**KABARAK UNIVERSITY**

**COMP 404: OBJECT ORIENTED PROGRAMMING WITH JAVA**

**STUDENT MANUAL**

**COMPILED BY: MR. MAKUPI D**

**History**

The Java programming Language evolved from a language named **Oak**. Oak was developed in the early nineties at Sun Microsystems as a platform-independent language aimed at allowing entertainment appliances such as video game consoles and VCRs to communicate. **Oak** was first slated to appear in television set-top boxes designed to provide video-on-demand services. Just as the deals with the set-top box manufacturers were falling through, the World Wide Web was coming to life. As **Oak’s** developers began to re cognize this trend, their focus shifted to the Internet and Web Runner, an **Oak**-enabled browser, was born. Oak’s name was changed to **Java** and Web Runner became the Hot Java web browser. The excitement of the Internet attracted software vendors such that Java development tools from many vendors quickly became available. That same excitement has provided the impetus for a multitude of software developers to discover Java and its many wonderful features. (The **GNU General Public License** (**GNU GPL** or simply **GPL**) is a widely used free software license)

**JAVA Principles**

There were five primary goals in the creation of the Java language:

1. It should be "simple, object oriented and familiar".
2. It should be "robust and secure".
3. It should be "architecture neutral and portable".
4. It should execute with "high performance".
5. It should be "interpreted, threaded, and dynamic".

**JAVA**

Java is a programming language used to develop software applications as well as applets that run on web pages. Originally developed by [**James Gosling**](http://en.wikipedia.org/wiki/James_Gosling) at [Sun Microsystems](http://en.wikipedia.org/wiki/Sun_Microsystems) and released in 1995 as a core component of Sun Microsystems' [Java platform](http://en.wikipedia.org/wiki/Java_(software_platform)). The language derives much of its [syntax](http://en.wikipedia.org/wiki/Syntax_(programming_languages)) from [C](http://en.wikipedia.org/wiki/C_(programming_language)) and [C++](http://en.wikipedia.org/wiki/C%2B%2B) but has a simpler [object model](http://en.wikipedia.org/wiki/Object_model) and fewer low-level facilities. Java applications are typically [compiled](http://en.wikipedia.org/wiki/Compiler) to [bytecode](http://en.wikipedia.org/wiki/Java_bytecode) ([class file](http://en.wikipedia.org/wiki/Class_(file_format))) that can run on any [Java Virtual Machine](http://en.wikipedia.org/wiki/Java_Virtual_Machine) (JVM) regardless of [computer architecture](http://en.wikipedia.org/wiki/Computer_architecture). Java is general-purpose, concurrent, class-based, and object-oriented, and is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere".

The original and [reference implementation](http://en.wikipedia.org/wiki/Reference_implementation_(computing)) Java [compilers](http://en.wikipedia.org/wiki/Compiler), virtual machines, and [class libraries](http://en.wikipedia.org/wiki/Library_(computing)) were developed by Sun from 1995. As of May 2007, in compliance with the specifications of the [Java Community Process](http://en.wikipedia.org/wiki/Java_Community_Process), Sun relicensed most of their Java technologies under the [GNU General Public License](http://en.wikipedia.org/wiki/GNU_General_Public_License). Others have also developed alternative implementations of these Sun technologies, such as the [GNU Compiler for Java](http://en.wikipedia.org/wiki/GNU_Compiler_for_Java) and [GNU Classpath](http://en.wikipedia.org/wiki/GNU_Classpath).

**Java is a high level language**

Java's syntax allows for the use of words and commands instead of just symbols and numbers; it is closer to human languages and further from machine language. The advantage to Java being a high level language is that it is easier to read, write, and maintain.

**Java is an object oriented language**

With Java, you can define your own reusable data structures called objects as well as define their attributes (properties) and things they can do (methods). You can also create relationships between various objects and data structures.

**Java is a web language AND a software development language**

With Java, you can create applets - small programs that run on web pages, as well as stand-alone software applications.

**Java is platform independent**

You can run the same Java programs on various operating systems without having to rewrite or recompile them, unlike other high level languages such as C and C++. Java is independent of specific hardware architecture and operating systems.

What makes Java's platform independence possible is the way operating systems interpret Java. Java source code will remain the same irregardless of what operating system you are writing a Java program for. Essentially, Java source code is not converted into machine language, but rather into a special form of instruction known as Java **byte code**. Java byte code is then interpreted by the **Java run-time environment**. The Java run-time environment is a Java interpreter which is also known as the **Java virtual machine**. The Java run-time environment interprets Java byte code and instructs the operating system what to do. This allows for Java's platform independence, since Java source code will be interpreted the same way on all operating systems by the Java run-time environment.

**Java and JavaScript**

There exists a common misconception that Java and JavaScript are the same language. They are not. Java is a language used to create applets that run in web pages, as well as stand-alone software applications, while JavaScript is a scripting language used to create dynamic and interactive content on web pages.

**Java file extensions**

Java source code files have a **.java** extension. Java source code files that have been compiled have a **.class** extension.

**Importance of java**

**Interact with the user**

Interact with the user. For example, you can ask the user for their name and print a custom message with it such as "Hello Roger!"

**Create graphical programs**

With Java, you can create graphical programs which can include various graphical components including buttons, textboxes, menus, checkboxes, and more. For example, you can create a simple text editing program such as Window's Notepad.

**Create applets**

An applet is a program that runs within another program. With Java, you can create applets that will run inside web pages. For example, you can create an applet that will get input from the user and store the data in a database.

**Read from and write to files**

You can read from and write to files. For example, you can store data that is input by the user in a text file and retrieve that data when the user accesses the program again.

**Communicate with databases**

Read data stored in a database or write new data to a database. For example, you can store a users name and e-mail address in a database, and allow them to retrieve this information and view it or change it, and the change will be reflected in the database.

To write your first program, you'll need:

**The Java SE Development Kit 6 (JDK 6)**

You can download the Windows version. (Make sure you download the **JDK**, *not* the JRE.).

**A text editor**

In the examples, we'll use Notepad, a simple editor included with the Windows platforms. You can easily adapt these instructions if you use a different text editor.

These two items are all you'll need to write your first application.

**Creating Your First Application**

Your first application, HelloWorldApp, will simply display the greeting "Hello world!".

To create this program, you will:

**Create a source file**

A source file contains code, written in the Java programming language, that you and other programmers can understand. You can use any text editor to create and edit source files.

**Compile the source file into a .class file**

The Java programming language *compiler* (javac) takes your source file and translates its text into instructions that the Java virtual machine can understand. The instructions contained within this file are known as *bytecodes*.

**Run the program**

The Java application *launcher tool* (java) uses the Java virtual machine to run your application.

**Create a Source File**

To create a source file:

First, start your editor. You can launch the Notepad editor from the **Start** menu by selecting **Programs > Accessories > Notepad**. In a new document, type in the following code:

/\*\*

 \* The HelloWorldApp class implements an application that

 \* simply prints "Hello World!" to standard output.

 \*/

class HelloWorldApp

{

    public static void main(String[] args)

{

        System.out.println("Hello World!"); // Display the string.

    }

}

**Be Careful When You Type**

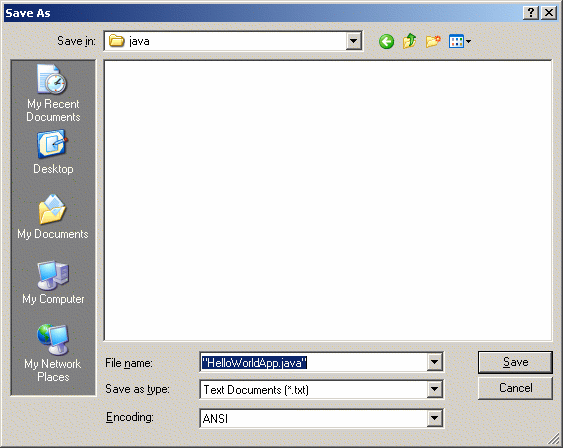
Type all code, commands, and file names exactly as shown. Both the compiler (javac) and launcher tool (java) are *case-sensitive*, so you must capitalize consistently.

HelloWorldApp helloworldapp

Save the code in a file with the name HelloWorldApp.java. To do this in Notepad, first choose the **File > Save As** menu item. Then, in the **Save As** dialog box:

1. Using the **Save in** combo box, specify the folder (directory) where you'll save your file. In this example, the directory is java on the C drive.
2. In the **File name** text field, type "HelloWorldApp.java", including the quotation marks.
3. From the **Save as type** combo box, choose **Text Documents (\*.txt)**.
4. In the **Encoding** combo box, leave the encoding as ANSI.

When you're finished, the dialog box should look like this.

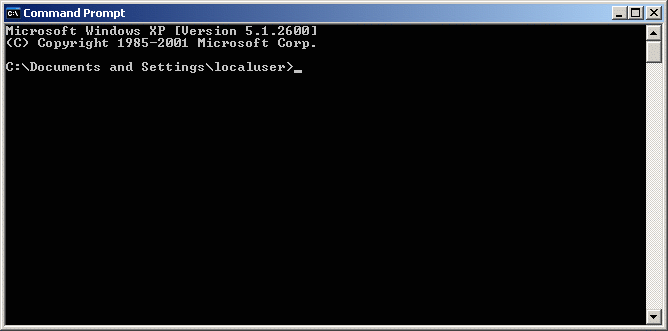


The Save As dialog just before you click **Save**.

Now click **Save**, and exit Notepad.

**Compile the Source File into a .class File**

Bring up a shell, or "command," window. You can do this from the **Start** menu by choosing **Command Prompt** (Windows XP), or by choosing **Run...** and then entering cmd. The shell window should look similar to the following figure.



A shell window.

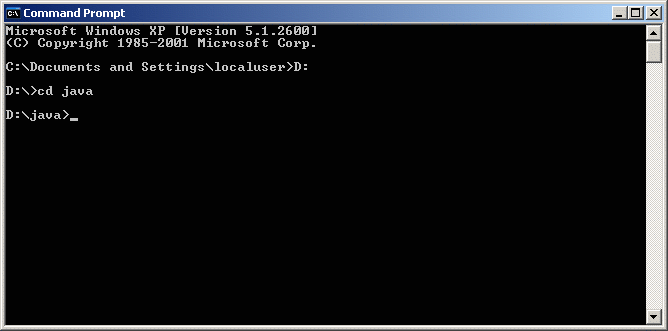
The prompt shows your *current directory*. When you bring up the prompt, your current directory is usually your home directory for Windows XP (as shown in the preceding figure).

To compile your source file, change your current directory to the directory where your file is located. For example, if your source directory is java on the C drive, type the following command at the prompt and press **Enter**:

cd C:\java

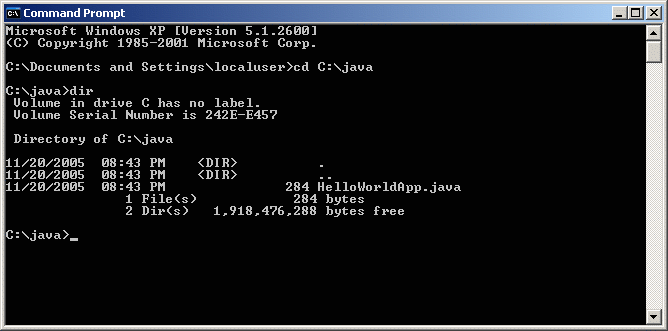
Now the prompt should change to C:\java>.

**Note:** To change to a directory on a different drive, you must type an extra command: the name of the drive. For example, to change to the java directory on the D drive, you must enter D:, as shown in the following figure.



Changing directory on an alternate drive.

If you enter dir at the prompt, you should see your source file, as the following figure shows.

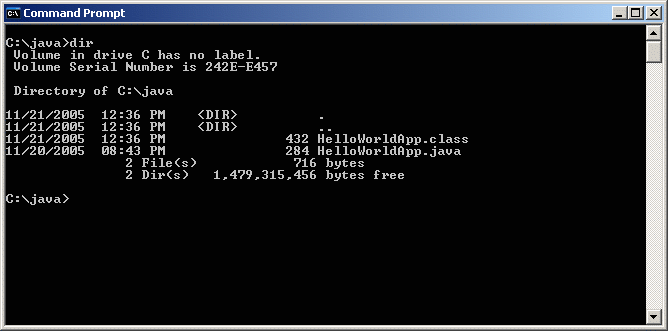


Directory listing showing the .java source file.

Now you are ready to compile. At the prompt, type the following command and press **Enter**.

javac HelloWorldApp.java

The compiler has generated a bytecode file, HelloWorldApp.class. At the prompt, type dir to see the new file that was generated, as shown in the following figure.



Directory listing, showing the generated .class file

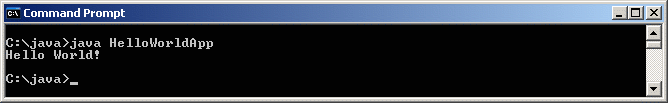
Now that you have a .class file, you can run your program.

**Run the Program**

In the same directory, enter the following command at the prompt:

java HelloWorldApp

The next figure shows what you should now see:



The program prints "Hello World!" to the screen.

**Object-Oriented Programming Concepts**

**Object**

An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life.

**Class**

A class is a blueprint or prototype from which objects are created.

**Inheritance**

Inheritance provides a powerful and natural mechanism for organizing and structuring your software.

**Interface**

An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface.

**Package**

A package is a namespace for organizing classes and interfaces in a logical manner. Placing your code into packages makes large software projects easier to manage.

**The fundamental elements of a Java program**

The source code of every Java program has to have a few fundamental elements. Every Java program should have:

**A class declaration**

A class is a grouping of related variables and functions (methods) that is used to achieve something. All the source code for a Java program will be placed within the class definition.

Syntax for declaring a class:

class nameOfClass

{

Code that will run the program goes here

}

**NOTE**: Class names should be descriptive and reflect the central purpose of a program.

**Example:**

class PrintText

{

}

In the above example a class named **PrintText** is declared.

**A main() method**

A method is a grouping of code that executes when it is called. The main() method is what makes a Java program work. When you insert the main() method into a Java program, it has to be used with a few special keywords, and has to contain a certain parameter.

**Example:**

class PrintText

{

**public static void main(String[] args)**

**{**

**}**

}

The above example does not do anything, but only contains the fundamental elements needed to create a Java program.

***Printing text***

To print text in a Java program, you can use either the **System.out.print()** method to print a single line of text or the **System.out.println()** method to print a single line of text followed by a line break.

*Syntax:*

System.out.print("textToPrint");

System.out.println("textToPrint");

**Example:**

class PrintText

{

public static void main(String[] args)

{

**System.out.println("Here is some text");**

**System.out.print("Here is some more text");**

}

}

The above code uses the **System.out.println()** method to print out one line of text followed by a line break, and then the **System.out.print()** method to print out another line of text.

**NOTE**: Every line of code in a Java program must end with a semicolon. If you don't end a line of code with a semicolon, an error will be generated!

**Including comments in Java code**

Comments in Java are declared so that code would be easier to understand and to navigate. Comments are not seen within code and can be placed anywhere within it. You can have single line comments and multi line comments.

**Single line comments**

Single line comments in Java are declared with two / symbols.

*Example:*

class PrintText

{

public static void main(String[] args)

{

*//print a single line of text*

*//followed by a line break*

System.out.println("Here is some text");

*//print another line of text*

*//with no line breaks afterwards*

System.out.print("Here is some more text");

}

}

**NOTE**: Single line comments can span only a single line.

**Multi line comments**

Multi line comments in Java are declared with a starting /\* and an ending \*/

Example:

**/\***

*This is a multi-line comment*

*This program will print*

*two lines of text*

*This multi-line comment*

*contains seven lines*

**\*/**

class PrintText

{

public static void main(String[] args)

{

System.out.println("Here is some text");

System.out.print("Here is some more text");

}

}

**VARIABLES.**

A variable is a container which holds information in a computer's memory. The value of a variable can change all throughout a program.

***Declaring variables***

In Java, different variables store different types of data. The type of data that is stored by a variable is signified with a **data type**.

**Java data types:**

|  |  |  |
| --- | --- | --- |
| **Data type** | **Type of data it stores** | **Size in memory** |
| boolean | true/false value | 1 bit |
| byte | byte size integer | 8 bits |
| char | a single character | 16 bits |
| double | double precision floating point decimal number | 64 bits |
| float | single precision floating point decimal number | 32 bits |
| Int | a whole number | 32 bits |
| long | a whole number (used for long numbers) | 64 bits |
| short | a whole number (used for short numbers) | 16 bits |

A variable is declared with the data type of the data it will store.

***Syntax:***

dataType varName;

**Example:**

char aCharacter;

int aNumber;

In this example, two variables are declared. The first variable is used to store a single character and is therefore of data type char. The second variable is used to store a whole number and is therefore of data type int. You can assign a value to a variable at the same time that it is declared. This process is known as **initialization**.

***Example of initializing a variable***:

char aCharacter = 'a';

int aNumber = 10;

In the above example a character variable named aCharacter is initialized with the value 'a' and a numeric variable is initialized with the value 10.

***Example of declaring a variable and then giving it a value***:

char aCharacter

aCharacter = 'a';

int aNumber

aNumber = 10;

In the above example a character variable named aCharacter is declared. On the next line this variable is assigned the value 'a' and a numeric variable named aNumber is declared and on the next line it is assigned the value 10.

**NOTE:** A variable must be declared with a data type. If you do not specify a data type for a variable, an error will be generated. The data type of a variable should be used only once with the variable name - during declaration. After a variable has been declared or initialized, you can refer to it by its name without the data type.

**NAMING VARIABLES**

When naming variables, several rules should be considered. These rules will make variable declaration easier, as well as clear up possible errors in code.

**Make sure that the variable name is descriptive**

If you do not give a variable a descriptive name, it will be hard to understand what the variable refers to. For example, if you wanted to create a variable which would hold a value describing the amount of chairs in a room, then the better choice for the variable name would be numChairs because it is more descriptive.

**Make sure the variable name is of appropriate length**

Make sure the variable name is long enough to be descriptive, but not too long.

**Do not use spaces in variable names.**

Java does not allow spaces in variable names, doing so will generate an error!

**Do not use special symbols in variable names such as !@#%^&\***

As is the rules with spaces, special symbols are not allowed in variable names, and using special symbols in variable names will generate an error. There is however one special symbol that can be used in variable names, and that symbol is the undersocre ( \_ ) symbol. Variable names can only contain letters, numbers and the underscore ( \_ ) symbol.

**Variable names can not start with an integer**

While integers can be used in variable names, it cannot be the first character in a variables name. Variable names can only start with a letter or the underscore ( \_ ) symbol.

**Distinguish between uppercase and lowercase**

Java is a case sensitive language which means that the variables varOne, VarOne, and VARONE are three separate variables!

**When referring to existing variables, be careful about spelling**

If you try to reference an existing variable and make a spelling mistake, an error will be generated!

**PRINTING VARIABLES**

Variables are printed by including the variable name in a *System.out.print()* or *System.out.println()* method. When printing the value of a variable, the variable name should NOT be included in double quotes.

**Example:**

class PrintText

{

public static void main(String[] args)

{

*//declare some variables*

byte aByte = -10;

int aNumber = 10;

char aChar = 'b';

boolean isBoolean = true;

*//print the variables*

System.out.println(aByte);

System.out.println(aNumber);

System.out.println(aChar);

System.out.println(isBoolean);

}

}

\

You can also print variables together with regular text. To do this, use the + symbol to join the text and variable values together.

**Example:**

class PrintText

{

public static void main(String[] args){

*//declare some variables*

byte aByte = -10;

int aNumber = 10;

char aChar = 'b';

boolean isBoolean = true;

*//print the variables*

System.out.println("aByte = " + aByte);

System.out.println("aNumber = " + aNumber);

System.out.println("aChar = " + aChar);

System.out.println("Is the isBoolean

variable a boolean variable? " + isBoolean);

}

}

**JAVA CONDITIONAL LOGIC**

The conditions in a program are not always the same. For example, when different users enter different data, how would you react to it accordingly? You need to be able to perform different actions based on certain conditions in a program. You can do this in Java using conditional logic.

* The if statement
* The else statement
* The else-if statement
* Using if, else, and else if together
* The switch statement
* The ternary operator

**The if statement**

The if statement tests if a certain condition is true or false and acts upon it accordingly.

***Syntax:***

if(condition)

{

Perform this action;

}

If the condition in the parenthesis is true, then the code following the condition will be executed, otherwise it will not.

**Example:**

class TestCondition

{

public static void main(String[] args)

{

*//declare a numeric variable*

int aNumber = 5;

*//check if aNumber equals 5 and if it is*

*//print a message accordingly*

if (aNumber == 5)

{

System.out.print("aNumber is equal to 5");

}

}

}

**NOTE**: Use two equal signs (==) in the condition when comparing values. One equal sign (=) is used to assign values, while two equal signs (==) are used to compare values.

**The else statement**

If the condition given in the parenthesis in an ***if statement*** is true, then the code following the condition will be executed. If you wanted one thing to happen if the condition is true, and another if the condition is false then else statement is used. The else statement works in conjunction with the if statement and executes certain code if the condition in the if statement is false.

***Syntax:***

if(condition)

{

Perform this action;

}

else

{

Perform this action if the above condition is false;

}

If the condition in the ***if statement*** is false, then the action dictated by the else statement will be performed.

**Example 1:**

class TestCondition

{

public static void main(String[] args)

{

*//declare a numeric variable*

int aNumber = 5;

*//check if aNumber equals 5 and if it is*

*//print a message accordingly*

if (aNumber == 5)

{

System.out.print("aNumber is equal to 5");

}

*//otherwise print*

*//a different message*

else

{

System.out.println("aNumber equals a

number other than 5");

}

}

}

**Example 2:**

class TestCondition

{

public static void main(String[] args){

*//declare a numeric variable*

int aNumber = 10;

*//check if aNumber equals 5 and if it is*

*//print a message accordingly*

if (aNumber == 5)

{

System.out.print("aNumber is equal to 5");

}

*//otherwise print*

*//a different message*

else

{

System.out.println("aNumber equals a

number other than 5");

}

}

}

**The else- if statement**

The if statement tests a single condition and performs an action if that condition is true and the else statement performs an action if the condition in the if statement is false. if there are more than two possibilities then else if statement is used. The ***else if statement*** is used in conjunction with the ***if statement***. Unlike the ***else statement***, it does not specifically perform a certain action if the condition in the ***if statement*** is false, but rather it performs an action if the condition in the ***if statement*** is another specific value specified in the ***else if statement*** itself.

***Syntax:***

if(condition is one value)

{

Perform this action;

}

else if(condition equals another value)

{

Perform this action;

}

**Example:**

class TestCondition

{

public static void main(String[] args)

{

int aNumber = 17;

if (aNumber == 5)

{

System.out.print("aNumber is equal to 5");

}

else if (aNumber == 7)

{

System.out.print("aNumber is equal to 7");

}

else if (aNumber == 20)

{

System.out.print("aNumber is equal to 20");

}

else

{

System.out.print("Not valid");

}

}

}

**Using if, else, and else if together**

It is possible to use the ***if, else, and else if statements*** together when you want to check a variable for a certain value many times. If it is not any of the checked values then the code specified by the else statement will be executed.

***Syntax:***

if(condition)

{

Perform this action;

}

else if(condition equals another value)

{

Perform this action;

}

else if(condition equals another value)

{

Perform this action;

}

else

{

if the condition was not equal to any of the values tested in

the if statement and all the else if statements, then perform

this action;

}

**Example:**

class TestCondition

{

public static void main(String[] args)

{

int aNumber = 5;

if (aNumber == 9)

{

System.out.print("aNumber is equal to 9");

}

else if (aNumber == 7)

{

System.out.print("aNumber is equal to 7");

}

else if (aNumber == 3)

{

System.out.print("aNumber is equal to 3");

}

else if (aNumber == 15)

{

System.out.print("aNumber is equal to 15");

}

else

{

System.out.print("aNumber is not equal to 9, 7, 3, or 15. aNumber is equal to " + aNumber);

}

}

}

**NOTE**: When using if, else if, and else statements - if the action dictated by these statements contains more than one line of code, it should be surrounded by curly braces, otherwise the curly braces are optional. Although, curly braces may not always be used with conditional statements. Doing so is good convention.

**Example:**

class TestCondition

{

public static void main(String[] args)

{

*//declare a numeric variable*

int aNumber = 5;

int anotherNumber = 7;

if (aNumber == 9)

{

System.out.println("aNumber is equal to 9");

System.out.print("anotherNumber is equal to 7");

}

else if (aNumber == 7) System.out.print("aNumber is

equal to 7");

else if (aNumber == 3)

{

System.out.println("aNumber is equal to 3");

System.out.print("anotherNumber is equal to 7");

}

else if (X == 15)

{

System.out.println("aNumber is equal to 15");

System.out.print("anotherNumber is equal to 7");

}

else System.out.print("aNumber is not equal to 9, 7, 3, or 15, it is equal to 5");

}

}

In the above example, some of the conditional statements dictate actions with one line of code - not sorrounded by curly braces, and some conditional statements dictate actions with more than one line of code - sorrounded by curly braces.

**The switch statement**

The switch statement is specifically designed for comparing one variable to a number of possible values. It can be thought of as a substitute for the ***if, else if, else structure***. There is an important keyword used within the switch structure, and that keyword is the **break** keyword. The **break** keyword is used to make sure that the switch structure will not fall through to the next possible value, even if that value is incorrect within the switch structure.

**Syntax:**

switch(variable)

{

case possible value:

Perform this action;

break;

case possible value:

perform this action;

break;

case possible value:

perform this action;

break;

default:

perform this action if none of the values match;

}

**Example:**

class TestCondition

{

public static void main(String[] args)

{

int aNumber = 7;

switch(aNumber)

{

case 1:

System.out.print("aNumber is equal to 1");

break;

case 2:

System.out.print("aNumber is equal to 2");

break;

case 3:

System.out.print("aNumber is equal to 3");

break;

case 7:

System.out.print("aNumber is equal to 7");

break;

default:

System.out.print("aNumber is not equal to any of the values specified");

}

}

}

**The ternary operator**

The ternary operator is used with the question mark symbol (?), it works the same way as the ***if-else structure.***

**Syntax:**

variable = (condition) ? value the variable will take if condition is true: value the variable will take if condition is false

**Example:**

class TestCondition

{

public static void main(String[] args)

{

int aNumber = 1;

int anotherNumber = 7;

**aNumber = (anotherNumber==10) ? 5: 15;**

System.out.print("aNumber = " + aNumber);

}

}

**Other operators**

Other than the ternary operator, there are other operators used when working with conditional logic as well as variables:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Name** | **Purpose** |
| % | Modulus | Performs division of two numbers and returns the remainder |
| && | And | Will return true if two or more conditions are true |
| || | Or | Will return true if at least one of two conditions is true |
| ++ | Increment | Will add one to a numeric variable |
| -- | Decrement | Will subtract one from a numeric variable |

**JAVA LOOPS**

Loops are specifically designed by a programmer to perform repetitive tasks with one set of code so as to save time. If you have to write a program which performs a repetitive task such as printing 1 to 100. Coding 100 lines to do this would be hard. There is no easier way to code repetitive tasks than implementing loops. Some of the loops used in java are as follows:

* The for loop
* The while loop
* The do-while loop
* Preventing endless loops
* Breaking out of a loop
* Continuing a loop

**The for loop**

The for loop is used to repeat a task several number of times. Ie it is used when the number of tasks to be performed is know in advance.

***Syntax:***

for(int a\_variable = initial\_value; a\_variable < end\_value;

a\_variable\_increment)

{

Code to be executed;

}

**Example:**

class ForLoopExample

{

public static void main(String[] args)

{

for(int a = 1; a < 11; a++)

{

System.out.println(a);

}

}

}

*The for loop has three parts*:

**Variable declaration**

The variable declaration is the first part of the loop which initializes the variable at the beginning of the loop to some value. This value is the starting point of the loop.

**Condition**

The condition is the second part of the loop, and it is the part that decides whether the loop will continue running or not. While the condition in the loop is true, it will continue running. Once the condition becomes false, the loop will stop running.

**Increment statement**

The increment statement is the third part of the loop. It is the part of the loop that changes the value of the variable created in the variable declaration part of the loop. The increment statement is the part of the loop which will eventually stop the loop from running.

Based on the above explanation of each part of the for loop, consider the example from earlier in the lesson and take it apart to see what each part does. Remember, that this loop prints to the screen the numbers 1 - 10.

class ForLoopExample

{

public static void main(String[] args)

{

for(int a = 1; a < 11; a++)

{

System.out.println(a);

}

}

}

**int a = 1**

This is the variable declaration part of the loop. A variable named **a** is declared with a value of 1.

**a < 11**

This is the condition of the loop. It states that as long as the variable **a** is less than 11, the loop should keep running.

**a++**

This is the increment statement part of the loop. It states that for every iteration of the loop, the value of the variable **a** should increase by 1. Recall that initially **a** is 1.

**The while loop**

The while loop works differently from the for loop. The for loop repeats a segment of code a specific number of times, while the while loop repeats a segment of code an unknown number of times. The code within a while loop will execute while the specified condition is true.

***Syntax:***

while(condition is true)

{

Execute this code;

}

**Example:**

class WhileLoopExample

{

public static void main(String[] args)

{

int num = 0;

while(num < 25)

{

num = num + 5;

System.out.println(num);

}

}

}

In the above code, a variable named **num** is initialized with the value of 0. The condition in the while loop is that while **num** is less than 25, 5 should be added to **num**. Once the value of **num** is greater than 25, the loop will stop executing.

**The do-while loop**

The do-while loop is similar to the while loop, but it does things in the reverse order. The mechanism of the while loop is - while a condition is true, perform a certain action. The mechanism of the do-while loop is – it performs a certain action while a condition is true. However, the code within the do-while loop will always be execute at least once, even if the specified condition is false. This is because the code is executed before the condition is tested.

***Syntax:***

do

{

Execute this code;

}

while (condition);

**Example:**

class DoWhileLoopExample

{

public static void main(String[] args)

{

int num = 0;

do

{

num = num + 5;

System.out.println(num);

}

while (num < 25);

}

}

In the above code, a variable named **num** is initialized with the value of 5. The condition in the do-while loop is that while **num** is less than 25, 5 should be added to **num**. Once the value of **num** is greater than 25, the loop will stop executing.

**Preventing endless loops**

A necessary precaution when working with a loop is to make sure that the loop is not endless. This occurs when the condition in a loop never becomes false. To prevent endless loops, you need to ensure that the condition in a loop will eventually become false.

***Example of an endless for loop:***

class EndlessForLoop

{

public static void main(String[] args)

{

for(int a = 10; a > 5; a++)

{

System.out.println(a);

} }}

The above example initializes the variable **a** with the value of 10, and states that as long as **a** is greater than 5, add 1 to it. Based on the variable declaration and the increment statement, **a** will always be greater than 5 in this loop, and thus, it is an endless loop.

***Example of an endless while loop:***

class EndlessWhileLoop

{

public static void main(String[] args)

{

int num = 50;

while(num > 10)

{

num = num + 5;

}

}

}

The above example initializes the variable **num** with the value of 50, and states that as long as **num** is greater than 10, add five to **num**. Based on the variable declaration and the increment statement, **num** will always be greater than 10 in this loop, and thus, it is an endless loop.

***Example of an endless do-while loop***:

class EndlessDoWhileLoop

{

public static void main(String[] args)

{

int num = 50;

do

{

num = num + 5;

}

while (num > 40);

}

}

The above example initializes the variable **num** with the value of 50, and states that as long as **num** is greater than 40, add 5 to **num**. Based on the variable declaration and the increment statement, **num** will always be greater than 40 in this loop, and thus, it is an endless loop.

**Breaking out of a loop**

You can completely break out of a loop when it is still running. This is achieved with the **break** keyword. Once a loop is exited, the first statement right after it will be executed. The **break** keyword provides an easy way to exit a loop if an error occurs, or if you found what you were looking for.

**Example:**

class BreakOutOfLoop

{

public static void main(String[] args)

{

for(int a = 1; a < 10; a++)

{

System.out.println(a);

if(a == 5)

{

break;

}

}

System.out.print("You have exited the loop");

}

}

In the above example, the FOR loop is set to iterate 9 times and print the current value of the variable **a** during each iteration. The IF statement within the loop states that when the variable **a** is equal to 5, it should break out of the loop.

**Continuing a loop**

While you can break out of a loop completely with the **break** keyword, there is another keyword used when working with loops - the **continue** keyword. Using the **continue** keyword in a loop will stop the loop at some point and continue with the next iteration of the loop from the beginning of it.

**Example:**

class ContinueLoop

{

public static void main(String[] args)

{

for(int a = 1; a < 10; a++)

{

if(a == 5)

{

continue;

}

System.out.println(a);

}

System.out.print("You have exited the loop");

}

}

In the above example, the for loop is set to iterate 9 times and print the current value of the variable **a** during each iteration. The if statement within the loop states that when the variable **a** is equal to 5, stop the loop and continue with the next iteration of the loop from the beginning of it. For this reason, all the numbers except the number 5 are printed.

**JAVA PACKAGES AND INTERFACES**

A **package** is a group of related classes and interfaces that work together to provide a wide variety of functionality for various purposes.

**Interface**

An interface is a collection of methods that have no implementation - they are created, but have no functionality. For one to define their functionality you do that individually in different classes. An interface is an abstract, and a programmer defines how its elements (methods) work according to their needs.

**Different types of Java packages**

Java provides many different packages including:

**Java.lang** – this package Provides classes that are fundamental to developing Java programs including the String class (for working with text or strings) and the Math class (for performing calculations and for working with numeric data). It is imported by default into all Java programs.

**java.io** - this package Provides classes for input and output and provides interaction with the user and reading and writing to files.

**java.awt** - this package Provides classes for creating graphical user interfaces, drawing graphics, and displaying images.

**java.awt.event** - this package Provides classes and interfaces for handling events like minimizing of a window, clicking a button, and moving the mouse.

**java.net** - this package Provides classes for creating communication and exchanging data over a computer network.

**Importing packages**

For a programmer to be able to use classes and interfaces located in a package, you have to import the package they are in. If you try to use a class or an interface without importing the package, your program will generate an error. Packages are imported using the import keyword.

**Methods of Packages Importing:**

There are three different methods used to import packages:

**Importing an entire package** - This means you will be able to use all the classes and interfaces in that particular package. To do this use the package name followed by the \* character. E.g.

*//imports the entire java.awt package*

import java.awt.\*;

**Importing individual classes** – in this you will be able to use just those classes you have specified. Eg.

*//imports the BufferedReader class from java.io package*

import java.io.BufferedReader;

**Importing individual interfaces** – the programmer will be able to use only those interfaces specified. E.g.

*//imports the ActionListener interface from java.awt.event package*

import java.awt.event.ActionListener;

**Importing packages in an actual program:**

**import java.util.\*;**

**import java.io.\*;**

**import java.awt.event.ActionListener;**

class ProgramWithPackages

{

public static void main(String[] args)

{

System.out.println("This program imports some packages");

}

}

**NOTE:**

The import statement should be the first thing in your code, even before the class declaration done.

**Java interfaces**

An interface is a collection of methods that have no implementation - they are just created, but have no functionality. What's the purpose of such methods? For you to define their functionality individually in different classes. An interface is abstract, and you define how it's elements (methods) work according to your needs.

**Different types of Java interfaces**

Java provides many different types of interfaces for a wide variety of functionality. These include:

**WindowListener** – this provides methods for indicating what happens when window actions such as minimizing, resizing, and closing occur. It is located in the java.awt.event package.

**ActionListener** – this provides methods for indicating what happens when action events such as the clicking of a button occur. It is located in the java.awt.event package.

**AudioClip** – this provides methods for playing sounds. It is located in the java.applet package.

**Declaring an interface**

While Java provides interfaces for you to use, you can also create your own interfaces. An interface is declared with the interface keyword.

***Syntax:***

interface nameOfInterface

{

//methods for interface here;

}

You can add methods to an interface the same way you would add methods to a class, except that the methods in an interface have no implementation.

**Adding methods to an interface:**

interface DataManager

{

public void printData(String data);

public String getUserInput();

}

Just like regular Java programs, interfaces should be declared in separate files, the name of the file an interface is declared in should have the same name as the interface (including the same capitalization), and should have a .java extension. For example, the code for the interface from above should be in a separate file named DataManager.java

**Using an interface**

To use an interface, a programmer will use the **implements** keyword with the name of the interface in the class declaration line in the code. You specify that a class *implements* the interface and you define in that class what the methods from the interface will do.

***Syntax:***

class nameOfClass implements nameOfInterface

{

}

**Example:**

class Sample **implements DataManager**

{

}

From the above example the Sample class can now use the methods declared in the DataManager interface and define their functionality.

**NOTE:**

Remember always to import the package that contains the interface you are implementing as in the example below.

import java.awt.event.\*;

class FrameWithEvents **implements WindowListener**

{

}

**JAVA USER INPUT**

The programs we have covered just display some data and that's it, there is no interaction. This part focuses on accepting user input and interacting with the user based on that input.

**User input package**

The package that needs to be imported to accept user input is java.io. The java.io package contains classes and interfaces used for input and output.

Syntax

**import java.io.\*;**

class GetUserInput

{

}

**User input classes**

To get user input, we use the BufferedReader and InputStreamReader classes.

**The BufferedReader class** - buffers the user's input to make it work more efficiently.

**The InputStreamReader class** - reads the user's input.

**Example of a on Getting some input from the use:**

**import java.io.\*;**

class GetUserInput

{

public static void main(String[] args)

{

*//the data that will be entered by the user*

String name;

*//an instance of the BufferedReader class*

*//will be used to read the data*

BufferedReader reader;

*//specify the reader variable*

*//to be a standard input buffer*

reader = new BufferedReader(new InputStreamReader(System.in));

*//ask the user for their name*

System.out.print("What is your name? ");

*//read the data entered by the user using*

*//the readLine() method of the BufferedReader class*

*//and store the value in the name variable*

name = reader.readLine();

System.out.print(name);

}

}

**Exceptions handling in java**

Using the above code, you still cannot accept user input because one thing is still missing, and that is you have to catch the exception (a situation in which something unexpected might happen). The exception we will be working with in this situation is the IOException which is the exception used when an input error occurs. Exceptions are handled by executing a specific set of code if there is an error.

**Get some input with exceptions:**

import java.io.\*;

class input

{

public static void main(String[] args)

{

*//the data that will be entered by the user*

String name;

*//an instance of the BufferedReader class*

*//will be used to read the data*

BufferedReader reader;

*//specify the reader variable*

*//to be a standard input buffer*

reader = new BufferedReader(new InputStreamReader(System.in));

*//ask the user for their name*

System.out.print("What is your name? ");

try

{

*//read the data entered by the user using*

*//the readLine() method of the BufferedReader class*

*//and store the value in the name variable*

name = reader.readLine();

*//print the data entered by the user*

System.out.println("Your name is " + name);

}

catch (IOException ioe)

{

*//statement to execute if an input/output exception occurs*

System.out.println("An unexpected error occured.");

}

}

}

**Java exceptions**

Exceptions are used to handle situations where something unexpected might happen, such as attempting to divide by zero, a file you're trying to access is not found etc.

**The various Java exceptions**

Java has classes that can be used to handle exceptions in various situations.

**1. ArithmeticException** – this is used to handle arithmetic errors such as attempting to divide a number by zero. It it located in the java.lang package.

**2. NoSuchMethodException** – this is used to handle a situation where a method to be used is not found. It is located in the java.lang package.

**3. IOException** – this is used to handle input and output errors. It is located in the java.io package.

**4. FileNotFoundException** – this is used to handle a situation where a file cannot be found. It is located in the java.io package.

**NOTE:**

The base class of all the exception classes is the class Exception and it is located in the java.lang package.

**The try-catch block in java**

The way to deal with exceptions is to use a try-catch block. The try-catch block will execute one set of code if an exception does not occur, and a different set of code if an exception does occur. The programmer Specifies within the try-catch block which exception he/she is watching for.

***Syntax:***

try

{

code to execute if exception doesn't occur;

}

catch(anExceptionThatMayOccur)

{

code to execute if exception does occur;

}

**Example:**

try

{

name = reader.readLine();

System.out.println("Your name is " + name);

}

//watch for an input/output exception

catch(IOException ioe)

{

System.out.println("An unexpected error has occured");

}

**The try-catch-finally block**

The try-catch-finally block is the same as the try-catch block with one addition. (The finally) The finally block contains a set of code that will execute whether an exception occurred or not.

***Syntax:***

try

{

code to execute if exception doesn't occur;

}

catch(anExceptionThatMayOccur)

{

code to execute if exception does occur;

}

finally

{

code to execute whether exception occurs or not;

}

**Example:**

try

{

name = reader.readLine();

System.out.println("Your name is " + name);

}

*//watch for an input/output exception*

catch(IOException ioe)

{

System.out.println("An unexpected error has occured");

}

finally

{

*//print this whether an exception occurs or not*

System.out.println("Thanks for stopping by!);

}

**JAVA GUI (Graphical User Interface) PROGRAMMING**

Most of the examples above run in a command prompt and displays some text output we will now focus on graphical programming.

**Graphical programs package**

The java.awt package is used to create graphical programs and contains classes for displaying several graphical components such as frames, textboxes, labels, and buttons, as well as drawing graphics and displaying images.

**Creating a frame**

In Java, a frame is a standard graphical window. It has the minimize, maximize, and close buttons in its top right corner and can be moved and resized by default. Frames are created using the Frame class.

**Frame class constructors**

**Frame()** – it is used to Creates a new instance of a Frame that is initially invisible.

**Frame(String title)** –this is used to Creates a new instance of a Frame that is initially invisible with the specified title.

**Frame class methods**

**void setResizable(boolean resizable)** - Sets whether or not a frame is resizable

**void setTitle(String title)** – it is used to sets the title of a frame

**void setVisible(boolean visible)** – it is used the frame whether or not a frame is visible

**void setSize(int width, int height)** – it used to set size i.e. the width & height of a frame

**String getTitle()** – it is used to Return the title of a frame

**Create a frame by extending the Frame class:**

import java.awt.\*;

class AFrame **extends** Frame

{

public static void main(String[] args)

{

AFrame frame = new AFrame();

frame.setSize(200, 200);

frame.setVisible(true);

}

}

**Create a frame by creating an instance of the Frame class:**

import java.awt.\*;

class Aframe

{

public static void main(String[] args)

{

Frame aFrame = new Frame();

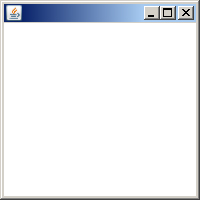
aFrame.setSize(200, 200);

aFrame.setVisible(true);

}

}

Both sets of code from above produce the same frame:



**NOTE:** Initially, a frame is not visible. You have to set the visiblity using the setVisible() method.

**GUI components**

There are various graphical components you can add to frames including labels, buttons, textboxes, and text areas. Each component is created through a class and each of these classes has methods to work with the component.

In the example below you will find usage of a few different classes used to create graphical components as well as the usage of some of their methods.

The layout of the frame is set to FlowLayout which means that the components in the frame will appear from left to right in the order in which they are added .

import java.awt.\*;

class FrameWithComponents

{

public static void main(String[] args)

{

Frame AFrame = new Frame("Frame with components");

**Label lblOne = new Label("This is a label");**

**Button btn1 = new Button("This is a button");**

**TextField tf1 = new TextField();**

**TextArea ta1 = new TextArea(12, 40);**

**tf1.setText("This is a textbox");**

**ta1.append("Number of columns in this textarea: " + ta1.getColumns());**

**//the add() method of the Frame class is**

**//used to add components to the frame**

**AFrame.add(lblOne);**

**AFrame.add(btn1);   
AFrame.add(tf1);**

**AFrame.add(ta1);**

AFrame.setSize(450, 300);

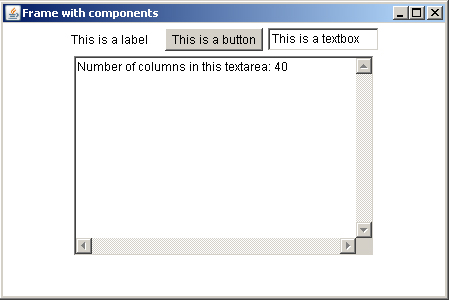
AFrame.setLayout(new FlowLayout());

AFrame.setVisible(true);

}

}

output:



**Java GUI layout**

Creating a graphical user interface and placing components on it is great, but how are you going to layout those components? For this task layout managers are used.

**Layout manager classes**

Layout manager classes are located in the java.awt package just like the classes used for displaying graphical components. So no need to import any extra packages when you're using layout managers

**Setting a layout**

You can set a frame to use a layout using the setLayout() method of the Frame class.

**Example:**

Frame AFrame = new Frame("Frame with components");

AFrame.setSize(450, 300);

**//set the layout of the frame to FlowLayout**

**AFrame.setLayout(new FlowLayout());**

AFrame.setVisible(true);

**The FlowLayout class**

The FlowLayout class is used to arrange components from left to right and if there is no more room, the next component will be wrapped onto a new line.

**FlowLayout class constructors:**

**FlowLayout()** - Creates a FlowLayout.

**FlowLayout(int alignment)** – this is used to Creates a FlowLayout with the specified alignment.

The Possible values include FlowLayout.LEFT, FlowLayout.CENTER, FlowLayout.RIGHT, FlowLayout.LEADING, FlowLayout.TRAILING.

**A frame using a flow layout:**

import java.awt.\*;

class FrameWithFlowLayout

{

public static void main(String[] args)

{

Frame AFrame = new Frame("Frame with components");

Label lblOne = new Label("This is a label");

Button btn1 = new Button("This is a button");

TextField tf1 = new TextField();

tf1.setText("This is a textbox");

AFrame.add(lblOne);

AFrame.add(btn1);

AFrame.add(tf1);

AFrame.setSize(450, 300);

**//set the layout of the frame to FlowLayout**

**//and align the components to the center**

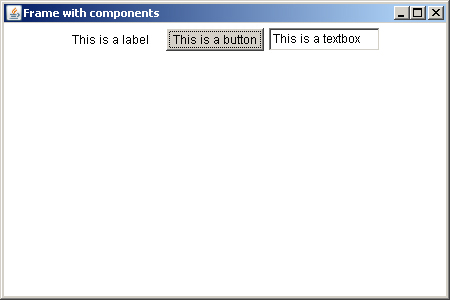
**AFrame.setLayout(new FlowLayout(FlowLayout.CENTER));**

AFrame.setVisible(true);

}

}

output:



**The GridLayout class**

The GridLayout class is used to arrange components in a grid of equally sized rectangular cells.

**GridLayout class constructors**:

**GridLayout()** - Creates a GridLayout which has one row by default.

**GridLayout(int rows, int columns)** - Creates a GridLayout with the specified number of rows and columns.

**A frame using a grid layout:**

import java.awt.\*;

class FrameWithGridLayout

{

public static void main(String[] args)

{

Frame AFrame = new Frame("Frame with components");

Label lblOne = new Label("This is a label");

Button btn1 = new Button("This is a button");

TextField tf1 = new TextField();

TextArea ta1 = new TextArea(12, 40);

tf1.setText("This is a textbox");

ta1.append("Number of columns in this textarea: " + ta1.getColumns());

AFrame.add(lblOne);

AFrame.add(btn1);

AFrame.add(tf1);

AFrame.add(ta1);

AFrame.setSize(480, 300);

**//set the layout of the frame to GridLayout**

**//specify the layout to have 2 rows and 2 columns**

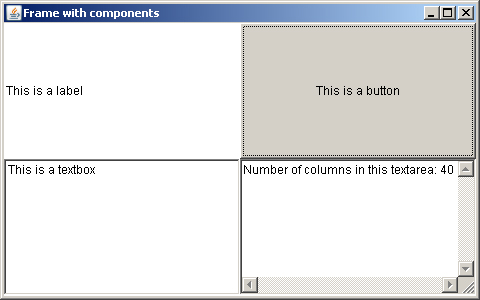
**AFrame.setLayout(new GridLayout(2, 2));**

AFrame.setVisible(true);

}

}

Output:



**Java graphics programming**

You can display various graphics including lines, rectangles, ovals, and images in Java programs.

**The Canvas class**

The first thing you will need is the Canvas class. This class is used to create an area in a frame to be used for displaying graphics.

**NOTE:** All the classes you will need to display graphics (as well as frames) are located in the java.awt package.

**Canvas class methods:**

**void setSize(width, height)** - Sets the size of the canvas

**void setBackground(Color c)** - Sets the background color of the canvas

**void setForeground(Color c)** - Sets the text color of the canvas

Add a canvas to a frame just like you would any other component:

Canvas C1 = new Canvas();

C1.setSize(120,120);

C1.setBackground(Color.white);

Frame F1 = new Frame();

F1.add(C1);

F1.setLayout(new FlowLayout());

F1.setSize(250,250);

F1.setVisible(true);

**Displaying graphics on a component**

Now that you have a Canvas (an area to display graphics on) how do you actually display those graphics? With the paint() method of the Frame class. The paint() method takes one attribute - an instance of the Graphics class. The Graphics class contain methods which are used for displaying graphics. The Graphics class lets a component draw on itself.

***Syntax:***

public void paint(Graphics g)

{

//methods for drawing graphics here;

}

**Drawing lines**

To draw lines, the drawLine() method of the Graphics class is used. This method takes four numeric attributes - the first two indicating the x/y starting point of the line, the last two indicating the x/y ending point of the line.

**Example:**

public void paint(Graphics g)

{

//draw a line starting at point 10,10 and ending at point 50,50.

g.drawLine(10, 10, 50, 50);

}

**Drawing rectangles**

To draw rectangles, the drawRect() method is used. This method takes four numeric attributes - the first two indicating the x/y starting point of the rectangle, the last two indicating the width and height of the rectangle.

**Example:**

public void paint(Graphics g)

{

//draw a rectangle starting at 100,100 width a width and height of 80

g.drawRect(100, 100, 80, 80);

}

**Filling a rectangle**

By default a rectangle will have no color on the inside (it will just look like a box). You can use the fillRect() method to fill a rectangle. The fillRect() method has four numeric attributes indicating the x/y starting position to begin filling and the height and width. Set these values the same as you did for the drawRect() method to properly fill the rectangle.

**Example:**

public void paint(Graphics g)

{

//draw a rectangle starting at 100,100 width a width and height of 80

g.drawRect(100, 100, 80, 80);

g.fillRect(100, 100, 80, 80);

}

The rectangle is filled, but we didn't set a color for it! To do this, we will use the setColor() method.

g.setColor(Color.orange);

**Drawing ovals**

To draw ovals, the drawOval() method is used. This method takes four numeric attributes - the first two indicating the x/y starting point of the oval, the last two indicating the width and height of the oval. Fill an oval with the fillOval() method which also takes four numeric attributes indicating the starting position to begin filling and the height and width. Set these values the same as you did for the drawOval() method to properly fill the oval.

**Example:**

public void paint(Graphics g)

{

g.setColor(Color.gray);

//draw an oval starting at 20,20 with a width and height of 100 and fill it

g.drawOval(20,20, 100, 100);

g.fillOval(20,20, 100, 100);

}

**Displaying images**

To display images, the Image class is used together with the Toolkit class. Use these classes to get the image to display. Use the drawImage() method to display the image.

**Example:**

public void paint(Graphics g)

{

Image img1 = Toolkit.getDefaultToolkit().getImage("sky.jpg");

//four attributes: the image, x/y position, an image observer

g.drawImage(img1, 10, 10, this);

}

**An entire Java graphics program:**

import java.awt.\*;

class GraphicsProgram extends Canvas

{

public GraphicsProgram()

{

setSize(200, 200);

setBackground(Color.white);

}

public static void main(String[] argS)

{

//GraphicsProgram class is now a type of canvas

//since it extends the Canvas class

//lets instantiate it

GraphicsProgram GP = new GraphicsProgram();

//create a new frame to which we will add a canvas

Frame aFrame = new Frame();

aFrame.setSize(300, 300);

//add the canvas

aFrame.add(GP);

aFrame.setVisible(true);

}

public void paint(Graphics g)

{

g.setColor(Color.blue);

g.drawLine(30, 30, 80, 80);

g.drawRect(20, 150, 100, 100);

g.fillRect(20, 150, 100, 100);

g.fillOval(150, 20, 100, 100);

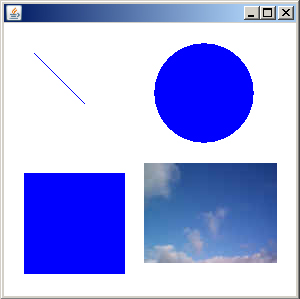
Image img1 = Toolkit.getDefaultToolkit().getImage("sky.jpg");

g.drawImage(img1, 140, 140, this);

}

}

Output:



**Java applets**

An applet is a Java program that runs on a webpage. Applets function like regular Java programs but with a few security restrictions (such as applets can't read or write files on your computer).

**The Applet class**

The Applet class is used to create applets. This class is located in the java.applet package. Applets are not created by instantiating an Applet object, rather by creating a class which extends the Applet class.

**Example (applet core structure):**

import java.applet.Applet;

class AnApplet extends Applet

{

}

**Applet class methods**

**init()** - Initializes an applet for usage and informs the applet that it has been loaded

**start()** - Informs an applet that it should start its execution

**stop()** - Informs an applet that it should stop its execution

**destroy()** - Informs an applet that it is being removed from memory and any resources it has allocated should be removed as well

**resize(int width, int height)** - Resizes the applet to the specified width and height

**Building an applet**

The first four methods discussed above are actually run by a web browser automatically. You can define the functionality of these methods in your applet code to specify what happens during each stage of an applet's existence.

The applet below uses these methods to print to a textarea what is currently happening.

**Applet with functionality:**

import java.applet.Applet;

import java.awt.TextArea;

public class AnApplet extends Applet

{

TextArea ta1 = new TextArea(12, 40);

//the \n is a 'newline character'

//it will start the text string on a new line

public void init()

{

add(ta1);

ta1.append("Applet has been initialized");

}

public void start()

{

ta1.append("\nApplet has been started");

}

public void stop()

{

ta1.append("\nApplet has been stopped");

}

public void destroy()

{

ta1.append("\nApplet has been destroyed");

}

}

**Placing an applet on a webpage**

This can be done using HTML's <applet> tag.

**Example:**

<html>

<head> <title>Applet on a webpage</title>

</head>

<body>

**<-- the code attribute denotes the name and location of the applet -->**

**<applet code="AnApplet.class" width="200" height="200">**

**</applet>**

</body>

</html>

Here is the applet:



If the applet doesn't load, try viewing this page in a different web browser.

**NOTE**: You should always refer to the compiled program file (the one with the .class extension) in an <applet> tag when placing an applet on a webpage. Not the .java source code file!

**Viewing an applet**

You can view an applet by viewing the webpage on to which the applet is placed in a web browser or you can use the Java appletviewer command line tool. In the command prompt type appletviewer and the name of the page where the applet is located.

**Example:**

appletviewer appletpage.html

The appletviewer will display just the applet, not the entire webpage.

**Java audio**

Java provides the ability to play audio in applications and applets.

**The AudioClip interface**

To be able to play audio in a program or applet, you first have to use the AudioClip interface and instantiate an AudioClip object with it. The AudioClip interface is located in the java.applet package.

**Example:**

AudioClip aClip;

**The newAudioClip method**

After instantiating an AudioClip object, you have to load the actual audio file to be played. This is achieved with the newAudioClip method - a static method of the Applet class which takes an instance of the URL class that loads the file.

**Example:**

aClip = Applet.newAudioClip(new URL("file:sound.wav"));

In the above example, an audio file named sound.wav is loaded as the audio file to be played.

Methods of the AudioClip interface

Once the audio file is uploaded, you can use the methods of the AudioClip interface to work with it.

**Methods of the AudioClip interface:**

**loop()** - will play an audio file continuously

**play()** - will play an audio file once

**stop()** - will stop an audio file while it is playing

**Complete audio file playing program:**

import java.awt.\*;

import java.awt.event.\*;

import java.applet.\*;

import java.net.\*;

class AudioFrame extends Frame implements ActionListener

{

**AudioClip bach;**

Button play, loop, stop;

public AudioFrame()

{

play = new Button("Play");

play.addActionListener(this);

add(play);

loop = new Button("Loop");

loop.addActionListener(this);

add(loop);

stop = new Button("Stop");

stop.addActionListener(this);

add(stop);

try{ **bach = Applet.newAudioClip(new URL("file:bach.mid"));**

}

//if there is a problem with the URL

//then this is the exception to be used catch (MalformedURLException mfe)

{

System.out.println("An error occured, please try again...");

}

setLayout(new FlowLayout());

setSize(220, 150);

setVisible(true);

}

public static void main(String[] args)

{

AudioFrame AF = new AudioFrame();

}

public void actionPerformed(ActionEvent e)

{

//the action event handler tracks which button

//is pressed and performs an action accordingly

if (e.getSource() == play)

{

**bach.play();**

}

if (e.getSource() == loop)

{

**bach.loop();**

}

if (e.getSource() == stop)

{ **bach.stop();**

}

}

}

Output:



**Java arrays**

Arrays are a very important concept in Java as well as many other programming languages. An array is a special type of variable which can store a list of values.

**The necessity of arrays**

If you are writing a program that has three variables which contain the names of Cars, your code for these variables would probably look like this:

String car1 = "Toyota Camry";

String car2 = "Honda Accord";

String car3 = "Mitsubishi Galant";

With Arrays, it is much simpler. An array gives you the ability to group together related variables into one set. Arrays are special variables which hold lists of information. Therefore, instead of the code above you can declare a 'Cars' array:

String[] Cars = new String[3];

Cars[0] = "Toyota Camry";

Cars[1] = "Honda Accord";

Cars[2] = "Mitsubishi Galant";

**Declaring arrays**

Arrays are declared with a data type, an array name, and square brackets [] specifying an array.

***Syntax:***

datatype arrayName[];

OR

datatype[] arrayName;

**Example:**

int evenNumbers[]; which can also be declared as

int[] evenNumbers;

The above example declares an array named evenNumbers of data type **int**. Whether you place the square brackets after the array name or after the data type makes no difference.

After an array is declared, an array object has to be assigned to it using the **new** keyword as well as the length of the array - specified in the square brackets. The length of the array denotes how many elements an array can hold.

***Syntax:***

arrayName = new datatype[numElementsInArray];

**Example:**

evenNumbers = new int[10];

The above example declares an array named evenNumbers of data type **int** which has a length of 10, therefore it can store 10 elements.

**Adding values to an array**

You can add values to an array by referring to the appropriate index of the array and assigning it a value.

***Syntax:***

arrayName[index] = value;

**Example:**

evenNumbers[3] = 20;

The above example will assign the value 20 to the 4th element in the evenNumbers array.

**NOTE**: Array indexes begin at 0, so the first element of an array is at index 0, the fifth element of an array is at index 4, and so on.

As an alternative to declaring an array and then assigning values to its elements, you can do both tasks on one line.

***Syntax:***

dataType[] arrayName = {value, value, value, value, etc.};

**Example:**

String[] Cars = {"Camry", "Accord", "Galant"};

The above example declares an array named Cars of data type String which stores three elements.

**Accessing an arrays elements**

You can access an array elements by referring to the array by its name and the appropriate index number of the element you wish to access.

**Example:**

class GetArrayElement

{

public static void main(String[] args)

{

String[] Cars = new String[3];  
 Cars[0] = "Toyota Camry";  
 Cars[1] = "Honda Accord";  
 Cars[2] = "Mitsubishi Galant";  
 System.out.println("Car: " + Cars[2]);

}

}

In the above example, the third element of the Cars arrays is printed by referring to the Cars array and the index number 2 in brackets.

**Modifying an arrays elements**

You can modify an array elements by referring to the array by its name and the appropriate index number of the element you wish to modify.

***Syntax:***

arrayName[index] = newValue;

**Example:**

class ModifyArrayElements

{

public static void main(String[] args)

{

int oddNumbers[] = new int[4];

oddNumbers[0] = 1;

oddNumbers[1] = 3;

oddNumbers[2] = 5;

oddNumbers[3] = 7;

*//print the initial value of oddNumbers[0]*

System.out.println("oddNumbers[0]: "

+ oddNumbers[0]);

*//print the initial value of oddNumbers[2]*

System.out.println("oddNumbers[2]: "

+ oddNumbers[2]);

*//change the value of oddNumbers[0] to 15*

oddNumbers[0] = 15;

*//change the value of oddNumbers[2]*

*// to oddNumbers[3] + 10*

oddNumbers[2] = oddNumbers[3] + 10;

*//print the new value of oddNumbers[0]*

System.out.println("new value of oddNumbers[0]: "

+ oddNumbers[0]);

*//print the new value of oddNumbers[2]*

System.out.println("new value of oddNumbers[2]: "

+ oddNumbers[2]);

}

}

In the above example, the initial value of two elements from the oddNumbers array is printed (oddNumbers[0] and oddNumbers[2]), the value of these elements is changed and the new value of each is then printed.

**Getting the length of an array**

The length of an array is the number of elements in it. To get the length of an array, use the length property of the Array object with the array whose length you want to find out.

**Example:**

class GetArrayLength

{

public static void main(String[] args)

{

int oddNumbers[] = new int[4];

oddNumbers[0] = 1;

oddNumbers[1] = 3;

oddNumbers[2] = 5;

oddNumbers[3] = 7;

*//print the length of the oddNumbers array*

System.out.print("The length of the oddNumbers array

is " + oddNumbers.length);

}

}

In the above example, the length property is used to return the length of the oddNumbers array.

**Copying Arrays**

The System class has an arraycopy method that you can use to efficiently copy data from one array into another:

public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)

The two Object arguments specify the array to copy *from* and the array to copy *to*. The three int arguments specify the starting position in the source array, the starting position in the destination array, and the number of array elements to copy.

The following program, ArrayCopyDemo, declares an array of char elements, spelling the word "decaffeinated". It uses arraycopy to copy a subsequence of array components into a second array:

class ArrayCopyDemo

{

public static void main(String[] args)

{

char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't', 'e', 'd' };

char[] copyTo = new char[7];

System.arraycopy(copyFrom, 2, copyTo, 0, 7);

System.out.println(new String(copyTo));

}

}

The output from this program is:

caffein

**Example**

**public class**CopyArray

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

{  
    **int**array1[]= {2,3,4,5,8,9};  
    **int**array2[] = **new int**[6];  
    System.out.println("array:");   
    System.out.print("[");  
    **for**(**int**i=0; i<array1.length; i++)

{  
    System.out.print(" "+array1[i]);  
}  
    System.out.print("]");  
    System.out.println("\narray1:");  
    System.out.print("[");  
    **for**(**int**j=0; j<array1.length; j++)

{  
    array2[j] = array1[j];  
    System.out.print(" "+ array2[j]);  
  }  
    System.out.print("]");  
  }  
}

**Passing Arrays to Methods**

Arrays are ***passed-by-reference*.**  *Passing-by-reference* means that when an array is passed as an argument, its **memory address location** is actually passed, referred to as its "reference".

In this way, the contents of an array CAN be changed inside of a method, since we are dealing directly with the actual array and not with a copy of the array.

int [ ] num = {1, 2, 3};  
testingArray(num);  **//Method call**  
System.out.println("num[0] = " + num[0] + "\n num[1] = " + num[1] + "\n num[2] =" + num[2]);  
. . .

**//Method for testing**   
public static void testingArray(int[ ] value)  
{  
    value[0] = 4;  
    value[1] = 5;  
    value[2] = 6;  
}

**Output:**  
num[0] = 4  
num[1] = 5  
num[2] = 6  
(The values in the array have been changed.  
Notice that nothing was "returned".)

**Example:**  Fill an array with 10 integer values.  Pass the array to a method that will add up the values and return the sum.

(In this example, no original data in the array was altered.  The array information was simply used to find the sum.)

*import java.io.\*;  
import BreezyGUI.\*;  
  
public class FindSum  
{  
     public static void main (String [ ] args)  
     {  
            int [ ]  number = new int [ 10];   // instantiate the array  
            int i;  
            int sum=0;*

*for ( i = 0; i < 10; i++ )            // fill the array  
               number[ i ] = Console.readInt("Enter number: " );*

*int sum = find\_sum(number);   // invoke the method  
           System.out.println("The sum is" +sum + ".");  
     }*

*public static int  find\_sum(int [ ] value)  //method definition to find sum  
    {  
          int i, total = 0;  
          for(i=0; i<10; i++)  
          {  
              total = total + value[ i ];  
          }  
  
          return (total);  
     }*

**Java strings**

A string is a grouping of text. You can store this group of a text in a variable that would then be known as a String variable. You can print strings, and you can also use various functions to perform operations on strings such as returning the string's length.

**Declaring a string**

Syntax:

String nameOfString = "stringValue";

String nameOfString = new String("stringValue");

**Example:**

String aString = "This is a string";

String aString = new String("This is a string");

Whether you declare a string as a variable or with the new keyword, it becomes an instance of the String class. A class is a special type of variable.

**Printing a string**

Use the System.out.print or System.out.println methods to print a string.

**Example:**

String myString = "I like pineapple";

*//print a string variable*

System.out.println(myString); *//print a non-variable string*

*//just put some text in double quotes and it's done*

System.out.println("Green is a great color");

**Concatenation**

Concatenation is the process by which two or more strings are joined together. This is achieved with the use of the + operator. You can use concatenation to join two or more strings into one or print two or more strings together.

**Example:**

String aString = "Here is some text. ";

String anotherString = "Here is some more text.";

*//declare a third string and*

*//combine into it the first two strings*

String combinedString = aString + anotherString;

System.out.println(combinedString);

*//print two strings together*

System.out.println("Notepad is a " + "simple text editor.");

**String functions**

The String class has various functions (methods) you can use to work with text.

**charAt(int index)** - Returns a character from a string at a specified index.

**equals()** - Compares two strings and returns true if they are the same.

**length()** - Returns the length of a string.

**toUpperCase()** - Converts all letters in a string to upper case.

**Example:**

String aString = "Here is some text";

String anotherString = "Here is some more text";

*//extract the third character*

*//from the aString string and print it*

System.out.println(**aString.charAt(2)**);

*//compare the two strings aString and anotherString*

*//and specify if they are the same or not*

System.out.println("The two strings match: " + **aString.equals(anotherString)**);

*//print the length of the aString string*

System.out.println("The length of the aString string variable is " + **aString.length()**);

*//print the anotherString string in all uppercase letters* System.out.println(**aString.toUpperCase()**);

**Java object-oriented programming (part 1)**

You can have simple data types that store some bit of information, but what if you wanted a much more advanced data type that represents an actual thing as opposed to just some piece of data? Where you can set attributes for it and things it does? You can do exactly this with object-oriented programming.

Java, being an object-oriented language, supports the creation of these advanced data types (objects). Objects consist of a set of variables (representing information about the object) and functions (representing what the object can do and things you can do with the object).

**Creating a class**

A class is a blueprint for an object. An object is a representation of a real world thing. An object can represent a car, a table, a book or any other real world concept. Before an object can be implemented, however, a blueprint has to be designed for it, and that blueprint is the class of an object. A class is a blueprint for an object stating the various properties (**variables**) of the object as well as what it can do (**methods**).

***Syntax for creating a class***:

class className

{

}

A class is created with the **class** keyword, followed by a single space, the name of the class to be created, an opening curly brace, and a closing curly brace. All the specifics and logistics of the class go in between these curly braces.

**Example of a class:**

class Book

{

}

The above code creates a class named Book.

Classes can be declared with certain keywords called **access modifiers** which dictate the level of access on a class.

**Class access modifiers:**

**Public**

The keyword **public** when placed in front of a class name specifies that a class can be accessed by any other class.

**Example:**

public class Aclass

{

}

**Protected**

The keyword **protected** when placed in front of a class name specifies that a class can only be accessed within the same package.

**Example:**

protected class Aclass

{

}

**Private**

The keyword **private** when placed in front of a class name specifies that a class can be accessed only within itself. It cannot be subclassed (extended) or instantiated.

**Example:**

private class Aclass

{

}

**Creating variables for a class**

Class variables are known as **member variables**, because they store important information in regards to the class.

**Example:**

class Book

{

String title;

int numPages;

}

The above code uses the Book class and creates two member variables for it. The first is title - a string variable which will store the title of the book. The second is numPages - a numeric variable which will store the number of pages in the book.

Variables can be declared with certain keywords called **access modifiers** which dictate the level of access on a variable.

**Variable access modifiers:**

**public**

The keyword **public** when placed in front of a variable specifies that the variable can be accessed directly by any other class.

**Example:**

public int aNumber;

**protected**

The keyword **protected** when placed in front of a variable specifies that the variable can be accessed by the class it is defined in, a subclass of the class it is defined in, or from classes within the same package.

**NOTE**: A package is a collection of related classes.

**Example:**

protected int aNumber;

**private**

The keyword **private** when placed in front of a variable specifies that the variable can be accessed only by the class it is declared in.

**Example:**

private int aNumber;

**Creating methods for a class**

Class methods are defined inside a class definition, they can be used to either retrieve data or manipulate data. Class methods which are used to retrieve data are known as **accessor methods**. Class methods which are used to manipulate data are known as **mutator methods**.

**Example:**

class Book

{

String title;

int numPages;

*//this is a mutator method*

public void setNumPages(int numOfPages)

{

numPages = numOfPages;

}

*//this is an accessor method*

public int getNumPages()

{

return numPages;

}

*//this is a mutator method*

public void setTitle(String theTitle)

{

title = theTitle;

}

*//this is an accessor method*

public String getTitle()

{

return title;

}

}

In the above example, four methods are created:

**setNumPages(int numOfPages)**

This method is used to set the number of pages in the book, this is done through its parameter numofPages. The method sets the variable numPages to whatever value you supply to it through its parameter numOfPages.

**getNumPages()**

This method is used to return the number of pages in the book, this is done through the use of the keyword **return** followed by the variable numPages - the variable that stores the value of how many pages there are in the book.

**setTitle(String theTitle)**

This method is used to set the title of the book; this is done through its parameter theTitle. The method sets the variable title to whatever value you supply to it through its parameter theTitle.

**getTitle()**

This method is used to return the title of the book; this is done through the use of the keyword **return** followed by the variable title - the variable that stores the title of the book.

Methods can be declared with certain keywords called access modifiers which dictate the level of access on a method.

**Method access modifiers:**

**public**

The keyword **public** when placed in front of a method specifies that the method can be accessed directly by any other class.

**Example:**

public void printNumber()

{

}

**protected**

The keyword **protected** when placed in front of a method specifies that the method can be accessed by the class it is defined in, a subclass of the class it is defined in, or from classes within the same package.

Example:

protected void printNumber()

{

}

**private**

The keyword **private** when placed in front of a method specifies that the method can be accessed only by the class it is declared in.

**Example:**

private void printNumber()

{

}

Another keyword used with methods is the **void** keyword. The **void** keyword is used to specify that a method will not return a value. To specify that a method does return a value, instead of the **void** keyword, the appropriate keyword corresponding to the data type that the method will return should be used. For example, a method that will return an integer should have the keyword **int** in front of it.

Generally, **mutator methods** do not return a value, and **accessor methods** do return a value.

The example from above contains two methods that do not return a value and two methods that do return a value. Ie.

class Book

{

String title;

int numPages;

*//does not return a value*

*//use the keyword void*

public void setNumPages(int numOfPages)

{

numPages = numOfPages;

}

*//returns an int value*

*//use the keyword int*

public int getNumPages()

{

return numPages;

}

*//does not return a value*

*//use the keyword void*

public void setTitle(String theTitle)

{

title = theTitle;

}

*//returns a String value*

*//use the name of the class (String)*

public String getTitle()

{

return title;

}

}

**Instantiating a class**

You can create objects of a class based on how you designed your class. The process of creating an object is called **instantiation**.

***Syntax:***

nameOfClass nameOfObject = new nameOfClass();

**Example:**

Book YellowPages = new Book();

The above example creates an object named YellowPages which is an instance of the Book class.

**Creating a class constructor**

What if you want an object to automatically have a certain value when it is instantiated? For example, if you wanted all instances of the Book class to automatically have 200 pages? This is what class costructors are for. Class constructors are used to set objects to have certain values when they are initialized. The method used for class constructors takes the same name as the class for which it will be a constructor.

**NOTE**: A class constructor should be declared within the class that it is a constructor for.

***Syntax***:

name\_Of\_Class\_To\_Create\_A\_Constructor\_For([parameters])

{

}

**Example:**

class Book

{

String title;

int numPages;

public Book(int numPages)

{

this.numPages = numPages;

}

}

In the above example, the Book class contains a constructor with one parameter. The value given to this parameter will be how many pages the book has.

Did you notice a new keyword in the above example? The keyword **this** is a special keyword used when creating classes. The purpose of it is to refer to the current class.

Now we can instantiate a new Book object and automatically set how many pages it has based on the class constructor.

***Book object with 340 pages:***

Book YellowPages = new Book(340);

**Using class variables**

You can use the variables of a class with an object by referring to them by name together with the object name.

***Syntax:***

objectName.variableName;

**Example:**

class Aclass

{

public static void main(String[] args)

{

Book ABook = new Book();

ABook.numPages = 200;

System.out.println("Number of pages in the book: " + ABook.numPages);

}

}

**NOTE**: You can refer directly to a class variable like in the above example only if the variable is set as public! You can still access class variables even if they are not set as public through encapsulation.

**Using the methods of a class with an object**

Just as with the variables of a class, you can use the methods of a class with an object by referring to them by name together with the object name.

**Syntax:**

objectName.methodName([parameters])

**Example:**

class Aclass

{

public static void main(String[] args)

{

Book ABook = new Book();

*//set the number of pages in the book to 300*

ABook.setNumPages(300);

*//set the title of the book to "Read Me"*

ABook.setTitle("Read Me");

System.out.println("Book title: " + ABook.getTitle());

System.out.println("Number of pages in the book: " + ABook.getNumPages());

}

}

**NOTE**: Because getTitle() and getNumPages() are used to return values, you can use them as values themselves. This is why they are used inside of the System.out.println() statements.

**Inheritance**

Inheritance is the process by which a class gets the properties and methods of another class. The idea behind inheritance is that it is not absolutely necessary to always build a class from scratch. Instead, you can take an existing class, add a few new features to it, and have a new class based on the already existing class. Inheritance is achieved through the use of the extends keyword. Recall the Book class with all its variables and methods from above.

class Book

{

Sting title;

int numPages;

public void setNumPages(int numOfPages)

{

numPages = numOfPages;

}

public int getNumPages()

{

return numPages;

}

public void setTitle(String theTitle)

{

title = theTitle;

}

public String getTitle()

{

return title;

}

}

This class is designed for objects which represent books, but what if you wanted to have a class whose objects represent soft cover books only? This is where inheritance comes into the picture. Instead of creating a whole new class for soft cover books, we will create a class for soft cover books which extends the Book class.

***The SoftCoverBook class***:

class SoftCoverBook **extends** Book

{

}

Now the SoftCoverBook class *inherits* all the variables and methods of the Book class, and they do not have to be declared again. Rather, they are automatically part of the SoftCoverBook class.

A class that extends another class is said to be its subclass and a class that is extended is said to be the superclass of that class. In the above example, SoftCoverBook is a subclass and Book is its superclass.

**NOTE:** All classes in Java inherits from the class Object whether this is explicitly declared or not. The Object class is the base class in Java.

**Polymorphism**

Polymorphism (meaning many forms) means that an instance of a class is also an instance of its superclass. For example, if you create an instance of the SoftCoverBook class, it is automatically an instance of the Book class because the SoftCoverBook class is a subclass of the Book class. Furthermore, since all classes in Java inherit from the Object class, every instance of the SoftCoverBook class as well as the Book class are automatically instances of the Object class.

**Encapsulation**

Encapsulation basically means data hiding. Through the use of encapsulation, you can declare class variables as private so that objects of the class (as well as other classes) cannot access them directly. You can then declare public methods within the same class that access these variables, and then objects of the class (as well as other classes) can access the private class variables by proxy through these public methods.

**Example:**

class Book

{

*//declare a private class variable*

private String title;

*//declare a public method to set the title of the book*

public void setTitle(String theTitle)

{

title = theTitle;

}

*//declare a public method to get the title of the book*

public String getTitle()

{

return title;

}

}

**Try to access private variable directly:**

class Aclass

{

public static void main(String[] args)

{

*//declare a new instance of the Book class*

Book ABook = new Book();

ABook.setTitle("Greatest book ever");

*//try to access the private variable title directly*

*//it will not work and an error will be generated*

System.out.println(ABook.title);

}

}

Output:

AClass.java:12: title has private access in Book System.out.println(ABook.title); ^ 1 error

As you can see, an error is generated when trying to access a private variable directly. But if we try to access it through a public method, it will work.

**Try to access private variable through public method:**

class Aclass

{

public static void main(String[] args)

{

*//declare a new instance of the Book class*

Book ABook = new Book();

ABook.setTitle("Greatest book ever");

*//try to access the private variable title through*

*//the public getTitle() method and it will be printed*

System.out.println("Book title: " + ABook.getTitle());

}

}

**JAVA EVENT HANDLING**

Event handling in Java refers to executing some code when specific things occur such as a window being minimized or a button being clicked.

To demonstrate event handling in action we will be using frames. A frame in Java is a standard graphical window. We will be catching window events like minimize and maximize and performing some action accordingly.

**Handling events**

There are several types of events that can happen in a Java program:

**Window events** - Occur when something happens with the program window such as maximizing the window, minimizing the window, or closing the window.

**Action events** - Occur when something happens with a component such as the clicking of a button

**Focus events** - Occur when a component gains or loses focus

**Mouse events** - Occur when something happens with the mouse such as moving the mouse or clicking the mouse

**Key events** - Occur when something happens with the keyboard such as a key is pressed or a key is released.

Each event type has it's own interface that you need to implement in a program to handle those events. These interfaces are located in the java.awt.event package.

**Window events** - WindowListener interface

**Action events** - ActionListener interface

**Focus events** - FocusListener interface

**Mouse events** - MouseListener, MouseMotionListener interfaces

**Key events** - KeyListener interface

Each interface has it's own methods to use to execute some code when certain events occur. For example, the KeyListener interface has a keyPressed method that can be used to execute some code when a key is pressed.

**Setting up the event functionality**

Let's start with a simple frame that will have window events. and then we will add event functionality to it.

import java.awt.\*;

import java.awt.event.\*;

class FrameWithEvents **implements WindowListener**

{

}

Now we need to use the methods of the WindowListener interface to specify what happens during window events.

*//Window event methods*

public void windowClosing(WindowEvent e)

{

System.out.println("The frame is closing.....");

}

public void windowClosed(WindowEvent e)

{

}

public void windowDeactivated(WindowEvent e)

{

}

............. .....................

**NOTE:** When you implement an interface, you have to define all of it's methods in your program.

**Making objects listen for events**

Now that we implemented the interface and set up the methods we need to specify which component will listen for these events and trigger the functionality accordingly. To do this, we will need to use an event listener. To use an event listener the addWindowListener() method will be used on the component that will listen for these events - the frame.

**Example**

aFrame.addWindowListener(this);

**An entire frame with events**

Here is the code for the entire frame. This frame utilizes all the window event methods. Try it and see how your command prompt will display different messages as you perform actions such as minimize and maximize on the frame.

import java.awt.\*;

import java.awt.event.\*;

class frame implements WindowListener

{

public frame()

{

Frame aFrame = new Frame();

aFrame.setSize(500, 500);

aFrame.addWindowListener(this);

aFrame.setVisible(true);

}

public static void main(String[] args)

{

frame FWE = new frame();

}

public void windowClosing(WindowEvent e)

{

System.out.println("The frame is closing.....");

//The following line of code

//specifies that the frame should be closed

((Window)e.getSource()).dispose();

}

public void windowClosed(WindowEvent e)

{

System.out.println("The frame has been closed!");

System.exit(0);

}

public void windowActivated(WindowEvent e)

{

System.out.println("The frame has been activated");

}

public void windowDeactivated(WindowEvent e)

{

System.out.println("The frame has been deactivated");

}

public void windowDeiconified(WindowEvent e)

{

System.out.println("The frame has been restored from a minimized state");

}

public void windowIconified(WindowEvent e)

{

System.out.println("The frame has been minimized");

}

public void windowOpened(WindowEvent e)

{

System.out.println("The frame is now visible");

}

}