**JAVA HISTORY:**

Java is an object-oriented programming language developed by Sun Microsystems in 1991. James Gosling, one of the inventors of this language. Originally Java was called “**Oak**”. Java was designed for the development of software for consumer electronic devices like TVs, VCRs and other electronic machines. Patrick Naughton, member of Java team discovered that the existing languages like C and C++ had limitations in terms of both reliability and portability. Java developed based on C and C++, but removed a number of features of C and C++. The goal of the Java development team is to make the Java language simple, portable and highly reliable. Java is the first programming language that is not tied to any particular hardware or operating system. Programs developed in Java can be executed anywhere on any system.

**Java Milestones**

|  |  |
| --- | --- |
| **Year** | **Development** |
| 1990 | Sun Microsystems decided to develop special software. James Gosling led the team to undertake the task. |
| 1991 | The team announced a new language called “Oak”. |
| 1992 | The team, known as Green Project team by Sun, demonstrated the application of their new language. |
| 1994 | A Web browser called “HotJava” was developed. |
| 1995 | Oak was renamed “JAVA”. |
| 1996 | Sun releases Java Development Kit 1.0. |
| 1997 | Sun releases Java Development Kit 1.1. |
| 1998 | Sun releases the Java 2 with the version 1.2. |
| 1999 | Sun releases Java 2 platform, Standard Edition (J2SE) and Enterprise Edition(J2EE) |
| 2000 | J2SE with SDK 1.3 was released |
| 2002 | J2SE with SDK 1.4 was released |
| 2004 | J2SE with JDK 5.0 was released |

**JAVA FEATURES:**

**Compiled and Interpreted:**

Java combines both these approaches thus making Java a two-stage system. First, Java compiler translates source code into what is known as bytecode instructions. Bytecode are not machine instructions and therefore, in the second stage, Java Interpreter generates machine code that can be directly executed by the machine that is running the Java program. We can thus say that Java is both a compiled and interpreted language.

**Object Oriented:**

In Java everything is an Object which has some data and behaviour. Java can be easily extended as it is based on Object Model.

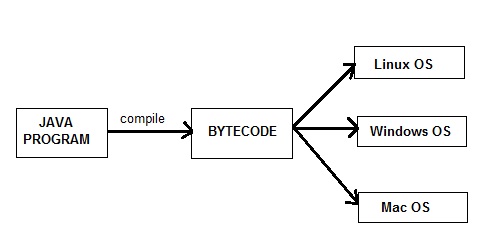
**Robust:**

Java makes an effort to eliminate error prone codes by emphasizing mainly on compile time error checking and runtime checking. But the main areas which Java improved were Memory Management and mishandled Exceptions by introducing automatic **Garbage Collector** and **Exception Handling**.

**Platform-Independent:**

Unlike other programming languages such as C, C++ etc which are compiled into platform specific machines. Java is guaranteed to be write-once, run-anywhere language.

On compilation Java program is compiled into bytecode. This bytecode is platform independent and can be run on any machine, plus this bytecode format also provide security. Any machine with Java Runtime Environment can run Java Programs.



**Secure:**

When it comes to security, Java is always the first choice. With java secure features it enable us to develop virus free, temper free system. Java program always runs in Java runtime environment with almost null interaction with system OS, hence it is more secure.

**Multi Threading:**

Java multithreading feature makes it possible to write program that can do many tasks simultaneously. Benefit of multithreading is that it utilizes same memory and other resources to execute multiple threads at the same time, like While typing, grammatical errors are checked along.

# Architectural Neutral:

Java compiler generates an architecture-neutral object file format, which makes the compiled code to be executable on many processors, with the presence of Java runtime system.

**Portable:**

# Java Bytecode can be carried to any platform. No implementation dependent features. Everything related to storage is predefined, example: size of primitive data types

**High Performance:**

Java enables high performance with the use of just-in-time compiler.

**Distributed**:

Java is designed for the distributed environment of the internet.

**JAVA PROGRAM STRUCTURE:**

**Documentation Section:** The documentation section consist a set of comment lines giving the name of the program, the author and other details.

**Package Statement:** The first statement allowed in a Java file is a Package statement. This statement declares a Package name and informs the compiler that the classes defined here belong to this package. Example:

*package sbvr;*

The package statement is optional.

**Import Statements:** The next thing after a package statement may be a number of import statements. This is similar to the #include statement in C. Example

*import student.test;*

**Interface Statements:** An interface is like a class but includes a group of methods declaration. This is also an optional section and is used only when we wish to implement the multiple inheritance features in the program.

**Class Definition:** A java program may contain multiple class definitions. Classes are the primary and essential elements of a java program.

**Main Method Class:** This class is the essential part of a java program. Main method is the starting point of a program.

**SIMPLE JAVA PROGRAM**

class begin

{

public static void main(String args[])

{

System.out.println(" JAVA PROGRAMMING");

}

}

**class** : class keyword is used to declare classes in Java

**Opening Brace:** Every class definition in Java begins with an opening brace “{“ and ends with a matching closing brace “}”.

**public** : It is an access specifier. Public means this function is visible to all.

**static** : static is again a keyword used to make a function static. To execute a static function you do not have to create an Object of the class. The **main()** method here is called by JVM, without creating any object for class.

**void** : It is the return type, meaning this function will not return anything.

**main** : main() method is the most important method in a Java program. This is the method which is executed, hence all the logic must be inside the main() method. If a java class is not having a main() method, it causes compilation error.

**System.out.println** : This is used to print anything on the console like *printf* in C language.

**JAVA VIRTUAL MACHINE:**

Java virtual Machine(JVM) is a virtual Machine that provides runtime environment to execute java byte code. The JVM doesn't understand Java typo, that's why you compile your \*.java files to obtain \*.class files that contain the bytecode understandable by the JVM.

JVM control execution of every Java program. It enables features such as automated exception handling, Garbage-collected heap.

Virtual Machine

Java Compiler

Java program

**Source code Byte code**

Java Interpreter

Bytecode

Machine Code

**Virtual machine Real Machine**

**JAVA TOKENS:**

Smallest individual units in a program are known as ***tokens***. Java language includes five types of tokens. They are

* Reserved keywords
* Identifiers
* Literals
* Operators
* Separators

**Keywords:**

Keywords are an essential part of language definition. They implement specific features of the language. Java language has reserved 50 words as keywords. We cannot use these keywords as identifiers in the program.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 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 | stircfp | super |
| switch | synchronized | this | throw | throws | transient | try | void |
| volatile | while |  |  |  |  |  |  |

**Identifiers:**

Identifiers are programmer-designed tokens. They are used for naming classes, methods, variables, objects, labels, packages and interfaces in a program. Java identifiers follow the following rules: They can have alphabets, digits, and the underscore and dollar sign characters.

1. They must not begin with a digit.
2. Uppercase and lowercase letters are distinct.
3. They can be of any length.

**Literals:**

Literal is also called as Constant. (Or) Literals in Java are a sequence of characters that represent constant values to be stored in variables. Types of literals as below:

1. Integer literals
2. Floating\_point literals
3. Character literals
4. String literals
5. Boolean literals

**Operators:** An operator is a symbol that takes one or more arguments and operates on them to produce a result.

**Separators:**

Separators are symbols used to indicate where group of code are divided and arranged. They basically define the shape and function of our code.

**IMPLEMENTING A JAVA PROGRAM:**

Implementation of a Java application program involves a series of steps. They include

* Creating the program
* Compiling the program
* Running the program

**Creating the program:**

We can create a program using any text editor. Save the program, ensuring that the filename contains the class name. Save the file with extension **.java**. This file is called source file.

**Compiling the program:**

To compile the program, we must run the Java compiler **javac,** with the source file on the command line as below

**javac filename.java**

If everything is OK, a class file is generated containing bytecode.

**Running the program:**

We need to use the Java interpreter to run a stand alone program. At the command prompt type

**java filename**

**JAVA DEVELOPERS FOLLOWED SOME NAMING CONVENTIONS:**

* Name of all public methods and instance variables start with a leading lowercase letter.

Examples: **average, sum**

* When more than one word is used in a name, the second and subsequent words are marked with a leading uppercase letters.

Examples: **dayTemperature, firstDayOfMonth**

* All private and local variables use only lowercase letters combined with underscore.

Examples: **length, batch\_strength**

* All classes and interfaces start with a leading uppercase letter.

Examples: **Student, Vehicle**

* Variables that represent constant values use all uppercase letters and underscores between words.

Examples: TOTAL, F\_MAX

* Packages should use lower case letters. If the name contains multiple words, it should be separated by dots(.) such as java.util, java.lang.

**LITERALS**

Any constant value which can be assigned to the variable is called iteral/constant.

**Integer Literal:**

Integer literals are also called as integer constant. An integer constant is a sequence of numerical digits. There are three types of integer literals:

1. Decimal integers
2. Octal integers
3. Hexadecimal integers

**Decimal integers** consist of set of digits, 0-9. Decimal integer may be either positive or negative.

Ex: 3445, 123, -123, -56, 0

An **Octal Integer** consists of any combination of digits from the set 0 to 7 with a leading 0. Octal values have sign

Ex: 034, 037, 0551

**Hexadecimal Integer** consists a set of digits 0 to 9 and alphabets A to F to represent the values of 10 to 15. Each Hexadecimal value begin with 0x.

Ex: 0x20xF5, 0x2, 0x9F

**Floating Point literal or Real literal:**

The floating point literals are also called as Real Constants. The numbers containing fractional parts are called as Real Constants. To represent fixed floating point values, we use real constants. These numbers are represented with a decimal value containing a decimal point. A real constant may be either positive or negative.

Ex: 3.1428, 435.38, -1.346

A real number may also be expressed in exponential notation.

Ex: 2.1565e2

e2 means multiply the number by 102

then the above value becomes 215.65

**Character** literal**:**

Any single character or symbol or digits enclosed within a pair of single quote marks are called Character Literal. Character Literals are also called Character Constants.

Ex: ‘B’, ‘6’, ‘@’ etc

**String** literal**:**

String Literals are also called as String Constants. A string constant is a sequence of characters enclosed between double quotes. The characters may be alphabets, digits, special characters and blank spaces.

Ex: “ Hai student”, “1985”,”good”

**Boolean** literal**:**

Boolean Literals represent only two values – true and false. It means we can store either true or false into a Boolean type variable.

**VARIABLES:**

A variable is an identifier that denotes a storage location used to store a data value. A variable may take different values at different times during the execution of the program.

Syntax to declare a variable:

**<Data Type> <Name of the Variable>**

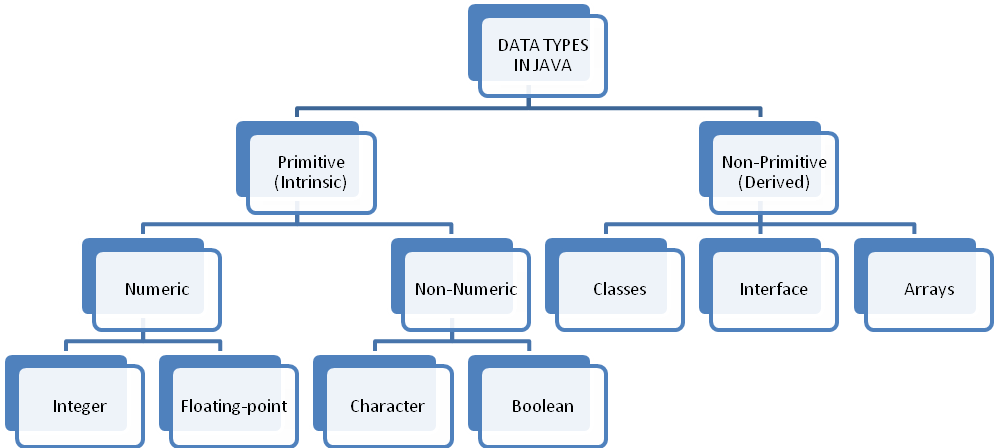
Some examples of variables names are:

int average;

float height;

char gender

**DATA TYPES:**

Every variable in java has a data type. Data types specify the size and type of values that can be stored. Primitive types also called intrinsic or built-in types. Non-Primitive also known as derived types or reference types.

**Integer Data types:**

Integer types can hold whole numbers such as 123, -96. These numbers should be either positive or negative values. Java supports 4 types of integers as shown below

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Size** | **Minimum value** | **Maximum value** |
| Byte | 1 byte | -128 | 127 |
| Short | 2 bytes | -32, 768 | 32, 767 |
| Int | 4 bytes | -2, 147, 483, 648 | 2, 147, 483, 647 |
| Long | 8 bytes | -9, 223, 372, 036, 854, 775, 808 | 9, 223, 372, 036, 854, 775, 807 |

**Floating Point Types:**

Floating point types can hold real numbers that is numbers containing fractional parts. Java supports two types of floating point declarations such as float and double.

*‘Float* ‘data type is used to represent single - precision numbers.

‘Double’ type represents double – precision.

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Size** | **Minimum value** | **Maximum value** |
| Float | 4 bytes | 3.4e-038 | 1.7e+0.38 |
| Double | 8 bytes | 3.4e-038 | 1.7e+0.38 |

**Character type:** Character type represents single character values. The data type char declares a variable of type char, it holds character constant. It occupies 2 bytes of memory. Range is from 0 to 65535.

**Boolean Type:**

This type is used to represent true or false values. If we declare a variable as Boolean type, it holds either true or false value. In java every conditional expression returns a Boolean value.

**OPERATORS**

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. An operator operates on operands.

Java operator can be classified into several categories as below:

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise operators
8. Special operators
9. **ARITHMETIC OPERATORS:** Arithmetic operators are used to construct mathematical expressions as in algebra. Arithmetic operations like addition, subtraction, multiplication, division are performed. List is shown below

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| + | Addition or unary plus |
| - | Subtraction or unary minus |
| \* | Multiplication |
| / | Division |
| % | Modulo division (Remainder) |

Arithmetic operators are used as shown below:

a + b, a – b, a \* b, a/b, a%b

**Integer Arithmetic:** when both the operands in single arithmetic expression such as a + b are integers, the expression is called an integer arithmetic, and the operation is called integer arithmetic.

**Example:**

a=14 and b =4

a-b=10

a+b=18

a\*b=56

a/b=3(decimal part truncated)

a%b=2 (remainder of integer division)

**Real Arithmetic:** An arithmetic operation involving only real operands is called real arithmetic. A real operand may assume values either in decimal or exponential notation. Since floating point values are rounded to the number of significant digits permissible, the final value is an approximation of the correct result.

**Output of the program is :**

a =20.5

b =6.4

a+b=26.9

a-b=14.1

a\*b=131.2

a/b=3.20313

a%b=1.3

**Example:**

class FloatPoint

{

public static void main(String args[])

{

float a=20.5F, b=6.4F;

System.out.println(“ a= “ +a);

System.out.println(“b= “ + b);

System.out.println(“a+b=”+ (a+b));

System.out.println(“a-b=”+ (a-b));

System.out.println(“a\*b=”+ (a\*b));

System.out.println(“a/b=”+ (a/b));

System.out.println(“a%b=”+ (a%b));

}

}

**Mixed-mode Arithmetic:** When one of the operand is real and other is integer, the expression is called a mixed-mode arithmetic expression. If either operand is of the type real, then the other operand is converted to real and the real arithmetic is performed. The result will be real. Thus

15/10.0 produces result 1.5

Whereas

15/10 produces result 1

1. **RELATIONAL OPERATORS:**

Relational operators are used for comparison of operations. When we need to compare two quantities with one another. The expressions which contain comparisons with relational operators are called relational expressions.

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| **<** | Less than |
| <= | Less than or equal to |
| > | Greater than |
| >= | Greater than or equal to |
| == | Equal to |
| != | Not equal to |

Java supports 6 Relational operators. These operators and their meanings are shown above.

All the relational operators in java return a value of type Boolean as result. So they return either true or false. All these relational operators compare the left operand with right operand and return the result.

**Example:**

class RelationalOperators

{

**Output:**

a < b is: true

a > b is : false

a== c is true

public static void main(String args[])

{

float a=15.0F, b= 20.75F, c= 15.0F;

System.out.println( “a < b is: ” +(a<b));

System.out.println(“ a > b is : “+(a>b));

System.out.println(“ a== c is “+(a==c));

}

1. **LOGICAL OPERATORS:**

Java has three logical operators, which are shown below

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| && | Logical AND |
| || | Logical OR |
| ! | Logical NOT |

The logical operators && and || are used when we want to form compound conditions by combining two or more relations. An example is:

a > b && x == 10

An expression of this kind which combines two or more relational expressions is termed as a logical expression or a compound relational expression.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Truth table for AND** | | |  | **Truth table for OR** | | |  | **Truth table for NOT** | |
| P | Q | P&&Q |  | P | Q | P||Q |  | P | Not P |
| T | T | T |  | T | T | T |  | T | F |
| T | F | F |  | T | F | T |  | F | T |
| F | T | F |  | F | T | T |  |
| F | F | F |  | F | F | F |  |

1. **ASSINGNMENT OPERATORS:**

Assignment operators are used to assign the value of an expression to a variable or constant. The operator = is used as assignment operator in java.

Operand 1 =Operand2

**Ex:** a=20, b=40, c= a+b

1. **INCREMENT AND DECREMENT OPERATORS:**

Java supports two operators increment and decrement operators which are used to increase or decrease the value of a variable by one.

Operators that operate on one operand is called **Unary Operators**.

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| ++ | Increment operator |
| -- | Decrement operator |

These increment and decrement operators are used in two different forms

* Pre-increment and post-increment
* Pre-decrement and post-decrement

Syntax for pre-increment/pre-decrement:

++<operand>, --<operand>

In this form, the expression is evaluated first prior to all other actions.

**Ex:** a=6 a=6

s=++a -3 s=--a -3

1. **CONDITIONAL OPERATROR:**

Conditional operator is also called as ternary operator. The expression is of the form

exp1 ? exp2 : exp3

exp1, exp2, exp3 are expressions

If the expression exp1 is true, exp2 is evaluated, if the expression exp1 is false, exp3 is evaluated.

**Example:**

class TernaryOperator

{

int a=10,b=30,max;

max=(a>b)?a:b;

System.out.println(max);

}

1. **BITWISE OPERATORS:**

Bitwise operators are used for manipulation of data values at bit level. These operators act upon individual bits of a number. These operators are used for testing the bits, or shifting them either to right or left side.

Bitwise operators compare two variables bit by bit and return a variable whose bits have been modified. We cannot use these operators on the data of type float.

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| & | Bitwise AND |
| | | Bitwise OR |
| ^ | Bitwise XOR |
| ~ | One’s complement |
| << | Left Shift |
| >> | Right Shift |
| >>> | Zero Fill Right Shift |

**Bitwise AND Operator (&):-** This operator performs AND operations on the individual bits of the numbers. The symbol for this operator is &, which is called ampersand. To understand the AND operation, see the truth table.

|  |  |  |
| --- | --- | --- |
| X | Y | X & Y |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Truth table gives relationship between the input and the output. From the truth table, we can conclude that multiplying the input bits, we can get the output bit.

**Example:**

X=10 0000 1010

Y=11 0000 1011

X&Y 0000 1010

**Bitwise OR Operator(|):-** This operator performs OR operation on the bits of the numbers. The symbol is “|”, which is called pipe symbol. To understand this operation, see the truth table given below. From the table, we can conclude that by adding the input bits, we can get the output bit.

|  |  |  |
| --- | --- | --- |
| X | Y | X|Y |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Example:

X=10 0000 1010

Y=11 0000 1011

X|Y 0000 1011

**Bitwise XOR(^):-** This operator performs exclusive OR (XOR) operation on the bits of the numbers. The symbol is “^”, which is called cap, carat symbol. To understand the XOR operation, see the truth table. From the truth table, we can conclude that when we have odd number of 1’s in the input bits, we can get the output bit as 1.

|  |  |  |
| --- | --- | --- |
| X | Y | X^Y |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Example:

X=10 0000 1010

Y=11 0000 1011

X^Y 0000 0001

1. **SPECIAL OPERATORS:**

Java supports some special operators, instanceof operator and dot operator.

**Instanceof Operator:** The instanceof operator is an object reference operator and returns true if the object on the left hand side is an instance of the class given on the right hand side. This operator allows us to determine whether the object belongs to particular class or not. **Example:**

Person instanceof student

Is true if the object person belongs to the class student, otherwise it is false.

**Dot Operator:** The dot operator (.) is used to access the instance variables and methods of class objects.

**Examples:**

Person.age //reference to the variable age

Person1.salaray() //reference to the method salary()

**ARITHMETIC EXPRESSION:**

It is a combination of variables, constants, and operators arranged as per the syntax of the language. Expressions are evaluated using an assignment statement of the form

*Variable=expression*

An arithmetic expression without any parentheses will be evaluated from left to right using the rules of precedence of operators. Two levels of priority

High priority \* / %

Low priority + -

**Example:**

class expression

{

public static void main(String args[])

{

int a, b,c;

a=23;

b=4;

c=22;

c=a+b+c;

a=a\*b/c;

b=b\*c+(a-b);

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

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

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

}

}

**DECISION MAKING AND BRANCHING**

A Java program is a set of statements, which are normally executed sequentially in the order in which they appear. This happens when options or repetitions of certain calculations are not necessary. We have a number of situations, where we may have to change the order of execution of statements based on certain conditions, or repeat of group of statements until certain specified conditions are met.

When a program breaks the sequential flow and jumps to another part of the code, it is called branching. When the branching is based on particular condition, it is known as conditional branching. If branching takes place without any decision, it is known as unconditional branching.

**Control or Decision making statements:**

1. If statement
2. If..else statement
3. Nesting of if.. else
4. Else if ladder
5. Switch statement
6. Conditional operator

**Simple IF:** The general form of the simple if statement is

The ‘statement - block’ may be a single statement or a group of 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:**  import java.util.Scanner;  class Example\_If  {  public static void main(String args[])  {  int x;  Scanner sc=new Scanner(System.in);  System.out.print(" Enter the value of x less than 10 ");  x=sc.nextInt();  if(x<10)  {  System.out.print(" Entered value less than 10 ");  }  System.out.println("ok");  } } |

**If-else statement:** The general form is

If the test expression is true, then the true-block statement(s) immediately following the if statement are execute. Otherwise, the false-block statement(s) are executed. In either case, either true-block or false-block will be executed, not both.

|  |  |
| --- | --- |
|  | **Example:**  import java.util.Scanner;  class Example\_If\_Else  {  public static void main(String args[])  {  int x;  Scanner sc=new Scanner(System.in);  System.out.print(" Enter the value of x less than 10 ");  x=sc.nextInt();  if(x<10)  {  System.out.print(" Entered value less than 10 ");  }  else  {  System.out.println(" Entered value greater than 10 ");  } } } |

**Nesting of if....else statement:** The general form of the nested if-else statement is

If the test condition1 is false statement -3 will be executed, otherwise it continues to perform the second test. If the condition – 2 true, the statement -1 will be evaluated, otherwise the statement -2 will be evaluated and the control is transferred to the statement – x.

|  |  |
| --- | --- |
|  | **Example:**  class IfElseNesting  {  public static void main(String args[])  {  int a=325, b=712, c=478;  System.out.println(" largest value is : ");  if(a>b)  {  if(a>c) {  System.out.println(a);  }  else {  System.out.println(c);  }  }  else {  if(c>b)  {  System.out.println(c);  }  else  {  System.out.println(b);  } } } } |

**Else if ladder:** The general form is

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

|  |  |
| --- | --- |
|  | **Example:**  import java.util.Scanner;  class Else\_If\_Ladder  {  public static void main(String args[])  {  int mark;  Scanner sc=new Scanner(System.in);  System.out.println(" Enter the mark ");  mark=sc.nextInt();  if(mark >= 75)  {  System.out.println(" Distinction ");  }  else if(mark >=60 && mark <75)  {  System.out.println(" First Class ");  }  else if (mark >=50 && mark <60)  {  System.out.println(" Second Class ");  }  else if(mark >=35 && mark <50)  {  System.out.println( " Third class");  }  else  {  System.out.println(" Fail ");  } } } |

**Switch Statement:**

The switch statement tests the value of a given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed.

The expression is an integer expression or characters. Value-1, value-2... are constants or constant expressions and are known as case labels.

Block-1, block-2 ... are statement lists and may contain zero or more statements. It is important to note that case labels end with a colon (:).

 The break statement at the end of each block signal the end of a particular case and causes an exit from the switch statement. The default is an optional case.

**Flowchart**

The general form is

**Example:**

**Java program to print week day using the switch statement.**

**import** java.util.Scanner;

**class** switchExample {

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

**int** dayno;

Scanner sc=**new** Scanner(System.*in*);

System.*out*.println(" Enter day number in a week \n");

dayno=sc.nextInt();

**switch**(dayno)

{

**case** 1: System.*out*.println( " sunday \n");

**break**;

**case** 2: System.*out*.println( " monday \n");

**break**;

**case** 3: System.*out*.println( " tuesday \n");

**break**;

**case** 4: System.*out*.println( " wednesday \n");

**break**;

Output:

Enter day number in a week

5

thrusday

**case** 5: System.*out*.println( " thrusday \n");

**break**;

**case** 6: System.*out*.println( " friday \n");

**break**;

**case** 7: System.*out*.println( " saturday \n");

**break**;

**default**:

System.*out*.println(" invalud day number");

}

}

}

**DECISION MAKING AND LOOPING:**

The process of repeatedly executing a block of statements is known as Looping.

The programme uses a loop construct to instruct the computer to perform repetitive tasks for a finite number of times based on conditions. Each loop construct contains two parts namely condition part and body part. The body part of the loop will be executed repeatedly as long as the given condition evaluates to true.

**Looping operations:**

1. while Loop
2. do ..while Loop
3. for Loop



**The while statement:** The general form is

The while is an entry-controlled loop statement. The simplest of all the looping structure in java is the while statement. The test condition is evaluated and if the condition is true, then the body of the loop is executed. After the execution of the body, the test condition is once again evaluated and if it is true, the body is executed once again. This process is repeated until the test condition becomes false and the control is transferred out of the loop.

**Example:**

class While\_Example

{

public static void main(String args[])

{

int i=1;

while(i<=10)

{

System.out.println( i);

i++;

}

}

}

****

**The do..while Statement:** The general form is

do..while loop is an exit-controlled loop statement as it allows the body of the loop to be executed for the first time without any condition.

At the end of the loop, the test condition in the while statement is evaluated. If the condition is true, the program continues to evaluate the body of the loop once again. This process continues as long as the condition is true. When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the while statement.

**Example:**

class Do\_While

{

public static void main(String args[])

{

int i=1;

do

{

System.out.println(i);

i++;

}

while(i<=10);

}

}

**The for statement:** The general form is



It is a entry-controlled loop.

Execution of the **for** statement is as follows:

1. Initialization of the control variable is done first.
2. The value of the control variable is tested using the test condition. The test condition is a relational expression. When the loop is true, the body of the loop is executed. Otherwise the loop is terminated.
3. When the body of the loop is executed, the control is transferred back to the **for** statement after evaluating the last statement in the loop. Now the control variable is incremented or decremented. The new value of the control variable is tested again to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test condition.

**Example:**

class for\_Example

{

public static void main(String args[])

{

int i;

for(i=1;i<=10;i++)

{

System.out.println( i);

}

}

}

**Difference b/w while and do-while:**

|  |  |
| --- | --- |
| **While** | **Do -while** |
| It is an entry-control loop | it is an exit-controlled loop |
| Test condition is tested first | Test condition is tested after executing the body of the loop |
| It is pre-testing loop | It is post-testing loop |
| It will not give guarantee to execute the body of the loop minimum once | It will give the guarantee to execute the body of the loop minimum once |
| Syntax: while (condition)  Simple or compound statement; | Syntax: do  Simple (or) compound statement;  while(condition); |

**Break Statement:**

The break statement is used to terminate the loop. When the **break** statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the break would only exit from the loop containing it. That is, the break will exit only a single loop.

**Syntax:** **break;**

Terminating the loop with break statement

**Example:**

class Break

{

public static void main(String args[])

{

int i;

for(i=1;i<10;i++)

{

if(i==7)

break;

System.out.print(i+" ");

}

}

}

**Continue statement:**

Like break statement, java supports another similar statement called the continue statement. However, unlike the break which caused the loop to be terminated, the continue, as the name implies, causes the loop to be continued with the next iteration after skipping any statements in between. The continue statement tells the compiler. “SKIP THE FOLLOWING STATEMENTS AND CONTIUNE WITH THE NEXT ITERATION”. The format of the continue statement is simply

**Continue;**

**Example:**

import java.util.Scanner;

class positive

{

public static void main(String args[])

**output:**

enter any number 5

enter any number -7

enter any number 6

sum is 11

{

int i,n,sum=0;

Scanner sc=new Scanner(System.in);

for(i=1;i<=3;i++)

{

System.out.print("\n enter any number");

n=sc.nextInt();

if(n<0)

continue;

else

sum=sum+n;

}

System.out.print("\n sum is "+sum);

}

}

**Difference b/w break and continue:**

|  |  |
| --- | --- |
| **Break** | **Continue** |
| Takes the control to the outside the loop | Takes the control to the beginning of the loop |
| Used in switch statement | Used only in loop statements |
| Associated with if condition in loops | Associated with if condition |
| Terminates the loop when break is encountered | Doesn’t terminates the loop when continue is encountered. |

**INPUT AND OUTPUT**

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

A stream is a sequence of data. In Java, a stream is composed of bytes. In Java, 3 streams are created for us automatically. All these streams are attached with the console.

**1) System.out:**standard output stream

This is the **standard output stream** that is used to produce the result of a program on an output device like the computer screen.

List of various print functions that we use to output statements.

**print():**This method prints the text on the console and the cursor remains at the end of the text at the console.

**Syntax:**

System.out.print(*parameter*);

**println():** It prints the text on the console and the cursor moves to the start of the next line at the console.

**Syntax:**

System.out.println(*parameter*);

**printf():**This is the easiest of all methods as this is similar to printf in C. Note that System.out.print() and System.out.println() take a single argument, but printf() may take multiple arguments.

**2) System.in:**standard input stream:

This is the **standard input stream** that is used to read characters from the keyboard or any other standard input device.

**3) System.err:**standard error stream

This is the **standard error stream** that is used to output all the error data that a program might throw, on a computer screen or any standard output device.

**Reading Input with Java.util.Scanner**

The scanner class is a class in java.util, which allows the user to read values of various types. A scanner object can read user input entered on the console or from a file. The scanner provides methods.

Scanner methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| nextBoolean() | Read and converts next token to a Boolean value. |
| nextInt() | Reads and converts next token to a integer value. |
| nextLong() | Reads and converts next token to a long value. |
| nextDouble() | Reads and converts next token to a double value. |
| nextString()or next() | Reads next token and returns it as a string. |
| nextLine() | Reads until the next new line and returns a String |
| hasNextBoolean() | Returns true if the next token is either ‘true” or “false” |
| hasNextInt() | Returns true if the next token is an integer |
| hasNextLong() | Returns true if the next token is a long |
| hasNextDouble() | Returns true if the next token is a real number |
| hasNextString() or hasNext() | Returns true if there is at least one more token of input |
| hasNextLine() | Returns true if there is another line of input |

Whenever using scanners, be sure to include the proper import line:

import java.util.Scanner;

We create Scanner in two ways:

To read from the console, use the following:

Scanner sc=new Scanner(System.in);

To read from the file, use the following:

Scanner sc=new Scanner(new FileStream(“filename.txt”));

**Example:**

import java.util.Scanner;

import java.io.\*;

public class Scanner\_Example

{

public static void main(String args[])

{

Scanner sc=new Scanner(System.in);

int i;

long li;

float r;

double d;

String str1;

String str2;

System.out.println(" Enter an integer value, long value, float value ");

System.out.println(" Enter double value, string values ");

System.out.println(" Separate each with a blank or return ");

i=sc.nextInt();

li=sc.nextLong();

r=sc.nextFloat();

d=sc.nextDouble();

str1=sc.nextLine();

str2=sc.nextLine();

System.out.println(" what i entered is shown below ");

System.out.println(" Integer value i= "+i +" long value li = "+li +" float value r= "+r );

System.out.println(" double value d= "+d+" string values str1 ="+str1+" string value str2 = "+str2);

} }

To accept data from the keyboard, i.e., System.in, we need to connect keyboard to an input stream object. Here, we can use InputStreamReader that can read data from the keyboard.

**InputStreamReader inr=new InputStreamReade(System.in);**

In the above statement, we are creating InputStreamReader object and connecting the keyboard (System.in) to it.

Next, connect InputStreamReader to BufferedReader, which is another input type of stream. We are using BufferedReader as it has got methods to read data properly, coming from the stream.

**BufferedReader br=new BufferedReader(inr);**

Here, we are creating BufferedReader object (br) and connecting the InputStreamReader object (inr) to it.

Above both statements are combined to produce single statement as below:

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

Now, we can read data coming from the keyboard using read() and readLine() methods available in BufferedReader class.

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

class io1

{

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

{

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

char ch=(char)br.read(); //reads a single character

System.out.println( "Enter the string ");

**import** java.io.\*;

**public** **class** BufferedReaderExample{

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

   InputStreamReader r=**new** InputStreamReader(System.in);

    BufferedReader br=**new** BufferedReader(r);         System.out.println("Enter your name");

   String name=br.readLine();

 System.out.println("Welcome "+name);

}

}

String s=br.readLine();

System.out.println(" Enter the numeric value");

int i=Integer.parseInt(br.readLine());

System.out.println("Enter the real values ");

float n=Float.parseFloat(br.readLine());

double du=Double.parseDouble(br.readLine());

System.out.println(“ The character value is “ +ch);

System.out.println(" The string value is " +s);

System.out.println(" The numereic value is "+i);

System.out.println(" The float value is "+n);

System.out.println(" The double value is "+du);

}

}

**PARTS OF JAVA**

Sun Microsystems Inc. has divided java into 3 parts – Java SE, Java EE, and Java ME.

1. **Java SE:** It is the Java Standard Edition that contains basic core Java classes. This edition is used to develop standard applets and applications.

2. **Java EE:** It is the Java Enterprise Edition and it contains classes that are beyond Java SE. In fact, we need Java SE in order to use many of the classes in Java EE. Java EE mainly concentrates on providing business solutions on a network.

3. **Java ME:** It stands for Java Micro Edition. Java ME is for developers who develop code for portable devices, such as a PDA or cellular phone. Code on these devices needs to be small in size and should take less memory.

**STRING.FORMAT()**

If we want only a string that consists of formatted output, then we can take the help of format() method of String class. The formatted characters supported by System.out.printf() are also usable with format() method. Since, format() method is static method, we can call it as: String.format(). This method returns a string that contains the formatted output which can be processed and used as the programmer wants it.

**Example:**

class FormattedOutput

{

public static void main(String args[])

{

int i=65; String name="Hello";

char ch='A';

String str=String.format(" i=%d%n name=%s%n ch=%c",i,name,ch); System.out.println(str);

}

}

**ARRAYS**

An array is a homogeneous collection of elements of same type that share a common name. The elements of the array are stored in consecutive memory locations and are referred by an index (also known as subscript). In java, arrays are created on dynamic memory, i.e., allocated at runtime by JVM.

**Array types:**

1. One Dimensional Arrays
2. Two dimensional Arrays
3. Multi Dimensional Arrays

**ONE DIMENSIONAL ARRAY:**

When array is declared with only one dimension (subscript) then it is called one dimensional array or single dimensional array.

**Declaration of one-dimensional array:**

**Syntax:**

**Type array\_name[size];**

**Type[size] array\_name;**

1. The type specifies the type of element that will be contained in the array, such as int, float, or char.
2. Size indicates the maximum number of elements that can be stored inside the array.
3. Array\_name is an identifier that specifies the name of the array.

Datatype[ ] array\_name=new datatype[size]

new operator is used to initialize an array.

**Ex**:

**int arr[5];**

0 1 2 3 4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

a[0] a[1] a[2] a[3] a[4]

**Initialization of arrays:**

After an array is declared, its elements must be initialized. Otherwise, they will contain garbage value.

Arrays can be initialized as follows:

**Syntax:**

Datatype[ ] array\_name={ elements separated by commas};

**Ex:**

int[] arr=new int[5];

int[] arr={1,2,3,4,5};

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |

arr[0] arr[1] arr[2] arr[3] arr[4]

**Array Length**

To know the length or size of an array we use a property named **length**.

Example: int x=arr.length;

**Program to display the elements of an array**

class array1

{

public static void main(String args[])

{

int arr[]={1,2,3,4,5,6};

int x=arr.length;

System.out.println(" The array elements are ");

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

System.out.println(arr[i]);

}

}

**Sorting an array**

class sortarray1

{

public static void main(String args[])

{

int arr[]={5,8,62,42,85,1};

int x=arr.length;

int i,j;

System.out.println(" The array elements are ");

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

{

for(j=i+1;j<x;j++)

{

if(arr[i]>arr[j])

{

int temp;

temp=arr[i];

arr[i]=arr[j];

arr[j]=temp;

} } }

System.out.println(" After sorting elements are ");

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

System.out.println(arr[i]);

} }

**TWO-DIMENSIONAL ARRAY**

When an array uses only two subscripts then it is called “Two dimensional array”. It can be viewed as table of elements, which contains rows and columns. A Two dimensional array is useful for matrix operations.

**Example:**

int table[2][3];

In the above example the table contains 2 rows and 3 columns, which mean, the table, can store 6 values.

0 1 2

|  |  |  |
| --- | --- | --- |
| 1  (0,0) | 2  (0,1) | 3  (0,2) |
| 6  (1,0) | 7  (1,1) | 8  (1,2) |

0

1

**Declaration of two-dimensional array:**

The general form is

**Syntax:**

Datatype arrayname[ ] [ ]= new datatype[row\_size ] [column\_size ];

**Example:**

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

**Initialization Of Two-Dimensional Array**

The general form is

**Syntax:**

Datatype arrayname[][]={list of values};

**Example:**

int arr[3][3]={1,2,3,4,5,6,7,8,9};

int arr[][]={{1,2},{3,4},{5,6}};

|  |  |
| --- | --- |
| **2-D program to display matrix**  class two\_dim  {  public static void main(String args[])  {  int arr[][]={{1,2,3},{4,5,6}};  int i,j,m,n;  for(i=0;i<2;i++)  {  for(j=0;j<3;j++)  {  System.out.print(arr[i][j]+ " ");  }  System.out.println();  } } }  **Program to multiply two matrices**  class array\_mul  {  public static void main(String args[])  {  int a[][]={{1,2},{4,5}};  int b[][]={{1,2},{4,5}};  int c[][]=new int[2][2];  int i,j,k;  System.out.println(" The multiplication of two matrices are Shown below ");  System.out.println("---------------------------");  for(i=0;i<2;i++)  {  for(j=0;j<2;j++)  {  c[i][j]=0;  for(k=0;k<2;k++)  {  c[i][j]=c[i][j]+(a[i][k]\*b[k][j]);  }  System.out.print(c[i][j]+"\t");  } System.out.println(); } } } | **Program on Transpose matrix**  class transpose  {  public static void main(String args[])  {  int a[][]= {{1,2,3},{4,5,6},{7,8,9}};  int i,j;  System.out.println (" After Transpose ");  for(i=0;i<3;i++)  {  for(j=0;j<3;j++)  {  System.out.print(a[j][i]+" ");  }  System.out.println(" ");  }  }}  **Program to add two matrices**  class array\_addition  {  public static void main(String args[])  {  int a[][]={{1,2,3,4},{5,6,7,8},{6,9,2,5}};  int b[][]={{2,5,2,6},{6,5,7,2},{8,5,7,9}};  int c[][]=new int[3][4];  int i,j;  System.out.println(" The addition of two matrices are Shown below ");  System.out.println("---------------------------");  for(i=0;i<3;i++)  {  for(j=0;j<4;j++)  {  c[i][j]=a[i][j]+b[i][j];  System.out.print(c[i][j]+ "\t");  }  System.out.println();  } }} |

**3D array:**

Three-dimensional array is the collection

of [two-dimensional arrays](https://code4coding.com/two-dimension-array-in-java-language/) in Java programming language.

**Sum of a diagonal elements in a matrix**

class sum\_diagonal

{

public static void main(String args[])

{

int a[][]={{1,2,3},{5,6,7},{9,10,11};

int i,j,sum=0;

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

{

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

{

if(i==j)

{

sum=sum+a[i][j];

} } }

System.out.print(" sum of diagonal "+sum); }

}

Three-dimensional array is also called the multidimensional

array.

**Syntax for 3D:**

datatype arrayname[ ][ ][ ]=new datatype[size1][size2][size3];

**Initializing a 3d array in Java**

**Method 1**

**Syntax**

array\_name[index\_1][index\_2][index\_3]=value;

**Example**

arr[0][0][0]=45;       //initialize first elements of 3 d array

we can initialize every index, like this

**Method 2**

int[][][] arr{ { {34,67,43}, {576,697,423},{576,697,423} },

{ {39,47,33},{376,987,453},{57,69,42}}

}

**Example:**

class threed

{

public static void main (String args[])

{

int arr[][][]={{{1,2,3},{4,5,6}},{{7,8,9},{2,4,6}}};

for(int i=0;i<2;i++)

{

for( int j=0;j<2;j++)

{

for(int k=0;k<3;k++)

{

System.out.printf("arr[%d][%d][%d]= %d ",i,j,k,arr[i][j][k]);

}

System.out.println();

}

System.out.println();

}}}

**COMMAND LINE ARGUMENTS:**

Command line arguments are parameters that are supplied to the application program at the time of invoking it for execution.

This is explained with a simple program

class comargtest

{

public static void main(String args[])

{

**Output:**

Number of arguments = 5

1: Java is simple!

2: Java is object\_oriented!

3: Java is robust!

4: Java is portable!

5: Java is secure!

int count,i=0;

String str;

count=args.length;

System.out.println("Number of arguments = " +count);

while(i<count)

{

str=args[i];

i=i+1;

System.out.println(i+": "+"Java is " + str + " ! ");

}

}

}

**Compile the program**

javac comargtest.java

**Run the program**

java comargtest simple object\_oriented robust portable secure

Arguments simple, object\_oriented, robust, portable, and secure are passed to the program through the array args. That is the element args[0] contains simple, args[1] contains object\_oriented and so on. Note that Java subscripts begin with 0 and not 1.

**STRINGS**

Strings represent a sequence of characters and these are enclosed with double quotes. In java, strings are class objects and implemented using three classes, namely String, StringBuffer and StringBuilder. String is a class in java.lang (language) package. But in java, all classes are also considered as data types. So we can take String as a datatype.

**The most direct way to create a string is to write:**

String greeting=”Hello World”;

Strings may be declared and created as follows:

**String stringname;**

**stringname=new String(“String”);**

**Example:**

String First\_name;

First\_name=new String(“ cnu “);

These two statements are combined as follows:

String First\_name=new String(“ cnu “);

The String class is immutable, so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters, then you should use String Buffer & String Builder Classes.

**STRING METHODS:**

The string class defines a number of methods that allows us to accomplish a variety of string manipulation tasks.

|  |  |
| --- | --- |
| **METHOD CALL** | **TASK PERFORMED** |
| s2=s1.toLowerCase(); | Converts the string s1 to all lowercase. |
| s2=s1.toUpperCase(); | Converts the string s1 to all uppercase. |
| s2=s1.replace(‘x’,’y’); | Replace all appearances of x with y |
| s2=s1.trim(); | Remove white spaces at the beginning and end of the string s1. |
| s1.equals(s2) | Returns true if s1 is equal to s2 |
| s1.equalsIgnoreCase(s2) | Returns true if s1=s2 ignoring the case of character |
| s1.length() | Gives the length of s1 |
| s1.charAt(n) | Gives nth character of s1 |
| s1.compareTo(s2) | Returns negative if s1<s2, positive if s1>s2, and zero if s1 is equal to s2 |
| s1.compareToIgnoreCase(s2) | Equals compareTo(), ignore the case characters |
| s1.concat(s2) | Concatenates s1 and s2 |
| s1.substring(n) | Gives substring starting from nth  character |
| s1.substring(n,m) | Gives substring starting from nth character up to mth (not including mth) |
| s1.indexof(‘x’) | Gives the position of the first occurrence of ‘x’ in the string s1 |
| s1.lastIndexOf('x') | Gives the last occurrence of 'x' in the string s1. |
| s1.startWith("sub") | Returns true, if a string beginning with the "sub " |
| s1.endsWith("sub ") | Returns true, if a string ending with the "sub" |

**Example:**

**Output:**

F:\java\_programs>javac str\_example.java

F:\java\_programs>java str\_example

The uppercase string ASHWIKA

The lowercase string nitiksha

Concating two stings NITIKSHAreddy

The length of a string 7

comparison of strings 19

string equals false

string equalignorecase false

charat w

substring TIKSHA

class str\_example

{

public static void main(String args[])

{

String s1="ashwika";

String s2="NITIKSHA";

String s3="reddy";

System.out.println(" The uppercase string " +s1.toUpperCase());

System.out.println(" The lowercase string " +s2.toLowerCase());

System.out.println(" Concating two stings " +s2.concat(s3));

System.out.println(" The length of a string " +s1.length());

System.out.println(" comparison of strings "+ s1.compareTo(s2));

System.out.println(" string equals "+ s1.equals(s2));

System.out.println(" string equalignorecase "+ s1.equalsIgnoreCase(s2));

System.out.println(" charat "+ s1.charAt(3));

System.out.println(" substring "+ s2.substring(2));

}

}

**STRING COMPARISON**

Different ways to compare String in Java:

1. **By Using equals() Method:** In java, String equals( ) method compares the two given strings based on the data / content. If all the contents of both the string are same then it returns true. If all characters are not matched then it returns false.

**Example:**

class StringEquals

{

public static void main(String args[])

{

String s1="Hello";

String s2="Hello";

String s3=new String("hello");

System.out.println(s1.equals(s2));

System.out.println(s1.equals(s3));

} }

1. **By using equalsIgnoreCase( ) method:**

The string equalsIgnoreCase( ) method compares two string irrespective of the case of the string. This method returns true if the argument is not null and the contents of both the strings are same ignoring case, else false.

class StringEqualsIgnoreCase

{

public static void main(String args[])

{

String s1="Hello";

String s2="Hello";

String s3=new String("hello");

System.out.println(s1.equalsIgnoreCase(s2));

System.out.println(s1.equalsIgnoreCase(s3));

} }

1. **By Using == Operator:**

The == operator compares references not values.

class DoubleEquals

{

public static void main(String args[])

{

String s1="Hello";

String s2="Hello";

String s3=new String("Hello");

System.out.println(s1==s2);

System.out.println(s1==s3);

} }

1. **By compareTo() Method**:

The java compares two strings based on the Unicode value of each character in the strings. If the compareTo() method returns negative if s1s2, and zero if s1=s2.

**Example:**

class StringCompareTo

{

public static void main(String args[])

{

String s1="Hello";

String s2="Hello";

String s3=new String("hello");

System.out.println(s1.compareTo(s2));

System.out.println(s1.compareTo(s3));

}

}

**IMMUTABILITY OF STRINGS IN JAVA**

Mutable objects are those objects whose contents can be modified. Immutable objects are those objects, once created cannot be modified. String class objects are immutable.

**Example:**

class Test

{

public static void main(String args[])

{

String s1="Data";

String s2="Base";

s1=s1+s2;

System.out.println(s1);

}

}

Here, JVM creates two objects, s1 and s2 separately. When s1+s2 is done, JVM creates a new object and stores the string DataBase in that object. But it does not modify the contents of the string s1. After creating the new object, the reference s1 is adjusted to refer to that new object. Observer that the contents of the string s1 are not modified, so the Strings are called immutable.

**OBJECT-ORIENTED PARADIGM:**

The major objective of object-oriented approach is to eliminate some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allow to flow freely around the system. It ties data more closely to the functions that operate on it and protects from unintentional modification by other functions. OOP allow us to decompose a problem into a number of entities called Objects and then build data and functions (known as methods in Java) around these entities. The combination of data and methods make up an object.

The data of an object accessed by the method associated with that object. However, methods of one object can access the methods of other objects.

Some of the features of object-oriented paradigm are:

1. Emphasis on data rather than procedure.
2. Programs are divided into what are known as Objects.
3. Data structures are designed such that they characterize the objects.
4. Methods that operate on the data of an object are tied together in the data structure.
5. Data is hidden and cannot be accessed by external functions.
6. Objects may communicate with each other through methods.
7. New data and methods can be easily added whenever necessary.
8. Follows bottom-up approach in program design.

**BASIC CONCEPTS OF OOPS**

The basic concepts of OOPs are

1. Objects
2. Classes
3. Data Abstraction
4. Data Encapsulation
5. Inheritance
6. Polymorphism
7. Dynamic Binding
8. Message Passing

**Objects:** Objects are basic run time entities in an object oriented programming. An object may be defined as an identifiable entity with some state and behaviour. We are often surrounded by a variety of objects. The computer we may be working on, the chair we sit on, a telephone, a clock, etc. are all objects.

**Classes:** The most important feature of Object-Oriented Programming is the Classes. A class is the way to bind the data and its associated functions together. It allows the data to be hidden if necessary from the external use. **(OR)**

A Class can be defined as a template/blue print that describes the behaviours/states that object of its type support.

**Data Abstraction:** Abstraction refers to the act of representing essential features without including the background details or explanations.

**Data Encapsulation:** The wrapping of data and functions into a single unit is known as encapsulation. The data is not accessible to the outside world and only those functions which are wrapped in the class can access it.

**Inheritance:** Inheritance is one of the most powerful features of OOP’s. Inheritance is a process of creating a new class from the existing class. The new class inherits all the capabilities of the existing class.

**Polymorphism:** Polymorphism comes from the Greek words “Poly” and “morphism”, “Poly” means many and “Morphism” means form i.e., many forms. In object oriented programming polymorphism refers to identically named methods have different behaviour depending on the type of object.

**SHAPE**

**TRIANGLE**

**RECTANGLE**

**CIRCLE**

**Dynamic Binding:** Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time.

**Message passing:** AMessage for an object is request in the receiving object that generates the desired result.

**BENEFITS OF OOP:**

1. The principle of data hiding helps the programmer to build secure programs.
2. It is easy to partition the work in a project based on objects.
3. Object-Oriented systems can be easily upgraded from small to large systems.
4. Through Inheritance, we can eliminate redundant code and extend the use of existing classes.
5. OOP provides an advantage in production and maintenance of software.
6. Object-Oriented program involves the identification and implementation of different classes of objects and their behaviour.
7. Software complexity can be easily managed.
8. It is possible to have multiple instances of an object to co-exist without any interference.
9. OOP leads to saving of development time and higher productivity.

**APPLICATIONS OF OOP:**

Applications of OOP are beginning to gain importance in many areas. The promising areas for application of OOP include:

1. Artificial intelligence and expert systems
2. Simulation and modelling studies.
3. Object-oriented database systems
4. Object-oriented operating systems
5. Real-time systems.
6. Office automation systems.
7. CAD/CAM systems
8. Multimedia applications
9. Graphical User Interface (GUI)
10. Computer – based training and education systems.

**DIFFERENCE BETWEEN PROCEDURE ORIENTED PROGRAMMING AND OBJECT ORIENTED PROGRMMING**

|  |  |
| --- | --- |
| **POP** | **OOP** |
| A program consists of many procedures. | A program consists of many objects. |
| A procedure is called a function that contains a set of instructions. | An object contains data and functions. |
| POP is called procedure oriented programming language. | OOP is called object oriented programming language. |
| It does not support for combing data and functions. | It supports for combining data and functions into a single unit. This mechanism is called encapsulation. |
| Large program is divided into functions. | Large program is divided into objects. |
| Data can be moved throughout the program. | Data which is hidden cannot be accessed by external functions. |
| Functions communicate with one to another | Objects communicate with one to another through functions |
| All these files must be in a single file. It does not support for separate compilation. | OOPS allows compiling the program with separate files. In separate compilation we need interface, implementation and application files. |
| Dynamic memory allocation is difficult. | Dynamic memory allocation is very easy. |
| It follows top-down programming approach. | It follows bottom-up programming approach. |
| It is less security. | It is more security |
| Data reusability is not possible. | Data reusability is possible. |
| Ex: C | Ex: C++, Java |

**CLASS, OBJECTS AND METHODS**

**DEFINING A CLASS:**

A class is a user-defined data type with a template that serves to define its properties. A class encloses both the data and functions that operate on the data, into a single unit. The variables are called data members and functions are called member functions in a class. The basic form of a class definition is:

***class <class name> extends <Super class> implements <interface name>***

***{***

***[fields declaration]***

***[methods declaration]***

*}*

The class name followed by the keyword ***class*** is the actual class which we want to define.

The keyword ***extends*** is used to extend the features of an existing class to the current class. We can use only one super class to extend at a time.

The keyword ***implements*** is used to implement the interfaces if any. We can use more than one interface to implement in the current class.

**FIELDS DECLARATION:**

Data is encapsulated in a class by placing data fields inside the body of the class definition. These variables are called instance variables because they are created whenever an object of the class is instantiated. We can declare the instance variables exactly the same way as we declare local variables. **Example**

class Rectangle

{

int length;

int width;

}

**METHODS DECLARATION:**

A class with only data fields has no life. The objects created by such a class cannot respond to any messages. We must therefore add methods that are necessary for manipulating the data contained in the class. Methods are declared inside the body of the class but immediately after the declaration of instance variables. The general form of a method declaration is

class Rectangle

{

int length;

int width;

void getdata (int x,int y)

{

length=x;

width=y;

}

int rectarea()

{

int area=length\*width;

return (area);

}

}

**type methodname [parameter list]**

**{**

**Method-body;**

**}**

Method declarations have four basic parts:

1. The name of the method (method name)
2. The type of the value the method returns (type)
3. A list of parameters (parameter-list)
4. The body of the method

**Examples:**

class Rectangle

{

int length;

int width;

void getdata (int x,int y)

{

length=x;

width=y;

}

}

**CREATING OBJECT:**

An instance of a class is called object. (Or)

An object in java is essentially a block of memory that contains space to store all the instance variables.

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. **Example**

**Rectangle rect1; //declare the object**

**rect1=new Rectangle(); //instantiate the object**

Both statements are combined into one as shown below

**Rectangle rect1=new Rectanlge();**

**ACCESSING CLASS MEMBERS:**

We can access the class members i.e. instance variables and methods in the following way

**Objectname.variablename=value;**

**Objectname.methodname(parameterlist);**

From the above statements, objectname is the name of the object, variablename is the name of the instance variable. Methodname is the method that we wish to call, and parameter-list is a comma separated list of actual values.

**Example:**

class Rectangle

{

int length; //DECLARATION OF VARIABLES

int width;

void getData(int x,int y) // DEFINITION OF METHOD

{

length=x;

width=y;

}

int rectarea()

{

int area=length\*width;

return(area);

}

}

class rectarea // CLASS WITH MAIN METHOD

{

public static void main(String args[])

{

int area1,area2;

Rectangle r=new Rectangle(); //CREATING OBJECTS

Rectangle r1=new Rectangle();

r.length=10; //ACCESSING VARIABLES

r.width=25;

area1=r.length\*r.width;

r1.getData(12,10); //ACCESSING METHODS

area2=r1.rectarea();

System.out.println("Area1 = " +area1);

System.out.println("Area2= "+area2);

}

}

**VISIBILITY CONTROL (ACCESS MODIFIERS OR ACCESS SPECIFIERS)**

**PRIVATE:** The data members and member functions that are declared as **private** can be accessed only within the class and not from the outside the class.

**PUBLIC:** The data members and member function that are declared as **public** can be accessed within the class as well as from outside the class.

**PROTECTED:** The data members and member functions declared as **protected** cannot be accessed outside the class, but can be accessed from a derived class. The protected keyword is frequently used in inheritance of class.

**CONSTRUCTOR:**

Java supports a special type of method, called a constructor that enables an object to initialize itself when it is created. Constructors have the same name as the class itself. Secondly, they do not specify any return type, not even void. Types of constructors are **default constructor and parameterized constructor.**

**Default constructor** is a constructor with no arguments or not parameter list.

**Parameterized constructor** is a constructor with arguments.

The below example show both Default Constructor and Parameterized Constructor

**Example:**

class Rectangle

{

int length;

int width;

Rectangle() // constructor with no arguments

{

System.out.println( “ Default constructor”);

}

Rectangle(int x, int y) //constructor with arguments

{

length=x;

width=y;

}

int rectarea()

{

return (length\*width);

}

}

class ex\_a

{

public static void main(String args[])

{

Rectangle r=new Rectangle(); //calling default constructor

Rectangle rect=new Rectangle(14,10); // calling parameterized constructor

int area1=rect.rectarea();

System.out.println(area1);

//System.out.println(rect.rectarea());

}

}

**METHODS OVERLOADING**

Method overloading means, that we can create multiple methods with same name but with different parameters lists.

**Example:**

class overloading

{

void dis()

{

class room

{

int len,bred;

room(int x, int y)

{

len=x;

bred=y;

}

room(int x)

{

len=bred=x;

}

int area()

{

return(len\*bred);

}

}

class overloa

{

public static void main(String args[])

{

int a1,a2;

room r=new room(2,3);

room r1=new room(4);

System.out.println(r.area());

System.out.println(r1.area()); } }

System.out.println(" method overloading");

}

void dis(int x)

{

System.out.println(" x value is " +x);

}

}

class overload

{

public static void main(String args[])

{

overloading ovr=new overloading();

ovr.dis();

ovr.dis(2);

}

}

**STATIC MEMBERS:**

The members that are declared static are called static members and methods are called as static methods. Static members can be accessed without any object of the class. Static members have only one storage area. Static members are initialized only once. Static members are called as class variables and static methods are called as class methods. Even static methods are called without using the object.

Static members and method is defined below:

static int a,b;

static int max(int x, int y);

**Example:**

class stat

{

static int a, b;

static int sum(int x, int y)

{

return(x+y);

}

static int sub(int x, int y)

{

return(x-y);

}

}

class static\_1

{

public static void main(String args[])

{

stat.a=10;

stat.b=20;

System.out.println(stat.sum(2,5));

System.out.println(stat.sub(5,2));

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

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

}

}

**Static Block:**

A static block is a block of statements declared as static, like

static

{

Statements:

}

JVM executes a static block on a highest priority basis. This means JVM first goes to a static block even before it looks for the main( ) method in the program.

**Example:**

class Test

{

static

{

System.out.println("Static block");

}

public static void main(String args[])

{

System.out.println("Static method");

}

}

We thought that the main() method is the first one that is given attention by the JVM. This is ok if static block is not present in the program. If a static block is present, then JVM executes it first of all. After that, it searches for the main( ) method is not found, it will display an error.

**Example:**

class Test1

{

static

{

System.out.println("Static block");

}

}

**THIS KEYWORD :**

* Java defined ‘’this’’ is a keyword, that can be used inside any method to refer the current object.
* ‘’this’’ is always a reference to the object on which method was invoked.
* ‘’this’’ reference is implicitly used to refer to both the instance variables and methods of current object.

**Example :**

class Box

{

int len, dep, hei;

Box(int len,int dep, int hei)

{

this.len=len;

this.dep=dep;

this.hei=hei;

}

int volume()

{

return len \* dep \* hei ;

}

}

class This1

{

public static void main(String[] args)

{

Box b=new Box(1,2,3);

int x=b.volume();

System.out.println(‘’volume=’’ +x);

}

}

**Argument passing :**

Arguments passed to a sub – routine (method) is calls into two categories.

1. Call by value
2. Call by reference

**PASSING PRIMITIVE DATA TYPES TO METHODS/ CALL BY VALUE :**

 If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

1. **class** Operation{
2. **int** data=50;
3. **void** change(**int** data)
4. {
5. data=data+100 ;
6. }
7. **public** **static** **void** main(String args[]){
8. Operation op=**new** Operation();
9. System.out.println("before change "+op.data);     op.change(500);
10. System.out.println("after change "+op.data);
11. }  }

**Example:**

class Test

{

void math( int i, int j)

{

i \*=2;

j/=2;

}

}

class CallByValue

{

Public static void main(String[] args)

{

Test t=new Test();

int a=4,b=7;

System.out.println(‘’Before calling a=’’+a+’’b==’’+b);

t.math(a,b);

System.out.println(‘’After calling a=’’+a+’’b=’’+b);

}

}

**PASSING OBJECTS TO METHODS/ CALL BY REFERENCE :**

In case of call by reference original value is changed if we made changes in the called method. If we pass object in place of any primitive value, original value will be changed. In this example we are passing object as a value.

1. **class** Operation2{
2. **int** data=50;
4. **void** change(Operation2 op){
5. op.data=op.data+100;//changes will be in the instance variable
6. }            **public** **static** **void** main(String args[]){

   Operation2 op=**new** Operation2();

     System.out.println("before change "+op.data);

   op.change(op);//passing object

   System.out.println("after change "+op.data);

   }  }

**Example :**

class Test

{

int a,b;

Test(int i,int j)

{ a=i;

b=j;

}

void math(Test x)

{

x . a \*=2;

x .b/=2;

}

}

class Cbr

{

Public static void main(String[] args)

{

Test t=new Test(4,5);

System.out.println(“Before calling a=”+t.a+”b=”+t.b);

t.math(t);

System.out.println(“After calling a=”+t.a+”b=”+t.b);

}

}

**PASSING ARRAYS TO METHODS**

You can pass arrays to a method just like normal variables. When we pass an array to a method as an argument, actually the address of the array in the memory is passed (reference). Therefore, any changes to this array in the method will affect the array.

**Example:**

public class Arrmethod

{

public static void addTen(int[] arr)// int[] denotes that the parameter is an array

{

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

{

arr[i] += 10;

}

}

public static void main(String[] args)

{

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

addTen(arr);//Simply pass the name of the array to the method

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

{

System.out.print(arr[i] + " ");

}

}

}

**RECURSION**

Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method.

1. **public** **class** RecursionExample3 {
2. **static** **int** factorial(**int** n){
3. **if** (n == 1)
4. **return** 1;
5. **else**
6. **return**(n \* factorial(n-1));
7. }
9. **public** **static** **void** main(String[] args) {

System.out.println("Factorial of 5 is: "+factorial(5));  }

}

1. **public** **class** RecursionExample2 {
2. **static** **int** count=0;
3. **static** **void** p(){
4. count++;
5. **if**(count<=5){
6. System.out.println("hello "+count);
7. p();
8. }
9. }

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

p();   }  }

**FACTORY METHODS IN JAVA**

A factory method is a method that creates and returns an object to the class to which it belongs. A single factory method replaces several constructors in the class by accepting different options form the user, while creating the object.

**Example:**

import java.text.\*;

class Circle

{

public static void main(String args[])

{

final double PI=(double)22/7;

double r=15.5;

double area=PI\*r\*r;

System.out.println("Area= "+area);

NumberFormat obj= NumberFormat.getNumberInstance();

obj.setMaximumFractionDigits(2);

obj.setMinimumIntegerDigits(7);

String str=obj.format(area);

System.out.println("Formatted Area= "+str);

}

}

**Output:**

Area= 755.0714285714286

Formatted Area= 0,000,755.07

Factory methods are static methods only. To format the output of the preceding program to display as many digits as we want in the integer and fraction parts. This is achieved by **NumberFormat** class. **NumberFormat** class of **java.text** package is useful to format the number values.

 Create **NumberFormat** object. For this, we should use the factory method

**getNumberInstance( ).**

 Decide how to format the area value. Depending on this, we should use any of

the **NumberFormat** class methods like.

**setMaximumIntegerDigits( );**

**setMinimumIntegerDigits( );**

**setMaximumFractionDigits( );**

**setMinimum FractionDigits( );**

These methods specify how many integer digits or fraction digits to be displayed in the output.

 Apply the format to the area value using **format( )** method. This method returns a string that contains formatted area value.

The above program formatted output area is 0,000,755.07

**INHERITANCE**

Deriving new classes from existing classes such that the new classes acquire all the features of existing classes is called **Inheritance**.

The existing class is known as the **base class** or **super class** or **parent class** and the new class is called the **subclass** or **derived class** or **child class**.

We use inheritance for 1) code reusability 2) method overriding

**Syntax of Java Inheritance**

**class subclass\_name extends superclass\_name**

**{**

**//methods and fields**

**}**

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

class Employee

{

float salary=40000;

void disp()

{

System.out.println(" hello world");

}

}

class Programmer extends Employee

{

int bonus=10000;

public static void main(String args[])

{

Programmer p=new Programmer();

p.disp();

System.out.println("Programmer salary is:"+p.salary);

System.out.println("Bonus of Programmer is:"+p.bonus);

}

}

**TYPES OF INHERITANCE**

**Single Inheritance:** Derivation of a class from only one base class is called Single inheritance.

class A

{

........

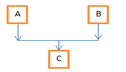
}

class B extends A

{

........

}

**Multiple Inheritance:** Derivation of a class from one or more base classes is called multiple inheritance. Java does not support multiple inheritance i.e. classes cannot have more than one super class. Java provides this concept implemented using interfaces.

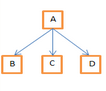
**Hierarchical Inheritance:** Derivation of several classes from one base class i.e. one class may be inherited by more than one class is called hierarchical inheritance.

class A

{

........

}

class B extends A

{

........

}

class C extends A

{

........

}

class D extends A

{

........

}

**Multilevel Inheritance:** Derivation of a class from another derived class is called Multilevel Inheritance.

class A

{

.......

}

class B extends A

{

.......

}

class C extends B

{

.......

}

**SUPER KEYWORD**

The super keyword refers to superclass (parent) objects.

It is used to call superclass methods, and to access the superclass constructor.

If we create an object to super class, we can access only the super class members, but not the sub class members. But if we create sub class object, all the members of both super and sub classes are available to it. So we always create an object to sub class in inheritance. Sometimes, the super class members and sub class members may have same name. In that case, by default, only sub class members are accessible. In such a case, how to access the super class members from sub class use the keyword **super**. The keyword **super** refers to super class members from a sub class. For example

* 1. The **super** can be used to refer to super class variables, as **super.varibale**
  2. The **super** can be used to refer to super class methods, as **super.method()**
  3. The **super** can be used to refer to super class constructor.

We need not call the default constructor of the super class, as it is by default available to sub class. To class the parameterized constructor, we can write, as **super(values);**

class Animal {

public void animalSound() {

System.out.println("The animal makes a sound");

}

}

class Dog extends Animal { public void animalSound() {

super.animalSound(); System.out.println("The dog says: bow wow");

}

}

public class Main {

public static void main(String args[]) {

Animal myDog = new Dog(); myDog.animalSound();

}

}

class One

{

int i=10;

void show()

{

System.out.println("super class method:i = "+i);

}

}

class Two extends One

{

int i=20;

void show()

{

System.out.println("sub class method:i = "+i); super.show();

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

}

}

class Super

{

public static void main(String args[])

{

Two t=new Two(); t.show();

}

}

**PROTECTED SPECIFIER**

The **private** members of the super class are not available to sub classes directly. But sometimes, there may be a need to access the data of super class in the sub class. For this purpose **protected** specifier is used. The **protected** is commonly used in super class to make the members of the super class available directly in its sub classes. We can think that the **protected** specifier works like **public** with respect to sub classes.

**Example:**

class Shape

{

protected double l;

Shape(double l)

{

this.l=l;

}

}

class Square extends Shape

{

Square(double l)

{

super(l);

}

void area()

{

System.out.println("Area of Square="+(l\*l));

}

}

class Rectangle extends Square

{

protected double b; Rectangle(double x, double y)

{

super(x); b=y;

}

void area()

{

System.out.println("Area of Rectangle="+(l\*b));

}

}

class Triangle extends Rectangle

{

Triangle(double x, double y)

{

super(x,y);

}

void area()

{

System.out.println("Area of Triangle="+(0.5\*l\*b));

}

}

class Override

{

public static void main(String args[])

{

Square s=new Square(5.5);

s.area();

Rectangle r=new Rectangle(5.5,6.0);

r.area();

Triangle t=new Triangle(5.5,6.0);

t.area();

}

}

**TYPE CASTING**

Converting one data type into another data type is called **type casting** or simply **casting**. To convert the data type we use cast operator. Cast operator means writing the data type between simple braces, before a variable or method whose value is to be converted.

# TYPES OF DATA TYPES:

There are two types of data types.

# Primitive data types (or) fundamental data types:

The data types which represent a single entity (or value) are called **primitive data types**. For example take **int** type, it can store only one integer value. Primitive data types are char, byte, short, int, long, float, double, boolean.

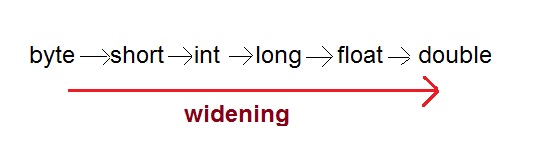
# Referenced data types or advanced data types:

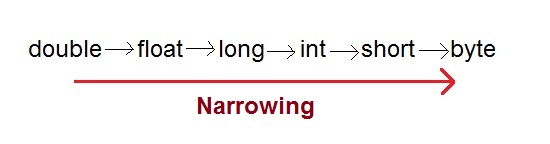
These data types represent several values. For example, take an array. It can store several values. Similarly take a class. It can store different values. So they are called advanced data types. We can access an array or an object of a class in memory through references. So, they are also called **referenced data types**.

We can convert a primitive data type into another primitive data type using casting. Similarly, it is possible to convert a referenced data type into another referenced data type by using casting. But we cannot convert a primitive data type into a referenced data type by using casting. For this purpose, methods of wrapper classes should be used.

# CASTING PRIMITIVE DATA TYPES:

It is possible to convert one primitive data type to another primitive data type. This is done in two ways, widening and narrowing.

* Widening Casting(Implicit)



* Narrowing Casting(Explicitly done)

#### Widening or Automatic type conversion

Automatic Type casting take place when,

* the two types are compatible
* the target type is larger than the source type

Example:

public class Test

{

public static void main(String[] args)

Output :

Int value 100

Long value 100

Float value 100.0

{

int i = 100;

long l = i; **//no explicit type casting required**

float f = l; **//no explicit type casting required**

System.out.println("Int value "+i);

System.out.println("Long value "+l);

System.out.println("Float value "+f);

}

}

#### Narrowing or Explicit type conversion

When you are assigning a larger type value to a variable of smaller type, then you need to perform explicit type casting.

**Example :**

public class Test

{

**Output :**

Double value 100.04

Long value 100

Int value 100

public static void main(String[] args)

{

double d = 100.04;

long l = (long)d; **//explicit type casting required**

int i = (int)l; **//explicit type casting required**

System.out.println("Double value "+d);

System.out.println("Long value "+l);

System.out.println("Int value "+i);

}

}

**ABSTRACT METHOD AND ABSTRACT CLASS**

An abstract method does not contain any body. It contains only the method header. So we can say it is an incomplete method. An abstract method must be redefined in a subclass, thus making overriding compulsory.

A class which is declared with the **abstract** keyword is known as an abstract class in [Java](https://www.javatpoint.com/java-tutorial). It can have abstract and non-abstract methods (method with the body).

Since, abstract class contains incomplete methods. It is not possible to estimate the total memory required to create the objects. So, JVM cannot create objects to an abstract class. We should create sub classes and all the abstract methods should be implemented in the sub classes.

**Example:**

abstract class Bank{

abstract int getRateOfInterest();

}

class SBI extends Bank{

int getRateOfInterest(){return 7;}

}

class PNB extends Bank{

int getRateOfInterest(){return 8;}

}

class TestBank{

public static void main(String args[]){

Bank b=new SBI();

System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");

Bank b1=new PNB();

System.out.println("Rate of Interest is: "+b1.getRateOfInterest()+" %");

}}

**INTERFACE**

The concept of multiple inheritance is not supported in java. The classes in java cannot have more than one superclass.

class A extends B extends C

{

---------------------------

---------------------------

}

A definition like above is not permitted in java. Java provides an alternate approach known as **Interfaces** to support the concept of multiple inheritance.

**DEFINING INTERFACES**

An ***interface*** is a collection of abstract methods and constants. A class implements an interface, thereby inheriting the abstract methods of the interface. An interface is not a class. Writing an interface is similar to writing a class. A class describes the attributes and behaviours of an object. An interface contains behaviours that a class implements.

**Important rules:**

1. You cannot instantiate an interface.
2. An interface does not contain constructors.
3. All of the methods in an interface are abstract.
4. All the variables of an interface are static and final.
5. An interface is not extended by a class, it is implemented by a class
6. An interface can extend multiple interfaces.

**Syntax:**

interface InterfaceName

{

variable declaration;

methods declaratioin;

}

From above, interface is the keyword and InterfaceName is any valid java variable.

Variables are declared as follows:

static final type VariableName=value;

**Example:**

interface item

{

static final int code=101;

static final String name="fan";

void display();

}

**Extending Interfaces:**

An interface can extend another interface, similarly to the way a class extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

**Syntax:**

interface one extends two

{

body of one

}

**Example:**

interface Itemconstants

{

int code=1000;

String name="fan";

}

interface Item extends Itemconstants

{

void display();

}

**Implementing Interfaces:**

Interfaces are used as “superclasses” whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface. This is done as follows:

**Syntax:**

class classname implements interfacename

{

body of classname

}

**Example:**

interface area

{

final static float pi=3.14f;

float compute(float x, float y);

}

class rectangle implements area

{

public float compute(float x, float y)

{

return x\*y;

}

}

class circle implements area

{

public float compute(float x,float y)

{

return (pi\*x\*y);

}

}

class tta

{

public static void main(String args[])

{

rectangle r=new rectangle();

circle cr=new circle();

System.out.println(r.compute(10,20));

System.out.println(cr.compute(10,30));

} }

**MULTIPLE INHERITANCE USING INTERFACES**

Derivation of a class from several base classes is called multiple inheritances. Java does not support multiple inheritance i.e. classes cannot have more than one super class. Java provides this concept implemented using interfaces.

**Example:**

interface Printable

{

void print();

}

interface Showable

{

void show();

}

class A implements Printable, Showable

{

public void print()

{

System.out.println("Hello");

}

public void show()

{

System.out.println("Welcome");

}

}

class MultiInterface

{

public static void main(String args[])

{

A obj = new A();

obj.print();

obj.show();

}

}

**PACKAGES**

A package represents a directory that contains related group of classes and interfaces. For example,

import java.io.\*;

**Advantages or Benefits**

1. Packages are useful to arrange related classes and interfaces in to a group. This puts together all the classes and interfaces performing the same task in the same package
2. Java package removes naming collision.
3. Packages can hide the classes and interfaces in a separate sub directory.
4. Packages provide a way for separating design from coding.

**TYPES OF PACKAGES:**

1. **Java API packages (built-in packages)**
2. **user-defined packages.**

# Built – in packages:

These are the packages which are already available in java language. These packages provide nearly all the necessary classes, interfaces and methods for the programmer to perform any task in the program. Frequently used packages are shown below:

|  |  |
| --- | --- |
| **Package name** | **Contents** |
| java.lang | They include classes for primitive types, strings, math functions, threads and exceptions. |
| java.util | Utility classes such as vectors, hash tables, random numbers, data, etc |
| java.io | Input/output support classes |
| java.awt | Set of classes for implementing graphical user interface. |
| java.net | Classes for networking |
| java.applet | Classes for creating and implementing applets. |
| java.sql | This package helps to connect database like oracle. |

# User – defined packages:

The users of the java language can also create their own packages. They are called user – defined packages. User – defined packages can also be imported into other classes and used exactly in the same way as the Built – in packages.

**Steps to create our own package**

1. Declare the package at the beginning of a file using the form package packagename:

2. Define the class that is to be put in the package and declare it public.

3. Create a subdirectory under the directory where the main source files are stored.

4. Store the classname.java file in the subdirectory created.

5. Compile the file. This creates .class file in the subdirectory.

**ACCESSING A PACKAGE**

The **import** statement is used to access the packages. The general form of import statement is as follows:

**Syntax:**

**import packagename.classname**

**or**

**import packagename.classname;**

**USING A PACKAGE**

Consider a program

Package package1;

public class Test

{

public void display( )

{

System.out.println("Class A");

}

}

This source file should be named **Test.java** and stored in the subdirectory **package1**. Now compile this java file. The resultant **Test.class** will be stored in the same subdirectory.

**Using the package1**

import package1. Test;

class PackageTest1

{

public static void main(String args[] )

{

Test obj = new Test( ) ;

obj.display( );

}

}

Save the above program as **PackageTest1.java** in the directory **Package** and compile the program to produce **PackageTest1.class**. Execute the program to produce the result.

**ADDING A CLASS TO A PACKAGE**

It is simple to add a class to an existing package. Consider the following package:

package p1;

public class A

{

public void disp()

{

System.out.println(" JAVA ");

}

}

**Adding a class B to a package**

package p1;

public class B

{

public void disp()

{

System.out.println(" PROGRAMMING ");

}

}

The package p1 contains classes A and B.

**INTERFACES IN PACKAGES**

It is also possible to write interfaces in a package. But whenever, we create an interface the implementation classes should also be created. We cannot create an object to the interface but we can create objects for implementation classes and use them.

Example:

// to implement interfaces in package package mypack;

import java.util.\*; interface MyDate

{

void showDate();

}

public class DateImpl implements MyDate

{

public void showDate()

{

Date d=new Date();

System.out.println(d);

}

}

When the preceding code is complied, DateImpl.class file is created in the same package mypack. DateImpl class contained showDate() method which can be called and used in any other program. DateDisplay is class where we want to use DateImpl class.

**Example:**

import mypack.\*; class DateDisplay

{

public static void main(String args[])

{

DateImpl obj=new DateImpl(); obj.showDate();

}

}

**CREATING SUB PACKAGE IN A PACKAGE**

We can create sub package in a package in the format: **package pack1.pack2;** Here, we are creating pack2 which is created inside pack1. To use the classes and interfaces of pack2, we can write import statement as: **import pack1.pack2**. This concept can be extended to create several sub packages.

**Creating a package:**

package pack1.pack2;

public class Sample

{

public void show()

{

System.out.println("Welcome to pack2");

}

}

When the preceding program is compiled, the java compiler creates sub directory with **pack1**. Inside this, there would be another sub directory with the name **pack2** is created. In this **pack2,** Sample class is stored. Suppose the user wants to use Sample class of **pack1.pack2** package, we can write the statement **import pack1.pack2.Sample;**

# Accessing a package:

import pack1.pack2.Sample; class UseSubPack

{

public static void main(String args[])

{

Sample s=new Sample(); s.show();

}

}

**CASTING REFERENCED DATA TYPES:**

A class is a referenced data type. Converting a class type into another class type is also possible through casting. But the classes should have some relationship between them by the way of inheritance. For example, we cannot convert a Dog class into a Horse class, as those classes do not have any relationship between them. And we can a convert Department class into a College, since Department is a sub class of College class. The classes Department, College & University have relationship by the way of inheritance.

**POLYMORPHISM**

Polymorphism came from the two Greek words ‘poly’ meaning many and morphos meaning forms. The ability to exist in different forms is called ‘polymorphism’. A variable, an object or a method can exist in different forms, thus performing various tasks depending on the context.

**Polymorphism with variables:**

When using variables, sometimes inherently the data type of the result is decided by the compiler and accordingly execution proceeds. For example

System.out.println(a+b);

Java compiler decides the data type of the result of the expression **a+b** depending on the data types of **a** and **b**. If **a** and **b** are int type, then **a+b** will also be taken as int type. If **a** and **b** are float type variables, then **a+b** will be taken as float type. If **a** is int **b** is float, then the compiler converts **a** also into float and then sum is found. Thus, the result **a+b** is exhibiting polymorphic nature. It may exist as an int or as a float or as some other data type depending on the context. This is also called **coercion**. Coercion is the automatic conversion between different data types done by the compiler.

Another example,

**float a=15.5f;**

**int x= (int)a;**

In, the second line, the actual data type of the variable is changed by using cast operator. Even if **a** is float type, it is converted into int type. If **a** is taken as it is, it becomes float type and because we converted, it can take the form of an int. This means **a** exists in two different forms. This also comes under polymorphism, which is called **conversion**. **Conversion** is an explicit change in the data type specified by the cast operator.

**Polymorphism using methods:**

If the same method performs different tasks, then that method is said to exhibit polymorphism. There may be two methods with the same name and they can perform different tasks. When we call the methods, we use same name but the task will be different depending on which method (body) is called.

Now, which method is called, the decision may happen either at compile time or at runtime. This will lead to two types of polymorphism, **Static polymorphism** and **Dynamic polymorphism.** The word **Static** represents at compile time and **Dynamic** represents at runtime.

**Dynamic polymorphism:** The polymorphism exhibited at runtime is called dynamic polymorphism. This means when a method is called, the method call is bound to the method body at the time of running the program, dynamically. In this case, java compiler does not know which method is called at the time of compilation. Only JVM knows at runtime which method is to be executed. Hence, this is also called **runtime polymorphism** or **dynamic binding**.

## Method overloading:

Writing two or more methods in the same class in such way that each method has same name but different method signature is called method overloading.

**Example:**

// Method overloading

class Sample

{

void add(int a, int b)

{

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

}

void add(int a, int b, int c)

{

System.out.println("Sum of three=" +(a+b+c));

}

}

class MethodOverload

**Output:**

Sum of two=25 Sum of three=45

{

public static void main(String args[])

{

Sample s=new Sample(); s.add(10,15);

s.add(10,15,20);

}

}

## Method overriding:

Writing two or more methods in super and sub classes such that the methods have same name and same signature is called method overriding.

**Example:**

// Method overriding

class One

{

void calculate(double x)

{

System.out.println("Square value=" +(x\*x));

}

}

**Output:**

Square root=5.0

class Two extends One

{

void calculate(double x)

{

System.out.println("Square root=" +Math.sqrt(x));

}

}

class MethodOverriding

{

public static void main(String args[])

{

Two t=new Two();

t.calculate(25);

}

}

# Static polymorphism:

The polymorphism exhibited at the compilation time is called static polymorphism. Here, the java compiler without ambiguity which method is called at the time of compilation. Of course, JVM executes the method later, but the compiler knows and can bind the method call with method code (body) at the time of compilation. So, it is called **static binding** or **compile time polymorphism**.

# Examples:

**Polymorphism with Static Methods**

A static method is a method whose single copy in memory is shared by all the objects of the class. Static methods belong to the class rather than to the objects. So they are also called class methods. When static methods are overloaded or overridden, since they do not depend on the objects, the java compiler need not wait till the objects are created to understand which method is called.

# Example:

// polymorphism with Static Methods

class One

{

static void calculate(double x)

{

System.out.println("Square value=" +(x\*x));

}

}

class Two extends One

{

static void calculate(double x)

{

System.out.println("Square root=" +Math.sqrt(x));

}

}

**Output:**

Square value=625.0

class StaticMethods

{

public static void main(String args[])

{

One o=new Two();

o.calculate(25);

}}

# Polymorphism with Private Methods

Private methods are the methods which are declared by using the access specifier **private**. This access specifier makes the method non available outside the class. So other programmers cannot access the private methods. Even private methods are not available in the sub classes. This means, there is no possibility to override the private methods of the super class in its sub classes. So only method overloading is possible in case of private methods.

# Example:

# // polymorphism with private methods

**class PrivateMethods**

# {

# private static void add(int a, int b)

**Output:**

Sum of two=25 Sum of three=45

# {

# System.out.println("Sum of two=" +(a+b));

# }

# private static void add(int a, int b, int c)

# {

# System.out.println("Sum of three=" +(a+b+c));

# }

# public static void main(String args[])

# { add(10,15);

# add(10,15,20);

# } }

# Polymorphism with Final Methods

# Methods which are declared as final are called final methods. Final methods cannot be overridden, because they are not available to the sub classes. Therefore, only method overloading is possible with final methods

# Example:

# class A

# {

# final static void method1()

# {

# System.out.println("Hello");

# }

# }

# class B

# {

# void method2()

# {

# A.method1();

# }

# }

# class FinalMethods

# {

# public static void main(String args[])

# {

# B b=new B();

# b.method2();

# }}

# FINAL CLASS

A final class is a class which is declared as **final**. The **final** keyword before a class prevents inheritance. This means sub classes cannot be treated to a final class.

**Example:**

final class Sample

{

final int x=10;

final void square()

{

System.out.println("square="+x\*x);

}

}

class TestFinal

{

public static void main(String args[])

{

Sample s=new Sample();

s.square();

}}

# OBJECT CLASS

There is a class with the name **Object** in **java.lang** package which is the super class of all classes in java. Every class in java is a direct or indirect sub class of the **Object** class. The **Object** class defines the methods to compare objects, to convert an object into a string, to notify threads (processes) regarding the availability of an object, etc.

**Object** class reference can store any reference of any object. This becomes an advantage when we want to write a method that needs to handle objects of unknown type. If we define a parameter of **Object** type, any class object can be passed to the method. Thus, the method can receive any type of object and handle it.

The methods of **Object** class are

|  |  |
| --- | --- |
| **Method** | **Description** |
| equals( ) | This method compares the references of two objects and if they  are equal, it returns true, otherwise false. |
| toString( ) | This method returns a string representation of an object. |
| getClass( ) | This method gives an object that contains the name of a class to  which an object belongs. |
| hashCode( ) | This method returns hash code number of object. |
| notify( ) | This method sends a notification to a thread which is waiting for  an object. |
| notifyAll( ) | This method sends a notification for all waiting threads for the  object. |
| wait( ) | This method causes a thread to wait till a notification is received  from a notify( ) or notifyAll( ) methods. |
| clone( ) | This method creates a bit wise exact copy of an existing object. |
| finalize( ) | This method is called by the garbage collector when an object is  removed from the memory. |

Example:

//equals( ) method compares Myclass objects references & same method is used to compare Integer class objects.

class Myclass

{

int x;

Myclass(int x)

{

this.x=x

}

}

class Compare

{

public static void main(String args[])

{

Myclass obj1=new Myclass(10);

Myclass obj2=new Myclass(10);

Integer obj3=new Integer(10);

Integer obj4=new Integer(10);

if(obj1.equals(obj2))

System.out.println("obj1 and obj2 are same");

else

System.out.println("obj1 and obj2 are not same");

if(obj3.equals(obj4))

System.out.println("obj3 and obj4 are same");

else

System.out.println("obj3 and obj4 are not same");

}

}

**JAR FILEs**

A JAR (Java Archive) file is a file that contains compressed version of **.class** file, audio files, image files or directories. We can imagine a .jar file as a zipped file (.zip) that is created by using WinZip software. Even, WinZip software can be used to extract the contents of .jar file. The difference is that a .jar file can be used but the .zip file cannot be used directly. The files should be extracted first from a .zip file, and then used.

# To create a .jar file and related commands as

1. To create .jar file, JavaSoft people have provided jar command, which can be used in following way

# Syntax: jar cf jarfilename inputfiles

Here, cf represents create file. For example, assuming our package **pack** is available in G:\JAVA\ directory, to convert into a jar file with the name pack.jar.

**Example:** G:\JAVA>jar cf pack.jar pack

Now, pack.jar file is created.

1. To view the contents of a .jar file, we can use the jar command as

# Syntax: jar tf jarfilename1`

Here, tf represents table view of file contents. For example, to view the contents of our pack.jar file, we can use

**Example:** G:\JAVA>jar tf pack.jar

Now, the contents of pack.jar are displayed as: META-INF/

META-INF/MANIFEST.MF

pack/

pack/Add.class

pack/Add.java

pack/Subtraction.class

pack/Subtraction.java

The first two entries represent that there is a manifest file created and added to pack.jar file. The third entry represents the sub directory with the name pack and the last four represent the file names in the directory pack.

When we create a .jar file, it automatically receives the default manifest file. There can be only one manifest file in an archive, and it always has the pathname: **META-INF/MANIFEST.MF**

This manifest file is useful to specify the information about other files which are packaged.

1. To extract the files from a .jar file, we can use

# Syntax: jar xf jarfilename

Here, xf represent extract files from the jar file. For example, to extract the contents of our pack.jar file, we can write:

# Example: G:\JAVA>jar xf pack.jar

This will create the following directories in G:\JAVA\META-INF

In software development, any package converted into a .jar file and stored in a separate sub directory. For, example converts our package pack into pack.jar file and stored it in a sub directory E:\temp.

Now set the CLASSPATH permanently to the pack.jar file by following the procedure.

1. First, go to Start Settings Control Panel
2. In control Panel, select System and double click on it, System properties dialog box appears
3. In this, select Advanced tab and then click on Environment variables button
4. Go to User variables and click on New button
5. Set the CLASSPATH variable to pack.jar and also the current directory, by typing at

Variable name: CLASSPATH Variable value: E:\TEMP\pack.jar;

1. Then click on the OK button
2. Then click on the OK button in the Environment variables window and System properties windows.
3. Close the Control Panel.

After setting the CLASSPATH permanently, it is available anywhere in that computer system. Our program (UsePack.java) which uses the package may present in any directory, it can be compiles and run without any problem.

**CREATING API DOCUMENT**

Application Programming Interface (API) document is a hypertext markup language (html) file that contains description of all the features of software, a product or a technology. API document is like a reference manual that is useful to all the users of the product to understand all the features and how to use them. For example, JavaSoft people have created API document separately for the three parts, Java SE, Java EE and Java ME after they are created java language. The java documentation can be downloaded from the following page of Sun Micro Systems site, <http://java.sun.com/javase/downloads/index.jsp>.

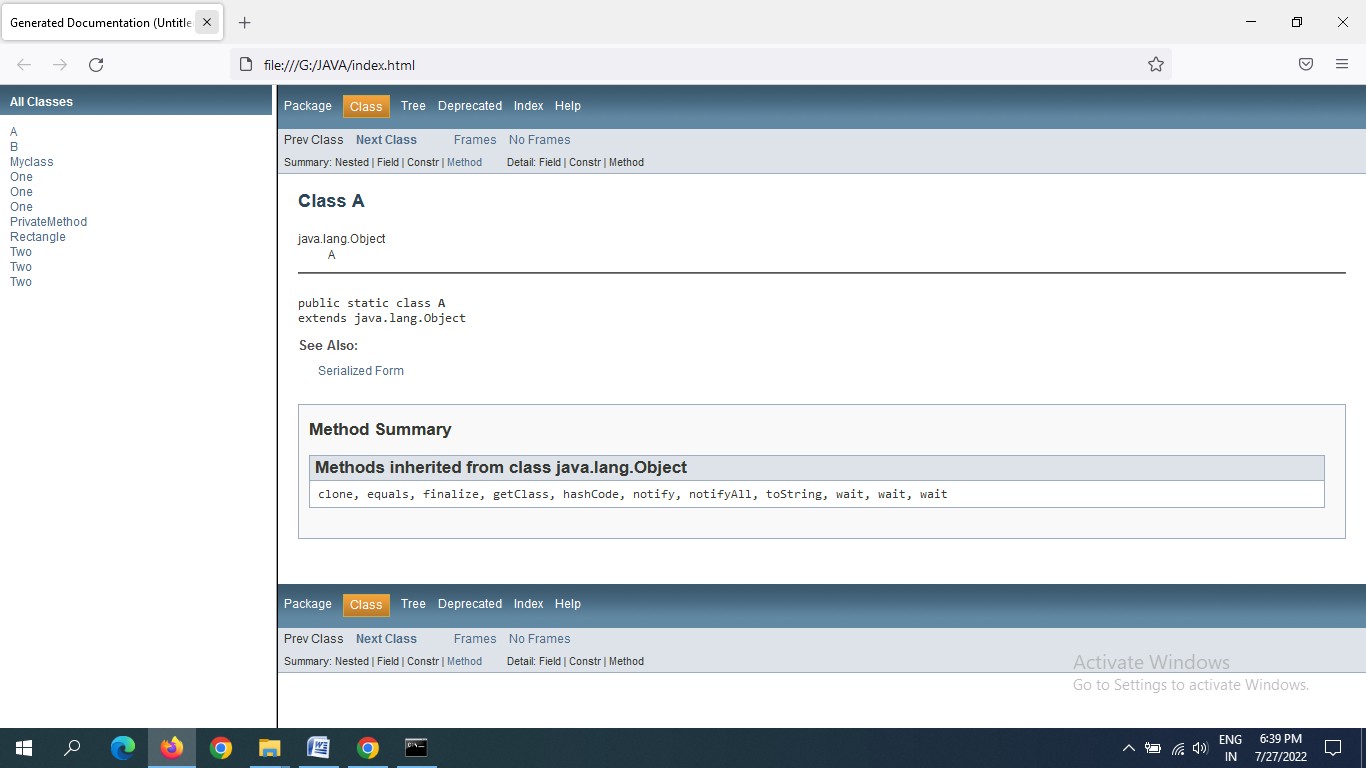
Every features of java language is described in API document. We can select any package, any class in that package and the description of the class along with the fields, constructors, methods will appear. When we click on any of these features, a detailed description of the features is displayed.

There are three important uses of API document.

1. API document is useful for the user to understand all the features of the product. The user can understand how to create an object to a class, which fields and methods are available in the class and how to use them.
2. API document makes programming easy. By understanding the methods of the API, the program will be able to use those methods and construct his programs easily.
3. Any third party vendor can provide implementation classes for the interfaces depending on the information available on the interfaces in the API.

To generate API document, we should give the following command:

# E:\JAVA>javadoc \*.java

Here, we are calling **javadoc** compiler to create the API document. When the preceding command is typed, then several .html files are created in G:\JAVA directory. Among them, observe **index.html** file and open it in the browser. It looks as

**EXCEPTION HANDLING**

As human beings, we commit many errors. A software engineer may also commit several errors while designing the project or developing the code. These errors are also called “bugs” and the process of removing them is called “debugging”.

An error may produce an incorrect output or may terminate the execution of the program abruptly or even may cause the system to crash.

**TYPES OF ERRORS:**

Errors may broadly be classified into three categories:

* Compile-time errors
* Run-time errors
* Logical errors

**COMPILE- TIME ERRORS:** The errors that are made while writing the program are known as "Compile-Time Errors". These errors include syntactical errors, misspelled identifiers and keywords, improper terminations of loops and blocks etc. These errors are detected and displayed by the Java compiler. It is necessary to fix all these errors before we successfully compile and run the program.

public class ChanceOfErrors

{

public static void main(String args[])

{

System.out.println("Hello Java) // Missing semicolon

}

}

In the above program, we missed to place a semicolon symbol to terminate the statement. It leads a compile-time error because when we compile the program then itself the error will be raised by the compiler.

**RUN-TIME ERRORS:** The errors that are raised at the time of executing a program are called as "Run-Time Errors". Sometimes, a program may compile successfully and creates the class file. But while executing that program, it may be terminated abruptly because of an error.

Run-Time errors lead to terminate a program abnormally even without the knowledge of user. It is very hard to find out run-time error. Most common run-time errors are:

1. Dividing an integer value by zero

2. Trying to access an element out of the bounds of an array

3. Trying to change the state of a thread illegally

**LOGICAL ERRORS:** Logical errors are the bugs. Which are not identified by the compiler but programmer can identify logical errors after seeing the wrong result.

Ex:

public class Logicerr

{

public static void main(String arg[ ])

{

double sal =5000.00;

sal=sal\* 15/100; // wrong use: sal+=sal\*15/100;

System.out.println(“ increment salaray =”+sal);

}

}

By comparing the output of a program with manually calculated results, a programmer can guess the presence of a logical error.

**EXCEPTIONS**

An exception is a runtime error. All exceptions occur only at runtime but some exceptions are detected at compile time and some others at runtime.

Exceptions that are checked at compilation time by the java compiler are called ‘**checked exception’.**

Exceptions that are checked by JVM are called ‘**unchecked exceptions’.**

If the exception object is not caught and handled properly, the interpreter will display an error message. If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as **exception handling.**

The basic concepts of exception handling are throwing an exception and catching it. This is shown in the below figure.



Java uses a keyword **try** to preface a block of code that is likely to cause an error condition and throw an exception. A catch block defined by the keyword catch “**catches**" the exception "throws" by the try block and handles it appropriately. The catch block is added immediately after the try block. The 'following example illustrates the use of simple try and catch statements: ………………….

………………….

Try

{

statement ; // generates an exception

}

catch(exception-type e)

{

Statement ; // processes the exception type

}

……………………………

……………………………

The try block can have one or more statements that could generate an exception. If any one statement generates an exception, the remaining statements in the block are skipped and exception jumps to the catch block that is placed next to the try block;

The catch block too can have one or more statements that are necessary to process the exception. Every try statement should be followed by at least one catch statement, otherwise compilation error will occur.

**Example:**

class exth

{

public static void main(String args[])

{

int a=10;

int b=5;

int c=5;

int x,y;

try

{

x=a/(b-c);

}

catch(ArithmeticException e)

{

System.out.println("division by zero");

}

y=a/(b+c);

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

}

**Handling Multiple Exceptions**

Most of the times there is possibility of more than one exception present in the program. In this case, the programmer should write multiple catch blocks to handle each one of them.

try

{

statement;

}

catch(Exception-Type-1 e)

{

statement;

}

catch(Exception-Type-2 e)

{

statement;

}

catch(Exception-Type-N e)

{

statement;

}

**Example:**

class exth

{

public static void main(String args[])

{

int a[]={5,10};

int b=5;

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");

}

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

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

}

}

**output:**

array index error

y=2

**Finally Keyword:**

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred.

A finally block appears at the end of the catch blocks and has following syntax:

try { //Protected code }

catch(ExceptionType1 e1)

{ //Catch block }

catch(ExceptionType2 e2)

{ //Catch block }

catch(ExceptionType3 e3)

{ //Catch block }

finally

{ //The finally block always executes. }

**Example:**

public class ExcepTest

{

public static void main(String args[])

{

int a[]=new int[2];

try

{

System.out.println("Access element three :"+ a[3]);

}

catch(ArrayIndexOutOfBoundsException e)

{

System.out.println("Exception thrown :"+ e);

}

finally

{

a[0]=6;

System.out.println("First element value: "+a[0]);

System.out.println("The finally statement is executed");

}

}

}

**Output:**

Exception thrown :java.lang.ArrayIndexOutOfBoundsException:3

First element value:6

The finally statement is executed

**THROW CLAUSE**

The Java **throw** keyword is used to throw an exception explicitly.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception.

**Example:**

class TestThrow1 {

public static void validate(int age) {

if(age<18) {

throw new ArithmeticException("Person is not eligible to vote");

}

else {

System.out.println("Person is eligible to vote!!");

}

}

public static void main(String args[]){

validate(13);

System.out.println("rest of the code...");

}

}

class sample

{

static void demo()

{

try{

System.out.println(" inside demo");

throw new ArithmeticException(" example ");

}

catch(ArithmeticException x)

{

System.out.println(x);

}

}

}

class throwsa

{

public static void main(String args[])

{

sample.demo();

} }

**THROWS CLAUSE**

The **Java throws keyword** is used to declare an exception. It gives information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.

**Syntax:**

return\_type method\_name() **throws** exception\_class\_name{

//method code

}

**Example:**

class TestThrows {

static void validate() throws Exception

{

System.out.println(" cal");

System.out.println(10/0);

}

public static void main(String args[])

{

try

{

validate();

}

catch(Exception e)

{

System.out.println(e);

}

System.out.println("rest of the code...");

}

}

**TYPES OF EXCEPTIONS**

1. Built-in exceptions
2. User-defined exceptions

Built-in exceptions: These are the exceptions which are already available in Java. List of important built-in exceptions.

|  |  |
| --- | --- |
| **Exception Type** | **Cause of Exception** |
| ArithmeticException | caused by math errors such as division by zero |
| ArraylndexOutOfBoundsException | Caused by bad array indexes |
| ArayStoreException | Caused when a program tries to store the wrong type of data in an array |
| ClassNotFoundException | Caused when we try to access a class whose definition is not found. |
| FileNotFoundException | Caused when an attempt to access a non existent file |
| IOException | Caused by general I/O failures, such as inability to read from a file |
| NoSuchFieldException | Caused when a class does not contain the field specified |
| NoSuchMethodException | Caused when accessing a method which is not found. |
| NullPointerException | Caused by referencing a null object |
| NumberFormatException | Caused when a conversion between strings and number fails |
| OutOfMemoryException | Caused when there's not enough memory to allocate a new object |
| SecurityException | Caused when an applet tries to perform an acting not allowed by the browser's security setting. |
| StackOverFlowException | Caused when the system runs out of stack space |
| StringIndexOutOfBoundsException | Caused when a program attempts to access a non existent character position in a string. |

-

**User –defined exceptions:**

User can create his own exceptions which are called ‘user-defined exceptions’.

When user wants to raise his own exception, he/she should create an object to his exception class and throw it using throw clause.

**Example:**

class InvalidAgeException extends Exception

{

public InvalidAgeException (String str)

{

super(str);

}

}

class CustEx

{

static void validate (int age) throws InvalidAgeException

{

if(age < 18){

throw new InvalidAgeException("age is not valid to vote");

}

else {

System.out.println("welcome to vote");

}

}

public static void main(String args[])

{

try

{

validate(13);

}

catch (InvalidAgeException ex)

{

System.out.println("Caught the exception");

System.out.println("Exception occured: " + ex);

}

System.out.println("rest of the code...");

}

}

**RE-THROWING AN EXCEPTION**

When an exception occurs in a try block, it is caught by a catch block. This means that the thrown exception is available to the catch block. The following code shows how to re-throw the same exception out from the catch block.

try

{

throw exception;

}

catch(Exception ex)

{

throw exception;

}

**Example:**

class A

{

void method1()

{

try

{

String str="hello";

char ch=str.charAt(5);

}

catch(StringIndexOutOfBoundsException sie)

{

System.out.println(" please see the index is within the range");

throw sie;

}

}

}

class B

{

public static void main(String args[])

{

A a=new A();

try

{

a.method1();

}

catch(StringIndexOutOfBoundsException sie)

{

System.out.println(" i caught rethorwn exception ");

}

}

}

**STREAMS**

Stream allows transporting data from one place to another. Different streams are needed to send or receive data. Without streams, it is not possible to move data in java.

Stream carries data from one place to another. Stream categorized as “input streams” and “output streams”. Input streams are the streams which receive or read data while output stream are the stream which send or write data. All streams are represented in java.io package.

Classification of stream is “byte streams” and “text streams”.

Byte streams represent data in the form of individual bytes. Text streams represent data as characters of each 2 bytes. If a class name ends with the word “**Stream**”, then it comes under byte streams. **InputStream** reads bytes and **OutputStream** writes bytes.

**For example:**

FileInputStream

FileOutputStream

BufferedInputStream

BufferedOutputStream

If a class ends with the word “**Reader**” or “**Writer**” then it is taken as a text stream.

**Reade**r reads text and **Writer** writes text.

**For example:**

FileReader

FileWriter

BufferedReader

BufferedWriter

Byte streams are used to handle any characters (text), images, audio and video files.

Text streams are used to handle text files like the ones we create in Notepad. They cannot handle images, audio or video files.

**CREATING A FILE USING FILEOUTPUTSTREAM**

FileOutputStream class belongs to byte stream and stores the data in the form of individual bytes. It can be used to create text file. The file stored on secondary storage devices like hard disk or CD. Follow the following steps to create a text file that stores some characters.

1. DataInputStream class is used for reading data from the keyboard as below.

DataInputStream dis=new DataInputStream(System.in)

1. FileOutputStream class which send data to the file.

FileOutputStream fout=new FileOutputStream(“myfile.txt”);

1. The next step is to read from DataInputStream and write into FileOutputStream.
2. Finally, any file should be closed after performing the operations.

**Example:**

import java.io.\*;

class CreateFile

{

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

{

DataInputStream dis=new DataInputStream(System.in);

FileOutputStream fout=new FileOutputStream("myfile.txt");

System.out.println("Enter text (@ at the end):");

char ch;

while((ch=(char)dis.read())!='@')

fout.write(ch);

fout.close();

}

}

**Reading Data From A File Using Fileinputstream**

FileInputStream is useful to read data from a file in the form of sequence of byte. It is possible to read data from a text file using FileInputStream.

Attach the file to a FileInputStream as shown here:

FileInputStream fin=new FileInputStream(‘myfile.txt”);

Read the data from the file.

ch=fin.read();

read() method reads all the characters from the file. When no more data available to read, the read() method returns -1.

Then we should attach the monitor to some output stream. We use System.out.

System.out.print(ch);

**Example:**

import java.io.\*;

class ReadFile

{

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

{

FileInputStream fin=new FileInputStream("myfile.txt");

System.out.println(" File contents ");

int ch;

while((ch=fin.read())!=-1)

System.out.print((char)ch);

fin.close();

}

}

**Create A File Using Filewriter**

FileWriter is useful to create a file by writing characters into it.

import java.io.\*;

class CreateFile1

{

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

{

String str=" This is about java and importance of java";

FileWriter fw=new FileWriter("text");

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

fw.write(str.charAt(i));

fw.close();

}

}

**Reading A File Using Filereader**

FileReader is a useful to read data in the form of characters from a ‘text’ file.

import java.io.\*;

class ReadFile1

{

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

{

int ch;

FileReader fr=null;

try

{

fr=new FileReader("text");

}

catch(Exception e)

{

System.out.println(" not found");

return;

}

while((ch=fr.read())!=-1)

System.out.print((char)ch);

fr.close();

}

}

**COUNTING NUMBER OF CHARACTERS IN A FILE**

import java.io.\*;

class Count

{

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

{

int ch;

boolean prev=true;

int char\_count=0;

int word\_count=0;

int line\_count=0;

FileInputStream fis=new FileInputStream(args[0]);

while((ch=fis.read()) !=-1)

{

if(ch!=' ')

++char\_count;

if(!prev && ch==' ')

++word\_count;

if(ch==' ')

prev=true;

else

prev=false;

if(ch=='\n')

++line\_count;

}

char\_count-=line\_count\*2;

word\_count+=line\_count;

System.out.println(" no of chars "+ char\_count);

System.out.println(" no of words "+ word\_count);

System.out.println(" no of lines "+ line\_count);

fis.close();

}

}

**FILE COPY**

Sometime we need to copy the entire data of a text file into another text file. Streams are useful in this case.

**Example:**

import java.io.\*;

class copyfile

{

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

{

int ch;

FileInputStream fis=new FileInputStream(args[0]);

FileOutputStream fos=new FileOutputStream(args[1]);

while((ch=fis.read())!=-1)

fos.write(ch);

fis.close();

fos.close();

System.out.print(" 1 file copied ");

}

}

**FILE CLASS**

File class of java.io package provides some methods to know the properties of a file or directory.

File class includes the following methods:

**boolean isFile():** This method returns true if the File object contains a filename, otherwise false.

**boolean isDirectory():** This method returns true if the File object contains a directory name.

**boolean canRead():** This method returns true if the File object contains a file which is readable.

**boolean canWrite():** This method returns true if the file is writeable.

**boolean can Excute():** This method returns true if the file is executable.

**boolean exists():** This method returns true when the File object contains a file or directory which physically exists in the computer.

**string getParent():** This method returns the name of the parent directory of a file or directory.

**string getPath():** This method gives the name of directory path of a file or directory.

**string getAbsolutePath():** This method give the absolute directory path of a file or directory location.

**boolean delete():** This method delete the file or directory whose name is in file object.

**boolean mkdir():** This method creates the directory whose name is give in File object.

**Example:**

import java.io.\*;

class FileProp

{

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

E:\>javac FileProp.java

E:\>java FileProp myfile.txt

File name :myfile.txt

path:myfile.txt

Absolute Path :E:\\myfile.txt

Parent :null

Exists :true

is file :true

is direcory :false

{

String fname=args[0];

File f=new File(fname);

System.out.println(" File name :"+f.getName());

System.out.println(" path:"+f.getPath());

System.out.println(" Absolute Path :"+f.getAbsolutePath());

System.out.println(" Parent :"+f.getParent());

System.out.println(" Exists :"+f.exists());

System.out.println(" is file :"+f.isFile());

System.out.println(" is direcory :"+f.isDirectory());

}

**}**

**SERIALIZATION OF OBJECTS**

Serialization is a process of converting the state of an object into a byte stream. We use a class **ObjectOutputStream** to store object into a file. **Serializable** interface should be implemented by the class whose objects are to be stored in the file. Interface is available in java.io package.

**Example:**

import java.io.\*;

class Employee implements Serializable

{

String name;

int num;

Employee(String name, int num)

{

this.name=name;

this.num=num;

}}

class SerializeDemo

{

public static void main(String args[])

{

Employee e=new Employee("sreenu",101);

try

{

FileOutputStream fos=new FileOutputStream("serial.ser");

ObjectOutputStream oos=new ObjectOutputStream(fos);

oos.writeObject(e);

oos.close();

fos.close();

}

catch(Exception e1)

{

System.out.println(e1.getMessage());

} }}

**De-serialization:**

De-serialization is a process of reading back the objects from a file.

**Example:**

import java.io.\*;

class Employee implements Serializable

{

String name;

int num;

Employee(String name, int num)

{

this.name=name;

this.num=num;

}}

class DeSerializeDemo

{

public static void main(String args[])

{

Employee e=new Employee("sreenu",101);

try

{

FileInputStream fis=new FileInputStream("serial.ser");

ObjectInputStream ois=new ObjectInputStream(fis);

Employee e2=(Employee)ois.readObject();

ois.close();

fis.close();

System.out.println(e2.name+" "+e2.num);

}

catch(Exception e1)

{

System.out.println(e1.getMessage());

} } }

An applet is a Java program that runs in a web browser. Applets can be transported over the internet from one computer to another computer and run using **Applet Viewer or any web browser** that supports Java. An applet, like any application program, can do many things for us. It can perform arithmetic operations, display graphics, play sounds, accept user input, create animation, and play interactive games.

**CREATING AN APPLET**

To create an applet, we need to write a java program and compile it to get byte code. Then include into a HTML page. This page is then stored in the web server. A client machine communicate with the web server, the server then send the HTML page that contains an applet. The applet is executed on the client’s web browser. Thus applets are executed at client side by the web browser.

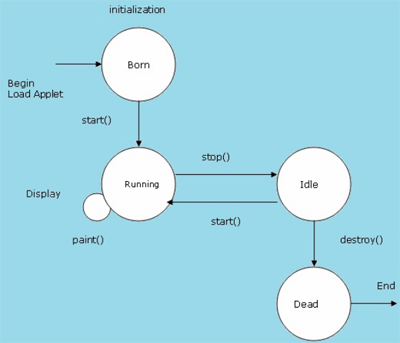
To create an applet, we have Applet class of java.applet package and JApplet class of javax.swing package. These classes use the following methods, which are automatically run by any applet program.

**public void init():** This is the first method to be called by the browser and it is executed only once. The applet is born. The programmer can use this method to initialize any variables, load images or fonts, set colors.

**public void start():** This method is called after init() method and each time the applet is revisited by the user. For example, when the user minimizes the web page, execution is stopped. When the user comes back to view the web page, start() method execution will resume.

**public void stop():** An applet becomes idle when it is stopped from running. Stopping occurs automatically when we leave the page containing the currently running applet. We can also do so by calling the stop() method explicitly.

**public void destroy():** This method is called when the applet is being terminated from memory. Destroying stage occurs only once. when we quit the browser, the destroy method is invoked.

***Executing init(), start(), stop() and destroy() methods in that sequence is called “Life cycle of an applet”. Note none of these methods are compulsory while writing an applet.***

Main method is not available in the case of applets. We can compile the applet code but we cannot run it using JVM.

To display the result of the applet code, the ***paint()*** method of the Applet class is called up.

The output may be text, graphics, or sound.

The syntax of ***paint()*** method which requires a Graphic object as an argument, is defined as follows

public void paint(Graphics g)

This requires that the applet code imports the ***java.awt*** package that contains the Graphic class.

**USES OF APPLETS**

1) Applets technology is a cross platform technology which allows us to work with different platforms very easily.

2) Applets technology is supported by majority of the web browsers available today.

3) Used for creating animation and games where the images can be displayed or moved giving a visual impression that they are alive.

**Disadvantages:**

1) To execute an applet, the browser needs a java plug-in

2) It cannot start until the Java Virtual Machine is running and this may have significant startup time the first time it is used.

3) Applets may require a specific JRE

**DIFFERENCE BETWEEN APPLETS AND APPLICATION**

|  |  |
| --- | --- |
| **Applet** | **Application Program** |
| 1) Do not Use main( ) method | 1) Main method is must |
| 2) cannot run independently | 2) Run independently |
| 3) Applets cannot read/write to the files in the local computer | 3) It can read and write a file from local computer |
| 4) No communication between Applets | 4) Communication is possible between two application programs |
| 5) Applets cannot use libraries from other languages (C or C++) | 5) Application programs can use libraries from other languages through native method. |
| 6) Internet browsers are required to run remote Applets. | 6) Browsers are not required to run application programs |
| 7) HTML files is required to run an applet. | 7) HTML files is not required |
| 8) Applets use the interface provided by the AWT | 8) Do not use the AWT interface |

**APPLET TAG**

The ***<Applet...>*** tag is useful to embed an applet into a HTML page. It has the following from:

<Applet code = "Hellojava.class"

width = 400 Height = 200

align=alignment (left,right, top, bottom, middle) >

<PARAM NAME =parameter name VALUE=its value>

</Applet>

**A SIMPLE APPLET**

Let us create a simple applet that displays a message in the applet frame. To display we need the **paint()** method of component class of **java.awt package**. All the methods of the applet and the applet class itself should be declared public, otherwise they are not available to the browser to execute.

import java.awt.\*;

import java.applet.\*;

public class Jap extends Applet

{

String str;

public void init()

{

str=" S.B.V.R.DEGREE COLLEGE";

setBackground(color.yellow);

}

public void paint(Graphics g)

{

g.drawString(str,10,100);

}

}

**Jap.html**

<html>

<APPLET CODE=jap.class WIDTH=400 HEIGHT=300>

</APPLET>

</html>

After building the program, run the applet and the applet viewer as shown below

F:\javac Jap.java

F:\appletviewer Jap.html

**AN APPLET WITH SWING COMPONENTS**

We create components using javax.swing package. We create two push buttons “Click Here” and “Click”. On clicking the buttons result will be displayed in the text field.

**Example: Myform.java**

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

public class Myform extends JApplet implements ActionListener

{

String str;

JTextField tf;

JButton b,b1;

Container c;

public void init()

{

JFrame f=new JFrame("Button Example");

c=f.getContentPane();

c.setLayout(null);

f.setSize(400,400);

f.setLayout(null);

f.setVisible(true);

tf=new JTextField();

tf.setBounds(50,50, 150,20);

b=new JButton("Click Here");

b.setBounds(50,100,95,30);

b1=new JButton("Click");

b1.setBounds(150,100,95,30);

c.add(b);c.add(b1);c.add(tf);

b.addActionListener(this);

b1.addActionListener(this);

}

public void actionPerformed(ActionEvent e)

{

str=e.getActionCommand();

if(str.equals("Click Here"))

{

tf.setText("Welcome to Javatpoint.");

}

else

{

tf.setText(" welcome badvel");

}

}}

**Myform.html**

<html>

<applet code="Myform.class" height=300 width=300></applet>

</html**>**

**ANIMATION IN APPLETS**

One of the uses of applets is performing animation and developing games. Animation represents moving the objects from one place to another so that the objects look alive. To animate an object, we should first load it into Image class object, as:

Image img=getImage(GetDocumentBase(),”plane.gif”);

Image class belongs to java.awt package and getImage() method belongs to Applet class.

GetDocumentBase() is a method of Applet class that gives the directory path where the image is located.

To display the image in the applet frame, we can use drawImage() method of Graphics class, as:

g.drawImage(img,x,y,obj);

**Example:**

**animate.java**

import java.awt.\*;

import java.applet.\*;

public class animate extends Applet

{

public void paint(Graphics g)

{

Image img=getImage(getDocumentBase(),"plane.gif");

for(int i=0;i<400;i++)

{

g.drawImage(img,i,0,null);

try

{

Thread.sleep(20);

}

catch(InterruptedException e){ }

}

}

}

**Animate.html**

<html>

<applet code="animate.class" height=300 width=300></applet>

</html**>**

**PASSING PARAMETERS TO APPLETS**

We can pass user-defined parameters to an applet using <PARAM..> tags. Each <PARAM..> tag has a ***name*** attribute and a ***value*** attribute.

**Syntax:** <Applet>

<PARAM name=... value= ………..>

</Applet>

**EXAMPLE:**

**Create an applet program with name parameter.java**

import java.awt.\*;

import java.applet.\*;

public class parameter extends Applet

{

String str;

public void init()

{

str=getParameter("string1");

if(str==null)

str="java";

str="hello "+str;

}

public void paint(Graphics g)

{

g.drawString(str,10,100);

}

}

Now let us create a HTML file that contains this applet.

<html>

<applet code="parameter.class" height=200 width=400>

<param name="string1" value="reddyagru"></applet>

</html>

Save the above the file as param.html

Run the applet using the appletviewer as follows

**appletviewer param.html**

**A SIMPLE GAME WITH AN APPLET**

**ButtonGame.java**

**import java.awt.\*;**

**import java.awt.event.\*;**

**import javax.swing.\*;**

**public class ButtonGame extends JApplet implements MouseMotionListener, MouseListener**

**{**

**JButton b;**

**JLabel lbl;**

**static int score=0;**

**public void init()**

**{**

**Container c=getContentPane();**

**ImageIcon li=new ImageIcon("happy.gif");**

**b=new JButton(" click me",li);**

**b.setFont(new Font("helvetica",Font.BOLD,30));**

**b.setBounds(400,300,250,75);**

**c.add(b);**

**lbl=new JLabel();**

**lbl.setFont(new Font("Impact",Font.PLAIN,30));**

**lbl.setText("score:"+score);**

**lbl.setBounds(550,20,150,50);**

**c.add(lbl);**

**b.addMouseMotionListener(this);**

**b.addMouseListener(this);**

**}**

**public void mouseDragged(MouseEvent e)**

**{**

**int x=(int)(600\*Math.random());**

**int y=(int)(500\*Math.random());**

**b.setBounds(x,y,250,75);**

**}**

**public void mouseMoved(MouseEvent e)**

**{**

**int x=(int)(600\*Math.random());**

**int y=(int)(500\*Math.random());**

**b.setBounds(x,y,250,75);**

**}**

**public void mouseClicked(MouseEvent e)**

**{**

**ImageIcon li=new ImageIcon("sad.gif");**

**b.setIcon(li);**

**lbl.setForeground(Color.red);**

**score+=100;**

**lbl.setText("score :"+score);**

**}**

**public void mouseEntered(MouseEvent e)**

**{**

**int x=(int)(600\*Math.random());**

**int y=(int)(500\*Math.random());**

**b.setBounds(x,y,250,75);**

**}**

**public void mouseExited(MouseEvent e)**

**{**

**ImageIcon li=new ImageIcon("happy.gif");**

**b.setIcon(li);**

**}**

**public void mousePressed(MouseEvent e)**

**{**

**}**

**public void mouseReleased(MouseEvent e)**

**{**

**}**

**}**

**ButtonGame.html**

**<html>**

**<applet code="ButtonGame.class" height=500 width=700></applet>**

**</html>**