References

Table of Contents

Session 1: Introduction to Java

Editor Hints in Netbeans IDE

Source	http://www.netbeans.org/kb/55/using-netbeans/editing.html#44438
Date of Retrieval	22/05/2007

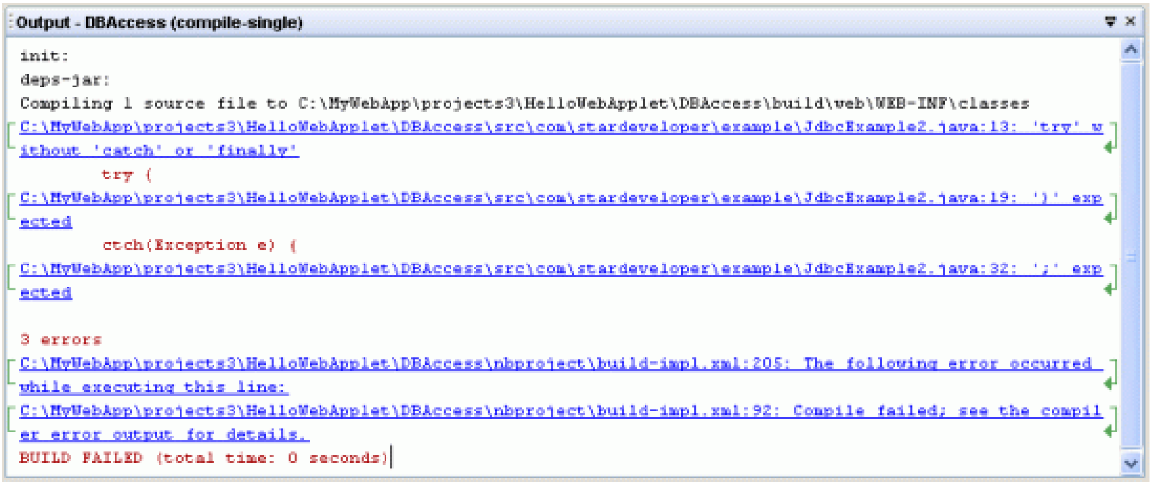
When the IDE detects an error for which it has identified a possible fix, a light bulb icon appears in the left margin of that line. You can click the light bulb or press Alt-Enter to display a list of possible fixes. If one of those fixes suits you, you can select it and press Enter to have the fix generated in your code.
Often, the "error" is not a coding mistake but a reflection of the fact that you have not gotten around to filling in the missing code. In those cases, the editor hints simply automate the entry of certain types of code.

*~~~ End of Article ~~~*

Fixing Compilation Errors in NetBeans

Source	http://www.netbeans.org/kb/55/using-netbeans/building.html#pgfId-1001843
Date of Retrieval	22/05/2007

The IDE displays output messages and any compilation errors in the Output Window. This multi-tabbed window is displayed automatically whenever you generate compilation errors, debug your program, generate Javadoc documentation, and so on. You can also open this window manually by choosing Window > Output (Ctrl-4).
One important function of the Output window is to notify you of errors found while compiling your program. The error message is displayed in blue underlined text and is linked to the line in the source code that caused the error, as illustrated in the image below. The Output window also provides links to errors found when running Ant build scripts. Whenever you click an error link in the Output window, the Source Editor jumps to the line containing the error automatically. You can also use the F12 and Shift-F12 keyboard shortcuts to move to the next and previous error in the file.



```
init:
deps-jar:
Compiling 1 source file to C:\MyWebApp\projects3\HelloWebApplet\DBAccess\build\web\WEB-INF\classes
C:\MyWebApp\projects3\HelloWebApplet\DBAccess\src\com\stardeveloper\example\JdbcExample2.java:13: 'try' w
ithout 'catch' or 'finally'
        try {
C:\MyWebApp\projects3\HelloWebApplet\DBAccess\src\com\stardeveloper\example\JdbcExample2.java:19: ')' exp
ected
        ctch(Exception e) {
C:\MyWebApp\projects3\HelloWebApplet\DBAccess\src\com\stardeveloper\example\JdbcExample2.java:32: ';' exp
ected

3 errors
C:\MyWebApp\projects3\HelloWebApplet\DBAccess\nbproject\build-impl.xml:205: The following error occurred
while executing this line:
C:\MyWebApp\projects3\HelloWebApplet\DBAccess\nbproject\build-impl.xml:92: Compile failed; see the compil
er error output for details.
BUILD FAILED (total time: 0 seconds)
```

Output window showing compilation errors

Every action that is run by an Ant script, such as compiling, running, and debugging files, sends its output to the same Output window tab. If you need to save the messages displayed in the Output window, you can copy and paste it to a separate file. You can also set Ant to print the command output for each new target to a new Output window tab by choosing Tools > Options, clicking the Ant node in the Miscellaneous category, and deselecting the checkbox for the Reuse Output Tabs from Finished Processes property.

*~~~ End of Article ~~~*

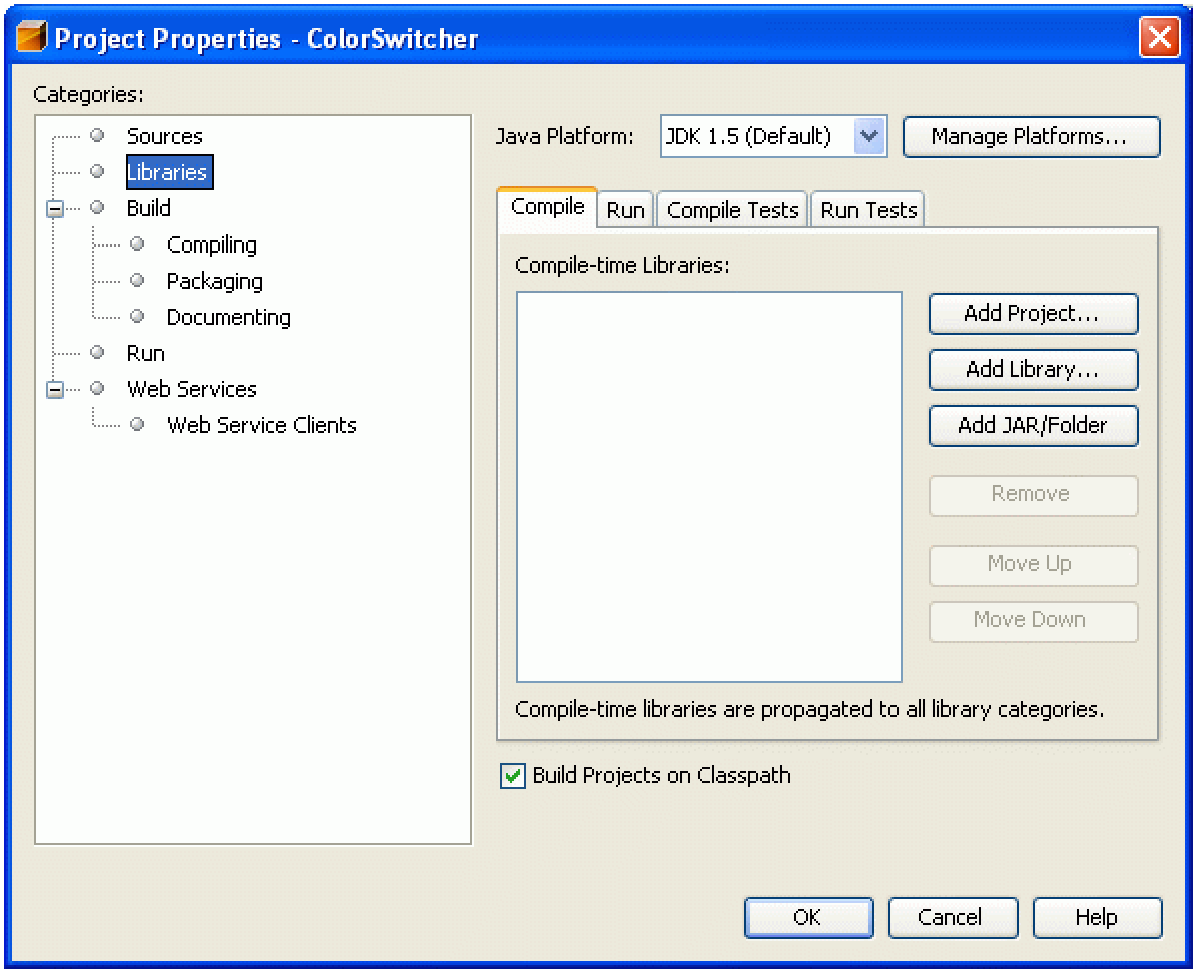
Managing Project's Classpath in NetBeans

Source	http://www.netbeans.org/kb/50/using-netbeans/project_setup.html#96961
Date of Retrieval	22/05/2007

Adding a group of class files to a project's classpath tells the IDE which classes the project should have access to during compilation and execution. The IDE also uses classpath settings to enable code completion, automatic highlighting of compilation errors, and refactoring. You can edit the classpath declarations for an existing project in the Project Properties dialog box.

- **Standard projects.** In standard projects, the IDE maintains separate classpaths for compiling and running your project, as well as compiling and running JUnit tests (for J2SE applications). The IDE automatically adds everything on your project's compilation classpath to the project's runtime classpath. You can add JAR files, libraries, and dependent projects to the project's compilation classpath in the Compile tab of the Project Properties dialog box. You can also right-click the Libraries node in the Projects window and add JAR files, libraries and projects to your project.
- **Free-form projects.** In free-form projects, your Ant script handles the classpath for all of your source folders. You declare the classpath in the New Project wizard when you set up your free-form project. The classpath settings for free-form projects only tell the IDE what classes to make available for code completion and refactoring. You can declare the classpath for free-form projects using the Java Sources Classpath panel in the Project Properties dialog box. For more, see Managing the Classpath in Free-form Projects below.

If you have attached Javadoc and source files to a JAR file in the Library Manager, the IDE automatically adds the Javadoc and source files to the project when you register the JAR file on a project's classpath. You can step into classes and look up Javadoc pages for the classes without configuring anything else.



Compile-time libraries are propagated to all library categories.

[✓] Build Projects on Classpath

Project Properties dialog box

*~~~ End of Article ~~~*

Setting the Target JDK in a Project in NetBeans

Source	http://www.netbeans.org/kb/55/using-netbeans/project_setup.html#51535
Date of Retrieval	22/05/2007

By default, the IDE uses the version of the J2SE platform (JDK) with which the IDE runs as the default Java platform for compilation, execution, and debugging. You can view your IDE's JDK version by choosing Help > About and clicking the Detail tab. The JDK version is listed in the Java field.

You can run the IDE with a different JDK version by starting the IDE with the --jdkhome jdk-home-dir switch from the command line or in your IDE-HOME/etc/netbeans.conf file. For more information, see Configuring IDE Startup Switches.

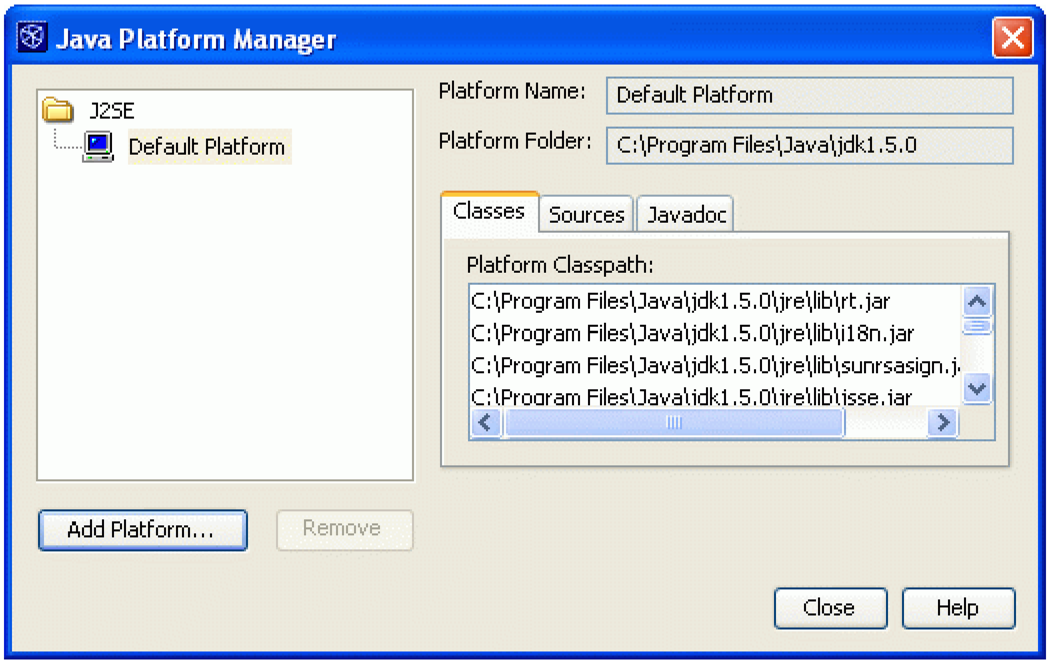
In the IDE, you can register multiple Java platforms and attach Javadoc and source code to each platform. Switching the target JDK for a standard project does the following:

- Offers the new target JDK's classes for code completion.
- If available, displays the target JDK's source code and Javadoc documentation.
- Uses the target JDK's executables (javac and java) to compile and execute your application.
- Compiles your source code against the target JDK's libraries.

You can switch the target JDK of your project by doing the following:

- **Standard projects.** In standard projects you switch the target JDK in the Libraries panel of the Project Properties dialog box.
- **Free-form projects.** In free-form projects you have to set the target JDK in the Ant script itself, then specify the source level in the Sources page of the Project Properties dialog box. You set the source level in the Project Properties dialog box because this is what the IDE uses to determine the JDK to use for your Javadoc and sources for your project. If the IDE cannot find a JDK corresponding to the source level specified, the IDE's default JDK is used.

To register a new Java platform, choose Tools > Java Platform Manager from the main menu. Specify the directory that contains the Java platform as well as the sources and Javadoc needed for debugging.



Platform Name: Default Platform

Platform Folder: C:\Program Files\Java\jdk1.5.0

Platform Classpath:

```
C:\Program Files\Java\jdk1.5.0\jre\lib\rt.jar
C:\Program Files\Java\jdk1.5.0\jre\lib\i18n.jar
C:\Program Files\Java\jdk1.5.0\jre\lib\sunrsasign.j
C:\Program Files\Java\jdk1.5.0\jre\lib\jsse.jar
```

Java Platform Manager

*~~~ End of Article ~~~*

Session 2: Variables and Operators

FORMATTING OUTPUT WITH THE NEW FORMATTER

Source	http://java.sun.com/developer/JDCTechTips/2004/tt1005.html#1
Date of Retrieval	22/05/2007

J2SE 5.0 introduces a new way to format output that is similar to that of the C language's printf. In this approach, each argument to be formatted is described using a string that begins with % and ends with the formatted object's type. This tip introduces you to the new Formatter class and to the syntax of the formatting that you perform with the class.

It's likely that you will most often use the new formatting approach in a call similar to either of the following:

```
System.out.format("Pi is approximately  %f", Math.Pi);
System.out.printf("Pi is approximately  %f", Math.Pi);
```

The printf() and the format() methods perform the same function. System.out is an instance of java.io.PrintStream. PrintStream, java.io.PrintWriter, and java.lang.String each have four new public methods:

```
format( String format, Object... args);
printf( String format, Object... args);
format( Locale locale, String format, Object... args);
printf( Locale locale, String format, Object... args);
```

These correspond to the format() method in the underlying worker class java.util.Formatter class:

```
format(String format, Object... args)
format(Locale l, String format, Object... args)
```

Although you'll likely use these methods from String, PrintStream, and PrintWriter, you'll find the documentation for the various available formatting options in the documentation for the Formatter class.

Let's begin with the following example of the format() method:

```
formatter.format("Pi is approximately %1$f," +"and e is about %2$f", Math.PI,
Math.E);
```

The format() method requires some of the new language features introduced in J2SE 5.0. One is varargs, which simplifies the way that an arbitray number of arguments can be passed to a method. Notice that a variable number of Object instances can be entered for formatting. Also notice that the objects in the example are autoboxed and unboxed. Autoboxing/Unboxing is also a new feature in J2SE 5.0. Autoboxing eliminates the need for manually converting a primitive type (such as int) to a wrapper class (such as Integer), unboxing automates the reverse process. In the example, Math.PI is a double which is autoboxed to a Double (to be treated as an Object). In addition, in the example the formatted output is written to java.lang.StringBuilder, yet another new feature introduced in J2SE 5.0.

The format itself is a String that includes zero or more formatting elements, each beginning with a %. Each formatting element is applied to one of the Objects passed in. Each formatting element has the general form:

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

The argument index is a positive integer that indicates the position of the argument in the argument list. The numbering begins with 1 for the first position, not with 0. So the first position in the previous code snippet is occupied by Math.PI, and is indicated by using 1$. The second position is occupied by Math.E, and is indicated by using 2$.

The width specifies the minimum number of characters to be written as output.

The precision is used to restrict the number of non-zero characters.

The conversion describes the type of the object being formatted. Much of this should be familiar to C programmers because this is a Java implementation of printf(). Common types include f for float, t for time, d for decimal, o for octal, x for hexadecimal, s for general, and c for a Unicode character. The following sample application, UsingFormatter, allows you to enter different formats from the command line and view the output. Notice that the application instantiates a destination -- in this example, a StringBuilder. It then instantiates a Formatter and associates it with the destination. The Formatter then formats some String and sends the output to the destination. The results of the conversion are then displayed to standard out.

```
package format;

import java.util.Formatter;

public class UsingFormatter {

  public static void main(String[] args) {
    if (args.length != 1) {
      System.err.println("usage: " +
        "java format/UsingFormatter ");
      System.exit(0);
    }
```

```
        String format = args[0];

        StringBuilder stringBuilder = new StringBuilder();
        Formatter formatter = new Formatter(stringBuilder);
        formatter.format("Pi is approximately " + format +
          ", and e is about " + format, Math.PI, Math.E);
        System.out.println(stringBuilder);
      }
    }
```

Compile and run this with the command line argument %f:

```
java format/UsingFormatter %f
```

You should get the result:

```
Pi is approximately 3.141593, and e is about 2.718282
```

Rerun this and set the precision to be two decimal places:

```
java format/UsingFormatter %.2f
```

You should see the following:

```
Pi is approximately 3.14, and e is about 2.72
```

Notice that the numbers are not just truncated. The value of e is rounded off to two decimal places. You can additionally specify the width by supplying the command line argument %6.2f. This time leading spaces are inserted because you specified that the number should use six characters, even though the precision restricts it to using only three characters and a decimal place. If you enter the command:

```
java format/UsingFormatter %6.2f
```

You should see this:

```
Pi is approximately   3.14, and e is about   2.72
```

The position can be used to specify which argument to format. Rerun UsingFormatter with the command line argument %1$.2f. This specifies that you want to use Math.PI twice. You should see the following output:

Pi is approximately 3.14, and e is about 3.14

You can change the Locale used to format the numbers by adding an import statement and calling a different constructor. Here's an example:

```
package format;

import java.util.Formatter;
import java.util.Locale;

public class UsingFormatter {

  public static void main(String[] args) {
    if (args.length != 1) {
      System.err.println("usage: " +
        "java format/UsingFormatter <format string>");
      System.exit(0);
    }
    String format = args[0];

    StringBuilder stringBuilder = new StringBuilder();
    Formatter formatter = new Formatter(stringBuilder,
                                Locale.FRANCE);
    formatter.format("Pi is approximately " + format +
      ", and e is about " + format, Math.PI, Math.E);
    System.out.println(stringBuilder);
  }
}
```

Compile and run this with the argument %.2f and you should see the decimal points changed to commas.

```
Pi is approximately 3,14, and e is about 2,72
```

As previously mentioned, you will typically not use the Formatter class explicitly. Instead, for example, you can directly use the printf() and format() methods in the PrintStream class. The following program, UsingSystemOut, is a rewritten version of the UsingFormatter program to use standard out:

```
package format;

public class UsingSystemOut {

  public static void main(String[] args) {
    if (args.length != 1) {
      System.err.println("usage: " +
        "java format/UsingSystemOut <format string>");
      System.exit(0);
    }
    String format = args[0];

    System.out.format("Pi is approximately " + format +
      ", and e is approximately " + format, Math.PI, Math.E);
  }
}
```

The behavior of UsingSystemOut is slightly different than that of UsingFormatter. The UsingFormatter program uses println(), the UsingSystemOut program does not. Because of that, if you run UsingSystemOut from the command line, you will notice that your next command prompt is on the same line as your output. You need to insert a new line. You can do this using formatted output by adding %n. Run UsingSystemOut with the command line argument %.2f%n:

```
java format/UsingSystemOut %.2f%n
```

You will see the following result:

```
Pi is approximately 3.14 , and e is about 2.72
```

You can replace the last method call with printf() if you prefer. There is no difference between System.out.format() and System.out.printf().

As a final example, let's take a look at how date and time objects can be formatted. These objects have the conversion type t or T. That letter is followed by a second letter that indicates which part of the time should be displayed and how it should be displayed. For example, you can display the hour in a variety of forms using tH, tI, tk, or tl, and the minute within the hour using tM. These can also be combined with tr, which displays tH:tM. Similarly, you can display the day of the week, the name of the month, and so on, using the format conversion keys detailed in the Formatter API.

Here is an example that displays the date by formatting the current time using tr for the hour and minute, tA for the day of the week, tB for the name of the month, te for the number of the day of the month, and tY for the year. These could all be preceded by 1$ to point to the first position. Instead, the %tr points to the first position. The < in the other format strings refers back to the position formatted previously.

```
package format;

import java.util.Calendar;

public class FormattingDates  {

  public static void main(String[] args) {
    System.out.printf("Right now it is %tr on " +
                      "%<tA, %<tB %<te, %<tY.%n",
                      Calendar.getInstance());
  }
}
```

Compile and run the FormattingDates program. You will see output that looks something like this:

```
Right now it is 01:55:19 PM on Wednesday, September 22, 2004.
```

This tip is intended to get you started using the new Formatter facility for formatting output. In some ways, the options should feel familiar to you from the old printf() days. Here as before, there are many possibilities available. You will only learn them by trying out different options and deciding which ones meet your needs.

*~~~ End of Article ~~~*

Java Operators: Performing Operations on Primitive Data Types

Source	http://www.informit.com/articles/article.asp?p=30868&rl=1
Date of Retrieval	22/05/2007

Operators work in conjunction with operands, or the literal values or variables involved in the operation. There are unary operators, which are operators that operate on a single operand, as well as operators that operate on two or more variables.

Arithmetic Operators

Arithmetic operators refer to the standard mathematical operators you learned in elementary school: addition, subtraction, multiplication, and division.

Addition

Addition, as you would expect, is accomplished using the plus sign (+) operator. The form of an addition operation is

```
operand + operand
```

For example:

```
// Add two literal values
int result = 5 + 5;

// Add two variables
int a = 5;
int b = 6;
int result = a + b;

// Add two variables and a literal
int result = a + b + 15;
```

An addition operation, can add two or more operands, whereas an operand can be a variable, a literal, or a constant.

Subtraction

Subtraction, again as, you would expect, is accomplished using the minus sign (–) operator. The form of a subtraction operation is

```
operand - operand
```

For example:

```
// Subtract a literal from a literal; the result is 5
int result = 10 - 5;

// Subtract a variable from another variable; the result is -1
```

```
int a = 5;
int b = 6;
int result = a - b;

// Subtract a variable and a literal from a variable
// The result is 5 – 6 – 15 = -1 – 15 = -16
int result = a - b - 15;
```

A subtraction operation can compute the difference between two or more operands, where an operand can be a variable, a literal, or a constant.

Multiplication

Multiplication is accomplished using the asterisk (*) operator. The form of a multiplication operation is

```
operand * operand
```

For example:

```
// Multiply two literal values; result is 25
int result = 5 * 5;

// Multiply two variables; result is 30
int a = 5;
int b = 6;
int result = a * b;

// Multiply two variables and a literal
// The result is 5 * 6 * 15 = 30 * 15 = 450
int result = a * b * 15;
```

A multiplication operation can multiply two or more operands, where an operand can be a variable, a literal, or a constant.

Division

Division is accomplished using the forward slash (/) operator. The form of a division operation is:

```
operand / operand
```

For example:

```
// Divide a literal by a literal; result is 5
int result = 10 / 2;

// Divide a variable by another variable; result is 3
int a = 15;
int b = 5;
int result = a / b;
```

When dividing integer types, the result is an integer type (see the previous chapter for the exact data type conversions for mathematical operations). This means that if you divide an integer unevenly by another integer, it returns the whole number part of the result; it does not perform any rounding. For example, consider the following two operations that both result to 1.

```
int result1 = 10 / 6; // Float value would be 1.6666
int result2 = 10 / 9; // Float value would be 1.1111
```

Both result1 and result2 resolve to be 1, even though result1 would typically resolve to 2 if you were rounding off the result. Therefore, be cognizant of the fact that integer division in Java results in only the whole number part of the result, any fractional part is dropped.

When dividing floating-point variables or values, this caution can be safely ignored. Floating-point division results in the correct result: The fractional part of the answer is represented in the floating-point variable.

```
float f = 10.0f / 6.0f; // result is 1.6666
double d = 10.0 / 9.0; // result is 1.1111
```

Note the appearance of the f following each literal value in the first line. When creating a floating-point literal value (a value that has a fractional element), the default assumption by the compiler is that the values are double. So, to explicitly tell the compiler that the value is a float and not a double, you can suffix the value with either a lowercase or uppercase F.

Modulus

If integer division results in dropping the remainder of the operation, what happens to it? For example if you divide 10 by 6:

```
int i = 10 / 6;
```

The Java result is 1, but the true result is 1 Remainder 4. What happened to the remainder 4?

Java provides a mechanism to get the remainder of a division operation through the modulus operator, denoted by the percent character (%). Although the previous example had a result of 1, the modulus of the operation would give you that missing 4. The form of a modulus operation is

```
operand % operand
```

For example:

```
int i = 10 / 6; // i = 1
int r = 10 % 6; // r = 4
```

Similar to the other arithmetic operators in this chapter, the modulus of an operation can be performed between variables, literals, and constants.

Increment and Decrement Operators

In computer programming it is quite common to want to increase or decrease the value of an integer type by 1. Because of this Java provides the increment and decrement operators that add 1 to a variable and subtract 1 from a variable, respectively. The increment operator is denoted by two plus signs (++), and the decrement operator is denoted by two minus signs (--). The form of the increment and decrement operators is

```
variable++;
++variable;
variable--;
--variable;
```

For example:

```
int i = 10;
i++; // New value of i is 11
```

You might notice that the variable could either be prefixed or suffixed by the increment or decrement operator. If the variable is always modified appropriately (either incremented by 1 or decremented by 1), then what is the difference? The difference has to do with the value of the variable that is returned for use in an operation.

Prefixing a variable with the increment or decrement operator performs the increment or decrement, and then returns the value to be used in an operation.

For example:

```
int i = 10;
int a = ++i; // Value of both i and a is 11
i = 10;
int b = 5 + --i; // Value of b is 14 (5 + 9) and i is 9
```

Suffixing a variable with the increment or decrement operator returns the value to be used in the operation, and then performs the increment or decrement. For example:

```
// Value of i is 11, but the value of a is 10
// Note that the assignment preceded the increment
int i = 10;
int a = i++;

// Value of i is 9 as before, but the value of b is 15 (5 + 10)
i = 10;
int b = 5 + i--;
```

Pay particular attention to your code when you use prefix and postfix versions of these operators and be sure that you completely understand the difference. That difference has led many programmers on a search for unexplained behavior in their testing!

Relational Operators

A very necessary part of computer programming is performing certain actions based off the value of a variable; for example if the nuclear reactor is about to blow up, then shut it down. The next chapter will speak at length about the mechanism for implementing this type of logic, but this section addresses the mechanism for comparing two variables through a set of relational operators.

Relational operators compare the values of two variables and return a boolean value. The general form of a relation operation is

```
LeftOperand RelationalOperator RightOperand
```

For example:

```
int a = 10;
int b = 10;
int c = 20;
boolean b1 = a == c; // false, 10 is not equal to 20
boolean b2 = a == b; // true, 10 is equal to 10
boolean b3 = a < c; // true, 10 is less than 20
boolean b4 = a < b; // false, 10 is not less than 10
boolean b5 = a <= b; // true, 10 is less than or equal to 10 (equal to)
boolean b6 = a != c; // true, 10 is not equal to 20
```

Bit-Wise Operators

The operators covered thus far in this chapter have been manipulating the interpreted values of these bits (for example, if a was an int with the value 10, a + 1 is equal to 11), but there are some things that can be done directly with those 1s and 0s. Java provides a set of bit-wise operators that looks at each bit in two variables, performs a comparison, and returns the result. The nature of the operator is defined by its truth table.

For the examples in this section, consider the following two bytes:

```
byte a = 10;
byte b = 6;
```

The bits for these values are

```
a = 0000 1010
b = 0000 0110
```

The bit-wise operators are going to define rules in the form of truth tables to apply to these two values for the purpose of building a result. The general form of a bit-wise operation is

```
result = operand
```

AND Operator

The AND operator specifies that both Signals A and B must be charged for the result to be charged. Therefore, AND-ing the bytes 10 and 6 results in 2, as follows:

```
a = 0000 1010 (10)
b = 0000 0110 (6)
  ---- ----
r = 0000 0010 (2)
```

OR Operator

The OR operator (|) defines a bit-wise comparison between two variables.

The OR operator specifies that the result is charged if either Signals A or B are charged. Therefore, OR-ing the bytes 10 and 6 results in 14, as follows:

```
a = 0000 1010 (10)
b = 0000 0110 (6)
  ---- ----
r = 0000 1110 (14)
```

Exclusive OR

The Exclusive OR (XOR) operator (^) defines a bit-wise comparison between two variables according to the truth table shown in Table 3.6.

The XOR operator specifies that the result is charged if Signal A or B is charged, but Signal A and B aren't both charged. It is called exclusive because it is charged only if one of the two is charged. Therefore, XOR-ing the bytes 10 and 6 results in 12, as follows:

```
a = 0000 1010 (10)
b = 0000 0110 (6)
  ---- ----
r = 0000 1100 (12)
```

NOT Operator

The NOT operator (~), also referred to as the bit-wise complement, flips the values of all the charges. If a bit is set to 1, it changes it to 0; if a bit is 0 it changes to 1.

The primary use for bit-wise operations originated in the data compression and communication applications. The problem was that you needed to either store or transmit the state of several different things. In the past, data storage was not as cheap as it is today and communication mechanisms were not over T3 or even a cable modem, but more like speeds of 1200 or 300 baud. To give you an idea about the difference in speed, cable modem and DSL companies tout that they achieve speeds about 20 times faster than 56K modems, but 56K modems are about 45 times faster than 1200-baud modems, and 180 times faster than 300-baud modems. So early communications were 3600 times slower than your cable modem. Thus, there was the need to compact data as much as possible!

If you have 8 different states to send to someone, you can simply assign a bit in a byte to each of the 8 states: 1 is defined to be true and 0 false (or on/off, and so on). Consider reporting the state of 8 different factory devices where devices 0 and 3 are active:

```
Device byte: 0000 1001 (9)
```

To determine whether device 3 is active, a Boolean expression can be determined with the following statement:

```
boolean is3Active = ( deviceByte & 8 ) == 8;
```

AND-ing the deviceByte (0000 1001) with the number 8 (0000 1000) returns 8, as the first bit is 0 (1 & 0 = 0), and using the equality operator it can be compared to 8 to see if that bit is set.

You should understand how bit-wise operators work (they are covered in Java certification exams) and, depending on what area of Java programming you delve into, you might use them in the future.

Logical Operators

Comparison operators enable you to compare two variables to determine whether they are equal or if one is greater than the other, and so on. But what happens when you want to check to see if a variable is in between a range of values? For example, consider validating that someone entered a correct value for an age field in your user interface. You might want to validate that the user is between the ages of 18 and 120; if someone claims to be 700 years old, you might want to check your calendar, prepare for rain, and see if anyone is building an ark! Furthermore, you might want to target a product to children and senior citizens, and therefore validate that the user's age is less than 8 or greater than 55.

To address this need, Java has provided a set of logical operators that enable you to make multiple comparisons and group the result into a single boolean value.

AND

The AND logical operator (&&) returns a true value if both of the variables it is comparing are true. The form of an AND operation is

```
boolean1 && boolean2
```

It compares two Boolean variables and returns true only if both of the Boolean variables are true. For example, consider verifying that someone is of age:

```
boolean isAdult = (age >= 18 ) && (age <= 120)
```

If the age is 20, then 20 is greater than 18 and less than 120, so isAdult is true. If the age is 17, then 17 is not greater than or equal to 18 although it is less than 120; both conditions are not satisfied, so isAdult is false.

OR

The OR logical operator (||) returns a true value if either of the variables it is comparing are true. The form of an OR operation is:

```
boolean1 && boolean2
```

It compares two Boolean variables and returns true if either of the Boolean variables are true. For example, consider verifying that someone is either a child or a senior citizen:

```
boolean isChildOrSenior = (age <= 10 ) || (age >= 55)
```

If the age is 7, then 7 is less than 18, so isChildOrSenior is true. If the age is 17, then 17 is not less than or equal to 10, and it is not greater than or equal to 55; neither condition is satisfied, so isChildOrSenior is false.

Not that this is an inclusive OR, not an exclusive OR; if either or both of the values are true, the result is true.

Shift Operators

After looking at the structure of memory and all the 1s and 0s with their binary assignment, you might notice something interesting. What happens if you move all the bits in your variable to the left? Consider the following:

```
0000 1010 = 8 + 2 = 10
0001 0100 = 16 + 4 = 20
0010 1000 = 32 + 8 = 40Now take that value and move it to the right:

0010 1000 = 32 + 8 = 40
0001 0100 = 16 + 4 = 20
0000 1010 = 8 + 2 = 10
0000 0101 = 4 + 1 = 5
0000 0010 = 2
```

The basic properties of the binary numbering system demonstrates that moving the bits to the left multiplies the value by two and moving the bits to the right divides the value by two. That is interesting and all, but what is the value in that?

Today video cards have high-performance, floating-point arithmetic chips built into them that can perform high-speed mathematical operations, but that was not always the case. In the early days of computer game programming, every ounce of performance had to be squeezed out of code to deliver a decent performing game. To understand just how much math is behind computer animation in game programming, consider a rudimentary flight simulator. From the cockpit we had to compute the location of all objects in sight and draw them. All the objects in the game were drawn with polygons or in some cases, triangles. Some objects in the game could have 50,000 triangles, which results in 150,000 lines, and to maintain 30 frames per second that requires 4,500,000 lines drawn per second. Whoa! How fast can you draw a line? If you can shave off 100 nanoseconds from the line drawing algorithm, that would result in a savings of

```
100ns * 4,500,000 lines = 450,000,000ns = .45 second
```

So, an increase of just 100 nanoseconds results in almost a half a second of savings per second! The bottom line is that every tiny bit of performance that can be improved can have dramatic results.

Okay, fine, now that you have a background on drawing lines, how does this relate to moving bits around?

It turns out that moving bits is exceptionally fast, whereas multiplication and division are very slow. Combine these two statements and what do you get? It would sure be better to make use of the fact that shifting bits can perform multiplication and division instead of using the multiplication and division operators!

That works great for multiplication by two, but how do you handle multiplications by other numbers?

It turns out that addition is also a very inexpensive operation, so through a combination of shifting bits and adding their results can substitute for multiplication in a much more efficient manner. Consider multiplying a value by 35: this is the same as shifting it 5 bits to the left (32), adding that to its value shifted 1 bit to the left (2), and adding that to its unshifted value (1). For more information on this peruse the selection of video game–programming books at your local bookstore.

Shift Left Operator

Shifting a value to the left (<<) results in multiplying the value by a power of two. The general form of a left shift operation is

```
value << number-of-bits
```

The number of bits specifies how many bits to shift the value over. Stated more simply shifting a value, n bits, is the same as multiplying it by 2n. For example:

```
int i=10;
int result = i << 2; // result = 40
```

Shift Right

Shifting a value to the right (>>) is the same as dividing it by a power of two. The general form of a right shift operation is

```
value >> number-of-bits
```

For example:

```
int i = 20;
int result = i >> 2; // result = 5
int j = -20;
int result2 = j >> 2; // result = -5Shift Right (Fill with 0s)
```

Shifting a value to the right has the effect of dividing a value by a power of two, and as you just saw, it preserved the sign of the negative number. Recall that this highest bit in each of Java's numeric data types specified the sign. So, if a right shift operator truly did shift the bits, it would move the sign bit to the right, and hence the number would not be equivalent to a division by a power of 2. The right shift operator performs the function that you would expect it to, but does not actually do a true shift of the value.

If your intent is not to perform division but to perform some true bit operations, this side effect is not desirable. To address this, Java has provided a second version of the right shift operator (>>>) that shifts the bits to the right and fills the new bits with zero.

Therefore, the right-shift operator (>>>) can never result in a negative number because the sign bit will always be filled with a zero.

Operator Precedence

There are all these operators and the good news is that they can all be used together. Consider multiplying a value by 10 and adding 5 to it:

```
int result = a * 10 + 5;
```

But what happens when you want to add 5 to a value, and then multiply it by 10? Does the following work?

```
int result = a + 5 * 10;
```

Consider for a moment that in the compiler it performs its calculations left to right, now how would you add 5 multiplied by 10 to a?

```
int result = 5 * 10 + a;
```

This is starting to get confusing and complicated! Furthermore it is not intuitive! When you study mathematics, you learn that certain operations are reflective, meaning that they can be performed in any order and offer the same result. Thus, the following two statements are equivalent:

```
int result = a + 5 * 10;
int result = 5 * 10 + a;
```

The confusion between these two statements is what operation to perform first; the multiplication or the division? How would you solve this in your math classes? You would simply instrument the statements with parentheses to eliminate the confusion:

```
int result = ( a + 5 ) * 10;
int result = ( 5 * 10 ) + a;
```

These statements are now read as follows: add 5 to a, and then multiply the result by 10; and multiply 5 by 10, and then add a to the result, respectively. In mathematics, the operation enclosed in parentheses is completed before any other operation.

The same mechanism can be implemented in Java, and it is! To qualify what operations to perform first, you can eliminate all ambiguity by explicitly using parentheses to denote what operations are grouped together. But that does not solve the problem of how to handle a statement without parentheses. What is the result of the following operation?

```
int result = 8 + 5 * 10;Is the result 130 or 58?
```

The answer is that there needs to be a set of rules that defines the order of operation execution. This is defined by what is called the operator precedence. The operator precedence defines what operators take precedence over other operators, and hence get executed first. All programming languages define an operator precedence, which is very similar between programming languages, and you must be familiar with it.

Java Operator Precedence

Unary operators are those operators that operate on a single variable, such as increment and decrement (++), positive and negative signs (+ –), the bit-wise NOT operator (~), the logical NOT operator (!), parentheses, and the new operator (to be discussed later).

Arithmetic operators are those operators used in mathematical operations. Here it is important to note that this table is read from left to right, therefore multiplication and division have greater precedence than addition and subtraction. Thus the answer to the aforementioned question:

```
int result = 8 + 5 * 10;
```

The result is 58; multiplication has a higher precedence than addition, so the multiplication is performed first followed by the addition. So, the compiler reads this as multiply 5 by 10 (50) and add 8 to the result (50 + 8 = 58).

Shift operators refer to the bit-wise shift left, shift right, and shift right and fill with zeros operators.

The logical comparison operators follow with the familiar greater than, less than, and equality variations. Comparison operators return a boolean value, so there is one additional operator added: instanceof; this operator will be addressed later when you have a little more Java under your belt.

Next are the bitwise AND, OR, and XOR operators followed by the logical AND and OR (&& ||), referred to as the short-circuit operators.

Next is a new category of operators in the ternary operators; the sole operator in this category is referred to as ternary because it uses three operands when computing its result. It is the following form:

```
a ? b : c;
```

This statement is read as follows: If a is true, then the result of this operation is b, otherwise the result is c. The ternary operator does not have to be comprised of single values, the only requirement is that the first value or operation resolves to be a boolean. Consider the following examples:

```
int result = ( 5 > 3 ) ? 2 : 1; // result is 2
int result = ( 5 < 3 ) ? 2 : 1; // result is 1
```

In the first example 5 is greater than 3, therefore the result is 2; in the second example 5 is not less than 3, so the result is 1. So more clearly written, the form of the ternary operator is:

```
(boolean expression) ? (return if true) : (return if false)
```

The ternary operator is rarely used, and is mainly inherited from Java's initial syntactical base from C/C++. It is a somewhat cryptic shortcut, but is perfectly legal, so be sure to understand how it is used.

The final sets of operators in the operator precedence hierarchy are the assignment operators. The assignment operators include the familiar assignment (=) operator as well as a set of additional assignment operators referred to generically as op= (operator equal). These new operators are shortcut operators used when performing an operation on a variable and assigning the result back to that variable. Consider adding 5 to the variable a; this could be accomplished traditionally as follows:

```
a = a + 5;
```

Because this is such a common operation Java provides a shortcut for it:

```
a += 5;
```

This is read: a plus equal 5, or explicitly a equals a plus 5. The operator equal operator can be applied to all the arithmetic, shift, and bit-wise operators.

Finally, whenever there is ambiguity or you desire a higher degree of readability, you can use parentheses to explicitly qualify the operator precedence yourself.

*~~~ End of Article ~~~*

Java Reference Type Casting

Source	http://javasoft.phpnet.us/res/tut_typecast.html
Date of Retrieval	22/05/2007

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 sibblings.

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 sibblings

  }
}
```

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 3: Decision-Making and Iterations

Java break and continue Statements

Source	http://www.selfimprovement.ch/tech/articleView.php?ArtID=418
Date of Retrieval	22/05/2007

Java's break and continue statements are used to alter the flow of control.

break

The break statement can be used in for, while or do-while loops. It is also used in switch structure. A break statement exits the loop without executing the rest of the code in the loop. In other words, if the program encounters a break inside a for loop, it exits the loop without executing the remaining statements even if the test condition of the loop is still valid.

continue

The continue statement can be used in for, while, and do-while loops. A continue statement skips the rest of the code in the loop and moves on to the next iteration. So, when the program encounters a continue statement inside a for loop, it will skip the remaining statements and simply move on the next iteration.

Example

```
public class BreakAndContinue {
   public static void main(String args[])
   {
      for (int count = 1; count <= 10; count++) {
         if (count == 6)
            break; // break out of the loop
         System.out.println(count);
      }
      for (int count = 1; count <= 10; count++) {
         if (count == 6)
            continue;
// skip remaining code this time but continue looping
         System.out.println(count);
      }
   }
}
```

*~~~ End of Article ~~~*

Java Control Flow Statements

Source	http://java.about.com/od/beginningjava/l/aa_control_1.htm
Date of Retrieval	22/05/2007

Java programs accomplish their tasks by manipulating program data using operators and making decisions by testing the state of program data. When a program makes a decision, it is determining, based on the state of the program data, whether certain lines of code should be executed. For example, a program may examine a variable called hasChanged to determine if it should execute a block of code that saves data to disk. If hasChanged is true, the data saving code is executed. If hasChanged is false, the data saving code is not executed.

At a simplistic level, a program executes statements sequentially in the order in which they appear. It starts with the first line of the "main()" method and ends with the last line, always moving from top to bottom. Most useful programs are vastly more complex than a simple linear sequence. Real programs contain numerous code blocks that must be executed only if certain conditions are present. Other code blocks must be executed repeatedly until certain conditions are met. A hypothetical example: A program loops through an array of employee models and increases the value stored in the salary variable by 10% for each employee that has a hire date of one year ago. (I said it was hypothetical).

Control flow statements are the tools programmers use to make decisions about which statements to execute and to otherwise change the flow of execution in a program. The four categories of control flow statements available in Java are selection, loop, exception, and branch.

Control Flow Statements

Selection and Loop

Selection and loop statements require a conditional expression that evaluates to true or false. If the conditional expression is true, the accompanying block of code is executed. If the conditional expression is false, the accompanying block of code is bypassed. Selection statements will execute the code block only once. Loop statements repeatedly execute the code block, testing the conditional statement each time. Once the conditional statement evaluates to false, the loop exits and execution of the program continues to the next statement following the loop.

Exception

Exception statements are used to gracefully handle unusual events or errors that arise while a program is running. For instance, if a program attempts to open a file but the file doesn't exist, this will cause an I/O error. Exception handling in Java is a two step process. First, an exceptional condition is detected during program execution by the JVM or application code. At this point a new exception object is created that describes the situation and is passed up the call stack using the throw statement. Each method in the call stack now has the option to handle it or let it continue to bubble up the call stack.

Branch

Branch statements explicitly redirect the flow of program execution. The most infamous branch statement in the history of computing is goto. In Java, goto is an unimplemented keyword, so it cannot be used in Java programs. In other languages, goto is used to redirect program execution to another line of code in the program by specifying a line number or other type of label. In Java, break, continue, and label: fill the role of goto, but without many of goto's negatives. return is a branching statement that redirects execution out of the current method and back to the calling method. It also returns a value to the calling method. This value is either a primitive, object, or void. void is used to return execution without returning a value. Also, whatever the return value is, its type must match the return type listed in the method declaration.

Let's take a look at control flow statements' general form:

```
statement_keyword(conditional_expression) {
  statements_to_execute
}
```

Notice there are three parts to the general form above. The statement_keyword is the Java language keyword that defines what type of control is being exerted on the statements_to_execute code block. The conditional_expression defines the conditions that must be present for statements_to_execute to be executed.

Notice statements_to_execute is wrapped in curly braces {}. If statements_to_execute consists of a single statement only, the curly braces are optional. Most Java programmers consider it to be good coding style to always include the curly braces. Leaving out curly braces can cause subtle bugs because most Java programmers expect the curly braces out of habit.

Here are some examples of control statements. We will cover all of these and more in upcoming tutorials.

```
//Example 1
if(conditional_expression) {
  statements_to_execute
}

//Example 2
if(conditional_expression) {
  statements_to_execute
}
else {
  statements_to_execute
}

//Example 3
while(conditional_expression) {
  statements_to_execute
}

//Example 4
for(int i = 0; conditional_expression; i++) {
  statements_to_execute
}
```

*~~~ End of Article ~~~*

Loops - Introduction

Source	http://www.leepoint.net/notes-java/flow/loops/loops.html
Date of Retrieval	22/05/2007

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 ~~~*

Session 4: Introducing Classes

Class and object initialization

Source	http://www.javaworld.com/javaworld/jw-11-2001/jw-1102-java101.html
Date of Retrieval	22/05/2007

Initialization prepares classes and objects for use during a program's execution. Although we tend to think of initialization in terms of assigning values to variables, initialization is so much more. For example, initialization might involve opening a file and reading its contents into a memory buffer, registering a database driver, preparing a memory buffer to hold an image's contents, acquiring the resources necessary for playing a video, and so on. Think of anything that prepares a class or an object for use in a program as initialization.

Java supports initialization via language features collectively known as initializers. Because Java handles class initialization differently from object initialization, and because classes initialize before objects, we first explore class initialization and class-oriented initializers. Later, we explore object initialization and object-oriented initializers.

Class initialization

A program consists of classes. Before a Java application runs, Java's class loader loads its starting class -- the class with a public static void main(String [] args) method -- and Java's byte code verifier verifies the class. Then that class initializes. The simplest kind of class initialization is automatic initialization of class fields to default values. Listing 1 demonstrates that initialization:

Listing 1. ClassInitializationDemo1.java

```
                    // ClassInitializationDemo1.java
class ClassInitializationDemo1
{
   static boolean b;
   static byte by;
   static char c;
   static double d;
   static float f;
   static int i;
   static long l;
   static short s;
   static String st;
   public static void main (String [] args)
```

```
  {
    System.out.println ("b = " + b);
    System.out.println ("by = " + by);
    System.out.println ("c = " + c);
    System.out.println ("d = " + d);
    System.out.println ("f = " + f);
    System.out.println ("i = " + i);
    System.out.println ("l = " + l);
    System.out.println ("s = " + s);
    System.out.println ("st = " + st);
  }
}
```

ClassInitializationDemo1's static keyword introduces a variety of class fields. As you can see, no explicit values assign to any of those fields. And yet, when you run ClassInitializationDemo1, you see the following output:

```
b = false
by = 0
c =
d = 0.0
f = 0.0
i = 0
l = 0
s = 0
st = null
```

The false, 0, 0.0, and null values are the type-oriented representations of default values. They represent the result of all bits automatically set to zero in each class field. And what automatically set those bits to zero? The JVM, after a class is verified. (Note: In the preceding output, you do not see a value beside c = because the JVM interprets c's default value as the nondisplayable null value.)

Class field initializers

After automatic initialization, the next simplest kind of class initialization is the explicit initialization of class fields to values. Each class field explicitly initializes to a value via a class field initializer.

*~~~ End of Article ~~~*

A Brief Introduction to Classes

Source	http://www.imsc.res.in/Computer/local/Docs/Java/java/javaOO/intro.html
Date of Retrival	22/05/2007

Following is the code for a class called SimplePoint that represents a point in 2D space:

```
public class SimplePoint {
    public int x = 0;
    public int y = 0;
}
```

This segment of code declares a class-- a new data type really-- called SimplePoint. The SimplePoint class contains two integer member variables, x and y. The public keyword preceding the declaration for x and y means that any other class can freely access these two members.

You create an object from a class such as SimplePoint by instantiating the class. When you create a new SimplePoint object (we show you how shortly), space is allocated for the object and its members x and y. In addition, the x and y members inside the object are initialized to 0 because of the assignment statements in the declarations of these two members.

Now, here's a class, SimpleRectangle, that represents a rectangle in 2D space:

```
public class SimpleRectangle {
    public int width = 0;
    public int height = 0;
    public SimplePoint origin = new SimplePoint();
}
```

This segment of code declares a class (another data type)--SimpleRectangle-- that contains two integer members, width and height. SimpleRectangle also contains a third member, origin, whose data type is SimplePoint. Notice that the class name SimplePoint is used in a variable declaration as the variable's type. You can use the name of a class anywhere you can use the name of a primitive type.

Just as width "is an" integer and height "is an" integer, origin "is a" SimplePoint. On the other hand, a SimpleRectangle object "has a" SimplePoint. The distinction between "is a" and "has a" is critical because only an object that "is a" SimplePoint can be used where a SimplePoint is called for.

As with SimplePoint, when you create a new SimpleRectangle object, space is allocated for the object and its members, and the members are initialized according to their declarations. Interestingly, the initialization for the origin member creates a SimplePoint object with this code: new SimplePoint().

The SimplePoint and SimpleRectangle classes are simplistic implementations for these classes. Both should provide a mechanism for initializing their members to values other than 0. Additionally, SimpleRectangle could provide a method for computing its area, and because SimpleRectangle creates a SimplePoint when it's created, the class should provide for the clean up of the SimplePoint when SimpleRectangle gets cleaned up. So, here's a new version of SimplePoint, called Point, that contains a constructor which you can use to initialize a new Point to a value other than (0,0):

```
public class Point {
   public int x = 0;
   public int y = 0;
        // a constructor!
   public Point(int x, int y) {
        this.x = x;
        this.y = y;
   }
}
```

Now, when you create a Point, you can provide initial values for it like this:

```
new Point(44, 78)
```

The values 44 and 78 are passed into the constructor and subsequently assigned to the x and y members of the new Point object as shown here:

Now, let's beef up the SimpleRectangle class. Here's a new version of SimpleRectangle, called Rectangle, that contains four constructors, a method to "move" the rectangle, a method to compute the area of the rectangle, and a finalize method to provide for clean up:

```
public class Rectangle {
   public int width = 0;
   public int height = 0;
   public Point origin;
        // four constructors
   public Rectangle() {
        origin = new Point(0, 0);
   }
   public Rectangle(Point p) {
        origin = p;
   }
   public Rectangle(int w, int h) {
        this(new Point(0, 0), w, h);
   }
   public Rectangle(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
   }
        // a method for moving the rectangle
```

```
    public void move(int x, int y) {
        origin.x = x;
        origin.y = y;
    }
        // a method for computing the area of the rectangle
    public int area(int x, int y) {
        return width * height;
    }
        // clean up!
    protected void finalize() throws Throwable {
        origin = null;
        super.finalize();
    }
}
```

The four constructors allow for different types of initialization. You can create a new Rectangle and let it provide default values for everything, or you can specify initial values for the origin, the width and the height, or for all three when you create the object. You'll see more of this version of the Rectangle class in the next section.

This section glossed over some details and left some things unexplained, but it provides the basis you need to understand the rest of this lesson. After reading this section, you should know that

objects are created from classes

an object's class is its type

how "is a" differs from "has a"

the difference between reference and primitive types.

You also should have a general understanding or a feeling for the following:

How to create an object from a class

What constructors are

What the code for a class looks like

What member variables are

How to initialize objects

What methods look like

Now, let's look in detail at the life cycle of an object, specifically how to create, use, and destroy an object.

*~~~ End of Article ~~~*

Java Classes and Objects

Source	http://java.about.com/od/beginningjava/l/aa_objintro_1.htm
Date of Retrieval	22/05/2007

Java is an object-based language, though, and real-world programs use dozens, if not hundreds, of classes. One cannot avoid using objects in Java, so an understanding of objects and object-oriented programming (OOP) is useful even to beginning Java programmers.

What is a Type?

A type is a category or grouping that describes a piece of data. Every datum in Java can be identified by its type. Data of the same type share common characteristics. If a datum is of type int, for example, then the programmer knows that it is a whole number that cannot be larger than 231 because these are characteristics that all ints share.

Java has two general categories of types: primitives and classes. Examples of primitives are int, float, and boolean. Primitive types' characteristics cannot be extended or modified by programmers. They represent simple pieces of data such as numbers, bytes, and characters. Classes are user-defined types and are created by programmers to represent more complex objects when simple primitives are insufficient. Many useful classes have already been defined by Sun engineers as part of the J2SE, J2EE, and J2ME libraries. Classes are frequently extended and modified by programmers.

Classes Describe Objects

Classes are the basic building blocks of Java programs. Classes can be compared to the blueprints of buildings. Instead of specifying the structure of buildings, though, classes describe the structure of "things" in a program. These things are then created as physical software objects in the program. Things worth representing as classes are usually important nouns in the problem domain. A Web-based shopping cart application, for example, would likely have classes that represent customers, products, orders, order lines, credit cards, shipping addresses and shipping providers.

Unlike data structures in other languages which only contain data, Java classes consist of both attributes and behaviors. Attributes represent the data that is unique to an instance of a class, while behaviors are methods that operate on the data to perform useful tasks.

Class Syntax

Use the following syntax to declare a class in Java:

```
  //Contents of SomeClassName.java
  [ public ] [ ( abstract | final ) ] class SomeClassName [ extends SomeParentClass ] [
implements SomeInterfaces ]
  {
    // variables and methods are declared within the curly braces
}
```

- A class can have public or default (no modifier) visibility.
- It can be either abstract, final or concrete (no modifier).
- It must have the class keyword, and class must be followed by a legal identifier.
- It may optionally extend one parent class. By default, it will extend java.lang.Object.
- It may optionally implement any number of comma-separated interfaces.
- The class's variables and methods are declared within a set of curly braces '{}'.
- Each .java source file may contain only one public class. A source file may contain any number of default visible classes.
- Finally, the source file name must match the public class name and it must have a .java suffix.

Here is an example of a Horse class. Horse is a subclass of Mammal, and it implements the Hoofed interface.

```
public class Horse extends Mammal implements Hoofed
{
  //Horse's variables and methods go here
}
```

What is an Object?

Objects are the physical instantiations of classes. They are living entities within a program that have independent lifecycles and that are created according to the class that describes them. Just as many buildings can be built from one blueprint, many objects can be instantiated from one class. Many objects of different classes can be created, used, and destroyed in the course of executing a program.

Attributes

Although several buildings may be built from the same blueprint and, hence, have the same attributes, such as twelve windows, exterior paint, and a driveway, each instance of the building may have different values for these attributes. One building may have twelve double-pane windows while another may have twelve single-pane windows. One building may have hunter green exterior paint while another may have white exterior paint. The attributes are identical but the values of the attributes are different. Similarly, a class describes what attributes an object will have but each object will have its own values for those attributes. In our shopping cart example, a customer class may specify that customer objects have a first name, last name, and phone number. Each Customer object, though, will have different values for first name, last name, and phone number.

Attributes are represented by non-local variables. These are variables that are not declared within method bodies.

Behaviors

A building has behaviors in addition to attributes: Toilets are flushed; furnaces fire to maintain temperature; doors open and close. Classes describe all of the behaviors of its objects. In our shopping cart example: A customer may place an order; a credit card may reject a purchase; an order may tally line items to generate a total.

Behaviors are represented by methods. An object may call, or invoke, its own methods, or it may call another object's methods that are visible to it.

Constructors

In order to be used by a program, an object must first be instantiated from its class definition. A special type of method called a constructor is used to define how objects are created. A constructor is called with the new keyword. new tells the JVM to allocate memory, initialize instance variables, and assign the object a reference code that uniquely identifies it within the JVM.

Let's create an instance of a Car:

```
Car myYugo = new Car();
```

Messages

Objects cooperate and communicate with other objects in a program by passing messages to one another. When an object invokes a method on itself or another object, it passes a message to the object that contains the target method. The message identifies the method to be invoked and any required data (known as arguments) that the method needs. When the method finishes executing, either a return value or void is passed back to the original invoking object, completing the message.

Let us create another Car object and pass an accelerate message to it.

```
Car myYugo = new Car();
int currentSpeed;
currentSpeed = myYugo.accelerate();
```

Class Inheritance via extends and implements

As you saw in the Horse example, a class can extend one other class and implement many Java interfaces. Extending and implementing is the Java mechanism for representing class inheritance. Inheritance represents an "is a" relationship between two classes. The Horse is a Mammal, and it is a Hoofed. When a child class, or subclass, extends a parent class, or super class, it inherits the parent class's visible variables and methods. The child class can read and write the parent class's visible attributes, it can pass messages to the parent class's visible methods, and it can even override (or re-implement) the parent class's visible methods. We'll cover inheritance in-depth in a future installment of the Java Programming Tutorial.

Now let's continue on and see a working example of using multiple objects in Java.

Now that we have covered some of the theory of objects, it is time to have a little fun. We'll create an object hierarchy with a Car super class and a Mustang subclass. Then we'll take them both for a test drive with the TestDrive class.

Below is the complete code for the Car, Mustang, and TestDrive classes. Create a subdirectory named objintro. Create a file in objintro for each class below. Copy the contents of the classes into the files, and save the files so the class and file name match (remember the .java extension) . So, Car would be saved into a file named Car.java, Mustang into Mustang.java, TestDrive into TestDrive.java. Compile the classes with the javac command. Then run TestDrive with the java command. See First Java Program for more details on compiling and executing Java programs.

Car

```
package objintro;

/**
 *  Represents a car.
 *  The attributes are speed and color.
 *  The methods are accelerate,
 *  decelerate, getSpeed, getColor, getAcceleration
 *  and getMaxSpeed.
 *
 *  Add your own attributes and behaviors
 *  for breaking and for shifting gears.
 *
 * @author Kevin Taylor guide@java.about.com
 */
public class Car {

  protected int speed = 0;
  protected String color = "black";
  private static final int MIN_SPEED = 0;
  private static final int CAR_MAX_SPEED = 100;
  private static final int CAR_ACCELERATION = 10;

  /**
   * Default constructor to create a new
   * Car object.
   */
  public Car() {
    //creates a new object
  }

  /**
   * Simulates pressing the accelerator.
```

```
     * @return the new speed
     */
    public int accelerate() {
        int newSpeed = speed + getAcceleration();
        if(newSpeed <= getMaxSpeed()) {
            speed = newSpeed;
        }
        else {
            speed = getMaxSpeed();
        }
        return speed;
    }

    /**
     * Simulates releasing the accelerator.
     * @return the new speed
     */
    public int decelerate() {
        if(speed > MIN_SPEED) {
            speed--;
        }
        return speed;
    }

    /**
     * @return the current speed
     */
    public int getSpeed() {
        return speed;
    }

    /**
     * @return the max speed
     */
    public int getMaxSpeed() {
        return CAR_MAX_SPEED;
    }

    /**
     * @return the max speed
     */
    public String getColor() {
        return color;
    }

    /**
     * @return the car's acceleration
     */
    public int getAcceleration() {
        return CAR_ACCELERATION;
    }
}
```

Mustang

```
package objintro;

/**
 * Represents a Mustang.
 * It overrides (changes) the getMaxSpeed method,
 * which sets an upper limit for the accelerate
 * method. It also overrides getAcceleration.
 *
 * @author Kevin Taylor guide@java.about.com
 */
public class Mustang extends Car {

    private static final int MUSTANG_MAX_SPEED = 150;
    private static final int MUSTANG_ACCELERATION = 20;

    /**
     * Constructors are used to initialize
     * the objects variables.
     *
     * The Mustang color is define when the constructor
     * is called, whereas the Car objects are always black.
     */
    public Mustang(String passedColor) {
        // creates a new instance of Mustang and
        // assigns a color
        color = passedColor;
    }
    /**
     * This method is implemented in the Car class and
     * in the Mustang class. The Mustang version overrides
     * the Car version.
     *
     * @return  The max speed for a Mustang.
     */
    public int getMaxSpeed() {
      return MUSTANG_MAX_SPEED;
    }

    /**
     * This method is implemented in the Car class and
     * in the Mustang class. The Mustang version overrides
     * the Car version.
     *
     * @return  The acceleration for a Mustang.
     */
    public int getAcceleration() {
      return MUSTANG_ACCELERATION;
    }
  }
```

TestDrive

```
  package objintro;

  /**
   * TestDrive demonstrates creating and calling
   * methods on Car and Mustang objects.
   *
   * @author Kevin Taylor java.guide@about.com
   */
  public class TestDrive {

    //The Java virtual machine (JVM) always starts
    //execution with the 'main' method of the class passed
    //as a argument to the java command
    public static void main(String []args) {
      TestDrive td = new TestDrive();
      td.start();
      //exit TestDrive
    }
    private void start() {
      //Create a Car
      Car yugo = new Car();
      //Take it for a drive
      System.out.println("Starting yugo test drive!");
      driveCar(yugo);

      //Create a Mustang
      //Remember, myMustang is a Mustang AND a Car
      Car pony = new Mustang("red");
      //Take it for a drive
      System.out.println("Starting mustang test drive!");
      driveCar(pony);
    }
    public static void driveCar(Car c) {
      System.out.println("Car color is: " + c.getColor());
      //press the accelerator 15 "times"
      for(int i = 0; i < 15; i++) {
        System.out.println("accelerating: " + c.accelerate());
      }
      //release the accelerator 5 "times"
      for(int i = 0; i < 5; i++) {
        ;
        System.out.println("decelerating: " + c.decelerate());
      }
      System.out.println("final cruising speed: " + c.getSpeed());
    }
  }
```

When you compile and run the program, you should see something like the following:

```
  ktaylor$ javac objintro/Car.java objintro/Mustang.java objintro/TestDrive.java
  ktaylor$ java objintro.TestDrive
  Starting yugo test drive!
  Car color is: black
  accelerating: 10
  accelerating: 20
  accelerating: 30
  accelerating: 40
  accelerating: 50
  accelerating: 60
  accelerating: 70
```

```
accelerating: 80
accelerating: 90
accelerating: 100
accelerating: 100
accelerating: 100
accelerating: 100
accelerating: 100
accelerating: 100
decelerating: 99
decelerating: 98
decelerating: 97
decelerating: 96
decelerating: 95
final cruising speed: 95
Starting mustang test drive!
Car color is: red
accelerating: 20
accelerating: 40
accelerating: 60
accelerating: 80
accelerating: 100
accelerating: 120
accelerating: 140
accelerating: 150
accelerating: 150
accelerating: 150
accelerating: 150
accelerating: 150
accelerating: 150
accelerating: 150
accelerating: 150
decelerating: 149
decelerating: 148
decelerating: 147
decelerating: 146
decelerating: 145
final cruising speed: 145
```

*~~~ End of Article ~~~*

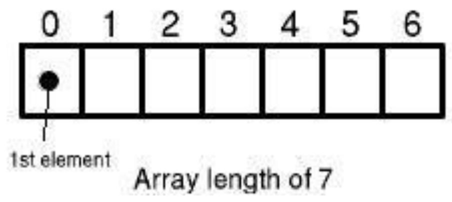
Session 5: Arrays

Java Arrays

Source	http://java.about.com/od/beginningjava/l/aa_array.htm
Date of Retrieval	22/05/2007

Fixed-Length Data Structures

As we saw in Java Variables, you can store program data in variables. Each variable has an identifier, a type, and a scope. When you have closely related data of the same type and scope, it is often convenient to store it together in a data structure instead of in individual variables. The most common data structure is the array. Arrays are fixed-length structures for storing multiple values of the same type. An array implicitly extends java.lang.Object so an array is an instance of Object. But arrays are directly supported language features. This means that their performance is on par with primitives and that they have a unique syntax that is different than objects.



Structure of Java Arrays
The Java array depicted above has 7 elements. Each element in an array holds a distinct value. In the figure, the number above each element shows that element's index. Elements are always referenced by their indices. The first element is always index 0. This is an important point, so I will repeat it! The first element is always index 0. Given this zero-based numbering, the index of the last element in the array is always the array's length minus one. So in the array pictured above, the last element would have index 6 because 7 minus 1 equals 6.

Java Array Declaration
An array variable is declared the same way that any Java variable is declared. It has a type and a valid Java identifier. The type is the type of the elements contained in the array. The [] notation is used to denote that the variable is an array. Some examples:

```
int[] counts;
String[] names;
int[][] matrix; //this is an array of arrays
```

Java Array Initialization
Once an array variable has been declared, memory can be allocated to it. This is done with the **new** operator, which allocates memory for objects. (Remember arrays are implicit objects.) The **new** operator is followed by the type, and finally, the number of elements to

allocate. The number of elements to allocate is placed within the [] operator. Some examples:

```
counts = new int[5];
names = new String[100];
matrix = new int[5][];
```

An alternate shortcut syntax is available for declaring and initializing an array. The length of the array is implicitly defined by the number of elements included within the {}. An example:

```
String[] flintstones = {"Fred", "Wilma", "Pebbles"};
```

Java Array Usage

To reference an element within an array, use the [] operator. This operator takes an int operand and returns the element at that index. Remember that array indices start with zero, so the first element is referenced with index 0.

```
int month = months[3]; //get the 4th month (April)
```

In most cases, a program will not know which elements in an array are of interest. Finding the elements that a program wants to manipulate requires that the program loop through the array with the **for** construct and examine each element in the array.

```
 String months[] =
    {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
     "July", "Aug", "Sep", "Oct", "Nov", "Dec"};

//use the length attribute to get the number
//of elements in an array
for(int i = 0; i < months.length; i++ ) {
   System.out.println("month: " + month[i]);
}
```

Using Java 5.0, the enhanced **for** loop makes this even easier:

```
 String months[] =
    {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
     "July", "Aug", "Sep", "Oct", "Nov", "Dec"};

// Shortcut syntax loops through array months
// and assigns the next element to variable month
// for each pass through the loop
for(String month: months) {
   System.out.println("month: " + month);
}
```

Arrays are a great way to store several to many values that are of the same type and that are logically related to one another: lists of invoices, lists of names, lists of Web page hits, etc. But, being fixed-length, if the number of values you are storing is unknown or changes, there are better data structures available in the Java Collections API. These include ArrayList and HashSet, amongst others.

*~~~ End of Article ~~~*

Multidimensional Array

Source	http://www.cafeaulait.org/javatutorial.html
Date of Retrieval	22/05/2007

Two Dimensional Arrays

The most common kind of multidimensional array is a two-dimensional array. If you think of a one-dimensional array as a column of values you can think of a two-dimensional array as a table of values like so:

	c0	c1	c2	c3
r0	0	1	2	3
r1	1	2	3	4
r2	2	3	4	5
r3	3	4	5	6
r4	4	5	6	7

Here we have an array with five rows and four columns. It has twenty total elements. However we say it has dimension five by four, not dimension twenty. This array is not the same as a four by five array like this one:

	c0	c1	c2	c3	c4
r0	0	1	2	3	4
r1	1	2	3	4	5
r2	2	3	4	5	6
r3	3	4	5	6	7

We need to use two numbers to identify a position in a two-dimensional array. These are the element's row and column positions. For instance if the above array is called J then J[0][0] is 0, J[0][1] is 1, J[0][2] is 2, J[0][3] is 3, J[1][0] is 1, and so on.

Here's how the elements in a four by five array called M are referred to:

M[0][0]	M[0][1]	M[0][2]	M[0][3]	M[0][4]

M[1][0]	M[1][1]	M[1][2]	M[1][3]	M[1][4]
M[2][0]	M[2][1]	M[2][2]	M[2][3]	M[2][4]
M[3][0]	M[3][1]	M[3][2]	M[3][3]	M[3][4]

Declaring, Allocating and Initializing Two Dimensional Arrays

Two dimensional arrays are declared, allocated and initialized much like one dimensional arrays. However we have to specify two dimensions rather than one, and we typically use two nested `for` loops to fill the array.

The array examples above are filled with the sum of their row and column indices. Here's some code that would create and fill such an array:

```
class FillArray {

  public static void main (String args[]) {

    int[][] M;
    M = new int[4][5];

    for (int row=0; row < 4; row++) {
      for (int col=0; col < 5; col++) {
        M[row][col] = row+col;
      }
    }

  }

}
```

Of course the algorithm you would use to fill the array depends completely on the use to which the array is to be put. Here is a program which calculates the identity matrix for a given dimension. The identity matrix of dimension N is a square matrix which contains ones along the diagonal and zeros in all other positions.

```
class IDMatrix {

  public static void main (String args[]) {

    double[][] ID;
    ID = new double[4][4];

    for (int row=0; row < 4; row++) {
      for (int col=0; col < 4; col++) {
        if (row != col) {
          ID[row][col]=0.0;
        }
        else {
          ID[row][col] = 1.0;
        }
      }
    }

  }
}
```

In two-dimensional arrays ArrayIndexOutOfBounds errors occur whenever you exceed the maximum column index or row index. Unlike two-dimensional C arrays, two-dimensional Java arrays are not just one-dimensional arrays indexed in a funny way.

Exercises

1. Write a program to generate the HTML for the above tables.

Multidimensional Arrays

You don't have to stop with two dimensional arrays. Java lets you have arrays of three, four or more dimensions. However chances are pretty good that if you need more than three dimensions in an array, you're probably using the wrong data structure. Even three dimensional arrays are exceptionally rare outside of scientific and engineering applications.

The syntax for three dimensional arrays is a direct extension of that for two-dimensional arrays. Here's a program that declares, allocates and initializes a three-dimensional array:

```
class Fill3DArray {

  public static void main (String args[]) {

    int[][][] M;
    M = new int[4][5][3];

    for (int row=0; row < 4; row++) {
      for (int col=0; col < 5; col++) {
        for (int ver=0; ver < 3; ver++) {
          M[row][col][ver] = row+col+ver;
        }
      }
    }

  }

}
```

We need three nested `for` loops here to handle the extra dimension.

The syntax for still higher dimensions is similar. Just add another pair pf brackets and another dimension.

Unbalanced Arrays

Like C Java does not have true multidimensional arrays. Java fakes multidimensional arrays using arrays of arrays. This means that it is possible to have *unbalanced arrays*. An unbalanced array is a multidimensional array where the dimension isn't the same for all rows. IN most applications this is a horrible idea and should be avoided.

Searching

One common task is searching an array for a specified value. Sometimes the value may be known in advance. Other times you may want to know the largest or smallest element.

Unless you have some special knowledge of the contents of the array (for instance, that it is sorted) the quickest algorithm for searching an array is straight-forward linear search. Use a `for` loop to look at every element of the array until you find the element you want. Here's a simple method that prints the largest and smallest elements of an array:

```
   static void printLargestAndSmallestElements (int[] n) {

    int max = n[0];
    int min = n[0];

    for (int i=1; i < n.length; i++) {
      if (max < n[i]) {
        max = n[i];
      }
      if (min > n[i]) {
        min = n[i];
      }
    }

    System.out.println("Maximum: " + max);
    System.out.println("Minimum: " + min);

    return;

  }
```

If you're going to search an array many times, you may want to sort the array, before searching it. We'll discuss sorting algorithms in the next section.

Sorting

All sorting algorithms rely on two fundamental operations, comparison and swapping. Comparison is straight-forward. Swapping is a little more complex. Consider the following problem. We want to swap the value of `a` and `b`. Most people propose something like this as the solution:

```
class Swap1 {

  public static void main(String args[]) {

   int a = 1;
   int b = 2;

   System.out.println("a = "+a);
   System.out.println("b = "+b);

   // swap a and b

   a = b;
```

```
    b = a;

    System.out.println("a = "+a);
    System.out.println("b = "+b);

  }

}
```

This produces the following output:

```
a = 1
b = 2
a = 2
b = 2
```

That isn't what you expected! The problem is that we lost track of the value 1 when we put the value of `b` into `a`. To correct this we need to introduce a third variable, `temp`, to hold the original value of `a`.

```
class Swap2 {

  public static void main(String args[]) {

    int a = 1;
    int b = 2;
    int temp;

    System.out.println("a = "+a);
    System.out.println("b = "+b);

    // swap a and b

    temp = a;
    a = b;
    b = temp;

    System.out.println("a = "+a);
    System.out.println("b = "+b);

  }

}
```

This code produces the output we expect:

```
a = 1
b = 2
a = 2
b = 1
```

Bubble Sort

Now that we've learned how to properly swap the values of two variables, let's proceed to sorting. There are **many** different sorting algorithms. One of the simplest and the most popular algorithms is referred to as *bubble sort*. The idea of bubble sort is to start at the top of the array. We compare each element to the next element. If its greater than that element then we swap the two. We pass through the array as many times as necessary to sort it.

The smallest value bubbles up to the top of the array while the largest value sinks to the bottom. (You could equally well call it a sink sort, but then nobody would know what you were talking about.) Here's the code:

```
import java.util.*;

class BubbleSort {

  public static void main(String args[]) {

   int[] n;
   n = new int[10];
   Random myRand = new Random();

   // initialize the array
   for (int i = 0; i < 10; i++) {
     n[i] = myRand.nextInt();
   }

   // print the array's initial order
   System.out.println("Before sorting:");
   for (int i = 0; i < 10; i++) {
     System.out.println("n["+i+"] = " + n[i]);
   }

   boolean sorted = false;
   // sort the array
   while (!sorted) {
     sorted = true;
     for (int i=0; i < 9; i++) {
       if (n[i] > n[i+1]) {
         int temp = n[i];
         n[i] = n[i+1];
         n[i+1] = temp;
         sorted = false;
       }
     }
   }

  // print the sorted array
  System.out.println();
  System.out.println("After sorting:");
   for (int i = 0; i < 10; i++) {
     System.out.println("n["+i+"] = " + n[i]);
   }

  }
}
```

In this case we have sorted the array in ascending order, smallest element first. It would be easy to change this to sort in descending order.

*~~~ End of Article ~~~*

Session 6: Packages and Access Specifiers

Access Levels for Interfaces

Source	http://www.artima.com/objectsandjava/webuscript/PackagesAccess1.html
Date of Retrieval	22/05/2007

Access Levels for Interfaces

Interfaces have slightly different rules for access levels, because every field and method defined by an interface is implicitly public. You can't use the keywords `private` or `protected` on the fields and methods of interfaces. If you leave off the `public` keyword when declaring interface members, as is officially recommended by the Java Language Specification, you do not get package access. You still get public access. Therefore, you can't hide any implementation details of a package inside an interface (You can't hide an interface's members). On the other hand, you can hide the entire interface. If you don't declare an interface public, the interface as a whole will only be available to other types in the same package. As with classes, you should make interfaces public only if they are needed by classes and interfaces defined in other packages. Here's an example of two interfaces. Interface `Soakable` is part of the internal implementation of a package. Interface `Washable` is part of the external implementation of the package:

```
// In Source Packet in file
// packages/ex5/com/artima/vcafe/dishes/Washable.java
package com.artima.vcafe.dishes;

public interface Washable {

    void wash();
}

// In Source Packet in file
// packages/ex5/com/artima/vcafe/dishes/Soakable.java
package com.artima.vcafe.dishes;

interface Soakable extends Washable {

    void soak();
}
```

In this example, `wash()` and `breakIt()` are not explicitly declared public, because they are public by default. Because the `Washable` interface as a whole is not explicitly declared as public, however, it has package access. Interface `Washable` is only be accessible to other types declared in the `com.artima.vcafe.dishes` package. Interface `Breakable`, because it is declared as public, is available to any type declared in any package.

*~~~ End of Article ~~~*

Rules for Class Member Access Levels

Source	http://www.artima.com/objectsandjava/webuscript/PackagesAccess1.html
Date of Retrieval	22/05/2007

Rules of Thumb for Class Member Access Levels

The most important rule of thumb concerning the use of access control modifiers is to keep data private unless you have a good reason not to. Keeping data private is the best way to maximize the robustness and ease of modification of your classes. If you keep data private, other classes can access a class's fields only through its methods. This enables the designer of a class to keep control over the manner in which the class's fields are manipulated. If fields are not private, other classes can change the fields directly, possibly in unpredictable and improper ways. Keeping data private also enables a class designer to more easily change the algorithms and data structures used by a class. Given that other classes can only manipulate a class's private fields indirectly, through the class's methods, other classes will depend only upon the external interface to the private fields provided by the methods. You can change the private fields of a class and modify the code of the methods that manipulate those fields. As long as you don't alter the signature and return type of the methods, the other classes that depended on the previous version of the class will still link properly. Making fields private is the fundamental technique for hiding the implementation of Java classes.

As mentioned in an earlier chapter, one other reason to make data private is because you synchronize access to data by multiple threads through methods. This justification for keeping data private will be discussed in Chapter 17.

As a general rule, the only good non-private field is a final one. Given that final fields cannot be changed after they are initialized, non-private final fields do not run the risk of improper manipulation by other classes. Other classes can use the field, but not change it.

A common use of non-private final fields is to define names to represent a set of valid values that may be passed to (or returned from) a method. As mentioned in Chapter 5, such fields are called constants and are declared static as well as non-private and final. A Java programmer will create constants in this manner in situations where a C++ programmer would have used an enumerated type or declared a "const" member variable.

Rules of thumb such as the ones outlined above are called rules of thumb for a reason: They are not absolute laws. Java allows you to declare fields in classes with any kind of access level, and you may very well encounter situations in which declaring a field private is too restrictive. One potential justification for non-private fields is simple trust. In some situations you may have absolute trust of certain other classes. For example, perhaps you are designing a small set of types that must work together closely to solve a particular problem. It may make sense to put all of these types in their own package, and allow them direct access to some of each other's fields. Although this would create interdependencies between the internal implementations of the classes, you may deem the level of interdependency to be acceptable. If later you change the internal implementation of one of the classes, you'll have to update the other classes that relied on the original implementation. As long as you don't grant access to the fields to classes outside the package, any repercussions of the implementation change will remain inside the package.

Nevertheless, the general rule of thumb in designing packages is to treat the types that share the same package with as much suspicion as types from different packages. If you don't trust classes from other packages to directly manipulate your class's fields, neither should you let classes from the same package directly manipulate them. Keep in mind that you usually can't prevent another programmer from adding new classes to your package, even if you only deliver class files to that programmer. If you leave all your fields with package access, a programmer using your package can easily gain access to those fields by creating a class and declaring it as a member of your package. Therefore, it is best to keep data private, except sometimes when the data is final, so that irrespective of what package classes are defined in, all classes must go through methods to manipulate each other's fields.

The methods you define in public classes should have whatever level of access control matches their role in your program. You should exploit the full range of access levels provided by Java on the methods of your public classes, assigning to each method the most restrictive access level it can reasonably have.

You can use the same rule of thumb to design classes that have package access. You must keep in mind, however, that for package-access classes, fields and methods declared public won't be accessible outside the package. Fields and methods declared protected won't be accessible to subclasses in other packages, because there won't be any subclasses in other packages. Only classes within the same package will be able to subclass the package-access class. Still, you should probably keep the same mindset when designing package-access classes as you do when designing public classes, because at some later time you may turn a package-access class into a public class.

*~~~ End of Article ~~~*

Session 7: Inheritance and Interfaces

Abstract Classes vs. Interface

Source	http://www.javaworld.com/javaworld/javaqa/2001-04/03-qa-0420-abstract.html
Date of Retrieval	22/05/2007

Abstract classes vs. interfaces

In Java, under what circumstances would you use abstract classes instead of interfaces? When you declare a method as abstract, can other nonabstract methods access it? In general, could you explain what abstract classes are and when you might use them?

Those are all excellent questions: the kind that everyone should ask as they begin to dig deeper into the Java language and object-oriented programming in general

Yes, other nonabstract methods can access a method that you declare as abstract.

But first, let's look at when to use normal class definitions and when to use interfaces. Then I'll tackle abstract classes.

Class vs. interface

Some say you should define all classes in terms of interfaces, but I think recommendation seems a bit extreme. I use interfaces when I see that something in my design will change frequently.

For example, the Strategy pattern lets you swap new algorithms and processes into your program without altering the objects that use them. A media player might know how to play CDs, MP3s, and wav files. Of course, you don't want to hardcode those playback algorithms into the player; that will make it difficult to add a new format like AVI. Furthermore, your code will be littered with useless case statements. And to add insult to injury, you will need to update those case statements each time you add a new algorithm. All in all, this is not a very object-oriented way to program.

With the Strategy pattern, you can simply encapsulate the algorithm behind an object. If you do that, you can provide new media plug-ins at any time. Let's call the plug-in class `MediaStrategy`. That object would have one method: `playStream(Stream s)`. So to add a new algorithm, we simply extend our algorithm class. Now, when the program encounters the new media type, it simply delegates the playing of the stream to our media strategy. Of course, you'll need some plumbing to properly instantiate the algorithm strategies you will need.
This is an excellent place to use an interface. We've used the Strategy pattern, which clearly indicates a place in the design that will change. Thus, you should define the strategy as an interface. You should generally favor interfaces over inheritance when you want an object to have a certain type; in this case, `MediaStrategy`. Relying on inheritance for type identity is dangerous; it locks you into a particular inheritance hierarchy. Java doesn't allow multiple inheritance, so you can't extend something that gives you a useful implementation or more type identity.

Interface vs. abstract class

Choosing interfaces and abstract classes is not an either/or proposition. If you need to change your design, make it an interface. However, you may have abstract classes that provide some default behavior. Abstract classes are excellent candidates inside of application frameworks.

Abstract classes let you define some behaviors; they force your subclasses to provide others. For example, if you have an application framework, an abstract class may provide default services such as event and message handling. Those services allow your application to plug in to your application framework. However, there is some application-specific functionality that only your application can perform. Such functionality might include startup and shutdown tasks, which are often application-dependent. So instead of trying to define that behavior itself, the abstract base class can declare abstract shutdown and startup methods. The base class knows that it needs those methods, but an abstract class lets your class admit that it doesn't know how to perform those actions; it only knows that it must initiate the actions. When it is time to start up, the abstract class can call the startup method. When the base class calls this method, Java calls the method defined by the child class.

Many developers forget that a class that defines an abstract method can call that method as well. Abstract classes are an excellent way to create planned inheritance hierarchies. They're also a good choice for nonleaf classes in class hierarchies.

*~~~ End of Article ~~~*

Effective Use of Abstract Classes and Interfaces

Source	http://javalive.com/modules/articles/article.php?id=17
Date of Retrieval	22/05/2007

Imagine you are the Transport Department who is responsible for the cars in the city. You come out with a specification of a car by stating its behavior and properties. I won't specify all the attributes but only two. Car manufacturers can use this specification to manufacture their own versions of cars. You define a car as:

A car is a vehicle where the driver can accelerate and where he can decelerate by braking.

To specify the car you define an interface:

```
interface Car {
    public void accelerate( double amt );
    public void brake( double amt );
    public double getSpeed();
}
```

By your interface definition, the driver can speed up the car by using the accelerate() method. The amt parameter specifies by what amount the speed has to be accelerated. Then the driver can slow down the car by using the brake() method. Again, the amt parameter specifies the amount of braking to be applied to slow down. Finally the getSpeed() method allows the driver to realize the speed at which he is driving.

After reading this specification, the manufacturer, Porsche comes out with a car that meets the specification you provided as follows:

```
class PorscheCar implements Car {
    private double _speed;           // stores the speed of the car

    public void accelerate( double amt ) {
        if( ( _speed + amt ) > 700.0 ) {
            _speed = 700.0;
            return;
        }
        _speed += amt;
    }

    public void brake( double amt ) {
        if( ( _speed - amt ) < 0.0 ) {
            _speed = 0.0;
            return;
        }
        _speed -= amt;
    }

    public double getSpeed() {
        return _speed;
    }
}
```

The car made by Porsche has a top speed of 700 mph and the car does not go beyond that. The braking also does not take place if the speed drops below 0.0 mph. Porsche meets your car specification at the moment, but they need not follow any rules. For example they could no braking to the car or no speed indication by providing an empty implementation for the brake() and getSpeed() methods. They can easily violate your specification.

When you are writing code, it is always safe NOT to trust any third-party implementation. You must always be aware that he could easily find ways to misuse your class or bypass your class's invariants to his own advantage. Hence you must take all possible precautions to prevent misuse.

So after Porsche started selling its cars, a large number of accidents are reported. All of them are due to over-speeding. To bring some sort of control and regulation you decide to extend the specifications of a car into two varieties:

1. Civilian car - must have a top speed limit of 200.0 mph.
2. Sports car - must have a top speed limit of 400.0 mph.

You also specify that both cars must have compulsory braking and must have a way for the driver to know its speed. You also may impose higher tax on sports cars and also make stringent rules about giving licenses to sports car drivers. But I am not going to get into those real-world details.

With your new specifications, you bring some invariants into the picture. The invariant for civilian cars are that their speed cannot exceed 200 mph and they must have compulsory braking and that sports car can have top speed up to 400 mph and also must have compulsory braking.

So how do u enforce these invariants on the manufacturers? Use interfaces?

No, interfaces are only contractual obligations. They do not enforce anything on the implementation. Hence Porsche or any other manufacturer still can violate your specifications. To prevent this from happening, you should use an abstract class. Therefore you specify your new civilian and sports car specifications as follows.

First you define an abstract class which enforces compulsory braking and speed indication:

```
abstract class GenericCar implements Car {
   // stores the speed, notice the use of protected modifier
   protected double _speed;

    // acceleration is still abstract
    public abstract void accelerate( double amt );

    // implement compulsary braking,
    // notice the use of final modifier to prevent
    // overriding.
    public final void brake( double amt ) {
        if( ( _speed - amt ) < 0.0 ) {
            _speed = 0.0;
            return;
        }
        _speed -= amt;
    }

    // implement compulsary speed indication
    public final double getSpeed() {
        return _speed;
    }
}
```

To analyze the GenericCar class, we enforce the compulsary speed indication and braking by providing a default, non-overridable(final) implementation of braking and speed indication. Preventing overriding is important here as we dont want manufacturers to find a loop hole in the specification.

Next, you implement your civilian car specification

```
abstract class CivilianCar extends GenericCar {
   // define a public constant for max speed
   public final static double MAX_SPEED = 200.0;

   /* implement the speed limit invariantfor acceleration to 200 mph
    * see the use of the pre-condition check for the invariant */
   public final void accelerate( double amt ) {
      // the precondition check
      if( ( _speed + amt ) > MAX_SPEED ) {
           accelerateImpl( MAX_SPEED - _speed );    // accelerate to max speed
           return;
        }
      accelerateImpl( amt );
```

```
  }

  // we want manufacturers to implement their own acceleration technique
  // hence this method
  protected abstract void accelerateImpl( double amt );
}
```

This is an interesting class. Here we are implementing the accelerate() method but only to enforce our pre-condition of speed limit. We dont want them to override the accelerate method but, we still want manufacturers to implement their own method of acceleration hence, we define the abstract accelerateImpl() method which needs to be implemented by subclasses to perform the actual acceleration.

Since the CivilianCar extends the GenericCar class, the compulsary braking and speed indication invariants are inherited and hence we dont define them again.

Now lets define our sports car specification...

```
abstract class SportsCar extends GenericCar {
    // define a public constant for max speed
    public final static double MAX_SPEED = 400.0;

    /* implement the speed limit invariantfor acceleration to 400 mph
     * see the use of the pre-condition check for the invariant */
    public final void accelerate( double amt ) {
        // the precondition check
        if( ( _speed + amt ) > MAX_SPEED ) {
          accelerateImpl( MAX_SPEED - _speed );    // accelerate to max speed
          return;
      }
        accelerateImpl( amt );
    }

    // we want manufacturers to implement their own acceleration technique
    // hence this method
    protected abstract void accelerateImpl( double amt );
}
```

The SportsCar class is very similar to our CivilianCar class, except that it increases the speed limit to 400.0 mph.

With these two specifications defined, we (i.e. the Transport Department) can now tell the public that they must specify what kind of car they wish to purchase. This choice of theirs could have a bearing on the taxation and pre-requisites to acquire the licence to drive the car.

And now our car manufactures must also choose the type of car they make. So lets redefine our Porsche car definition to suit the new specification of a sports car:

```
class NewPorscheCar extends SportsCar {

    // notice how we implement the acceleration
    public void accelerateImpl( double amt ) {
        if( ( _speed + amt ) > SportsCar.MAX_SPEED ) {
            _speed = SportsCar.MAX_SPEED;
            return;
        }
        _speed += amt;
    }
}
```

So you see how the invariants are enforced on the new NewPorscheCar. Porsche is a good manufacturer hence they have their own check for the speed limit. But a bad manufacturer who wants to take advantage of our specifications, may intentionally not do so. For example:

```
class BadSportsCar extends SportsCar {

    public void accelerateImpl( double amt ) {
        // observe there are no checks
        _speed += amt;
    }
}
```

The BadSportsCar accelerates the car without checking the speed. This attempt of the class will be defeated by its superclass, i.e. SportsCar because even if the BadSportsCar does not check the speed, the superclass will. Lets see how. I am writing a test program to test our specification:

```
public class TestCarSpecs {
    public static void main( String[] args ) {
        Car newPorscheCar = new NewPorscheCar();
        Car badSportsCar = new BadSportsCar();

        System.out.println( "Now testing NewPorscheCar..." );
        testCar( newPorscheCar );

        System.out.println();

        System.out.println( "Now testing BadSportsCar..." );
        testCar( badSportsCar );
    }

    private static void testCar( Car car ) {
        double accelerateBy = 100;
        for( int ctr = 0; ctr < 10; ctr++ ) {
            car.accelerate( accelerateBy );
            System.out.println( "Car's speed is: " + car.getSpeed() );
        }
    }
}
```

Now when you run this program, you will see that both NewPorcheCar and BadSportsCar do not exceed 400 mph as per the specifications of a sports car. Even the badly designed BadSportsCar obeys the specification speed limits. You may see an output like:

```
Quote:
Now testing NewPorscheCar...
Car's speed is: 100.0
Car's speed is: 200.0
Car's speed is: 300.0
Car's speed is: 400.0
Car's speed is: 400.0
Car's speed is: 400.0
Car's speed is: 400.0
Car's speed is: 400.0
Car's speed is: 400.0
Car's speed is: 400.0

Now testing BadSportsCar...
Car's speed is: 100.0
Car's speed is: 200.0
Car's speed is: 300.0
Car's speed is: 400.0
Car's speed is: 400.0
Car's speed is: 400.0
Car's speed is: 400.0
Car's speed is: 400.0
Car's speed is: 400.0
Car's speed is: 400.0
```

This is how we can use interfaces and abstract classes to our advantage and make safe designs possible. Here is a list recommendations I strongly suggest you consider when designing your projects:

1. Interfaces must be re-inforced by abstract classes.
2. All implementations MUST be accessed via their interfaces.
3. Special care must be taken to recognize and enforce type invariants when designing the class hierarchy.
4. Never trust third-party implementations. Always safe-guard your types.
5. Always remember to clearly document details about type invariants and how they should be enforced via JavaDocs.

*~~~ End of Article ~~~*

OOPs Principle

Source	http://www.devarticles.com/c/a/Java/An-Overview-of-Java/1/
Date of Retrieval	22/05/2007

An Overview of Java - The Three OOP Principles

All object-oriented programming languages provide mechanisms that help you implement the object-oriented model. They are encapsulation, inheritance, and polymorphism. Let's take a look at these concepts now.

Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface. To relate this to the real world, consider the automatic transmission on an automobile. It encapsulates hundreds of bits of information about your engine, such as how much you are accelerating, the pitch of the surface you are on, and the position of the shift lever. You, as the user, have only one method of affecting this complex encapsulation: by moving the gear-shift lever. You can't affect the transmission by using the turn signal or windshield wipers, for example. Thus, the gear-shift lever is a well-defined (indeed, unique) interface to the transmission. Further, what occurs inside the transmission does not affect objects outside the transmission. For example, shifting gears does not turn on the headlights! Because an automatic transmission is encapsulated, dozens of car manufacturers can implement one in any way they please. However, from the driver's point of view, they all work the same. This same idea can be applied to programming. The power of encapsulated code is that everyone knows how to access it and thus can use it regardless of the implementation details—and without fear of unexpected side effects.

In Java the basis of encapsulation is the class. Although the class will be examined in great detail later in this book, the following brief discussion will be helpful now. A *class* defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. For this reason, objects are sometimes referred to as *instances of a class.*Thus, a class is a logical construct; an object has physical reality.

When you create a class, you will specify the code and data that constitute that class. Collectively, these elements are called *members* of the class. Specifically, the data defined by the class are referred to as *member variables* or *instance variables.*The code that operates on that data is referred to as *member methods* or just *methods.* (If you are familiar with `C/C++`, it may help to know that what a Java programmer calls a *method,* a `C/C++` programmer calls a *function.*) In properly written Java programs, the methods define how the member variables can be used. This means that the behavior and interface of a class are defined by the methods that operate on its instance data.

Since the purpose of a class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class. Each method or variable in a class may be marked private or public. The *public* interface of a class represents everything that external users of the class need to know, or may know. The *private* methods and data can only be accessed by code that is a member of the class. Therefore, any other code that is not a member of the class cannot access a private method or variable. Since the private members of a class may only be accessed by other parts of your program through the class' public methods, you can ensure that no improper actions take place. Of course, this means that the public interface should be carefully designed not to expose too much of the inner workings of a class (see Figure 2-1).

Inheritance

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. As mentioned earlier, most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Golden Retriever is part of the classification *dog,* which in turn is part of the *mammal* class, which is under the larger class *animal.* Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. Let's take a closer look at this process.

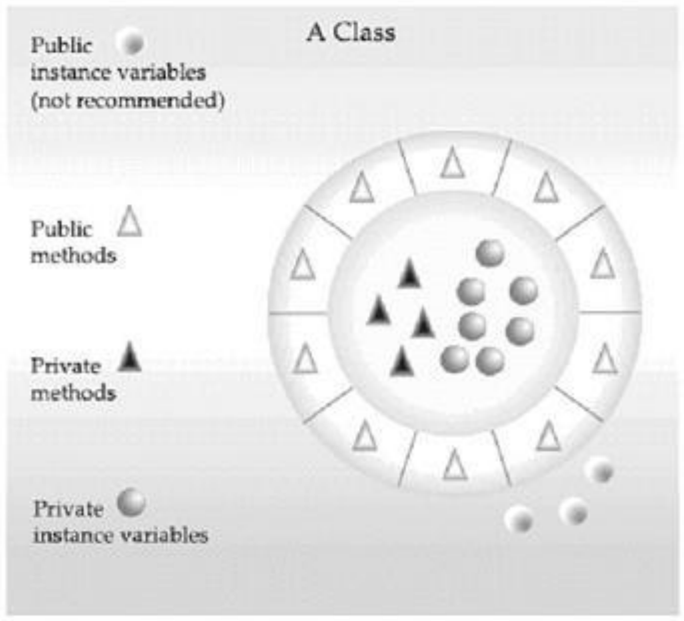
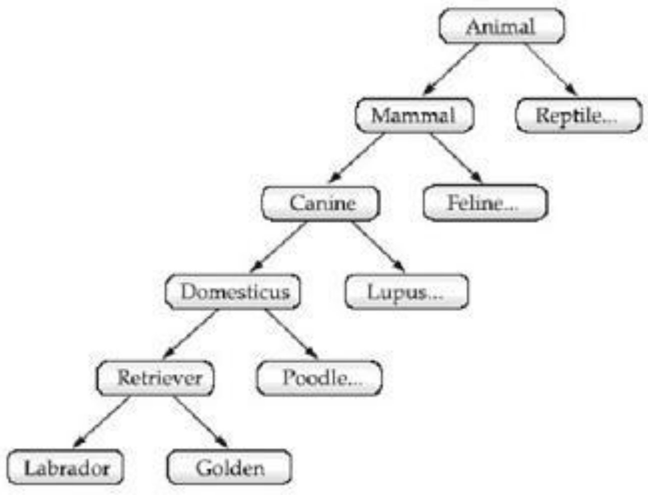

A Class
Public instance variables (not recommended)
Public methods
Private methods
Private instance variables

Figure 2-1. *Encapsulation: public methods can be used to protect private data*

Most people naturally view the world as made up of objects that are related to each other in a hierarchical way, such as animals, mammals, and dogs. If you wanted to describe animals in an abstract way, you would say they have some attributes, such as size, intelligence, and type of skeletal system. Animals also have certain behavioral aspects; they eat, breathe, and sleep. This description of attributes and behavior is the *class* definition for animals.

If you wanted to describe a more specific class of animals, such as mammals, they would have more specific attributes, such as type of teeth, and mammary glands. This is known as a *subclass* of animals, where animals are referred to as mammals' *superclass.*

Since mammals are simply more precisely specified animals, they *inherit* all of the attributes from animals. A deeply inherited subclass inherits all of the attributes from each of its ancestors in the *class hierarchy.*



Inheritance interacts with encapsulation as well. If a given class encapsulates some attributes, then any subclass will have the same attributes *plus* any that it adds as part of its specialization (see Figure 2-2). This is a key concept which lets object-oriented programs grow in complexity linearly rather than geometrically. A new subclass inherits all of the attributes of all of its ancestors. It does not have unpredictable interactions with the majority of the rest of the code in the system.

Polymorphism

Polymorphism (from the Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. Consider a stack (which is a last-in, first-out list). You might have a program that requires three types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs. In a non– object-oriented language, you would be required to create three different sets of stack routines, with each set using different names. However, because of polymorphism, in Java you can specify a general set of stack routines that all share the same names.

More generally, the concept of polymorphism is often expressed by the phrase "one interface, multiple methods." This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a *general class of action.* It is the compiler's job to select the *specific action* (that is, method) as it applies to each situation. You, the programmer, do not need to make this selection manually. You need only remember and utilize the general interface.

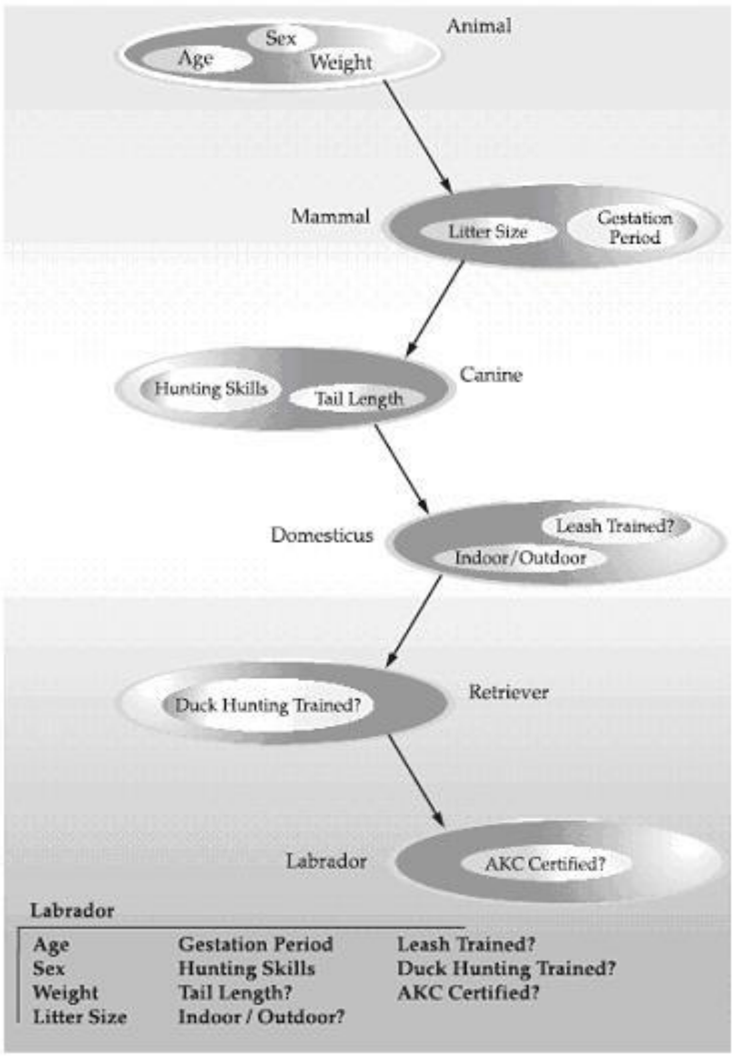


Figure 2-2. *Labrador inherits the encapsulation of all of its superclasses*

Extending the dog analogy, a dog's sense of smell is polymorphic. If the dog smells a cat, it will bark and run after it. If the dog smells its food, it will salivate and run to its bowl. The same sense of smell is at work in both situations. The difference is what is being smelled, that is, the type of data being operated upon by the dog's nose! This same general concept can be implemented in Java as it applies to methods within a Java program.

When properly applied, polymorphism, encapsulation, and inheritance combine to produce a programming environment that supports the development of far more robust and scaleable programs than does the process-oriented model. A well-designed hierarchy of classes is the basis for reusing the code in which you have invested time and effort developing and testing. Encapsulation allows you to migrate your implementations over time without breaking the code that depends on the public interface of your classes. Polymorphism allows you to create clean, sensible, readable, and resilient code.

Of the two real-world examples, the automobile more completely illustrates the power of object-oriented design. Dogs are fun to think about from an inheritance standpoint, but cars are more like programs. All drivers rely on inheritance to drive different types (subclasses) of vehicles. Whether the vehicle is a school bus, a Mercedes sedan, a Porsche, or the family minivan, drivers can all more or less find and operate the steering wheel, the brakes, and the accelerator. After a bit of gear grinding, most people can even manage the difference between a stick shift and an automatic, because they fundamentally understand their common superclass, the transmission.

People interface with encapsulated features on cars all the time. The brake and gas pedals hide an incredible array of complexity with an interface so simple you can operate them with your feet! The implementation of the engine, the style of brakes, and the size of the tires have no effect on how you interface with the class definition of the pedals.

The final attribute, polymorphism, is clearly reflected in the ability of car manufacturers to offer a wide array of options on basically the same vehicle. For example, you can get an antilock braking system or traditional brakes, power or rack-and-pinion steering, 4-, 6-, or 8-cylinder engines. Either way, you will still press the break pedal to stop, turn the steering wheel to change direction, and press the accelerator when you want to move. The same interface can be used to control a number of different implementations.

As you can see, it is through the application of encapsulation, inheritance, and polymorphism that the individual parts are transformed into the object known as a car. The same is also true of computer programs. By the application of object-oriented principles, the various parts of a complex program can be brought together to form a cohesive, robust, maintainable whole.

As mentioned at the start of this section, every Java program is object-oriented. Or, put more precisely, every Java program involves encapsulation, inheritance, and polymorphism. Although the short example programs shown in the rest of this chapter and in the next few chapters may not seem to exhibit all of these features, they are nevertheless present. As you will see, many of the features supplied by Java are part of its built-in class libraries, which do make extensive use of encapsulation, inheritance, and polymorphism.

*~~~ End of Article ~~~*

Polymorphism

Source	http://www.developer.com/tech/article.php/966001
Date of Retrieval	22/05/2007

What is polymorphism?

The meaning of the word polymorphism is something like *one name, many forms*.

How does Java implement polymorphism?

Polymorphism manifests itself in Java in the form of multiple methods having the same name.

In some cases, multiple methods have the same name, but different formal argument lists *(overloaded methods)*.

In other cases, multiple methods have the same name, same return type, and same formal argument list *(overridden methods)*.

Three distinct forms of polymorphism

From a practical programming viewpoint, polymorphism manifests itself in three distinct forms in Java:

- Method overloading
- Method overriding through inheritance
- Method overriding through the Java interface

Method overloading

I will begin the discussion of polymorphism with method overloading, which is the simpler of the three. I will cover method overloading in this lesson and will cover polymorphism based on overridden methods and interfaces in subsequent lessons.

Method overloading versus method overriding

Don't confuse method *overloading* with method *overriding*.

Java allows you to have two or more method definitions in the same scope with the same name, provided that they have different formal argument lists.

More specifically, here is what Roberts, Heller, and Ernest have to say about overloading methods in their excellent book entitled The Complete Java 2 Certification Study Guide:

"A valid overload differs in the number or type of its arguments. Differences in argument names are not significant. A different return type is permitted, but is not sufficient by itself to distinguish an overloading method."

Similarly, here is what they have to say about method overriding:
"A valid override has identical argument types and order, identical return type, and is not less accessible than the original method. The overriding method must not throw any checked exceptions that were not declared for the original method."
You should read these two descriptions carefully and make certain that you recognize the differences.

Compile-time polymorphism

Some authors refer to method overloading as a form of *compile-time polymorphism,* as distinguished from *run-time polymorphism.*

This distinction comes from the fact that, for each method invocation, the compiler determines which method *(from a group of overloaded methods)* will be executed, and this decision is made when the program is compiled. *(In contrast, I will tell you later that the determination of which overridden method to execute isn't made until runtime.)*

Selection based on the argument list

In practice, the compiler simply examines the types, number, and order of the parameters being passed in a method invocation, and selects the overloaded method having a matching formal argument list.

A sample program

I will discuss a sample program named **Poly01** to illustrate method overloading. A complete listing of the program can be viewed in Listing 4 near the end of the lesson.

Within the class and the hierarchy

Method overloading can occur both within a class definition, and vertically within the class inheritance hierarchy. *(In other words, an overloaded method can be inherited into a class that defines other overloaded versions of the method.)* The program named **Poly01** illustrates both aspects of method overloading.

Class B extends class A, which extends Object

Upon examination of the program, you will see that the class named **A** extends the class named **Object**. You will also see that the class named **B** extends the class named **A**.

The class named **Poly01** is a driver class whose **main** method exercises the methods defined in the classes named **A** and **B**.

Once again, this program is not intended to correspond to any particular real-world scenario. Rather, it is a very simple program designed specifically to illustrate method overloading.

Will discuss in fragments

As is my usual approach, I will discuss this program in fragments.

The code in Listing 1 defines the class named **A**, which explicitly extends **Object**.

```
class A extends Object{
  public void m(){
    System.out.println("m()");
  }//end method m()
}//end class A
```

Listing 1

Redundant code

Recall that explicitly extending **Object** is not required *(but it also doesn't hurt anything)*.

By default, the class named **A** would extend the class named **Object** automatically, unless the class named **A** explicitly extends some other class.

The method named m()

The code in Listing 1 defines a method named **m()**. Note that this version of the method has an empty argument list *(it doesn't receive any parameters when it is executed)*. The behavior of the method is simply to display a message indicating that it has been invoked.

The class named B

Listing 2 contains the definition for the class named **B**. The class named **B** extends the class named **A**, and inherits the method named **m** defined in the class named **A**.

```
class B extends A{
  public void m(int x){
    System.out.println("m(int x)");
  }//end method m(int x)
  //--------------------------------//

  public void m(String y){
    System.out.println("m(String y)");
  }//end method m(String y)
}//end class B
```

Listing 2

Overloaded methods

In addition to the inherited method named **m**, the class named **B** defines two overloaded versions of the method named **m**:

- m(int x)

- m(String y)

(Note that each of these two versions of the method receives a single parameter, and the type of the parameter is different in each case.)

As with the version of the method having the same name defined in the class named **A**, the behavior of each of these two methods is simply to display a message indicating that it has been invoked.

The driver class

Listing 3 contains the definition of the driver class named **Poly01**.

```
public class Poly01{
  public static void main(
                      String[] args){
    B var = new B();
    var.m();
    var.m(3);
    var.m("String");
  }//end main
}//end class Poly01

Listing 3
```

Invoke all three overloaded methods

The code in the **main** method

- Instantiates a new object of the class named **B**
- Successively invokes each of the three overloaded versions of the method named **m** on the reference to that object.

One version is inherited

Note that the overloaded version of the method named **m**, defined in the class named **A**, is inherited into the class named **B**. Hence, it can be invoked on a reference to an object instantiated from the class named **B**.

Two versions defined in class B

The other two versions of the method named **m** are defined in the class named **B**. Thus, they also can be invoked on a reference to an object instantiated from the class named **B**.

The output

As you would expect, the output produced by sending messages to the object asking it to execute each of the three overloaded versions of the method named **m** is:

```
m()
m(int x)
m(String y)
```

Note that the values of the parameters passed to the methods do not appear in the output. Rather, the parameters are used solely to make it possible for the compiler to select the correct version of the overloaded method to execute in each case.

This output confirms that each overloaded version of the method is properly selected for execution based on the matching of method parameters to the formal argument list of each method.

*~~~ End of Article ~~~*

Session 8: More on Classes

Inner Classes

Source	http://www.javaranch.com/campfire/StoryInner.jsp
Date of Retrieval	22/05/2007

One Object shares deeply personal feelings. Next... on a very, special, Campfire Story.

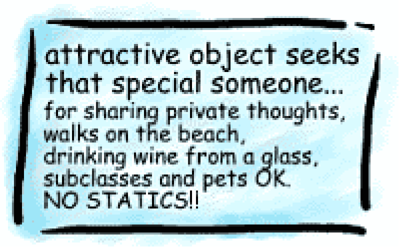
Being an object like me isn't as fun as you might think.

It gets lonely...

Out here...
on the heap...
alone.

Not to mention the horror, the emotional devastation when you feel your last reference slip away and it hits you -- you've just become food for the garbage collector.

But you know what helps? Having an inner class. An inner class can ease the loneliness... as long as somebody makes an *instance* of that inner class. All I really want is someone to bond with.


attractive object seeks that special someone...
for sharing private thoughts,
walks on the beach,
drinking wine from a glass,
subclasses and pets OK.
NO STATICS!!

Someone to **share my most private thoughts** (and variables and methods). Someone who knows EVERYTHING about me. An intimate relationship shared between two objects -- an outer and an inner.

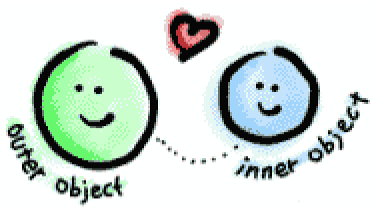
I'm very protective of my inner class. If somebody wants to instantiate my inner class, they MUST go through me-- an object of the outer class.

My inner class can't exist on its own. I, as an instance of the outer class, can live on my own (however unhappily). You don't have to make an instance of an inner class in order to have an instance of the outer class. But you can NEVER make an instance of my inner class without an outer object to "bind" it to.

My inner class needs me.

We have that special bond.

It makes life on the garbage-collectible heap bearable.



Here's how to do a little object matchmaking of your own:

```
class Outer
{
    private int size ;
    private String thoughts = "My outer thoughts";

    class Inner
    {
        String innerThoughts = "My inner thoughts";

        void doStuff()
        {
            // inner object has its own "this"
            System.out.println( innerThoughts );

            // and it also has a kind of "outer this"
            // even for private data of outer class
            System.out.println(thoughts);
        }
    }
}
```

OK, but nothing really happens until somebody makes an instance of BOTH classes...

```
class TestMe
{

    public static void main( String args[] )
    {
        // instantiate me, the outer object
        Outer o = new Outer();

        // Inner i = new Inner();
        // NO! Can't instantiate Inner by itself!

        Outer.Inner i = o.new Inner();
        // now I have my special inner object
        i.doStuff();
        // OK to call methods on inner object
    }

}
```

You can also instantiate both the outer class and inner class at the same time:

```
  Inner i = new Outer().new Inner();
```

I know that looks odd, but it shows that you need an outer object, so that you can ask it to make an inner object. In this example, you didn't even keep a reference to the outer object... only the inner object, "i". The inner object "i" still knows its outer object... its "outer this". (By the way, there is no keyword "outer this" -- that's just a concept for the way inner objects behave. They behave as if the outer object's variables were their own.)

I hate static!

You've probably heard about **static inner classes**. Well, they don't deserve to be called inner classes!

A static inner class (an inner class marked as static) looks like this:

```
class Outer
{
    static class Inner
    {
    }
}
```

I don't like them because they don't give me that special object-to-object bond. In fact, static inner classes aren't even supposed to be called inner classes at all. Technically, they are "**top-level nested classes**".

A static nested class can be instantiated, but the object created doesn't share any special relationship with an outer object.

The static nested class is tied only to the outer **class, *not an instance*** of the outer class.

```
     Outer.Inner i = new Outer.Inner();
```

That's why you can make an instance of the static nested class *without* having an instance of the outer class, just the way you can call static methods of a class without having any instances of that class. A top-level nested class is little more than another way to control namespace.

But let's go back to inner classes; they're so much more meaningful. And did you know that I can bond with an instance of my inner class even when I don't know the NAME of my inner class? For convenience, you can get an **instance** of an inner class and **make that inner class** at the **same time.**

It works like this...

Imagine you (the programmer) are making your nice GUI program and you decide that you need to know when the user clicks your GO button. "I reckon I need an ActionListener object", you say to yourself. So you type:

```
 goButton.addActionListener([object goes here]);
```

And then you slap your forehead as you realize... "I can't make an **instance**... I don't even HAVE an ActionListener **class**!"

You never made yourself a class that implements the ActionListener interface.

Not a problem.

You can create a new class which implements the ActionListener interface, AND make an instance of that new class -- **all inside the parameter to the Button object's addActionListener() method.**
How cool is that?

It looks like this:

```
goButton.addActionListener
(
    new ActionListener()
    {
        public void actionPerformed( ActionEvent e )
        {
            doImportantStuff();
        }
    }
);
```

It works like this:

```
  new ActionListener()
```

says to the compiler: "Create an instance of a new, unnamed class which **implements the ActionListener interface**..."

And after that opening curly brace (shown above in green) you define the new unnamed class...

```
public void actionPerformed( actionEvent e )
{
    doImportantStuff();
}
```

That actionPerformed method is the same one you would be forced to define in any class which implements the ActionListener interface. But this new class has no name. That's why its called an **anonymous** inner class.

And notice that you did not say "new MyActionClass()". You said, "new ActionListener()". But you aren't making an instance of ActionListener, you're making an instance of your new anonymous class which implements the ActionListener interface.

"But wait!" you say, "What if I don't want to implement an interface... what if I want to make an anonymous inner class that **extends** another class?"

Once again, No Problem-o.

Whatever you say after the "new" as in "new Something()", if Something is an interface, then the anonymous class implements that interface (and must define all the methods of that interface). But if Something is a class, then your anonymous class automatically becomes a subclass of that class. This is perfect for the event adapter classes like WindowAdapter.

Finally, don't forget to close the parameter to the `goButton.addActionListener(`

by finishing off with a closing parentheses and the semicolon ending that statement...

```
   );
```

Programmers always seem to forget that last little semicolon way down there, since the statement started waaaaaay up above somewhere.

Danger Danger!

Now I feel compelled to warn you about one thing with anonymous inner classes, or any other inner class that you define **inside** a method (as opposed to inside the class but NOT inside a method). **The inner class can't use local variables from the method in which the inner class**

is defined!
After all, at the end of that method, the local variables will be blown away. Poof. Gone. History. That inner object you created from that inner class might still be alive and kickin' out on the heap long after that local variable has gone out of scope.
You can, however, use local variables that are declared final, because the compiler takes care of that for you in advance. But that's it -- no method parameters from that method, and no local variables.

Another warning about all inner classes is that they can't declare any static members unless they are compile-time constants and are primitives or Strings. (This does not apply to static nested classes, of course). But don't worry -- the compiler will stop you if you try.

Finally, you should know that some programmers really dislike inner classes, and especially anonymous ones. Some folks claim they're not very object-oriented, and others get their feathers ruffled over "encapsulation violations" because the inner object can access the private data of the outer object.

Well you know what I say?
That's the point!

It's that special relationship, that intimate bond, that makes inner classes so practical. Otherwise you'd have to make constructors for your inner class, and pass references to variables, etc. when you instantiate the inner class... all the things you have to do with a plain old non-inner class, which you have to treat like an outsider.

But when you're trying to decide if inner classes are right for you, please... think of me...

the poor, lonely, object...

on the heap...

alone .

*~~~ End of Article ~~~*

Static/Class Methods

Source	http://www.leepoint.net/notes-java/flow/methods/50static-methods.html
Date of Retrieval	22/05/2007

There are two types of methods.

- **Instance methods** are associated with an object and use the instance variables of that object. This is the default.
- **Static methods** use no instance variables of any object of the class they are defined in. If you define a method to be static, you will be given a rude message by the compiler if you try to access any instance variables. You can access static variables, but except for constants, this is unusual. Static methods typically take all they data from parameters and compute something from those parameters, with no reference to variables. This is typical of methods which do some kind of generic calculation. A good example of this are the many utility methods in the predefined `Math` class. (See Math and java.util.Random).

Qualifying a static call

From outside the defining class, an instance method is called by prefixing it with an *object*, which is then passed as an implicit parameter to the instance method, eg, `inputTF.setText("");`

A static method is called by prefixing it with a *class name*, eg, `Math.max(i,j);`. Curiously, it can also be qualified with an object, which will be ignored, but the class of the object will be used.

Example

Here is a typical static method.

```
class MyUtils {
    . . .
    //================================================= mean
    public static double mean(int[] p) {
        int sum = 0;  // sum of all the elements
        for (int i=0; i<p.length; i++) {
            sum += p[i];
        }
        return ((double)sum) / p.length;
    }//endmethod mean
    . . .
}
```

The only data this method uses or changes is from parameters (or local variables of course).

Why declare a method `static`

The above `mean()` method would work just as well if it wasn't declared `static`, as long as it was called from within the same class. If called from outside the class and it wasn't declared static, it would have to be qualified (uselessly) with an object. Even when used within the class, there are good reasons to define a method as static when it could be.

- **Documentation**. Anyone seeing that a method is static will know how to call it (see below). Similarly, any programmer looking at the code will know that a `static` method can't interact with instance variables, which makes reading and debugging easier.
- **Efficiency**. A compiler will usually produce slightly more efficient code because no implicit object parameter has to be passed to the method.

Calling static methods

There are two cases.

Called from within the same class

Just write the static method name. Eg,

```
// Called from inside the MyUtils class
double avgAtt = mean(attendance);
```

Called from outside the class

If a method (static or instance) is called from another class, something must be given before the method name to specify the class where the method is defined. For instance methods, this is the object that the method will access. For static methods, the class name should be specified. Eg,

```
// Called from outside the MyUtils class.
double avgAtt = MyUtils.mean(attendance);
```

If an object is specified before it, the object value will be ignored and the the class of the object will be used.

Accessing static variables

Altho a `static` method can't access instance variables, it can access `static` variables. A common use of `static` variables is to define "constants". Examples from the Java library are `Math.PI` or `Color.RED`. They are qualified with the class name, so you know they are `static`. Any method, `static` or not, can access static variables. Instance variables can be accessed only by instance methods.

*~~~ End of Article ~~~*

Session 9: Exceptions

Exceptions

Source	http://www.cs.wisc.edu/~cs302/io/Exceptions.html
Date of Retrieval	22/05/2007

Introduction

Exceptions are *objects* that store information about the occurrence of an unusual or error condition. They are *thrown* when that error or unusual condition occurs. You have likely experienced the occurrence of an exception occasionally throughout your programming in this course. Examples include: **`NullPointerException`** and **`ArithmeticException`**.

Until now, any exception that occurred *always* resulted in a program crash. However, Java and other programming languages have mechanisms for handling exceptions that you can use to keep your program from crashing. In Java, this is known as *catching an exception*.

If any exception occurs and is not caught, the program will crash.

There are two types of exceptions in Java, *unchecked* exceptions and *checked* exceptions.

Unchecked Exceptions

Unchecked exceptions are any class of exception that extends the **`RuntimeException`** class at some point in its inheritance hierarchy. This includes the most common exceptions. An **`ArithmeticException`** occurs when a program tries to divide by zero. A **`NullPointerException`** occurs when you try and access an instance data member or method of a reference variable that does not yet reference an object.

Unchecked exceptions can occur anywhere in a program and in a typical program can be very numerous. Therefore, the cost of checking these exceptions can be greater than the benefit of handling them. Thus, Java compilers do not require that you declare or catch unchecked exceptions in your program code. Unchecked exceptions may be handled as explained for *checked* exceptions in the following section.

All Exceptions that extend the RuntimeException class are unchecked exceptions.

Checked Exceptions

Checked exceptions are exceptions that do *not* extend the **`RuntimeException`** class. Checked exceptions *must* be *handled* by the programmer to avoid a compile-time error. One example of a checked exception is the **`IOException`** that may occur when the ***`readLine`*** method is called on a

`BufferedReader` object. Read more about the ***`readLine`*** method in the section on console input in the Java I/O page. All other exceptions you may have experienced were examples of *unchecked* exceptions.

Handling Checked Exceptions

There are two ways to *handle* checked exceptions. You may declare the exception using a *throws clause* or you may *catch the exception*. To declare an exception, you add the keyword *throws* followed by the class name of the exception to the method header. See example below. Any method that calls a method that *throws* a checked exception must also *handle* the checked exception in one of these two ways.

In the example below, either call to the ***`readLine`*** method may cause an **`IOException`** to occur. Each method of handling the checked exception is shown.

The throws Clause

The **throws IOException** clause in the method header tells the compiler that we know this exception may occur and if it does, the exception should be thrown to the caller of this method instead of crashing the program. The *throws clause* is placed after the parameter list and before the opening brace of the method. If more than one type of checked exception needs to be declared, separate the class names with commas.

```
/**
 *  Fills a Data object with Strings of data that
 *  are read from a BufferedReader object.
 */
void fillData ( BufferedReader in, Data data )
    throws IOException
{
    String newData = in.readLine();
    while ( newData != null )
    {
        data.addData( newData );
        newData = in.readLine();
    }
}
```

If an exception is always handled using the throws clause, then eventually the exception will propagate all the way back to the ***`main`*** method of the application. If the ***`main`*** method *throws the exception* and the exception actually occurs while the program is running, then the program will crash.

The try-catch Statement

An alternative to simply passing the exception back to the caller of your method, is to *catch* the exception yourself. In this case, your method will do something to correct the error or recover and continue the program's execution, if the exception occurs. To *catch* an exception, you must first indicate what code may cause the exception. This is called a *try* block. Together, they are referred to as a **try-catch** statement.

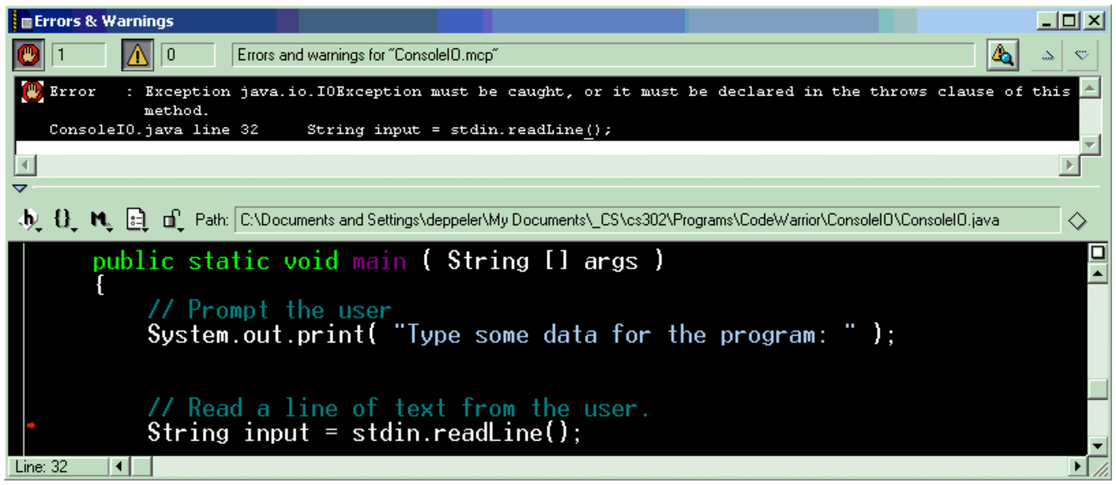
The *try* block surrounds the statement and any related statements that may cause the exception to occur. The *catch* block follows the try block and defines the code that should be executed, if the exception occurs. If more than one type of checked exception needs to be caught, write a separate catch clause for each type. See How to Catch Multiple Types of Exceptions.

```
/**
 *  Fills a Data object with Strings of data that
 *  are read from a BufferedReader object.
 */
void fillData ( BufferedReader in, Data data )
{
    try
    {
        String newData = in.readLine();
        while ( newData.equals("") )
        {
            data.addData( newData );
            newData = in.readLine();
        }
    }
    catch ( IOException e )
    {
      System.out.println(e);
      System.out.println("Unable to finish adding data.");
    }
}
```

All Exceptions that DO NOT extend the **`RuntimeException`** *class are checked exceptions.* The compiler requires a throws clause or a try-catch statement for any call to a method that may cause a checked exception to occur. *If a checked exception occurs and is not caught before it reaches the* **`main`** *method of the application, the program will crash.*

How do I know if it is a checked exception or an unchecked exception?

Experienced programmers seem to just know which exceptions are checked. But, there are two ways that novice programmers can gain this experience. One way that we learn which exceptions are *checked*, is from the compiler. The compiler will produce an error when we call a method that may throw *checked* exceptions. For example, our Console IO program from the Java I/O page will produce the following compile-time error message, if we forget the *throws IOException* clause in the method header.


Errors & Warnings
1
0
Errors and warnings for "ConsoleIO.mcp"
Path: C:\Documents and Settings\deppeler\My Documents_CS\cs302\Programs\CodeWarrior\ConsoleIO\ConsoleIO.java
Line: 32

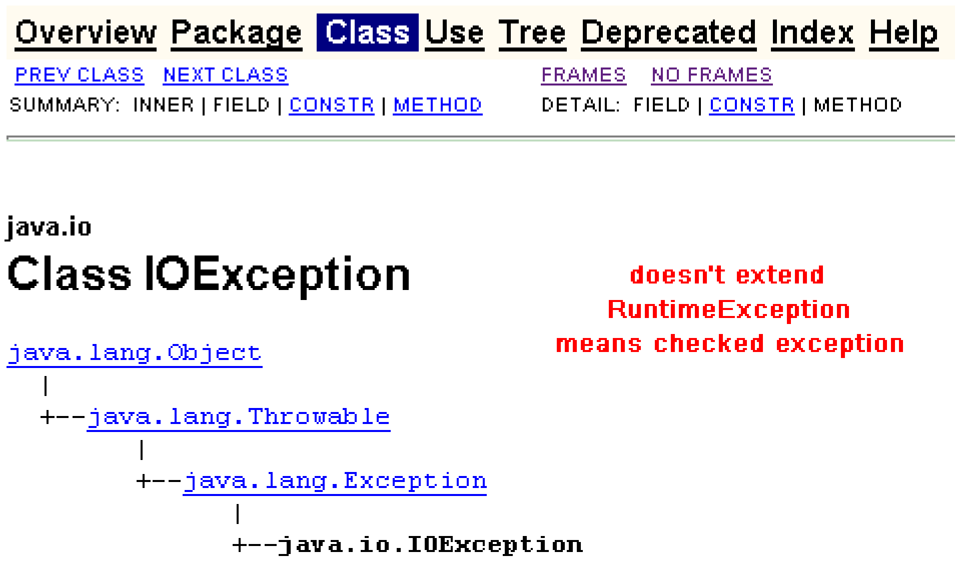
```
Error   : Exception java.io.IOException must be caught, or it must be declared in the throws clause of this
          method.
ConsoleIO.java line 32     String input = stdin.readLine();
```

```
public static void main ( String [] args )
{
    // Prompt the user
    System.out.print( "Type some data for the program: " );


    // Read a line of text from the user.
    String input = stdin.readLine();
```

Thus, an **`IOException`** is a checked exception and we now know that the ***`readLine`*** method of the **`BufferedReader`** class may throw such exceptions. From now on, we will always *catch* or include the *throws IOException* clause in the method header of any method that calls the ***`readLine`*** method. This also means that any method that calls any of our methods that *throws IOException* will have to handle the exception.

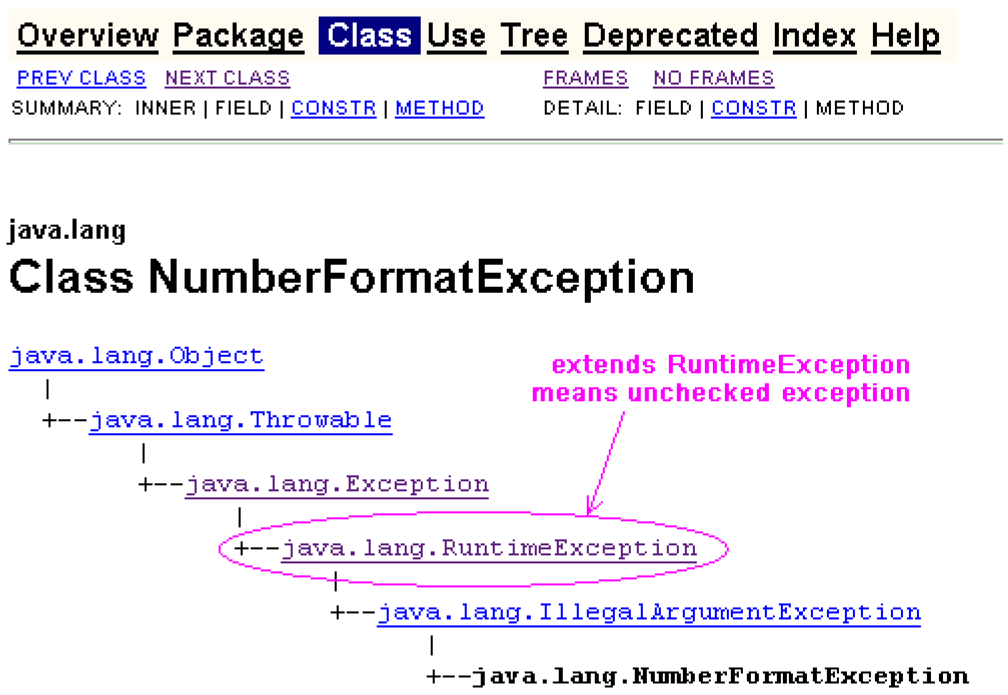
A second way we can learn which exceptions are checked exceptions is to view the inheritance hierarchy for the exception class of interest. Use your browser to view the javadoc pages for the **`BufferedReader`** class and navigate to the method detail documentation for the ***`readLine`*** method. The method summary information doesn't show, but the method detail information shows that the ***`readLine`*** method *throws IOException*. This information tells you what exceptions may be thrown by the ***`readLine`*** method. But, is the **`IOException`** a checked exception? Click on the link to **`IOException`** to find out for sure. You will see the following, which shows that the **`IOException`** class extends the **`Exception`** class, but it does not extend the **`RuntimeException`** class. Since, checked exceptions are those types of exceptions that do not extend the **`RuntimeException`** class, we know this is a checked exception.


Overview Package Class Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS
FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD
doesn't extend RuntimeException means checked exception

java.io
Class IOException

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--java.io.IOException
```

Compare and contrast the inheritance hierarchy of the **`IOException`** class (shown above) with that of the **`NumberFormatException`** class (shown below). Notice, that the **`NumberFormatException`** class does have the **`RuntimeException`** class in its inheritance hierarchy and is thus an unchecked exception.



Overview Package **Class** Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

java.lang
Class NumberFormatException

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--java.lang.RuntimeException
                    |
                    +--java.lang.IllegalArgumentException
                          |
                          +--java.lang.NumberFormatException
```

Viewing the javadoc pages will help you identify if a method may cause a *checked* exception. But, there are many *unchecked* exceptions that may occur that are not documented in any method header. They may not be easy to find. Some of the most common unchecked exceptions are: **`ArithmeticException: Divide By Zero`** and **`NullPointerException`**. It is up to the programmer to understand when these type of exceptions can occur and to employ defensive programming to ensure that they do not occur.

When you catch an exception, you [the programmer] have the ability to perform some other action including *fixing* the error condition before continuing with the program's execution. This opportunity to resolve the problem means that the program can continue to perform the operations for which it was intended. However, there are some exceptions that you will not be able to resolve and catching those exceptions will give you the opportunity to save some current state of execution and report the error to the user before exiting the program gracefully. You can always choose not to catch an exception. If that exception occurs, the program will crash.

How to catch a NumberFormatException

To **catch** a **`NumberFormatException`**, put a *try* statement block around the code that may cause the exception and a *catch* clause after the try block that describes what to do if the exception occurs. Here's a code fragment that shows how to catch a **`NumberFormatException`** that may occur when the ***`parseInt`*** method is called on a **`String`**:

```
System.out.print( "Enter an integer: " );
String input = stdin.readLine();

try
{
    int x = Integer.parseInt( input );
}
catch ( NumberFormatException e )
```

```
    {
        System.out.println( e.getMessage()
            + " is not a valid format for an integer." );
    }
```

In this example, the user is prompted for an integer. The program then *tries* to parse that string into an `int` value. If the string of characters has a decimal point or letters or some other invalid `int` format, then a **`NumberFormatException`** is thrown. If that happens, the flow of control transfers to the `catch ( NumberFormatException e ) {...}` block of code. In this example, a message is displayed to the user and the program then continues after the catch block without crashing.

The code fragment above will keep your program from crashing if the user enters an invalid `int` format, but it does not give the user another chance to enter a valid integer format. A repetition statement is necessary, if you wish to repeat the prompt and let the user have another chance to enter a valid integer.

A **while loop** that repeats until a valid integer is entered is a good choice. The entire *try-catch* statement must be nested inside the loop. The preferred way to control this type of loop is with a `boolean` variable that is set true when a valid integer has been entered. Using a specific integer as a sentinel value for ending the while loop is only okay if there is an integer that can be used that is not otherwise a valid choice for the user.

```
    // Code fragment to prompt a user for an integer
    // and store that integer in a variable x that can be
    // used after the loop has gotten a valid integer from the user.

    // used to mark when a valid integer has been entered
    boolean valid = false;

    // x will hold the integer entered by the user
    // It can be initialized to anything, since it will be overwritten
    int x = -1;

    // Get a valid integer from the user and put into x
    while ( ! valid )
    {
        try
        {
            System.out.print( "Enter an integer: " );

            x = Integer.parseInt( stdin.readLine() );

            valid = true;    // this statement only executes if the string
                             // entered does not cause a NumberFormatException
        }
        catch ( NumberFormatException e )
        {
            // A NumberFormatException occurred, print an error message
            System.out.println( e.getMessage()
                + " is not a valid format for an integer." );
        }

    } // repeats until a valid integer has been entered
```

Notes: The declaration of `valid` must precede its use inside the loop condition expression. The declaration of `x` must also be outside of the loop. This is so that the value of `x` can be used after the loop has finished executing. This is known as the *scope* of the variable. If a variable is declared inside of a statement block, then it will only have *scope (be visible)* to code that is also inside of the block and after the declaration. The statement to set the `boolean` variable named `valid` to `true` must also be inside of the `try` block. If this statement were placed after the try-catch statement, then it would be executed even if a **`NumberFormatException`** occurred.

How to Catch Multiple Types of Exceptions

You can catch more than one type of exception after the same try block. However, the order that the exceptions are listed in the catch clauses is sometimes important. More specific classes of exceptions must be listed before more general classes that could catch the same exception. Here is a code fragment for opening a file for reading.

```
 try
 {
     // Prompt the user and read a file name
     System.out.print( "Type a file name: " );
     String fileName = stdin.readLine();

      // Create a FileReader object for the specified file
     FileReader fileReader = new FileReader( fileName );

     // Create a BufferedReader object
     BufferedReader inFile = new BufferedReader( fileReader );

     // Intialize line number and read the first line of the file
     int lineNo = 0;
     String line = inFile.readLine();

     // Read entire file line-by-line and echo to screen with line numbers
     while ( line != null )
     {
         System.out.println( ++lineNo + "\t" + line );
         line = inFile.readLine();
     }

     // Close the file reader for safety
     fileReader.close();
}
catch ( FileNotFoundException e )
{
     System.out.println( e.getMessage() + " FILE NOT FOUND" );
}
catch ( IOException e )
{
     System.out.println( e + " IO EXCEPTION" );
}
catch ( Exception e )
{
     System.out.println( e + " EXCEPTION" );
}
```

The class **`Exception`** is the most general type of exception. The **`FileNotFoundException`** class is a more specific type of exception and will not catch *all* exceptions that could occur. In the example above, exceptions that are not caught by the **`FileNotFoundException`** catch clause will fall through and possibly be caught by the catch clause for the **`IOException`** class. It they are not caught by the **`IOException`** class, they will be caught by the catch clause for the **`Exception`** class.

It is a compile-time error to place a more general exception clause (for a *super*class) before a more specific exception clause (for a *sub*class). Use the inheritance tree provided by the javadoc to determine which exception class is more specific. The classes that are lower on the inheritance hierarchy are *sub*classes and are more specific than those classes listed higher in the inheritance hierarchy. See the NumberFormatException hierarchy figure and notice that **`NumberFormatException`** is a subclass of **`IllegalArgumentException`**, which is a subclass of **`RuntimeException`** and so on.

Finally

After the *try-catch* statement, you may optionally include a *finally* clause. A *finally* clause is used to perform operations that must happen whether or not an exception occurs in the *try* block. If a *finally* clause is used, it must follow all *catch* clauses for a given *try* block. There can only be one *finally* clause for a given *try-catch* statement block.

```
try
{
    ...
}
catch ( IOException e )
{
    ...
}
catch ( Exception e )
{
    ...
}
finally
{
    // any statements that you always want to occur
}
```

Conclusion

Exceptions are a good way to separate the code that handles errors or very special cases from the code that is most commonly executed. This can help the programmer focus on the intended purpose of the code without having to write many nested control statements to verify information at each step of execution. Read Chapter 8, "Exceptions and Assertions" of Wu's "An Introduction to Object-Oriented Programming with Java" textbook for more information about handling exceptions in Java.

*~~~ End of Article ~~~*

Java Exceptions

Source	http://www.smartdataprocessing.com/lessons/l5.htm
Date of Retrieval	22/05/2007

Let's say a Java class reads a file with the customer's data. What's going to happen if someone deletes this file?

Will the program crash with that scary multi-line error message, or will it stay alive displaying a user friendly message like this one: "Dear friend, for some reason I could not read the file customer.txt. Please make sure that the file exists"? In many programming languages, error processing depends on the skills of a programmer.

Java forces software developers to include error processing code, otherwise the programs will not even compile.

Error processing in the Java is called exception handling.

You have to place code that may produce errors in a so-called try/catch block:

```
try{
  fileCustomer.read();
  process(fileCustomer);
}
catch (IOException e){
  System.out.println("Dear friend, I could not read the file
customer.txt...");
}
```

In case of an error, the method read() throws an exception. In this example the catch clause receives the instance of the class IOException that contains information about input/output errors that have occured. If the catch block exists for this type of error, the exception will be caught and the statements located in a catch block will be executed. The program will not terminate and this exception is considered to be taken care of.

The print statement from the code above will be executed only in case of the file read error. Please note that method process() will not even be called if the read fails.

Reading the Stack Trace

If an unexpected exception occurs that's not handled in the code, the program may print multiple error messages on the screen. Theses messages will help you to trace all method calls that lead to this error. This printout is called a stack trace. If a program performs a number of nested method calls to reach the problematic line, a stack trace can help you trace the workflow of the program, and localize the error.

Let's write a program that will intentionally divide by zero:

```
class TestStackTrace{
  TestStackTrace()
  {
    divideByZero();
  }

  int divideByZero()
  {
    return 25/0;
  }

  static void main(String[]args)
  {
    new TestStackTrace();
  }
}
```

Below is an output of the program - it traced what happened in the program stack before the error had occurred. Start reading it from the last line going up. It reads that the program was executing methods main(), init() (constructor), and divideByZero(). The line numbers 14, 4 and 9 (see below) indicate where in the program these methods were called. After that, the ArithmeticException had been thrown - the line number nine tried to divide by zero.

```
c:\temp>java TestStackTrace
  Exception in thread "main"
  java.lang.ArithmeticException: / by zero
     at TestStackTrace.divideByZero(TestStackTrace.java:9)
     at TestStackTrace.(TestStackTrace.java:4)
     at TestStackTrace.main(TestStackTrace.java:14)
```

Exception Hierarchy

All exceptions in Java are classes implicitely derived from the class Throwable which has immediate subclasses

Error and Exception.

Subclasses of the class Exception are called listed exceptions and have to be handled in your code.

Subclasses of the class Error are fatal JVM errors and your program can't fix them. Programmers also can define their own exceptions.

How you are supposed to know in advance if some Java method may throw a particular exception and the try/catch block should be used? Don't worry - if a method throws an exception and you did not put this method call in a try/catch block, the Java compiler will print an error message similar to this one:

"Tax.java": unreported exception: java.io.IOException; must be caught or declared to be thrown at line 57.

Try/Catch Block

There are five Java keywords that could be used for exceptions handling: try, catch, finally, throw, and throws.

By placing a code in a try/catch block, a program says to a JVM: "Try to execute this line of code, and if something goes wrong, and this method throws exceptions, please catch them, so that I could report this situation to a user." One try block can have multiple catch blocks, if more than one problem occurs. For example, when a program tries to read a file, the file may not be there - FileNotFoundException, or it's there, but the code has reached the end of the file - EOFException, etc.

```
public void getCustomers(){
  try{
    fileCustomers.read();
  }catch(FileNotFoundException e){
    System.out.println("Can not find file Customers");
  }catch(EOFException e1){
    System.out.println("Done with file read");
  }catch(IOException e2){
    System.out.println("Problem reading  file " +
                                 e2.getMessage());
  }
}
```

If multiple catch blocks are handling exceptions that have a subclass-superclass relationship (i.e. EOFException is a subclass of the IOException), you have to put the catch block for the subclass first as shown in the previous example.

A lazy programmer would not bother with catching multiple exception, but will rather write the above code like this:

```
public void getCustomers(){
  try{
    fileCustomers.read();
  }catch(Exception e){
    System.out.println("Problem reading  file " +
                        e.getMessage());
  }
}
```

Catch blocks receive an instance of the Exception object that contains a short explanation of a problem, and its method getMessage() will return this info. If the description of an error returned by the getMessage() is not clear, try the method toString() instead.

If you need more detailed information about the exception, use the method printStackTrace(). It will print all internal method calls that lead to this exception (see the section "Reading Stack Trace" above).

Clause throws

In some cases, it makes more sense to handle an exception not in the method where it happened, but in the calling method. Let's use the same example that reads a file. Since the method read() may throw an IOException, you should either handle it or declare it:

```
class CustomerList{
  void getAllCustomers() throws IOException{
    file.read(); // Do not use try/catch  if you are not handling exceptions
here
  }

  public static void main(String[] args){
    System.out.println("Customer List");

    try{
      // Since the  getAllCustomers()declared exception,
      // we have to either  handle  it over here, or re-
      // throw it (see the throw clause explanation below)

     getAllCustomers();
   }catch(IOException e){
     System.out.println("Sorry, the  Customer List is not
                                          available");
  }
}
```

In this case, the IOException has been propagated from the getAllCustomers() to the main() method.

Clause finally

A try/catch block could be completed in different ways

1. the code inside the try block successfully ended and the program continues,
2. the code inside the try block ran into a return statement and the method is exited,
3. an exception has been thrown and code goes to the catch block, which handles it
4. an exception has been thrown and code goes to the catch block, which throws another exception to the calling method.

If there is a piece of code that must be executed regardless of the success or failure of the code, put it under the clause finally:

```
try{
  file.read();
}catch(Exception e){
  printStackTrace();
}
finally{
 file.close();
}
```

The code above will definitely close the file regardless of the success of the read operation. The finally clause is usually used for the cleanup/release of the system resources such as files, database connections, etc..

If you are not planning to handle exceptions in the current method, they will be propagated to the calling method. In this case, you can use the finally clause without the catch clause:

```
void myMethod () throws IOException{
  try{
    file.read();
  }
  finally{
    file.close();
  }
}
```

Clause throw

If an exception has occurred in a method, you may want to catch it and re-throw it to the calling method. Sometimes, you might want to catch one exception but re-throw another one that has a different description of the error (see the code snippet below).

The throw statement is used to throw Java objects. The object that a program throws must be Throwable (you can throw a ball, but you can't throw a grand piano). This technically means that you can only throw subclasses of the Throwable class, and all Java exceptions are its subclasses:

```
class CustomerList{

  void getAllCustomers() throws Exception{
    try{
      file.read(); // this line may throw an exception
    } catch (IOException e) {
      // Perform some internal processing of this error, and _
      throw new  Exception (
        "Dear Friend, the file has problems..."+
                               e.getMessage());
      }
    }

    public static void main(String[] args){
      System.out.println("Customer List");

      try{
        // Since the  getAllCustomers() declares an
        // exception, wehave  to either  handle  it, or re-throw it
         getAllCustomers();
      }catch(Exception e){
        System.out.println(e.getMessage());
      }
   }
}
```

User-Defined Exceptions

Programmers could also create user-defined exceptions, specific to their business. Let's say you are in business selling bikes and need to validate a customers order. Create a new class TooManyBikesException that is derived from the class Exception or Throwable, and if someone tries to order more bikes than you can ship - just throw this exception:

```
class TooManyBikesException extends Exception{
  TooManyBikesException (String msgText){
     super(msgText);
  }
}

class BikeOrder{
  static  void validateOrder(String bikeModel,int quantity) throws
TooManyBikesException{

  // perform  some data validation, and if you do not like
  // the quantity for the specified model, do  the following:

  throw new TooManyBikesException("Can not ship" +
    quantity+" bikes of the model " + bikeModel +);
  }
}

class Order{
   try{
     bikeOrder.validateOrder("Model-123", 50);

     // the next line will be skipped in case of an exception
     System.out.println("The order is valid");
   } catch(TooManyBikes e){
      txtResult.setText(e.getMessage());
   }
}
```

In general, to make your programs robust and easy to debug, you should always use the exception mechanism to report and handle exceptional situations in your program. Be specific when writing your catch clauses - catch as many exceptional situations as you can predict. Just having one catch (Exception e) statements is not a good idea.

*~~~ End of Article ~~~*