**Unit I**

**Chapter 1 Java Fundamentals**

**The Origins of Java**

* Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank and Mike Sheridan at Sun Microsystems in 1991.
* It was initially named as “Oak” but renamed as “Java” in 1995.
* It is platform-independent language that could be used to create software to be embedded in consumer electronic devices such as toasters, microwave ovens and remote controls.
* It is portable and cross-platform language that produces code which can run on a variety of CPUs under different environments.
* It is portable and hence well suited for World Wide Web.

**How Java Relates to C and C++**

* Java is directly related to both C and C++. It inherits syntax from C and C++. This makes it easy and familiar to programmers.
* Its object model is adapted from C++.
* It is influenced by C++ but not the extended version of C++.

**How Java Relates to C#**

* Many features of C# are common to Java
* Both Java and C# share general syntax of C++
* Both support distributed programming and utilize the same object model.
* Java and C# are optimized for two different types of computing environments.

**Java’s Contribution to the Internet**

Java had a profound effect on the Internet. Java innovated new type of network program called applet. Java addressed the portability and security issues associated with the Internet.

* **Java Applets**: An Applet is a special kind of program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible browser. If the user clicks a link that contains an Applet, the applet will be automatically downloaded and run in the browser. Applet is a small program that is used to display data provided by the server, to handle user inputs or to provide simple functions such as EMI calculator that execute locally. There are two categories of objects that are transmitted between the server and the client: passive information and active or dynamic. Reading an e-mail is example for passive data. Applet is a dynamic self-executing program, yet it is initiated by the server.
* **Security**: A normal program that is downloaded may contain Viruses, Trojan horses or other harmful code which can gain an unauthorized access to the recourses. To download and execute applets safely some protection mechanism is needed. This protection is achieved by confining an applet to the Java execution environment and not allowing it access to the other parts of the computer.
* **Portability**: Portability is a major aspect of the Internet because there are different types of computers and operating systems connected to it. A Java program need to be executed on any computer connected to the Internet. It is not practical to have different versions of applet for different computers. The same code must work in all computers.

**Java’s Magic: The Bytecode**

The key that allows Java to solve both security and portability problems is the output of a Java compiler is not executable code. It is bytecode. Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called Java Virtual Machine (JVM). JVM is an interpreter for bytecode. Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only JVM needs to be implemented for each platform. Details of the JVM differ from platform to platform but all understand the same bytecode.

The fact that a Java program is executed by the JVM makes it secure. Because the JVM is in control, it can contain the program and prevent it from generating side effects outside the system. The bytecode is highly optimized and the use of bytecode enables the JVM to execute programs much faster.

**The Java Buzzwords (Features of Java)**

|  |  |
| --- | --- |
| **Simple** | Java has a concise set of features that makes it easy to learn and use |
| **Secure** | Java provides a secure means of creating Internet applications |
| **Portable** | Java contains the object-oriented programming philosophy |
| **Object-Oriented** | Java encourages error-free programming by being strictly |
| **Robust** | Java provides strict compile time and run time checking of data types, exception handling and garbage collection mechanism |
| **Multithreaded** | Java can handle multiple tasks simultaneously by using multithreaded programs |
| **Architecture-neutral** | Java is not tied to a specific machine or operating system |
| **Interpreted** | Java supports cross-platform code through the use of Java bytecode |
| **High Performance** | Java bytecode is highly optimized for speed of execution |
| **Distributed** | Java is designed with the distributed environment of the Internet in mind |
| **Dynamic** | Java is capable of dynamically linking in the new class libraries, methods and objects. |

**Object Oriented Programming**

The center of Java is OOP. All OOP languages including Java have three features in common: encapsulation, polymorphism and inheritance.

1. **Encapsulation**: This mechanism binds together the code and the data it manipulates and keeps both safe from outside interference and misuse. Java’s basic unit of encapsulation us the class. It specifies both the data and the code that will operate on that data. Objects are instances of a class. The code and data that constitute a class are called members. Data is referred to as member variables and the code is referred to as methods.
2. **Polymorphism**: It allows one interface to a general class of action. This mechanism is referenced by the phrase “one interface, multiple methods”. It is possible to define a generic interface to a group of related activities. It is the job of the compiler to select the specific action based on the situation.
3. **Inheritance**: It is the process by which one object can acquire the properties of another object. It supports the concept of hierarchical classification. Without the use of hierarchies, each object needs explicit definition of all its characteristics. Using inheritance, an object need only define those qualities unique within its class. It can inherit its general attributes from its parent.

**A Simple Java Program**

class Example{

public static void main(String args[]) {

System.out.println(“This is my First Java Program”);

}}

For most computer languages, the name of the file that holds the source code is arbitrary. However, this is not the case with Java. The Java compiler requires that a source file use the **.java** extension. So the name of the above program must be Example. In Java, all code must reside inside a class. The name if the main class should match the name of the file that holds the program.

To compile the program, execute the compiler, **javac**, specifying the name of the source file on the command line as

javac Example.java

The javac compiler creates a file called Example. Class that contains the bytecode version of the program. Bytecode is not executable code. Bytecode must be executed by a Java Virtual Machine. To run the program, Java interpreter should be used as follows: java Example

public static void main(String args[])

All Java applications begin execution by calling **main()** method. The **public** keyword is an access modifier which determines how the program can access the members of the class. If the keyword is public, then it can be accessed by code outside the class in which it is declared. In this case, main() must be declared as public, since it must be called by code outside its class when the program is started. The keyword **static** allows main() to be called before an object of the class has been created. This is necessary because main() is called by the JVM before any objects are made. The keyword void tells the compiler main() does not return any value.

Any information that is needed to be passed to a method is received by variables specified within the set of parenthesis that follow the name of the method. In main() there is only one parameter **args[]** of type array of **String**. In this case args[] receives any command-line arguments present when the program is executed.

System.out.println(“This is my First Java Program”);

This line outputs the string “This is my First Java Program” on the screen. println() is the method used to display output, **System** is a predefined class that provides access to the system and **out** is the output stream connected to the console. Thus **System.out** is an object that encapsulates console output.

**Java Keywords**

Fifty keywords are currently defined in Java. Keywords cannot be used as names for variable, class or method.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| abstract | assert | boolean | break | byte | case |
| catch | char | class | const | continue | default |
| do | double | else | enum | extends | final |
| finally | float | for | goto | if | implements |
| import | instanceof | int | interface | long | native |
| new | package | private | protected | public | return |
| short | static | strictfp | super | switch | synchronized |
| this | throw | throws | transient | try | void |
| volatile | while |  |  |  |  |

In addition to the keywords, Java reserves the following: **true**, **false** and **null**.

**Identifiers in Java**

* An identifier is a name given to a method, a variable or any other user-defined item.
* Identifier can be from one to several character long.
* It may start with an alphabet, an underscore or a dollar sign
* It cannot contain space or any other special character
* Keywords cannot be used as identifiers. Names of any standard methods like println cannot be used as identifier

**The Java Class Libraries**

Java class library contains set of pre-defined classes that contain many built-in methods. These methods provide support for I/O, string handling, networking and graphics.

**Chapter-2 Data Types and Operators**

**Java’s Primitive Data Types**

|  |  |
| --- | --- |
| **Type** | **Meaning** |
| boolean | Represents true/false values |
| byte | 8-bit integer |
| char | Character |
| double | Double-precision floating point |
| float | Single-precision floating point |
| int | Integer |
| long | Long integer |
| short | Short integer |

Java strictly specifies a range and behaviour for each primitive type, which all implementations o the JVM must support. This is because Java programs must be portable. For example, an int is same in all execution environments. This allows programs to be fully portable.

**Integers:** Java defines four integer types: byte, short, int and long. The range allows both positive and negative values. Java does not support unsigned integers.

|  |  |  |
| --- | --- | --- |
| **Type** | **Width** | **Range** |
| byte | 8 Bits | -128 to 127 |
| short | 16 Bits | -32,768 to 32,767 |
| int | 32 Bits | -2,147,483,648 to 2,147,483,647 |
| long | 64 Bits | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |

**Floating Point Types**: Floating-point types can represent numbers that have fractional components. There are two kinds of floating-point type float and double which single- and double-precision numbers respectively are. Type float is 32-bit wide and double is 64-bit wide. Of the two, double is most commonly used because all math functions in Java’s class library use double values.

**Characters**: Java uses Unicode which defines a character set that can represent all the characters found in all human languages. In Java, char is an unsigned 16-bit type having a range of 0 to 65,536. A character variable can be assigned value by enclosing the character in single quotes. For example, to assign value X to variable ch we use the following Java statement: char ch=’X’;

**The Boolean Type**: The Boolean type represents true/false values. Example: boolean b= true;

**Literals**

* Literals refer to fixed values. They are also called as constants. Integer literals are specified a numbers without fractional part. For example -100, 999 are valid integer literals. By default integer literals are of type int. If long integers are to be used, an L or l is to be appended. Example: 12L. 12 is of type int and 12L is of type long.
* By default, floating point numbers are of type double. To specify a float literal, F or f is to be appended to the constant. Example: 10.3 is of type double and 10.3F is of type float.
* A hexadecimal literal must begin with 0x or 0X where as an octal literal always begins with 0. Example hex=0X45C; oct=011;
* Java also supports sting literals. A string is a set of characters enclosed within double quotes. For example “Good Morning”

**Escape sequences**: Java supports some special backslash character constants that are used in output methods. A list of such backslash character constants are given below. Note that each of them represents one character, although they consist of two characters. These characters combinations are known as escape sequences.

‘\b’ backspace ‘\t’ horizontal tab

‘\f’ form feed ‘\’’ Single quote

‘\n’ new line ‘\”’ double quote

‘\r’ carriage return ‘\\’ backslash

**Variables**

In Java, variables are the names of storage locations. After designing suitable variable names, we must declare them to the compiler. Declaration does three things.

* It tells the compiler what the variable name is
* It specifies what type of data the variable will hold
* The place of declaration decides the scope of the variable
* A variable must be declared before it is used in the program. This is necessary because the compiler must know what type of data it contains.

Ex: **int** count; **float** height; **boolean** flag **short** sno=123;

**Dynamic Initialization**

Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

**Example:**

// Demonstration of dynamic initialization

class DynInit

{ public static void main(String args)

{ double radius=4, height=5;

double volume = 3.1416 \* radius \* radius \* height; // volume is initialized dynamically

system.out.println(“Volume = ” +volume); } }

In the above program, there are three local variables namely height, radius and volume. The first two are initialized by a constant and the volume is initialized dynamically.

**The Scope and Lifetime of Variables**

Java allows variables to be decaled within any block. Scope determines the visibility of objects to the other parts of the program. It also determines the lifetime of those objects. In Java variables can be declared within any block. Variables declared inside the block are not visible to the code which is defined outside the block.

**Example:**

// Demonstration of block scope

Class ScopeDemo{

public static void main (String args[]){

int x=10; // known to all code within main

if (x==10){ // Start a new scope

int y=20; // Known to only this block

System.out.println(“X =” +x+ ”Y =”+y); // Both x and y are visible here4

}

Y=100; // Error!! Y is not known here

System.out.println(“X =” +x); // x is still known here } }

**Operators**

An operator is a symbol that tells the compiler to perform a specific mathematical or logical manipulation. Java has four general classes of operators: arithmetic, bitwise, relational and logical. Java also defines some additional operators that handle certain special situations.

**Arithmetic Operators:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Meaning** | **Operator** | **Meaning** |
| + | Addition (also unary plus) | % | Modulus |
| - | Subtraction(also unary minus) | ++ | Increment |
| \* | Multiplication | -- | Decrement |
| / | Division |  |  |

These can be applied to any built-in numeric data types. They can also be used on objects of type char. When / is applied to integers, the result will be an integer. So 10/3 will be 3. The increment operator ++ adds 1 to its operand and the decrement operator – subtracts 1. Both increment and decrement operators either prefix or follow the operand. There is no difference whether the increment is applied as prefix or postfix. However, when they are a part of an expression, there is a difference.

x=10; x=10;

y=++x; // In this case y becomes 11 y=x++; // y= 10 but x= 11

**Relational and Logical Operators**

|  |  |  |  |
| --- | --- | --- | --- |
| **Relational Operators** | | **Logical Operators** | |
| **Operator** | **Meaning** | **Operator** | **Meaning** |
| == | Equal to | & | AND |
| !- | Not equal to | | | OR |
| > | Greater than | ^ | XOR (Exclusive OR) |
| < | Less than | || | Short circuit OR |
| >= | Greater than or equal to | && | Short circuit AND |
| <= | Less than or equal to | ! | NOT |

Relational operators can be applied to all numeric as well as char data types. For logical operations, operands must be of type Boolean and the result of logical operation is of type Boolean.

Logical operators work according to the following truth table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **P** | **Q** | **P & Q** | **P | Q** | **P ^ Q** | **!P** |
| False | False | False | False | False | True |
| False | True | False | True | True | True |
| True | False | False | True | True | False |
| True | True | True | True | False | False |

Short circuit operators work the same way as their normal counterparts. Only difference is that the normal operands will always evaluate each operand but short circuit versions will evaluate the second operand only when necessary.

**The Assignment Operator**

The assignment operator is =. The general form is var=expression; the type of var must be compatible with the type of expression.

Example: int x, y,z;

x=y=z=100; // will set 100 to variables x,y and z

**Shorthand Assignments**

An expression of the kind x=x+10;can be written as x+=10; The operator pair += tells the compiler to assign x+10 to the variable x. Java has the following arithmetic and logical shorthand operators.

|  |  |  |  |
| --- | --- | --- | --- |
| += | -= | \*= | /= |
| %= | &= | |= | ^= |

**Type Conversion in Assignments**

When compatible types are mixed in an assignment, the value of the right side is automatically converted to the type of the left side. Not all types are compatible in Java. For example, int and boolean are not compatible.

When one type of data is assigned to another type of variable, an automatic type conversion will take place if

* The two types are compatible.
* The Destination type is larger than the source type.

For example, int type is always large enough to hold all valid byte values, and both int and byte are integer types, so an automatic conversion byte to int will take place.

**Example:**

//To demonstrate type conversion

class TypeDemo{

public static void main(String args[]) {

{ long L, double D;

L=123456L;

D=L; // Automatic conversion from long to double;

D=123456.0;

L=D; //Illegal. No automatic type conversion from double to long } }

**Casting Incompatible Types**

A cast is an instruction to the compiler to convert one type into another. It has the following form:

*(target-type) expression;*

Here, target-type specifies the desired type to convert the specified expression to.

Example:

double x,y;

// --------

(int)(x/y);

Here, even though x and y are of type double, the caste converts the outcome of the expression to int. The parentheses surrounding x/y are necessary. Otherwise, the cast to int would apply only to x. The cast is necessary here because there is no automatic conversion from double to int.

When a cast involves a narrowing conversion, information might be lost. For example, when casting a long into short, information will be lost if the value held in long type of variable is greater than the range of short because higher order bits are removed. When floating point value is cast to an integer type, the fractional component will be lost due to truncation. If 1.23 is assigned to an integer, the resulting value will be 1.

**Operator Precedence**

++ (Postfix), --(Postfix) Highest

++ (Prefix) --(Prefix) ~ ! + (Unary) – (Unary) (type-cast)

\* / %

+ -

>> >>> <<

> > = < <= instance of

== !=

&

^

|

&&

||

?:

= += -= \*= /= %= &= |= Lowest

**Type Conversion in Expressions**

It is possible to mix two or more different types of data in an expression. When different types of data are mixed, they are converted to the same data type. This is through Java’s type promotion rules. Char, byte , short values are promoted to int. If one of the operand is long, the whole expression is promoted to long. If one operand is of type float, the entire expression is promoted to float. If any of the operand is double, the result is double. Type promotion only affects the evaluation of expression.

**Example**:

//To demonstrate type conversion in expressions

class Demo{

public static void main(String args[]) {

{

byte b; int i;

No cast is needed because result is already evaluated to int.

b=10;

Cast is needed because result is already evaluated to int

i=b\*b;

b=10;

b=(byte)(b\*b);

}

}

**I/O Basics**

Most real applications of Java are not text-based, console programs. Rather, they are graphically oriented programs that rely upon Java’s Abstract Window Toolkit (AWT) or Swing for interaction with the user. Although text-based programs are excellent as teaching examples, they do not constitute an important use for Java in the real world. Also, Java’s support for console I/O is limited and somewhat awkward to use—even in simple example programs. Text-based console I/O is just not very important to Java programming.

**Streams**

Java programs perform I/O through streams. A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection. Streams are a clean way to deal with input/output. Java implements streams within class hierarchies defined in the **java.io** package.

**Byte Streams and Character Streams**

Java defines two types of streams: byte and character. *Byte streams* provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. *Character streams* provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.

**The Byte Stream Classes**

Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**. Each of these abstract classes has several concrete subclasses that handle the differences between various devices, such as disk files, network connections, and even memory buffers. To use the stream classes, **java.io**. must be imported

The abstract classes **InputStream** and **OutputStream** define several key methods thatthe other stream classes implement. Two of the most important are **read( )** and **write( )**,which, respectively, read and write bytes of data. Both methods are declared as abstractinside **InputStream** and **OutputStream**. They are overridden by derived stream classes.

**The Character Stream Classes**

Character streams are defined by using two class hierarchies. At the top are two abstract classes, **Reader** and **Writer**. These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these.

The abstract classes **Reader** and **Writer** define several key methods that the other streamclasses implement. Two of the most important methods are **read( )** and **write( )**, which read and write characters of data, respectively. These methods are overridden by derived stream classes.

**The Predefined Streams**

All Java programs automatically import the **java.lang** package. This package defines a class called **System**, which encapsulates several aspects of the run-time environment. **System** also contains three predefined stream variables: **in**, **out**, and **err**. These fields are declared as **public**, **static**, and **final** within **System**. This means that they can be used by any other part of the program and without reference to a specific **System** object.

**System.out** refers to the standard output stream. By default, this is the console. **System.in** refers to standard input, which is the keyboard by default. **System.err** refers to the standarderror stream, which also is the console by default. However, these streams may be redirectedto any compatible I/O device.

**System.in** is an object of type **InputStream**; **System.out** and **System.err** are objects of type **PrintStream**. These are byte streams, even though they typically are used to read and write characters from and to the console. As you will see, you can wrap these within character based streams, if desired.

**Reading Console Input**

In Java 1.0, the only way to perform console input was to use a byte stream, and older code that uses this approach persists. Today, using a byte stream to read console input is still technically possible, but doing so is not recommended. The preferred method of reading console input is to use a character-oriented stream, which makes your program easier to internationalize and maintain.

In Java, console input is accomplished by reading from **System.in**. To obtain a character based stream that is attached to the console, wrap **System.in** in a **BufferedReader** object. You can not construct a Buffered Reader directly fron S**ystem.in**. Instead, you must first convert it into a character stream. To do this, you will use **InputStreamreader** object that is linked to **System.in**, use the constructor shown next:

InputStreamReader(InputStream *inputStream*)

Next, using the object produced by **InputStreamReader,** construct a **BufferedReader** using the constructor shown here:

BufferedReader(Reader *inputReader*)

Here, *inputReader* is the stream that is linked to the instance of **BufferedReader** that is being created. **Reader** is an abstract class. One of its concrete subclasses is **InputStreamReader**, which converts bytes to characters. Because **System.in** refers to an object of type **InputStream**, it can be used for *inputStream.* Putting it all together, the following line of code creates a **BufferedReader** that is connected to the keyboard:

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

After this statement executes, **br** is a character-based stream that is linked to the console through **System.in**.

**Reading Characters**

To read a character from a **BufferedReader**, use **read( )**. The version of **read( )** that we will be using is

int read( ) throws IOException

Each time that **read( )** is called, it reads a character from the input stream and returns it as an integer value. It returns –1 when the end of the stream is encountered.

The following program demonstrates **read( )** by reading characters from the console until the user types a “q” Notice that any I/O exceptions that might be generated are simply thrown out of **main( )**.

// Use a BufferedReader to read characters from the console.

import java.io.\*;

class BRRead

{

public static void main(String args[]) throws IOException

{

char c;

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

System.out.println("Enter characters, 'q' to quit.");

// read characters

do {

c = (char) br.read();

System.out.println(c);

} while(c! = 'q');

} }

**Reading Strings**

To read a string from the keyboard, use the version of **readLine( )** that is a member of the **BufferedReader** class. Its general form is shown here:

String readLine( ) throws IOException

It returns a **String** object.

The following program demonstrates **BufferedReader** and the **readLine( )** method; the program reads and displays lines of text until you enter the word “stop”:

// Read a string from console using a BufferedReader.

import java.io.\*;

class BRReadLines {

public static void main(String args[])

throws IOException

{

// create a BufferedReader using System.in

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

String str;

System.out.println("Enter lines of text.");

System.out.println("Enter 'stop' to quit.");

do {

str = br.readLine();

System.out.println(str);

} while(!str.equals("stop")); }

}

**Writing Console Output**

Console output is most easily accomplished with **print( )** and **println( )**, described earlier, which are used in most of the examples in this book. These methods are defined by the class **PrintStream** (which is the type of object referenced by **System.out**). Even though **System.out** is a byte stream, using it for simple program output is still acceptable. However, a character-based alternative is described in the next section. Because **PrintStream** is an output stream derived from **OutputStream**, it also implements the low-level method **write( )**. Thus, **write( )** can be used to write to the console. The simplest form of **write( )** defined by **PrintStream** is shown here: void write(int *byteval*) This method writes to the stream the byte specified by *byteval.* Although *byteval* is declared as an integer, only the low-order eight bits are written. Here is a short example that uses **write( )** to output the character “A” followed by a newline to the screen:

// Demonstrate System.out.write().

class WriteDemo {

public static void main(String args[]) {

int b;

b = 'A';

System.out.write(b);

System.out.write('\n');

}

}

**Program Control Statements**

**Input Characters from the Keyboard**

To read a character from the keyboard, System.in.read() can be used. System.in is the input object attached to the keyboard. The read() method waits until the user presses a kay and then returns the result. The character is returned as integer. So it must be cast into char to assign it to a variable of type char. By default, console input is line buffered. The term buffer here refers to a small portion of memory that is used to characters before they are read by a program. When the user presses ENTER key, the character is returned to the program.

**Example:**

// Read a character from the keyboard

class KbInput{

public static void main(String args[]) throws java.io.IOException

{

char ch;

System.out.println(“Type a character !!!! “);

ch=(char) in.read(); // This reads a character from the keyboard

System.out.println(“Your Key is “ +ch); } }

Because System.in.read() is being used, the program must specify the throws java.io.IOException clause. This line is necessary to handle input errors.

**The simple if Statement**

The if statement is a powerful decision making statement and is used to control the flow of execution of statements. It takes the following form:

if(test-expression)

{

statement-block;

}

statement-x;

The statement-block may be a single statement or a group statements. If the test-expression is true, the statement-block will be executed; otherwise the statement-block will be skipped and the execution will jump to statement-x;

**Example:**

if (category==SPORTS)

marks=marks+bonus\_marks;

System.out.println(“Marks = ” +marks);

**The if ..... else Statement**

if(test-expression)

{ True-block statements; }

else

{ False-block statements; }

statement-x;

If the test-expression is true, then the true-block statements are executed; otherwise, the false-block statements are executed. In either case, either true-block or false-block will be executed, not both. In both the cases, the control will be transferred subsequently to statement-x.

**Example:**

if (code==1)

boy=boy+1;

else

girl=girl+1;

**The Nested if Statement**

A nested if statement is an if statement that is the target of another if or else.

Syntax:

if (test condition-1)

{ if (test condition-2)

{ statement-1; }

else

{ statement-22; }

else { statement-3;}

statement-x;

If condition-1 is false, then statement-3 will be executed; otherwise it checks condition-2. If condition-2 is true, statement-1 will be executed; otherwise statement-2.

**Example:**

if(gender=’F’)

{ if (balance>5000)

bonus=0.05 \* balance;

else

bonus=0.02 \* balance; }

else

bonus=0.03 \* balance;

balance=balance+bonus;

**The if .... else if Ladder**

if(condition-1)[

statement-1;

else if (condition-2)

statement-2;

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

else if (condition-n)

statement-n;

else

default-statement;

statement-x;

This is known as the else if ladder. The conditions are evaluated from top to downwards. As soon as true condition is found, the statement associated with it is executed and the control is transferred to statement-x. When all conditions become false, then the final else containing the default statement will be executed.

Example:

// Demonstrate if-else-if ladder

class Ladder {

public static void main (String args[]){

for(int x=0; x<6; x++) {

**Output:**

x is one

x is two

x is three

x is four

x is not between 1 and 4

if(x==1)

System.out.println(“x is one”);

else if (x==2)

System.out.println(“x is two”);

else if (x==3)

System.out.println(“x is three”);

else if (x==4)

System.out.println(“x is four”);

else

System.out.println(“x is not between 1 and 4); }}}

**The switch Statement**

The test expression is continuously tested against a list of constants. When a match is found, the statement sequence associated with that match is executed. The general format of switch statement:

switch(expression)

{

case value-1: block-1;

break;

case value-2: block-2;

break;

...........

default: default-block;

}

statement-x

The expression controlling the switch must be of byte, short int or char. Each value specified in the case statement must be unique. If a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed. The break statement signals the end of particular case and transfers the control to statement-x following the switch.

The default is an optional case. When present, it will be executed if the value of the expression does not match with any of the case values. If not present, no action takes place when all matches fail and the control goes to statement-x.

**Output:**

i is zero

i is one

i is two

i is three

i is four

i is five or more

i is five or more

i is five or more

i is five or more

i is five or more

**Example**:

//Demonstrate switch statement

class SwitchDemo{

public static void main(String args[]){

for(int i=0; i< 10; i++)

switch(i) {

case 0 : System.out.println(“i is zero”);

break;

case 1 : System.out.println(“i is one”);

break;

case 2: System.out.println(“i is two”);

break;

case 3 : System.out.println(“i is three”);

break;

case 4 : System.out.println(“i is four”);

break;

default: System.out.println(“i is five or more ”); } } }

Technically break statement is optional. When included within a case, it causes the program flow to exit from the entire switch statement and resume at the next statement outside the switch. However, if break is not present with a case, then all statements following the matching case will be executed until a break or end of switch is encountered.

**Example**:

//Demonstrate switch statement without break

class NoBreak

{

public static void main(String args[])

{

for(int i=0; i<=5; i++)

{

switch(i)

{

case 0 : System.out.println(“i is less than 1”);

**Output:**

i is less than 1

i is less than 2

i is less than 3

i is less than 4

i is less than 5

i is less than 2

i is less than 3

i is less than 4

i is less than 5

i is less than 3

i is less than 4

i is less than 5

i is less than 4

i is less than 5

i is less than 5

case 1 : System.out.println(“i is less than 2”);

case 2 : System.out.println(“i is less than 3”);

case 3 : System.out.println(“i is less than 4”);

case 4 : System.out.println(“i is less than 5”);

}

System.out.println();

}

} }

We can have empty cases as shown in the example below

switch(i)

{ case 0 :

case 1 :

case 2 : System.out.println(“i is 0 1 or 2”);

break;

case 3 : System.out.println(“i is 3”); }

**The for Loop**

The general form of for loop for repeating a single statement is

for(initialization;condition;iteration) statement;

For repeating a block, the general form is

for(initialization;condition;iteration)

{ statement- block ; }

The initialization is usually an assignment statement that sets the initial value of the loop control variable. The condition is a Boolean expression that determines whether or not the loop will repeat. The iteration expression defines the amount by which loop control variable will change each time the loop is repeated. These three sections must be separated by semicolons. The for loop will continue to execute as long as the condition evaluates to true. Once the condition becomes false, loop will be terminated and control will be transferred to the statement following for.

**Example:**

//Demonstration of for loop

class ForDemo{

public static void main(String args[]) {

int i;

for(i=0; i<5; i++)

System.out.print(i+ “ ”);} } Output: 0 1 2 3 4

For loop can proceed in a positive or negative fashion.

**Example:**

//Demonstration of for loop

class NegFor{

public static void main(String args[]) {

int i;

for(i=10; i>5; i--)

System.out.print(i+ “ ”);} } Output: 10 9 8 7 6

In case of for loop, the condition always gets tested in the beginning. This means that, the code inside the loop may not be executed at all if the condition is false in the beginning.

Example:

for(count=10; count <5; count++)

x+=count; // This statement will not get executed at all

**Some Variations on the for Loop**

We can use multiple loop control variables in a for loop. Here commas separate the two statements.

**Example:**

Output

i=0 j=10

i=1 j=9

i=2 j=8

i=3 j=7

i=4 j=6

class MulFor{

public static void main(String args[])

{

int i,j;

for(i=0, j=10; i<j; i++,j--)

System.out.print(“i= ” +i +” j= ”+j);

} }

The condition controlling the loop can be any valid Boolean expression. It does not need to involve the loop control variable.

**Example:**

// Loop until ‘S’ is pressed

class ForTest

{

public static void main(String args[]) throws java.io.IOException{

int i;

for(i=0, (char)System.in.read()!=’S’; i++)

System.out.print(“i= ” +i);

} }

**Missing Pieces**

In Java, it is possible for any or all of the initialization, condition or iteration portions of the for loop to be blank.

**Example-1:**

//Demonstrate for loop with some empty parts

class Empty

{

public static void main(String args[]) {

int i;

for(i=0; i<10;) { // Iteration expression is missing here

System.out.println(“Pass ”+i); i++;

}}}

**Example-2:**

//Demonstrate for loop with some empty parts

class Empty2

{

public static void main(String args[]) {

int i=0;

for(; i<10;)

{ // Initialization and iteration expressions are missing

System.out.println(“Pass ”+i); i++;

}}}

Placing initialization outside the loop is done only when initial value is derived through complex process which can not be contained in the for loop.

**The Infinite Loop**

An infinite loop can be created using for by leaving conditional expression empty. The structure of an infinite loop is as follows:

for(;;) // Intentionally infinite loop

{ .......

}

This loop will run for ever. Use of break statement inside such loop can terminate the loop.

**Loops with No Body**

In Java, the body associated with a for loop can be empty.

//Demonstrate for loop with no body

class Empty3{

public static void main(String args[]) {

int i, sum=0;

for(i=0; i<=10; sum+=i++);

System.out.println(“Sum = ”+sum); }}

**Declaring Loop Control Variables Inside the for Loop**

Often the variable that controls a for loop is only needed for the purposes of the loop and is not used elsewhere. When this is the case, it is possible to declare the variable inside the initialization portion of the for loop.

**Example:**

// Declare a loop control variable inside the for.

class ForVar {

public static void main(String args[]) {

int sum=0,fact=1;

for(int i=1;i<=5;i++) // i is declared inside the loop

{ sum+=i;

fact\*=i; // i is available only till here }

System.out.println("Sum = " + sum); // i is not known to this part

System.out.println("Factorial = " + fact); } }

When a variable is declared inside a for loop, the scope of that variable ends when the for statement does. Outside the for loop, the variable will cease to exist. If the variable is to be used elsewhere in the program, it has to be declared outside the for loop.

**The while Loop**

**Syntax**: while(condition) statement;

Where the statement may be a may be a single statement or a block of statements. Condition is a valid Boolean expression that controls the loop. Loop repeats while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.

**Example:**

//Demonstration of while loop

class WhileDemo

{

public static void main(String args[]) {

char ch= ‘a’;

while (ch<= ‘z’){

System.out.print (ch + “ ”);

ch++;

} } }

**The do-while Loop**

In do... while loop, condition is always tested at the bottom of the loop. This means that do-while loop will always execute at least once. The general form of do-while is

do

{

statements;

} while(condition);

Braces are not necessary when only one statement is present, but they are used to increase the readability. The do-while loop executes as long as the conditional expression is true.

// Demonstrate the do-while loop

class DoWhileDemo

{

public static void main(String args[])throws java.io.IOException {

char ch;

do {

System.out.println(“Enter a character”);

ch=(char) System.in.read();

} while (ch!=’q’);

} }

**Use of break to Exit a Loop**

It is possible to force an intermediate exit from a loop, bypassing any remaining code in the body of the loop using break statement. When a break statement is encountered inside a loop, the loop is terminated and the program control resumes at the next statement following the loop.

**Example:**

//Using break to exit the loop

class BreakDemo{

public static void main(String args[]) {

int num=100;

for(int 1=0;i<num;i++){

if(i\*i>=num) break; // terminate loop if i\*i >=100

System.out.println(i + “ ”); } }

Break statement can be used with any of Java’s loops including infinite loop.

//Read input until a q is received

class Break2{

public static void main(String args[]) {

char ch;

for(;;){

ch=(char) System.in.read();

if(ch==’q’) break; // terminate loop if ch is q }

System.out.println(“q is pressed ”); } }

**Labelled break Statement (break as a form of goto)**

In Java, we can give a label to a block of statements. A label is any valid Java variable name. Label is given to a loop by placing it before the loop with a colon at the end.

loop1: for(..........)

{ .....................;

break loop1;

}

**//using break with a label**

class Break4

{

Public static void main(String args[])

**The output is,**

i is 1

After block one.

i is 2

After block two.

After block one.

i is 3

After block three.

After block two.

After block one.

After for.

{

int i;

for(i=1;i<4;i++)

{

one:{

two: {

three: {

system.out.println(“\n i is “+i);

if(i==1) break one;

if(i==2) break two;

if(i==3) break three;

// this is never reached

System.out.println(“won’t print”);

}

System.out.println(“After block three.”);

}

System.out.println(“After block two.”);

}

System.out.println(“After block one.”);

}

System.out.println(“After for”);

}

}

Precisely where you put a label is very important-especially when working with loops. For example:

class Break6{

public static void main(String args[])

{

int x=0,y=0;

Stop1: for(x=0;x<5;x++){

for(y=0;y<5;y++) {

if(y==2) break stop1;

System.out.println(“x and y: “+ x + ” ”+ y);

}

}

System.out.println();

// now, put label immediately before

{

for(x=0;x<5;x++)

stop2: {

for(y=0;y<5;y++) {

if(y==2) break stop2;

System.out.println(“x and y: “+ x + ” ”+y); }} }}

**The output is,**

x and y : 0 0

x and y : 0 1

x and y : 0 0

x and y : 0 1

x and y : 1 0

x and y : 1 1

x and y : 2 0

x and y : 2 1

x and y : 3 0

x and y : 3 1

x and y : 4 0

x and y : 4 1

**The continue Statement**

The continue statement forces the next iteration of the loop to take place skipping any code between itself and the conditional expression that controls the loop. The following program uses continue to print even numbers from 1 to 100:

**Example:**

//Use of continue

Output:

2

4

6

8 ........ .100

class contDemo {

public static void main(String args[]) {

for(int i=1;i<=100;i++) {

if((i%2)!=0) continue;

System.out.println( i); } } }

**Use of continue and break statements**

// Demonstration of labelled continue

Output:

\*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

class BreakCont{

public static void main(Sting args[]) {

LOOP1: for(int i=1;i<=100;i++) {

System.out.println( “ ”);

if(i>=6) break;

for(int j=1; j<100;j++) {

System.out.print( “ \* ”);

if(j==i) continue LOOP1 } } } }

**Nested Loops**

Output:

Factors of 2 :

Factors of 3 :

Factors of 4 : 2

Factors of 5 :

Factors of 6 : 2 3

Factors of 7 :

Factors of 8 : 2 4

Factors of 9 : 3

Factors of 10: 2 5

Writing one loop inside the scope of other loop is called nesting of loops.

// Demonstration of nested loops

class Nest{

public static void main(String args[]) {

for(int i=2;i<=10;i++) {

System.out.print(“Factots of “ + i +”: ”);

for(int j=2; j<i;j++)

if(i%j==0) System.out.print(j+ “ ”);

System.out.println();} } }

**Unit II**

**Arrays**

An array is a collection of variables of the same type, referred to by a common name. An array organizes the data in such a way that it can be easily manipulated.

**One-Dimensional Arrays**

A one dimensional array is a list of related variables. It can be declared using following format:

type array-name[]=new type[size];

Here, type declares the element type of the array. The element type determines the data type of each element contained in the array. The number of elements that the array will hold is determined by size. Since arrays are implemented as objects, the creation of an array is a two-step process.

1. An array reference variable is declared
2. Memory is allocated to the array declared.

int sample[]=new int[10];

It is possible to break the declaration into two steps as follows:

int sample[];

sample=new int[10];

Individual elements can be accessed by the use of an index. An index describes the position of an element within an array. Array index starts from 0.

Output

This is Sample[0] : 0

This is Sample[1] : 1

This is Sample[2] : 2

This is Sample[3] : 3

This is Sample[4] : 4

This is Sample[5] : 5

This is Sample[6] : 6

This is Sample[7] : 7

This is Sample[8] : 8

This is Sample[9] : 9

**Example**

//Demonstrate one dimensional array

class ArrayDemo{String args[]) {

int sample[]=new int[10];

int i;

for(i=0;i<10;i++)

sample[i]=i;

for(i=0;i<10;i++)

System.out.println(“This is Sample[ ”+i+“] :”+sample[i]);

} }

An array can be initialized at the time of creation. The general format is

type array-name[] ={ val1, val2, val3, ....., valN}

Here, the initial values are specified bay val1 through valN. They are assigned in sequence left to right in index order. Java automatically allocates an array large enough to hold the initializers. There is no need to explicitly use new operator.

**Example:** //Demonstrate Use of Array initializers

class Minimax {

public static void main(String args[]) {

int nums[]={ 99, 0, -10, 1023, 19, -99, 5678, 800, 784, -9 };

int min, max;

min=max=nums[0];

for(int i=;i<10;i++) {

if(max<nums[i]) max=nums[i];

if(min>num[i]) min=num[i]); }

System.out.println(“Min and Max : ” +min +” “ +max); } }

Array boundaries are strictly enforced in Java; it is a run-time error to overrun or underrun the end of an array.

// Demonstrate Array overrun

class ArrayErr{

public static void main (String args[]) {

int sample[]=new int[10];

int i;

for(i=0;i<100;i++) // This will generate ArrayIndexOutOfBoundsException

sample[i]=i; } }

**Two-Dimensional Arrays**

Simplest form of multi-dimensional array is two dimensional array. It can be declared as follows:

int table[][]=new int[10][10];

**Example:**

//Demonstrate two dimensional array

class TwoD{

public static void main(String args[]) {

int t, i;

int table[][]=new int[3][4];

for(t=0;t<3;t++) {

for(i=0;i<4;i++) {

table[t][i]=t\*4+i+1;

System.out.print(table[t][i] + “ ”); } }

System.out.println(); } }

**Irregular Arrays**

When allocating memory for a multidimensional array, only the memory for the leftmost dimension needs to be specified. The remaining dimensions can be allocated separately. We need not allocate the same number of elements for each index. The length of each array can be controlled by the programmer.

**Example**

//Demonstrate irregular arrays

class ArrDemo2{

public static void main(String args[]) {

int riders[][]=new int[6][];

riders[0]= new int[10];

riders[1]= new int[10];

riders[2]= new int[10];

riders[3]= new int[10];

riders[4]= new int[5];

riders[5]= new int[5];

int i,j;

for(i=0;i<4;i++)

for(j=0;j<10;j++)

riders[i][j]=i+j+10; // initializing with s

for(i=4;i<6;i++)

for(j=0;j<5;j++)

riders[i][j]=i+j\*10;

for(i=0;i<4;i++){

for(j=0;j<10;j++)

System.out.print(riders[i][j]);

System.out.println(); }

for(i=4;i<6;i++) {

for(j=0;j<5;j++)

System.out.print(riders[i][j]);

System.out.println(); } } }

**Arrays of Three or more Dimensions**

Java allows arrays with three or more dimensions. The general format is:

type name[][][] .....[]=new type[size1][size2].......[sizeN];

Example: int muldim[][][]=new int[4][10][3];

**Alternate Array Declaration Syntax**

There is a second form that can be used to declare an array:

type [] var-name;

The following statements are equivalent:

int count[] = new int[10];

int [] count = new int[10]; // Both are one and the same

The alternative declaration form offers convenience when declaring several arrays at the same time.

For example:

int[] num1, num2, num3; // This statement declares 3 arrays.

**Assigning Array References**

When one array reference variable is assigned to other, we are changing what object that variable refers to.

Example

classArrRef{

public static void main(String args[]) {

int i;

int num1[]=new int[10];

int num2[]=new int[10];

for (i=0;i<10;i++)

num1[i]=i;

for (i=0;i<10;i++)

Output

Here is num1 0 1 2 3 4 5 6 7 8 9

Here is num2 -1 -2 -3 -4 -5 -6 -7 -8 -9

Here is num2 after change

0 1 2 3 4 5 6 7 8 9

Here is num2 after change

0 1 2 -99 4 5 6 7 8 9

num2[i]= -i;

System.out.print(“Here is num1 :”);

for (i=0;i<10;i++)

System.out.print (num1[i] + “ ”;

System.out.println();

System.out.print(“Here is num2: ”);

for (i=0;i<10;i++)

System.out.print (num2[i] + “ ”);

System.out.println();

num2=num1;

System.out.println(“Here is num2 after change”);

for (i=0;i<10;i++)

System.out.print (num2[i] + “ ”);

System.out.println();

num2[3]=-99;

System.out.println(“Here is num2 after change”);

for (i=0;i<10;i++)

System.out.print (num2[i] + “ ” ); }}

**Using the Length Member**

Because, arrays are implemented as objects, the length instance variable associate with each array contains the number of elements that the array can hold.

**Example:**

//Demonstrate Array Length

class LengthDemo{

public static void main(String args[]){

int list[]={ 1, 0, 99, 8, -7, 74};

for(int i=0; i<list.lenght;i++)

System.out.print(list[i] + “ ”) } }

**The For-Each Style for Loop**

When working with arrays, it is common to encounter situations in which each element in an array must be examined from start to finish. Java defines a second form of for loop that streamlines this operation. This form is “for-each” style loop. It cycles through an array in a strictly sequential manner from start to finish. The general form of this loop is :

for(type itr-var : collection) statement-block

Here, type specifies the type, and itr-var specifies the name of the iteration variable that will receive the elements from a collection, one at a time, from beginning to end. With each iteration of the loop, the next element in the collection is retrieved and stored in itr-var. The loop repeats until all the elements in the collection are obtained. The type must be same or compatible with that of array elements.

class ForEach{

public static void main(String args[]){

Output

1. 2 3 4 5 6 7 8 9 10

Sum= 55

int list[]={ 1,2,3,4,5,6,78,9,10};

sum=0;

for(int x:list)

{ System.out.print(x + “ ”);

sum+=x; }

System.out.prinln(“\n” + “Sum= ”+sum); } }

Although the for-each loop iterates until all elements in the array have been examined, it is possible to terminate the loop early by using a break statement.

for(int x:list)

{ System.out.print(x + “ ”);

sum+=x; }

if(x==5) break; } }

The iteration variable of for-each loop is read only. We can’t change the contents of the array by assigning the iteration variable a new value.

class Nochange{

public static void main(String args[]){

int list[]={ 1,2,3,4,5,6,78,9,10};

Output

1 2 3 4 5 6 7 8 9 10

1 2 3 4 5 6 7 8 9 10

for(int x:list)

{ System.out.print(x + “ ”);

x=x\*10; }

Sustem.out.println();

for(int x: list)

System.out.print(x + “ ”); } }

**Strings**

In Java, strings are objects. The syntax for creating strings is given below:

String str = new String(“Hello”);

String str2 = new String(str1);

**Operating on Strings**

|  |  |
| --- | --- |
| boolean equals(str) | Returns true if the invoking string contains same character sequence as str |
| int length() | Obtains the length of the string |
| char charAt(index) | Obtains the character at specified index |
| int compareTo(str) | Returns less than zero if the invoking string is less than str, greater than zero if the invoking string is greater than str and zero if the strings are equal |
| int indexOf(str) | Searches the invoking string for the substring specified by str. Returns the index of the first match or -1 on failure |
| int lastIndexOf(str) | Searches the invoking string for the substring specified by str. Returns the index of the last match or -1 on failure |

class StrOp {

public static void main(String args[]) {

String str1= “ Java is #1 programming language”;

String str2=new String(Str1);

String str3= “Java Strings are powerful”;

int res, idx;

char ch;

System.out.println(“Length of str1 = “ +str1.lenght());

for(int i=0; i<str1.length(); i++)

System.out.print(str1.charAt(i));

System.out.println();

if(str1.equals(str2))

System.out.println(“str1 is equal to str2 ”);

else

System.out.println(“str1 is not equal to str2 ”);

if(str1.equals(str3))

Output

Length of str1 = 31

str1 is equal to str2

str1 is not equal to str3

srt1 is greater than str3

The first occurrence of One 0

The last occurrence of One 14

System.out.println(“str1 is equal to str3 ”);

else

System.out.println(“str1 is not equal to str3 ”);

res=str1.compareTo(str3);

if(res==0)

System.out.println(“ str1 and str3 are equal”);

else if(result < 0)

System.out.println(srt1 is less than str3);

else

System.out.println(srt1 is greater than str3);

str2=”One Two Three One”;

idx=str2.indexOf(“One”);

System.out.println(“The first occurrence of One” +idx);

idx=str2.lastIndexOf(“One”);

System.out.println(“The last occurrence of One” +idx); } }

**Using Command-Line Arguments**

In certain cases, we may need to provide inputs at the time of execution. This is achieved in Java by using command line arguments. Command line arguments are parameters that are supplied to a program at the time of execution.

**Example**

// Demonstrate Command Line arguments

class ComLineTest {

public static void main(String args[]) {

int count;

count=args.length;

System.out.println(“There are ” +count “ Number of arguments”);

for(int i=0;i<count;i++)

System.out.println( i + “ : Java is ” +args[i]) + ” !”); } }

After compilation, we can supply arguments to the example above. Any arguments provided in the command line are passed to the array args[] as its elements. We can simply access the array elements the array elements and use them in the program as we wish.

The above program can be executed as follows:

java ComLineTest Simple Obejet\_Oriented Distributed Robust Secure Portable Multithreaded Dynamic

Upon execution, the command line arguments Simple, Object\_Oriented etc are passed to the program through the array args. That is, args[0] contains Simple, args[1] contains Object\_Oriented and so on. These elements are accessed using the loop variable i **as name =args[i];**

The output will be

There are 8 Number of arguments

1 : Java is Simple !

2: Java is Object\_Oriented !

3: Java is Distributed !

4: Java is Robust !

5: Java is Secure !

6: Java is Portable !

7: Java is Multithreaded !

8: Java is Dynamic !

**Classes Objects and Methods**

**Class Fundamentals**

Java is a true Object-Oriented language and therefore the underlying structure of all Java programs is classes. Anything we wish to represent in a Java program must be encapsulated in a class that defines the state and behaviour of the basic program components known as objects. A class contains both data (referred to as attributes), and executable code (referred to as methods). Classes create objects and objects use methods to communicate between them.

Classes provide a convenient method for packing together a group of logically related data items and functions that work on them. In Java, the data items are called fields and the functions are called methods. Calling a specific method in an object is described as sending the object a message or message passing.

A class is a template that defines the form of an object. It specifies both the data and the code that will operate on the data. Java uses a class specification to construct objects. Objects are instances of a class. Class is a logical representation. The physical representation of that class will be created when an object is created.

**Declaration**

A class is a user-defined data type with a template that serves to define its properties. Once the class type has been defined, we can create ‘variables’ of that type using declarations that are similar to the basic type declarations. In Java, these variables are termed as *instances* of classes. Which are the actual *objects*. Unlike data structures in other languages which only contain data, Java classes consist of both *attributes* and *behaviors*. *Attributes* represent the data that is unique to an instance of a class, while *behaviors* are methods that operate on the data to perform useful tasks.

A class is created by using the keyword class. The general form is as follows:

class classname

{

type var1;

type var2; ......

type var n;

return-type method-name-1(arguments)

{

body of the method ;

}

return-type method-name-2(arguments)

{

body of the method ;

} ......

return-type method-name-n(arguments)

{

body of the method ;

}

}

The class with only data fields has no life. The objects created by such a class cannot respond to any message. Methods are necessary for manipulating data contained in a class. Methods are declared inside the body of the class after declaring instance variables. Method declaration should have return type, method name list of parameters and body. The method name main() is reserved. A method contains one or more Java statements but performs only one task.

**Example:**

class Rectangle

{

int length, width;

int area()

{

int a=height \* width;

return a;

}

}

**Creating Objects**

An object in Java is essentially a block of memory that contains space to store all the instance variables. Creating an object is also referred to as *instantiating* an object.

Objects in Java are created using the ***new*** operator. The ***new*** operator creates an object of the specified class and returns a reference to that object. This reference is then stored in a variable.

**Example:**

Rectangle rect1;

rect1= new Rectangle();

Both statements can be combined in to a single statement as follows:

Rectangle rect1= new Rectangle();

The method Rectangle() is the default constructor of the class. We can created any number of objects of type Rectangle.

**Accessing Class Members**

The variables and methods inside a class can be accessed from outside using *dot* operator. The format is as shown below:

objectname.variablename;

objectname.methodname(parameter-list);

**Example:**

rect1.length=10;

rect1.width=50;

rect2.lenght=60;

rect2.width=50;

The two objects rect1 and rect2 store different values as shown below:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| rect1 | length | 10 |  | rect2 | length | 60 |
|  | width | 50 |  |  | width | 50 |

**Example**

class Rectangle

{

int length, width;

void getData(int x, int y)

{

length=x;

width=y;

Output

Area1 : 150

Area2 : 240

}

int rectArea()

{

int area=length \*width;

return(area);

}}

class RectArea

{

public static void main(String args[])

{

int area1, area2;

Rectangle rect1 = new Rectangle();

Rectangle rect2 = new Rectangle();

rect1.lenght = 15;

rect1.width=10;

area1 = rect1.lenght \* rect1.widht;

rect2.getData(20, 12);

area2=rect2.rectArea();

System.out.println(“Area1 : ” +area1);

System.out.println(“Area2 : ” +area2);

} }

**Constructors**

In order to be used by a program, an object must first be *instantiated* from its class definition. A special type of method called a *constructor* is used to define how objects are created. Constructors are special methods which have same name as that of class. They do not specify return type, not even void. This is because they return the instance of the class itself.

**Example:**

class Rectangle

{

int length, width;

Rectangle(int x, int y)

{

Output

Area : 150

width=y;

}

int rectArea()

{

int area=length \*width;

return(area);

}}

class rectArea

{

public static void main(String args[])

{

Rectangle rect1 = new Rectangle(15, 10);

int area= rect1.rectArea();

System.out.println(“Area : ” +area1);

} }

**Garbage collection and Finalizers**

In java, objects are dynamically allocated from a pool of free memory by using the new operator. Memory is not infinite and the free memory can be exhausted. Thus, new operator may fail to allocate memory because there is insufficient free memory to create desired object. For this reason, the key component of any dynamic allocation scheme is the recovery free memory from unused objects. Java uses garbage collection mechanism to reclaim unused objects automatically. When no reference to an object exists, that object is assumed to be no longer needed and the memory occupied by the object is released. This recycled memory can be used for a subsequent allocation. For efficiency, the garbage collector will usually run only when two conditions are met.: there are object to recycle and there is a need to recycle them. Garbage collection requires time, so the Java run-time system does it only when it is appropriate.

**The finalize() method**

It is possible to define a method that will be called just before an object’s final destruction by the garbage collector. This method is called finalize() method and it can be used to make sure that an object terminates cleanly. Syntax of adding a finalize() method is given below:

protected void finalize()

{

// Finalization code will be written here

}

The keyword protected prevents the access to finalize() by code defined outside the class. The finalize() method is called just before garbage collection. It is not called when an object goes out of scope. We cannot come to know when finalize() gets executed.

**Example:**

class FDemo

{

int x;

FDemo(int i) { x=i;

}

protected void finalize()

{

System.out.println(“ Finalizing ” +x);

}

void generator(int i)

{

FDemo ob= new FDemo(i);

} }

class Finalize

{

public static void main(String args[])

{

int count;

FDemo ob = new FDemo(0);

for (count=1; count < 100000; count++)

ob.generator(count);

} }

**The this Keyword**

When a method is called, it is automatically passed an implicit argument that is a reference to the invoking object. This reference is called as **this**. Java syntax permits the name of the parameter or the local variable to be the same as the instance variable. When this happens, the local variable hides instance variable. The instance variable can be accessed by referring it through **this** keyword.

**Example:**

class Pwr

{

double b, val;

int e;

Pwr(double b, int e)

{

this.b=b;

this.e=e;

val=1; // can also be written as this.val=1

if(e==0) return

for(; e>0; e--) val=val \* b;

}

double get\_pwr()

{ return this.val ;

} }

In this version, the names of parameters and names of instance variables are same. To uncover the instance variables, this keyword is used.

**Controlling Access to Class Members**

In its support for encapsulation, the class provides two major benefits. First, it links data with code that manipulates it. Second, it provides means by which access to members can be controlled. Restricting access to members of a class is a fundamental part of object-oriented programming as helps prevents the misuse of an object. By allowing access to data only through a well-defined set of methods, it is not possible for code outside the class to access and modify values of members directly.

**Java’s Access Modifiers**

The member access control is achieved through the use of three access modifiers: *public, private and protected.* Protected will be applied only in case of inheritance. When a member of a class is modified public specifier, that member can be accessed by any other code in the program. When a member of a class is specified as private, that member can be accessed only by any other members in its class. The default access specifier is public. Access modifier precedes the rest of the type specification.

***public***

Any variable or method is visible to the entire class in which it is defined. We can make it visible to all the classes outside the class by simply declaring the variable or method as ***public***. The variable or method declared as public has the widest possible visibility and accessible everywhere. In fact, this is what we would like to prevent in many programs.

***Ex : public int*** number***;***

***public void*** sum***()***

{ …..

}

***friendly***

When no access modifier is specified, the member defaults to a limited version of public visibility known as ***friendly***(default) level of access. The difference between the ‘public access’ and the ‘friendly access’ is that the public modifier makes fields visible in all classes, regardless of their packages while the friendly access makes fields only in the same package, but not in other packages.

[Note : A package is a group of related classes stored separately.]

***protected***

The visibility level of protected field lies in between the public access and friendly access. That is, the ‘protected’ modifiers makes the fields visible not only to all classes and subclasses in the same package but also to subclasses in other packages. Note that non-subclasses in other packages cannot access the ‘protected’ members.

***private***

private fields enjoy the highest degree of protection. They are accessible only within their own class. They cannot be inherited by subclasses and therefore not accessible in subclasses. A method declared as private behaves like a method declared as final. It prevents the method from being subclasses. Also note that we cannot override a non-private method in a subclass and then make it private.

***private protected***

This gives visibility level in between ‘protected’ access and ‘private’ access. This modifier makes the fields visible in all subclasses regardless of what package they are in. Remember, these fields are not accessible by other classes in the same package.

**Rules of Thumb**

* Use public if the field is to be visible everywhere in the current package and also subclasses in other packages
* Use default if the field is to be visible everywhere in the current package only
* Use private protected if the field is to be visible only in subclasses, regardless of packages
* Use private if the field is not to be visible anywhere except in its own class.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Access**  **Modifier**  **Access Location** | **public** | **protected** | **friendly**  **(default)** | **private protected** | **private** |
| Same class | Yes | Yes | Yes | Yes | Yes |
| Subclass in same location | Yes | Yes | Yes | Yes | No |
| Other classes in same package | Yes | Yes | Yes | No | No |
| Subclasses in other packages | Yes | Yes | No | Yes | No |
| Non-subclasses in other packages | Yes | No | No | No | No |

class MyClass

{

private int a; // a is private. Cannot be accessed outside the class

public int b; // b is public

int c; // c is public by default

void seta(int a){ this.a=a;

}

int getA()

{ return(a);

} }

class AccessDemo

{

public static void main(String args[])

{

MyClass ob= new MyClass();

Ob.setA(-99); // Allowed to modify a only through access function;

Ob.b=20; // allowed because b is public

Ob.c=34; // allowed because c is public

Ob.a=10; // Not allowed because a is private.

}}

This program will not compile because we access to the member ‘a’ outside the class which is not allowed.

**Pass Objects to Methods**

Till now only simple parameters are passed to methods. However objects can be passed to methods as parameters**.**

// Demonstrate objects as parameters to methods

class Block

{

int a, b,c;

Output

ob1 is same dimension of ob2 true

ob1 is same dimension of ob3 false

ob1 is same volume as ob3 true

int volume ;

Block(int i, int j, int k)

{

a=i; b=j; c=k;

volume= a\*b\*c;

}

boolean sameBlock(Block ob)

{

if ((ob.a==a) && (ob.b==b) && (ob.c==c)) return true;

else

return false;

}

boolean sameVolume(Block ob)

{

if ((ob.volume==volume) return true;

else

return false;

} }

class PassOb

{

public static void main(String args[])

{

Block ob1=new Block(10, 2, 5);

Block ob2=new Block(10, 2, 5);

Block ob3=new Block(4,5, 5);

System.out.println( “ob1 is same dimension of ob2 ” +ob1.sameBlock(ob2);

System.out.println( “ob1 is same dimension of ob3 ” +ob1.sameBlock(ob3);

System.out.println( “ob1 is same volume as ob3 ” +ob1.sameBlock(ob3);

} }

**Pass-by Value and Pass-by Reference**

Call by value or pass by value approach copies the vaue of an argument into the formal parameter of the subroutine. Therefore changes made to the parameter have no effect. When arguments are passed as primitive data types, the method is pass-by value.

class Test

{

void NoChange(int i, int j)

{

i=i+10;

Output

a and b before function call 15 20

a and b after function call 15 20

j=-j;

} }

class PassByVlaue

{

public static void main(String args[])

{

Test ob1 = new Test();

int a=15, b=20;

System.out.println(“a and b before function call ”+a + “ ” + b);

Ob.noChange(a,b);

System.out.println(“a and b after function call ”+a + “ ” + b);

} }

When an object is passed to a method, then changes made to the object by the method will be reflected. When a variable of type class is created, a reference to the object is made. This method is called pass by reference and changes made to the object inside the method will affect the object used as an argument.

class Test

{

int a, b;

Test(int i, int j)

Output

a and b before function call 15 20

a and b after function call 35 -20

{

a=i;

b=j;

}

void change(Test ob)

{

ob.a=ob.a+ob.b;

Ob.b=-ob.b

} }

class PassByRef

{

public static void main(String args[])

{

Test ob1 = new Test(15, 20);

System.out.println(“a and b before function call ”+ob1.a + “ ” + ob1.b);

Ob.change(ob);

System.out.println(“a and b after function call ”+ob1.a + “ ” + ob1.b);

} }

**Returning Objects**

A method can return any type of data, including class types that you create. For example, in the following program, the **incrByTen( )** method returns an object in which the value of **a** is ten greater than it is in the invoking object.

// Returning an object.

class Test

{

int a;

Test(int i)

{

a = i;

}

Test incrByTen()

{

Test temp = new Test(a+10);

return temp;

} }

class RetOb

Output

ob1.a: 2

ob2.a: 12

ob2.a after second increase: 22

{

public static void main(String args[])

{

Test ob1 = new Test(2);

Test ob2;

ob2 = ob1.incrByTen();

System.out.println("ob1.a: " + ob1.a);

System.out.println("ob2.a: " + ob2.a);

ob2 = ob2.incrByTen();

System.out.println("ob2.a after second increase: " + ob2.a);

}

}

**Overloading Methods**

If two methods of a class (whether both declared in the same class, or both inherited by a class, or one declared and one inherited) have the same name but different signatures, then the method name is said to be *overloaded.* In Java, it is possible to create methods that have the same name, but different parameter lists and different definitions. Method overloading is used when objects are required to perform similar tasks but using different input parameters. When we call a method in an object, Java matches up the method name first and then the number and type of parameters to decide which one of the definitions to execute. This process is known as ***polymorphism***. Methods are overridden on a signature-by-signature basis

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters.

While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

The important points to note:

* A difference in return type only is not sufficient to constitute an overload and is illegal.
* You should restrict your use of overloaded method names to situations where the methods really are performing the same basic function with different data.
* The language treats methods with overloaded names as totally different methods and as such they can have different return types. However, since overloaded methods perform the same job with different data sets, they should produce same return type normally. There is one particular condition, however, under which it is sensible to define different return types. This is the situation where the return type is derived from the argument type and is exactly parallel with the arithmetic operators.
* Overloaded methods may call one another simply by providing a normal method call with an appropriately formed argument list.

class OverloadDemo

{

void test() {

System.out.println("No parameters");

}

void test(int a)

{

System.out.println("a: " + a);

}

void test(int a, int b)

{

System.out.println("a and b: " + a + " " + b);

}

double test(double a)

**Output**

No parameters

a: 10

a and b: 10 20

double a: 123.25

Result of ob.test(123.25): 15190.5625

{

System.out.println("double a: " + a);

return a\*a;

} }

class Overload

{

public static void main(String args[])

{

OverloadDemo ob = new OverloadDemo();

double result;

ob.test();

ob.test(10);

ob.test(10, 20);

result = ob.test(123.25);

System.out.println("Result of ob.test(123.25): " + result);

} }

The method test( ) is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one double parameter. The fact that the fourth version of test( ) also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.

When an overloaded method is called, Java looks for a match between the arguments used to call the method and the parameter of the method. However, this match need not always be exact. In some cases, Java’s automatic type conversions can play a role in overload resolution.

// Automatic type conversions apply to overloading.

class OverloadDemo

{

void test()

{

System.out.println("No parameters");

}

// Overload test for two integer parameters.

void test(int a, int b)

{

System.out.println("a and b: " + a + " " + b);

}

// overload test for a double parameter

void test(double a)

{

System.out.println("Inside test(double) a: " + a);

} }

class Overload

{

public static void main(String args[]) {

Output

No parameters

a and b: 10 20

Inside test(double) a: 88

Inside test(double) a: 123.2

OverloadDemo ob = new OverloadDemo();

int i = 88;

ob.test();

ob.test(10, 20);

ob.test(i); // this will invoke test(double)

ob.test(123.2); // this will invoke test(double)

} }

This version of **OverloadDemo** does not define **test(int)**. Therefore, when **test( )** is called with an integer argument inside **Overload**, no matching method is found. However, Java can automatically convert an integer into a **double**, and this conversion can be used to resolve the call. Therefore, after **test(int)** is not found, Java elevates **i** to **double** and then calls **test(double)**.

**Overloading Constructors**

In addition to overloading normal methods, we can also overload constructor methods. In fact, for most real-world classes, overloaded constructors will be the norm, not the exception.

// Demonstrate Constructor Overloading

class MyClass

{

int x;

MyClass()

{

Output

Inside MyClass()

Inside MyClass(int)

Inside MyClass(double)

Inside MyClass(int,int)

ob1.x=0

ob2.x=88

ob3.x=17

ob4.x=8

System.out.println(“Inside MyClass()”);

x=0;

}

MyClass(int i)

{

System.out.println(“Inside MyClass(int)”);

x=i;

}

MyClass(double d)

{

System.out.println(“Inside MyClass(double)”);

x= (int) d;

}

MyClass(int i, int j)

{

System.out.println(“Inside MyClass(int, int)”);

x=i\*j;

}}

class OvLoadConsDemo

{

public static void main(String args[])

{

MyClass ob1=new MyClass();

MyClass ob2=new MyClass(88);

MyClass ob3=new MyClass(17.23);

MyClass ob4=new MyClass(2,4);

System.out.printl(ob1.x : ” +ob1.x);

System.out.printl(ob2.x : ” +ob2.x);

System.out.printl(ob3.x : ” +ob3.x);

System.out.printl(ob4.x : ” +ob4.x);

}}

MyClass() is overloaded four ways, each constructing an object differently. The proper constructor is based upon the parameters specified when new is executed.

**Recursion**

In Java, a method can call itself. This process is called recursion and a method that calls itself is said to be recursive. When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method. Recursive versions of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional function calls. Many recursive calls to a method could cause a stack overrun. Because storage for parameters and local variables is on the stack and each new call creates a new copy of these variables, it is possible that the stack could be exhausted. The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative versions.

//Recursion Demo

class Factorial

{

int factR(int n)

Output

Factorial of 3 6

Factorial of 4 24

Factorial of 5 120

{

if(n==1) return 1;

else return(n\*factR(n-1));

} }

class Recursion

{

public static void main(String args[])

{

Factorial f = new Factorial();

System.out.println(“Factorial of 3 ” + f.factR(3));

System.out.println(“Factorial of 4” + f.factR(4));

System.out.println(“Factorial of 5 ” + f.factR(5));

} }

**Understanding static**

There will be times when we want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static**. When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object. both methods and variables can be declared **static**. The most common example of a **static** member is **main( )**. **main( )** is declared as **static** because it must be called before any objects exist. Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

//Demonstrate static members

class StaticDemo

{

int x; // x is a normal variable

static int y; // y is a static variable

int sum()

{ return x+y; }

}

class SDemo

{

public static void main(String args[])

{

StaticDemo ob1 = new StaticDemo();

StaticDemo ob2 = new StaticDemo();

Ob1.x=10;

Output

ob1.sum=60

ob2.sum=70

ob1.sum=110

ob2.sum=120

Ob2.x=20;

StaticDemo.y=50;

System.out.println(“ob1. Sum =”+ob1.sum());

System.out.println(“ob2. Sum =”+ob2.sum());

StaticDemo.y=100;

System.out.println(“ob1. Sum =”+ob1.sum());

System.out.println(“ob2. Sum =”+ob2.sum());

} }

Static variable y is shared by both ob1 and ob2. Changing it affects the entire class.

**Static Methods**

Methods declared as static have several restrictions:

* They can directly call only other static methods.
* They can directly access only static data.
* They do not have a **this** reference.

The difference between the static method and a normal method is that the static method is called through its class name, without any object of that class being created.

// Demonstrate static method

Output

Value is 1024

After calling the function 512

Value is 4

After calling the function 2

class SMethod

{

static int val = 1024;

static int valDiv2()

{ return val/2;

} }

class SDemo

{

public static void main(String args[])

{

System.out.println(“Value is ”+SMethod.val);

System.out.println(“After calling the function ”+SMethod.valDiv2());

SMethod.val=4;

System.out.println(“Value is ”+SMethod.val);

System.out.println(“After calling the function ”+SMethod.valDiv2());

}}

Static methods can refer only static members. The following will not compile

class StaticError{

int denom=3;

static int val = 1024;

static int ValDiv() {

return val/denom;

} }

This will generate a compile time error as the static method refers to a non static member.

**Static Blocks**

Static blocks are needed to initialize static variables before any of the static methods of a class are used. A static block is executed when the class is first loaded. Thus, it is executed before the class can be used for any other purpose.

//Demonstrate static block

class SBlock

{

static double r2;

static double r3;

static

{ System.out.println(“Inside Static Block”);

r2=Math.sqrt(2.0);

r3=Math.sqrt(3.0);

}

SBlock(String msg)

{

System.out.println(msg);

} }

class SDemo

{

public static void main(String args[]) {

SBlock ob = new SBlock(“Inside Constructor “);

System.out.println(“Root of 2 =”+SBlock.r2);

System.out.println(“Root of 3 =”+SBlock.r3);

}}

**Nested and Inner Classes**

Nested class is a class that is declared within another class. A nested does not exist independently of its enclosing class. Thus the scope of a nested class is bounded by its outer class. It has access to all the variables and methods of its outer class.

// Nested Demo

class Outer

{

int nums[];

Outer(int n[])

{

nums = n;

}

void analyze()

{

Inner inob = new Inner();

System.out.println(“Minimum :”+inob.min());

System.out.println(“Maximum :”+inob.max());

System.out.println(“Average :”+inob.avg());

Output

Minimum : 1

Maximum : 9

Average : 5

}

class Inner

{

int min()

{

m=nums[0];

for(int I =1; i<nums.length;i++)

if (nums[i]<m) m=nums[i];

return m ;

}

int max()

{

m=nums[0];

for(int I =1; i<nums.length;i++)

if (nums[i]>m) m=nums[i];

return m ;

}

int avg()

{

int a=0;

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

a+=nums[i];

return a/nums.length;

} } }

class NestDemo

{

public static void main (String args[])

{

int x[]={3,2,1,5,6,9,7,8};

Outer outob = new Outer(x);

outob.analyze();

} }

**Varargs: Variable-Length Arguments**

Sometimes it may be necessary to create a method that takes a variable number of arguments. This can be handled in two ways. If the maximum number of arguments is small and known, the methods can be overloaded. If the maximum number of arguments is large and unknown, we can construct a method that takes variable number of arguments.

A variable-length argument is specified by three periods (…)

//Demonstrate variable number of arguments

Output

No. of args : 1

arg 0 : 10

No. of args : 3

arg 0 : 1

arg 1 : 2

arg 2 : 3

No. of args 0

class VarArgs

{

static void vaTest(int … v) //

{

System.out.println(“No. of args ” +v.length());

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

System.out.print(“ arg ” + i + “ :” +v[i]);

}

public static void main(Sting args[])

{

vaTest(10);

vaTest(1,2,3);

vaTest();

} }

**Inheritance**

**Inheritance Basics**

Inheritance is one of the basic principles of object-oriented programming. A class that is inherited by is called superclass. The class that does the inheritance is called subclass. It inherits all the variables and methods defined by the superclass and adds its own unique elements.

Java supports inheritance by using the keyword extends. The general form of a class declaration that inherits a superclass is given below:

class subclass-name extends superclass-name {

// body of the class

}

//Demonstrate single inheritance

class Shape{

int height, width;

void showDim() {

System.out.println(“Width = “ +width + “Height =”+height); } }

class Triangle extends Shape{

String style;

void showStyle(){

System.out.println(“Triangle is ” +style); }

int area() {

return(height\*width /2); } }

Output

Height = 4 Width = 6

Triangle is filled

Area =12

Height = 8 Width = 12

Triangle is outlined

Area =48

class InheritDemo {

public static void main(String args[]) {

Triangle t1 = new Triangle();

Triangle t2 = new Triangle();

t1.height=4;

t1.width=6;

t1.style=”filled”;

t2.height=8;

t2.width=12;

t2.style=”outlined”;

t1.showDim();

t1.showStyle();

System.out.println(“Area =”+t1.area());

t2.showDim();

t2.showStyle();

System.out.println(“Area =”+t2.area()); } }

The class Shapes contains two members namely height and width and a method showDim(). The class triangle inherits all the members and methods of the class Shape and it has its own member named style and methods named area() and showStyle().

**Member Access and Inheritance**

All the public members of the base class are inherited by the derived classes where as the members declared with the modifier private cannot be accessed by the derived class. The following example program will not be compiled

class Shapes {

private int height;

private int width;

void showDim() {

System.out.println(“Width = “ +width + “Height =”+height); } }

class Triangle extends Shape {

Sting style;

int area() {

return width \* height /2; // Error !Private members cant be accessed } }

This program will not compile because width and height can not be inherited by the derived class since they are private.

**Constructors and Inheritance**

Both superclasses and subclasses can have their own constructors. The constructor of the superclass constructs the superclass portion of the object and that of the subclass constructs the subclass part. When only the subclass defines a constructor, it just constructs the object of the subclass. The superclass portion of the object is constructed by calling the default constructor.

class Shape{

int height, width;

void showDim() {

System.out.println(“Width = “ +width + “Height =”+height); } }

class Triangle extends Shape {

Triangle(String s, int h, int w) {

style=s;

height =h;

width=w; }

void showStyle(){

System.out.println(“Triangle is ” +style); }

int area()

{return(height\*width/2); } }

class ConDemo {

public static void main(String args[]) {

Triangle t1 = new Triangle(“filled”, 4, 6);

Triangle t2 = new Triangle(“outlined”, 8,12);

t1.height=4;

Output

Height = 4 Width = 6

Triangle is filled

Area =12

Height = 8 Width = 12

Triangle is outlined

Area =48

t1.width=6;

t1.style=”filled”;

t2.height=8;

t2.width=12;

t2.style=”outlined”;

t1.showDim();

t1.showStyle();

System.out.println(“Area =”+t1.area());

t2.showDim();

t2.showStyle();

System.out.println(“Area =”+t2.area()); } }

The constructor Triangle() can be used to initialize objects of the derived class but it can not be used to initialize the objects of the base class.

**Use of super()**

When both superclass and subclass define constructors, the process is bit complicated. The subclass constructor uses the keyword super to invoke the constructor of the superclass. The keyword super can be used only within the derived class and if included, it must always be the first statement executed inside a subclass constructor.

//To demonstrate super()

class Room {

int length, breadth;

Room(int x, int y) {

length=x;

breadth=y; }

int area() {

return(length\*breadth); }}

Output

Area =168

Volume =1680

class BedRoom extends Room{

int height;

BedRoom(int x, int y, int z) {

super(x,y);

height=z; }

int volume() {

return(length \* breadth\*height); } }

class InhTest{

public static void main(String args[]) {

BedRoom r1= new BedRoom(14,12,10);

System.out.println(“Area =”, +r1.area());

System.out.println(“Volume =”, +r1.volume()); } }

**Using super to Access Superclass Members**

We can access the superclass member with the help of super key word. This has the following form:

**super**.*member*

Here member is either a method or an instance variable.

// using super to access member of superclass

class A {

int i; }

class B extends A {

int i; // This i belongs to class B which hides the i of A

B(int a, int b) {

super.i=a;

Output

i in superclass 1

i in subclass 2

i=b; }

void show() {

System.out.println(“i in superclass ”+super.i);

System.out.println(“i in subclass ”+ i); } }

class UseSuper {

public static void main(String args[]) {

B ob1=new(1,2);

ob1.show(); } }

Although variable i of B is hidden by variable i of A, ***super*** allows access to the i defined in the superclass. The keyword ***super*** can also be used to call methods that are hidden by a subclass.

**Multilevel Inheritance**

It is possible to build hierarchies that contain many layers of inheritance. This phenomenon is called multilevel inheritance. In this situation there will be two or more levels of inheritance. If there are classes A B and C, in such a way that B is the subclass of A and C is the subclass of B we call it as multilevel inheritance.

//Demonstrate Multilevel Inheritance

class Shape{

int height, width;

void showDim() {

System.out.println(“Width = “ +width + “Height =”+height); }

Shape()

{ height=width=0; }

Shape(int x, int y)

{ height=x; width=y; }

void showDim()

Sustem.out.println(“Height = ”+height+” Width = ”+width);}}

class Triangle extends Shape {

Triangle() {

super();

style=”none”; }

Output

Height = 4 Width = 6

Triangle is filled

Color is blue

Area =12

Height = 8 Width = 12

Triangle is outlined

Color is red

Area =48

class Triangle extends Shape {

Triangle(String s, int h, int w) {

super(h, w);

style=s;

void showStyle(){

System.out.println(“Triangle is ” +style); }

int area()

{return(height\*width/2); } }

class colTriangle extends Triangle {

String colr;

colTriangle(String s1, String s2, int x, int y)

{ super(s2, x,y);

colr=s1; }

void showColor(){

System.out.println(“Color is ” +colr);} }

class MulDemo {

public static void main(String args[]) {

colTriangle t1 = new Triangle(“blue”, “filled”, 4, 6);

Triangle t2 = new Triangle(“red”, “outlined”, 8,12);

t1.showDim();

t1.showStyle();

t1.showColor();

System.out.println(“Area =”+t1.area());

t2.showDim();

t2.showStyle();

t1.showColor();

System.out.println(“Area =”+t2.area()); } }

**Method Overriding**

In a class hierarchy when a method in a subclass has the same return type and signature as a method in it’s superclass, then the method in the subclass is said to override the method in the superclass. when a overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass**.** The version of the method defined by the superclass will be hidden.

//Demonstrate method overriding

class A {

int i,j;

A(int x, int y) {

i=x,

j=y; }

void show() {

System.out.println(“Values of i and j are “ +i+ “ ” +j); } }

class B extends A {

int k;

Output

k= 30

B(int x, int y, int z) {

super(x,y);

k=z; }

void show() {

System.out.println(“”k= ”+k); } }

class overrideDemo{

public static void main(String args[]){

B ob1=new B(10,20,30);

Ob1.show() //refers t the function written in the subclass

} }

The show() in the subclass overrides the show() in the superclass. If we want to access the show() of superclass we can use the super() as follows:

B(int x, int y, int z) {

Output

Values of i and j 10 20

k= 30

super(x,y);

k=z; }

void show() {

super();

System.out.println(“”k= ”+k); } }

**Final variables and methods**

All methods and variables can be overridden by the subclasses. If we wish to prevent overriding the members if superclass we can declare them as final using the keyword ***final*** as modifier.

Example

final int size = 100;

final void showStatus()

Making a method final ensures that the functionality defined in this method will never be altered in any way. Similarly, the value of a final variable can never be changed. Final variables behave like class variables and they do not take any space on individual objects of the class.

class A{

final void meth() {

System.out.println(“This is a final method”); }

Class B extends A {

void meth() { // Error can’t override!!! } }

**Abstract Methods and Classes**

If we modify a method as final we ensure that the method is not redefined in a subclass. Java allows us to do something that is exactly opposite to this. That is, we can indicate a method must always be redefined in a subclass, thus making overriding compulsory. This is done using the modifier keyword ***abstract***.

Example:

abstract class Shape {

........

........

abstract void show() {

........ } }

While using abstract classes, we must satisfy the following conditions:

* Abstract classes can not be used to instantiate objects directly. For example:

Shape s1=new Shape() is illegal because Shape is an abstract class according to the above example.

* The abstract methods of an abstract class must be redefined in its subclass
* We can not declare abstract constructors or abstract static methods.

**Unit III**

**Packages and Interfaces**

***Packages*** are groups of related classes. Packages help us to organize code and provide another layer of encapsulation. An **interface** defines a set of methods that will be implemented by a class. Interface does not implement any method by itself. Packages and interfaces give the users greater control over the organization of programs.

**Packages**

In programming, it is helpful to group related pieces of program together. In Java, this is done by using a package. Package serves two purposes:

1. It provides a mechanism by which related pieces of a program can be organized as a unit. Classes defined within a package must be accessed through their package name.
2. Package participates in Java’s access control mechanism. Classes defined within a package can be made private to that package and not accessible by code outside the package. Thus package provides a means by which classes can be encapsulated.

In general, when a class is named, we are allocating a name from namespace. A namespace defines a declarative region. In Java, no two classes can use the same name from the same namespace. Thus, within a given namespace, each class name must be unique. Since package usually contains related classes, Java defines special access rights to the code within a package. This enables the programmer to create self-contained group of related classes that keep their operation private.

**Defining a Package**

All classes in Java belong to some package. When no package statement is specified, the default package is used. A package can be created by adding a **package** command at the top of the Java source file. The classes declared within that file will belong to the specified package. Since a package specifies a namespace, the names of the classes inside it become the part of the namespace.

The general form of package statement is

**package** pkg;

where pkg is the name of the package.

Example: package myPack;

**Accessing a Package**

It is to be noted that a Java system package can be accessed either using a fully qualified class name for using a shortcut approach through the import statement. We use import statement when there are many references to a particular package or the package name is too long and unwieldy.

The same approach can be used to access the user-defined packages as well. The ***import*** statement can be used to search a list of packages for a particular class. The general form of import statement for searching a class is :

**import** package1 [.package2] [.package3].classname;

Here *package1* is the name of the top level package, *package2* is the name of the package that is inside the *pacakge1*, and so on. We can have any number of packages in a package hierarchy, Finally, the explicit classname is specified. Note that the statement must end with a semicolon and the import statement should appear before any class definitions in a source file. Multiple import statements are allowed.

Following is an example of importing a particular class :

**import** firstPackage.secondPackage.MyClass;

After using this statement, all the members of the class MyClass can be directly accessed using the class name or its objects directly without using the package name.

We can also use another approach as follows

**import** packagename.\*;

Here, *packagename* may denote a single package or a hierarchy of packages. The \* indicate that the compiler should search this entire package hierarchy when it encounters a class name. This implies that we can access all classes contained in the above package directly.

**Adding a class to a package.**

It is simple to add a classs to an existing package. For example consider

package p1;

public class ClassA

{

…..

}

The above package **p1** contains one class **ClassA**. Suppose we want to add another class **ClassB** to this package. This can be done as follows :

* 1. Define the class and make it public.
  2. Place the package statement p1 before the class defintion as follows

package p1;

public class ClassB

{

……

}

**Packages and Member Access**

The visibility of an element is determined by its access specification—private, public, protected or by default. If a member has no explicit access modifier, then it is visible within its package and not outside the package. The members which are declared s public are visible everywhere including different classes of different package. There is no restriction on their use or access. A private member is available only to the other members of its class. A member specified as protected is accessible within its package and to all subclasses including subclasses in another package.

Class Member Access

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Private Member | Default Member | Protected Member | Public Member |
| Visible within same class | Yes | Yes | Yes | Yes |
| Visible within same package by subclass | No | Yes | Yes | Yes |
| Visible within same package by non-subclass | No | Yes | Yes | Yes |
| Visible within different package by subclass | No | No | Yes | Yes |
| Visible within different package by non-subclass | No | No | No | Yes |

**Built-in Packages**

Java defines a large number of standard classes that are available to all programs. This class library is referred to as the Java API (Application Programming Interface) The Java. API is stored in packages. At the top of the package hierarchy is **java.**

|  |  |
| --- | --- |
| **Subpackage** | **Description** |
| java.lang | Langage support classes. These are classes that Java compiler itself uses and therefore they are automatically imported. They include classes for primitive types, strings, math functions, threads and exceptions. |
| java.util | Language utility classes as vectors, hash tables, random numbers, date, etc. |
| java.io | Input/output support classes. They provide facilities for the input and output of data |
| java.awt | Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on. |
| java.net | Classes for networking. They include classes for communicating with local computers as well as with internet servers. |
| java.applet | Classes for creating and implementing applets |

**Interfaces**

Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. An interface doesn’t provide any implementation. To implement interface, classes must provide bodies for the methods described by the interface. Two classes may implement the same interface in different ways but each class supports the same set of methods. By providing interface keyword, Java allows the programmers to utilize “One interface, multiple methods” aspect of polymorphism.

This is the general form of an interface:

access interface name {

return-type method-name1(parameter-list);

return-type method-name2(parameter-list);

type final-varname1 = value;

type final-varname2 = value;

// ...

return-type method-nameN(parameter-list);

type final-varnameN = value;

}

When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as **public**, the interface can be used by any other code. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface. *name* is the name of the interface, and can be any valid identifier. Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods.

Variables can be declared inside of interface declarations. They are implicitly **public**, **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized. All methods and variables are implicitly **public**.

Example: public interface Series {

int getNext();

void reset();

void setStart(int x); }

**Implementing Interfaces**

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, we must include the **implements** clause in a class definition and then create the methods defined by the interface. The general form of a class that includes the **implements** clause looks like this:

class *classname* [extends *superclass*] [implements *interface* [,*interface...*]] {

// class-body

}

// Implementing an interface

public interface Series {

int getNext();

void reset();

void setStart(int x); }

class ByTow implements Series {

int start, val;

ByTwo() {

start=0; val=0; }

**Output:**

Next value is 2

Next value is 4

Next value is 6

Next value is 8

Next value is 10

Resetting

Next value is 2

Next value is 4

Next value is 6

Next value is 8

Next value is 10

Starting at 100

Next value is 102

Next value is 104

Next value is 106

Next value is 108

Next value is 110

public int getNext() {

val+=2;

return val; }

public void reset() {

val=start; } public void setStart(int x) {

start=x;

val=x; } }

class SDemo {

public static void main(String args[]) {

ByTwo ob = new ByTwo();

for(int i=0; i<5; i++)

System.out.println(“Next value is ”+ob.getNext());

System.out.println(“\n Resetting”);

ob.reset();

for(int i=0; i<5; i++)

System.out.println(“Next value is ”+ob.getNext());

System.out.println(“\n Starting at 100”);

ob.setStart(100);

for(int i=0; i<5; i++)

System.out.println(“Next value is ”+ob.getNext()); } }

**Using Interface References**

We can create interface reference variables. Such a variable can refer to an object that implements its interface. When a method on an object is called through an interface reference, the version of the version of the method implemented by the object is executed.

// Demonstrate interface references

public interface Series {

int getNext();

void reset();

void setStart(int x); }

class ByTow implements Series {

int start, val;

ByTwo() {

start=0; val=0; }

public int getNext() {

val+=2;

return val; }

public void reset() {

val=start; }

public void setStart(int x) {

start=x;

val=x; } }

class SDemo2 **{**

public static void main(String args[]) {

ByTwo two = new ByTwo();

Series ob;

Ob=two;

for(int i=0; i<5; i++)

System.out.println(“Next value is ”+ob.getNext());

} }

**Extending an Interface**

One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain. Following is an example:

// One interface can extend another.

interface A {

void meth1();

void meth2();

}

// B now includes meth1() and meth2() -- it adds meth3().

interface B extends A {

void meth3();

}

// This class must implement all of A and B

class MyClass implements B {

public void meth1() {

System.out.println("Implement meth1().");

}

public void meth2() {

System.out.println("Implement meth2()."); }

public void meth3() {

System.out.println("Implement meth3()."); } }

class IFExtend {

public static void main(String arg[]) {

MyClass ob = new MyClass();

ob.meth1();

ob.meth2();

ob.meth3(); } }

**Exception Handling**

**Exception Handling Fundamentals**

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. All exception classes are derived from a class called **Throwable**. There are two direct subclasses of Throwable class: **Exception** and **Error**. Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Program statements that require monitoring are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. The code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, the keyword **throw** can be used. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

**Using try and catch**

This is the general form of an exception-handling block:

try {

// block of code to monitor for errors

}

catch (*ExceptionType1 exOb*) {

// exception handler for *ExceptionType1*

}

catch (*ExceptionType2 exOb*) {

// exception handler for *ExceptionType2*

}

// ...

finally {

// block of code to be executed after try block ends

}

Here, ExceptionType is the type of exception that has occurred. When an exception is thrown, it is caught by the corresponding catch statement which then processes the exception. When no exception is thrown, try block ends normally and catch statements are bypassed. Thus catch statements are executed only if an exception is thrown.

**A Simple Exception Example**

// Demonstrate Exception Handling

class EDemo1 {

public static void main(String args[]) {

Output:

Before exception is generated

Array index out of bounds !!

After the catch statement

int nums[] = new int[4];

try {

System.out.println(“Before exception is generated”);

nums[7]=10; // code that generates exception

System.out.println(“This won’t be displayed”); }

catch (ArrayIndexOut OfBoundsException e) {

System.out.println(“Array index out of bounds !! ”); }

System.out.println(“After the catch statement”); } }

**The Consequences of an Uncaught Exception**

If an exception is caught, it prevents abnormal program termination. If a program doesn’t catch any exception, it will be caught by the JVM which terminates the program and displays stack trace and an error message.

Output:

Exception in thread “main” java.lang.ArrayIndexOutOfBoundsException : 7 at NotHandled.main(NotHandled.java : 9

class NotHandled {

public static void main(String args[]) {

int nums[] = new int[4];

System.out.println(“Before exception is generated”);

nums[7]=10; // code that generates exception } }

**Use of Exception Handling**

The main benefit of using exception handling is that, it enables the program to respond to an error and then continue running.

// To handle exceptions gracefully

class EDemo2 {

public static void main(String args[]) {

Output:

4/2 = 2

Can’t divide by zero

16/4=4

32/4=8

Can’t divide by zero

128/8= 16

int nr[] {4, 8, 16, 32, 64, 128};

int dr[] {2, 0, 4, 4, 0, 8};

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

try

{

System.out.println( nr[i] +”/” + dr[i] +”=” +nr[i]/dr[i]); }

catch (ArithmeticException e) {

System.out.println(“Can’t Divide by zero ”); }

} } }

**Using Multiple catch Statements**

// Multiple catch statements

class EDemo3 {

public static void main(String args[]) {

Output:

4/2 = 2

Can’t divide by zero

16/4=4

32/4=8

Can’t divide by zero

128/8= 16

No Matching Element

No Matching Element

int nr[] {4, 8, 16, 32, 64, 128, 265, 512};

int dr[] {2, 0, 4, 4, 0, 8};

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

try

{

System.out.println( nr[i] +”/” + dr[i] +”=” +nr[i]/dr[i]); }

catch (ArithmeticException e) {

System.out.println(“Can’t Divide by zero ”); }

catch (ArrayIndexOutOfBoundsException e) {

System.out.println(“No Matching Element ”); }

} } }

**Catching Subclass Exceptions**

A catch clause for a superclass will also match any of its subclasses. Since the superclass of all exceptions is Throwable, to catch all possible exception, we can use Throwable. If subclass exception has to be caught, it must be put first in the catch sequence.

class EDemo4 {

public static void main(String args[]) {

Output:

4/2 = 2

Some Exception Occurred

16/4=4

32/4=8

Some Exception Occurred

128/8= 16

No Matching Element

No Matching Element

int nr[] {4, 8, 16, 32, 64, 128, 265, 512};

int dr[] {2, 0, 4, 4, 0, 8};

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

try

{

System.out.println( nr[i] +”/” + dr[i] +”=” +nr[i]/dr[i]); }

catch (ArrayIndexOutOfBoundsException e) {

System.out.println(“No Matching Element ”); }

catch (Throwable e) {

System.out.println(“Some Exception occurred ”); }

} } }

Output:

4/2 = 2

Can’t divide by zero

16/4=4

32/4=8

Can’t divide by zero

128/8= 16

No Matching Element

Fatal Error

Program terminated

**Nesting of try Blocks**

One try block can be nested within another. An exception generated by the inner

try block is not caught by the catch statement associated with that try.

class EDemo5 {

public static void main(String args[]) {

int nr[] {4, 8, 16, 32, 64, 128, 265, 512};

int dr[] {2, 0, 4, 4, 0, 8};

try

{

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

try {

System.out.println( nr[i] +”/” + dr[i] +”=” +nr[i]/dr[i]);

}

catch (ArithmeticException e) {

System.out.println(“Can’t divide by zero ”); } } }

catch (ArrayIndexOutOfBounds e) {

System.out.println(“No matching element ”);

System.out.println(“Fatal Error \n Program terminated ”);}

} }

In the example given above, if division by zero exception is occurred, it is handled by the inner catch block and the program execution continues. However, the array boundary error is caught by the outer catch, which causes the program to terminate.

**Throwing an Exception**

It is possible to throw an exception manually using the **throw** statement.

// manually throw and exception

calss ThrowDemo {

public static void main(String args[]) {

try{

Output:

Before throw.

Exception caught

After try/catch block

System.out.println(“Before throw. ”);

throw new ArithmeticException(); }

catch (ArithmeticException e) {

System.out.println(“Exception caught ”); }

System.out.println(“After try /catch block ”); }

**Rethrowing an Exception**

An exception caught by catch statement can be rethrown so that it can be caught by an outer catch. The reason for rethrowing is to allow multiple handlers.

**The Throwable Class**

All exceptions are subclasses of Throwable. Throwable supports several methods. The commonly used methods defined by Throwable class are listed below:

|  |  |
| --- | --- |
| **Method** | **Description** |
| Throwable fillinStackTrace() | Returns a Throwable object that contains complete stack trace |
| String getLocalizedMessage() | Returns a localized description of the exception |
| void printStackTrace() | Displays the stack trace |
| void printStackTrace(PrintStream s) | Sends the stack trace to specified stream |
| void printStackTrace(PrintWriter s) | Sends the stack trace to specified stream |
| String toString() | Returns the string object containing a complete description of the exception. This method is called by println() when outputting Throwable object. |

**Java’s Built-in Exceptions**

Inside the standard package **java.lang**, Java defines several exception classes. The most general of these exceptions are subclasses of the standard type **RuntimeException**. As previously explained, these exceptions need not be included in any method’s **throws** list. In the language of Java, these are called ***unchecked exceptions***because the compiler does not check to see if a method handles or throws these exceptions. Exceptions defined by **java.lang** that must be included in a method’s **throws** list if that method can generate one of these exceptions and does not handle it itself. These are called ***checked exceptions****.*

|  |  |
| --- | --- |
| **Exception** | **Meaning** |
| ArithmeticException | Arithmetic error, such as divide-by-zero. |
| ArrayIndexOutOfBoundsException | Array index is out-of-bounds. |
| ArrayStoreException | Assignment to an array element of an incompatible type. |
| ClassCastException | Invalid cast. |
| EnumConstantNotPresentException | An attempt is made to use an undefined enumeration value |
| IllegalArgumentException | Illegal argument used to invoke a method. |
| IllegalMonitorStateException | Illegal monitor operation, such as waiting on an unlocked thread. |
| IllegalStateException | Environment or application is in incorrect state. |
| IllegalThreadStateException | Requested operation not compatible with current thread state. |
| IndexOutOfBoundsException | Some type of index is out-of-bounds. |
| NegativeArraySizeException | Array created with a negative size. |
| NullPointerException | Invalid use of a null reference. |
| NumberFormatException | Invalid conversion of a string to a numeric format. |
| SecurityException | Attempt to violate security. |
| StringIndexOutOfBounds | Attempt to index outside the bounds of a string. |
| TypeNotPresentException. | Type not found |
| UnsupportedOperationException | An unsupported operation was encountered |
| The unchecked exceptions defined in **java.lang** | |

|  |  |
| --- | --- |
| **Exception** | **Meaning** |
| ClassNotFoundException | Class not found. |
| CloneNotSupportedException | Attempt to clone an object that does not implement the Cloneable |
| IllegalAccessException. | Access to a class is denied. |
| InstantiationException | Attempt to create an object of an abstract class or interface. |
| InterruptedException | One thread has been interrupted by another thread. |
| NoSuchFieldException | A requested field does not exist. |
| NoSuchMethodException | A requested method does not exist. |
| The checked exceptions defined in **java.lang** | |

**Using Finally**

To specify a block of code when a try/catch block is exited, **finally** statement can be used. When finally block is included, this is guaranteed to execute whether or not an exception is thrown. Whose general form is

try {

// block of code to monitor for errors }

catch (ExceptionType1 exOb) {

// exception handler for ExceptionType1 }

catch (ExceptionType2 exOb) {

// exception handler for ExceptionType2 }

// ...

finally {

// block of code to be executed after try block ends }

The **finally** block will be executed whenever the execution leaves a try/catch block.

// Use of finally

class UseFinally {

public static void main(Sting args[]) {

Output:

Array Index Error

Y = 2

int a[]={5,10};

try{

int x=a[2]/b-a[1]; }

catch(ArithmeticException e) {

System.out.println(“Division by zero”); }

catch(ArrayIndexOutOfBoundsException e) {

System.out.println(“Array Index Error”); }

catch(ArrayStoreException e) {

System.out.println(“Wrong Data type”); }

finally{

int y=a[1]/a[0];

System.out.println(“Y = ”+y);

**Creating Exception Subclasses (Creating User-Defined Exceptions)**

Although Java’s built-in exceptions handle most common errors, we can create our own exception types to handle situations specific to our applications. This is quite easy to do: just define a subclass of **Exception** (which is, of course, a subclass of **Throwable**). These subclasses don’t need to actually implement anything—it is their existence in the type system that allows us to use them as exceptions. The **Exception** class does not define any methods of its own. It inherits those methods provided by **Throwable**.

//Demonstrate user-defined exception handling

import java.lang.Exception

class MyException extends Exception

{

MyException(String mag)

super(msg); } }

class UserException {

public static void main(String args[])

{

int x=5, y=1000;

try{

float z = float(x)/float(y);

if(z<0.01)

{

throw new MyException(“Number is too small”); }

}

catch(MyException e)

{

Output:

Caught My Exception

Number is too small

I am always here

System.out.println(“Caught My Exception”);

System.out.println(e.getMessage()); }

finally{

System.out.println(“I am always here”; }

} }

**Multithreaded Programming**

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a ***thread****,* and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

**Multithreading Fundamentals**

There are two distinct types of multitasking: ***process-based*** and ***thread-based***. A process is a program that is executing. A process-based multi-tasking allows the computer to run two or more programs concurrently. In thread-based multi tasking environment, a single program can perform two or more tasks at the same time. The advantage of multithreading is that it makes one to write efficient programs which utilize the idle time that is present in most of the programs. I/O devices are much slower than the CPU. Sometimes, a program spends majority of its execution time waiting to send or receive the information to or from a device. By using multithreading, the program can execute another task.

Threads exist in several states. A thread can be *running*. It can be ready to run as soon as it gets CPU time. A running thread can be *suspended*, which temporarily suspends its activity. A suspended thread can then be *resumed*, allowing it to pick up where it left off. A thread can be *blocked* when waiting for a resource. At any time, a thread can be *terminated*, which halts its execution immediately. Once terminated, a thread cannot be resumed.

**The Thread Class and Runnable Interface**

Java’s multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. **Thread** encapsulates a thread of execution. Since you can’t directly refer to the ethereal state of a running thread. To create a new thread, the program will either extend **Thread** or implement the **Runnable** interface. The **Thread** class defines several methods that help manage threads. Few of them are listed below:

|  |  |
| --- | --- |
| **Method** | **Meaning** |
| final String getName () | Obtain a thread’s name. |
| final int getPriority () | Obtain a thread’s priority. |
| final isAlive() | Determine if a thread is still running. |
| final void join() | Wait for a thread to terminate. |
| void run() | Entry point for the thread. |
| static void sleep(long milliseconds) | Suspend a thread for a period of time. |
| void start() | Start a thread by calling its run method. |

All processes have at least one thread of execution, which is usually called the main thread, because it is the one that is executed when the program begins. From the main thread other threads can be created.

**Creating a Thread**

There are two methods in Java by which we can create threads.

Output

I from Thread X = 0

I from Thread X = 1

I from Thread X = 2

I from Thread X = 3

I from Thread X = 4

I from Thread X = 5

End of main

1. By implementing Runnable interface.
2. By extending Thread class.

The Runnable interface defines only one method called run()

// Demonstrate creation of a Thread

class X implements Runnable {

public void run() {

for(int i=0; i<=5; i++)

System.out.println(“I from thread X = ”+ i); } }

class ThreadDemo {

public static void main(String args[]) {

X rn = new X();

Thread t1=new Thread(rn);

t1.start();

System.out.println(“End of main”); } }

**Creating Multiple Threads**

The above program creates a single thread. Multiple threads can be created in same way. The following program illustrates creation of multiple threads.

class Square implements Runnable {

public void run() {

for(int i=1; i<=5; i++)

System.out.println(“Square of ”+ i + “ = ”+ i\*i); }

}

Output

Square of 1 = 1

Square of 2 = 4

Square of 3 = 9

Square of 4 = 16

Square of 5 = 25

Cube of 1 =1

Cube of 2 =8

Cube of 3 =27

Cube of 4 =64

Cube of 5 =125

End of main

class Cube implements Runnable {

public void run() {

for(int i=1; i<=5; i++)

System.out.println(“Cube of ”+ i + “ = ”+ i\*i\*i); }

}

class ThreadDemo1 {

public static void main(String args[]) {

Square s = new Square();

Cube c = new Cube();

Thread t1=new Thread(s);

Thread t2=new Thread(c);

t1.start();

t2.start();

System.out.println(“End of main”); } }

**Determining When a Thread Ends**

Thread provides two means by which we can determine whether the thread has ended.

1. The isAlive() method which waits for the child thread to terminate.
2. The join() method which waits until the thread on which it is called terminates. The calling thread is waiting until the specified thread joins it.

**Thread Priority**

Each thread is associated with a priority. The priority of a thread determines how much CPU time a thread receives relative to other active threads. Low-priority threads receive a little and high-priority threads receive a lot of CPU time. When high priority thread waits for a resource, low-priority thread may run. When high priority thread gains access to the resource, it may preempt the low-priority thread and resume execution.

When a child thread is started, its priority setting is equal to that of its parent thread. The priority of a thread can be changed by calling setPriority(). This method has general form

final void setPriority(int level)

Here, *level* specifies the new priority setting for the calling thread. The value of *level* must be within the range MIN\_PRIORITY and MAX\_PRIORITY with values 1 and 10 respectively. The defaultpriority value is NORM\_PRIORITY which is 5.

The current priority of a thread can be obtained by using getPriority() whose general form is

final int getPriority()

//Demonstrate Thread Priority

class A extends Thread {

public void run() {

for(int i=0; i<=4;i++)

System.out.println(“From Thread A i = ”+i);

System.out.println(“Exit from A”); } }

class B extends Thread {

public void run() {

for(int j=0; j<=4;j++)

System.out.println(“From Thread B j = ”+j);

System.out.println(“Exit from B”); } }

class C extends Thread {

public void run() {

for(int k=0; j<=4;k++)

System.out.println(“From Thread C k = ”+k);

System.out.println(“Exit from C”); } }

class ThreadPriority {

public static void main(String args[]) {

A threadA = new A();

B threadB = new B();

C threadC = new C();

threadC.setPriority(Thread.MAX\_PRIORITY);

threadB.setPriority(threadA.getPriority()+1);

threadA.setPriority(Thread.MIN\_PRIORITY);

System.out.println(“Thread A Started”);

threadA.start();

System.out.println(“Thread B Started”);

threadB.start();

System.out.println(“Thread C Started”);

threadC.start(); } }

In the above example, thread is assigned with priority value 10, thread with 6 and threadA with 1 respectively.

**Synchronization**

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called ***synchronization****.* The most common reason for synchronization is when two or more threads need to access to a shared resource that can be used by only one thread at a time. For example, when a thread is writing to a file, the second thread must be prevented from doing so at the same time. Java provides unique, language-level support for it.

Key to synchronization is the concept of the monitor (also called a *semaphore*). A *monitor* is an object that is used as a mutually exclusive lock. Only one thread can *own* amonitor at a given time. When a thread acquires a lock, it is said to have *entered* the monitor.

All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor. These other threads are said to be *waiting* for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires. There are two ways to synchronize a code.

1. Using Synchronized Methods
2. Using Synchronized Statement

**Synchronized Methods**

We can make a method synchronized by prefixing the keyword synchronized. When a method is called, the calling thread enters the object’s monitor, which locks the object. While locked, no other object can enter the method. When the thread enters from the method, the monitor unlocks the object, allowing it to be used by another thread.

//Use of Synchronize to control the access

import java.io.\*;

class SumArray{

private int sum;

synchronized int SumArray(int nums[]) {

sum=0;

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

sum+=nums[i];

System.out.println("Running total for" Thread.currentThread().getName() + "is " + sum);

try{

Thread.sleep(100) }

catch(InterruptedException e)

{ System.out.println("Thread Interrupted"); } }

return (sum); } }

class MyThread implements Runnable

{ Thread thrd;

static SumArray sa = new SumArray();

int a[];

int answer;

MyThread(String Name, int nums[]) {

thrd=new Thread(this, Name);

a=nums;

thrd.start(); }

public void run(){

int sum;

System.out.println(thrd.getName() + "Starting");

answer=sa.SumArray(a);

System.out.println("Sum for "+ thrd.getName() + answer);

System.out.println(thrd.getName() + "Terminating"); } }

class Sync {

public static void main(String args[]) {

int a[]={1,2,3,4,5,6,7,8};

int b[]={1,3,5,7,9};

MyThread t1 = new MyThread3("Child #1", a);

MyThread t2 = new MyThread3("Child #2", b);

try {

t1.thrd.join();

t2.thrd.join(); }

catch(InterruptedException e) {

System.out.println("Main Thread Interrupted !!"); } }}

**The Synchronized Statement**

While creating **synchronized** methods within classes that is an easy and effective means of achieving synchronization, it will not work in all cases. For example, if we want to synchronize access to objects of a class that was not designed for multithreaded access that is, the class does not use **synchronized** methods. Further, this class was not created by us , but by a third party and we do not have access to the source code. Thus, we can’t add **synchronized** to the appropriate methods within the class. The solution to this problem is quite easy: We simply put calls to the methods defined by this class inside a **synchronized** block.

This is the general form of the **synchronized** statement:

synchronized(*object*) {

// statements to be synchronized }

Here, *object* is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of *object* occurs only after the current thread has successfully entered *object*’s monitor.

**Thread Communication Using wait(), notify() and NotifyAll()**

When a thread T is executing inside a synchronized method and needs access to a resource R which is temporaritly unavailable. Then T enters into the polling loop that waits for R, T ties up the object preventing other threads’ access to it. This solution is less optimal. A better solution is gien by an elegant inter-process communication mechanism via the **wait()**, **notify( )**, and **notifyAll( )** methods. These methods are implemented as **final** methods in **Object**, so all classes have them. All three methods can be called only from within a **synchronized** context. Although conceptually advanced from a computer science perspective, the rules for using these methods are actually quite simple:

• **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**.

• **notify( )** wakes up a thread that called **wait( )** on the same object.

• **notifyAll( )** wakes up all the threads that called **wait( )** with the highest priority thread gaining access to object.

These methods are declared within **Object**, as shown here:

final void notify( )

final void notifyAll( )

Additional forms of **wait( )** exist that allow us to specify a period of time to wait.

Although **wait( )** normally waits until **notify( )** or **notifyAll( )** is called, there is a possibility that in very rare cases the waiting thread could be awakened due to a *spurious wakeup*. In this case, a waiting thread resumes without **notify( )** or **notifyAll( )** having been called. In essence, the thread resumes for no apparent reason.

**Suspending, Resuming, and Stopping Threads**

Sometimes, suspending execution of a thread is useful. For example, a separate thread can be used to display the time of day. If the user doesn’t want a clock, then its thread can be suspended. Whatever the case, suspending a thread is a simple matter. Once suspended, restarting the thread is also a simple matter. The methods used achieve this are suspend() resume() anhd stop(). They have the following form:

final void resume()

final void suspend()

final void stop()

**Unit IV**

**Applets and Swing**

**Applet Basics**

* Applets are small programs that are designed for transmission over the Internet and run within a browser
* There are two general varieties
  + 1. AWT Based (It is original toolkit)
    2. Swing Based (Lightweight alternative)
* There are two ways to run an applet
  1. Using a browser
  2. Using the appletviewer tool provided by JDK

**How Applets differ from Console-based Programs**

* All applets created are subclasses of Applet
* Applets do not need a main method
* Applets must be run under an applet viewer or a Java-compatible browser
* User I/O is not accomplished with Java’s stream I/O classes. Instead, Applets use interface provided by the AWT or Swing
* Applets are event-driven. That is, an Applet waits until an event occurs.
* Run-time system notifies the applet about an event by calling event handler
* User initiates interaction with the Applet. These interactions are sent to the applet as events to which the applet must respond. In case of console-based program, when the program needs input, it will prompt the user.

**Example:**

import java.awt.\*;

import java.applet.\*;

public class SampleApplet extends Applet {

public void paint(Graphics g) {

g.dawString(“Java Applets are easy and simple”, 20,20); } }

The first import statement imports Abstract Window Toolkit classes. Applets interact with the users through AWT. AWT contains support for a window-based graphical user interface. The second import statement imports the applet package. This package contains Applet class and we must create a sub-class of Applet class. this program can be complied in the same way as a console-based program however, it requires an HTML file to load the applet. The HTML code for the above example is

<applet code=”SampleApplet.class” width=200 height=300>

</applet>

**A Complete Applet Skeleton**

All applets must override a set of methods that provide the basic mechanism to control the execution. These lifecycle methods are init(), start(), stop() and destroy() which are defined by the Applet class. The paint() method is inherited from AWT component class.

import java.awt.\*;

import java.applet.\*;

public class AppSk extends Applet

{

public void init() { // for initialization}

public void start(){ //to start or resume }

public void stop() { // to suspend execution }

public void destroy() { // to perform shutdown activities }

public void paint() { //to redisplay contents of window }

}

* init () is the first method to be called. It contains code to initialize variables and perform any other startup activities
* start() is called after init(). It is also called to re-start the applet after it has been stopped
* start() might be called more than once during the life cycle of an applet.
* When page containing an applet is left, the stop() method is called.
* However stop() dose not mean that applet is terminated. It might be restarted with a call to start() method if user returns to page
* The destroy() method is called when the applet is no longer needed.
* paint() method is called by the run-time system or when ever an applet must re-draw its output.
* The paint() method is called automatically whenever the window needs to be refreshed. The programmer never calls paint() .
* **repaint()**:-Programmer calls repaint() in order to obtain a rendering. repaint() then again call paint() to service repaint() method.
* The **update()** method : It is called e applet when some portion of the applet window is to be redrawn

**Using the Status Window**

In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running. This is done by calling showStatus() method defined by Applet. Its syntax is

showStatus(String msg);

**Example**

import java.awt.\*;

import java.applet.\*;

public class StatusWindow extends Applet {

public void paint(Graphics g) {

g.dawString(“This is an Applet Window”, 20,20);

showStatus(“This is shown in the status window”); } }

**Passing Parameters to Applets**

Parameters can be passes to an applet using PARAM attribute of APPLET tag , specifying the name of the parameter and the value. Parameters are received by an applet using getParameter() method defend by Applet. Its general form is :

String getParameter(String paramName);

Here, paramName is the name of the parameter. It returns the value in the form of a String object. If any other types of parameters are passed, they need to be converted to String.

**Example**

import java.awt.\*;

import java.applet.\*;

/\* <applet code =Param.class width =300 height=80>

<param name=author value=”Herbert Shildt” >

<param name=purpose value=”To Demonstrate Parameters” >

<param name=version value=2 > </applet> \*/

public class Param extends Applet {

String author, purpose;

int ver;

public void start(){

String temp;

author=getParameter(“author”);

if (author==null)

author=”not found”

purpose=getParameter(“purpose”);

if (purpose==null)

purpose=”not found”;

try{

if(temp!=null)

ver=Integer.parseInt(temp);

else

ver=0; }

catch(NumberFormatException e)

{ ver=-1; } }

public void paint(Graphics g) {

g.dawString(“Purpose : ”+purpose, 10,20);

g.dawString(“By : ”+author, 10,40);

g.dawString(“Version : ”+ver, 10,60); }}

**The Applet Class**

* All applets are the subclasses of the Applet class.
* Applet inherits Component, Container and Panel superclasses defined by the AWT.
* An applet has access to full functionality of the AWT

|  |  |
| --- | --- |
| **Method** | **Description** |
| void destroy() | Called by the browser just before an applet is terminated. If overridden, this method will perform any cleanup before destruction. |
| AccessibleContext getAccessibleContext() | Returns the accessibility context for the invoking object. |
| AppletContext getAppletContext() | Returns the context associated with the applet. |
| String getAppletInfo() | Returns the string that describes the applet. |
| AudioClip getAudioClip(URL url) | Returns the AudioClip object that encapsulates the audio clip found at the location specified by url. |
| AudioClip getAudioClip(URL url, String ClipName) | Returns the AudioClip object that encapsulates the audio clip found at the location specified by url and having the name specified by clipName. |
| URL getCodeBase() | Returns the URL associated with the invoking applet. |
| URL getDocumentBase() | Returns the URL of the HTML document that invokes the applet. |
| Image getImage(URL url) | Returns an image object that encapsulates the image found at the location specified by url. |
| Image getImage(URL url , String imageName) | Returns an image object that encapsulates the image found at the location specified by url and having the name specified by imageName. |
| Locate getLocate() | Returns a Locate object that is used by various locate sensitive classes and methods. |
| String getParameter(String paramName) | Returns the parameter associated with paramName. Null is returned if the specified parameter is not found. |
| String[][] getParameterInfo() | If overridden, returns a String table that describes the parameters recognised by the applet. Each entry in the table must consist of three strings that contain the name of the parameter, a description and an explanation of its purpose. The default implementation returns null. |
| void init() | Called when an applet begins execution. It is the first method called for an applet. |
| boolean isActive() | Returns true if the applet has been started. It returns false if the applet has been stopped. |
| Static final AudioClip newAudioclip(URL url) | Returns an AudioClip object that encapsulates the audio clip found at the location specified by url. This method is similar to getAudioClip() except that it is static and cab be executed without the need for an Applet object. |
| void play(URL url) | If an audio clip is found at the location specified by url, the clip is played. |
| void play(URL url, String clipName) | If an audio clip is found at the location specified by url with th name specified by clipName, the clip is played. |
| void resize(Dimension dim) | Resizes the applet according to the dimension specified by dim. Dimension is a class stored inside java.awt. It contains two integer fields: width and height. |
| final void setStub(AppletStub stubObj) | Makes stubObj the stub for the applet. This method is used by the run-time system and is not usually called by the applet. A stub is a small piece of code that provides linkage between applet and the browser. |
| void showStatus(String str) | Displays str in the status window of the browser or applet viewer. If the browser does not support a status window, then no action takes place. |
| void start() | Called by the browser when an applet should start(or resume) execution. It is automatically called after init() when an applet first begins. |
| void stop() | Called by the browser to suspend execution of the applet. once stopped, an applet is restarted when the browser calls start(). |

**Event Handling**

Java applets are event-driven. Most events to which the program will respond are generated by the user. These events are passed to the program in a variety of ways. There are several types of events, including those generated by the mouse, the keyboard, and various controls like push buttons. AWT-based events are supported by the **java.awt.event** package.

**The Delegation Event Model**

This model defines standard and consistent mechanisms to generate and process events. A **source** generates an event and sends it to one or more **listeners**. Listener waits until it receives an event. Once received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to “delegate” the processing of an event to a separate piece of code. In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them.

**Events**

In the delegation model, an event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Many other user operations could also be cited as examples.

Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, software or hardware failure occurs, or an operation is completed.

**Event Sources**

A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event. A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

public void addTypeListener(TypeListener el)

Here, Type is the name of the event, and el is a reference to the event listener. For example, the method that registers a keyboard event listener is called addKeyListener( ). The method that registers a mouse motion listener is called addMouseMotionListener( ). When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as multicasting the event. In all cases, notifications are sent only to listeners that register to receive them.

A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

public void removeTypeListener(TypeListener el)

Here, Type is the name of the event, and el is a reference to the event listener. For example, to remove a keyboard listener, removeKeyListener( ) is to be called.

**Event Listeners**

A listener is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.

The methods that receive and process events are defined in a set of interfaces found in java.awt.event. For example, the MouseMotionListener interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface.

**Event Classes**

The classes that represent events are at the core of Java’s event handling mechanism. Thus, a discussion of event handling must begin with the event classes At the root of the Java event class hierarchy is EventObject, which is in java.util. It is the superclass for all events. The package java.awt.event defines several types of events generated by various user interface elements.

**Event Listener Interface**

Event listeners receive event notifications. Listeners for AWT-based events are created by implementing one or more of the interfaces defined by the java.awt.event package. When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its arguments. Commonly used event classes in java.awt.event are given below:

|  |  |
| --- | --- |
| **Event Class** | **Description** |
| ActionEvent | Generated when a button is pressed, a list item is double-clicked, or a menu item is selected. |
| AdjustmentEvent | Generated when a scroll bar is manipulated |
| ComponentEvent | Generated when a component is hidden, moved, resized, or becomes visible. |
| ContainerEvent | Generated when a component is added to or removed from a container |
| FocusEvent | Generated when a component gains or loses keyboard focus. |
| InputEvent | Abstract superclass for all component input event classes. |
| ItemEvent | Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected |
| KeyEvent | Generated when input is received from the keyboard. |
| MouseEvent | Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component. |
| MouseWheelEvent | Generated when the mouse wheel is moved. |
| TextEvent | Generated when the value of a text area or text field is changed. |
| WindowEvent | Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit. |

Commonly used event listener interfaces are listed in the table below

|  |  |
| --- | --- |
| **Interface** | **Description** |
| ActionListener | Defines one method to receive action events. Action events are generated by push buttons menus etc |
| AdjustmentListener | Defines one method to receive adjustment events. |
| ComponentListener | Defines four methods to recognize when a component is hidden, moved, resized, or shown. |
| ContainerListener | Defines two methods to recognize when a component is added to or removed from a container. |
| FocusListener | Defines two methods to recognize when a component gains or loses keyboard focus. |
| ItemListener | Defines one method to recognize when the state of an item changes. |
| KeyListener | Defines three methods to recognize when a key is pressed, released, or typed. |
| MouseListener | Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released. |
| MouseMotionListener | Defines two methods to recognize when the mouse is dragged or moved. |
| MouseWheelListener | Defines one method to recognize when the mouse wheel is moved. |
| TextListener | Defines one method to recognize when a text value changes. |
| WindowListener | Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit. |

**Applet Programming Using Delegation Event Model**

Using the delegation event model is actually quite easy. Just follow these two steps:

1. Implement the appropriate interface in the listener so that it will receive the type of event desired.
2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

A source may generate several types of events. Each event must be registered separately. Also, an object may register to receive several types of events, but it must implement all of the interfaces that are required to receive these events.

**Handling Mouse and Mouse Motion Events**

To handle mouse events, we must implement the MouseListener and the MouseMotionListener interfaces. The MouseListener interface defines five methods.

|  |  |
| --- | --- |
| void mouseClicked(MuseEvent me) | Invoked when a mouse button is clicked |
| void mouseEntered(MouseEvent me) | Invoked when mouse enters a component |
| void mouseExited(MouseEvent me) | Invoked when mouse exits from a component |
| void mousePressed(MouseEvent me) | Invoked when a mouse button is pressed |
| void mouseReleased(MouseEvent me) | Invoked when a mouse button is released |

If a mouse button is clicked, mouseClicked() is invokeThe MouseListener interface defines five methods. If a mouse button is clicked, mouseClicked() is invoked.

Example

import java.awt.event.\*;

import java.applet.\*;

/\*

<applet code="MouseEvents" width=300 height=100>

</applet>

\*/

public class MouseEvents extends Applet

implements MouseListener, MouseMotionListener {

String msg = "";

int mouseX = 0, mouseY = 0; // coordinates of mouse

public void init() {

addMouseListener(this);

addMouseMotionListener(this); }

public void mouseClicked(MouseEvent me) {

// save coordinates

mouseX = 0;

mouseY = 10;

msg = "Mouse clicked.";

repaint(); }

public void mouseEntered(MouseEvent me) {

// save coordinates

mouseX = 0;

mouseY = 10;

msg = "Mouse entered.";

repaint(); }

public void mouseExited(MouseEvent me) {

// save coordinates

mouseX = 0;

mouseY = 10;

msg = "Mouse exited.";

repaint(); }

public void mousePressed(MouseEvent me) {

// save coordinates

mouseX = me.getX();

mouseY = me.getY();

msg = "Down";

repaint(); }

public void mouseReleased(MouseEvent me) {

// save coordinates

mouseX = me.getX();

mouseY = me.getY();

msg = "Up";

repaint(); }

public void paint(Graphics g) {

g.drawString(msg, mouseX, mouseY); } }

**The transient and volatile Modifiers**

These modifiers handle special situations. When an instance variable is declared as transient, then its value need not persist when an object is stored. A transient field does not affect the state of an object. The volatile modifier tells the compiler that a variable can be changed unexpectedly by other parts of the program. One of these situations involve multithreaded programs. In a multithreaded program, sometimes two or more threads share same variable. In such situations that variable can be declared as volatile.

**instanceof** : Sometimes it is useful to know the type of an object during run time. It is useful in the following situations:

1. One thread generates various types of objects and another thread may process these objects. In this situation, it is useful for the processing thread to know the type of each object received.
2. Knowledge of an object type at run time is important in case of casting. A superclass reference can refer to subclass objects, it is not always possible to know at compile time whether a cast is valid or not.

The instanceof keyword addresses these types of situations. Its general form is

objref instanceof type 🡪 here, objref is a reference to an instance of a class, and type is a class or interface. If objref is of specific type, then instanceof operator returns true else it returns false.

**strictfp**: When Java 2 was released, the floating point model was relaxed slightly. It doesn’t require the truncation of certain intermediate values that occur during computation. This prevents overflow and underflow. By modifying a class, method or interface with strictfp, we can ensure floating point calculations take place precisely.

**assert**: The assert keyword is used to create an assertion, which is a condition that is expected to be true during the execution of the program. For example, we may require a method which always returns a positive integer value. We can test this by asserting that the return value is greater than zero using an assert statement. At run time, if the condition is actually true, no other action takes place. However, if the condition is false, an AssertionError is thrown. Assertion is often used in testing to verify that some expected condition is actually met.

**Syntax:** assert condition;

**Exampe**: assert n>0;

**Native Methods**

We may require to call a subroutine that is written in a language other than Java. Such subroutine will exist as executable code. This is called native code. We may wish to call a native method for faster execution or we may use a special third party library such as a statistical package. Java provides native keyword, which is used to declare native code methods. Once declared, these methods can be called from a Java program just as a Java method. To declare a native method, one just needs native keyword and method declaration but body of the method is not needed.

**Example** : public native int meth();

The AWT supports the following types of controls:

* Labels
* Push buttons
* Check boxes
* Choice lists
* Scroll bars
* Text editing

These controls are subclasses of **Component**.

**Adding and Removing Controls**

To include a control in a window, it must be added to the window. To do this, first an instance of the desired control must be created and then it must be added it to a window by calling **add( )**, which

is defined by **Container**.

Component add(Component compObj)

Here, *compObj* is an instance of the control to be added. A reference to *compObj* is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed.

A control which is no longer needed can be removed from the window. To do this, **remove( )** is used. This method is also defined by **Container**. It has this general form:

void remove(Component obj)

Here, *obj* is a reference to the control you want to remove. All controls can be removed by calling **removeAll( )**.

**Responding to Controls**

Except for labels, which are passive, all controls generate events when they are accessed by the user. For example, when the user clicks on a push button, an event is sent that identifies the push button. The program simply implements the appropriate interface and then registers an event listener for each control that need to be monitored.

**The HeadlessException**

Most of the AWT controls described in this chapter now have constructors that can throw a **HeadlessException** when an attempt is made to instantiate a GUI component in a non-interactive environment (such as one in which no display, mouse, or keyboard is present). This exception can be used to write code that can adapt to non-interactive environments.

**Labels**

The easiest control to use is a label. A *label* is an object of type **Label**, and it contains a string,

which it displays. Labels are passive controls that do not support any interaction with the

user. **Label** defines the following constructors:

Example:

Label( ) throws HeadlessException

Label(String str) throws HeadlessException

Label(String str, int how) throws HeadlessException

The text in a label can be changed by using the **setText( )** method. The caption of the current label can by calling **getText( )**. These methods are shown here:

void setText(String *str*)

String getText( )

For **setText( )**, *str* specifies the new label. For **getText( )**, the current label is returned. The alignment of the string within the label can be changed by calling **setAlignment( )**.

To obtain the current alignment, call **getAlignment( )**. The methods are as follows:

void setAlignment(int how)

int getAlignment( )

Here, *how* must be one of the alignment constants shown earlier.

The following example creates three labels and adds them to an applet window:

// Demonstrate Labels

import java.awt.\*;

import java.applet.\*;

/\*

<applet code="LabelDemo" width=300 height=200>

</applet>

\*/

public class LabelDemo extends Applet {

public void init() {

Label one = new Label("One");

Label two = new Label("Two");

Label three = new Label("Three");

// add labels to applet window

add(one);

add(two);

add(three); } }

**Using Buttons**

A *push button* is a component that contains a label and that generates an event when it is pressed. Push buttons are objects of type **Button**. Button defines these two constructors:

Button( ) throws HeadlessException

Button(String *str*) throws HeadlessException

The first version creates an empty button. The second creates a button that contains *str* as a label.

After a button has been created, its label can be set by calling **setLabel( )**. The text associated with a button can be retrieved by calling **getLabel( )**.

These methods are as follows:

void setLabel(String *str*)

String getLabel( )

Here, *str* becomes the new label for the button.

**Handling Buttons**

Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered an interest in receiving action event notifications from that component.

Each listener implements the **ActionListener** interface. That interface defines the **actionPerformed()** method, which is called when an event occurs. An **ActionEvent** object is supplied as the argument to this method. It contains both a reference to the button that generated the event and a reference to the *action command string* associated with the button. By default, the action command string is the label of the button. Usually, either the button reference or the action command string can be used to identify the button.

Here is an example that creates three buttons labeled “Yes”, “No”, and “Undecided”. Each time one is pressed, a message is displayed that reports which button has been pressed.

// Demonstrate Buttons

import java.awt.\*;

import java.awt.event.\*;

import java.applet.\*;

/\*

<applet code="ButtonDemo" width=250 height=150>

</applet>

\*/

public class ButtonDemo extends Applet implements ActionListener {

String msg = "";

Button yes, no, maybe;

public void init() {

yes = new Button("Yes");

no = new Button("No");

maybe = new Button("Undecided");

add(yes);

add(no);

add(maybe);

yes.addActionListener(this);

no.addActionListener(this);

maybe.addActionListener(this); }

public void actionPerformed(ActionEvent ae) {

String str = ae.getActionCommand();

if(str.equals("Yes")) {

msg = "You pressed Yes."; }

else if(str.equals("No")) {

msg = "You pressed No.";

}

else {

msg = "You pressed Undecided.";}

repaint(); }

public void paint(Graphics g) {

g.drawString(msg, 6, 100); } }

**Applying Check Boxes**

A *check box* is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents. The state of a check box can be changed by clicking on it. Check boxes can be used individually or as part of a group. Check boxes are objects of the **Checkbox** class.

**Checkbox** supports these constructors:

Checkbox( ) throws HeadlessException

Checkbox(String *str*) throws HeadlessException

Checkbox(String *str*, boolean *on*) throws HeadlessException

The first form creates a check box whose label is initially blank. The state of the check box is unchecked. The second form creates a check box whose label is specified by *str.* The state of the check box is unchecked. The third form allows to set the initial state of the check box. If *on* is **true**, the check box is initially checked; otherwise, it is cleared. The value of *on* determines the initial state of the check box.

To retrieve the current state of a check box, call **getState( )**. To set its state, call **setState( )**. The current label associated with a check box can be obtained by calling **getLabel( )**. To set the label, **setLabel( )** can be called. These methods are as follows:

boolean getState( )

void setState(boolean *on*)

String getLabel( )

void setLabel(String *str*)

Here, if *on* is **true**, the box is checked. If it is **false**, the box is cleared. The string passed in *str* becomes the new label associated with the invoking check box.

**Handling Check Boxes**

Each time a check box is selected or deselected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the **ItemListener** interface. That interface defines the **itemStateChanged( )** method. An **ItemEvent** object is supplied as the argument to this method. It contains information about the event (for example, whether it was a selection or de-selection).

**CheckboxGroup**

It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called

*radio buttons,* because they act like the station selector on a car radio—only one station can be selected at any one time. To create a set of mutually exclusive check boxes, Check box groups are objects of type **CheckboxGroup**. Only the default constructor is defined, which creates an empty group. One can determine which check box in a group is currently selected by calling

**getSelectedCheckbox( )**. A check box can that is selected can be obtained by calling **setSelectedCheckbox( )**. These methods are as follows:

Checkbox getSelectedCheckbox( )i

void setSelectedCheckbox(Checkbox *which*)

**Choice Controls**

The **Choice** class is used to create a *pop-up list* of items from which the user may choose. Thus, a **Choice** control is a form of menu. When inactive, a **Choice** component takes up only enough space to show the currently selected item. When the user clicks on it, the whole list of choices pops up, and a new selection can be made. Each item in the list is a string that appears as a left-justified label in the order it is added to the **Choice** object. **Choice** only defines the default constructor, which creates an empty list. To add a selection to the list, call **add( )**. It has this general form:

void add(String *name*)

Here, *name* is the name of the item being added. Items are added to the list in the order in which calls to **add( )** occur.

To determine which item is currently selected, you may call either **getSelectedItem( )** or **getSelectedIndex( )**. These methods are shown here:

String getSelectedItem( )

int getSelectedIndex( )

The **getSelectedItem( )** method returns a string containing the name of the item.

**getSelectedIndex( )** returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected.

To obtain the number of items in the list, call **getItemCount( )**. You can set the currently selected item using the **select( )** method with either a zero-based integer index or a string that will match a name in the list. These methods are shown here:

int getItemCount( )

void select(int *index*)

void select(String *name*)

Given an index, the name associated with the item at that index can be obtained by calling **getItem()**, which has this general form:

String getItem(int *index*)

Here, *index* specifies the index of the desired item.

**Using Lists**

The **List** class provides a compact, multiple-choice, scrolling selection list. Unlike the **Choice** object, which shows only the single selected item in the menu, a **List** object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections. **List** provides these constructors:

List( ) throws HeadlessException

List(int *numRows*) throws HeadlessException

List(int *numRows*, boolean *multipleSelect*) throws HeadlessException

The first version creates a **List** control that allows only one item to be selected at any one time. In the second form, the value of *numRows* specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed). In the third form, if *multipleSelec*t is **true**, then the user may select two or more items at a time. If it is **false**, then only one item may be selected.

To add a selection to the list, call **add( )**. It has the following two forms:

void add(String *name*)

void add(String *name*, int *index*)

Here, *name* is the name of the item added to the list. The first form adds items to the end of the list. The second form adds the item at the index specified by *index.* Indexing begins at zero. If the value specified is –1 the item gets added to the end of the list.

For lists that allow only single selection, you can determine which item is currently selected by calling either **getSelectedItem( )** or **getSelectedIndex( )**. These methods are shown here:

String getSelectedItem( )

int getSelectedIndex( )

The **getSelectedItem( )** method returns a string containing the name of the item. If more than one item is selected, or if no selection has yet been made, **null** is returned. **getSelectedIndex( )**  returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, –1 is returned.

For lists that allow multiple selection, you must use either **getSelectedItems( )** or **getSelectedIndexes( )**, shown here, to determine the current selections:

String[ ] getSelectedItems( )

int[ ] getSelectedIndexes( )

**getSelectedItems( )** returns an array containing the names of the currently selected items. **getSelectedIndexes( )** returns an array containing the indexes of the currently selected items.

To obtain the number of items in the list, call **getItemCount( )**. You can set the currently selected item by using the **select( )** method with a zero-based integer index. These methods

are shown here:

int getItemCount( )

void select(int *index*)

Given an index, the name associated with the item at that index can be obtained by calling **getItem()**, which has this general form:

String getItem(int *index*)

Here, *index* specifies the index of the desired item.

**Managing Scroll Bars**

*Scroll bars* are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. A scroll bar is actually a composite of several individual parts. Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow. The current value of the scroll bar relative to its minimum and maximum values is indicated by the *slider box* (or *thumb*) for the scroll bar. The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value. In the background space on either side of the thumb, the user can click to cause the thumb to jump in that direction by some increment larger than 1. Typically, this action translates into some form of page up and page down. Scroll bars are encapsulated by the **Scrollbar** class.

**Scrollbar** defines the following constructors:

Scrollbar( ) throws HeadlessException

Scrollbar(int *style*) throws HeadlessException

Scrollbar(int *style*, int *initialValue*, int *thumbSize*, int *min*, int *max*) throws HeadlessException

The first form creates a vertical scroll bar. The second and third forms allow you to specify the orientation of the scroll bar. If *style* is **Scrollbar.VERTICAL**, a vertical scroll bar is created. If *style* is **Scrollbar.HORIZONTAL**, the scroll bar is horizontal. In the third form of the constructor, the initial value of the scroll bar is passed in *initialValue.* The number of units represented by the height of the thumb is passed in *thumbSize.* The minimum and maximum values for the scroll bar are specified by *min* and *max.*

The minimum and maximum values of the scroll bar can be retrieved via **getMinimum( )** and

**getMaximum( )**, shown here:

int getMinimum( )

int getMaximum( )

By default, 1 is the increment added to or subtracted from the scroll bar each time it is scrolled up or down one line. The increment can be changed by calling **setUnitIncrement( )**.

By default, page-up and page-down increments are 10. This value can be changed by calling

**setBlockIncrement( )**. These methods are shown here:

void setUnitIncrement(int *newIncr*)

void setBlockIncrement(int *newIncr*)

**Handling Scroll Bars**

To process scroll bar events, the **AdjustmentListener** interface is to be implemented. Each time a user interacts with a scroll bar, an **AdjustmentEvent** object is generated. Its **getAdjustmentType( )** method can be used to determine the type of the adjustment. The types of adjustment events are as follows:

* BLOCK\_DECREMENT A page-down event has been generated.
* BLOCK\_INCREMENT A page-up event has been generated.
* TRACK An absolute tracking event has been generated.
* UNIT\_DECREMENT The line-down button in a scroll bar has been pressed.
* UNIT\_INCREMENT The line-up button in a scroll bar has been pressed.

**Using a TextField**

The **TextField** class implements a single-line text-entry area, usually called an *edit control.* Text fields allow the user to enter strings and to edit the text using the arrow keys, cut andpaste keys, and mouse selections. **TextField** is a subclass of **TextComponent**. **TextField**defines the following constructors:

TextField( ) throws HeadlessException

TextField(int *numChars*) throws HeadlessException

TextField(String *str*) throws HeadlessException

TextField(String *str*, int *numChars*) throws HeadlessException

The first version creates a default text field. The second form creates a text field that is *numChars* characters wide. The third form initializes the text field with the string contained in *str.* The fourth form initializes a text field and sets its width. **TextField** (and its superclass **TextComponent**) provides several methods that allow utilizing a text field. To obtain the string currently contained in the text field, **getText( )**can be called. To set the text, **setText( )** can becalled. These methods are as follows:

String getText( )

void setText(String *str*)

Here, *str* is the new string. The user can select a portion of the text in a text field. Also, you can select a portion of text under program control by using **select( )**. Your program can obtain the currently selected text by calling **getSelectedText( )**. These methods are shown here:

String getSelectedText( )

void select(int *startIndex*, int *endIndex*)

the editability of a textfield cab be determined by calling **isEditable( )**. These methods

are shown here:

boolean isEditable( )

void setEditable(boolean *canEdit*)

**isEditable( )** returns **true** if the text may be changed and **false** if not. In **setEditable( )**, if *canEdit* is **true**, the text may be changed. If it is **false**, the text cannot be altered. When we don’t want the text to be displayed, such as a password, the echoing of the characters as they are typed can be disabled by calling **setEchoChar( )**. This method specifies a single character that the **TextField** will display when characters are entered (thus, the actual characters typed will not be shown). The echo character can be retrieved by calling the **getEchoChar( )** method. These methods are as follows:

void setEchoChar(char *ch*)

boolean echoCharIsSet( )

char getEchoChar( )

Here, *ch* specifies the character to be echoed.

**Using a TextArea**

Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called **TextArea**. Following are the constructors for this**:**

TextArea( ) throws HeadlessException

TextArea(int *numLines*, int *numChars*) throws HeadlessException

TextArea(String *str*) throws HeadlessException

TextArea(String *str*, int *numLines*, int *numChars*) throws HeadlessException

TextArea(String *str*, int *numLines*, int *numChars*, int *sBars*) throws HeadlessException

Here, *numLines* specifies the height, in lines, of the text area, and *numChars* specifies its width, in characters. Initial text can be specified by *str.* In the fifth form, we can specify the scroll *sBars* must be one of these values

SCROLLBARS\_BOTH SCROLLBARS\_NONE

SCROLLBARS\_HORIZONTAL\_ONLY SCROLLBARS\_VERTICAL\_ONLY

**TextArea** is a subclass of **TextComponent**. Therefore, it supports the **getText( )**, **setText( )**,

**getSelectedText( )**, **select( )**, **isEditable( )**, and **setEditable( )** methods described earlier.

**TextArea** adds the following methods:

void append(String *str*)

void insert(String *str*, int *index*)

void replaceRange(String *str*, int *startIndex*, int *endIndex*)

The **append( )** method appends the string specified by *str* to the end of the current text. **insert( )** inserts the string passed in *str* at the specified index. To replace text, call **replaceRange( )**. Itreplaces the characters from *startIndex* to *endIndex*–1, with the replacement text passed in *str.*

**Understanding Layout Managers**

A layout manager automatically arranges the controls within a window by using some type of algorithm. It is very tedious to manually lay out a large number of components and in some cases, the width and height information is not available when arranging the controls, Each **Container** object has a layout manager associated with it. A layout manager is an instance of any class that implements the **LayoutManager** interface. The layout manager is set by the **setLayout( )** method. If no call to **setLayout( )** is made, then the default layout manager is used. Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it. The **setLayout( )** method has the following general form:

void setLayout(LayoutManager *layoutObj*)

Here, *layoutObj* is a reference to the desired layout manager.

Each layout manager keeps track of a list of components that are stored by their names. The layout manager is notified each time when a component is added to a container.

**FlowLayout**

**FlowLayout** is the default layout manager.  **FlowLayout** implements a simple layout style, which is similar tohow words flow in a text editor. The direction of the layout is governed by the container’scomponent orientation property, which, by default, is left to right, top to bottom. Therefore,by default, components are laid out line-by-line beginning at the upper-left corner. In allcases, when a line is filled, layout advances to the next line. A small space is left betweeneach component, above and below, as well as left and right. Here are the constructors for

**FlowLayout**:

FlowLayout( )

FlowLayout(int *how*)

FlowLayout(int *how*, int *horz*, int *vert*)

The first form creates the default layout, which centers components and leaves five pixels of space between each component. The second form lets you specify how each line is aligned.

Valid values for *how* are as follows:

FlowLayout.LEFT

FlowLayout.CENTER

FlowLayout.RIGHT

FlowLayout.LEADING

FlowLayout.TRAILING

**BorderLayout**

The **BorderLayout** class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north, south, east, and west. The middle area is called the center. Here are the constructors defined

by BorderLayout:

BorderLayout( )

BorderLayout(int horz, int vert)

The first form creates a default border layout. The second allows you to specify the horizontal and vertical space left between components in horz and vert, respectively. BorderLayout defines the following constants that specify the regions:

BorderLayout.CENTER BorderLayout.SOUTH

BorderLayout.EAST BorderLayout.WEST

BorderLayout.NORTH

When adding components, we can specify the regions with **add( )**, which is defined by **Container**:

void add(Component *compObj*, Object *region*)

Here, *compObj* is the component to be added, and *region* specifies where the component will be added

**GridLayout**

**GridLayout** lays out components in a two-dimensional grid. When we instantiate a GridLayout, we define the number of rows and columns. The constructors supported by GridLayout are shown here:

GridLayout( )

GridLayout(int *numRows*, int *numColumns*)

GridLayout(int *numRows*, int *numColumns*, int *horz*, int *vert*)

The first form creates a single-column grid layout. The second form creates a grid layout with the specified number of rows and columns. The third form allows to specify the horizontal and vertical space left between components in *horz* and *vert,* respectively. Either *numRows* or *numColumns* can be zero. Specifying *numRows* as zero allows for unlimited-length columns. Specifying *numColumns* as zero allows for unlimited-length rows.

// Demonstrate GridLayout

import java.awt.\*;



import java.applet.\*;

/\*

<applet code="GridLayoutDemo" width=300 height=200>

</applet>

\*/

public class GridLayoutDemo extends Applet {

static final int n = 4;

public void init() {

setLayout(new GridLayout(n, n));

setFont(new Font("SansSerif", Font.BOLD, 24));

for(int i = 0; i < n; i++) {

for(int j = 0; j < n; j++) {

int k = i \* n + j;

if(k > 0)

add(new Button(" " + k)); }

} } }

**Menu Bars and Menus**

Atop-level window can have a menu bar associated with it. A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu. This concept is implemented in the AWT by the following classes: **MenuBar**, **Menu**, and **MenuItem**. In general, a menu bar contains one or more **Menu** objects. Each **Menu** object contains a list of **MenuItem** objects. Each **MenuItem** object represents something that can be selected by the user. Since **Menu** is a subclass of **MenuItem**, a hierarchy of nested submenus can be created.

To create a menu bar, first create an instance of **MenuBar**. This class only defines the default constructor. Next, create instances of **Menu** that will define the selections displayed on the bar. Following are the constructors for **Menu**:

Menu( ) throws HeadlessException

Menu(String *optionName*) throws HeadlessException

Menu(String *optionName*, boolean *removable*) throws HeadlessException

Here, *optionName* specifies the name of the menu selection. If *removable* is **true**, the menu can be removed and allowed to float free. Otherwise, it will remain attached to the menu bar. (Removable menus are implementation-dependent.) The first form creates an empty menu.

Individual menu items are of type **MenuItem**.

It defines these constructors:

MenuItem( ) throws HeadlessException

MenuItem(String *itemName*) throws HeadlessException

MenuItem(String *itemName*, MenuShortcut *keyAccel*) throws HeadlessException

Here, *itemName* is the name shown in the menu, and *keyAccel* is the menu shortcut for this item.

A menu item can be enabled or disabled by using the **setEnabled( )** method. Its form is shown here:

void setEnabled(boolean *enabledFlag*)

If the argument *enabledFlag* is **true**, the menu item is enabled. If **false**, the menu item is disabled.

The status on an item is obtained by calling **isEnabled( )**.

**isEnabled( )** returns **true** if the menu item on which it is called is enabled. Otherwise, it returns **false**.

The name of a menu item can be changed by calling **setLabel( )**. The current name of the menu item can be obtained by using **getLabel( )**. These methods are as follows:

void setLabel(String *newName*)

String getLabel( )

Here, *newName* becomes the new name of the invoking menu item. **getLabel( )** returns the current name.

Menus only generate events when an item of type **MenuItem** is selected. They do not generate events when a menu bar is accessed to display a drop-down menu, for example. Each time a menu item is selected, an **ActionEvent** object is generated. By default, the action command string is the name of the menu item. However, you can specify a different action command string by calling **setActionCommand( )** on the menu item. Each time a check box menu item is checked or unchecked, an **ItemEvent** object is generated. Thus, you must implement the **ActionListener** and/or **ItemListener** interfaces in order to handle these menu events.

The **getItem( )** method of **ItemEvent** returns a reference to the item that generated this

event. The general form of this method is shown here:

Object getItem( )

**Introducing Swing**

Swing was a response to deficiencies present in Java’s original GUI subsystem: the Abstract Window Toolkit. The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface. One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents, or peers. This means that the look and feel of a component is defined by the platform, not by Java. Because the AWT components use native code resources, they are referred to as heavyweight.

**Limitations of AWT Components**

1. Because of variations between operating systems, a component might look, or even act, differently on different platforms. This potential variability threatened the overarching philosophy of Java: write once, run anywhere.
2. The look and feel of each component was fixed (because it is defined by the platform) and could not be (easily) changed.
3. The use of heavyweight components caused some frustrating restrictions. For example, a heavyweight component is always rectangular and opaque.

**Advantages of Swing**

Swing was created to address the limitations present in the AWT. It does this through two key features: lightweight components and a pluggable look and feel. Together they provide an elegant, yet easy-to-use solution to the problems of the AWT.

**Swing Components Are Lightweight**

Swing components are lightweight. This means that they are written entirely in Java and do not map directly to platform-specific peers. Lightweight components are rendered using graphics primitives; they can be transparent, which enables nonrectangular shapes. Thus, lightweight components are more efficient and more flexible. Because lightweight components do not translate into native peers, the look and feel of each component is determined by Swing, not by the underlying operating system. This means that each component will work in a consistent manner across all platforms.

**Swing Supports a Pluggable Look and Feel**

Because each Swing component is rendered by Java code rather than by native peers, the look and feel of a component is under the control of Swing. This fact means that it is possible to separate the look and feel of a component from the logic of the component, and this is what Swing does. Separating out the look and feel provides a significant advantage: it becomes possible to change the way that a component is rendered without affecting any of its other aspects. In other words, it is possible to “plug in” a new look and feel for any given component without creating any side effects in the code that uses that component.

The pluggable look of swing is made possible because it uses the classic model-view-controller (MVC) architecture. In MVC terminology, the model corresponds to the state information associated with the component. For example, in the case of a check box, the model contains a field that indicates if the box is checked or unchecked. The view determines how the component is displayed on the screen, including any aspects of the view that are affected by the current state of the model. The controller determines how the component reacts to the user. For example, when the user clicks a check box, the controller reacts by changing the model to reflect the user’s choice (checked or unchecked). This then results in the view being updated. By separating a component into a model, a view, and a controller, the specific implementation of each can be changed without affecting the other two. For instance, different view implementations can render the same component in different ways without affecting the model or the controller.

Although the MVC architecture and the principles behind it are conceptually sound, the high level of separation between the view and the controller is not beneficial for Swing components. Instead, Swing uses a modified version of MVC that combines the view and the controller into a single logical entity called the UI delegate. For this reason, Swing’s approach is called either the Model-Delegate architecture or the Separable-Model architecture. Therefore, although Swing’s component architecture is based on MVC, it does not use a classical implementation of it.

**Components and Containers**

Swing defines two types of items: components and containers. Component is an independent visual control, such as push button or text field. A container holds a group of components. If a component needs to be displayed, it must be held within a container. Because containers are also components, a container can also hold other containers. This enables Swing to define containment hierarchy.

**Components**

Swing components are derived from the JComponent class. JComponent provides the functionality that is common to all components. For example, JComponent supports the pluggable look and feel. JComponent inherits the AWT classes Container and Component. Thus, a Swing component is built on and compatible with an AWT component. All of Swing’s components are represented by classes defined within the package javax.swing. The following table shows the class names for Swing components (including those used as containers).

|  |  |  |  |
| --- | --- | --- | --- |
| JApplet JColorChooser | JButton | JCheckBox | JCheckBoxMenuItem |
| JColorChooser | JComboBox | JComponent | JDesktopPane |
| JDialog | JEditorPane | JFileChooser | JFormattedTextField |
| JFrame | JInternalFrame | JLabel | JLayer |
| JLayeredPlane | JList | JMenu | JMenuBar |
| JMenuItem | JOptionPlane | JPlane | JPasswordField |
| JPopupMenu | JProgressBar | JRadioButton | JRadioButtonMenuItem |
| JRootPane | JScrollBar | JSCrollPane | JSeperator |
| JSlider | JSpinner | JSplitPane | JTabbedPane |
| JTable | JTextArea | JTextField | JtextPane |
| JToggleButton | JToolBar | JToolTip | JTree |
| JViewport | JWindow |  |  |

**Containers**

Swing defines two types of containers. The first are top-level containers: **JFrame**, **JApplet**, **JWindow**, and **JDialog**. These containers do not inherit **JComponent**. They do, however, inherit the AWT classes **Component** and **Container**. Unlike Swing’s other components, which are lightweight, the top-level containers are heavyweight. This makes the top-level containers a special case in the Swing component library. As the name implies, a top-level container must be at the top of a containment hierarchy. A top-level container is not contained within any other container. Furthermore, every containment hierarchy must begin with a top-level container. The one most commonly used for applications is **JFrame**. The one used for applets is **JApplet**.

The second type of containers supported by Swing are lightweight containers. Lightweight containers doinherit **JComponent**. Examples of a lightweight container is **JPanel**, **JScrollPane** and **JRootPane** Lightweight containers are often used to organize and manage groups of related components because a lightweight container can be contained within another container. Thus lightweight containers such as **JPanel** can be usedto create subgroups of related controls that are contained within an outer container.

**Layout Managers**

Layout manager controls the position of component within a container. Java offers several layout managers. Most are provided by AWT. Here is a list of a few layout managers available:

|  |  |
| --- | --- |
| FlowLayout | A simple layout that positions components left-to-right, top-to-bottom |
| BorderLayout | Positions the components within the center or the borders of the container. |
| GridLayout | Lays out components within a grid |
| GirdBagLayout | Lays out different size components within a flexible grid |
| BoxLayout | Lays out components vertically or horizontally within a box |
| SpringLayout | Lays out components subject to a set of constraints |

* BorderLayout is the default layout. It defines five locations to which a component can be added. They are center, north, south east and west. By default, it is center. To add a component to one of the regions, it is necessary to specify the name of the region.
* FlowLayout lays out components one row at a time, top to bottom. When one row is full, layout advances to the next row.

// A simple Swing application.

import javax.swing.\*;

class SwingDemo {

SwingDemo() {

// Create a new JFrame container.

JFrame jfrm = new JFrame("A Simple Swing Application");

// Give the frame an initial size.

jfrm.setSize(275, 100);

// Terminate the program when the user closes the application.

jfrm.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

// Create a text-based label.

JLabel jlab = new JLabel(" Swing means powerful GUIs.");

// Add the label to the content pane.

jfrm.add(jlab);

// Display the frame.

jfrm.setVisible(true); }

public static void main(String args[]) {

// Create the frame on the event dispatching thread.

SwingUtilities.invokeLater(new Runnable() {

public void run() {

new SwingDemo(); }

});

} }

**Explanation**:

The **javax.swing** package contains the components and models defined by Swing. It defines classes that implement labels, buttons and other controls. This package will be included in all programs that use swing.

JFrame jfrm = new JFrame("A Simple Swing Application");

Will create a container called jfrm that defines a rectangular window, with title bar, maximize, minimize and close buttons with title specified by the constructor.

jfrm.setSize(275, 100);

The setSize() method sets dimensions of the window, which are specified in pixels. Its general form is void setSize(int width, int height);

By default, when top-level window is closed, the window is removed from the screen but application is not terminated. One can make the application terminate by specifying setDefaultCloseOperation() which has the general format void setDefaultCloseOperation(int what). The value passed in what determines what happens when the window is closed. There are several options:

JFrame.DESPOSE\_ON\_CLOSE, JFrame.HIDE\_ON\_CLOSE, JFrame.EXIT\_ON\_CLOSE, JFrame.DO\_NOTHING\_ON\_CLOSE are some of the options.

**Use of JButton**

One of the most commonly used Swing controls is the push button. It is an instance of JButton. Swing push buttons can contain text, image or both. JButton supplies several constructors. The one which is most commonly used is

**JButton**(String msg) The msg specifies the string that will be displayed inside the button.

When push button is pressed, it generates ActionEvent which is defined by AWT and also Swing. JButton provides the following methods, which are used to add or remove an action listener:

**JButton.addActionListener(ActionListener al);**

**JButton.removeActionListener(ActionListener al)** Here *al* specifies an object that will receive an instance of a class that implements ActionListener interface. The ActionListener interface defines actionPerformed() method. This method is called when a button is pressed.

**Example**: JButton reset = new JButton(“Reset”);

reset.addActionListener(this);

When we add more controls to a container, we must set the layout manager for the content pane as follows:

jfrm.setLayout(new FlowLayout());

**JTextField**

It is also a commonly used control. It defines several constructors. The most commonly used one is

JTextField(int cols) Here, cols specifies the width of text field in columns. A text which is longer than columns can be entered because cols specifies the physical size of the text field on screen. It is used for accepting input from the user. When accepting input, if enter key is pressed, an ActionEvent is generated. Therefore JTextField provides addActionListener() and removeActionListener() methods. To handle action events, actionPerformed() method defined by ActionListener interface must be implemented.

JTextField has an action command string associated with it. By default, the action command is the current content of the text field. We can set action command to a fixed value by calling setActionCommand() method which has a general format

void setActionCommand(String cmd)

The string passed in cmd becomes the new action command string. The text in the text field remains unaffected.

To obtain the string that is currently displayed in the text field, getText() method can be called. To set the contents to a specified value setText() method can be used.

**Example:**

JTextField jtf= new JTextField(10); -- defines a text field which is 10 columns wide.

jtf.setActionCommand(“myTF”); - This will make myTF as action command when event is generated.

jtf.setText(“Use text field to input”); - This will set the contents of jtf as Use text field to input

String str= jtf.getText(); - Whatever is stored in text field jtf will be transferred to str which is of type string.

**JCheckBox**

In swing, a check box is a special type of button which belongs to the type JCheckBox and inherits the AbstractButton and ToggleButton classes. Check box defines several constructors. The commonly used one is

JCheckBox(String str)

It creates a check box that has text specified by str as label. When the user selects or deselects a check box, an **ItemEvent** is generated. You can obtain a reference to the **JCheckBox** that generated the event by calling **getItem( )** on the **ItemEvent** passed to the **itemStateChanged( )** method defined by **ItemListener**. This interface specifies only one method, **itemStateChanged()**.

void itemStateChanged(ItemEvent ie)

The item event received is ie. To obtain reference to the item that is changed, getItem() can be used on the ItemEvent object. This has the following syntax:

Object getItem() 🡪 the reference returned must cast to the component class being handled. In this case JCheckBox.

The getText() and setText() methods can be used to obtain and set the contents of the check box.

The easiest way to determine the state of a check box is by calleing isSelected() method. It will return true, if the check box is selected else it will return false. It has the following syntax:

boolean isSelected()

Example:

JCheckBox jcb = new JCheckBox(“Alpha”); this creates a check box named jcb with label Alpha

jcb.addItemListener(this); this will set action to the check box jcb. When this check box is checked, itemEvent is generated whci can be caught as follows;

public void itemStateChanged(ItemEvent ie)

{JCheckBox cb = (JCheckBox) ie.getItem();

if(cb.isSelected())

jtf.setText(cb.getText() + “Is selected”);

else

jtf.setText( “ check box is not selected”); }

**JList**

In Swing, the basic list class is called **JList**. It supports the selection of one or more items from a list. Although the list often consists of strings, it is possible to create a list of just about any object that can be displayed inside a **JScrollPane**. This way, long lists will automatically be scrollable, which simplifies GUI design. It also makes it easy to change the number of entries in a list without having to change the size of the **JList** component.

A **JList** generates a **ListSelectionEvent** when the user makes or changes a selection. This event is also generated when the user deselects an item. It is handled by implementing **ListSelectionListener**. This listener specifies only one method, called **valueChanged( )**, which is shown here:

void valueChanged(ListSelectionEvent *le*)

Here, *le* is a reference to the object that generated the event. Although **ListSelectionEvent** does provide some methods of its own, normally we will interrogate the **JList** object itselfto determine what has occurred. Both **ListSelectionEvent** and **ListSelectionListener** arepackaged in **javax.swing.event**.

By default, a **JList** allows the user to select multiple ranges of items within the list, but we can change this behavior by calling **setSelectionMode( )**, which is defined by **JList**. It is shown here:

void setSelectionMode(int *mode*)

Here, *mode* specifies the selection mode. It must be one of these values defined by **ListSelectionModel** interface of the package javax.swing.

SINGLE\_SELECTION, SINGLE\_INTERVAL\_SELECTION, MULTIPLE\_INTERVAL\_SELECTION

The default, multiple-interval selection, lets the user select multiple ranges of items within a list. With single-interval selection, the user can select one range of items. With single selection, the user can select only a single item. Of course, a single item can be selected in the other two modes, too. It’s just that they also allow a range to be selected.

The index of the first item selected can be obtained, which will also be the index of the only selected item when using single-selection mode, by calling **getSelectedIndex( )**, shown here:

int getSelectedIndex( )

Indexing begins at zero. So, if the first item is selected, this method will return 0. If no item is selected, –1 is returned

**Use of Anonymous Inner Classes to Handle Events**

Till now, event handling was straight forward, in which the main class of application will implement the listener interface itself and all events are sent to an instance of that class. We can also implement listeners through the use of anonymous inner classes. Anonymous inner classes are inner classes that don’t have a name. Instead, an instance of the class is generated on the fly as needed. Anonymous inner classes make implementing some types of event handlers much easier. For example, when the JBtton named jbtn is clicked, the action listener can be implemented as follows:

Jbtn.addActionListenerr(new ActionerListener() {

public void actionPerformed(ActionEvent ae)

{ // code to handle action event;} } );

Here anonymous inner class is created that implements the ActionListener interface. One advantage of using anonymous inner class is that the component that invokes the method is already known. In the example above there is no need to call getActionCommand() to determine which component generated the event, because this implementation of ActionPerformed() will only be called by events generated by jbtn.

**Create a Swing Applet**

The second type of applications that commonly use Swing are Swing applets. They are similar to AWT-based applets. They extend JApplet class. Swing applets use the four life cycle methods init(), start() stop() and destroy(). JApplet will not override the paint() method. All interactions with components in a Swing applet take place on the event-dispatching thread.

// A simple Swing-based applet

import javax.swing.\*;

import java.awt.\*;

import java.awt.event.\*;

/\*

This HTML can be used to launch the applet:

<object code="MySwingApplet" width=220 height=90>

</object>

\*/

public class MySwingApplet extends JApplet {

JButton jbtnAlpha;

JButton jbtnBeta;

JLabel jlab;

// Initialize the applet.

public void init() {

try {

SwingUtilities.invokeAndWait(new Runnable () {

public void run() {

makeGUI(); // initialize the GUI

}

});

} catch(Exception exc) {

System.out.println("Can't create because of "+ exc);

}

}

// This applet does not need to override start(), stop(),

// or destroy().

// Set up and initialize the GUI.

private void makeGUI() {

// Set the applet to use flow layout.

setLayout(new FlowLayout());

// Make two buttons.

jbtnAlpha = new JButton("Alpha");

jbtnBeta = new JButton("Beta");

// Add action listener for Alpha.

jbtnAlpha.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent le) {

jlab.setText("Alpha was pressed.");

}

});

// Add action listener for Beta.

jbtnBeta.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent le) {

jlab.setText("Beta was pressed.");

}

});

// Add the buttons to the content pane.

add(jbtnAlpha);

add(jbtnBeta);

// Create a text-based label.

jlab = new JLabel("Press a button.");

// Add the label to the content pane.

add(jlab);

}

}

**\*\*\*\*\*\*\*\*\*\***