

# Introduction to Java Training



with examples and  
hands-on exercises

---

**WEBUCATOR**

Copyright © 2021 by Webucator. All rights reserved.

No part of this manual may be reproduced or used in any manner without written permission of the copyright owner.

**Version:** 2.2.3

## **The Authors**

### ***Steven Claflin***

Steve has been providing training in C, C++, Java, database, and web-related programming for over fifteen years. His programming background includes web applications using GWT, Java, JavaScript, and jQuery, as well as embedded programming for programmable logic controllers. In the last few years, Steve has been working extensively with Ajax, jQuery, and GWT. Steve received undergraduate degrees from MIT and an MBA from the Harvard Business School.

Steve has been working with Webucator since 2005. He provides both onsite and online training. He has provided training for employees of many organizations such as Johnson & Johnson, ESPN, Cisco Systems, Verizon, Robert Half International, and more. Students enjoy Steve's thorough explanations, real-world examples, and hands-on approach.

### ***Brian Hoke (Editor)***

Brian Hoke is Principal of Bentley Hoke, a web consultancy in Syracuse, New York. The firm serves the professional services, education, government, nonprofit, and retail sectors with a variety of development, design, and marketing services. Core technologies for the firm include PHP and Wordpress, JavaScript and jQuery, Ruby on Rails, and HTML5/CSS3. Previously, Brian served as Director of Technology, Chair of the Computer and Information Science Department, and Dean of Students at Manlius Pebble Hill School, an independent day school in DeWitt, NY. Before that, Brian taught at Insitut auf dem Rosenberg, an international boarding school in St. Gallen, Switzerland. Brian holds degrees from Hamilton and Dartmouth colleges.

### ***Roger Sakowski (Editor)***

Roger has over 35 years of experience in technical training, programming, data management, network administration, and technical writing for companies such as NASA, Sun Microsystems, Bell Labs, GTE, GE, and Lucent among other Fortune 100 companies.

## **Class Files**

Download the class files used in this manual at  
<https://static.webucator.com/media/public/materials/classfiles/JVA102-2.2.3.zip>.

## **Errata**

Corrections to errors in the manual can be found at  
<https://www.webucator.com/books/errata/>.

# Table of Contents

LESSON 1. Java Introduction.....	1
Conventions in These Notes.....	1
The Java Environment - Overview.....	2
Writing a Java Program.....	3
Obtaining The Java Environment.....	4
Setting up Your Java Environment.....	5
Creating a Class that Can Run as a Program.....	11
Useful Stuff Necessary to Go Further.....	12
Using an Integrated Development Environment.....	13
📄 <b>Exercise 1: Running a Simple Java Program</b> .....	<b>15</b>
Using the Java Documentation.....	17
LESSON 2. Java Basics.....	19
Basic Java Syntax.....	19
Variables.....	24
Data.....	28
Constants and the final Keyword.....	32
Mathematics in Java.....	33
Creating and Using Methods.....	44
Variable Scope.....	50
📄 <b>Exercise 2: Method Exercise</b> .....	<b>52</b>

LESSON 3. Java Objects.....	55
Objects.....	55
Object-oriented Languages.....	57
Object-oriented Programs.....	58
Encapsulation.....	59
📄 <b>Exercise 3: Object Definition.....</b>	<b>62</b>
References.....	65
Defining a Class.....	68
More on Access Terms.....	70
Adding Data Members to a Class.....	71
Standard Practices for Fields and Methods.....	73
Java Beans.....	74
Bean Properties.....	75
📄 <b>Exercise 4: Payroll01: Creating an Employee Class .....</b>	<b>78</b>
Constructors.....	81
Instantiating Objects Revisited.....	83
Important Note on Constructors.....	84
📄 <b>Exercise 5: Payroll02: Adding an Employee Constructor.....</b>	<b>85</b>
Method Overloading.....	87
📄 <b>Exercise 6: Payroll03: Overloading Employee Constructors.....</b>	<b>91</b>
The this Keyword.....	94
Using this to Call Another Constructor.....	95
📄 <b>Exercise 7: Payroll04: Using the this Reference.....</b>	<b>99</b>
static Elements.....	101
The main Method.....	102
📄 <b>Exercise 8: Payroll05: A static field in Employee .....</b>	<b>104</b>
Garbage Collection.....	108
Java Packages.....	108
Compiling and Executing with Packages.....	110
Working with Packages.....	111
📄 <b>Exercise 9: Payroll06: Creating an employees Package.....</b>	<b>115</b>
Variable Argument Lists (varargs).....	117
📄 <b>Exercise 10: Payroll07: Using KeyboardReader in Payroll .....</b>	<b>127</b>
String, StringBuffer, and StringBuilder .....	129
Creating Documentation Comments and Using javadoc.....	130
📄 <b>Exercise 11: Payroll08: Creating and Using javadoc Comments.....</b>	<b>135</b>
Primitives and Wrapper Classes .....	142

LESSON 4. Comparisons and Flow Control Structures.....	145
Boolean-valued Expressions.....	145
Comparison Operators.....	146
Comparing Objects.....	146
Conditional Expression Examples.....	147
Complex boolean Expressions.....	148
Simple Branching.....	149
The if Statement.....	150
if Statement Examples.....	151
📄 <b>Exercise 12: Game01: A Guessing Game.....</b>	<b>153</b>
📄 <b>Exercise 13: Payroll-Control01: Modified Payroll.....</b>	<b>155</b>
Two Mutually Exclusive Branches.....	159
📄 <b>Exercise 14: Game02: A Revised Guessing Game.....</b>	<b>162</b>
Comparing a Number of Mutually Exclusive options - The switch Statement.....	164
📄 <b>Exercise 15: Game03: Multiple Levels.....</b>	<b>169</b>
Comparing Two Objects.....	171
Conditional Expression.....	174
📄 <b>Exercise 16: Payroll-Control02: Payroll With a Loop.....</b>	<b>181</b>
📄 <b>Exercise 17: Game04: Guessing Game with a Loop.....</b>	<b>183</b>
Additional Loop Control: break and continue .....	185
Continuing a Loop.....	186
Classpath, Code Libraries, and Jar Files.....	188
📄 <b>Exercise 18: Creating and Viewing a jar File.....</b>	<b>190</b>
Compiling to a Different Directory.....	190
LESSON 5. Arrays.....	193
Defining and Declaring Arrays.....	193
Instantiating Arrays.....	194
Initializing Arrays.....	196
Working With Arrays.....	197
Enhanced for Loops - the For-Each Loop .....	199
Array Variables.....	200
Copying Arrays.....	202
📄 <b>Exercise 19: Using the args Array.....</b>	<b>204</b>
📄 <b>Exercise 20: Game-Arrays01: A Guessing Game with Random Messages.....</b>	<b>207</b>
Arrays of Objects.....	209
📄 <b>Exercise 21: Payroll-Arrays01: An Array of employees.....</b>	<b>211</b>
Multi-Dimensional Arrays.....	213
Multidimensional Arrays in Memory.....	214
Example - Printing a Picture.....	216
Typecasting with Arrays of Primitives .....	218

LESSON 6. Inheritance.....	221
Inheritance.....	221
Inheritance Examples.....	222
Payroll with Inheritance.....	224
Derived Class Objects.....	225
Polymorphism.....	227
Creating a Derived Class.....	229
Inheritance Example - A Derived Class .....	230
Inheritance and Access.....	231
Inheritance and Constructors - the super Keyword.....	231
Example - Factoring Person Out of Employee.....	234
📄 <b>Exercise 22: Payroll-Inheritance01: Adding Types of Employees.....</b>	<b>241</b>
Derived Class Methods that Override Base Class Methods.....	250
Inheritance and Default Base Class Constructors.....	252
The Instantiation Process at Runtime.....	254
Typecasting with Object References.....	255
More on Overriding.....	258
Object Typecasting Example.....	259
Checking an Object’s Type: Using instanceof .....	261
Typecasting with Arrays of Objects.....	263
📄 <b>Exercise 23: Payroll-Inheritance02: Using the Employee Subclasses.....</b>	<b>264</b>
Other Inheritance-related Keywords.....	268
📄 <b>Exercise 24: Payroll-Inheritance03: Making our base classes abstract .....</b>	<b>271</b>
Methods Inherited from Object .....	274
LESSON 7. Interfaces.....	279
Interfaces.....	279
Creating an Interface Definition.....	280
Implementing Interfaces.....	282
Reference Variables and Interfaces.....	284
Interfaces and Inheritance.....	287
📄 <b>Exercise 25: Exercise: Payroll-Interfaces01.....</b>	<b>289</b>
Some Uses for Interfaces.....	295
Annotations.....	303
Annotation Details .....	303
Using Annotations.....	304

LESSON 8. Exceptions.....	307
Exceptions.....	307
Handling Exceptions.....	308
Exception Objects.....	310
Attempting Risky Code - try and catch .....	311
Guaranteeing Execution of Code - The finally Block.....	315
Letting an Exception be Thrown to the Method Caller.....	318
Throwing an Exception.....	319
📄 <b>Exercise 26: Payroll-Exceptions01: Handling NumberFormatException in Payroll.....</b>	<b>321</b>
📄 <b>Exercise 27: Payroll-Exceptions01, continued.....</b>	<b>323</b>
Exceptions and Inheritance.....	332
Creating and Using Your Own Exception Classes.....	334
📄 <b>Exercise 28: Payroll-Exceptions02.....</b>	<b>337</b>
Rethrowing Exceptions.....	344
Initializer Blocks.....	345
Logging.....	347
Log Properties.....	353
Assertions.....	354
LESSON 9. Collections.....	357
Collections.....	357
Using the Collection Classes.....	361
Using the Iterator Interface.....	363
Creating Collectible Classes.....	367
Generics.....	371
Bounded Types.....	374
Extending Generic Classes and Implementing Generic Interfaces .....	375
Generic Methods.....	375
Variations on Generics - Wildcards .....	376
Type Erasure.....	377
Multiple-bounded Type Parameters.....	379
📄 <b>Exercise 29: Payroll Using Generics.....</b>	<b>380</b>
LESSON 10. Inner Classes.....	385
Inner Classes, aka Nested Classes.....	385
Inner Class Syntax.....	387
Instantiating an Inner Class Instance from within the Enclosing Class.....	389
Inner Classes Referenced from Outside the Enclosing Class.....	390
Referencing the Outer Class Instance from the Inner Class Code .....	391
Better Practices for Working with Inner Classes.....	393
Enums.....	405



# LESSON 1

## Java Introduction

---

### Topics Covered

- ☑ The Java Runtime Environment.
- ☑ How a Java program is created, compiled, and run.
- ☑ The Java Development Kit Standard Edition.
- ☑ A simple Java program.

### Introduction

In this lesson, you will learn about the Java Runtime Environment and how a Java program is created, compiled, and run, how to download, install, and set up the Java Development Kit Standard Edition, and how to create a simple Java program.

*Evaluation  
Copy*

---

### 1.1. Conventions in These Notes

Code is listed in a monospace font, both for code examples and for Java keywords mentioned in the text.

The standard Java convention for names is used:

- Class names are listed with an initial uppercase letter.
- Variable and function names are listed with an initial lowercase letter.
- The first letters of inner words are capitalized (e.g., `maxValue`).

For syntax definitions:

- Terms you must use as is are listed in normal monospace type.
- Terms that you must substitute for, either one of a set of allowable values, or a name of your own, or code, listed in *italics*.

- The following generic terms are used - you must substitute an appropriate term.

Generic Terms	Substitution Options
access	An access word from: public, protected, private, or it can be omitted
modifiers	one or more terms that modify a declaration; these include the access terms as well as terms like: static, transient, or volatile
dataType	A data type word; this can be a primitive, such as int, or the name of a class, such as Object; variants of this include: returnType and paramType
variableName	The name of a variable; variants on this include paramName and functionName
ClassName	The name of a class; there will be variants of this used for different types of examples, such as: BaseClassName, DerivedClassName, InterfaceName, and ExceptionClassName
code	Executable code goes here
. . .	In an example, omitted code not related to the topic

Don't worry if some (or all) of the terms in the table above don't make sense right now - we'll cover all of this material in detail as we move through the course.



## 1.2. The Java Environment - Overview

A Java program is run differently than a traditional executable program.

Traditional programs are invoked through the operating system (OS).

- They occupy their own memory space.
- They are tracked as an individual process by the OS.

Traditional programs are compiled from source code into a machine and OS-specific binary executable file.

- To run the program in different environments, the source code would be compiled for that specific target environment.

When you run a Java program, the OS is actually running the *Java Runtime Engine*, or *JRE*, as an executable program; it processes your compiled code through the *Java Virtual Machine* (usually referred to as the *JVM*).

A Java source code file is compiled into a *bytecode* file.

- You can consider bytecode as a sort of generic machine language.
- The compiled bytecode is read by the JVM as a data file.
- The JVM interprets the bytecode at runtime, performing a final mapping of bytecode to machine language for whatever platform it is running on.
- Thus, Java programs are portable - they can be written in any environment, compiled in any environment, and run in any environment (as long as a JVM is available for that environment).
- The JVM manages its own memory area and allocates it to your program as necessary.
- Although this involves more steps at runtime, Java is still very efficient, much more so than completely interpreted languages like JavaScript, since the time-consuming parsing of the source code is done in advance.

---

### 1.3. Writing a Java Program

Java source code is written in plain text, using a text editor.

- This could range from a plain text editor like Notepad, to a programmer's editor such as TextPad, EditPlus, or Crimson Editor, to a complex integrated development environment (IDE) like NetBeans, Eclipse, or JDeveloper.
- The source code file should have a `.java` extension.

The `javac` compiler is then used to compile the source code into *bytecode*. The following command, run from the command line (the "Command Prompt", also known as "cmd", on a Windows computer; "Terminal" on a Mac), would compile the source-code file `MyClass.java` into the bytecode file `MyClass.class`:

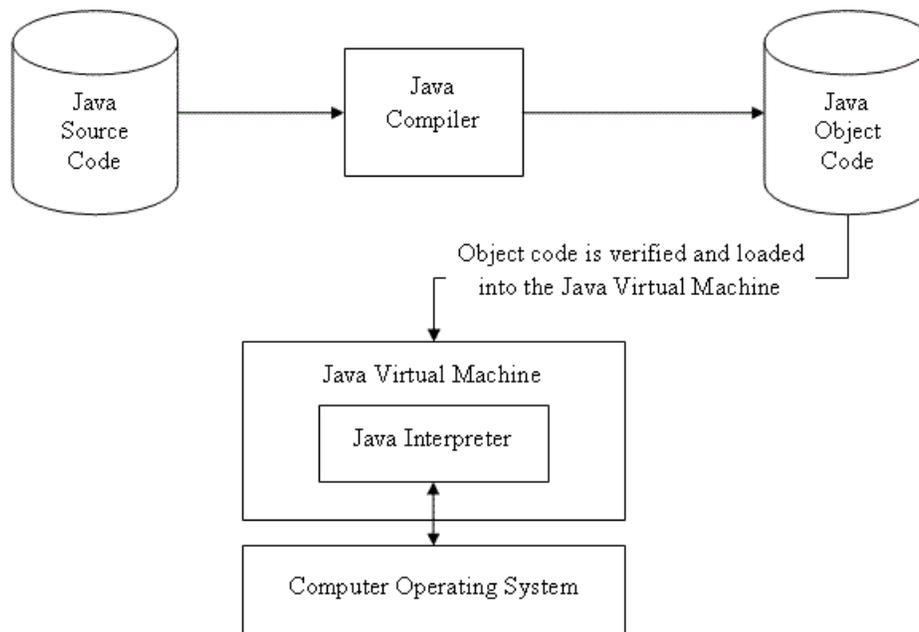
```
javac MyClass.java
```

- The bytecode is stored in a file with an extension `.class`
- Bytecode is universal - one bytecode file will run on any supported platform.

You then run the `java` runtime engine, which will then interpret the bytecode to execute the program. This command runs the compiled bytecode:

```
java MyClass
```

- The executable program you are running is: `java`.
- The class name tells the JVM what class to load and run the `main` method for.
- You must have a JVM made specifically for your hardware/OS platform.



## 1.4. Obtaining The Java Environment

You can download the JDK (Java Development Kit), including the compiler and runtime engine, for free from [aws.amazon.com](https://docs.aws.amazon.com/corretto/latest/corretto-11-ug/downloads-list.html) at: <https://docs.aws.amazon.com/corretto/latest/corretto-11-ug/downloads-list.html>. Look for the download of the **Coretto 11 JDK** for your operating system. For Windows 10, here is the `msi` file to download:

**Amazon Corretto**  
Corretto 11 User Guide

What Is Amazon Corretto 11?  
List of Patches for Amazon Corretto 11

- ▶ Linux
- ▶ Windows
- ▶ macOS
- ▶ Docker
- Downloads**
- Document History

**Note:** Permanent URL's are redirected (HTTP 302) to actual artifact's URL.

These links can be used in scripts to pull the latest version of Amazon Corretto 11.

Platform	Type	Download Link
		<a href="https://corretto.aws/downloads/latest/corretto-11-x86-linux-jdk.tar.gz">https://corretto.aws/downloads/latest/corretto-11-x86-linux-jdk.tar.gz</a>
Windows x64	JDK	<a href="https://corretto.aws/downloads/latest/corretto-11-x64-windows-jdk.msi">https://corretto.aws/downloads/latest/corretto-11-x64-windows-jdk.msi</a>
		<a href="https://corretto.aws/downloads/latest/corretto-11-x64-windows-jdk.zip">https://corretto.aws/downloads/latest/corretto-11-x64-windows-jdk.zip</a>
macOS x64	JDK	<a href="https://corretto.aws/downloads/latest/corretto-11-x64-macos-jdk.pkg">https://corretto.aws/downloads/latest/corretto-11-x64-macos-jdk.pkg</a>
		<a href="https://corretto.aws/downloads/latest/">https://corretto.aws/downloads/latest/</a>

For Mac users, installation instructions can be found at <https://docs.aws.amazon.com/corretto/latest/corretto-11-ug/macos-install.html>.

For Windows users, after you have downloaded the msi file, double-click the file in your downloads folder to install Java on your computer. Accept the defaults. The location of the installation will be `C:\Program Files\Amazon Corretto\jdk11.0.9_12`. This is your Java home. **Note:** You might have a different sequence of numbers after "jdk11.". This means you have a different release and build of the Corretto JDK and that is fine.

You can also download the API documentation.

- The documentation lists all the standard classes in the API, with their data fields and methods, as well as other information necessary to use the class. You will find this documentation helpful as you work through some of the exercises.
- You can view the docs online at <https://docs.oracle.com/en/java/javase/11/docs/api/index.html>



## 1.5. Setting up Your Java Environment

Java programs are compiled and run from an operating system prompt, unless you have installed an IDE that will do this for you directly.

After you have installed the JDK, you will need to set at least one environment variable in order to be able to compile and run Java programs.

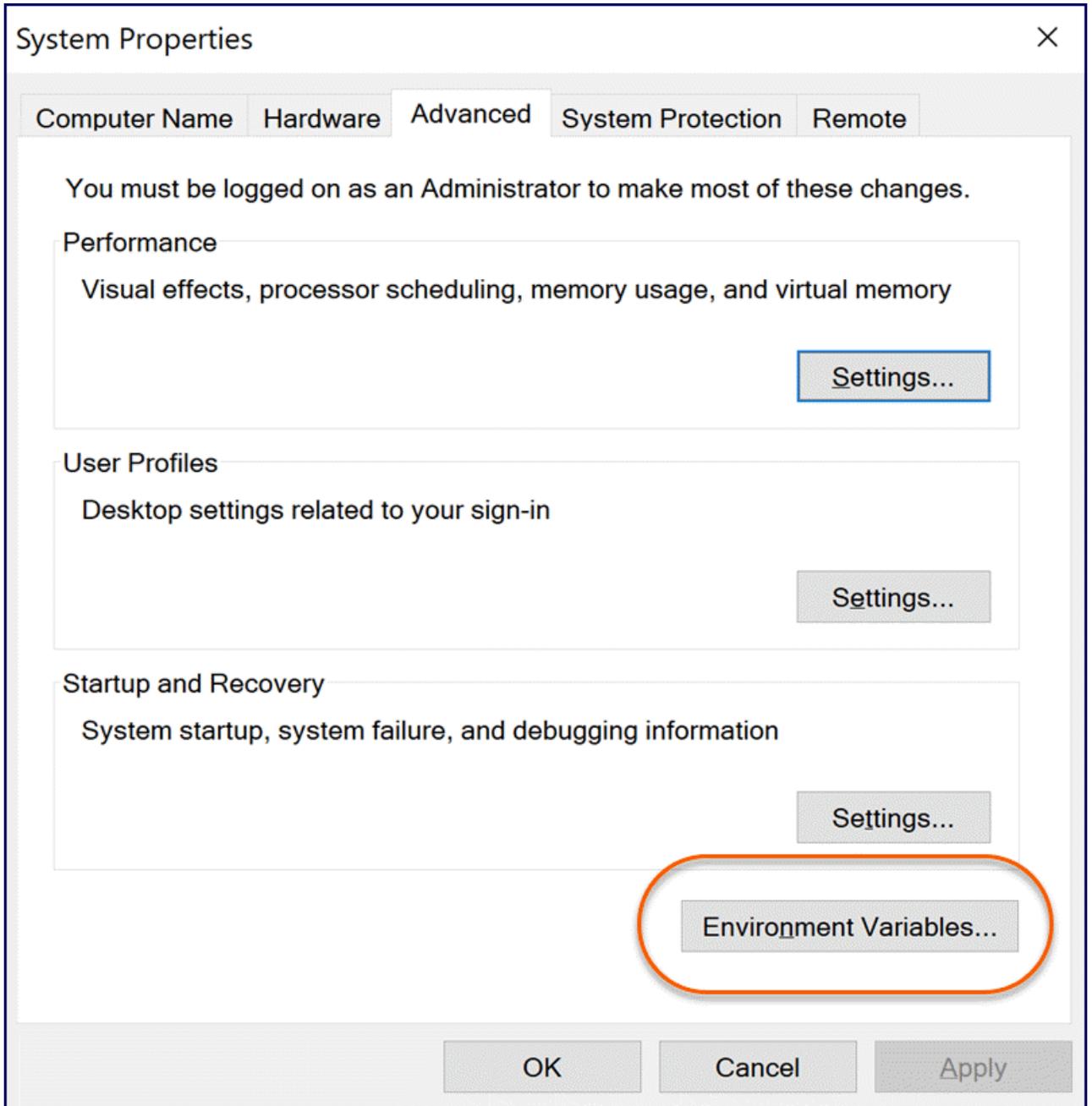
For more complex projects that pull together elements from different sources, you must set an additional environment variable or two. Note that:

- a Path environment variable enables the operating system to find the JDK executables when your working directory is not the JDK's binary directory.
- CLASSPATH is Java's analog to Path, the compiler and JVM use it to locate Java classes.
  - Often you will not need to set this, since the default setting is to use the JDK's library jar file and the current working directory.
  - But, if you have additional Java classes located in another directory (a third-party library, perhaps), you will need to create a classpath that includes not only that library, but the current working directory as well (the current directory is represented as a dot).
- Many IDE's and servers expect to find a JAVA\_HOME environment variable.
  - This would be the JDK directory (the one that contains bin and lib).
  - The Path is then set from JAVA\_HOME plus \bin.
  - This makes it easy to upgrade your JDK, since there is only one entry you will need to change.

#### Best Practice: Setting Path from JAVA\_HOME

The instructions below will show how you create the JAVA\_HOME environment variable and add a Path entry from JAVA\_HOME on Windows.

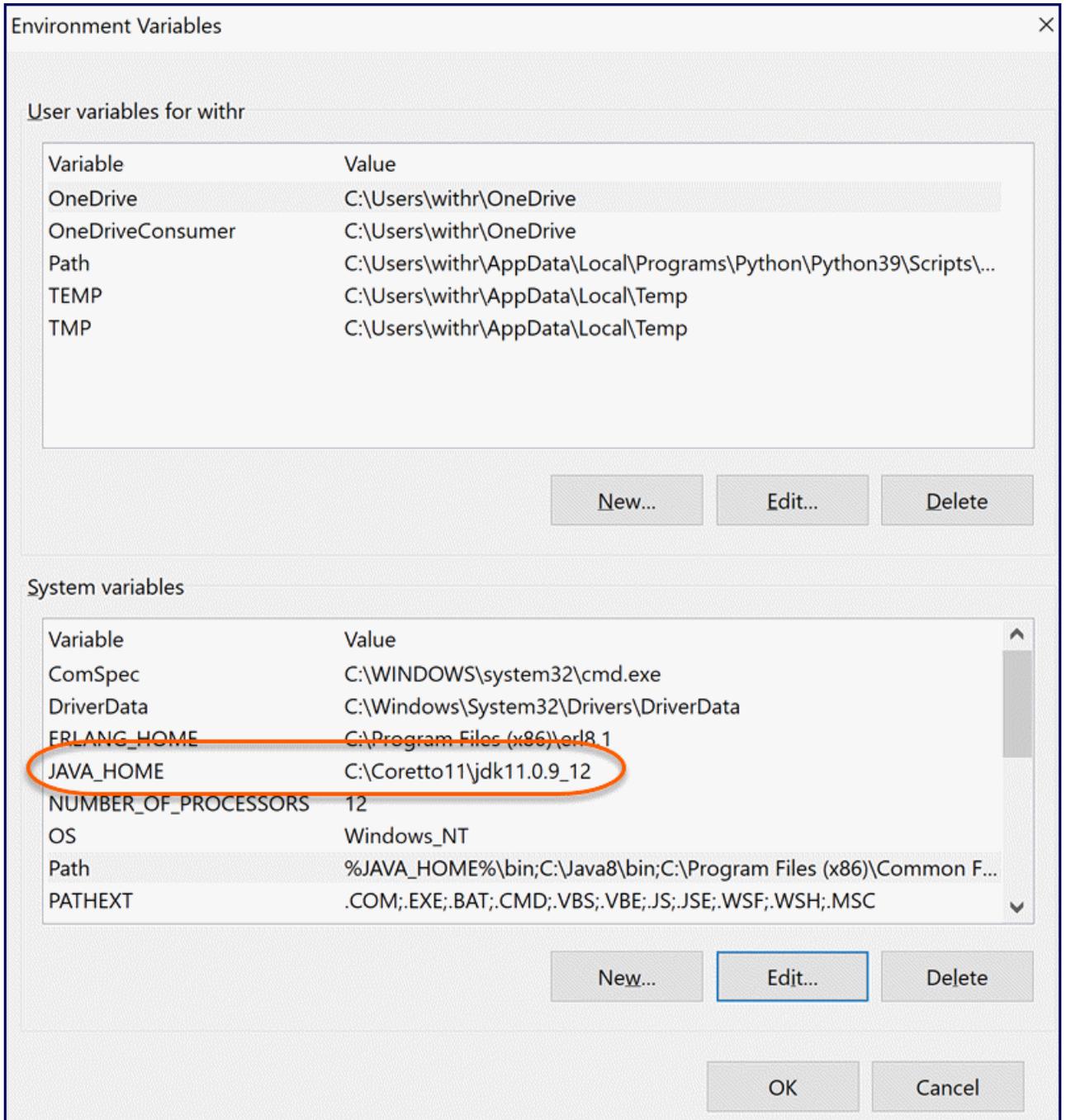
1. In the **Search** field in the lower left hand corner, type env and select **Best Match** at the top of the search results panel.
  - You should see the **System Properties** panel:



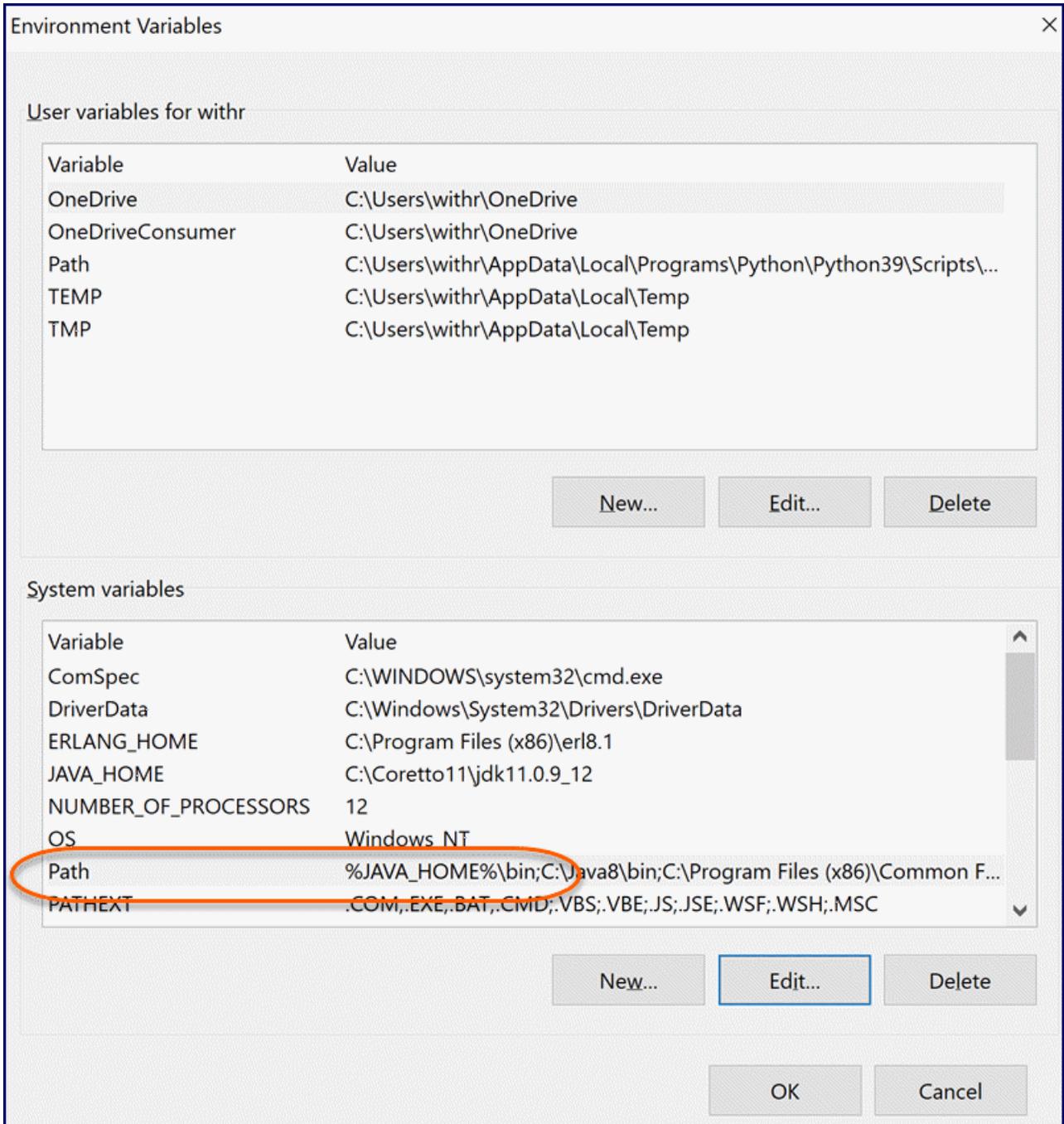
- Click the **Environment Variables...** button.
2. Check both the **User** and **System** variable lists to see if JAVA\_HOME or Path already exist.
  3. If JAVA\_HOME exists, check to see that it matches the JDK path from the installation - in our example above, that would be the path C:\Program Files\Amazon Coret

to \jdk11.0.9\_12 (**Note:** your sequence of numbers after "jdk11." might be different and that is fine).

- A. If JAVA\_HOME exists under **System variables**, click **Edit**, if not, click **Add**
- B. For **Variable name**, enter JAVA\_HOME
- C. For **Variable value**, enter the JDK directory, e.g., C:\Program Files\Amazon Corretto\jdk11.0.9\_12.
- D. Click **OK**.
- E. Your Java home is now set:



- Next, add an entry to the Path environment variable to reference the bin directory of the JAVA\_HOME system variable you just created. In the screenshot below, we edited the existing Path System variable, adding %JAVA\_HOME%\bin at the beginning of the the existing paths:



5. Click **OK** to save your changes.
6. Once you have completed these steps, open a command prompt and type `java -version` and press enter; you should see the response shown in the following screenshot:

```
Command Prompt
Microsoft Windows [Version 10.0.18363.1139]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\withr>java -version
openjdk version "11.0.9.1" 2020-11-04 LTS
OpenJDK Runtime Environment Corretto-11.0.9.12.1 (build 11.0.9.1+12-LTS)
OpenJDK 64-Bit Server VM Corretto-11.0.9.12.1 (build 11.0.9.1+12-LTS, mixed mode)

C:\Users\withr>
```

If you experience an error, go back and confirm that you have correctly followed the steps above. Additional assistance for Windows users is available at <https://docs.aws.amazon.com/corretto/latest/corretto-11-ug/windows-7-install.html>.



## 1.6. Creating a Class that Can Run as a Program

### ❖ 1.6.1. The main() Method

In order to run as a program, a class must contain a method named `main`, with a particular argument list. This is similar to the C and C++ languages.

The method declaration goes inside your class definition, and looks like:

```
public static void main(String[] args) {
    (code goes here)
}
```

- It must be `public`, because it will be called from outside your class (by the JVM).

- The `static` keyword defines an element (could be data or functional) that will exist regardless of whether an object of a class has been instantiated. In other words, declaring `main` as `static` allows the JVM to call the method and therefore execute the program. That doesn't mean that an object of that class cannot be instantiated, it is just not required.
- The `String[] args` parameter states that there will be an array of `String` objects given to the method - these are the command line arguments.
- To run the program, you would first compile a file named `ClassName.java` with the following from the command line:

```
javac ClassName.java
```

and then use the following from the command line to run the program:

```
java ClassName
```

For example, if we had an executable class called `Hello`, in a file called `Hello.java` that compiled to `Hello.class`, you could run it with:

```
java Hello
```

Evaluation  
Copy

As above, if terms like “static” and “instantiated” don't make too much sense right now, don't worry - we will cover these concepts in detail soon.



## 1.7. Useful Stuff Necessary to Go Further

### ❖ 1.7.1. `System.out.println()`

In order to see something happen, we need to be able to print to the screen.

There is a `System` class that is automatically available when your program runs (everything in it is `static`).

- It contains, among other things, input and output streams that match `stdin`, `stdout`, and `stderr` (standard output, standard input, and standard error).

`System.out` is a static reference to the standard output stream.

As an object, `System.out` contains a `println(String)` method that accepts a `String` object, and prints that text to the screen, ending with a newline (linefeed).

- There is also a `print(String)` method that does not place a newline at the end.

You can print a `String` directly, or you can build one from pieces.

- It is worth noting that a `String` will automatically be created from a quote-delimited series of characters.
- Values can be appended to a `String` by using the `+` sign; if one item is a `String` or quote-delimited series of characters, then the rest can be any other type of data.



## 1.8. Using an Integrated Development Environment

*Integrated Development Environments*, or *IDEs*, can greatly facilitate the task of creating applications. Mid-level code editors, such as Crimson Editor, TextPad, or Edit-Plus, provide syntax highlighting, automatic indenting, parentheses and curly-brace matching, and may have tools to compile and execute programs.

High-end environments, such as Eclipse, NetBeans, or JDeveloper, offer many additional features, such as project management, automatic deployment for web applications, code refactoring, code assist, etc. But, these additional capabilities come with the price of additional complexity in the environment, particularly because they force you to create *projects* for even the simplest applications.

To use the class files in these environments, we must first have a *workspace*, which is a collection of projects. Workspaces have various configuration settings that will cut across all projects within all projects they contain, such as which version of Java is in use, any added libraries, etc. (a project is usually one application).

Both Eclipse and NetBeans use the workspace concept, but you can start with any empty directory - the IDE will add its own special files as you create projects.

In the Windows Explorer, navigate to your ClassFiles directory. If it does not contain a Workspace subdirectory, create it.

To use Eclipse with the class files, you can direct it to the workspace (ClassFiles/Workspace) when it opens (if it asks), or use *File, Switch Workspace* if Eclipse is already running. Each chapter's demo folder should be treated as a project, as should each Solution folder.

One limitation of these environments is that no project can contain the same Java class twice, so our progressive versions of the Payroll application solution require their own individual project. Also, due to the way Java's package structures work, each Demo folder must be a separate project, as with each subdirectory under Solutions.

To use the Exercises folder as an Eclipse project:

1. Select *File, New ... Java Project* from the menu (if *Java Project* is not immediately available, choose *Project ...* first - the menu items are usage-sensitive).
2. Name the project Exercises.
3. check *Create project from existing source*.
4. Browse to the Exercises directory, and *OK* it.
5. You can then *Finish* the dialog box.

As we encounter each chapter, we can create a project using the Demos directory for that chapter the same way we just created the Exercises project..

To create a Java class within a project, use *File, New*, and then *Class* if that is available, or *Other...* and then *Class*. Provide a name and then *OK*.

In general, with the several applications that we will build upon progressively (Game and Payroll), you can continue to work with the same files, and add to them as we cover additional topics. If you want to check the solutions, you can either open those files in a separate editor, like Notepad, or create a project for that Solutions subdirectory. If you wish to save a particular stage of the application for future reference, you can just make a copy of the Exercises directory.

The examples and instructions going forward in this course assume the use of a basic text editor and the command line. You are free to choose which tool to use. If Eclipse becomes a problem, move to a simple editor. The focus of this course is on the Java Programming Language and not a particular tool.

# Exercise 1: Running a Simple Java Program

 5 to 15 minutes

---

1. Use a text editor or IDE to open `Java-Introduction/Exercises/Hello.java`.
2. Note that the file is intentionally blank.
3. Enter the following code:

```
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
}
```

4. Save the file.
5. In a command prompt window, change to the directory where `Hello.java` is stored and type the following

```
javac Hello.java
```

and press *Enter*. A prompt on the next line without any error messages indicates success.

6. Next, type

```
java Hello
```

7. Press *Enter*. You should see the message `Hello World` print in the window.

```
public class Hello
```

- All Java code must be within a class definition.
- A class defines a type of object.
- A public class can be accessed by any other class.

- A public class must be in its own file, whose name is the name of the class, plus a dot and an file extension of java (e.g., Hello.java).

```
{  
.  
.  
.  
}
```

- Curly braces denote a block of code (the code inside the braces).
- Code within braces usually belongs to whatever immediately precedes them.

```
public static void main(String[] args) {
```

- Words followed by parentheses denote a function.
- To be executable by itself, a class must have a function defined in this fashion; the name `main` means that execution will start with the first step of this function, and will end when the last step is done.
- It is `public` because it needs to be executed by other Java objects (the JVM itself is a running Java program, and it launches your program and calls its `main` function).
- It is `static` because it needs to exist even before one of these objects has been created.
- It does not return an answer when it is done; the absence of data is called `void`.
- The `String[] args` represents the additional data that might have been entered on the command line.

```
System.out.println("Hello World!");
```

- `System` is an class within the JVM that represents system resources.
- It contains an object called `out`, which represents output to the screen (what many environments call *standard out*).
- Note that an object's ownership of an element is denoted by using the name of the object, a dot, and then the name of the element.
- `out` in turn contains a function, `println`, that prints a line onto the screen (and appends a newline at the end).
- A function call is denoted by the function name followed by parentheses; any information inside the parentheses is used as input to the function (called arguments or parameters to the function).

- The argument passed to `println()` is the string of text "Hello World!".
- Note that the statement ends in a semicolon (as all do Java statements).



## 1.9. Using the Java Documentation

Oracle provides extensive Java documentation. The home page for Java documentation in English is at <https://docs.oracle.com/en/java/>. Documentation for the platform (JDK Standard Edition, or "SE") that you will use in this course is at <https://docs.oracle.com/javase/10/> for the current version - version 10 as of this writing.

This page offer a wealth of resources - videos, downloadable books, etc. - to help you learn Java. Perhaps the most useful is the API reference - a detailed list of all of the built-in features of Java available to you when programming in the language:

<https://docs.oracle.com/javase/10/docs/api/overview-summary.html>

If you view that page, you will see a list of classes on the left as hyperlinks. Clicking a class name will bring up the documentation for that class on the right.

For example, click `java.base`, then `java.lang`, then out under "Field Summary" to get to this page:

<https://docs.oracle.com/javase/10/docs/api/java/lang/System.html#out>

From this page you can find the `println` methods we looked at above in a table with short descriptions. Select one to see the detailed description. The multiple versions are an example of *method overloading*, which we will cover in an upcoming lesson. Another lesson will cover documenting your own classes with the same tool that created this documentation.

In this lesson, you have learned:

## Conclusion

In this lesson you have learned:

1. About the Java Runtime Environment.
2. How to download, install, and set up the Java Development Kit Standard Edition.

3. How to create a simple Java program.

Evaluation Copy

# LESSON 2

## Java Basics

---

### Topics Covered

- ✓ Java's basic syntax rules, including statements, blocks, and comments.
- ✓ Declaring variables.
- ✓ Constructing statements using variables and literal values.
- ✓ Primitive data types.
- ✓ The `String` class.
- ✓ Java's operators and operator precedence.
- ✓ Converting data from one type to another.
- ✓ Simple methods.

Evaluation  
Copy

### Introduction

In this lesson, you will learn Java's basic syntax rules, including statements, blocks, and comments. You will also learn to declare variables and to construct statements using variables and literal (constant) values. You will get familiar with the primitive data types, as well as the `String` class, and come to understand many of Java's operators and the concept of operator precedence. You will learn the rules that apply when data is converted from one type to another. Finally, you will learn to declare, write, and use simple methods.



## 2.1. Basic Java Syntax

### ❖ 2.1.1. General Syntax Rules

Java is *case-sensitive*. `main()`, `Main()`, and `MAIN()` would all be different methods (the term we use for functions in Java).

There are a limited number of reserved words that have a special meaning within Java. You may not use these words for your own variables or methods.

Some examples of reserved words are:

- `public`
- `void`
- `static`
- `do`
- `for`
- `while`
- `if`

Most keyboard symbol characters (the set of characters other than alphabetic or numeric) have a special meaning.

Identifiers - names we use for methods, variables, or classes - may contain alphabetic characters, numeric characters, currency characters, and connecting characters such as the underscore ( `_` ) character:

- Identifiers may not begin with a numeric character.
- Note that the set of legal characters draws from the entire Unicode character set.
- Also note that it is probably impossible to write a succinct set of rules about what are valid characters, other than to say a character, that when passed to `Character.isJavaIdentifierPart(char ch)`, results in a `true` value.

The compiler parses your code by separating it into individual entities called *tokens* or *symbols* in computer science jargon:

- Identifiers (of classes, variables, and methods).
- Command keywords.
- Single or compound symbols (compound symbols are when an operation is signified by a two-symbol combination.)

Tokens may be separated by spaces, tabs, carriage returns, or by use of an *operator* (such as `+`, `-`, etc.).

## Note

Since identifiers may not contain spaces, tabs, or carriage returns, or operator characters, these characters imply a separation of what came before them from what comes after them.

Extra *whitespace* is ignored. Once the compiler knows that two items are separate, it ignores any additional separating whitespace characters (spaces, tabs, or carriage returns).

### ❖ 2.1.2. Java Statements

- A Java statement is one step of code, which may span across multiple lines.
- Java statements end with a semicolon (the ; character).
- It is OK to have multiple statements on a single line.
- Program execution is done statement by statement, one statement at a time from top to bottom (if there is more than one statement on a line, execution goes from left to right).
- Within a statement, execution of the individual pieces is not necessarily left to right. There are concepts called *operator precedence* and *associativity* that determine the order of operations within a statement.

### ❖ 2.1.3. Blocks of Code

A block of code:

- Is enclosed in curly braces - start with { and end with }.
- Consists of zero, one, or more statements.
- Behaves like a single statement to the outside world..
- May be nested (e.g., it may contain one or more other blocks of code inside).
- Generally belong to whatever comes before it. Although it is perfectly OK to create a block that does not, this is almost never done.

A complete method is a single block of code, most likely with nested blocks.

The diagram below illustrates how blocks of code can be nested:

```

public class Hello {
    public static void main(String[] args) {
        if (Math.random( ) < 0.5) {
            System.out.println("Hello World!");
        }
        System.out.println("End of Program");
    }
}

```

If you want, go ahead and modify your Hello World program to match this example.

## ❖ 2.1.4. Comments

A comment:

- Is additional, non-executable text in a program used to document code.
- May also be used to temporarily disable a section of code (for debugging).

### Block Comments

Block comments are preceded by `/*` and followed by `*/`.

Some rules for block comments:

- May not be nested.
- May be one or more lines.

#### **Block Comment**

```

/* this is a block comment
 * asterisk on this line not necessary, but looks nice
 */

```

- May span part of a line, for example, to temporarily disable part of a statement:

#### **Block Comment Within a Line**

```

x = 3 /* + y */ ;

```

## Single-line Comments

A single line can be a comment by preceding the comment with two forward slashes: `//`. Note that:

- The comment ends at the end of that line.
- Single-line comments may be nested inside block comments.

### Comment-to-end-of-line Within a Block Comment

```
y = 7;
/*
 * temporarily disable this line
 * which has a comment to end of line
x = 3 + y; // add 3 for some reason
*/
```

Java specifies a third type of comment, the *javadoc* comment:

- Java contains a self-documentation utility, *javadoc*, which builds documentation from comments within the code.
- javadoc comments begin with `/**` and end with `*/`.
- Comments only work as javadoc comments when placed at specific locations in your code (immediately above anything documentable - the class itself and its members), otherwise they are treated as ordinary comments.

### Javadoc Comment

```
/**
 * Represents a person, with a first name and last name.
 */
public class Person {
    /** The person's first name */
    public String firstName;
    /** The person's last name */
    public String lastName;
}
```



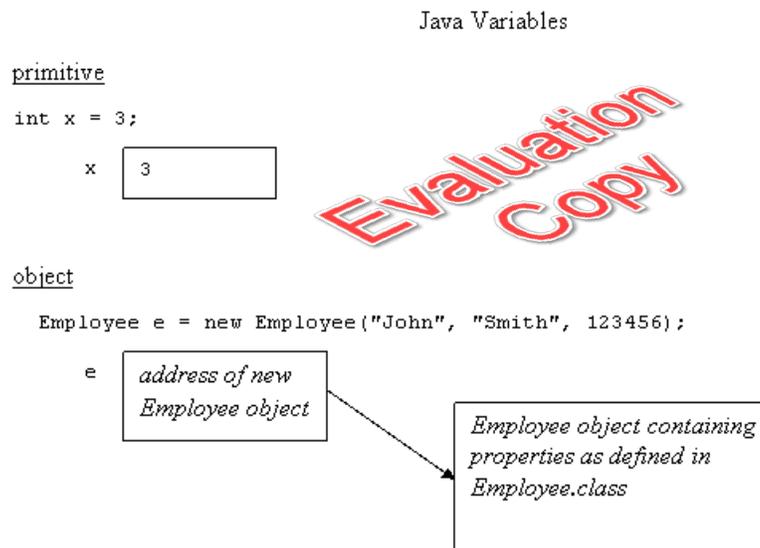
## 2.2. Variables

Variables store data that your code can use.

There are two fundamental categories of variables, *primitive data* and *references*:

- With primitive data, the compiler names a memory location and uses it to store the actual data - numeric values such as integers, floating point values, and the code values of single individual text characters are stored as primitives.
- With references, the data is accessed indirectly - the compiler selects a memory location, associates it with the variable name, and stores in it a value that is effectively the memory address of the actual data - in Java, all objects and arrays are stored using references.

In the diagram below, the boxes are areas in memory:



### ❖ 2.2.1. Declaring Variables

Variables must be *declared* before they are used.

A declaration informs the compiler that you wish to:

- Create an *identifier* that will be accepted by the compiler within a section of code (exactly what that section is depends on how and where the variable is declared; that is the concept of *scope*, which will be addressed later).

- Associate that identifier with a specified type of data, and enforce restrictions related to that in the remainder of your code.
- Create a memory location for the specified type of data.
- Associate the identifier with that memory location.

Java uses many specific data types; each has different properties in terms of size required and handling by the compiler:

- The declaration specifies the name and datatype.
- Generally the declaration also defines the variable, meaning that it causes the compiler to allocate storage space in memory of the requested type's size.
- A declaration may also assign an initial value (to *initialize* the variable).
- Multiple variables of the same type can be declared in a comma-separated list.

Code	Effect
<code>int a;</code>	declares the name <code>a</code> to exist, and allocates a memory location to hold a 32-bit integer
<code>int a = 0;</code>	same as above, and also assigns an initial value of 0
<code>int a = 0, b, c = 3;</code>	declares three integer variables and initializes two of them

Note that different languages have different rules regarding variables that have not been initialized:

- Some languages just let the value be whatever happened to be in that memory location already (from some previous operation).
- Some languages initialize automatically to 0.
- Some languages will produce a compiler or runtime error.
- Java uses both of the last two: local variables within methods must be initialized before attempting to use their value, while variables that are fields within objects are automatically set to zero.

## ❖ 2.2.2. Advanced Declarations

Local variables, fields, methods, and classes may be given additional *modifiers*; keywords that determine special characteristics.

Any modifiers must appear first in any declaration, but multiple modifiers may appear in any order.

Evaluation Copy

Keyword	Usage	Comments
final	local variables, fields, methods, classes	<p>The name refers to a fixed item that cannot be changed.</p> <p>For a variable, that means that the value cannot be changed.</p> <p>For a method, the method cannot be overridden when extending the class.</p> <p>A final field does not, however, have to be initialized immediately; the initial assignment may be done once during an object's construction.</p>
static	fields, methods, inner classes	<p>Only for fields and methods of objects.</p> <p>One copy of the element exists regardless of how many instances are created.</p> <p>The element is created when the class is loaded.</p>
transient	fields	<p>The value of this element will not be saved with this object when serialization is used (for example, to save a binary object to a file, or send one across a network connection).</p>
volatile	fields	<p>The value of this element may change due to outside influences (other threads), so the compiler should not perform any caching optimizations.</p>
public, protected, private	fields, methods, classes	<p>Specifies the level of access from other classes to this element - covered in depth later.</p>
abstract	methods, classes	<p>Specifies that a method is required for a concrete extension of this class, but that the method will not be created at this level of inheritance - the class must be extended to realize the method.</p> <p>For a class, specifies that the class itself may not be instantiated; only an extending class that is not abstract may be instantiated (a class must be abstract if one or more of its methods are abstract) - covered in depth later</p>

Keyword	Usage	Comments
native	methods	The method is realized in native code (as opposed to Java code) - there is an external tool in the JDK for mapping functions from a DLL to these methods.
strictfp	methods, classes	For a method, it should perform all calculations in strict floating point (some processors have the ability to perform floating point more accurately by storing intermediate results in a larger number of bits than the final result will have; while more accurate, this means that the results might differ across platforms).  For a class, this means that all methods are strictfp.
synchronized	methods, code blocks	No synchronized code may be accessed from multiple threads for the same object instance at the same time.

Evaluation  
\*  
Copy

## 2.3. Data

### ❖ 2.3.1. Primitive Data Types

- The *primitive* data types store single values at some memory location that the compiler selects and maps to the variable name you declared .
- Primitive values are not objects - they do not have fields or methods.
- A primitive value is stored at the named location, while an object is accessed using a *reference*.
- An object reference variable does not store the object's data directly - it stores a reference to the block of data, which is somewhere else in memory (technically, the reference stores the memory address of the object, but you never get to see or use the address).

## Primitives Data Types

Primitive Type	Storage Size	Comments
boolean	1 bit	not usable mathematically, but can be used with logical and bitwise operators
char	16 bits	unsigned, not usable for math without converting to <code>int</code>
byte	8 bits	signed
short	16 bits	signed
int	32 bits	signed
long	64 bits	signed
float	32 bits	signed
double	64 bits	signed
void	None	not really a primitive, but worth including here

### ❖ 2.3.2. Object Data Types

Objects can be data, which can be stored in variables, passed to methods, or returned from methods.

Evaluation  
Copy

#### References

As we will see later, objects are stored differently than primitives. An object variable stores a *reference* to the object (the object is located at some other memory location, and the reference is something like a memory address).

#### Text Strings

A sequence of text, such as a name, an error message, etc., is known as a *string*.

In Java, the `String` class is used to hold a sequence of text characters.

A `String` object:

- Is accessed through a reference, since it is an object.
- Has a number of useful methods, for case-sensitive or case-insensitive comparisons, obtaining substrings, determining the number of characters, converting to upper or lower case, etc.

- Is *immutable*; that is, once created it cannot be changed (but you can make your variable reference a different `String` object at any time).

### ❖ 2.3.3. Literal Values

A value typed into your code is called a *literal* value.

The compiler makes certain assumptions about literals:

- `true` and `false` are literal boolean values.
- `null` is a literal reference to nothing (for objects).
- A numeric value with no decimal places becomes an `int`, unless it is immediately assigned into a variable of a smaller type and falls within the valid range for that type.
- A value with decimal places becomes a `double`.
- To store a text character, you can put apostrophes around the character.

Code	Effect
<code>char e = 'X';</code>	Creates a 16-bit variable to hold the Unicode value for the uppercase X character.

You can add modifiers to values to instruct the compiler what type of value to create (note that all the modifiers described below can use either uppercase or lowercase letters).

Modifying prefixes enable you to use a different number base:

Prefix	Effect
<code>0X</code> or <code>0x</code>	A base 16 value; the extra digits can be either uppercase or lowercase, as in <code>char c = 0x1b;</code>
<code>0</code>	A base 8 value, as in <code>int i = 0765;</code>

#### Notes

- Using these prefixes will always result in number that is considered positive (so that `0x7F` for a byte would be OK, but `0x80` would not, since the latter would have a value of 128, outside the range of byte).
- A long value would need the `L` modifier as discussed below.

Modifying suffixes create a value of a different type than the default:

Suffix	Effect
<b>L or l</b>	a long value (uses 64 bits of storage), as in <code>long l = 1234567890123456L;</code> Note: An <code>int</code> value will always implicitly be promoted to a <code>long</code> when required, but the reverse is not true; the above notation is necessary because the literal value is larger than 32 bits
<b>F or f</b>	A float value, as in <code>float f = 3.7F;</code>

## Escape Sequences for Character Values

There are a number of *escape sequences* that are used for special characters:

Escape Sequence	Resulting Character
<code>\b</code>	Backspace
<code>\f</code>	Form feed
<code>\n</code>	Linefeed character - note that it produces exactly one character, Unicode 10 ( <code>\u000A</code> in hex)
<code>\r</code>	Carriage return
<code>\t</code>	Tab
<code>\"</code>	Quote mark
<code>\'</code>	Apostrophe
<code>\\</code>	Backslash
<code>\uNNNN</code>	Unicode value, where N is a base 16 digit from 0 through F; valid values are <code>\u0000</code> through <code>\uFFFF</code>
<code>\NNN</code>	Value expressed in octal; ranging from <code>\000</code> to <code>\377</code>

The escape sequences can either be used for single characters or within strings of text:

### Using Escape Sequences

```
char c = '\u1234';  
System.out.println("\t\tHello\n\t\tWorld");
```



## 2.4. Constants and the `final` Keyword

Java has a means for defining *constants*, which are like variables in that they have names, but are not changeable once set.

If a variable is declared as `final`, it cannot be changed:

- Even though the variable's value is not changeable once a value has been established, you are allowed to set a unique value *once*.
- Local variables within methods may be declared as `final`.
- Constants' values may be set in an explicit initialization, in a separate line of code, or, as method parameters passed when the method is called.
- Fields within a class may be declared as `final`.

Fields of a class may be declared as `public static final` - that way they are available to other classes, but cannot be changed by those other classes. An example is `Math.PI`.

Classes and methods may also be marked as `final`. We will cover this later.

### Demo 2.1: Java-Basics/Demos/FinalValues.java

---

```
1.  public class FinalValues {
2.
3.      // a constant
4.      final int scale = 100;
5.
6.      // value below is dynamically created,
7.      // but cannot change afterward
8.      final int answer = (int)(Math.random() * scale);
9.
10.     public static void main(String[] args) {
11.         FinalValues fv = new FinalValues();
12.         System.out.println(fv.answer);
13.
14.         // line below would not compile:
15.         // fv.answer = 44;
16.     }
17. }
```

---

## Code Explanation

---

The class has two final fields, `scale` and `answer`. The `scale` is fixed at 100, while the `answer` is initialized dynamically, but, once established, the value cannot be changed. Try removing the comment from the line that attempts to set it to 44, and you will see the compiler error message that results.

---



## 2.5. Mathematics in Java

Looks and behaves like algebra, using variable names and math symbols:

### Some Math Statements

```
int a, b, c, temp;  
a = b/c + temp;  
b = c * (a - b);
```

### ❖ 2.5.1. Basic Rules

Evaluation  
Copy

- What goes on the left of the = sign is called an *lvalue*. Only things that can accept a value can be an *lvalue* (usually this means a variable name - you can't have a calculation like `a + b` in front of the equal sign).
- Math symbols are known as *operators*; they include:

Operator	Purpose (Operation Performed)
+	for addition
-	for subtraction
*	for multiplication
/	for division
%	for modulus (remainder after division)
( <b>and</b> )	for enclosing a calculation

- Note that integer division results in an integer - any remainder is discarded.

## ❖ 2.5.2. Expressions

An expression is anything that can be evaluated to produce a value. Every expression yields a value.

Examples (note that the first few of these are not complete statements):

Two simple expressions:

```
a + 5  
5/c
```

An expression that contains another expression inside, the  $(5/c)$  part:

```
b + (5/c)
```

A statement is an expression; this one that contains another expression inside - the  $b + (5/c)$  part, which itself contains an expression inside it (the  $5/c$  part):

```
a = b + (5/c);
```

Since an *assignment* statement (using the = sign) is an expression, it also yields a value (the value stored is considered the result of the expression), which allows things like this:

```
d = a = b + (5/c);
```

- The value stored in  $a$  is the value of the expression  $b + (5/c)$ .
- Since an assignment expression's value is the value that was assigned, the same value is then stored in  $d$ .
- The order of processing is as follows:
  1. Retrieve the value of  $b$ .
  2. Retrieve the  $5$  stored somewhere in memory by the compiler.
  3. Retrieve the value of  $c$ .
  4. perform  $5 / c$
  5. Add the held value of  $b$  to the result of the above step.

6. Store that value into the memory location for a.
7. Store that same value into the memory location for d.

Here is a moderately complicated expression; let's say that a, b, and c are all double variables, and that a is 5.0, b is 10.0, and c is 20.0:

```
d = a + b * Math.sqrt(c + 5);
```

1. Since the `c + 5` is in parentheses, the compiler creates code to evaluate that first, **but** to perform the `c + 5` operation, both elements must be the same type of data, so the thing the compiler creates is a conversion for the 5 to 5.0 as a double.

```
d = a + b * Math.sqrt(c + 5.0);
```

2. Then the compiler creates code to evaluate `20.0 + 5.0` (at runtime it would become `25.0`), reducing the expression to:

```
d = a + b * Math.sqrt(25.0);
```

3. Next, the compiler adds code to call the `Math.sqrt` method to evaluate its result, which will be 5.0, so the expression reduces to:

```
d = a + b * 5.0;
```

4. Note that the evaluated result of a method is known as the *return value*, or *value returned*.
5. Multiplication gets done before addition, the compiler creates that code next, to reduce the expression to:

```
d = a + 50.0;
```

6. Then the code will perform the addition, yielding:

```
d = 55.0;
```

7. And finally, the assignment is performed so that 55.0 is stored in d.

As implied by the examples we have seen so far, the order of evaluation of a complex expression is not necessarily from left to right. There is a concept called *operator precedence* that defines the order of operations.

### ❖ 2.5.3. Operator Precedence

Operator precedence specifies the order of evaluation of an expression.

Every language has a “table of operator precedence” that is fairly long and complex, but most languages follow the same general rules:

- Anything in parentheses gets evaluated before what is outside the parentheses.
- Multiply or divide get done before add and subtract.

#### Example

- In the expression  $a = b + 5/c$ , the  $5/c$  gets calculated first, then the result is added to  $b$ , then the overall result is stored in  $a$ .
- The equal sign ( $=$ ) is an operator; where on the table do you think it is located?

The basic rule programmers follow is: when in doubt about the order of precedence, use parentheses.

Try the following program:

#### Demo 2.2: Java-Basics/Demos/ExpressionExample.java

---

```
1. public class ExpressionExample {
2.     public static void main(String[] args) {
3.         double a = 5.0, b = 10.0, c = 20.0;
4.         System.out.println("a+b is " + (a + b));
5.         System.out.println("a+b/c is " + (a + b / c));
6.         System.out.println("a*b+c is " + (a * b + c));
7.         System.out.println("b/a+c/a is " + (b / a + c / a));
8.     }
9. }
```

---

### ❖ 2.5.4. Multiple Assignments

Every expression has a value. For an assignment expression, the value assigned is the expression’s overall value. This enables chaining of assignments:

`x = y = z + 1;` is the same as `y = z + 1; x = y;`

`i = (j = k + 1)/2;` is the same as `j = k + 1; i = j/2;`

Quite often, you may need to calculate a value involved in a test, but also store the value for later use:

#### **Storing a Result And Continuing an Operation**

```
double x;  
if ( (x = Math.random()) < 0.5 ) {  
    System.out.println(x);  
}
```

You might wonder why not just generate the random number when we declare `x`? In this case, that would make sense, but in a loop the approach shown above might be easier such that:

- generates the random number and stores it in `x`.
- as an expression, that results in the same value, which is then tested for less than `0.5`, and the loop body executes if that is true.
- the value of `x` is then available within the block.
- after the loop body executes, another random number is generated as the process repeats.

It is usually not necessary to code this way, but you will see it often.

### ❖ 2.5.5. Order of Evaluation

The order of operand evaluation is always left to right, regardless of the precedence of the operators involved.

## Demo 2.3: Java-Basics/Demos/EvaluationOrder.java

---

```
1.  public class EvaluationOrder {
2.      public static int getA() {
3.          System.out.println("getA is 2");
4.          return 2;
5.      }
6.      public static int getB() {
7.          System.out.println("getB is 3");
8.          return 3;
9.      }
10.     public static int getC() {
11.         System.out.println("getC is 4");
12.         return 4;
13.     }
14.     public static void main(String[] args) {
15.         int x = getA() + getB() * getC();
16.         System.out.println("x = " + x);
17.     }
18. }
```

---

### Code Explanation

---

The operands are first evaluated in left to right order, so that the functions are called in the order `getA()`, then `getB()`, and, lastly, `getC()`

But, the returned values are combined together by multiplying the results of `getB()` and `getC()`, and then adding the result from `getA()`

---

## ❖ 2.5.6. Bitwise Operators

Java has a number of operators for working with the individual bits within a value.

Operator	Example	Effect	Bit Pattern
&	Bitwise AND, combines individual bits with an AND operation, so that in the resulting value, a bit position is only 1 if that position had a 1 for both operands.		
	<code>int a = 2;</code>	has the 2 bit set	<code>0...00000010</code>
	<code>int b = 6;</code>	has the 2 and 4 bits set	<code>0...00000110</code>
	<code>int c = a &amp; b;</code>	results in 2	<code>0...00000010</code>
	Bitwise OR, combines individual bits with an OR operation, so that in the resulting value, a bit position is 1 if that position had a 1 for either operand.		
	<code>int a = 2;</code>	has the 2 bit set	<code>0...00000010</code>
	<code>int b = 4;</code>	has the 4 bit set	<code>0...00000100</code>
	<code>int c = a   b;</code>	results in 6	<code>0...00000110</code>
^	Bitwise exclusive OR (XOR), combines individual bits so that any position that is the same in both operands yields a 0, any bits that differ yield a 1 in that position; this is often used in encryption, since repeating the operation on the result yields the original value again		
	<code>int a = 3;</code>	has the 1 and 2 bits set	<code>0...00000011</code>
	<code>int b = 6;</code>	has the 2 and 4 bits set	<code>0...00000110</code>
	<code>int c = a ^ b;</code>	results in 5	<code>0...00000101</code>
	<code>int d = c ^ b;</code>	results in 3 again	<code>0...00000011</code>
~	Bitwise complement. Reverses the state of all bits.		
	<code>int a = 0;</code>	has no bits set	<code>0...00000000</code>
	<code>int b = ~a;</code>	has all bits set	<code>1...11111111</code>

## Bitwise Shift Operators

These operators shift the bits left or right within a 32-bit `int` value (they do not work with any other type).

Operator	Description		
Example	Effect	Bit Pattern	
<code>&lt;&lt;</code>	left shift the bits by the second operand		
<code>int a = 4;</code>	has the 4 bit set	<code>0...00000100</code>	
<code>int b = a &lt;&lt; 2;</code>	now has the 16 bit set	<code>0...00001000</code>	
<code>&gt;&gt;</code>	right shift the bits by the second operand with <i>sign-extension</i> (if the first bit is a 1, new bits introduced to fill in on the left come in as 1 bits)		
<code>int a = -126;</code>	has all bits except the rightmost set to 1	<code>10...0000010</code>	
<code>int b = a &gt;&gt; 1;</code>	now has all bits set to 1 (the 0 rolled off the right end, and a 1 was added on the left to match the original leftmost bit; the resulting value is 63)	<code>110...000001</code>	
<code>&gt;&gt;&gt;</code>	right shift the bits by the second operand without sign-extension (bits added on the left always come in as 0)		
	<code>int a = -1;</code>	has all bits set to 1	<code>11...1111111</code>
	<code>int b = a &gt;&gt;&gt; 31;</code>	now has all bits set to 0 except the rightmost (all bits except the first rolled off the right end, and 0's were added on the left)	<code>0000000...01</code>

You can try out the bitwise operators and bitwise shift operators with the file `Java-Basics/Demos/Bitwise.java`.

## ❖ 2.5.7. Compound Operators

Combine multiple effects in one operation: calculation and storage into memory

Operator	Purpose (Operation Performed)
++	increment a variable
--	decrement a variable; note that ++ and -- can precede or follow a variable
	if preceding (called <i>prefix</i> ), apply the operator and then use the resulting value
	if following (called <i>postfix</i> ), retrieve the value of the variable first, use it as the result of the expression, and then apply the operator
+=	add an amount to a variable
-=	subtract an amount from a variable
*=	multiply a variable by an amount
/=	divide a variable by an amount
%=	set variable equal to remainder after division by an amount
&=	perform bitwise AND between left and right, store result into left operand
=	perform bitwise OR between left and right, store result into left operand
^=	perform bitwise XOR between left and right, store result into left operand
>>=	shift the variable's bits to the right by an amount with sign-extension
>>>=	shift the variable's bits to the right by an amount without sign-extension
<<=	shift the variable's bits to the left by an amount

### Compound Operator Examples

Statement	Result
<code>i++;</code>	increment <code>i</code>
<code>i--;</code>	decrement <code>i</code>
<code>j = i++;</code>	retrieve current value of <code>i</code> , hold it as the result of the expression, assign its value to <code>j</code> , then increment <code>i</code>
<code>j = ++i;</code>	increment <code>i</code> , then <code>j = i;</code>
<code>x *= 3;</code>	<code>x = x * 3;</code>
<code>y -= z + 5;</code>	<code>y = y - (z + 5);</code>

It is inevitable that a certification exam will ask a question that requires understanding the steps involved in a postfix increment or decrement: try the following program:

### Demo 2.4: Java-Basics/Demos/IncrementTest.java

---

```
1. public class IncrementTest {
2.     public static void main(String[] args) {
3.         int i = 0, j;
4.
5.         i = i++;
6.         System.out.println("i = " + i);
7.
8.         j = i++ + i;
9.         System.out.println("i = " + i + ", j = " + j);
10.    }
11. }
```

---

#### Code Explanation

---

- In the first statement, `i = i++;`, the original value of `i` (0) is retrieved and held. Then, `i` is incremented. But, after that, the held value of 0 is put back into `i`, overwriting the 1!
  - In the second part, `j = i++ + i;`, the operands are evaluated from left to right, but each operand is *fully evaluated* before proceeding to the next. Which means that the increment part of `i++` has taken effect before the second appearance of `i` in the equation, so that the expression reduces to `0 + 1` (recall that the previous operation resulted in `i` being 0).
- 

### ❖ 2.5.8. Expressions that Mix Data Types: Typecasting

Expressions will often mix different types of data, for example:

```
double x = 15.6 / 4;
double y = 12.2 + 15 / 4;
```

The compiler must choose a specific type of data (either integer or floating-point, and the specific size) for each individual expression it evaluates within a statement.

- In general, the processor will choose the larger or more complex unit to work with.

- In the first case the value 4 will be *promoted* to a `double`, which will then be used in the division.
- The conversion is done *one expression at a time* within a statement; so, in the second example, the division will be performed with integer math, resulting in a value of 3, which will then be promoted to a `double` to finish the statement.

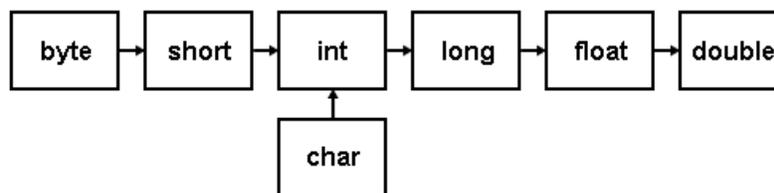
The process of changing the data type of a value is known as a *typecasting* (or *casting*):

- A *widening cast* converts a smaller or less precise value into a larger or more precise type - this is done automatically (the compiler does an *implicit cast*, as above).
- A *narrowing cast* converts a larger or more precise value into a smaller or less precise type (such as from `double` to `int`) - this must be coded with an *explicit cast*.
- To code an explicit typecast, put the desired type in parentheses in front of the expression to be converted; for example:

```
int a = (int) (15.6 / 4);
```

- The division will be evaluated as before, meaning the 4 is promoted to a `double`, and the result of the division is a `double`, but a `double` can't be stored in `a`, so the cast to `int` is necessary.
- Note that casting is an operation, therefore it has a precedence (which is fairly high on the operator precedence table).

The allowable sequence of implicit casts is shown below:



## Small Integral Types and Expressions

One aspect of Java that is important to understand is the result of expressions involving the small integral types: `byte`, `char`, and `short` any expression involving these types,

other than the compound assignment expressions, is done with `int` values, so that the result will always be an `int`.

Try the following (it is the same as the earlier postfix increment example, using `byte` instead of `int`):

```
byte i = 0, j = 6, k;  
i++;  
k = i + j;  
System.out.println("i = " + i + ", j = " + j);
```

#### Note

The increment expression is accepted by the compiler; it is the simple addition in the third line that causes an error.

The Java Language Specification states the promotion rules as follows (note that this concept does not apply to all operators; again, the operators that include assignment do not use it):

When an operator applies *binary numeric promotion* to a pair of operands, each of which must denote a value of a numeric type, the following rules apply, in order, using widening conversions to convert operands as necessary:

- If either operand is of type `double`, the other is converted to `double`.
- Otherwise, if either operand is of type `float`, the other is converted to `float`.
- Otherwise, if either operand is of type `long`, the other is converted to `long`.
- Otherwise, both operands are converted to type `int`.



## 2.6. Creating and Using Methods

The *method* is the basic complete unit of code to perform one task within an object:

- In procedural languages, these are called functions, but in object-oriented programming (OOP) languages like Java they are usually called methods.

- Almost all executable code is in some method.
- Technically, there is one other place an object-oriented program could have executable code, but that is an advanced topic.
- Examples: calculate a trigonometry function like cosine, print data on the screen, read from a file, open a window.
- Methods are defined by a name followed by parentheses.
- Inputs to the method go inside the parentheses; they are called *parameters* or *arguments* to the method.

Using a method in your code, causing it to run, is known as *calling* the method.

A method call is an expression, so it may result in a value (the method call is evaluated like any other expression).

- this value is called the *return value*, and the method is said to return a value.
- the following is an example of a method used as an expression within a larger expression:

```
z = Math.sin(Math.PI / Math.sqrt(x));
```

Evaluation  
Copy

Methods must be called with arguments that match their specified form, known as the *function signature*:

- the signature is the combination of the name of the method with the pattern of its parameters.
- the documentation for library-supplied methods will tell you the signature for those methods.
- when you call a method, the name of the method and the parameters you pass to it will determine which method will be called, based on the available signatures.
- for each argument, Java expects a value - which could be a literal value, a variable, or expression; it must be either the correct type or a value that can be implicitly typecast to the correct type.

All methods in Java must be defined within a class definition. They have complete access to all other elements of the class (fields and other methods, regardless of the access modifier used for that element).

## Demo 2.5: Java-Basics/Demos/UseMethodsExample.java

---

```
1. public class UseMethodsExample {
2.     public static void main(String[] args) {
3.         int x = 16;
4.         double y = Math.sqrt(x);
5.         System.out.println("Square root of " + x + " is " + y);
6.
7.         double z = Math.sin(Math.PI / 2);
8.         System.out.println("Sine of pi/2 is " + z);
9.     }
10. }
```

---

### Code Explanation

---

This class calls two methods from the `Math` class: `sqrt` and `sin`, both of which expect one parameter which is a `double`.

When we call `sqrt` and pass an integer, the compiler converts that to a `double` to provide the type of data that `sqrt` expects

---

Note that even if your program does not call any methods, it has one method that will be called: `main()`. Things to note:

- The definition of `main()` is that it provides a start point and end point for your program.
- Program execution starts with the first statement in `main()`.
- If the last statement in `main()` is executed, the program ends.
- The `main()` does have arguments (any other words typed on the command line).

### ❖ 2.6.1. Creating Methods

There are three things you need to decide for any method:

1. what it does.
2. what inputs it needs.
3. what answer it gives back.

## general format for a method

```
        [modifiers] dataType methodName(parameterList) {  
methodBody  
    return result;  
}
```

- Where emphasized words are concepts; fixed-font words are keywords, and brackets [ ] indicate optional elements:
- *modifiers* include the accessibility of this method from outside the class (`public`, `private`, `protected`, or left blank) as well as other possibilities listed earlier in this section.
- *dataType* is a type of data, like `int`.
- *methodName* is the name of your method.
- *parameterList* is a comma-separated list of parameters; each parameter is listed with its data type first, then its name.
- *methodBody* is the set of statements that perform the task of the method.
- `return` is a keyword that says to use the *result* expression value as the result of calling the method, and send that value back to the code that called the method (as an expression, a call to a method evaluates to the returned value; i.e., calling `Math.sqrt(4)` evaluates to the value that `sqrt` returned).
- if a method declares that it returns a value of a certain type, then it must explicitly return a value of that type.

## Method Examples

### A Simple Method

```
public void sayHello() {  
    System.out.println("Hello");  
}
```

- `public` methods are accessible to any other class.
- `void` is a keyword for a returned data type that means no data at all - this method does not calculate a result.
- The name of the method is `sayHello`.

- This method could be called as follows:

Code	When Used
<code>sayHello();</code>	from within the class
<code>x.sayHello();</code>	from outside the class, for an instance x of this class

#### **Method with a Parameter**

```
public void showNumber(int number) {
    System.out.println("The number is: " + number);
}
```

- again, this method does not calculate a result.
- this method could be called as follows:

Code	When Used
<code>showNumber(5);</code>	from within the class
<code>x.showNumber(5);</code>	from outside the class, for an instance x of this class

#### **Method with Parameters and a Return Value**

```
public int calculateSum(int num1, int num2) {
    int answer = num1 + num2;
    return answer;
}
```

- This method does calculate a result; the data type it calculates is `int`.
- This method must be given two parameters, both of which are `int` data types.
- This method could be called as follows:

Code	When Used
<code>int value = calculateSum(4, 6);</code>	from within the class
<code>int value = x.calculateSum(4, 6);</code>	from outside the class, for an instance x of this class

## Return Values

Note that the listing of a data type word in front of a name is something we have seen before, in declaring variables. The purpose is the same - to state what type of data this element provides when used in an expression.

The first statement below states that the value of `a` is an `int`, so that the compiler knows what memory size to allocate and what type of processing to use with it when it evaluates the second statement.

```
int a;  
int b = a / 2;
```

The effect is no different using a method; recall the function signature (the first line) of our `calculateSum` method:

```
public int calculateSum(int num1, int num2)
```

It states that the result of evaluating `calculateSum` is an `int`, so that the compiler knows what memory size to allocate for the result, and what type of processing to use with it when it evaluates a statement like:

```
int b = calculateSum(4, 6) / 2;
```

When this statement gets processed, the `calculateSum` method will run and return its result (which will be the value 10 as an `int`).

Thus the statement in effect is reduced to:

```
int b = 10 / 2;
```

## Method Parameters

Method parameters are also declarations. They declare a type of data received by the method, and also provide a name for each value so it can be used within the method.

Back to our method:

```
public int calculateSum(int num1, int num2) {  
    int answer = num1 + num2;  
    return answer;  
}
```

- The parameter list declares two variables that will exist in this method: num1 and num2.
- The order is important - num1 will be the first value given to the method, num2 will be the second.
- The difference between method parameter declarations and the variable declarations we saw before is that the method parameters receive their values when the method is called.



## 2.7. Variable Scope

The variable scope rules are similar to those in C++.

Variables can be declared at any point in your code, not just at the top of a block:

- *Local variables* (those within methods) are not visible to any code that precedes the declaration.
- Object elements are visible to any code within the object, regardless of the order of declaration.

Variables declared within a set of curly braces cease to exist after the closing brace has been reached (it is said that they go *out of scope*); therefore, local variables exist only within that method.

Variables can be declared in the control portion of a for loop, and will exist for the duration of the loop

Parameters to a method are local variables within that method

It is legal to use the same variable name in different scopes, as long as the two scopes have no irresolvable conflicts. Some explanation:

- **Non-overlapping scopes.** For example, two different methods could each have a local variable called `firstName`.

- **Overlapping scopes.** It is valid for a method to have a variable whose name conflicts with a property name for that class - in that case, the local variable *hides* the property, but there is a special syntax that allows the method to access the property; we will cover this later.
- It is legal for a method and a field of a class to have the same name, although it is not considered a good practice to do so.
- An example of an irresolvable conflict is declaring a local variable within a block when the same name is already in scope in that method as a local variable.

## Demo 2.6: Java-Basics/Demos/MethodExample.java

---

```
1. public class MethodExample {
2.     public static void sayHello() {
3.         System.out.println("Hello");
4.     }
5.     public static void showNumber(int number) {
6.         System.out.println("The number is: " + number);
7.     }
8.     public static int calculateSum(int num1, int num2) {
9.         int answer = num1 + num2;
10.        return answer;
11.    }
12.    public static void main(String[] args) {
13.        sayHello();
14.        showNumber(5);
15.        int b = calculateSum(4, 6);
16.        System.out.println("The sum of 4 and 6 is:" + b);
17.        b = calculateSum(4, 6) / 2;
18.        System.out.println("That divided by 2 is: " + b);
19.    }
20. }
```

---

## Exercise 2: Method Exercise

 5 to 10 minutes

---

1. Open the file `Java-Basics/Exercises/Calculator.java`.
2. In it create a method called `add` that accepts two double parameters, adds them together, and returns the result - mark it as `public` and `static`, and returning a `double`.
3. In a similar manner, create methods called `subtract`, `multiply`, and `divide`.
4. In `main`, create three double variables `a`, `b`, and `c`, initialized with values of your choice, and `c`, which we can leave uninitialized.
5. Call each of your methods in turn, passing `a` and `b`, and accepting the returned value into `c`, and printing all three values.



## Solution: Java-Basics/Solutions/Calculator.java

---

```
1. public class Calculator {
2.     public static void main(String[] args) {
3.         double a = 10.0, b = 20.0, c;
4.         c = add(a, b);
5.         System.out.println(a + " + " + b + " = " + c);
6.         c = subtract(a, b);
7.         System.out.println(a + " - " + b + " = " + c);
8.         c = multiply(a, b);
9.         System.out.println(a + " * " + b + " = " + c);
10.        c = divide(a, b);
11.        System.out.println(a + " / " + b + " = " + c);
12.    }
13.
14.    public static double add(double x, double y) {
15.        return x + y;
16.    }
17.    public static double subtract(double x, double y) {
18.        return x - y;
19.    }
20.    public static double multiply(double x, double y) {
21.        return x * y;
22.    }
23.    public static double divide(double x, double y) {
24.        return x / y;
25.    }
26. }
```

---

## Conclusion

In this lesson you have learned:

- About Java's basic syntax rules, variables and their declaration, and primitive types.
- To write simple statements using variables and operators.
- To understand the rules that apply when data is converted from one type to another.
- To declare, write, and use simple methods.

# LESSON 3

## Java Objects

---

### Topics Covered

- ☑ General Object-Oriented concepts.
- ☑ Declaring object classes.
- ☑ Defining methods and fields.
- ☑ Creating and using instances of classes.
- ☑ Accessing objects using reference variables.
- ☑ Defining different protection levels for elements.
- ☑ Defining constructors.
- ☑ Overloading methods and constructors.
- ☑ The `this` self-reference.
- ☑ Declaring and using `static` elements.
- ☑ Creating packages of classes.

Evaluation  
Copy

### Introduction

In this lesson, you will learn general Object-Oriented concepts, to declare object classes, defining methods and fields, to create instances of objects and use them in a program, to access objects using reference variables, to define different protection levels for elements, to define constructors, to overload methods and constructors, to use the `this` self-reference, to declare and use static elements, and to create packages of classes.



### 3.1. Objects

In general, the concept of an object is: *a collection of data along with the functions to work with that data*

- as opposed to purely procedural languages where the data and functions are separate, and the data is passed to the function to work with

In Java, all code must be written inside object definitions

- a *class* is an object definition, containing the data and function elements necessary to create an object
- an *instance* of an object is one object created from the class definition (the process is known as *instantiation*)
- the data and function elements are known as *members*
- data members are also known as *fields*, *properties*, or *attributes*, and function members as *methods*

Methods can work with any fields of an object, as well as any other method.

Conceptually, data members (fields) are like *nouns* and methods (functions) are like *verbs*. Data members define the attributes or state that a given class can hold information about. A Person class might have data members for `firstName`, `lastName`, `age`, etc. A BankAccount class could store info like `accountBalance`, `interestRate`, or `accountOwner` - where data member `accountOwner` might itself be an instance of a class like Person.

Members define what the class can do - in particular how instances of the class can communicate with other objects. An instance of class Person might have methods that returns the person's full name (the `firstName` field concatenated with a space and the `lastName`) or to set the person's age. An instance of BankAccount might have methods that allow for withdrawing funds (reducing the `accountBalance`) or for returning the Person who is the owner of the account.

A *constructor* is a function that defines the steps necessary to instantiate one object of that class

- if you do not explicitly create one, a default constructor that does nothing will be created automatically

Each class should generally be defined in its own file if it is to be used by other classes

- the file name must match the name of the class, plus a `.java` extension

The data members of a class may be any type of data, objects or *primitives*

It is traditional in Java that only class names begin with capital letters

- variable names, both object and primitive, begin with lowercase letters, as do method names
- multi-word names, like first name, are done in *Camel-case* (as in `firstName`)



## 3.2. Object-oriented Languages

For a whirlwind tour of Object-Oriented Programming (OOP), an object-oriented language is supposed to include three major concepts:

*Encapsulation* - that data and the functions to work with that data are contained within the same entity; encapsulation includes another concept: data hiding or protection - where direct access to data is restricted in favor of methods that manipulate the data

*Polymorphism* - that there can be different forms of the same thing, so that you could have one variable that could store several different types of objects; or a method that accepts no arguments and another method with the exact same name that accepts an integer argument (and maybe another that accepts a string and a double-precision argument, etc.).

*Inheritance* - that an object definition can use another object definition as a starting point and build upon that base object (in which case the original object definition is called the *base class*, *parent class*, or *superclass* and the new class is called the *derived class*, the *child class*, or *subclass*, respectively) - note that a derived class can be used as a base for further inheritance, creating a chain.

Java uses all three of these concepts.

Java does not allow any code that is not part of an object - all code in your program must be inside a class definition.

Java forces inheritance on every class - if you do not explicitly inherit from a class, then by default Java assumes that you are inheriting from the class called `Object` (which does not do much, but does have several useful methods).

- Every class is descended from `Object`, since whatever class you inherit from must have inherited from something, which would be either `Object` or another class that inherited from another class, etc., the ultimate parent of which is the `Object` class.

- The concept of polymorphism implies that you could store any type of object in a variable whose type was `Object`.



### 3.3. Object-oriented Programs

An object-oriented program replaces the concept of a linear sequence of steps with steps that create instances of classes, connect them together, and set them to communicate with each other.

For example, in an HR/Payroll application, we might have object classes that represent employees, dependents, checks, departments, financial accounts, etc.

The act of creating a check for an employee could trigger a number of actions:

- Update the employee's accumulated pay, etc., fields - done in an employee object;
- Update accumulated department totals - performed in a department object;
- Debit a financial account - done in an account object;
- Cause a physical check to be printed - a check object would be created encapsulating the information, and sent to a printing module (in some sort of object dedicated to similar generic operations);
- Upon successful printing of the check, notifications would be sent to the above objects to finalize their operations.

Encapsulation is the concept that the data and operations that work on or with that data are bundled together into one unit - the object.

Objects are said to have *state* and *behavior*.

- The state is the current set of values for all the data values.
- Terms commonly used to describe the state values are:
  - *data members*
  - *fields*
  - *properties*
  - *attributes*

- The behavior is the set of operations that the object can perform (the functional code for the object). There are commonly called *methods*, but also sometimes called *functions*.



## 3.4. Encapsulation

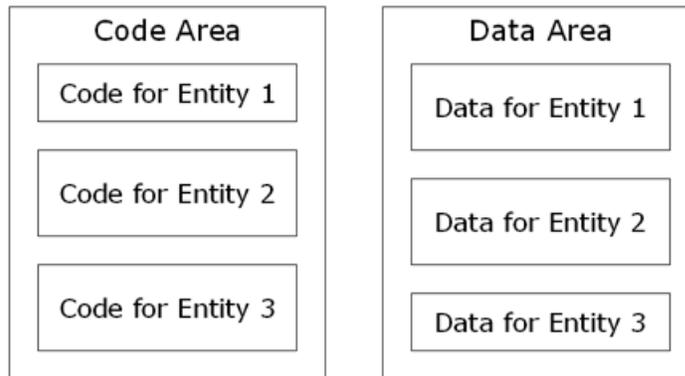
In a procedural program, the executable code is usually kept in a separate area from the data values.

- In general, the programmer will call the code in the entity where its data is located.

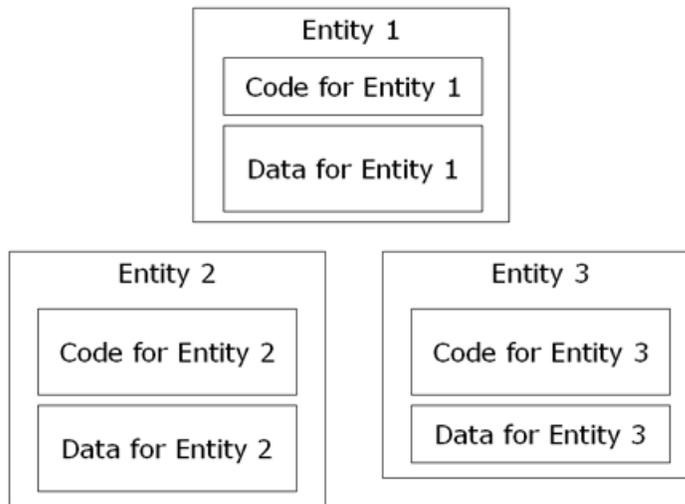
In an object-oriented program, it will appear to the programmer that the data values are kept with the executable code.

- Each object has a memory area, which can be viewed as having a data block and a code block (note that this is a dramatic simplification - in practice the data is indeed kept separate from the code, but the compiler and/or runtime environment handles the details necessary to match an object's code with its data).
- An entity's code automatically knows where its data is.

### Memory Organization in a Procedural Program



### Apparent Memory Organization in an Object-Oriented Program



Encapsulation is often said to include the concept of *data hiding*, where the data elements of an object are not directly visible to the outside world. Instead, methods are provided to retrieve and set the values of the data elements.

- Forcing external code to go through code inside the class allows for *validation* of data; for example, if a field must have a value that is greater than or equal to zero, the method to set that field can perform a test on incoming data to ensure that it is valid

Consider our hypothetical `BankAccount` class from above: we certainly wouldn't want external code to have direct access to the `accountBalance` field. A request to a `BankAccount`

object to make a withdrawal should certainly first make sure that the requested amount to withdraw were not more than the value of `accountBalance`.

### ❖ 3.4.1. OOP as a Messaging System

One way to view an object-oriented environment is as a messaging system. Objects can send each other messages by calling their methods. In the following example, the code sends object `p` three messages, first to set its first name value to “James”, then to set its last name to “Berger”, and finally to ask it what the full name is (the result of which it passes on in a message to `System.out`, requesting that it be printed).

```
Person p = new Person();  
p.setFirstName("James");  
p.setLastName("Berger");  
System.out.println(p.getFullName());
```

## Exercise 3: Object Definition

 5 to 15 minutes

---

We would like to develop a simple employee database system that will include the ability to pay an employee (print a check for them, and processing necessary for bookkeeping). As a first step, we will define a type of object to represent an employee.

1. Make a list of what data fields we would be likely to need (names and the type of data they would hold)
2. In a separate list, what methods might be associated with an employee's data?

For this exercise, you are writing *pseudocode* - code not meant to be compiled, but rather a sketch or outline of the fields and methods we will need going forward.



## Solution: Java-Objects/Solutions/ObjectDefinition/Employee.java

---

```
1.  public class Employee {
2.
3.  /*
4.  Possible elements of an employee class include:
5.
6.  Data fields:
7.
8.      String first name
9.      String last name
10.     String middle name or initial
11.     String social security number
12.     int employee id
13.     Date hire date
14.     Date date of birth
15.     char? byte? gender
16.     char? byte? employee type
17.     int department
18.     double periodic pay amount
19.     double ytd amounts for pay and all withholdings
20.     char? byte? benefits plan selection
21.     String employee review information
22.
23.
24.  Methods:
25.
26.     get and set values for all properties
27.     pay them
28.     print various reports or report lines
29.
30.
31.  */
32.
33. }
```



---

### Code Explanation

---

The above is only a partial listing of what could very likely be a large and complex class in a real application.

---



## 3.5. References

When you create an object, it occupies some memory location that the JVM allocates at runtime.

If you then have a reference and set it equal to that object, the variable's memory location stores the numeric address of the memory location where the object is stored (often it is said that it points to the object's memory location; it is more appropriate to say that it references the object).

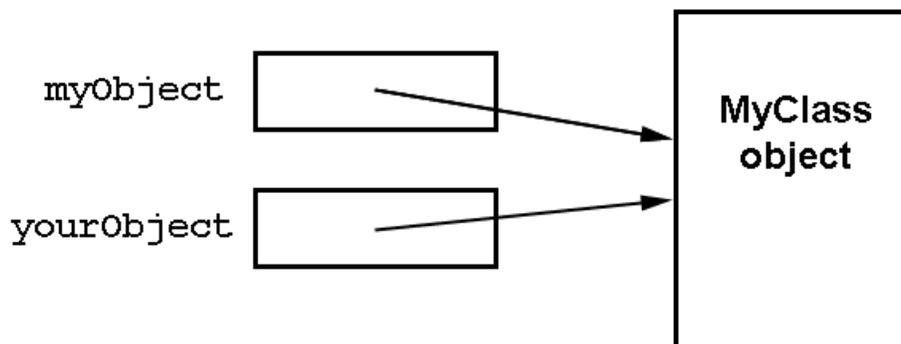
If you set one object variable equal to another, you do not have two copies of the object - you have one object and two variables that point to that object.

If you had the following:

```
MyClass myObject = new MyClass();  
MyClass yourObject = myObject;
```

You would not end up with two copies of the same data, you would end up with two references to the same object (since all you really did was store another copy of the original object's memory address)

So, if you then changed the properties of `myObject`, the properties as seen through `yourObject` would change as well.



Note: there is a `clone()` method available to copy an object's contents to a new object. The `clone()` method is inherited from the `Object` class. It's overridden in the JDK base classes so it returns a copy as you would expect. It's not appropriate for creating copies

of Employee or any classes associated with our exercises until we cover the Object class later in the course.

Although this is a technicality, it is an important one: certain Java classes are *immutable*, or *final*; that is, they cannot be changed once they are created. Note that:

- String objects are immutable, as are the *wrapper classes* that hold primitive data types inside objects.
- You can still set a String reference variable to a new value, but what you are doing is setting the reference to point to a different object.
- The important point to note is that if there were other references to the original String, they would be unchanged - this follows from how references work, but is counter to how some languages work.

### ❖ 3.5.1. Reference Example

In the following code, the new keyword creates an object, in this case a new StringBuffer object.

#### **Demo 3.1: Java-Objects/Demos/References.java**

---

```
1. public class References {
2.     public static void main(String[] args) {
3.         StringBuffer sb1 = new StringBuffer("Hello");
4.         StringBuffer sb2 = null;
5.         sb2 = sb1;
6.         sb2.append(" World");
7.         System.out.println(sb1);
8.     }
9. }
```

---

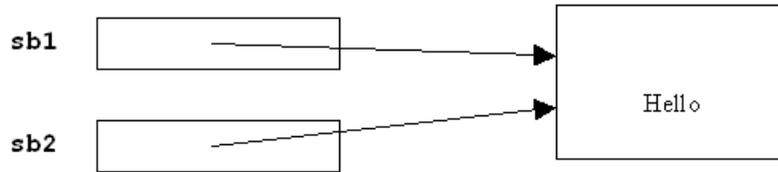
#### Code Explanation

---

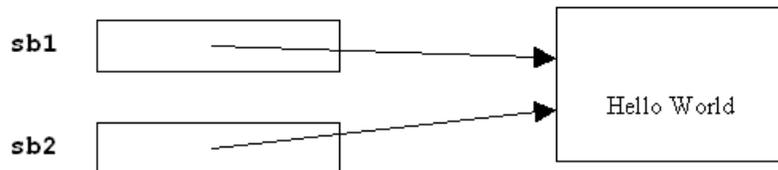
The first line creates a StringBuffer object containing the text Hello, then creates a reference variable called sb1 that points to that new object.

The second line creates a reference variable called sb2 that points to nothing (it is set to null).

The third line then sets sb2 to point to the same object that sb1 points to.



The last line changes `sb2`; the effect of this change is seen via `sb1` as well.



### ❖ 3.5.2. Reference Expressions

A reference is a simple form of expression. References that result from more complex expressions may be used as well. For, example, with an array of `String` references called `names`, a method for one element can be called in this manner:

#### Array Element Reference Expression

```
names[1].toUpperCase()
```

Similarly, if a method returns a reference, a method may be called directly on the returned value.

#### **Demo 3.2: Java-Objects/Demos/ReferencesChained.java**

```
1. public class ReferencesChained {
2.     public static void main(String[] args) {
3.         String ending = "World";
4.         StringBuffer sb1 = new StringBuffer("Hello");
5.         StringBuffer sb2 = null;
6.         sb2 = sb1;
7.         sb2.append(" ").append(ending);
8.         System.out.println(sb1);
9.     }
10. }
```

## Code Explanation

---

The `StringBuffer.append` method returns a reference to the same `StringBuffer` object, so that another call to `append` or any other method can be *chained* to the result.

---



## 3.6. Defining a Class

The basic syntax for defining a class is:

### **Defining a Class**

```
[modifiers] class ClassName {  
    (field definitions, constructors, and methods go here)  
}
```

Example (for overall format only, we will examine the pieces later):

### **Demo 3.3: Java-Objects/Demos/BookBasic.java**

---

```
1.  public class BookBasic {  
2.      private int itemCode;  
3.      private String title;  
4.      public int getItemCode() {  
5.          return itemCode;  
6.      }  
7.      public void setItemCode (int newItemCode) {  
8.          if (newItemCode > 0) itemCode = newItemCode;  
9.      }  
10.     public String getTitle() {  
11.         return title;  
12.     }  
13.     public void setTitle (String newTitle) {  
14.         title = newTitle;  
15.     }  
16.     public void display() {  
17.         System.out.println(itemCode + " " + title);  
18.     }  
19. }
```

---

Normally the access term for the class should be `public`, which means that the class can be used in other classes.

- A `public` class definition must be in a file with a name that matches the class name, but classes with other access levels do not have that requirement.

### ❖ 3.6.1. Access to Data

Class definitions may have different access levels - this determines where an object of the class may be instantiated or otherwise used.

- Class definitions are usually made freely available to all code, defined as `public`.

All data and function members in a class are available to any code written for that class.

- Note that the normal scope rules for *local* variables apply to Java - that is, a method's internal variables are never available to other methods.

There are four possible access states. Three are declared with access keywords:

1. `public` - the member may be freely accessed by code from any other class.
2. `protected` - the member may not be accessed by code from any other class, except:
  - Classes that *inherit* from this class.
  - Classes in the same *package* (a package is a collection of classes - basically all the classes stored in the same directory).
3. `private` - the member may not be accessed by code in any class, including classes that inherit from this one.

Then there is the fourth possibility, the *default access* used if you use no access word at all. Note that:

- Any code in the same *package* (the same directory) can access the member.
- This type of access is often called *package access*, and is also sometimes called *default* or *friendly access*.

Example - the example shown previously defines a class that will be publicly available, so that any class could instantiate one. The class listed below can use that class in its code, but cannot directly access the `title` or `itemCode` fields, so it must call the `setTitle` and

setItemCode methods to set property values, and getTitle and getItemCode to retrieve values:

### Demo 3.4: Java-Objects/Demos/UseBookBasic.java

---

```
1.  public class UseBookBasic {
2.      public static void main(String[] args) {
3.          BookBasic b = new BookBasic();
4.          b.setItemCode(5011);
5.          b.setTitle("Fishing Explained");
6.          System.out.println(b.getItemCode() + " " + b.getTitle());
7.          System.out.print("From display: ");
8.          b.display();
9.      }
10. }
```

---



## 3.7. More on Access Terms

The access terms control where the name of the item may be written in your code. Note that:

- If the item is `public`, then the name of the item may be written within code for any other class.
- If the item is `private`, then the name of the item may not be written within the code for any other class.
- *Package access* is the default if no access term is specified; this allows the name of the item to be written within code for any class in the same package (which maps to the same directory).
- If the item is `protected`, then the name of the item may not be written within the code for any other class, except one that extends this class, or a class in the same package.

For example:

- If a class definition is `public`, then you can write the name of the class in code in some other class (so you can use the class).
- If it is `private`, then you can't write the name in any other class, so you couldn't use it (in fact, a `private` class definition can only be used in one very special situation, to be covered later).

For another example:

- If the class is `public`, and contains a `public` element, you can write the name of the class in another class, and you can write the name of the element in that class as well - and if the element is `private`, then you can't write its name, so you can't use it.
- So, the `UseBookBasic` class code, which has a `BookBasic` variable `b`, can use `b.getItemCode()` and `b.setItemCode(5011)`, but not `b.itemCode`, since `itemCode` is a data member declared as `private` in class `BookBasic`.

Note: this does not mean that a `private` item cannot be *affected* by the code in another class; it just can't have its name written in that other class.

- The `BookBasic` example shows how public methods can provide controlled access to a private element - in this case to ensure that the value of `itemCode` is never set to a negative value.



## 3.8. Adding Data Members to a Class

Data members are added using an access specifier, a data type keyword, and the field name.

### Adding Data Members

```
[modifiers]
class ClassName {
[modifiers]
    dataType fieldName1, fieldName2, . . . ;
[modifiers]
    dataType fieldName3 = value, . . . ;
(etc.)
}
```

- Note that multiple members of the same type may be defined with one statement.
- Variables may be given initializing values when declared.
- Primitive data elements are initialized to 0 unless explicitly set to a value (and remember that for a boolean value a 0 bit means `false`).

- Member elements that are object reference variables are automatically set to `null` if not given a specific initial value (`null` is a keyword).

Revisiting our `BookBasic` example:

```
class BookBasic {  
    private int itemCode;  
    private String title;  
    . . .  
}
```

- This class has an integer variable `itemCode` and a reference to a `String` object called `title`.
- Neither of these can be directly accessed from outside the class definition, since they are `private`.

### ❖ 3.8.1. Adding Method Members (Functions) to a Class

Methods may be added in a similar manner. Note that in Java, unlike C++, the method body must be defined when the function is *declared* - you can't postpone that until later.

#### Adding Methods

```
[modifiers] class ClassName {  
  
[modifiers]  
    dataType fieldName1, fieldName2, . . . ;  
[modifiers]  
    dataType fieldName3 = value, . . . ;  
  
[modifiers]  
    returnType methodName(paramType paramName, . . . ) {  
  
        [method body code goes here]  
  
    }  
  
    [more methods here]
```

Methods may freely access all data and function members of the class.



## 3.9. Standard Practices for Fields and Methods

It is considered a good practice to have fields be `private`, with access provided by “set and get” methods that are `public`.

Java has a naming convention that, for a field named `value`, there should be methods `setValue` (accepting a parameter of the same type as `value`) and `getValue()` which would return a value whose type is the same as `value`). Note that:

- Get methods do not take any parameters; set methods take one parameter whose type matches the type of the field.
- For `boolean` `true/false` values, the convention also uses `is` and `has` as prefixes for the get methods (as in `isEnabled()` for a field called `enabled`).

It is also considered a good practice to reuse existing methods for any steps that work with data in a way other than simple retrieval. If we had, for example, a method that accepted both `itemCode` and `title`, we would call the `setItemCode()` and `setTitle()` methods rather than access those fields directly. That way if our approach to setting a value changes, the effect is automatically applied through all methods that do that.

```
public void setItemCodeTitle (int newItemCode, String newTitle) {  
    setItemCode(newItemCode);  
    setTitle(newTitle);  
}
```

### ❖ 3.9.1. Order of Elements within a Class

There is no requirement that the elements of a class be in any specific order. Any field or method can be accessed from a method, even if the referenced element is declared later. The compiler makes a pass through the code to identify all the elements, and then goes back through the code again armed with that information.

The most common practice is to list fields first, then constructors, then get/set methods, and then all the other methods. (There is a less commonly held belief that fields should be listed last.) Generally the static fields are listed before the non-static ones.

If there are dependencies between fields due to the value of one field being used to initialize another field, then that would impose an order on those fields.

#### **Field Initializer Dependencies**

```
private int range = 100;
private Random r = new Random();
private int answer = r.nextInt(range);
// answer must come after range and r,
// since it depends on them
```



## 3.10. Java Beans

If you follow the standard naming conventions, you are most of the way to creating a *Java Bean* class. Note that:

- A bean is an object that can be created and used by various tools, both at development time and at runtime.
- Java has a mechanism known as *reflection*, *inspection*, or *introspection* that allows for classes to be instantiated and used without knowing at compile time what the class name is.
  - A bean can be instantiated using a String variable containing the class name.
  - Or a bean can be delivered to an application, for example, across a network connection.
  - At runtime, for any object, its methods and fields can be determined by the JVM, as it inspects the object.
  - Tools can then match field names to get and set methods and treat the fields as *properties* (to this point, we have used the terms fields and properties more or less interchangeably - in bean-speak fields are data values and properties are private fields that can be accessed by set and/or get methods).
- Example uses of beans include:
  - Java-based GUI development tools can create instances of GUI component beans such as text fields, radio buttons, etc., and create a properties table where the programmer can enter values into text fields for values such as

background color, foreground color, text value, etc. (keep in mind that the environment's JVM can inspect the component objects).

- Java Server Pages can instantiate a bean and set and retrieve properties to display in a dynamic web page .
- To be a bean, a class must:
  - Have a constructor that takes no parameters (the object will be created empty, and populated from the set methods - if there was a `Rectangle` class, for example, with height and width properties that were `int`, the tool wouldn't know how to call a constructor that took both parameters (which is the height and which is the width?).
  - Follow the naming convention stated above.
  - A bean can also have additional methods that code with advance knowledge of the class.



## 3.11. Bean Properties

Properties are defined by `public` or `protected` get and set methods, and usually map to `private` fields.

- A *read-only property* would have a get method, but no method to set the value - the property might or might not be *backed by a field*.

## Demo 3.5: Java-Objects/Demos/Rectangle.java

---

```
1.  public class Rectangle {
2.
3.      private int height;
4.      private int width;
5.
6.      public Rectangle() { }
7.
8.      public int getHeight() {
9.          return height;
10.     }
11.
12.     public void setHeight(int newHeight) {
13.         height = newHeight;
14.     }
15.
16.     public int getWidth() {
17.         return width;
18.     }
19.
20.     public void setWidth(int newWidth) {
21.         width = newWidth;
22.     }
23.
24.     public int getArea() {
25.         return height * width;
26.     }
27.
28. }
```

---

### Code Explanation

---

The class has a constructor that takes no parameters, plus methods to get and set the height and width. The area is a read-only property that is not backed by a field - it is calculated each time it is needed.

---

## Demo 3.6: Java-Objects/Demos/UseRectangle.java

---

```
1. public class UseRectangle {
2.
3.     public static void main(String[] args) {
4.         Rectangle r = new Rectangle();
5.         r.setHeight(100);
6.         r.setWidth(250);
7.
8.         System.out.println(
9.             "Rectangle of height " + r.getHeight() +
10.            " and width " + r.getWidth() +
11.            " has an area of " + r.getArea());
12.     }
13.
14. }
```

---

Evaluation  
Copy

### Code Explanation

---

This is a simple main class to create and use a `Rectangle`. A Java Server Page might use a similar approach to populate the height and width from values submitted from a form, and send back a page showing the area.

---

- Less often, you might encounter a *write-only property* - a simplistic example might be a security bean that uses a key to encrypt and decrypt messages: you could set the key, but not retrieve it (and other methods would set an incoming encrypted message and retrieve the decrypted version, or set an outgoing message and retrieve the encrypted result).

# Exercise 4: Payroll01: Creating an Employee Class

🕒 10 to 15 minutes

---

After review with management, it is decided that we will store the following information for each employee:

- Employee ID (an integral number, automatically assigned when a new employee is added).
- First name and last name.
- Department number (an integral value, call it dept).
- Pay rate (a floating-point value, using a `double`).

In addition, we would like:

- A method called `getPayInfo()` that will return a sentence with the employee's name, id, department, and pay rate amount.
  - A method called `getFullName()` that will return the first and last names separated by a space.
1. Open the two files (which are mostly empty) in `Java-Objects/Exercises/Payroll01/`
  2. Define a class called `Employee` with these characteristics, using standard practices for limiting data access and for method naming.
  3. In order to be useful, there should be methods to set and get all properties (except setting the employee id, which will happen automatically in a manner to be determined later; for now, just let it default to 0, but still provide a method to retrieve it).
  4. Create another class called `Payroll` with a main method, which should:
    - Instantiate an `Employee` object.
    - Set values for all the properties (name, department, and pay rate).
    - Call the `getPayInfo()` method to see the results.



## Solution: Java-Objects/Solutions/Payroll01/Employee.java

---

```
1.  public class Employee {
2.      private int id = 0;
3.      private String firstName;
4.      private String lastName;
5.      private int dept;
6.      private double payRate;
7.
8.      public int getId() { return id; }
9.
10.     public String getFirstName() { return firstName; }
11.
12.     public void setFirstName(String fn) {
13.         firstName = fn;
14.     }
15.
16.     public String getLastName() { return lastName; }
17.
18.     public void setLastName(String ln) {
19.         lastName = ln;
20.     }
21.
22.     public int getDept() { return dept; }
23.
24.     public void setDept(int dp) {
25.         dept = dp;
26.     }
27.
28.
29.     public double getPayRate() { return payRate; }
30.
31.     public void setPayRate(double pay) {
32.         payRate = pay;
33.     }
34.
35.     public String getFullName() {
36.         return firstName + " " + lastName;
37.     }
38.
39.     public String getPayInfo() {
40.         return "Employee " + id + " dept " + dept + " " +
41.             getFullName() + " paid " + payRate;
42.     }
43.
44. }
```

## Solution: Java-Objects/Solutions/Payroll01/Payroll.java

---

```
1. public class Payroll {
2.     public static void main(String[] args) {
3.         Employee e1 = new Employee();
4.         e1.setFirstName("John");
5.         e1.setLastName("Doe");
6.         e1.setPayRate(6000.0);
7.         e1.setDept(2);
8.         System.out.println(e1.getPayInfo());
9.     }
10. }
```

---



## 3.12. Constructors

Every class should have at least one method: the *constructor*. This specifies code that must run to instantiate (usually to initialize) a new object from the class definition.

The constructor is a function with the same name as the class.

### Creating a Constructor

```
    [modifiers] class ClassName {
[modifiers]
    ClassName(paramType paramName, . . . ) {
    [code goes here]
    }
}
```

- The name of the constructor is the the same as the name of the class.
- Constructors have no return type and do not return a value.
- They may take input parameters, usually to populate the fields for the new object.
- They may use the standard access keywords - usually they are `public` (but in some circumstances involving inheritance they are sometimes `protected` or `private`).

The methods in an object are available at construction time. While a constructor could set field values directly, it is considered a better practice to call the set methods to store any

values. That way, any validation necessary on the data can be accomplished without duplicating the validating code. The lesson on flow control will cover this in detail.

### Demo 3.7: Java-Objects/Demos/BookWithConstructor.java

---

```
1.  public class BookWithConstructor {
2.
3.      private int itemCode;
4.      private String title;
5.
6.      public BookWithConstructor(int newItemCode, String newTitle) {
7.          setItemCode(newItemCode);
8.          setTitle(newTitle);
9.      }
10.     public int getItemCode() {
11.         return itemCode;
12.     }
13.     public void setItemCode (int newItemCode) {
14.         itemCode = newItemCode;
15.     }
16.     public String getTitle() {
17.         return title;
18.     }
19.     public void setTitle (String newTitle) {
20.         title = newTitle;
21.     }
22.     public void display() {
23.         System.out.println(itemCode + " " + title);
24.     }
25. }
```

---

#### Code Explanation

---

Note how the constructor can make use of the existing methods: the constructor (method `BookWithConstructor`) calls methods `setItemCode` and `setTitle` to set the item code and title, respectively. Thus if there were any validation code - a condition that ensured the item code were greater than 0, say - then that validation code would be written only once (in method `setItemCode`) and reused by the constructor. This strategy makes for code that is easier to maintain.

---

## Demo 3.8: Java-Objects/Demos/UseBookWithConstructor.java

---

```
1. public class UseBookWithConstructor {
2.     public static void main(String[] args) {
3.         BookWithConstructor b =
4.             new BookWithConstructor(5011, "Fishing Explained");
5.         b.display();
6.     }
7. }
```

---

### Code Explanation

---

This class creates an object of class `BookWithConstructor` by using the `new` keyword and the `BookWithConstructor` constructor.

---



## 3.13. Instantiating Objects Revisited

You instantiate an object from a class using the `new` keyword.

### Instantiating a New Object

```
objectReferenceVariable = new ClassName(parameters);
```

- Note that the type of the object reference variable must match the type of the class being instantiated (or, for later, match a parent of that class).
- You can pass parameters as long as the parameter list matches a constructor for that class.
- Note that while the object reference is often stored in a variable, it does not have to be - an object can be instantiated in any situation where an object reference could be used, for example:

```
new BookWithConstructor(1234, "Help is on the Way").display();
System.out.println(new Integer(5));
```

When a new instance is created, the following sequence of events takes place (note that this includes only concepts we have covered thus far; later, we will expand the sequence as we cover additional topics):

1. Memory is allocated in an appropriately sized block.
2. The entire block is set to binary zeros (so every field is implicitly initialized to 0).
3. Explicit initializations of fields take place.
4. The constructor runs.
5. The expression that created the instance evaluates to the address of the block.

---

\*

### 3.14. Important Note on Constructors

If you create no constructors at all, you will get a default constructor that accepts no arguments and does nothing (but at least you can use it to instantiate an object).

If you write a constructor that accepts arguments, then you lose the default constructor. Note that:

- If you still want to have a no-argument constructor that does nothing, you must explicitly write it.
- Try modifying `UseBookWithConstructor.java` to add line like `BookWithConstructor c = new BookWithConstructor();`.

# Exercise 5: Payroll02: Adding an Employee Constructor

 10 to 15 minutes

---

1. Open the two files located in `Java-Objects/Exercises/Payroll02/`.
2. Modify your `Employee` class to use a constructor that accepts parameters for the first and last names, department, and pay rate.
3. Also add a constructor that takes no parameters and does nothing (as we discussed above).

```
public Employee() { }
```

4. In `Payroll`, modify `main` to add another `Employee` variable that receives an instance created using this constructor, then print their pay info.

## Solution: Java-Objects/Solutions/Payroll02/Employee.java

---

```
1.  public class Employee {
2.      private int id = 0;
3.      private String firstName;
4.      private String lastName;
5.      private int dept;
6.      private double payRate;
7.
8.      public Employee() { }
9.
10.     public Employee(String firstName, String lastName,
11.         int dept, double payRate) {
12.         setFirstName(firstName);
13.         setLastName(lastName);
14.         setDept(dept);
15.         setPayRate(payRate);
16.     }
17.
18.     public int getId() { return id; }
19.
20.     public String getFirstName() { return firstName; }
21.
22.     public void setFirstName(String fn) {
23.         firstName = fn;
24.     }
25.
26.     public String getLastName() { return lastName; }
27.
28.     public void setLastName(String ln) {
29.         lastName = ln;
30.     }
31.
32.     public int getDept() { return dept; }
33.
34.     public void setDept(int dp) {
35.         dept = dp;
36.     }
37.
38.
39.     public double getPayRate() { return payRate; }
40.
41.     public void setPayRate(double pay) {
42.         payRate = pay;
43.     }
44.
```

```
45.     public String getFullName() {
46.         return firstName + " " + lastName;
47.     }
48.
49.     public String getPayInfo() {
50.         return "Employee " + id + " dept " + dept + " " +
51.             getFullName() + " paid " + payRate;
52.     }
53.
54. }
```

---

### Solution: Java-Objects/Solutions/Payroll02/Payroll.java

---

```
1.     public class Payroll {
2.         public static void main(String[] args) {
3.
4.             Employee e1 = new Employee();
5.             e1.setFirstName("John");
6.             e1.setLastName("Doe");
7.             e1.setPayRate(6000.0);
8.             e1.setDept(2);
9.             System.out.println(e1.getPayInfo());
10.
11.            Employee e2 = new Employee("Jane", "Smith", 15, 6500.0);
12.            System.out.println(e2.getPayInfo());
13.        }
14.    }
```

---



## 3.15. Method Overloading

Methods can be *overloaded* to have several versions, all with the same name. Note that:

- The difference is in the parameter lists, also known as the *function signature*.
- Differences in return types are not enough to create a second version of a function, the parameter lists must be different.

```
[modifiers]
returnType methodName(paramList) { . . . }
[modifiers]
returnType methodName(differentParamList) { . . . }
```

Constructors are often overloaded, so that an object can be instantiated from different combinations of parameters.

This is an example of *polymorphism* - the concept that the same name may have several forms. Method overloading produces a *functional polymorphism*, since the same method name can be used in different ways. While polymorphism is mainly used to describe having one variable that can store different but related types due to inheritance, the concept also applies to overloaded methods at a smaller scale.

The combination of the method name and pattern of types in the parameter list is called the *function signature*. Within a class, every method and constructor must have a unique signature. Note that the return type is *not* considered part of the signature.

It is not required to keep the same return type for different versions of the same method. For example, the `Math` class has multiple `max` methods to determine the maximum of two parameters passed in. As you might expect, they are:

#### **Math.max methods**

```
public static double max(double a, double b)
public static float max(float a, float b)
public static int max(int a, int b)
public static long max(long a, long b)
```

While technically this is not considered overloading, that is a minor issue of semantics.

Continuing our example:

## Demo 3.9: Java-Objects/Demos/BookMultiConstructor.java

---

```
1.  public class BookMultiConstructor {
2.      private int itemCode;
3.      private String title;
4.      public BookMultiConstructor(int newItemCode, String newTitle) {
5.          setItemCode(newItemCode);
6.          setTitle(newTitle);
7.      }
8.      public BookMultiConstructor(String newTitle) {
9.          setItemCode(0);
10.         setTitle(newTitle);
11.     }
12.     public int getItemCode() {
13.         return itemCode;
14.     }
15.     public void setItemCode (int newItemCode) {
16.         if (newItemCode > 0) itemCode = newItemCode;
17.     }
18.     public String getTitle() {
19.         return title;
20.     }
21.     public void setTitle (String newTitle) {
22.         title = newTitle;
23.     }
24.     public void display() {
25.         System.out.println(itemCode + " " + title);
26.     }
27. }
```

---

### Code Explanation

---

We have overloaded this class's constructor - offering several different versions. External classes that make use of our `BookMultiConstructor` class can create a new instance of the class by calling the constructor that takes the item code and title or the constructor that takes just the title.

---

## Demo 3.10: Java-Objects/Demos/UseBookMultiConstructor.java

---

```
1. public class UseBookMultiConstructor {
2.     public static void main(String[] args) {
3.         BookMultiConstructor b =
4.             new BookMultiConstructor(5011, "Fishing Explained");
5.         b.display();
6.         // note the immediate call to a method on a new instance
7.         new BookMultiConstructor("Dogs I've Known").display();
8.     }
9. }
```

---

Evaluation  
Copy

### Code Explanation

---

This class instantiates two objects of class `BookMultiConstructor`, one from each of the two different constructors.

---

In the next exercise, we'll ask you to add more constructors to the `Employee` class.

# Exercise 6: Payroll03: Overloading Employee Constructors

 10 to 15 minutes

---

1. Open the files found in `Java-Objects/Exercises/Payroll03/`
2. Add more `Employee` constructors:
  - One that takes values for first and last names only.
  - One that takes values for first name, last name, and department.
  - One that takes values for first name, last name, and pay rate.
  - Note that, in practice, you can use the parameter lists for constructors to help enforce what you would consider valid combinations of properties - for example, if you would not want an employee to exist in a state where they had name and department information, but no pay rate, then the absence of that particular constructor would help ensure that condition.
  - Judicious use of copy-paste-edit can speed up this process, but be careful to make every necessary edit if you do this!
3. You will find yourself writing somewhat repetitive code, setting the same values the same way in several different constructors - we will address this issue a little later in the course.
4. In `Payroll`, create and pay additional instances using these constructors.

## Solution: Java-Objects/Solutions/Payroll03/Employee.java

---

```
1.  public class Employee {
2.      private int id = 0;
3.      private String firstName;
4.      private String lastName;
5.      private int dept;
6.      private double payRate;
7.
8.      public Employee() { }
9.
10.     public Employee(String firstName, String lastName) {
11.         setFirstName(firstName);
12.         setLastName(lastName);
13.     }
14.
15.     public Employee(String firstName,String lastName, int dept) {
16.         setFirstName(firstName);
17.         setLastName(lastName);
18.         setDept(dept);
19.     }
20.
21.     public Employee(String firstName, String lastName, double payRate) {
22.         setFirstName(firstName);
23.         setLastName(lastName);
24.         setPayRate(payRate);
25.     }
26.
27.     public Employee(String firstName, String lastName,
28.         int dept, double payRate) {
29.         setFirstName(firstName);
30.         setLastName(lastName);
31.         setDept(dept);
32.         setPayRate(payRate);
33.     }
34.
35.     public int getId() { return id; }
36.
37.     public String getFirstName() { return firstName; }
38.
39.     public void setFirstName(String fn) {
40.         firstName = fn;
41.     }
42.
43.     public String getLastName() { return lastName; }
44.
```

```
45.     public void setLastName(String ln) {
46.         lastName = ln;
47.     }
48.
49.     public int getDept() { return dept; }
50.
51.     public void setDept(int dp) {
52.         dept = dp;
53.     }
54.
55.
56.     public double getPayRate() { return payRate; }
57.
58.     public void setPayRate(double pay) {
59.         payRate = pay;
60.     }
61.
62.     public String getFullName() {
63.         return firstName + " " + lastName;
64.     }
65.
66.     public String getPayInfo() {
67.         return "Employee " + id + " dept " + dept + " " +
68.             getFullName() + " paid " + payRate;
69.     }
70.
71. }
```

---

## Code Explanation

---

We now have five constructors, one for each of the combinations of parameters we will accept.

---

## Solution: Java-Objects/Solutions/Payroll03/Payroll.java

---

```
1. public class Payroll {
2.     public static void main(String[] args) {
3.         Employee e1 = new Employee();
4.         e1.setFirstName("John");
5.         e1.setLastName("Doe");
6.         e1.setPayRate(6000.0);
7.         e1.setDept(2);
8.         System.out.println(e1.getPayInfo());
9.
10.        Employee e2 = new Employee("Jane", "Smith", 15, 6500.0);
11.        System.out.println(e2.getPayInfo());
12.
13.        Employee e3 = new Employee("Bob", "Jones", 5400.0);
14.        System.out.println(e3.getPayInfo());
15.
16.        Employee e4 = new Employee("Bill", "Meelater", 4);
17.        System.out.println(e4.getPayInfo());
18.
19.    }
20. }
```

---

Evaluation  
Copy

### Code Explanation

---

Employee e1 uses the default constructor (the one that takes no parameters, so that the set methods must be called to populate the object's data fields). e2 uses the full constructor. e3 uses the constructor that omits department - since we didn't set it, it prints as 0. e4 leaves off the pay rate, and, again, since we didn't set it, it prints as 0.0.

---



## 3.16. The this Keyword

The `this` keyword provides a reference to the current object. It is used to resolve name conflicts.

If you have a method that receives an input parameter `String s`, but `s` is already a member variable `String` reference.

- ```
class X {
    String s;
    public void setS(String s) {
        this.s = s;
    }
}
```

- Within the function, the parameter `s` is a local variable that will disappear when the function ends.
- It hides the existence of the `s` member field.
- But, we can pass the local variable's value into the field `s` by using the `this` reference to resolve `s` as the one that belongs to this object.
- Some programmers always use this approach with constructor functions and set methods - the benefit is that the code is more self-documenting, since it would use the most appropriate variable name in both the class definition and in the method argument.

The `this` keyword is also used when the object's code needs to pass a reference to itself to the outside. Say you are creating `MyClass`. Some other class, `YourClass`, has a public method called `useMyClass` that needs a reference to a `MyClass` object; in other words, something like:

```
public void useMyClass(MyClass x) { . . . }
```

Your code for `MyClass` could do:

```
YourClass yC = new YourClass(); yC.useMyClass(this);
```

The `this` keyword is also used in a constructor function to call another constructor from the same class.



## 3.17. Using `this` to Call Another Constructor

As we have seen, to avoid duplicating code in different functions, any function in a class may call other functions in that class.

- A function that is not a constructor can only call other functions that are *not* constructors (although it can still cause a constructor to run by instantiating an object).
- A constructor may only call *one other constructor*, and it must be done *as the first line* in the function.
- The `this` keyword is used instead of the name of the constructor, then arguments are passed in a normal fashion.

The following example uses `this` to provide the object with a reference to itself, as well as to chain to another constructor:

## Demo 3.11: Java-Objects/Demos/BookUsingThis.java

---

```
1.  public class BookUsingThis {
2.
3.      private int itemCode;
4.      private String title;
5.
6.      public BookUsingThis(int itemCode, String title) {
7.          setItemCode(itemCode);
8.          setTitle(title);
9.      }
10.     public BookUsingThis(String title) {
11.         this(0, title);
12.         /*
13.            the line above is a call to the first constructor,
14.            supplying 0 as the value for the itemCode parameter
15.         */
16.     }
17.
18.     public int getItemCode() {
19.         return itemCode;
20.     }
21.     public void setItemCode (int itemCode) {
22.         if (itemCode > 0) this.itemCode = itemCode;
23.     }
24.     public String getTitle() {
25.         return title;
26.     }
27.     public void setTitle (String title) {
28.         this.title = title;
29.     }
30.     public void display() {
31.         System.out.println(itemCode + " " + title);
32.     }
33. }
```

---

### Code Explanation

---

The constructor that only accepts the title calls the other constructor, passing 0 as the item code. It would also be possible to set up the chain of constructors in the reverse direction, as in:

### Constructors Chained from Simplest to More Complex

```
public BookUsingThis(int itemCode, String title) {
    this(title);
    setItemCode(itemCode);
}
public BookUsingThis(String title) {
    setTitle(title);
}
```

This approach is good if the default values of the fields (either zero or as set by an initializer) are acceptable - note that the constructor that accepts only title never sets the item code.

---

### **Demo 3.12: Java-Objects/Demos/UseBookUsingThis.java**

---

```
1. public class UseBookUsingThis {
2.     public static void main(String[] args) {
3.         BookUsingThis b = new BookUsingThis(5011, "Fishing Explained");
4.         b.display();
5.         new BookUsingThis("Dogs I've Known").display();
6.     }
7. }
```

---

### Code Explanation

---

The second book uses the constructor that accepts only the title. Since the new operator results in a reference to the book, we can chain a call to display to that result.

---

# Exercise 7: Payroll04: Using the this Reference

 10 to 15 minutes

---

1. Now we can clean up our Employee class definition: open the two files in Java-Objects/Exercises/Payroll04/
2. Modify the set methods to use the same name for each parameter as the associated field.
3. Modify the constructors to eliminate redundant code - for example:

```
public Employee(String firstName, String lastName, int dept) {  
    this(firstName, lastName); setDept(dept);  
}
```

## Solution: Java-Objects/Solutions/Payroll04/Employee.java

---

```
1.  public class Employee {
2.      private int id = 0;
3.      private String firstName;
4.      private String lastName;
5.      private int dept;
6.      private double payRate;
7.
8.      public Employee() {
9.      }
10.     public Employee(String firstName, String lastName) {
11.         setFirstName(firstName);
12.         setLastName(lastName);
13.     }
14.     public Employee(String firstName,String lastName, int dept) {
15.         this(firstName, lastName);
16.         setDept(dept);
17.     }
18.     public Employee(String firstName, String lastName, double payRate) {
19.         this(firstName, lastName);
20.         setPayRate(payRate);
21.     }
22.     public Employee(String firstName, String lastName,
23.         int dept, double payRate) {
24.         this(firstName, lastName, dept);
25.         setPayRate(payRate);
26.     }
27.
28.     public int getId() { return id; }
29.
30.     public String getFirstName() { return firstName; }
31.
32.     public void setFirstName(String firstName) {
33.         this.firstName = firstName;
34.     }
35.
36.     public String getLastName() { return lastName; }
37.
38.     public void setLastName(String lastName) {
39.         this.lastName = lastName;
40.     }
41.
42.     public int getDept() { return dept; }
43.
44.     public void setDept(int dept) {
```

```
45.     this.dept = dept;
46.   }
47.
48.
49.   public double getPayRate() { return payRate; }
50.
51.   public void setPayRate(double payRate) {
52.     this.payRate = payRate;
53.   }
54.
55.   public String getFullName() {
56.     return firstName + " " + lastName;
57.   }
58.
59.   public String getPayInfo() {
60.     return "Employee " + id + " dept " + dept + " " +
61.           getFullName() + " paid " + payRate;
62.   }
63.
64. }
```

---

Evaluation  
\*  
Copy

## 3.18. static Elements

The keyword `static` states that one and only one of this element should be available at all times - whether there is one object of the class instantiated, many objects, or none at all.

### Declaring Static Elements

```
[modifiers] static dataType fieldName;
[modifiers] static returnType methodName(paramType paramName, . . .)
```

If a data member is `static`, there is one memory location that is shared among all instances of a class. This allows the separate instances of one class a way of sharing data with each other. That memory location exists even if no objects of the class exist - this is useful for creating constants. Any initialization of this value occurs just once, when the *class is loaded* into the JVM.

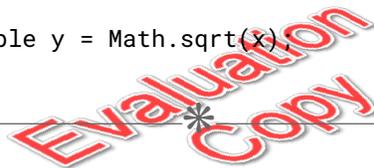
If a method is `static` and `public`, then it is available even when no objects have been instantiated.

If a class has `static` fields, you may want to have methods to work with those fields. It would make sense that those methods be `static` as well.

Utility functions, like the random number and trigonometric functions in `Math`, have to be part of a class, but you wouldn't want to have to instantiate some special object just to generate a single random number or calculate a cosine. For more sophisticated random number generation, where repeatability of a sequence of random values might be desirable, there is a `Random` class.

One caveat with this approach is that the function cannot access any of the other elements (data members or methods) that are not also `static`, since it is not guaranteed that they will exist when the function is called. Such an element is referenced using the name of the class, a dot, then the element name:

```
double x = Math.random(); double y = Math.sqrt(x);
```



### 3.19. The `main` Method

Now we can see that `main`, as a `static` method, can be run without any instance of the class. In fact, no instance is created automatically - the JVM only needs to load the class into memory.

If you create a class that has fields or methods not marked as `static`, and try to access them from `main`, you will get a rather cryptic error message about a non-`static` element referenced from a `static` context. The problem is that there isn't necessarily an instance of your class in memory, or there may be more than one. To call a non-`static` element, you must first create an instance and access the element for that instance.

The following example draws from the GUI world, where the `JFrame` class has a `setVisible` method that this class inherits. The `main` method creates an instance of its own class to display.

### **main Class Instance Example**

```
import javax.swing.*;
public class MyGUI extends JFrame{
    public static void main(String[] args) {
        MyGUI gui = new MyGUI();
        gui.setVisible(true);
    }
}
```

**Evaluation  
Copy**

# Exercise 8: Payroll05: A static field in Employee

 5 to 10 minutes

---

1. Open the files in Java-Objects/Exercises/Payroll05/.
2. Add a private and static integer field called `nextId` to the `Employee` class (give it an initial value of 1).
3. Modify the declaration of the `id` field as follows:

```
private int id = nextId++;
```

4. What happens when each new `Employee` gets instantiated?
5. Run and test the application - you should see incrementing employee ids.
6. Now add the following to `Employee.java`:

```
static Methods for Employee  
public static int getNextId() {  
    return nextId;  
}  
public static void setNextId(int nextId) {  
    Employee.nextId = nextId;  
}
```

Notice the syntax we use to resolve the name conflict between the `nextId` parameter and the static `nextId` field.

7. In `Payroll`, before we create the first employee, call `Employee.setNextId(22)` to set the value that will be used for the first employee.



## Solution: Java-Objects/Solutions/Payroll05/Employee.java

---

```
1.  public class Employee {
2.      private static int nextId = 1;
3.      private int id = nextId++;
4.
5.      public static void setNextId(int nextId) {
6.          Employee.nextId = nextId;
7.      }
8.      public static int getNextId() {
9.          return nextId;
10.     }
11.
12.     private String firstName;
13.     private String lastName;
14.     private int dept;
15.     private double payRate;
16.
17.     public Employee() {
18.     }
19.     public Employee(String firstName, String lastName) {
20.         setFirstName(firstName);
21.         setLastName(lastName);
22.     }
23.     public Employee(String firstName, String lastName, int dept) {
24.         this(firstName, lastName);
25.         setDept(dept);
26.     }
27.     public Employee(String firstName, String lastName, double payRate) {
28.         this(firstName, lastName);
29.         setPayRate(payRate);
30.     }
31.     public Employee(String firstName, String lastName, int dept, double payRate)
32.         {
33.         this(firstName, lastName, dept);
34.         setPayRate(payRate);
35.     }
36.     public int getId() { return id; }
37.
38.     public String getFirstName() { return firstName; }
39.
40.     public void setFirstName(String firstName) {
41.         this.firstName = firstName;
42.     }
43. }
```

```
44.     public String getLastName() { return lastName; }
45.
46.     public void setLastName(String lastName) {
47.         this.lastName = lastName;
48.     }
49.
50.     public int getDept() { return dept; }
51.
52.     public void setDept(int dept) {
53.         this.dept = dept;
54.     }
55.
56.
57.     public double getPayRate() { return payRate; }
58.
59.     public void setPayRate(double payRate) {
60.         this.payRate = payRate;
61.     }
62.
63.     public String getFullName() {
64.         return firstName + " " + lastName;
65.     }
66.
67.     public String getPayInfo() {
68.         return "Employee " + id + " dept " + dept + " " +
69.             getFullName() + " paid " + payRate;
70.     }
71.
72. }
```

---

## Code Explanation

---

When the `Employee` class is loaded, the `nextId` field is initialized to 1. This initialization *only happens once*, when the class is loaded.

Then, when each employee is created, the explicit initializer for that employee's `id` reads the current value of `nextId`, and then increments `nextId` to be ready for the next employee.

---



## 3.20. Garbage Collection

Java takes care of reclaiming memory that is no longer in use.

Your program is not making memory-allocation calls directly to the operating system - the JVM requests a fairly large block at start time, and handles memory allocation from that block. Note that:

- When necessary, it can request additional blocks from the OS.
- There are command line options to set the initial and maximum sizes.

When an object is no longer reachable through your code, it becomes subject to garbage collection. Note that:

- The JVM has a low-priority thread that checks the *graph* of objects to find orphaned objects.
- Those that are found are marked as available to be collected.
- A separate process will run when necessary to reclaim that memory.
- If your program has small memory requirements, it is possible that garbage collection will never run.

You can request garbage collection by calling `System.gc()`. Note that this may not have any effect; it is not guaranteed to run when you ask - the call is merely a request.

You can specify code to run when an object is collected by writing a `finalize()` method in a class. Note that:

- There is no guarantee that this method will ever run.
- When the program ends, any objects still reachable will not be collected, nor will any objects marked for collection but not yet collected - thus finalization will not occur for those objects.



## 3.21. Java Packages

As you have seen, Java programs may involve a large number of files.

Packages help organize files within a program, as well as to collect objects into groups to improve reusability of code.

A package structure is a tree structure that maps to a folder/directory structure. Note that:

- Package names correspond to directory names.
- A dot is used between levels of a multilevel structure.
- Searching for files starts at the root level of the structure during compiling or class-load at runtime.
- You can't use packages or directory structures without explicitly placing a class in a package via your code.
- Classes are always referenced from the root of the structure, and the *fully-qualified name* of the class contains the relative path elements to reach the class from the root point, separated by dot characters.

To assign a class to a package, use a package statement as *the first* non-blank, non-comment line of code.

| Code                            | Effect                                                                                                                                                                                                                                                        |
|---------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>package test;</code>      | Puts this class in the package called <code>test</code> in order to be found, the class file must be in a directory named <code>test</code> that directory must be located in one of the classpath locations.                                                 |
| <code>package test.util;</code> | Puts this class in the package called <code>test.util</code> in order to be found, the class file must be in a subdirectory of the <code>test</code> directory named <code>util</code> ; <code>test</code> must be located in one of the classpath locations. |

The accepted convention is that package names be all lowercase.

### Example

```
package test;  
public class XYZ { . . . }
```

- `XYZ.java` should be in a directory called `test` that is located within a directory listed in our CLASSPATH.
- The fully-qualified name of the class is `test.XYZ`.

When the compiler and JVM look for the files, they will search every CLASSPATH entry for a directory called `test` containing `XYZ.java` (or `XYZ.class`, if that is what is needed).



## 3.22. Compiling and Executing with Packages

The default behavior of the compiler in JDK is to compile source code in a directory structure into class files in the same structure. Note that:

- You can use the `-d` option with `javac` to put the class files into a separate, parallel directory structure.

### Compiling With a Destination Directory

```
javac -d rootofstructure *.java
```

- It will start the structure at the directory called `rootofstructure`.
- It will create the package directories if they don't already exist.
  - In earlier versions of the JDK, all the source code, regardless of package, had to be in the same directory for that to work.

To run an executable Java class that belongs to a package, your command prompt should be at the directory level that matches the *root* of the package structure. You would then treat the class name as `packagename.ClassName`.

Example - the directory `C:\MyProject` is the root level of a package structure, and the main class, `XYZ.class`, is in the package called `test` (therefore `test` is a subdirectory of `MyProject`).

To run `XYZ.class`, the command line would look like the following:

### Running a main Class That is in a Package

```
C:\MyProject>java test.XYZ
```



## 3.23. Working with Packages

Once a package structure is created, all classes in it must be referenced with respect to the root of the structure. You have two choices:

- Explicitly use the fully-qualified name of the class.
- *import* either that specific class or the entire package.

Say your directory `C:\Bookstore` is the root of your project.

- `C:\Bookstore\UseBookInPackage.java` is the main class.
- A directory called `types` contains `BookInPackage.java`; that is, `C:\Bookstore\types\BookInPackage.java`.

### Declaring a package

The `BookInPackage` class would start like this:

```
package types;
public class BookInPackage {
    . . .
}
```

Evaluation  
Copy

The package statement must be the first non-comment, non-blank line in the file.

### Referencing Packaged Classes - Importing and Fully Qualified Names

In `UseBookInPackage.java`, you could import the entire `types` package, or just the class(es) you need

```
import types.*;
```

*or*

```
import types.BookInPackage;
```

*then*

```
public class UseBookInPackage {
    public static void main(String[] args) {
        BookInPackage b = new BookInPackage("My Favorite Programs");
        . . .
    }
}
```

Or, instead of importing, you could explicitly reference the class by its full name, which precedes the class name with the package structure (using a dot character as the directory separator)

```
public class UseBookInPackage {
    public static void main(String[] args) {
        types.BookInPackage b = new types.BookInPackage("My Favorite Programs");
        . . .
    }
}
```

This approach is necessary if your program uses two classes with the same name (such as `java.sql.Date` and `java.util.Date`).

All classes in the Java API are in packages. We have not had to deal with imports to this point because the package `java.lang` is special - it does not need to be imported. (This is where `System`, `String`, the numeric wrapper classes like `Integer`, and a few more generally useful classes reside.)

The package at the root level is called the *default package*.

### When the main Class is in a Package

If your main class is in a package, then it must be compiled and run from the root of the structure.

For example, say your directory `C:\Bookstore` is the root of your project, but it contains no classes.

It contains a subdirectory called `bookstore`, which contains `Bookstore.java` (the main class)

```
package bookstore; public class Bookstore { . . . }
```

- If that directory contained the `types` directory, the remainder of the code would be as before - we could import our `types.Book` class.

To compile the program from the `C:\Bookstore` directory, use the standard operating system directory separator character:

```
javac bookstore\Bookstore.java
```

To run the program, you must be in the `C:\Bookstore` directory; use the dot character as the separator character:

```
java bookstore.Bookstore
```

Note that the preferred practice is to have *no* classes in the default package.

- Because the package has no name, there is no way to access it or any of its classes from any other package!
- Because no other class needs to reference the main class, for now we will leave that in the default package.

## static Imports

*Static imports* enable you to import the static elements of a class. This can be used both for fields (usually constants) and methods. The syntax is:

### **Static Imports**

```
import static packages.Class.element;  
import static packages.Class.*;
```

These will import the specified elements so that they may be used within the current class, not only without the fully qualified package structure, but without even needing the class name.

## Demo 3.13: Java-Objects/Demos/TestStaticImport.java

---

```
1.  import static java.lang.Math.*;
2.
3.  public class TestStaticImport {
4.
5.      public static void main(String[] args) {
6.          double sin90 = sin(PI / 2);
7.          double sqrt = sqrt(4);
8.
9.          System.out.println("Sine of 90 degrees is " + sin90);
10.         System.out.println("Square root of 4.0 is " + sqrt);
11.     }
12.
13. }
```

---

### Code Explanation

---

We have imported every static element from `java.lang.Math`, giving us access to `Math.PI`, `Math.sin(double)`, and `Math.sqrt(double)`. Notice that the local variable `sqrt` does not conflict with the imported method, since without the parentheses for the parameter list, we are using the variable, with parentheses we are calling the method. If we did write a `sqrt` method in this class, that method would take precedence over the imported method; calling `sqrt` would invoke our local method, and we would need to qualify the call to `sqrt` as `Math.sqrt` in order to use the `Math` version.

---

### Other Package Considerations

It is a recommended convention that classes created for external distribution use a package structure that begins with the creating organization's domain name in reverse order.

Package names should be in all lowercase, to avoid conflict with the names of classes or interfaces.

Package access does not necessarily limit you to working in the same directory; you could have the same directory structure from a different root point in the classpath.

## Exercise 9: Payroll06: Creating an employees Package

⌚ 5 to 10 minutes

---

1. Open the files found in `Java-Objects/Exercises/Payroll06/`
2. Create a package (a directory) called `employees`; put the `Employee.java` file into this package (leave `Payroll` where it is).
3. This will require not only creating the directory and moving the file, but also adding an `import` statement to `Payroll` and a `package` statement to `Employee`.
4. To compile, start in the project root directory (the directory containing `Payroll`). If you compile `Payroll` as usual, it will also find and compile `Employee`. To compile just `Employee`, you would still work in the project root directory, but execute.

### Compiling a Packaged Class

```
javac employees\Employee.java
```

5. Run `Payroll` in the usual fashion.

## Solution: Java-Objects/Solutions/Payroll06/employees/Employee.java

---

```
1.  package employees;
2.
3.  public class Employee {
4.      private static int nextId = 1;
5.      private int id = nextId++;
6.      private String firstName;
7.      private String lastName;
8.      private int dept;
9.      private double payRate;
10.
11.
-----Lines 12 through 75 Omitted-----
76. }
```

---

### Code Explanation

---

The first line of `Employee.java` declares that it is a member of package `employees`.

---

## Solution: Java-Objects/Solutions/Payroll06/Payroll.java

---

```
1.  import employees.*;
2.
3.  public class Payroll {
4.      public static void main(String[] args) {
5.          Employee e1 = new Employee();
6.
-----Lines 7 through 23 Omitted-----
24.
25.     }
26. }
```

---

### Code Explanation

---

The first line of `Payroll.java` imports all of package `employees` - which, in our case, is just `Employee.java`.

---



## 3.24. Variable Argument Lists (varargs)

Java offers a syntax for variable argument lists, called *varargs*. This enables you to write methods that take a variable number of arguments, in addition to any fixed arguments, as long as the variable arguments are all of the same type.

- There may be 0 or more fixed parameters at the beginning of the parameter list, as in a regular parameter list.
- There may be 0 or 1 varargs entries at the end of the parameter list; this single entry supports 0 or more parameters of the specified type.
- The varargs parameter is specified as `type...`, as in `int...`
- Within the method, the varargs entry is treated as an array.

### Demo 3.14: Java-Objects/Demos/Varargs.java

---

```
1.  public class Varargs {
2.
3.  public static void showDataOnly(int... values) {
4.    for (int i = 0; i < values.length; i++) {
5.      System.out.println("  " + values[i]);
6.    }
7.  }
8.
9.  public static void showDataWithMessage(String message, int... values) {
10.   System.out.println(message);
11.   showDataOnly(values);
12.  }
13.
14.  public static void main(String[] args) {
15.
16.   System.out.println("Below printed from showDataOnly");
17.   showDataOnly(2, 4, 5, 1, 7);
18.
19.   showDataWithMessage(
20.     "This printed with showDataWithMessage", 2, 4, 5, 1, 7);
21.
22.   showDataWithMessage("No values this time");
23.
24.   int[] data = { 4, 5, 1, 3 };
25.   showDataWithMessage("This time we passed an array", data);
26.  }
27. }
```

---

## Code Explanation

---

There are two `varargs` methods. The first has only the one `varargs` parameter, `values`. Within its code, `values` can be treated as an array. When we call the method, we can pass it any number of integer values, 0 or more.

The second method takes one `String`, followed by the `varargs` parameter. When we call it, we must pass at least the message, and can pass any number of integers after that. Note that the second method prints the message, and then delegates printing the values to the first method.

In `main`, we also try another approach, which is to pass an array for the `varargs` parameter. This also works. Because of this, it is not legal to write a method overloads like `myMethod(int[] x)` and `myMethod(int... x)`, since they could both be invoked by passing an array, even though the array version *could not* be invoked by passing a comma-separated list of values!

If some of the code above - arrays and `for` loops - looks a bit confusing, not to worry; we will cover these aspects of Java in detail later in the course. The important feature here is that we can supply an indeterminate number of parameters to a method.

---

### ❖ 3.24.1. Keyboard I/O Using the `Console` Class

The `Console` class provides an easy way to access the keyboard. If a console is available to the JVM, the `System.console()` method will return an instance of the `Console` class,

otherwise, it will return `null`. (Unfortunately, Eclipse is one environment for which a console is not available, due to the way that your programs are launched within it.)

| Console Class Methods    |                                                         |                                                                                                                                                                                              |
|--------------------------|---------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void</code>        | <code>flush()</code>                                    | Flushes the console and forces any buffered output to be written immediately.                                                                                                                |
| <code>Console</code>     | <code>format(String format, Object...args)</code>       | Writes a formatted string to this console's output stream using the specified format string and arguments. Returns this instance so that method calls may be chained.                        |
| <code>Console</code>     | <code>printf(String format, Object...args)</code>       | A convenience method to write a formatted string to this console's output stream using the specified format string and arguments. Returns this instance so that method calls may be chained. |
| <code>Reader</code>      | <code>reader()</code>                                   | Retrieves the unique <code>Reader</code> object associated with this console.                                                                                                                |
| <code>String</code>      | <code>readLine()</code>                                 | Reads a single line of text from the console.                                                                                                                                                |
| <code>String</code>      | <code>readLine(String format, Object...args)</code>     | Provides a formatted prompt, then reads a single line of text from the console.                                                                                                              |
| <code>char[]</code>      | <code>readPassword()</code>                             | Reads a password or passphrase from the console with echoing disabled.                                                                                                                       |
| <code>char[]</code>      | <code>readPassword(String format, Object...args)</code> | Provides a formatted prompt, then reads a password or passphrase from the console with echoing disabled.                                                                                     |
| <code>PrintWriter</code> | <code>writer()</code>                                   | Retrieves the unique <code>PrintWriter</code> object associated with this console.                                                                                                           |

## String and PrintStream Methods

The `format` method listed above is also available as a static method in the `String` class, and, for convenience, in the `PrintStream` class (of which `System.out` is an instance). `String.format` returns the formatted `String` result, while the `PrintStream` version prints the formatted result and returns the `PrintStream` reference for chaining.

`PrintStream` also has the `printf` method, which again returns its own reference after printing the output.

The discussion below is a brief overview of formatting strings. The `String` class documentation of the `format` method has a link to a full discussion of all the formatting options.

## Format Strings

The methods that accept a *format string* use the *varargs* (variable argument list) syntax. The format string contains placeholders, indicated by the `%` signs, at which values will be substituted. The values used will be any additional parameters you pass to the method as indicated by the `Object...` parameter. The percent sign is followed by one or more symbols that determine the type and format of the output for that value. Format specifiers are of the form:

### Format Strings

```
%[
    argument_index$
    ][
    flags
    ][
    width
    ][
    .precision
    ]conversion
```

Evaluation  
Copy

Where (in order of importance/frequency of use):

- *Conversion* is a one or two letter code indicating the type of data - some conversion codes are:
  - `d`: integer
  - `f`: floating point
  - `e`: floating point in scientific notation
  - `s`: string
  - `b`: boolean (using `true` and `false` as output strings)
- There is a large selection of two-letter codes for date/time values, some of which are:
  - `tY`: a four-digit year

- ty: a two-digit year
  - tm: a two digit month number
  - tB: a full month name
  - tb: a three-letter month name abbreviation
  - td: a two-digit day of month (with leading zero if necessary)
  - te: a one or two digit day of month
  - tD: a date formatted as month/day/year (as in 02/05/2010)
  - tr: a time formatted as hours:minutes:seconds AM/PM (as in 09:05:00 AM)
- *Width* is the minimum character width of the field.
  - *Precision* is the number of digits after the decimal point (these characters and the decimal point are included in the character count toward the width).
  - *Flags* modify the output , they include:
    - -: indicates left-alignment within a field, without this all values, even strings, are right-aligned
    - 0: pad a numeric field with 0 characters
    - (space): leave a space for a sign in positive numbers
    - ,: include grouping characters every three digits left of the decimal point
    - (: indicate negative values in parentheses
    - +: always include a sign on a number
  - *Argument\_index* is the position of the argument in the parameter list, overriding the default of using the arguments in order - this is particularly useful with dates and times, where several pieces of the same value may be printed (day, month, year, etc.).
    - A dollar sign follows the argument index, in order to distinguish from a width number.
    - Note that the first value parameter is 1\$.
  - %% prints a single percent sign.
  - %n produces a platform-specific end-of line sequence.

Note that many of the controls are *locale-specific*, so that month names, day names, the use of comma and dot for thousands or decimal separator, etc., will be determined by the machines locale setting.

## Demo 3.15: Java-Objects/Demos/TestConsole.java

---

```
1.  import java.io.Console;
2.  import java.util.Date;
3.
4.  public class TestConsole {
5.      public static void main(String[] args) {
6.          Console console = System.console();
7.
8.          // using unformatted I/O
9.          String name = console.readLine("What is your name? ");
10.         console.printf("Hello, " + name + "\n\n");
11.
12.         // using formatted I/O
13.         int age = Integer.parseInt(
14.             console.readLine("Tell me %s, how old are you? ", name));
15.         console.printf("Wow, you don't look a day over %d!%n%n", age - 10);
16.
17.         // using formatted date/time
18.         console.printf("Today is %tD%nand it is now %tr%n%n",
19.             new Date(), new Date());
20.
21.         // using argument order indicators
22.         console.printf("Today is %1$tD%nand it is now %1$tr%n%n",
23.             new Date());
24.
25.     }
26. }
```

---

### Code Explanation

---

The output on line 10 concatenates strings to substitute values. The output on line 15 uses a formatting string with a series of substitution values instead.

The output on line 18 supplies the same date twice, in order to print two different parts of it, while the output on line 22 uses the parameter index so that we need only supply the date once.

---

So, in the formatting method parameter list, `String format`, `Object...args`, the format string is the first parameter, and the values to be formatted are the rest. Because their type is `Object`, they can actually be any type of object. But, how come it accepts primitives like integers? Hold onto that thought for just a while longer, we will cover that later in this lesson.

---

## ❖ 3.24.2. Keyboard Input Without the Console

Java input is very “low-level” - it deals with raw bytes of data, but, we can use classes designed to make input easier.

Also, in general, input is error-prone, requiring our program to handle *exceptions*, situations requiring special attention. Even though the keyboard will never throw an exception, as an instance of input stream, its methods are marked as throwing `IOException`.

Since it would be nice to have interactive programs, the following code gives an example of reading data from the keyboard.

### Demo 3.16: Java-Objects/Demos/InputTest.java

---

```
1.  import java.io.*;
2.  public class InputTest {
3.      public static void main(String[] args)
4.          throws Exception {
5.          BufferedReader in =
6.              new BufferedReader(
7.                  new InputStreamReader(System.in));
8.          System.out.println("Enter a string");
9.          String s = in.readLine();
10.         System.out.println("Enter an integer");
11.         int i = Integer.parseInt(in.readLine());
12.         System.out.println("Enter a character");
13.         char c = (char) in.read();
14.         System.out.println("s=" + s + " i=" + i + " c=" + c);
15.     }
16. }
```

---

Briefly:

- The `import` statement enables you to use classes not in the default set (in this case the input-related classes) - the API documentation lists the package for each class.
- The raw input class `System.in` is *filtered* using an `InputStreamReader`, which allows it to recognize input as text characters (the basic `InputStream` is plugged into a more complex object that does additional processing).
- Then, the resulting `InputStreamReader` is plugged into a `BufferedReader` that buffers keyboard input until it recognizes the Enter keystroke.

- Lines can now be read using `readLine()`, and characters can be read using `read()` (note that they are read as integers, thus the typecast).
- A sequence of digits can be parsed using a static method of the `Integer` class.
- Since the whole input and parsing process is error-prone, we will avoid issues by having `main` *throw an exception* (much more on exceptions later).

## A Keyboard Reader Class

The following class, `KeyboardReader.java`, can be used to read from the keyboard:

## Demo 3.17: Java-Objects/Demos/util/KeyboardReader.java

---

```
1.  package util;
2.  import java.io.*;
3.
4.  public class KeyboardReader {
5.
6.      private static BufferedReader in =
7.          new BufferedReader(new InputStreamReader(System.in));
8.
9.      public KeyboardReader() { }
10.
11.     public static String getPromptedString(String prompt) {
12.         String response = null;
13.         System.out.print(prompt);
14.         try {
15.             response = in.readLine();
16.         } catch (IOException e) {
17.             System.out.println("IOException occurred");
18.         }
19.         return response;
20.     }
21.
22.     public static char getPromptedChar(String prompt) {
23.         return getPromptedString(prompt).charAt(0);
24.     }
25.
26.     public static int getPromptedInt(String prompt) {
27.         return Integer.parseInt(getPromptedString(prompt));
28.     }
29.
30.     public static float getPromptedFloat(String prompt) {
31.         return Float.parseFloat(getPromptedString(prompt));
32.     }
33.     public static double getPromptedDouble(String prompt) {
34.         return Double.parseDouble(getPromptedString(prompt));
35.     }
36. }
```

---

### Code Explanation

---

The approach shown above is used in a class with all static elements. A static `BufferedReader` is created when the class is loaded, and used by a set of static methods that retrieve specific types of data from the keyboard (`String`, `char`, `int`, `float`, and `double`).

The `getPromptedString` method prints the prompt and retrieves the `String` entered in response. It handles the `IOException` that `readLine` is specified as throwing (even though the keyboard can never actually throw that exception, the methods in the classes involved, like `BufferedReader` and `InputStream` (used behind the scenes), can throw those exceptions when used with other types of streams, like those coming from files. This is one case where inheritance can actually get in the way, by adding a bit of complexity as the cost of having a system that will work for all types of input streams.

Since all keyboard input starts as a `String`, the methods to retrieve other types of data reuse `getPromptedString`, and then process the result to obtain the required type of data. One benefit of this approach is that the potential for an exception has been handled, so we don't have to worry about that again in these methods.

The remainder of our demos and exercises use `KeyboardReader`. (For those using Eclipse, using `KeyboardReader` still makes sense, rather than the console. Besides, there are not very many console-based programs being written, so the use of the console has limited reach in today's development projects.) Feel free to modify these to use `Console` I/O - you might want to create a revised `KeyboardReader` that uses the `Console` methods, for example.

---

# Exercise 10: Payroll07: Using KeyboardReader in Payroll

 10 to 20 minutes

---

1. Open the files in Java-Objects/Exercises/Payroll07/
2. Add another employee to your payroll, this time using the `KeyboardReader` class to prompt the user for all data; for example, to read a double-precision value for a number:

```
double num = KeyboardReader.getPromptedDouble("Please enter a number: ");
```

3. `KeyboardReader` is in the `util` package; you will need to import the class in `Payroll.java`.
4. There are no changes to `Employee.java`.

## Solution: Java-Objects/Solutions/Payroll07/Payroll.java

---

```
1.  import employees.*;
2.  import util.*;
3.
4.  public class Payroll {
5.      public static void main(String[] args) {
6.          String fName = null;
7.          String lName = null;
8.          int dept = 0;
9.          double payRate = 0.0;
10.
11.         fName = KeyboardReader.getPromptedString("Enter first name: ");
12.         lName = KeyboardReader.getPromptedString("Enter last name: ");
13.         dept = KeyboardReader.getPromptedInt("Enter department: ");
14.         payRate = KeyboardReader.getPromptedDouble("Enter pay rate: ");
15.         Employee e1 = new Employee(fName, lName, dept, payRate);
16.
17.         fName = KeyboardReader.getPromptedString("Enter first name: ");
18.         lName = KeyboardReader.getPromptedString("Enter last name: ");
19.         dept = KeyboardReader.getPromptedInt("Enter department: ");
20.         payRate = KeyboardReader.getPromptedDouble("Enter pay rate: ");
21.         Employee e2 = new Employee(fName, lName, dept, payRate);
22.
23.         fName = KeyboardReader.getPromptedString("Enter first name: ");
24.         lName = KeyboardReader.getPromptedString("Enter last name: ");
25.         dept = KeyboardReader.getPromptedInt("Enter department: ");
26.         payRate = KeyboardReader.getPromptedDouble("Enter pay rate: ");
27.         Employee e3 = new Employee(fName, lName, dept, payRate);
28.
29.         System.out.println(e1.getPayInfo());
30.         System.out.println(e2.getPayInfo());
31.         System.out.println(e3.getPayInfo());
32.
33.     }
34. }
```

---

### Code Explanation

---

This code goes a bit beyond the requested changes, creating three employees instead of just one.

---



## 3.25. String, StringBuffer, and StringBuilder

Java has three primary classes for working with text.

The `String` class is optimized for retrieval of all or part of the string, but not for modification (it is assumed to be written once and read many times). Note that:

- This comes at the expense of flexibility.
- Once created, a `String` object is immutable, that is, its contents cannot be changed.
- Methods like `toUpperCase()` that seem like they would change the contents actually return a brand-new `String` object with the new contents.

The `StringBuffer` and `StringBuilder` classes are optimized for changes. Note that:

- They are essentially the same as far as available methods, but `StringBuilder` is not safe for multithreaded applications while `StringBuffer` is (but `StringBuilder` is faster because of that).
- They are useful for building a long string from multiple pieces, or for text-editing applications.
- Among the useful methods are:
  - `append(String s)` and other forms accepting various primitive data types and a form that accepts another `StringBuffer` object.
  - `insert(int position, String s)` to insert text at (in front of) the specified position; again, various forms for different types of data.
  - `delete(int start, int end)` to delete characters from the start position up to but not including the end position.
  - `toString()` will convert the contents of the object to a `String` object.
- Note that the modifying methods listed above modify the contents of the object, but also return a `this` reference, (a reference to the same object) - this enables chaining of method calls:

```
StringBuffer sb = new StringBuffer();
sb.append("Hello").append(" World");
System.out.println(sb.toString());
```



## 3.26. Creating Documentation Comments and Using javadoc

Java documentation can be created as part of the source code. If you embed *javadoc comments* in your code the javadoc documentation engine supplied with the jdk will create a set of interlinked HTML pages, one for each class.

### ❖ 3.26.1. Javadoc Comments

A javadoc comment is a block comment beginning with `/**`, and ending with `*/`

- They may be used before any documentable item: (classes, fields, constructors, and methods - if used in other places they will be ignored by the documentation engine).
- Use them to provide descriptions of the item.
- Many HTML tags are valid within the comment - in particular, formatting tags like `<em>` and `<code>` are often used, as well as lists.
- There are a number of special *block tags* that begin with `@`, such as:
  - `@param` documents a method or constructor parameter.
  - `@return` documents a return value.
  - `{@link BookWithJavadoc}` creates a hyperlink to the specified class page.

#### javadoc Command Syntax

```
javadoc options filelist
```

Command line options include:

- `-d destinationdirectory` to produce the output in a specific directory, which need not already exist (default is current directory).
- Options for access levels to document (each option generates documentation for that level and every more accessible level - e.g., protected documents `public`, `package`, and protected elements)
  - `-private`
  - `-package`
  - `-protected`

- o -public

The file list can use file names, package names, wildcards, like \*.java (separate multiple elements with spaces).

The following command places the output in a subdirectory docs of the current directory, and documents the “default package” and the employees package (use this in the upcoming exercise):

Evaluation  
Copy

**Sample javadoc Command Line**

```
javadoc -d docs -private employees *.java
```

See Oracle’s How To (<http://www.oracle.com/technetwork/articles/java/index-137868.html>) reference for more information.

## Demo 3.18: Java-Objects/Demos/BookWithJavadoc.java

---

```
1.  /**
2.   * Represents a Book in inventory,with an item code and a price
3.   */
4.  public class BookWithJavadoc {
5.
6.   /**
7.    * The book's item code
8.    */
9.    private int itemCode;
10.
11.   /**
12.    * The title of the book
13.    */
14.    private String title;
15.
16.   /**
17.    * Creates a book instance.
18.    * @param itemCode the book's item code. It is expected
19.    * that the value will be non-negative; a negative value
20.    * will be rejected.
21.    * @param title the title of the book
22.    */
23.    public BookWithJavadoc(int itemCode, String title) {
24.        setItemCode(itemCode);
25.        setTitle(title);
26.    }
27.
28.   /**
29.    * Creates a book instance, The item code will be set to 0.
30.    * @param title the title of the book
31.    */
32.    public BookWithJavadoc(String title) {
33.        setItemCode(0);
34.        setTitle(title);
35.    }
36.
37.   /**
38.    * Retrieves the item code for the book.
39.    * @return the book's item code
40.    */
41.    public int getItemCode() {
42.        return itemCode;
43.    }
44.
```

```
45. /**
46.  * Sets the item code for the book. It is expected that the value will
47.  * be non-negative; a negative value will be rejected.
48.  * @param itemCode the book's item code
49.  */
50.  public void setItemCode (int itemCode) {
51.      if (itemCode > 0) this.itemCode = itemCode;
52.  }
53.
54. /**
55.  * Retrieves the title of the book.
56.  * @return the title of the book
57.  */
58.  public String getTitle() {
59.      return title;
60.  }
61.
62. /**
63.  * Sets the title of the book.
64.  * @param title the title of the book
65.  */
66.  public void setTitle (String title) {
67.      this.title = title;
68.  }
69.  public void display() {
70.      System.out.println(itemCode + " " + title);
71.  }
72. }
```

---

## Code Explanation

---

Even though some of the documentation will often be trivial, parameters and return values should always be documented. Any restrictions on parameter values, such as the non-negative parameters, should be mentioned.

---

## Demo 3.19: Java-Objects/Demos/UseBookWithJavadoc.java

---

```
1.  /**
2.   * Tests the {@link BookWithJavadoc} class.
3.   */
4.  public class UseBookWithJavadoc {
5.
6.      /**
7.       * Tests the {@link BookWithJavadoc} class.
8.       */
9.      public static void main(String[] args) {
10.         BookWithJavadoc b = new BookWithJavadoc(5011, "Fishing Explained");
11.         b.display();
12.     }
13. }
```

---

### Code Explanation

---

Note the use of the `{@link}` tag, since nothing else (parameter or return types) would mention the `BookWithJavadoc` class otherwise.

---

# Exercise 11: Payroll08: Creating and Using javadoc Comments

 5 to 10 minutes

---

1. Add javadoc comments to all the fields and methods in `Employee`. Don't worry about making them completely descriptive, but do document parameters and return values for the methods. Note that judicious use of copy and paste can speed up the process.
2. Run `javadoc` and view `index.html` in a browser.

## Solution: Java-Objects/Solutions/Payroll08/employees/Employee.java

---

```
1. package employees;
2. /**
3.    Represents an Employee
4. */
5. public class Employee {
6.    /**
7.    Holds the id of the next instance
8.    */
9.    private static int nextId = 1;
10. /**
11. Stores the current next id value in this instance and
12. increments <code>nextId</code>
13. */
14.    private int id = nextId++;
15. /**
16. Stores the first name
17. */
18.    private String firstName;
19. /**
20. Stores the last name
21. */
22.    private String lastName;
23. /**
24. Stores the department number
25. */
26.    private int dept;
27. /**
28. Stores the pay rate
29. */
30.    private double payRate;
31.
32. /**
33. Constructs an empty Employee
34. */
35.    public Employee() {
36.    }
37.
38. /**
39. Constructs an Employee with first and last names
40. @param firstName the first name
41. @param lastName the last name
42. */
43.    public Employee(String firstName, String lastName) {
44.        setFirstName(firstName);
```

Evaluation  
Copy

```

45.     setLastName(lastName);
46.   }
47.
48.   /**
49.    Constructs an Employee with first and last names, and department number
50.    @param firstName the first name
51.    @param lastName the last name
52.    @param dept the department number
53.    */
54.    public Employee(String firstName,String lastName, int dept) {
55.        this(firstName, lastName);
56.        setDept(dept);
57.    }
58.
59.   /**
60.    Constructs an Employee with first and last names, department number,
61.    and pay rate
62.    @param firstName the first name
63.    @param lastName the last name
64.    @param payRate the pay rate
65.    */
66.    public Employee(String firstName, String lastName, double payRate) {
67.        this(firstName, lastName);
68.        setPayRate(payRate);
69.    }
70.
71.   /**
72.    Constructs an Employee with first and lsst names, department number,
73.    and pay rate
74.    @param firstName the first name
75.    @param lastName the last name
76.    @param dept the department number
77.    @param payRate the pay rate
78.    */
79.    public Employee(
80.        String firstName, String lastName, int dept, double payRate) {
81.        this(firstName, lastName, dept);
82.        setPayRate(payRate);
83.    }
84.
85.   /**
86.    Retrieves the next id
87.    @return the nextId value
88.    */
89.    public static int getNextId() {

```



```
90.     return nextId;
91.   }
92.
93.   /**
94.   Retrieves the next id
95.   @return the nextId value
96.   */
97.   public static void setNextId(int nextId) {
98.       Employee.nextId = nextId;
99.   }
100.
101.   /**
102.   Retrieves the id
103.   @return the id
104.   */
105.   public int getId() { return id; }
106.
107.   /**
108.   Retrieves the first name
109.   @return the first name
110.   */
111.   public String getFirstName() { return firstName; }
112.
113.   /**
114.   Sets the first name
115.   @param firstName the first name
116.   */
117.   public void setFirstName(String firstName) {
118.       this.firstName = firstName;
119.   }
120.
121.   /**
122.   Retrieves the last name
123.   @return the last name
124.   */
125.   public String getLastName() { return lastName; }
126.
127.   /**
128.   Sets the last name
129.   @param lastName the last name
130.   */
131.   public void setLastName(String lastName) {
132.       this.lastName = lastName;
133.   }
134.
```

Evaluation  
Copy

```
135. /**
136. Retrieves the department
137. @return the department
138. */
139. public int getDept() { return dept; }
140.
141. /**
142. Sets the department
143. @param dept the department
144. */
145. public void setDept(int dept) {
146.     this.dept = dept;
147. }
148.
149.
150. /**
151. Retrieves the pay rate
152. @return the pay rate
153. */
154. public double getPayRate() { return payRate; }
155.
156. /**
157. Sets the pay rate
158. @param payRate the pay rate
159. */
160. public void setPayRate(double payRate) {
161.     this.payRate = payRate;
162. }
163.
164. /**
165. Retrieves the full name as first name and last name separated by a space
166. @return the full name
167. */
168. public String getFullName() {
169.     return firstName + " " + lastName;
170. }
171.
172. /**
173. Retrieves a pay information string
174. @return a String with the pay information
175. */
176. public String getPayInfo() {
177.     return "Employee " + id + " dept " + dept + " " +
178.         getFullName() + " paid " + payRate;
179. }
```

Evaluation  
Copy

180.  
181. }

*Evaluation  
Copy*

---

## Solution: Java-Objects/Solutions/Payroll08/Payroll.java

---

```
1.  import employees.*;
2.  import util.*;
3.
4.  /**
5.   * Tests the {@link Employee} class.
6.   */
7.
8.  public class Payroll {
9.
10.     public static void main(String[] args) {
11.         String fName = null;
12.         String lName = null;
13.         int dept = 0;
14.         double payRate = 0.0;
15.
16.         fName = KeyboardReader.getPromptedString("Enter first name: ");
17.         lName = KeyboardReader.getPromptedString("Enter last name: ");
18.         dept = KeyboardReader.getPromptedInt("Enter department: ");
19.         payRate = KeyboardReader.getPromptedDouble("Enter pay rate: ");
20.         Employee e1 = new Employee(fName, lName, dept, payRate);
21.
22.         fName = KeyboardReader.getPromptedString("Enter first name: ");
23.         lName = KeyboardReader.getPromptedString("Enter last name: ");
24.         dept = KeyboardReader.getPromptedInt("Enter department: ");
25.         payRate = KeyboardReader.getPromptedDouble("Enter pay rate: ");
26.         Employee e2 = new Employee(fName, lName, dept, payRate);
27.
28.         fName = KeyboardReader.getPromptedString("Enter first name: ");
29.         lName = KeyboardReader.getPromptedString("Enter last name: ");
30.         dept = KeyboardReader.getPromptedInt("Enter department: ");
31.         payRate = KeyboardReader.getPromptedDouble("Enter pay rate: ");
32.         Employee e3 = new Employee(fName, lName, dept, payRate);
33.
34.         System.out.println(e1.getPayInfo());
35.         System.out.println(e2.getPayInfo());
36.         System.out.println(e3.getPayInfo());
37.
38.     }
39. }
```



## 3.27. Primitives and Wrapper Classes

Since it is sometimes useful to treat a primitive value as if it were an object, the API provides a set of *wrapper classes* that store a primitive value inside an object, and also have useful methods for that type of primitive. Note that:

- There is a matching wrapper class for each primitive type.
- There are also two additional wrapper classes that allow for arbitrarily large values for floating point values and integer values, respectively: `BigDecimal` and `BigInteger`.
- The wrapper classes each have a constructor that take a value of the associated primitive type.
- All of the numeric wrapper classes have *all of the following methods* - inherited from the `Number` class - to retrieve numeric values:
  - `public byte byteValue()`
  - `public short shortValue()`
  - `public int intValue()`
  - `public long longValue()`
  - `public float floatValue()`
  - `public double doubleValue()`
- The other wrapper classes each have one method for retrieving the value:
  - `Boolean` has `public boolean booleanValue()`
  - `Character` has `public char charValue()`

## Primitives and Wrapper Classes

| Primitive Type | Storage Size     | Wrapper Class            | Comments                                                                                                                                        |
|----------------|------------------|--------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| boolean        | 1 bit            | Boolean                  | Not usable mathematically, but can be used with logical and bitwise operators                                                                   |
| char           | 16 bits          | Character                | Unsigned, not usable for math without converting to int                                                                                         |
| byte           | 8 bits           | Byte                     | Signed                                                                                                                                          |
| short          | 16 bits          | Short                    | Signed                                                                                                                                          |
| int            | 32 bits          | Integer                  | Signed                                                                                                                                          |
| long           | 64 bits          | Long                     | Signed                                                                                                                                          |
| float          | 32 bits          | Float                    | Signed                                                                                                                                          |
| double         | 64 bits          | Double                   | Signed                                                                                                                                          |
| void           | None             |                          |                                                                                                                                                 |
| N/A            | object reference | BigInteger<br>BigDecimal | Can store arbitrarily large or small values - cannot be used with math operators like +, -, etc., but have methods to perform those operations. |

### ❖ 3.27.1. Autoboxing and Unboxing

In earlier version of Java, you were required to explicitly code the construction of a wrapper class object from your primitive value, and to explicitly retrieve the primitive value:

#### Manually Boxing and Unboxing

```
int x = 33;  
Integer xObj = new Integer(x);  
// then, later on  
int y = xObj.intValue();
```

While you can still take this approach, Java now includes *Autoboxing*, where the primitive to object conversion and the reverse conversion will occur automatically (the compiler will put in the necessary steps)

### Autoboxing and Unboxing

```
int x = 33;  
Integer xObj = x;  
// then, later on  
int y = xObj;
```

Now, back to the formatting methods and the `Object...` parameter. Because of autoboxing, it can accept primitives, which will be boxed into wrapper objects. Hence, an `int` can now be used as an `Object` value for the `varargs` parameter - the compiler will add code to box the `int` into an `Integer`, which is then acceptable as the `Object` expected by the method.

While a useful coding convenience, there are some tricky rules regarding autoboxing when combined with implicit typecasting and method overloading, which are beyond the scope of this course.

## Java Mid-Term Exam

Please take the mid-term exam before moving on with the course.

Evaluation  
Copy

## Conclusion

In this lesson, we have learned how to define and use classes, using:

- Fields.
- Methods.
- Constructors.
- `public` and `private` access.
- Overloading.
- The `this` reference.
- Packages.
- The `Console` class.
- Autoboxing.
- Javadoc comments.

# LESSON 4

## Comparisons and Flow Control Structures

---

### Topics Covered

- ☑ Boolean values and expressions.
- ☑ Complex boolean expressions using AND and OR.
- ☑ Flow control statements using if and if... else structures.
- ☑ Comparing objects for equality and identity.
- ☑ Loops using while, do... while, and for loop structures.

### Introduction

In this lesson, you will learn about boolean values and expressions, how to create complex boolean expressions using AND and OR logic, how to write flow control statements using if and if... else structures., how to compare objects for equality and identity, and to write loops using while, do... while, and for loop structures.



## 4.1. Boolean-valued Expressions

Java has a data type called `boolean`. Note the following:

- Possible values are `true` or `false`.
- `boolean` can be operated on with logical or bitwise operators, but cannot be treated as a 0 or 1 mathematically.
- `boolean` can be specified as a parameter type or return value for a method.
- A `boolean` value will print as `true` or `false`.

The result of a conditional expression (such as a comparison operation) is a `boolean` value. Note that:

- Simple comparisons, such as greater than, equal to, etc., are allowed.

- Values are returned from functions.
- Complex combinations of simpler conditions are allowed.

Conditional expressions are used for program control. That means they provide the ability for a program to branch based on the runtime condition or iterate through a block of code until the condition is false.



## 4.2. Comparison Operators

This chart shows the comparison operators and the types of data they can be applied to.

| Operator | Purpose (Operation Performed)                   | Types of Data |                                |                      |
|----------|-------------------------------------------------|---------------|--------------------------------|----------------------|
|          |                                                 | boolean       | Numeric Primitives<br>and char | Object<br>References |
| ==       | is equal to (note the <i>two</i> equals signs!) | X             | X                              | X                    |
| !=       | is not equal to                                 | X             | X                              | X                    |
| >        | is greater than                                 |               | X                              |                      |
| <        | is less than                                    |               | X                              |                      |
| >=       | is greater than or equal to                     |               | X                              |                      |
| <=       | is less than or equal to                        |               | X                              |                      |

### Warning

It is unwise to use == or != with floating-point data types, as they can be imprecise.



## 4.3. Comparing Objects

The operators == and != test if two references point to *exactly the same object* in memory; they test that the numeric values of the two references are the same.

The `equals(Object o)` method compares the contents of two objects to see if they are the same (you can override this method for your classes to perform any test you want).



## 4.4. Conditional Expression Examples

The following are conditional expression examples.

If `a = 3`, `b = 4`; then:

- `a == b` will evaluate to `false`
- `a != b` will evaluate to `true`
- `a > b` will evaluate to `false`
- `a < b` will evaluate to `true`
- `a >= b` will evaluate to `false`
- `a <= b` will evaluate to `true`

```
String s = "Hello";  
String r = "Hel";  
String t = r + "lo";
```

Evaluation  
Copy

`s == t` will evaluate to `false`

`s != t` will evaluate to `true` (they are not the same object - they are two different objects that have the same contents)

`s.equals(t)` will evaluate to `true`

```
String s = "Hello";  
String t = s;
```

`s == t` will evaluate to `true`

`s != t` will evaluate to `false` (they are the same object)

`s.equals(t)` will evaluate to `true`

Note: Java will *intern* a literal String that is used more than once in your code; that is, it will use the same location for all occurrences of that String

```
String s = "Hello";  
String t = "Hello";
```

`s == t` will evaluate to `true`, because the String object storing "Hello" is stored only once, and both `s` and `t` reference that location

`s != t` will evaluate to `false` (they are the same object)

`s.equals(t)` will evaluate to `true`



## 4.5. Complex boolean Expressions

Java has operators for combining boolean values with AND and OR logic, and for negating a value (a NOT operation). Note that:

- There are also bitwise operators for AND, OR, and bitwise inversion (changing all 1 bits to 0, and vice versa).
- Since a boolean is stored as one bit, the bitwise AND and OR operators will give the same logical result, but will be applied differently (see below).
- For some reason, the bitwise NOT cannot be applied to a boolean value.

| Logical                 |             | Bitwise            |                         |
|-------------------------|-------------|--------------------|-------------------------|
| <code>&amp;&amp;</code> | logical AND | <code>&amp;</code> | bitwise AND             |
| <code>  </code>         | logical OR  | <code> </code>     | bitwise OR              |
| <code>!</code>          | logical NOT | <code>~</code>     | bitwise NOT (inversion) |

Examples:

| Code                                                      | Effect                                                                                       |
|-----------------------------------------------------------|----------------------------------------------------------------------------------------------|
| Testing if a value falls within a range                   |                                                                                              |
| <code>( a &gt;= 0 ) &amp;&amp; ( a &lt;= 10 )</code>      | is true if a falls between 0 and 10; it must satisfy both conditions                         |
| Testing if a value falls outside a range                  |                                                                                              |
| <code>( a &lt; 0 )    ( a &gt; 10 )</code>                | is true if a falls outside the range 0 to 10; it may be either below or above that range     |
| <code>!( ( a &gt;= 0 ) &amp;&amp; ( a &lt;= 10 ) )</code> | inverts the test for a between 0 and 10; so a must be outside the range instead of inside it |

The `&&` and `||` operations are called *short-circuiting*, because if the first condition determines the final outcome, the second condition is not evaluated.

#### Note

To force both conditions to be evaluated, use `&` and `|` for AND and OR, respectively.

Example: to test if a reference is `null` before calling a `boolean` valued method on it.

```
String s = request.getParameter("shipovernite");  
if (s != null && s.equalsIgnoreCase("YES")) { . . . }
```

`equalsIgnoreCase` will be called only if the reference is not `null`.



## 4.6. Simple Branching

The result of a conditional expression can be used to control program flow.



## 4.7. The if Statement

```
if (condition) statement;
```

or

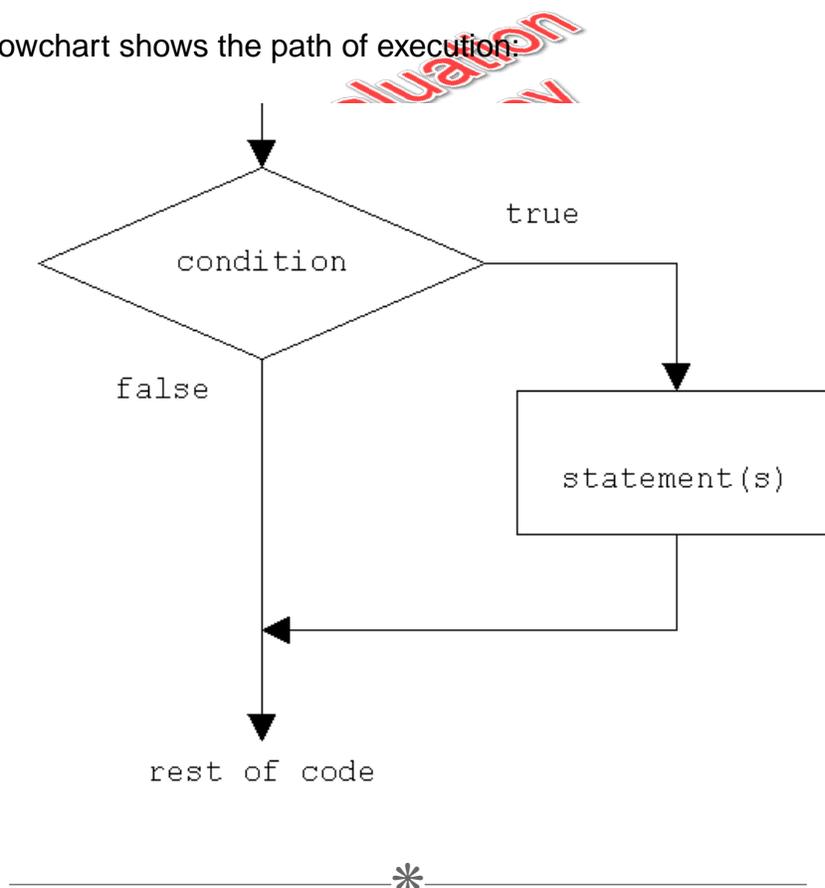
```
if (condition) {  
    zero to many statements;  
}
```

Causes “one thing” to occur when a specified condition is met.

The one thing may be a single statement or a block of statements in curly braces. The remainder of the code (following the `if` and the statement or block it owns) is then executed regardless of the result of the condition.

The conditional expression is placed within parentheses.

The following flowchart shows the path of execution:



## 4.8. if Statement Examples

### ❖ 4.8.1. Absolute Value

If we needed the absolute value of a number (a number that would always be positive):

#### **Demo 4.1: Java-Control/Demos/If1.java**

---

```
1.  public class If1 {
2.      public static void main(String[] args) {
3.          int a = 5;
4.          System.out.println("original number is: " + a);
5.          if ( a < 0 ) { a = -a; }
6.          System.out.println("absolute value is: " + a);
7.          a = -10;
8.          System.out.println("original number is: " + a);
9.          if ( a < 0 ) { a = -a; }
10.         System.out.println("absolute value is: " + a);
11.     }
12. }
```

---

### ❖ 4.8.2. Random Selection

Task: write a brief program which generates a random number between 0 and 1. Print out that the value is in the low, middle, or high third of that range (`Math.random()` will produce a `double` value from 0.0 up to but not including 1.0).

- Although it is a bit inefficient, just do three separate tests: (note that `Math.random()` will produce a number greater than or equal to 0.0 and less than 1.0).

## Demo 4.2: Java-Control/Demos/If2.java

---

```
1. public class If2 {
2.     public static void main(String[] args) {
3.         double d = Math.random();
4.         System.out.print("The number " + d);
5.         if (d < 1.0/3.0)
6.             System.out.println(" is low");
7.         if (d >= 1.0/3.0 && d < 2.0/3.0)
8.             System.out.println(" is middle");
9.         if (d >= 2.0/3.0)
10.            System.out.println(" is high");
11.     }
12. }
```

---

# Exercise 12: Game01: A Guessing Game

 15 to 20 minutes

---

Write a program called `Game` that will ask the user to guess a number and compare their guess to a stored integer value between 1 and 100.

1. Open the file `Java-Control/Exercises/Game01/Game.java`.
2. Use a field called `answer` to store the expected answer.
3. For now, just hard code the stored value; we will create a random value later (your code will be easier to debug if you know the correct answer).
4. Create a method called `play()` that holds the logic for the guessing. Use the `KeyboardReader` class (provided for you in the `util` directory) to ask for a number between 1 and 100, read the result, and tell the user if they are too low, correct, or too high.
5. Create a `main` method, have it create a new instance of `Game` and call `play()`.

## Solution: Java-Control/Solutions/Game01/Game.java

---

```
1.  import util.*;
2.
3.  public class Game {
4.      private int answer = 67;
5.
6.      public void play() {
7.          int guess;
8.          guess = KeyboardReader.getPromptedInt("Enter a number 1 -100: ");
9.          if (guess < answer) System.out.println("Too low");
10.         if (guess > answer) System.out.println("Too high");
11.         if (guess == answer) System.out.println("Correct!");
12.     }
13.
14.     public static void main(String[] args) {
15.         new Game().play();
16.     }
17. }
```

---

**Evaluation  
Copy**

### Code Explanation

---

Each of the the three possible cases is tested individually as shown below. All three tests will always be performed. In the next version of this program we will use a more efficient approach, making the tests mutually exclusive so that processing stops when one is true.

#### Game01 Logic

```
if (guess < answer) System.out.println("Too low");
if (guess > answer) System.out.println("Too high");
if (guess == answer) System.out.println("Correct!");
```

---

In the next exercise, we'll ask you to use `if` statements to validate values passed as parameters to some of the "set" methods in the `Employee` class.

# Exercise 13: Payroll-Control01: Modified Payroll

 10 to 15 minutes

---

We will modify our payroll program to check the pay rate and department values.

1. The `Employee` class should protect itself from bad data, so modify `setPayRate` and `setDept` to check the incoming data (and ignore it if it is less than 0).
2. Test your program to see what happens with negative input values.
3. The code using `Employee` should avoid sending it bad data, so also change `main` in `Payroll.java` to check for acceptable pay rate and department values (print an error message for any negative entry, then just set that variable to 0 for lack of anything better that we can do right now. Later we will find a way to ask the user for a new value instead)

## Solution:

### Java-Control/Solutions/Payroll-Control01/employees/Employee.java

---

```
1.  package employees;
2.
3.  public class Employee {
4.      private static int nextId = 1;
5.      private int id = nextId++;
6.      private String firstName;
7.      private String lastName;
8.      private int dept;
9.      private double payRate;
10.
11.     public Employee() {
12.     }
13.     public Employee(String firstName, String lastName) {
14.         setFirstName(firstName);
15.         setLastName(lastName);
16.     }
17.     public Employee(String firstName,String lastName, int dept) {
18.         this(firstName, lastName);
19.         setDept(dept);
20.     }
21.     public Employee(String firstName, String lastName, double payRate) {
22.         this(firstName, lastName);
23.         setPayRate(payRate);
24.     }
25.     public Employee(String firstName, String lastName, int dept, double payRate) {
26.         this(firstName, lastName, dept);
27.         setPayRate(payRate);
28.     }
29.
30.     public static int getNextId() {
31.         return nextId;
32.     }
33.
34.     public static void setNextId(int nextId) {
35.         Employee.nextId = nextId;
36.     }
37.
38.     public int getId() { return id; }
39.
40.     public String getFirstName() { return firstName; }
41.
42.     public void setFirstName(String firstName) {
43.         this.firstName = firstName;
```

```
44.     }
45.
46.     public String getLastName() { return lastName; }
47.
48.     public void setLastName(String lastName) {
49.         this.lastName = lastName;
50.     }
51.
52.     public int getDept() { return dept; }
53.
54.     public void setDept(int dept) {
55.         if (dept > 0) this.dept = dept;
56.     }
57.
58.
59.     public double getPayRate() { return payRate; }
60.
61.     public void setPayRate(double payRate) {
62.         if (payRate >= 0) this.payRate = payRate;
63.     }
64.
65.     public String getFullName() {
66.         return firstName + " " + lastName;
67.     }
68.
69.     public String getPayInfo() {
70.         return "Employee " + id + " dept " + dept + " " +
71.             getFullName() + " paid " + payRate;
72.     }
73.
74. }
```

---

## Code Explanation

---

The `Employee` class should protect itself from bad incoming data, so the `setPayRate` method simply ignores a value less than 0. A better practice, which we will add later, would be to throw an exception when an illegal value is passed in. Note the benefit of coding the constructors to use the `setPayRate` method; we do not have to go back and revise their code as well. (`setDept` has similar changes.)

---

## Solution: Java-Control/Solutions/Payroll-Control01/Payroll.java

---

```
1.  import employees.*;
2.  import util.*;
3.
4.  public class Payroll {
5.      public static void main(String[] args) {
6.          Employee e1 = null, e2 = null, e3 = null;
7.          String fName = null;
8.          String lName = null;
9.          int dept = 0;
10.         double payRate = 0.0;
11.
12.         fName = KeyboardReader.getPromptedString("Enter first name: ");
13.         lName = KeyboardReader.getPromptedString("Enter last name: ");
14.         dept = KeyboardReader.getPromptedInt("Enter department: ");
15.         if (dept <= 0) {
16.             System.out.println("Department must be > 0");
17.             dept = 0;
18.         }
19.         payRate = KeyboardReader.getPromptedDouble("Enter payRate: ");
20.         if (payRate < 0.0) {
21.             System.out.println("Pay rate must be >= 0");
22.             payRate = 0.0;
23.         }
24.         e1 = new Employee(fName, lName, dept, payRate);
25.         System.out.println(e1.getPayInfo());
26.
27.         fName = KeyboardReader.getPromptedString("Enter first name: ");
28.         lName = KeyboardReader.getPromptedString("Enter last name: ");
29.         dept = KeyboardReader.getPromptedInt("Enter department: ");
30.         if (dept <= 0) {
31.             System.out.println("Department must be > 0");
32.             dept = 0;
33.         }
34.         payRate = KeyboardReader.getPromptedDouble("Enter pay rate: ");
35.         if (payRate < 0.0) {
36.             System.out.println("Pay rate must be >= 0");
37.             payRate = 0.0;
38.         }
39.         e2 = new Employee(fName, lName, dept, payRate);
40.         System.out.println(e2.getPayInfo());
41.
42.         fName = KeyboardReader.getPromptedString("Enter first name: ");
43.         lName = KeyboardReader.getPromptedString("Enter last name: ");
44.         dept = KeyboardReader.getPromptedInt("Enter department: ");
```

```
45.     if (dept <= 0) {
46.         System.out.println("Department must be > 0");
47.         dept = 0;
48.     }
49.     payRate = KeyboardReader.getPromptedDouble("Enter pay rate: ");
50.     if (payRate < 0.0) {
51.         System.out.println("Pay rate must be >= 0");
52.         payRate = 0.0;
53.     }
54.     e3 = new Employee(fName, lName, dept, payRate);
55.     System.out.println(e3.getPayInfo());
56. }
57. }
```

---

## Code Explanation

---

Even though `Employee` now protects itself from illegal incoming values, it should really be the responsibility of the programmers for the classes using `Employee` to avoid sending it bad values. So, the `payRate` value is tested here as well. While this may seem to decrease efficiency (since, presumably, a non-object-oriented program might be able to avoid testing the value twice in a row), the maintainability aspects of OOP are still considered to outweigh the loss in efficiency.

---



## 4.9. Two Mutually Exclusive Branches

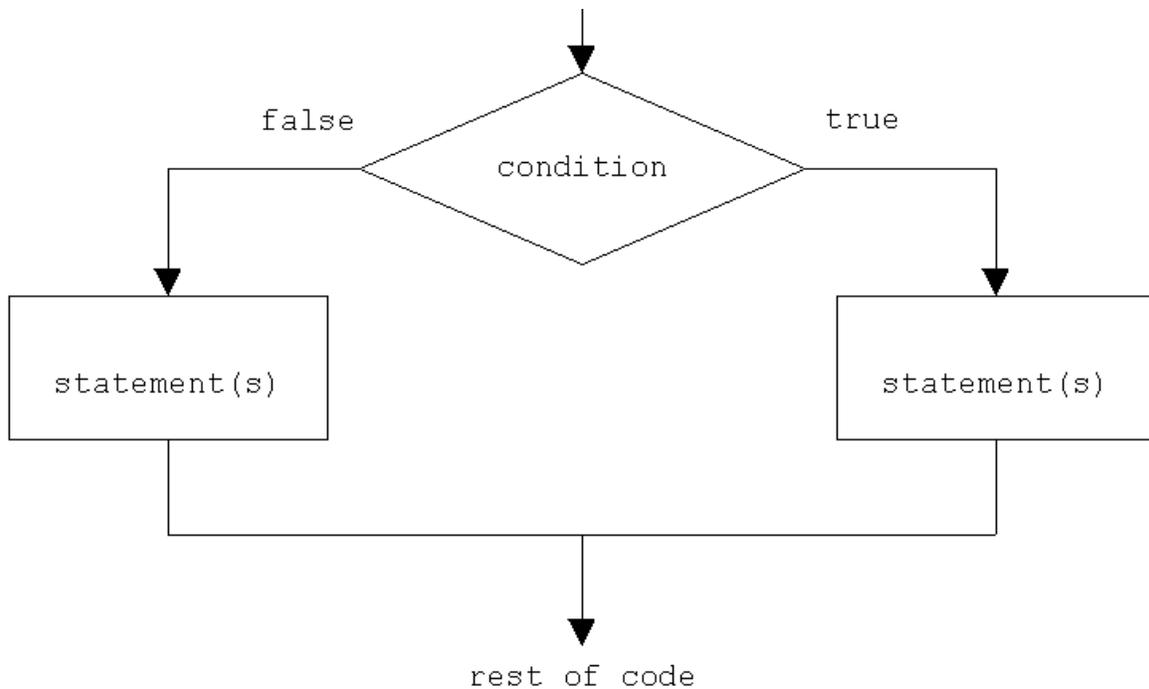
### ❖ 4.9.1. The `if ... else` Statement

```
        if (condition) {
statement;
} else if (condition) {
statement;
} else {
statement;
}
```

The if ... else Statement does “one thing” if a condition is true, and a different thing if it is false.

It is never the case that both things are done. The “one thing” may be a single statement or a block of statements in curly braces.

A statement executed in a branch may be any statement, including another if or if ... else statement.



This program tells you that you are a winner on average once out of every four tries.

### Demo 4.3: Java-Control/Demos/IfElse1.java

```
1. public class IfElse1 {
2.     public static void main(String[] args) {
3.         double d = Math.random();
4.         if (d < .25) System.out.println("You are a winner!");
5.         else System.out.println("Sorry, try again!");
6.     }
7. }
```

## ❖ 4.9.2. Nested `if ... else` Statements - Comparing a Number of Mutually Exclusive Options

You can *nest* `if ... else` statements, so that an `if` or `else` clause contains another test. Both the `if` and the `else` clause can contain any type of statement, including another `if` or `if ... else`.

You can test individual values or ranges of values. Once an `if` condition is true, the rest of the branches will be skipped. You could also use a sequence of `if` statements without the `else` clauses (but this doesn't by itself force the branches to be mutually exclusive).

Here is the low/middle/high example rewritten using `if ... else`

### Demo 4.4: Java-Control/Demos/IfElse2.java

---

```
1.  public class IfElse2 {
2.      public static void main(String[] args) {
3.          double d = Math.random();
4.          System.out.print("The number " + d);
5.          if (d < 1.0/3.0)
6.              System.out.println(" is low");
7.          else if (d < 2.0/3.0)
8.              System.out.println(" is middle");
9.          else
10.             System.out.println(" is high");
11.     }
12. }
```

---

#### Code Explanation

---

Note that we only need one test for the middle value. If we reach that point in the code, `d` must be greater than `1.0/3.0`.

---

Similarly, we do not test the high third at all. The original version worked because there was no chance that more than one message would print; that approach is slightly less efficient because all three tests will always be made. In the `if ... else` version, comparing stops once a match has been made.

# Exercise 14: Game02: A Revised Guessing Game

 10 to 15 minutes

---

1. Revise your number guessing program to use `if . . . else` logic (you can test for *too low* and *too high*, and put the message for *correct* in the final `else` branch).
2. Once you have done that, here is a way to generate a random answer between 1 and 100:

A. At the top:

```
import java.util.*;
```

B. Add a `private` field for a random number generator:

```
private Random r = new Random();
```

C. Then, you can initialize the `answer` field:

```
answer = r.nextInt(100) + 1;
```

the `nextInt(int n)` method generates a number greater than or equal to 0 and less than `n`, so `r.nextInt(100)` would range from 0 through 99; we need to add 1 to raise both ends of the range.

D. You might want to print the expected correct answer to aid debugging.

Note that until we cover looping, there will be no way to truly “play” the game, since we have no way to preserve the value between runs.



## Solution: Java-Control/Solutions/Game02/Game.java

---

```
1. import util.*;
2. import java.util.*;
3.
4. public class Game {
5.     private Random r = new Random();
6.     private int answer = r.nextInt(100) + 1;
7.
8.     public void play() {
9.         System.out.println(answer);
10.        int guess;
11.        guess = KeyboardReader.getPromptedInt("Enter a number 1 -100: ");
12.        if (guess < answer) System.out.println("Too low");
13.        else if (guess > answer) System.out.println("Too high");
14.        else System.out.println("Correct!");
15.    }
16.
17.    public static void main(String[] args) {
18.        new Game().play();
19.    }
20. }
```

---

Evaluation  
\*  
Copy

## 4.10. Comparing a Number of Mutually Exclusive options - The switch Statement

### ❖ 4.10.1. The switch Statement

A switch expression (usually a variable) is compared against a number of possible values. It is used when the options are each a single, constant value that is exactly comparable (called a *case*).

The switch expression must be a `String`, `byte`, `char`, `short`, or `int`. Cases may only be `String`, `byte`, `char`, `short`, or `int` values; in addition, their magnitude must be within the range of the switch expression data type and cannot be used with floating-point datatypes or `long` and cannot compare an option that is a range of values, unless it can be stated as a list of possible values, each treated as a separate case.

Cases are listed under the switch control statement, within curly braces, using the `case` keyword. Once a match is found, all executable statements below that point are executed,

including those belonging to later cases; this allows stacking of multiple cases that use the same code.

The `break;` statement is used to jump out of the switch block, thus skipping executable steps that are not desired. The `default` case keyword catches all cases not matched above - note that the `default` case does not need to be the last thing in the `switch`. Note that technically speaking, the cases are *labeled lines*; the switch jumps to the first label whose value matches the switch expression.

```
switch ( expression ) {
case constant1:
    statements;
    break;
case constant2:
    statements;
    break;
. . .
case constant3:
case constant4:
    //do these following statements for case constant3 & constant4
    statements;
    break;
. . .
[default:
    statements;]
}
```



## ❖ 4.10.2. switch Statement Examples

### Demo 4.5: Java-Control/Demos/Switch1.java

---

```
1.  import util.KeyboardReader;
2.
3.  public class Switch1 {
4.      public static void main(String[] args) {
5.          char c = 'X';
6.          c = KeyboardReader.getPromptedChar("Enter a letter a - d: ");
7.          switch (c) {
8.              case 'a':
9.              case 'A': System.out.println("A is for Aardvark");
10.                 break;
11.             case 'b':
12.             case 'B': System.out.println("B is for Baboon");
13.                 break;
14.             case 'c':
15.             case 'C': System.out.println("C is for Cat");
16.                 break;
17.             case 'd':
18.             case 'D': System.out.println("D is for Dog");
19.                 break;
20.             default: System.out.println("Not a valid choice");
21.         }
22.     }
23. }
```

---

Three points to note:

- Stacking of cases for uppercase and lowercase letters allows both possibilities.
- `break;` statements separate code for different cases.
- The `default:` clause is used to catch all other cases not explicitly handled.

Here is a revised version that moves the `default` to the top, so that a bad entry is flagged with an error message, but then treated as an 'A' - note that there is no `break` below the `default` case.

## Demo 4.6: Java-Control/Demos/Switch2.java

---

```
1.  import util.KeyboardReader;
2.
3.  public class Switch2 {
4.      public static void main(String[] args) {
5.          char c = 'X';
6.          c = KeyboardReader.getPromptedChar("Enter a letter a - d: ");
7.          switch (c) {
8.              default: System.out.println("Not a valid choice - assuming 'A'");
9.              case 'a':
10.                 case 'A': System.out.println("A is for Aardvark");
11.                     break;
12.                 case 'b':
13.                 case 'B': System.out.println("B is for Baboon");
14.                     break;
15.                 case 'c':
16.                 case 'C': System.out.println("C is for Cat");
17.                     break;
18.                 case 'd':
19.                 case 'D': System.out.println("D is for Dog");
20.                     break;
21.             }
22.         }
23.     }
```

---

Another example is taking advantage of the “fall-through” behavior without a break statement.

## Demo 4.7: Java-Control/Demos/Christmas.java

---

```
1.  import util.KeyboardReader;
2.
3.  public class Christmas {
4.      public static void main(String[] args) {
5.          int day = KeyboardReader.getPromptedInt("What day of Christmas? ");
6.          System.out.println(
7.              "On the " + day + " day of Christmas, my true love gave to me:");
8.          switch (day) {
9.              case 12: System.out.println("Twelve drummers drumming,");
10.             case 11: System.out.println("Eleven pipers piping,");
11.             case 10: System.out.println("Ten lords a-leaping,");
12.             case 9: System.out.println("Nine ladies dancing,");
13.             case 8: System.out.println("Eight maids a-milking,");
14.             case 7: System.out.println("Seven swans a-swimming,");
15.             case 6: System.out.println("Six geese a-laying,");
16.             case 5: System.out.println("Five golden rings,");
17.             case 4: System.out.println("Four calling birds,");
18.             case 3: System.out.println("Three French hens,");
19.             case 2: System.out.println("Two turtle doves, and a ");
20.             case 1: System.out.println("Partridge in a pear tree!");
21.         }
22.     }
23. }
```

---

## Exercise 15: Game03: Multiple Levels

 15 to 30 minutes

---

What if we want to offer gamers multiple levels of difficulty in our game? We could make the random number multiplier a property of the `Game` class, and set a value into it with a constructor, after asking the user what level they'd like to play.

1. Open `Java-Control/Exercises/Game03/Game.java`.
2. Add a `range` field to the `Game` class (an `int`).
3. Add a constructor that accepts a `level` parameter; use a `char`.
4. Within that constructor, use a `switch` to process the incoming `level`:
  - Uppercase or lowercase `B` means *Beginner*, set the range to 10.
  - `I` means *Intermediate*; set the range to 100.
  - `A` means *Advanced*; set the range to 1000.
  - Any other value results in a *Beginner* game; after all, if they can't answer a simple question correctly, how could we expect them to handle something more advanced.
  - You could put the default option stacked above the "B" cases, so that you can print an error message and then fall through to the 'B' logic
5. Use `range` as the parameter when you call the `Random` object's `nextInt` method (move the logic that creates the `Random` object and generates the answer to this constructor).
6. The prompt given by the `play` method should now take the `range` into account.
7. In the `main` method, ask the user for the level and call the new constructor with their response.

## Solution: Java-Control/Solutions/Game03/Game.java

---

```
1.  import util.*;
2.  import java.util.*;
3.
4.  public class Game {
5.      private Random r = new Random();
6.      private int answer;
7.      private int range = 10;
8.
9.      public Game(char level) {
10.         switch (level) {
11.             default:
12.                 System.out.println("Invalid option: " +
13.                     level + ", using Beginner");
14.             case 'b':
15.             case 'B':
16.                 range = 10;
17.                 break;
18.             case 'i':
19.             case 'I': range = 100;
20.                 break;
21.             case 'a':
22.             case 'A': range = 1000;
23.                 break;
24.         }
25.         Random r = new Random();
26.         answer = r.nextInt(range) + 1;
27.         //System.out.println(answer);
28.     }
29.
30.     public void play() {
31.         int guess;
32.         guess = KeyboardReader.getPromptedInt("Enter a number 1 - " + range + ": ");
33.         if (guess < answer) System.out.println("Too low");
34.         else if (guess > answer) System.out.println("Too high");
35.         else System.out.println("Correct!");
36.     }
37.
38.     public static void main(String[] args) {
39.         char level = KeyboardReader.getPromptedChar("What level (B, I, A)? ");
40.         new Game(level).play();
41.     }
42. }
```



## Code Explanation

---

The `switch` tests for the three letters, stacking cases for uppercase and lowercase values. The `default` catches all other responses and falls through to the *Beginner* logic.

---



## 4.11. Comparing Two Objects

When comparing two objects, the `==` operator compares the *references* and not the values of the objects. Similarly, `!=` tests to see if the references point to two different objects (even if they happen to have the same internal values).

The `Object` class defines an `equals(Object)` method intended to compare the contents of the objects. The code written in the `Object` class simply compares the references using `==`. This method is overridden for most API classes to do an appropriate comparison. For example, with `String` objects, the method compares the actual characters up to the end of the string. For classes you create, you would override this method to do whatever comparisons you deem appropriate.

## Demo 4.8: Java-Control/Demos/Rectangle.java

---

```
1.  public class Rectangle {
2.      private int height;
3.      private int width;
4.
5.      public Rectangle(int height, int width) {
6.          this.height = height;
7.          this.width = width;
8.      }
9.
10.     public int getArea() {
11.         return height * width;
12.     }
13.
14.     public boolean equals(Object other) {
15.         if (other instanceof Rectangle) {
16.             Rectangle otherRect = (Rectangle) other;
17.             return this.height == otherRect.height &&
18.                 this.width == otherRect.width;
19.         }
20.         else return false;
21.     }
22. }
```

---

Evaluation  
Copy

### Code Explanation

---

The `equals` method compares another object to this object.

This example is necessarily somewhat complicated. It involves a few concepts we haven't covered yet, including inheritance. Because the `equals` method inherited from `Object`, it takes a parameter which is an `Object`, we need to keep that type for the parameter (technically, we don't need to do it by the rules of Java inheritance, but because other tools in the API are coded to pass an `Object` to this method). But, that means that someone could call this method and pass in something else, like a `String`.

So, the first thing we need to do is test that the parameter was actually a `Rectangle`. If so, we can work with it; if not, there is no way it could be equal, so we return `false`.

We will cover `instanceof` later, but, for now, we can assume it does what it implies: test that the object we received was an instance of `Rectangle`. Even after that test, in order to treat it as a `Rectangle`, we need to do a typecast to explicitly store it into a `Rectangle`

variable (and again, we will cover object typecasting later). Once we have it in a `Rectangle` variable, we can check its `height` and `width` fields.

In yet another complication, if we write an `equals` method, we should really write a `public int hashCode()` method as well, since their meanings are interrelated; two elements that compare as equal should produce the same `hashCode`.

As an aside, note that private data in one instance of a class *is* visible to other instances of the same class; it is only *other classes* that cannot see the private elements.

---

## Demo 4.9: Java-Control/Demos/ObjectEquivalenceIdentity.java

---

```
1.  class ObjectEquivalenceIdentity {
2.
3.      public static void main(String[] args) {
4.          Rectangle r1 = new Rectangle(10, 20);
5.          Rectangle r2 = new Rectangle(10, 20);
6.          if (r1 == r2)
7.              System.out.println("Same object");
8.          else
9.              System.out.println("Different objects");
10.         if (r1.equals(r2))
11.             System.out.println("Equal");
12.         else
13.             System.out.println("Not equal");
14.     }
15. }
```

---

### Code Explanation

---

Since all the important work is done in `Rectangle`, all we need to do here is instantiate two and compare them using both `==` and the `equals` method to see the differing results.

The output should be:

```
Different objects
Equal
```

---

## ❖ 4.11.1. Testing Strings for Equivalence

In the example below, we show a brief program which has a word stored as a string of text. The program prints a message asking for a word, reads it from the user's input, then tests if they are the same or not.

- You can preset the message assuming that they are incorrect, then change it if they are correct

### Demo 4.10: Java-Control/Demos/StringEquals.java

---

```
1. import util.KeyboardReader;
2.
3. public class StringEquals {
4.     public static void main(String[] args) {
5.         String s = "Hello";
6.         String t = KeyboardReader.getPromptedString("Enter a string: ");
7.         String message = "That is incorrect";
8.         if (s.equals(t)) message = "That is correct";
9.         System.out.println(message);
10.    }
11. }
```

---

- Try this, then change `equals(t)` to `equalsIgnoreCase(t)`



## 4.12. Conditional Expression

Java uses the same conditional expression (sometimes called a “ternary operator”) as C and C++

```
condition ? expressionIfTrue : expressionIfFalse
```

This performs a conditional test in an expression, resulting in the first value if the condition is true, or the second if the condition is false

Note: due to operator precedence issues, it is often best to enclose the entire expression in parentheses. Here's a brief example:

## Demo 4.11: Java-Control/Demos/Conditional.java

---

```
1. public class Conditional {
2.     public static void main(String[] args) {
3.         int i = -6;
4.         String s;
5.         s = "i is " + ((i >= 0) ? "positive" : "negative");
6.         System.out.println(s);
7.     }
8. }
```

---

Note that the parentheses around the test are not necessary by the operator precedence rules, but may help make the code clearer.

The parentheses around the entire conditional expression *are* necessary; without them, precedence rules would concatenate the boolean result onto the initial string, and then the `?` operator would be flagged as an error, since the value to its left would not be a boolean.

### ❖ 4.12.1. while and do . . . while Loops

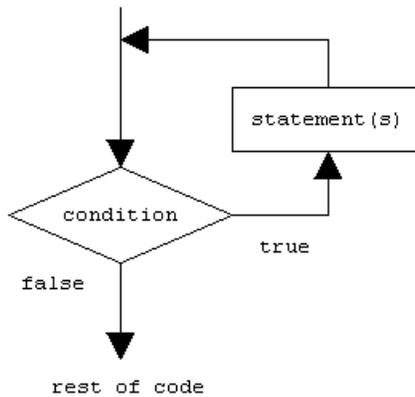
Again, the loops in Java are pretty much the same as in C - with the exception that the conditional expressions must evaluate to only `boolean` values

#### while Loops

```
while (condition) statement;
```

*or*

```
while (condition){
    block of statements
}
```



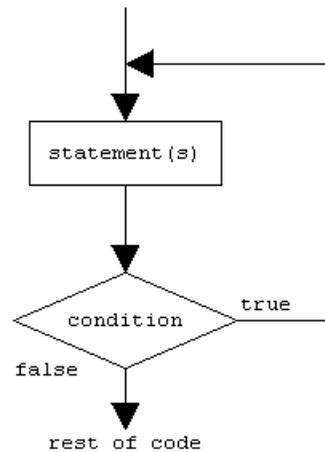
- The loop continues as long as the expression evaluates to true.
- The condition is evaluated before the start of each pass.
- It is possible that the body of the loop never executes at all (if the condition is false the first time).

### do ... while loops

```
do statement; while (condition);
```

*or*

```
do {
  block of statements
} while (condition);
```

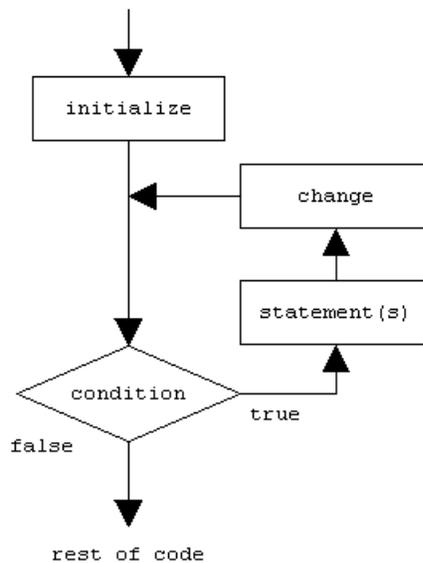


- The condition is evaluated after the end of each pass.
- The body of the loop will always execute at least once, even if the condition is false the first time.

#### ❖ 4.12.2. for Loops

A for loop uses a counter to progress through a series of values:

- A value is initialized, tested each time through, and then modified (usually incremented) at the end of each pass, before it is tested again.
- The for loop does not do anything that cannot be done with a while loop, but it puts everything that controls the looping at the top of the block.



```
for (initialize; condition; change)
    statement;
```

or

```
for (initialize; condition; change) {
    block of statements
}
```

Evaluation Copy

for loops can use a variable declared outside of the control portion of the loop or in the control portion. The latter gives the variable *block-level scope* (existing only for the duration of the loop).

#### Counter Variable Declared Before Loop

```
int j;
for (j = 0; j < 12; j++ )
    System.out.print("j = " + j);
```

#### Block Level Counter Variable

```
for (int j = 0; j < 12; j++ )
    System.out.print("j = " + j);
```

A `for` loop may use multiple control variables by using the *sequence* operator, the comma ( , ):

#### **Multiple Counters in a for Loop**

```
for (int j = 0, k = 12; j <= 12; j++, k-- )  
    System.out.println(j + " " + k);
```

Note: if you use a block-scope variable (such as above) for the first counter, the additional variables will be block scope as well, and also will be the same type of data - i.e., the variable `k` above also exists only for the duration of the block. There is no way to declare two counters of different types at block-level scope.

Note that neither `while` nor `do . . . while` loops allow declaring a looping variable with this type of scope. The looping variable must be declared in advance.

### ❖ 4.12.3. For-Each Loops

The *for-each* loop allows us to loop through an array or collection class instance using a simplified syntax:

```
for (type  
    variable : arrayOrCollection) {  
    body of loop  
}
```

The looping variable is not a counter - it will contain each element of the array or collection in turn (the actual value and not an index to it, so its type should be the same as the type of items in the array or collection). You can read the `:` character as if it is the word “from”. We will cover this type of loop in more depth in the Arrays section.

The looping variable must be declared within the parentheses controlling the loop - you cannot use a preexisting variable.

Since the looping variable is a local variable, it gets a copy of each value from the array or collection. Therefore you cannot use the *for-each* loop to write values back to the array/collection - assigning a new value to the variable in the body of the loop is only overwriting the local copy.

## Demo 4.12: Java-Control/Demos/Loops1.java

---

```
1.  public class Loops1 {
2.      public static void main(String[] args) {
3.          int i = 1;
4.
5.          System.out.println("While loop:");
6.          while (i <= 512) {
7.              System.out.println("i is " + i);
8.              i *= 2;
9.          }
10.         System.out.println("i is now " + i);
11.
12.         System.out.println("Do while loop:");
13.         do {
14.             i = i - 300;
15.             System.out.println("i is now " + i);
16.         }
17.         while (i > 0);
18.
19.         System.out.println("For loop:");
20.         for (i = 0; i < 12; i++)
21.             System.out.print(" " + i);
22.         System.out.println();
23.
24.         System.out.println("For loop that declares a counter:");
25.         for (int j = 0; j < 12; j++)
26.             System.out.print(" " + j);
27.         System.out.println();
28.
29.         System.out.println("For-each loop:");
30.         String[] names = { "Jane", "John", "Bill" };
31.         for (String oneName : names)
32.             System.out.println(oneName.toUpperCase());
33.     }
34. }
```

---

# Exercise 16: Payroll-Control02: Payroll With a Loop

 20 to 40 minutes

---

1. Open the files in `Java-Control/Exercises/Payroll-Control02/`.
2. Revise your payroll program to use only one `Employee` variable.
3. Use a loop to repeatedly create a new instance to store in it, populate it from the keyboard, and display it on the screen (you can ask after each one if the user wants to enter another, or just use a loop that has a fixed number of iterations).
4. Also, change the logic for reading the pay rate and department values from the keyboard to use `do . . . while` loops that will continue to loop as long as the user enters invalid values (note that you will need to separate declaring the variables from populating them - declare each of them before their loop starts, in order for the variable to be available to the test at the end of the loop).

## Solution: Java-Control/Solutions/Payroll-Control02/Payroll.java

---

```
1.  import employees.*;
2.  import util.*;
3.
4.  public class Payroll {
5.      public static void main(String[] args) {
6.          Employee e;
7.          String fName, lName;
8.          int dept;
9.          double payRate;
10.
11.         for (int i = 0; i < 3; i++) {
12.             fName = KeyboardReader.getPromptedString("Enter first name: ");
13.             lName = KeyboardReader.getPromptedString("Enter last name: ");
14.             do {
15.                 dept = KeyboardReader.getPromptedInt("Enter department: ");
16.                 if (dept <= 0) System.out.println("Department must be >= 0");
17.             } while (dept <= 0);
18.             do {
19.                 payRate = KeyboardReader.getPromptedDouble("Enter pay rate: ");
20.                 if (payRate < 0.0) System.out.println("Pay rate must be >= 0");
21.             } while (payRate < 0.0);
22.             e = new Employee(fName, lName, dept, payRate);
23.             System.out.println(e.getPayInfo());
24.         }
25.     }
26. }
```

---

# Exercise 17: Game04: Guessing Game with a Loop

 10 to 20 minutes

---

1. Open `Java-Control/Exercises/Game04/Game.java`.
2. Revise the number guessing program to force the user to guess until they are correct - that is, to keep guessing as long as they are incorrect.
3. Then add more logic to ask if they want to play again, read a “Y” or “N” character as their answer, and loop as long as they enter a “Y”.

## Solution: Java-Control/Solutions/Game04/Game.java

---

```
1.  import util.*;
2.  import java.util.*;
3.
4.  public class Game {
5.      private Random r = new Random();
6.      private int answer;
7.      private int range = 10;
8.
9.      public Game(char level) {
10.         switch (level) {
11.             default:
12.                 System.out.println("Invalid option: " +
13.                     level + ", using Beginner");
14.             case 'b':
15.             case 'B':
16.                 range = 10;
17.                 break;
18.             case 'i':
19.             case 'I': range = 100;
20.                 break;
21.             case 'a':
22.             case 'A': range = 1000;
23.                 break;
24.         }
25.         Random r = new Random();
26.         answer = r.nextInt(range) + 1;
27.         //System.out.println(answer);
28.     }
29.
30.     public void play() {
31.         int guess;
32.         do {
33.             guess = KeyboardReader.getPromptedInt("Enter a number 1 - " + range + ": ");
34.             if (guess < answer) System.out.println("Too low");
35.             else if (guess > answer) System.out.println("Too high");
36.         } while (guess != answer);
37.         System.out.println("Correct!");
38.     }
39.
40.     public static void main(String[] args) {
41.         char playAgain = 'Y';
42.         do {
43.             char level = KeyboardReader.getPromptedChar("What level (B, I, A)? ");
44.             new Game(level).play();
```

Evaluation  
Copy

```
45.     playAgain = KeyboardReader.getPromptedChar("Play again (y/n)?: ");
46.     } while (playAgain == 'Y' || playAgain == 'y');
47.     }
48. }
```

---

## Code Explanation

---

In the `main` method, inside the `do ... while` loop we prompt the user to enter 'y' or 'n'; the loop runs again if the user enters 'y' or 'Y'.

In the `play` method, inside the `do ... while` loop we prompt the user to enter a guess; the loops runs again if the user-entered value is not equal to answer, the randomly-generated number to be guessed.

---



## 4.13. Additional Loop Control: `break` and `continue`

### ❖ 4.13.1. Breaking Out of a Loop

The `break` statement will end a loop early and execution jumps to the first statement following the loop. The following example prints random digits until a random value is less than 0.1.

#### Demo 4.13: Java-Control/Demos/Break.java

---

```
1.  public class Break {
2.      public static void main(String[] args) {
3.          for ( ; ; ) { // creates an infinite loop
4.              // no initialization, no test, no increment
5.              double x = Math.random();
6.              if (x < 0.1) break;
7.              System.out.print((int) (x * 10) + " ");
8.          }
9.          System.out.println("Done!");
10.     }
11. }
```

---

## Code Explanation

---

This code loops, generating and printing a random number for each iteration. (Note that `Math.random()` returns a random number greater than or equal to `0.0` and less than `1.0`.) If the number is less than `0.1`, we break out before printing it.

---

### Demo 4.14: Java-Control/Demos/BreakNot.java

---

```
1. public class BreakNot {
2.     public static void main(String[] args) {
3.         double x = 0.0;
4.         for ( x = Math.random(); x >= 0.1; x = Math.random() ) {
5.             System.out.print((int) (x * 10) + " ");
6.         }
7.         System.out.println("Done!");
8.     }
9. }
```

---

## Code Explanation

---

This code avoids the break, by creating and testing the random number in the control part of the loop. As part of the iteration condition, if the number is less than `0.1`, the loop simply ends “normally”.

---



## 4.14. Continuing a Loop

If you need to stop the current iteration of the loop, but continue the looping process, you can use the `continue` statement. Note that:

- It is normally based on a condition of some sort.
- Execution skips to the end of this pass, but continues looping.
- It is usually a better practice to reverse the logic of the condition, and place the remainder of the loop under control of the `if` statement.

In the following example, we ask the user to enter 10 non-negative numbers, skipping (with `continue`) the processing of each user-entered number if the number supplied is less than `0`:

## Demo 4.15: Java-Control/Demos/Continuer.java

---

```
1.  import util.*;
2.
3.  public class Continuer {
4.      public static void main(String[] args) {
5.          int count = 0;
6.          do {
7.              int num = KeyboardReader.getPromptedInt("Enter an integer: ");
8.              if (num < 0) continue;
9.              count++;
10.             System.out.println("Number " + count + " is " + num);
11.         } while (count < 10);
12.         System.out.println("Thank you");
13.     }
14.
15.     /*
16.     // a better way
17.
18.     public static void main(String[] args) {
19.         int count = 0;
20.         do {
21.             int num = KeyboardReader.getPromptedInt("Enter an integer: ");
22.             if (num >= 0) {
23.                 count++;
24.                 System.out.println("Number " + count + " is " + num);
25.             }
26.         } while (count < 10);
27.         System.out.println("Thank you");
28.     }
29.     */
30.
31. }
```

---

A better way to handle the loop is shown in the commented out version of main - try removing the comment and commenting out the original method.

But, `continue` is easier to use in nested loops, because you can label the level that will be continued.

### break and continue in Nested Loops

In normal usage, `break` and `continue` only affect the current loop; a `break` in a nested loop would break out of the inner loop, not the outer one

But, you can label a loop, and break or continue at that level. A label is a unique identifier followed by a colon character.

Try the following example as is, then reverse the commenting on the break lines

### Demo 4.16: Java-Control/Demos/BreakOuter.java

---

```
1.  public class BreakOuter {
2.      public static void main(String[] args) {
3.
4.          outer: for (int r = 0; r < 10; r++) {
5.              for (int c = 0; c < 20; c++) {
6.                  double x = Math.random();
7.
8.                  //if (x < 0.02) break;
9.                  if (x < 0.02) break outer;
10.                 System.out.print((int) (x * 10) + " ");
11.             }
12.             System.out.println();
13.         }
14.         System.out.println("Done!");
15.     }
16. }
```

---

Evaluation  
Copy

---

## 4.15. Classpath, Code Libraries, and Jar Files

By now, you may have noticed that every demo and solution folder contains its own copy of `util` and `KeyboardReader`. Not only is it inefficient, but it means that we would have to locate and update each copy if we wanted to change `KeyboardReader`.

A better solution would be to have one master copy somewhere that all of our exercises could access.

### ❖ 4.15.1. Using CLASSPATH

Java has a `CLASSPATH` concept that enables you to specify multiple locations for `.class` files, at both compile-time and runtime.

By default, Java uses `rt.jar` in the current Java installation's `jre/lib` directory, and assumes that the classpath is the current directory (the *working directory* in an IDE). A *jar*

*file* is a Java archive file and is similar to a zip file. A jar file contains the compressed contents of one or more packages.

Many Integrated Development Environments (IDEs) such as Eclipse and IntelliJ take care of the classpath for the developer. For example, Eclipse provides a “Build Path” property that permits you to specify external jar files (as well as choose from various libraries that contain multiple jar files supplied with the environment).

## ❖ 4.15.2. Creating a jar File

You can bundle one or packages in a *jar* file. You can also add other resources such as image files.

A jar file contains files in a zipped directory structure. The root level of the file structure should match the root of the package structure; for example, to put `KeyboardReader` in a jar file, we would want to change directory in a command prompt to the parent directory of the `util` folder.

Next, we create a jar file called `utilities.jar` for all files in the `util` package (just `KeyboardReader`, in this case).

### **Command to Create a jar File**

```
jar -cvf utilities.jar util\*.class
```

- The options are *create*, *verbose*, and use a list of *files* supplied at the end of the command.

# Exercise 18: Creating and Viewing a jar File

 5 to 10 minutes

---

1. Run the command shown above to create `utilities.jar`.
2. To view the contents of your new jar file run the following command using the *table of contents* option:

### Command to List jar file contents

```
jar -tvf utilities.jar
```



## 4.16. Compiling to a Different Directory

You can supply a destination directory to the Java compiler, in order to keep your source and compiled files separate:

### Compiler -d Option

```
javac -d destination  
      FileList
```

The destination directory can be a relative path from the current directory, or an absolute path. It must already exist, but any subdirectories necessary for package structures will be created.

Also, if you have existing compiled files in the source directories (like `employees/Employee.class`), then the compiler won't recompile those classes (and therefore they won't end up in the destination path) unless they are older than the associated `.java` file. You would need to “clean” the directories by deleting the `.class` files first.

## Conclusion

In this section, we have learned about boolean-valued expressions, and used them to control branches and loops.

We also learned how to create and view jar files.





# LESSON 5

## Arrays

---

### Topics Covered

- ☑ Understanding how arrays are implemented in Java.
- ☑ Declaring, instantiating, populating, and using arrays of primitives.
- ☑ Arrays of objects.
- ☑ Multi-dimensional arrays.

### Introduction

In this lesson, you will learn to declare, instantiate, populate, and use arrays of primitives and objects, and to work with multi-dimensional arrays.

Evaluation  
\*  
Copy

### 5.1. Defining and Declaring Arrays

An array stores a group of data items all of the same type. An array is an object.

- An array variable does not actually store the array - it is a reference variable that points to an array object.
- Declaring the array variable does not create an array object instance; it merely creates the reference variable - the array object must be instantiated separately.
- Once instantiated, the array object contains a block of memory locations for the individual elements.
- If the individual elements are not explicitly initialized, they will be set to zero.
- Arrays can be created with a size that is determined dynamically (but once created, the size is fixed).

Declare an array variable by specifying the type of data to be stored, followed by square brackets [ ].

### Array Declaration Syntax

```
dataType[] variableName;
```

You can read the [] as the word “array”.

To declare a variable for an array of integers:

```
int[] nums;
```

...which you can read as "int array nums".

To declare a variable for an array of String objects:

```
String[] names;
```

...which you can read as "String array names" - the array holds String references.

You may also put the brackets after the variable name (as in C/C++), but that is less clearly related to how Java actually works.

```
int nums[]; // not recommended, but legal
```

But, that syntax does allow the following, which is legal, but seems like a bad practice.

```
int nums[], i, j; // declares one array and two single int values
```



## 5.2. Instantiating Arrays

Instantiate an array object using `new`, the data type, and an array size in square brackets

```
int[] nums;  
nums = new int[10];
```

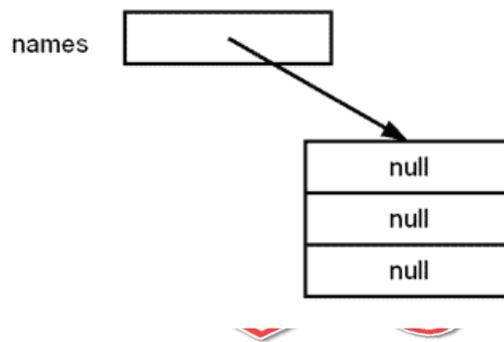
The second line constructs a new array object with 10 integer elements, all initialized to 0, and stores the reference into `nums`.

```
int[] moreNums;  
int size = 7;  
moreNums = new int[size];
```

You can declare and instantiate all at once:

```
String[] names = new String[3];
```

The elements of the array, `String` references, are initialized to `null`.



As objects, arrays also have a useful property: `length`:

- In the above example, `names.length` would be 3.
- The property is fixed (i.e., it is read-only).

You can reassign a new array to an existing variable:

```
int[] nums;  
nums = new int[10];  
nums = new int[20];
```

The original ten-element array is no longer referenced by `nums`, since it now points to the new, larger array.



## 5.3. Initializing Arrays

An array can be initialized when it is created

The notation looks like this:

```
String[] names = { "Joe", "Jane", "Herkimer" };
```

*or*

```
String[] names = new String[] { "Joe", "Jane", "Herkimer" };
```

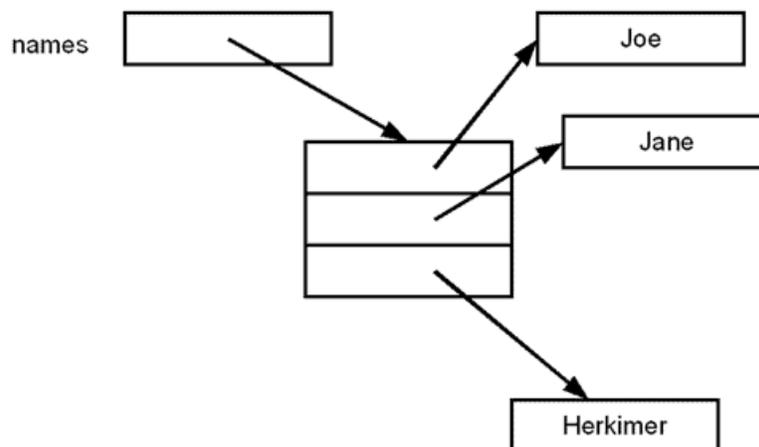
This automatically creates an array of length 3, because there were 3 items supplied.

```
int[] nums = new int[] { 2, 4, 6, 8, 10, 12 };
```

This array will have a length of 6.

If a new array is being assigned to an existing variable, you cannot use the shorter variant, you must use the `new` keyword and the data type:

```
String[] names;  
names = new String[] { "Joe", "Jane", "Herkimer" };
```



For arrays of other types of objects:

```
Book[] titles;
titles = new Book[] {
    new Book(5011, "Fishing Explained"),
    new Book(1234, "Help is on the Way")
};
```

If we were to run the code above, we would end up with an array named "titles" each of whose two elements is an object of type Book.



## 5.4. Working With Arrays

Array elements are accessed through the array reference, by their array index:

- The index is the element number, placed within brackets.
- Elements are numbered from 0 to one less than the specified size.

```
String[] names = new String[3];
```

The valid elements are 0, 1, and 2, as in:

```
names[0] = "Sam";
names[1] = "Sue";
names[2] = "Mary";
```

You could access array elements in a for loop with:

```
for (int i = 0; i < 3; i++) System.out.println(names[i]);
```

Or, better programming practice would be to use the length property:

```
for (int i = 0; i < names.length; i++) System.out.println(names[i]);
```

The compiler does not check to ensure that you stay within the bounds of the array, but the JVM does check at runtime - if you try to exceed the bounds of the array, an exception will occur.

Note that a zero-length array is valid:

```
Book[] titles = new Book[] { };  
Book[] moreTitles = new Book[0];
```

You might create a zero-length array as the return value from a method typed as returning an array, when there are no items to return (as opposed to returning `null`).

### Demo 5.1: Java-Arrays/Demos/Arrays1.java

---

```
1.  import java.util.*;  
2.  public class Arrays1 {  
3.      public static void main(String[] args) {  
4.          Random r = new Random();  
5.          int[] nums = new int[10];  
6.          for (int i = 0; i < nums.length; i++) {  
7.              nums[i] = r.nextInt(100);  
8.          }  
9.          System.out.println("Element 7 is: " + nums[7]);  
10.         String[] names = new String[3];  
11.         names[0] = "Joe";  
12.         names[1] = "Jane";  
13.         names[2] = "Herkimer";  
14.         for (int i = 0; i < names.length; i++) {  
15.             System.out.println(names[i]);  
16.         }  
17.         //this line should throw an exception  
18.         System.out.println(names[6]);  
19.     }  
20. }
```

---

#### Code Explanation

---

In the demo above, line 18 throws an exception: since array `names` has three elements, trying to access `names[6]` gives an error.

---



## 5.5. Enhanced for Loops - the For-Each Loop

In the last lesson we touched briefly on the *for-each* loop, which loops through a collection of values without using an index. Instead, the loop variable represents each individual value.

The syntax uses a loop variable and a collection of values, separated by a colon character (which can be read as the word “from”). Some things to note:

- The collection of values can be any array or an instance of one of the Java *Collections classes* (to be discussed later)
- The looping variable must be declared in the parentheses that create the loop - you cannot use a preexisting variable as the loop variable
- The looping variable will represent each item from the collection or array in turn.

```
for (dataType loopVariable : collectionOfSameType) code using loopVariable;
```

### Warning

You cannot write into the array using a for-each loop. The looping variable you declare receives a copy of the data in the array, so, if you change its value, you are only changing the local copy.

In the following demo, we use a for-each loop to print each element of an array of `ints` to the screen:

## Demo 5.2: Java-Arrays/Demos/ArraysForEach.java

---

```
1. import java.util.*;
2. public class ArraysForEach {
3.     public static void main(String[] args) {
4.         Random r = new Random();
5.         int[] nums = new int[10];
6.         for (int i = 0; i < nums.length; i++) {
7.             nums[i] = r.nextInt(100);
8.         }
9.         for (int num : nums) {
10.            System.out.println(num);
11.        }
12.    }
13. }
```

---

### Code Explanation

**Evaluation  
Copy**

We use a for-each loop to iterate over array `nums`, an array of `int` of length 10. For each iteration of the loop, `num` holds the value of the current element.

---



## 5.6. Array Variables

The array as a whole can be referenced by the array name without the brackets, for example, as a parameter to or return value from a function

## Demo 5.3: Java-Arrays/Demos/Arrays2.java

---

```
1. public class Arrays2 {
2.     public static void main(String[] args) {
3.         String[] names = new String[3];
4.         names[0] = "Joe";
5.         names[1] = "Jane";
6.         names[2] = "Herkimer";
7.         printArray(names);
8.     }
9.     public static void printArray(String[] data) {
10.        for (int i = 0; i < data.length; i++) {
11.            System.out.println(data[i].toUpperCase());
12.        }
13.    }
14. }
```

---

### Code Explanation

---

The array names is passed to printArray, where it is received as data.

Note also the syntax to access a method directly for an array element:  
data[i].toUpperCase()

---

Since an array reference is a variable, it can be made to refer to a different array at some point in time

```
String[] names = new String[3];
names[0] = "Joe";
names[1] = "Jane";
names[2] = "Herkimer";
printArray(names);
names = new String[2];
names[0] = "Rudolf";
names[1] = "Ingrid";
printArray(names);
```



## 5.7. Copying Arrays

You can use `System.arraycopy` to copy an array into another.

You might do this to expand an array by creating a larger one and copying the contents of the smaller one into it (but any references to the original array will need to be changed to point to the new array).

The declaration is:

### **System.arraycopy Usage**

```
public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
```

### **Demo 5.4: Java-Arrays/Demos/CopyArray.java**

---

```
1. // puts two copies of a small array into a larger one
2. public class CopyArray {
3.     public static void main(String[] args) {
4.         int nums[] = { 6, 2, 4, 9, 1 };
5.         int biggerNums[] = new int[10];
6.         System.arraycopy(nums, 0, biggerNums, 0, nums.length);
7.         System.arraycopy(nums, 0, biggerNums,
8.             nums.length, nums.length);
9.         for (int i = 0; i < biggerNums.length; i++)
10.            System.out.println(biggerNums[i]);
11.     }
12. }
```

---

### Code Explanation

---

1. The first `arraycopy` line copies from `nums` into the first half of `biggerNums`. The number of items copied is determined by `nums.length`

```
System.arraycopy(nums, 0, biggerNums, 0, nums.length);
```

2. The second `arraycopy` line again copies from `nums`, but now into the second half of `biggerNums`. The starting location for the “paste” is `nums.length`, since that is where we left off the first time. The number of items copied is again determined by `nums.length`

```
System.arraycopy(nums, 5, biggerNums, nums.length, nums.length);
```

---

In the following exercise you'll write code to loop over any arguments supplied when running a compiled program.

Evaluated  
copy

## Exercise 19: Using the args Array

 5 to 10 minutes

---

1. Note that there has been an array declaration in all of our examples thus far - the array of strings passed to main: `args`
  - These are the additional words, if any, on the command line when running the `java` command.
2. Open `Java-Arrays/Exercises/Args/HelloArgs.java`.
3. Modify the class to loop through `args` to print each element.
4. Run the program with something like `java HelloArgs From Me` and see what happens.



## Solution: Java-Arrays/Solutions/Args/HelloArgs.java

---

```
1. public class HelloArgs {
2.     public static void main(String[] args) {
3.
4.         // using indexed loop
5.         for (int i = 0; i < args.length; i++)
6.             System.out.println(args[i]);
7.
8.         // using for-each loop
9.         for (String a : args)
10.            System.out.println(a);
11.     }
12. }
```

---

### Code Explanation

---

The `for` loop iterates through the array; each separate word on the command line becomes one element in the array (note that `java HelloArgs` is *not* part of the array). We also provide a solution using a `for-each` loop.

If no arguments were supplied from the command line - that is, if you were to run the program with the command `java HelloArgs` - then the code would still work without error; the loops would simply not execute (i.e. would execute 0 times.) In this case, the program would run successfully but print nothing to the screen.

---

# Exercise 20: Game-Arrays01: A Guessing Game with Random Messages

🕒 15 to 20 minutes

---

In this exercise, you will update the guessing-game program by using arrays to store a variety of messages to be printed to the user.

1. Modify your guessing game program to hold an array of several different String messages, all of which have some form of message for “Correct”.
2. Generate a random number in the range from 0 to the size of the array.
3. Use this value to select one message to print when they guess correctly.
4. Continue this approach for “Too Low” and “Too High”.

## Solution: Java-Arrays/Solutions/Game-Arrays01/Game.java

---

```
1.  import util.*;
2.  import java.util.*;
3.
4.  public class Game {
5.      private Random r = new Random();
6.      private int answer;
7.      private static String [] correctMessages =
8.          { "Correct", "Right on!", "Most Excellent!" };
9.      private static String [] lowMessages =
10.         { "Too low", "Try higher next time" };
11.     private static String [] highMessages =
12.         { "Too high", "Try lower next time" };
13.     private int range = 10;
14.
15.     public Game(char level) {
16.         switch (level) {
17.             default:
18.                 System.out.println("Invalid option: " +
19.                     level + ", using Beginner");
20.             case 'b':
21.             case 'B':
22.                 range = 10;
23.                 break;
24.             case 'i':
25.             case 'I': range = 100;
26.                 break;
27.             case 'a':
28.             case 'A': range = 1000;
29.                 break;
30.         }
31.         Random r = new Random();
32.         answer = r.nextInt(range) + 1;
33.         //System.out.println(answer);
34.     }
35.
36.     public void play() {
37.         int guess;
38.         do {
39.             guess = KeyboardReader.getPromptedInt("Enter a number 1 - " + range + ": ");
40.             if (guess < answer)
41.                 System.out.println(lowMessages[r.nextInt(lowMessages.length)]);
42.             else if (guess > answer)
43.                 System.out.println(highMessages[r.nextInt(highMessages.length)]);
44.         } while (guess != answer);
```

Evaluation  
Copy

```
45.     System.out.println(correctMessages[r.nextInt(correctMessages.length)]);
46. }
47.
48. public static void main(String[] args) {
49.     char playAgain = 'Y';
50.     do {
51.         char level = KeyboardReader.getPromptedChar("What level (B, I, A)? ");
52.         new Game(level).play();
53.         playAgain = KeyboardReader.getPromptedChar("Play again (y/n)?: ");
54.     } while (playAgain == 'Y' || playAgain == 'y');
55. }
56. }
```

---

### Code Explanation

---

There are three arrays of messages; note that they are not all the same size. The code to handle too high, too low, and correct generates a random number between 0 and the appropriate array's length, by using the Random object's `nextInt` method, and passing the length of the array.

---



## 5.8. Arrays of Objects

If an array contains objects, those objects' properties and methods may be accessed.

The notation uses the array variable name, the index in brackets, a dot, and the property or method. The following demo gives a quick example of this:

## Demo 5.5: Java-Arrays/Demos/Arrays2.java

---

```
1.  public class Arrays2 {
2.      public static void main(String[] args) {
3.          String[] names = new String[3];
4.          names[0] = "Joe";
5.          names[1] = "Jane";
6.          names[2] = "Herkimer";
7.          printArray(names);
8.      }
9.      public static void printArray(String[] data) {
10.         for (int i = 0; i < data.length; i++) {
11.             System.out.println(data[i].toUpperCase());
12.         }
13.     }
14. }
```

---

### Code Explanation

---

Since `names` is an array of `Strings` and since `toUpperCase()` is a valid `String` method, we can use `data[i].toUpperCase()` to return the all-upper-case version of the `i`th element (the `i`th `String`) in `names`.

---

## Exercise 21: Payroll-Arrays01: An Array of employees

 20 to 30 minutes

---

1. Modify the payroll program to use an array of Employee objects with a size of 3 or more (later we will come up with a more flexible solution that allows for the number of employees to change dynamically).
2. Use a for loop to populate and display the data.
3. After the loop is complete, prompt the user to enter a last name.
4. Loop through the array to find the first element with the matching last name and display it; print a “not found” message if there is no matching last name.

## Solution: Java-Arrays/Solutions/Payroll-Arrays01/Payroll.java

---

```
1.  import employees.*;
2.  import util.*;
3.
4.  public class Payroll {
5.      public static void main(String[] args) {
6.          Employee[] e = new Employee[5];
7.          String fName = null;
8.          String lName = null;
9.          int dept = 0;
10.         double payRate = 0.0;
11.
12.         for (int i = 0; i < e.length; i++) {
13.             fName = KeyboardReader.getPromptedString("Enter first name: ");
14.             lName = KeyboardReader.getPromptedString("Enter last name: ");
15.             do {
16.                 dept = KeyboardReader.getPromptedInt("Enter department: ");
17.                 if (dept <= 0) System.out.println("Department must be >= 0");
18.             } while (dept <= 0);
19.             do {
20.                 payRate = KeyboardReader.getPromptedDouble("Enter pay rate: ");
21.                 if (payRate < 0.0) System.out.println("Pay rate must be >= 0");
22.             } while (payRate < 0.0);
23.             e[i] = new Employee(fName, lName, dept, payRate);
24.             System.out.println(e[i].getPayInfo());
25.         }
26.
27.         lName = KeyboardReader.getPromptedString("Enter last name to find: ");
28.         boolean notFound = true;
29.         for (int i = 0; i < e.length; i++) {
30.             if (lName.equalsIgnoreCase(e[i].getLastName())) {
31.                 System.out.println("Found: " + e[i].getPayInfo());
32.                 notFound = false;
33.                 break;
34.             }
35.         }
36.         if (notFound) System.out.println("Not found");
37.     }
38. }
```

---

## Code Explanation

---

The code uses `e[i].getPayInfo()` to print the pay information for employee `i`. Element `i` of the array is one employee reference. It uses the same approach to call `e[i].getLastName()` for each employee to compare with the requested name.

---



## 5.9. Multi-Dimensional Arrays

Arrays may have more than one dimension, for such things as:

- A graphic image that is `x` pixels across and `y` pixels vertically.
- Weather information modeled in a 3-dimensional space, with measurements for these axes: North/South, East/West, and altitude.

Declare a multidimensional array as:

```
datatype[][] ... [] arrayName;
```

*Evaluation Copy*

Arrays are objects, and, like other objects, declaring a variable does not instantiate the array - that must be done separately. To instantiate a multidimensional array:

```
arrayName = new datatype[size1][size2] ... [sizeN];
```

The most significant dimension is listed first; the least significant dimension listed last.

The code below could be used to declare an array to store an image that is 640 pixels across and 480 pixels down - in a graphic the image data is stored sequentially across each row; each row is a 640 pixel block; there are 480 of these blocks in our image:

```
int[][] picture = new int[480][640];
```

This code might be used for an image where the data is stored in three layers, each of which is an entire 480 by 640 array:

```
int[][][] picture = new int[3][480][640];
```



## 5.10. Multidimensional Arrays in Memory

Recall that a matched pair of brackets after a data type means:

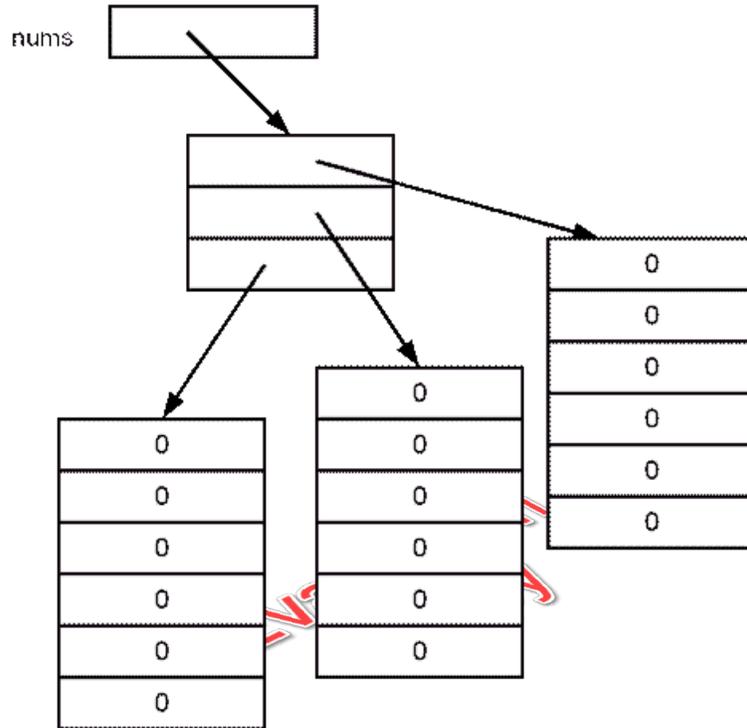
- The array contains elements of that type.
- The array itself is a type of data.

In Java, a two-dimensional array is actually a single array of array reference variables, each of which points to a single dimensional array.

Consider the following example:

```
int[][] nums = new int[3][6];
```

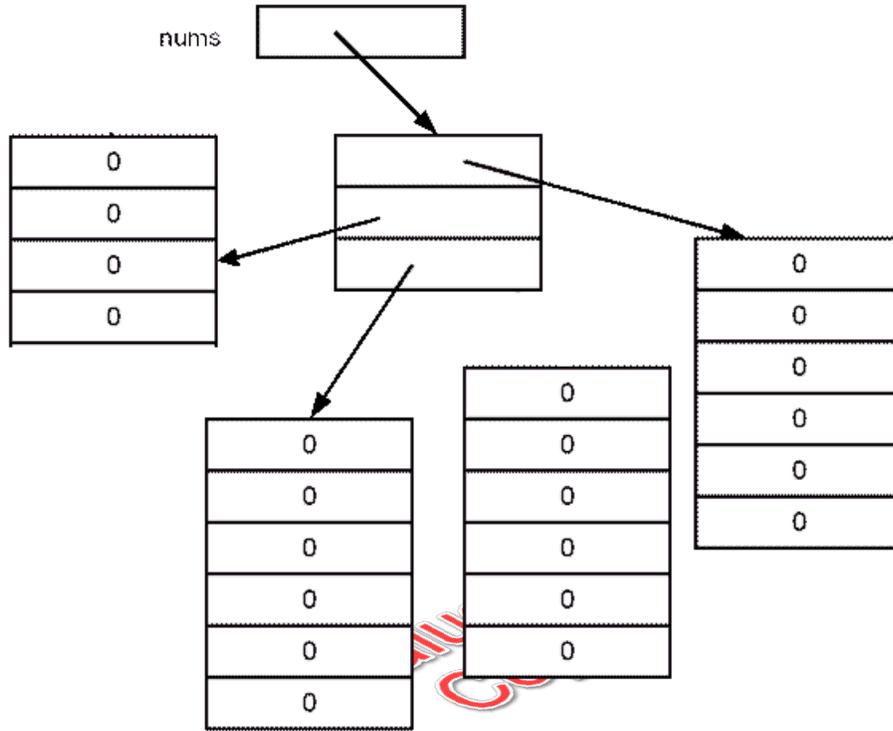
This is an array of 3 elements, each of which is an array of 6 int elements as shown in the diagram below:



### Note

It is possible to replace any of the one-dimensional elements with a different one, or that the second-dimension arrays each have a different length - the following line would replace one of the arrays with another of a different length:

```
nums[1] = new int[4];
```

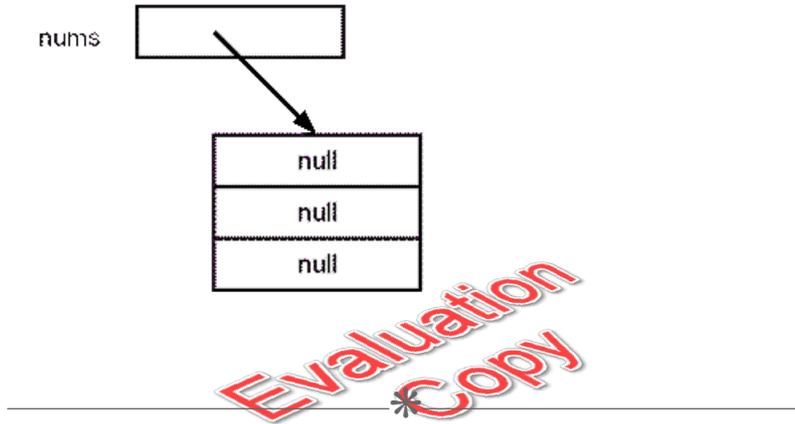


## 5.11. Example - Printing a Picture

This example uses a two-dimensional array preloaded with text characters that make up a picture

- There is a loop that processes each row (the first, or most significant, dimension of the array, each element of which is an array of characters).
- Within that loop, another loop prints each character without ending the line.
- Then, when the inner loop is done, a newline is printed.





## 5.12. Typecasting with Arrays of Primitives

It is not possible to typecast an array of one type of primitive to an array of another type of primitive. For example, the following will cause compiler errors if the comment marks are removed:

## Demo 5.7: Java-Arrays/Demos/ArrayTypecast.java

---

```
1. // typecasts with arrays
2.
3. public class ArrayTypecast {
4.
5.     public static void main(String[] args) {
6.
7.         int i = 5;
8.         double d;
9.
10.        d = i;
11.        i = (int) d;
12.
13.        int inums[] = { 1, 2, 3, 4, 5 };
14.        double[] dnums;
15.
16.        // line below fails
17.        //dnums = inums;
18.
19.        dnums = new double[] { 1.1, 2.2, 3.3, 4.4 };
20.
21.        // line below fails
22.        //inums = (int[]) dnums;
23.
24.    }
25. }
```



---

### Code Explanation

Neither an implicit or explicit typecast can be performed. With a single `int i`, the *copy* of it that is given to `d` can be expanded to a `double`. But, with the `int[] inums`, the value that would be given to `dnums` is just a *copy of the reference* to `inums`, so there is no way that each of the individual elements can be expanded to `double`.

The next chapter will discuss typecasting from arrays of one type of object to another.

---

## Conclusion

In this lesson you have learned how to declare, instantiate and work with arrays of primitives and objects.



# LESSON 6

## Inheritance

---

### Topics Covered

- ☑ Object-oriented programming (OOP).
- ☑ The role of inheritance in a program.
- ☑ Declaring and extending Java classes.
- ☑ Polymorphism.

### Introduction

In this lesson, you will learn about the OOP concept of inheritance, to examine the role of inheritance in a program, to declare and use Java classes that extend existing classes, and about the concept and role of polymorphism in Java.



## 6.1. Inheritance

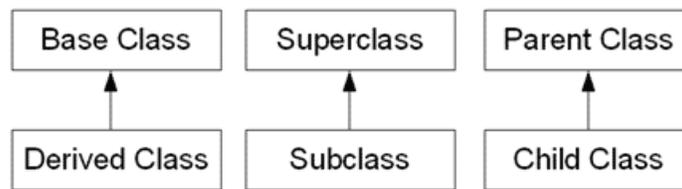
*Inheritance* creates a new class definition by building upon an existing definition (you *extend* the original class).

The new class can, in turn, can serve as the basis for another class definition.

- All Java objects use inheritance.
- Every Java object can trace back up the inheritance tree to the generic class `Object`.

The keyword `extends` is used to base a new class upon an existing class

Several pairs of terms are used to discuss class relationships (these are not keywords).



- Note that traditionally the arrows point from the inheriting class to the base class, and the base class is drawn at the top - in the *Unified Modeling Language (UML)* the arrows point from a class to another class that it depends upon (and the derived class depends upon the base class for its inherited code).

A derived class instance may be used in any place a base class instance would work - as a variable, a return value, or parameter to a method.

Inheritance is used for a number of reasons (some of the following overlap):

- To model real-world hierarchies.
- To have a set of pluggable items with the same “look and feel,” but different internal workings.
- To allow customization of a basic set of features.
- When a class has been distributed and enhancements would change the way existing methods work (breaking existing code using the class).
- To provide a “common point of maintenance.”

When extending a class, you can add new fields and methods and you can change the behavior of existing methods (which is called *overriding* the methods).

- You can declare a method with the same signature and write new code for it.
- You can declare a field again, but this does not replace the original field; it *shadows* it (the original field exists, but any use of that name in this class and its descendants refers to the memory location of the newly declared element).



## 6.2. Inheritance Examples

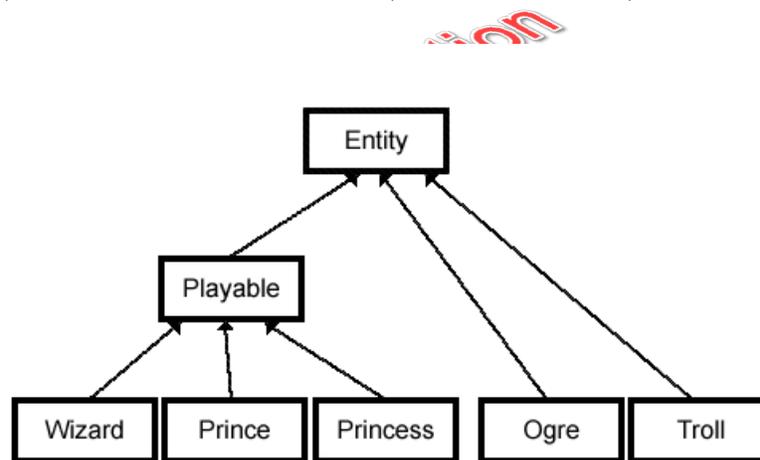
Say you were creating an arcade game, with a number of different types of beings that might appear - wizards, trolls, ogres, princesses (or princes), frogs, etc. All of these entities

would have some things in common, such as a name, movement, ability to add/subtract from the player's energy - this could be coded in a base class `Entity`.

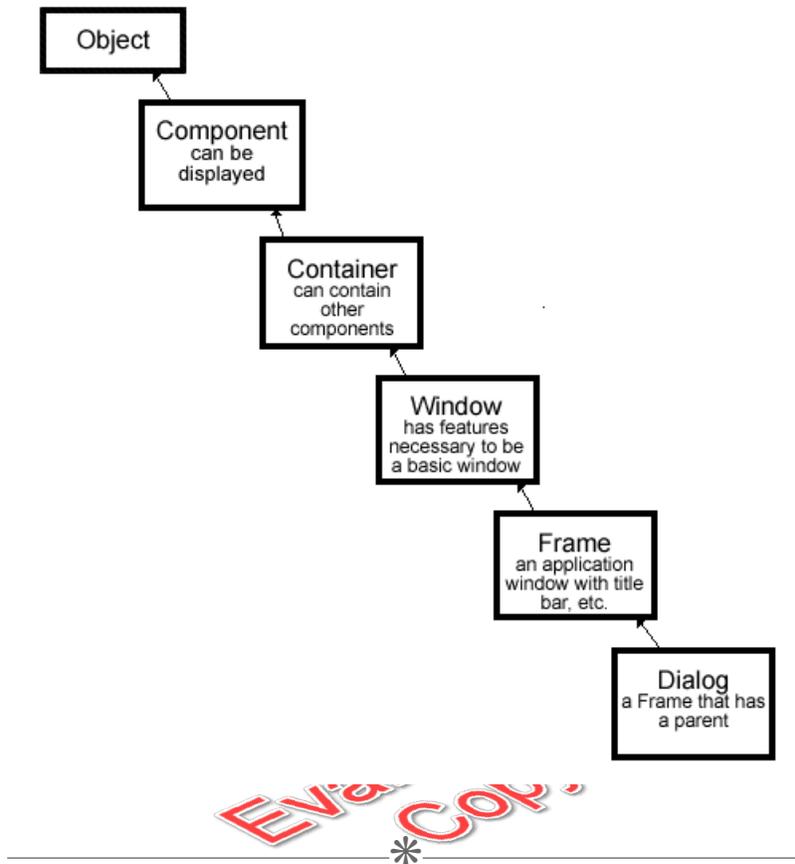
For entities that can be chosen and controlled by a player (as opposed to those that merely appear during the course of the game but can't be chosen as a character) a new class `Playable` could extend `Entity` by adding the control features.

Then, each individual type of entity would have its own special characteristics; those that can be played would extend `Playable`, the rest would simply extend `Entity`.

You could then create an array that stored `Entity` objects, and fill it with randomly created objects of the specific classes. For example, your code could generate a random number between 0 and 1; if it is between 0.0 and 0.2, create a Wizard, 0.2 - 0.4 a Prince, etc.



The Java API is a set of classes that make extensive use of inheritance. One of the classes used in the GUI is `Window`; its family tree looks like this:

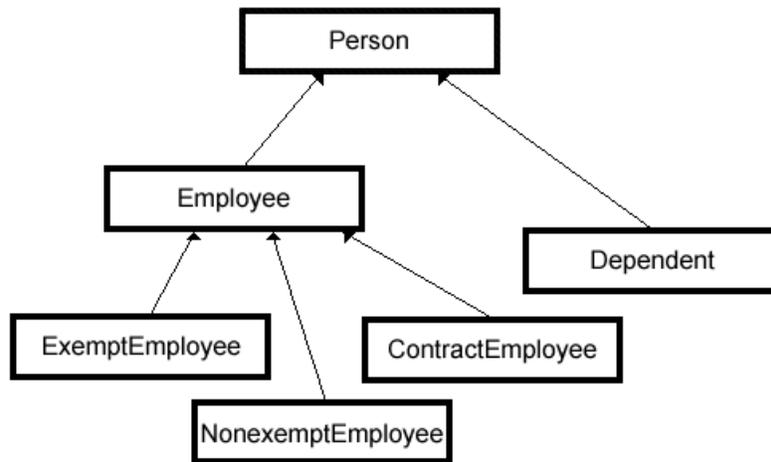


### 6.3. Payroll with Inheritance

Our payroll program could make use of inheritance if we had different classes of employees: exempt employees, nonexempt employees, and contract employees:

- They all share basic characteristics such as getting paid (albeit via different algorithms), withholding, and having to accumulate year-to-date numbers for numerous categories.
- But they have different handling regarding payment calculations, benefits, dependents, etc.
- Exempt employees get a monthly salary, while nonexempt get a wage \* hours; contract employees are handled similarly to nonexempt, but cannot have benefits or dependents.

This would leave us with an inheritance scheme as follows:



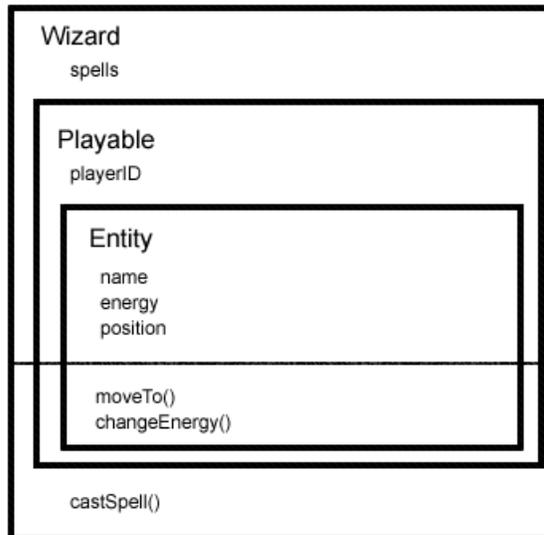
Note that a scheme with `ContractEmployee` extending `NonexemptEmployee` might also be a reasonable approach



## 6.4. Derived Class Objects

You can view a derived class object as having a complete base class object inside it. Let's assume that the `Entity` class defines the fields `name`, `energy`, and `position`, and methods `moveTo()` and `changeEnergy()`.

The `Playable` class adds a field `playerID`, and the `Wizard` class adds a `spells` field (an array of spells they can cast) and a `castSpell()` method.



Any Wizard object contains all the elements inside its box, include those of the base classes. So, for example, the complete set of properties in a Wizard object is:

- name
- energy
- position
- playerID
- spells

A Wizard reference to a Wizard object has access to any public elements from any class in the inheritance chain from Object to Wizard. Code inside the Wizard class has access to all elements of the base classes (except those defined as private in the base class - those are present, but not directly accessible).

A more complete description of access levels is coming up.

Note: although it appears that a base class object is physically located inside the derived class instance, it is not actually implemented that way.



## 6.5. Polymorphism

### ❖ 6.5.1. Inheritance and References

If a derived class extends a base class, it is not only considered an instance of the derived class, but an instance of the base class as well. The compiler knows that all the features of the base class were inherited, so they are still there to work in the derived class (keeping in mind that they may have been changed).

This demonstrates what is known as an "is a" relationship - a derived class object *is a* base class instance as well.

This is an example of *polymorphism*: one reference can store several different types of objects. For example, in the arcade game example, for any character that is used in the game, an Entity reference variable could be used, so that at runtime, any subclass can be instantiated to store in that variable.

- Entity shrek = new Ogre();  
Entity merlin = new Wizard();

- For the player's character, a Playable variable could be used.

```
Playable charles = new Prince();
```

When this is done, however, the only elements immediately available through the reference are those known to exist; that is, those elements defined in the *reference type* object. Note that:

- The compiler decides what to allow you to do with the variable based upon the *type declared for the variable*.
- `merlin.moveTo()` would be legal, since that element is guaranteed to be there.
- `merlin.castSpell()` would not be legal, since the definition of Entity does not include it, even though the actual object does have that capability.

- The following example gives a hint as to why this is the case:

```
Entity x;  
if (Math.random() < 0.5)  
    x = new Wizard();  
else  
    x = new Troll();
```

There is no way the compiler could determine what type of object would actually be created.

- The variables names above, shrek, merlin, and charles, are probably not good choices: presumably we know shrek is an ogre, and always will be, so the type might as well be `Ogre` (unless, of course, he could transmogrify into something else during the game!).

## ❖ 6.5.2. Dynamic Method Invocation

When a method is called through a reference, the JVM looks to the actual class of the instance to find the method. If it doesn't find it there, it backs up to the ancestor class (the class this class extended) and looks there (and if it doesn't find it there, it backs up again, potentially all the way to `Object`).

Sooner or later, it will find the method, since if it wasn't defined somewhere in the chain of inheritance, the compiler would not have allowed the class to compile.

In this manner, what you could consider the most advanced (or most derived) version of the method will run, even if you had a base class reference.

So, for our arcade game, an `Entity` reference could hold a `Wizard`, and when the `moveTo` method is called, the `Wizard` version of `moveTo` will run.

An interesting aspect of dynamic method invocation is that it occurs even if the method is called from base class code. If, for example:

- The `Entity` class `moveTo` method called its own `toString` method.
- The `Ogre` class didn't override `moveTo`, but did override `toString`.
- For an `Ogre` stored in an `Entity` variable, the `moveTo` method was called.

The Entity version of `moveTo` would run, but its call to `toString` would invoke the `toString` method from `Ogre!`



## 6.6. Creating a Derived Class

The syntax for extending a base class to create a new class is:

```
[modifiers] class DerivedClassName extends BaseClassName {  
    (new or revised field and method definitions go here)  
}
```

If you do not specify a class to extend, Java assumes that you are extending `Object` by default.

Your new class can use the fields and methods contained in the original class (subject to the note coming up in a few pages about access keywords), add new data fields and methods, or replace fields or methods.

A derived class object may be stored in a base class reference variable without any special treatment. If you then want to store that object in a derived class reference again, you can force that with a typecast.

Java doesn't allow *multiple inheritance*, where one class inherits from two or more classes. Note that:

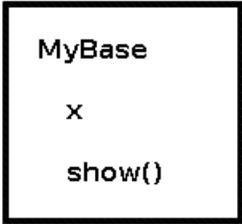
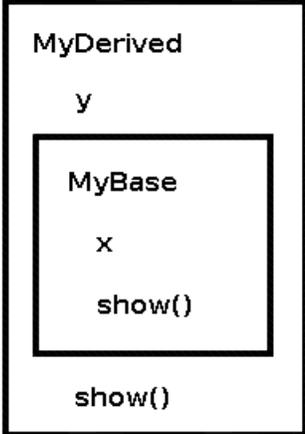
- It does have a concept called an *interface*, which defines a set of method names.
- A class may implement an interface, defining those methods in addition to whatever other methods are in the class.
- This allows for several otherwise unrelated classes to have the same set of method names available, and to be treated as the same type of object for that limited set of methods. We'll cover interfaces later in this course.



## 6.7. Inheritance Example - A Derived Class

When a derived class is created, an object of the new class will in effect contain all the members of the base and derived classes. In some cases, the accessibility modifiers can limit the availability of base class members in the derived class, but we will cover that issue later.

The following maps out the relation between the derived class and the base class (note that the diagrams show the apparent memory allocation, in a real implementation the base class memory block is not inside the derived class block).

| Class Code                                                                                                                                                                     | Apparent Memory Allocation                                                                                                                                                                                                                                                                                                                                                                                                       |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>public class MyBase {     public int x;     public void show() {         System.out.println("x = " + x);     } }</pre>                                                    |  <p>A rectangular box representing the memory allocation for the MyBase class. Inside the box, the text 'MyBase' is at the top, followed by 'x' and 'show()' below it.</p>                                                                                                                                                                      |
| <pre>class MyDerived extends MyBase {     public int y;     public void show() {         System.out.println("x = " + x);         System.out.println("y = " + y);     } }</pre> |  <p>A rectangular box representing the memory allocation for the MyDerived class. Inside the box, the text 'MyDerived' is at the top, followed by 'y'. Below 'y' is a smaller rectangular box representing the inherited MyBase class, containing 'MyBase', 'x', and 'show()'. Below this inner box, the text 'show()' is written again.</p> |

Since everything in MyBase is public, code in the MyDerived class has free access to the x value from the MyBase object inside it, as well as y and show() from itself.

The `show()` method from `MyBase` is also available to code within `MyDerived`, but some work is required to get it, since it is hidden by the `show()` method added within `MyDerived`.



## 6.8. Inheritance and Access

When inheritance is used to create a new (derived) class from an existing (base) class, everything in the base class is also in the derived class. It may not be accessible; however, the access in the derived class depends on the access in the base class:

| Base class access            | Accessibility in derived class |
|------------------------------|--------------------------------|
| <code>public</code>          | <code>public</code>            |
| <code>protected</code>       | <code>protected</code>         |
| <code>private</code>         | Inaccessible                   |
| Unspecified (package access) | Unspecified (package access)   |

Note that `private` elements become inaccessible to the derived class - this does not mean that they disappear, or that there is no way to affect their values, just that they can't be referenced by name in code within the derived class

Also note that a class can extend a class from a different package



## 6.9. Inheritance and Constructors - the `super` Keyword

Since a derived class object contains the elements of a base class object, it is reasonable to want to use the base class constructor as part of the process of constructing a derived class object.

Constructors are “not inherited”. In a sense, this is a moot point, since they would have a different name in the new class, and can't be called by name under any circumstances, so, for example, when one calls `new Integer(int i)` they shouldn't expect a constructor named `Object(int i)` to run.

Within a derived class constructor, however, you can use `super( parameterList )` to call a base class constructor. Note that:

- It must be done as the first line of a constructor.
- Therefore, you can't use both `this()` and `super()` in the same constructor function.
- If you do not explicitly invoke a form of super-`constructor`, then `super()` (the form that takes no parameters) will run.
- For the superclass, its constructor will either explicitly or implicitly run a constructor for its superclass.
- So, when an instance is created, *one constructor will run at every level of the inheritance chain*, all the way from `Object` up to the current class.

## Demo 6.1: Java-Inheritance/Demos/Inheritance1.java

---

```
1.  class MyBase {
2.      private int x;
3.      public MyBase(int x) {
4.          this.x = x;
5.      }
6.      public int getX() {
7.          return x;
8.      }
9.      public void show() {
10.         System.out.println("x=" + x);
11.     }
12. }
13.
14. class MyDerived extends MyBase {
15.     private int y;
16.     public MyDerived(int x) {
17.         super(x);
18.     }
19.     public MyDerived(int x, int y) {
20.         super(x);
21.         this.y = y;
22.     }
23.     public int getY() {
24.         return y;
25.     }
26.     public void show() {
27.         System.out.println("x = " + getX());
28.         System.out.println("y = " + y);
29.     }
30. }
31.
32. public class Inheritance1 {
33.     public static void main(String[] args) {
34.         MyBase b = new MyBase(2);
35.         b.show();
36.         MyDerived d = new MyDerived(3, 4);
37.         d.show();
38.     }
39. }
```

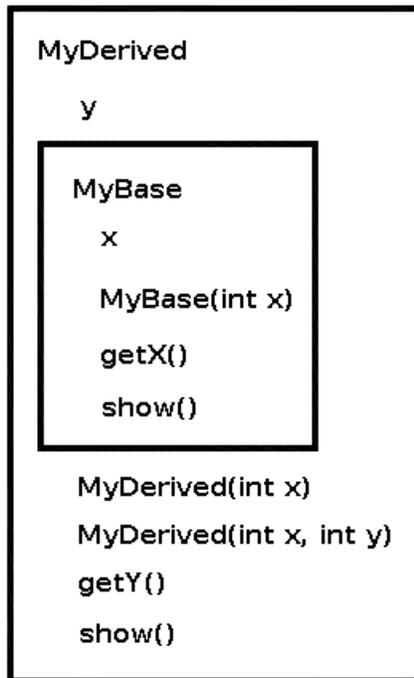
Evaluation  
Copy

---

### Code Explanation

---

The diagram below shows the structure of our improved classes:



- A MyDerived object has two constructors available, as well as both the getX and getY methods and the show method.
- Both MyDerived constructors call the super constructor to handle storage of x.
- The show method in the derived class overrides the base class version.
- Note that you can tell which class's version of show() is running by the different spacing around the = sign.
- x from the base class is not available in the derived class, since it is private in MyBase, so the show method in MyDerived must call getX() to obtain the value.



## 6.10. Example - Factoring Person Out of Employee

Since personnel and probably other people in our overall corporate software suite (contact management, perhaps) would have some basic personal attributes in common, we will pull the first and last name fields into a base class representing a person.

1. Create a new class, `Person`, in the `employees` package (that may not be the best place for it, but for convenience we will put it there).
2. Cut the first and last name fields and the associated get and set methods out of `Employee` and paste them here (including `getFullName`).
3. Create a constructor for `Person` that sets the first and last names; it would probably be a good idea to create a default constructor as well.
4. Declare `Employee` to extend `Person`; change the constructor that accepts the first and last name fields to call a super-constructor to accomplish that task.
5. Note that if `getPayInfo()` uses `firstName` and `lastName` directly, you will need to revise it to call the `get` methods for those values - why?
6. We can test the code using a simplified version of `Payroll`.

## Demo 6.2: Java-Inheritance/Demos/employees/Person.java

---

```
1. package employees;
2.
3. public class Person {
4.     private String firstName;
5.     private String lastName;
6.
7.     public Person() {
8.     }
9.
10.    public Person(String firstName, String lastName) {
11.        setFirstName(firstName);
12.        setLastName(lastName);
13.    }
14.
15.    public String getFirstName() { return firstName; }
16.
17.    public void setFirstName(String firstName) {
18.        this.firstName = firstName;
19.    }
20.
21.    public String getLastName() { return lastName; }
22.
23.    public void setLastName(String lastName) {
24.        this.lastName = lastName;
25.    }
26.
27.    public String getFullName() {
28.        return firstName + " " + lastName;
29.    }
30.
31. }
```



---

### Code Explanation

This class includes the name fields and related set and get methods.

---

## Demo 6.3: Java-Inheritance/Demos/employees/Employee.java

---

```
1.  ="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="https://www.we  ←←
    bucator.com/Schemas/Courseware https://www.webucator.com/assets/wcwc/Au  ←←
    thoring/ExternalFile.xsd"><![CDATA[package employees;
2.
3.  public class Employee extends Person {
4.      private static int nextId = 1;
5.      private int id = nextId++;
6.      private int dept;
7.      private double payRate;
8.
9.      public Employee() {
10.     }
11.     public Employee(String firstName, String lastName) {
12.         super(firstName, lastName);
13.     }
14.     public Employee(String firstName,String lastName, int dept) {
15.         super(firstName, lastName);
16.         setDept(dept);
17.     }
18.     public Employee(String firstName, String lastName, double payRate) {
19.         super(firstName, lastName);
20.         setPayRate(payRate);
21.     }
22.     public Employee(
23.         String firstName, String lastName, int dept, double payRate) {
24.         this(firstName, lastName, dept);
25.         setPayRate(payRate);
26.     }
27.
28.     public static int getNextId() {
29.         return nextId;
30.     }
31.
32.     public static void setNextId(int nextId) {
33.         Employee.nextId = nextId;
34.     }
35.
36.     public int getId() { return id; }
37.
38.     public int getDept() { return dept; }
39.
40.     public void setDept(int dept) {
41.         this.dept = dept;
42.     }
```

Evaluation  
Copy

```
43.
44. public double getPayRate() { return payRate; }
45.
46. public void setPayRate(double payRate) {
47.     this.payRate = payRate;
48. }
49.
50. public String getPayInfo() {
51.     return "Employee " + id + " dept " + dept + " " +
52.         getFullName() +
53.         " paid " + payRate;
54. }
55. }
```

---

Evaluation  
Copy

---

### Code Explanation

---

Since this class now extends `Person`, the name-related elements are already present, so we remove them from this code. We took advantage of the `Person` constructor that accepts first and last names in the corresponding `Employee` constructor.

Note that since `getPayInfo` calls `getFullName`, which is now inherited and publicly accessible, that code did not need to change.

---

## Demo 6.4: Java-Inheritance/Demos/Payroll.java

---

```
1.  import employees.*;
2.  import util.*;
3.
4.  public class Payroll {
5.      public static void main(String[] args) {
6.          String fName = null;
7.          String lName = null;
8.          int dept = 0;
9.          double payRate = 0.0;
10.         double hours = 0.0;
11.
12.         Employee e = null;
13.
14.         fName = KeyboardReader.getPromptedString("Enter first name: ");
15.         lName = KeyboardReader.getPromptedString("Enter last name: ");
16.         dept = KeyboardReader.getPromptedInt("Enter department: ");
17.         do {
18.             payRate = KeyboardReader.getPromptedFloat("Enter pay rate: ");
19.             if (payRate < 0.0) System.out.println("Pay rate must be >= 0");
20.         } while (payRate < 0.0);
21.         e = new Employee(fName, lName, dept, payRate);
22.         System.out.println(e.getPayInfo());
23.     }
24. }
```

---

### Code Explanation

---

No changes need to be made to `Payroll` to take advantage of the addition of the inheritance hierarchy that we added. The only changes we made were for the sake of brevity.

To revisit the sequence of events when instantiating an `Employee` using the constructor that accepts the first and last names, department, and pay rate:

1. Memory for an `Object` is allocated.
2. Any `Object` initializers would run.
3. The `Object()` constructor would run.
4. Memory for a `Person` is allocated.
5. If there were any `Person` initializers, they would run.

6. The Person constructor would run, because that was the version selected by the Employee constructor we called.
  7. Memory for an Employee is allocated.
  8. If there were any Employee initializers, they would run.
  9. Any additional steps in the Employee constructor we called would run.
-

# Exercise 22: Payroll-Inheritance01: Adding Types of Employees

🕒 30 to 45 minutes

---

We wish to improve our payroll system to take account of the three different types of employees we actually have: exempt, nonexempt, and contract employees. Rather than use some sort of identifying code as a property, OOP makes use of inheritance to handle this need, since at runtime a type can be programmatically identified. So we will create three new classes that extend `Employee`:

1. `ExemptEmployee`
2. `NonexemptEmployee`
3. `ContractEmployee`

In our company, exempt employees get a monthly salary, nonexempt an hourly rate that is multiplied by their hours, as do contract employees. There won't be any real difference between nonexempt and contract employees within their code. Realistically, in a real application, there would be, but, even if there weren't, it would still be a good idea to have separate classes, since the class type itself becomes a bit of information about the object. In the exception class hierarchy, there are many classes that have no internal differences; The multiple classes serve merely to identify the type of problem that occurred.

We can inherit much of the logic from `Employee`, such as the pay rate fields and methods, as well as the name-related elements indirectly gained from `Person`. But, `ContractEmployee` and `NonexemptEmployee` will both add logic related to hours, and all three classes will override the `getPayInfo` method.

Also, the solution code builds upon the `Person` base class from the preceding example. You can use the `Person.java` file from the example and edit `Employee.java` to match.

1. Open the files in `Java-Inheritance/Exercises/Payroll-Inheritance01/`.
2. Copy the `Person` class from our earlier example, and edit `Employee` to inherit from `Person`.
3. Create three new classes, `ExemptEmployee`, `NonexemptEmployee`, and `ContractEmployee` that extend `Employee`.

- One possible approach is to copy the `Employee.java` file into a new file, `ExemptEmployee.java`, and modify as appropriate.
  - Then copy that into `NonexemptEmployee.java` and edit.
  - Then copy `NonexemptEmployee.java` into `ContractEmployee.java` and make the required changes.
4. Add an `hours` field for non-exempt and contract employees, along with the associated get and set methods (note that the `payRate` field will hold a monthly amount for exempt and an hourly amount for nonexempt and contractors).
  5. Revise the `getPayInfo` method for each new class.
    - For all three, it should identify which type of employee it is.
    - For nonexempt and contract employees, calculate `payRate * hours` as the numeric value for their pay.
  6. Add constructors as you see fit, making use of calls to the appropriate super-constructors.
  7. Note that `Payroll.java` has been updated for you to properly test this new version of the program. It prompts the user to enter three employees: first an `ExemptEmployee`, then a `NonexemptEmployee`, then finally a `ContractEmployee`.



## Solution:

### Java-Inheritance/Solutions/Payroll-Inheritance01/employees/ExemptEmployee.java

```
1.   ="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="https://www.we  ←←
      bucator.com/Schemas/Courseware https://www.webucator.com/assets/wcwc/Au  ←←
      thoring/ExternalFile.xsd"><![CDATA[package employees;
2.
3.   public class ExemptEmployee extends Employee {
4.
5.       public ExemptEmployee() {
6.           }
7.       public ExemptEmployee(String firstName, String lastName) {
8.           super(firstName, lastName);
9.       }
10.      public ExemptEmployee(String firstName,String lastName, int dept) {
11.          super(firstName, lastName, dept);
12.      }
13.      public ExemptEmployee(String firstName,
14.          String lastName, double payRate) {
15.          super(firstName, lastName, payRate);
16.      }
17.      public ExemptEmployee(String firstName, String lastName, int dept,
18.          double payRate) {
19.          super(firstName, lastName, dept, payRate);
20.      }
21.      public String getPayInfo() {
22.          return "Exempt Employee " + getId() + " dept " + getDept() +
23.              " " + getFirstName() + " " + getLastName() +
24.              " paid " + getPayRate();
25.      }
26.  }
```

---

## Code Explanation

---

The primary thing to notice about this file is that it rewrites each constructor with one line to call the equivalent form of `super`.

---

## Solution:

### Java-Inheritance/Solutions/Payroll-Inheritance01/employees/NonexemptEmployee.java

```
1.   ="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="https://www.we  ←←
      bucator.com/Schemas/Courseware https://www.webucator.com/assets/wcwc/Au  ←←
      thoring/ExternalFile.xsd"><![CDATA[package employees;
2.
3.   public class NonexemptEmployee extends Employee {
4.
5.       private double hours;
6.
7.       public NonexemptEmployee() {
8.           }
9.       public NonexemptEmployee(String firstName, String lastName) {
10.          super(firstName, lastName);
11.       }
12.      public NonexemptEmployee(String firstName,
13.          String lastName, int dept) {
14.          super(firstName, lastName, dept);
15.       }
16.      public NonexemptEmployee(String firstName, String lastName,
17.          double payRate, double hours) {
18.          super(firstName, lastName, payRate);
19.          setHours(hours);
20.       }
21.      public NonexemptEmployee(String firstName, String lastName, int dept,
22.          double payRate, double hours) {
23.          super(firstName, lastName, dept, payRate);
24.          setHours(hours);
25.       }
26.
27.      public double getHours() {
28.          return hours;
29.       }
30.      public void setHours(double hours) {
31.          this.hours = hours;
32.       }
33.
34.      public String getPayInfo() {
35.          return "Non-Exempt Employee " + getId() + " dept " + getDept() +
36.              " " + getFirstName() + " " + getLastName() +
37.              " paid " + getPayRate() * hours;
38.       }
39.   }
```

## Code Explanation

---

In addition to rewriting the existing constructors, this class adds another that accepts hours, as well as appropriate methods to get and set that value.

---

## Solution:

### Java-Inheritance/Solutions/Payroll-Inheritance01/employees/ContractEmployee.java

```
1.   ="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="https://www.we  ←←
      bucator.com/Schemas/Courseware https://www.webucator.com/assets/wcwc/Au  ←←
      thoring/ExternalFile.xsd"><![CDATA[package employees;
2.
3.   public class ContractEmployee extends Employee {
4.
5.       private double hours;
6.
7.       public ContractEmployee() {
8.           }
9.       public ContractEmployee(String firstName, String lastName) {
10.          super(firstName, lastName);
11.       }
12.      public ContractEmployee(String firstName,
13.          String lastName, int dept) {
14.          super(firstName, lastName, dept);
15.       }
16.      public ContractEmployee(String firstName, String lastName,
17.          double payRate, double hours) {
18.          super(firstName, lastName, payRate);
19.          setHours(hours);
20.       }
21.      public ContractEmployee(String firstName, String lastName, int dept,
22.          double payRate, double hours) {
23.          super(firstName, lastName, dept, payRate);
24.          setHours(hours);
25.       }
26.
27.      public double getHours() {
28.          return hours;
29.       }
30.      public void setHours(double hours) {
31.          this.hours = hours;
32.       }
33.
34.      public String getPayInfo() {
35.          return "Contract Employee " + getId() + " dept " + getDept() +
36.              " " + getFirstName() + " " + getLastName() +
37.              " paid " + getPayRate() * hours;
38.       }
39.   }
```

## Code Explanation

---

This class is virtually identical to `NonexemptEmployee`. Realistically, there would be differences due to benefits, dependents, etc.

Given the similarity between `ContractEmployee` and `NonexemptEmployee`, it might be worth refactoring to have a common base class for hourly employees, especially if there were situations where we would want to work with all hourly employees, regardless of type.

---

## Solution: Java-Inheritance/Solutions/Payroll-Inheritance01/Payroll.java

```
1.  = "http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="https://www.we  ←←
    bucator.com/Schemas/Courseware https://www.webucator.com/assets/wcwc/Au  ←←
    thoring/ExternalFile.xsd"><![CDATA[import employees.*;
2.  import util.*;
3.
4.  public class Payroll {
5.      public static void main(String[] args) {
6.          Employee[] e = new Employee[5];
7.          String fName = null;
8.          String lName = null;
9.          int dept = 0;
10.         double payRate = 0.0;
11.         double hours = 0.0;
12.
13.         ExemptEmployee ee = null;
14.         NonexemptEmployee ne = null;
15.         ContractEmployee ce = null;
16.
17.         fName = KeyboardReader.getPromptedString("Enter first name: ");
18.         lName = KeyboardReader.getPromptedString("Enter last name: ");
19.         dept = KeyboardReader.getPromptedInt("Enter department: ");
20.         do {
21.             payRate = KeyboardReader.getPromptedDouble("Enter pay rate: ");
22.             if (payRate < 0.0) System.out.println("Pay rate must be >= 0");
23.         } while (payRate < 0.0);
24.         ee = new ExemptEmployee(fName, lName, dept, payRate);
25.
26.         System.out.println(ee.getPayInfo());
27.
28.         fName = KeyboardReader.getPromptedString("Enter first name: ");
29.         lName = KeyboardReader.getPromptedString("Enter last name: ");
30.         dept = KeyboardReader.getPromptedInt("Enter department: ");
31.         do {
32.             payRate = KeyboardReader.getPromptedDouble("Enter pay rate: ");
33.             if (payRate < 0.0) System.out.println("Pay rate must be >= 0");
34.         } while (payRate < 0.0);
35.         do {
36.             hours = KeyboardReader.getPromptedDouble("Enter hours: ");
37.             if (hours < 0.0) System.out.println("Hours must be >= 0");
38.         } while (hours < 0.0);
39.         ne = new NonexemptEmployee(fName, lName, dept, payRate, hours);
40.
41.         System.out.println(ne.getPayInfo());
42.
```

```
43.     fName = KeyboardReader.getPromptedString("Enter first name: ");
44.     lName = KeyboardReader.getPromptedString("Enter last name: ");
45.     dept = KeyboardReader.getPromptedInt("Enter department: ");
46.     do {
47.         payRate = KeyboardReader.getPromptedDouble("Enter pay rate: ");
48.         if (payRate < 0.0) System.out.println("Pay rate must be >= 0");
49.     } while (payRate < 0.0);
50.     do {
51.         hours = KeyboardReader.getPromptedDouble("Enter hours: ");
52.         if (hours < 0.0) System.out.println("Hours must be >= 0");
53.     } while (hours < 0.0);
54.     ce = new ContractEmployee(fName, lName, dept, payRate, hours);
55.     System.out.println(ce.getPayInfo());
56.
57.     }
58. }
```

---

## Code Explanation

---

Our main class prompts the user to enter three employees, one of each of the new types.

---

Evaluation  
\*  
Copy

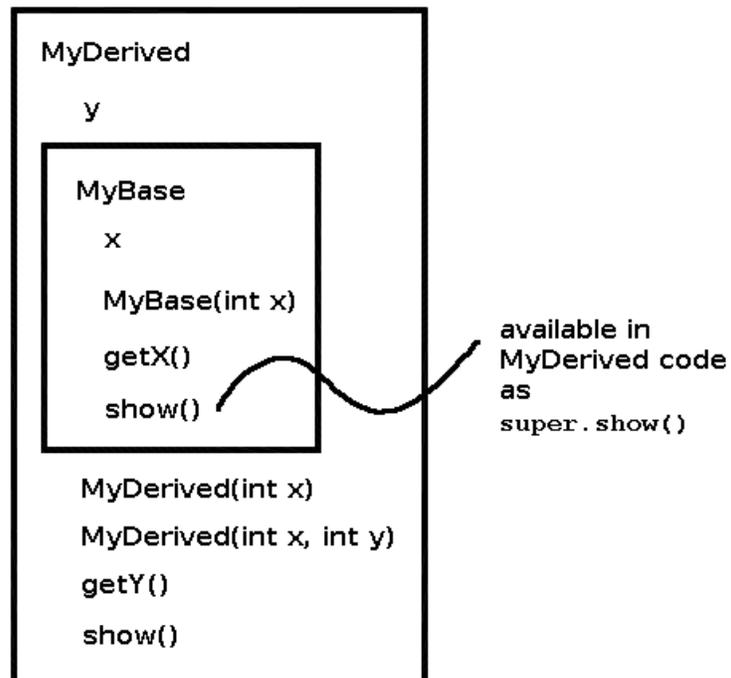
## 6.11. Derived Class Methods that Override Base Class Methods

As we saw before, you can create a method in the derived class with the same name as a base class method. Note that:

- The new method *overrides* (and hides) the original method.
- You can still call the base class method from within the derived class if necessary, by adding the `super` keyword and a dot in front of the method name.
- The base class version of the method is not available to outside code.
- You can view the `super` term as providing a reference to the base class object buried inside the derived class.
- You cannot do `super.super` to back up two levels.
- You cannot change the return type when overriding a method, since this would make polymorphism impossible.

Example: a revised MyDerived using super.show()

```
class MyDerived extends MyBase {  
    int y;  
    public MyDerived(int x) {  
        super(x);  
    }  
    public MyDerived(int x, int y) {  
        super(x);  
        this.y = y;  
    }  
    public int getY() {  
        return y;  
    }  
    public void show() {  
        super.show();  
        System.out.println("y = " + y);  
    }  
}
```



## 6.12. Inheritance and Default Base Class Constructors

One base class constructor will *a/ways* run when instantiating a new derived class object.

- If you do not explicitly call a base class constructor, the no-arguments base constructor will be automatically run, without the need to call it as `super()`.
- But if you do explicitly call a base class constructor, the no-arguments base constructor will not be automatically run.
- The no-arguments (or no-args for short) constructor is often called the default constructor, since it is the one that will run by default (and also because you are given it by default if you write no constructors).

## Demo 6.5: Java-Inheritance/Demos/Inheritance2.java

---

```
1.  class Purple {
2.      protected int i = 0;
3.      public Purple() {
4.          System.out.println("Purple() running and i = " + this.i);
5.      }
6.      public Purple(int i) {
7.          this.i = i;
8.          System.out.println("Purple(i) running and i = " + this.i);
9.      }
10. }
11.
12. class Violet extends Purple {
13.     Violet() {
14.         System.out.println("Violet() running and i = " + this.i);
15.     }
16.     Violet(int i) {
17.         System.out.println("Violet(i) running and i = " + this.i);
18.     }
19. }
20.
21. public class Inheritance2 {
22.
23.     public static void main(String[] args) {
24.         System.out.println("new Violet():");
25.         new Violet();
26.
27.         System.out.println();
28.         System.out.println("new Violet(4):");
29.         new Violet(4);
30.     }
31. }
```

---

### Code Explanation

---

Each constructor prints a message so that we can follow the flow of execution. Note that using `new Violet()` causes `Purple()` to run, and that `new Violet(4)` also causes `Purple()` to run.

For the sake of simplicity, the `i` field has been made protected, but this is not considered a good practice.

---

If your base class has constructors, but no no-arguments constructor, then the derived class must call one of the existing constructors with `super(args)`, since there will be no default constructor in the base class.

If the base class has a no-arguments constructor that is `private`, it will be there, but not be available, since `private` elements are hidden from the derived class. So, again, you must explicitly call an available form of base class constructor, rather than relying on the default.

Try the above code with the `Purple()` constructor commented out or marked as `private`.



## 6.13. The Instantiation Process at Runtime

In general, when an object is instantiated, an object is created for each level of the inheritance hierarchy. Each level is completed before the next level is started, and the following takes place *at each level*:

1. The memory block for that level is allocated (for derived classes, this means it is sized for the added elements, since the inherited elements were in the base class memory block.)
2. The entire block is zeroed out.
3. Explicit initializers for that level run, which may involve executable code, for example: `private double d = Math.random();`
4. The constructor for that level runs. Note:
  - Since the class code has already been loaded, and any more basic code has been completed, any methods in this class or inherited from superclasses are available to be called from the constructor.
  - Note that if this level's constructor calls a superconstructor, all you are really doing is selecting which form of superconstructor will run at the appropriate time. Timing wise, that superconstructor was run before we got to this point.

When the process has completed, the expression that created the instance evaluates to the address of the block for the last unit in the chain.

## ❖ 6.13.1. Inheritance and static Elements

static methods in a class may not be overridden in a derived class. This is because the static method linkages are not resolved with the same dynamic mechanism that non-static methods use. The linkage is established at compile time.



## 6.14. Typecasting with Object References

Object references can be typecast only along a chain of inheritance. If class `MyDerived` is derived from `MyBase`, then a reference to one can be typecast to the other.

An *upcast* converts from a derived type to a base type, and will be done implicitly, because it is guaranteed that everything that could be used in the base is also in the derived class.

```
Object o;  
String s = new String("Hello");  
o = s;
```

Evaluation  
Copy

A *downcast* converts from a parent type to a derived type, and must be done explicitly.

```
Object o = new String("Hello");  
String t;  
t = (String) o;
```

even though `o` came from a `String`, the compiler only recognizes it as an `Object`, since that is the type of data the variable is declared to hold.

As a memory aid, if you draw your inheritance diagrams with the parent class on top, then an upcast moves up the page, while a downcast moves down the page.

### ❖ 6.14.1. More on Object Typecasts

The compiler will not check your downcasts, other than to confirm that the cast is at least feasible. For instance, there is no possibility of a typecast between `String` and `Integer`, since neither is a base class to the other.

But, the JVM will check at runtime to make sure that the cast will succeed. A downcast could fail at runtime because you might call a method that is not there, so the JVM checks when performing the cast in order to fail as soon as possible, rather than possibly having some additional steps execute between casting and using a method.

```
Object o = new Integer(7);
String t, u;

// compiler will allow the following line, but it will fail at runtime
t = (String) o;

// we will never reach this line
u = t.toUpperCase();
```

Since `t` was not actually a `String`, this would fail with a runtime exception during the cast operation. At that point, the runtime engine sees that it cannot make the conversion and fails.

You can use a typecast in the flow of an operation, such as:

```
MyBase rv = new MyDerived(2, 3);
System.out.println(
    "x=" + rv.getX() + " y=" + ((MyDerived) rv).getY() );
```

`((MyBase) rv)` casts `rv` to a `MyDerived`, for which the `getY()` method is run.

Note the parentheses enclosing the inline typecast operation; this is because the dot operator is higher precedence than the typecast; without them we would be trying to run `rv.getY()` and typecast the result (technically, Java does not consider the dot an operator, but a *separator*, like curly braces, parentheses, and square brackets - the net effect is the same, since separators get applied before operators).

## ❖ 6.14.2. Typecasting, Polymorphism, and Dynamic Method Invocation

The concept of *polymorphism* means “one thing - many forms.”

In this case, the one thing is a base class variable; the many forms are the different types derived from that base class that can be stored in the variable.

Storing a reference in a variable of a base type does not change the contents of the object, just the compiler's identification of its type - it still has its original methods and fields.

You must explicitly downcast the references back to their original class in order to access their unique properties and methods. If you have upcast, to store a derived class object with a base class reference, the compiler will not recognize the existence of derived class methods that were not in the base class.

The collection classes, such as `Vector`, are defined to store `Objects`, so that anything you store in them loses its identity. You must downcast the reference back to the derived type in order to access those methods. The introduction of *generics* provides a solution to this annoyance (more on this in the Collections lesson).

During execution, using a base class reference to call to a method that has been overridden in the derived class will result in the derived class version being used. This is called *dynamic method invocation*.

The following example prints the same way twice, even though two different types of variables are used to reference the object:

### Demo 6.6: Java-Inheritance/Demos/Inheritance3.java

---

```
1. public class Inheritance3 {
2.     public static void main(String[] args) {
3.         MyDerived mD = new MyDerived(2, 3);
4.         MyBase mB = mD;
5.         mB.show();
6.         mD.show();
7.     }
8. }
```

---

Situations where you would often see a base class reference include:

- Parameters to methods that only need to work with base class elements of the incoming object.
- As fields of a class where the class code will only work with the base class elements of the object.
- Return values from methods where the actual class returned is not important to the user, and you want to hide the actual class from the user - for example, a `java.net.Socket` object has a public `OutputStream` `getOutputStream()` method that returns an instance of a specialized class that will send data across the network connection that the `Socket` holds; to use this class you only need to know that it

is an `OutputStream`, since there is no way you could use this class without the `Socket` it is attached to.



## 6.15. More on Overriding

### ❖ 6.15.1. Changing Access Levels on Overridden Methods

You can change the access level of a method when you override it, but only to make it more accessible.

- You can't restrict access any more than it was in the base class.
- So, for example, you could take a method that was protected in the base class and make it `public`.
- For example, if a method was `public` in the base class, the derived class may not override it with a method that has protected, `private` or package access.

This avoids a logical inconsistency:

- Since a base class variable can reference a derived class object, the compiler will allow it to access something that was `public` in the base class.
- If the derived class object actually referenced had changed the access level to `private`, then the element ought to be unavailable.
- This logic could be applied to any restriction in access level, not just `public` to `private`.

As a more specific example of why this is the case, imagine that `ExemptEmployee` overrode `public String getPayInfo()` with `private String getPayInfo()`.

The compiler would allow

#### **Why You Cannot Restrict Access When Extending a Class**

```
Employee e = new ExemptEmployee();
```

```
// getPayInfo was public in Employee, so compiler should allow this  
e.getPayInfo();
```

- Because `Employee`, the type on the variable `e`, says that `getPayInfo` is `public`.
- But, now at runtime, it shouldn't be accessible, since it is supposed to be `private` in `ExemptEmployee`.

## ❖ 6.15.2. Redefining Fields

A field in a derived class may be redefined, with a different type and/or more restrictive access - when you do this you are creating a second field that hides the first; this is called *shadowing* instead of overriding.

- A new field is created that hides the existence of the original field.
- Since it actually is a new field being created, the access level and even data type may be whatever you want - they do not have to be the same as the original field.
- I don't know of any good reason to do this *deliberately*, but it is possible.
  - A strange thing happens when you use a base class reference to such a class where the field was accessible (for example, `public`).
  - The base class reference sees the base class version of the field!

But, if you were to extend a class from the Java API or other library, you wouldn't necessarily know what fields it had - this facility allows you to use whatever field names you want, and, as long as the base class versions were `private`, you would not get any adverse effects.



## 6.16. Object Typecasting Example

Say our game program needed to store references to `Entity` objects.

- The exact type of `Entity` would be unknown at compile time.
- But during execution we would like to instantiate different types of `Entity` at random.

Perhaps our code for `Entity` includes a method `move()` that moves it to a new location. Many of the entities move the same way, but perhaps some can fly, etc. We could write a generally useful form of `moveTo` in the `Entity` class, but then override it as necessary in some of the classes derived from `Entity`.

## Demo 6.7: Java-Inheritance/Demos/EntityTest.java

---

```
1.  class Entity {
2.      private String name;
3.      public Entity(String name) { this.name = name; }
4.      public String getName() { return name; }
5.      public void moveTo() {
6.          System.out.println("I am " + name + ". Here I go!");
7.      }
8.  }
9.
10. class Playable extends Entity {
11.     public Playable(String name) { super(name); }
12.     public void moveTo() {
13.         System.out.println("I am Playable " + getName() + ". Here we go!");
14.     }
15. }
16.
17. class Ogre extends Entity {
18.     public Ogre(String name) { super(name); }
19. }
20.
21. class Troll extends Entity {
22.     public Troll(String name) { super(name); }
23. }
24.
25. class Prince extends Playable {
26.     public Prince(String name) { super(name); }
27.     // does not override public void moveTo()
28. }
29.
30. class Princess extends Playable {
31.     public Princess(String name) { super(name); }
32.     public void moveTo() {
33.         System.out.println("I am Princess " + getName() +
34.             ". Watch as I and my court move!");
35.     }
36. }
37.
38. class Wizard extends Playable {
39.     public Wizard(String name) { super(name); }
40.     public void moveTo() {
41.         System.out.println("I am the Wizard " + getName() +
42.             ". Watch me translocate!");
43.     }
44. }
```

```

45.
46. public class EntityTest {
47.     public static void main(String[] args) {
48.         String[] names = { "Glogg", "Blort", "Gruff",
49.                             "Gwendolyne", "Snow White", "Diana",
50.                             "Merlin", "Houdini", "Charles", "George" };
51.         for (int i = 0; i < 10; i++) {
52.             int r = (int) (Math.random() * 5);
53.             Entity e = null;
54.             switch (r) {
55.                 case 0: e = new Ogre(names[i]); break;
56.                 case 1: e = new Troll(names[i]); break;
57.                 case 2: e = new Wizard(names[i]); break;
58.                 case 3: e = new Prince(names[i]); break;
59.                 case 4: e = new Princess(names[i]); break;
60.             }
61.             e.moveTo();
62.         }
63.     }
64. }

```

---

The compiler allows the calls to the `moveTo` method because it is guaranteed to be present in any of the subclasses - since it was created in the base class.

- If not overridden in the derived class, then the base class version will be used.
- If the method is overridden by the derived class, then the derived class version will be used.

At runtime, the JVM searches for the method implementation, starting at the actual class of the instance, and moving up the inheritance hierarchy until it finds where the method was implemented, so the most derived version will run.



## 6.17. Checking an Object's Type: Using `instanceof`

Given that you can have base class references to several different derived class types, you will eventually come to a situation where you need to determine exactly which derived class is referenced - you may not know it at the time the program is compiled.

- In the above example, perhaps wizards, ogres, trolls, etc. have their own special methods.

- How would you know which method to call if you had an Entity reference that could hold any subclass at any time?

The `instanceof` operator is used in comparisons - it gives a `boolean` answer when used to compare an object reference with a class name.

#### instanceof Operator Syntax

```
referenceExpression instanceof ObjectType
```

- It will yield `true` if the reference points to an instance of that class.
- It will also give `true` if the object is a derived class of the one tested.
- If the test yields `true`, then you can safely `typecast` to call a derived class method (you still need to `typecast` to call the method - the compiler doesn't care that you performed the test).

For example:

```
if (e[i] instanceof Wizard) ((Wizard) e[i]).castSpell();
```

There is another method of testing:

- Every object has a `getClass()` method that returns a `Class` object that uniquely identifies each class.
- Every class has a `class` pseudo-property that provides the same `Class` object as above; this object is the class as loaded into the JVM (technically, `class` isn't a field, but syntax-wise it is treated somewhat like a `static` field).
- With this method, a derived class object's class will compare as not equal to the base class.

```
if (e[i].getClass().equals(Wizard.class))  
    ((Wizard) e[i]).castSpell();
```

It is rare that you would need this type of test.



## 6.18. Typecasting with Arrays of Objects

Unlike with arrays of primitives, it is possible to typecast an array of one type of object to an array of another type of object, if the types are compatible. The following example converts an array of strings to an array of objects and back again:

### Demo 6.8: Java-Inheritance/Demos/ObjectArrayTypecast.java

---

```
1.   ="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="https://www.we  ←←
      bucator.com/Schemas/Courseware https://www.webucator.com/assets/wcwc/Au  ←←
      thoring/ExternalFile.xsd"><![CDATA[public class ObjectArrayTypecast {
2.
3.   public static void main(String[] args) {
4.
5.       Object[] objects;
6.       String[] strings = { "A", "B", "C", "D" };
7.
8.       objects = strings;
9.       for (Object o : objects) System.out.println(o);
10.
11.      strings = null;
12.      strings = (String[]) objects;
13.      for (String s : strings) System.out.println(s);
14.  }
15. }
```

---

#### Code Explanation

---

Both upcasts and downcasts can be made. Because the arrays store references, the physical attributes, like the size, of each element remains the same regardless of what type of object the element references. Note that, on line 9, there is an implicit call to the `toString()` method when we print the object to the screen in this context.

---



# Exercise 23: Payroll-Inheritance02: Using the Employee Subclasses

⌚ 45 to 60 minutes

---

1. Open the files in `Java-Inheritance/Exercises/Payroll-Inheritance02/`.
2. We have reinstated the array of `Employee` objects in the `main` method of `Payroll`.
3. Create instances of the non-exempt employees, exempt employees, and contract employees to fill the array. You can ask which type of employee using `getPromptedChar`.
4. Create a report that lists all employees, grouped by type, by looping through the array three times.
  - The first time, show all exempt employees and their pay information.
  - The second time, print only the non-exempt employees and their pay information.
  - The third time, print contract employees and their pay information.
  - Since you'll be writing the same loop multiple times, you could try both indexed loops and for-each loops (the solution uses for-each loops).



## Solution: Java-Inheritance/Solutions/Payroll-Inheritance02/Payroll.java

```
1.  import employees.*;
2.  import util.*;
3.
4.  public class Payroll {
5.      public static void main(String[] args) {
6.          Employee[] e = new Employee[5];
7.          String fName = null;
8.          String lName = null;
9.          int dept = 0;
10.         double payRate = 0.0;
11.         double hours = 0.0;
12.
13.         for (int i = 0; i < e.length; i++) {
14.             char type = KeyboardReader.getPromptedChar("Enter type: E, N, or C: ");
15.             if (type != 'e' && type != 'E' && type != 'n' && type != 'N' && type != 'c'
16.                 && type != 'C') {
17.                 System.out.println("Please enter a valid type");
18.                 i--;
19.                 continue;
20.             }
21.             fName = KeyboardReader.getPromptedString("Enter first name: ");
22.             lName = KeyboardReader.getPromptedString("Enter last name: ");
23.             do {
24.                 dept = KeyboardReader.getPromptedInt("Enter department: ");
25.                 if (dept <= 0) System.out.println("Department must be >= 0");
26.             } while (dept <= 0);
27.             do {
28.                 payRate = KeyboardReader.getPromptedDouble("Enter pay rate: ");
29.                 if (payRate < 0.0) System.out.println("Pay rate must be >= 0");
30.             } while (payRate < 0.0);
31.
32.             switch (type) {
33.                 case 'e':
34.                     e[i] = new ExemptEmployee(fName, lName, dept, payRate);
35.                     break;
36.                 case 'n':
37.                 case 'N':
38.                     do {
39.                         hours = KeyboardReader.getPromptedDouble("Enter hours: ");
40.                         if (hours < 0.0)
41.                             System.out.println("Hours must be >= 0");
42.                     } while (hours < 0.0);
43.                 }
```

```

44.     e[i] = new NonexemptEmployee(fName, lName, dept, payRate, hours);
45.     break;
46.     case 'c':
47.     case 'C':
48.     do {
49.         hours = KeyboardReader.getPromptedDouble("Enter hours: ");
50.         if (hours < 0.0)
51.             System.out.println("Hours must be >= 0");
52.     } while (hours < 0.0);
53.     e[i] = new ContractEmployee(fName, lName, dept, payRate, hours);
54. }
55.
56. }
57.
58. System.out.println();
59. System.out.println("Exempt Employees");
60. System.out.println("=====");
61. for (Employee emp : e) {
62.     if (emp instanceof ExemptEmployee) {
63.         System.out.println(emp.getPayInfo());
64.     }
65. }
66. System.out.println();
67. System.out.println("Nonexempt Employees");
68. System.out.println("=====");
69. for (Employee emp : e) {
70.     if (emp instanceof NonexemptEmployee) {
71.         System.out.println(emp.getPayInfo());
72.     }
73. }
74. System.out.println();
75. System.out.println("Contract Employees");
76. System.out.println("=====");
77. for (Employee emp : e) {
78.     if (emp instanceof ContractEmployee) {
79.         System.out.println(emp.getPayInfo());
80.     }
81. }
82. }
83. }

```

---

## Code Explanation

We reduced the amount of code by recognizing that some of the data entry logic is common to all three types - asking for first and last names and department before the `switch`. This

required some additional shuffling of declarations, plus some restructuring to avoid asking for the name information all over again if the employee type is not valid (we used `continue` to avoid processing an employee if the type wasn't valid, and backed up the loop counter to go over the same array location on the next iteration).

For the report, the same logic is essentially repeated three times:

```
System.out.println();
System.out.println("Exempt Employees");
System.out.println("=====");
for (Employee emp : e) {
    if (e instanceof ExemptEmployee) {
        System.out.println(e[i].getPayInfo());
    }
}
```

The `instanceof` test enables us to isolate just one type of employee to print; the others will be skipped over.

---

## 6.19. Other Inheritance-related Keywords

### ❖ 6.19.1. `abstract`

`abstract` states that the item cannot be realized in the current class, but can be if the class is extended. Note:

- For a class, it states that the class can't be instantiated (it serves merely as a base for inheritance).
- For a method, it states that the method is not implemented at this level.
- The `abstract` keyword cannot be used in conjunction with `final`.

#### `abstract` Classes

`abstract` Classes are used when a class is used as a common point of maintenance for subsequent classes, but either structurally doesn't contain enough to be instantiated, or conceptually doesn't exist as a real physical entity.

```
public abstract class XYZ { ... }
```

- We will make the Employee class abstract in the next exercise: while the concept of an employee exists, nobody in our payroll system would ever be just an employee, they would be exempt, nonexempt, or contract employees.
- While you cannot instantiate an object from an abstract class, you can still create a reference variable whose type is that class.

## abstract Methods

The method cannot be used in the current class, but only in a inheriting class that overrides the method with code.

```
public abstract String getPayInfo();
```

- The method is not given a body, just a semicolon after the parentheses.
- If a class has an abstract method, then the class must also be abstract.
- You can extend a class with an abstract method without overriding the method with a concrete implementation, but then the class must be marked as abstract.

## ❖ 6.19.2. final

final is used to mark something that cannot be changed.

### final Classes

The class cannot be extended.

```
public final class XYZ { ... }
```

### final Methods

The method cannot be overridden when the class is extended.

```
public final void display() { ... }
```

## final Properties

`final` Properties marks the field as a constant.

```
public static final int MAX = 100;
```

A `final` value can be initialized in a constructor or initializer:

```
public final double randConstant = Math.random();
```

or

```
public final double randConstant;
```

Then, in a constructor:

```
randConstant = Math.random();
```

Evaluation  
Copy

Note: `String` and the wrapper classes use this in two ways:

1. The class is `final`, so it cannot be extended.
2. The internal field storing the data is `final` as well, but set by the constructor (this makes the instance *immutable* - the contents cannot be changed once set)

In some cases, a declaration of `final` enables the compiler to optimize methods, since it doesn't have to leave any "hooks" in for potential future inheritance.

Note that `final` and `abstract` cannot be used for the same element, since they have opposite effects.

In the next exercise we'll ask you to apply this concept - to make some of the classes and method in our Payroll program `abstract`.

## Exercise 24: Payroll-Inheritance03: Making our base classes abstract

 5 to 10 minutes

---

1. Open the files in `Java-Inheritance/Exercises/Payroll-Inheritance03/`
2. Mark the `Person` and `Employee` classes as abstract.
3. Make the `getPayInfo()` method abstract in `Employee`.

## Solution:

### Java-Inheritance/Solutions/Payroll-Inheritance03/employees/Person.java

```
1.   ="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="https://www.we  ←←
      bucator.com/Schemas/Courseware https://www.webucator.com/assets/wcwc/Au  ←←
      thoring/ExternalFile.xsd"><![CDATA[package employees;
2.
3.   public abstract class Person {
4.       private String firstName;
5.       private String lastName;
6.
7.       public Person() {
8.           }
9.
10.      public Person(String firstName, String lastName) {
11.          setFirstName(firstName);
12.          setLastName(lastName);
13.      }
14.
15.      public String getFirstName() { return firstName; }
16.
17.      public void setFirstName(String firstName) {
18.          this.firstName = firstName;
19.      }
20.
21.      public String getLastName() { return lastName; }
22.
23.      public void setLastName(String lastName) {
24.          this.lastName = lastName;
25.      }
26.
27.      public String getFullName() {
28.          return firstName + " " + lastName;
29.      }
30.
31.  }
```



---

### Code Explanation

We mark class `Person` as `abstract` - it is a class that serves as a base for derived classes, but is never meant to be instantiated as an actual object. As we see here, a class marked as `abstract` doesn't necessarily need to have an abstract method.

---

## Solution:

### Java-Inheritance/Solutions/Payroll-Inheritance03/employees/Employee.java

```
1.    ="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="https://www.we  ←←
      bucator.com/Schemas/Courseware https://www.webucator.com/assets/wcwc/Au  ←←
      thoring/ExternalFile.xsd"><![CDATA[package employees;

2.
3.    public abstract class Employee extends Person {
4.        private static int nextId = 1;
5.        private int id = nextId++;
6.        private int dept;
7.        private double payRate;
8.
9.        public Employee() {
10.           }
11.       public Employee(String firstName, String lastName) {
12.           super(firstName, lastName);
13.       }
14.       public Employee(String firstName,String lastName, int dept) {
15.           super(firstName, lastName);
16.           setDept(dept);
17.       }
18.       public Employee(String firstName, String lastName, double payRate) {
19.           super(firstName, lastName);
20.           setPayRate(payRate);
21.       }
22.       public Employee(String firstName,
23.           String lastName, int dept, double payRate) {
24.           this(firstName, lastName, dept);
25.           setPayRate(payRate);
26.       }
27.
28.       -----Lines 28 through 48 Omitted-----
49.
50.       public abstract String getPayInfo();
51.     }
```

---

### Code Explanation

Class Employee is also marked as abstract; as with Person, Employee is never meant to be instantiated. We also mark method setPayRate as abstract, since each class that

derives from `Employee` - `ContractEmployee`, `ExemptEmployee`, and `NonexemptEmployee` - should implement their own appropriate version of `setPayRate`.

---



## 6.20. Methods Inherited from `Object`

There are a number of useful methods defined for `Object`.

Some are useful as is, such as:

`Class getClass()` - returns a `Class` object (a representation of the class that can be used for comparisons or for retrieving information about the class).

Others are useful when overridden with code specific to the new class:

`Object clone()` - creates a new object that is a copy of the original object. This method must be overridden, otherwise an exception will occur (the `Object` version of `clone` throws a `CloneNotSupportedException`).

The issue is whether to perform a *shallow copy* or a *deep copy* - a shallow copy merely copies the same reference addresses, so that both the original object and the new object point to the same internal objects; a deep copy makes copies of all the internal objects (and then what if the internal objects contained references to objects ...? ).

`boolean equals(Object)` - does a comparison between this object and another. If you don't override this method, you get the same result as if you used `==` (that is, the two references must point to the same object to compare as equal - two different objects with the same field values would compare as unequal) - that is how the method is written in the `Object` class. You would override this method with whatever you need to perform a comparison.

`int hashCode()` - returns an integer value used by collection objects that store elements using a *hashtable*. Elements that compare as the same using the `equals(Object)` method should have the same hashcode.

`String toString()` - converts this object to a string representation.

This method is called by some elements in the Java API when the object is used in a situation that requires a `String`, for example, when you concatenate the object with an existing `String`, or send the object to `System.out.println()`.

Note that the call to `toString` is not made automatically as a typecast to a `String` - the only behavior built into the syntax of Java is string concatenation with the `+` sign (so one of the operands must already be a `String`); the code in `println` is explicitly written to call `toString`.

If you don't override this method, you will get a strange string including the full class name and the *hashCode* for the object `void finalize()` - called by the JVM when the object is garbage-collected. This method might never be called (the program may end without the object being collected).

There are also several methods (`wait`, `notify`, and `notifyAll`) related to locking and unlocking an object in multithreaded situations.

## Demo 6.9: Java-Inheritance/Demos/ObjectMethods.java

---

```
1.   ="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="https://www.we  ←←
      bucator.com/Schemas/Courseware https://www.webucator.com/assets/wcwc/Au  ←←
      thoring/ExternalFile.xsd"><![CDATA[public class ObjectMethods {
2.   int id;
3.   String name;
4.   int age;
5.
6.   ObjectMethods(int id, String name, int age) {
7.       this.id = id;
8.       this.name = name;
9.       this.age = age;
10.  }
11.  public boolean equals(Object x) {
12.      if (x == this) return true;
13.      else if (x instanceof ObjectMethods) {
14.          ObjectMethods omx = (ObjectMethods) x;
15.          return id == omx.id && name.equals(omx.name) && age == omx.age;
16.      }
17.      else return false;
18.  }
19.  public int hashCode() {
20.      return id + age * 1000;
21.  }
22.  public String toString() {
23.      return id + " " + name + " is " + age + " years old";
24.  }
25.  public Object clone() {
26.      return new ObjectMethods(id, name, age);
27.  }
28.  public static void main(String[] args) {
29.      ObjectMethods om1 = new ObjectMethods (1, "John", 6);
30.      ObjectMethods om2 = new ObjectMethods (1, "John", 6);
31.      ObjectMethods om3 = new ObjectMethods (2, "Jane", 5);
32.      ObjectMethods om4 = (ObjectMethods)om3.clone();
33.      System.out.println("Printing an object: " + om1);
34.      if (om1.equals(om2))
35.          System.out.println("om1 equals(om2)");
36.      if (om1.equals(om3))
37.          System.out.println("om1 equals(om3)");
38.      if (om1.equals("John"))
39.          System.out.println("om1 equals(\"John\")");
40.      if (om3.equals(om4))
41.          System.out.println("om3 equals(om4) which was cloned from om3");
42.      System.out.println("object class is: " + om1.getClass());
```

```
43.     }  
44.     }
```

---

## Code Explanation

---

The `clone` method returns `Object` rather than `ObjectMethods`, since that is how it was declared in the `Object` class, and you can't change the return type when overriding - thus the typecast on the returned value.

Similarly, the parameter to `equals` is `Object`, rather than `ObjectMethods`. This is not required by Java's syntax rules, but rather a convention that enables other classes to work with this class. For example, the Collections API classes use the `equals` method to determine if an object is already in a set. If we wrote the method as `equals(ObjectMethods om)` instead of `equals(Object o)`, the collections classes would call `equals(Object o)` as inherited from `Object`, which would test for identity using an `==` test.

The `hashCode` method was written out of a sense of duty - the behavior of `hashCode` should be consistent with `equals`, meaning that if two items compare as equal, then they should have the same hash code. We will revisit this in the lesson on Collections.

---

## Conclusion

In this lesson, you have learned:

- About the role of inheritance in a Java program.
- How to declare and use classes that extend existing classes .
- About methods inherited from `Object`.



# LESSON 7

## Interfaces

---

### Topics Covered

- ☑ The concept of interfaces.
- ☑ Defining and writing code using your own interfaces.
- ☑ Interfaces in the Java API.

### Introduction

In this lesson, you will learn about the concept of interfaces, to define and write code using your own interfaces, and to use interfaces in the Java API.

—\*—

### 7.1. Interfaces

Evaluation  
Copy

*Interfaces* define a standardized set of commands that a class will obey.

The commands are a set of methods that a class *implements*.

The interface definition states the names of the methods and their return types and argument signatures. Often there is no executable body for methods; this is left to each class that implements the interface. (As of Java 8, an interface can fully define a method but we do not cover that in this course.)

Once a class implements an interface, the Java compiler knows that an instance of the class will contain the specified set of methods. Therefore, it will allow you to call those methods for an object referenced by a variable whose type is the interface.

One useful way to think of an interface is as a *contract*: any class that implements the interface agrees to (must) implement all of the methods and fields defined in the interface.

Implementing an interface enables a class to be “plugged in” to any situation that requires a specific behavior (manifested through the set of methods).

An analogy: a serial interface on a computer defines a set of pin/wire assignments and the control signals that will be used. Note that:

- The actual devices that can be used may do entirely different tasks: mouse, keyboard, monitor, etc.
- But they are all controlled through the same digital instruction mechanism; the individual wires are specified to carry specific signals.

Using an interface rather than inheritance to specify a certain set of methods allows a class to inherit from some other class.

- In other words, if a class needs two different sets of methods, so it can behave like two different types of things, it could inherit one set from class A, and use an interface B to specify the other.
- You could then reference one of these objects with either an A reference or a B reference.

Interfaces can also specify constants that are `public`, `static`, and `final`.

---

## 7.2. Creating an Interface Definition

To create an interface definition:

- Define it like a Java class, in its own file that matches the interface name.
- Use the keyword `interface` instead of `class`.
- Declare methods using the same approach as abstract methods.
  - Note the semicolon after each method declaration - and that no executable code is supplied (and no curly braces).
  - The elements will automatically be `public` and `abstract`, and cannot have any other state; it is OK to specify those terms, but not necessary (usually `public` is specified and `abstract` is not - that makes it easy to copy the list of methods, paste them into a class, and modify them).
  - As of Java 8 the interface can include static and default methods. This is beyond the scope of this course.
- The access level for the entire interface is usually `public`.

- It may be omitted, in which case the interface is only available to other classes in the same package (i.e., in the same directory).
- Note, for the sake of completeness, there are situations where the interface definition could be protected or private; these involve what are called *inner classes*.

### **Defining an Interface**

```
[modifiers] interface InterfaceName {  
  
    // declaring methods  
  
    [public abstract] returnType methodName(arguments);  
  
    // defining constants  
  
    [public static final]  
    type fieldName = value;  
}
```

Example:

### **Demo 7.1: Java-Interfaces/Demos/Printable.java**

---

```
1. public interface Printable {  
2.     void printAll();  
3. }
```

---

### Code Explanation

---

This interface requires only one method. Any class implementing `Printable` must contain a `public void printAll()` method in order to compile.

---

Because the above interface is defined as `public`, its definition must be in its own file, even though that file will be tiny.

An interface definition may also define fields that are automatically `public static final` - these are used as constants.



## 7.3. Implementing Interfaces

A class definition may, in addition to whatever else it does, implement one or more interfaces.

Once a class states that it implements an interface, it must supply all the methods defined for that interface, complete with executable code.

- Note: it actually does not have to implement all of them, but in that case the class cannot be instantiated - it must be declared as an `abstract` class that can only be used as a base class (where some derived class would then fully implement the interface).

To implement an interface:

- Add that the class implements the interface to the class declaration.
- Add the methods specified by the interface to the body of the class. Note that you *do* need to specify the access terms on methods in a class that implements an interface.

### Implementing an Interface

```
[modifiers] class ClassName implements InterfaceName {  
  
    any desired fields  
  
    // implement required methods  
    [modifiers]  
        returnType methodName1(arguments) {  
            executable code  
        }  
  
    any other desired methods  
  
}
```

It is important to note that a class may implement an interface in addition to whatever else it might do, so it could have additional fields and methods not associated with the interface.

A class may implement more than one interface - that merely adds to the list of required methods. Use a comma-separated list for the interface names.

## Implementing Multiple Interfaces

```
[modifiers] class ClassName implements Interface1Name, Interface2Name {  
  
    // must implement all methods from all implemented interfaces
```

### ❖ 7.3.1. Implementing Interfaces - Example

The complete example will use three separate files (the third file will be shown shortly):

#### **Demo 7.2: Java-Interfaces/Demos/Printable.java**

---

```
1.  public interface Printable {  
2.      void printAll();  
3.  }
```

---

#### **Demo 7.3: Java-Interfaces/Demos/PrintableThings.java**

---

```
1.  class Person implements Printable {  
2.      private String name = new String("Bill");  
3.      private int age = 22;  
4.      public void printAll() {  
5.          System.out.println("Name is " + name + ", age is " + age);  
6.      }  
7.  }  
8.  class Stock implements Printable {  
9.      private String tickerSymbol = new String("XYZ");  
10.     private int shares = 100;  
11.     private int currentPrice = 4000; // in pennies  
12.     public void printAll() {  
13.         System.out.println(tickerSymbol + " " + shares +  
14.             " shares at " + currentPrice);  
15.         System.out.println("Value: " + currentPrice * shares);  
16.     }  
17.     public void sell() {  
18.         System.out.println(tickerSymbol + " sold");  
19.     }  
20. }
```

---

## Code Explanation

---

This file contains two classes with package access. Since the classes are not public, they can both be in the same file, and the file name does not need to match either class name. This is done purely as a convenience; it is not a good programming practice in general, but is sometimes useful if one class is *highly coupled* (interrelated) with the other, which is not the case here. Both classes implement the `Printable` interface, but are otherwise not related. `Stock` has another method not related to `Printable`.

---



## 7.4. Reference Variables and Interfaces

An interface is like a class where the internal structure and some of the behavior is hidden.

- Interfaces are listed like classes in the API documentation.
- They compile to a `.class` file, and get loaded by the same process that loads true classes.

Since a class that implements an interface is a class in all other respects, you can create a reference variable for that class, as usual.

You can also create a reference variable whose type is the interface name.

- Only the methods defined in the interface are visible through a variable whose type is the interface.
  - For a `Printable` variable containing a `Stock` instance, the `sell` method is not visible, since it is not declared in `Printable`.
- Any constants defined by the interface can be accessed without a prefix from code within the class, since implementing the interface makes them part of this class.

To access an interface-implementing class with an interface class reference:

### **Creating a Reference Variable for an Interface**

```
[modifiers]
InterfaceName variableName;
```

Example:

- Both Person and Stock implement Printable.
- Therefore, we can create a reference variable to a Printable, and assign either a Person or a Stock object to it.
- We can then call the `printAll()` method from the Printable reference variable, since the compiler knows that method will exist, no matter which type of object is actually stored in the variable.

```
Person p = new Person();  
Stock s = new Stock();  
Printable pr;  
pr = p;
```

or

```
pr = s;
```

or

```
pr = new Person();
```

Evaluation  
Copy

### ❖ 7.4.1. Calling an Interface Method

If you have a variable that is declared as a reference to the interface type, you can use it to call an interface method.

- Note that you cannot call any of the additional methods that are not defined by the interface.

## Demo 7.4: Java-Interfaces/Demos/PrintableTest.java

---

```
1. public class PrintableTest {
2.     public static void main(String[] args) {
3.         Person p = new Person();
4.         Stock s = new Stock();
5.
6.         p.printAll();
7.         s.printAll();
8.
9.         Printable pr;
10.        pr = p;
11.        pr.printAll();
12.        pr = s;
13.        pr.printAll();
14.    }
15. }
```

---

### Code Explanation

---

Once `pr` has been assigned a `Printable` instance, we can call `pr.printAll()`;

- We cannot directly call the `sell()` method when `pr` refers to a `Stock`, since the compiler would not associate it with a variable whose type was `Printable`.

Note: to compile this, use `*.java`; since the name of the file containing `Stock` and `Person` is `PrintableThings.java`, the compiler won't be able to find those classes, since it would be looking for `Person.java` and `Stock.java`.

---

Note: you can test the type of object actually contained in an interface reference, and typecast it back to that type.

- for instance, to use the `sell()` method for a `Stock`:

```
if (pr instanceof Stock) ((Stock) pr).sell();
```



## 7.5. Interfaces and Inheritance

If a class implements an interface, then all subclasses of it will also automatically implement the interface.

- They are guaranteed to have the necessary methods available.
- It is a good practice to specify that the derived class implements the interface, just for self-documentation of the code (also for purposes of javadoc, if the base class is not in the same group of files).

An interface definition can inherit from another interface.

- The new interface then adds fields and methods to the existing (base) definition.
- A class that implements the new interface must implement all methods from the base interface as well as all the additional methods from the new interface definition.
- An interface can actually extend multiple base interfaces, in which case the combination of all the methods will be required for any implementing class.

The following interface extends the `Printable` interface and adds another required method (the new method overloads `printAll` to print to a specified destination instead of to `System.out`):

### Demo 7.5: Java-Interfaces/Demos/Printable2.java

---

```
1. import java.io.PrintStream;
2. public interface Printable2 extends Printable {
3.     public void printAll(PrintStream p);
4. }
```

---

A class implementing `Printable2` must define both versions of `printAll`.

## Demo 7.6: Java-Interfaces/Demos/Printable2Test.java

---

```
1.  import java.io.PrintStream;
2.  class Cat implements Printable2 {
3.      public void printAll() {
4.          printAll(System.out);
5.      }
6.      public void printAll(PrintStream out) {
7.          out.println("Meow");
8.      }
9.  }
10.
11. public class Printable2Test {
12.     public static void main(String[] args) {
13.         Printable2 c = new Cat();
14.         c.printAll(System.err);
15.         c.printAll();
16.     }
17. }
```

---

Evaluation  
Copy

# Exercise 25: Exercise: Payroll-Interfaces01

 30 to 40 minutes

---

It turns out that our hypothetical system is to be used for all payments our company makes, not just payroll checks, and things like invoices will be paid through the system as well.

1. Open the files in `Java-Interfaces/Exercises/Payroll-Interfaces01/`.
2. Within the `finance` package directory, create an interface called `Payable`.
  - This interface should define the `public String getPayInfo()` method that our employee classes already implement.
3. Specify that all the employee classes implement `Payable`.
4. The `Java-Interfaces/Exercises/Payroll-Interfaces01` directory contains a package called `vendors` with a class named `Invoice`. Modify the payroll program by adding an array `inv` of several invoices (you can just hard code them).
5. The `finance` directory contains a file named `CheckPrinter.java`. The class has a `printChecks(Payable[])` method that you can call twice, once for employees and again for invoices.

## Solution:

### Java-Interfaces/Solutions/Payroll-Interfaces01/finance/Payable.java

---

```
1. package finance;
2.
3. public interface Payable {
4.
5.     public String getPayInfo();
6.
7. }
```

---

Evaluation  
Copy

#### Code Explanation

---

This interface declares the `public String getPayInfo()` method that our employee classes already implement. No change is required to any other code, but we can now be sure that any object of a class that implements interface `Payable` will be able to call method `getPayInfo()`.

---

## Solution:

### Java-Interfaces/Solutions/Payroll-Interfaces01/employees/Employee.java

```
1.  package employees;
2.  import finance.Payable;
3.
4.  public class Employee extends Person implements Payable {
5.      private static int nextId = 1;
6.      private int id = nextId++;
7.      private int dept;
8.      private double payRate;
9.
10.     public Employee() {
11.     }
12.     public Employee(String firstName, String lastName) {
13.         super(firstName, lastName);
14.     }
15.     public Employee(String firstName, String lastName, int dept) {
16.         super(firstName, lastName);
17.         setDept(dept);
18.     }
19.     public Employee(String firstName, String lastName, double payRate) {
20.         super(firstName, lastName);
21.         setPayRate(payRate);
22.     }
23.     public Employee(String firstName, String lastName,
24.         int dept, double payRate) {
25.         this(firstName, lastName, dept);
26.         setPayRate(payRate);
27.     }
28.
29.     public static int getNextId() {
30.         return nextId;
31.     }
32.
33.     public static void setNextId(int nextId) {
34.         Employee.nextId = nextId;
35.     }
36.
37.     public int getId() { return id; }
38.
39.     public int getDept() { return dept; }
40.
41.     public void setDept(int dept) {
42.         this.dept = dept;
43.     }
```

```
44.
45.     public double getPayRate() { return payRate; }
46.
47.     public void setPayRate(double payRate) {
48.         this.payRate = payRate;
49.     }
50.
51.     public String getPayInfo() {
52.         return "Employee " + id + " dept " + dept + " " +
53.             getFirstName() + " " + getLastName() +
54.             " paid " + payRate;
55.     }
56. }
```

---

Evaluation  
Copy

## Code Explanation

---

The class has been marked as implementing `Payable`. Although it would not be required, we should (and do) mark the derived classes the same way, to have more self-documenting code.

---

## Solution: Java-Interfaces/Solutions/Payroll-Interfaces01/Payroll.java

---

```
1.  import employees.*;
2.  import vendors.Invoice;
3.  import finance.*;
4.  import util.*;
5.
6.  public class Payroll {
7.      public static void main(String[] args) {
8.          Employee[] e = new Employee[5];
9.          String fName = null;
10.         String lName = null;
11.         int dept = 0;
12.         double payRate = 0.0;
13.         double hours = 0.0;
14.
15.         for (int i = 0; i < e.length; i++) {
16.             char type = KeyboardReader.getPromptedChar("Enter type: E, N, or C: ");
17.             if (type != 'e' && type != 'E' &&
18.                 type != 'n' && type != 'N' &&
19.                 type != 'c' && type != 'C') {
20.                 System.out.println("Please enter a valid type");
21.                 i--;
22.                 continue;
23.             }
24.             fName = KeyboardReader.getPromptedString("Enter first name: ");
25.             lName = KeyboardReader.getPromptedString("Enter last name: ");
26.             do {
27.                 dept = KeyboardReader.getPromptedInt("Enter department: ");
28.                 if (dept <= 0) System.out.println("Department must be >= 0");
29.             } while (dept <= 0);
30.             do {
31.                 payRate = KeyboardReader.getPromptedDouble("Enter pay rate: ");
32.                 if (payRate < 0.0) System.out.println("Pay rate must be >= 0");
33.             } while (payRate < 0.0);
34.
35.             switch (type) {
36.                 case 'e':
37.                 case 'E':
38.                     e[i] = new ExemptEmployee(fName, lName, dept, payRate);
39.                     break;
40.                 case 'n':
41.                 case 'N':
42.                     do {
43.                         hours = KeyboardReader.getPromptedDouble("Enter hours: ");
44.                         if (hours < 0.0)
```

```

45.     System.out.println("Hours must be >= 0");
46.     } while (hours < 0.0);
47.     e[i] = new NonexemptEmployee(fName, lName, dept, payRate, hours);
48.     break;
49.     case 'c':
50.     case 'C':
51.     do {
52.         hours = KeyboardReader.getPromptedDouble("Enter hours: ");
53.         if (hours < 0.0)
54.             System.out.println("Hours must be >= 0");
55.         } while (hours < 0.0);
56.         e[i] = new ContractEmployee(fName, lName, dept, payRate, hours);
57.     }
58.
59.     System.out.println(e[i].getPayInfo());
60. }
61.
62. System.out.println();
63. System.out.println("Exempt Employees");
64. System.out.println("=====");
65. for (Employee emp : e) {
66.     if (emp instanceof ExemptEmployee) {
67.         System.out.println(emp.getPayInfo());
68.     }
69. }
70. System.out.println();
71. System.out.println("Nonexempt Employees");
72. System.out.println("=====");
73. for (Employee emp : e) {
74.     if (emp instanceof NonexemptEmployee) {
75.         System.out.println(emp.getPayInfo());
76.     }
77. }
78. System.out.println();
79. System.out.println("Contract Employees");
80. System.out.println("=====");
81. for (Employee emp : e) {
82.     if (emp instanceof ContractEmployee) {
83.         System.out.println(emp.getPayInfo());
84.     }
85. }
86.
87. Invoice[] inv = new Invoice[4];
88. inv[0] = new Invoice("ABC Co.", 456.78);
89. inv[1] = new Invoice("XYZ Co.", 1234.56);

```



```
90.     inv[2] = new Invoice("Hello, Inc.", 999.99);
91.     inv[3] = new Invoice("World, Ltd.", 0.43);
92.
93.     CheckPrinter.printChecks(e);
94.     CheckPrinter.printChecks(inv);
95. }
96.
97. }
```

---

## Code Explanation

---

The last several lines of code create an array of invoices, then use the `CheckPrinter` to print employees and invoices separately. It would also be possible to create an array of `Payable` objects, and add in the elements from both the employees and invoices arrays, but that seems like an unnecessary complication for this application.

---



## 7.6. Some Uses for Interfaces

### ❖ 7.6.1. Interfaces and Event-Handling

A real-world use of interfaces is for event-handling:

- An object that can generate an event maintains a list of objects that would like to listen for that event (they will be notified when the event occurs by having one of their methods called).
- The object that generates the event *fires* it by going through its list of objects that want to handle the event, and calling a specified interface method for each object.
- A class may handle an event if it implements the interface that is expected for that event - therefore it will have the specified method.
- You *register* an object to handle an event by passing a reference to it to the event-generating object's method that adds a handler.

Assuming that there is an event type called `XXXEvent`:

- The handler interface would probably be named `XXXListener`.
- The method to register a listener would usually be called `addXXXListener`.

- The method generating the event probably has a protected utility method called `fireXXXEvent` that it uses to trigger the event notifications (and it is available for you to call if you extend the class).

The `ActionListener` interface is used for GUI events like button clicks.

- The event is fired by the GUI object calling the `actionPerformed` method for any registered listeners (the code to do this is already built into the GUI classes, and the Java API defines the interface shown below).

#### **The ActionListener Interface**

```
public interface ActionListener {
    public void actionPerformed(ActionEvent e);
}
```

A class can listen for events if it implements `ActionListener`.

- It can either register itself with the event-generating object, or code outside the class can register it - the example below shows how it would register itself using this:

#### **A Class Implementing ActionListener**

```
public class MyClass implements ActionListener {
    ...
    public void actionPerformed(ActionEvent e) {
        System.out.println("Event occurred");
    }
    someOtherMethod() {
        guiComponent.addActionListener(this);
    }
}
```

For the class that fires the event, registering is done with the `addActionListener(ActionListener)` method, which receives a reference to an `ActionListener` object:

- It adds that reference to a list (maybe a `java.util.Vector`) of listeners.
- When the time comes to fire the event, it walks through the list, calling `actionPerformed()` for each element on the list (and passing a reference to an event object that it creates).

For the sake of completeness, when the listener interface has multiple methods, there are often abstract classes that implement most or all of the methods as do-nothing methods - so that all you need to do is extend the class and implement the methods that you choose.

## ❖ 7.6.2. Interfaces and “Pluggable Components”

### The `TableModel` interface

The Swing classes contain a component called `JTable`, which displays a spreadsheet-like grid. Note that:

- It uses a *Model-View-Controller* approach to separate these sections of logic into individual classes.
- The `TableModel` interface defines a set of methods that allow a `JTable` (the controller) to query a data model to find out information in order to display it.
- The interface forms a framework for a discussion that will take place between the controller and the model (like, “How many rows do you have?” and, “How many columns?”, followed by “What’s the value at column 0, row 0?”, etc.).

Below are some of the methods from `TableModel`:

| <b>public interface TableModel</b>                                                                        |
|-----------------------------------------------------------------------------------------------------------|
| <code>int getColumnCount()</code>                                                                         |
| Returns the number of columns in the model.                                                               |
| <code>int getRowCount()</code>                                                                            |
| Returns the number of rows in the model.                                                                  |
| <code>String getColumnName(int columnIndex)</code>                                                        |
| Returns the name of the column at <code>columnIndex</code> .                                              |
| <code>Class&lt;?&gt; getColumnClass(int columnIndex)</code>                                               |
| Returns the most specific superclass for all the cell values in the column.                               |
| <code>Object getValueAt(int rowIndex, int columnIndex)</code>                                             |
| Returns the value for the cell at <code>columnIndex</code> and <code>rowIndex</code> .                    |
| <code>boolean isCellEditable(int rowIndex, int columnIndex)</code>                                        |
| Returns true if the cell at <code>rowIndex</code> and <code>columnIndex</code> is editable.               |
| <code>void setValueAt(Object aValue, int rowIndex, int columnIndex)</code>                                |
| Sets the value in the cell at <code>columnIndex</code> and <code>rowIndex</code> to <code>aValue</code> . |

You can see the conversation that will take place between the controller and the model. Note that:

- The controller will ask the model for the number of rows and columns, and, with that information, ask for the value at each location.
- It will ask for the type of data with `getColumnClass`, so it can determine from its settings how to display the values (instances of `Number`, which `Integer`, `Double`, etc., extend, get right-aligned, `Boolean` columns use a check box, all others get left-aligned - these settings are configurable).
- It will get a heading for each column with `getColumnName`.
- If a cell is double-clicked, it can ask if the cell is editable with `isCellEditable`.
  - If it is, when the user is done editing, it can put the new data into the model using `setValueAt`.

## Demo 7.7: Java-Interfaces/Demos/TableModelExample.java

---

```
1.  import java.util.*;
2.  import java.awt.*;
3.  import java.awt.event.*;
4.  import javax.swing.*;
5.  import javax.swing.table.*;
6.
7.  public class TableModelExample {
8.
9.      public static void main(String[] args) {
10.
11.          DemoTableModel model = new DemoTableModel();
12.
13.          new TableGUI("Table Model Example", model).setVisible(true);
14.          new TableConsole(model);
15.          new TableHTML(model);
16.      }
17.  }
18.
19.  class DemoTableModel extends AbstractTableModel {
20.
21.      String[] titles = { "Name", "Active", "Grade" };
22.      String[] names = { "Mary", "Joe", "Sue" };
23.      Boolean[] actives = { new Boolean(true), new Boolean(false),
24.                           new Boolean(true) };
25.      Integer[] grades = { new Integer(99), new Integer(87),
26.                          new Integer(89) };
27.      public int getRowCount() { return names.length; }
28.
29.      public int getColumnCount() { return 3; }
30.
31.      public String getColumnName(int col) {
32.          return titles[col];
33.      }
34.
35.      public Object getValueAt(int row, int column) {
36.          if (column == 0) return names[row];
37.          else if (column == 1) return actives[row];
38.          else return grades[row];
39.      }
40.
41.      public void setValueAt(Object v, int row, int column) {}
42.
43.      public Class getColumnClass(int column) {
44.          if (column == 0) return String.class;
```

```

45.         else if (column == 1) return Boolean.class;
46.         else return Integer.class;
47.     }
48.
49.     public boolean isCellEditable(int row, int column) {
50.         return false;
51.     }
52.
53. }
54.
55. class TableGUI extends JFrame {
56.
57.     public TableGUI(String title, DemoTableModel model) {
58.         super(title);
59.         JTable jt;
60.         this.setDefaultCloseOperation(EXIT_ON_CLOSE);
61.         jt = new JTable(model);
62.         setSize(600,170);
63.         getContentPane().add(jt);
64.     }
65. }
66.
67. class TableConsole {
68.
69.     TableModel model;
70.
71.     public TableConsole(TableModel model) {
72.
73.         int rows = model.getRowCount();
74.         int cols = model.getColumnCount();
75.
76.         for (int c = 0; c < cols; c++) {
77.             System.out.print(fixedStringLength(model.getColumnName(c), 15));
78.         }
79.         System.out.println();
80.         System.out.println("-----");
81.         for (int r = 0; r < rows; r++) {
82.             for (int c = 0; c < cols; c++) {
83.                 System.out.print(
84.                     fixedStringLength(model.getValueAt(r, c).toString(), 15));
85.             }
86.             System.out.println();
87.         }
88.     }
89.     private String fixedStringLength(String in, int size) {

```

Evaluation  
Copy

```

90.     if (in.length() > size) in = in.substring(0, 15);
91.     char[] blankArray = new char[size - in.length()];
92.     for (int i = 0; i < blankArray.length; i++) blankArray[i] = ' ';
93.     return in + String.valueOf(blankArray);
94. }
95. }
96.
97. class TableHTML {
98.
99.     TableModel model;
100.
101.     public TableHTML(TableModel model) {
102.         java.io.PrintStream out = System.out;
103.         int rows = model.getRowCount();
104.         int cols = model.getColumnCount();
105.         out.println("<html><Head><title>My Table</title></head>");
106.         out.println(
107.             "<body><table border='1' cellpadding='8' cellspacing='0'><tr>");
108.         for (int c = 0; c < cols; c++) {
109.             out.print("<th>" + model.getColumnName(c) + "</th>");
110.         }
111.         out.println("</tr>");
112.         for (int r = 0; r < rows; r++) {
113.             out.println("<tr>");
114.             for (int c = 0; c < cols; c++) {
115.                 System.out.print("<td>" + model.getValueAt(r, c) + "</td>");
116.             }
117.             out.println("</tr>");
118.         }
119.         out.println("</table></body></html>");
120.     }
121. }

```

---

## Code Explanation

---

For convenience, all the classes are in one file.

The `DemoTableModel` class implements `TableModel` by extending `AbstractTableModel`, thus gaining implementations of several methods (like those relating to model change event listener lists), then adding the remaining methods.

The model is based on parallel arrays of student data: name, grade, and active or not—each array represents one column of data, and element 0 in each array is the same student.

The `titles` array holds column names.

`getColumnCount` returns 3, because we know that in advance.

`getRowCount` returns the length of one of the data arrays.

For `getColumnName`, we return an appropriate string from the `titles` array.

For `getValueAt`, we pick an array based on the `column` number, and return the element at the `row` index.

`getColumnClass` returns a class object that matches the type of data for each array.

`isCellEditable` returns false, and `setValueAt` does nothing, because our model is not editable.

We then have three possible views of the data: a Swing GUI view that uses a `JTable`, a console view that prints column-aligned data, and an HTML view that produces HTML code to the console (you can copy that and paste it into a file to view in a browser, like in `tablemodel.html`).

Since the `JTable` is the whole reason `TableModel` exists, it knows what to do with the model. The `TableConsole` and `TableHTML` view objects have to explicitly call the appropriate methods in order to display the data.

---

### ❖ 7.6.3. Marker Interfaces

It is actually possible to have an interface that requires no methods at all! This creates what is called a *marker interface*.

A declaration that a class implements the interface makes it an instance of that interface, so that it can be passed as a parameter to a method expecting an instance of the interface, or as a return value from a method that declares it returns an instance of the interface.

An example from the API is `Serializable`

- An object that implements `Serializable` may be turned into a serial data stream, perhaps to save in a file or send across a network connection.
- The `writeObject` method of `ObjectOutputStream` accepts a parameter whose type is `Object`, but throws an exception if it doesn't implement `Serializable`.

- The serialization mechanism is recursive, so not only must the object be an instance of `Serializable`, but any of its object fields must also reference objects that are `Serializable` (or marked as `transient`), and any of their fields ...



## 7.7. Annotations

An annotation is a piece of descriptive data (metadata) about a class, field, or method. It is somewhat like a comment, except that individual annotations are predefined, reusable, and can have effects on either the compilation process or the use of the class once compiled. If you have used an IDE like Eclipse or NetBeans, you may have seen the `@Override` annotation on editor-supplied template code. This particular annotation tells the compiler that the method that immediately follows is meant to override a base class method (or a method required by an interface). If it does not (because perhaps you spelled the name incorrectly, or got the parameter list wrong), then a compiler error is issued.

Annotations provide Java with a means to achieve, at least to some extent, *Aspect-Oriented Programming*, or AOP. AOP recognizes cross-cutting concerns, that is, aspects of an element that cut across classes that might not be related by inheritance or implementation of an interface.

An example is a Java web service. While servlets usually extend a Java EE base class (and will always implement the `Servlet` interface), there is no specified base class or interface for a web service. Instead, configuration information informs the web server that a specific class is intended to be used as a web service, and the server takes steps to make that happen.



## 7.8. Annotation Details

- Annotations are defined as a sort of interface, but an `@` symbol precedes the `interface` keyword in the declaration (as in `@interface`). When used, an `@` symbol is prepended to the name.
- They can be parameterized with *optional elements*.
  - A parameter element named `value` is special - if it is the only element, then it does not need to be named when used (a single value passed to the annotation will be assumed to be the `value`).

- For annotations accepting only one parameter, that parameter should be named `value`.
- An annotation with no parameters serves as a *marker*, much like implementing the `Serializable` interface.
- They are used as modifiers preceding any *target* code entities that are declared: class, field, method, constructor, method parameters, return values, package, or local variables.
- Based on their specified *retention policy*, they can be discarded after compilation (the `SOURCE` policy), or preserved into the compiled class (`CLASS` persists, but the JVM isn't required to keep the information after the class is loaded, and `RUNTIME` annotations do remain with the class in the JVM).
  - `@Override` is an example of the source type, since it is only needed by the compiler
  - A runtime type that you might encounter the effect of is `@Deprecated`, which states that an element is deprecated - you will receive a compiler warning if your code uses an element marked with this annotation (and, since you might be accessing the element in an already-compiled class, this annotation must persist into the compiled class file).
  - `@SuppressWarnings` is another source annotation, with an optional element to specify what types of warnings are to be suppressed.
- The annotation definitions are themselves annotated: `-@Target` and `@Retention` are used before the annotation definition to specify which type of element receives the annotation, and what the retention is.



## 7.9. Using Annotations

To apply an annotation to a class or element, precede the item with the name of the annotation, prefixed with the `@` symbol.

If the annotation takes parameters, supply them in parentheses, as a comma separated list of `parameterName=parameterValue`. If the only parameter is called `value`, then you can just supply the `parameterValue`, without specifying it by name.

### Using an Annotation

```
@AnnotationName(  
    value=parameterValue  
    , parameter2Name=parameter2Value, ...)
```

### Demo 7.8: Java-Interfaces/Demos/AnotherClass.java

---

```
1.  public class AnotherClass {  
2.      public void myMethod() {  
3.          System.out.println("This method is overridden in MyClass");  
4.      }  
5.  }  
6.  
7.  class MyClass extends AnotherClass {  
8.      @Override  
9.      public void myMethod() {  
10.         System.out.println("myMethod overrides the method in AnotherClass");  
11.     }  
12.  
13.     public void myMethod(int i) {  
14.         System.out.println("myMethod is an overload");  
15.     }  
16. }
```

---

### Code Explanation

---

The `@Override` annotation indicates that `myMethod()` in `MyClass` overrides `myMethod()` in `AnotherClass`.

---

### Conclusion

In this lesson, you have learned:

- How to declare, implement, and use interfaces.
- About the basic use of annotations.



# LESSON 8

## Exceptions

---

### Topics Covered

- ☑ Exception handling in Java.
- ☑ Writing try... catch structures to catch expected exceptions.
- ☑ Runtime exception classes vs. other exception classes.
- ☑ finally blocks.
- ☑ Thrown exceptions.
- ☑ Custom exception classes.
- ☑ Initializer blocks.

### Introduction

In this lesson, you will learn about the exception handling mechanism in Java, to write try... catch structures to catch expected exceptions, to distinguish runtime exception classes from other exception classes, to use finally blocks to guarantee execution of code, about thrown exceptions, to define custom exception classes, and about initializer blocks.



## 8.1. Exceptions

*Exceptions* are generated when a recognized condition, usually an error condition, occurs during the execution of a method. There are a number of standard error conditions defined in Java, and you may define your own error conditions as well.

When an exception is generated, it is said to be *thrown*.

Java syntax includes a system for managing exceptions, by tracking the potential for each method to throw specific exceptions. Note that:

- For each method that could throw an exception, your code must inform the Java compiler that it could throw that specific exception.
- The compiler marks that method as potentially throwing that exception, and then requires any code calling the method to handle the possible exception.

There are two ways to handle an exception:

- You can try the “risky” code, catch the exception, and do something about it, after which the propagation of the exception ceases.
- You can mark the method with the risky statement indicating that it throws an exception. Any statement that calls a method capable of throwing an exception must deal with it.

So, if you use a method in your code that is marked as throwing a particular exception, the compiler will not allow that code unless you handle the exception.

Once an exception is thrown, it propagates backward up the chain of methods, from callees to callers, until it is caught. Note that:

- If the exception occurs in a try block, the JVM looks to the catch block(s) that follow to see if any of them match the exception type.
- The first one that matches will be executed.
- If none match, then this method ends, and execution jumps to the method that called this one, at the point where the call was made.

If an exception is not caught in your code (which would happen if `main` was marked as throwing the exception) then the JVM will catch the exception, end that thread of execution, and print a *stack trace*.

There are cases where the compiler does not enforce these rules. Exceptions that fit this category are called *unchecked exceptions*.



## 8.2. Handling Exceptions

Let's say we are writing a method called `getThatInt(ResultSet rs)`. (`ResultSet`, the type of the passed-on parameter, is a built-in Java interface - a “table of data representing a database result set, which is usually generated by executing a statement that queries the database”.) We want to use the method `getInt(int column)` from the `ResultSet` passed in as a parameter:

```
public int getThatInt(ResultSet rs) {
    int i = 0;
    return rs.getInt(3);
}
```

A look at the API listing for `ResultSet` tells us that the `getInt()` ([https://docs.oracle.com/javase/10/docs/api/java/sql/ResultSet.html#getInt\(int\)](https://docs.oracle.com/javase/10/docs/api/java/sql/ResultSet.html#getInt(int))) method throws `SQLException`, so we must handle that in our code, in one of the two following ways:

1. Use try and catch

```
public int getThatInt(ResultSet rs) {
    int i = 0;
    try {
        i = rs.getInt(3);
    }
    catch (SQLException e) {
        System.out.println("Exception occurred!");
        System.out.println(e.getMessage());
        e.printStackTrace();
    }
    return i;
}
```

Evaluation  
Copy

2. Declare that the method will throw the exception and let our caller handle it

```
public int getThatInt(ResultSet rs) throws SQLException {
    int i = 0;
    i = rs.getInt(3);
    return i;
}
```

Note that although you are required to “handle” the exception, you aren’t necessarily required to do anything useful about it!

Your decision as to which approach to use should be based on where you think responsibility for handling the exception lies. In the example above, the second approach is probably better, so that the code that works more closely with the SQL handles the exception.



## 8.3. Exception Objects

When an exception is thrown, an exception object is created and passed to the catch block much like a parameter to a method. Note that:

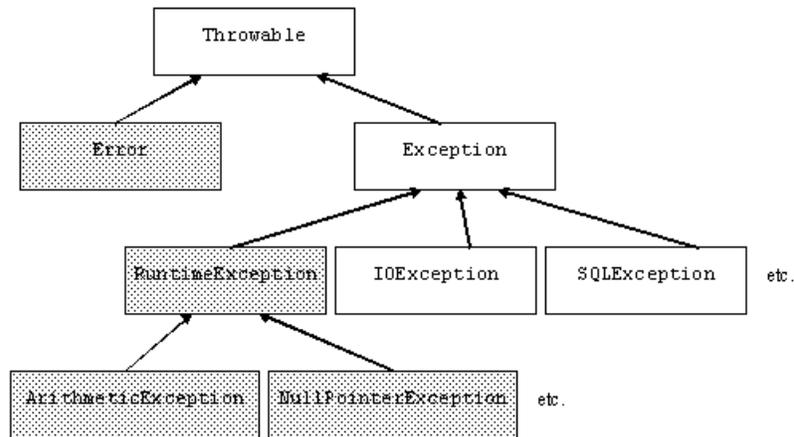
- The occurrence of an exception generates an object (an instance of a class in the exception hierarchy) containing information about the exception.
- The exception object is passed to the code designated to catch the exception, which may then use methods of the exception object to help handle the situation.

There is an API class called `Exception`. Note that:

- All exception classes inherit from `Exception`, which itself inherits from `Throwable`.
- Another class, `Error`, also inherits from `Throwable`.
- Your code must handle most exceptions, but generally should not attempt to handle `Error` subtypes (like `OutOfMemoryError` or `StackOverflowError`).
- `RuntimeException` is a subclass of `Exception` that is a base class for all the exception classes that you are not obligated to handle, but still might want to anyway (examples are `ArithmeticException`, from dividing by zero, `NullPointerException`, and `ArrayIndexOutOfBoundsException`).

So, there are several classes of exceptions you are not required to handle (shaded in the image below). Note that:

- These extend either `Error` or `RuntimeException`.
- The ones you are required to handle are called *checked exceptions*.
- Generally, runtime exceptions can be prevented by good coding practices:
  - Avoid null pointer exceptions by checking the reference first.
  - Check array indexes before using them to avoid `ArrayIndexOutOfBoundsException`.
  - Looking at the documentation for allowable parameter values and testing them before passing them to a method will prevent `IllegalArgumentException`.



## 8.4. Attempting Risky Code - try and catch

If a method is going to resolve a potential exception internally, the line of code that could generate the exception is placed inside a try block.

- There may be other code inside the try block, before and/or after the risky line(s). Any code that depends upon the risky code's success should be in the try block, since it will automatically be skipped if the exception occurs.

### try Block Structure

```

try {
    code
        risky code
        code that depends on the risky code succeeding
}
  
```

There is usually at least one catch block immediately after the try block. A catch block must specify what type of exception it will catch.

### **catch Block Structure**

```
catch (ExceptionClassName exceptionObjectName) {  
    code using methods from exceptionObjectName  
}
```

- There can be more than one catch block, each one marked for a specific exception class.
- The exception class that is caught can be any class in the exception hierarchy, either a general (base) class, or a very specific (derived) class.
- The catch block(s) must handle all checked exceptions that the try block is known to throw unless you want to throw that exception back to the method that called this one.
- It is possible to have a try block without any catch blocks if you have a finally block, but any checked exceptions still need to be caught, or the method needs to declare that it throws them. We will cover finally later in this section.

If an exception occurs within a try block, execution jumps to the first catch block whose exception class matches the exception that occurred (using an instanceof test). Any steps remaining in the try block are skipped.

If no exception occurs, then the catch blocks are skipped. The catch blocks will also be skipped if an exception that is not caught occurs, such as a RuntimeException, or an exception that the method declared it throws.

You cannot catch an exception that would not occur in the try block, but you can mark a method as throwing an exception that it doesn't (this leaves open the possibility that an extending class can override the method and actually throw the exception).

### **❖ 8.4.1. try ... catch Blocks and Variable Scope/Initialization**

If you declare a variable within a try block, it will not exist outside the try block, since the curly braces define the scope of the variable. You will often need that variable later, if nowhere else other than the catch or finally blocks, so you would need to declare the variable before the try.

If you declare but don't initialize a variable before a try block, and the only place you set a value for that variable is in the try block, then it is possible when execution leaves the try ... catch structure that the variable never received a value:

- You would get a “possibly uninitialized value” error message from the compiler, since it actually keeps track of that sort of thing.
- Usually this happens with object references; you would generally initialize them to null.

### Demo 8.1: Java-Exceptions/Demos/ExceptionTest.java

---

```
1. public class ExceptionTest {
2.     public static void main(String[] args) {
3.         int i, j, x = 5, y = 5, z = 0;
4.         try {
5.             i = x/y;
6.             System.out.println("x/y = " + i);
7.             j = x/z;
8.             System.out.println("x/z = " + j);
9.         }
10.        catch(ArithmeticException e) {
11.            System.out.println("Arithmetic Exception!");
12.        }
13.        System.out.println("Done with test");
14.    }
15. }
```

---

#### Code Explanation

*Evaluation Copy*

The program will print the first result, then fail while performing the division for the second equation. Execution will jump to the catch block to print our message on the screen.

Note: `ArithmeticException` is one of the few you are not required to catch, but you can still catch it if you wish.

---

### ❖ 8.4.2. Example - An Exception You Must Handle

The preceding example used a `RuntimeException` which your code is not obligated to handle.

Most methods in the I/O classes throw `IOException` which is an exception you must handle.

Our `KeyboardReader` class has `try` and `catch` to handle this, essentially stifling the exception, since it is unlikely, if not impossible, to actually get an `IOException` from the keyboard.

## Demo 8.2: Java-Exceptions/Demos/IOExceptionTest.java

---

```
1.  import java.io.IOException;
2.
3.  public class IOExceptionTest {
4.
5.      public static void main(String[] args) {
6.
7.          int num = 0;
8.
9.          num = System.in.read(); // comment out this line
10.
11.         try {
12.             num = System.in.read();
13.             System.out.println("You entered " + (char) num);
14.         }
15.         catch (IOException e) {
16.             System.out.println("IO Exception occurred");
17.         }
18.     }
19. }
```

---

### Code Explanation

The line marked to comment out throws `IOException`, but is not in a `try` block, so the compiler rejects it. The second read attempt is within a `try` block, as it should be.

- Try to compile this code as-is and then comment out the indicated line.
  - There is no way we can force an `IOException` from the keyboard to test the catch block.
- 

### ❖ 8.4.3. Using Multiple catch Blocks

It is possible that a statement might throw more than one kind of exception. You can list a sequence of catch blocks, one for each possible exception. Remember that there is an object hierarchy for exceptions. Since the first one that matches is used and the others skipped, you can put a derived class first and its base class later (you will actually get a compiler error if you list a more basic class before a derived class, as it is “unreachable code”).

## Demo 8.3: Java-Exceptions/Demos/MultiCatchTest.java

---

```
1.  import util.KeyboardReader;
2.
3.  public class MultiCatchTest {
4.      public static void main(String[] args) {
5.          int num1, num2;
6.          try {
7.              num1 = KeyboardReader.getPromptedInt("Enter a number: ");
8.              num2 = KeyboardReader.getPromptedInt("Enter another number: ");
9.              System.out.println(num1 + " / " + num2 + " = " + num1/num2);
10.         }
11.         catch (NumberFormatException e) {
12.             System.out.println("Number Format Exception occurred");
13.         }
14.         catch (ArithmeticException e) {
15.             System.out.println("Divide by Exception occurred");
16.         }
17.         catch (Exception e) {
18.             System.out.println("General Exception occurred");
19.         }
20.     }
21. }
```

---

Evaluation  
Copy

### Code Explanation

---

The code in the `try` block could throw `NumberFormatException` during the parsing (if the user entered “d” instead of a number, for example), and `ArithmeticException` while doing the division (if the user entered “0” for `num2`), so we have catch blocks for those specific cases. The more generic catch block for `Exception` would catch other problems, like `NullPointerException`.

---



## 8.5. Guaranteeing Execution of Code - The `finally` Block

To guarantee that a line of code runs, whether an exception occurs or not, use a `finally` block after the `try ... catch` blocks.

The code in the `finally` block will *almost always* execute.

- If an exception causes a catch block to execute, the finally block will be executed after the catch block.
- If an uncaught exception occurs, the finally block executes, and then execution exits this method and the exception is thrown to the method that called this method.
- If either the try block or a catch block executes a return statement, the finally block executes before we leave the method.
- If either the try block or a catch block calls System.exit, the finally block will not execute.
- For the sake of completeness, if a finally block executes a return while an uncaught exception is pending, the exception is stifled; that is, it just disappears.

#### Complete try ... catch ... finally Structure

```
try {  
    risky code block  
}  
catch (ExceptionClassName exceptionObjectName) {  
    code to resolve problem  
}  
finally {  
    code that will always execute  
}
```

Evaluation  
Copy

In summary, note the following:

- A try block is followed by zero or more catch blocks.
  - If the catch block exception classes caught are related, the blocks must be listed in inheritance order from most derived to most basic.
- There may one finally block as the last block in the structure.
- There must be at least one block from the combined set of catch and finally after the try.

It's possible to have a try block followed by a finally block, with no catch block. This is used to prevent an unchecked exception, or an exception the method declared it throws, from exiting the method before cleanup code can be executed.

## Demo 8.4: Java-Exceptions/Demos/FinallyTest.java

---

```
1.  import util.KeyboardReader;
2.
3.  public class FinallyTest {
4.      public static void main(String[] args) {
5.          System.out.println("Returned value is " + go());
6.      }
7.
8.      public static int go() {
9.          int choice = 0;
10.         try {
11.             String name = KeyboardReader.getPromptedString("Enter your name: ");
12.             System.out.println("MENU:");
13.             System.out.println("1 - normal execution");
14.             System.out.println("2 - uncaught ArithmeticException");
15.             System.out.println("3 - return from try block");
16.             System.out.println("4 - call System.exit");
17.             System.out.println(
18.                 "5 - return 5 from finally after ArithmeticException");
19.             System.out.println(
20.                 "6 - return 6 from finally after try returns -1");
21.             System.out.println("X - catch NumberFormatException");
22.             choice = KeyboardReader.getPromptedInt("Enter your choice: ");
23.
24.             if (choice == 1) System.out.println("Hello " + name);
25.             else if (choice == 2) System.out.println("1 / 0 = " + 1/0);
26.             else if (choice == 3) return 3;
27.             else if (choice == 4) System.exit(1);
28.             else if (choice == 5) System.out.println("1 / 0 = " + 1/0);
29.             else if (choice == 6) return -1;
30.         }
31.         catch (NumberFormatException e) {
32.             System.out.println("Number Format Exception occurred");
33.         }
34.         finally {
35.             System.out.println("Goodbye from finally block");
36.             if (choice == 5) return 5;
37.             if (choice == 6) return 6;
38.         }
39.         return 0;
40.     }
41. }
```

---

## Code Explanation

---

The program shows a menu of possible execution paths you can trigger. The “Goodbye from finally block” message will always appear except from an explicit call to `System.exit` in the `try` block:

- When no exception occurs, the `finally` block will execute after the `try`.
- If an uncaught exceptions occurs, like when we divide by zero, the `finally` block will execute before we are thrown out of the method.
- If we execute a `return` from the `try` block, the `finally` block still executes before we leave.
- Catching a `NumberFormatException` is still followed by executing the `finally` block.
- But, calling `System.exit` in the `try` block causes the JVM to shut down without executing the `finally` block.
- When we force an uncaught exception but return from the `finally` block, we do not get a stack trace, indicating that the `ArithmeticException` just disappeared (compare choices 2 and 5).
- When both the `try` block and the `finally` block execute a `return` statement, the value from the `finally` block is the one actually returned.

EVALUATION COPY



## 8.6. Letting an Exception be Thrown to the Method Caller

A method that generates an exception can be written to not catch it. Instead it can let it be thrown back to the method that called it.

The possibility that a method may throw an exception must be defined with the method.

### Declaring a Method to Throw an Exception

```
[modifiers]
returnType
functionName(arguments)
throws ExceptionClassName {
    body including risky code
}
```

In this case, an instance of `ExceptionClassName` or a class that extends it may be thrown. Stating that a method throws `Exception` is about as generic as you can get. (Stating that it throws `Throwable` is as generic as you can get, but not recommended). A method can throw more than one type of exception; in these cases, you would use a comma-separated list of exception types.

In this way, the method is now marked as throwing that type of exception, and a code that calls this method will be obligated to handle it.

When you extend a class and override a method, you cannot add exceptions to the `throws` list, but a base class method can list exceptions that it does not throw in the expectation that an overriding method will throw the exception. This is another example of the “inheritance cannot restrict access” principle we saw earlier.

If `main()` throws an exception then the JVM, which runs under Java rules, will handle the exception by printing a *stack trace* and closing down the offending thread. In a single-threaded program, this will shut down the JVM.

---

Evaluation  
Copy

## 8.7. Throwing an Exception

The keyword `throw` is used to trigger the exception-handling process (or, “raise” the exception, as it is often termed in other languages).

That word is followed by an instance of a throwable object - an instance of a class that extends `Throwable`. Usually, a new instance of an appropriate exception class is created to contain information about the exception.

For example, suppose a `setAge()` method expects a nonnegative integer; we can have it throw an `IllegalArgumentException` if it receives a negative value. It makes sense for the method that calls `setAge()` to do something about the problem, since it is where the illegal number came from.

So, we can declare `setAge()` as `throws IllegalArgumentException`.

```
public void setAge(int age) throws IllegalArgumentException {
    if (age < 0)
        throw new IllegalArgumentException("Age must be >= 0");
    else
        this.age = age;
}
```

Evaluation  
Copy

# Exercise 26: Payroll-Exceptions01: Handling NumberFormatException in Payroll

 5 to 10 minutes

---

Our application currently does not handle bad numeric data that might be entered by the user. The parsing methods, e.g., `parseInt`, all throw `NumberFormatException` in the event bad data is passed to the method.

We need to fix this potential problem!

The `NumberFormatException` should be caught and the user should then be prompted to re-enter the numeric data, e.g., a department number.

You will not write code in this exercise. Rather, you will think about where the try/catch for `NumberFormatException` should be placed.

1. Open the files on `Java-Exceptions/Exercises/Payroll-Exceptions01/`.
2. Where would you put the code to catch the `NumberFormatException`? In the main method of `Payroll.java` or in the methods of `KeyboardReader.java`?

## Solution

---

Placing the code in the `main` of our `Payroll.java` program places the responsibility of catching bad data on the client application. Therefore the programmer would have to place any code that requests numeric input in a loop that encapsulates a `try/catch` block.

A good approach is to overload the `get` methods in `KeyboardReader.java` to accept an error message string and display the message string to the user in the event of a `NumberFormatException`. Furthermore, we can place the prompt in a loop that re-prompts the user until the data is numeric. The original method would still be available if the programmer wanted to customize the exception handling.

We will use this approach to catch a potential `NumberFormatException`.

## Exercise 27: Payroll-Exceptions01, continued

 15 to 20 minutes

---

1. Go ahead and add a second version of each get numeric method in `KeyboardReader` that accepts an error message string as a second parameter; have it loop on each numeric input request until it succeeds without throwing the exception (and printing the error message each time the exception occurs).
2. Then modify `Payroll` to call these methods.

### Challenge

This approach still doesn't solve the problem of valid department numbers. For example, the department number must be in the range of 1 through 5, inclusive. Can you think of an approach that would solve this problem? (Hint: interfaces are a powerful tool ...).

## Solution:

### Java-Exceptions/Solutions/Payroll-Exceptions01/util/KeyboardReader.java

```
1. package util;
2. import java.io.*;
3.
4. public class KeyboardReader {
5.
6.     private static BufferedReader in =
7.         new BufferedReader(new InputStreamReader(System.in));
8.
9.     public KeyboardReader() {
10.    }
11.
12.     public static String getPromptedString(String prompt) {
13. String response = null;
14.     System.out.print(prompt);
15.     try {
16.         response = in.readLine();
17.     } catch (IOException e) {
18.         System.out.println("IOException occurred");
19.     }
20.     return response;
21. }
22.
23.     public static char getPromptedChar(String prompt) {
24.         return getPromptedString(prompt).charAt(0);
25.     }
26.
27.     public static int getPromptedInt(String prompt) {
28.         return Integer.parseInt(getPromptedString(prompt));
29.     }
30.
31.     public static float getPromptedFloat(String prompt) {
32.         return Float.parseFloat(getPromptedString(prompt));
33.     }
34.     public static double getPromptedDouble(String prompt) {
35.         return Double.parseDouble(getPromptedString(prompt));
36.     }
37.
38.     public static int getPromptedInt(String prompt, String errMsg) {
39.         for ( ; ; ) {
40.             try {
41.                 return Integer.parseInt(getPromptedString(prompt));
42.             } catch (NumberFormatException nfe) {
43.                 System.out.println(errMsg);
```

```
44.     }
45.   }
46. }
47.
48. public static float getPromptedFloat(String prompt, String errMsg) {
49.     for( ; ; ) {
50.         try {
51.             return Float.parseFloat(getPromptedString(prompt));
52.         } catch (NumberFormatException nfe) {
53.             System.out.println(errMsg);
54.         }
55.     }
56. }
57.
58. public static double getPromptedDouble(String prompt, String errMsg) {
59.     for( ; ; ) {
60.         try {
61.             return Double.parseDouble(getPromptedString(prompt));
62.         } catch (NumberFormatException nfe) {
63.             System.out.println(errMsg);
64.         }
65.     }
66. }
67. }
```

---

## Code Explanation

---

We added overloaded versions of the `getPromptedXXX` numeric methods, allowing the calling code to optionally specify an error message to display and using a `try/catch` block to ensure that the requested type of input is entered.

---

## Solution: Java-Exceptions/Solutions/Payroll-Exceptions01/Payroll.java

```
1.  import employees.*;
2.  import vendors.*;
3.  import finance.*;
4.  import util.*;
5.
6.  public class Payroll {
7.      public static void main(String[] args) {
8.          Employee[] e = new Employee[5];
9.          String fName = null;
10.         String lName = null;
11.         int dept = 0;
12.         double payRate = 0.0;
13.         double hours = 0.0;
14.
15.         for (int i = 0; i < e.length; i++) {
16.             char type = KeyboardReader.getPromptedChar(
17.                 "Enter type: E, N, or C: ");
18.             if (type != 'e' && type != 'E' &&
19.                 type != 'n' && type != 'N' &&
20.                 type != 'c' && type != 'C') {
21.                 System.out.println("Please enter a valid type");
22.                 i--;
23.                 continue;
24.             }
25.             fName = KeyboardReader.getPromptedString("Enter first name: ");
26.             lName = KeyboardReader.getPromptedString("Enter last name: ");
27.             dept = KeyboardReader.getPromptedInt(
28.                 "Enter department: ", "Department must be numeric");
29.             do {
30.                 payRate = KeyboardReader.getPromptedDouble(
31.                     "Enter pay rate: ", "Pay rate must be numeric");
32.                 if (payRate < 0.0) System.out.println("Pay rate must be >= 0");
33.             } while (payRate < 0.0);
34.
35.             switch (type) {
36.                 case 'e':
37.                     e[i] = new ExemptEmployee(fName, lName, dept, payRate);
38.                     break;
39.                 case 'n':
40.                     case 'N': do {
41.                         hours = KeyboardReader.getPromptedDouble(
42.                             "Enter hours: ", "Hours must be numeric");
43.                         if (hours < 0.0)
44.                             System.out.println("Hours must be >= 0");
```

```

45.         } while (hours < 0.0);
46.         e[i] = new NonexemptEmployee(
47.             fName, lName, dept, payRate, hours);
48.         break;
49.     case 'c':
50.     case 'C': do {
51.         hours = KeyboardReader.getPromptedDouble(
52.             "Enter hours: ", "Hours must be numeric");
53.         if (hours < 0.0)
54.             System.out.println("Hours must be >= 0");
55.         } while (hours < 0.0);
56.         e[i] = new ContractEmployee(
57.             fName, lName, dept, payRate, hours);
58.     }
59.
60.     System.out.println(e[i].getPayInfo());
61. }
62.
63. System.out.println();
64. System.out.println("Exempt Employees");
65. System.out.println("=====");
66. for (Employee emp : e) {
67.     if (emp instanceof ExemptEmployee) {
68.         System.out.println(emp.getPayInfo());
69.     }
70. }
71. System.out.println();
72. System.out.println("Nonexempt Employees");
73. System.out.println("=====");
74. for (Employee emp : e) {
75.     if (emp instanceof NonexemptEmployee) {
76.         System.out.println(emp.getPayInfo());
77.     }
78. }
79. System.out.println();
80. System.out.println("Contract Employees");
81. System.out.println("=====");
82. for (Employee emp : e) {
83.     if (emp instanceof ContractEmployee) {
84.         System.out.println(emp.getPayInfo());
85.     }
86. }
87.
88. Invoice[] inv = new Invoice[4];
89. inv[0] = new Invoice("ABC Co.", 456.78);

```

```
90.     inv[1] = new Invoice("XYZ Co.", 1234.56);
91.     inv[2] = new Invoice("Hello, Inc.", 999.99);
92.     inv[3] = new Invoice("World, Ltd.", 0.43);
93.
94.     CheckPrinter.printChecks(e);
95.     CheckPrinter.printChecks(inv);
96. }
97.
98. }
```

---

## Code Explanation

---

The revised code uses the new overloads of the `getPromptedXXX` methods.

---

## Challenge Solution:

**Java-Exceptions/Solutions/Payroll-Exceptions01-challenge/util/IntValidator.java**

---

```
1.  package util;
2.
3.  public interface IntValidator {
4.      public boolean accept(int candidate);
5.  }
```

---

## Code Explanation

---

This interface specifies a method that will be used to validate integers. A validator for a specific field (like department) would implement this with code to test for legal values for that field. The package contains similar interfaces for floats and doubles.

---

## Challenge Solution:

Java-Exceptions/Solutions/Payroll-Exceptions01-challenge/employees/DeptValidator.java

```
1. package employees;
2. import util.IntValidator;
3.
4. public class DeptValidator implements IntValidator {
5.
6.     @Override
7.     public boolean accept(int dept) {
8.         return dept > 0 && dept <= 5;
9.     }
10.
11. }
12. /* Sample usage in Payroll
13.
14. dept = KeyboardReader.getPromptedInt(
15.     "Enter department: ", "Dept must be numeric",
16.     new DeptValidator(), "Valid depts are 1 - 5");
17.
18. */
```

---

### Code Explanation

---

This class implements the `accept` method of the `IntValidator` interface in order to ensure department numbers are in the range of 1 through 5 inclusive.

---

## Challenge Solution:

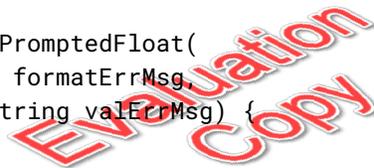
### Java-Exceptions/Solutions/Payroll-Exceptions01-challenge/util/KeyboardReader.java

```
1. package util;
2. import java.io.*;
3.
4. public class KeyboardReader {
5.
6.     private static BufferedReader in =
7.         new BufferedReader(new InputStreamReader(System.in));
8.
9.     public KeyboardReader() { }
10.
11.     public static String getPromptedString(String prompt) {
12.         String response = null;
13.         System.out.print(prompt);
14.         try {
15.             response = in.readLine();
16.         } catch (IOException e) {
17.             System.out.println("IOException occurred");
18.         }
19.         return response;
20.     }
21.
22.     public static char getPromptedChar(String prompt) {
23.         return getPromptedString(prompt).charAt(0);
24.     }
25.
26.     public static int getPromptedInt(String prompt) {
27.         return Integer.parseInt(getPromptedString(prompt));
28.     }
29.
30.     public static int getPromptedInt(String prompt, String errMsg) {
31.         for ( ; ; ) {
32.             try {
33.                 return Integer.parseInt(getPromptedString(prompt));
34.             } catch (NumberFormatException e) {
35.                 System.out.println(errMsg);
36.             }
37.         }
38.     }
39.     public static int getPromptedInt(
40.         String prompt, String formatErrMsg,
41.         IntValidator val, String valErrMsg) {
42.         for ( ; ; ) {
43.             try {
```

```

44.     int num = Integer.parseInt(getPromptedString(prompt));
45.     if (val.accept(num)) return num;
46.     else System.out.println(valErrMsg);
47.     } catch (NumberFormatException e) {
48.         System.out.println(formatErrMsg);
49.     }
50. }
51. }
52.
53. public static float getPromptedFloat(String prompt) {
54.     return Float.parseFloat(getPromptedString(prompt));
55. }
56. public static float getPromptedFloat(String prompt, String errMsg) {
57.     for ( ; ; ) {
58.         try {
59.             return Float.parseFloat(getPromptedString(prompt));
60.         } catch (NumberFormatException e) {
61.             System.out.println(errMsg);
62.         }
63.     }
64. }
65. public static float getPromptedFloat(
66.     String prompt, String formatErrMsg,
67.     FloatValidator val, String valErrMsg) {
68.     for ( ; ; ) {
69.         try {
70.             float num = Float.parseFloat(getPromptedString(prompt));
71.             if (val.accept(num)) return num;
72.             else System.out.println(valErrMsg);
73.         } catch (NumberFormatException e) {
74.             System.out.println(formatErrMsg);
75.         }
76.     }
77. }
78.
79. public static double getPromptedDouble(String prompt) {
80.     return Double.parseDouble(getPromptedString(prompt));
81. }
82. public static double getPromptedDouble(String prompt, String errMsg) {
83.     for ( ; ; ) {
84.         try {
85.             return Double.parseDouble(getPromptedString(prompt));
86.         } catch (NumberFormatException e) {
87.             System.out.println(errMsg);
88.         }

```



```
89.     }
90.     }
91.     public static double getPromptedDouble(
92.         String prompt, String formatErrMsg,
93.         DoubleValidator val, String valErrMsg) {
94.         for ( ; ; ) {
95.             try {
96.                 double num = Double.parseDouble(getPromptedString(prompt));
97.                 if (val.accept(num)) return num;
98.                 else System.out.println(valErrMsg);
99.             } catch (NumberFormatException e) {
100.                System.out.println(formatErrMsg);
101.            }
102.        }
103.    }
104. }
```

---

## Code Explanation

---

Here is the modified `KeyboardReader` that uses the validator interfaces.

---

*Evaluation  
\*  
Copy*

---

## 8.8. Exceptions and Inheritance

If a base class method throws an exception, that behavior will also occur in any derived classes that do not override the method.

An overriding method may throw the same exception(s) that the base class method threw.

An overriding method cannot add new exceptions to the `throws` list. Similar to placing more strict access on the method, this would restrict the derived class object in ways that a base class reference would be unaware of.

If the derived class method does not throw the exception that the base class threw, it can either:

1. Retain the exception in the `throws` list, even though it does not throw it; this would enable subclasses to throw the exception.
2. Remove the exception from its `throws` list, thus blocking subsequent extensions from throwing that exception.

If you have a base class method that does not throw an exception, but you expect that subclasses might, you can declare the base class to throw that exception.

### ❖ 8.8.1. Exception Class Constructors and Methods

There are several forms of constructors defined in the base class for the exception hierarchy.

| Constructor                                             | Description                                                                                                                                                                                                     |
|---------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Throwable()</code>                                | Constructs a new <code>Throwable</code> with <code>null</code> as its detail message.                                                                                                                           |
| <code>Throwable(String message)</code>                  | Constructs a new <code>Throwable</code> with the specified detail message.                                                                                                                                      |
| <code>Throwable(String message, Throwable cause)</code> | Constructs a new <code>Throwable</code> with the specified detail message and cause.                                                                                                                            |
| <code>Throwable(Throwable cause)</code>                 | Constructs a new <code>Throwable</code> with the specified cause and a detail message of <code>(cause==null ? null : cause.toString())</code> (which typically contains the class and detail message of cause). |

The forms involving a cause are used in situations like Servlets and Java Server Pages, where a specific exception is thrown by the JSP engine, but it may be rooted in an exception from your code.

- In both cases, a method in a base class is overridden by your code since the writers of the base class did not know what specific exceptions your code might throw, and didn't want to specify something too broad like `throws Exception`, they settled on `throws IOException`, `ServletException` (or `JSPException` for Java Server Pages).
- You would try and catch for your expected exceptions and repackage them inside `ServletException` objects if you did not want to handle them

A Exception object has several useful methods:

| Method                                           | Description                                                                                                                                                                                                                                                                                                             |
|--------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>getMessage()</code>                        | Prints the message that was associated with the exception (many of the exceptions that deal with outside resources pass on the message from the outside) - for example, when you connect to a database and run a query, that could generate an error in the database; <code>getMessage()</code> will show that message. |
| <code>printStackTrace()</code>                   | Prints to the standard error stream the trace of what function called what function, etc., leading up to the exception. There are variations of this method where you may specify a destination for the printing (note that stack trace includes the message).                                                          |
| <code>printStackTrace(PrintStream stream)</code> | Same as above, but prints to the specified output stream (which could be hooked to a log file, for example).                                                                                                                                                                                                            |

Also worth noting is that the Java Logging API has logging methods that will accept a `Throwable` parameter and make a log entry with the stack trace.

Evaluation  
Copy

## 8.9. Creating and Using Your Own Exception Classes

You can create your own exception class by extending an existing exception class.

### Declaring an Exception Class

```
[modifiers] class NewExceptionClassName extends ExceptionClassName {  
    create constructors that usually delegate to super-constructors  
}
```

You could then add any fields or methods that you wish, although often that is not necessary.

You must, however, override any constructors you wish to use: `Exception()`, `Exception(String message)`, `Exception(String message, Throwable cause)`, `Exception(Throwable cause)`. Usually you can just call the corresponding super-constructor.

If you extend `RuntimeException` or one of its subclasses, your exception will be treated as a runtime exception (it will not be checked).

When a situation arises for which you would want to throw the exception, use the `throw` keyword with a new object from your exception class, for example:

### Throwing an Exception

```
throw new ExceptionClassName(messageString);
```

## **Demo 8.5: Java-Exceptions/Demos/NewExceptionTest.java**

---

```
1.  class NewException extends Exception {
2.      NewException() {
3.          super();
4.      }
5.      NewException(String message) {
6.          super(message);
7.      }
8.      NewException(String message, Throwable cause) {
9.          super(message, cause);
10.     }
11.     NewException(Throwable cause) {
12.         super(cause);
13.     }
14. }
15. public class NewExceptionTest {
16.     public void thrower() throws NewException {
17.         if (Math.random() < 0.5) {
18.             throw new NewException("This is my exception");
19.         }
20.     }
21.     public static void main(String[] args) {
22.         NewExceptionTest t = new NewExceptionTest();
23.         try {
24.             t.thrower();
25.         }
26.         catch(NewException e) {
27.             System.out.println("New Exception: " + e.getMessage());
28.         }
29.         finally {
30.             System.out.println("Done");
31.         }
32.     }
33. }
```



## Code Explanation

---

The `thrower` method randomly throws a `NewException`, by creating and throwing a new instance of `NewException`.

`main` tries to call `thrower`, and catches the `NewException` when it occurs.

---

## Exercise 28: Payroll-Exceptions02

 20 to 30 minutes

---

Our payroll program can now handle bad numeric input for pay rate and department number, as well as invalid data. There is no guarantee that later code will remember to do this. We can now leverage encapsulation by checking the data in a setter method. If the data is invalid, we can throw a custom exception that must be caught by the program creating Employee objects.

1. Open the files in `Java-Exceptions/Exercises/Payroll-Exceptions02/`.
2. In the `util` package, create an exception class for `InvalidValueException`. Note that the Java API already contains a class for this purpose, `IllegalArgumentException`, but it is a `RuntimeException` - we would like ours to be a checked exception.
3. In `Employee.java` modify the setter method for pay rate to throw the `InvalidValueException` if the data fails validation.
4. In `Employee.java` and its subclasses, change the constructors that call `setPayRate` to throw that exception.
5. In `Payroll.java`, encapsulate the code in the scope of the `for` loop in a `try..catch` block.
6. The solution uses the validators from the previous challenge exercise; that code is included in the files in `Java-Exceptions/Exercises/Payroll-Exceptions02/`.

## Solution:

### Java-Exceptions/Solutions/Payroll-Exceptions02/util/InvalidValueException.java

---

```
1. package util;
2. public class InvalidValueException extends Exception {
3.
4.     public InvalidValueException() { super(); }
5.     public InvalidValueException(String message) { super(message); }
6.     public InvalidValueException(Throwable cause) { super(cause); }
7.     public InvalidValueException(String message, Throwable cause) {
8.         super(message, cause);
9.     }
10.
11. }
```

---

## Solution:

### Java-Exceptions/Solutions/Payroll-Exceptions02/employees/Employee.java

```
1. package employees;
2. import finance.Payable;
3. import util.*;
4.
5. public class Employee extends Person implements Payable {
6.     private static int nextId = 1;
7.     private int id = nextId++;
8.     private int dept;
9.     private double payRate;
10.
11.     public Employee() {
12.     }
13.     public Employee(String firstName, String lastName) {
14.         super(firstName, lastName);
15.     }
16.     public Employee(String firstName,String lastName, int dept) {
17.         super(firstName, lastName);
18.         setDept(dept);
19.     }
20.     public Employee(String firstName, String lastName, double payRate) throws In
        validValueException {
21.         super(firstName, lastName);
22.         setPayRate(payRate);
23.     }
24.     public Employee(String firstName, String lastName, int dept, double payRate)
        throws InvalidValueException {
25.         this(firstName, lastName, dept);
26.         setPayRate(payRate);
27.     }
28.
29.     public static int getNextId() {
30.         return nextId;
31.     }
32.
33.     public static void setNextId(int nextId) {
34.         Employee.nextId = nextId;
35.     }
36.
37.     public int getId() { return id; }
38.
39.     public int getDept() { return dept; }
40.
41.     public void setDept(int dept) {
```

```
42.     this.dept = dept;
43. }
44.
45. public double getPayRate() { return payRate; }
46.
47. public void setPayRate(double payRate) throws InvalidValueException {
48.     DoubleValidator val = new PayRateValidator();
49.     if (!val.accept(payRate))
50.         throw new InvalidValueException(payRate + " is an invalid pay rate");
51.     this.payRate = payRate;
52. }
53.
54. public String getPayInfo() {
55.     return "Employee " + id + " dept " + dept + " " +
56.         getFullName() + " paid " + payRate;
57. }
58. }
```

---

## Code Explanation

---

The declaration that `setPayRate` throws `InvalidValueException` requires that a constructor that calls `setPayRate` must throw (or catch) that exception.

---

## Solution:

### Java-Exceptions/Solutions/Payroll-Exceptions02/employees/ExemptEmployee.java

---

```
1. package employees;
2.
3. import util.*;
4.
5. public class ExemptEmployee extends Employee {
6.
7.     public ExemptEmployee() {
8.     }
9.     public ExemptEmployee(String firstName, String lastName) {
10.        super(firstName, lastName);
11.    }
12.    public ExemptEmployee(String firstName,String lastName, int dept) {
13.        super(firstName, lastName, dept);
14.    }
15.    public ExemptEmployee(String firstName, String lastName, double payRate) throws
        InvalidValueException {
16.        super(firstName, lastName, payRate);
17.    }
18.    public ExemptEmployee(String firstName, String lastName, int dept, double
        payRate) throws InvalidValueException {
19.        super(firstName, lastName, dept, payRate);
20.    }
21.    public String getPayInfo() {
22.        return "Exempt Employee " + getId() + " dept " + getDept() +
23.            " " + getFirstName() + " " + getLastName() +
24.            " paid " + getPayRate();
25.    }
26. }
```

---

## Code Explanation

---

Calling a constructor in the super class that might throw our exception requires that the constructor in a subclass must throw (or catch) that exception. The other classes, not shown, should be similarly updated.

---

## Solution: Java-Exceptions/Solutions/Payroll-Exceptions02/Payroll.java

```
1.  import employees.*;
2.  import vendors.*;
3.  import util.*;
4.  import finance.*;
5.
6.  public class Payroll {
7.      public static void main (String[] args) {
8.          Employee[] e = new Employee[5];
9.          String fName = null;
10.         String lName = null;
11.         int dept = 0;
12.         double payRate = 0.0;
13.         double hours = 0.0;
14.
15.         for (int i = 0; i < e.length; i++) {
16.             try {
17.                 char type = KeyboardReader.getPromptedChar ("Enter type: E, N, or C: ");
18.                 if (type != 'e' && type != 'E' && type != 'n' && type != 'N' && type != 'c'
19.                     && type != 'C') {
20.                     System.out.println ("Please enter a valid type");
21.                     i--;
22.                     continue;
23.                 }
24.                 fName = KeyboardReader.getPromptedString ("Enter first name: ");
25.                 lName = KeyboardReader.getPromptedString ("Enter last name: ");
26.                 dept = KeyboardReader.getPromptedInt ("Enter department: ",
27.                 "Department must be numeric", new DeptValidator (),
28.                 "Valid departments are 1 - 5");
29.                 payRate = KeyboardReader.getPromptedDouble (
30.                 "Enter pay rate: ", "Pay rate must be numeric",
31.                 new PayRateValidator(), "Pay rate must be >= 0");
32.
33.                 switch (type) {
34.                     case 'e':
35.                     case 'E':
36.                         e[i] = new ExemptEmployee (fName, lName, dept, payRate);
37.                         break;
38.                     case 'n':
39.                     case 'N':
40.                         do {
41.                             hours = KeyboardReader.getPromptedDouble (
42.                             "Enter hours: ", "Hours must be numeric");
43.                             if (hours < 0.0)
44.                                 System.out.println ("Hours must be >= 0");
```

```

44.     }
45.     while (hours < 0.0);
46.     e[i] = new NonexemptEmployee (fName, lName, dept, payRate, hours);
47.     break;
48.     case 'c':
49.     case 'C':
50.     do {
51.         hours = KeyboardReader.getPromptedDouble (
52.             "Enter hours: ", "Hours must be numeric");
53.         if (hours < 0.0)
54.             System.out.println ("Hours must be >= 0");
55.     }
56.     while (hours < 0.0);
57.     e[i] = new ContractEmployee (fName, lName, dept, payRate, hours);
58.     }
59.
60.     System.out.println (e[i].getPayInfo ());
61. } catch (InvalidValueException ex) {
62.     System.out.println (ex.getMessage ());
63.     i--; //failed, so back up counter to repeat this employee
64. }
65. }
66.
67. System.out.println ();
68. System.out.println ("Exempt Employees");
69. System.out.println ("=====");
70. for (Employee emp : e) {
71.     if (emp instanceof ExemptEmployee) {
72.         System.out.println (emp.getPayInfo ());
73.     }
74. }
75. System.out.println ();
76. System.out.println ("Nonexempt Employees");
77. System.out.println ("=====");
78. for (Employee emp : e) {
79.     if (emp instanceof NonexemptEmployee) {
80.         System.out.println (emp.getPayInfo ());
81.     }
82. }
83. System.out.println ();
84. System.out.println ("Contract Employees");
85. System.out.println ("=====");
86. for (Employee emp : e) {
87.     if (emp instanceof ContractEmployee) {
88.         System.out.println (emp.getPayInfo ());

```

```
89.     }
90.     }
91.
92.     Invoice[] inv = new Invoice[4];
93.     inv[0] = new Invoice ("ABC Co.", 456.78);
94.     inv[1] = new Invoice ("XYZ Co.", 1234.56);
95.     inv[2] = new Invoice ("Hello, Inc.", 999.99);
96.     inv[3] = new Invoice ("World, Ltd.", 0.43);
97.
98.     CheckPrinter.printChecks (e);
99.     CheckPrinter.printChecks (inv);
100.  }
101. }
```

---

## Code Explanation

---

Since we are already checking the values of the pay rate and hours, we shouldn't expect to see any exceptions thrown, so it is reasonable to put the entire block that gets the employee data and creates an employee in a try block. If we decrement the counter upon a failure, then that employee's data will be requested again.

You might want to test your logic by temporarily changing one of the test conditions you use when reading input (like `hours > 0` to `hours > -20`), so that you can see the result (keep count of how many employees you are asked to enter).

---



## 8.10. Rethrowing Exceptions

An exception may be *rethrown*.

When we throw an exception, it does not necessarily have to be a new object. We can reuse an existing one.

This allows us to partially process the exception and then pass it up to the method that called this one to complete processing. This is often used in servlets and JSPs to handle part of the problem (possibly just log it), but then pass the problem up to the servlet or JSP container to abort and send an error page.

### **Rethrowing an Exception**

```
String s = "1234X";
try {
    Integer.parseInt(s);
} catch (NumberFormatException e) {
    System.out.println("Bad number, passing buck to JVM");
    throw e;
}
```

The stack trace will still have the original information. The `fillInStackTrace` method for the exception object will replace the original information with information detailing the line on which `fillInStackTrace` was called as the origin of the exception.



## 8.11. Initializer Blocks

Class properties that are object types can be initialized with a newly constructed object.

```
public class MyClass {
    private Random rand = new java.util.Random();
    private MegaString ms =
        new MegaString("Hello " + rand.nextInt(100));
    private int x = rand.nextInt(100);
    . . .
}
```

The `MegaString` class constructor code will run whenever a `MyClass` object is instantiated.

But what if the object's constructor throws an exception?

```
class MegaString{
    public MegaString(String s) throws Exception {
        . . .
    }
}
```

- The MyClass code won't compile - you cannot put a property declaration into a try ... catch structure and there is no place to state that the property declaration throws an exception.

You can use an *initializer block* to handle this situation.

#### **Initializer Block Enclosing try ... catch**

```
public class MyClass {
    private java.util.Random rand = new java.util.Random();
    private MegaString ms = null;
    {
        try { ms = new MegaString()"Hello " + rand.nextInt(100); }
        catch (Exception e) { . . . }
    }
    private int x = rand.nextInt(100);
    . . .
}
```

This is not absolutely necessary, since the initialization could be done in a constructor, where a try ... catch would be legal. But then it would need to be done in every constructor, which someone adding another constructor later might forget.

Initializers are run in the order in which they appear in the code, whether standalone initializers, or initializers in a field declaration so, in the above code:

1. The Random object gets created for the first field.
2. The MegaString gets the first generated random number.
3. x gets the second generated random number.

### ❖ 8.11.1. Static Initializer Blocks

If a field is static, and is populated with a newly constructed object, that object's constructor code will run when the class loads. In our example, if we make the MegaString property static, its constructor will run when the class loads.

```
public class MyClass {
    private static MegaString sms = new MegaString("Goodbye");
    . . .
}
```

- Again, this won't compile, but now there is no way even to defer the issue to the constructors, since the element is `static`.

You can use a *static initializer block* to handle this problem.

#### **Static Initializer Block Enclosing try ... catch**

```
public class MyClass {
    private static MegaString sms = null;
    static {
        try { sms = new MegaString("Hello"); }
        catch (Exception e) { . . . }
    }
    . . .
}
```

Again, the initializers are run in the order in which they appear in the code, when the class is loaded.



## 8.12. Logging

Java has a built-in logging feature, based on the original open-source logging framework *Log4j*. This feature enables you to create various types of loggers, and then log messages and caught exceptions at a number of different levels of severity. The loggers can be configured as to what severity level they will accept - less severe levels will be ignored.

Loggers are normally named, using a hierarchical dot-separated naming strategy. The logging classes are in the `java.util.logging` package. Key classes in addition to `Logger` are *handlers* and *formatters*.

### ❖ 8.12.1. Creating a Logger

Loggers are usually created using the factory method `Logger.getLogger(String name)`. If the named logger already exists, it will be returned. Otherwise, a new logger with that name will be created and returned. A common practice is to use a class name for the logger.

### Creating a Logger

```
Logger log1 = Logger.createLogger(XXX.class.getName());  
Logger log2 = Logger.createLogger(this.getClass().getName());
```

You can programmatically configure the logger's filtering level with:

### Setting Log Levels

```
log.setLevel(Level.FINE);
```

The levels are constant instances of the `Level` class, and the following values represent the levels in descending order:

- SEVERE (the highest value)
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST (the lowest value)

Evaluation  
Copy

There are also `Level` constants for `ALL` and `OFF`; `ALL` is basically the same as `FINEST`, while `OFF` does what you would expect.

Enabling a given level causes the logger to log everything from that level up to the most severe level.

There are logging methods with the same names as the levels, except in all lowercase:

### Making a Log Entry

```
log.fine("Starting iteration " + i);
```

## ❖ 8.12.2. Logger Hierarchy and Naming

There is a hierarchical structure for loggers. There is a *root logger*, whose name is the empty string. Child logs would have a name without any dots, and grandchildren would have a name with one dot in the middle (the grandchildren would be children of the logger

whose name was the part of the grandchild's name before the dot). Information logged at a specific level is also passed up the hierarchy to its parent, so all log entries eventually end up at the root log.

Each logger can have one or more *handlers*, which will produce the log output, and set its own filtering level for entries it will accept *directly*. All entries received from child loggers will be accepted, even if the child logger accepts a finer level.

### ❖ 8.12.3. Log Handlers

Log handlers process the logging, by filtering it, then formatting it via a formatter, and sending it to a destination. The `ConsoleHandler` is the default handler; `FileHandler` is another handler that is often used. Handlers also have a minimum log filtering level. The default level for the console and file handlers is `INFO`, so, the above line of code would not actually result in a log entry using the console logger. Everything passed by the logger to the handler is filtered again by the handler, so a handler can restrict logging levels even further than the logger did.

To create and add a file handler to a logger:

#### Adding a Log Handler

```
FileHandler fLogHandler;  
try {  
    fLogHandler = new FileHandler("test.log");  
    fLogHandler.setLevel(Level.FINER);  
    log.addHandler(fLogHandler);  
} catch (SecurityException e2) {  
    e2.printStackTrace();  
} catch (IOException e2) {  
    e2.printStackTrace();  
}
```

Evaluation  
Copy

To obtain a handler other than the default, you would supply command line options to the JVM, either with specific instructions or by identifying a properties file. You can also edit the master properties file `JAVA_HOME/jre/lib/logging.properties`.

### ❖ 8.12.4. Log Formatters

Each handler has a formatter that formats the output of the handler. The default formatter for a console handler is a `SimpleFormatter` that produces a brief two-line output of each

log entry. For file handlers, the default is an XMLFormatter, which produces XML records for each entry. You can also write custom formatters.

#### Adding a Log Formatter

```
fLogChildHandler.setFormatter(new SimpleFormatter());
```

Evaluation  
Copy

## Demo 8.6: Java-Exceptions/Demos/Logging.java

---

```
1.  import java.io.IOException;
2.  import java.util.logging.*;
3.
4.  public class Logging {
5.      public static void main(String[] args) {
6.          Logger logMain = Logger.getLogger(Logging.class.getName());
7.          logMain.setLevel(Level.INFO);
8.          Logger logChild =
9.              Logger.getLogger(Logging.class.getName() + ".debug");
10.         FileHandler fLogMainHandler = null, fLogChildHandler = null;
11.         Formatter fmtChild = new SimpleFormatter();
12.         try {
13.             fLogMainHandler =
14.                 new FileHandler("LoggingTestMain.log");
15.             fLogMainHandler.setLevel(Level.INFO);
16.             logMain.addHandler(fLogMainHandler);
17.             fLogChildHandler =
18.                 new FileHandler("LoggingTestChild.%g.log", 300, 5);
19.             fLogChildHandler.setLevel(Level.ALL);
20.             logChild.addHandler(fLogChildHandler);
21.             fLogChildHandler.setFormatter(fmtChild);
22.             logMain.setLevel(Level.INFO);
23.             logChild.setLevel(Level.FINER);
24.         } catch (SecurityException e2) {
25.             e2.printStackTrace();
26.         } catch (IOException e2) {
27.             e2.printStackTrace();
28.         }
29.
30.         logMain.info("Starting application ");
31.         for (int i = 0; i < 8; i++) {
32.             logChild.fine("Fine at iteration " + i);
33.             logChild.finer("Finer at iteration " + i);
34.             logChild.finest("Finest at iteration " + i);
35.             if (i == 4) try { System.in.read(); } catch (Exception e) { }
36.         }
37.         logMain.info("Ending application ");
38.         if (fLogMainHandler != null) fLogMainHandler.close();
39.         if (fLogChildHandler != null) fLogChildHandler.close();
40.
41.     }
42. }
```

---

## Code Explanation

---

This example creates a logger named with the class name, `Logging`. It also creates a logger with the name `Logging.child`, which will be a child logger of `Logging`. Remember that `Logging` is a child of the root logger, which has defaults of `INFO` level, console handler, and simple formatter, the output of which you will see in the console window. The `Logging` logger uses a file handler, which defaults to an xml formatter.

`logMain`'s level is set to `INFO`, and its handler's level is set to `FINE`. But, the logger only sends it `INFO` and above, so no lesser messages will appear. The file `LoggingTestMain.log` in your working directory will look like the following:

### **File Handler XML Output**

```
<?xml version="1.0" encoding="windows-1252" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2011-02-05T13:10:15</date>
  <millis>1296929415119</millis>
  <sequence>0</sequence>
  <logger>Logging</logger>
  <level>INFO</level>
  <class>Logging</class>
  <method>main</method>
  <thread>10</thread>
  <message>Starting application </message>
</record>
<record>
  <date>2011-02-05T13:10:15</date>
  <millis>1296929415151</millis>
  <sequence>11</sequence>
  <logger>Logging</logger>
  <level>INFO</level>
  <class>Logging</class>
  <method>main</method>
  <thread>10</thread>
  <message>Ending application </message>
</record>
</log>
```

Evaluation  
Copy

The child logger sets a file naming pattern for its file handler, in which the `%g` stands for generation number, as well as setting a maximum file size of 300 bytes and specifying a

rotation of five files. The program has been set up to pause halfway through, so you can examine the child log files. When you type a character and enter, it will continue. If you review the files again, you will see that the lowest numbered file is the most recent.

The SimpleFormatter we used for the child logger produces output like this:

**Simple Formatter output**

```
Feb 5, 2011 4:35:36 PM Logging main
FINE: Fine at iteration 0
Feb 5, 2011 4:35:36 PM Logging main
FINER: Finer at iteration 0
Feb 5, 2011 4:35:36 PM Logging main
FINE: Fine at iteration 1
Feb 5, 2011 4:35:36 PM Logging main
FINER: Finer at iteration 1
```

Evaluation  
Copy



## 8.13. Log Properties

You can create a properties file to manage handlers and formatters for named logs. You can modify the `logging.properties` under your JDK directory, or create a separate file. The following example uses a separate properties file. To test the program, run the following on the command line:

```
java -Djava.util.logging.config.file=logProperties.properties LogProperties
```

## Demo 8.7: Java-Exceptions/Demos/LogProperties.java

---

```
1. import java.io.IOException;
2. import java.util.logging.*;
3.
4. public class LogProperties {
5.     public static void main(String[] args) {
6.         Logger logMain = Logger.getLogger(LogProperties.class.getName());
7.         Logger logChild =
8.             Logger.getLogger(LogProperties.class.getName() + ".debug");
9.         System.out.println(logMain.getName());
10.        logMain.info("Starting application ");
11.        for (int i = 0; i < 5; i++) {
12.            logChild.fine("Fine at iteration " + i);
13.            logChild.finer("Finer at iteration " + i);
14.            logChild.finest("Finest at iteration " + i);
15.        }
16.        logMain.info("Ending application ");
17.
18.    }
19. }
```

---

### Code Explanation

The Java code is much simpler, since we just request the loggers. The configuration has been handled by the processing of the properties file.

---



## 8.14. Assertions

Assertions are code lines that test that a presumed state actually exists

- If the state is not as presumed, then an `AssertionError` will be thrown.
- Assertions are not intended for testing values that could be expected; they are intended to be used when it is believed that a state exists, but we are not absolutely sure we have covered all the possible avenues that affect the state.

To use an assertion in code:

## Assertion Syntax

```
assert (condition)[: messageExpression];
```

The optional `messageExpression` will be converted to a `String` and passed as the message to the `AssertionError` constructor, so it cannot be a call to a function declared as `void`.

For example, perhaps we are using a third-party function that is specified to return a double value between 0 and 1, but we'd like to guarantee that is the case.

## Demo 8.8: Java-Exceptions/Demos/AssertionTest.java

---

```
1.  public class AssertionTest {
2.
3.      public static void main(String[] args) {
4.          double value = thirdPartyFunction();
5.          assert (value >= 0 && value < 1) :
6.              " thirdPartyFunction value " + value + " out of range";
7.          System.out.println("Value is " + value);
8.      }
9.
10.     public static double thirdPartyFunction() {
11.         return 5.0;
12.     }
13. }
```



## Code Explanation

---

If the function returns a value outside of our expected range, like 5.0, the assertion condition will evaluate to `false`, so an `AssertionError` will be thrown with the message “thirdPartyFunction value 5.0 out of range.”

Assertions are used for debugging a program, and usually not enabled for production runs. You must specifically enable assertions when you run the program, by using the command line switch `-enableassertions` (or `-ea`).

```
java -enableassertions ClassName
java -ea ClassName
```

Oracle’s “rules” for assertions emphasize that they will be disabled most of the time:

- They should not be used for checking the values passed into `public` methods, since that check will disappear if assertions are not enabled.
- The assertion code shouldn't have any side effects required for normal operation.

## Conclusion

In this lesson, you have learned:

- How Java's exception handling mechanism works.
- How to try and catch exceptions.
- About the various types of checked and unchecked exceptions.
- How to write exception classes.
- How to throw exceptions.
- How to use assertions.

# LESSON 9

## Collections

---

### Topics Covered

- ☑ The Collections API.
- ☑ Lists, Sets, and Maps.
- ☑ Iterators.
- ☑ Comparable and Comparator classes.
- ☑ About generic classes.

### Introduction

In this lesson, you will learn about the Collections API, how to work with Lists, Sets, and Maps, how to use Iterators, how to define and use Comparable and Comparator classes, and about generic classes.

Evolution  
Code



## 9.1. Collections

The Java *Collections API* is a set of classes and interfaces designed to store multiple objects.

There are a variety of classes that store objects in different ways:

- *Lists* store objects in a specific order.
- *Sets* reject duplicates of any objects already in the collection.
- *Maps* store objects in association with a key, which is later used to look up and retrieve the object (note that if an item with a duplicate key is put into a map, the new item will replace the old item).

The basic distinctions are defined in several interfaces in the `java.util` package:

- `Collection` is the most basic generally useful interface that most, but not all, collection classes implement.
  - It specifies a number of useful methods, such as those to add and remove elements, get the size of the collection, determine if a specific object is contained in the collection, or return an `Iterator` that can be used to loop through all elements of the collection (more on this later).
  - Methods include: `add(Object o)`, `remove(Object o)`, `contains(Object o)`, and `iterator()`.
  - Note that removing an element removes an object that compares as equal to a specified object, rather than by position.
  - `Collection` extends the `Iterable` interface, which only requires one method, which supplies an object used to iterate through the collection.
- The `List` interface adds the ability to insert and delete at a specified index within the collection, or retrieve from a specific position.
- The `Set` interface expects that implementing classes will modify the add methods to prevent duplicates, and also that any constructors will prevent duplicates (the add method returns a `boolean` that states whether the operation succeeded or not; i.e., if the object could be added because it was not already there).
  - Sets do not support any indexed retrieval.
- The `Map` interface does not extend `Collection`, because its adding, removing, and retrieving methods use a key value to identify an element.
  - Methods include: `get(Object key)`, `put(Object key, Object value)`, `remove(Object key)`, `containsKey(Object key)`, `containsValue(Object value)`, `keySet()`, and `entrySet()`.
  - Maps do not support any indexed retrieval.

In addition to the `List`, `Set`, and `Map` categories, there are also several inferences you can make from some of the collection class names:

- A name beginning with *Hash* uses a hashing and mapping strategy internally, although the key values usually have no meaning (the hashing approach attempts to provide an efficient and approximately equal lookup time for any element).
- A name beginning with *Linked* uses a linking strategy to preserve the order of insertion and is optimized for insertion/deletion as opposed to appending or iterating.

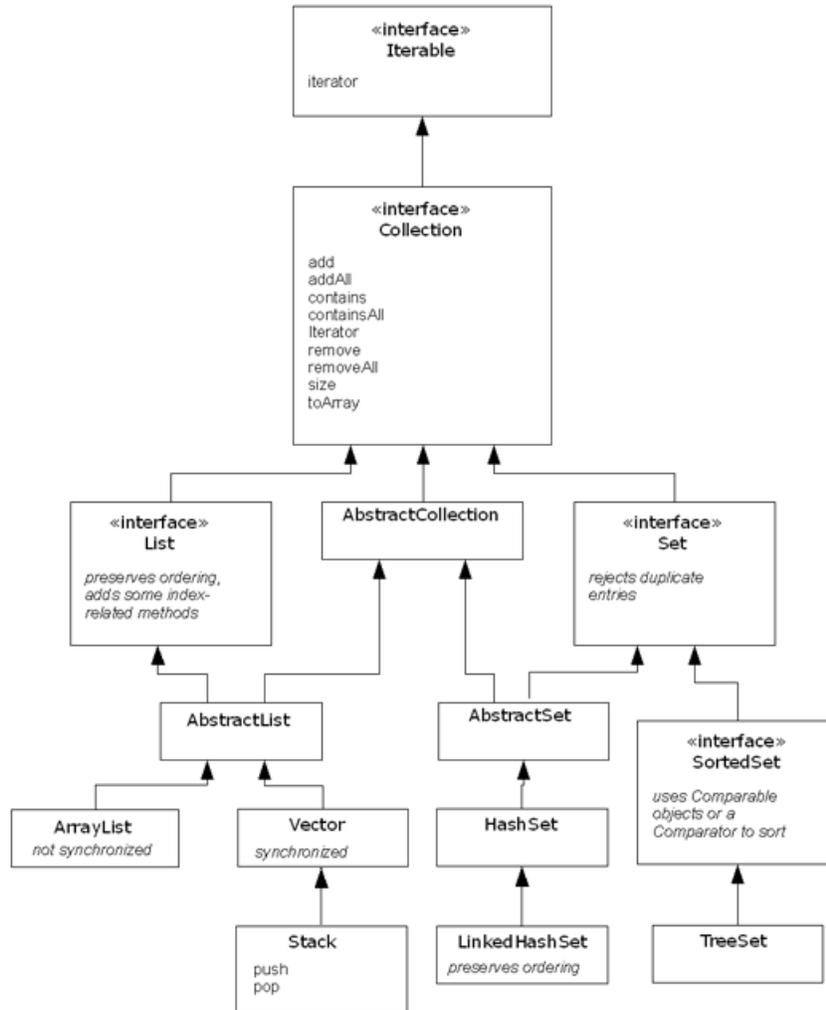
- A name beginning with *Tree* uses a binary tree to impose an ordering scheme, either the *natural order* of the elements (as specified by the `Comparable` interface implemented by many classes including `String` and the numeric wrapper classes) or an order dictated by a special helper class object (a `Comparator`).

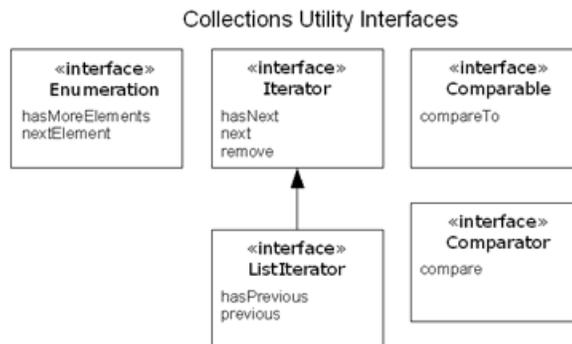
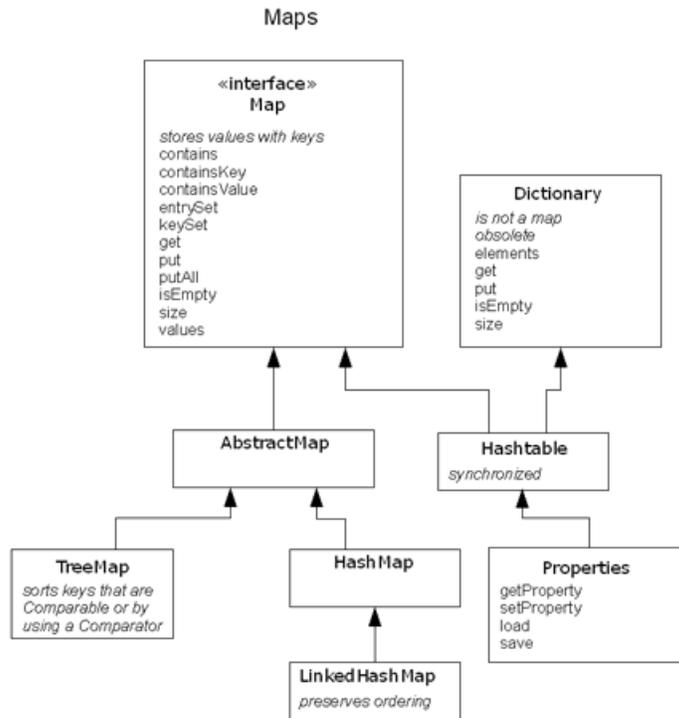
Several additional interfaces are used to define useful helper classes:

- `Enumeration` provides a pair of methods that enable you to loop through a collection in a standardized manner, regardless of the type of collection (you test if there are more elements, and, if so, retrieve the next element).
  - This interface is less frequently used now that the following interface has been added to the API.
  - But, many of the elements in J2EE (servlets in particular) predate the `Iterator` concept and were specified to use enumerations, so they will still appear in new code.
- `Iterator` is an improved version that allows an object to be removed from the collection through the iterator.

The following diagrams show some of the classes and interfaces that are available:

## Lists and Sets





## 9.2. Using the Collection Classes

The following are several examples of collection classes:

Vector stores objects in a linear list, in order of addition. Some additional notes on Vector are:

- You can insert and delete objects at any location.

- All methods are synchronized, so that the class is *thread-safe*.
- Vector predates the collections framework, but was marked to implement `List` when the collections API was developed.
  - As a result, there are many duplicate methods, like `add` and `addElement`, since the names chosen for `Collection` methods didn't all match existing method names in `Vector`.
- Vector was used extensively in the past, and therefore you will see it in a lot of code, but most recent code uses `ArrayList` instead.

`ArrayList`, like `Vector`, stores objects in a linear list, in order of addition. Methods are not synchronized, so that the class is not inherently thread-safe (but there are now tools in the Collections API to provide a thread-safe wrapper for any collection, which is why `Vector` has fallen into disuse).

`TreeSet` stores objects in a linear sequence, sorted by a comparison, with no duplicates. `TreeSet` also:

- Stores `Comparable` items or uses a separate `Comparator` object to determine ordering.
- Uses a *balanced tree* approach to manage the ordering and retrieval.

#### Note

There is no tree list collection, because the concepts of insertion order and natural order are incompatible. Since sets reject duplicates, any comparison algorithm should include a guaranteed tiebreaker (for example, to store employees in last name, first name order: to allow for two Joe Smiths, we should include the employee id as the final level of comparison).

`TreeMap` stores objects in a `Map`, where any subcollection or iterator obtained will be sorted by the key values. Note that:

- Keys must be `Comparable` items or have a separate `Comparator` object.
- For maps, an iterator is not directly available. You must get either the key set or entry set, from which an iterator is available.

`Hashtable` stores objects in a `Map`. Note also that:

- All methods are synchronized, so that the class is thread-safe.
- `Hashtable` predates the collections framework, but was marked to implement `Map` when the collections API was developed.
- Like `Vector`, `Hashtable` has some duplicate methods.
- `Hashtable` extends an obsolete class called `Dictionary`.

`HashSet` uses hashing strategy to manage a `Set`. Note also that:

- `HashSet` rejects duplicate entries.
- Entries are managed by an internal `HashMap` that uses the entry `hashCode` values as the keys.
- Methods are not synchronized.



### 9.3. Using the Iterator Interface

*Iterators* provide a standard way to loop through all items in a collection, regardless of the type of collection. Note that:

- All collection classes implement an `iterator()` method that returns the object's iterator (declared as `Iterator iterator()`).
- This method is specified by the `Iterable` interface, which is the base interface for `Collection`.
- For maps, the collection itself is not `Iterable`, but the set of keys and the set of entries are.
- You can use a for-each loop for any `Iterable` object.

Without an iterator, you could still use an indexed loop to walk through a list using the `size()` method to find the upper limit, and the `get(int index)` method to retrieve an item, but other types of collections may not have the concept of an index, so an iterator is a better choice.

There are two key methods supplied by an iterator:

1. `boolean hasNext()`, which returns `true` until there are no more elements.

- Object `next()`, which retrieves the next element and also moves the iterator's internal pointer to it.

Some collections allow you to remove an element through the iterator. The `remove` method is marked in the docs as *optional*. By the definition of interfaces it has to be present - but the optional status indicates that it may be implemented to merely throw an `UnsupportedOperationException`.

In any case, if the collection is modified from a route other than via the iterator (perhaps by using `remove(int index)`, or even just using the `add` method), a `ConcurrentModificationException` is thrown.

The `Enumeration` class is an older approach to this concept; it has methods `hasMoreElements()` and `nextElement()`.

The following demo gives an example of using various collection classes and an iterator.

#### Note

When compiling the next few demos, Java will offer a warning similar to the following:

- Note: `CollectionsTest.java` uses unchecked or unsafe operations.
- Note: Recompile with `-Xlint:unchecked` for details.

Because we are here using the collection classes without specifying a type. You can ignore the warning.

## Demo 9.1: Java-Collections/Demos/CollectionsTest.java

---

```
1.  import java.util.*;
2.
3.  public class CollectionsTest {
4.      public static void main(String[] args) {
5.          List l = new ArrayList();
6.          Map m = new TreeMap();
7.          Set s = new TreeSet();
8.
9.          l.add(new Integer(1));
10.         l.add(new Integer(4));
11.         l.add(new Integer(3));
12.         l.add(new Integer(2));
13.         l.add(new Integer(3));
14.
15.         m.put(new Integer(1), "A");
16.         m.put(new Integer(4), "B");
17.         m.put(new Integer(3), "C");
18.         m.put(new Integer(2), "D");
19.         m.put(new Integer(3), "E");
20.
21.         System.out.println("Adding to Set");
22.         System.out.println("Adding 1: " + s.add(new Integer(1)));
23.         System.out.println("Adding 4: " + s.add(new Integer(4)));
24.         System.out.println("Adding 3: " + s.add(new Integer(3)));
25.         System.out.println("Adding 2: " + s.add(new Integer(2)));
26.         System.out.println("Adding 3: " + s.add(new Integer(3)));
27.
28.         System.out.println("List");
29.         Iterator i = l.iterator();
30.         while (i.hasNext()) System.out.println(i.next());
31.
32.         System.out.println("Map using keys");
33.         i = m.keySet().iterator();
34.         while (i.hasNext()) System.out.println(m.get(i.next()));
35.
36.         System.out.println("Map using entries");
37.         i = m.entrySet().iterator();
38.         while (i.hasNext()) System.out.println(i.next());
39.
40.         System.out.println("Set");
41.         i = s.iterator();
42.         while (i.hasNext()) System.out.println(i.next());
43.     }
44. }
```

---

## Code Explanation

---

This program demonstrates the three types of collections. As is commonly done, the variables are typed as the most basic interfaces (`List`, `Set`, and `Map`).

We attempt to add the same sequence of values to each: 1, 4, 3, 2, and 3. Note that:

- They are deliberately out of sequence and contain a duplicate.
- For the `Map`, the numbers are the keys, and single-letter strings are used for the associated values.

### Note

In the output, first note the `true` and `false` values resulting from attempting to add the values to the `Set`. Also note which value associated with the key 3 is retained in the `Map` listing.

An iterator is then obtained for each and the series printed out (for the `Map`, we try two different approaches: iterating through the keys and retrieving the associated entries, and iterating directly through the set of entries).

Note the order of the values for each, and also which of the duplicates was kept or not. Note also that:

- The `List` object stores all the objects and iterates through them in order of addition.
- The `Map` object stores by key; since a `TreeMap` is used, its iterator returns the elements sorted in order by the keys.
- Since one key is duplicated, the later of the two values stored under that key is kept.
- The `Set` object rejects the duplicate, so the first item entered is kept (although in this case it would be hard to tell from the iterator listing alone which one was actually stored).



## 9.4. Creating Collectible Classes

### ❖ 9.4.1. hashCode and equals

Both the `equals(Object)` method and `hashCode()` methods are used by methods in the Collections API. Note also that:

- The Map classes use them to determine if a key is already present.
- The Set classes use them to determine if an object is already in the set.
- All collections classes use them in `contains` and related methods.

The behavior of `hashCode` should be consistent with `equals`, meaning that two objects that compare as equal should return the same hash code. The reverse is not required; two objects with the same hash code might be unequal, since hash codes provide "bins" for storing objects.

This is critical when writing collectible classes. For example, the implementation of `HashSet`, which should reject duplicate entries, compares a candidate entry's hashcode against that of each object currently in the collection. It will only call `equals` if it finds a matching hash code. If no hash code matches, it assumes that the candidate object must not match any already present in the set.

### ❖ 9.4.2. Comparable and Comparator

Sorted collections can sort elements in two ways:

1. By the natural order of the elements - objects that implement the `Comparable` interface have a natural order.
2. By using a third-party class that implements `Comparator`.

`Comparable` specifies one method, `compareTo(Object o)`, that returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object, respectively.

`Comparator` specifies one method:

1. `int compare(Object a, Object b)` returns a negative integer, zero, or a positive integer if `a` is less than, equal to, or greater than `b`, respectively.

In addition the class for an object stored in a collection should have a `boolean equals(Object o)` method, to determine if two comparator objects can be considered equal. However, note that the behavior of the `compare` method should be *consistent with equals*, so that `compare(a, b)` should return 0 when `a.equals(b)` returns `true`.

As an example: `TreeSet` uses a tree structure to store items. The tree part of the name is just for identifying the algorithm used for storage; you cannot make use of any of the node-related behaviors from the outside. There are several forms of constructors, most notably:

- `TreeSet()` uses the natural order of the items, so they must implement the `Comparable` interface (and the items should be mutually comparable, to avoid casting exceptions being thrown during a comparison).
- `TreeSet(Comparator c)` - uses the specified `Comparator` to compare the items for ordering (and, again, the mutually comparable caveat applies).

## Demo 9.2: Java-Collections/Demos/UseComparable.java

---

```
1. import java.util.*;
2.
3. public class UseComparable {
4.     public static void main(String[] args)
5.         throws java.io.IOException {
6.         String[] names = { "Sue", "Bill", "Tom", "Dave", "Andy",
7.             "Mary", "Beth", "Bill", "Mike" };
8.         TreeSet sl = new TreeSet(Arrays.asList(names));
9.         Iterator it = sl.iterator();
10.        while (it.hasNext()) {
11.            System.out.println(it.next());
12.        }
13.    }
14. }
```

---

### Code Explanation

---

Since `String` implements `Comparable`, the names will appear in alphabetical order when we iterate through the set.

The `Arrays` class provides useful methods for working with arrays, some of which will return a collection backed by the array (the `Collections` classes contain useful methods for working with collections, some of which perform the reverse operation).

---

## Creating a Comparable Class

Objects instantiated from classes that implement the `Comparable` interface may be stored in ordered collections.

The results of the `compareTo` method should match the results of the object's `equals` method. You should implement this method so that if two objects return 0 from `compareTo`, they should also be considered equal in `equals`.

The *natural order* of the objects is determined by this method.

## Creating and Using a Comparator Class

Classes that implement the `Comparator` interface may be also used with ordered collections, but the collection must be constructed with an explicit reference to an instance of the `Comparator`. The `Comparator` is a separate class that will compare two instances of your class to determine the ordering.

The interface specifies `int compare(Object a, Object b)`. The method returns a negative value for an object considered less than the object `b`, a positive value for `b` considered greater than `a`, and 0 if they are equal.

## Demo 9.3: Java-Collections/Demos/UseComparator.java

---

```
1.  import java.util.*;
2.
3.  public class UseComparator {
4.
5.      public static void main(String[] args) throws java.io.IOException {
6.          String[] names = { "Sue", "Bill", "Tom", "Dave", "Andy",
7.                             "Mary", "Beth", "Bill", "Mike" };
8.
9.          TreeSet s2 = new TreeSet(new ReverseComparator());
10.         s2.addAll(Arrays.asList(names));
11.
12.         Iterator it = s2.iterator();
13.         while (it.hasNext()) {
14.             System.out.println(it.next());
15.         }
16.     }
17. }
18.
19. class ReverseComparator implements Comparator {
20.     public int compare(Object o1, Object o2) {
21.         if (o1 instanceof String && o2 instanceof String)
22.             return -((String)o1).compareTo((String)o2);
23.         else throw new ClassCastException("Objects are not Strings");
24.     }
25. }
```

---

### Code Explanation

---

The compare method of our comparator makes use of the existing compareTo method in String and simply inverts the result. Note that the objects being compared are tested to make sure both are String objects. If the test fails, the method throws a ClassCastException.

---

## Demo 9.4: Java-Collections/Demos/UseComparableAndComparator.java

---

```
1.  import java.util.*;
2.
3.  public class UseComparableAndComparator {
4.
5.      public static void main(String[] args) throws java.io.IOException {
6.          String[] names = { "Sue", "Bill", "Tom", "Dave", "Andy",
7.                              "Mary", "Beth", "Bill", "Mike" };
8.
9.          TreeSet s1 = new TreeSet(Arrays.asList(names));
10.         Iterator it = s1.iterator();
11.         while (it.hasNext()) {
12.             System.out.println(it.next());
13.         }
14.
15.         TreeSet s2 = new TreeSet(new ReverseComparator());
16.         s2.addAll(Arrays.asList(names));
17.
18.         it = s2.iterator();
19.         while (it.hasNext()) {
20.             System.out.println(it.next());
21.         }
22.     }
23. }
```

---

Evaluation  
Copy

### Code Explanation

---

Since all the collections store are references, it will not use a lot of memory to store the same references in different collections. This creates an analog to a set of table indexes in a database

---



## 9.5. Generics

*Generics* allow data types to be parameterized for a class. In earlier versions of Java, the collection methods to store objects all received a parameter whose type was `Object`. Therefore, the methods to retrieve elements were typed to return `Object`. To use a retrieved element, you had to typecast the returned object back to whatever it actually was (and somehow you had to know what it actually was).

Generics use a special, new syntax where the type of object is stated in angle brackets after the collection class name.

Instead of `ArrayList`, there is `ArrayList<E>`, where the `E` can be replaced by any type. Within the class, method parameters and return values can be parameterized with the same type.

- ```
public class ArrayList<E> {  
    . . .  
    public void add(int index, E element) { . . . }  
    . . .  
    public E get(int index) { }  
    . . .  
}
```

For example, an `ArrayList` of `String` objects would be `ArrayList<String>`.

## Demo 9.5: Java-Collections/Demos/GenericCollectionsTest.java

---

```
1.  import java.util.*;
2.
3.  public class GenericCollectionsTest {
4.
5.      public static void main(String[] args) {
6.
7.          List<String> ls = new ArrayList<String>();
8.
9.          ls.add("Hello");
10.         ls.add("how");
11.         ls.add("are");
12.         ls.add("you");
13.         ls.add("today");
14.
15.         // using iterator
16.         StringBuffer result = new StringBuffer();
17.         Iterator<String> is = ls.iterator();
18.         while (is.hasNext())
19.             result.append(is.next().toUpperCase()).append(' ');
20.         result.append('?');
21.         System.out.println(result);
22.
23.         // using for-each loop
24.         result = new StringBuffer();
25.         for (String s : ls) result.append(s.toLowerCase()).append(' ');
26.         result.append('?');
27.         System.out.println(result);
28.
29.         // old way
30.         List l = new ArrayList();
31.
32.         l.add("Hello");
33.         l.add("how");
34.         l.add("are");
35.         l.add("you");
36.         l.add("today");
37.
38.         // using iterator
39.         result = new StringBuffer();
40.         Iterator i = l.iterator();
41.         while (i.hasNext())
42.             result.append(((String)i.next()).toUpperCase()).append(' ');
43.         result.append('?');
44.         System.out.println(result);
```

```
45.
46.     // using for-each loop
47.     result = new StringBuffer();
48.     for (Object o : l)
49.         result.append(((String)o).toLowerCase()).append(' ');
50.     result.append('?');
51.     System.out.println(result);
52.
53.     }
54. }
```

---

## Code Explanation

---

As you can see, the objects retrieved from the `ArrayList` are already typed as being `String` objects. Note the following:

- We need to have them as `String` objects in order to call `toUpperCase`, whereas our previous examples only printed the objects, so being typed as `Object` was okay.
  - Without generics, we must `typecast` each retrieved element in order to call `toUpperCase`.
  - To use an iterator, we would declare the variable to use the same type of element as the collection we draw from, as in `Iterator<String>`.
- 



## 9.6. Bounded Types

A type parameter may set bounds on the type used, by setting an upper limit (in inheritance diagram terms) on the class used. The `extends` keyword is used to mean that the class must either be an instance of the specified boundary class, or extend it, or, if it is an interface, implement it:

### **A Bounded Generic Type**

```
public class EmployeeLocator<T extends Employee> { . . . }
public class CheckPrinter<T extends Payable> { . . . }
```

In the first case, the class may be parameterized with `Employee`, or any class that extends `Employee`. In the second, the class may be parameterized with `Payable` or any type that implements the `Payable` interface.



## 9.7. Extending Generic Classes and Implementing Generic Interfaces

When extending a generic class or implementing a generic interface, you can maintain the generic type, as in `public class ArrayList<T> implements List<T>`. In this case, types are still stated in terms of `T`.

You can lock in the generic type: `public class EmployeeList extends ArrayList<Employee>` or `public class StringList implements java.util.List<String>`. In these cases, methods would use the fixed type. For example, if you overrode `add(E)` in `ArrayList<E>` in the above `EmployeeList`, it would be `add(Employee)`.



## 9.8. Generic Methods

Methods may be generic, whether or not they are in a generic class. The syntax is somewhat ugly, since it requires listing the type variable before the return type and requires that at least one parameter to the method be of the generic type (that is how the compiler knows what the type is).

```
public <T> T chooseRandomItem(T[] items) {  
    Random r = new Random();  
    return items[r.nextInt(items.length)];  
}
```

The above method is parameterized with type `T`. The type for `T` is established by whatever type of array is passed in; if we pass in a `String` array, then `T` is `String`. The method will then randomly pick one to return.

The type may be bounded with extends.



## 9.9. Variations on Generics - Wildcards

In the documentation for a collections class, you may see some strange type parameters for methods or constructors, such as:

### **Additional Generics Syntax**

```
public boolean containsAll(Collection<?> c)
public boolean addAll(Collection<? extends E> c)
public TreeSet(Comparator<? super E> comparator)
```

The question mark is a *wildcard*, indicating that the actual type is unknown. But, we at least know limits in the second two cases. The extends keyword in this usage actually means “is, extends, or implements” (which is the same criteria the instanceof operator applies). The super keyword means essentially the opposite: that the type parameter of the other class is, or is more basic than, this class’s type. The usages with extends and super are called *bounded wildcards*.

This syntax only occurs when the variable is itself a generic class. The wildcards then state how that class’s generic type relates to this class’s type.

Why this is necessary leads down a long and winding path. To start, consider the following:

```
List<Employee> exEmps = new ArrayList<ExemptEmployee>();
```

This seems reasonable at first glance, but then consider if this line followed:

```
exEmps.add(new ContractEmployee());
```

Perfectly legal as far as the compiler is concerned, since ContractEmployee fits within the Employee type that the exEmps variable requires, but now we have a contract employee in a list instance that is supposed to hold only exempt employees. So, an instance of a class parameterized with a derived class is *not* an instance of that class parameterized with the base class, even though individual instances of the derived class can be used in the

base-parameterized generic class; e.g., our `List<Employee>` can add *individual* exempt employees.

For the first wildcard case above, `public boolean containsAll(Collection<?> c)`, it does no harm for us to see if our collection contains all the elements of some other collection that may contain an entirely unrelated type (but few, if any, of the items would compare as equal). Note that the `contains` method accepts an `Object` parameter, not an `E`, for this same reason.

The `extends` term in `public boolean addAll(Collection<? extends E> c)` means that the unknown class *is, extends, or implements* the listed type. For instance, we could add all the elements of an `ArrayList<ExemptEmployee>` to an `ArrayList<Employee>`. That makes sense, since we could add individual exempt employees to a basic employee collection. But, since we don't actually know what the parameterized type of the incoming collection is (it is the `?` class), we cannot call any methods on that object that depend on its parameterized type. So we can't add to that collection; we can only read from it.

The `super` term is seen less often; it means that the parameterized type of the incoming collection must be of the same type or a more basic type. The `TreeSet` constructor can accept a `Comparator` for its actual type, or any type more basic (e.g., a `Comparator<Object>` can be used for a `TreeSet` of anything, since its `compare` method will accept any type of data). It is, however, likely that many "acceptable" comparators will end up throwing a `ClassCastException` at runtime if they can't actually compare the types involved. For example, if we had a `Comparator<Employee>` class that compared employee ids, we might wish to use it in a `TreeSet<ExemptEmployee>`, where it would be perfectly valid and eliminate the need to write a comparator for every `Employee` subtype that would contain the same code, comparing `Employee` ids.



## 9.10. Type Erasure

One discomfoting aspect of generics is that they are handled entirely by the compiler. The compiler enforces the type restrictions on values passed to methods and allows you to use return values without typecasting. But, once the compiler is done, the actual class used will be the regular class without any generic type. In other words, at runtime, an `ArrayList<Employee>` is actually implemented as a plain `ArrayList`. There are no separate classes created for `ArrayList<Employee>` as opposed to `ArrayList<ExemptEmployee>`. The compiler enforced what went in, and therefore allowed assumptions about what we retrieved from it.

The concept that the knowledge of the type disappears after compilation is called *type erasure*.

This means that you *can* use a generic class without specifying the type parameter. But, the compiler will give a warning about using a “raw type”. Mixing raw and parameterized types can lead to runtime problems.

Another aspect of type erasure is that you cannot write any code that will depend at runtime on knowledge of the type. In the following example, neither method is legal:

```
public class Eraser<t> {
    T createEraser() {
        return new T();
    }

    T[] createEraserArray(int size) {
        return new T[size];
    }
}
```

We might expect a problem with the first one, since who's to say that there actually is a default constructor for `T`? But, there is a problem with the type having been erased. At runtime, we don't know what `T` is (and, again, separate classes don't get created for each type we use the class with, so the compiler can't build in that knowledge for each usage).

The second example doesn't require a constructor but still has the same issue of not knowing the appropriate type at runtime.

There are a few tricks available to get around this type of issue much of the time. If the method accepts a parameter of type `T`, then it can create a `T` via reflection, since it can get the class instance from the parameter and call `newInstance`. The rather strange looking `<T> T[] toArray(T[] a)` method in `List<E>` is an example. It uses the type of the incoming dummy array to create a new array of type `T`, to hold all the elements from the list. It cannot use the incoming array, since it might not be the correct size, so a new array must be created.

Note that in the above method declaration, the type `T` is not the parameterized type of the class (which is `E`). This is an example of a *generic method*. Generic methods can exist in both parameterized and nonparameterized classes. The `<T>` listed in front of the return type `T` specifies that this method uses its own independent generic type. At least one

parameter must be of type  $T$  - that is how  $T$  is identified. So, if we had an `ArrayList<ExemptEmployee>` called `exEmps`, we could get an `Employee` array from it with:

```
Employee[] emps = exEmps.toArray( new Employee[] { } );
```

The compiler will see `Employee[]` used for the  $T$ -parameterized type and know that an `Employee` array is being returned. The `toArray` method will return an array that has been created from `exEmps`.

### ❖ 9.10.1. instanceof Tests with Generic Classes

Because of type erasure, it is not legal to test if an object is an instance of a class with a specific type. So, the following is not legal:

```
if (emps instanceof List<Employee>) { . . . }
```

Since the type will not be known at runtime, there is no way we can test for it. Similarly, there is no class object for something like `List<Employee>`; there is just the class object for `List`.



## 9.11. Multiple-bounded Type Parameters

A type parameter in a generic class can be bounded with multiple types, all of which must be satisfied by the actual type. In order for this to work, at most one of the types may be a class, the rest, or all, must be interfaces. The class used must then be an instance of all the listed types (if a class is listed, it must extend that class, and it must implement all the implemented interfaces). The listed items are separated with an `&` character, which is somewhat intuitive if you read it as “and”.

```
public class EnvelopePrinter<T extends Person & Payable> { . . . }
```

This class can use types that are or extend `Person`, and implement `Payable` (such as our `Employee` class, but not our `Invoice` class, since `Invoice` doesn't extend `Person`).

## Exercise 29: Payroll Using Generics

 15 to 25 minutes

We can modify our payroll application to use generic lists instead of arrays for our employees, invoices, and payables.

1. Open the files in `Java-Collections/Exercises/Payroll-Collections01/`
2. Make the `Employee[]` variable a `List<Employee>`, and populate it with a new `ArrayList<Employee>`.
3. Since lists have no fixed size, you will need to change the first for loop, perhaps to a fixed number of iterations.
4. Modify the lines that add employees from using array indices to using the `add` method.
5. Turn the `Invoice[]` into a `List<Invoice>` populated with a `Vector<Invoice>`, similar to what we did in we did in step 1.
6. Modify the lines that populate that list.
7. Create an `ArrayList<Payable>` in a `List<Payable>` `payments` variable.
8. Then call that list's `addAll` method once and pass in the employee list.
9. Then add all the invoices the same way.
10. Since we might not want to modify `CheckPrinter`, you can generate a `Payable` array for it with:

```
CheckPrinter.printChecks( payments.toArray(new Payable[] { }) );
```

The parameter is a “dummy” array used to tell the generic method what type of array to create. Note in the `Collection` documentation that this method is typed with `T` rather than the `E` used in the rest of the class. This method has its own *local type*, which is determined by the type of the array passed in.



## Solution: Java-Collections/Solutions/Payroll-Collections01/Payroll.java

```
1.  import employees.*;
2.  import vendors.*;
3.  import util.*;
4.  import finance.*;
5.  import java.util.*;
6.
7.  public class Payroll {
8.      public static void main(String[] args) {
9.          List<Employee> e = new ArrayList<Employee>();
10.         Employee empl = null;
11.         String fName = null;
12.         String lName = null;
13.         int dept = 0;
14.         double payRate = 0.0;
15.         double hours = 0.0;
16.
17.         for (int i = 0; i < 5; i++) {
18.             try {
19.                 char type =
20.                     KeyboardReader.getPromptedChar("Enter type: E, N, or C: ");
21.                 if (type != 'e' && type != 'E' &&
22.                     type != 'n' && type != 'N' &&
23.                     type != 'c' && type != 'C') {
24.                     System.out.println("Please enter a valid type");
25.                     i--;
26.                     continue;
27.                 }
28.                 fName = KeyboardReader.getPromptedString("Enter first name: ");
29.                 lName = KeyboardReader.getPromptedString("Enter last name: ");
30.                 dept = KeyboardReader.getPromptedInt(
31.                     "Enter department: ", "Department must be numeric",
32.                     new DeptValidator(), "Valid departments are 1 - 5");
33.                 do {
34.                     payRate = KeyboardReader.getPromptedDouble("Enter pay rate: ",
35.                         "Pay rate must be numeric");
36.                     if (payRate < 0.0) System.out.println("Pay rate must be >= 0");
37.                 } while (payRate < 0.0);
38.
39.                 switch (type) {
40.                     case 'e':
41.                     case 'E':
42.                         empl = new ExemptEmployee(fName, lName, dept, payRate);
43.                         e.add(empl);
44.                         break;
```

```

45.     case 'n':
46.     case 'N':
47.         do {
48.             hours = KeyboardReader.getPromptedDouble("Enter hours: ");
49.             if (hours < 0.0)
50.                 System.out.println("Hours must be >= 0");
51.         } while (hours < 0.0);
52.         empl = new NonexemptEmployee(fName, lName, dept, payRate, hours);
53.         e.add(empl);
54.         break;
55.     case 'c':
56.     case 'C':
57.         do {
58.             hours = KeyboardReader.getPromptedDouble("Enter hours: ");
59.             if (hours < 0.0)
60.                 System.out.println("Hours must be >= 0");
61.         } while (hours < 0.0);
62.         empl = new ContractEmployee(fName, lName, dept, payRate, hours);
63.         e.add(empl);
64.     }
65.
66.     System.out.println(empl.getPayInfo());
67. } catch (InvalidValueException ex) {
68.     System.out.println(ex.getMessage());
69.     i--; //failed, so back up counter to repeat this employee
70. }
71. }
72.
73. System.out.println();
74. System.out.println("Exempt Employees");
75. System.out.println("=====");
76. for (Employee emp : e) {
77.     if (emp instanceof ExemptEmployee) {
78.         System.out.println(emp.getPayInfo());
79.     }
80. }
81. System.out.println();
82. System.out.println("Nonexempt Employees");
83. System.out.println("=====");
84. for (Employee emp : e) {
85.     if (emp instanceof NonexemptEmployee) {
86.         System.out.println(emp.getPayInfo());
87.     }
88. }
89. System.out.println();

```

```
90. System.out.println("Contract Employees");
91. System.out.println("=====");
92. for (Employee emp : e) {
93.     if (emp instanceof ContractEmployee) {
94.         System.out.println(emp.getPayInfo());
95.     }
96. }
97.
98. List<Invoice> inv = new Vector<Invoice>();
99. inv.add(new Invoice("ABC Co.", 456.78));
100. inv.add(new Invoice("XYZ Co.", 1234.56));
101. inv.add(new Invoice("Hello, Inc.", 999.99));
102. inv.add(new Invoice("World, Ltd.", 0.43));
103.
104. List<Payable> payments = new ArrayList<Payable>();
105. payments.addAll(e);
106. payments.addAll(inv);
107.
108. CheckPrinter.printChecks( payments.toArray(new Payable[] { }) );
109.
110. }
111. }
```

Evaluation  
Copy

---

## Code Explanation

Notice that we didn't have to change the for-each loops; they work equally well for Iterable objects, like lists, as they do for arrays.

Also, the `addAll` method accepts any `Collection`, so it can take an `ArrayList` or a `Vector` equally well. And, since it accepts `Collection<? extends E>`, it can take collections of either `Employee` or `Invoice` (both implement the `Payable` type used for `E` in the the receiving collection).

---

## Conclusion

In this lesson, you have learned how to use the classes and interfaces in the Collections API and how to work with generic classes.

# LESSON 10

## Inner Classes

---

### Topics Covered

- Inner classes.
- enum classes.

### Introduction

In this lesson, you will learn about the creation and use of inner classes and how to create and use enum classes.



### 10.1. Inner Classes, aka Nested Classes

*Inner classes*, also known as *nested classes* are classes defined within another class.

They may be defined as `public`, `protected`, `private`, or with *package access*.

They may only be used “in the context” of the *containing class* (*outer class* or *enclosing class*), unless they are marked as `static`.

- The outer class can freely instantiate inner-class objects within its code; these objects are automatically associated with the outer class instance that created them.
- Code in some other class can instantiate an inner class object associated with a specific instance of the outer class if the inner-class definition is `public` (and its containing class is `public` as well).
- If the inner class is `static`, then it can be instantiated without an outer class instance; otherwise, the inner class object must be attached to an instance of the outer class.

Inner classes may be used to do the following:

- Create a type of object that is only needed within one class, usually for some short-term purpose.
- Create a utility type of object that cannot be used elsewhere (which would allow the programmer to change it without fear of repercussions in other classes).
- Create one-of-a-kind interface implementations (such as individualized event handlers).
- Allow a sort of multiple inheritance, since the inner class may extend a different class than the outer class extends, and an inner class instance would have access to its `private` elements as well as the `private` elements of the outer class object to which it is attached.
- Implement one-to-many relationships where the classes are *tightly coupled* (meaning that code for one or both of the classes needs access to many of the private elements of the other class) - the outer class would be the “one” side of the relationship, with the inner class being the “many” side.
- Provide a specialized form of *callback*, with which a class may pass very limited access to some of its internal components. Note that:
  - The collections classes provide an *iterator*, a class that implements the `Iterator` interface to loop through the elements in the collection using `hasNext` and `next` methods.
  - Given that the internal structure of the collection may be complex, implementing the iterator as an inner class enables it to navigate the structure, while not exposing any other aspects of the collection to the outside world.

Inner class code has free access to all elements of the outer class object that contains it, by name (no matter what the access level of the elements is).

Outer class code has free access to all elements in any of its inner classes, no matter what their access term.

An inner class compiles to its own class file, separate from that of the outer class (the name of the file will be `OuterClassName$InnerClassName.class`, although within your code the name of the class will be `OuterClassName.InnerClassName`); you cannot use the dollar sign version of the name in your code.

An inner class occupies its own memory block, separate from the outer-class memory block.

An inner class may extend one class, which might be unrelated to the class the outer class extends.

An inner class can implement one or more interfaces and, if treated as an instance of one of its interfaces, external code may have no knowledge that the object actually comes from an inner class.



## 10.2. Inner Class Syntax

### Inner Class Syntax

```
[modifiers] class OuterClassName {
    code
    [modifiers] class InnerClassName
        [extends BaseClassToInner]
        [implements SomeInterface[, MoreInterfaces, ...]] {
            fields and methods
        }
}
```

The *definition* of the inner class is always available for the outer class to use. Note that:

- No inner class objects are automatically instantiated with an outer class object.
- Outer class code may instantiate any number of inner class objects - none, one, or many.

## Demo 10.1: Java-InnerClasses/Demos/MyOuter.java

---

```
1.  public class MyOuter {
2.      private int x;
3.      MyOuter(int x, int y) {
4.          this.x = x;
5.          new MyInner(y).privateDisplay();
6.      }
7.      public class MyInner {
8.          private int y;
9.          MyInner(int y) {
10.             this.y = y;
11.         }
12.         private void privateDisplay() {
13.             System.out.println("privateDisplay x = " + x + " and y = " + y);
14.         }
15.         public void publicDisplay() {
16.             System.out.println("publicDisplay x = " + x + " and y = " + y);
17.         }
18.     }
19. }
```

---

Evaluation  
Copy

### Code Explanation

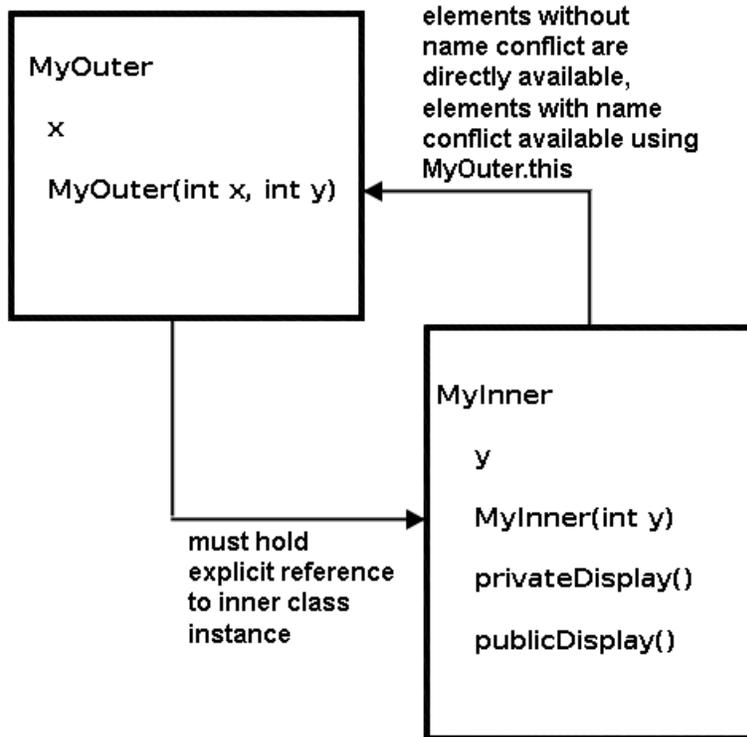
---

This is a simple example of an inner class

- MyOuter has one property, x; the inner class MyInner has one property, y.
- The MyOuter constructor accepts two parameters; the first is used to populate x.
- It creates one MyInner object, whose y property is populated with the second parameter.
- Note that the inner class has free access to the private outer class x element.
- The outer class has free access to the private inner class privateDisplay() method.

The connection between the two classes is handled automatically.

The following diagram maps out the memory used by the example.



## 10.3. Instantiating an Inner Class Instance from within the Enclosing Class

An inner class instance may be directly instantiated from code in the enclosing class, without any special syntax:

### Instantiating an Inner Class Instance from Within the Enclosing Class

```
[modifiers] class OuterClassName {
    code
    [modifiers] class InnerClassName {
        code
    }
    public void someMethod() {
        InnerClassName variable = new InnerClassName();
    }
}
```

Such an instance is automatically associated with the enclosing class instance that instantiated it.

### **Demo 10.2: Java-InnerClasses/Demos/Inner1.java**

---

```
1. public class Inner1 {
2.     public static void main(String[] args) {
3.         new MyOuter(1, 2);
4.     }
5. }
```

Evaluation  
Copy

#### Code Explanation

---

This code simply creates an instance of the outer class, MyOuter.

The MyOuter constructor creates an instance of MyInner as mentioned earlier.

---



## 10.4. Inner Classes Referenced from Outside the Enclosing Class

If the access term for the inner class definition is `public` (or the element is accessible at package access or `protected` level to the other class), then other classes can hold references to one or more of these inner class objects.

- If the inner class is `static`, then it can exist without an outer class object, otherwise any inner class object must belong to an outer class instance.

For code that is not in the outer class, a reference to a `static` or non-`static` inner class object must use the outer class name, a dot, then the inner class name:

#### Inner Class Reference Syntax

```
OuterClassName.InnerClassName innerClassVariable
```

If the inner class has an accessible constructor, then you can instantiate one from outside of the enclosing class, although the syntax is ugly, and there is rarely a need for this capability.



## 10.5. Referencing the Outer Class Instance from the Inner Class Code

If inner class code needs a reference to the outer class instance to which it is attached, use the name of the outer class, a dot, and `this`. Remember that if there is no name conflict, there is no need for any special syntax.

For code in `MyInner` to obtain a reference to its `MyOuter`:

#### Outer Class this Reference

```
MyOuter.this
```

### ❖ 10.5.1. `static` Inner Classes

An inner class may be marked as `static`.

A `static` inner class may be instantiated without an instance of the outer class. Note that:

- `static` members of the outer class are visible to the inner class, no matter what their access level.

- Non-static members of the outer class are not available, since there is no instance of the outer class from which to retrieve them.

To create a static inner class object from outside the enclosing class, you must still reference the outer class name.

#### Instantiating an Inner Class Object From External Code

```
new OuterClassName.InnerClassName(arguments)
```

An inner class may not have static members unless the inner class is itself marked as static.

### **Demo 10.3: Java-InnerClasses/Demos/StaticInnerTest.java**

---

```
1.  class StaticOuter {
2.      public StaticInner createInner() {
3.          return new StaticInner();
4.      }
5.
6.      static class StaticInner {
7.          public static void display() {
8.              System.out.println("StaticOuter.Inner display method");
9.          }
10.     }
11. }
12.
13. class StaticInnerTest {
14.     public static void main(String[] args) {
15.
16.         new StaticOuter.StaticInner().display();
17.
18.         StaticOuter so = new StaticOuter();
19.
20.         //so.new StaticInner().display();
21.
22.         so.createInner().display();
23.
24.
25.     }
26. }
```

---

## Code Explanation

---

We have a class `StaticOuter` that declares a static inner class `StaticInner`. `StaticOuter` has a method that will create instances of `StaticInner`. But, `StaticInner` also has a public constructor. Note that:

- We can directly instantiate a `StaticOuter.StaticInner` object without an outer class instance.
- Code for a `StaticOuter` can create a `StaticInner`, but the inner class object has no attachment to the outer class object that created it.
- Note the commented out line; you cannot create a static inner class instance attached to an instance of its enclosing class.



## 10.6. Better Practices for Working with Inner Classes

It is easiest if inner-class objects can always be instantiated from the enclosing class object. You can create a *factory method* to accomplish this.

## Demo 10.4: Java-InnerClasses/Demos/FactoryInnerOuter.java

---

```
1.  class FactoryOuter {
2.      FactoryInner[] fi = new FactoryInner[3];
3.      protected int lastIndex = 0;
4.      private int x = 0;
5.      public FactoryOuter(int x) {
6.          this.x = x;
7.      }
8.      public int getX() {
9.          return x;
10.     }
11.     public void addInner(int y) {
12.         if (lastIndex < fi.length) {
13.             fi[lastIndex++] = new FactoryInner(y);
14.         }
15.         else throw new RuntimeException("FactoryInner array full");
16.     }
17.     public void list() {
18.         for (int i = 0; i < fi.length; i++) {
19.             System.out.print("I can see into the inner class where y = " +
20.                 fi[i].y + " or call display: ");
21.             fi[i].display();
22.         }
23.     }
24.     public class FactoryInner {
25.         private int y;
26.         protected FactoryInner(int y) {
27.             this.y = y;
28.         }
29.         public void display() {
30.             System.out.println("FactoryInner x = " + x + " and y = " + y);
31.         }
32.     }
33. }
34. public class FactoryInnerOuter {
35.     public static void main(String[] args) {
36.         FactoryOuter fo = new FactoryOuter(1);
37.         fo.addInner(101);
38.         fo.addInner(102);
39.         fo.addInner(103);
40.         fo.list();
41.         //fo.addInner(104);
42.     }
43. }
```

## Code Explanation

---

For convenience, this file contains both the main class and the `FactoryOuter` class (with package access). Note that:

- An instance of `FactoryOuter` contains a three element array of `FactoryInner` objects.
- The `addInner` method instantiates a `FactoryInner` object and adds it to the array (note that is still automatically associated with the `FactoryOuter` instance by the JVM, but we need our own mechanism for keeping track of the inner class instances we create).
  - A better approach would be to use one of the collections classes instead of an array, to avoid running out of room in the array.

---

This is exactly the sort of thing that happens when you obtain an iterator from a collection class. In order to successfully navigate what is most likely a complex internal structure, the object will need access to the private elements. So, an inner class is used, but all you need to know about the object is that it implements the `Iterator` interface.

Let's look at how we might apply these concepts to our "Payroll" program:

## Demo 10.5:

### Java-InnerClasses/Demos/PayrollInnerClass/employees/Employee.java

```
1.  package employees;
2.  import finance.TransactionException;
3.
4.  public class Employee {
5.
6.      private static int nextId = 1;
7.
8.      public static void setNextId(int nextId) {
9.          Employee.nextId = nextId;
10.     }
11.
12.     private int id = nextId++;
13.     private String firstName;
14.     private String lastName;
15.     private double payRate;
16.
17.     private double ytdPay;
18.     private Payment[] payments = new Payment[12];
19.     private int paymentCount = 0;
20.
21.     public Employee() { }
22.
23.     public Employee(String firstName, String lastName, double payRate) {
24.         setFirstName(firstName);
25.         setLastName(lastName);
26.         setPayRate(payRate);
27.     }
28.
29.     public static int getNextId() {
30.         return nextId;
31.     }
32.
33.     public int getId() { return id; }
34.
35.     public String getFirstName() { return firstName; }
36.
37.     public void setFirstName(String firstName) {
38.         this.firstName = firstName;
39.     }
40.
41.     public String getLastName() { return lastName; }
42.
43.     public void setLastName(String lastName) {
```

```

44.     this.lastName = lastName;
45. }
46.
47. public String getFullName() { return firstName + " " + lastName; }
48.
49. public double getPayRate() { return payRate; }
50.
51. public void setPayRate(double payRate)
52.     throws IllegalArgumentException {
53.     if (payRate >= 0) {
54.         this.payRate = payRate;
55.     }
56.     else {
57.         throw new IllegalArgumentException(
58.             "Pay amount: " + payRate + " must not be negative");
59.     }
60. }
61.
62. public double getYtdPay() { return ytdPay; }
63.
64. public Payment createPayment() {
65.     Payment p = new Payment(payRate);
66.     payments[paymentCount++] = p;
67.     return p;
68. }
69.
70. public void printPaymentHistory() {
71.     for (Payment p : payments) {
72.         System.out.println(p);
73.     }
74. }
75.
76. public class Payment {
77.     private double amount;
78.     private boolean posted;
79.
80.     public Payment(double amount) {
81.         this.amount = amount;
82.     }
83.
84.     public boolean process() throws TransactionException {
85.         if (!posted) {
86.             ytdPay += amount;
87.             posted = true;
88.             System.out.println(getFullName() + " paid " + amount);

```

*Evaluation  
Copy*

```
89.         return true;
90.     } else {
91.         throw new TransactionException("Transaction already processed");
92.     }
93. }
94.
95.     public String toString() {
96.         return getFullName() + " payment of " + amount;
97.     }
98.
99. }
100. }
```

---

## Code Explanation

---

Payment is an inner class to a simplified Employee and, as an inner class, has free access to all private elements of Employee. Unlike a standalone payment class, this class can retrieve the employee name from the outer class instance. We also use this access to defer updating the year-to-date amounts until the payment is posted, via the process method.

To get this degree of interaction between two separate classes would be difficult, since it would mean that either:

1. The ability to update ytdPay would have to be publicly available.
2. Employee and Payment would have to be in the same package, with updating ytdPay achieved by using package access.

Note that we have also separated the concepts of creating a payment from actually posting it. This gives us better control over transactions - note that a payment cannot be processed twice.

---

## Demo 10.6: Java-InnerClasses/Demos/PayrollInnerClass/Payroll.java

---

```
1.  import employees.*;
2.  import finance.*;
3.
4.  public class Payroll {
5.      public static void main(String[] args) {
6.
7.          Employee.setNextId(22);
8.          Employee e = new Employee("John", "Doe", 6000.0);
9.
10.         // loop to pay each month
11.         for (int month = 0; month < 12; month++) {
12.             Employee.Payment p = e.createPayment();
13.             try {
14.                 p.process();
15.
16.                 // HR error causes attempt to process June paycheck twice
17.                 if (month == 5) p.process();
18.             }
19.             catch (TransactionException te) {
20.                 System.out.println(te.getMessage());
21.             }
22.             System.out.println("Ytd pay: " + e.getYtdPay());
23.         }
24.
25.         System.out.println("Employee Payment History:");
26.
27.         e.printPaymentHistory();
28.     }
29.
30. }
```

---

### Code Explanation

---

We have only one employee for simplicity. As we loop for each month, a payment is created for each. We try to process the June payment twice (remember that the array is zero-based, so January is month 0; this matches the behavior of the `java.util.Date` class). The second attempt to process the payment should throw an exception which our `catch` block handles.

We retrieve and print the year-to-date pay each time we process a payment.

At the end, we have the `Employee` object print the entire payment history created by our calls to the inner class' `process` method.

---

In the next demo, we'll look at how we could enhance our "Payroll" program further, using an inner class that implements an interface.

EVALUATION COPY

## Demo 10.7:

### Java-InnerClasses/Demos/PayrollInnerClassInterface/employees/Employee.java

```
1.  package employees;
2.  import finance.*;
3.
4.  public class Employee {
5.
6.      private static int nextId = 1;
7.
8.      public static void setNextId(int nextId) {
9.          Employee.nextId = nextId;
10.     }
11.
12.     private int id = nextId++;
13.     private String firstName;
14.     private String lastName;
15.     private double payRate;
16.
17.     private double ytdPay;
18.     private Payment[] payments = new Payment[12];
19.     private int paymentCount = 0;
20.
21.     public Employee() { }
22.
23.     public Employee(String firstName, String lastName, double payRate) {
24.         setFirstName(firstName);
25.         setLastName(lastName);
26.         setPayRate(payRate);
27.     }
28.
29.     public static int getNextId() {
30.         return nextId;
31.     }
32.
33.     public static void setNextId(int nextId) {
34.         Employee.nextId = nextId;
35.     }
36.
37.     public int getId() { return id; }
38.
39.     public String getFirstName() { return firstName; }
40.
41.     public void setFirstName(String firstName) {
42.         this.firstName = firstName;
43.     }
```

```
44.
45.     public String getLastName() { return lastName; }
46.
47.     public void setLastName(String lastName) {
48.         this.lastName = lastName;
49.     }
50.
51.     public String getFullName() { return firstName + " " + lastName; }
52.
53.     public double getPayRate() { return payRate; }
54.
55.     public void setPayRate(double payRate)
56.         throws IllegalArgumentException {
57.         if (payRate >= 0) {
58.             this.payRate = payRate;
59.         }
60.         else {
61.             throw new IllegalArgumentException(
62.                 "Pay amount: " + payRate + " must not be negative");
63.         }
64.     }
65.
66.     public double getYtdPay() { return ytdPay; }
67.
68.     public Payment createPayment() {
69.         Payment p = new Payment(payRate);
70.         payments[paymentCount++] = p;
71.         return p;
72.     }
73.
74.     public void printPaymentHistory() {
75.         for (Payment p : payments) {
76.             System.out.println(p);
77.         }
78.     }
79.
80.     public class Payment implements Payable {
81.         private double amount;
82.         private boolean posted;
83.
84.         public Payment(double amount) {
85.             this.amount = amount;
86.         }
87.
88.         public boolean process() throws TransactionException {
```

```
89.     if (!posted) {
90.         ytdPay += amount;
91.         posted = true;
92.         System.out.println(getFullName() + " paid " + amount);
93.         return true;
94.     } else {
95.         throw new TransactionException("Transaction already processed");
96.     }
97. }
98.
99.     public String toString() {
100.         return getFullName() + " payment of " + amount;
101.     }
102.
103. }
104. }
```

---

## Code Explanation

---

This code goes one step further to create a Payment inner class that implements the Payable interface.

---

## Demo 10.8:

### Java-InnerClasses/Demos/PayrollInnerClassInterface/Payroll.java

---

```
1.  import employees.*;
2.  import finance.*;
3.
4.  public class Payroll {
5.      public static void main(String[] args) {
6.
7.          Employee.setNextId(22);
8.          Employee e = new Employee("John", "Doe", 6000.0);
9.
10.         // loop to pay each month
11.         for (int month = 0; month < 12; month++) {
12.             Payable p = e.createPayment();
13.             try {
14.                 p.process();
15.
16.                 // HR error causes attempt to process June paycheck twice
17.                 if (month == 5) p.process();
18.             }
19.             catch (TransactionException te) {
20.                 System.out.println(te.getMessage());
21.             }
22.             System.out.println("Ytd pay: " + e.getYtdPay());
23.         }
24.
25.         System.out.println("Employee Payment History:");
26.
27.         e.printPaymentHistory();
28.     }
29.
30. }
```

---

#### Code Explanation

---

The only difference here is that we declare the variable holding the payments as `Payable`, hiding the fact that it is an inner class.

---



## 10.7. Enums

The `enum` element provides a set of predefined constants for indicating a small set of mutually exclusive values or states.

### ❖ 10.7.1. Why Another Syntax Element for a Set of Constants?

Before enums, there were other ways to handle this kind of situation - but the other approaches all had some sort of flaw, particularly as involves *type-safety*.

- Interfaces defining only constants were commonly used, and several are grandfathered into the API, like `SwingConstants`. But, there is a minor problem that if you implement an interface in order to gain the constants, you then have additional `public` elements in that class that you wouldn't really want to provide to the outside world.

```
public class MyFrame extends JFrame implements SwingConstants { . . . }  
  
MyFrame frame = new MyFrame();  
// frame.HORIZONTAL is now publicly available
```

While not terrible, there isn't any real meaning to `frame.HORIZONTAL`, or any reason we would want to make it available to the outside world.

- Using plain old integers seems straightforward enough, but, if you perhaps have a method that requires one of that set of values to be passed in, the parameter would be typed as `int`. A caller could supply any `int` value, including ones you wouldn't expect.

```
private int LEFT = 0;  
private int RIGHT = 1;  
private int CENTER = 2;  
  
public void setAlignment(int align) { ... }  
  
// compiler would allow:  
setAlignment(99);
```

Java enums provide a type-safe way of creating a set of constants, since they are defined as a `class`, and therefore are a type of data.

A disadvantage to this approach is that the set of values is written into the code. For sets of values that may change, this would require recompiling the code, and would invalidate any serialized instances of the enum class. For example, if we offered a choice of benefits plans to our employees, the set of available plans would not be a good candidate for an enum, since it is likely that the set of available plans would eventually change.

## ❖ 10.7.2. Defining an enum Class

To create a simple enum class:

1. Declare like an ordinary class, except using the keyword `enum` instead of `class`.
2. Within the curly braces, supply a comma-separated list of names, ending in a semicolon.

One instance of the enum class will be created to represent each item you listed, available as a static field of the class, using the name you supplied which will be the individual values. Each instance can provide an integral value, with sequential indexes starting at 0, in the order that the names were defined - there is no way to change this, but there is a route to get specific values which have a complex internal state.

### Declaring a Simple enum Class

```
public enum EnumName {  
    value1, value2, value3, . . . ;  
}
```

### A Simple Enum

```
public enum Alignment { left, right, center; }
```

There will be three instances of the class created: `Alignment.left`, `Alignment.right`, and `Alignment.center`. An `Alignment` type variable can hold any of these three values.

Enums automatically extend the `Enum` class from the API, and they inherit several useful methods:

- Each object has a `name` method that returns the name of that instance (as does the `toString` method).
- The `ordinal` method returns that enum object's position in the set, the integer index mentioned above.

There are also several other methods that will be present, although they are not listed in the documentation for Enum.

- A public static `EnumName[] values()` method that returns an array containing all the values, in order (so that the array index would match the `ordinal` value for that object). This method is not inherited, but is built specifically for each enum class.
- A public `EnumName valueOf(String name)` method that returns the instance whose name matches the specified name (this is not the uglier method you will see in the documentation, but another built specifically for each instance - the one listed in the documentation is actually used internally by the simpler form).

The reason for the last two methods not being in the documentation has to do with generics and *type erasure* - the methods cannot be declared in the Enum base class in a way that would allow the use of the as-yet unknown subclass.

Individual values from the set may be accessed as `static` elements of the enum class. The JVM will instantiate *exactly* one instance of each value from the set. Therefore, they can be used in comparisons with `==`, or in `switch` statements (using the `equals` method is preferred to `==`, since it will serve as a reminder that you are dealing with true objects, not integers).

Although enums may be top-level classes, they are often created as inner classes, as in the following example, where the concept of the enum is an integral part of a new `BookWithEnum` class. When used as an inner class, they are automatically `static`, so that an instance of an inner enum does not have access to instance elements of the enclosing class.

## Demo 10.9: Java-InnerClasses/Demos/BookWithEnum.java

---

```
1.  public class BookWithEnum {
2.      private int itemCode;
3.      private String title;
4.      private double price;
5.      private Category category;
6.
7.      public enum Category { required, supplemental, optional, unknown };
8.
9.      public BookWithEnum(
10.         int itemCode, String title,
11.         double price, Category category) {
12.         setItemCode(itemCode);
13.         setTitle(title);
14.         setPrice(price);
15.         setCategory(category);
16.     }
17.     public BookWithEnum(String title) {
18.         setItemCode(0);
19.         setTitle(title);
20.         setPrice(0.0);
21.         setCategory(Category.unknown);
22.     }
23.     public int getItemCode() {
24.         return itemCode;
25.     }
26.     public void setItemCode (int itemCode) {
27.         if (itemCode > 0) this.itemCode = itemCode;
28.     }
29.     public String getTitle() {
30.         return title;
31.     }
32.     public void setTitle (String title) {
33.         this.title = title;
34.     }
35.     public void setPrice(double price) {
36.         this.price = price;
37.     }
38.     public double getPrice() {
39.         return price;
40.     }
41.     public void setCategory(Category category) {
42.         this.category = category;
43.     }
44.     public void setCategory(String categoryName) {
```

Evaluation  
Copy

```
45.     this.category = Category.valueOf(categoryName);
46.     }
47.     public Category getCategory() {
48.         return category;
49.     }
50.     public void display() {
51.         System.out.println(itemCode + " " + title + ": " + category +
52.             ", Price: $" + price);
53.     }
54. }
```

---

## Code Explanation

---

The `Category` enum is defined as an inner class to `BookWithEnum`. The full names of the complete set of values are: `BookWithEnum.Category.required`, `BookWithEnum.Category.supplemental`, `BookWithEnum.Category.optional`, and `BookWithEnum.Category.unknown`. From within the `BookWithEnum` class, they may be accessed as: `Category.required`, `Category.supplemental`, `Category.optional`, and `Category.unknown`.

We set the category for a book constructed without one as `Category.unknown`, and provide methods to get the value, and to set it with either an enum object or from a string.

Note that enums may be used in `switch` statements - for the cases you use only the short name for the value.

---

### ❖ 10.7.3. More Complex Enums

Enums are more than just a set of integer constants. They are actually a set of unique object instances, and, as objects, can have multiple fields. So, an enum is a class with a fixed number of possible instances, each with its own unique state, and each of the possible instances is created automatically and stored as static field under the same name. (In design pattern terms, an enum is a *Flyweight* - a class where only a limited number of fixed states exist.)

To create a more complex enum class:

1. Declare as before.

2. Declare any additional fields and accessor methods as with a regular class. While you can actually write mutator methods to create what is called a *mutable enum*, this practice is strongly discouraged.
3. Write one constructor.
4. Within the curly braces, again supply a comma-separated list of names, which will be the individual values, but this time with a parameter list. The enum values will be constructed with the data you provide.

## Conclusion

In this lesson, you have learned:

- How to declare and use inner classes.
- How to define and use enum classes.