References

Table of Contents

Session 1: Introduction to Java

Objects Overview

Source	http://freecourseware.uwc.ac.za/freecourseware/information-systems/java-platform-introduction/objects-overview-1/objects-overview
Date of Retrieval	21/01/2013

An object is a bundle of related variables and functions that belong together, i.e. they are grouped into a single entity.

Software objects are often used to model real-world objects that are all around you. A computer is sometimes called *The Universal Machine* because it can be programmed to simulate any other machine. Objects are a large part of how this is achieved, because they can be designed to be just like their real-world counterparts, and to have the same virtual abilities and qualities as the real-world object physically has. Look around - you'll see plenty of real-world objects like your computer, your cellphone, the table the computer is sitting on, etc.

Real objects all have two characteristics:

i. **State** - all the things that make them what they are, e.g. name, size, color, alive, etc.

ii. **Behavior** - all the things that they can do, e.g. fall, break, stop, run, bark, etc.

Software objects are modeled on real objects in that they also have state and behavior. Java objects store their state in *variables*, and they *implement* their behavior through *methods*. A method is just a function (a subroutine of Java code) associated in some particular way with the object.

If you were writing a game that simulated driving a number of different kinds of racing cars, you might decide to simulate the cars with an *Automobile* object, and the actions of the driver through a *Driver* object. Objects also allow you to model abstract concepts. For example, an *event* can be modeled as an object to allow control of a user interface, where an event object would represent the user clicking a mouse button, or pressing a key on the keyboard. Here is a representation of a general software object:

ClassName
variables (state)
...
...
behavior (methods)
...
...

Everything that the software object knows (state) and can do (behavior) is expressed by the variables and the methods within that object. A software object that modeled a real-world automobile would have variables that indicated the automobile's current state: speed 75 km/h, 4th gear, engine speed 1800 rpm, fuel in tank 39 liters. These variables are called *instance variables* because they will contain the state of a particular automobile object, and in object-oriented terminology a particular object is called an *instance* (of a class). The following figure shows an automobile modeled as a software object:

Automobile (object)
variables (state)
speed 75 km/h
5th gear
engine rpm 1800
fuel in tank 39 litres
behavior (methods)
...
...
...

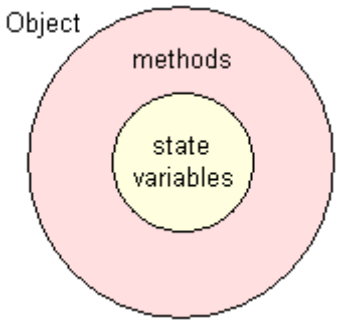
A software automobile would also have methods for braking, accelerating, starting the engine, stopping the engine, changing gear, adding fuel to the tank:

Automobile (object)
variables (state)
speed 75 km/h
5th gear
engine rpm 1800
fuel in tank 39 litres
behavior (methods)
start engine
change gear
accelerate
brake
add fuel to tank
stop engine

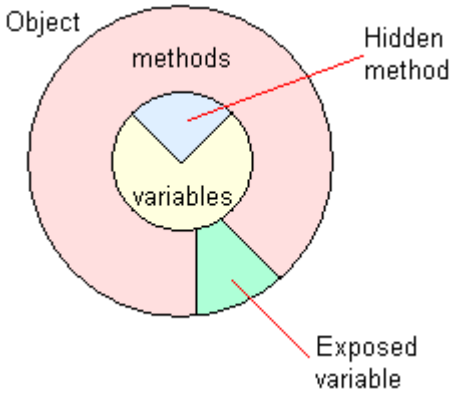
Note that the automobile doesn't have a method for simply changing it's speed because the speed is a function of a lot of factors, e.g. whether the brakes are on, how fast the engine is turning, what gear it's in, how steep the hill is, etc. Our automobile has instead a method for accelerating, i.e. changing the engine revolutions (a better name would

be `changeRPM`). This is a single method that allows both an increase and a decrease in engine r.p.m.; decreases are achieved by a negative argument for the method.

Actually, the table type representation of an object that you saw above, is not the best representation. In Java, an object's variables are screened - protected if you like - by the object's methods from any outside interference. We say that the variables are *encapsulated* by the methods. What this really means is that the methods must stand between the outside world and the variables, so that the state of the object cannot be manipulated in an unexpected or unauthorized way.



Unfortunately, this is not always practical and some objects may want to expose some variables and hide some methods. Java allows this when necessary by having several access levels to the variables and methods of objects. The *access level* determines which other objects and classes are allowed to use the methods and change the state of an object.



There are two chief benefits to this system of encapsulation.

- Firstly, the source code for any object can be stored and maintained independently of the code for any other object. Also, objects can be passed around the system, i.e. you can give or lend your automobile to someone else, and it will still work.
- The object has a *public* interface which other objects must use to communicate with it. *Private* information and methods can be kept hidden from all other objects, which means that the values and methods can be modified at any time without

affecting any other part of the code. You don't have to understand how the engine works in order to be able to drive an automobile.

~~~ End of Article ~~~

Under the JVM Hood - Classloaders

Source	http://www.javacodegeeks.com/2012/12/under-the-jvm-hood-classloaders.html
Date of Retrieval	21/01/2013

Classloaders are a low level and often ignored aspect of the Java language among many developers**. A classloader is just a plain java object**.

Yes, it's nothing clever, well other than the system classloader in the JVM, a classloader is just a java object! It's an abstract class, ClassLoader, which can be implemented by a class you create. Here is the API:

```
01 public abstract class ClassLoader {
02
03  public Class loadClass(String name);
04
05  protected Class defineClass(byte[] b);
06
07  public URL getResource(String name);
08
09  public Enumeration getResources(String name);
10
11  public ClassLoader getParent()
12
13 }
```

Looks pretty straightforward, right? Let's take a look method by method. The central method is loadClass which just takes a String class name and returns you the actual Class object. This is the method which if you've used classloaders before is probably the most familiar as it's the most used in day to day coding. defineClass is a final method in the JVM that takes a byte array from a file or a location on the network and produces the same outcome, a Class object.

A classloader can also find resources from a classpath. It works in a similar way to the loadClass method. There are a couple of methods, getResource and getResources, which return a URL or an Enumeration of

URLs which point to the resource which represents the name passed as input to the method.

Every classloader has a parent; getParent returns the classloaders parent, which is not Java inheritance related, rather a linked list style connection. We will look into this in a little more depth later on.

Classloaders are lazy, so classes are only ever loaded when they are requested at runtime. Classes are loaded by the resource which invokes the class, so a class, at runtime, could be loaded by multiple classloaders depending on where they are referenced from and which classloader loaded the classes which referen… oops, I've gone cross-eyed! Let's look at some code.

```
01 public class A {
02
03  public void doSmth() {
04
05     B b = new B();
06
07     b.doSmthElse();
08
09  }
10
11 }
```

Here we have class A calling the constructor of class B within the doSmth of it's methods. Under the covers this is what is happening

```
A.class.getClassLoader().loadClass("B");
```

The classloader which originally loaded class A is invoked to load the class B.

~~~ End of Article ~~~

How to Set Path for Java Unix/Linux and Windows?

Source	http://javarevisited.blogspot.in/2011/10/how-to-set-path-for-java-unix-linux-and.html
Date of Retrieval	21/01/2013

PATH is one of fundamental Environment variable on shell or DOS but it's commonly associated with Java mainly because if we try to run a java program which doesn't include Java executable in PATH then we say PATH is not set for Java and we need to set path for Java. I have also seen developer getting confused over path and classpath in java. Though both path and classpath provides run-time settings for any java environment which is required to compile and execute Java program they are completely different to each other. Classpath is usually used to find out classes and mostly associated with lib part while PATH is used to find the executable or command to be executed. In order to compile and run java program from command line your PATH environment variable must have "javac" and "java" on it. In this Java PATH tutorial we will see what is PATH for Java, **How to Set Path for Java** and how to troubleshoot PATH related issues.

What is Path in Java

First of all PATH is not specific to java it's a shell concept and also available in Windows and DOS. It's represented by Environment variable called "**PATH**" and that's why it's known as path. Whenever you type a command in shell in UNIX or Linux or in command prompt in windows machine, command will be looked on PATH and if shell is not able to find the command in PATH it says "not recognized" or wrong command. Now for compiling and running we use two java commands "javac" and "java" and these commands doesn't come by default with windows or Unix instead they comes when you install JDK in your machine. Now to successfully compile and run a java program in either windows or Linux you must have these two commands or executable in your PATH environment variable and that is called *Setting Path for Java*.

Setting Path for Java in Unix/Linux and Windows

How to check if "java" or "javac" is in PATH

Javac and Java command resides under /bin directory or your Java installation directory. In my machine its "C:\Program Files\Java\jdk1.6.0_26\bin"

If you have this bin directory in path it means java and javac will be in path and Path is set to run Java Program. There are two ways you verify whether java is in path or not.

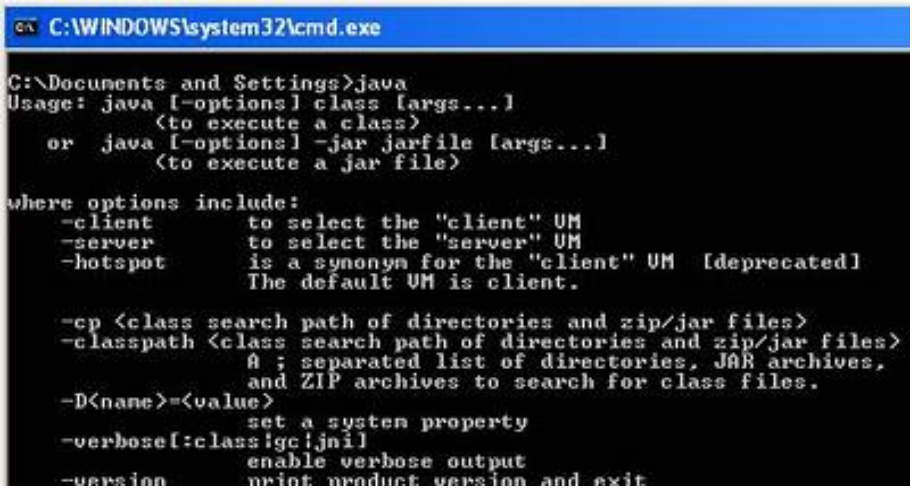
1) Simple and easy way

Open a command prompt window by pressing start -->run-->cmd and then typing "java" or "javac" in command prompt as shown in below example

```
C:\Documents and Settings>java
Usage: java [-options] class [args...]
           (to execute a class)
   or  java [-options] -jar jarfile [args...]
```

```
            (to execute a jar file)

where options include:
    -client       to select the "client" VM
    -server       to select the "server" VM
    -hotspot      is a synonym for the "client" VM [deprecated]
```



If it displays lot of output means Java is in your path and similarly you can check for "javac", on the other hand if java is not in your system's path you will get below output in UNIX

```
stock_trader$ javac
javac: not found

and in windows
C:\Documents and Settings>javac
'javac' is not recognized as an internal or external command,
operable program or batch file.
```

If Java or Javac is not in your path then you can add them into path by adding "bin" directory or your JDK installation directory into environment variable "PATH". Both windows and UNIX use same name.

How to set PATH for Java in windows

I say pretty easy just add bin directory of your JDK installation directory into PATH environment variable. You can do this either from command prompt or by using windows advanced environment editor

1) Setting Java PATH using command prompt in windows

Use "set" command to set value of PATH environment variable as shown in below example:

```
C:\Documents and Settings>set PATH=%PATH%; C:\Program
Files\Java\jdk1.6.0_26\bin
```

%PATH% is actually representing current path and we are appending java bin directory into PATH. Note that every path in windows is comma (;) separated while in UNIX it would be colon (:) separated

2) Setting Java PATH using windows environment variable editor

Use short cut "window key + pause/break" --> Advanced --> Environment Variables --->PATH

Just append the path of java installation directory here and you are done. Open a command prompt and type java or javac and you can see the output.

How to set Java PATH in UNIX or Linux

Setting PATH for java in UNIX is similar to the way we did for it on windows using command prompt. In UNIX just open any shell and execute below command

```
set PATH=${PATH}:/home/opt/jdk1.6.0_26/bin
```

Remember here each element in PATH is colon separated.

~~~ End of Article ~~~

Session 2: Application Development in Java

Edit - compile – test

Source	http://freecourseware.uwc.ac.za/freecourseware/information-systems/java-platform-introduction/edit-compile-test
Date of Retrieval	03/04/2013

The process of building a complete and functioning Java application begins with a text editor. The code is typed into the editor interface, and then saved as `.java` source files. Source files are submitted to the compiler, which adds any *library files* required by the code, and then produces compiled `.class` files. These are interpreted by the Java interpreter at run time, and the results are output as required.
This process is not always as smooth and simple as it sounds, however. It is very difficult to produce a faultless program that runs perfectly the first time it is invoked. Rather, a process of editing source files, trying to compile them, testing them once they successfully compile, and then returning to the editor to fix whatever problems may have been discovered, and then starting the whole process again, is followed. There are two kinds of problem that can cause this process to be rough - *syntax errors*, and *logic errors*. Let's examine the kinds of things that can go wrong in this process, and cause us to have to return to the editor to fix them.

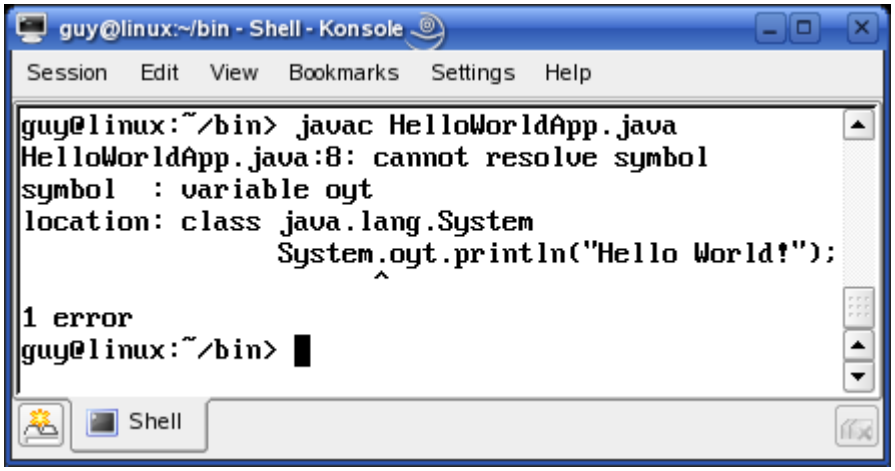
Syntax errors

Syntax errors are mistakes in the way code is typed; these can be spelling errors in the names of classes and methods, punctuation in the wrong place or left out, incorrect use of brackets, incorrect use of method arguments, etc. Open your HelloWorldApp.java source file, and misspell the name of the class variable in the method call, like this:

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

Re-save the file, and now compile it with the Java compiler by using `javac HelloWorldApp.java` just as before.

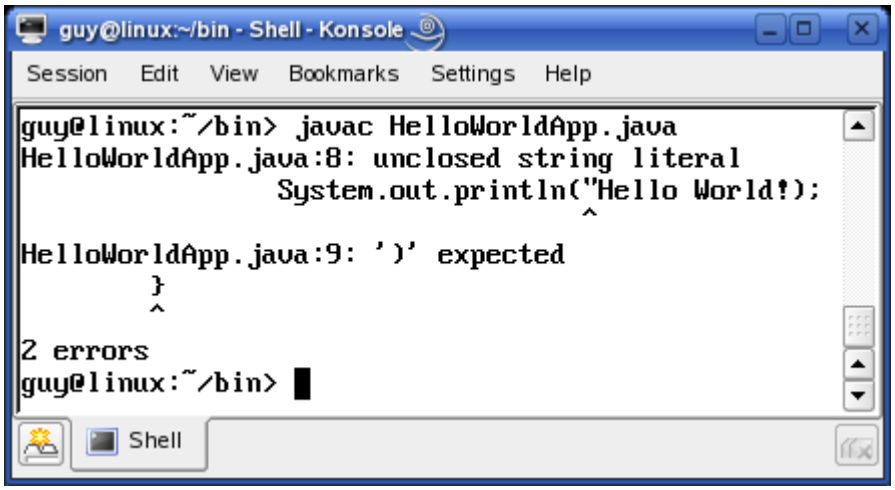
This is what you should see:



The compiler has refused to compile the source code because we have made a *syntax* error by misspelling the name of the class variable in the method call. The Java compiler explains this and uses the caret ^ to point out where the error is in the code, and also tells us which line in the program code contains the problem - line 8 in this case;
see `HelloWorldApp.java:8` in the image above. Fix this error, and then delete the second quotation mark that delimits the string "Hello World!" - like this:

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

Save and compile the file again - this is what should happen:



Once again, the syntax error - missing string delimiter - causes the compiler to refuse to compile the program. Notice that the compiler has detected **two** errors; there is really only one error in the file, but this

error has caused the rest of the code not to make proper sense any more. The compiler tries to make sense of the rest of the code, but can't - so it sends the second error message along with the first. Sometimes a single syntax error will cause the compiler to send a large number of error messages to the console. The best thing to do when dealing with errors is to fix the first one in the list, recompile, see how many error messages have disappeared, fix the first one in the remaining list, and so on until the code compiles without any errors.

Logic errors

Now fix the missing quote error, and then change the string inside the quotes to read:

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

and then save and recompile the file. This time, the code compiles with no problems, but if you now run it you will see

```
Hell World!
```

appear on the screen. The program has compiled and run, but the result is incorrect nonetheless. This is a *run-time* or *logic* error, and errors of this type are not trapped by the compiler because it cannot make any decisions about the logic that the programmer is following in her code. We still have to go back to the editor and fix this error nonetheless.

This process of writing, compiling, running and fixing code is known as the **edit - compile - test** cycle, and in a large program it will need to be followed many times before the code is completely ready to be run. This cycle is a normal part of the programming experience, and you should be prepared to follow it for as many times as it takes to get your code fully debugged. Careful attention to detail as you generate your code will help you to minimize the number of repetitions of this cycle, but be prepared to loop through it a few times! Java makes this process much easier by having an excellent compiler that traps all syntax errors, and points out exactly what and where they are into the bargain.

This means that when it comes time to run your code for the first time, you can be almost certain that any errors that occur are *logic* errors - the number of things that can go wrong has been cut in half by the compiling process.

~~~ End of Article ~~~

javac - The Java Compiler

Source	http://www.cis.upenn.edu/~bcpierce/courses/629/jdkdocs/tooldocs/win32/javac.html
Date of Retrieval	21/01/2013

javac compiles Java programs.

SYNOPSIS

```
javac [ options ] filename.java ...
javac_g [ options ] filename.java ...
```

DESCRIPTION

The **javac** command compiles Java source code into Java bytecodes. You then use the Java interpreter - the **java** command - to interpret the Java bytecodes.

Java source code must be contained in files whose filenames end with the `.java` extension. The file name must be constructed from the class name, as *classname*`.java`, if the class is public or is referenced from another source file.

For every class defined in each source file compiled by **javac**, the compiler stores the resulting bytecodes in a *class file* with a name of the form *classname*`.class`. Unless you specify the **-d** option, the compiler places each class file in the same directory as the corresponding source file.

When the compiler must refer to your own classes you need to specify their location. Use the **-classpath** option or **CLASSPATH** environment variable to do this. The class path is a sequence of directories (or zip files) which **javac** searches for classes not already defined in any of the files specified directly as command arguments. The compiler looks in the class path for both a source file and a class file, recompiling the source (and regenerating the class file) if it is newer.

Set the property `javac.pipe.output` to `true` to send output messages to `System.out`. Set `javac.pipe.output` to `false`, that is, do not set it, to send output messages to `System.err`.

javac_g is a non-optimized version of **javac** suitable for use with debuggers like **jdb**.

OPTIONS

-classpath *path*

Specifies the path **javac** uses to look up classes needed to run **javac** or being referenced by other classes you are compiling. Overrides the default or the **CLASSPATH** environment variable if it is set. Directories are separated by

semi-colons. It is often useful for the directory containing the source files to be on the class path. You should always include the system classes at the end of the path. For example:

```
javac -classpath .;C:\users\dac\classes;C:\tools\java\classes ...
```

-d *directory*

Specifies the root directory of the class file hierarchy. In other words, this is essentially a destination directory for your compiled classes. For example, doing:

```
javac -d C:\users\dac\classes MyProgram.java
```

causes the class files for the classes in the `MyProgram.java` source file to be saved in the directory `C:\users\dac\classes`.

Note that the **-d** and **-classpath** options have independent effects. The compiler reads only from the class path, and writes only to the destination directory. It is often useful for the destination directory to be on the class path. If the **-d** option is not specified, the source files should be stored in a directory hierarchy which reflects the package structure, so that the resulting class files can be easily located.

-encoding *encoding name*

Specify the source file encoding name, such as `EUCJIS\SJIS`. If this option is not specified, then the platform default converter is used.

-g

Enables generation of debugging tables. Debugging tables contain information about line numbers and local variables - information used by Java debugging tools. By default, only line numbers are generated, unless optimization (**-O**) is turned on.

-nowarn

Turns off warnings. If used the compiler does not print out any warnings.

-O

Optimizes compiled code by inlining static, final and private methods. Note that your classes may get larger in size.

-verbose

Causes the compiler and linker to print out messages about what source files are being compiled and what class files are being loaded.

-depend

Causes recompilation of class files on which the source files given as command line arguments recursively depend. Without this option, only files that are directly depended on and missing or out-of-date will be recompiled. Recompilation does not extend to missing or out-of-date files only depended on by already up-to-date class files.

-J*javaoption*

Passes through the string *javaoption* as a single argument to the Java interpreter which runs the compiler. The argument should not contain spaces. Multiple argument words must all begin with the prefix**-J**, which is stripped. This is useful for adjusting the compiler's execution environment or memory usage.

EXAMPLES

Each package name has a corresponding directory name. In the following examples, the source files are located at `c:\jdk\src\java\awt\*.java`.

Compiling One or More Packages

To compile a package, the source files (`*.java`) for that package must be located in a directory having the same name as the package. If a package name is made up of several identifiers (separated by dots), each identifier represents a different directory. Thus, all java.awt classes must reside in a directory named java\awt\.

First, change to the parent directory of the top-most package. Then run javac, supplying one or more package names.

```
% cd c:\jdk\src
% javac java.awt java.awt.event
```

Compiles the public classes in packages `java.awt` and `java.awt.event`.

Compiling One or More Classes

Change to the directory holding the class. Then run javac, supplying one or more class names.

```
% cd c:\jdk\src\java\awt
% javac Button.java Canvas.java
```

Compiles the two classes.

ENVIRONMENT VARIABLES

CLASSPATH

Used to provide the system a path to user-defined classes. Directories are separated by semi-colons, for example,

```
.;C:\users\dac\classes;C:\tools\java\classes
```

~~~ End of Article ~~~

HelloWorldApp.java Examined

Source	http://freecourseware.uwc.ac.za/freecourseware/information-systems/java-platform-introduction/helloworldapp.java-examined
Date of Retrieval	03/04/2013

```
/**
 * The HelloWorldApp class implements an application that
 * displays "Hello World!" to the standard output.
 */
public class HelloWorldApp {
    public static void main(String[] args) {
        // Display "Hello World!"
        System.out.println("Hello World!");
    }
}
```

Defining a class

The lines of code that produce the object blueprint called a **class** in Java, are called the *class definition*. All the functionality and attributes of the class are defined in the class definition. A Java class definition must contain the word **class** and the name of the class. The simplest possible class definition in Java is:

```
class ClassName {
        ...  }
```

Java is an OOP, and therefore almost everything to do with Java has something to do with an object, or is actually an object. For our purposes every standalone piece of code in Java is a class - including the most simple application such as `HelloWorldApp` above. This is required because object messaging is the means of communication between code blocks, so in order to be able to work coherently all Java programs must consist of class definitions and their instantiation.

The `HelloWorldApp` class definition appears at the top of this page. The line:

```
public class HelloWorldApp {
```

begins the `HelloWorldApp` class definition block. This is a simple class, but no matter how simple a class in Java may be, it still has to conform to the rules for the defining of classes.

This class has a single **method**, *viz.* the **main()** method.
A *main()* method contains the code that is used to control and manipulate other objects in order to produce the results you want the program to produce. Every Java program must have a *main()* method, and it's always defined in the same way:

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

The words `public static void` comprise the *method signature* for the *main()* method:

1. `public` means that the method can be called from outside the class, i.e. by another object or some other code such as the Java Interpreter for example.
2. `static` indicates that the method is a class method and not an instance method, i.e. there is only one copy of the *main()* method, which is used every time the *main()* method is called by any code anywhere.
3. `void` indicates that *main()* doesn't return any value - ever.

How the *main()* method gets called (used)

When the Java interpreter executes a Java application, the very first thing it does is to call the application's *main()* method. The *main()* method then calls all the other methods that are required to make the application function.

If you forget to define a *main()* method in your program, the compiler will complain and refuse to compile the code. Please note that one application can contain definitions for many classes, but only one class in any application should contain a *main()* method. This class is the controlling class for that application. `HelloWorldApp` is a very simple class that in itself is the entire application, and this application calls only one method from another class, namely the *println()* method of the *out* object of the *System* package.

Arguments to the *main()* method

The *main()* method accepts a single argument - an array of elements of type *String* called *args*. Each string in this array is called a *command line element*, and these strings are used by the application to affect the way the application runs without requiring the application to be recompiled. In essence, these strings are switches which tell *main()* how to do its work. For example, you may have written a program that sorts names

alphabetically, and the command line element
`-descending`
would tell *main()* to sort the names in descending order. You have seen this in action when you ran Java from the console with the command line element `-version` in order to have the Java version information returned to the screen. The `HelloWorldApp` class ignores the command line elements, so we won't examine this any further at this point.

Classes and objects in HelloWorldApp

Most Java applications define several objects and contain supporting code that allows the objects to communicate to achieve the desired result. `HelloWorldApp` is so simple that it has no need to define any extra classes. It does however make use of another existing Java class, the *System* class in the *java.lang* package, which is part of the Java API (Application Programming Interface) that is supplied with the Java environment. The *System* class allows your Java program to interact with whichever operating system your computer is running in a standard way, so that there is no need for different methods to be used to communicate with each different system.

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

uses a *class variable*, viz. the *System.out* construct. Notice that `HelloWorldApp` doesn't instantiate the System class, i.e. there is no

```
System myVar = new System();
```

instruction. *System.out* is a class variable and can therefore be called directly from its class without instantiation, because it is associated with the class rather than any instance of the class. *System.out* is an *object variable* because it holds an object - an instance of the Java *PrintStream* object, in fact; *println()* is a method of the *PrintStream* object, and this method writes characters to the standard output stream. Usually this results in something being printed on the screen, as it does in our case - the string "Hello World!" (the argument to the `println()` method) is printed on the screen.

The `HelloWorldApp` class definition is closed off with a closing brace }, and that's all she wrote. If you are feeling a little confused with all this talk of class variables and instantiation of `PrintStream` objects, don't worry - you'll get used to the concepts as we go along, and will soon be understanding and using them confidently yourself after a little practice.

~~~ End of Article ~~~

Session 3: Variables and Operators

Shift and Bitwise Operators

Source	http://enos.itcollege.ee/~jpoial/docs/tutorial/java/nutsandbolts/bitwise.html
Date of Retrieval	21/01/2013

A shift operator performs bit manipulation on data by shifting the bits of its first operand right or left. The next table summarizes the shift operators available in the Java programming language.

Shift Operators		
Operator	**Use**	**Description**
`<<`	`op1 << op2`	Shifts bits of `op1` left by distance `op2`; fills with `0` bits on the right side
`>>`	`op1 >> op2`	Shifts bits of `op1` right by distance `op2`; fills with highest (sign) bit on the left side
`>>>`	`op1 >>> op2`	Shifts bits of `op1` right by distance `op2`; fills with `0` bits on the left side

Each operator shifts the bits of the left operand over by the number of positions indicated by the right operand. The shift occurs in the direction indicated by the operator itself. For example, the following statement shifts the bits of the integer 13 to the right by one position.

```
13 >> 1;
```

The binary representation of the number 13 is 1101. The result of the shift operation is 1101 shifted to the right by one position — 110, or 6 in decimal. The left bits are filled with 0s as needed.

The following table shows the four operators the Java programming language provides to perform bitwise functions on their operands.

Logical Operators		
Operator	**Use**	**Operation**
`&`	`op1 & op2`	Bitwise `AND` if both operands are numbers; conditional `AND` if both operands are boolean

`|`	`op1 | op2`	Bitwise `OR` if both operands are numbers; conditional `OR` if both operands are boolean
`^`	`op1 ^ op2`	Bitwise exclusive `OR` (`XOR`)
`~`	`~op`	Bitwise complement

When its operands are numbers, the `&` operation performs the bitwise `AND` function on each parallel pair of bits in each operand. The `AND` function sets the resulting bit to 1 if the corresponding bit in both operands is 1, as shown in the following table.

The Bitwise AND Function		
Bit in `op1`	**Corresponding Bit in `op2`**	**Result**
0	0	0
0	1	0
1	0	0
1	1	1

Suppose you were to `AND` the values 13 and 12, like this: `13 & 12`. The result of this operation is 12 because the binary representation of 12 is 1100, and the binary representation of 13 is 1101.

```
   1101     //13
 & 1100     //12
 ------
   1100     //12
```

If both operand bits are 1, the `AND` function sets the resulting bit to 1; otherwise, the resulting bit is 0. So, when you line up the two operands and perform the `AND` function, you can see that the two high-order bits (the two bits farthest to the left of each number) of each operand are 1. Thus, the resulting bit after performing the `AND` function is also 1. The low-order bits evaluate to 0 because either one or both bits in the operands are 0.

When both of its operands are numbers, the `|` operator performs the *inclusive or* operation, and `^` performs the *exclusive or* (`XOR`) operation. *Inclusive or* means that if either of the two bits is 1, the result is 1. The following table shows the results of an *inclusive or* operation.

The Bitwise Inclusive OR Function

Bit in op1	Corresponding Bit in op2	Result
0	0	0
0	1	1
1	0	1
1	1	1

Exclusive or means that if the two operand bits are different, the result is 1; otherwise, the result is 0. The following table shows the results of an *exclusive or* operation.

The Bitwise Exclusive OR (XOR) Function		
Bit in op1	**Corresponding Bit in op2**	**Result**
0	0	0
0	1	1
1	0	1
1	1	0

Finally, the complement operator (`~`) inverts the value of each bit of the operand: If the operand bit is 1, the result is 0; if the operand bit is 0, the result is 1. For example, `~1011` (11) is `0100` (4).

Among other things, bitwise manipulations are useful for managing sets of boolean flags. Suppose, for example, that your program had several boolean flags that indicated the state of various components in your program: is it visible, is it draggable, and so on. Rather than define a separate boolean variable to hold each flag, you could define a single variable — `flags` — for all of them. Each bit within `flags` would represent the current state of one of the flags. You would then use bit manipulations to set and to get each flag.

First, set up constants that indicate the various flags for your program. These flags should each be a different power of 2 to ensure that each bit is used by only one flag. Define a variable, `flags`, whose bits would be set according to the current state of each flag. The following code sample initializes `flags` to 0, which means that all flags are `false` (none of the bits are set).

```
static final int VISIBLE = 1;
static final int DRAGGABLE = 2;
static final int SELECTABLE = 4;
static final int EDITABLE = 8;

int flags = 0;
```

To set the visible flag when something became visible, you would use this statement.

```
flags = flags | VISIBLE;
```

To test for visibility, you could then write the following.

```
if ((flags & VISIBLE) == VISIBLE) {
    ...
}
```

Here's the complete Bitwise Demo program that includes this code.

```
public class BitwiseDemo {

    static final int VISIBLE = 1;
    static final int DRAGGABLE = 2;
    static final int SELECTABLE = 4;
    static final int EDITABLE = 8;

    public static void main(String[] args) {
        int flags = 0;

        flags = flags | VISIBLE;
        flags = flags | DRAGGABLE;

        if ((flags & VISIBLE) == VISIBLE) {
            if ((flags & DRAGGABLE) == DRAGGABLE) {
                 System.out.println("Flags are Visible "
                                    + "and Draggable.");
            }
        }

        flags = flags | EDITABLE;

        if ((flags & EDITABLE) == EDITABLE) {
           System.out.println("Flags are now also Editable.");
        }
    }
}
```

Here's the output from this program.

```
Flags are Visible and Draggable.
Flags are now also Editable.
```

~~~ End of Article ~~~

Java Operators

Source	http://diegolemos.net/2012/01/03/java-6-operators/
Date of Retrieval	07/03/2013

As in mathematics, operators in Java are used to produce new values from operands. Choosing the right operator on the right time will help you to write clear code in a fast way.

Assignment Operators

The most common assignment operator is the = operator. This operator simply assign values to variables. Nevertheless, the = operator can be composed with other operators like +, -, *, /, >>, <<, & and | forming the compound operators: +=, -=, *=, /=, >>=, <<=, &= and |=.

```
int i = 0;

int j = 0;

i += 2;          // Which is the same that i = i + 2;

j *= 10 + 4;     // Which is the same that j = j * (10 +
4)
```

Relational Operators

Relational operators are often used on if tests and result in a boolean value (true or false). Java has six relational operators which are: >, >=, <, <=, ==, and !=. All of them can perform comparisons on integers, floating-points or characters. Only the last two (== and !=) are used to perform comparisons on boolean values, or object reference variables.

```
char sex = 'f';

if(sex)

Relational r1 = new Relational();

Relational r2 = new Relational();

if(r1 == r2) {} // false

r1 = r2;        // now they hold the same reference

if(r1 == r2) {} // true
```

instanceof Operator

The instanceof operator verifies if an object is of a particular type. To evaluate the type of the object (on the left), the operator uses a IS-A test for a class or interface (on the right). The IS-A test returns true if the object inherits from the class or implements the interface used on the test. Otherwise, the result is false.

```
class A {}

class B extends A {}

public class InstanceOf {

    public static void main(String... args) {

            A a = new B();

           B b;

          if(a instanceof B) {    // the IS-A test passes

            b = (B)a;   // we can then downcast
```

```
            }

    }

    }
```

Arithmetic Operators

Beyond the basic arithmetic operators +, -, *, and /, there are three other operators also widely used which are:

- the Remainder operator (%);
- the String Concatenation operator (+);
- the Increment and Decrement operators (++ and --, respectively).

The remainder operator % returns the remainder of a division operation.

```
int x = 10;

int y = 4;

int rem = x % y;     // which will be 2
```

Observation: expressions are evaluated from left to right. The operators *, /, and % have higher priority then + and - operators. Parentheses can be used to change the evaluation order.
The operator + can act as addition operator or string concatenation operator, which depends on the operands. If there is a string among the operands, the + becomes a string concatenation operator. If the operands are numbers, the + will perform the addition.

```
int i = 5;

int j = 5;

System.out.println("String" + i);    // will print String5
```

```
System.out.println("String" + (i + j)); // will print
//String10 because the parentheses
```

Finally, the increment and decrement operators will increment or decrement variables by one. The operator can prefix or postfix the operand. When the prefix approach is used, the operator is placed before the variable, and the operation is performed before the variable is used. When the postfix approach is employed, the operator is placed after the variable, and the increment or decrement operation takes place after the variable is used.

```
public class IncrementDecrement {

public static void main(String... args) {

            int i = 0;

                if(++i == 1) {

System.out.println("++i is performed before the test");

        }

System.out.println("i = " + i--); // output : i = 1

    }

}
```

Conditional Operators

The ternary conditional operator will decide which value to assign or return after evaluating a boolean expression. Here is the structure of the conditional operator:

(boolean expression) ? value if true : value if false

The first value is assigned or returned if the result of the expression is true. Otherwise, the second value is used. The parentheses are optionals and it is possible to nest conditional operators.

```
public class ConditionalTest {
public static void main(String... args) {
        int i = 4;
        int j = 5;
        String result = ++i == 4 ? "i=4" :
       (i + j++ > 10) ? "i+j>10" : "i+j    }
}
```

Logical Operators

Logical operators in Java basically evaluate an expression and return a boolean value.

Short-Circuit Logical Operators

Short-circuit operators evaluate boolean expressions and return boolean values. Short-circuit operators are represented as follow:

- || short-circuit OR
- && short-circuit AND

A short-circuit AND (&&) will evaluate the left side of an expression first. If the result is false, the operator will not waste time evaluating the right side and will return false. The right side will be evaluated only if the result of the left side is true. If the result of both sides is true, the hole expression is evaluated as true. If one of them results false, the expression is evaluated as false.

As the short-circuit AND, the short-circuit OR (||) will start evaluating the left side of the expression. Nevertheless, if the result is true the right side will not be analyzed and the result of the whole expression will be true. The right side will be evaluated only if the result of the expression on left side is false. So, if there is one true, the global result is true. If both sides are false, the final result is false.

```
public class ShortCircuitLogicalOperator {
```

```
    public static void main(String... args) {
        if(check(9) && check(11)) {
System.out.println("First IF");
if(false || check(13)) {
System.out.println("Second IF");
        }
if(value % 2 < 2) {
return true;
}
        return false;
}
}
```

The output of the code above is:

```
First IF

Second IF
```

Not Short-Circuit Logical Operators

Not short-circuit operators are represented as follows:

- | non-short-circuit OR
- & non-short-circuit AND

As not short-circuit operators will evaluate logical expressions like the operators && and ||. On the other hand, not short-circuit operators will evaluated ALWAYS both sides of the expression. Non-short-circuit OR (|) will return false if both sides of the logical expression are false. It returns true if one side is true. Non-short-circuit AND (&) returns true only if both sides are evaluated as true. Otherwise, it returns false.

```
1 public class NotShortCircuitLogicalOperator {

2    public static void main(String... args) {
```

```
3         int i = 0;

5         if((i++ < 1 & ++i > 1) | i == 2) {

6             System.out.println("Little puzzle");

7         }

8     }

9 }
```

Boolean Invert Logical Operator

The unary boolean invert (!) operator evaluates only boolean expression and returns its opposite value. So, if the result is true, it gets false and vice-versa.

Exclusive-OR (XOR) Logical Operator

The exclusive-OR (^) operator also used with boolean expressions and it evaluates BOTH sides of an expression. For an exclusive-OR (^) expression be `true`, EXACTLY one operand must be `true`.

```
01     public class ExclusiveOR {

03     public static void main(String... args) {

04     byte size1 = -127;

05     byte size2 = 127;

07    if(checkSize(size1) ^ checkSize(size2)) {

08       System.out.println("There is a size that passes the" + "
check AND a size that does not match. ");

}
```

```
}

    private static boolean checkSize(byte i) {

     if(i < 127) {

         return true;

     }

       return false;

  }

}
```

Bitwise Operators

Bitwise operators are operators that evaluate the bit value of each operand. They are used in expressions with integer values. If the operand is smaller than int, the primitive value will be converted to an int. Following a list with the bitwise operators in Java.

- ~ unary bitwise complement
- & bitwise AND
- | bitwise OR
- ^ bitwise XOR
- Shift operators: <<, >>, and >>>

The unary bitwise complement simply invert the bit value. The bitwise operators AND, OR and XOR evaluate bits using simple logic. The shift operator << shifts bits of to left putting zeros on the right side (low order position). The shift operator >> shifts bits to the right side saving the number sign. In other words, a negative number remains negative after the operation. Finally, the shift operator >>> move bits to right without saving the number sign. In this case, zeros are inserted as most

significant bits (on the left). All the operators can be also used with the assignment (=) operator.
Bitwise operators are generally used to pack a lot of information into a single variable, like masks, flags, etc.

```
public class BitwiseOperators {

    public static void main(String... args) {

        short i = 127;

        long j = 0;

byte k = 85; // binary: 0101 0101

        int l = 99; // binary: 0000 0000 0000 0000 0000 0000 0110
0011

int m = -99; // binary: 1111 1111 1111 1111 1111 1111 1001 1101

        System.out.println("~i  = " + ~i);

        System.out.println("i&j = " + (i&j));

        System.out.println("i|j = " + (i|j));

System.out.println("i^j = " + (i^k));

j |= i;

System.out.println("j (after j |= i) = " + j);

System.out.println("m>>4 = " + (m>>4)); // binary: 1111 1111 1111
1111 1111 1111 1111 1001

System.out.println("m>>>12 = " + (m>>>12)); // binary: 0000 0000
0000 1111 1111 1111 1111 1111
}
}
```

The output of the code above will be:

```
~i  = -128

i&j = 0

i|j = 127

i^j = 42

j (after j |= i) = 127

m>>4 = -7

m>>>2 = 1048575
```

~~~ End of Article ~~~

Java Reference Type Casting

Source	http://javasoft.phpnet.us/res/tut_typecast.html
Date of Retrieval	21/01/2013

In java one object reference can be cast into another object reference. The cast can be to its own class type or to one of its subclass or superclass types or interfaces. There are compile-time rules and runtime rules for casting in java. The casting of object references depends on the relationship of the classes involved in the same hierarchy. Any object reference can be assigned to a reference variable of the type Object, because the Object class is a superclass of every Java class.

There can be 2 types of casting

- Upcasting
- Downcasting

When we cast a reference along the class hierarchy in a direction from the root class towards the children or subclasses, it is a downcast.

When we cast a reference along the class hierarchy in a direction from the sub classes towards the root, it is an upcast. We need not use a cast operator in this case.

The compile-time rules are there to catch attempted casts in cases that are simply not possible. This happens when we try to attempt casts on objects that are totally unrelated (that is not subclass super class relationship or a class-interface relationship)

At runtime a ClassCastException is thrown if the object being cast is not compatible with the new type it is being cast to.

Below is an example showing when a ClassCastException can occur during object casting

```
                //X is a supper class of Y and Z which are siblings.

public class RunTimeCastDemo{
        public static void main(String args[]){
                X x = new X();
                Y y = new Y();
                Z z = new Z();

                X xy = new Y();  // compiles ok (up the hierarchy)
                X xz = new Z();  // compiles ok (up the hierarchy)
//              Y yz = new Z();   incompatible type (siblings)

//              Y y1 = new X();   X is not a Y
//              Z z1 = new X();   X is not a Z

                X x1 =  y;       // compiles ok (y is subclass of X)
```

```
            X x2 =  z;       // compiles ok (z is subclass of X)

            Y y1 = (Y) x;    // compiles ok but produces runtime error
            Z z1 = (Z) x;    // compiles ok but produces runtime error
            Y y2 = (Y) x1;   // compiles and runs ok (x1 is type Y)
            Z z2 = (Z) x2;   // compiles and runs ok (x2 is type Z)
//          Y y3 = (Y) z;     inconvertible types (siblings)
//          Z z3 = (Z) y;     inconvertible types (siblings)

            Object o  = z;
            Object o1 = (Y)o; // compiles ok but produces runtime error

      }
}
```

Casting Object References: Implicit Casting using a Java Compiler

In general an implicit cast is done when an Object reference is assigned (cast) to:

A reference variable whose type is the same as the class from which the object was instantiated.

An Object as Object is a super class of every Java Class.

A reference variable whose type is a super class of the class from which the object was instantiated.

A reference variable whose type is an interface that is implemented by the class from which the object was instantiated.

A reference variable whose type is an interface that is implemented by a super class of the class from which the object was instantiated.

Consider an interface Vehicle, a super class Car and its subclass Ford. The following example shows the automatic conversion of object references handled by the java compiler

```
interface Vehicle {
}
class Car implements Vehicle {
 }

class Ford extends Car {
 }
```

Let c be a variable of type Car class and f be of class Ford and v be an vehicle interface reference. We can assign the Ford reference to the Car variable:

I.e. we can do the following

Example 1

```
c = f; //Ok Compiles fine
```

Where c = new Car();

And, f = new Ford();

The compiler automatically handles the conversion (assignment) since the types are compatible (sub class - super class relationship), i.e., the type Car can hold the type Ford since a Ford is a Car.

Example 2

v = c; //Ok Compiles fine

c = v; // illegal conversion from interface type to class type results in compilation error

Where c = new Car();

And v is a Vehicle interface reference (Vehicle v)

The compiler automatically handles the conversion (assignment) since the types are compatible (class – interface relationship), i.e., the type Car can be cast to Vehicle interface type since Car implements Vehicle Interface. (Car is a Vehicle).

Casting Object References: Explicit Casting

Sometimes we do an explicit cast in java when implicit casts don't work or are not helpful for a particular scenario. The explicit cast is nothing but the name of the new "type" inside a pair of matched parentheses. As before, we consider the same Car and Ford Class

```
class Car {
      void carMethod(){
    }
}
```

```
class Ford extends Car {
      void fordMethod () {
   }
}
```

We also have a breakingSystem() function which takes Car reference (Superclass reference) as an input parameter.

The method will invoke carMethod() regardless of the type of object (Car or Ford Reference) and if it is a Ford object, it will also invoke fordMethod(). We use the instanceof operator to determine the type of object at run time.

```
public void breakingSystem (Car obj) {
   obj.carMethod();
   if (obj instanceof Ford)

      ((Ford)obj).fordMethod ();
}
```

To invoke the fordMethod(), the operation (Ford)obj tells the compiler to treat the Car object referenced by obj as if it is a Ford object. Without the cast, the compiler will give an error message indicating that fordMethod() cannot be found in the Car definition.

The following java shown illustrates the use of the cast operator with references.

Note: Classes Honda and Ford are Siblings in the class Hierarchy. Both these classes are subclasses of Class Car. Both Car and HeavyVehicle Class extend Object Class. Any class that does not explicitly extend some other class will automatically extends the Object by default. This code instantiates an object of the class Ford and assigns the object's reference to a reference variable of type Car. This assignment is allowed as Car is a superclass of Ford.

In order to use a reference of a class type to invoke a method, the method must be defined at or above that class in the class hierarchy. Hence an object of Class Car cannot invoke a method present in Class Ford, since the method fordMethod is not present in Class Car or any of its superclasses. Hence this problem can be colved by a simple downcast by casting the Car object reference to the Ford Class Object reference as done in the program.

Also an attempt to cast an object reference to its Sibling Object reference produces a ClassCastException at runtime, although compilation happens without any error.

```
 class Car extends Object{
        void carMethod() {
   }
}

class HeavyVehicle extends Object{

}

class Ford extends Car {
        void fordMethod () {
                System.out.println("I am fordMethod defined in Class Ford");
   }
}

class Honda extends Car {
        void fordMethod () {
                System.out.println("I am fordMethod defined in Class Ford");
        }
}

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

```
        Car obj = new Ford();
//    Following will result in compilation error
//    obj.fordMethod();        //As the method fordMethod is undefined for the Car Type
//  Following will result in compilation error
//    ((HeavyVehicle)obj).fordMethod();    //fordMethod is undefined in the HeavyVehicle Type
// Following will result in compilation error

        ((Ford)obj).fordMethod();

    //Following will compile and run
//        Honda hondaObj = (Ford)obj;        Cannot convert as they are siblings

  }
}
```

One common casting that is performed when dealing with collections is, you can cast an object reference into a String.

```
import java.util.Vector;

public class StringCastDemo{
        public static void main(String args[]){
                String username = "asdf";
                String password = "qwer";
                Vector v = new Vector();
                v.add(username);
                v.add(password);


//              String u = v.elementAt(0); Cannot convert from object to String
                Object u = v.elementAt(0);  //Cast not done
                System.out.println("Username : " +u);


                String uname = (String) v.elementAt(0); // cast allowed
                String pass = (String) v.elementAt(1); // cast allowed

                System.out.println();
                System.out.println("Username : " +uname);
                System.out.println("Password : " +pass);
        }
}
```

Output

Username : asdf
Username : asdf
Password : qwer

~~~ End of Article ~~

Session 4: Decision-Making Constructs

Using Strings in switch Statements in Java SE 7

Source	http://guchex.com/mrinflictor/post/32/using-strings-in-switch-statements-in-java-se-7
Date of Retrieval	07/03/2013

Switch statements can provide a clean, easy to read way to choose between a set of discrete choice-paths in Java. Something that is commonly switched on is user selection - when there are only a finite number of options, using 'switch' can be much more concise than a long series of if-then-else statements.

For a quick example, here is code that will return the Spanish language equivalent of a given integer between one and five.

```
int number;

...

String spanishName;

switch(number){

        case 1: spanishName = "uno";

                        break;

        case 2: spanishName = "dos";

                        break;

        case 3: spanishName = "tres";

                        break;

        case 4: spanishName = "cuatro";

                        break;

        case 5: spanishName = "cinco";

                        break;

        default: spanishName = "Invalid integer";
```

```
            break;

}

System.out.println(spanishName);
```

As an aside, this example is meant to demonstrate syntax - there are a number of more efficient ways to do the same thing (enums and toString come to mind).
Even to someone not intimately familiar with the Java language, that code is fairly easy to understand - each possibility is represented with its own 'case' statement and a 'default' statement is triggered when the input applies to none of the given cases.

Prior to the most recent release of Java (SE 7), however, you have been unable to switch on a String - it must first be converted with an additional step into an integer by using something like an enum, as shown by the following example.

```
public enum ClassRank

{

    FRESHMEN, SOPHMORE, JUNIOR, SENIOR;

}

CLASSRANK givenRank;

...

String dormName;

switch (ClassRank.valueOf(givenRank)){

        case FRESHMEN:

                dormName = "Howell Hall";

                 break;

        case SOPHMORE:

                dormName = "Towers Hall";

                break;

        case JUNIOR:
```

```
                dormName = "North Avenue Aparments";
                break;
        case SENIOR:
                dormName = "Off-campus party-pad";
                break;
        default:
                dormName = "Woodruff";
                break;
}   System.out.println(dormName);
```

You can see that by using the inherited valueOf(String) method in the enum, we are able to convert a String to an integer within the switch statement itself. Then, rather than using String literals to define the cases, we use the enum values.

With the introduction of SE 7, however, we are now able to use Strings directly as the cases in switch statements. The following provides a short example that is equivalent to the enum example above.

```
        String givenRank = "FRESHMEN";
        String dormName;
        switch (givenRank){
            case "FRESHMEN":
                dormName = "Howell Hall";
                 break;
            case "SOPHMORE":
                dormName = "Towers Hall";
                break;
            case "JUNIOR":
```

```
                dormName = "North Avenue Aparments";
                break;
            case "SENIOR":
                dormName = "Off-campus party-pad";
                break;
            default:
                dormName = "Woodruff";
                break;
        }
        System.out.println(dormName);
```

As you can see, switching on a string is a small change that allows an increase in readability to code where there are a number of paths to be taken based on a discrete number of string choices.

~~~ End of Article ~~~

Session 5: Looping Constructs

Loops - Introduction

Source	http://www.leepoint.net/notes-java/flow/loops/loops.html
Date of Retrieval	21/01/2013

The purpose of loop statements is to repeat Java statements many times. There are several kinds of loop statements in Java.

while statement - Test at beginning

The while statement is used to repeat a block of statements while some condition is true. The condition must become false somewhere in the loop, otherwise it will never terminate.

```
//... While loop to build table of squares.
String result = "";   // StringBuilder would be more efficient.
int i = 1;
while (i <= 20) {
   result = result + i + " squared is " + (i * i) + "\n";
   i++;
}
JOptionPane.showMessageDialog(null, "Tables of squares\n" + result);
```

The following example has an assignment inside the condition. Note that "=" is assignment, not comparison ("=="). This is a common coding idiom when reading input.

```
//... Add a series of numbers.
JOptionPane.showMessageDialog(null, "Enter ints.  Cancel to end");
String valStr;
int sum = 0;
while ((valStr = JOptionPane.showInputDialog(null, "Number?")) != null) {
   sum += Integer.parseInt(valStr.trim());
}
JOptionPane.showMessageDialog(null, "Sum is " + sum);
```

for statement - Combines three parts

Many loops consist of three operations surrounding the body: (1) initialization of a variable, (2) testing a condition, and (3) updating a value before the next iteration. The for loop groups these three common parts together into one statement, making it more readable and less error-prone than the equivalent while loop. For repeating code a known number of times, the for loop is the right choice.

```
//... For loop to build table of squares.
String result = "";   // StringBuilder would be more efficient.
for (int i = 1; i <= 20; i++) {
   result += i + " squared is " + (i * i) + "\n";
}
JOptionPane.showMessageDialog(null, "Tables of squares\n" + result);
```

do..while statement - Test at end

When you want to test at the end to see whether something should be repeated, the do..while statement is the natural choice.

```
String ans;
do {
   . . .
   ans = JOptionPane.showInputDialog(null, "Do it again (Y/N)?");
} while (ans.equalsIgnoreCase("Y"));
```

"foreach" statement - Java 5 data structure iterator

Java 5 introduced what is sometimes called a "for each" statement that accesses each successive element of an array, List, or Set without the bookkeeping associated with iterators or indexing.

```
//... Variable declarations.
JTextArea nameTextArea = new JTextArea(10, 20);
String[] names = {"Michael Maus", "Mini Maus"};

//... Display array of names in a JTextArea.
for (String s : names) {
   nameTextArea.append(s);
   nameTextArea.append("\n");
}
```

Similar to the 'if' statement

There are three general ideas that you will see in many parts of Java.

Braces {} to enclose multiple statements in the body.

Indentation to show the extent of the body clearly.

Boolean (true/false) conditions to control whether the body is executed.

Scope of loop indicated with braces {}

If the body of a loop has more than one statement, you must put the statements inside braces. If there is only one statement, it is not necessary to use braces {}. However, many programmers think it is a good idea to always use braces to indicate the scope of

statements. Always using braces allows the reader to relax and not worry about the special single statement case.

Indentation. All statements inside a loop should be indented one level (eg, 4 spaces), the same as an if statement.

~~~ End of Article ~~~

Performance Improvement Techniques in Loops

Source	http://www.precisejava.com/javaperf/j2se/Loops.htm
Date of Retrieval	21/01/2013

Overview of loops

Loops provide efficient way for repeating a piece of code as many times as required. Java has three types of loop control structures they are: for loop, while loop and do-while loop. The for loop is used when we know in advance how many iterations are required. The while loop is used when we do not know in advance the number of iterations required so each time before entering the loop the condition is checked and if it is true then the loop is executed. The do-while loop is always executed at least once and then the condition is checked at the end of the loop. Loops have a considerable effect on the performance of the program let us look at some points that focus on optimizing while using the loop control structures.

Optimization techniques in loops

- Always use an int data type as the loop index variable whenever possible because it is efficient when compared to using byte or short data types. Because when we use byte or short data type as the loop index variable they involve implicit type cast to int data type.
- When using arrays it is always efficient to copy arrays using System.arraycopy() than using a loop. The following example shows the difference

```
package com.performance.loop;
 // This class tests the loop copy versus System.arraycopy()
 public class loopTest1{
      public static void main(String s[]){
                           long start,end;
            int[] a=new int[2500000];
            int[] b= new int[2500000];
            for(int i=0;i<a.length;i++){
                  a[i]=i;
            }
            start=System.currentTimeMillis();
                              for(int j=0;j<a.length;j++){
            b[j]=a[j];
            }
            end=System.currentTimeMillis();
            System.out.println(end-start + " milli seconds for loop copy
       ");
                           int[] c= new int[2500000];
            start=System.currentTimeMillis();
```

```
            System.arraycopy(a,0,c,0,c.length);

            end=System.currentTimeMillis();

            System.out.println(end-start + " milli seconds for
       System.arraycopy() ");

      }

}
```

The output is

```
110 milli seconds for loop copy

50 milli seconds for System.arraycopy()
```

- Always avoid anything that can be done outside of the loop like method calls, assigning values to variables, or testing for conditions.
- Method calls are very costly and you only make it worse by putting them in a loop. So as far as possible avoid method calls in a loop.
- It is better to avoid accessing array elements in a loop the better option would be to use a temporary variables inside the loop and modify the array values out of the loop. It is fast to use a variable in a loop than accessing an array element.
- Try to compare the terminating condition with zero if you use non-JIT or HotSpot virtual machine, here is an example to prove the point. JIT or HotSpot virtual machines are optimized for general loops so you do not have to bother about the terminating condition.

```
package com.performance.loop;
// Note that this class may give different results in latest JDK
versions as discussed above.
 public class loopTest2{

      public static void main(String s[]){

            long start,end;

            int[] a=new int[2500000];

            start=System.currentTimeMillis();

            for(int i=0;i<a.length;i++){

            a[i]+=i;

            }

            end=System.currentTimeMillis();

            System.out.println(end-start + " millis with i<a.length ");

                        int[] b=new int[2500000];
                        start=System.currentTimeMillis();
            for(int i=b.length-1;i>=0;i--){
            b[i]+=i;

            }

            end=System.currentTimeMillis();

            System.out.println(end-start + " millis with i>=0");

       }
```

```
}
```

The output is

```
100 millis with i<250000

60 millis with i>=0
```

- Avoid using method calls in loops for termination condition this is costly instead use a temporary variable and test the loop termination with the temporary variable.
- When using short circuit operators to test for loop termination tests always put the expression that will most likely evaluate to false at extreme left. This saves all the following expressions from being tested in case there is an && operator and if there are only || operators then put the expression which is most likely to evaluate to true in the extreme left.
- Avoid using try-catch inside the loops instead place the loops inside the try-catch for better performance

Key Points

1. Use integer as loop index variable.
2. Use System.arraycopy() to copy arrays.
3. Avoid method calls in loops.
4. It is efficient to access variables in a loop when compared to accessing array elements.
5. Compare the termination condition in a loop with zero if you use non-JIT or HotSpot VMs.
6. Avoid using method calls to check for termination condition in a loop
7. When using short circuit operators place the expression which is likely to evaluate to false on extreme left if the expression contains &&.
8. When using short circuit operators place the expression which is likely to evaluate to true on extreme left if the expression contains only ||.
9. Do not use exception handling inside loops.

~~~ End of Article ~~~

Session 6: Classes and Objects

Abstraction

Source	http://cnx.org/content/m11785/latest/
Date of Retrieval	07/03/2013

Abstraction is the process of hiding the details and exposing only the essential features of a particular concept or object. Computer scientists use abstraction to understand and solve problems and communicate their solutions with the computer in some particular computer language. We illustrate this process by way of trying to solve the following problem using a computer language called Java.

Problem: Given a rectangle 4.5 ft wide and 7.2 ft high, compute its area.

We know the area of a rectangle is its width times its height. So all we have to do to solve the above problem is to multiply 4.5 by 7.2 and get the answer. The question is how to express the above solution in Java, so that the computer can perform the computation.

Data Abstraction

The product of 4.5 by 7.2 is expressed in Java as: 4.5 * 7.2. In this expression, the symbol * represents the multiplication operation. 4.5 and 7.2 are called number ***literals***. Using DrJava, we can type in the expression 4.5 * 7.2 directly in the interactions window and see the answer.

Now suppose we change the problem to compute the area of a rectangle of width 3.6 and height 9.3. Has the original problem really change at all? To put it in another way, has the ***essence*** of the original problem changed? After all, the formula for computing the answer is still the same. All we have to do is to enter 3.6 * 9.3. What is it that has ***not*** change (the ***invariant***)? And what is it that has changed (the ***variant***)?

Type Abstraction

The problem has not changed in that it still deals with the same geometric shape, a rectangle, described in terms of the same dimensions, its width and height. What vary are simply the values of the width and the height. The formula to compute the area of a rectangle given its width and height does not change:

width * height

It does not care what the actual specific values of width and height are. What it cares about is that the values of width and height must be such that the multiplication operation makes sense. How do we express the above invariants in Java?

We just want to think of the width and height of a given rectangle as elements of the set of real numbers. In computing, we group values with common characteristics into a set and called it a ***type***. In Java, the type ***double*** is the set of real numbers that are implemented inside the computer in some specific way. The details of this internal representation is immaterial for our purpose and thus can be ignored. In addition to the type double, Java provides many more pre-built types such as ***int*** to represent the set of integers and ***char*** to represent the set of characters. We will examine and use them as their need arises in future examples. As to our problem, we only need to restrict ourselves to the type double.

We can define the width and the height of a rectangle as double in Java as follows.

```
double width;
double height;
```

The above two statements are called ***variable*** definitions where width and height are said to be variable names. In Java, a variable represents a memory location inside the computer. We define a variable by first declare its type, then follow the type by the name of the variable, and terminate the definition with a semi-colon. This is a Java syntax rule. Violating a syntax rule constitutes an error. When we define a variable in this manner, its associated memory content is initialized to a default value specified by the Java language. For variables of type double, the default value is 0.

FINGER EXERCISE:

Use the interactions pane of DrJava to evaluate width and height and verify that their values are set to 0.

Once we have defined the width and height variables, we can solve our problem by writing the expression that computes the area of the associated rectangle in terms of width and height as follows.

```
width * height
```

Observe that the two variable definitions together with the expression to compute the area presented in the above directly translate the description of the problem -two real numbers representing the width and the height of a rectangle- and the high-level thinking of what the solution of the problem should be -area is the width times the height. We have just expressed the invariants of the problem and its solution. Now, how do we vary width and height in Java? We use what is called the ***assignment*** operation. To assign the value 4.5 to the variable width and the value 7.2 to the variable height, we write the following Java assignment statements.

```
width = 4.5;
```

```
height = 7.2;
```

The syntax rule for the assignment statement in Java is: first write the name of the variable, then follow it by the equal sign, then follow the equal sign by a Java expression, and terminate it with a semi-colon. The semantic (i.e. meaning) of such an assignment is: evaluate the expression on the right hand side of the equal sign and assign the resulting value into the memory location represented by the variable name on the left hand side of the equal side. It is an error if the type of the expression on the right hand side is not ***a subset*** of the type of the variable on the left hand side.

Now if we evaluate width * height again (using the Interactions Window of DrJava), we should get the desired answer. Life is good so far, though there is a little bit of inconvenience here: we have to type the expression width * height each time we are asked to compute the area of a rectangle with a given width and a given height. This may be OK for such a simple formula, but what if the formula is something much more complex, like computing the length of the diagonal of a rectangle? Re-typing the formula each time is quite an error-prone process. Is there a way to have the computer memorize the formula and perform the computation behind the scene so that we do not have to memorize it and rewrite it ourselves? The answer is yes, and it takes a little bit more work to achieve this goal in Java.

What we would like to do is to build the equivalent of a black box that takes in as inputs two real numbers (recall type ***double***) with a button. When we put in two numbers and depress the button, "magically" the black box will compute the product of the two input numbers and spit out the result, which we will interpret as the area of a rectangle whose width and height are given by the two input numbers. This black box is in essence a specialized calculator that can only compute one thing: the area of a rectangle given a width and a height. To build this box in Java, we use a construct called a class, which looks like the following.

```
class AreaCalc {
   double rectangle(double width, double height) {
      return width * height;
   }
}
```

What this Java code means is something like: `AreaCalc` is a ***blue print*** of a specialized computing machine that is capable of accepting two input `doubles`, one labeled width and the other labeled height, computing their product and returning the result. This computation is given a name: `rectangle`. In Java parlance, it is called a ***method*** for the class `AreaCalc`.

Here is an example of how we use AreaCalc to compute area of a rectangle of width 4.5 and height 7.2. In the Interactions pane of DrJava, enter the following lines of code.

```
AreaCalc calc = new AreaCalc();
calc.rectangle(4.5, 7.2)
```

The first line of code defines `calc` as a variable of type `AreaCalc` and assign to it an ***instance*** of the class `AreaCalc`. ***new*** is a keyword in Java. It is an example of what is called a class operator. It operates on a class and creates an instance (also called ***object***) of the given class. The second line of code is a call to the object `calc` to perform the `rectangle` task where `width` is assigned the value 4.5 and `height` is assigned the value 7.2. To get the area of a 5.6 by 8.4 rectangle, we simply use the same calculator `calc` again:

```
calc.rectangle(5.6, 8.4);
```

So instead of solving just one problem -given a rectangle 4.5 ft wide and 7.2 ft high, compute its area- we have built a "machine" that can compute the area of ***any*** given rectangle. But what about computing the area of a right triangle with height 5 and base 4? We cannot simply use this calculator. We need another specialized calculator, the kind that can compute the area of a circle.

There are at least two different designs for such a calculator.

- create a new class called AreaCalc2 with one method called ***rightTriangle*** with two input parameters of type ***double***. This corresponds to designing a different area calculator with one button labeled ***rightTriangle*** with two input slots.
- add to AreaCalc a method called rightTriangle with two input parameters of type ***double***. This corresponds to designing an area calculator with two buttons: one labeled ***rectangle*** with two input slots and the other labeled ***rightTriangle***, also with two input slots.

In either design, it is the responsibility of the calculator user to pick the appropriate calculator or press the appropriate button on the calculator to correctly obtain the area of the given geometric shape. Since the two computations require exactly the same number of input parameters of exactly the same type, the calculator user must be careful not get mixed up. This may not be too much of an inconvenience if there are only two kinds of shape to choose from: rectangle and right triangle. But what if the user has to choose from hundreds of different shapes? or better yet an ***open-ende*** number of shapes? How can we, as programmers, build a calculator that can handle an ***infinite*** number of shapes? The answer lies in ***abstraction***. To motivate how conceptualize the problem, let us digress and contemplate the behavior of a child!

Modeling a Person

For the first few years of his life, Peter did not have a clue what birthdays were, let alone his own birth date. He was incapable of responding to your inquiry on his birthday. It was his parents who planned for his elaborate birthday parties months in advance. We can think of Peter then as a rather "dumb" person with very little intelligence and capability. Now Peter is a college student. There is a piece of memory in his brain that stores his birth date: it's September 12, 1985! Peter is now a rather smart person. He can figure out how many more months till his next birthday and e-mail his wish list two

months before his birth day. How do we model a "smart" person like Peter? Modeling such a person entails modeling

- a birth date and
- the computation of the number of months till the next birth day given the current month.

A birth date consists of a month, a day and a year. Each of these data can be represented by an integer, which in Java is called a number of type ***int***. As in the computation of the area of a rectangle, the computation of the number of months till the next birth day given the current month can be represented as a method of some class. What we will do in this case that is different from the area calculator is we will lump both the data (i.e. the birth date) and the computation involving the birth date into one class. The grouping of data and computations on the data into one class is called encapsulation. Below is the Java code modeling an intelligent person who knows how to calculate the number of months before his/her next birth day. The line numbers shown are there for easy referencing and are not part of the code.

```
1  public class Person {
2      /**
3      * All data fields are private in order to prevent code
outside of this
4      * class to access them.
5      */
6      private int _bDay;   // birth day
7      private int _bMonth; // birth month; for example, 3
means March.
8      private int _bYear;  // birth year

9       /**
10     * Constructor: a special code used to initialize the
fields of the class.
11     * The only way to instantiate a Person object is to call
new on the constructor.
12     * For example: new Person(28, 2, 1945) will create a
Person object with
13     * birth date February 28, 1945.
14     */
15     public Person(int day, int month, int year) {
16         _bDay = day;
17         _bMonth = month;
18         _bYear = year;
19     }

20     /**
21     * Uses "modulo" arithmetic to compute the number of
months till the next
22     * birth day given the current month.
```

```
23     * @param currentMonth an int representing the current
month.
24     */
25     public int nMonthTillBD(int currentMonth) {
26         return (_bMonth - currentMonth + 12) % 12;
27     }
28 }
```

We now explain what the above Java code means.

- line 1 defines a class called Person. The opening curly brace at the end of the line and the matching closing brace on line 28 delimit the contents of class Person. The key word ***public*** is called an ***access specifier*** and means all Java code in the system can reference this class.
- lines 2-5 are comments. Everything between /* and */ are ignored by the compiler.
- lines 6-8 define three integer variables. These variables are called ***fields*** of the class. The key word private is another access specifier that prevents access by code outside of the class. Only code inside of the class can access them. Each field is followed by a comment delimited by // and the end-of-line. So there two ways to comment code in Java: start with /* and end with */ or start with // and end with the end-of-line.
- lines 9-14 are comments.
- lines 15-19 constitute what is called a ***constructor***. It is used to initialize the fields of the class to some particular values. The name of the constructor should spell exactly like the class name. Here it is ***public***, meaning it can be called by code outside of the class Person via the operator new. For example, `new Person(28, 2, 1945)` will create an instance of a Person with _bDay = 28, _bMonth = 2 an d_bYear = 1945.
- lines 20-24are comments.
- line 23 is a special format for documenting the parameters of a method. This format is called the javadoc format. We will learn more about javadoc in another module.
- lines 25-27 constitute the definition of a method in class Person.

- line 26 is the formula for computing the number of months before the next birthday using the remainder operator %. x % y gives the remainder of the integer division between the dividend x and the divisor y.

~~~ End of Article ~~~

GC with Automatic Resource Management in Java 7

Source	http://www.javacodegeeks.com/2011/08/gc-with-automatic-resource-management.html
Date of Retrieval	07/03/2013

This post provides a brief overview of a new feature introduced in Java 7 called Automatic Resource Management or ARM. The post delves how ARM tries to reduce the code that a developer has to write to efficiently free the JVM heap of allocated resources.
One of the sweetest spots of programming in the Java programming language is automatic handling of object de-allocation. In Java world this is more popularly known as garbage collection; it basically means that developers do not have to worry about de-allocating the object allocated by their code. As soon as a developer is finished with using the object he can nullify all references to the object and then the object becomes eligible for garbage collection.
Garbage collection has a flip side to it however. Unlike in C/C++ where the coder has complete control of memory allocation and de-allocation (malloc, free, new, delete etc), in Java the developer does not have significant control over the process of de-allocation of objects. The JVM manages the process of garbage collecting of unused objects and it is really up to the whims of the JVM when to run a cycle of garbage collection. True, there are method calls like System.gc() or Runtime.getRuntime().gc() that indicates that garbage collection will be run, but these methods merely serve to remind the JVM that - ”maybe you need to run a garbage collection now, just a suggestion, no pressure!”. The JVM is fully authorized to disregard such requests and is coded to run garbage collection only when it really sees fit. Hence in practice, developers are always advised not to build their program logic believing System.gc() or Runtime.getRuntime().gc() will trigger a full garbage collection.
There is no denying how much good automatic garbage collection has done to enhance the productivity of developers. However there are some corner cases where garbage collection is not sufficient to maintain a “clean” heap, free of unused objects. Especially if the objects deal with some form of native resources that is served by the underlying operating system. These objects include, but are not limited to IO streams, database connections etc. For these kind of objects developers must release the resources explicitly. Typically these are done through try-catch blocks.
Let us look at a small example that closes an InputStream after finishing the processing of the stream:

```
01 InputStream in = null;
02
03 try
04 {
05     in = new FileInputStream(new File("test.txt");
06     //do stuff with in
07 }
08 catch(IOException ie)
09 {
10     //SOPs
11 }
12 finally
13 {
14     //do cleanup
```

```
15 }
```

The above looks good and clean; however as soon as we try to close the input stream via in.close() in the finally block, we need to surround it with a try-catch block that catches the checked exception, IOException. Thus the code sample transforms to:

```
01 InputStream in = null;
02
03 try
04 {
05     in = new FileInputStream(new File("test.txt"));
06     //do stuff with in
07 }
08 catch(IOException ie)
09 {
10     //SOPs
11 }
12 finally
13 {
14     try
15     {
16         in.close();
17     }
18     catch(IOException ioe)
19     {
20         //can't do anything about it
21     }
22 }
```

Now the above code looks bloated, and with multiple kinds of checked exceptions in different hierarchy, we need more catch clauses. Very soon the code becomes lengthy and difficult to maintain, not to mention the code losing its initial clean and no-nonsense look that even appealed to the eye.

But there is a good news.

Java 7 makes this easier with the new try-catch block. With this feature we can avoid the finally block itself. This is how we do it:

```
1 try(InputStream in = new FileInputStream(new File("test.txt"))
2 {
3     //do stuff with in
4 }
5 catch(IOException ie)
6 {
7     //SOPs
8 }
```

The above block of code will do the cleanup part itself. This is made possible by the introduction of a new interface, java.lang.AutoCloseable which defines a single method, void close() throws Exception. Objects which are subtypes of this interface can be automatically close() using the above syntax. The above feature is applicable to objects of any class that implement the AutoCloseable interface.

The best part is that even if we initialize multiple AutoCloseable instances in the try() block, it will call the close() method for all the objects, even if some close() method on some object throw any exception.

Coming to the handling of the exceptions, if there was any IOExceptions in our try block as well as in the implicit finally block (where the AutoCloseables are actually being closed), the exception thrown will be the one that was thrown in the try block rather than the one in the implicit finally block.
However we can still have the details of the implicit finally block's exception from the method Throwable.getSuppressed() which is added as a new method in Java 7.

~~~ End of Article ~~~

Session 7: Methods and Access Specifiers

Access Modifiers

Source	http://en.wikibooks.org/wiki/Java_Programming/Scope
Date of Retrieval	04/01/2013

Access modifiers

You surely would have noticed by now, the words `public`, `protected` and `private` at the beginning of class's method declarations used in this book. These keywords are called the **access modifiers** in the Java language syntax, and they define the scope of a given item.

For a class

- If a class has **public** visibility, the class can be referenced by anywhere in the program.
- If a class has **package** visibility, the class can be referenced only in the package where the class is defined.
- If a class has **private** visibility, (it can happen only if the class is defined nested in another class) the class can be accessed only in the outer class.

For a variable

- If a variable is defined in a **public** class and it has **public** visibility, the variable can be referenced anywhere in the application through the class it is defined in.
- If a variable has **protected** visibility, the variable can be referenced only in the sub-classes and in the same package through the class it is defined in.
- If a variable has **package** visibility, the variable can be referenced only in the same package through the class it is defined in.
- If a variable has **private** visibility, the variable can be accessed only in the class it is defined in.

For a method

- If a method is defined in a **public** class and it has **public** visibility, the method can be called anywhere in the application through the class it is defined in.
- If a method has **protected** visibility, the method can be called only in the sub-classes and in the same package through the class it is defined in.
- If a method has **package** visibility, the method can be called only in the same package through the class it is defined in.
- If a method has **private** visibility, the method can be called only in the class it is defined in.

For an interface

The interface methods and interfaces are always `public`. You do not need to specify the access modifier it will default to `public`. For clarity it is considered a good practice to put the `public` keyword.

The same way all member variables defined in the Interface by default will become `static final` once inherited in a class.

Summary

	Class	Nested class	Method, or Member variable	Interface	Interface method signature
`public`	visible from anywhere	same as its class	same as its class	visible from anywhere	visible from anywhere
`protected`	N/A	its class and its subclass	its class and its subclass, and from its package	N/A	N/A
package	only from its package	only from its package	only from its package	N/A	N/A
`private`	N/A	only from its class	only from its class	N/A	N/A

The cases in blue are the default.

~~~ End of Article ~~~

Overloading

Source	http://freecourseware.uwc.ac.za/freecourseware/information-systems/java-platform-introduction/overloading
Date of Retrieval	04/01/2013

When implementing a Java class, it sometimes becomes necessary to write more than one constructor for the class. Take our `BankAccount` class for example. This class has two constructors:

```
//default bank account with zero initial balance
BankAccount()
//bank account with initial balance
BankAccount(double initialBalance)
```

Java decides which constructor to use based on the parameters supplied by the user. When we write more than one constructor for the same class, we say that we are *overloading* the constructor.

Other methods can be overloaded as well.

Class *Die*

Consider a standard die with six faces and spots on each face - the spots are arranged on the die so that any two opposite faces add up to 7. The die can be thrown to expose the number of spots on the uppermost face. This is a way to randomly choose any number between 1 and 6, because each number on the die has an equal chance of being thrown, provided the die is fair.

We can model a die with a Java class, because Java has the `Math.random()` method, which generates a random number between 0 and 1, exclusive. To produce a random integer between 1 and a second higher integer, we must multiply the value generated by `Math.random()` by the higher integer, add 1, and then cast the result to an integer because `Math.random()` generates a`double`:

```
//random number between 1 & 6
int result = (int)(Math.random() * 6 + 1);
```

Returning to class `Die`:

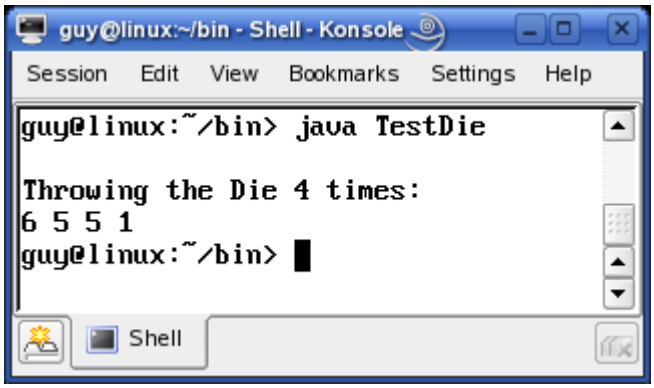
```
/**
 * A Die can be thrown to produce a random number between 1 and 6.
 */
public class Die {

    private final int SIDES = 6;

    /**
     * Constructs a six-sided Die.
     */
    public Die(){
    }

    /**
     * Throws the Die.
     * @return a random integer from 1 to 6, inclusive.
     */
    public int throwMe() {
        int result = (int)(Math.random() * SIDES + 1);
        return result;
    }

}
/**
 * Tests the Die class.
 */
public class TestDie {
    public static void main(String[] args){
        Die myDie = new Die();
        System.out.println("\nThrowing the Die 4 times:\n");
        for(int k=1; k<=4; k++){
            System.out.print(myDie.throwMe() + " ");
        }
        System.out.println();
    }
}
```



Note the use of `System.out.print(myDie.throwMe() + " ");` - this method always prints its argument on the current line, so each throw of the `Die` is printed onto the same line, separated by a space.

This is fine as far as it goes, but a die can have more or less than 6 faces - in some role playing games 10- or 12-sided dice are used. If we want to build a Java class that represents a

die, we have to take all the possibilities into account. To make our class more complete, we should allow for all possibilities. Also, sometimes we might want to throw the `Die` more than once, and we might want to know what the total is of all the throws. Consider the revised `Die` class below:

```
/**
 * A Die can be thrown one or more times to produce random integers from 1
 * to the number of sides of the die, inclusive.
 */
public class Die {

    private int sides;
    private int pipCount;

    /**
     * Constructs a new default Die with 6 sides.
     */
    public Die() {
        sides = 6;
        pipCount = 0;
    }

    /**
     * Constructs a new Die with integer number of sides numSides.
     * @param numSides the number of sides to the Die.
     */
    public Die(int numSides) {
        sides = numSides;
        pipCount = 0;
    }

    /**
     * Throws the Die once.
     * @return a random integer from 1 to the number of sides, inclusive.
     */
    public int throwMe() {
        int result = (int)(Math.random() * sides + 1);
        pipCount = result;
        return result;
    }
```

```
    /**
     * Throws the Die more than once in the same turn.
     * @param times the number of times die is to be thrown
     * @return a string containing <code>times</code> random integer values
     * from 1 to the number of sides inclusive, in the order in which the throws
     * occurred.
     */
     public String throwMe(int times) {
        String results = "";
        int hold = 0;
        pipCount = 0;
        for(int k = 0; k < times; k++){
            hold = (int)(Math.random() * sides + 1);
            results += hold + " ";
            pipCount += hold;
        }
        return results;
     }

     /**
     * Accessor for the total pip count of the last (series of) throw(s).
     * @return the total pip count
     */
     public int getPipCount(){
        return pipCount;
     }

    /**
     * Accessor for the number of sides the die has.
     * @return the number of sides
     */
    public int getSides(){
        return sides;
    }

}
/**
 * Tests the Die class.
 */

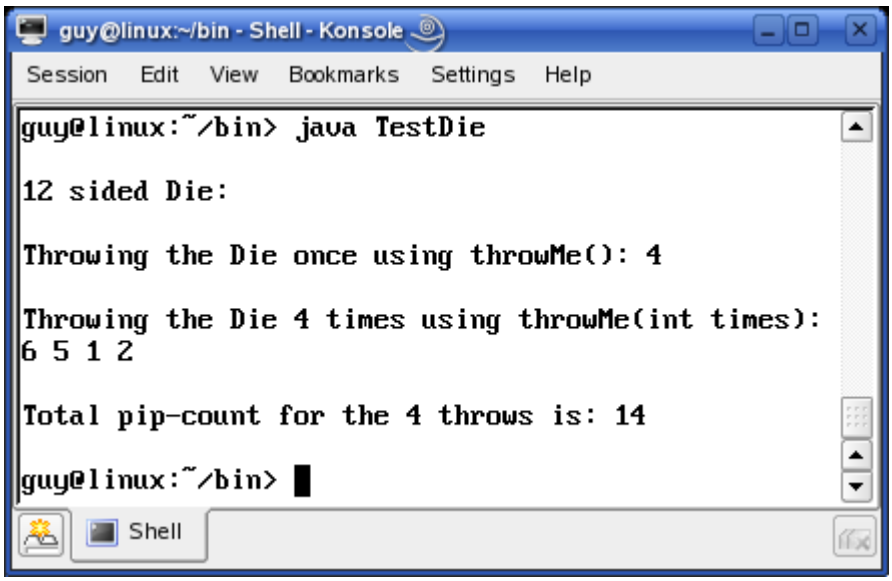
 public class TestDie {

    public static void main(String[] args){

        Die myDie = new Die(12);
        System.out.println("\n12-sided die:\n");
        System.out.println("Throwing die once using throwMe(): " +
                           myDie.throwMe());
        System.out.println("\nThrowing die 4 times using throwMe(int times):\n" +
                           myDie.throwMe(4));
        System.out.println("\n\nTotal pip count for the 4 throws is: " +
                           myDie.getPipCount());

    }

 }
```



The `Die` class now has an overloaded constructor and an overloaded `throwMe()` method. Overloaded methods carry the same name, but have different parameters and possibly different return types (a constructor is a special kind of method). In this case we have:

```
//default Die
public Die(){
   sides = 6;
   pipCount = 0;
}

//n-sided Die
public Die(int numSides){
   sides = numSides;
   pipCount = 0;
}
```

When you place a call to the class constructor, Java sees two constructors of the same name. It then checks to see what the parameters of the two constructors are, and will base its decision on which constructor to use by the arguments and their data types present in the call to the constructor. In the `TestDie` class above,

`Die myDie = new Die(12);`

calls the second of the two constructors because it carries an integer argument - the first constructor has no parameters, so Java cannot use that one.

In the case of the `throwMe()` method, the situation is essentially similar - we have one method that takes no arguments, and one which takes an integer argument; Java will select which method to use based on the arguments you supply in the method call. Note that the two `throwMe()` methods apart from having different parameters, in this case have different return types: the first returns an integer, the second a string.

```
public int throwMe() { }
```

```
public String throwMe(int times) { }
```

Factors which affect the choice of overloaded methods are:

1. The number of parameters
2. The data type(s) of parameters

Note that merely changing the return data type is not sufficient to overload a method, and the Java compiler will not allow overloaded methods with the same parameters and merely different return data types.

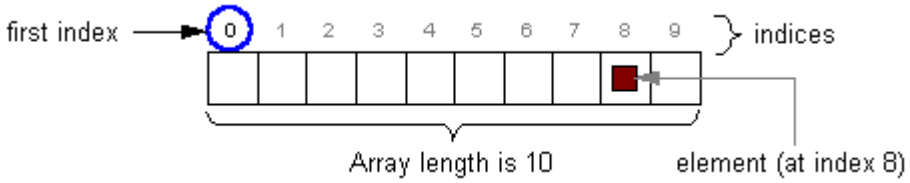
~~~ End of Article ~~~

Session 8: Arrays and Strings

Java Arrays

Source	http://freecourseware.uwc.ac.za/freecourseware/information-systems/java-platform-introduction/java-arrays
Date of Retrieval	04/01/2013

An *array* is a structure that holds multiple values of the same type. The length of an array is established when the array is created (at runtime). After creation, an array is a fixed-length structure. If you need a structure that can change its size dynamically you cannot use an array. Java has a `Collections` framework that has dynamic collection objects such as a `Vector`, but they are outside the scope of this tutorial.



An *array element* is one of the values within an array and is accessed by its position within the array. Arrays are objects in Java, so an array variable merely contains a reference to an array object.

Java arrays may hold data of a single data type only - you may not store strings and integers in the same array, for example. When declaring an array variable the data type followed by a set of square brackets is used:

```
//declare 5-element integer array
int[] myArray = new int[5];
```

The numeral 5 in `new int[5]` sets the length (number of elements) of the array. Note that Java arrays are zero-based, i.e. the indices are numbered from zero to the length of the array minus 1 (see image above). An array may be declared and directly initialized in the following way:

```
//declare and initialize 4-element string array
String[] anArray = {"one","two","three","four"};
```

Accessing an array element is done by using the index number of the element in square brackets after the name of the array variable:

```
//extract 3rd element of array
String hold = anArray[2];
//assign value to 3rd element of array
anArray[2] = "blue";
```

The size of an array can be accessed by the `length` property of the array:

```
int len = anArray.length;
```

Note that unlike the `length()` method of the `String` object, the length of an array is a property; this is often forgotten when writing code, and errors ensue:

```
int len = myArray.length();  // Error!
```

ARRAYS OF OBJECTS

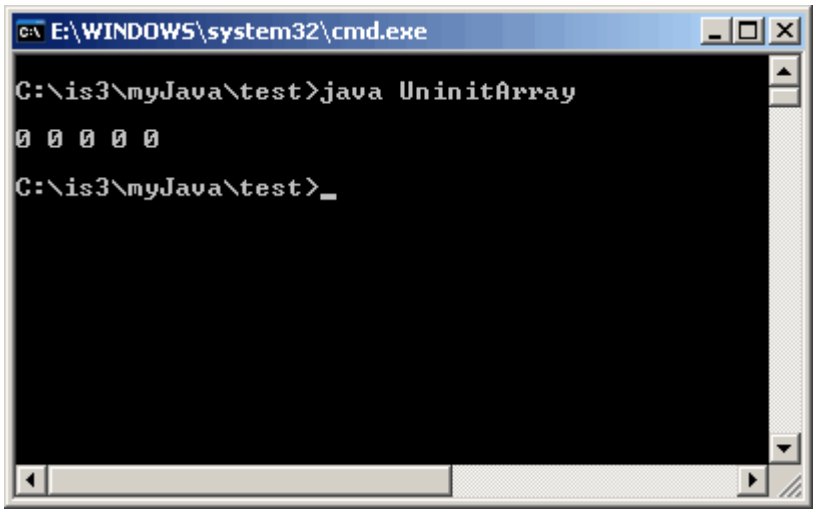
An array of objects is possible in Java. For example, an array of strings is an array of *String* objects. There is a caveat however - defining an array of objects is not sufficient to initialize the array:

```
String [] stringArray = new String[4];
for (k = 0; k < stringArray.length; k++) {
   System.out.println(stringArray[k].toUpperCase());  //Error!
}
```

In the code block above, `stringArray` exists and has a length of 4; the array is completely empty however, and the call to the *String* object method `toUpperCase()` generates a`NullPointerException`. An array of objects must therefore be populated with objects after declaration if any array methods or manipulation of array values is to be attempted.

In contrast to arrays of objects, number arrays are automatically initialized to hold zeros in all elements at instantiation. In the class `UninitArray` below a 5-element integer array is declared, but no values are specifically assigned to any of the elements. In the `for(){}` loop each array element is accessed and printed to the same line in the console - as you can see, 5 zeros are the output.

```
/**
 * Tests declaring an integer array and accessing its elements
 * without formally initialising them.
 */
public class UninitArray {
    public static void main(String[] args){
        int[] myArray = new int[5];
        System.out.println();
        for(int k = 0; k < myArray.length; k++){
            System.out.print(myArray[k] + " ");
        }
        System.out.println();
    }
}
```



ARRAYS OF ARRAYS

In reality arrays are not confined to a single dimension. An array can be built which produces a matrix - like this:

Index	0	1	2	3
0	233	142	209	99
1	258	411	116	368
2	400	347	236	539

The red numerals are the column and row indices. This is a 2-dimensional array - it has rows and columns, specifically 3 rows and 4 columns. The blue value 209 is addressed by using the row index followed by the column index: [0][2]

In Java, multi-dimensional arrays are not supported; however, arrays of arrays achieve the same end, i.e. we produce the matrix by making a 1-dimensional array have other arrays as its

elements. In the case above we do this by defining a 3-element array and then placing each of the 4-element arrays that make up the three rows of the matrix, into the elements of the 3-element array. The sub-arrays within an array must be explicitly created. Note that the length of the primary array *must* be specified at declaration if the array is not explicitly populated at declaration; the length of the secondary arrays can be determined when the primary array is populated. To declare an array of arrays:

```
int[][] arrOfArrays = new int[4][];
                OR
String[][] myArray = {
   {"ab","cd","ef"}, {"gh","ij"}, {"kl","mn","op","qr"}
};  //explicitly populated at declaration
```

To access a value in an array of arrays:

```
//accesses the 3rd value in the 2nd sub-array
int hold = intArray[1][2];
```

~~~ End of Article ~~~

Java Collections Framework

Source	http://en.wikipedia.org/wiki/Java_collections_framework
Date of Retrieval	04/01/2013

The **Java collections framework** (JCF) is a set of classes and interfaces that implement commonly reusable collection data structures.

Although it is a framework, it works in a manner of a **library**. The JCF provides both interfaces that define various collections and classes that implement them.

History

Collection implementations in pre-JDK 1.2 versions of the Java platform included few data structure classes, but did not contain a collections framework. The standard methods for grouping Java objects were via the array, the `Vector`, and the `Hashtable` classes, which unfortunately were not easy to extend, and did not implement a standard member interface.

To address the need for reusable collection data structures, several independent frameworks were developed, the most used being Doug Lea's *Collections package*, and ObjectSpace *Generic Collection Library* (JGL), whose main goal was consistency with the C++ Standard Template Library (STL).

The collections framework was designed and developed primarily by Joshua Bloch, and was introduced in JDK 1.2. It reused many ideas and classes from Doug Lea's *Collections package*, which was deprecated as a result. Sun choose not to use the ideas of JGL, because they wanted a compact framework, and consistency with C++ was not one of their goals.

Doug Lea later developed a concurrency package, comprising new Collection-related classes. An updated version of these concurrency utilities was included in JDK 5.0 as of JSR 166.

Architecture

Almost all collections in Java are derived from the `java.util.Collection` interface. Collection defines the basic parts of all collections. The interface states the add() and remove() methods for adding to and removing from a collection respectively. Also required is the toArray() method, which converts the collection into a simple array of all the elements in the collection. Finally, the contains() method checks if a specified element is in the collection. The Collection interface is a subinterface of `java.util.Iterable`, so any Collection may be the target of a for-each statement. (The Iterable interface provides the iterator() method used by for-each statements.) All collections have an iterator that goes through all of the elements in the collection. Additionally, Collection is a generic. Any collection can be written to store any class. For example, Collection<String> can hold strings, and the elements from the collection can be used as strings without any casting required.

There are three main types of collections:

- Lists: always ordered, may contain duplicates and can be handled the same way as usual arrays

- Sets: cannot contain duplicates and provide random access to their elements
- Maps: connect unique keys with values, provide random access to its keys and may host duplicate values

List Interface

Lists are implemented in the JCF via the `java.util.List` interface. It defines a list as essentially a more flexible version of an array. Elements have a specific order, and duplicate elements are allowed. Elements can be placed in a specific position. They can also be searched for within the list. Two concrete classes implement List. The first is `java.util.ArrayList`, which implements the list as an array. Whenever functions specific to a list are required, the class moves the elements around within the array in order to do it. The other implementation is `java.util.LinkedList`. This class stores the elements in nodes, each of which has a pointer to the previous and next nodes in the list. The list can be traversed by following the pointers, and elements can be added or removed simply by changing the pointers around to place the node in its proper place.

Queue Interfaces

The `java.util.Queue` interface defines the queue data structure, which stores elements in the order in which they are inserted. New additions go to the end of the line, and elements are removed from the front. It creates a first-in first-out system. This interface is implemented by `java.util.LinkedList`, `java.util.ArrayDeque`, and `java.util.PriorityQueue`. LinkedList, of course, also implements the List interface and can also be used as one. But it also has the Queue methods. ArrayDeque implements the queue as an array. Both LinkedList and ArrayDeque also implement the `java.util.Deque` interface, giving it more flexibility.

`java.util.Queue` can be used more flexibly with its subinterface, `java.util.BlockingQueue`. The BlockingQueue interface works like a regular queue, but additions to and removals from the queue are blocking. If remove is called on an empty queue, it can be set to wait either a specified time or indefinitely for an item to appear in the queue. Similarly, adding an item is subject to an optional capacity restriction on the queue, and the method can wait for space to become available in the queue before returning.

The `java.util.Queue` interface is expanded by the `java.util.Deque` subinterface. Deque creates a double-ended queue. While a regular queue only allows insertions at the rear and removals at the front, the deque allows insertions or removals to take place both at the front and the back. A deque is like a queue that can be used forwards or backwards or both at once. Additionally, both a forwards and a backwards iterator can be generated. The Deque interface is implemented by `java.util.ArrayDeque` and `java.util.LinkedList`.

The `java.util.concurrent.BlockingDeque` interface works similarly to `java.util.concurrent.BlockingQueue`. The same methods for insertion and removal with time limits for waiting for the insertion or removal to become possible are provided. However, the interface also provides the flexibility of a deque. Insertions and removals can take place at both ends. The blocking function is combined with the deque function.

PriorityQueue Class

`java.util.PriorityQueue` implements `java.util.Queue`, but also alters it. Instead of elements being ordered by the order in which they are inserted, they are ordered by priority. The method used to determine priority is either the compareTo() method in the elements or a method given in the constructor. The class creates this by using a heap to keep the items sorted.

Set Interfaces

Java's `java.util.Set` interface defines the set. A set can't have any duplicate elements in it. Additionally, the set has no set order. As such, elements can't be found by index. Set is implemented by `java.util.HashSet`, `java.util.LinkedHashSet`, and `java.util.TreeSet`. HashSet uses a hash table. More specifically, it uses a `java.util.HashMap` to store the hashes and elements and to prevent duplicates. `java.util.LinkedHashSet` extends this by creating a doubly linked list that links all of the elements by their insertion order. This ensures that the iteration order over the set is predictable. `java.util.TreeSet` uses a red-black tree implemented by a `java.util.TreeMap`. The red-black tree makes sure that there are no duplicates. Additionally, it allows TreeSet to implement `java.util.SortedSet`.

The `java.util.Set` interface is extended by the `java.util.SortedSet` interface. Unlike a regular set, the elements in a sorted set are sorted, either by the element's compareTo() method, or a method provided to the constructor of the sorted set. The first and last elements of the sorted set can be retrieved, and subsets can be created via minimum and maximum values, as well as beginning or ending at the beginning or ending of the sorted set. The SortedSet interface is implemented by`java.util.TreeSet`.

`java.util.SortedSet` is extended further via the `java.util.NavigableSet` interface. It's similar to SortedSet, but there are a few additional methods. The floor(), ceiling(), lower(), and higher() methods find an element in the set that's close to the parameter. Additionally, a descending iterator over the items in the set is provided. As with SortedSet, `java.util.TreeSet` implements NavigableSet.

Map Interfaces

Maps are defined by the `java.util.Map` interface in Java. Maps are simple data structures that associate a key with a value. The element is the value. This lets the map be very flexible. If the key is the hash code of the element, the map is essentially a set. If it's just an increasing number, it becomes a list. Maps are implemented by `java.util.HashMap`, `java.util.LinkedHashMap`, and `java.util.TreeMap`. HashMap uses a hash table. The hashes of the keys are used to find the values in various buckets. LinkedHashMap extends this by creating a doubly linked list between the elements. This allows the elements to be accessed in the order in which they were inserted into the map. TreeMap, in contrast to HashMap and LinkedHashMap, uses a red-black tree. The keys are used as the values for the nodes in the tree, and the nodes point to the values in the map.

The `java.util.Map` interface is extended by its subinterface, `java.util.SortedMap`. This interface defines a map that's sorted by the keys provided. Using, once again, the compareTo() method or a method provided in the

constructor to the sorted map, the key-value pairs are sorted by the keys. The first and last keys in the map can be called. Additionally, submaps can be created from minimum and maximum keys. SortedMap is implemented by `java.util.TreeMap`.

The `java.util.NavigableMap` interface extends `java.util.SortedMap` in various ways. Methods can be called that find the key or map entry that's closest to the given key in either direction. The map can also be reversed, and an iterator in reverse order can be generated from it. It's implemented by `java.util.TreeMap`.

~~~ End of Article ~~~

Java OOP: String and StringBuffer

Source	http://cnx.org/content/m45237/latest/?collection=col11441/latest
Date of Retrieval	04/01/2013

Introduction

A string in Java is an object. Java provides two different string classes from which objects that encapsulate string data can be instantiated:

- **String**
- **StringBuffer**

The **String** class is used for strings that are not allowed to change once an object has been instantiated *(an immutable object).* The **StringBuffer** class is used for strings that may be modified by the program.

You can't modify a String object, but you can replace it

While the contents of a **String** object cannot be modified, a reference to a **String** object can be caused to point to a different **String** object as illustrated in the sample program shown in **Listing 1**. Sometimes this makes it appear that the original **String** object is being modified.

Listing 1: File String01.java.

```
/*File String01.java Copyright 1997, R.G.Baldwin
This application illustrates the fact that while a String
object cannot be modified, the reference variable can be
modified to point to a new String object which can have
the appearance of modifying the original String object.

The program was tested using JDK 1.1.3 under Win95.

The output from this program is

Display original string values
THIS STRING IS NAMED str1
This string is named str2
Replace str1 with another string
Display new string named str1
THIS STRING IS NAMED str1 This string is named str2
Terminating program

**********************************************************/

class String01{
  String str1 = "THIS STRING IS NAMED str1";
  String str2 = "This string is named str2";
```

Listing 1: File String01.java.

```
  public static void main(String[] args){
    String01 thisObj = new String01();
    System.out.println("Display original string values");
    System.out.println(thisObj.str1);
    System.out.println(thisObj.str2);
    System.out.println("Replace str1 with another string");
    thisObj.str1 = thisObj.str1 + " " + thisObj.str2;
    System.out.println("Display new string named str1");
    System.out.println(thisObj.str1);
    System.out.println("Terminating program");
  }//end main()
}//end class String01
```

1

It is important to note that the following statement does not modify the original object pointed to by the reference variable named **str1** .

NOTE:

```
thisObj.str1 = thisObj.str1 + " " + thisObj.str2;
```

Rather, this statement creates a new object, which is concatenation of two existing objects and causes the reference variable named **str1** to point to the new object instead of the original object.

The original object then becomes eligible for garbage collection *(unless there is another reference to the object hanging around somewhere).*

Many aspects of string manipulation can be accomplished in this manner, particularly when the methods of the **String** class are brought into play.

Why are there two string classes?

According to *The Java Tutorial* by Campione and Walrath:

NOTE:

"Because they are constants, Strings are typically cheaper than StringBuffers and they can be shared. So it's important to use Strings when they're appropriate."

Creating String and StringBuffer objects

The **String** and **StringBuffer** classes have numerous overloaded constructors and many different methods. I will attempt to provide a sampling of constructors and methods that will prepare you to explore other constructors and methods on your own.

The next sample program touches on some of the possibilities provided by the wealth of constructors and methods in the **String** and **StringBuffer** classes.

At this point, I will refer you to Java OOP: Java Documentation where you will find a link to online Java documentation. Among other things, the online documentation provides a list of the overloaded constructors and methods for the **String** and **StringBuffer** classes.

As of Java version 7, there are four overloaded constructors in the **StringBuffer** class and about thirteen different overloaded versions of the **append** method. There are many additional methods in the **StringBuffer** class including about twelve overloaded versions of the **insert** method.

As you can see, there are lots of constructors and lots of methods from which to choose. One of your challenges as a Java programmer will be to find the right methods of the right classes to accomplish what you want your program to accomplish.

The sample program named String02

The sample program shown in **Listing 2** illustrates a variety of ways to create and initialize **String** and **StringBuffer** objects.

Listing 2: File String02.java.

```
/*File String02.java Copyright 1997, R.G.Baldwin
Illustrates different ways to create String objects and
StringBuffer objects.

The program was tested using JDK 1.1.3 under Win95.

The output from this program is as follows.  In some cases,
manual line breaks were inserted to make the material fin
this presentation format.

Create a String the long way and display it
String named str2

Create a String the short way and display it
String named str1

Create, initialize, and display a StringBuffer using new
StringBuffer named str3

Try to create/initialize StringBuffer without
using new - not allowed

Create an empty StringBuffer of default length
Now put some data in it and display it
StringBuffer named str5

Create an empty StringBuffer and specify length
when it is created
Now put some data in it and display it
StringBuffer named str6

Try to create and append to StringBuffer without
using new -- not allowed

**********************************************************/
```

Listing 2: File String02.java.

```
class String02{
  void d(String displayString){//method to display strings
    System.out.println(displayString);
  }//end method d()

  public static void main(String[] args){
    String02 o = new String02();//obj of controlling class

    o.d("Create a String the long way and display it");
    String str1 = new String("String named str2");
    o.d(str1 + "\n");

    o.d("Create a String the short way and display it");
    String str2 = "String named str1";
    o.d(str2 + "\n");

    o.d("Create, initialize, and display a StringBuffer " +
                                              "using new");
    StringBuffer str3 = new StringBuffer(
                                "StringBuffer named str3");
    o.d(str3.toString()+"\n");

    o.d("Try to create/initialize StringBuffer without " +
                               "using new - not allowed\n");
    //StringBuffer str4 = "StringBuffer named str4";x

    o.d("Create an empty StringBuffer of default length");
    StringBuffer str5 = new StringBuffer();

    o.d("Now put some data in it and display it");
    //modify length as needed
    str5.append("StringBuffer named str5");
    o.d(str5.toString() + "\n");

    o.d("Create an empty StringBuffer and specify " +
                              "length when it is created");
    StringBuffer str6 = new StringBuffer(
                       "StringBuffer named str6".length());
    o.d("Now put some data in it and display it");
    str6.append("StringBuffer named str6");
    o.d(str6.toString() + "\n");

    o.d("Try to create and append to StringBuffer " +
                      "without using new -- not allowed");
    //StringBuffer str7;
    //str7.append("StringBuffer named str7");
  }//end main()
}//end class String02
```

2

Alternative String instantiation constructs

The first thing to notice is that a **String** object can be created using either of the following constructs:

NOTE:

Alternative String instantiation constructs

```
String str1 = new String("String named str2");

String str2 = "String named str1";
```

The first approach uses the **new** operator to instantiate an object while the shorter version doesn't use the new operator.

Later I will discuss the fact that

- the second approach is not simply a shorthand version of the first construct, but that
- they involve two different compilation scenarios with the second construct being more efficient than the first.

Instantiating StringBuffer objects

The next thing to notice is that a similar alternative strategy does not hold for the **StringBuffer** class.

For example, it is not possible to create a **StringBuffer** object without use of the **new** operator. *(It is possible to create a reference to a **StringBuffer** object but it is later necessary to use the **new** operator to actually instantiate an object.)*

Note the following code fragments that illustrate allowable and non-allowable instantiation scenarios for **StringBuffer** objects.

NOTE:

Instantiating StringBuffer objects

```
//allowed
StringBuffer str3 = new StringBuffer(
                      "StringBuffer named str3");

//not allowed
//StringBuffer str4 = "StringBuffer named str4";


o.d("Try to create and append to StringBuffer " +
            "without using new -- not allowed");
//StringBuffer str7;
//str7.append("StringBuffer named str7");
```

Declaration, memory allocation, and initialization

To review what you learned in an earlier module, three steps are normally involved in creating an object *(but the third step may be omitted).*

- declaration
- memory allocation

- initialization

The following code fragment performs all three steps:

NOTE:

Declaration, memory allocation, and initialization

```
StringBuffer str3 =
  new StringBuffer("StringBuffer named str3");
```

The code

StringBuffer str3 declares the type and name of a reference variable of the correct type for the benefit of the compiler.

The **new** operator allocates memory for the new object.

The constructor call

StringBuffer("StringBuffer named str3") constructs and initializes the object.

Instantiating an empty StringBuffer object

The instantiation of the **StringBuffer** object shown **above** uses a version of the constructor that accepts a **String** object and initializes the **StringBuffer** object when it is created.

The following code fragment instantiates an empty **StringBuffer** object of a default capacity and then uses a version of the **append** method to put some data into the object. *(Note that the data is actually a **String** object -- a sequence of characters surrounded by quotation marks.)*

NOTE:

Instantiating an empty StringBuffer object

```
//default initial length
StringBuffer str5 = new StringBuffer();

//modify length as needed
str5.append("StringBuffer named str5");
```

It is also possible to specify the capacity when you instantiate a **StringBuffer** object.

Some authors suggest that if you know the final length of such an object, it is more efficient to specify that length when the object is instantiated than to start with the default length and then require the system to increase the length "on the fly" as you manipulate the object.

This is illustrated in the following code fragment. This fragment also illustrates the use of the **length** method of the **String** class just to make things interesting. *(A simple integer value for the capacity of the **StringBuffer** object would have worked just as well.)*

NOTE:

Instantiating a StringBuffer object of a non-default length

```
StringBuffer str6 = new StringBuffer(
            "StringBuffer named str6".length());
str6.append("StringBuffer named str6");
```

Accessor methods

The following quotation is taken directly from *The Java Tutorial* by Campione and Walrath.

NOTE:

"An object's instance variables are encapsulated within the object, hidden inside, safe from inspection or manipulation by other objects. With certain well-defined exceptions, the object's methods are the only means by which other objects can inspect or alter an object's instance variables. Encapsulation of an object's data protects the object from corruption by other objects and conceals an object's implementation details from outsiders. This encapsulation of data behind an object's methods is one of the cornerstones of object-oriented programming."

The above statement lays out an important consideration in good object-oriented programming.

The methods used to obtain information about an object are often referred to as *accessor methods.*

Constructors and methods of the String class

I told you in an **earlier section** that the **StringBuffer** class provides a large number of overloaded constructors and methods. The same holds true for the **String** class.

Once again, I will refer you to Java OOP: Java Documentation where you will find a link to online Java documentation. Among other things, the documentation provides a list of the overloaded constructors and methods for the **String** class

String objects encapsulate data

The characters in a **String** object are not directly available to other objects. However, as you can see from the documentation, there are a large number of methods that can be used to access and manipulate those characters. For example, in an earlier sample program ***(Listing 2)***, I used the **length** method to access the number of characters stored in a **String** object as shown in the following code fragment.

NOTE:

```
StringBuffer str6 = new StringBuffer(
                    "StringBuffer named str6".length());
```

In this case, I applied the **length** method to a literal string, but it can be applied to any valid representation of an object of type **String** .

I then passed the value returned by the **length** method to the constructor for a **StringBuffer** object.

As you can determine by examining the argument lists for the various methods of the **String** class,

- some methods return data stored in the string while
- other methods return information about that data.

For example, the **length** method returns information about the data stored in the **String** object.

Methods such as **charAt** and **substring** return portions of the actual data.

Methods such **toUpperCase** can be thought of as returning the data, but returning it in a different format.

Creating String objects without calling the constructor

Methods in other classes and objects may create **String** objects without an explicit call to the constructor by the programmer. For example the **toString** method of the **Float** class receives a **float** value as an incoming parameter and returns a reference to a **String** object that represents the **float** argument.

Memory management by the StringBuffer class

If the additional characters cause the size of the **StringBuffer** to grow beyond its current capacity when characters are added, additional memory is automatically allocated.

However, memory allocation is a relatively expensive operation and you can make your code more efficient by initializing **StringBuffer** capacity to a reasonable first guess. This will minimize the number of times memory must be allocated for it.

When using the **insert** methods of the **StringBuffer** class, you specify the index *before which* you want the data inserted.

The toString method

Frequently you will need to convert an object to a **String** object because you need to pass it to a method that accepts only **String** values *(or perhaps for some other reason).*

All classes inherit the **toString** method from the **Object** class. Many of the classes *override* this method to provide an implementation that is meaningful for objects of that class.

In addition, you may sometimes need to *override* the **toString** method for classes that you define to provide a meaningful **toString** behavior for objects of that class.

I explain the concept of overriding the **toString** method in detail in the module titled Java OOP: Polymorphism and the Object Class.

Strings and the Java compiler

In Java, you specify literal strings between double quotes as in:

NOTE:

Literal strings

```
"I am a literal string of the String type."
```

You can use literal strings anywhere you would use a **String** object.

You can also apply **String** methods directly to a literal string as in an **earlier program** that calls the **length** method on a literal string as shown below.

NOTE:

Using String methods with literal strings

```
StringBuffer str6 = new StringBuffer(
              StringBuffer named str6".length());
```

Because the compiler automatically creates a new **String** object for every literal string, you can use a literal string to initialize a**String** object *(without use of the new operator)* as in the following code fragment from a **previous program**:

NOTE:

```
String str1 = "THIS STRING IS NAMED str1";
```

The above construct is equivalent to, but more efficient than the following, which, according to *The Java Tutorial* by Campione and Walrath, ends up creating two **String** objects instead of one:

NOTE:

```
String str1 = new String("THIS STRING IS NAMED str1");
```

In this case, the compiler creates the first **String** object when it encounters the literal string, and the second one when it encounters **new String()** .

Concatenation and the + operator

The plus (+) operator is overloaded so that in addition to performing the normal arithmetic operations, it can also be used to concatenate strings.

This will come as no surprise to you because we have been using code such as the following since the beginning of this group of *Programming Fundamentals* modules:

NOTE:

```
String cat = "cat";

System.out.println("con" + cat + "enation");
```

According to Campione and Walrath, Java uses **StringBuffer** objects behind the scenes to implement concatenation. They indicate that the above code fragment compiles to:

NOTE:

```
String cat = "cat";
System.out.println(new StringBuffer().append("con").
                       append(cat).append("enation"));
```

Fortunately, that takes place behind the scenes and we don't have to deal directly with the syntax.

~~~ End of Article ~~

Java OOP: Command-Line Arguments

Source	http://cnx.org/content/m45246/latest/?collection=col11441/latest
Date of Retrieval	04/01/2013

Preface

Although the use of command-line arguments is rare is this time of Graphical User Interfaces *(GUI)* , they are still useful for testing and debugging code. This module explains the use of command-line arguments in Java.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following link to easily find and view the listing while you are reading about it.

Listings

- **Listing 1**. Illustration of command-line arguments.

Discussion

Familiar example from DOS

Java programs can be written to accept command-line-arguments.

DOS users will be familiar with commands such as the following:

NOTE:

Familiar DOS command

```
copy fileA fileB
```

In this case, **copy** is the name of the program to be executed, while **fileA** and **fileB** are command-line arguments.

Java syntax for command-line arguments

The Java syntax for supporting command-line arguments is shown below *(note the formal argument list for the **main** method).*

NOTE:

Java syntax for command-line arguments

```
public static void main(String[] args){
. . .
}//end main method
```

In Java, the formal argument list for the **main** method must appear in the method signature whether or not the program is written to support the use of command-line arguments. If the argument isn't used, it is simply ignored.

Where the arguments are stored

The parameter **args** contains a reference to a one-dimensional array object of type **String** .

Each of the elements in the array *(including the element at index zero)* contains a reference to an object of type **String.** Each object of type String encapsulates one command-line argument.

The number of arguments entered by the user

Recall from an earlier module on arrays that the number of elements in a Java array can be obtained from the **length** property of the array. Therefore, the number of arguments entered by the user is equal to the value of the **length** property. If the user didn't enter any arguments, the value will be zero.

Command-line arguments are separated by the space character. If you need to enter an argument that contains a space, surround the entire argument with quotation mark characters as in *"My command line argument"* .

The first command-line argument is encapsulated in the **String** object referred to by the contents of the array element at index 0, the second argument is referred to by the element at index 1, etc.

Sample Java program

The sample program in **Listing 1** illustrates the use of command-line arguments.

Listing 1: Illustration of command-line arguments.

```
/*File cmdlin01.java Copyright 1997, R.G.Baldwin
This Java application illustrates the use of Java
command-line arguments.

When this program is run from the command line as follows:

java cmdlin01 My command line arguments

the program produces the following output:

My
command
line
arguments
**************************************************/
class cmdlin01 { //define the controlling class
  public static void main(String[] args){ //main method
    for(int i=0; i < args.length; i++)
    System.out.println( args[i] );
  }//end main
}//End cmdlin01 class.
```

The output from running this program for a specific input is shown in the comments at the beginning of the program.

~~~ End of Article ~~~

Session 9: Modifiers and Packages

Different Types of Variables in Java

Source	http://anotherjavaduke.wordpress.com/2012/02/25/different-types-of-variables-in-java/
Date of Retrieval	04/01/2013

Following are the kinds of variables categorized according to their scope

- Static variables
- Instance variables
- Method local and Method parameters
- Block variables

Following code shows the variables on go [Code Snippet 1]

```
01 public class Variables {
02  static int staticVariable = 1;
03  int instanceVariable = 2;
04
05  public void methodName(int methodParameter) {
06       int methodLocalVariable = 3;
07
08  if (true) {
09       int blockVariable = 4;
10  }
11
12  }
13
14 }
```

Static variables

These variables are declared at the top level. They begin their life when first class loaded into memory and ends when class is unloaded. As they remain in memory till class exists, so these variables often called Class variables. There only one copy of these variables exist

These variables are having highest scope that is they can be accessed from any method/ block in a class

When no explicit assignment made while declaration they are initialized to default values, depending on their type.

E.g. As from Code Snippet 1, *staticVariable* declared as the variable of this type.

Instance variables

These are also declared as the top level variable. They begin their life when object/.instance of class created and ends when object is destroyed. Each copy of this variable belongs an object.

Same as static ones, they initialized to default values depending on their type

They are having comparatively less scope as they are accessed only within instance method blocks.

E.g. As from Code Snippet 1, *instanceVariable* is of this type.

Method local variables / Method parameters

Method local variables are declared anywhere inside method. Their life is started point they declared / initialized and ends when method completes.

These variables are not assigned any default value as case in case of static and instance variables. Therefore, they must be initialized when declared or later but before accessing/ using in an expression otherwise compiler will complain.

```
01 // other code of class
02 public void methodName(int methodParameter) {
03  int methodLocalVariable ;
04  System.out.println(methodLocalVariable);
05
06  }
07 // other code of class
08
09 // If you try to compile above code snippet, compiler will throw
   following
10 // VariablesDemo.java:9: variable methodLocalVariable might not have
   been initialized
11 // System.out.println(methodLocalVariable);
12 //                        ^
```

Method parameters are local variables to method only except their declaration in parameter list of method and they get value upon invocation.

They are only accessible only in method that they are declared.

E.g. As from Code Snippet 1, methodLocalVariable and methodParameter is of this type

Block variables

These are variables that are declared inside any block. They can be accessed only within that block only.

Just like method local variables, they are not assigned any default value. Therefore, it is mandatory to assign value prior to using same in expression/ accessing.

E.g. As from Code Snippet 1, *blockVariable* is this kind.

~~~ End of Article ~~~

Packages

Source	http://en.wikibooks.org/wiki/Java_Programming/Packages
Date of Retrieval	04/01/2013

Java Package / Name Space

Usually a Java application is built by many developers and it is common that third party modules/classes are integrated. The end product can easily contain hundreds of classes. Class name collision is likely to happen. To avoid this, a Java class can be put in a "name space". This "name space" in Java is called the **`package`**.

The Java package needs to be unique across Vendors to avoid name collisions. For that reason Vendors usually use their domain name in reverse order. That is guaranteed to be unique. For example a company called 'Your Company Inc.', would use a package name something like this:`com.yourcompany.yourapplicationname.yourmodule.YourClass`.

To put a class in a package, the `package` keyword is used at the top of each class file. For Example,

```
package com.yourcompany.yourapplication.yourmodule;
```

When we want to reference a Java class that is defined outside of the current package name space, we have to specify which package name space that class is in. So we could reference that class with something like*com.yourcompany.yourapplication.yourmodule.YourClass*. To avoid having to type in the package name each time when we want to reference an outside class, we can declare which package the class belongs to by using the **`import`** Java keyword at the top of the file. For Example,

```
import com.yourcompany.yourapplication.yourmodule.YourClass;
```

Then we can refer to that class by just using the class name `YourClass`.

In rare cases it can happen that you need to reference two classes having the same name in different packages. In those cases, you cannot use the import keyword for both classes. One of them needs to be referenced by typing in the whole package name. For Example,

```
package com.mycompany.myapplication.mymodule;
...
import com.yourcompany.yourapplication.youmodule.SameClassName;
...
SameClassName yourObjectRef = new SameClassName();
com.hercompany.herapplication.hermodule.SameClassName herObjectRef =
   new com.hercompany.herapplication.hermodule.SameClassName();
```

The Java package has one more interesting characteristic; the package name corresponds where the actual file is stored on the file system. And that is actually how the compiler and

the class loader find the Java files on the file system. For example, the class*com.yourcompany.yourapplication.yourmodule.YourClass* is stored on the file system in the corresponding directory: *com/yourcompany/yourapplication/yourmodule/YourClass*. Because of this, package names should be lowercase, since in some operating systems the directory names are not case sensitive.

Wildcard imports

It is possible to import an entire package, using an asterisk:

```
import javax.swing.*;
```

While it may seem convenient, it may cause problems if you make a typographical error. For example, if you use the above import to use JFrame, but then type `JFraim frame=new JFraim();`, the Java compiler will report an error similar to "Cannot find symbol: JFraim". Even though it seems as if it was imported, the compiler is giving the error report at the first mention of JFraim, which is half-way through your code, instead of the point where you imported JFrame along with everything else in javax.swing.

If you change this to `import javax.swing.JFraim;` the error will be at the import instead of within your code.

Furthermore, if you `import javax.swing.*;` and `import java.util.*;`, and javax.swing.Queue is later added in a future version of Java, your code that uses Queue (java.util) will fail to compile. This particular example is fairly unlikely, but if you are working with non-Sun libraries, it may be more likely to happen.

Importing packages from .jar files

If you are importing library packages or classes that reside in a `.jar` file, you must ensure that the file is in the current classpath (both at compile- and execution-time). Apart from this requirement, importing these packages and classes is the same as if they were in their full, expanded, directory structure.

Example:

To compile and run a class from a project's top directory (that contains the two directories `/source` and `/libraries`) you could use the following command:

```
javac -classpath libraries/lib.jar source/MainClass.java
```

And then to run it, similarly:

```
java -classpath libraries/lib.jar source/MainClass
```

(The above is simplified, and demands that MainClass be in the default package, or a package called 'source', which isn't very desirable.)

Class Loading / Name Space

A fully qualified class name consists of the package name plus the class name. For example, the fully qualified class name of `HashMap` is `java.util.HashMap`. Sometime it can happen that two classes have the same name, but they can not belong to the same package, otherwise it would be the same class. It can be said that the two classes with the same name are in different name spaces. In the above example, the `HashMap` class is in the `java.util` name space.

Let be two `Customer` classes with different name spaces (in different packages):

- `com.bluecompany.Customer`
- `com.redcompany.Customer`

When we need to use both classes in the same program file, we can use the `import` keyword only for one of the classes. For the other we need to use the fully qualified name.

The runtime identity of a class in Java 2 is defined by the fully qualified class name and its defining class loader. This means that the same class, loaded by two different class loaders, is seen by the Virtual Machine as two completely different types.

~~~ End of Article ~~~

Java OOP: Packages

Source	http://cnx.org/content/m45229/latest/?collection=col11441/latest
Date of Retrieval	04/01/2013

Introduction

Before you can understand much about packages, you will need to understand the *classpath environment variable,* so that is where I will begin the discussion.

After learning about the classpath environment variable, you will learn how to create your own packages.

Classpath environment variable

The purpose of the *classpath environment variable* is to tell the JVM and other Java applications *(such as the javac compiler)* where to find class files and class libraries. This includes those class files that are part of the JDK and class files that you may create yourself.

I am assuming that you already have some familiarity with the use of environment variables in your operating system. All of the discussion in this module will be based on the use of a generic form of Windows. *(By generic, I mean not tied to any particular version of Windows.)* Hopefully you will be able to translate the information to your operating system if you are using a different operating system.

NOTE:

In a nutshell

Environment variables provide information that the operating system uses to do its job.

There are usually a fairly large number of environment variables installed on a machine at any given time. If you would like to see what kind of environment variables are currently installed on your machine, bring up a command-line prompt and enter the command **set**. This should cause the names of several environment variables, along with their settings to be displayed on your screen.

While you are at it, see if any of those items begin with **CLASSPATH=.** If so, you already have a classpath environment variable set on your machine, but it may not contain everything that you need.

I am currently using a Windows 7 operating system and no classpath environment variable is set on it. I tend to use the **–cp** switch option *(see **Listing 4**)* in the JDK to set the classpath on a temporary basis when I need it to be set.

Rather than trying to explain all of the ramifications regarding the classpath, I will simply refer you to an Oracle document on the topic titled 'Setting the class path'.

I will also refer you to Java OOP: The Guzdial-Ericson Multimedia Class Library where I discuss the use of the classpath environment variable with a Java multimedia class library.

Some rules

There are some rules that you must follow when working with the classpath variable, and if you fail to do so, things simply won't work.

For example, if your class files are in a jar file, the classpath must end with the name of that jar file.

On the other hand, if the class files are not in a jar file, the classpath must end with the name of the folder that contains the class files.

Your classpath must contain a fully-qualified path name for every folder that contains class files of interest, or for every jar file of interest. The paths should begin with the letter specifying the drive and end either with the name of the jar file or the name of the folder that contains the class files. .

If you followed the default JDK installation procedure and are simply compiling and executing Java programs in the current directory you probably won't need to set the classpath. By default, the system already knows *(or can figure out)* how to allow you to compile and execute programs in the current directory and how to use the JDK classes that come as part of the JDK.

However, if you are using class libraries other than the standard Java library, are saving your class files in one or more different folders, or are ready to start creating your own packages, you will need to set the classpath so that the system can find the class files in your packages.

Developing your own packages

One of the problems with storing all of your class files in one or two folders is that you will likely experience name conflicts between class files.

Every Java program can consist of a large number of separate classes. A class file is created for each class that is defined in your program, even if they are all combined into a single source file.

It doesn't take very many programs to create a lot of class files, and it isn't long before you find yourself using the same class names over again. If you do, you will end up overwriting class files that were previously stored in the folder.

For me, it only takes two GUI programs to get in trouble because I tend to use the same class names in every program for certain standard operations such as closing a **Frame** or processing an **ActionEvent.** For the case of the **ActionEvent,** the body of the class varies from one application to the next so it doesn't make sense to turn it into a library class.

So we need a better way to organize our class files, and the Java package provides that better way.

The Java package approach allows us to store our class files in a hierarchy of folders *(or a jar file that represents that hierarchy)*while only requiring that the classpath variable point to the top of the hierarchy. The remaining hierarchy structure is encoded into our programs using package directives and import directives.

Now here is a little jewel of information that cost me about seven hours of effort to discover when I needed it badly.

When I first started developing my own packages, I spent about seven hours trying to determine why the compiler wouldn't recognize the top-level folder in my hierarchy of package folders.

I consulted numerous books by respected authors and none of them was any help at all. I finally found the following statement in the Java documentation *(when all else fails, read the documentation).* By the way, a good portion of that wasted seven hours was spent trying to find this information in the documentation which is voluminous.

NOTE:

The following text was extracted directly from the JDK 1.1 documentation

If you want the CLASSPATH to point to class files that belong to a package, you should specify a path name that includes the path to the directory one level above the directory having the name of your package.

For example, suppose you want the Java interpreter to be able to find classes in the package mypackage. If the path to the mypackage directory is C:\java\MyClasses\mypackage, you would set the CLASSPATH variable as follows:

set CLASSPATH=C:\java\MyClasses

If you didn't catch the significance of this, read it again. When you are creating a classpath variable to point to a folder containing classes, it must point to the folder. However, when you are creating a classpath variable to point to your package, it must point to the folder that is one level above the directory that is the top-level folder in your package.

Once I learned that I had to cause the classpath to point to the folder immediately above the first folder in the hierarchy that I was including in my package directives, everything started working.

The package directive

So, what is the purpose of a package directive, and what does it look like?

NOTE:

Purpose of a package directive

The purpose of the package directive is to identify a particular class *(or group of classes contained in a single source file (compilation unit))* as belonging to a specific package.

This is very important information for a variety of reasons, not the least of which is the fact that the entire access control system is wrapped around the concept of a class belonging to a specific package. For example, code in one package can only access public classes in a different package.

Stated in the simplest possible terms, a package is a group of class files contained in a single folder on your hard drive.

At compile time, a class can be identified as being part of a particular package by providing a package directive at the beginning of the source code.

A package directive, if it exists, must occur at the beginning of the source code *(ignoring comments and white space).* No text other than comments and whitespace is allowed ahead of the package directive.

If your source code file does not contain a package directive, the classes in the source code file become part of the *default package.* With the exception of the default package, all packages have names, and those names are the same as the names of the folders that contain the packages. There is a one-to-one correspondence between folder names and package names. The default package doesn't have a name.

Some examples of package directives that you will see in upcoming sample programs follow:

NOTE:

Example package directives

```
package Combined.Java.p1;
package Combined.Java.p2;
```

Given the following as the classpath on my hypothetical machine,

CLASSPATH=.;c:\Baldwin\JavaProg

These two package directives indicate that the class files being governed by the package directives are contained in the following folders:

NOTE:

```
c:\Baldwin\JavaProg\Combined\Java\p1
c:\Baldwin\JavaProg\Combined\Java\p2
```

Notice how I concatenated the package directive to the classpath setting and substituted the backslash character for the period when converting from the package directive to the fully-qualified path name.

Code in one package can refer to a class in another package *(if it is otherwise accessible)* by qualifying the class name with its **package name as follows** :

NOTE:

```
Combined.Java.p2.Access02 obj02 =
       new Combined.Java.p2.Access02();
```

Obviously, if we had to do very much of that, it would become very burdensome due to the large amount of typing required. As you already know, the *import directive* is available to allow us to specify the package containing the class just once at the beginning of the source file and then refer only to the class name for the remainder of that source file.

The import directive

This discussion will be very brief because you have been using import directives since the very first module. Therefore, you already know what they look like and how to use them.

If you are interested in the nitty-gritty details *(such as what happens when you provide two import directives that point to two different packages containing the same class file name)* , you can consult the Java Language Reference by Mark Grand, or you can download the Java language specification from Oracle's Java website.

The purpose of the import directive is to help us avoid the **burdensome typing requirement** described in the previous section when referring to classes in a different package.

An import directive makes the definitions of classes from other packages available on the basis of their file names alone.

You can have any number of import directives in a source file. However, they must occur after the package directive *(if there is one)* and before any class or interface declarations.

There are essentially two different forms of the import directive, one with and the other without a wild card character *(*)*. These two forms are illustrated in the following box.

NOTE:

Two forms of import directives

```
import java.awt.*
import java.awt.event.ActionEvent
```

The first import directive makes all of the class files in the **java.awt** package available for use in the code in a different package by referring only to their file names.

The second import directive makes only the class file named **ActionEvent** in the **java.awt.event** package available by referring only to the file name.

Compiling programs with package directives

So, how do you compile programs containing package directives? There are probably several ways. I am going to describe one way that I have found to be successful.

First, you must create your folder hierarchy to match the package directive that you intend to use. Remember to construct this hierarchy downward relative to the folder specified at the end of your classpath setting. If you have forgotten the **critical rule** in this respect, go back and review it.

Next, place source code files in each of the folders where you intend for the class files associated with those source code files to reside. *(After you have compiled and tested the program, you can remove the source code files if you wish.)*

You can compile the source code files individually if you want to, but that isn't necessary.

One of the source code files will contain the *controlling class.* The controlling class is the class that contains the **main** method that will be executed when you run the program from the command line using the JVM.

Make the directory containing that source code file be the current directory. *(If you don't know what the current directory is, go out and get yourself a **DOS For Dummies** book and read it.)*

Each of the source code files must contain a package directive that specifies the package that will contain the compiled versions of all the class definitions in that source code file. Using the instructions that I am giving you, that package directive will also describe the folder that contains the source code file.

Any of the source code files containing code that refers to classes in a different package must also contain the appropriate import directives, or you must use fully-qualified package names to refer to those classes.

Then use the **javac** program with your favorite options to compile the source code file containing the controlling class. This will cause all of the other source code files containing classes that are linked to the code in the controlling class, either directly or indirectly, to be compiled also. At least an attempt will be made to compile them all. You may experience a few compiler errors if your first attempt at compilation is anything like mine.

Once you eliminate all of the compiler errors, you can test the application by using the **java** program with your favorite options to execute the controlling class.

Once you are satisfied that everything works properly, you can copy the source code files over to an archive folder and remove them from the package folders if you want to do so.

Finally, you can also convert the entire hierarchy of package folders to a jar file if you want to, and distribute it to your client. If you don't remember how to install it relative to the classpath variable, go back and review that part of the module.

Once you have reached this point, how do you execute the program. I will show you how to execute the program from the command line in the sample program in the next section. *(Actually I will encapsulate command-line commands in a batch file and execute the batch file. That is a good way to guard against typing errors.)*

Sample program

The concept of packages can get very tedious in a hurry. Let's take a look at a sample program that is designed to pull all of this together.

This application consists of three separate source files located in three different packages. Together they illustrates the use of package and import directives, along with **javac** to build a standalone Java application consisting of classes in three separate packages.

(If you want to confirm that they are really in different packages, just make one of the classes referred to by the controlling class non-public and try to compile the program.)

In other words, in this sample program, we create our own package structure and populate it with a set of cooperating class files.

A folder named **jnk** is a child of the root folder on the M-drive.

A folder named **SampleCode** is a child of the folder named **jnk.**

A folder named **Combined** is a child of the folder named **SampleCode.**

A folder named **Java** is a child of the folder named **Combined.**

Folders named **p1** and **p2** are children of the folder named **Java.**

The file named **Package00.java,** shown in **Listing 1** is stored in the folder named **Java.**

Listing 1: File: Package00.java.

```
/*File Package00.java Copyright 1997, R.G.Baldwin
Illustrates use  of package and import directives to
build an application consisting of classes in three
separate packages.

The output from running the program follows:

Starting Package00
Instantiate obj of public classes in different packages
Constructing Package01 object in folder p1
Constructing Package02 object in folder p2
```

```
Back in main of Package00
**********************************************************/
package Combined.Java; //package directive

//Two import directives
import Combined.Java.p1.Package01;//specific form
import Combined.Java.p2.*; //wildcard form

class Package00{
  public static void main(String[] args){ //main method
    System.out.println("Starting Package00");

    System.out.println("Instantiate obj of public " +
                       "classes in different packages");
    new Package01();//Instantiate objects of two classes
    new Package02();// in different packages.

    System.out.println("Back in main of Package00");

  }//end main method
}//End Package00 class definition.
```

1

The file named **Package01.java,** shown in **Listing 2** is stored in the folder named **p1**.

Listing 2: File Package01.java.

```
/*File Package01.java Copyright 1997, R.G.Baldwin
See discussion in file Package00.java
**********************************************************/
package Combined.Java.p1;
public class Package01 {
  public Package01(){//constructor
    System.out.println(
             "Constructing Package01 object in folder p1");
  }//end constructor
}//End Package01 class definition.
```

2

The file named **Package02.java,** shown in **Listing 3** is stored in the folder named **p2.**

Listing 3: File Package02.java.

```
/*File Package02.java Copyright 1997, R.G.Baldwin
See discussion in file Package00.java
**********************************************************/
package Combined.Java.p2;
public class Package02 {
  public Package02(){//constructor
    System.out.println(
              "Constructing Package02 object in folder p2");
  }//end constructor
}//End Package02 class definition.
```

3

The file named **CompileAndRun.bat**, shown in **Listing 4** is stored in the folder named **SampleCode**.

Listing 4: File: CompileAndRun.bat.

```
echo off
rem This file is located in folder named M:\SampleCode,
rem which is Parent of folder Combined.

del Combined\Java\*.class
del Combined\Java\p1\*.class
del Combined\Java\p2\*.class

javac -cp M:\jnk\SampleCode Combined\Java\Package00.java

java -cp M:\jnk\SampleCode Combined.Java.Package00

pause
```

4

The controlling class named **Package00** is stored in the package named **Combined.Java,** as declared in **Listing 1**.

The class named **Package01** is stored in the package named **Combined.Java.p1,** as declared in **Listing 2**.

The class named **Package02** is stored in the package named **Combined.Java.p2,** as declared in **Listing 3**.

The controlling class named **Package00** imports **Combined.Java.p1.Package01** and **Combined.Java.p2.***, as declared in **Listing 1**.

Code in the **main** method of the controlling class in **Listing 1** instantiates objects of the other two classes in different packages. The constructors for those two classes announce that they are being constructed.

The two classes being instantiated are **public.** Otherwise, it would not be possible to instantiate them from outside their respective packages.

This program was tested using JDK 7 under Windows by executing the batch file named **CompileAndRun.bat**.

The classpath is set to the parent folder of the folder named **Combined** *(M:\jnk\SampleCode)* by the **-cp** switch in the file named **CompileAndRun.bat**.

The output from running the program is shown in the comments at the beginning of **Listing 1**.

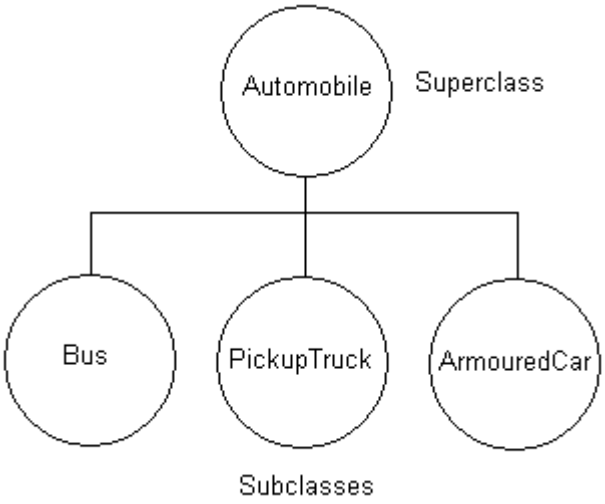
~~~ End of Article ~~~

Session 10: Inheritance and Polymorphism

Inheritance

Source	http://freecourseware.uwc.ac.za/freecourseware/information-systems/java-platform-introduction/inheritance
Date of Retrieval	04/01/2013

Remember the discussion we had on the `Automobile` superclass and some of its possible subclasses?



Let's take a look at building and subclassing a Java class.

Class *BankAccount*

Let's build a Java class `BankAccount`, which we can then subclass by way of illustration of the principles of *inheritance*. Inheritance is central to OOP, and no text on OOP would be complete without a treatment of this important concept. Inheritance is embodied by the following:

A subclass inherits the public variables and methods of the superclass, and has access to the public constructors of the superclass.

It is important to understand that inheritance is founded on the *public* variables and methods of the superclass; a subclass has no direct access to the private variables and methods of the superclass. The superclass has no access to the variables and methods of its subclass.

Consider a simple `BankAccount` class, which embodies:

1. A private variable `balance`, the amount of money in the `BankAccount`
2. Constructors for:
   - A `BankAccount` with a zero initial balance
   - A `BankAccount` with an initial balance other than zero
3. Methods for:
   - depositing money in the `BankAccount`
   - withdrawing money from the `BankAccount`
   - returning the current balance in the `BankAccount`

```
/** Creates a bank account with a balance that can be adjusted by
 * deposits and withdrawals.
 * @author G. Hearn July 2003
 */
public class BankAccount
{

   /** Constructs a bank account with a zero initial balance.
    */
   public BankAccount()
   {
      balance = 0;
   }

   /** Constructs a bank account with a given balance.
    * @param initialBalance the amount deposited at opening
    */
   public BankAccount(double initialBalance)
   {
      balance = initialBalance;
   }

   /** Deposits money into the bank account.
    * @param amount the amount being deposited
    */
   public void deposit(double amount)
   {
      balance += amount;
   }

   /** Withdraws money from the bank account.
    * @param amount the amount to be withdrawn
    */
   public void withdraw(double amount)
   {
      balance -= amount;
   }
```

```
    /** Returns the current balance in the bank account.
     * @return the current balance
     */
    public double getBalance()
    {
       return balance;
    }

    /** Holds the amount of money currently in the bank account.
     */
    private double balance;

}
```

This is a fairly simple class, which is self-explanatory. Note the two constructors.

Subclassing class *BankAccount*

When subclassing another class, we rely on inheriting the public methods and variables of the class, and add only those variables and methods to the subclass that provide the extra or specialized functionality of the subclass. `BankAccount` contains the functionality that is common to most kinds of bank account, so to produce a sensible and useful subclass, we need a situation where we recognize a type of bank account that requires some specialization that `BankAccount` doesn't provide us with. A savings account is usually an interest-bearing account, but `BankAccount` doesn't have (and doesn't need) a means to calculate interest on the balance in the account.

To deal with this, we write a subclass of the `BankAccount` class, called `SavingsAccount` which contains the extra functionality. Being a subclass of `BankAccount`, `SavingsAccount` will have methods:

i. *void deposit(double amount)*

ii. *void withdraw(double amount)*

iii. *double getBalance()*

These are the public methods of `BankAccount`, and they are *inherited* by `SavingsAccount`. Additionally, `SavingsAccount` must include a variable that holds the interest rate applicable to the `SavingsAccount`, must have a method which calculates the interest and adds it to the balance in the account, and another which retrieves the interest rate. To write a subclass, we must include the keyword `extends` and the name of the class we are extending in the class definition:

```
public class SavingsAccount extends BankAccount {
        ...
}
```

Here is the source code for class `SavingsAccount`:

```
/**This class was designed to extend the BankAccount class, to experiment
 * with inheriting methods. The class has its own extra method <i>getInterest()</i>
 * which calculates the balance + simple interest accrued on the balance in
 * the account.
 * @author G. Hearn July 2003
 */

public class SavingsAccount extends BankAccount
{
    /**
     * The interest rate expressed as a decimal fraction to apply to the balance.
     */
    private double interestRate;

    /**
     * Constructs a savings account with a given balance, and a given interest rate
     * expressed as a decimal fraction, e.g. 5.6% = 0.056
     * @param rate the interest rate
     * @param initialBalance the initial balance in the account
     */
    public SavingsAccount(double rate, double initialBalance)
    {
        interestRate = rate;
        this.deposit(initialBalance);
    }

    /**
     * Calculates the balance + simple interest for the account.
     * @return the balance + simple interest
     */
    public double getInterest()
    {
        double balWithInterest = this.getBalance() * (1 + interestRate);
        return balWithInterest;
    }



    /**
     * Returns the interest rate of the SavingsAccount.
     * @return the interest rate
     */
    public double getInterestRate(){
        return interestRate;
    }
}
```

Note that although `balance` is a private variable of class `BankAccount`, class `SavingsAccount` has no private variable of its own called `balance`. Where will the `SavingsAccount` store its current balance? In the private `BankAccount` variable `balance` - this is possible because `BankAccount` has a public method called `deposit()` which adds money to the private `balance` variable of

the `BankAccount` object. Even though a subclass has no *direct* access to the private variables of its superclass, if the superclass has a public method that accesses the private variable, we can use this public method to access the superclass private variable. This is as it should be - private variables can only be accessed by the methods of their class, and if we use the inherited superclass method `deposit()`, we are adding money to the superclass `balance`. We can see the balance in the `SavingsAccount` by using the `BankAccount` public method `getBalance()`.

To use the inherited methods of the superclass, we employ the keyword `this`, which refers to the current instance of the class, i.e. the current instance of `SavingsAccount`. For example, in the constructor for `SavingsAccount`:

```
public SavingsAccount(double rate, double initialBalance) {
    interestRate = rate;
    this.deposit(initialBalance);
}
```

~~~ End of Article ~~~

Java OOP: Abstract Methods, Abstract Classes, and Overridden Methods

Source	http://cnx.org/content/m44205/1.2/
Date of Retrieval	04/01/2013

Preview

The program that I will explain in this module produces no graphics and does not require the use of Ericson's media library.

OOP concepts

The program illustrates the following OOP concepts:

- Extending an abstract class.
- Parameterized constructor.
- Defining an abstract method in the superclass and overriding it in a subclass.
- Overridden **toString** method.

Program specifications

Write a program named **Prob04** that uses the class definition shown in **Listing 1** to produce the output on the command-line screen shown in **Figure 1**.

Program output on the command line screen.

```
Prob04
Dick
Baldwin
95
95
```

Figure 1: Program output on the command line screen.

Pseudo random data

Because the program generates and uses a pseudo random data value each time it is run, the actual values displayed in the last two lines of **Figure 1** will differ from one run to the next. However, in all cases, the two values must match.

New classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob04** given below.

Discussion and sample code

Will explain in fragments

I will explain this program in fragments. A complete listing is provided in **Listing 5** near the end of the module.

I will begin with the driver class named **Prob04**, which is shown in its entirety in **Listing 1**.

LISTING 1: **Source code for class Prob04.**

```
import java.util.*;

abstract class Prob04{
  public static void main(String[] args){
    Random generator = new Random(new Date().getTime());
    int randomNumber = (byte)generator.nextInt();

    Prob04 objRef = new Prob04MyClass(randomNumber);

    System.out.println(objRef);
    System.out.println(objRef.getData());
    System.out.println(randomNumber);
  }//end main

  //Declare the signature of an abstract class.
  public abstract int getData();

}//end class Prob04
```

The import directive

The import directive at the beginning of **Listing 1** is required because the program requires access to the **Random** class and the **Date** class, both of which are defined in the **java.util** package.

Lazy programming practice

It would be better programming practice to provide two explicit import directives, one for the **Random** class and the other for the **Date** class. However, if you are lazy like I apparently was when I wrote this program, you can use the wildcard character *(*)* to *import* all of the classes in a package.

An *abstract* method

I'm going to begin by skipping down to the second line from the bottom in Listing 1 and explain the declaration of the *abstract* method named **getData.**

Purpose of an abstract method

The purpose of an abstract method declaration is to establish the signature of a method that must be overridden in every *(non-abstract)* subclass of the class in which the abstract method is declared.

An incomplete method

As you can see the abstract method has no body. Therefore, it is incomplete, has no behavior, and cannot be executed.

An abstract method must be overridden in a subclass in order to be useful.

Override in different ways

The same abstract method can be overridden in different ways in different subclasses. In other words, the behavior of the overridden version can be tailored to *(appropriate for)* the class in which it is overridden.

A guarantee

The existence of an abstract method in a superclass guarantees that every *(non-abstract)* subclass of that superclass will have a **concrete** *(executable)* version of a method having that same signature.

An *abstract* class

The class named **Prob04** is declared *abstract* in **Listing 1**.

Any class can be declared abstract. The consequence of declaring a class abstract is that it is not possible to instantiate an object of the class.

Must be declared abstract...

More importantly, a class must be declared abstract if it contains one or more abstract method declarations. The idea here is that it must not be possible to instantiate objects containing incomplete *(non-executable)* methods.

The main method

As you have seen in previous modules, the driver class for every Java application must contain a method named **main** with a signature matching that shown in **Listing 1**.

A pseudo-random number generator

I will leave it as an exercise for the student to go to the javadocs and read up on the class named **Random**, along with the class named **Date** and the method named **getTime.**

Why pseudo-random?

I refer to this as a pseudo-random number generator because the sequence will probably repeat after an extremely large number of values has been generated.

An object of the class Random

Briefly, however, the first statement in the **main** method in **Listing 1** instantiates an object that will return a pseudo-random number each time certain methods are called on the object.

Seeding the generator

The value passed as a parameter to the **Random** constructor represents the current time and guarantees that the series of pseudo-random values returned by the methods will be different each time the program is run. This is commonly known as *seeding* the generator.

Get and save a pseudo random value

The next statement in **Listing 1** calls the **nextInt** method on the generator object to get and save the next value of type **int** in the pseudo-random sequence.

Cast to type byte

This value is cast to type **byte**, which discards all but the eight least significant bits of the **int** value. When it is stored in the variable named **randomNumber** of type **int**, the sign is extended through the most significant 24 bits and it becomes a value of type int that is guaranteed to be of relatively small magnitude.

Why cast to byte?

I cast the random value to type byte simply to cause the values that are displayed to be smaller and easier to compare visually.

Instantiate an object of type Prob04MyClass

The next statement in **Listing 1** instantiates an object of the class named **Prob04MyClass**, passing the random value as a parameter to the constructor. At this point, I will put the explanation of the class named **Prob04** on temporary hold and explain the class named **Prob04MyClass**, which begins in **Listing 2**.

LISTING 2: **Beginning of the class named Prob04MyClass**.

```
class Prob04MyClass extends Prob04{
  private int data;

  public Prob04MyClass(int inData){//constructor
    System.out.println("Prob04");
    System.out.println("Dick");
    data = inData;
  }//end constructor
```

Extends the abstract class named Prob04

First note that the class named **Prob04MyClass** extends the abstract class named **Prob04**.

Among other things, this means that either this class must override the abstract method named **getData** that was declared in the superclass, or this class must also be declared abstract.

Does it override getData?

Seeing that this class isn't declared abstract, we can surmise at this point that it does override the abstract method named **getData.** We will see more about this later.

Beginning of the class named Prob04MyClass

The class definition in **Listing 2** begins by declaring a private instance variable of type **int** named **data.** Note that it does not initialize the variable. Therefore, the value is automatically initialized to an **int** value of zero.

The constructor

Then **Listing 2** defines the constructor for the class. The first two statements in the constructor cause the first two lines of text shown in **Figure 1** to be displayed on the command line screen.

Save the incoming parameter value

The last line in the constructor saves the incoming value in the instance variable named **data,** overwriting the default value of zero that it finds there.

This statement is more in keeping with the intended usage of a constructor than the first two statements. The primary purpose of a constructor is to assist in the initialization of the *state of an object,* which depends on the values stored in its variables.

Override the abstract getData method

Listing 3 overrides the abstract **getData** method declared in the abstract superclass named **Prob04** and inherited into the subclass named **Prob04MyClass.**

LISTING 3: **Override the abstract getData method.**

```
  public int getData(){//overridden abstract method
    return data;
  }//end getData()
```

Very simple behavior

Although the overridden version of the method simply returns a copy of the value stored in the private instance variable named **data**, it is concrete and can be executed. We will see later that it is called in the **main** method of the driver class named**Prob04** in **Listing 1**.

Override the toString method

The ultimate superclass of every class is the predefined system class named **Object**. The **Object** class defines eleven methods with default behavior, including the method named **toString**.

Listing 4 overrides the inherited **toString** method, overriding the default behavior of the method insofar as objects of the class named **Prob04MyClass** are concerned.

LISTING 4: **Override the toString method.**

```
  public String toString(){//overridden method
    return "Baldwin";
  }//end overloaded toString()

}//end class Prob04MyClass
```

Default behavior of the toString method

If the **toString** method had not been overridden in the **Prob04MyClass** class, calling the **toString** method on an object of the class would return a string similar to that shown in Figure 2.

Default behavior of the toString method .

```
Prob04MyClass@42e816
```

Figure 2: Default behavior of the toString method.

Figure 2 shows the default behavior of the **toString** method as defined in the **Object** class. For this program, only the six hexadecimal digits at the end would change from one run to the next.

More on the default behavior of the toString method

Furthermore, if the **toString** method had not been overridden in the **Prob04MyClass** class, the output produced by the program on the command line screen would be similar to that shown in **Figure 3** instead of that shown in **Figure 1**.

More on the default behavior of the toString method.

```
Prob04
Dick
Prob04MyClass@42e816
-34
-34
```

Figure 3: More on the default behavior of the toString method.

Compare to see the difference

If you compare **Figure 3** with **Figure 1**, you will see that the difference results from the fact that the overridden version of the **toString** method in **Listing 4** returns *"Baldwin"* as a string rather than returning the default string shown in **Figure 2**.

The end of the class named Prob04MyClass

Listing 4 signals the end of the class definition for the class named **Prob04MyClass.** Therefore, it is time to return to the explanation of the driver class shown in **Listing 1**.

Display information about the object

When the **Prob04MyClass** constructor returns, **Listing 1** calls the **println** method passing a reference to the new object as a parameter.

Many overloaded *(not overridden)* versions of println

There are many overloaded versions of the **println** method, each of which requires a different type of incoming parameter or parameters.

For example, different overloaded versions of the method know how to receive incoming parameters of each of the different primitive types, convert them to characters, and display the characters on the screen.

An incoming parameter of type Object

There is also an overloaded version of the **println** method that requires an incoming parameter of type **Object**. That is the version of the method that is executed when the reference to this object is passed to the method in **Listing 1**.

One object, several types

Recall that the reference to this object can be treated as its true type, or as the type of any superclass. Therefore, the reference can be treated as any of the following types:

- **Prob04MyClass**
- **Prob04**
- **Object**

Will satisfy type requirement...

Because it can be treated as type **Object**, it will satisfy the type requirement for the overloaded version of the **println** method that requires an incoming parameter of type **Object** .

Call the toString method

The first thing that this version of the **println** method does is to call the **toString** method on the incoming reference. Then it displays the string value returned by the **toString** method on the screen.

In this case, the overridden **toString** method returns the string *"Baldwin"*, which is what you see, displayed in **Figure 1**.

Runtime polymorphism

This is a clear example of an OOP concept known as *runtime polymorphism.*

Runtime polymorphism is much too complicated to explain in this module. However, I explain it in detail in my online OOP modules and I strongly recommend that you study it there until you thoroughly understand it.

A critical concept

It is critical that you understand runtime polymorphism if you expect to go further in Java OOP.

It is almost impossible to write a useful Java application without making heavy use of runtime polymorphism, because that is the foundation of the event driven Java graphical user interface system.

Call the overridden getData method

The next statement in **Listing 1** calls the overridden **getData** method and displays the return value.

As you saw earlier, this method returns a copy of the random value that was received and saved by the constructor for the**Prob04MyClass** class in **Listing 2**.

Display the original random value

Finally, the last statement in the **main** method in **Listing 1** displays the contents of the instance variable named **randomNumber**. This variable contains the random value that was passed to the constructor for the **Prob04MyClass** earlier in**Listing 1**.

The two values must match

Therefore, the final two statements in the **main** method in **Listing 1** display the same random value. This is shown in the command line screen output in **Figure 1**.

The program terminates

After displaying this value, the **main** method terminates causing the program to terminate.

Run the program

I encourage you to copy the code from **Listing 5**, compile it and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

You have learned about abstract methods, abstract classes, and overridden methods in this module. Very importantly, you have learned about overriding the **toString** method.

What's next?

You will learn more about indirection, array objects, and casting in the next module.

Miscellaneous

This section contains a variety of miscellaneous information.

NOTE:

Housekeeping material

- Module name: Java OOP: Abstract Methods, Abstract Classes, and Overridden Methods
- File: Java3008.htm
- Published: August 2, 2012
- Revised: --

NOTE:

Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listings

A complete listing of the program discussed in this module is shown in **Listing 5** below.

LISTING 5: **Complete program listing.**

```
/*File Prob04 Copyright 2001, R.G.Baldwin
Rev 12/16/08
**********************************************************/
import java.util.*;

abstract class Prob04{
  public static void main(String[] args){
    Random generator = new Random(new Date().getTime());
    int randomNumber = (byte)generator.nextInt();

    Prob04 objRef = new Prob04MyClass(randomNumber);
    System.out.println(objRef);
    System.out.println(objRef.getData());
    System.out.println(randomNumber);
  }//end main

  //Declare the signature of an abstract class.
  public abstract int getData();

}//end class Prob04
//========================================================//

class Prob04MyClass extends Prob04{
  private int data;

  public Prob04MyClass(int inData){//constructor
    System.out.println("Prob04");
    System.out.println("Dick");
    data = inData;
  }//end constructor

  public int getData(){//overridden abstract method
    return data;
  }//end getData()

  public String toString(){//overridden method
    return "Baldwin";
  }//end overloaded toString()

}//end class Prob04MyClass
```

~~~ End of Article ~~~

Session 11: Interfaces and Nested Classes

Interfaces

Source	http://en.wikibooks.org/wiki/Java_Programming/Interfaces
Date of Retrieval	07/01/2013

Interfaces

Java does not allow you to create a subclass from two classes. There is no multiple inheritance. The major benefit of that is that all Java objects can have a common ancestor. That class is called `Object`. All Java classes can be up-casted to `Object`. Example:

```
class MyObject
{
 ...
}
```

When you type the above code, it actually means the following:

```
class MyObject extends Object // The compiler adds 'extends Object'. if not specified
{
 ...
}
```

So, it can be guaranteed that all the `Object` class methods are available in all Java classes. This makes the language simpler.

To mimic multiple inheritance, Java offers *interfaces*, which are similar to abstract classes. In interfaces all methods are abstract by default, without the **`abstract`** keyword. Interfaces have no implementation and no variables, but constant values can be defined in interfaces. However, a single class can implement as many interfaces as required.

```
public interface MyInterface
{
  public static final String CONSTANT = "This value can not be changed";

  public String methodOne(); // This method must be implemented by the class implementing this interface
  ...
}
```

...

```
public class MyObject implements MyInterface
{
  // Implement MyInterface interface
  public String methodOne()
  {
```

```
    ...
  }
}
```

All constants are assumed to be `final` `static` and all methods `public`, even if the keywords are missing.

~~~ End of Article ~~~

Nested Classes

Source	http://scjp.wikidot.com/nested-classes
Date of Retrieval	07/01/2013

- Classes that are declared inside the body of a class are called "nested classes".
- The following are the main reasons behind the concept of nested classes in Java:
    - Grouping of logical classes
        - When designing a class, we always try to create well-focused classes - with a unique behavior and purpose. Let us imagine we have designed classes A and B on the same principle. Later we found, class B can only be used by class A. In such cases, we can simply put class B inside class A.
    - Encapsulation
        - By writing nested classes, we have hidden them from the world and made them available only to their enclosing class.
    - Maintainability and re-usability of code
        - Encapsulation brings the beauty of maintainability of code. In our earlier example, we can write another class B which is visible t o the entire world. This has nothing to do with the one already present inside class A.
- Nested class is of 2 kinds:
    - Inner class
    - Static nested class
- Inner class is of 3 types:
    - Inner class
    - Method-local inner class
    - Anonymous inner class
- Nested class behaves just like any other member of its enclosing(outer) class.
- It has access to all the members of its enclosing class.

Inner Class

- We define the term "inner class" to the nested class that is:
    - declared inside the body of another class.
    - not declared inside a method of another class.
    - not a static nested class.
    - not an anonymous inner class.
- An example:

```
class Outer{
    class Inner{
    }
}
```

- When we compile the above code we get 2 class files:
    - Outer.class
    - Outer$Inner.class
- Notice that inner class is tied to its outer class though it is still a separate class.

- An inner class cannot have any kind of static code including the public static void main(String[] args).
- Only classes with "public static void main(String[] args)" can be called using "java" command.
- In our earlier example, Inner class didn't have a static main method. So, we can't call java Outer$Inner!
- The inner class is just like any other member of its enclosing class.
- It has access to all of its enclosing class' members including private.

Instantiating an Inner Class

- Since inner class can't stand on its own, we need an instance of its outer class to tie it.
- There are 2 ways to instantiate an instance of inner class:
  - From within its outer class
  - From outside its outer class

```
class Outer{
    private String s = "Outer string"; //Outer instance variable
    Inner i1 = new Inner();
    void getS(){
        System.out.println(s);
    }
    void getInnerS(){
        System.out.println(i1.s);
    }
    class Inner{
        private String s = "Inner string"; //Inner instance variable,
uninitialized
        void getS(){
            System.out.println(s);
        }
        void getOuterS(){
            System.out.println(Outer.this.s);
        }
    }
    public static void main(String[] args){
        Outer o = new Outer();
        Outer.Inner oi = o.new Inner();//can also be new Outer().new
Inner();
        o.getS();
        oi.getS();
        o.getInnerS();
        oi.getOuterS();
    }
}
Output:
Outer string
Inner string
Inner string
Outer string
```

- Another excellent example:

```
class InnerTest{
    public static void main(String[] args){
        Outer o = new Outer();
        Outer.Inner i = o.new Inner();//one way
        i.seeOuter();
```

```
        i = new Outer().new Inner();//another way
        i.seeOuter();
    }
}
class Outer{
    private String s = "outer s";
    void makeAndSeeInner(){
        System.out.println(this);//refers to Outer.this
        Inner i = new Inner();//No need of Outer.this explicitly, because,
Outer.this already exists here.
        i.seeOuter();
    }
    void seeInner(){
        System.out.println(s);//How to see Inner s here? You can't, because
Inner.this not present.
    }
    strictfp class Inner{
        private String s = "inner s";
        void seeOuter(){
            System.out.println(this);
            System.out.println(Outer.this.s);//Need to mention Outer
because this refers to Inner.this here.
            System.out.println(s);//Or, this.s
        }
    }
    abstract class AbInner{
        private String s = "abstract inner s";
        void seeOuter(){
            System.out.println(this);//this refers to the subclass not the
abstract class.
            System.out.println(Outer.this.s);
            System.out.println(s);
        }
        abstract void abMethod();
    }
    class Some extends AbInner implements Runnable, Animal{//can extend and
implement
        public void run(){}
        void abMethod(){
            System.out.println(this);
            System.out.println(super.s);
        }
    }
    public static void main(String[] args){
        Inner i = new Outer().new Inner();
        //Inner i = new Inner();//can't exist w/o outer class instance
        i.seeOuter();
        Outer o = new Outer();
        o.makeAndSeeInner();
        o.seeInner();
        //new Outer().makeAndSeeInner();
        Some so = new Outer().new Some();
        so.seeOuter();
        so.abMethod();
    }
}
interface Animal{
}
```

Run it and check your output. How to run? Look into the code and you'll definitely understand how to do that. :)

Method-Local Inner Classes

- A method-local inner class is defined within a method of the enclosing class.
- For the inner class to be used, you must instantiate it and that instantiation must happen within the same method, but after the class definition code.
- A method-local inner class cannot use variables declared within the method (including parameters) unless those variables are marked final.
- The only modifiers you can apply to a method-local inner class are abstract and final. (Never both at the same time, though.)

Anonymous Inner Classes

- Inner classes that are declared without a name are called anonymous inner classes.
- Anonymous inner classes have no name, and their type must be either a subclass of the named type or an implementer of the named interface.
- An anonymous inner class is always created as part of a statement; don't forget to close the statement after the class definition with a curly brace. This is a rare case in Java, a curly brace followed by a semicolon.
- Because of polymorphism, the only methods you can call on an anonymous inner class reference are those defined in the reference variable class (or interface), even though the anonymous class is really a subclass or implementer of the reference variable type.
- An anonymous inner class can extend one subclass or implement one interface, unlike non-anonymous classes (inner or otherwise); an anonymous inner class cannot do both. In other words, it cannot both extend a class and implement an interface, nor can it implement more than one interface.
- An argument-local inner class is declared, defined, and automatically instantiated as part of a method invocation. The key to remember is that the class is being defined within a method argument, so the syntax will end the class definition with a curly brace, followed by a closing parenthesis to end the method call, followed by a semicolon to end the statement: });

Static Nested Classes

- Nested classes that are declared "static" are called static nested classes.
- Static nested classes are inner classes marked with the static modifier.
- A static nested class is not an inner class; it's a top-level nested class.
- Because the nested class is static, it does not share any special relationship with an instance of the outer class. In fact, you don't need an instance of the outer class to instantiate a static nested class.
- Instantiating a static nested class requires using both the outer and nested class names as follows:

```
BigOuter.Nested n = new BigOuter.Nested();
```

- A static nested class cannot access non-static members of the outer class, since it does not have an implicit reference to any outer instance (in other words, the nested class instance does not get an outer this reference).
- An example:

```
class StaticOuter{
    String a = "StaticOuter string";
    static String b = "StaticOuter static string";
```

```
    void seeStaticInner(){
        //System.out.println(nonstatic);//cannot find symbol
        //System.out.println(StaticInner.nonstatic);//nonstatic is not
static to access like this!
        System.out.println(new StaticInner().nonstatic);//OK
        System.out.println(StaticInner.s);//OK, s is static
    }
    public static void main(String[] args){
        //System.out.println(s);//Doesn't compiles without writing the
exact location of s
        System.out.println(StaticInner.s);
        StaticOuter so = new StaticOuter();
        so.seeStaticInner();
    }
    static class StaticInner{
        String nonstatic = "StaticInner nonstatic string";
        static String s = "StaticInner static string";
        public static void main(String... args){
            //System.out.println(nonstatic);//cannot be referenced from a
static context
            System.out.println(s);
            System.out.println(b);//OK, b is a static string. But not 'a'
which is non-static!
        }
    }
}
class SomeOther{
    public static void main(String[] args){
        System.out.println(StaticOuter.StaticInner.s);//Write the exact
location of s
        StaticOuter.StaticInner si = new StaticOuter.StaticInner();//To
access nonstatic members we need an object or 'this'
        System.out.println(si.nonstatic);//No 'this' exists in static
context!
        System.out.println(si.s);//'si' is simply a fake! To the end it
will be StaticOuter.StaticInner!
    }
}
Output:
$java StaticOuter
StaticInner static string
StaticInner nonstatic string
StaticInner static string

$java StaticOuter$StaticInner
StaticInner static string
StaticOuter static string

$java SomeOther
StaticInner static string
StaticInner nonstatic string
StaticInner static string
```

~~~ End of Article ~~~

Session 12: Exceptions

Java Programming/Exceptions

Source	http://en.wikibooks.org/wiki/Java_Programming/Exceptions
Date of Retrieval	07/01/2013

The ideal time to catch an error is at compile time, before you even try to run the program. However, not all errors can be detected at compile time. The rest of the problems must be handled at run time through some formality that allows the originator of the error to pass appropriate information to a recipient who will know how to handle the difficulty properly.

Improved error recovery is one of the most powerful ways that you can increase the robustness of your code. Error recovery is a fundamental concern for every program you write, but it's especially important in Java, where one of the primary goal is to create program components for others to use. *To create a robust system, each component must be robust*. By providing a consistent error-reporting model using exceptions, Java allows components to reliably communicate problems to client code.

Flow of code execution

In Java, there are two main flows of code executions.

- Normal main sequential code execution, the program doing what it meant to accomplish
- Exception handling code execution, the main program flow was interrupted by an error or some other condition that prevent the continuation of the normal main sequential code execution.

Exception

Exceptions are Java's way of error handling. Whenever an unexpected condition occurs, an exception can be thrown with an exception object as a parameter. It means that the normal program control flow stops and the search for a `catch` block begins. If that is not found at the current method level the search continues at the caller method level, until a matching `catch` block is found. If none is found the exception will be handled by the JVM, and usually the java program terminates.

When a `catch` "matching" block is found, that block will be executed, the exception object is passed to the block as a parameter. Then normal program execution continues after the `catch` block.

Exception Object

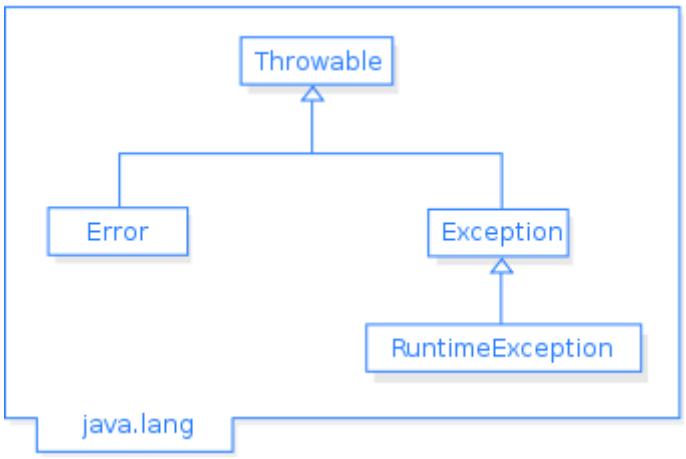
This is the object that is "thrown" as a parameter from the error, and passed to the `catch` block. Exception object encapsulates the information about the error's location and its nature. All Exception objects must be inherited from the `java.lang.Throwable`. See the UML diagram below.

Matching rule

A thrown exception object can be caught by the `catch` keyword and specifying the exception object's class or its super-class.

Naming convention

It is good practice to add Exception to all exception classes. Also the name of the exception should be meaningful, should represent the problem. For example `CustomerNotFoundException` indicate that customer was not found.



~~~ End of Article ~~~

Throwing and Catching Exceptions

Source	http://en.wikibooks.org/wiki/Java_Programming/Throwing_and_Catching_Exceptions
Date of Retrieval	07/01/2013

Language compilers are adept at pointing out most of the erroneous code in a program; however there are some errors that only become apparent when the program is executed. Consider the code in Listing 1.1; here, the program defines a method **divide** that does a simple division operation taking two integers as parameter arguments and returning the result of their division. It can safely be assumed that when the **divide(4, 2)** statement is called, it would return the number **2**. However, consider the next statement, where the program relies upon the provided command line arguments to generate a division operation. What if the user provides the number zero (**0**) as the second argument? We all know that division by zero is impossible, but the compiler couldn't possibly have anticipated the user providing zero as an argument.

Listing 1.1: A simple division operation

Exceptions/ExceptionTutorial01.java

```
1. public class ExceptionTutorial01
2. {
3.   public static int divide(int a, int b)
4.   {
5.     return a / b;
6.   }
7.   public static void main(String[] args)
8.   {
9.     System.out.println( divide(4, 2) );
10.          if(args.length > 1)
11.          {
12.            int arg0 = Integer.parseInt(args[0]);
13.            int arg1 = Integer.parseInt(args[1]);
14.            System.out.println( divide(arg0, arg1) );
15.          }
16.        }
17.      }
```

Output for: java ExceptionTutorial01 1 0

```
2
Exception in thread "main" java.lang.ArithmeticException: / by zero
     at ExceptionTutorial01.divide(ExceptionTutorial01.java:5)
```

```
    at ExceptionTutorial01.main(ExceptionTutorial01.java:14)
```

Such *exceptional code* that results in erroneous interpretations at program runtime usually results in errors that are called *exceptions* in Java. When the Java interpreter encounters an exceptional code, it halts execution and displays information about the error that occurs. This information is known as a **stack trace**. The **stack trace** in the above example tells us more about the error, such as the thread -`"main"` - where the exception occurred, the type of exception - `java.lang.ArithmeticException`, a comprehensible display message - `/ by zero`, and the exact methods and the line numbers where the exception may have occurred.

Exception arguments

In the code above, the exception object for `java.lang.ArithmeticException` was generated by the Java interpreter itself. However, there are times when you would need to explicitly create your own exceptions. As with any object in Java, you always create exceptions on the heap using **new**, which allocates storage and calls a constructor. There are two constructors in all standard exceptions:

1. The default constructor; and,
2. A constructor taking a string argument so that you can place pertinent information in the exception.

Listing 1.2: Instance of an exception object with the default constructor

```
new Exception();
```

Listing 1.3: Instance of an `Exception` object by passing string in constructor

```
new Exception("Something unexpected happened");
```

This string can later be extracted using various methods, as you'll see.

The keyword **throw** produces a number of interesting results. After creating an exception object with **new**, you give the resulting reference to **throw**. The object is, in effect, "returned" from the method, even though that object type isn't normally what the method is designed to return. A simplistic way to think about exception handling is as a different kind of return mechanism, although you get into trouble if you take that analogy too far. You can also exit from ordinary scopes by throwing an exception. In either case, an exception object is returned, and the method or scope exits.

Listing 1.4: Throwing an exception results in an unexpected return from the method

```
throw new Exception();
```

Anything after the **throw** statement would not be executed, unless the *thrown* exception is handled properly. Any similarity to an ordinary return from a method ends here, because *where* you return is some place completely different from where you return for a normal method call, i.e., you end up in an appropriate exception handler that might be far away - many levels on the call stack - from where the exception was thrown.

In addition, you can throw any type of **Throwable**, which is the exception root class. Typically, you'll throw a different class of exception for each different type of error. The information about the error is represented both inside the exception object and implicitly in the name of the exception class, so someone in the bigger context can figure out what to do

with your exception. Often, the only information is the type of exception, and nothing meaningful is stored within the exception object.

Catching an exception

To see how an exception is caught, you must first understand the concept of a *guarded region*. This is a section of code that might produce exceptions and is followed by the code to handle those exceptions.

The `try` block

If you are inside a method and you throw an exception (or another method that you call within this method throws an exception) that method will exit in the process of throwing. If you don't want a **throw** to exit the method, you can set up a special block within that method to capture the exception. This is called the *try block* because you "try" your various method calls there. The **try** block is an ordinary scope preceded by the keyword **try**.

Listing 1.5: A basic try block

```
try
{
  // Code that might generate exceptions
}
```

If you were checking for errors carefully in a programming language that didn't support exception handling, you'd have to surround every method call with setup and error-testing code, even if you call the same method several times. With exception handling, you put everything in a **try** block and capture all the exceptions in one place. This means your code is much easier to write and read because the goal of the code is not confused with the error checking.

Exception handlers

Of course, the thrown exception must end up some place. This "place" is the *exception handler*, and there's one for every exception type you want to catch. Exception handlers immediately follow the **try** block and are denoted by the keyword **catch**:

Listing 1.6: Exception handling with catch blocks

```
try
{
  // Suppose the code here throws two exceptions:
  // NullPointerException and NumberFormatException,
  // then each is handled in a separate catch block.
}
catch(NullPointerException ex)
{
  // Exception handling code for the NullPointerException
}
catch(NumberFormatException ex)
{
  // Exception handling code for the NumberFormatException
}
// etc...
```

Using the syntax in Listing 1.6, we can write the potentially problematic division code in Listing 1.1 as under.

Listing 1.7: Catching 'division by zero' errors.

```
int result = 0;
try
{
  result = a / b;
}
catch(ArithmeticException ex)
{
  result = 0;
}
return result;
```

Using the code in Listing 1.7, the program would now not abruptly terminate but continue its execution whilst returning zero for a division by zero operation. I know it's not the correct answer, but hey, it saved your program from terminating abnormally.

Exception classes in the JCL

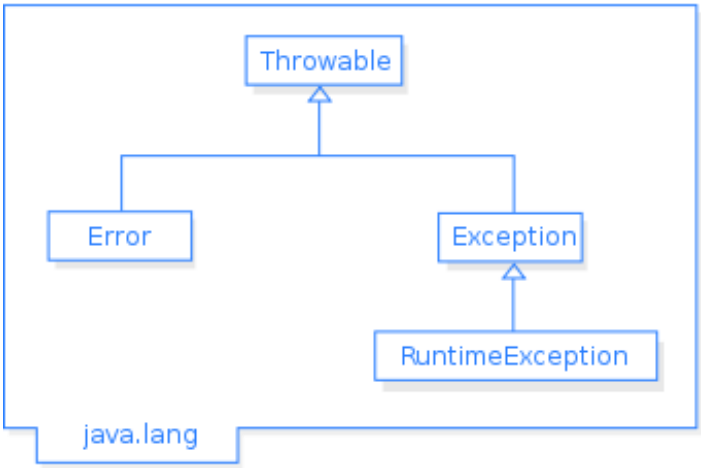


Figure 1.1: The exception classes and their inheritance model in the JCL.

The box 1.1 below talks about the various exception classes within the `java.lang` package of the JCL.

Box 1.1: The Java exception classes

Throwable

The Throwable class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java

Virtual Machine or can be thrown by the Java throw statement.

A throwable contains a snapshot of the execution stack of its thread at the time it was created. It can also contain a message string that gives more information about the error. Finally, it can contain a cause: another throwable that caused this throwable to get thrown. The cause facility is new in release 1.4. It is also known as the chained exception facility, as the cause can, itself, have a cause, and so on, leading to a "chain" of exceptions, each caused by another

Error

An Error indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions.

Exception

The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch. Also this is the class that a programmer may want to extend when adding business logic exceptions.

RuntimeException

RuntimeException is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine. A method is not required to declare in its throws clause any subclasses of RuntimeException that might be thrown during the execution of the method but not caught.

Catch clauses

There are three distinct types of catch clauses used by Java programmers. We have already explored the first rule in the code listings above. Given below are all three clauses; we will explain all three in greater detail in the next sections.

1. The *catch-list* clause
2. The *catch-any* clause
3. The *multi-catch* clause

The *catch-list* clause

Using the **catch-list** clause, you provide one handle for every one type of exception. So, for instance if a *guarded block* generates a **NullPointerException** or an **ArithmeticException**, you have to provide two exception handling **catch blocks** - one for each exception. Refer to code listing 1.6 for an example.

The *catch-any* clause

There are times when too many blocks are just verbose, especially when all exceptions are handled in a similar manner. What you need here is a **catch-any** clause, where all exceptions are handled in a single **catch block**. Because all exceptions in Java are the sub-class

Note:
You can also use the `java.lang.Throwable` class here, since **Throwable** is the parent class for the *application-specific* **Exception** classes. However, this is discouraged in Java programming circles. This is because **Throwable** happens to also be the parent class for the *non-application specific* **Error** classes which are not meant to be handled explicitly as they are catered for by the JVM itself.

of `java.lang.Exception` class, you can have a single **catch block** that catches an exception of type **Exception** only. Hence the compiler is fooled into thinking that this block can handle any exception. Using this, the code in Listing 1.6 can be rewritten as follows:

Listing 1.8: The catch-any clause

```
try
{
  // ...
}
catch(Exception ex)
{
  // Exception handling code for ANY exception
}
```

The *multi-catch* clause

As of JDK 7, Java added another convenient feature in their exception handling routines - the **multi-catch** clause. This is a combination of the previous two **catch** clauses and lets you handle exceptions in a single handler while also maintaining their types. So, instead of being boxed into a parent Exception super-class, they retain their individual types.

Listing 1.9: The multi-catch clause

```
try
{
  // ...
}
catch(NullPointerException | NumberFormatException ex)
{
  // Exception handling code specifically for the NullPointerException
  // and the NumberFormatException
}
```

Example of handling exceptions

Let's examine the following code:

```
public void methodA() throws SomeException
{
    // Method body
}
public void methodB() throws CustomException, AnotherException
{
```

```
    // Method body
}
public void methodC()
{
    methodB();
    methodA();
}
```

In the code sample, `methodC` is invalid. Because `methodA` and `methodB` pass (or throw) exceptions, `methodC` must be prepared to handle them. This can be handled in two ways: a **`try-catch`** block, which will handle the exception within the method and a **`throws`** clause which would in turn throw the exception to the caller to handle. The above example will cause a compilation error, as Java is very strict about exception handling. So the programmer forced to handle any possible error condition at some point.

A method can do two things with an exception. Ask the calling method to handle it by the **`throws`** declaration. Or handle the exception inside the method by the **`try-catch`** block.

To construct a throws declaration, add `throws ExceptionName` (additional exceptions can be added with commas). To construct a **`try-catch`** block, use the following syntax:

```
try
{
  // Guarded region that will catch any exception that
  // occurs within this code.
}
catch(Exception ex)
{
  // Caught exceptions are handled here
}
finally
{
  // This clause is optional. Code here is executed
  // regardless of exceptions being thrown
}
```

To work correctly, the original code can be modified in multiple ways. For example, the following:

```
public void methodC() throws CustomException, SomeException
{
  try
  {
    methodB();
  }
  catch(AnotherException e)
  {
    // Handle caught exceptions.
  }
  methodA();
}
```

The `AnotherException` from `methodB` will be handled locally,
while `CustomException` and `SomeException` will be thrown to the caller to handle it.

Application Exceptions

Application Exception classes should extend the `java.lang.Exception` class. Some of the JDK classes also throw exception objects inherited from `java.lang.Exception`. If any of those Exception object is thrown, **it must be caught by the application**some point, by a catch-block. The compiler will enforce that there is a catch-block associated with an exception thrown, if the thrown exception object is inherited
from `java.lang.Exception` and it is not the `java.lang.RuntimeException` or its inherited objects. However, `java.lang.RuntimeException` or its inherited objects can be caught by the application, but that is not enforced by the compiler.

Let's see what is the catching criterion for a catch block to catch the "thrown" exception?

A catch-block will "catch" a thrown exception if and only if:

- the thrown exception object is the same as the exception object specified by the catch-block
- the thrown exception object is the subtype of the exception object specified by the catch-block

```
try
{
  throw new Exception("This will be caught below");
}
catch(Exception ex)
{
  // The thrown object is the same that what is specified at the catch-
block
}
try
{
  throw new NullPointerException("This will be caught below");
}
catch(Exception e)
{
  // NullPointerException is subclass of the Exception class.
}
```

There can be more than one catch-block for a try-block. The catching blocks evaluated sequentially one by one. If a catch-block catches the exception, the others will not be evaluated.

Example:

```
try
{
  throw new NullPointerException("This will be caught below");
}
catch(Exception e)
{
```

```
  // The NullPointerException thrown in the code above is caught here...
}
catch(NullPointerException e)
{
  // ..while this code is never executed and the compiler returns an error
}
```

Because `NullPointerException` is a subclass of the `Exception` class, all `NullPointerExceptions` will be caught by the first catch-block.

Instead the above code should be rewritten as follows:

```
try
{
  throw new NullPointerException("This will be caught below");
}
catch(NullPointerException e)
{
  // The above NullPointerException will be caught here...
}
catch(Exception e)
{
  // ..while other exception are caught here.
}
```

Runtime Exceptions

The `java.lang.RuntimeException` exception class is inherited from `java.lang.Exception`. It is a special exception class, because catching this exception class or its subclasses are not enforced by the Java compiler.

runtime exception

> Runtime exceptions are usually caused by data errors, like arithmetic overflow, divide by zero,... . Runtime exceptions are not business related exceptions. In a well debugged code, runtime exceptions should not occur. Runtime exceptions should only be used in the case that the exception could be thrown by and only by something hard-coded into the program. These should not be able to be triggered by the software's user(s).

NullPointerException

`NullPointerException` is a `RuntimeException`. In Java, a special **`null`** can be assigned to an object reference.`NullPointerException` is thrown when an application attempts to use an object reference, having the **`null`** value. These include:

- Calling an instance method on the object referred by a null reference.
- Accessing or modifying an instance field of the object referred by a null reference.
- If the reference type is an array type, taking the length of a null reference.
- If the reference type is an array type, accessing or modifying the slots of a null reference.
- If the reference type is a subtype of `Throwable`, throwing a null reference.

Applications should throw instances of this class to indicate other illegal uses of the null object.

```
Object obj = null;
obj.toString();  // This statement will throw a NullPointerException
```

The above code shows one of the pitfall of Java, and the most common source of bugs. No object is created and the compiler does not detect it. `NullPointerException` is one of the most common exceptions thrown in Java.

Why do we need `null`?

The reason we need it is because many times we need to create an object reference, before the object itself is created. Object references cannot exist without a value, so we assign the **`null`** value to it.

```
public Customer getCustomer()
{
  Customer customer = null;
  try
  {
    ...
    customer = createCustomer();
    ...
  }
  catch(Exception ex)
  {
    ...
  }
  return customer;
}
```

In the above code we want to create the Customer inside the try-block, but we also want to return the object reference to the caller, so we need to create the object reference outside of the try-block, because of the scoping rule in Java. Incorrect error-handling and poor contract design can be a pitfall with any programming language; this is also true for Java.

~~~ End of Article ~~~

Session 13: New Date and Time API

ZonedDateTime API

Source	http://www.threeten.org/articles/zoned-date-time.html
Date of Retrieval	30/09/2016

The `ZonedDateTime` class represents a date-time with a time-zone. This class stores all date and time fields, to a precision of nanoseconds, and a time-zone, with a zone offset used to handle ambiguous local date-times. For example, the value "2nd October 2007 at 13:45.30.123456789 +02:00 in the Europe/Paris time-zone" can be stored in a `ZonedDateTime`.

This class handles conversion from the local time-line of `LocalDateTime` to the instant time-line of Instant. The difference between the two time-lines is the offset from UTC/Greenwich, represented by a `ZoneOffset`.

A `ZonedDateTime` holds state equivalent to three separate objects, a `LocalDateTime`, a `ZoneId` and the resolved `ZoneOffset`. The offset and local date-time are used to define an instant when necessary. The `zone ID` is used to obtain the rules for how and when the offset changes. The offset cannot be freely set, as the zone controls which offsets are valid.

Creating a ZonedDateTime

There are a number of ways to create a `ZonedDateTime`.

Current Date Time with Zone information

Current date-time with zone information can be created using the following now methods:

```
// output is based on code run in Europe/London time zone

ZonedDateTime a = ZonedDateTime.now();                          //2013-03-02T21:52:25.371+01:00[Europe/London]
ZonedDateTime b = ZonedDateTime.now(ZoneId.of("Europe/Paris")); //2013-03-02T22:52:25.374+02:00[Europe/Paris]

ZonedDateTime c = ZonedDateTime.now(aClock );                   //2013-07-02T16:52:25.374-04:00[America/Indiana/Indianapolis]
// where Clock aClock = Clock.fixed(Instant.now(),
ZoneId.of("America/Indiana/Indianapolis"));
```

The first method (a) uses the Java default time-zone, as per `TimeZone.getDefault()`. The second method (b) allows the time-zone to be explicitly controlled. The third method (c) uses a `Clock` object, which provides further control over the current instant and time-zone. This will come handy in testing scenarios as we can provide an alternate clock.

Using fields

If you want to hard code the creation of `ZonedDateTime`, or have fields available, these factory methods can be used:

```
ZonedDateTime a = ZonedDateTime.of(aLocalDateTime,
ZoneId.of("Europe/Paris"));                                    // 2013-12-
18T14:30+01:00[Europe/Paris]
// where LocalDateTime aLocalDateTime = LocalDateTime.of(2013, 12, 18, 14,
30)


ZonedDateTime b = ZonedDateTime.ofInstant(Instant.ofEpochSecond(1),
ZoneId.of("Europe/Paris"));              // 1970-01-
01T01:00:01+01:00[Europe/Paris


ZonedDateTime c = ZonedDateTime.ofInstant(aLocalDateTime,
preferredZoneOffset, ZoneId.of("Europe/Paris"));   // 2013-12-
18T14:30+01:00[Europe/Paris]
// where ZoneOffset preferredZoneOffset = ZoneOffset.ofHours(1)


ZonedDateTime d = ZonedDateTime.ofLocal(aLocalDateTime,
ZoneId.of("Europe/Paris"), preferredZoneOffset);     // 2013-12-
18T14:30+01:00[Europe/Paris]
ZonedDateTime e = ZonedDateTime.ofStrict(aLocalDateTime,
preferredZoneOffset, ZoneId.of("Europe/Paris"));  // 2013-12-
18T14:30+01:00[Europe/Paris]
```

Remember that `preferredZoneOffset` can be applied only if available in the valid offset from UTC/GMT of the `localDateTime` as defined by `zoneRules` of given Zone. If it is invalid then it is corrected to available offset and that value is used except for method (e) which throws an exception in this case.

Although the methods (d) and (e) has got similar arguments, there is a fundamental difference in the way offset is applied. For `localDateTime` in Europe/London the valid offset with Paris is one hour for this time. For method (d) if we provide an invalid offset say 2 hours, it will provide correct offset of one hour and use it while method (e) will throw an exception

```
ZonedDateTime f = ZonedDateTime.ofLocal(aLocalDateTime,
ZoneId.of("Europe/Paris"), ZoneOffset.ofHours(2));  // 2013-12-
18T14:30+01:00[Europe/Paris]
ZonedDateTime g = ZonedDateTime.ofStrict(aLocalDateTime,
ZoneOffset.ofHours(2), ZoneId.of("Europe/Paris")); // throws
DateTimeException
```

~~~ End of Article ~~~

TimeZone Class

Source	http://www.zhuwu.me/blog/posts/handle-time-zone-in-java
Date of Retrieval	30/09/2016

It is important to process and present date and time correctly when building an application, especially when your application is going to be used by users from more than one time zones. This article will give you more insights on how to deal time zone in a correct way in Java.

Before we dig into time zone, we need to understand timestamp in Java. Timestamp is defined as number of milliseconds since 00:00:00 Thursday, 1 January 1970 UTC (this time is also known as Epoch time). From the definition, it is easy to understand that a timestamp is not related to any time zone. For example, time stamp of 29 Feb 2016, 12:34:56 (UTC+8) is 1456720496000, which is identical to time stamp of 29 Feb 2016, 05:34:56 (UTC+1).

Since the same time in different time zone has the same timestamp, the timestamp is a good reference point for us when dealing date and time in Java. The only thing we need to cater is how to translate a time zone based time into timestamp and how to present the time stamp in human readable format according to target time zone. Java provides us the following classes to do the manipulations:

Class	Functionality	Set Time Zone
Date	Wrapper of timestamp.	Not applicable.
SimpleDateFormat	1) Parse a string to a Date object according to certain pattern. 2) Display a Date object according to certain pattern.	Supply time zone when calling setTimeZone method.
Calendar	Maintain and manipulate a set of calendar fields such as year, month, day of month, hour, minute, second, millisecond, etc.	1) Supply time zone when calling getInstance method. 2) Supply time zone when calling setTimezone method.

Maybe the target users of your application come from one time zone only, but you still need to take good care of time zone because time zone can be changed by authority. For example, here is an example program to calculate difference in seconds between two times (please note line 8 to set time zone for the `SimpleDateFormat` object):

```
import java.text.SimpleDateFormat;

import java.text.ParseException;

import java.util.Date;

import java.util.TimeZone;

public class Main{

  public static void main(String[] args) throws ParseException {

    SimpleDateFormat sf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
```

```
    sf.setTimeZone(TimeZone.getTimeZone("Asia/Singapore"));
    String time1 = "1981-12-31 23:59:59";
    String time2 = "1982-01-01 00:30:00";
    Date date1 = sf.parse(time1);
    Date date2 = sf.parse(time2);
    long second1 = date1.getTime() /1000;
    long second2 = date2.getTime() /1000;
    System.out.println(second2 - second1);
  }
}
```

You may think the program should output 30 * 60 + 1 = 1801 at first glance since it looks like these two times have 30 minutes and 1 second in difference. However, the program outputs 1 instead of 1801. Why? Referring to http://www.timeanddate.com/time/change/singapore/singapore?year=1982, we know that Singapore changed its time zone from UTC+7:30 to UTC+8 on 1 January 1982, and clocks were turned forward 30 minutes to Friday, 1 January 1982, 00:30:00 when local time was about to read 1 January 1982, 00:00:00. Thus, 31 December 1981, 23:59:59 and 1 Janurary 1982, 00:30:00 have only 1 second in difference!

Thus, as a best practice, we should always store timestamp instead of string with certain date format when persistence is required, and display it out according to required time zone. Always be careful when you parse a date string.

~~~ End of Article ~~~

Session 14: Annotations and Base64 Encoding

Creating Custom Annotations and Using Them

Source	http://isagoksu.com/2009/creating-custom-annotations-and-making-use-of-them/
Date of Retrieval	30/09/2016

How to Create Custom Annotations?

There are a lot of documentation about this part in the Internet. All you have to do is basically creating an annotation class like below:

```
public @interface Copyright {
   String info() default "";
}
```

And that's it. Now it's ready to use! Now you can put copyright information to your classes :) Since we didn't define any @Target , you can use this annotation anywhere in your classes by default. If you want your annotation to be only available for class-wise or method-wise, you should define @Target annotation. Keep in mind, there are plenty ofElementType options are available in the SDK.

If you want your annotation to be available in more than one place, just use array syntax as in:

```
@Target({ ElementType.PARAMETER, ElementType.LOCAL_VARIABLE })
```

One thing you may already notice is annotations are interfaces, so you don't implement anything in them.

How to Make Use of Your Custom Annotations?

Up to here, you can find lots of examples. Okaaay, now let's do something useful :) For instance, let's re-implement JUnit's @Test annotation. As you guys already know, @Test annotation is a marker annotation. Basically it marks the method as test method. If you're expecting any exceptions, you would set expect attribute in the annotation. You can try anything here, I'm just using this example since everyone knows how @Test annotation works.

First let's define our annotation:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {
   Class expected();
}
```

You might notice that I used @Retention. This annotation marks our annotation to be retained by JVM at runtime. This will allow us to use Java reflections later on.

Now we need to write our annotation parser class. This class will parse our annotation and trigger some other invocations related to what we want. Keep in mind that if you have more than one custom annotation, then it's also wise to have separate parsers for each annotation you define. So I'll create one for this! The basic idea behind the annotation parser is using Java reflections to access the annotation information/attributes etc. So here is an example parser for our @Test annotation:

```
public class TestAnnotationParser {
   public void parse(Class<?> clazz) throws Exception {
      Method[] methods = clazz.getMethods();
      int pass = 0;
      int fail = 0;
```

```
      for (Method method : methods) {
         if (method.isAnnotationPresent(Test.class)) {
            try {
               method.invoke(null);
               pass++;
            } catch (Exception e) {
               fail++;
            }
         }
      }
   }
}
```

That's all you need. You parser is ready to use too. But wait a minute we didn't implement anything about the annotation attributes. This part is a bit tricky. Because you cannot directly access those attributes from the object graph. Luckily invocation helps us here. You can only access these attributes by invoking them. Sometimes you might need to cast the class to the annotation type too. I'm sure you'll figure out when you see it:) Anyways here is a bit more logic to take our expected attribute into account:

```
// ...
// this is how you access to the attributes
Test test = method.getAnnotation(Test.class);
// we use Class type here because our attribute type
// is class. If it would be string, you'd use string
Class expected = test.expected();
try {
   method.invoke(null);
   pass++;
} catch (Exception e) {
   if (Exception.class != expected) {
       fail++;
   } else {
       pass++;
   }
}
// ...
```

Now everything is ready to use. Below example demonstrates how you use Parser with your test classes:

```
public class Demo {
   public static void main(String [] args) {
      TestAnnotationParser parser = new TestAnnotationParser();
      parser.parse(MyTest.class);
      // you can use also Class.forName
      // to load from file system directly!
   }
}
```

Yeah, I hope you enjoyed. Don't hesitate to shoot me an email if you've a better approach? Thanks! Here is the full parser class implementation:

```
public class TestAnnotationParser {
   public void parse(Class<?> clazz) throws Exception {
      Method[] methods = clazz.getMethods();
```

```
      int pass = 0;
      int fail = 0;
      for (Method method : methods) {
         if (method.isAnnotationPresent(Test.class)) {
            // this is how you access to the attributes
            Test test = method.getAnnotation(Test.class);
            Class expected = test.expected();
            try {
               method.invoke(null);
               pass++;
            } catch (Exception e) {
               if (Exception.class != expected) {
                  fail++;
               } else {
                  pass++;
               }
            }
         }
      }
   }
}
```

Edit: Also after receiving some emails, I guess I should add a full working example :) So here is one. Just copy paste and run the show :)

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@interface Test {
   String info() default "";
}
class Annotated {
   @Test(info = "AWESOME")
   public void foo(String myParam) {
      System.out.println("This is " + myParam);
   }
}
class TestAnnotationParser {
   public void parse(Class> clazz) throws Exception {
      Method[] methods = clazz.getMethods();
      for (Method method : methods) {
         if (method.isAnnotationPresent(Test.class)) {
            Test test = method.getAnnotation(Test.class);
            String info = test.info();
            if ("AWESOME".equals(info)) {
                System.out.println("info is awesome!");
                // try to invoke the method with param
                method.invoke(
                   Annotated.class.newInstance(),
                   info
                );
            }
```

```
            }
        }
    }
}
public class Demo {
    public static void main(String[] args) throws Exception {
        TestAnnotationParser parser = new TestAnnotationParser();
        parser.parse(Annotated.class);
    }
}
```

~~~ End of Article ~~~

Session 15: Functional Programming in Java

Lambda Expressions and Functional Interfaces: Tips and Best Practices

Source	http://www.baeldung.com/java-8-lambda-expressions-tips
Date of Retrieval	30/09/2016

Overview

Now that Java 8 has reached wide usage, patterns, and best practices have begun to emerge for some of its headlining features. In this tutorial, we will take a closer look to functional interfaces and lambda expressions.

2. Prefer Standard Functional Interfaces

Functional interfaces, which are gathered in the **java.util.function** package, satisfy most developers' needs in providing target types for lambda expressions and method references. Each of these interfaces is general and abstract, making them easy to adapt to almost any lambda expression. Developers should explore this package before creating new functional interfaces.

Consider an interface *Foo*:

```
1    @FunctionalInterface
2    public interface Foo {
3        String method(String string);
4    }
```

and a method *add()* in some class *UseFoo*, which takes this interface as a parameter:

```
1    public String add(String string, Foo foo) {
2        return foo.method(string);
3    }
```

To execute it, you would write:

```
1    Foo foo = parameter -> parameter + " from lambda";
2    String result = useFoo.add("Message ", foo);
```

Look closer and you will see that *Foo* is nothing more than a function that accepts one argument and produces a result. Java 8 already provides such an interface in *Function<T,R>* from the java.util.function package.

Now we can remove interface *Foo* completely and change our code to:

```
1    public String add(String string, Function<String, String> fn) {
2        return fn.apply(string);
3    }
```

To execute this, we can write:

```
1    Function<String, String> fn =
2      parameter -> parameter + " from lambda";
3    String result = useFoo.add("Message ", fn);
```

3. Use the *@FunctionalInterface* Annotation

Annotate your functional interfaces with *@FunctionalInterface*. At first, this annotation seems to be

useless. Even without it, your interface will be treated as functional as long as it has just one abstract method.

But imagine a big project with several interfaces – it's hard to control everything manually. An interface, which was designed to be functional, could accidentally be changed by adding of other abstract method/methods, rendering it unusable as a functional interface.

But using the *@FunctionalInterface* annotation, the compiler will trigger an error in response to any attempt to break the predefined structure of a functional interface. It is also a very handy tool to make your application architecture easier to understand for other developers.

So, use this:

```
1   @FunctionalInterface
2   public interface Foo {
3       String method();
4   }
```

instead of just:

```
1   public interface Foo {
2       String method();
3   }
```

4. Don't Overuse Default Methods in Functional Interfaces

You can easily add default methods to the functional interface. This is acceptable to the functional interface contract as long as there is only one abstract method declaration:

```
1   @FunctionalInterface
2   public interface Foo {
3       String method();
4       default void defaultMethod() {}
5   }
```

Functional interfaces can be extended by other functional interfaces if their abstract methods have the same signature. For example:

```
1    @FunctionalInterface
2    public interface FooExtended extends Baz, Bar {}
3
4    @FunctionalInterface
5    public interface Baz {
6        String method();
7        default void defaultBaz() {}
8    }
9
10   @FunctionalInterface
11   public interface Bar {
12       String method();
13       default void defaultBar() {}
14   }
```

Just as with regular interfaces, extending different functional interfaces with the same default method can be problematic. For example, assume that interfaces*Bar* and *Baz* both have a default method *defaultCommon().* In this case, you will get a compile-time error:

```
1   interface Foo inherits unrelated defaults for defaultCommon() from types Baz and Bar...
```

To fix this, *defaultCommon()* method should be overridden in the *Foo* interface. You can, of course, provide a custom implementation of this method. But if you want to use one of the parent interfaces' implementations (for example, from the *Baz*interface), add following line of code to the *defaultCommon()* method's body:

```
1   Baz.super.defaultCommon();
```

But be careful. **Adding too many default methods to the interface is not a very good architectural decision.** It is should be viewed as a compromise, only to be used when required, for upgrading existing interfaces without breaking backward compatibility.

5. Instantiate Functional Interfaces with Lambda Expressions

The compiler will allow you to use an inner class to instantiate a functional interface. However, this can lead to very verbose code. You should prefer lambda expressions:

```
Foo foo = parameter -> parameter + " from Foo";
```

over an inner class:

```
Foo fooByIC = new Foo() {
    @Override
    public String method(String string) {
        return string + " from Foo";
    }
};
```

The lambda expression approach can be used for any suitable interface from old libraries. It is usable for interfaces like *Runnable*, *Comparator*, and so on.**However, this doesn't mean that you should review your whole older codebase and change everything.**

6. Avoid Overloading Methods with Functional Interfaces as Parameters

Use methods with different names to avoid collisions; let's look at an example:

```
public interface Adder {
    String add(Function<String, String> f);
    void add(Consumer<Integer> f);
}

public class AdderImpl implements Adder {

    @Override
    public  String add(Function<String, String> f) {
        return f.apply("Something ");
    }

    @Override
    public void add(Consumer<Integer> f) {}
}
```

At first glance, this seems reasonable. But any attempt to execute any of*AdderImpl's* methods:

```
String r = adderImpl.add(a -> a + " from lambda");
```

ends with an error with the following message:

```
reference to add is ambiguous both method
add(java.util.function.Function<java.lang.String,java.lang.String>)
in fiandlambdas.AdderImpl and method
add(java.util.function.Consumer<java.lang.Integer>)
in fiandlambdas.AdderImpl match
```

To solve this problem, you have two options. The **first** is to use methods with different names:

```
String addWithFunction(Function<String, String> f);

void addWithConsumer(Consumer<Integer> f);
```

The **second** is to perform casting manually. This is not preferred.

```
String r = Adder.add((Function) a -> a + " from lambda");
```

7. Don't Treat Lambda Expressions as Inner Classes

Despite our previous example, where we essentially substituted inner class by a lambda expression, the two concepts are different in an important way: scope.

When you use an inner class, it creates a new scope. You can overwrite local variables from the enclosing scope by instantiating new local variables with the same names. You can also use the keyword ***this*** inside your inner class as a reference to its instance.

However, lambda expressions work with enclosing scope. You can't overwrite variables from the enclosing scope inside the lambda's body. In this case, the keyword ***this*** is a reference to an enclosing instance.

For example, in the class *UseFoo* you have an instance variable *value:*

```
1   private String value = "Enclosing scope value";
```

Then in some method of this class place the following code and execute this method.

```
1    public String scopeExperiment() {
2        Foo fooIC = new Foo() {
3            String value = "Inner class value";
4
5            @Override
6            public String method(String string) {
7                return this.value;
8            }
9        };
10       String resultIC = fooIC.method("");
11
12       Foo fooLambda = parameter -> {
13           String value = "Lambda value";
14           return this.value;
15       };
16       String resultLambda = fooLambda.method("");
17
18       return "Results: resultIC = " + resultIC +
19         ", resultLambda = " + resultLambda;
20   }
```

If you execute the *scopeExperiment()* method, you will get the following result: *Results: resultIC = Inner class value, resultLambda = Enclosing scope value*

As you can see, by calling *this.value* in IC, you can access a local variable from its instance. But in the case of the lambda, *this.value* call gives you access to the variable *value* which is defined in the *UseFoo* class, but not to the variable*value* defined inside the lambda's body.

8. Keep Lambda Expressions Short And Self-explanatory

If possible, use one line constructions instead of a large block of code. Remember**lambdas should be an expression, not a narrative.** Despite its concise syntax, **lambdas should precisely express the functionality they provide.**

This is mainly stylistic advice, as performance will not change drastically. In general, however, it is much easier to understand and to work with such code.

This can be achieved in many ways – let's have a closer look.

8.1. Avoid Blocks of Code in Lambda's Body

In an ideal situation, lambdas should be written in one line of code. With this approach, the lambda is a self-explanatory construction, which declares what action should be executed with what data (in the case of lambdas with parameters).

If you have a large block of code, the lambda's functionality is not immediately clear.

With this in mind, do the following:

```
1   Foo foo = parameter -> buildString(parameter);
1   private String buildString(String parameter) {
2       String result = "Something " + parameter;
3       //many lines of code
4       return result;
5   }
```

instead of:

```
1   Foo foo = parameter -> { String result = "Something " + parameter;
2       //many lines of code
3       return result;
4   };
```

However, please don't use this "one-line lambda" rule as dogma. If you have two or three lines in lambda's definition, it may not be valuable to extract that code into another method.

8.2. Avoid Specifying Parameter Types

A compiler in most cases is able to resolve the type of lambda parameters with the help of **type inference**. Therefore, adding a type to the parameters is optional and can be omitted.

Do this:

```
1   (a, b) -> a.toLowerCase() + b.toLowerCase();
```

instead of this:

```
1   (String a, String b) -> a.toLowerCase() + b.toLowerCase();
```

8.3. Avoid Parentheses Around a Single Parameter

Lambda syntax requires parentheses only around more than one parameter or when there is no parameter at all. That is why it is safe to make your code a little bit shorter and to exclude parentheses when there is only one parameter.

So, do this:

```
1   a -> a.toLowerCase();
```

instead of this:

```
1   (a) -> a.toLowerCase();
```

8.4. Avoid Return Statement and Braces

Braces and ***return*** statements are optional in one-line lambda bodies. This means, that they can be omitted for clarity and conciseness.

Do this:

```
1   a -> a.toLowerCase();
```

instead of this:

```
1   a -> {return a.toLowerCase()};
```

8.5. Use Method References

Very often, even in our previous examples, lambda expressions just call methods which are already implemented elsewhere. In this situation, it is very useful to use another Java 8 feature: **method references**.

So, the lambda expression:

```
1   a -> a.toLowerCase();
```

could be substituted by:

```
1   String::toLowerCase;
```

This is not always shorter, but it makes the code more readable.

9. Use "Effectively Final" Variables

Accessing a non-final variable inside lambda expressions will cause the compile-time error. **But it doesn't mean that you should mark every target variable as*final.***

According to the "**effectively final**" concept, a compiler treats every variable as*final,* as long as it is assigned only once.

It is safe to use such variables inside lambdas because the compiler will control their state and trigger a compile-time error immediately after any attempt to change them.

For example, the following code will not compile:

```
1   public void method() {
2       String localVariable = "Local";
3       Foo foo = parameter -> {
4           String localVariable = parameter;
5           return localVariable;
6       };
7   }
```

The compiler will inform you that:

```
1   Variable 'localVariable' is already defined in the scope.
```

This approach should simplify the process of making lambda execution thread-safe.

10. Protect Object Variables from Mutation

One of the main purposes of lambdas is use in parallel computing – which means that they're really helpful when it comes to thread-safety.

The "effectively final" paradigm helps a lot here, but not in every case. Lambdas can't change a value of an object from enclosing scope. But in the case of mutable object variables, a state could be changed inside lambda expressions.

Consider the following code:

```
1   int[] total = new int[1];
2   Runnable r = () -> total[0]++;
3   r.run();
```

This code is legal, as *total* variable remains "effectively final". But will the object it references to have the same state after execution of the lambda? No!

Keep this example as a reminder to avoid code that can cause unexpected mutations.

11. Conclusion

In this tutorial, we saw some best practices and pitfalls in Java 8's lambda expressions and functional interfaces. Despite the utility and power of these new features, they are just tools. Every developer should pay attention while using them.

The complete **source code** for the example is available in this GitHub project – this is a Maven and Eclipse project, so it can be imported and used as-is.

~~~ End of Article ~~~

Session 16: Stream API

Grouping

Source	http://www.leveluplunch.com/java/examples/java-util-stream-groupingBy-example/
Date of Retrieval	30/09/2016

A common operation that you have become familiar with in SQL is the GROUP BY statement which is used in conjunction with the aggregate functions such as count. It might look something like this:

```
SELECT column_name, count(column_name)

FROM table

GROUP BY column_name;
```

In java 8 the idea of grouping objects in a collection based on the values of one or more of their properties is simplified by using a Collector. A collector is another new interface introduced in Java 8 for defining how to perform a reduction operation on a stream and with the functional nature allows you to achieve a grouping with a single statement. It might even spark debates with your DBA for control or their lack of understanding. Should it happen in code or in the database? In the example below, we will perform a number of group by statements to show the simplicity and flexibility of the interface.

```
class StudentClass {


    private String teacher;

    private double level;

    private String className;


    public StudentClass(String teacher, double level, String className) {

        super();

        this.teacher = teacher;

        this.level = level;

        this.className = className;

    }


    //...

}


private List<StudentClass> studentClasses;
```

```
@Before
public void setUp() {

    studentClasses = new ArrayList<>();

    studentClasses.add(new StudentClass("Kumar", 101, "Intro to Web"));
    studentClasses.add(new StudentClass("White", 102, "Advanced Java"));
    studentClasses.add(new StudentClass("Kumar", 101, "Intro to Cobol"));
    studentClasses.add(new StudentClass("Sargent", 101, "Intro to Web"));
    studentClasses.add(new StudentClass("Sargent", 102, "Advanced Web"));

}
```

Group by teacher name

This example will show how to "group classes by a teacher's name". Here you will pass the `groupingBy` method a function in the form of a method reference extracting each teacher name to the corresponding `StudentClass` which will return 1 key to many elements. This is similar to guava's Multimap collection which allows for easy mapping of a single key to multiple values.

```
@Test
public void group_by_teacher_name() {

    Map<String, List<StudentClass>> groupByTeachers = studentClasses
            .stream().collect(
                    Collectors.groupingBy(StudentClass::getTeacher));
    logger.info(groupByTeachers);
    assertEquals(1, groupByTeachers.get("White").size());
}
```

Output:

```
{
White=[StudentClass {teacher=White, level=102.0, className=Advanced Java}],
Kumar=[StudentClass{teacher=Kumar, level=101.0, className=Intro to Web},
StudentClass{teacher=Kumar, level=101.0, className=Intro to Cobol}],
Sargent=[StudentClass{teacher=Sargent, level=101.0, className=Intro to
Web}, StudentClass{teacher=Sargent, level=102.0, className=Advanced Web}]
}
```

Group by class level

Similar to above, this example will show "show all classes by level".

```
@Test
public void group_by_level() {

    Map<Double, List<StudentClass>> groupByLevel = studentClasses.stream()
            .collect(Collectors.groupingBy(StudentClass::getLevel));

    logger.info(groupByLevel);

    assertEquals(3, groupByLevel.get(101.0).size());
}
```

Output:

```
102.0=[StudentClass{teacher=White, level=102.0, className=Advanced Java},
StudentClass{teacher=Sargent, level=102.0, className=AdvancedWeb}],
101.0=[StudentClass{teacher=Kumar, level=101.0,className=Intro to Web},
StudentClass{teacher=Kumar, level=101.0,className=Intro to Cobol},
StudentClass{teacher=Sargent, level=101.0, className=Intro to Web}]
```

groupBy aggregate

This example will "count the number of classes per level". In sql, it might look like this:

```
SELECT CLASS_LEVEL, count(CLASS_LEVEL)
FROM STUDENT_CLASS
GROUP BY CLASS_LEVEL;
```

In an overloaded `groupingBy` method, you can pass a second collector. Collectors have various reduce operations which can be passed, in this case `Collectors.counting` which will count the number of classes in each level.

```
@Test
public void group_by_count() {

    Map<Double, Long> groupByLevel = studentClasses.stream().collect(
            Collectors.groupingBy(StudentClass::getLevel,
                    Collectors.counting()));

    logger.info(groupByLevel);
```

```
    assertEquals(2.0, groupByLevel.get(102.0), 0);
}
```

Output:

<{102.0=2, 101.0=3}>

~~~ End of Article ~~~

Java 8 Map

Source	http://www.leveluplunch.com/java/tutorials/042-java8-stream-map/
Date of Retrieval	30/09/2016

The map method is a transform operation and is a functional programming technique that applies a function to each one of streams elements returning the results. A common use case map is utilized for is transforming object in java.

Starting simple

For our first snippet, lets multiply each element in the list by 10 by passing in a lambda function to the Stream.map() that accepts a parameter then multiples it by 10. Next, to display the results by using java8 print stream.

```
@Test
public void map_times_10() {

    List<Integer> myList = Arrays.asList(1, 2, 3, 4, 5);

    Stream<Integer> multipleOf10 = myList.stream().map((x) -> x * 10);

    multipleOf10.forEach(System.out::println);
}
```

Output:

```
10
20
30
40
50
```

To be more usable, we can refactoring our code by moving the lambda expression and declare a function. In addition, if we were returning a list as a data type, we can convert the stream to list and then again, output the values in the list where we should see the same result as above.

```
01 @Test
02 public void map_times_10_refactor() {
03  List<Integer> myList = Arrays.asList(1, 2, 3, 4, 5);
04 Function<Integer, Integer> times10 = (x) -> x * 10;
```

```
   List<Integer> multipleOf10 = myList.stream().map(times10)
05
   .collect(Collectors.toList());

   multipleOf10.forEach(System.out::println); }
06
```

Output:

```
10
20
30
40
50
```

Squaring values

Similar in fashion, lets create a function that will square all the values within a stream. We will create a java 8 function that accepts one parameter and then multiplies it by itself. Next we will `Stream.collect()`, a reduction operation, the result of the map and convert the stream to a set.

```
@Test
public void map_square_to_set() {

   List<Integer> myList = Arrays.asList(1, 2, 3, 4, 5);
   Function<Integer, Integer> square = (x) -> x * x;

   Set<Integer> squaresList = myList.stream().map(square)
         .collect(Collectors.toSet());

   squaresList.forEach(System.out::println);
}
```

Output:

```
16
1
4
9
```

25

Map one object to another

If you are looking for a way to convert from object A to object B java 8 makes it easy. Below we will show how to convert a list to map by first creating a POJO named JavaScriptFrameworks. Next initializing a List with three objects we will use a stream() to call the Collectors.toMap. The first parameter will accept a function that will generate the key while second parameter will generate the value. Finally we will loop over the elements using the for each

```
class JavaScriptFrameworks {

    private Integer rank;
    private String description;

    public JavaScriptFrameworks(Integer rank, String description) {
        super();
        this.rank = rank;
        this.description = description;
    }
}

@Test
public void objectAObjectB() {

    List<JavaScriptFrameworks> framework = new ArrayList<JavaScriptFrameworks>();
    framework.add(new JavaScriptFrameworks(1, "Angular"));
    framework.add(new JavaScriptFrameworks(2, "React"));
    framework.add(new JavaScriptFrameworks(3, "EmberJs"));
    Map<Integer, JavaScriptFrameworks> mappedFramework = framework.stream()
            .collect(
                    Collectors.toMap(JavaScriptFrameworks::getRank,
                            (p) -> p));

    mappedFramework.forEach((k, v) -> System.out.println("rank = " + k
            - " value = " + v));
```

```
}
```

Output:

```
rank = 1 value = JavaScriptFrameworks [rank=1, description=Angular]
rank = 2 value = JavaScriptFrameworks [rank=2, description=React]
rank = 3 value = JavaScriptFrameworks [rank=3, description=EmberJs]
```

Java 8 primitive streams

If you are working within your favorite ide, you will notice three other map methods exists. mapToInt, mapToDouble, and mapToLong are methods designed to return primitive data streams. This is helpful if you are looking to perform a reduce operations on a stream.

Building on the squares snippet above passing in square function to map will produce a stream of Integers. Then calling mapToInt will return an IntStream where we can call max to return the maximum value in stream.

```
@Test
public void mapto() {

    List<Integer> myList = Arrays.asList(1, 2, 3, 4, 5);
    Function<Integer, Integer> square = (x) -> x * x;

    IntStream squaresIntStream = myList.stream().map(square)
            .mapToInt(Integer::intValue);

    System.out.println(squaresIntStream.max().getAsInt());;
}
```

Output:

```
25
```

Mapping values using guava

Before java 8, you were left with transforming values in java using a imperative programming technique unless you were using a library such as guava. If you are familiar with the library, the code below will provide an example for reference.

```
@Test
public void map_square_guava() {
```

```
    List<Integer> myList = Arrays.asList(1, 2, 3, 4, 5);
    //com.google.common.base.Function
    Function<Integer, Integer> square = new Function<Integer, Integer>() {
        @Override
        public Integer apply(Integer value) {
            return value * value;
        }
    };
    List<Integer> squaresList = FluentIterable.from(myList)
            .transform(square).toList();
    System.out.println(squaresList);
}
```

Output:

```
[1, 4, 9, 16, 25]
```

Converting from object to object or primitive streams, this tutorial scratched the surface in showing basic ways to use Stream.map function. Thanks for joining in today's levelup, have a great day!

~~~ End of Article ~~~

Session 17: More on Functional Programming

Functional Interface

Source	https://java2practice.com/2014/03/16/java-8-functional-interface-example/
Date of Retrieval	30/09/2016

To Support lambda expressions in Java 8, they introduced Functional Interfaces.

An interface which has Single Abstract Method can be called as Functional Interface.

Runnable, Comparator,Cloneable are some of the examples for Functional Interface. We can implement these Functional Interfaces by using Lambda expression.

For example:

```
Thread t =new Thread(new Runnable(){
   public void run(){
     System.out.println("Runnable implemented by using Lambda Expression");
   }
});
```

This is the old way of creating a Thread.

As Runnable is having Single Abstract Method, we can consider this as a Functional Interface and we can use Lambda expression like below.

```
Thread t = new Thread(()->{
   System.out.println("Runnable implemented by using Lambda Expression");
});
```

Here instead of passing `Runnable` object we just passed lambda expression.

Declaring our own Functional Interfaces:

We can declare our own Functional Interface by defining Single Abstract Method in interface.

```
public interface FunctionalInterfaceTest{
void display();
}
//Test class to implement above interface
public class FunctionInterfaceTestImpl {
      public static void main(String[] args){
     //Old way using anonymous inner class
     FunctionalInterfaceTest fit = new FunctionalInterfaceTest(){
```

```
        public void display(){
           System.out.println("Display from old way");
        }};
     fit.display();//outputs: Display from old way
     //Using lambda expression
     FunctionalInterfaceTest newWay = () -> {System.out.println("Display
from new Lambda Expression");}
        newWay.display();//outputs : Display from new Lambda Expression
     }
}
```

We can annotate with `@FunctionalInterface` annotation, to tell compile time errors. It is optional

for ex:

```
@FunctionalInterface
public interface FunctionalInterfaceTest{
   void display();
   void anotherDisplay();//shows an error, FunctionalInterface should have
only one abstarct method.
}
```

Default Method:

Functional interface cannot have more than one abstract method but it can have more than one default methods.

Default methods are introduced in Java 8, to add new methods to interface without disturbing the implemented classes.

```
interface DefaultInterfaceTest{
  void show();
  default void display(){
     System.out.println("Default method from interface can have body..!");
  }
}
public class DefaultInterfaceTestImpl implements DefaultInterfaceTest{
   public void show(){
         System.out.println("show method");
   }
   //we dont need to provide any implementation to default method.
   public static void main(String[] args){
          DefaultInterfaceTest obj = new DefaultInterfaceTestImpl();
```

```
          obj.show();//out puts: show method
          obj.display();//outputs : Default method from interface can have
body..!
        }
}
```

Main use of default method is without forcing the implemented class , we can add a method to an interface.

Multiple Inheritance:

If the same default method is there in two interfaces and one class is implementing that interface, then it will throw an error.

```
//Normal interface with show method
 interface Test{
   default void show(){
     System.out.println("show from Test");
  }
}
 //Another interface with same show method
 interface AnotherTest{
    default void show(){
      System.out.println("show from Test");
   }
 }
 //Main class to implement above two interfaces
 class Main implements Test, AnotherTest{
//here is an ambiguity which show method has to inherit here
}
```

This class wont compile because there is an ambiguity between `Test`, `AnotherTest` interfaces `show()` method, to resolve this we need to override `show()` method to `Main` Class.

```
class Main implements Test, AnotherTest{
    void show(){
      System.out.println("Main show method");
   }
 }
```

~~~ End of Article ~~~

Session 18: Additional Features of Java 8

Math Utilities

Source	http://docstore.mik.ua/orelly/java/exp/ch07_02.htm
Date of Retrieval	30/09/2016

Java supports integer and floating-point arithmetic directly. Higher-level math operations are supported through the `java.lang.Math` class. Java provides wrapper classes for all primitive data types, so you can treat them as objects if necessary. Java also provides the `java.util.Random` class for generating random numbers.

Java handles errors in integer arithmetic by throwing an `ArithmeticException`:

```
int zero = 0;


try {
    int i = 72 / zero;
}
catch ( ArithmeticException e ) {        // division by zero
}
```

To generate the error in the above example, we created the intermediate variable `zero`. The compiler is somewhat crafty and would have caught us if we had blatantly tried to perform a division by zero.

Floating-point arithmetic expressions, on the other hand, don't throw exceptions. Instead, they take on the special out-of-range values shown inTable 7.3.

Table 7.3: Special Floating-Point Values

Value	Mathematical representation
`POSITIVE_INFINITY`	1.0/0.0
`NEGATIVE_INFINITY`	-1.0/0.0
`NaN`	0.0/0.0

The following example generates an infinite result:

```
double zero = 0.0;
double d = 1.0/zero;


if ( d == Double.POSITIVE_INFINITY )
    System.out.println( "Division by zero" );
```

The special value `NaN` indicates the result is "not a number." The value `NaN` has the special distinction of not being equal to itself (`NaN != NaN`). Use`Float.isNaN()` or `Double.isNaN()` to test for `NaN`.

java.lang.Math

The `java.lang.Math` class serves as Java's math library. All its methods are `static` and used directly ; you can't instantiate a Math object. We use this kind of degenerate class when we really want methods to approximate normal functions in C. While this tactic defies the principles of object-

oriented design, it makes sense in this case, as it provides a means of grouping some related utility functions in a single class. Table 7.4 summarizes the methods in `java.lang.Math`.

Table 7.4: Methods in java.lang.Math

Method	Argument type(s)	Functionality
`Math.abs(a)`	`int`, `long`, `float`, `double`	Absolute value
`Math.acos(a)`	`Double`	Arc cosine
`Math.asin(a)`	`Double`	Arc sine
`Math.atan(a)`	`Double`	Arc tangent
`Math.atan2(a,b)`	`Double`	Converts rectangular to polar coordinates
`Math.ceil(a)`	`Double`	Smallest whole number greater than or equal to a
`Math.cos(a)`	`Double`	Cosine
`Math.exp(a)`	`Double`	Exponential number to the power of a
`Math.floor(a)`	`Double`	Largest whole number less than or equal to a
`Math.log(a)`	`Double`	Natural logarithm of a
`Math.max(a, b)`	`int`, `long`, `float`, `double`	Maximum
`Math.min(a, b)`	`int`, `long`, `float`, `double`	Minimum
`Math.pow(a, b)`	`Double`	`a` to the power of `b`
`Math.random()`	None	Random number generator
`Math.rint(a)`	`Double`	Converts double value to integral value in double format
`Math.round(a)`	`float`, `double`	Rounds
`Math.sin(a)`	`Double`	Sine
`Math.sqrt(a)`	`Double`	Square root
`Math.tan(a)`	`Double`	Tangent

`log()`, `pow()`, and `sqrt()` can throw an `ArithmeticException`. `abs()`, `max()`, and `min()` are overloaded for all the scalar values, `int`, `long`, `float`, or`double`, and return the corresponding type. Versions of `Math.round()` accept either `float` or `double` and return `int` or `long` respectively. The rest of the methods operate on and return `double` values:

```
double irrational = Math.sqrt( 2.0 );
int bigger = Math.max( 3, 4 );
long one = Math.round( 1.125798 );
```

For convenience, `Math` also contains the `static final double` values `E` and `PI`:

```
double circumference = diameter * Math.PI;
```

java.math

If a `long` or a `double` just isn't big enough for you, the `java.math` package provides two classes, `BigInteger` and `BigDecimal`, that support arbitrary-precision numbers. These are full-featured classes with a bevy of methods for performing arbitrary-precision math. In the following example, we use `BigInteger` to add two numbers together.

```
try {
    BigDecimal twentyone = new BigDecimal("21");
    BigDecimal seven = new BigDecimal("7");
    BigDecimal sum = twentyone.add(seven);

    int twentyeight = sum.intValue();
}
catch (NumberFormatException nfe) { }
catch (ArithmeticException ae) { }
```

Wrappers for Primitive Types

In languages like Smalltalk, numbers and other simple types are objects, which makes for an elegant language design, but has trade-offs in efficiency and complexity. By contrast, there is a schism in the Java world between class types (i.e., objects) and primitive types (i.e., numbers, characters, and boolean values). Java accepts this trade-off simply for efficiency reasons. When you're crunching numbers you want your computations to be lightweight; having to use objects for primitive types would seriously affect performance. For the times you want to treat values as objects, Java supplies a wrapper class for each of the primitive types, as shown in Table 7.5.

Table 7.5: Primitive Type Wrappers

Primitive	Wrapper
Void	java.lang.Void
boolean	java.lang.Boolean
Char	java.lang.Character
Byte	java.lang.Byte
Short	java.lang.Short
Int	java.lang.Integer
Long	java.lang.Long
Float	java.lang.Float
Double	java.lang.Double

An instance of a wrapper class encapsulates a single value of its corresponding type. It's an immutable object that serves as a container to hold the value and let us retrieve it later. You can construct a wrapper object from a primitive value or from a `String` representation of the value. The following code is equivalent:

```
Float pi = new Float( 3.14 );
Float pi = new Float( "3.14" );
```

Wrapper classes throw a `NumberFormatException` when there is an error in parsing from a string:

```
try {
    Double bogus = new Double( "huh?" );
}
catch ( NumberFormatException e ) {     // bad number
}
```

You should arrange to catch this exception if you want to deal with it. Otherwise, since it's a subclass

of `RuntimeException`, it will propagate up the call stack and eventually cause a run-time error if not caught.

Sometimes you'll use the wrapper classes simply to parse the `String` representation of a number:

```
String sheep = getParameter("sheep");

int n = new Integer( sheep ).intValue();
```

Here we are retrieving the value of the `sheep` parameter. This value is returned as a `String`, so we need to convert it to a numeric value before we can use it. Every wrapper class provides methods to get primitive values out of the wrapper; we are using `intValue()` to retrieve an `int` out of`Integer`. Since parsing a `String` representation of a number is such a common thing to do, the `Integer` and `Long` classes also provide the `static` methods `Integer.parseInt()` and `Long.parseLong()` that read a `String` and return the appropriate type. So the second line above is equivalent to:

```
int n = Integer.parseInt( sheep );
```

All wrappers provide access to their values in various forms. You can retrieve scalar values with the methods `doubleValue()`, `floatValue()`,`longValue()`, and `intValue()`:

```
Double size = new Double ( 32.76 );

double d = size.doubleValue();

float f = size.floatValue();

long l = size.longValue();

int i = size.intValue();
```

The code above is equivalent to the primitive `double` value cast to the various types. For convenience, you can cast between the wrapper classes like `Double` class and the primitive data types.

Another common use of wrappers occurs when we have to treat a primitive value as an object in order to place it in a list or other structure that operates on objects.

As you'll see shortly, a `Vector` is an extensible array of `Objects`. We can use wrappers to hold numbers in a `Vector`, along with other objects:

```
Vector myNumbers = new Vector();


Integer thirtyThree = new Integer( 33 );

myNumbers.addElement( thirtyThree );
```

Here we have created an `Integer` wrapper so that we can insert the number into the `Vector` using `addElement()`. Later, when we are taking elements back out of the `Vector`, we can get the number back out of the `Integer` as follows:

```
Integer theNumber = (Integer)myNumbers.firstElement();

int n = theNumber.intValue();           // n = 33
```

Random Numbers

You can use the `java.util.Random` class to generate random values. It's a pseudo-random number generator that can be initialized with a 48-bit seed.[1] The default constructor uses the current time as a seed, but if you want a repeatable sequence, specify your own seed with:

[1] The generator uses a linear congruential formula. See *The Art of Computer Programming*, Volume 2 "Semi-numerical Algorithms," by Donald Knuth (Addison-Wesley).

```
long seed = mySeed;

Random rnums = new Random( seed );
```

This code creates a random-number generator. Once you have a generator, you can ask for random values of various types using the methods listed in Table 7.6.

Table 7.6: Random Number Methods

Method	Range
`nextInt()`	-2147483648 to 2147483647
`nextLong()`	-9223372036854775808 to 9223372036854775807
`nextFloat()`	-1.0 to 1.0
`nextDouble()`	-1.0 to 1.0

By default, the values are uniformly distributed. You can use the `nextGaussian()` method to create a Gaussian distribution of `double` values, with a mean of 0.0 and a standard deviation of 1.0.

The `static` method `Math.random()` retrieves a random `double` value. This method initializes a `private` random-number generator in the `Math` class, using the default `Random` constructor. So every call to `Math.random()` corresponds to a call to `nextDouble()` on that random number generator.

~~~ End of Article ~~~