# DAYANANDA SAGAR COLLEGE OF ENGINEERING

*(An Autonomous Institute Affiliated to VTU, Belagavi)*

Shavige Malleshwara Hills, Kumaraswamy Layout, Bengaluru-560078

## Department of Information Science and Engineering

(Accredited by NBA & NAAC with 'A' Grade)

DSCE

Dept. of Information Science & Engg

# (2020-2021)

## COURSE NAME:INTRODUCTION TO JAVA

## COURSE CODE: 18IS6IEJVA

**Faculty:Mrs. Vani K.A, Asst.Prof**

**Mrs. Radhika T V, Asst.Prof**

# CONTENTS

# UNIT-1

# INTRODUCTION

## Introduction to Java

- Java is a successor of C & C++,it derived the syntax from C & object oriented features like encapsulation, inheritance, polymorphism data abstraction from C++.
- Java is an object-oriented programming language developed by Sun Microsystems and released in 1995. Java runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX
- Java also provides for portable programming with applets. Applets appear in a Web page much in the same way as images do, but unlike images, applets are dynamic and interactive.

## Creation of java

- James Gosling initiated the Java language project in June 1991 for use in one of his many set-top box projects.
- The language, initially called Oak ,also went by the name Green and ended up later being renamed as Java.
- Sun released the first public implementation as Java 1.0 in 1995. It promised **Write Once, Run Anywhere** (WORA), providing no-cost run-times on popular platforms.

## Why java? Or features of java

### Java is:

- **Object Oriented:** In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
- **Platform independent:** Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run.
- **Simple:** Java is designed to be easy to learn. If you understand the basic concept of OOP Java would be easy to master.
- **Secure:** With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
- **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:** Being architectural-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary which is a POSIX subset.
- **Robust:** Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.

- **Multithreaded:** With Java's multithreaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.
- **Interpreted:** Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light weight process.
- **High Performance:** With the use of Just-In-Time compilers, Java enables high performance.
- **Distributed:** Java is designed for the distributed environment of the internet.
- **Dynamic:** Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

**Java Architecture**

**Compilation and interpretation in Java**

As shown in Figure-1 Java combines both the approaches of compilation and interpretation. First, java compiler compiles the source code into bytecode. At the run time, Java Virtual Machine (JVM) interprets this bytecode and generates machine code which will be directly executed by the machine in which java program runs. So java is both compiled and interpreted language.

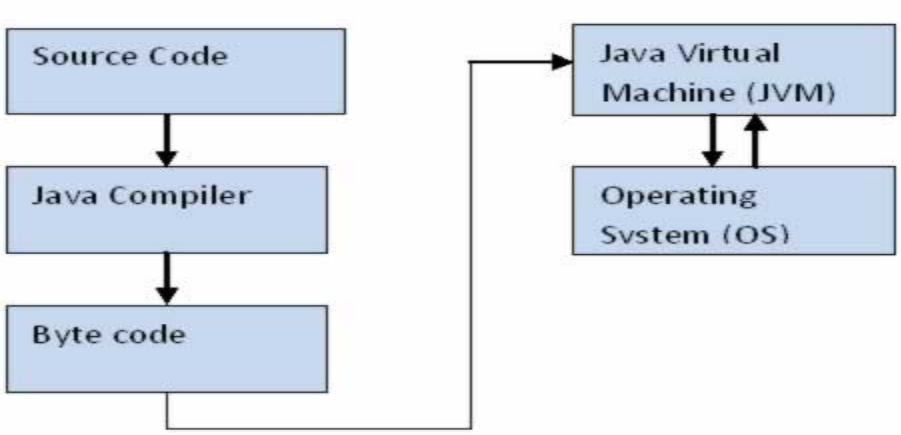



Figure 1: Java Architecture

**JDK**

- The Java Development Kit (JDK) is a software development environment used for developing Java applications and applets.
- It includes the Java Runtime Environment (JRE), an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (javadoc) and other tools needed in Java development.

As shown in Figure-2 JDK is composed of:

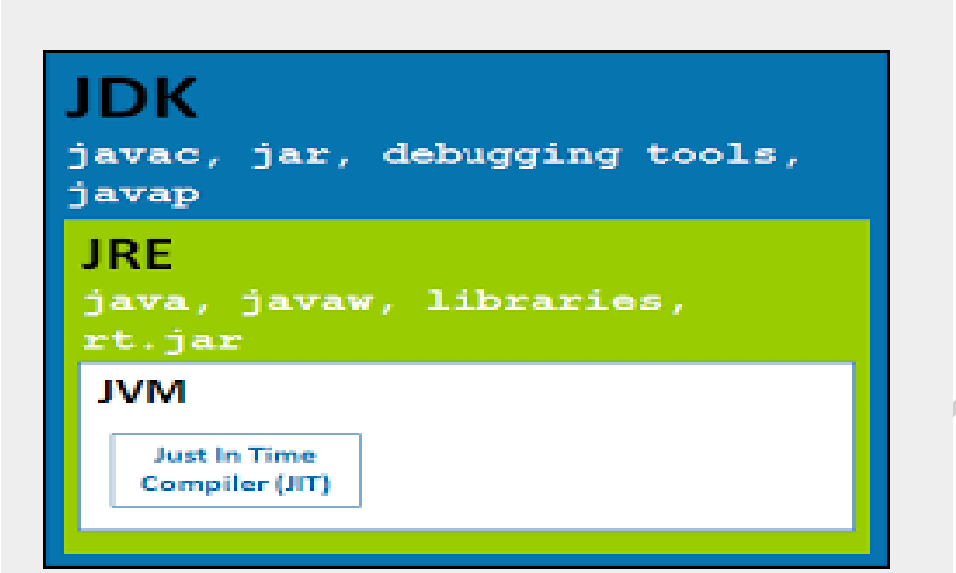



Figure -2 :JAVA DEVELOPMENT KIT

**JAVA is Interpreted**

- Java source code is compiled into bytecode, and the bytecode is interpreted by a platform specific JVM. The JVM acts as a interpreter for bytecode.

**Byte code**: is a highly optimized set of INSTRUCTIONS executed by JVM.Once a Java program has been converted to bytecode, it can be transferred across a network and executed by Java Virtual Machine (JVM). Bytecode files generally have a *.class* extension.

**Java Virtual Machine (JVM)**

As specified in Figure-3 JVM is a component which provides an environment for running Java programs. JVM interprets the byte code into machine code which will be executed the machine in which the Java program runs.

- This software component is the key to achieve Platform Independence feature of Java. Hence a Java program written and compiled on a Java-compliant platform can be run on any other Java-compliant platform.
- JVM is mainly composed of two components - Classloader and Execution Engine. The first component loads the .class files after converting them into the implementation dependent internal data structures. The Execution Engine uses these data structures are converts the bytecode instructions into machine-level instructions for the underlying Operating System which are subsequently executed by the OS.

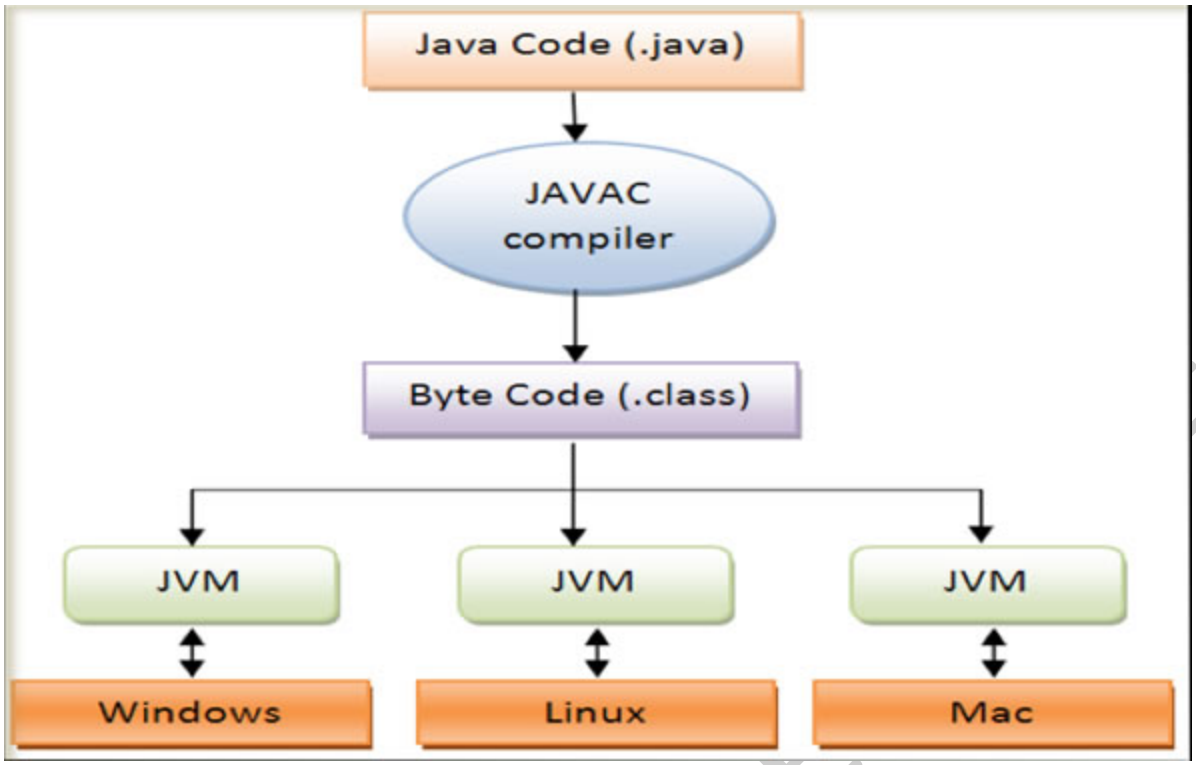



Figure-3: Java Virtual Machine

**Object oriented programming:**

Any program can be constructed based on 2 paradigms:

- Process oriented model
- Object oriented model

In Process oriented model it takes a program as a series of linear steps. This can be thought as code acting on data.

In Object oriented model it organizes a program around its data & a set of well defined interfaces to that data. This can be thought as data controlling access to the code.

**The three OOP principles:**

The OO model can be implemented using OOP principles they are:

- Encapsulation
- Inheritance
- Polymorphism

**Encapsulation:**

The whole idea behind encapsulation is to hide the implementation details from users. Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class. Access to the data and code is tightly controlled by an interface. The main benefit of encapsulation is the ability to modify our implemented code without breaking the code of others who use our code. With this feature Encapsulation gives maintainability, flexibility and extensibility to our code.

Eg:

```
public class EncapsulationDemo{
   private int ssn;
   private String empName;
   private int empAge;

   //Getter and Setter methods
   public int getEmpSSN(){
      return ssn;
   }

   public String getEmpName(){
      return empName;
   }

   public int getEmpAge(){
      return empAge;
   }

   public void setEmpAge(int newValue){
      empAge = newValue;
   }

   public void setEmpName(String newValue){
      empName = newValue;
   }

   public void setEmpSSN(int newValue){
      ssn = newValue;
   }
}
public class EncapsTest{
   public static void main(String args[]){
       EncapsulationDemo obj = new EncapsulationDemo();
       obj.setEmpName("Mario");
       obj.setEmpAge(32);
```

```
        obj.setEmpSSN(112233);
        System.out.println("Employee Name: " + obj.getEmpName());
        System.out.println("Employee SSN: " + obj.getEmpSSN());
        System.out.println("Employee Age: " + obj.getEmpAge());
    }
}
```

**Output:**

Employee Name: Mario
Employee SSN: 112233
Employee Age: 32

**Polymorphism :**

Polymorphism in java is a concept by which we can perform a *single action by different ways.* There are two types of polymorphism in java: compile time polymorphism and runtime polymorphism. Method overloading is a perfect example of compile time polymorphism. Method is a perfect example of runtime polymorphism.

Eg:Method Overloading

```
class DisplayOverloading
{
   public void disp(char c)
   {
       System.out.println(c);
   }
   public void disp(char c, int num)
   {
       System.out.println(c + " "+num);
   }
}
class Sample
{
  public static void main(String args[])
  {
     DisplayOverloading obj = new DisplayOverloading();
     obj.disp('a');
     obj.disp('a',10);
  }
}
```

**Output:**

a
a 10

**Method overriding with examples**

```
class Human{
  public void eat()
  {
    System.out.println("Human is eating");
  }
}
class Boy extends Human{
  public void eat(){
    System.out.println("Boy is eating");
  }
  public static void main( String args[]) {
    Boy obj = new Boy();
    obj.eat();
  }
}
```

**Output:**

Boy is eating

**Inheritance:** Inheritance can be defined as the process where one object acquires the properties of another. With the use of inheritance the information is made manageable in a hierarchical order.

To implement inheritance a keyword **extends** would be used in java.

Eg:

```
                class Vehicle
        {
                String vehicleType;
        }
                public class Car extends Vehicle {

                String modelType;
                public void showDetail()
```

```
{
        vehicleType = "Car";        //accessing Vehicle class member
        modelType = "sports";
        System.out.println(modelType+" "+vehicleType);
}
        public static void main(String[] args)
{
        Car car =new Car();
        car.showDetail();
}
}
```

**Output**

sports Car

**Simple java Programs:**

**Eg1:**

```
Class eg
{
Public static void main(String args[])
{
System.out.printlln("This Is Simple Java Pgm");
}
}
```

**Compiling the above program**

To compile the above program javac compiler is used

**C:\>javac eg.java**

The **javac** command compiles Java source code into Java bytecodes and creates eg.class file.

To run this program use

C:\> java eg

This displays an output: This Is Simple Java Pgm

**Eg2:**

Hello World example :

```
class HelloWorld {
public static void main (String args[]) {
System.out.println("Hello World! ");
}
}
```

Here the entire program is enclosed in a class definition—here, a class called Hello World.The body of the program (here, just the one line) is contained in a method (function) called main(). In Java applications, as in a C or C++ program, main() is the first method (function) that is run when the program is executed

**Output:** Hello World!

**Data types and other tokens**

Variables are locations in memory in which values can be stored. They have a name, a type, and a value. Java has three kinds of variables: instance variables, class variables, and local variables.

Data types: java defines 8 primitive types of data:
Integers, Floating point numbers, characters, Boolean

**Integer types.**

The width of an integer type should not be thought as the amount of storage it consumes but rather as the behavior it defines for variables and expressions of that type.

byte 8 bits —128 to 127
short 16 bits —32,768 to 32,767
int 32 bits —2,147,483,648 to 2,147,483,647
long 64 bits—9223372036854775808 to 9223372036854775807

This is a example for using Integer Data types

Find the addition of two numbers */

```
class addition

{

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

```
{
int i=8;
int j=7;
int k;
k= i + j ;
System.out.println(" The addition of " + i + "and" + j + " is" + k);
}
}
```

The output will be

The addition of 8 and 7 is 15

**Floating-point**

This is used for numbers with a decimal part. There are two floating-point types: float (32 bits, single-precision) and double (64 bits, double-precision).

**float**: The float data type is a single-precision 32-bit

- Float is mainly used to save memory in large arrays of floating point numbers.
- Default value is 0.0f.
- Float data type is never used for precise values such as currency.

eg: float empsal,temperature;

```
/* This is example for using floating-point Data Types
Find the square root of given numbers */
class squareroot
{
```

```
public static void main(String args[])
{
double x,y,z;
x=13;
y=17;
z=Math.sqrt(x+y);
System.out.println(" The Square Root is " +z);
}
}
```

The output will be

The Square Root is 5.477225575051661

**double**: The double data type is a double-precision 64-bit

- This data type is generally used as the default data type for decimal values, generally the default choice.
- Double data type should never be used for precise values such as currency.
- Default value is 0.0d.

eg: double d1 = 123.4

```
class area
{
public static void main (String args[])
double pi,r,a;
r=8.3;
```

```
pi=3.1416;

a=pi*r*r

System.out.println("area of circle is "+a);

}
```

**char data type** is a single 16-bit Unicode character.

- Minimum value is '\u0000' (or 0).
- Maximm value is '\uffff' (or 65,535 inclusive).
- Char data type is used to store any character.
- Example: char letterA ='A'

**Boolean Type**

The Boolean type holds either true or false using the reserved word true or false. The expression of the variable using boolean datatype will be either true or false.

For example the boolean data type will be assigned as

```
boolean b;

b=true;
```

or

```
boolean b= true;
```

**Literals**

Literals in Java are a sequence of characters (digits, letters, and other characters) that represent constant values to be stored in variables. Java language specifies five major types of literals. Literals can be any number, text, or other information that represents a value. This means what you type is what you get. We will use literals in addition to variables in Java statement. While writing a source code as a character sequence, we can specify any value as a literal such as an integer. They are:

**Integer literals:**

Integer data types consist of the following primitive data types: int,long, byte, and short. byte, int, long, and short can be expressed in decimal(base10), hexadecimal(base 16) or octal(base 8) number systems as well.Prefix 0 is used to indicate octal and prefix 0x indicates hexadecimal when using these number systems for literals.
Examples:

```
int decimal = 100;
int octal = 0144;
int hexa =  0x64;
```

**Floating-point literals:**
Floating-point numbers are like real numbers in mathematics, for example, 4.13179, -0.000001. Java has two kinds of floating-point numbers: float and double. The default type when you write a floating-point literal is double, but you can designate it explicitly by appending the D (or d) suffix. However, the suffix F (or f) is appended to designate the data type of a floating-point literal as float. We can also specify a floating-point literal in scientific notation using Exponent (short E ore), for instance: the double literal 0.0314E2 is interpreted as:

$0.0314 * 10^2$ (i.e 3.14).
6.5E+32 (or 6.5E32)  Double-precision floating-point literal
7D  Double-precision floating-point literal
.01f  Floating-point literal

**Character literals:**
char data type is a single 16-bit Unicode character. We can specify a character literal as a single printable character in a pair of single quote characters such as 'a', '#', and '3'. You must know about the ASCII character set. The ASCII character set includes 128 characters including letters, numerals, punctuation etc. Below table shows a set of these special characters.

| Escape | Meaning |
|---|---|
| \n | New line |
| \t | Tab |
| \b | Backspace |
| \r | Carriage return |
| \f | Formfeed |
| \\ | Backslash |
| \' | Single quotation mark |
| \" | Double quotation mark |
| \d | Octal |
| \xd | Hexadecimal |
| \ud | Unicode character |
| | |

**String Literals:**

The set of characters in represented as String literals in Java. Always use "double quotes" for String literals. There are few methods provided in Java to combine strings, modify strings and to know whether to strings have the same values.

| | |
|---|---|
| "" | The empty string |
| "\"" | A string containing |
| "This is a string" | A string containing 16 characters |
| "This is a " + "two-line string" | actually a string-valued constant expression, formed from two string literals |

**Operators and Expressions:**

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators

The Arithmetic Operators:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Assume integer variable A holds 10 and variable B holds 20, then:

```
public class Test {
  public static void main(String args[]) {
    int a = 10;
    int b = 20;
    int c = 25;
    int d = 25;
    System.out.println("a + b = " + (a + b) );
    System.out.println("a - b = " + (a - b) );
    System.out.println("a * b = " + (a * b) );
```

```
    System.out.println("b / a = " + (b / a) );
    System.out.println("b % a = " + (b % a) );
    System.out.println("c % a = " + (c % a) );
    System.out.println("a++   = " +  (a++) );
    System.out.println("b--   = " +  (a--) );
    // Check the difference in d++ and ++d
    System.out.println("d++   = " +  (d++) );
    System.out.println("++d   = " +  (++d) );
  }
}
```

Output is :

```
a + b = 30
a - b = -10
a * b = 200
b / a = 2
b % a = 0
c % a = 5
a++   = 10
b--   = 11
d++   = 25
++d   = 27
```



## The Relational Operators:

There are following relational operators supported by Java language

Assume variable A holds 10 and variable B holds 20, then:

```
public class Test {

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

```
    int a = 10;
    int b = 20;
    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("b >= a = " + (b >= a) );
    System.out.println("b <= a = " + (b <= a) );
  }
}
```

This would produce the following result:

```
a == b = false
a != b = true
a > b = false
a < b = true
b >= a = true
b <= a = false
```

The Bitwise Operators:

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; and b = 13; now in binary format they will be as follows:

a = 0011 1100

b = 0000 1101

---

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

The following table lists the bitwise operators:

Assume integer variable A holds 60 and variable B holds 13 then:

```
public class Test {

  public static void main(String args[]) {
    int a = 60;         /* 60 = 0011 1100 */
    int b = 13;         /* 13 = 0000 1101 */
    int c = 0;

    c = a & b;       /* 12 = 0000 1100 */
    System.out.println("a & b = " + c );

    c = a | b;       /* 61 = 0011 1101 */
    System.out.println("a | b = " + c );

    c = a ^ b;       /* 49 = 0011 0001 */
    System.out.println("a ^ b = " + c );

    c = ~a;          /*-61 = 1100 0011 */
    System.out.println("~a = " + c );

    c = a << 2;     /* 240 = 1111 0000 */
```

```
    System.out.println("a << 2 = " + c );

    c = a >> 2;     /* 215 = 1111 */
    System.out.println("a >> 2  = " + c );

    c = a >>> 2;     /* 215 = 0000 1111 */
    System.out.println("a >>> 2 = " + c );
  }
}
```

This would produce the following result:

```
a & b = 12
a | b = 61
a ^ b = 49
~a = -61
a << 2 = 240
a >> 15
a >>> 15
```

The Logical Operators:

The following table lists the logical operators:

Assume Boolean variables A holds true and variable B holds false, then:

```
public class Test {

  public static void main(String args[]) {
    boolean a = true;
    boolean b = false;

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

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

    System.out.println("!(a && b) = " + !(a && b));
  }
}
```

This would produce the following result:

```
a && b = false
a || b = true
!(a && b) = true
```

**Assignment operators:**

```
public class Test {

  public static void main(String args[]) {
    int a = 10;
    int b = 20;
    int c = 0;

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

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

    c -= a ;
    System.out.println("c -= a = " + c );
```

```
c *= a ;
System.out.println("c *= a = " + c );

a = 10;
c = 15;
c /= a ;
System.out.println("c /= a = " + c );

a = 10;
c = 15;
c %= a ;
System.out.println("c %= a  = " + c );

c <<= 2 ;
System.out.println("c <<= 2 = " + c );

c >>= 2 ;
System.out.println("c >>= 2 = " + c );

c >>= 2 ;
System.out.println("c >>= a = " + c );

c &= a ;
System.out.println("c &= 2  = " + c );

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

c |= a ;
```

```
    System.out.println("c |= a   = " + c );
  }
}
```

This would produce the following result:

```
c = a + b = 30
c += a  = 40
c -= a = 30
c *= a = 300
c /= a = 1
c %= a  = 5
c <<= 2 = 20
c >>= 2 = 5
c >>= 2 = 1
c &= a  = 0
c ^= a   = 10
c |= a   = 10
```

### Conditional Operator ( ? : ):

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as:

```
variable x = (expression) ? value if true : value if false
```

Following is the example:

```
public class Test {

  public static void main(String args[]){
    int a , b;
```

```
    a = 10;
    b = (a == 1) ? 20: 30;
    System.out.println( "Value of b is : " +  b );

    b = (a == 10) ? 20: 30;
    System.out.println( "Value of b is : " + b );
  }
}
```

This would produce the following result:

```
Value of b is : 30
Value of b is : 20
```

**Precedence of Java Operators:**

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example, x = 7 + 3 * 2; here x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

| Category | Operator | Associativity |
|---|---|---|
| Postfix | () [] . (dot operator) | Left toright |
| Unary | ++ - - ! ~ | Right to left |
| Multiplicative | * / % | Left to right |
| Additive | + - | Left to right |
| Shift | >> >>> << | Left to right |
| Relational | > >= < <= | Left to right |

| | | |
|---|---|---|
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | \| | Left to right |
| Logical AND | && | Left to right |
| Logical OR | \|\| | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= *= /= %= >>= <<= &= ^= \|= | Right to left |
| Comma | , | Left to right |

**Control Statements:**

The while Loop:

A while loop is a control structure that allows you to repeat a task a certain number of times.

Syntax:

The syntax of a while loop is:

```
while(Boolean_expression)
{
  //Statements
}
```

When executing, if the *boolean_expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.

Here, key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```
public class Test {

  public static void main(String args[]) {
    int x = 10;

    while( x < 20 ) {
      System.out.print("value of x : " + x );
      x++;
      System.out.print("\n");
    }
  }
}
```

This would produce the following result:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

**The do...while Loop:**

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax:

The syntax of a do...while loop is:

```
do
{
  //Statements
}while(Boolean_expression);
```

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

If the Boolean expression is true, the flow of control jumps back up to do, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

Example:

```
public class Test {

  public static void main(String args[]){
    int x = 10;

    do{
      System.out.print("value of x : " + x );
      x++;
      System.out.print("\n");
    }while( x < 20 );
  }
```

```
}
```

This would produce the following result:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
```

value of x : 17

value of x : 18

value of x : 19

## The for Loop:

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

A for loop is useful when you know how many times a task is to be repeated.

Syntax:

The syntax of a for loop is:

```
for(initialization; Boolean_expression; update)
{
  //Statements
}
```

Here is the flow of control in a for loop:

- The initialization step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement past the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the Boolean expression.
- The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

Example:

```
public class Test {

  public static void main(String args[]) {

    for(int x = 10; x < 20; x = x+1) {
      System.out.print("value of x : " + x );
      System.out.print("\n");
    }
  }
}
```

This would produce the following result:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
```

```
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

## Enhanced for loop in Java:

As of Java 5, the enhanced for loop was introduced. This is mainly used for Arrays.

Syntax:

The syntax of enhanced for loop is:

```
for(declaration : expression)
{
  //Statements
}
```

- **Declaration:** The newly declared block variable, which is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
- **Expression:** This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

Example:

```
public class Test {

  public static void main(String args[]){
    int [] numbers = {10, 20, 30, 40, 50};

    for(int x : numbers ){
      System.out.print( x );
      System.out.print(",");
```

```
    }
    System.out.print("\n");
    String [] names ={"James", "Larry", "Tom", "Lacy"};
    for( String name : names ) {
      System.out.print( name );
      System.out.print(",");
    }
  }
}
```

This would produce the following result:

```
10,20,30,40,50,
James,Larry,Tom,Lacy,
```

The break Keyword:

The *break* keyword is used to stop the entire loop. The break keyword must be used inside any loop or a switch statement.

The break keyword will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

The syntax of a break is a single statement inside any loop:

```
break;
```

Example:

```
public class Test {

  public static void main(String args[]) {
    int [] numbers = {10, 20, 30, 40, 50};
```

```
    for(int x : numbers ) {
      if( x == 30 ) {
                break;
      }
      System.out.print( x );
      System.out.print("\n");
    }
  }
}
```

This would produce the following result:

```
10
20
```

The continue Keyword:

The *continue* keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

- In a for loop, the continue keyword causes flow of control to immediately jump to the update statement.
- In a while loop or do/while loop, flow of control immediately jumps to the Boolean expression.

Syntax:

The syntax of a continue is a single statement inside any loop:

```
continue;
```

Example:

```
public class Test {

  public static void main(String args[]) {
    int [] numbers = {10, 20, 30, 40, 50};

    for(int x : numbers ) {
      if( x == 30 ) {
                continue;
      }
      System.out.print( x );
      System.out.print("\n");
    }
  }
}
```

This would produce the following result:

```
10
20
40
50
```

## UNIT-2

# CLASSES AND OBJECTS

**What is a Class in JAVA**

A class is a user defined blueprint or prototype from which objects are created. Objects are real life entities(or) it is an instance of class.

In general, class declarations can include these components, in order:

**Modifiers** : A class can be public or has default access

**Class name:**The name should begin with a initial letter

**Superclass(if any):**The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.

**Interfaces(if any):**A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.

**Body:**The class body surrounded by braces, { }.

An object consists of :

**State**: It is represented by attributes of an object. It also reflects the properties of an object.

**Behavior**: It is represented by methods of an object. It also reflects the response of an object with other objects.

**Identity**: It gives a unique name to an object and enables one object to interact with other objects.

Following is a sample of a class.

Example

```
public class Dog
{
 String breed;
```

```
int age;
String color;
void barking()
{
 }
 void hungry()
 {
}
void sleeping()
{
 }
 }
```

A class can contain any of the following variable types.

**Local variables**

**Instance variables**

**Class variables**

Creating an Object

There are three steps when creating an object from a class −

**Declaration** − A variable declaration with a variable name with an object type.

**Instantiation**− The 'new' keyword is used to create the object.

**Initialization**− The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

```
public class Puppy

{
 public Puppy(String name)
{ // This constructor has one parameter, name. System.out.println("Passed Name is :" + name );
}
 public static void main(String []args)
 { // Following statement would create an object myPuppy Puppy myPuppy = new Puppy( "tommy" );
 }
 }
```

Java - Constructors

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

Syntax

Following is the syntax of a constructor −

```
class ClassName
 {
ClassName()
{
 }
}
```

Java allows two types of constructors namely −

No argument Constructors

Parameterized Constructors

No argument Constructors

```
Public class MyClass
{
Int num;
MyClass()
 {
 num = 100;
}
}
```

You would call constructor to initialize objects as follows

```
public class ConsDemo
{
public static void main(String args[])
{
MyClass t1 = new MyClass();
MyClass t2 = new MyClass(); System.out.println(t1.num + " " + t2.num);
}
```

}

## Parameterized Constructors

Here is a simple example that uses a constructor −

// A simple constructor. class MyClass { int x; // Following is the constructor MyClass(int i ) { x = i; } }
You would call constructor to initialize objects as follows −

public class ConsDemo { public static void main(String args[]) { MyClass t1 = new MyClass( 10 );
MyClass t2 = new MyClass( 20 ); System.out.println(t1.x + " " + t2.x); } }
this keyword in java

**this** is a **reference variable** that refers to the currUsage of java this keyword

Here is given the 6 usage of java this keyword.

- this can be used to refer current class instance variable.
- this can be used to invoke current class method (implicitly)
- this() can be used to invoke current class constructor.
- this can be passed as an argument in the method call.
- this can be passed as argument in the constructor call.
- this can be used to return the current class instance from the method.

Understanding the problem without this keyword

Let's understand the problem if we don't use this keyword by the example given below:

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
rollno=rollno;
name=name;
fee=fee;
```

```
}
void display(){System.out.println(rollno+""+name+""+fee);} class TestThis1{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:
0 null 0.0 0 null 0.0

So, we are using this keyword to distinguish local variable and instance variable.

```
Class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+""+name+""+fee);}
}

class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:
111 ankit 5000 112 sumit 6000

this: to invoke current class method

```
class A{
void m(){System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();
```

```
}
}
class TestThis4{
public static void main(String args[]){
A a=new A();
a.n();
}}
```

o/p:

Hello n
Hello m

this() : to invoke current class constructor

```
class A{
A(){System.out.println("hello a");}
A(int x){
this();
System.out.println(x);
}
}
class TestThis5{
public static void main(String args[]){
A a=new A(10);
}}
```

o/p:

Hello a
10

Garbage Collection in Jav

the programmer need not to care for all those objects which are no longer in use. Garbage collector destroys these objects.

**Eligibility for garbage collection :** An object is said to be eligible for GC(garbage collection) iff it is unreachable.

**Ways to make an object eligible for GC**

different ways to make an object eligible for garbage collection.

Nullifying the reference variable

Re-assigning the reference variable

Object created inside method

**Ways for requesting JVM to run Garbage Collector**

We can also request JVM to run Garbage Collector. There are two ways to do it :

**Using *System.gc()*method** : System class contain static method*gc()* for requesting JVM to run Garbage Collector.

**Using *Runtime.getRuntime().gc()*method** :Runtime class allows the application to interface with the JVM in which the application is running. Hence by using its gc() method, we can request JVM to run Garbage Collector.

**finalize method**

It is a **method** that the **Garbage Collector** always calls just **before** the deletion/destroying the object which is eligible for Garbage Collection, so as to perform **clean-up activity** . Clean-up activity means closing the resources associated with that object like Database Connection, Network Connection or we can say resource de-allocation.

Once the finalize method completes immediately Garbage Collector destroy that object. finalize method is present in Object class and its syntax is:

**protected void finalize throws Throwable{}**

Method Overloading in Java

Different ways to overload the method

There are two ways to overload the method in java

By changing number of arguments

By changing the data type

1.) Method Overloading: changing no. of arguments

```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
classTestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

Output:

22 33

2.) Method Overloading: changing data type of arguments

```
class Adder{
```

```
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

**o/p:22 24.9**

# UNIT 3
## INHERITANCE ,INTERFACE, EXCEPTION HANDLING

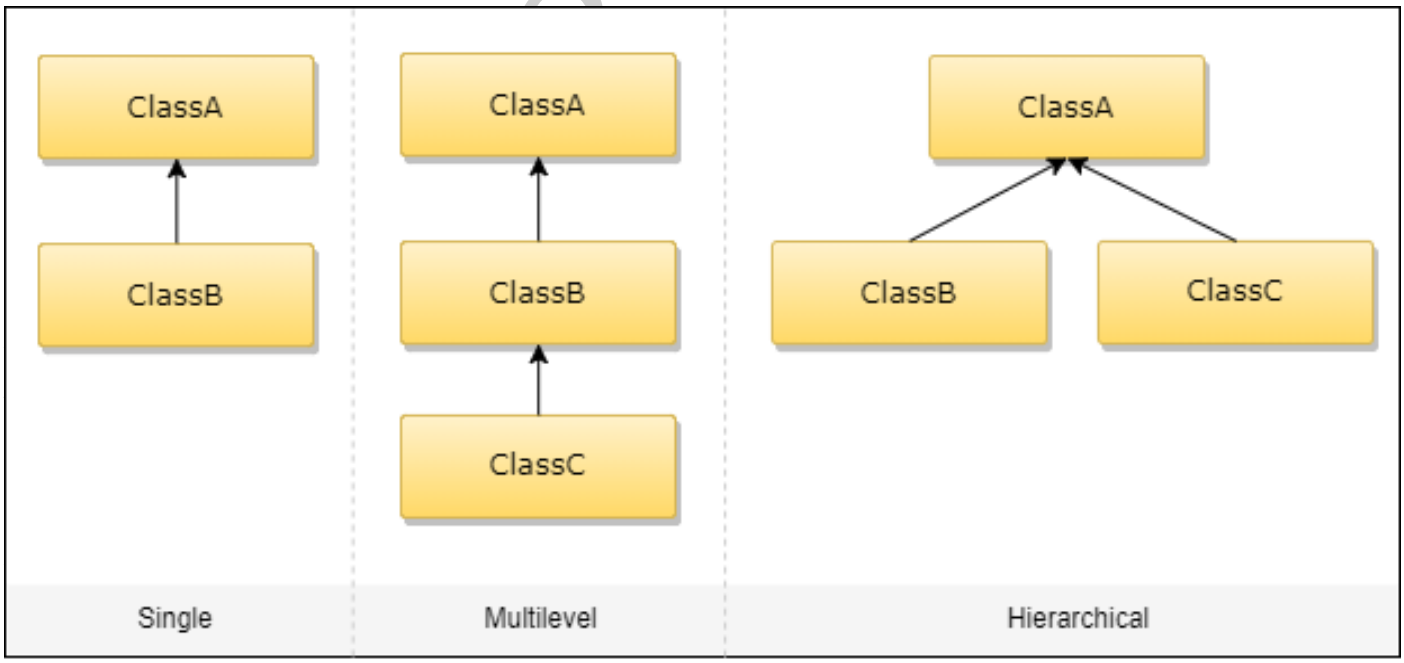
Inheritance can be defined as the procedure or mechanism of acquiring all the properties and behavior of one class to another, i.e., acquiring the properties and behavior of child class from the parent class

The keyword extends used to inherit the properties of the base class to derived class.

Syntax

```
class base
{
  .....
  .....
}
class derive extends base
 {
  .....
  .....
}
```

Java supports three types of inheritance




Java Inheritance

## Single Inheritance

When a single class gets derived from its base class, then this type of inheritance is termed as single inheritance. The figure drawn above has class A as the base class, and class B gets derived from that base class.

EXAMPLE:

```
class Teacher {
void teach() {
 System.out.println("Teaching subjects");
}
}
class Students extends Teacher {
 void listen() {
  System.out.println("Listening to teacher");
 }
}
class CheckForInheritance {
 public static void main(String args[]) {
  Students s1 = new Students();
  s1.teach();
  s1.listen();
 }
}
```



## Multi-level Inheritance

In this type of inheritance, a derived class gets created from another derived class and can have any number of levels.

EXAMPLE:

```
class Teacher {
 void teach() {
  System.out.println("Teaching subject");
 }
}
class Student extends Teacher {
 void listen() {
  System.out.println("Listening");
 }
}
class homeTution extends Student {
 void explains() {
  System.out.println("Does homework");
 }
}
class CheckForInheritance {
 public static void main(String argu[]) {
  homeTution h = new himeTution();
  h.explains();
  d.teach();
  d.listen();
 }
}
```

## Hierarchical Inheritance

In this type of inheritance, there are more than one derived classes which get created from one single base class.

```
class Teacher {
 void teach() {
  System.out.println("Teaching subject");
 }
}
class Student extends Teacher {
 void listen() {
  System.out.println("Listening");
 }
}
class Principal extends Teacher {
 void evaluate() {
  System.out.println("Evaluating");
 }
}
class CheckForInheritance {
public static void main(String argu[]) {
 Principal p = new Principal();
 p.evaluate();
 p.teach();
 // p.listen(); will produce an error
}
}
```

## INTERFACE

An interface can contain any number of methods.An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.The byte code of an interface appears in a **.class** file.Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including −

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

### 3.1.1 Declaring Interfaces

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface −

**Example**

Following is an example of an interface −

```
/* File name : NameOfInterface.java */
import java.lang.*;
// Any number of import statements

public interface NameOfInterface {
   // Any number of final, static fields
```

```
  // Any number of abstract method declarations\
}
```

Interfaces have the following properties −

- An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

**Example**

```
/* File name : Animal.java */
interface Animal {
  public void eat();
  public void travel();
}
```



### 3.1.2 Implementing Interfaces

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

**Example**

```
/* File name : MammalInt.java */
public class MammalInt implements Animal {

  public void eat() {
    System.out.println("Mammal eats");
  }

  public void travel() {
    System.out.println("Mammal travels");
  }

  public int noOfLegs() {
    return 0;
  }

  public static void main(String args[]) {
    MammalInt m = new MammalInt();
    m.eat();
    m.travel();
  }
}
```

This will produce the following result −

**Output**

```
Mammal eats
Mammal travels
```

When overriding methods defined in interfaces, there are several rules to be followed −

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
- An implementation class itself can be abstract and if so, interface methods need not be implemented.

When implementation interfaces, there are several rules −

1.A class can implement more than one interface at a time.
2.A class can extend only one class, but implement many interfaces.
3.An interface can extend another interface, in a similar way as a class can extend another class.

### 3.1.3 Extending Interfaces

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

The following Sports interface is extended by Hockey and Football interfaces.

**Example**

```
// Filename: Sports.java
public interface Sports {
   public void setHomeTeam(String name);
   public void setVisitingTeam(String name);
}

// Filename: Football.java
public interface Football extends Sports {
   public void homeTeamScored(int points);
   public void visitingTeamScored(int points);
   public void endOfQuarter(int quarter);
}

// Filename: Hockey.java
public interface Hockey extends Sports {
   public void homeGoalScored();
   public void visitingGoalScored();
   public void endOfPeriod(int period);
   public void overtimePeriod(int ot);
}
```

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

### 3.1.4 Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as −

**Example**

```
public interface Hockey extends Sports, Event
```

**Tagging Interfaces**

The most common use of extending interfaces occurs when the parent interface does not contain any methods. For example, the MouseListener interface in the java.awt.event package extended java.util.EventListener, which is defined as −

**Example**

```
package java.util;
public interface EventListener
{}
```

An interface with no methods in it is referred to as a **tagging** interface. There are two basic design purposes of tagging interfaces −

**Creates a common parent** − As with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends EventListener, the JVM knows that this particular interface is going to be used in an event delegation scenario.

**Adds a data type to a class** − This situation is where the term, tagging comes from. A class that implements a tagging interface does not need to define any methods sincetheinterfacedoesnothaveanysincetheinterfacedoesnothaveany, but the class becomes an interface type through polymorphism.

## 3.2 Exception handling: Fundamentals

Exception handling is one of the most important feature of java programming that allows us to handle the runtime errors caused by exceptions.

An Exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs program execution gets terminated. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled in Java. By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

If an exception occurs, which has not been handled by programmer then program execution gets terminated and a system generated error message is shown to the user. For example look at the system generated exception below:

**An exception generated by the system is given below:**

Exception in thread "main" java.lang.ArithmeticException: / by zero at
ExceptionDemo.main(ExceptionDemo.java:5)
ExceptionDemo : The class name
main : The method name
ExceptionDemo.java : The filename
java:5 : Line number

### 3.2.1 Types of exceptions:

There are two types of exceptions in Java:

1)Checked exceptions

2)Unchecked exceptions

## Checked exceptions

All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not. If these exceptions are not handled/declared in the program, you will get compilation error. For example, SQLException, IOException, ClassNotFoundException etc.

## Unchecked Exceptions

Runtime Exceptions are also known as Unchecked Exceptions. These exceptions are not checked at compile-time so compiler does not check whether the programmer has handled them or not but it's the responsibility of the programmer to handle these exceptions and provide a safe exit. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

### 3.2.2 try-catch block

#### Try block

The try block contains set of statements where an exception can occur. A try block is always followed by a catch block, which handles the exception that occurs in associated try block. A try block must be followed by catch blocks or finally block or both.

## Syntax of try block

```
try{
  //statements that may cause an exception
}
```

**Catch block**

A catch block is where you handle the exceptions, this block must follow the try block. A single try block can have several catch blocks associated with it. You can catch different exceptions in different catch blocks. When an exception occurs in try block, the corresponding catch block that handles that particular exception executes. For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

## Syntax of try catch in java

```
try
{
    //statements that may cause an exception
}
catch (exception(type) e(object))
{
    //error handling code
}
```

**Nested try catch block in Java**

When a try catch block is present in another try block then it is called the nested try catch block.

**Syntax of Nested try Catch**

```
....
//Main try block
try {
  statement 1;
  statement 2;   //try-catch block inside another try block
  try {
```

```
    statement 3;
    statement 4;
    //try-catch block inside nested try block
    try {
      statement 5;
      statement 6;
    }
    catch(Exception e2) {
      //Exception Message
    }
  }
  catch(Exception e1) {
    //Exception Message
  }

}
//Catch of Main(parent) try block
catch(Exception e3) {
    //Exception Message
}
....
```



**throw**: Throw keyword is used to transfer control from try block to catch block.

**Syntax of throw keyword:**

throw new exception_class("error message");

**throws**: Throws keyword is used for exception handling without try & catch block. It specifies the exceptions that a method can throw to the caller and does not handle itself.

**finally**: It is executed after catch block. We basically use it to put some common code when there are multiple catch blocks.

**Syntax of Finally block**

```
try {
    //Statements that may cause an exception
}
catch {
   //Handling exception
}
finally {
   //Statements to be executed
}
```

# UNIT -4

# INPUT/OUTPUT & PACKAGES:

## I/O Basics

JAVA brings various Streams with its I/O package that helps the user to perform all the input-output operations. These streams support all the types of objects, data-types, characters, files etc to fully execute the I/O operations.

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

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

Here is a list of the various print functions that we use to output statements:

•print(): This method in Java is used to display a text on the console. This text is passed as the parameter to this method in the form of String. This method prints the text on the console and the cursor remains at the end of the text at the console. The next printing takes place from just here.

Syntax:

**Syntax:** System.out.print(parameter);

```
// Java code to illustrate print()
import java.io.*;
class Demo_print {
    public static void main(String[] args)
    {
        // using print()
```

```
            // all are printed in the
            // same line
            System.out.print("GfG! ");
            System.out.print("GfG! ");
            System.out.print("GfG! ");
        }
}
```

**Output:**GfG! GfG! GfG!

**Println():** This method in Java is also used to display a text on the console. It prints the text on the console and the cursor moves to the start of the next line at the console. The next printing takes place from the next line.

**Syntax:**System.out.println(parameter);

```
// Java code to illustrate println()
import java.io.*;
class Demo_print {
    public static void main(String[] args)
    {
    // using println()
            // all are printed in the
            // different line
            System.out.println("GfG! ");
            System.out.println("GfG! ");
            System.out.println("GfG! ");
    }
} Output: GfG!
```

**GfG!**

**GfG!**

**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. This is used to format the output in Java.

**Example:**

```
// A Java program to demonstrate working of printf() in Java
class JavaFormatter1 {
    public static void main(String args[])
    {
        int x = 100;
        System.out.printf( "Printing simple"+ " integer: x = %d\n", x);
        // this will print it upto
        // 2 decimal places
        System.out.printf("Formatted with"+ " precison: PI = %.2f\n",
                Math.PI);
        float n = 5.2f;
        // automatically appends zero
        // to the rightmost part of decimal
System.out.printf("Formatted to "+ "specific width: n = %.4f\n", n);
        n = 2324435.3f;
        // here number is formatted from
        // right margin and occupies a
        // width of 20 characters
System.out.printf("Formatted to "+ "right margin: n = %20.4f\n", n);
```

```
    }
}
```

**O/P:**

Printing simple integer: x = 100

Formatted with precision:PI=3.14

Formatted to specific width:n=5.2000

Formatted to right marcgin:n=2324435.2500

**System.err:** 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.

This stream also uses all the 3 above-mentioned functions to output the error data:

- print()
- println()
- printf()

**Example:**

```
// Java code to illustrate standard
// input output streams

import java.io.*;
public class SimpleIO {
    public static void main(String args[]) throws IOException
    {       // InputStreamReader class to read input
        InputStreamReader inp = null;
        // Storing the input in inp
        inp = new InputStreamReader(System.in);
        System.out.println("Enter characters, "+ " and '0' to quit.");
```

```
            char c;
            do {
                c = (char)inp.read();
                System.out.println(c);
            } while (c != '0');
        }
}
```

Enter characters, and '0' to quit.

G

**e**

**e**

**k**

**s**

**f**

**o**

**r**

**G**

**e**

**e**

**k**

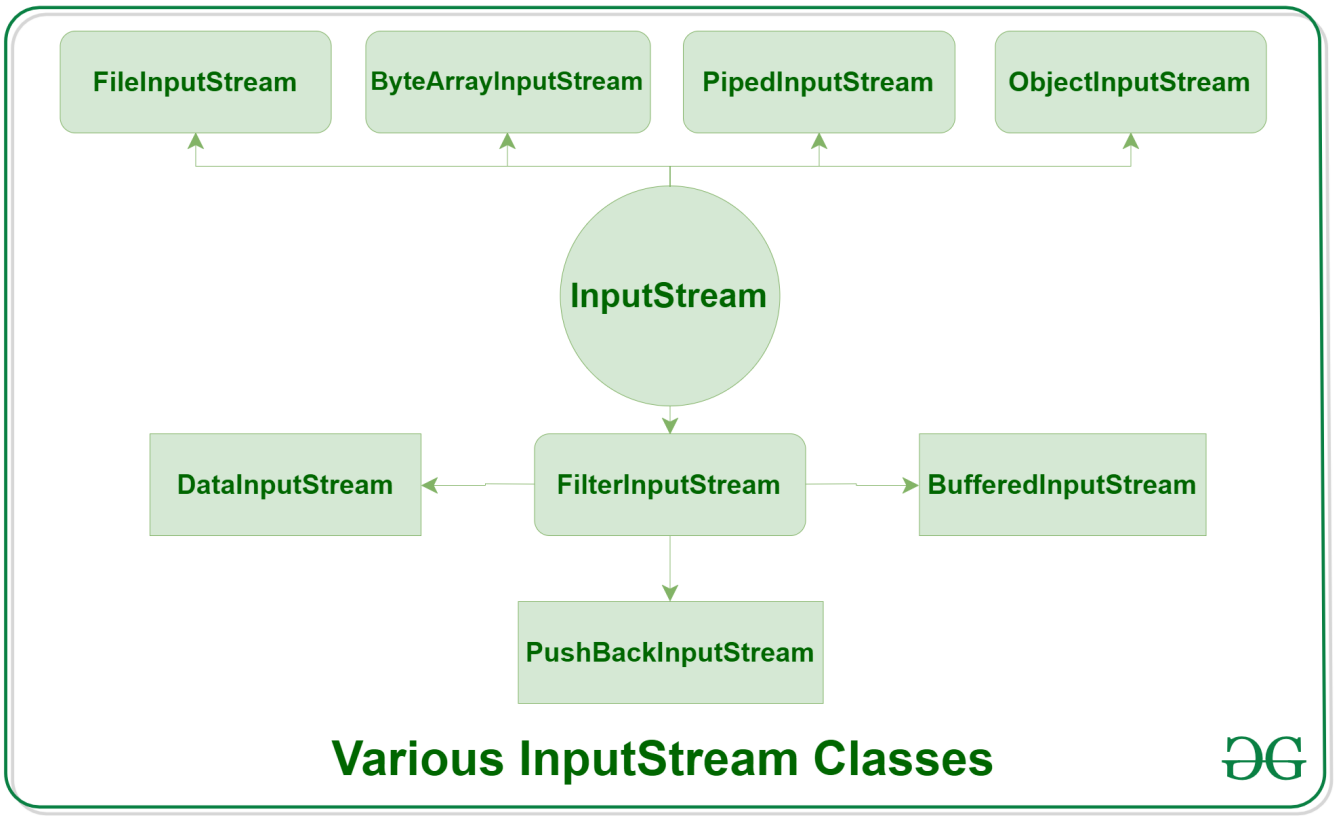
**s**



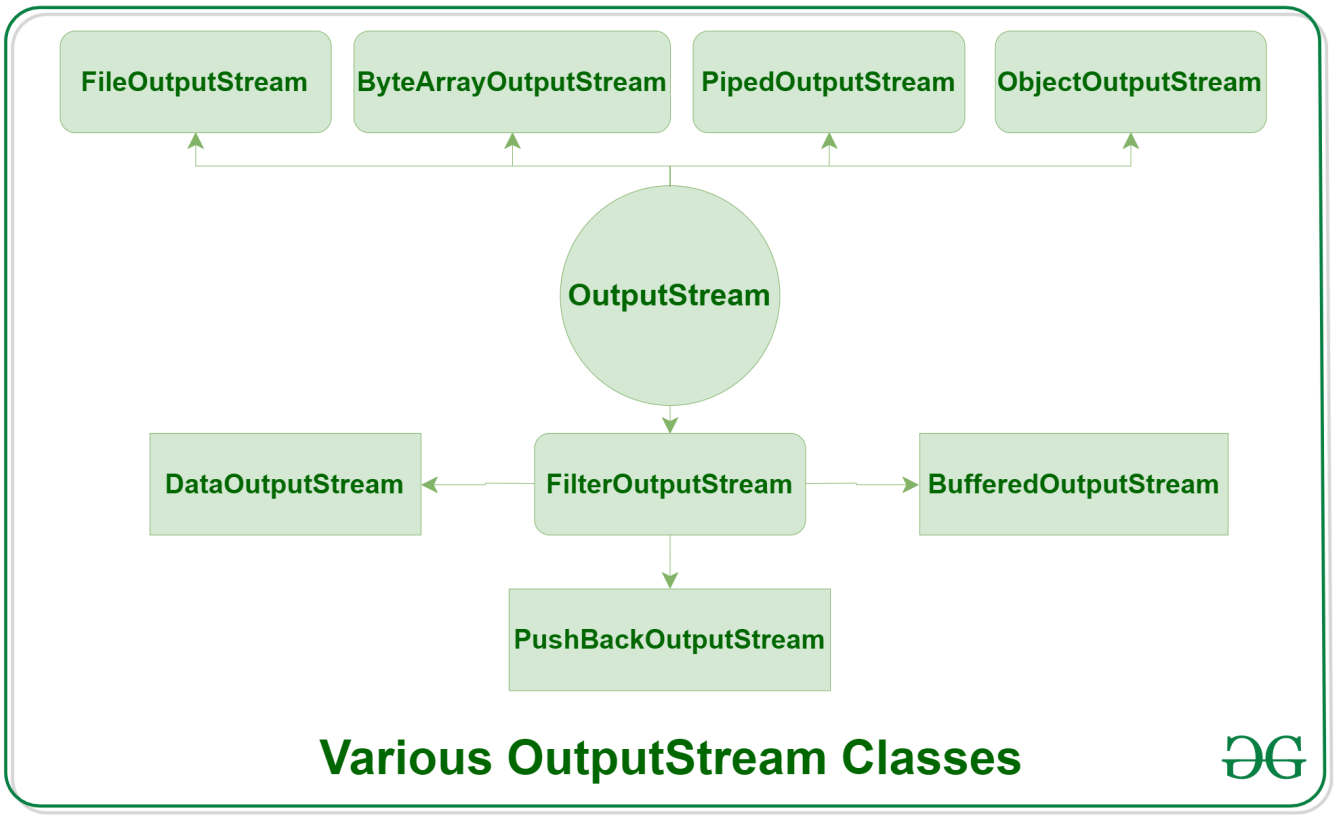
**Types of Streams:**

- Depending on the type of operations, streams can be divided into two primary classes:

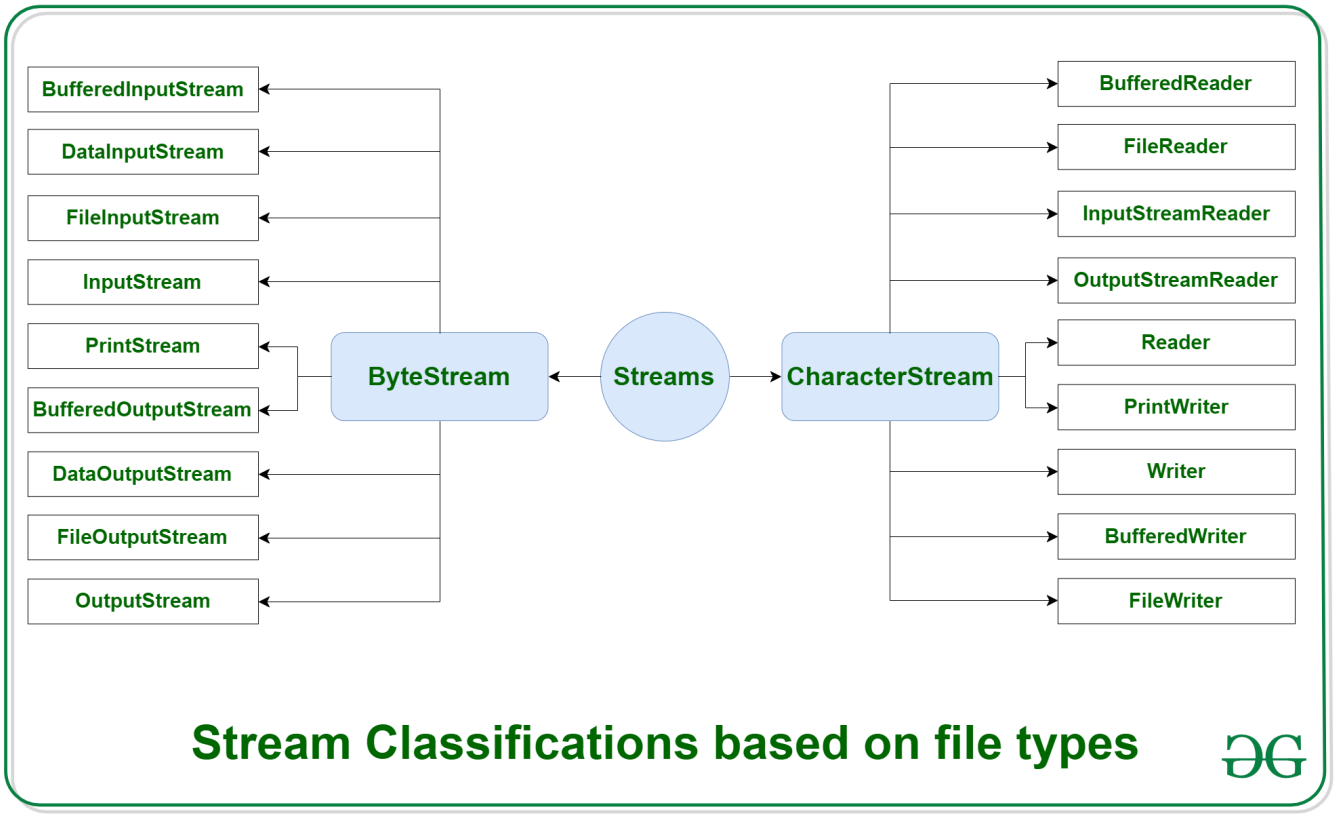
1.Input Stream: These streams are used to read data that must be taken as an input from a source array or file or any peripheral device. For eg., FileInputStream, BufferedInputStream, ByteArrayInputStream etc.




**Output Stream:**These streams are used to write data as outputs into an array or file or any output peripheral device. For eg., FileOutputStream, BufferedOutputStream, ByteArrayOutputStream etc.




**Depending on the types of file**, Streams can be divided into two primary classes which can be further divided into other classes as can be seen through the diagram below followed by the explanations.




**ByteStream:** This is used to process data byte by byte (8 bits). Though it has many classes, the FileInputStream and the FileOutputStream are the most popular ones. The FileInputStream is used to read from the source and FileOutputStream is used to write to the destination. Here is the list of various ByteStream Classes:

| STREAM CLASS | DESCRIPTION |
|---|---|
| BufferedInputStream | It is used for Buffered Input Stream. |
| DataInputStream | It contains method for reading java standard datatypes. |
| FileInputStream | This is used to reads from a file |
| InputStream | This is an abstract class that describes stream input. |
| PrintStream | This contains the most used print() and println() method |
| BufferedOutputStream | This is used for Buffered Output Stream. |
| DataOutputStream | This contains method for writing java standard data types. |
| FileOutputStream | This is used to write to a file. |
| OutputStream | This is an abstract class that describe stream output. |

## PACKAGES

### What is Package in Java?

Java package is a mechanism of grouping similar type of classes, interfaces, and sub-classes collectively based on functionality.

Using packages while coding offers a lot of advantages like:

- **Re-usability:**The classes contained in the packages of another program can be easily reused
- **Name Conflicts:**Packages help us to uniquely identify a class, for example, we can have company.sales.Employee and company.marketing.Employee classes
- **Controlled Access:** Offers access protection such as protected classes, default classes and private class
- **Data Encapsulation:**They provide a way to hi de classes, preventing other programs from accessing classes that are meant for internal use only
- **Maintainance:**With packages, you can organize your project better and easily locate related classes.

Elaborate on **Types of Packages in Java?**

Based on whether the package is defined by the user or not, packages are divided into two categories:

1.Built-in Packages

2.User Defined Packages

### Built-in Packages

Built-in packages or predefined packages are those that come along as a part of Java Development Kit to simplify the task of Java programmer. They consist of a huge number of predefined classes and interfaces that are a part of Java API's. Some of the commonly used built-in packages are java.lang, java.io, java.util, java.applet, etc. Here's a simple program using a built-in package.

```
package Edureka;

import java.util.ArrayList;

class BuiltInPackage {
```

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

        ArrayList<Integer> myList = new ArrayList<>(3);

        myList.add(3);
        myList.add(2);
        myList.add(1);

        System.out.println("The elements of list are: " + myList);
    }
}
```

**Output:**

The elements of list are: [3, 2, 1]

The ArrayList class belongs to java.util package. To use it, we have to import the package using the import statement. The first line of the code *import java.util.ArrayList* imports the java.util package and uses Array List Class which is present in the sub package util.

## User Defined Packages

User-defined packages are those which are developed by users in order to group related classes, interfaces and sub packages. With the help of an example program, let's see how to create packages, compile Java programs inside the packages and execute them.

## How to Create a Package in Java

Creating a package in Java is a very easy task. Choose a name for the package and include a package command as the first statement in the Java source file.

```
package MyPackage;
```

The package statement simply specifies to which package the classes defined belongs to..

## Including a Class in Java Package

To create a class inside a package, you should declare the package name as the first statement of your program. Then include the class as part of the package. But, remember that, a class can have only one package declaration. Here's a simple program to understand the concept

```
package MyPackage;


public class Compare {
  int num1, num2;

  Compare(int n, int m) {
     num1 = n;
     num2 = m;
}
public void getmax(){
    if ( num1 > num2 ) {
       System.out.println("Maximum value of two numbers is " + num1);
  }
    else {
       System.out.println("Maximum value of two numbers is " + num2);
    }
}

public static void main(String args[]) {

       Compare current[] = new Compare[3];

       current[1] = new Compare(5, 10);
       current[2] = new Compare(123, 120);
                for(int i=1; i < 3 ; i++)
            {
            current[i].getmax();
            }
       }
}
```

**Output:**

Maximum value of two numbers is 10

Maximum value of two numbers is 123

## How can you Create a class inside package while importing another package

Well, it's quite simple. You just need to import it. Once it is imported, you can access it by its name. Here's a sample program demonstrating the concept.

```
package Edureka;

import MyPackage.Compare;

public class Demo{
   public static void main(String args[]) {
      int n=10, m=10;
      Compare current = new Compare(n, m);
      if(n != m) {
          current.getmax();
      }
      else {
         System.out.println("Both the values are same");
      }
}
}
```

**Output:**

Both the values are same

## Using fully qualified name while importing a class

Here's an example to understand the concept. I am going to use the same package that I have declared earlier in the blog, *MyPackage*.

```
package Edureka;

public class Demo{
   public static void main(String args[]) {
      int n=10, m=11;
      //Using fully qualified name instead of import
      MyPackage.Compare current = new MyPackage.Compare(n, m);
      if(n != m) {
          current.getmax();
      }
      else {
         System.out.println("Both the values are same");
      }
}
}
```

**Output:**

Maximum value of two numbers is 11

**Static Import in Java**

Static import feature was introduced in java from version 5. It facilitates the Java programmer to access any static member of a class directly without using the fully qualified name

```
package MyPackage;

import static java.lang.Math.*; //static import
import static java.lang.System.*;// static import
public class StaticImportDemo {
      public static void main(String args[]) {
        double val = 64.0;
        double sqroot = sqrt(val); // Access sqrt() method directly
        out.println("Sq. root of " + val + " is " + sqroot);
        //We don't need to use 'System.out
      }
   }
```

**Output:**

Sq. root of 64.0 is 8.0

**Explain Access Protection in Java Packages**

You might be aware of various aspects of Java's access control mechanism and its access specifiers. Packages in Java add another dimension to access control. Both classes and packages are a means of data encapsulation. While packages act as containers for classes and other subordinate packages, classes act as containers for data and code. Because of this interplay between packages and classes, Java packages addresses four categories of visibility for class members:

- Sub-classes in the same package
- Non-subclasses in the same package
- Sub-classes in different packages
- Classes that are neither in the same package nor sub-classes

The table below gives a real picture of which type access is possible and which is not when using packages in Java:

| | Private | No Modifier | Protected | Public |
|---|---|---|---|---|
| **Same Class** | Yes | Yes | Yes | Yes |
| **Same Package Subclasses** | No | Yes | Yes | Yes |
| **Same Package Non-Subclasses** | No | Yes | Yes | Yes |
| **Different Packages Subclasses** | No | No | Yes | Yes |
| **Different Packages Non- Subclasses** | No | No | No | Yes |

# UNIT 5

# STRING HANDLING

### String Constructor

The String class supports several constructors. To create an empty String, you call the default constructor.

For example, String s = new String(); will create an instance of String with no characters in it. Frequently, you will want to create strings that have initial values. The String class provides a variety of constructors to handle this. To create a String initialized by an array of characters, use the constructor shown here:

String(char chars[ ])

Here is an example: char chars[] = { 'a', 'b', 'c' };

String s = new String(chars);

This constructor initializes s with the string “abc”. You can specify a subrange of a character array as an initializer using the following constructor:

String(char chars[ ], int startIndex, int numChars)

Here, startIndex specifies the index at which the subrange begins, and numChars specifies the number of characters to use.

Here is an example:

char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };

String s = new String(chars, 2, 3);

This initializes s with the characters cde. You can construct a String object that contains the same character sequence as another String object using this constructor:

String(String strObj)

Here, strObj is a String object. Consider this example:

```
// Construct one String from another.
 class MakeString
{
public static void main(String args[])
{
char c[] = {'J', 'a', 'v', 'a'};
```

```
String s1 = new String(c);
String s2 = new String(s1);
System.out.println(s1);
System.out.println(s2);
}
}
```

**The output from this program is as follows:**

**Java**

**Java**

As you can see, s1 and s2 contain the same string. Even though Java's char type uses 16 bits to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set.

Because 8-bit ASCII strings are common, the String class provides constructors that initialize a string when given a byte array.

Their forms are shown here:

String(byte asciiChars[ ])

String(byte asciiChars[ ],

int startIndex, int numChars)

Here, asciiChars specifies the array of bytes.

The second form allows you to specify a subrange. In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform.

The following program illustrates these constructors:

```
// Construct string from subset of char array.
class SubStringCons
{
public static void main(String args[])
{
byte ascii[] = {65, 66, 67, 68, 69, 70 };
String s1 = new String(ascii);
System.out.println(s1);
String s2 = new String(ascii, 2, 3);
```

```
System.out.println(s2);
}
}
```

**This program generates the following output:**

**ABCDEF**

**CDE**

## 5.2 String Length:

The length of a string is the number of characters that it contains. To obtain this value, call the length( ) method, shown here:

int length( )

The following fragment prints “3”, since there are three characters in the string s:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

## 5.3 Special String Operations:

The earlier examples showed how to explicitly create a String instance from an array of characters by using the new operator. However, there is an easier way to do this using a string literal. For each string literal in your program, Java automatically constructs a String object. Thus, you can use a string literal to initialize a String object.

For example, the following code fragment creates two equivalent strings:

```
char chars[] = { 'a', 'b', 'c' };
String s1 = new String(chars);
String s2 = "abc"; // use string literal
```

Because a String object is created for every string literal, you can use a string literal any place you can use a String object. For example, you can call methods directly on a quoted string as if it were an object reference, as the following statement shows. It calls the length( ) method on the string “abc”. As expected, it prints “3”.

```
System.out.println("abc".length());
```

- **String Concatenation**

In general, Java does not allow operators to be applied to String objects. The one exception to this rule is the + operator, which concatenates two strings, producing a String object as the result. This allows you to chain together a series of + operations.

For example, the following fragment concatenates three strings:

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

This displays the string "He is 9 years old."

One practical use of string concatenation is found when you are creating very long strings. Instead of letting long strings wrap around within your source code, you can break them into smaller pieces, using the + to concatenate them.

Here is an example:

```
// Using concatenation to prevent long lines.
class ConCat
{
public static void main(String args[])
{
String longStr = "This could have been " + "a very long line that would have " + "wrapped around. But string concatenation " + "prevents this.";
System.out.println(longStr);
}
}
```

### 5.4.1 String Concatenation with Other Data Types

You can concatenate strings with other types of data.

For example,

consider this slightly different version of the earlier example:

```
int age = 9;
String s = "He is " + age + " years old.";
```

System.out.println(s);

In this case, age is an int rather than another String, but the output produced is the same as before. This is because the int value in age is automatically converted into its string representation within a String object. This string is then concatenated as before. The compiler will convert an operand to its string equivalent whenever the other operand of the + is an instance of String.

Be careful when you mix other types of operations with string concatenation expressions, however. You might get surprising results. Consider the following:

String s = "four: " + 2 + 2;

System.out.println(s);

This fragment displays

four: 22

rather than the four:

4

that you probably expected.

Here's why. Operator precedence causes the concatenation of "four" with the string equivalent of 2 to take place first. This result is then concatenated with the string equivalent of 2 a second time. To complete the integer addition first, you must use parentheses, like this:

String s = "four: " + (2 + 2);

Now s contains the string "four: 4"

## 5.5 String Conversion and toString( )

Every class implements toString( ) because it is defined by Object. However, the default implementation of toString( ) is seldom sufficient. For most important classes that you create, you will want to override toString( ) and provide your own string representations

. Fortunately, this is easy to do.

The toString( ) method has this general form:

String toString( )

To implement toString( ), simply return a String object that contains the human-readable string that appropriately describes an object of your class.

By overriding toString( ) for classes that you create, you allow them to be fully integrated into Java's programming environment.

For example, they can be used in print( ) and println( ) statements and in concatenation expressions. The following program demonstrates this by overriding toString( ) for the Box class:

```
// Override toString() for Box class.
 class Box
{
 double width;
double height;
 double depth;
 Box(double w, double h, double d)
 {
width = w;
height = h;
 depth = d;
 }
 public String toString()
 {
 return "Dimensions are " + width + " by " +
depth + " by " + height + ".";
}
 }
 class toStringDemo { public static void main(String args[])
 {
 Box b = new Box(10, 12, 14);
 String s = "Box b: " + b;// concatenate Box object
 System.out.println(b); // convert Box to string
System.out.println(s);
 }
 }
```

**The output of this program is shown here:**

**Dimensions are 10.0 by 14.0 by 12.0**

**Box b: Dimensions are 10.0 by 14.0 by 12.0**

## 5.6 Character Extraction

The String class provides a number of ways in which characters can be extracted from a String object. Each is examined here. Although the characters that comprise a string within a String object cannot be indexed as if they were a character array, many of the String methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero.

### 5.6.1 charAt( )

To extract a single character from a String, you can refer directly to an individual character via the charAt( ) method. It has this general form:

char charAt(int where)

Here, where is the index of the character that you want to obtain. The value of where must be nonnegative and specify a location within the string. charAt( ) returns the character at the specified location.

For example,

```
char ch;
ch = "abc".charAt(1);
```

assigns the value “b” to ch.

### 5.6.2 getChars( )

If you need to extract more than one character at a time, you can use the getChars( ) method.

It has this general form:

void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)

Here, sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring. Thus, the substring contains C the characters from sourceStart through sourceEnd1. The array that will receive the characters – is specified by target. The index within target at which the substring will be copied is passed in targetStart. Care must be taken to assure that the target array is large enough to hold the number of characters in the specified substring. The following program demonstrates getChars( ):

class getCharsDemo

```
{
public static void main(String args[])
{
String s = "This is a demo of the getChars method.";
int start = 10;
int end = 14;
char buf[] = new char[end - start];
s.getChars(start, end, buf, 0);
System.out.println(buf);
}
}
```

**Here is the output of this program:**

**Demo**

### 5.6.3 **getBytes( )**

There is an alternative to getChars( ) that stores the characters in an array of bytes. This method is called getBytes( ), and it uses the default character-to-byte conversions provided by the platform.

Here is its simplest form:

byte[ ] getBytes( )

Other forms of getBytes( ) are also available. getBytes( ) is most useful when you are exporting a String value into an environment that does not support 16-bit Unicode characters. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

### 5.6.4 **toCharArray( )**

If you want to convert all the characters in a String object into a character array, the easiest way is to call toCharArray( ). It returns an array of characters for the entire string.

It has this general form:

char[ ] toCharArray( )

This function is provided as a convenience, since it is possible to use getChars( ) to achieve the same result.