Chapter One: Introduction to Java Programming

What is Java

- Java is a **programming language** and a **platform**.
- Java is a high level, robust, secured and object-oriented programming language.
- **Platform**: Any hardware or software environment in which a program runs, is known as a platform. Since Java has its own runtime environment (JRE) and API, it is called platform.

Where it is used?

According to Sun, 3 billion devices run java. There are many devices where java is currently used. Some of them are as follows:

1. Desktop Applications such as acrobat reader, media player, antivirus etc.
2. Web Applications such as irctc.co.in, javatpoint.com etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games etc.

Types of Java Applications

There are mainly 4 type of applications that can be created using java programming:

1) Standalone Application : It is also known as desktop application or window-based application. An application that we need to install on every machine such as media player, antivirus etc. AWT and Swing are used in java for creating standalone applications.

2) Web Application : An application that runs on the server side and creates dynamic page, is called web application. Currently, servlet, jsp, struts, jsf etc. technologies are used for creating web applications in java.

3) Enterprise Application : An application that is distributed in nature, such as banking applications etc. It has the advantage of high level security, load balancing and clustering. In java, EJB is used for creating enterprise applications.

4) Mobile Application : An application that is created for mobile devices. Currently Android and Java ME are used for creating mobile applications.

History of Java

Java history is interesting to know. The history of java starts from Green Team. Java team members (also known as **Green Team**), initiated a revolutionary task to develop a language for digital devices such as set-top boxes, televisions etc.For the green team members, it was an advance concept at that time. But, it was suited for internet programming. Later, Java technology as incorporated by Netscape.

Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc. There are given the major points that describes the history of java.

1) **James Gosling**, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.

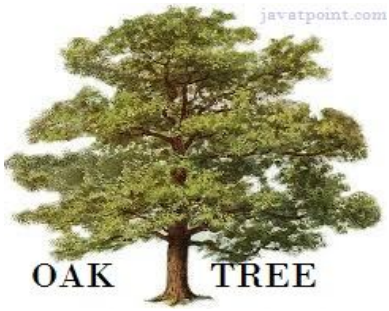
2) Originally designed for small, embedded systems in electronic appliances like set-top boxes.

3) Firstly, it was called **"Greentalk"** by James Gosling and file extension was .gt.

4) After that, it was called **Oak** and was developed as a part of the Green project.

Why Oak name for java language?

5) **Why Oak?** Oak is a symbol of strength and choosen as a national tree of many countries like U.S.A., France, Germany, Romania etc.



6) In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.

Why Java name for java language?

7) **Why they choosed java name for java language?** The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA" etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say. According to James Gosling "Java was one of the top choices along with **Silk**". Since java was so unique, most of the team members preferred java.

8) Java is an island of Indonesia where first coffee was produced (called java coffee).

9) Notice that Java is just a name not an acronym.

10) Originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.

11) In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.

12) JDK 1.0 released in(January 23, 1996).

Java Version History

There are many java versions that has been released. Current stable release of Java is Java SE 8.

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan, 1996)
3. JDK 1.1 (19th Feb, 1997)
4. J2SE 1.2 (8th Dec, 1998)
5. J2SE 1.3 (8th May, 2000)
6. J2SE 1.4 (6th Feb, 2002)
7. J2SE 5.0 (30th Sep, 2004)
8. Java SE 6 (11th Dec, 2006)
9. Java SE 7 (28th July, 2011)
10. Java SE 8 (18th March, 2014)

Features of Java

There is given many features of java. They are also known as java buzzwords. The Java Features given below are simple and easy to understand.

Simple : According to Sun, Java language is simple because syntax is based on C++ (so easier for programmers to learn it after C++). removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading etc.

No need to remove unreferenced objects because there is Automatic Garbage Collection in java.

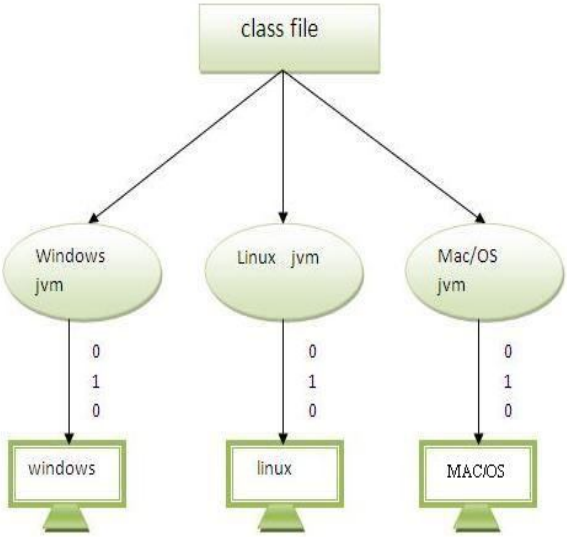
Object-oriented : Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behaviour. Object-oriented programming(OOPs) is a methodology that simplify software development and maintenance by providing some rules. Basic concepts of OOPs are:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

Platform Independent : A platform is the hardware or software environment in which a program runs. There are two types of platforms software-based and hardware-based. Java provides software-based platform. The Java platform differs from most other platforms in the sense that it's a software-based platform that runs on top of other hardware-based platforms.It has two components:

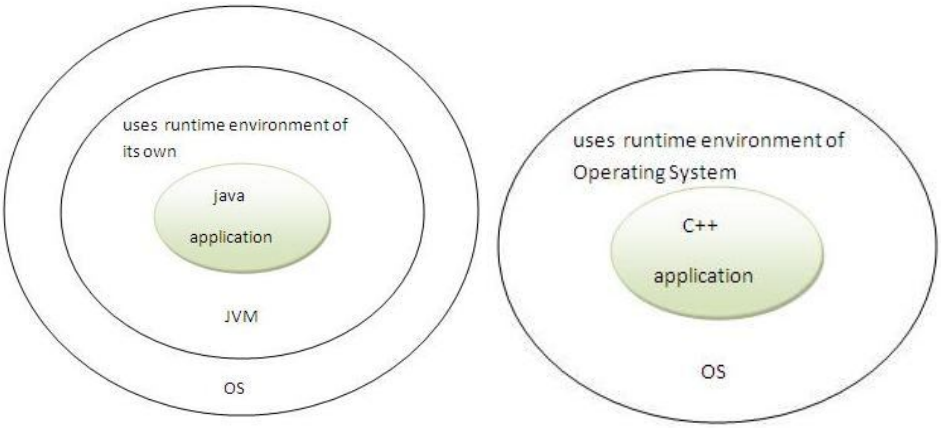
1. Runtime Environment
2. API(Application Programming Interface)

Java code can be run on multiple platforms e.g.Windows, Linux,Sun Solaris,Mac/OS etc. Java code is compiled by the compiler and converted into bytecode.This bytecode is a platform independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere(WORA).



Secured: Java is secured because:

- No explicit pointer
- Programs run inside virtual machine sandbox.



- **Classloader**- adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier**- checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager-** determines what resources a class can access such as reading and writing to the local disk.

These security are provided by java language. Some security can also be provided by application developer through SSL,JAAS,cryptography etc.

Multi-threaded : A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it shares the same memory. Threads are important for multi-media, Web applications etc.

Robust : Robust simply means strong. Java uses strong memory management. There are lack of pointers that avoids security problem. There is automatic garbage collection in java. There is exception handling and type checking mechanism in java. All these points makes java robust.

Architecture-neutral : There is no implementation dependent features e.g. size of primitive types is set.

Portable : We may carry the java bytecode to any platform.

High-performance : Java is faster than traditional interpretation since byte code is "close" to native code still somewhat slower than a compiled language (e.g., C++)

Distributed : We can create distributed applications in java. RMI and EJB are used for creating distributed applications. We may access files by calling the methods from any machine on the internet.

Simple Program of Java : In this page, we will learn how to write the simple program of java. We can write a simple hello java program easily after installing the JDK.

To create a simple java program, you need to create a class that contains main method. Let's understand the requirement first.

Requirement for Hello Java Example

For executing any java program, you need to

- install the JDK if you don't have installed it, download the JDK and install it.
- set path of the jdk/bin directory.
- create the java program
- compile and run the java program

Creating hello java example

Let's create the hello java program:

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

save this file as Simple.java

To compile:	javac Simple.java
To execute:	java Simple

Output: Hello Java

Understanding first java program

Let's see what is the meaning of class, public, static, void, main, String[], System.out.println().

- **class** keyword is used to declare a class in java.
- **public** keyword is an access modifier which represents visibility, it means it is visible to all.
- **static** is a keyword, if we declare any method as static, it is known as static method. The core advantage of static method is that there is no need to create object to invoke the

static method. The main method is executed by the JVM, so it doesn't require to create object to invoke the main method. So it saves memory.

- **void** is the return type of the method, it means it doesn't return any value.
- **main** represents startup of the program.
- **String[] args** is used for command line argument. We will learn it later.
- **System.out.println()** is used print statement. We will learn about the internal working of System.out.println statement later.

To write the simple program, open notepad by **start menu -> All Programs -> Accessories -> notepad** and write simple program as displayed below:

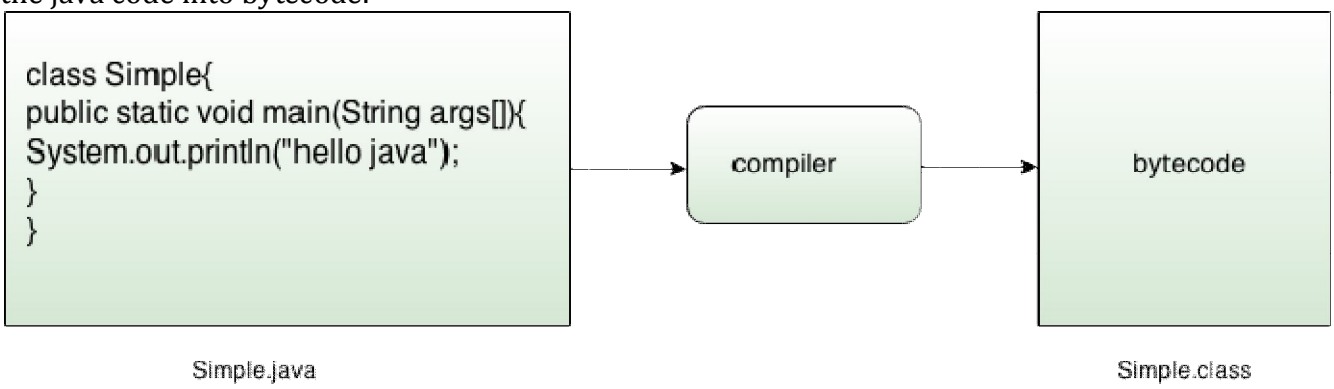
Internal Details of Hello Java Program

1. Internal Details of Hello Java

In the previous page, we have learned about the first program, how to compile and how to run the first java program. Here, we are going to learn, what happens while compiling and running the java program. Moreover, we will see some question based on the first program.

What happens at compile time?

At compile time, java file is compiled by Java Compiler (It does not interact with OS) and converts the java code into bytecode.



How to set path in Java

The path is required to be set for using tools such as javac, java etc.

If you are saving the java source file inside the jdk/bin directory, path is not required to be set because all the tools will be available in the current directory.

But If you are having your java file outside the jdk/bin folder, it is necessary to set path of JDK.

There are 2 ways to set java path:

1. **temporary**
2. **permanent**

1) How to set Temporary Path of JDK in Windows

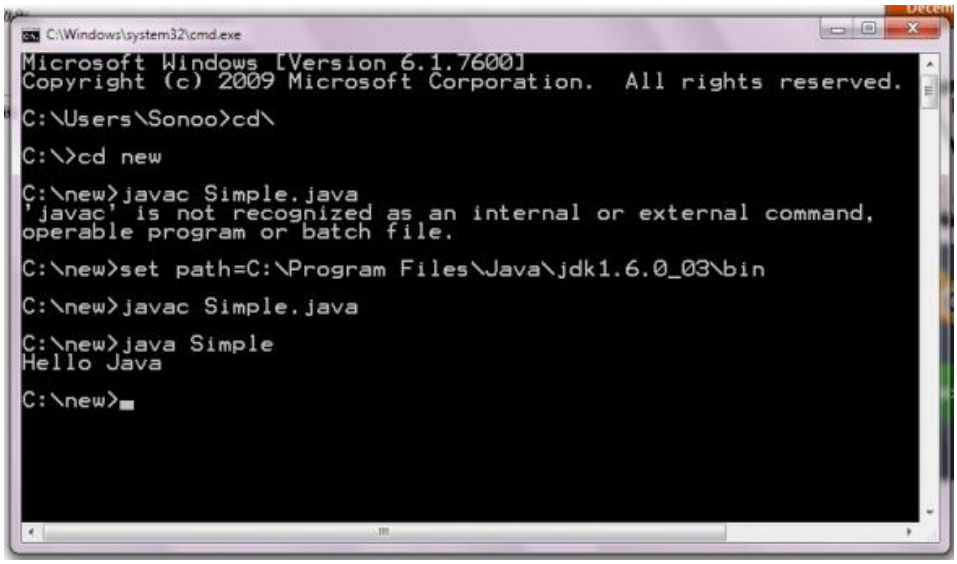
To set the temporary path of JDK, you need to follow following steps:

- Open command prompt
- copy the path of jdk/bin directory
- write in command prompt: set path=copied_path

For Example:

set path=C:\Program Files\Java\jdk1.6.0_23\bin

Let's see it in the figure given below:



2) How to set Permanent Path of JDK in Windows

For setting the permanent path of JDK, you need to follow these steps:

- Go to MyComputer properties -> advanced tab -> environment variables -> new tab of user variable -> write path in variable name -> write path of bin folder in variable value -> ok -> ok -> ok

Difference between JDK, JRE and JVM

Understanding the difference between JDK, JRE and JVM is important in Java. We are having brief overview of JVM here. If you want to get the detailed knowledge of Java Virtural Machine, move to the next page. Firstly, let's see the basic differences between the JDK, JRE and JVM.

JVM

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms. JVM, JRE and JDK are platform dependent because configuration of each OS differs. But, Java is platform independent.

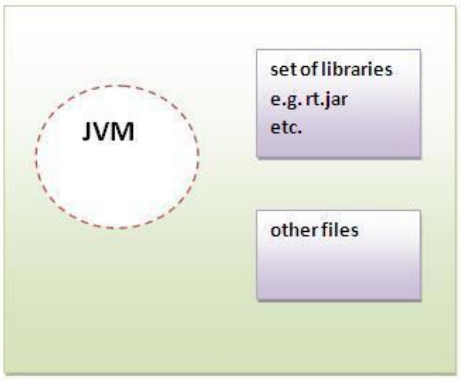
The JVM performs following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JRE

JRE is an acronym for Java Runtime Environment.It is used to provide runtime environment.It is the implementation of JVM.It physically exists.It contains set of libraries + other files that JVM uses at runtime.

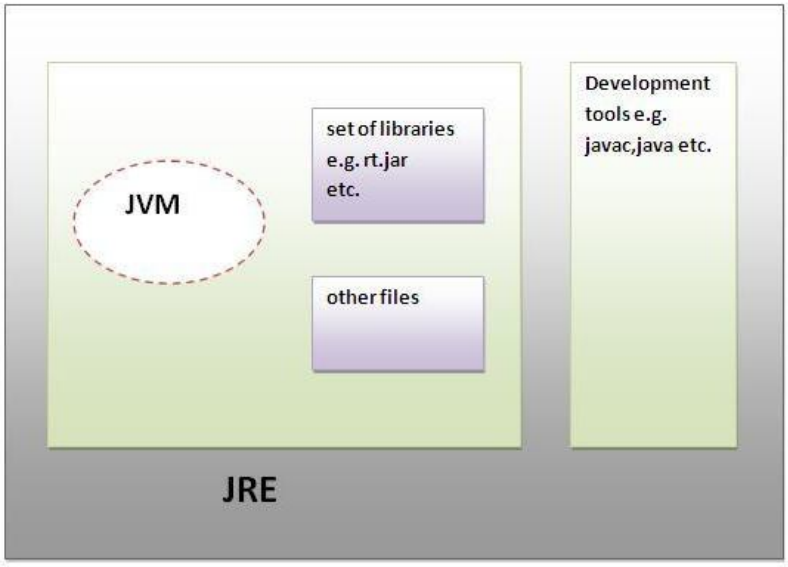
Implementation of JVMs are also actively released by other companies besides Sun Micro Systems.



JRE

JDK

JDK is an acronym for Java Development Kit.It physically exists.It contains JRE + development tools.



JDK

JVM (Java Virtual Machine) : JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms (i.e.JVM is plateform dependent).

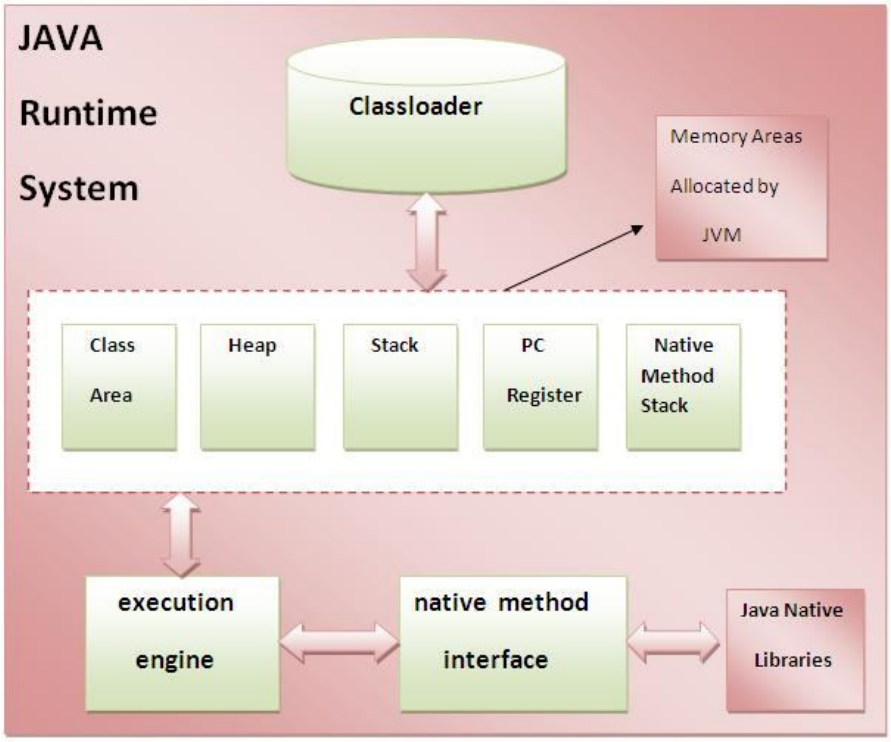
What is JVM?

It is:

1. **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.
2. **An implementation** Its implementation is known as JRE (Java Runtime Environment).
3. **Runtime Instance** Whenever you write java command on the command prompt to run the java class, and instance of JVM is created.

Internal Architecture of JVM

Let's understand the internal architecture of JVM. It contains classloader, memory area, execution engine etc.



1) Classloader: **Classloader is a subsystem of JVM that is used to load class files.**

2) Class(Method) Area: **Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.**

3) Heap:**It is the runtime data area in which objects are allocated.**

4) Stack: Java Stack stores frames.It holds local variables and partial results, and plays a part in method invocation and return.

5) Program Counter Register: **PC (program counter) register. It contains the address of the Java virtual machine instruction currently being executed.**

6) Native Method Stack: **It contains all the native methods used in the application.**

7) Execution Engine: It contains:

1) A virtual processor

2) Interpreter: Read bytecode stream then execute the instructions.

3) Just-In-Time(JIT) compiler: It is used to improve the performance.JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation.Here the term ?compiler? refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

JDK Basic Tools

These tools are the foundation of the JDK. They are the tools you use to create and build applications.

Tool Name	Brief Description
javac	The compiler for the Java programming language.
java	The launcher for Java applications. In this release, a single launcher is used both for development and deployment. The old deployment launcher, jre, is no longer provided.
javadoc	API documentation generator.
apt	Annotation processing tool.
appletviewer	Run and debug applets without a web browser.
jar	Create and manage Java Archive (JAR) files.
jdb	The Java Debugger.
javah	C header and stub generator. Used to write native methods.
javap	Class file disassembler

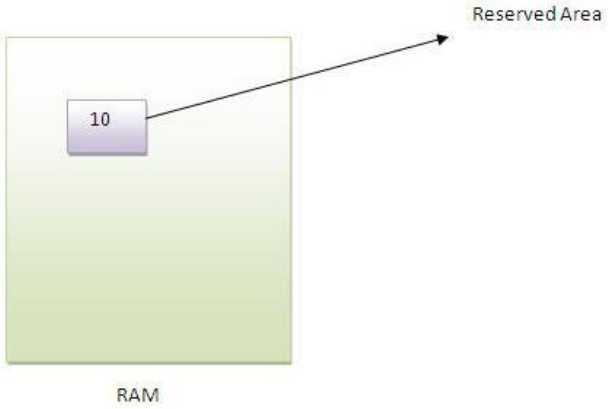
Chapter Two : Java Programming Fundaments

Variable and Datatype in Java

In this page, we will learn about the variable and java data types. Variable is a name of memory location. There are three types of variables: local, instance and static. There are two types of datatypes in java, primitive and non-primitive.

Variable

Variable is name of reserved area allocated in memory.

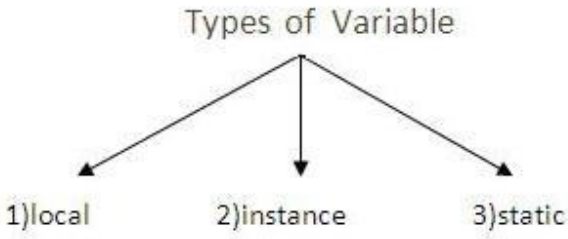


1. `int data=50;//Here data is variable`

Types of Variable

There are three types of variables in java

- **local variable**
- **instance variable**
- **static variable**



Local Variable

A variable that is declared inside the method is called local variable.

Instance Variable

A variable that is declared inside the class but outside the method is called instance variable .

It is not declared as static.

Static variable

A variable that is declared as static is called static variable. It cannot be local.

We will have detailed learning of these variables in next chapters.

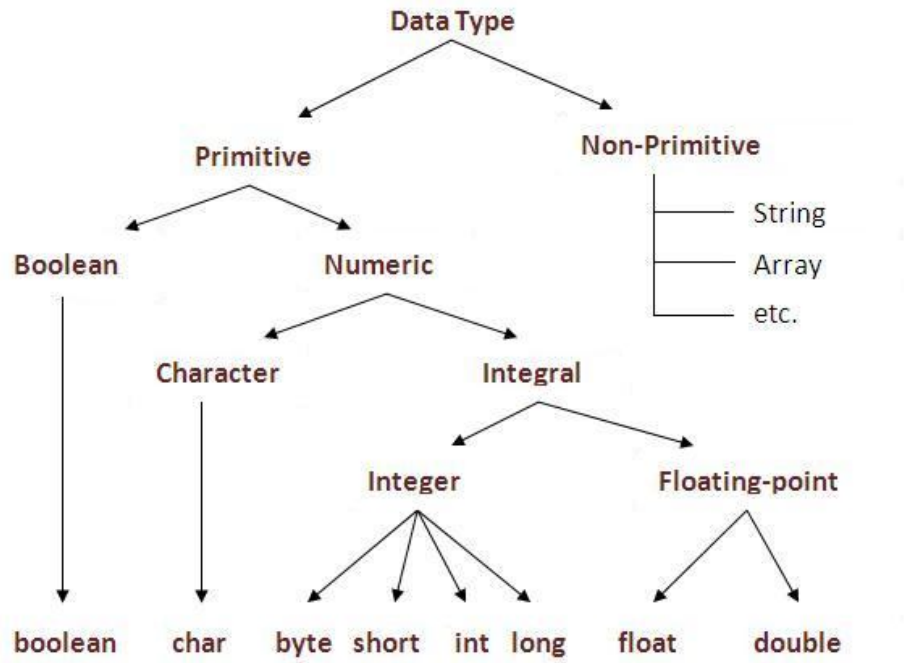
Example to understand the types of variables

```
class A
{
    int data=50;//instance variable
    static int m=100;//static variable
    void method(){
    int n=90;//local variable
}
}//end of class
```

Data Types in Java

In java, there are two types of data types

- primitive data types
- non-primitive data types



Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Operators in java

Operator in java is a symbol that is used to perform operations. There are many types of operators in java such as unary operator, arithmetic operator, relational operator, shift operator, bitwise operator, ternary operator and assignment operator.

Operators	Precedence
postfix	*expr*++ *expr*--
unary	++*expr* --*expr* +*expr* -*expr* ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Java Programs

Java programs are frequently asked in the interview. These programs can be asked from control statements, array, string, oops etc. Let's see the list of java programs.

1) Fibonacci series in Java

In fibonacci series, *next number is the sum of previous two numbers* for example 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 etc. The first two numbers of fibonacci series are 0 and 1.

There are two ways to write the fibonacci series program in java:

- Fibonacci Series without using recursion
- Fibonacci Series using recursion

Fibonacci Series in Java without using recursion

Let's see the fibonacci series program in java without using recursion.

```
class FibonacciExample1{
public static void main(String args[])
{
int n1=0,n2=1,n3,i,count=10;
System.out.print(n1+" "+n2);//printing 0 and 1

for(i=2;i<count;++i)//loop starts from 2 because 0 and 1 are already printed
{
n3=n1+n2;
System.out.print(" "+n3);
n1=n2;
n2=n3;
}

}}
```

Output:

```
0 1 1 2 3 5 8 13 21 34
```

2) Prime number

Prime Number Program in Java

Prime number in Java: **Prime number** is a number that is greater than 1 and divided by 1 or itself. In other words, prime numbers can't be divided by other numbers than itself or 1. For example 2, 3, 5, 7, 11, 13, 17.... are the prime numbers.

Note: 0 and 1 are not prime numbers. 2 is the only even prime number because all the numbers can be divided by 2.

Let's see the prime number program in java. In this java program, we will take a number variable and check whether the number is prime or not.

```
1. class PrimeExample{
        1.  public static void main(String args[]){
        2.   int i,m=0,flag=0;
        3.   int n=17;//it is the number to be checked
        4.   m=n/2;
        5.   for(i=2;i<=m;i++){
        6.    if(n%i==0){
        7.    System.out.println("Number is not prime");
        8.    flag=1;
        9.    break;
        10.   }
        11.  }
        12.  if(flag==0)
        13.  System.out.println("Number is prime");
        14. }
        15. }
```

Output:

```
Number is prime
```

3) Palindrome number

Palindrome Program in Java

Palindrome number in java: A **palindrome number** is *a number that is same after reverse*. For example 545, 151, 34543, 343, 171, 48984 are the palindrome numbers.

Palindrome number algorithm

- Get the number to check for palindrome
- Hold the number in temporary variable
- Reverse the number
- Compare the temporary number with reversed number
- If both numbers are same, print "palindrome number"
- Else print "not palindrome number"

Let's see the palindrome program in java. In this java program, we will get a number variable and check whether number is palindrome or not.

```
class PalindromeExample{
1.  public static void main(String args[]){
2.   int r,sum=0,temp;
3.   int n=454;//It is the number variable to be checked for palindrome
4.
5.   temp=n;
6.   while(n>0){
7.    r=n%10;  //getting remainder
8.    sum=(sum*10)+r;
9.    n=n/10;
10.  }
11.  if(temp==sum)
12.   System.out.println("palindrome number ");
13.  else
14.   System.out.println("not palindrome");
15. }
16. }
```

Output: palindrome number

4) Factorial number

Factorial Program in Java

Factorial Program in Java: Factorial of n is the *product of all positive descending integers*.

Factorial of *n* is denoted by n!. For example:

1. 4! = 4*3*2*1 = 24
2. 5! = 5*4*3*2*1 = 120

Here, 4! is pronounced as "4 factorial", it is also called "4 bang" or "4 shriek".

The factorial is normally used in Combinations and Permutations (mathematics).

There are many ways to write the factorial program in java language. Let's see the 2 ways to write the factorial program in java.

- Factorial Program using loop
- Factorial Program using recursion

Factorial Program using loop in java

Let's see the factorial Program using loop in java.

```
1.  class FactorialExample{
2.  public static void main(String args[]){
3.  int i,fact=1;
4.  int number=5;//It is the number to calculate factorial
5.  for(i=1;i<=number;i++){
6.  fact=fact*i;
7.  }
8.  System.out.println("Factorial of "+number+" is: "+fact);   } }
```

Armstrong Number in Java

Armstrong Number in Java: **Armstrong number** is *a number that is equal to the sum of cubes of its digits* for example 0, 1, 153, 370, 371, 407 etc.

Let's try to understand why **153** is an Armstrong number.

1. 153 = (1*1*1)+(5*5*5)+(3*3*3)
2. where:
3. (1*1*1)=1
4. (5*5*5)=125
5. (3*3*3)=27
6. So:
7. 1+125+27=153

Let's try to understand why **371** is an Armstrong number.

1. 371 = (3*3*3)+(7*7*7)+(1*1*1)
2. where:
3. (3*3*3)=27
4. (7*7*7)=343
5. (1*1*1)=1
6. So:
7. 27+343+1=371

Let's see the java program to check Armstrong Number.

```
1.  class ArmstrongExample{
2.   public static void main(String[] args)  {
3.    int c=0,a,temp;
4.    int n=153;//It is the number to check armstrong
5.    temp=n;
6.    while(n>0)
7.    {
8.    a=n%10;
9.    n=n/10;
10.   c=c+(a*a*a);
11.   }
12.   if(temp==c)
13.   System.out.println("armstrong number");
14.   else
15.      System.out.println("Not armstrong number");
16.  }
17. }
```

Java Keyword

There are few keywords in Java programming language. Remember, we cannot use these keywords as identifiers in the program. The keywords const and goto are reserved though; they are not being currently used.

The brief description of each one of the keyword is given below.

abstract: When a class is not to be instantiated, use abstract keyword but rather extended by other classes. This keyword is used to in a method declaration to declare a method without providing the implementation.

assert: To make the assumed value of a condition explicit, assert keyword is used. An AssertionError is thrown if the condition is not true.

boolean: This keyword is used to pertain to an expression or variable that can have only a true or false value.

byte: This is an an 8-bit integer. This keyword is used to declare an expression, method return value, or variable of type byte.

case: This keyword is used to defines a group of statements. The value defined by the enclosing switch statement should match with the specified value.

catch: This keyword is used to handle the exceptions that occur in a program preceding try keyword. When the class of the thrown exception is assignment compatible with the exception class declared by the catch clause then only the code is executed.

char: This Java keyword is used to declare an expression, method return value, or variable of type character.

class: This keyword is used to define the implementation of a particular kind of object. const: This keyword has been deprecated from Java programming language.

continue: This keyword is used for the continuation of the program at the end of the current loop body.

default: If the value defined by the enclosing switch statement does not match any value specified by a

case keyword in the switch statement, default keyword is used to define a group of statements to begin the execution.

do: Used to declare a loop that will iterate a block of statements. The loop's exit condition is specified with the while keyword. The loop will execute once before evaluating the exit condition.
double: A 64-bit floating point value. A Java keyword used to declare an expression, method return value, or variable of type double-precision floating point number.
else: This keyword is used to test the condition. It is used to define a statement or block of statements that are executed in the case that the test condition specified by the if keyword evaluates to false.

enum: Enumerations extend the base class Enum.This Java keyword is used to declare an enumerated
type.
extends: To specify the superclass in a class declaration, extends keyword is used. It is also used in an interface declaration to specify one or more superinterfaces.
final: It is used to define an entity once that cannot be altered nor inherited later. Moreover, a final class cannot be subclassed, a final method cannot be overridden, and a final variable can occur at most once as a left-hand expression. All methods in a final class are implicitly final.
finally: This keyword is used when the finally block is executed after the execution exits the try block and any associated catch clauses regardless of whether an exception was thrown or caught.
break: Used to resume program execution at the statement immediately following the current enclosing block or statement. If followed by a label, the program resumes execution at the statement immediately following the enclosing labeled statement or block.

Java Naming conventions:

Java **naming convention** is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method etc.
But, it is not forced to follow. So, it is known as convention not rule.
All the classes, interfaces, packages, methods and fields of java programming language are given according to java naming convention.

Advantage of naming conventions in java

By using standard Java naming conventions, you make your code easier to read for yourself and for other programmers. Readability of Java program is very important. It indicates that**less time** is spent to figure out what the code does.

Name	Convention
class name	should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc.
interface name	should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc.
method name	should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc.
variable name	should start with lowercase letter e.g. firstName, orderNumber etc.
package name	should be in lowercase letter e.g. java, lang, sql, util etc.
constants name	should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY etc.

Standard Java Naming Conventions

The below list outlines the standard Java naming conventions for each identifier type:

Packages: Names should be in lowercase. With small projects that only have a few packages it's okay to just give them simple (but meaningful!) names:

```
package sunilcoreproject
package suniladvancedproject
```

In software companies and large projects where the packages might be imported into other classes, the names will normally be subdivided. Typically this will start with the company domain before being split into layers or features:

```
package com.sunilcom.utilities
```

Classes: Names should be in SP3112. Try to use nouns because a class is normally representing something in the real world:

```
class Customer
```

class Account

Interfaces: Names should be in SP3112. They tend to have a name that describes an operation that a class can do:

interface SP3112(Comparable)
interface SP3112(Enumerable)

Note that some programmers like to distinguish interfaces by beginning the name with an "I":

interface ISP3112(Comparable)
interface I SP3112(Enumerable)

Methods: Names should be in mixed case. Use verbs to describe what the method does:

```
void calculateTax()
string getSurname()
```

Variables: Names should be in mixed case. The names should represent what the value of the variable represents:

```
string firstName
int orderNumber
```

Only use very short names when the variables are short lived, such as in for loops:

```
for (int i=0; i<20;i++)
{
  //i only lives in here
}
```

Constants: Names should be in uppercase.

```
static final int DEFAULT_WIDTH
static final int MAX_HEIGHT
```

Structure of Java Program.

```
/**
This is a sample java program save his file as SP3112.java
*/
//importing user packages if user want to use

packages sunil.*;
```

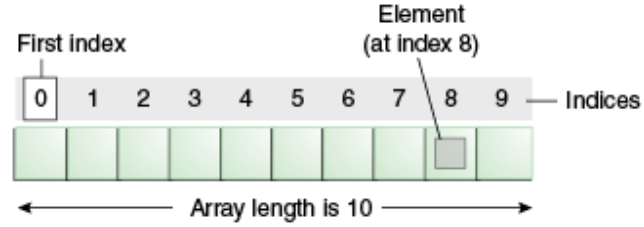
```
//importing java default packages
import java.lang.*;

//importig default and specific package

                    import java.lang.Scanner;


//writing class name it should be capital because all keyword in java staring with capital is known as class in java
public class SP3112

{

//  A java program will start from here.
// jvm machine need this method to run program and all properties of user define class we access in this method with class name or class object

public static void main(String args[])

{
        System.out.println("  Welcome to Java-Sample  sunil ……..!!! ");
}

}       // ending of java programming
```

Java Array

Normally, array is a collection of similar type of elements that have contiguous memory location.

Java array is an object the contains elements of similar data type. It is a data structure where we store similar elements. We can store only fixed set of elements in a java array.

Array in java is index based, first element of the array is stored at 0 index.



Advantage of Java Array

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
- **Random access:** We can get any data located at any index position.

Disadvantage of Java Array

- **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in java

Syntax to Declare an Array in java

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

Instantiation of an Array in java

1. arrayRefVar=**new** datatype[size];

Example of single dimensional java array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

```
1. class Testarray{
2. public static void main(String args[]){

3. int a[]=new int[5];//declaration and instantiation
4. a[0]=10;//initialization
5. a[1]=20;
6. a[2]=70;
7. a[3]=40;
8. a[4]=50;

9. //printing array
10. for(int i=0;i<a.length;i++)//length is the property of array
11. System.out.println(a[i]);

12. }}
```

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

1. `int a[]={33,3,4,5};//declaration, instantiation and initialization`
2. Let's see the simple example to print this array.

```
3. class Testarray1{
4. public static void main(String args[]){

5. int a[]={33,3,4,5};//declaration, instantiation and initialization

6. //printing array
7. for(int i=0;i<a.length;i++)//length is the property of array
8. System.out.println(a[i]);

9. }}
```

Passing Array to method in java

We can pass the java array to method so that we can reuse the same logic on any array.

Let's see the simple example to get minimum number of an array using method.

```
1. class Testarray2{
2. static void min(int arr[]){
3. int min=arr[0];
4. for(int i=1;i<arr.length;i++)
5. if(min>arr[i])
6. min=arr[i];

7. System.out.println(min);
8. }

9. public static void main(String args[]){

10. int a[]={33,3,4,5};
11. min(a);//passing array to method

12. }}
```

Multidimensional array in java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in java

```
1. dataType[][] arrayRefVar; (or)
2. dataType [][]arrayRefVar; (or)
3. dataType arrayRefVar[][]; (or)
4. dataType []arrayRefVar[];
```

5. Example to instantiate Multidimensional Array in java

```
6. int[][] arr=new int[3][3];//3 row and 3 column
```

```
7. Example to initialize Multidimensional Array in java
8. arr[0][0]=1;
9. arr[0][1]=2;
10. arr[0][2]=3;
11. arr[1][0]=4;
12. arr[1][1]=5;
13. arr[1][2]=6;
14. arr[2][0]=7;
15. arr[2][1]=8;
16. arr[2][2]=9;
```

Example of Multidimensional java array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
1. class Testarray3{
2. public static void main(String args[]){

3. //declaring and initializing 2D array
4. int arr[][]={{1,2,3},{2,4,5},{4,4,5}};

5. //printing 2D array
6. for(int i=0;i<3;i++){
7. for(int j=0;j<3;j++){
8. System.out.print(arr[i][j]+" ");
9. }
10. System.out.println();
11. }

12. }}
```

Copying a java array

We can copy an array to another by the arraycopy method of System class.

Syntax of arraycopy method

```
1. public static void arraycopy(
2. Object src, int srcPos,Object dest, int destPos, int length
3. )
```

```
4. Example of arraycopy method
5. class TestArrayCopyDemo {
6. public static void main(String[] args) {
7. char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
    a. 'i', 'n', 'a', 't', 'e', 'd' };
8. char[] copyTo = new char[7];

9. System.arraycopy(copyFrom, 2, copyTo, 0, 7);
10. System.out.println(new String(copyTo));
```

```
11. }
12. }
```

Addition of 2 matrices in java

Let's see a simple example that adds two matrices.

```
1. class Testarray5{
2. public static void main(String args[]){
3. //creating two matrices
4. int a[][]={{1,3,4},{3,4,5}};
5. int b[][]={{1,3,4},{3,4,5}};

6. //creating another matrix to store the sum of two matrices
7. int c[][]=new int[2][3];

8. //adding and printing addition of 2 matrices
9. for(int i=0;i<2;i++){
10. for(int j=0;j<3;j++){
11. c[i][j]=a[i][j]+b[i][j];
12. System.out.print(c[i][j]+" ");
13. }
14. System.out.println();//new line
15. }

16. }}
```

Java String

Java String provides a lot of concepts that can be performed on a string such as compare, concat, equals, split, length, replace, compareTo, intern, substring etc.

In java, string is basically an object that represents sequence of char values.

An array of characters works same as java string. For example:

```
1. char[] ch={'j','a','v','a','t','p','o','i','n','t'};
2. String s=new String(ch);
```

is same as:

```
1. String s="javatpoint";
```

The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence*interfaces.

The java String is immutable i.e. it cannot be changed but a new instance is created. For mutable class, you can use StringBuffer and StringBuilder class.

We will discuss about immutable string later. Let's first understand what is string in java and how to create the string object.

What is String in java

Generally, string is a sequence of characters. But in java, string is an object that represents a sequence of characters. String class is used to create string object.

How to create String object?

There are two ways to create String object:

1. By string literal
2. By new keyword

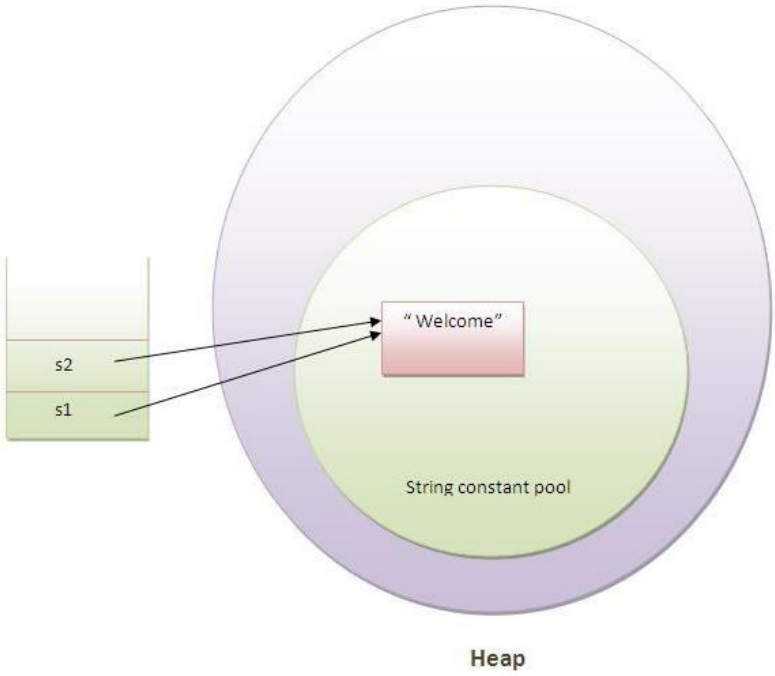
1) String Literal

Java String literal is created by using double quotes. For Example:

```
1. String s="welcome";
```

Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance is returned. If string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

```
1. String s1="Welcome";
2. String s2="Welcome";//will not create new instance
```



In the above example only one object will be created. Firstly JVM will not find any string object with the value "Welcome" in string constant pool, so it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create new object but will return the reference to the same instance.

Note: String objects are stored in a special memory area known as string constant pool.

Java String Example

```
1. public class StringExample{
2. public static void main(String args[]){
3. String s1="java";//creating string by java string literal

4. char ch[]={'s','t','r','i','n','g','s'};
5. String s2=new String(ch);//converting char array to string

6. String s3=new String("example");//creating java string by new keyword

7. System.out.println(s1);
8. System.out.println(s2);
9. System.out.println(s3);
10. }}
```

Java Control Statements

Java supports different types of control statements: the decision making statements (if-then, if-then-else and switch), looping statements (while, do-while and for) and branching statements (break, continue and return).

Control Statements

The control statement are used to control the flow of execution of the program. This execution order depends on the supplied data values and the conditional logic. Java contains the following types of control statements:

A.Selection Statements
B- Repetition Statements
C- Branching Statements

A] Selection statements:

1. If Statement:

This is a control statement to execute a single statement or a block of code, when the given condition is true and if it is false then it skips if block and rest code of program is executed .

Syntax:

```
if(conditional_expression){
<statements>;
...;          }
```

Example: If n%2 evaluates to 0 then the "if" block is executed. Here it evaluates to 0 so if block is executed. Hence "This is even number" is printed on the screen.

```
int n = 10;

if(n%2 = = 0){

    System.out.pri
ntln("This        is
even number");}
```

2.**If-else Statement:**

The "if-else" statement is an extension of if statement that provides another option when 'if' statement evaluates to "false" i.e. else block is executed if "if" statement is false.

```
Syntax:
  if(conditional_expression){
      <statements>;

      ...;
   }
  else{
      <statements>;
        .;      }
```

Example: If n%2 doesn't evaluate to 0 then else block is executed. Here n%2 evaluates to 1 that is not equal to 0 so else block is executed. So "This is not even number" is printed on the screen.

```
int n = 11;

if(n%2 = = 0){
    System.out.println("This
is even number"); }else{

    System.out.println("This is not even number"); }
```

3. Switch Statement:

This is an easier implementation to the if-else statements. The keyword "switch" is followed by an expression that should evaluates to byte, short, char or int primitive data types ,only. In a switch block there can be one or more labeled cases. The expression that creates labels for the case must be unique. The switch expression is matched with each case label. Only the matched case is executed ,if no case matches the the default statement (if present) is executed.

```
Syntax:
   switch(control_expression){
      case expression 1:
         <statement>;
      case expression 2:
      <statement>;

      case expression n:
      <statement>;
      default:
         <statement>;
   }//end switch
```

Example: Here expression "day" in switch statement evaluates to 5 which matches with a case labeled "5" so code in case 5 is executed that results to output "Friday" on the screen.

B] Repetition Statements:

1. while loop statements:

This is a looping or repeating statement. It executes a block of code or statements till the given condition is true. The expression must be evaluated to a boolean value. It continues testing the condition and executes the block of code. When the expression results to false control comes out of loop.

```
Syntax:
  while(expression){
      <statement>;
      ...;
      ...;    }
```

Example: Here expression i<=10 is the condition which is checked before entering into the loop statements. When i is greater than value 10 control comes out of loop and next statement is executed. So here i contains value "1" which is less than number "10" so control goes inside of the

loop and prints current value of i and increments value of i. Now again control comes back to the loop and condition is checked. This procedure continues until i becomes greater than value "10". So this loop prints values 1 to 10 on the screen.

```
int i = 1;

//print 1 to 10

while (i <= 10){
   System.out.pr
   intln("Num "
   + i);
   i++;}
```

2.do-while loop statements:

This is another looping statement that tests the given condition past so you can saythat the dowhile looping statement is a past-test loop statement. First the do block statements are executed then the condition given in while statement is checked. So in this case, even the condition is false in the first attempt, do block of code is executed at least once.

```
Syntax:
   do{
      <statement>;
         ...;

   }while (expression);
```

Example: Here first do block of code is executed and current value "1" is printed then the condition i<=10 is checked. Here "1" is less than number "10" so the control comes back to do block. This process continues till value of i becomes greater than 10.

```
int i = 1;
do{
System.out.println("N
um: " + i);
    i++;
}while(i <= 10);
```

3. for loop statements:

This is also a loop statement that provides a compact way to iterate over a range of values. From a user point of view, this is reliable because it executes the statements within this block repeatedly till the specified conditions is true .

Syntax:

```
for (initialization; condition;
increment or decrement){
<statement>;
   ...;
        }
```

initialization: The loop is started with the value specified. condition: It evaluates to either 'true' or 'false'. If it is false then the loop is terminated. increment or decrement: After each iteration, value increments or decrements.

Example: Here num is initialized to value "1", condition is checked whether num<=10. If it is so then control goes into the loop and current value of num is printed. Now num is incremented and checked again whether num<=10.If it is so then again it enters into the loop. This process continues till num>10. It prints values 1 to10 on the screen.

```
for (int num = 1; num <= 10; num++)
{

    System.out.println("Num: " +
num);

}
```

C] Branching Statements:

1. Break statements:

The break statement is a branching statement that contains two forms: labeled and unlabeled. The break statement is used for breaking the execution of a loop (while, do-while and for) . It also terminates the switch statements.

Syntax:

break; // breaks the innermost loop or switch statement.
break label; // breaks the outermost loop in a series of nested loops.

Example: When if statement evaluates to true it prints "data is found" and comes out of the loop and executes the statements just following the loop.

```
int num[] = (2,9,1,4,25,50);

int search = 4;

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

   if ( num[i] = = search){

      System.out.println("data is found!");

      break;

   }

}
```

2. Continue statements:

This is a branching statement that are used in the looping statements (while, do-while and for) to skip the current iteration of the loop and resume the next iteration .

Syntax:

continue;

Example:

```
int num[] = (2,9,1,4,25,50);

int search = 4;
for (int i = 1;  i < num.length; i++){

   if (num[i] != search){
      continue;
   }
 if (found = = search){
   System.out.println("data is found!");
   break;  } }
```

3. Return statements:

It is a special branching statement that transfers the control to the caller of the method. This statement is used to return a value to the caller method and terminates execution of method. This has two forms: one that returns a value and the other that can not return. the returned value type must match the return type of method.

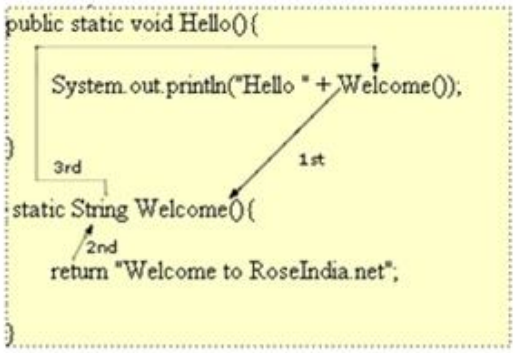
Syntax:

return; or return values;

return; //This returns nothing. So this can be used when method is declared with void return type.

return expression; //It returns the value evaluated from the expression.

Example: Here Welcome() function is called within println() function which returns a String value "Welcome to sunil3112.net". This is printed to the screen.



Chapter Three: Classes and Objects

Java OOPs Concepts

In this page, we will learn about basics of OOPs. Object Oriented Programming is a paradigm that provides many concepts such as **inheritance**, **data binding**, **polymorphism** etc.

Simula is considered as the first object-oriented programming language. The programming paradigm where everything is represented as an object, is known as truly object-oriented programming language.

Smalltalk is considered as the first truly object-oriented programming language.

OOPs (Object Oriented Programming System)

Object means a real word entity such as pen, chair, table etc.**Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

Collection of objects is called class. It is a logical entity.

Inheritance

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

When **one task is performed by different ways**i.e. known as polymorphism. For example: to convense the customer differently, to draw something e.g. shape or rectangle etc.

In java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something e.g. cat speaks meaw, dog barks woof etc.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

In java, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation.

For example: capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Object and Class in Java

In this page, we will learn about java objects and classes. In object-oriented programming technique, we design a program using objects and classes.

Object is the physical as well as logical entity whereas class is the logical entity only.

Object in Java

An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tengible and intengible). The example of integible object is banking system.

An object has three characteristics:

- **state:** represents data (value) of an object.

- **behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
- **identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But,it is used internally by the JVM to identify each object uniquely.

For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.

Object is an instance of a class. Class is a template or blueprint from which objects are created. So object is the instance(result) of a class.

Class in Java

A class is a group of objects that has common properties. It is a template or blueprint from which objects are created.

A class in java can contain:

- **data member**
- **method**
- **constructor**
- **block**
- **class and interface**

Syntax to declare a class:

```
class <class_name>{
   data member;
   method;
}
```

Simple Example of Object and Class

In this example, we have created a Student class that have two data members id and name. We are creating the object of the Student class by new keyword and printing the objects value.

```
class Student1{
 int id;//data member (also instance variable)
 String name;//data member(also instance variable)

 public static void main(String args[]){
  Student1 s1=new Student1();//creating an object of Student
```

```
7.    System.out.println(s1.id);
8.    System.out.println(s1.name);
9.   } }
```

Instance variable in Java

A variable that is created inside the class but outside the method, is known as instance variable.Instance variable doesn't get memory at compile time.It gets memory at runtime when object(instance) is created.That is why, it is known as instance variable.

Method in Java

In java, a method is like function i.e. used to expose behaviour of an object.

Advantage of Method

- Code Reusability
- Code Optimization
-

new keyword

The new keyword is used to allocate memory at runtime.

Example of Object and class that maintains the records of students

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method on it. Here, we are displaying the state (data) of the objects by invoking the displayInformation method.

```
class Student2
{
int rollno;  String name;
void insertRecord(int r, String n){  //method
rollno=r;
name=n;
}

void displayInformation(){System.out.println(rollno+" "+name);}//method
public static void main(String args[]){
Student2 s1=new Student2();
Student2 s2=new Student2();

s1.insertRecord(111,"Karan");
s2.insertRecord(222,"Aryan");
```

```
s1.displayInformation();
s2.displayInformation();

} }
```

Another Example of Object and Class

There is given another example that maintains the records of Rectangle class. Its exaplanation is same as in the above Student class example.

```
1. class Rectangle{
2. int length;
3. int width;

4. void insert(int l,int w){
5. length=l;
6. width=w;
7. }

8. void calculateArea(){System.out.println(length*width);}

9. public static void main(String args[]){
10. Rectangle r1=new Rectangle();
11. Rectangle r2=new Rectangle();

12. r1.insert(11,5);
13. r2.insert(3,15);

14. r1.calculateArea();
15. r2.calculateArea();
16. }
17. }
```

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By factory method etc.

We will learn, these ways to create the object later.

Difference between object and class

There are many differences between object and class. A list of differences between object and class are given below:

No.	Object	Class
1)	Object is an **instance** of a class.	Class is a **blueprint or template** from which objects are created.
2)	Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair etc.	Class is a **group of similar objects**.
3)	Object is a **physical** entity.	Class is a **logical** entity.
4)	Object is created through **new keyword** mainly e.g. Student s1=new Student();	Class is declared using**class keyword** e.g. class Student{}
5)	Object is created **many times** as per requirement.	Class is declared **once**.
6)	Object **allocates memory when it is created**.	Class **doesn't allocated memory when it is created**.
7)	There are **many ways to create object** in java such as new keyword, newInstance() method, clone() method, factory method and deserialization.	There is only **one way to define class** in java using class keyword.

Constructor in Java

Constructor in java is a *special type of method*that is used to initialize the object.

Java constructor is *invoked at the time of object creation*. It constructs the values i.e. provides data for the object that is why it is known as constructor.

Rules for creating java constructor

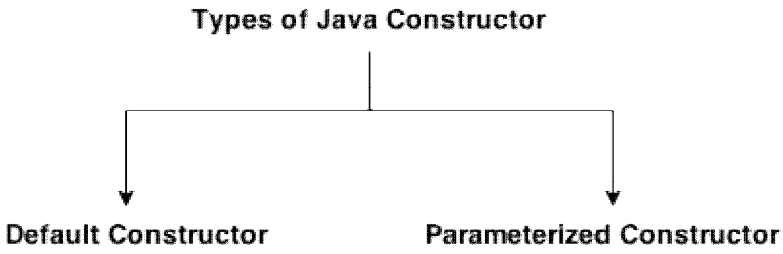
There are basically two rules defined for the constructor.

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

Types of java constructors

There are two types of constructors:

1. Default constructor (no-arg constructor)
2. Parameterized constructor



Java Default Constructor

A constructor that have no parameter is known as default constructor.

Syntax of default constructor:

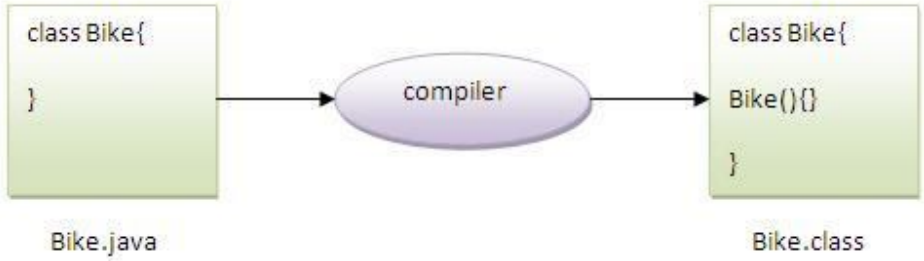
1. <class_name>(){}
2. Example of default constructor

3. In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
4. class Bike1{
5. Bike1(){System.out.println("Bike is created");}
6. public static void main(String args[]){
7. Bike1 b=new Bike1();
8. }
```

```
9.  }
```

Rule: If there is no constructor in a class, compiler automatically creates a default constructor.



What is the purpose of default constructor?

Default constructor provides the default values to the object like 0, null etc. depending on the type.

Example of default constructor that displays the default values

```
1.  class Student3
2.  {
3.  int id;
4.  String name;

5.  void display(){System.out.println(id+" "+name);}

6.  public static void main(String args[]){
7.  Student3 s1=new Student3();
8.  Student3 s2=new Student3();
9.  s1.display();
10. s2.display();
11. }
12. }
```

Explanation: In the above class,you are not creating any constructor so compiler provides you a default constructor.Here 0 and null values are provided by default constructor.

Java parameterized constructor

A constructor that have parameters is known as parameterized constructor.

Why use parameterized constructor?

Parameterized constructor is used to provide different values to the distinct objects.

Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
4. class Student4{
5. int id;
6. String name;

7. Student4(int i,String n){
8. id = i;
9. name = n;
10. }
11. void display(){System.out.println(id+" "+name);}

12. public static void main(String args[]){
13. Student4 s1 = new Student4(111,"Karan");
14. Student4 s2 = new Student4(222,"Aryan");
15. s1.display();
16. s2.display();
17. }
18. }
```

Difference between constructor and method in java

There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
Constructor is used to initialize the state of an object.	Method is used to expose b of an object.
Constructor must not have return type.	Method must have return ty
Constructor is invoked implicitly.	Method is invoked explicitl
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by any case.
Constructor name must be same as the class name.	Method name may or may same as class name.

Java static keyword

The **static keyword** in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

---

1) Java static variable

If you declare any variable as static, it is known static variable.

- The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees,college name of students etc.
- The static variable gets memory only once in class area at the time of class loading.

Advantage of static variable

It makes your program **memory efficient** (i.e it saves memory).

Understanding problem without static variable

```
1. class Student{
2. int rollno;
3. String name;
4. String college="ITS";
5. }
```

Suppose there are 500 students in my college, now all instance data members will get memory each time when object is created.All student have its unique rollno and name so instance data member is good.Here, college refers to the common property of all objects.If we make it static,this field will get memory only once.

Java static property is shared to all objects.

Example of static variable

```
1. //Program of static variable

2. class Student8{
3. int rollno;
4. String name;
```

```
5.  static String college ="ITS";

6.  Student8(int r,String n){
7.  rollno = r;
8.  name = n;
9.  }
10. void display (){System.out.println(rollno+" "+name+" "+colleg
    e);}

11. public static void main(String args[]){
12. Student8 s1 = new Student8(111,"Karan");
13. Student8 s2 = new Student8(222,"Aryan");

14. s1.display();
15. s2.display();
16. }
17. }
```

Program of counter without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable, if it is incremented, it won't reflect to other objects. So each objects will have the value 1 in the count variable.

```
1.  class Counter{
2.  int count=0;//will get memory when instance is created

3.  Counter(){
4.  count++;
5.  System.out.println(count);
6.  }

7.  public static void main(String args[]){

8.  Counter c1=new Counter();
9.  Counter c2=new Counter();
10. Counter c3=new Counter();  }  }
```

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.

Example of static method

```
1. //Program of changing the common property of all objects(static field).

2. class Student9{
3. int rollno;
4. String name;
5. static String college = "ITS";

6. static void change(){
7. college = "BBDIT";
8. }

9. Student9(int r, String n){
10. rollno = r;
11. name = n;
12. }

13. void display (){System.out.println(rollno+" "+name+" "+college);}

14. public static void main(String args[]){
15. Student9.change();

16. Student9 s1 = new Student9 (111,"Karan");
17. Student9 s2 = new Student9 (222,"Aryan");
18. Student9 s3 = new Student9 (333,"Sonoo");

19. s1.display();
20. s2.display();
21. s3.display();
22. }
23. }
```

Another example of static method that performs normal calculation

```
1. //Program to get cube of a given number by static method
2.
3. class Calculate{
4.  static int cube(int x){
5.  return x*x*x;
6.  }
7.
```

```
8.  public static void main(String args[]){
9.  int result=Calculate.cube(5);
10. System.out.println(result);
11. }
12.}
```

this keyword in java

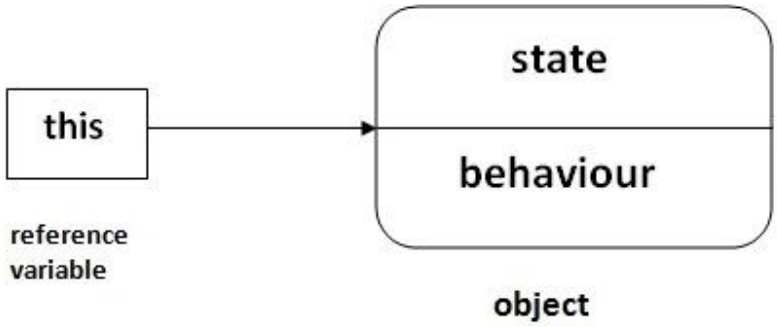
There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.

Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this keyword can be used to refer current class instance variable.
2. this() can be used to invoke current class constructor.
3. this keyword can be used to invoke current class method (implicitly)
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this keyword can also be used to return the current class instance.

Suggestion: If you are beginner to java, lookup only two usage of this keyword.



1) The this keyword can be used to refer current class instance variable.

If there is ambiguity between the instance variable and parameter, this keyword resolves the problem of ambiguity.

Understanding the problem without this keyword

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

```
1. class Student10{
2. int id;
3. String name;

4. Student10(int id,String name){
5. id = id;
```

```
6.  name = name;
7.  }
8.  void display(){System.out.println(id+" "+name);}

9.  public static void main(String args[]){
10. Student10 s1 = new Student10(111,"Karan");
11. Student10 s2 = new Student10(321,"Aryan");
12. s1.display();
13. s2.display();
14. }
15. }
```

In the above example, parameter (formal arguments) and instance variables are same that is why we are using this keyword to distinguish between local variable and instance variable.

Solution of the above problem by this keyword

```
1.  //example of this keyword
2.  class Student11{
3.  int id;
4.  String name;

5.  Student11(int id,String name){
6.  this.id = id;
7.  this.name = name;
8.  }
9.  void display(){System.out.println(id+" "+name);}
10. public static void main(String args[]){
11. Student11 s1 = new Student11(111,"Karan");
12. Student11 s2 = new Student11(222,"Aryan");
13. s1.display();
14. s2.display();
15. }
16. }
```

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

Program where this keyword is not required

```
1.  class Student12{
2.  int id;
3.  String name;

4.  Student12(int i,String n){
5.  id = i;
```

```
6.  name = n;
7.  }
8.  void display(){System.out.println(id+" "+name);}
9.  public static void main(String args[]){
10. Student12 e1 = new Student12(111,"karan");
11. Student12 e2 = new Student12(222,"Aryan");
12. e1.display();
13. e2.display();
14. }
15. }
```

---

2) this() can be used to invoked current class constructor.

The this() constructor call can be used to invoke the current class constructor (constructor chaining). This approach is better if you have many constructors in the class and want to reuse that constructor.

```
1.  //Program of this() constructor call (constructor chaining)

2.  class Student13{
3.  int id;
4.  String name;
5.  Student13(){System.out.println("default constructor is invoked");}

6.  Student13(int id,String name){
7.  this ();//it is used to invoked current class constructor.
8.  this.id = id;
9.  this.name = name;
10. }
11. void display(){System.out.println(id+" "+name);}

12. public static void main(String args[]){
13. Student13 e1 = new Student13(111,"karan");
14. Student13 e2 = new Student13(222,"Aryan");
15. e1.display();
16. e2.display();
17. }
18. }
```

Where to use this() constructor call?

The this() constructor call should be used to reuse the constructor in the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see

the example given below that displays the actual use of this keyword.

```
1. class Student14{
2. int id;
3. String name;
4. String city;

5. Student14(int id,String name){
6. this.id = id;
7. this.name = name;
8. }
9. Student14(int id,String name,String city){
10. this(id,name);//now no need to initialize id and name
11. this.city=city;
12. }
13. void display(){System.out.println(id+" "+name+" "+city);}

14. public static void main(String args[]){
15. Student14 e1 = new Student14(111,"karan");
16. Student14 e2 = new Student14(222,"Aryan","delhi");
17. e1.display();
18. e2.display();
19. }
20. }
```

Rule: Call to this() must be the first statement in constructor.

```
1. class Student15{
2. int id;
3. String name;
4. Student15(){System.out.println("default constructor is invoked");}

5. Student15(int id,String name){
6. id = id;
7. name = name;
8. this ();//must be the first statement
9. }
10. void display(){System.out.println(id+" "+name);}

11. public static void main(String args[]){
12. Student15 e1 = new Student15(111,"karan");
13. Student15 e2 = new Student15(222,"Aryan");
14. e1.display();
15. e2.display();
16. }
```

```
17. }
```

Access Modifiers in java

There are two types of modifiers in java: **access modifiers** and **non-access modifiers**.

The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

1. private
2. default
3. protected
4. public

There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc. Here, we will learn access modifiers.

---

1) private access modifier

The private access modifier is accessible only within class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error.

```
1. class A{
2. private int data=40;
3. private void msg(){System.out.println("Hello java");}
4. }

5. public class Simple{
6. public static void main(String args[]){
7. A obj=new A();
8. System.out.println(obj.data);//Compile Time Error
9. obj.msg();//Compile Time Error
10. }
11. }
```

Role of Private Constructor

2. If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
1. class A{
2. private A(){}//private constructor
3. void msg(){System.out.println("Hello java");}
4. }
5. public class Simple{
6. public static void main(String args[]){
7. A obj=new A();//Compile Time Error
} }
```

Note: A class cannot be private or protected except nested class.

2) default access modifier

If you don't use any modifier, it is treated as **default** bydefault. The default modifier is accessible only within package.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
1. //save by A.java
2. package pack;
3. class A{
4.  void msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. package mypack;
8. import pack.*;
9. class B{
10. public static void main(String args[]){
11.  A obj = new A();//Compile Time Error
12.  obj.msg();//Compile Time Error
13. }
14.}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) protected access modifier

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
1. //save by A.java
2. package pack;
3. public class A{
4. protected void msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. package mypack;
8. import pack.*;

9. class B extends A{
10. public static void main(String args[]){
11. B obj = new B();
12. obj.msg();
13. }
14. }
```

Output:Hello

4) public access modifier

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

```
//save by A.java

package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
//save by B.java

package mypack;
import pack.*;

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

```
    A obj = new A();
    obj.msg();
   }
}
```

Understanding all java access modifiers

Let's understand the access modifiers by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Java access modifiers with method overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

```
1. class A{
2. protected void msg(){System.out.println("Hello java");}
3. }

4. public class Simple extends A{
5. void msg(){System.out.println("Hello java");}//C.T.Error
6. public static void main(String args[]){
7. Simple obj=new Simple();
8. obj.msg();
9. }
10.}
```

The default modifier is more restrictive than protected. That is why there is compile time error.

Encapsulation in Java

Encapsulation in java is a *process of wrapping code and data together into a single unit*, for example capsule i.e. mixed of several medicines.

We can create a fully encapsulated class in java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

The **Java Bean** class is the example of fully encapsulated class.

Advantage of Encapsulation in java

By providing only setter or getter method, you can make the class **read-only or write-only**.

It provides you the **control over the data**. Suppose you want to set the value of id i.e. greater than 100 only, you can write the logic inside the setter method.

Simple example of encapsulation in java

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

```
1. //save as Student.java
2. package com.javatpoint;
3. public class Student{
4. private String name;
5.   public String getName(){
6. return name;  }
7. public void setName(String name){
8. this.name=name  }
9. }
10. //save as Test.java
11. package com.javatpoint;
12. class Test{
13. public static void main(String[] args){
14. Student s=new Student();
15. s.setname("vijay");
16. System.out.println(s.getName());
17. }  }
```

Chapter Four: Inheritance & Polymorphism

Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object.

The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.

Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

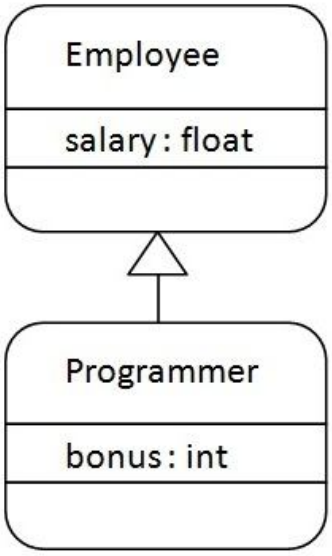
Syntax of Java Inheritance

```
1. class Subclass-name extends Superclass-name
2. {
3. //methods and fields
4. }
```

The **extends keyword** indicates that you are making a new class that derives from an existing class.

In the terminology of Java, a class that is inherited is called a super class. The new class is called a subclass.

Understanding the simple example of inheritance



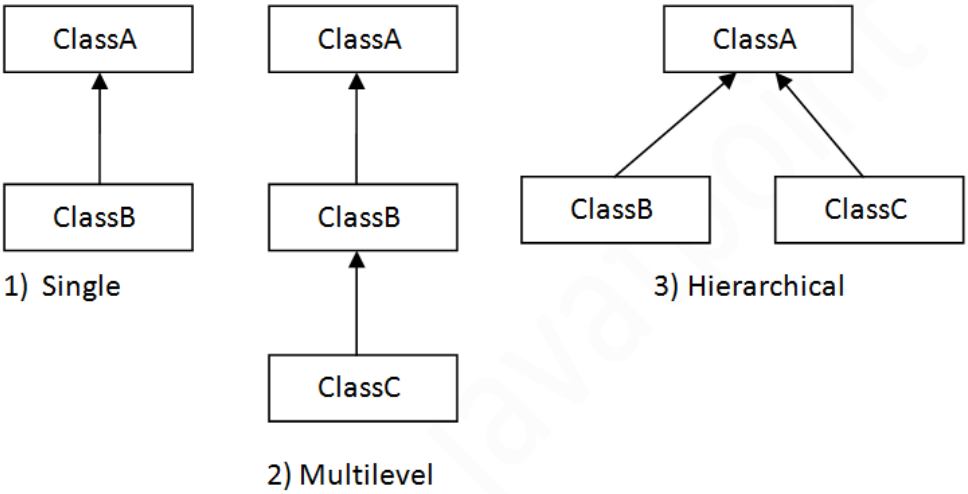
As displayed in the above figure, Programmer is the subclass and Employee is the superclass. Relationship between two classes is **Programmer IS-A Employee**.It means that Programmer is a type of Employee.

```
1. class Employee{
2. float salary=40000;
3. }
4. class Programmer extends Employee{
5. int bonus=10000;
6. public static void main(String args[]){
7. Programmer p=new Programmer();
8. System.out.println("Programmer salary is:"+p.salary);
9. System.out.println("Bonus of Programmer is:"+p.bonus);
10. }
11. }
```

Types of inheritance in java

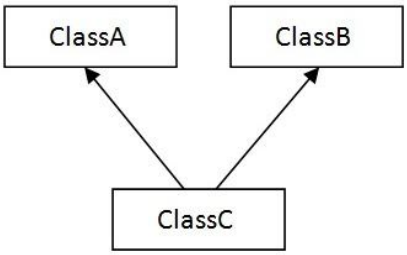
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

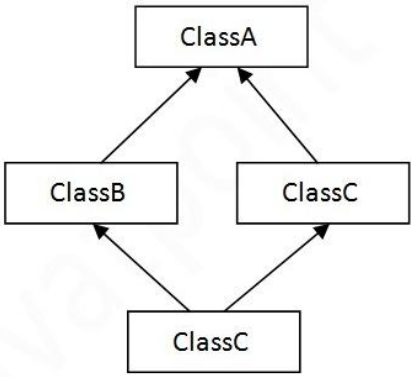


Note: Multiple inheritance is not supported in java through class.

When a class extends multiple classes i.e. known as multiple inheritance. For Example:



4) Multiple



5) Hybrid

.

Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**.

In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. method must have same name as in the parent class
2. method must have same parameter as in the parent class.
3. must be IS-A relationship (inheritance).

Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

```
1. class Vehicle{
2. void run(){System.out.println("Vehicle is running");}
3. }
4. class Bike extends Vehicle{

5. public static void main(String args[]){
6. Bike obj = new Bike();
7. obj.run();
8. } }
```

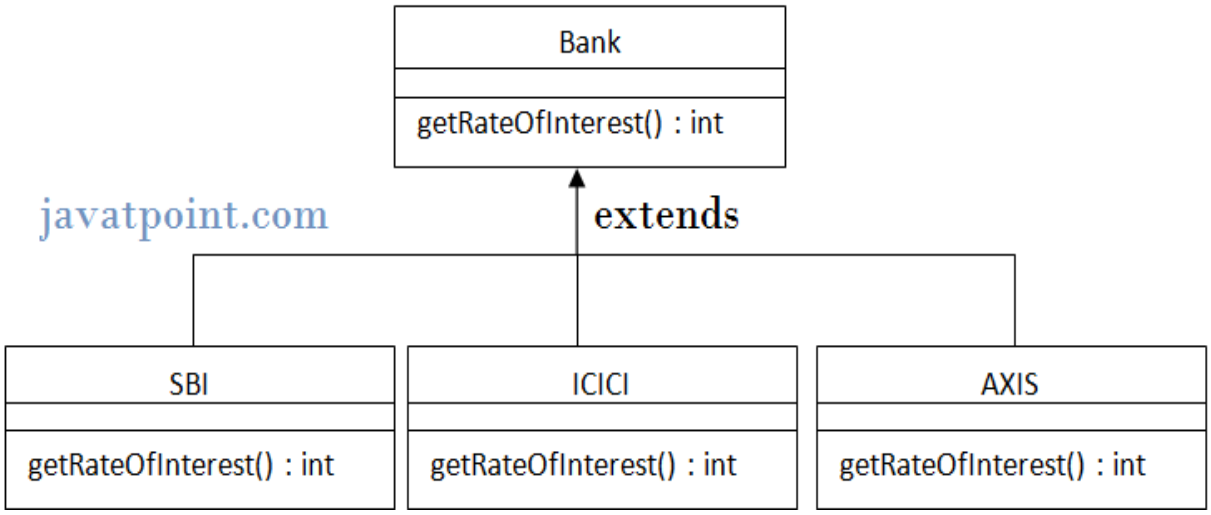
Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method is same and there is IS-A relationship between the classes, so there is method overriding.

```
1. class Vehicle{
2. void run(){System.out.println("Vehicle is running");}
3. }
4. class Bike2 extends Vehicle{
5. void run(){System.out.println("Bike is running safely");}

6. public static void main(String args[]){
7. Bike2 obj = new Bike2();
8. obj.run();
9. }
```

Real example of Java Method Overriding

Consider a scenario, Bank is a class that provides functionality to get rate of interest. But, rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.




```
1. class Bank{
2. int getRateOfInterest(){return 0;}
3. }

4. class SBI extends Bank{
5. int getRateOfInterest(){return 8;}
```

```
6. }

7. class ICICI extends Bank{
8. int getRateOfInterest(){return 7;}
9. }
10. class AXIS extends Bank{
11. int getRateOfInterest(){return 9;}
12. }

13. class Test2{
14. public static void main(String args[]){
15. SBI s=new SBI();
16. ICICI i=new ICICI();
17. AXIS a=new AXIS();
18. System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
19. System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
20. System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
21. }
22. }
```

Method Overloading in Java : If a class have multiple methods by same name but different parameters, it is known as

Method Overloading.If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behaviour of the method because its name differs. So, we perform method overloading to figure out the program quickly.

Advantage of method overloading?

Method overloading **increases the readability of the program**.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

In java, Methood Overloading is not possible by changing the return type of the method.

1)Example of Method Overloading by changing the no. of arguments

In this example, we have created two overloaded methods, first sum method performs addition of two numbers and second sum method performs addition of three numbers.

```
class Calculation{
 void sum(int a,int b){System.out.println(a+b);}
 void sum(int a,int b,int c){System.out.println(a+b+c);}

 public static void main(String args[]){
 Calculation obj=new Calculation();
 obj.sum(10,10,10);
 obj.sum(20,20);

 }
}
```

2) Example of Method Overloading by changing data type of argument

In this example, we have created two overloaded methods that differs in data type. The first sum method receives two integer arguments and second sum method receives two double arguments.

```
1. class Calculation2{
2. void sum(int a,int b){System.out.println(a+b);}
3. void sum(double a,double b){System.out.println(a+b);}

4. public static void main(String args[]){
5. Calculation2 obj=new Calculation2();
6. obj.sum(10.5,10.5);
7. obj.sum(20,20);

8. }
9. }
```

Super keyword in java

The **super** keyword in java is a reference variable that is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.

Usage of java super Keyword

1. super is used to refer immediate parent class instance variable.
2. super() is used to invoke immediate parent class constructor.
3. super is used to invoke immediate parent class method.

1) super is used to refer immediate parent class instance variable.

Problem without super keyword

```
1. class Vehicle{
2. int speed=50;
3. }
4. class Bike3 extends Vehicle{
5. int speed=100;
6. void display(){
7. System.out.println(speed);//will print speed of Bike
8. }
9. public static void main(String args[]){
10. Bike3 b=new Bike3();
11. b.display();
12. }
13. }
```

Solution by super keyword

```
//example of super keyword

class Vehicle{
int speed=50;  }

class Bike4 extends Vehicle{
int speed=100;

void display(){
System.out.println(super.speed);//will print speed of Vehicle
now  }
public static void main(String args[]){
Bike4 b=new Bike4();
b.display();

} }
```

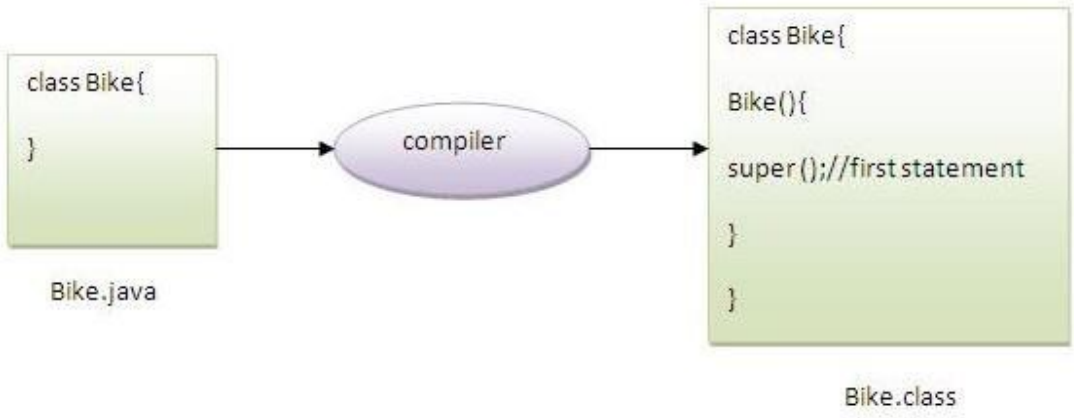
2) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor as given below:

```
1. class Vehicle{
2. Vehicle(){System.out.println("Vehicle is created");}
3. }

4. class Bike5 extends Vehicle{
5. Bike5(){
```

```
6.  super();//will invoke parent class constructor
7.  System.out.println("Bike is created");
8.  }
9.  public static void main(String args[]){
10. Bike5 b=new Bike5();

11. } }
```

Note: super() is added in each class constructor automatically by compiler.



As we know well that default constructor is provided by compiler automatically but it also adds super() for the first statement.If you are creating your own constructor and you don't have either this() or super() as the first statement, compiler will provide super() as the first statement of the constructor.

3) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used in case subclass contains the same method as parent class as in the example given below:

```
1.  class Person{
2.  void message(){System.out.println("welcome");}
3.  }

4.  class Student16 extends Person{
5.  void message(){System.out.println("welcome to java");}

6.  void display(){
7.  message();//will invoke current class message() method
8.  super.message();//will invoke parent class message() method
9.  }
```

```
10. public static void main(String args[]){
11. Student16 s=new Student16();
12. s.display();
13. }
14. }
```

In case there is no method in subclass as parent, there is no need to use super. In the example given below message() method is invoked from Student class but Student class does not have message() method, so you can directly call message() method.

Program in case super is not required

```
1. class Person{
2. void message(){System.out.println("welcome");}
3. }
4. class Student17 extends Person{
5. void display(){
6. message();//will invoke parent class message() method
7. }

8. public static void main(String args[]){
9. Student17 s=new Student17();
10. s.display();
11. }
12. }
```

Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
1. class Bike9{
2. final int speedlimit=90;//final variable
3. void run(){
4. speedlimit=400;
5. }
6. public static void main(String args[]){
7. Bike9 obj=new  Bike9();
8. obj.run();
9. }
10. }//end of class
```

---

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
1. class Bike{
2. final void run(){System.out.println("running");}
3. }

4. class Honda extends Bike{
5. void run(){System.out.println("running safely with 100kmph");}

6. public static void main(String args[]){
7. Honda honda= new Honda();
```

```
8. honda.run();
9. }
10.}
```

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
1. final class Bike{}

2. class Honda1 extends Bike{
3. void run(){System.out.println("running safely with 100kmph");}

4. public static void main(String args[]){
5. Honda1 honda= new Honda();
6. honda.run();
7. }
8. }
```

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

```
1. class Bike{
2. final void run(){System.out.println("running...");}
3. }
4. class Honda2 extends Bike{
5. public static void main(String args[]){
6. new Honda2().run();
7. }
8. }
```

Polymorphism in Java

Polymorphism in java is a concept by which we can perform a *single action by different ways*. Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in java: compile time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

If you overload static method in java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

Runtime Polymorphism in Java

Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

When reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:


```
1. class A{}
2. class B extends A{}
1. A a=new B();//upcasting
```

Example of Java Runtime Polymorphism

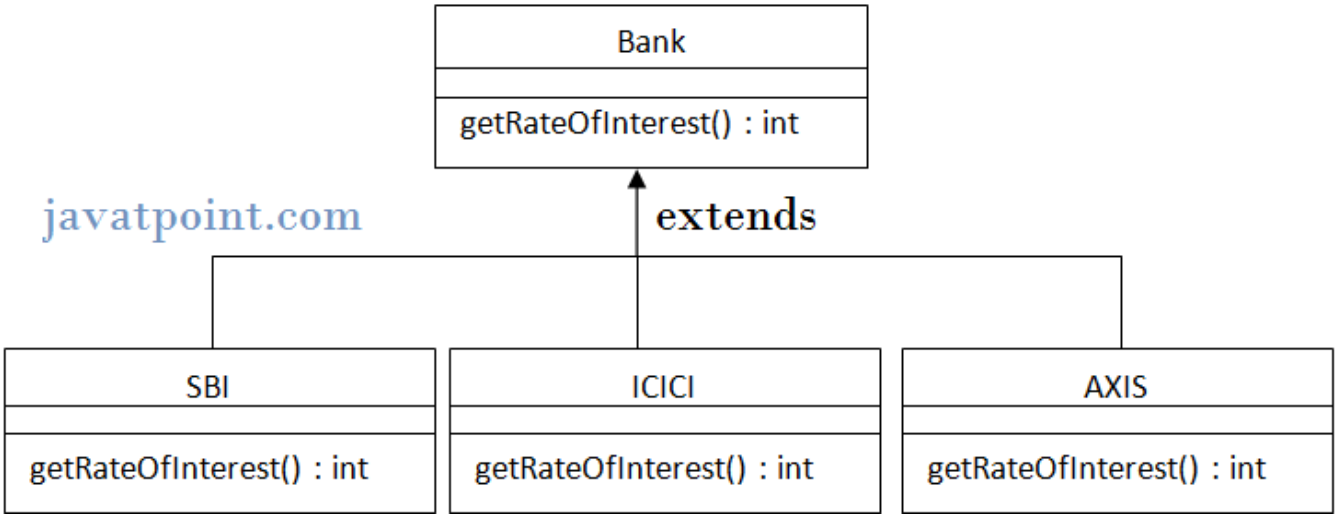
In this example, we are creating two classes Bike and Splendar. Splendar class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
1. class Bike{
2. void run(){System.out.println("running");}
3. }
4. class Splender extends Bike{
5. void run(){System.out.println("running safely with 60km");}

6. public static void main(String args[]){
7. Bike b = new Splender();//upcasting
8. b.run();  } }
```

Real example of Java Runtime Polymorphism

Consider a scenario, Bank is a class that provides method to get the rate of interest. But, rate of interest may differ according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.



Note: It is also given in method overriding but there was no upcasting.

```
1.  class Bank{
2.  int getRateOfInterest(){return 0;}
3.  }
4.  class SBI extends Bank{
5.  int getRateOfInterest(){return 8;}
6.  }

7.  class ICICI extends Bank{
8.  int getRateOfInterest(){return 7;}
9.  }
10. class AXIS extends Bank{
11. int getRateOfInterest(){return 9;}
12. }
13. class Test3{
14. public static void main(String args[]){
15. Bank b1=new SBI();
16. Bank b2=new ICICI();
17. Bank b3=new AXIS();
18. System.out.println("SBI Rate of Interest: "+b1.getRateOfInterest());
19. System.out.println("ICICI Rate of Interest: "+b2.getRateOfInterest());
20. System.out.println("AXIS Rate of Interest: "+b3.getRateOfInterest()); } }
```

Java Runtime Polymorphism with data member

Method is overridden not the datamembers, so runtime polymorphism can't be achieved by data members.

In the example given below, both the classes have a datamember speedlimit, we are accessing the datamember by the reference variable of Parent class which refers to the subclass object. Since we are accessing the datamember which is not overridden, hence it will access the datamember of Parent class always.

Rule: Runtime polymorphism can't be achieved by data members.

```
1. class Bike{
2. int speedlimit=90;
3. }
4. class Honda3 extends Bike{
5. int speedlimit=150;

6. public static void main(String args[]){
7. Bike obj=new Honda3();
8. System.out.println(obj.speedlimit);//90
9. }
```

Let's see the simple example of Runtime Polymorphism with multilevel inheritance.

```
1. class Animal{
2. void eat(){System.out.println("eating");}
3. }
4. class Dog extends Animal{
5. void eat(){System.out.println("eating fruits");}
6. }
7. class BabyDog extends Dog{
8. void eat(){System.out.println("drinking milk");}
9. public static void main(String args[]){
10. Animal a1,a2,a3;
11. a1=new Animal();
12. a2=new Dog();
13. a3=new BabyDog();

14. a1.eat();
15. a2.eat();
16. a3.eat();
```

17.} }

Method Overloading vs Method Overriding

Difference between method overloading and method overriding in java

There are many differences between method overloading and method overriding in java. A list of differences between method overloading and method overriding are given below:

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding,*parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

Java Method Overloading example

```
1. class OverloadingExample{
2. static int add(int a,int b){return a+b;}
3. static int add(int a,int b,int c){return a+b+c;}
4. }
5. Java Method Overriding example
6. class Animal{
7. void eat(){System.out.println("eating...");}
8. }
9. class Dog extends Animal{
10. void eat(){System.out.println("eating bread...");} }
```

Static Binding and Dynamic Binding

Connecting a method call to the method body is known as binding.

There are two types of binding

1. static binding (also known as early binding).
2. dynamic binding (also known as late binding).

Understanding Type

Let's understand the type of instance.

1) variables have a type

Each variable has a type, it may be primitive and non-primitive.

```
1. int data=30;
```

Here data variable is a type of int.

2) References have a type

```
1. class Dog{
2.  public static void main(String args[]){
3.   Dog d1;//Here d1 is a type of Dog
4.  }
5. }
```

3) Objects have a type

An object is an instance of particular java class,but it is also an instance of its superclass.

```
1. class Animal{}
2.
3. class Dog extends Animal{
4.  public static void main(String args[]){
5.   Dog d1=new Dog();
6.  }
7. }
```

Here d1 is an instance of Dog class, but it is also an instance of Animal.

static binding

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

Example of static binding

```
1. class Dog{
2. private void eat(){System.out.println("dog is eating...");}

3. public static void main(String args[]){
4. Dog d1=new Dog();
5. d1.eat();
6. }
7. }
8.
```

Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding

```
1. class Animal{
2. void eat(){System.out.println("animal is eating...");}
3. }

4. class Dog extends Animal{
5. void eat(){System.out.println("dog is eating...");}

6. public static void main(String args[]){
7. Animal a=new Dog();
8. a.eat();
9. }
10. }
```

Abstract class in Java

A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).

Before learning java abstract class, let's understand the abstraction in java first.

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstaction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class in Java

A class that is declared as abstract is known as **abstract class**. It needs to be extended and its method implemented. It cannot be instantiated.

Example abstract class

```
1. abstract class A{}
```

---

abstract method

A method that is declared as abstract and does not have implementation is known as abstract method.

Example abstract method

```
1. abstract void printStatus();//no body and abstract
```

Example of abstract class that has abstract method

In this example, Bike the abstract class that contains only one abstract method run. It implementation is provided by the Honda class.

```
1. abstract class Bike{
2. abstract void run();
3. }

4. class Honda4 extends Bike{
5. void run(){System.out.println("running safely..");}

6. public static void main(String args[]){
7. Bike obj = new Honda4();
```

```
8. obj.run();
9. }
10.}
```

Understanding the real scenario of abstract class

In this example, Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes. Mostly, we don't know about the implementation class (i.e. hidden to the end user) and object of the implementation class is provided by the **factory method**.

A **factory method** is the method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

File: TestAbstraction1.java

```
1. abstract class Shape{
2. abstract void draw();
3. }
4. //In real scenario, implementation is provided by others i.e. unknown by en
   d user
5. class Rectangle extends Shape{
6. void draw(){System.out.println("drawing rectangle");}
7. }

8. class Circle1 extends Shape{
9. void draw(){System.out.println("drawing circle");}
10.}

11.//In real scenario, method is called by programmer or user
12.class TestAbstraction1{
13.public static void main(String args[]){
14.Shape s=new Circle1();//In real scenario, object is provided through metho
   d e.g. getShape() method
15.s.draw();
16.} }
```

Another example of abstract class in java

File: TestBank.java

```
1. abstract class Bank{
2. abstract int getRateOfInterest();
3. }
```

```
4. class SBI extends Bank{
5. int getRateOfInterest(){return 7;}
6. }
7. class PNB extends Bank{
8. int getRateOfInterest(){return 7;}
9. }

10. class TestBank{
11. public static void main(String args[]){
12. Bank b=new SBI();//if object is PNB, method of PNB will be invoked
13. int interest=b.getRateOfInterest();
14. System.out.println("Rate of Interest is: "+interest+" %");
15. }}
```

Interface in Java

An **interface in java** is a blueprint of a class. It has static constants and abstract methods only.

The interface in java is **a mechanism to achieve fully abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve fully abstraction and multiple inheritance in Java.

Java Interface also **represents IS-A relationship**.

It cannot be instantiated just like abstract class.

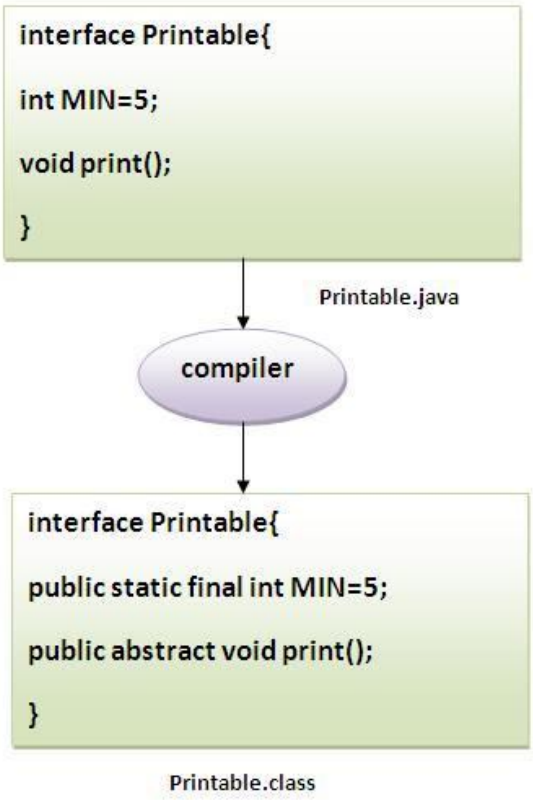
Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve fully abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

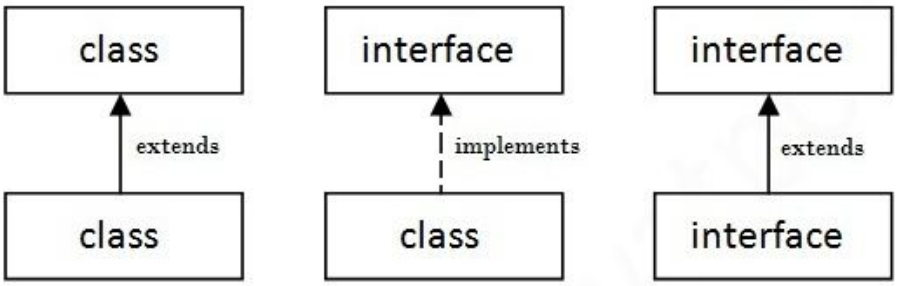
The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

In other words, Interface fields are public, static and final bydefault, and methods are public and abstract.



Understanding relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.



Simple example of Java interface

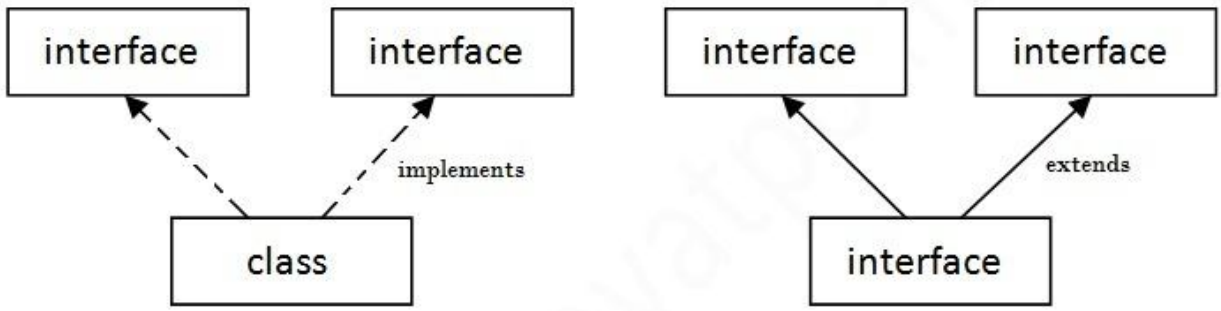
In this example, Printable interface have only one method, its implementation is provided in the A class.

```
1. interface printable{
2. void print();
3. }
```

```
4. class A6 implements printable{
5. public void print(){System.out.println("Hello");}

6. public static void main(String args[]){
7. A6 obj = new A6();
8. obj.print();
9. }
10.}
```

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Multiple Inheritance in Java

```
1. interface Printable{
2. void print();
3. }

4. interface Showable{
5. void show();
6. }

7. class A7 implements Printable,Showable{

8. public void print(){System.out.println("Hello");}
9. public void show(){System.out.println("Welcome");}

10. public static void main(String args[]){
11. A7 obj = new A7();
12. obj.print();
13. obj.show();
```

```
14. }
15. }
```

Q) Multiple inheritance is not supported through class in java but it is possible by interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class. For example:

```
1.  interface Printable{
2.  void print();
3.  }
4.  interface Showable{
5.  void print();
6.  }

7.  class TestTnterface1 implements Printable,Showable{
8.  public void print(){System.out.println("Hello");}
9.  public static void main(String args[]){
10. TestTnterface1 obj = new TestTnterface1();
11. obj.print();
12. }
13. }
```

Interface inheritance

A class implements interface but one interface extends another interface .

```
1.  interface Printable{
2.  void print();
3.  }
4.  interface Showable extends Printable{
5.  void show();
6.  }
7.  class Testinterface2 implements Showable{

8.  public void print(){System.out.println("Hello");}
9.  public void show(){System.out.println("Welcome");}

10. public static void main(String args[]){
11. Testinterface2 obj = new Testinterface2();
12. obj.print();
13. obj.show();
```

```
14.} }
```

Nested Interface in Java

Note: An interface can have another interface i.e. known as nested interface. We will learn it in detail in the nested classes chapter. For example:

```
1. interface printable{
2.  void print();
3.  interface MessagePrintable{
4.    void msg();
5.  }
6. }
```

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can have static methods, main method and constructor**.	Interface **can't have static methods, main method or constructor**.
5) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
6) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

7) **Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Chapter Five : Exception Handling in Java

The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

In this page, we will learn about java exception, its type and the difference between checked and unchecked exceptions.

What is exception

Dictionary Meaning: Exception is an abnormal condition.

In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is exception handling

Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.

Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

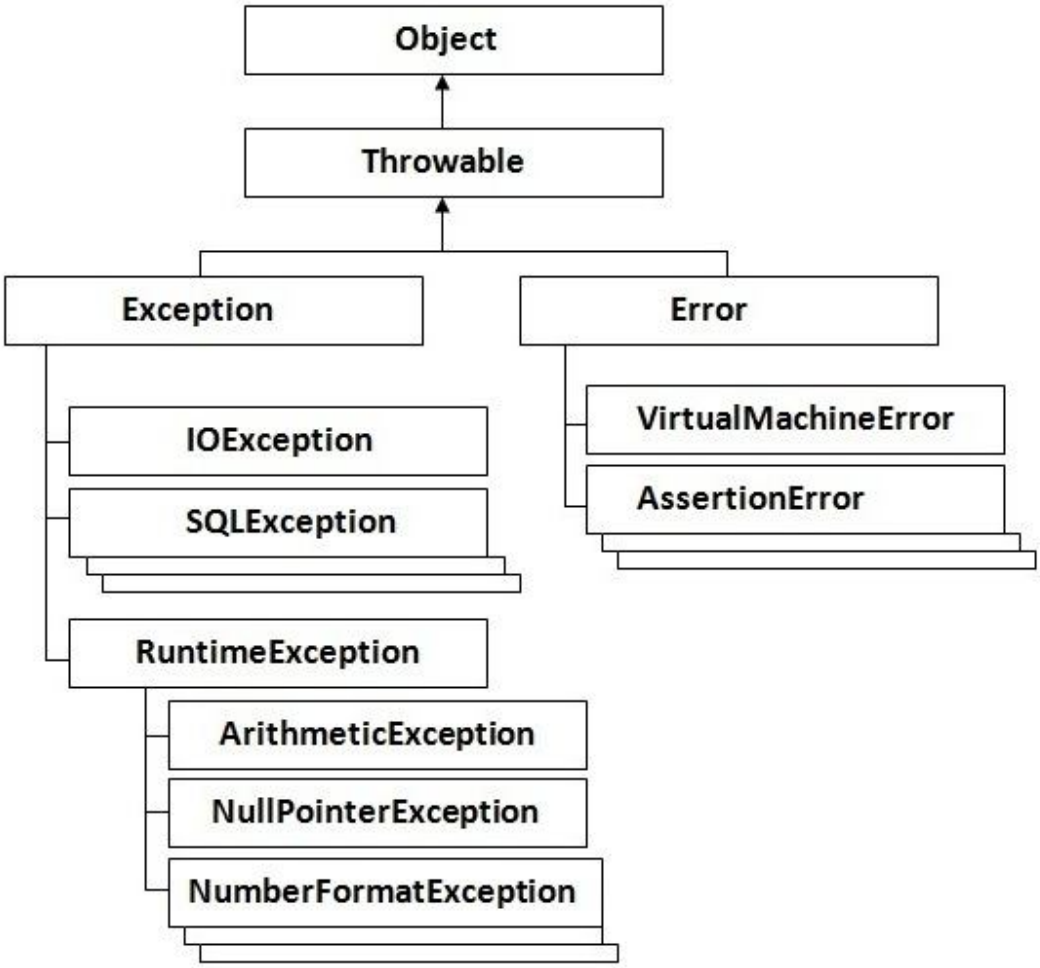
1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

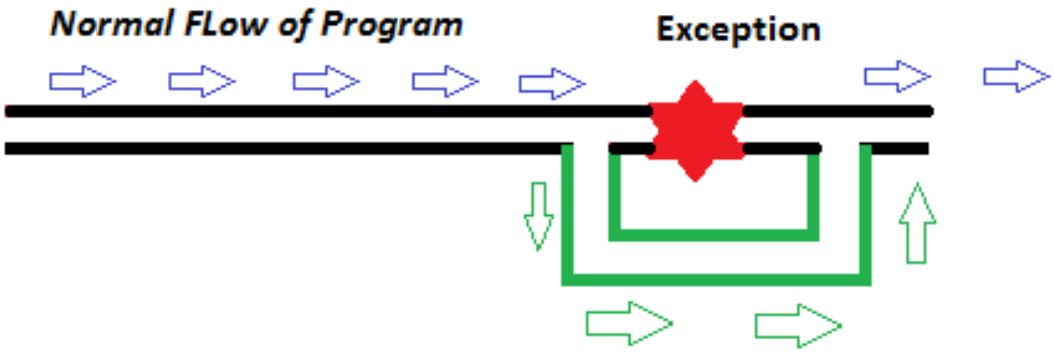
Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.

Do You Know ?

- What is the difference between checked and unchecked exceptions ?
- What happens behind the code int data=50/0; ?
- Why use multiple catch block ?
- Is there any possibility when finally block is not executed ?
- What is exception propagation ?
- What is the difference between throw and throws keyword ?
- What are the 4 rules for using exception handling with method overriding ?

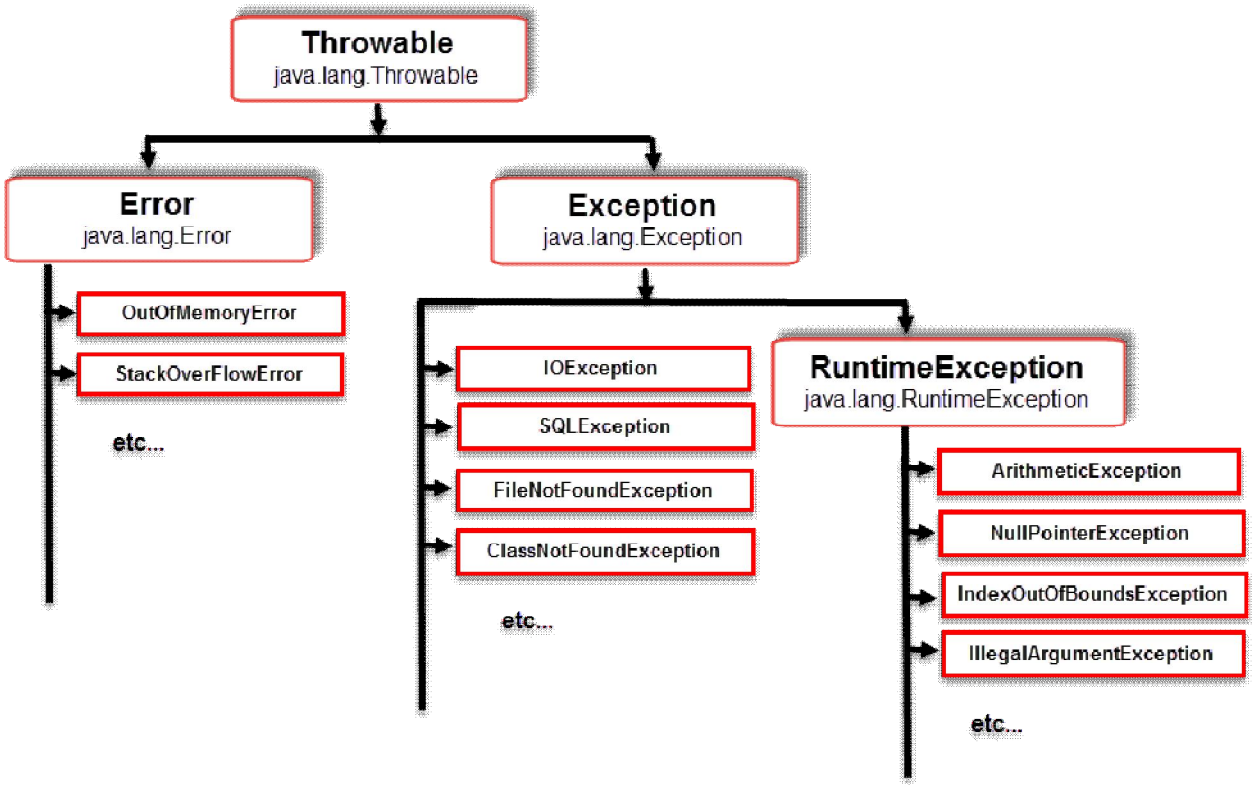
Hierarchy of Java Exception classes





EXCEPTION HANDLING

Alternate way to continue flow of program



Types of Exception

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. **Checked Exception**
2. **Unchecked Exception**
3. **Error**

Difference between checked and unchecked exceptions

1) Checked Exception

The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Common scenarios where exceptions may occur

There are given some scenarios where unchecked exceptions can occur. They are as follows:

1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
1. int a=50/0;//ArithmeticException
```

---

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

```
1. String s=null;
2. System.out.println(s.length());//NullPointerException
```

---

3) Scenario where NumberFormatException occurs

The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

```
1. String s="abc";
2. int i=Integer.parseInt(s);//NumberFormatException
```

---

4) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

```
1. int a[]=new int[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException
```

Java Exception Handling Keywords

There are 5 keywords used in java exception handling.

1. try
2. catch
3. finally
4. throw
5. throws

Java try-catch

Java try block

Java try block is used to enclose the code that might throw an exception. It must be used within the method.

Java try block must be followed by either catch or finally block.

Syntax of java try-catch

```
1. try{
2. //code that may throw exception
3. }catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
1. try{
2. //code that may throw exception
3. }finally{}
```

Java catch block

Java catch block is used to handle the Exception. It must be used after the try block only.

You can use multiple catch block with a single try.

Problem without exception handling

Let's try to understand the problem if we don't use try-catch block.

```
1. public class Testtrycatch1{
2.  public static void main(String args[]){
3.    int data=50/0;//may throw exception
4.    System.out.println("rest of the code...");
5. }
6. }
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
```

As displayed in the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed).

There can be 100 lines of code after exception. So all the code after exception will not be executed.

Solution by exception handling

Let's see the solution of above problem by java try-catch block.

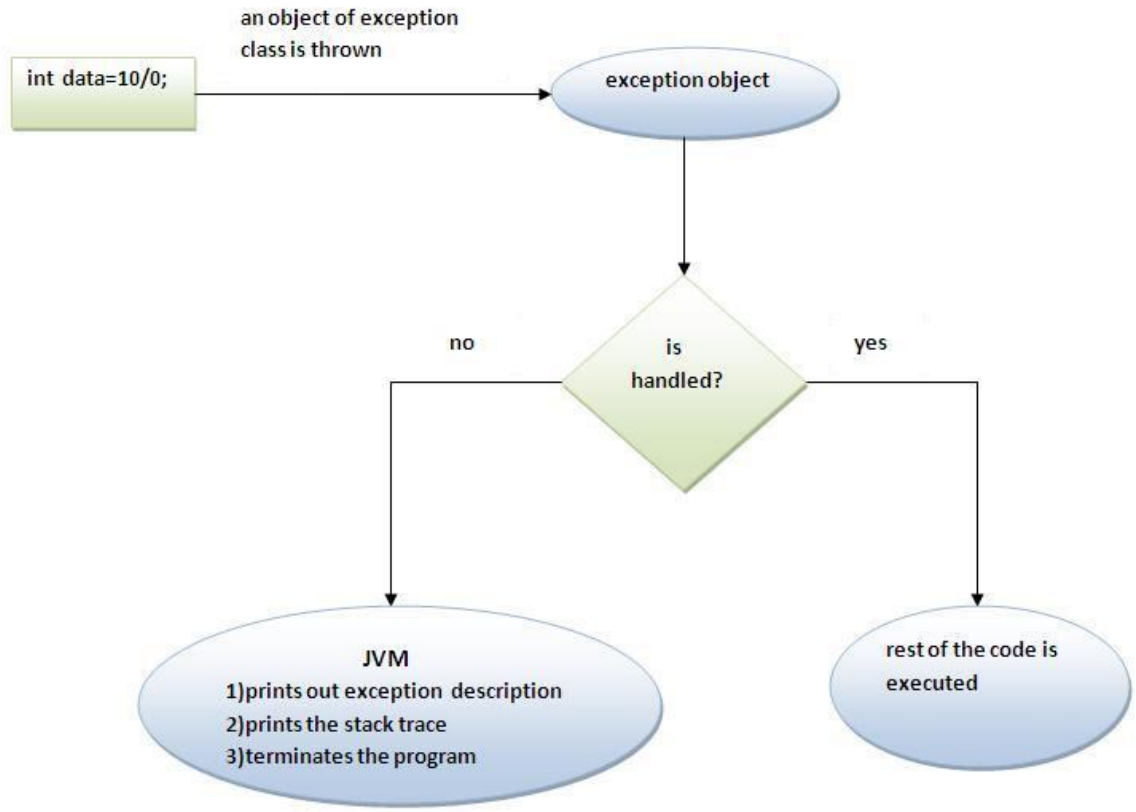
```
1. public class Testtrycatch2{
2.  public static void main(String args[]){
3.   try{
4.     int data=50/0;
5.   }catch(ArithmeticException e){System.out.println(e);}
6.   System.out.println("rest of the code...");
7. }
8. }
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.

Internal working of java try-catch block



The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

Java catch multiple exceptions

Java Multi catch block

If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block.

Let's see a simple example of java multi-catch block.

```
1. public class TestMultipleCatchBlock{
2. public static void main(String args[]){
3. try{
4. int a[]=new int[5];
```

```
5.  a[5]=30/0;
6.  }
7.  catch(ArithmeticException e){System.out.println("task1 is completed");}
8.  catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
9.  catch(Exception e){System.out.println("common task completed");}

10. System.out.println("rest of the code...");
11. }
12. }
```

Output:task1 completed

rest of the code...

Rule: At a time only one Exception is occured and at a time only one catch block is executed.

Rule: All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception .

```
1.  class TestMultipleCatchBlock1{
2.   public static void main(String args[]){
3.    try{
4.     int a[]=new int[5];
5.     a[5]=30/0;
6.    }
7.    catch(Exception e){System.out.println("common task completed");}
8.    catch(ArithmeticException e){System.out.println("task1 is completed");}
9.    catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
10.   System.out.println("rest of the code...");
11. }
12. }
```

Output:

Compile-time error

Java Nested try block

The try block within a try block is known as nested try block in java.

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

```
....
try
{
statement 1;
statement 2;
try
{
statement 1;
statement 2;
}
catch(Exception e)
{
}
}
catch(Exception e)
{
}
....
```

Java nested try example

Let's see a simple example of java nested try block.

```
class Excep6{
public static void main(String args[]){
try{
try{
System.out.println("going to divide");
int b =39/0;
}catch(ArithmeticException e){System.out.println(e);}

try{
int a[]=new int[5];
a[5]=4;
}catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}

System.out.println("other statement);
}catch(Exception e){System.out.println("handeled");}
```

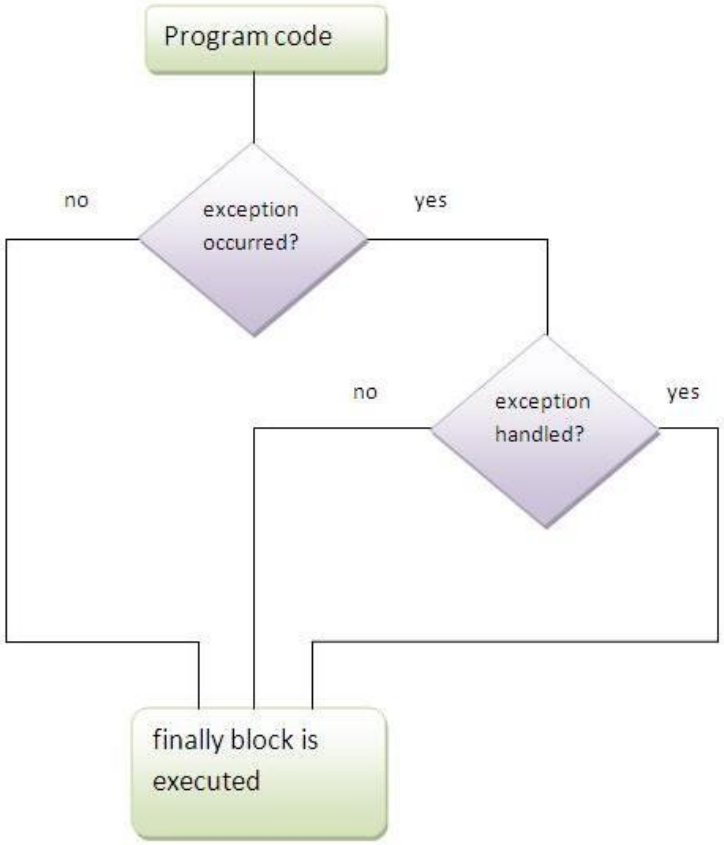
```
14. System.out.println("normal flow..");
15. }
16. }
```

Java finally block

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block must be followed by try or catch block.



Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).

Why use java finally

- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Usage of Java finally

Let's see the different cases where java finally block can be used.

Case 1

Let's see the java finally example where **exception doesn't occur**.

```
1. class TestFinallyBlock{
2. public static void main(String args[]){
3. try{
4. int data=25/5;
5. System.out.println(data);
6. }
7. catch(NullPointerException e){System.out.println(e);}
8. finally{System.out.println("finally block is always executed");}
9. System.out.println("rest of the code...");
10. }
11. }
```

Output:5

finally block is always executed

rest of the code...

Case 2

Let's see the java finally example where **exception occurs and not handled**.

```
1. class TestFinallyBlock1{
2. public static void main(String args[]){
3. try{
4. int data=25/0;
5. System.out.println(data);
6. }
7. catch(NullPointerException e){System.out.println(e);}
8. finally{System.out.println("finally block is always executed");}
9. System.out.println("rest of the code...");
10. }
11. }
```

Output:finally block is always executed

Exception in thread main java.lang.ArithmeticException:/ by zero

Case 3

Let's see the java finally example where **exception occurs and handled**.

```
1. public class TestFinallyBlock2{
    1. public static void main(String args[]){
    2. try{
    3. int data=25/0;
    4. System.out.println(data);
    5. }
```

```
6. catch(ArithmeticException e){System.out.println(e);}
7. finally{System.out.println("finally block is always executed");}
8. System.out.println("rest of the code...");
9. }
10. }
```

Java throw exception

Java throw keyword

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

We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

```
1. throw exception;
```

Let's see the example of throw IOException.

```
1. throw new IOException("sorry device error);
```

java throw keyword example

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
1. public class TestThrow1{
2. static void validate(int age){
3. if(age<18)
4. throw new ArithmeticException("not valid");
5. else
6. System.out.println("welcome to vote");
7. }
8. public static void main(String args[]){
9. validate(13);
10. System.out.println("rest of the code...");
11. }
12. }
```

The **Java throws keyword** is used to declare an exception. It gives an 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 normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Syntax of java throws

```
1. return_type method_name() throws exception_class_name{
2. //method code
3. }
```

Which exception should be declared

Ans) checked exception only, because:

- **unchecked Exception:** under your control so correct your code.
- **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception.

Java throws example

Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

```
1.  import java.io.IOException;
2.  class Testthrows1{
3.    void m()throws IOException{
4.      throw new IOException("device error");//checked exception
5.    }
6.    void n()throws IOException{
7.      m();
8.    }
9.    void p(){
10.    try{
11.     n();
12.    }catch(Exception e){System.out.println("exception handled");}
13.  }
14.  public static void main(String args[]){
15.   Testthrows1 obj=new Testthrows1();
16.   obj.p();
17.   System.out.println("normal flow...");
18.  }
19. }
```

Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.

There are two cases:

1. **Case1:**You caught the exception i.e. handle the exception using try/catch.
2. **Case2:**You declare the exception i.e. specifying throws with the method.

Case1: You handle the exception

- In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

```
1. import java.io.*;
2. class M{
3.  void method()throws IOException{
4.   throw new IOException("device error");
5.  }
6. }
7. public class Testthrows2{
8.    public static void main(String args[]){
9.     try{
10.     M m=new M();
11.     m.method();
12.    }catch(Exception e){System.out.println("exception handled");}
13.
14.    System.out.println("normal flow...");
15.  }
16.}
```

Output:exception handled

normal flow...

Case2: You declare the exception

- A)In case you declare the exception, if exception does not occur, the code will be executed fine.
- B)In case you declare the exception if exception occures, an exception will be thrown at runtime because throws does not handle the exception.

A)Program if exception does not occur

```
1. import java.io.*;
2. class M{
3. void method()throws IOException{
4. System.out.println("device operation performed");
5. }
6. }
7. class Testthrows3{
```

```
8.  public static void main(String args[])throws IOException{//declare except
    ion
9.  M m=new M();
10. m.method();

11. System.out.println("normal flow...");
12. }
13. }
```

```
Output:device operation performed
    normal flow...
```

B)Program if exception occurs

```
1.  import java.io.*;
2.  class M{
3.  void method()throws IOException{
4.  throw new IOException("device error");
5.  }
6.  }
7.  class Testthrows4{
8.  public static void main(String args[])throws IOException{//declare exception

9.  M m=new M();
10. m.method();

11. System.out.println("normal flow...");
12. }
13. }
```

```
Output:Runtime Exception
```

Difference between throw and throws

Que) Can we rethrow an exception?

Yes, by throwing same exception in catch block.

Difference between throw and throws in Java

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

No.	throw	throws

1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Java throw example

```
1. void m(){
2. throw new ArithmeticException("sorry");
3. }
```

Java throws example

```
1. void m()throws ArithmeticException{
2. //method code
3. }
```

Java throw and throws example

```
1. void m()throws ArithmeticException{
2. throw new ArithmeticException("sorry");
3. }
```

Difference between final, finally and finalize

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

Java final example

```
1. class FinalExample{
2. public static void main(String[] args){
3. final int x=100;
4. x=200;//Compile Time Error
5. }}
```

Java finally example

```
1. class FinallyExample{
2. public static void main(String[] args){
3. try{
4. int x=300;
5. }catch(Exception e){System.out.println(e);}
6. finally{System.out.println("finally block is executed");}
7. }}
```

Java finalize example

```
1. class FinalizeExample{
2. public void finalize(){System.out.println("finalize called");}
3. public static void main(String[] args){
4. FinalizeExample f1=new FinalizeExample();
5. FinalizeExample f2=new FinalizeExample();
6. f1=null;
7. f2=null;
8. System.gc();
9. }}
```

ExceptionHandling with MethodOverriding in Java

There are many rules if we talk about methodoverriding with exception handling. The Rules are as follows:

- **If the superclass method does not declare an exception**
  - If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
- **If the superclass method declares an exception**
  - If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

If the superclass method does not declare an exception

1) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.

```
1. import java.io.*;
2. class Parent{
3. void msg(){System.out.println("parent");}
```

```
4.  }

5.  class TestExceptionChild extends Parent{
6.  void msg()throws IOException{
7.  System.out.println("TestExceptionChild");
8.  }
9.  public static void main(String args[]){
10. Parent p=new TestExceptionChild();
11. p.msg();
12. }
13. }
```

Output:Compile Time Error

2) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.

```
1.  import java.io.*;
2.  class Parent{
3.  void msg(){System.out.println("parent");}
4.  }

5.  class TestExceptionChild1 extends Parent{
6.  void msg()throws ArithmeticException{
7.  System.out.println("child");
8.  }
9.  public static void main(String args[]){
10. Parent p=new TestExceptionChild1();
11. p.msg();
12. }
13. }
```

Output:child

If the superclass method declares an exception

1) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

Example in case subclass overridden method declares parent exception

```
1.  import java.io.*;
2.  class Parent{
3.  void msg()throws ArithmeticException{System.out.println("parent");}
```

```
4.  }

5.  class TestExceptionChild2 extends Parent{
6.  void msg()throws Exception{System.out.println("child");}

7.  public static void main(String args[]){
8.  Parent p=new TestExceptionChild2();
9.  try{
10. p.msg();
11. }catch(Exception e){}
12. }
13. }
```

Output:Compile Time Error

Example in case subclass overridden method declares same exception

```
1.  import java.io.*;
2.  class Parent{
3.  void msg()throws Exception{System.out.println("parent");}
4.  }

5.  class TestExceptionChild3 extends Parent{
6.  void msg()throws Exception{System.out.println("child");}

7.  public static void main(String args[]){
8.  Parent p=new TestExceptionChild3();
9.  try{
10. p.msg();
11. }catch(Exception e){}
12. }
13. }
```

Output:child

Example in case subclass overridden method declares subclass exception

```
1.  import java.io.*;
2.  class Parent{
3.  void msg()throws Exception{System.out.println("parent");}
4.  }

5.  class TestExceptionChild4 extends Parent{
6.  void msg()throws ArithmeticException{System.out.println("child");}
```

```
7.  public static void main(String args[]){
8.  Parent p=new TestExceptionChild4();
9.  try{
10. p.msg();
11. }catch(Exception e){}
12. }
13. }
```

Output:child

Example in case subclass overridden method declares no exception

```
1.  import java.io.*;
2.  class Parent{
3.  void msg()throws Exception{System.out.println("parent");}
4.  }

5.  class TestExceptionChild5 extends Parent{
6.  void msg(){System.out.println("child");}

7.  public static void main(String args[]){
8.  Parent p=new TestExceptionChild5();
9.  try{
10. p.msg();
11. }catch(Exception e){}
12. }
13. }
```

Output:child

Java Custom Exception

If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

By the help of custom exception, you can have your own exception and message.

Let's see a simple example of java custom exception.

```
1.  class InvalidAgeException extends Exception{
2.  InvalidAgeException(String s){
3.  super(s);
4.  }
5.  }
6.  class TestCustomException1{

7.  static void validate(int age)throws InvalidAgeException{
8.  if(age<18)
```

```
9.      throw new InvalidAgeException("not valid");
10.     else
11.     System.out.println("welcome to vote");
12.     }

13.     public static void main(String args[]){
14.     try{
15.     validate(13);
16.     }catch(Exception m){System.out.println("Exception occured: "+m);}

17.     System.out.println("rest of the code...");
18.     }
19.     }
```

Output:Exception occured: InvalidAgeException:not valid

rest of the code...

Chapter 6: Java Threading

The java programming language allows us to create a program that contains one or more parts that can run simultaneously at the same time. This type of program is known as a multithreading program. Each part of this program is called a thread. Every thread defines a separate path of execution in java. A thread is explained in different ways, and a few of them are as specified below.

- A thread is a light wieght process.
- A thread may also be defined as follows.
- A thread is a subpart of a process that can run individually.

In java, multiple threads can run at a time, which enables the java to write multitasking programs. The multithreading is a specialized form of multitasking. All modern operating systems support multitasking. There are two types of multitasking, and they are as follows.

- **Process-based multitasking**
- **Thread-based multitasking**

It is important to know the difference between process-based and thread-based multitasking. Let's distinguish both.

Process-based multitasking	Thread-based multitasking
It allows the computer to run two or more programs concurrently	It allows the computer to run two or more threads concurrently
In this process is the smallest unit.	In this thread is the smallest unit.
Process is a larger unit.	Thread is a part of process.
Process is heavy weight.	Thread is light weight.
Process requires seperate address space for each.	Threads share same address space.
Process never gain access over idle time of CPU.	Thread gain access over idle time of CPU.
Inter process communication is expensive.	Inter thread communication is not expensive.

In the upcoming tutorials, we learn things like, what are the different phases in a thread life cycle? how to create threads in java? and how to synchronize multiple threads?

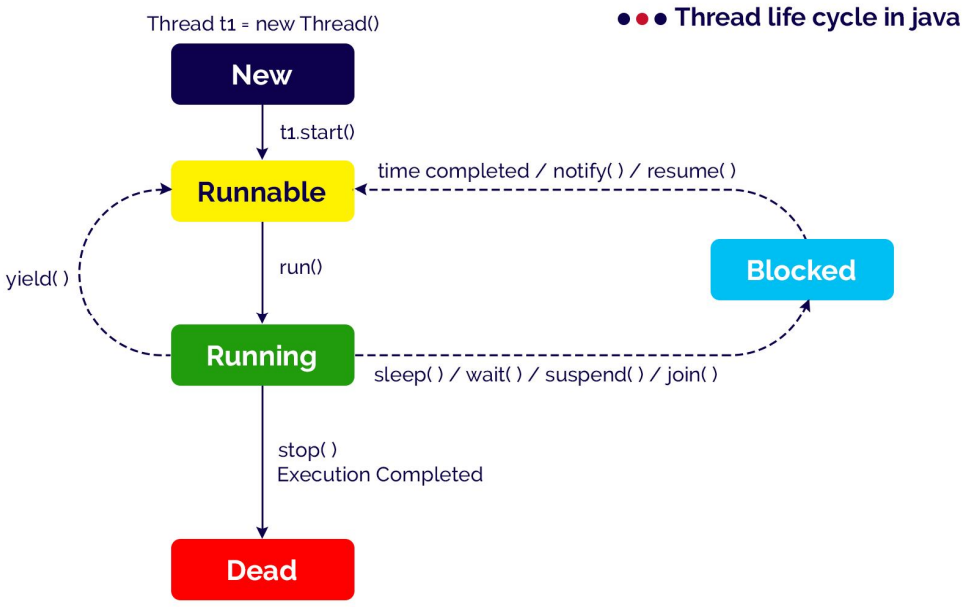
Why Threads are used?

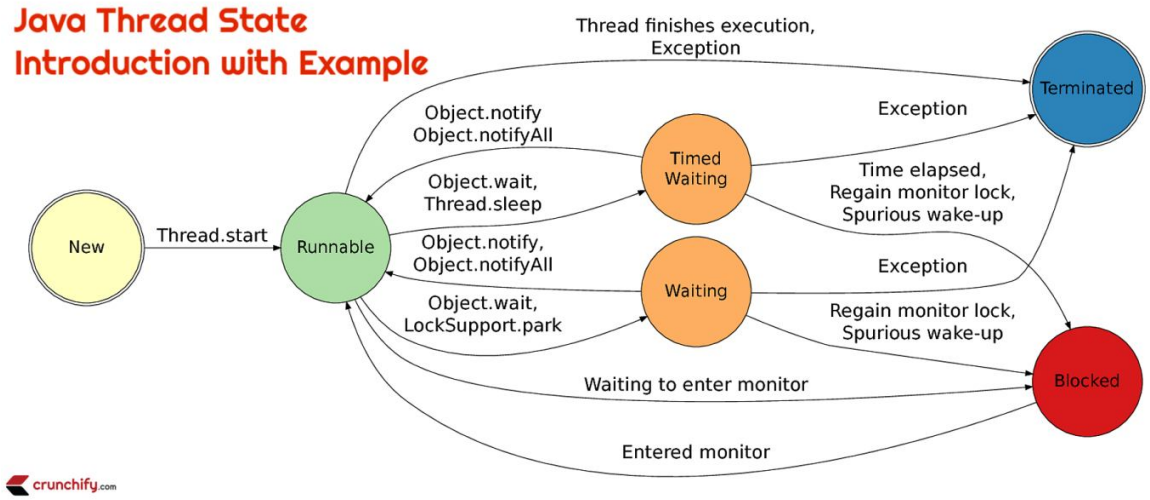
Now, we can understand why threads are being used as they had the advantage of being lightweight and can provide communication between multiple threads at a Low Cost contributing to effective multi-tasking within a shared memory environment.

Life Cycle Of Thread

There are different states Thread transfers into during its lifetime, let us know about those states in the following lines: in its lifetime, a thread undergoes the following states, namely:

1. New State
2. Active State
3. Waiting/Blocked State
4. Timed Waiting State
5. Terminated State





We can see the working of different states in a Thread in the above Diagram, let us know in detail each and every state:

1. New State

By Default, a Thread will be a new state, in this state, code has not yet been run and the execution process is not yet initiated.

2. Active State

A Thread that is a new state by default gets transferred to Active state when it invokes the start() method, his Active state contains two sub-states namely :

- **Runnable State:** In This State, The Thread is ready to run at any given time and it's the job of the Thread Scheduler to provide the thread time for the runnable state preserved threads. A program that has obtained Multithreading shares slices of time intervals which are shared between threads hence, these threads run for some short span of time and wait in the runnable state to get their schedules slice of a time interval.
- **Running State:** When The Thread Receives CPU allocated by Thread Scheduler, it transfers from "Runnable" state to "Running" state. and after the expiry of its given time slice session, it again moves back to the "Runnable" state and waits for its next time slice.

3. Waiting/Blocked State

If a Thread is inactive but on a temporary time, then either it is at waiting or blocked state, for example, if there are two threads, T1 and T2 where T1 need to communicate to the camera and other thread T2 already using a camera to scan then T1 waits until T2 Thread completes its work, at this state T1 is parked in waiting for the state, and in another scenario, the user called two Threads T2 and T3 with the same functionality and both had same time slice given by Thread Scheduler then both Threads T1, T2 is in a blocked state. When there are multiple threads parked in Blocked/Waiting state Thread Scheduler clears Queue by rejecting unwanted Threads and allocating CPU on a priority basis.

4. Timed Waiting State

Sometimes the longer duration of waiting for threads causes starvation, if we take an example like there are two threads T1, T2 waiting for CPU and T1 is undergoing Critical Coding operation and if it does not exit CPU until its operation gets executed then T2 will be exposed to longer waiting with undetermined certainty, In order to avoid this starvation situation, we had Timed Waiting for the state to avoid that kind of scenario as in Timed Waiting, each thread has a time period for which sleep() method is invoked and after the time expires the Threads starts executing its task.

5. Terminated State

A thread will be in Terminated State, due to the below reasons:

- Termination is achieved by a Thread when it finishes its task Normally.
- Sometimes Threads may be terminated due to unusual events like segmentation faults, exceptions...etc. and such kind of Termination can be called Abnormal Termination.
- A terminated Thread means it is dead and no longer available.

What is Main Thread?

As we are familiar that, we create Main Method in each and every Java Program, which acts as an entry point for the code to get executed by JVM, Similarly in this Multithreading Concept, Each Program has one Main Thread which was provided by default by JVM, hence whenever a program is being created in java, JVM provides the Main Thread for its Execution.

How to Create Threads using Java Programming Language?

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).

18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. **public void run():** is used to perform action for a thread.

Starting a thread:

The **start() method** of Thread class is used to start a newly created thread. It performs the following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

1) Java Thread Example by extending Thread class

FileName: Multi.java

```
1. class Multi extends Thread{
2. public void run(){
3. System.out.println("thread is running...");
4. }
5. public static void main(String args[]){
6. Multi t1=new Multi();
7. t1.start();
8.  }
9. }
```

Output:

```
thread is running...
```

2) Java Thread Example by implementing Runnable interface

FileName: Multi3.java

```
1. class Multi3 implements Runnable{
2. public void run(){
```

```
3. System.out.println("thread is running...");
4. }
5.
6. public static void main(String args[]){
7. Multi3 m1=new Multi3();
8. Thread t1 =new Thread(m1);   // Using the constructor Thread(Runnable r)
9. t1.start();
10. }
11. }
```

Output:

```
thread is running...
```

If you are not extending the Thread class, your class object would not be treated as a thread object. So you need to explicitly create the Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

3) Using the Thread Class: Thread(String Name)

We can directly use the Thread class to spawn new threads using the constructors defined above.

FileName: MyThread1.java

```
1. public class MyThread1
2. {
3. // Main method
4. public static void main(String argvs[])
5. {
6. // creating an object of the Thread class using the constructor Thread(String name)
7. Thread t= new Thread("My first thread");
8.
9. // the start() method moves the thread to the active state
10. t.start();
11. // getting the thread name by invoking the getName() method
12. String str = t.getName();
13. System.out.println(str);
14. }
15. }
```

Output:

```
My first thread
```

4) Using the Thread Class: Thread(Runnable r, String name)

Observe the following program.

FileName: MyThread2.java

```
1. public class MyThread2 implements Runnable
2. {
3. public void run()
4. {
5. System.out.println("Now the thread is running ...");
6. }
7.
8. // main method
9. public static void main(String argvs[])
10. {
11. // creating an object of the class MyThread2
12. Runnable r1 = new MyThread2();
13.
14. // creating an object of the class Thread using Thread(Runnable r, String name)
15. Thread th1 = new Thread(r1, "My new thread");
16.
17. // the start() method moves the thread to the active state
18. th1.start();
19.
20. // getting the thread name by invoking the getName() method
21. String str = th1.getName();
22. System.out.println(str);
23. }
24. }
```

Java Thread Priority

In a java programming language, every thread has a property called priority. Most of the scheduling algorithms use the thread priority to schedule the execution sequence. In java, the thread priority range from 1 to 10. Priority 1 is considered as the lowest priority, and priority 10 is considered as the highest priority. The thread with more priority allocates the processor first.

The java programming language Thread class provides two methods setPriority(int), and getPriority( ) to handle thread priorities.

The Thread class also contains three constants that are used to set the thread priority, and they are listed below.

- *MAX_PRIORITY* - It has the value 10 and indicates highest priority.
- *NORM_PRIORITY* - It has the value 5 and indicates normal priority.
- *MIN_PRIORITY* - It has the value 1 and indicates lowest priority.

 The default priority of any thread is 5 (i.e. NORM_PRIORITY).

setPriority( ) method

The setPriority( ) method of Thread class used to set the priority of a thread. It takes an integer range from 1 to 10 as an argument and returns nothing (void).

The regular use of the setPriority( ) method is as follows.

Example

```
threadObject.setPriority(4);
or
threadObject.setPriority(MAX_PRIORITY);
```

getPriority( ) method

The getPriority( ) method of Thread class used to access the priority of a thread. It does not takes anyargument and returns name of the thread as String.

The regular use of the getPriority( ) method is as follows.

Example

```
String threadName = threadObject.getPriority();
```

Look at the following example program.

Example

```
class SampleThread extends Thread{
        public void run() {
                System.out.println("Inside SampleThread");
                System.out.println("Current Thread: " +
Thread.currentThread().getName());
        }
}

public class My_Thread_Test {

        public static void main(String[] args) {
                SampleThread threadObject1 = new SampleThread();
                SampleThread threadObject2 = new SampleThread();
                threadObject1.setName("first");
                threadObject2.setName("second");

                threadObject1.setPriority(4);
                threadObject2.setPriority(Thread.MAX_PRIORITY);
```

```
                threadObject1.start();
                threadObject2.start();

        }
}
```

Java Thread Synchronisation

The java programming language supports multithreading. The problem of shared resources occurs when two or more threads get execute at the same time. In such a situation, we need some way to ensure that the shared resource will be accessed by only one thread at a time, and this is performed by using the concept called synchronization.

 The synchronization is the process of allowing only one thread to access a shared resource at a time.

In java, the synchronization is achieved using the following concepts.

- Mutual Exclusion
- Inter thread communication

In this tutorial, we discuss mutual exclusion only, and the interthread communication will be discussed in the next tutorial.

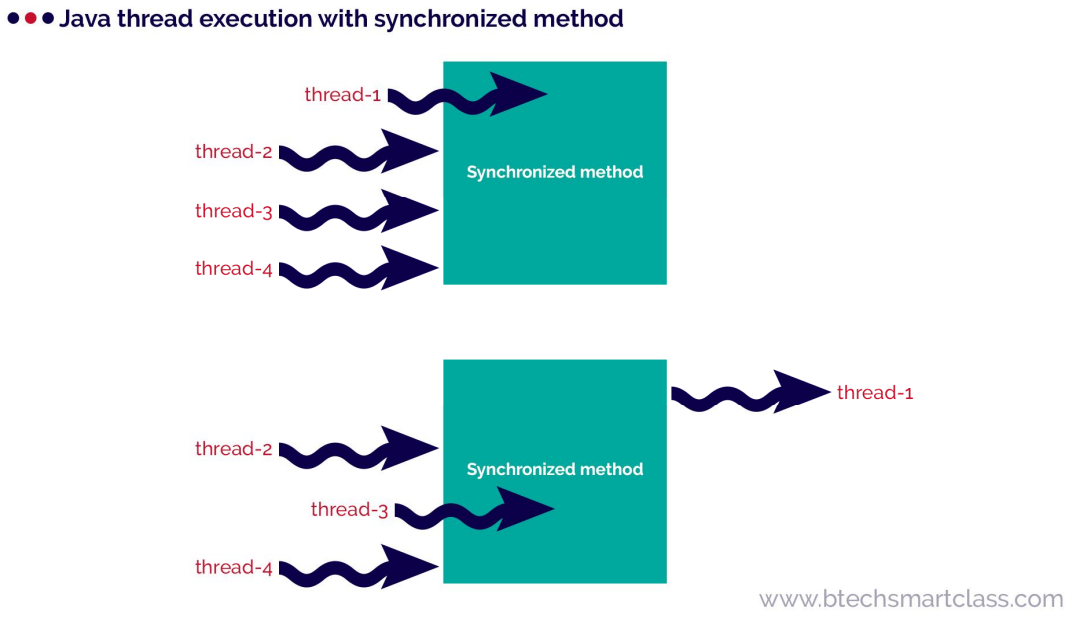
Mutual Exclusion

Using the mutual exclusion process, we keep threads from interfering with one another while they accessing the shared resource. In java, mutual exclusion is achieved using the following concepts.

- **Synchronized method**
- **Synchronized block**

Synchronized method

When a method created using a synchronized keyword, it allows only one object to access it at a time. When an object calls a synchronized method, it put a lock on that method so that other objects or thread that are trying to call the same method must wait, until the lock is released. Once the lock is released on the shared resource, one of the threads among the waiting threads will be allocated to the shared resource.

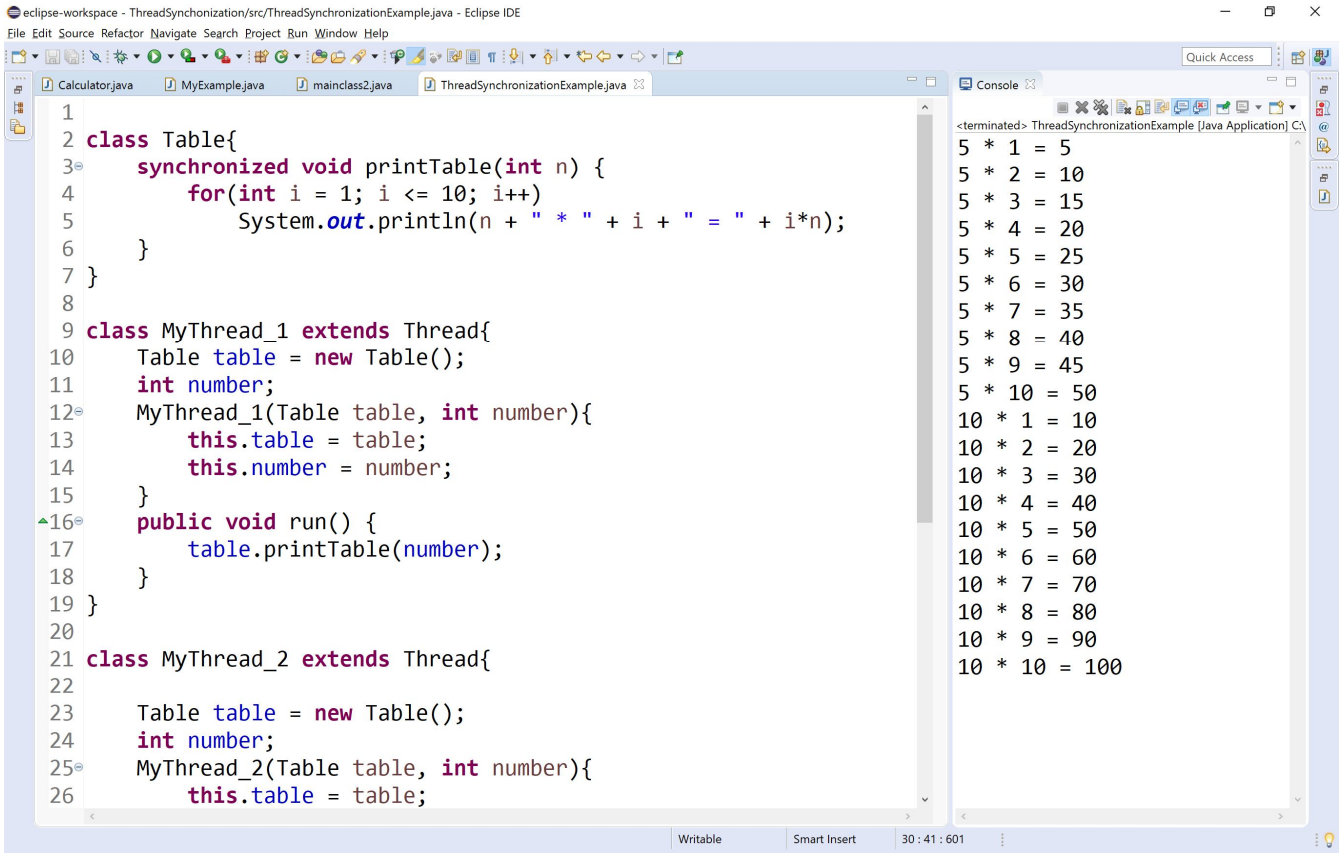


In the above image, initially the thread-1 is accessing the synchronized method and other threads (thread-2, thread-3, and thread-4) are waiting for the resource (synchronized method). When thread-1 completes it task, then one of the threads that are waiting is allocated with the synchronized method, in the above it is thread-3.

Example

```
class Table{
        synchronized void printTable(int n) {
                for(int i = 1; i <= 10; i++)
                        System.out.println(n + " * " + i + " = " + i*n);
        }
}

class MyThread_1 extends Thread{
        Table table = new Table();
        int number;
        MyThread_1(Table table, int number){
                this.table = table;
                this.number = number;
        }
        public void run() {
                table.printTable(number);
        }
}

class MyThread_2 extends Thread{

        Table table = new Table();
        int number;
        MyThread_2(Table table, int number){
```

```
                this.table = table;
                this.number = number;
        }
        public void run() {
                table.printTable(number);
        }
}
public class ThreadSynchronizationExample {

        public static void main(String[] args) {
                Table table = new Table();
                MyThread_1 thread_1 = new MyThread_1(table, 5);
                MyThread_2 thread_2 = new MyThread_2(table, 10);
                thread_1.start();
                thread_2.start();
        }
}
```



Synchronized block

The synchronized block is used when we want to synchronize only a specific sequence of lines in a method. For example, let's consider a method with 20 lines of code where we want to synchronize only a sequence of 5 lines code, we use the synchronized block.

The folllowing syntax is used to define a synchronized block.

Syntax

```
synchronized(object){
    ...
    block code
    ...
}
```

Look at the following example code to illustrate synchronized block.

Example

```
import java.util.*;

class NameList {
    String name = "";
    public int count = 0;

    public void addName(String name, List<String> namesList){
        synchronized(this){
            this.name = name;
            count++;
        }
        namesList.add(name);
    }

    public int getCount(){
        return count;
    }
}
public class SynchronizedBlockExample {
        public static void main (String[] args)
    {
        NameList namesList_1 = new NameList();
        NameList namesList_2 = new NameList();
        List<String> list = new ArrayList<String>();
        namesList_1.addName("Rama", list);
        namesList_2.addName("Seetha", list);
        System.out.println("Thread1: " + namesList_1.name + ", " +
namesList_1.getCount() + "\n");
        System.out.println("Thread2: " + namesList_2.name + ", " +
namesList_2.getCount() + "\n");
    }
}
```

Java Inter Thread Communication

Inter thread communication is the concept where two or more threads communicate to solve the problem of polling. In java, polling is the situation to check some condition repeatedly, to take appropriate action, once the condition is true. That means, in inter-thread communication, a thread waits until a condition becomes true such that other threads can execute its task. The inter-thread communication allows the synchronized threads to communicate with each other.

Java provides the following methods to achieve inter thread communication.

- wait( )
- notify( )
- notifyAll( )

The following table gives detailed description about the above methods.

Method	Description
void wait( )	It makes the current thread to pause its execution until other thread in the same monitor calls notify( )
void notify( )	It wakes up the thread that called wait( ) on the same object.
void notifyAll()	It wakes up all the threads that called wait( ) on the same object.

 Calling notify( ) or notifyAll( ) does not actually give up a lock on a resource.

Let's look at an example problem of producer and consumer. The producer produces the item and the consumer consumes the same. But here, the consumer can not consume until the producer produces the item, and producer can not produce until the consumer consumes the item that already been produced. So here, the consumer has to wait until the producer produces the item, and the producer also needs to wait until the consumer consumes the same. Here we use the inter-thread communication to implement the producer and consumer problem.

The sample implementation of producer and consumer problem is as follows.

Example

```
class ItemQueue {
        int item;
        boolean valueSet = false;

        synchronized int getItem()

        {
                while (!valueSet)
                        try {
                                wait();
                        } catch (InterruptedException e) {
                                System.out.println("InterruptedException caught");
                        }
                System.out.println("Consummed:" + item);
```

```
                valueSet = false;
                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e) {
                        System.out.println("InterruptedException caught");
                }
                notify();
                return item;
        }

        synchronized void putItem(int item) {
                while (valueSet)
                        try {
                                wait();
                        } catch (InterruptedException e) {
                                System.out.println("InterruptedException caught");
                        }
                this.item = item;
                valueSet = true;
                System.out.println("Produced: " + item);
                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e) {
                        System.out.println("InterruptedException caught");
                }
                notify();
        }
}

class Producer implements Runnable{
        ItemQueue itemQueue;
        Producer(ItemQueue itemQueue){
                this.itemQueue = itemQueue;
                new Thread(this, "Producer").start();
        }
        public void run() {
                int i = 0;
                while(true) {
                        itemQueue.putItem(i++);
                }
        }
}
class Consumer implements Runnable{

        ItemQueue itemQueue;
        Consumer(ItemQueue itemQueue){
                this.itemQueue = itemQueue;
                new Thread(this, "Consumer").start();
        }
        public void run() {
                while(true) {
                        itemQueue.getItem();
                }
        }
}

class ProducerConsumer{
```

```
        public static void main(String args[]) {
                ItemQueue itemQueue = new ItemQueue();
                new Producer(itemQueue);
                new Consumer(itemQueue);

        }
}
```

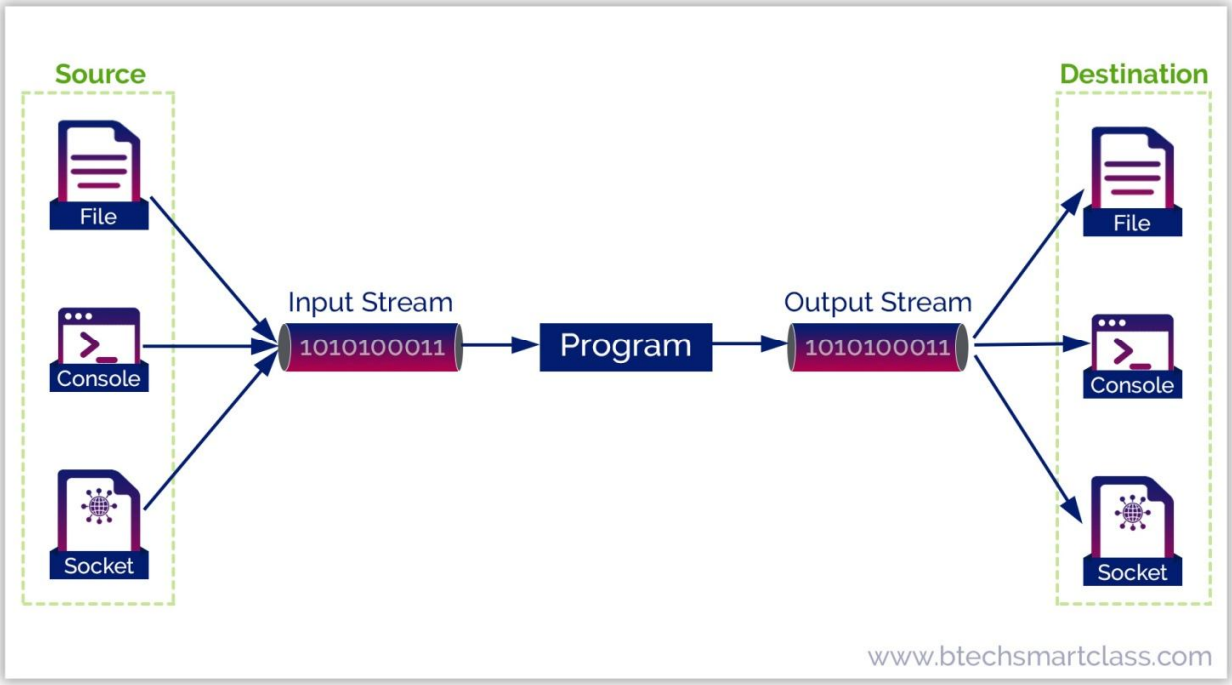
Chapter 7: Java I/O (File Handling)

Stream in java

In java, the IO operations are performed using the concept of streams. Generally, a stream means a continuous flow of data. In java, a stream is a logical container of data that allows us to read from and write to it. A stream can be linked to a data source, or data destination, like a console, file or network connection by java IO system. The stream-based IO operations are faster than normal IO operations.

The Stream is defined in the java.io package.

To understand the functionality of java streams, look at the following picture.



In java, the stream-based IO operations are performed using two separate streams input stream and output stream. The input stream is used for input operations, and the output stream is used for output operations. The java stream is composed of bytes.

In Java, every program creates 3 streams automatically, and these streams are attached to the console.

- **System.out: standard output stream for console output operations.**
- **System.in: standard input stream for console input operations.**
- **System.err: standard error stream for console error output operations.**

The Java streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.

Java provides two types of streams, and they are as follows.

- **Byte Stream**
- **Character Stream**

The following picture shows how streams are categorized, and various built-in classes used by the java IO system.

Byte Stream in java

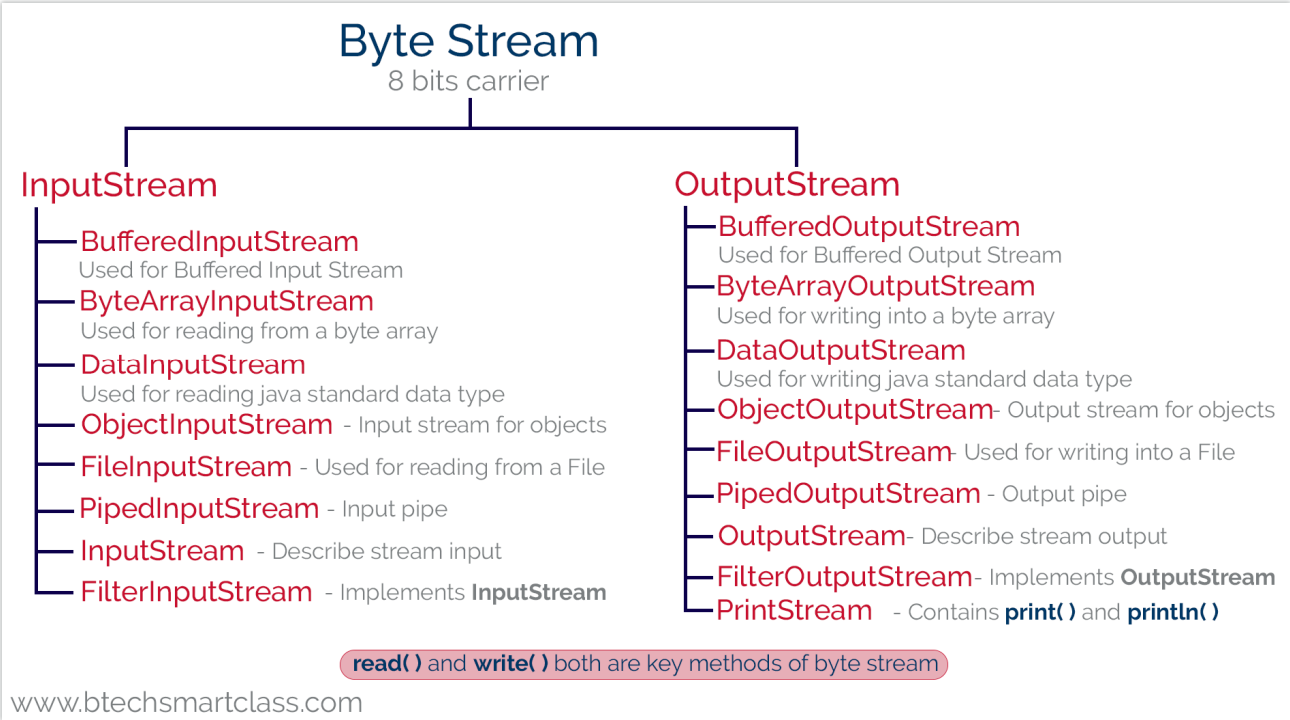
In java, the byte stream is an 8 bits carrier. The byte stream in java allows us to transmit 8 bits of data.

In Java 1.0 version all IO operations were byte oriented, there was no other stream (character stream).

The java byte stream is defined by two abstract classes, InputStream and OutputStream. The InputStream class used for byte stream based input operations, and the OutputStream class used for byte stream based output operations.

The InputStream and OutputStream classes have several concreate classes to perform various IO operations based on the byte stream.

The following picture shows the classes used for byte stream operations.



InputStream class

The InputStream class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

S.No.	Method with Description
1	**int available()** It returns the number of bytes that can be read from the input stream.
2	**int read()** It reads the next byte from the input stream.
3	**int read(byte[] b)** It reads a chunk of bytes from the input stream and store them in its byte array, b.
4	**void close()** It closes the input stream and also frees any resources connected with this input stream.

OutputStream class

The OutputStream class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

S.No.	Method with Description
1	**void write(int n)** It writes byte(contained in an int) to the output stream.
2	**void write(byte[] b)** It writes a whole byte array(b) to the output stream.
3	**void flush()** It flushes the output steam by forcing out buffered bytes to be written out.
4	**void close()** It closes the output stream and also frees any resources connected with this output stream.

Reading data using BufferedInputStream

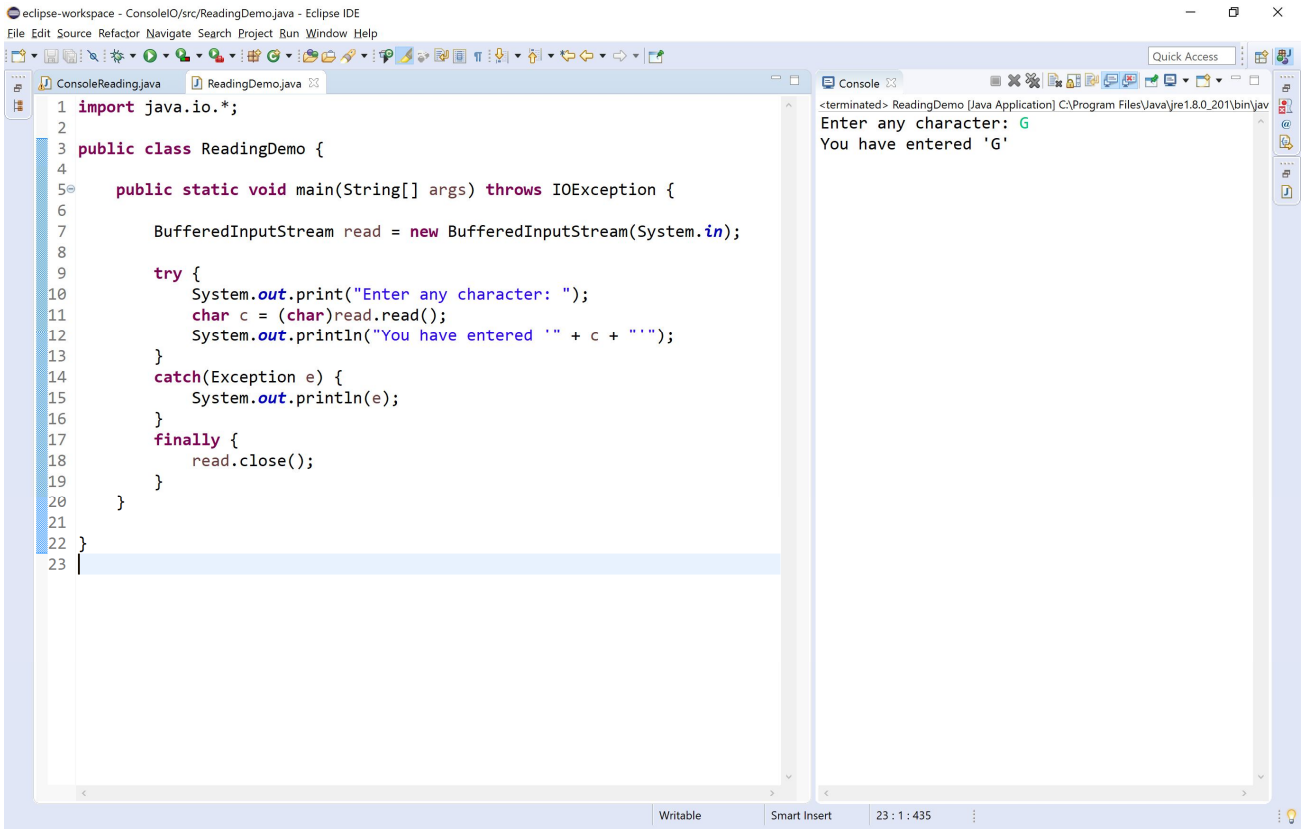
We can use the BufferedInputStream class to read data from the console. The BufferedInputStream class use a method read( ) to read a value from the console, or file, or socket.

Let's look at an example code to illustrate reading data using BufferedInputStream.

Example 1 - Reading from console

```
import java.io.*;

public class ReadingDemo {

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

                BufferedInputStream read = new BufferedInputStream(System.in);

                try {
                        System.out.print("Enter any character: ");
                        char c = (char)read.read();
                        System.out.println("You have entered '" + c + "'");
```

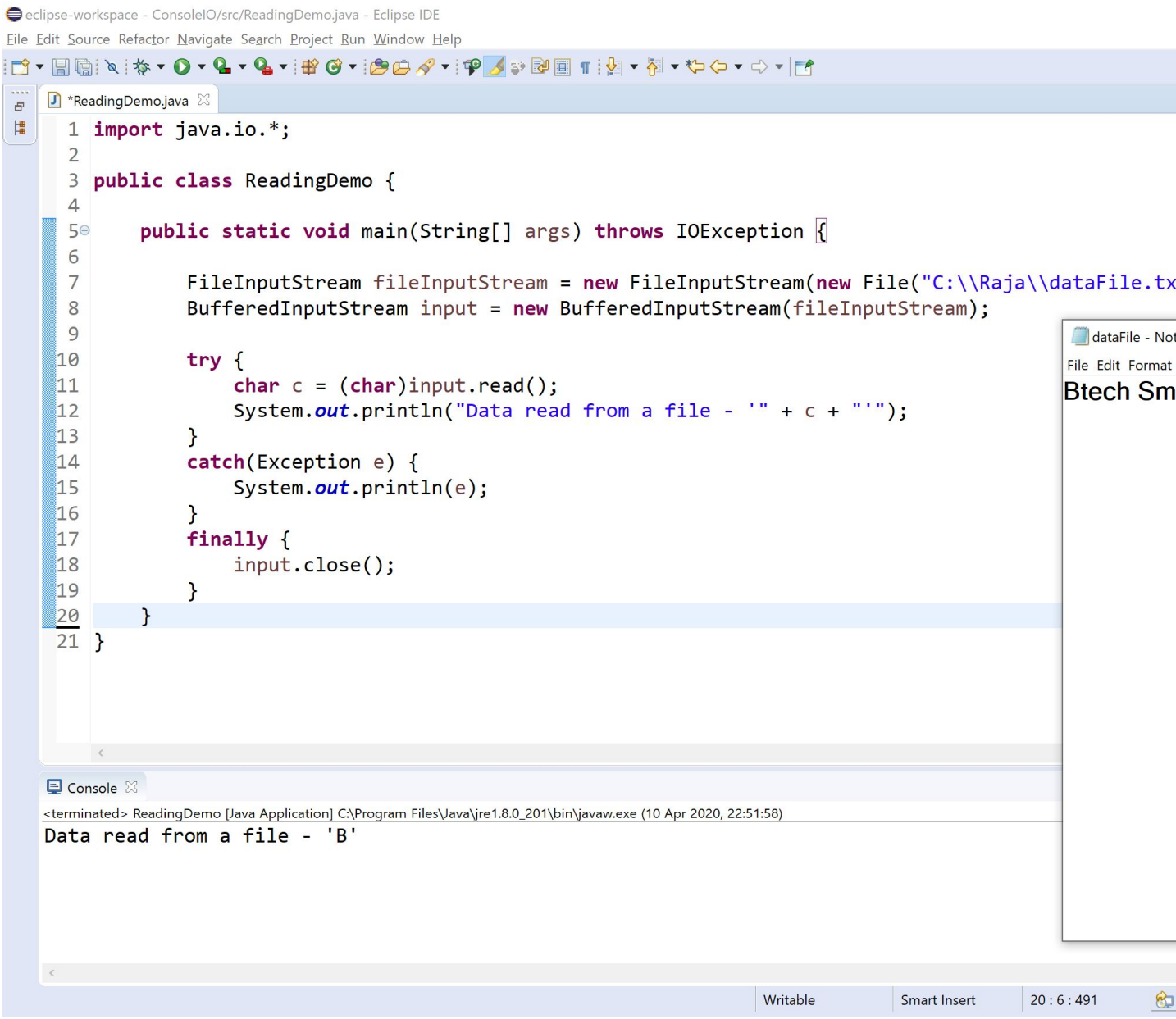
```
                }
                catch(Exception e) {
                        System.out.println(e);
                }
                finally {
                        read.close();
                }
        }

}
```



Example 2 - Reading from a file

```
import java.io.*;

public class ReadingDemo {

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

                FileInputStream fileInputStream = new FileInputStream(new
File("C:\\Raja\\dataFile.txt"));
                BufferedInputStream input = new
BufferedInputStream(fileInputStream);
```

```
                try {
                        char c = (char)input.read();
                        System.out.println("Data read from a file - '" + c + "'");
                }
                catch(Exception e) {
                        System.out.println(e);
                }
                finally {
                        input.close();
                }
        }
}
```



Writing data using BufferedOutputStream

We can use the BufferedOutputStream class to write data into the console, file, socket. The BufferedOutputStream class use a method write( ) to write data.

Let's look at an example code to illustrate writing data into a file using BufferedOutputStream.

Example - Writing data into a file

```
import java.io.*;

public class WritingDemo {

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

		String data = "Java tutorials by BTech Smart Class";
		BufferedOutputStream out = null;
		try {
			FileOutputStream fileOutputStream = new FileOutputStream(new
File("C:\\Raja\\dataFile.txt"));
			out = new BufferedOutputStream(fileOutputStream);

			out.write(data.getBytes());
			System.out.println("Writing data into a file is success!");

		}
		catch(Exception e) {
			System.out.println(e);
		}
		finally {
			out.close();
		}
	}
}
```

Character Stream in java

In java, when the IO stream manages 16-bit Unicode characters, it is called a character stream. The unicode set is basically a type of character set where each character corresponds to a specific numeric value within the given character set, and every programming language has a character set.

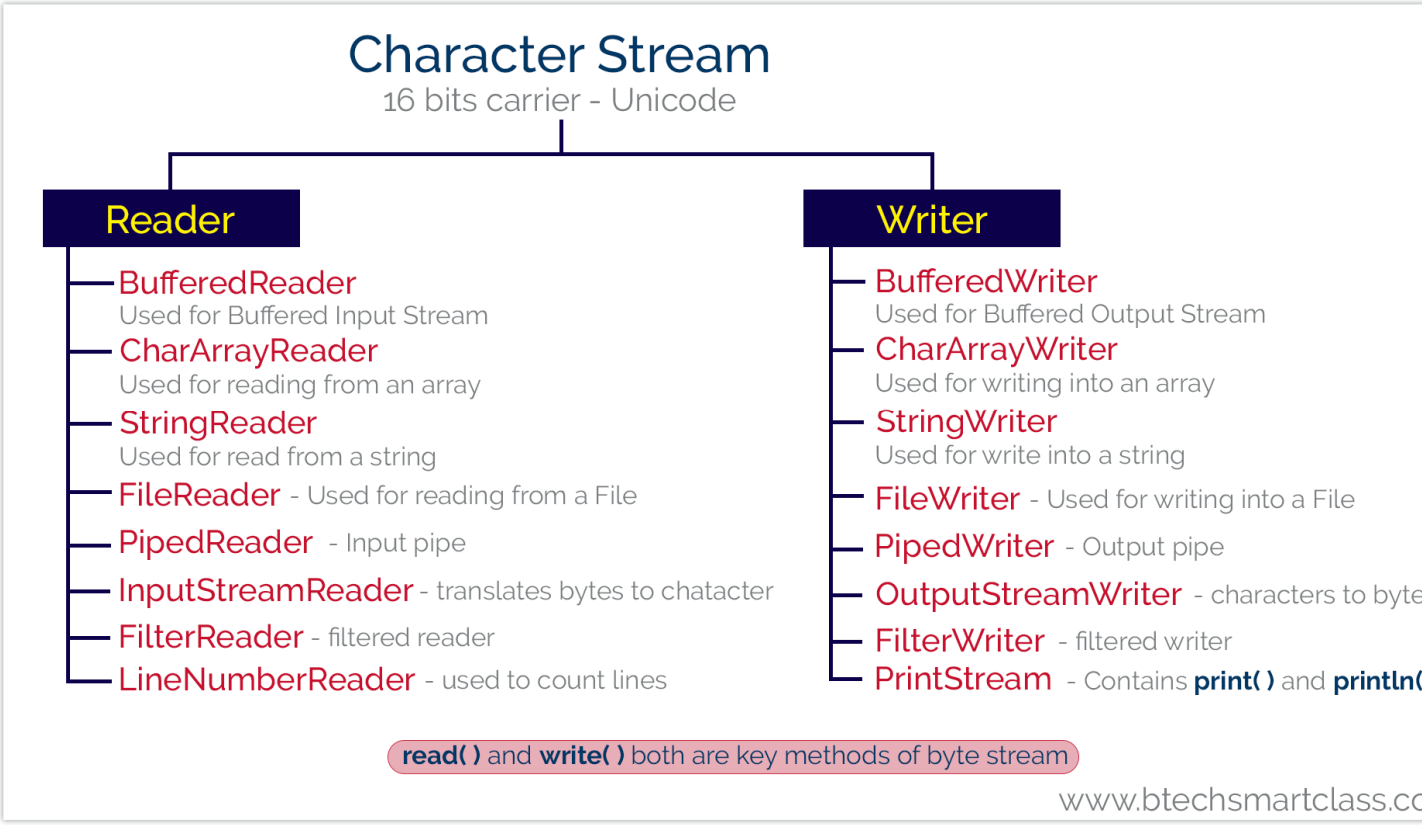
In java, the character stream is a 16 bits carrier. The character stream in java allows us to transmit 16 bits of data.

The character stream was introduced in Java 1.1 version. The charater stream

The java character stream is defined by two abstract classes, Reader and Writer. The Reader class used for character stream based input operations, and the Writer class used for charater stream based output operations.

The Reader and Writer classes have several concreate classes to perform various IO operations based on the character stream.

The following picture shows the classes used for character stream operations.



Reader class

The Reader class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

S.No.	Method with Description
1	**int read()** It reads the next character from the input stream.

S.No.	Method with Description
2	**int read(char[] cbuffer)** It reads a chunk of charaters from the input stream and store them in its byte array, cbuffer.
3	**int read(char[] cbuf, int off, int len)** It reads charaters into a portion of an array.
4	**int read(CharBuffer target)** It reads charaters into into the specified character buffer.
5	**String readLine()** It reads a line of text. A line is considered to be terminated by any oneof a line feed ('\n'), a carriage return ('\r'), or a carriage returnfollowed immediately by a linefeed.
6	**boolean ready()** It tells whether the stream is ready to be read.
7	**void close()** It closes the input stream and also frees any resources connected with this input stream.

Writer class

The Writer class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

S.No.	Method with Description
1	**void flush()** It flushes the output steam by forcing out buffered bytes to be written out.
2	**void write(char[] cbuf)** It writes a whole array(cbuf) to the output stream.
3	**void write(char[] cbuf, int off, int len)** It writes a portion of an array of characters.
4	**void write(int c)** It writes single character.

S.No.	Method with Description
5	**void write(String str)** It writes a string.
6	**void write(String str, int off, int len)** It writes a portion of a string.
7	**Writer append(char c)** It appends the specified character to the writer.
8	**Writer append(CharSequence csq)** It appends the specified character sequence to the writer
9	**Writer append(CharSequence csq, int start, int end)** It appends a subsequence of the specified character sequence to the writer.
10	**void close()** It closes the output stream and also frees any resources connected with this output stream.

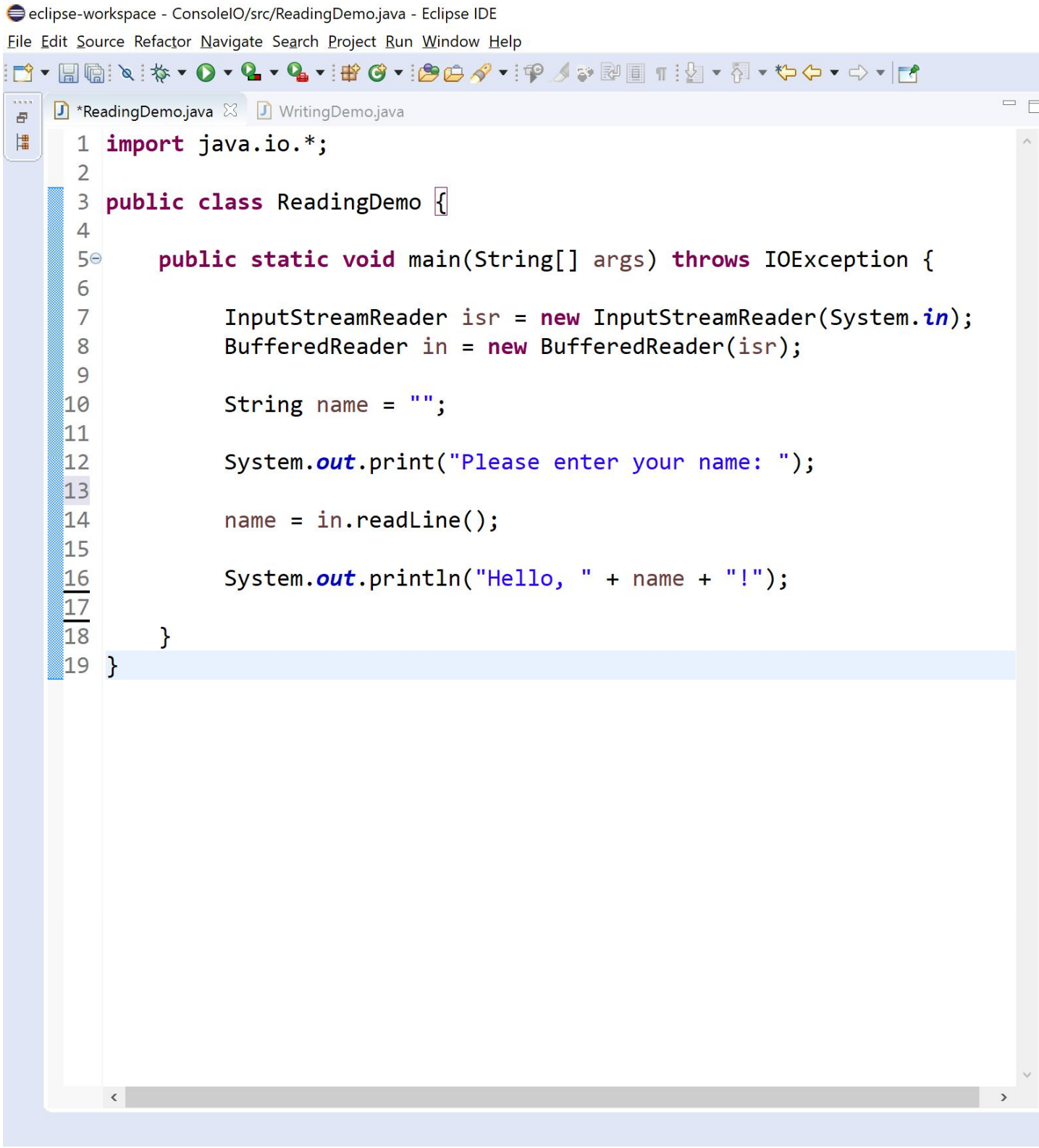
Reading data using BufferedReader

We can use the BufferedReader class to read data from the console. The BufferedInputStream class needs InputStreamReaderclass. The BufferedReader use a method read( ) to read a value from the console, or file, or socket.

Let's look at an example code to illustrate reading data using BufferedReader.

Example 1 - Reading from console

```
import java.io.*;

public class ReadingDemo {

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

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

            String name = "";

            System.out.print("Please enter your name: ");

            name = in.readLine();

            System.out.println("Hello, " + name + "!");      }}
```


eclipse-workspace - ConsoleIO/src/ReadingDemo.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
*ReadingDemo.java
WritingDemo.java
1 import java.io.*;
2
3 public class ReadingDemo {
4
5 public static void main(String[] args) throws IOException {
6
7 InputStreamReader isr = new InputStreamReader(System.in);
8 BufferedReader in = new BufferedReader(isr);
9
10 String name = "";
11
12 System.out.print("Please enter your name: ");
13
14 name = in.readLine();
15
16 System.out.println("Hello, " + name + "!");
17
18 }
19 }

Example 2 - Reading from a file

```
import java.io.*;

public class ReadingDemo {

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

                Reader in = new FileReader();

                try {
                        char c = (char)input.read();
                        System.out.println("Data read from a file - '" + c + "'");
                }
                catch(Exception e) {
                        System.out.println(e);
                }
                finally {
                        input.close();
                }
        }
}
```

Console IO Operations in Java

Reading console input in java

In java, there are three ways to read console input. Using the 3 following ways, we can read input data from the console.

- Using BufferedReader class
- Using Scanner class
- Using Console class

Let's explore the each method to read data with example.

1. Reading console input using BufferedReader class in java

Reading input data using the BufferedReader class is the traditional technique. This way of the reading method is used by wrapping the System.in (standard input stream) in an InputStreamReader which is wrapped in a BufferedReader, we can read input from the console.

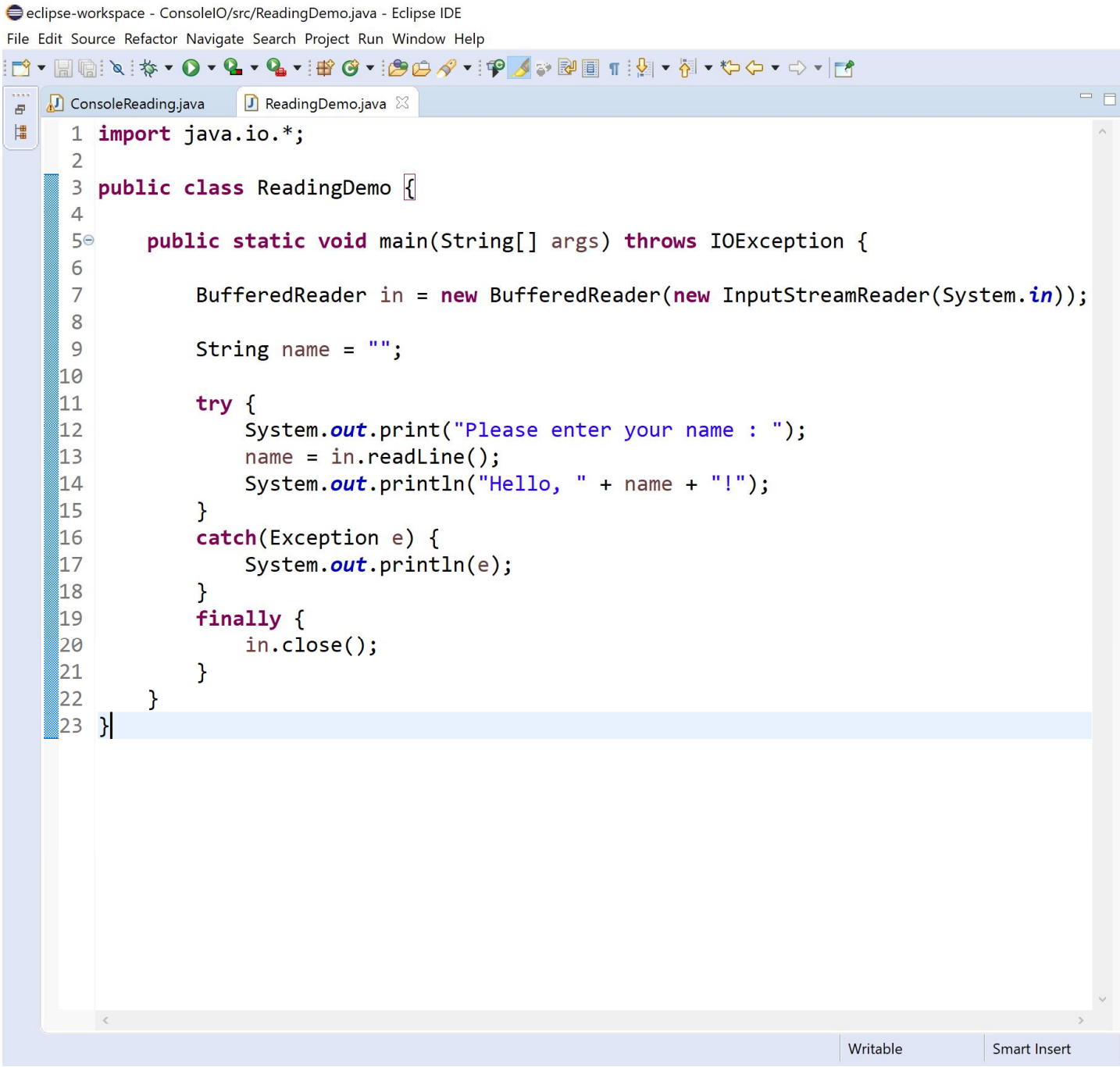
The BufferedReader class has defined in the java.io package.

Consider the following example code to understand how to read console input using BufferedReader class.

Example

```
import java.io.*;

public class ReadingDemo {

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

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

		String name = "";

		try {
			System.out.print("Please enter your name : ");
			name = in.readLine();
			System.out.println("Hello, " + name + "!");
		}
		catch(Exception e) {
			System.out.println(e);
		}
		finally {
```

```
                    in.close();
                }
            }
}
```


eclipse-workspace - ConsoleIO/src/ReadingDemo.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
ConsoleReading.java
ReadingDemo.java
1 import java.io.*;
2
3 public class ReadingDemo {
4
5 public static void main(String[] args) throws IOException {
6
7 BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
8
9 String name = "";
10
11 try {
12 System.out.print("Please enter your name : ");
13 name = in.readLine();
14 System.out.println("Hello, " + name + "!");
15 }
16 catch(Exception e) {
17 System.out.println(e);
18 }
19 finally {
20 in.close();
21 }
22 }
23 }
Writable
Smart Insert

2. Reading console input using Scanner class in java

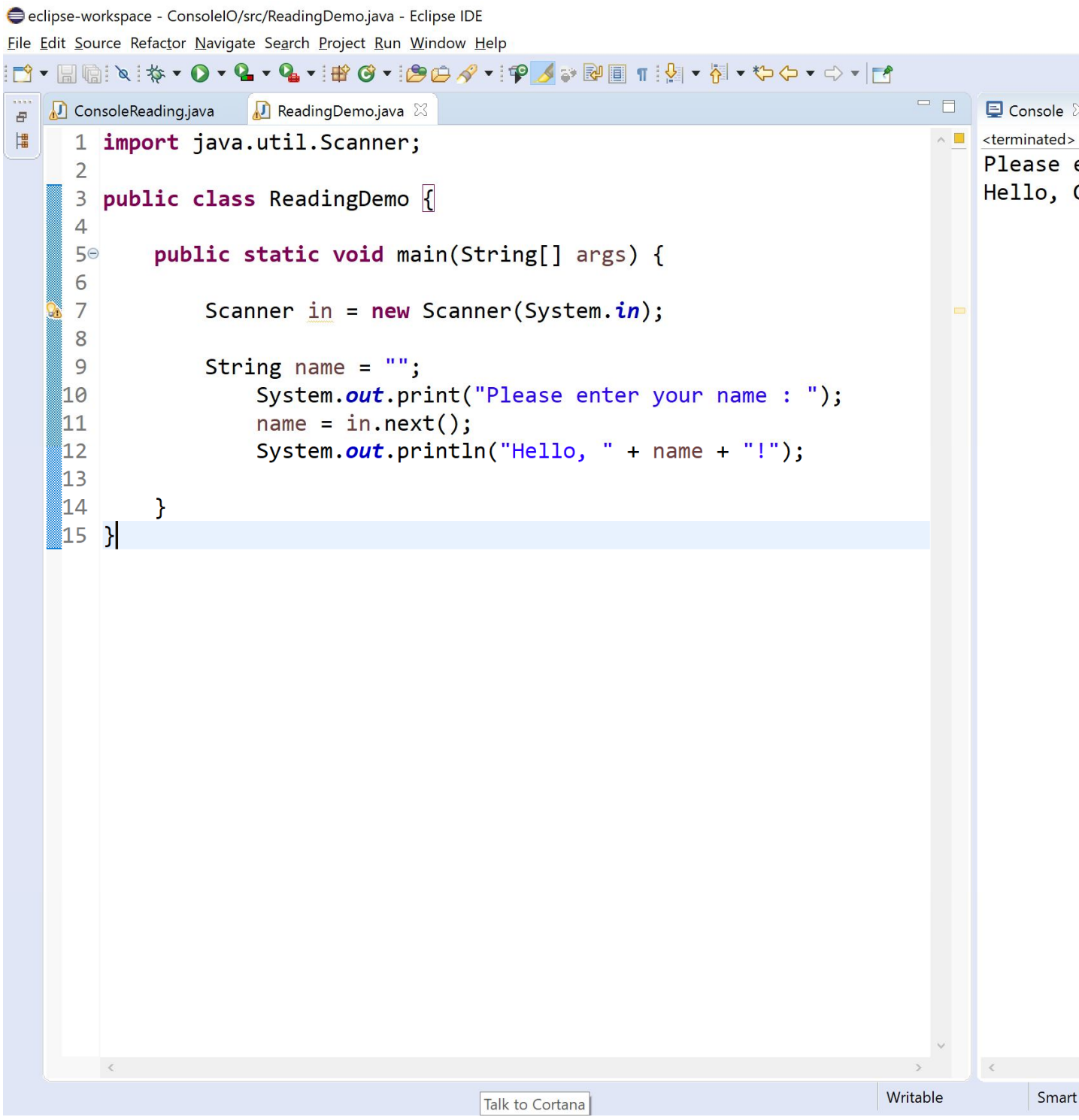
Reading input data using the Scanner class is the most commonly used method. This way of the reading method is used by wrapping the System.in (standard input stream) which is wrapped in a Scanner, we can read input from the console.

The Scanner class has defined in the java.util package.

Consider the following example code to understand how to read console input using Scanner class.

Example

```
import java.util.Scanner;

public class ReadingDemo {

	public static void main(String[] args) {

		Scanner in = new Scanner(System.in);

		String name = "";
			System.out.print("Please enter your name : ");
			name = in.next();
			System.out.println("Hello, " + name + "!");

	}
}
```


eclipse-workspace - ConsoleIO/src/ReadingDemo.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
ConsoleReading.java
ReadingDemo.java
Console
<terminated>
Please e
Hello, C
1 import java.util.Scanner;
2
3 public class ReadingDemo {
4
5 public static void main(String[] args) {
6
7 Scanner in = new Scanner(System.in);
8
9 String name = "";
10 System.out.print("Please enter your name : ");
11 name = in.next();
12 System.out.println("Hello, " + name + "!");
13
14 }
15 }
Talk to Cortana
Writable
Smart

3. Reading console input using Console class in java

Reading input data using the Console class is the most commonly used method. This class was introduced in Java 1.6 version.

The Console class has defined in the java.io package.

Consider the following example code to understand how to read console input using Console class.

Example

```
import java.io.*;

public class ReadingDemo {

        public static void main(String[] args) {

                String name;
         Console con = System.console();

                if(con != null) {
                        name = con.readLine("Please enter your name : ");
                        System.out.println("Hello, " + name + "!!");
                }
                else {
                        System.out.println("Console not available.");
                }
        }
}
```

Writing console output in java

In java, there are two methods to write console output. Using the 2 following methods, we can write output data to the console.

- Using print() and println() methods
- Using write() method

Let's explore the each method to write data with example.

1. Writing console output using print() and println() methods

The PrintStream is a bult-in class that provides two methods print() and println() to write console output. The print() and println() methods are the most widely used methods for console output.

Both print() and println() methods are used with System.out stream.

The print() method writes console output in the same line. This method can be used with console output only.

The println() method writes console output in a separete line (new line). This method can be used with console ans also with other output sources.

Let's look at the following code to illustrate print() and println() methods.

Example

```
public class WritingDemo {

	public static void main(String[] args) {

		int[] list = new int[5];

		for(int i = 0; i < 5; i++)
			list[i] = i*10;

		for(int i:list)
			System.out.print(i);   //prints in same line

		System.out.println("");
		for(int i:list)
			System.out.println(i); //Prints in separate lines

	}
}
```

2. Writing console output using write() method

Alternatively, the PrintStream class provides a method write() to write console output.

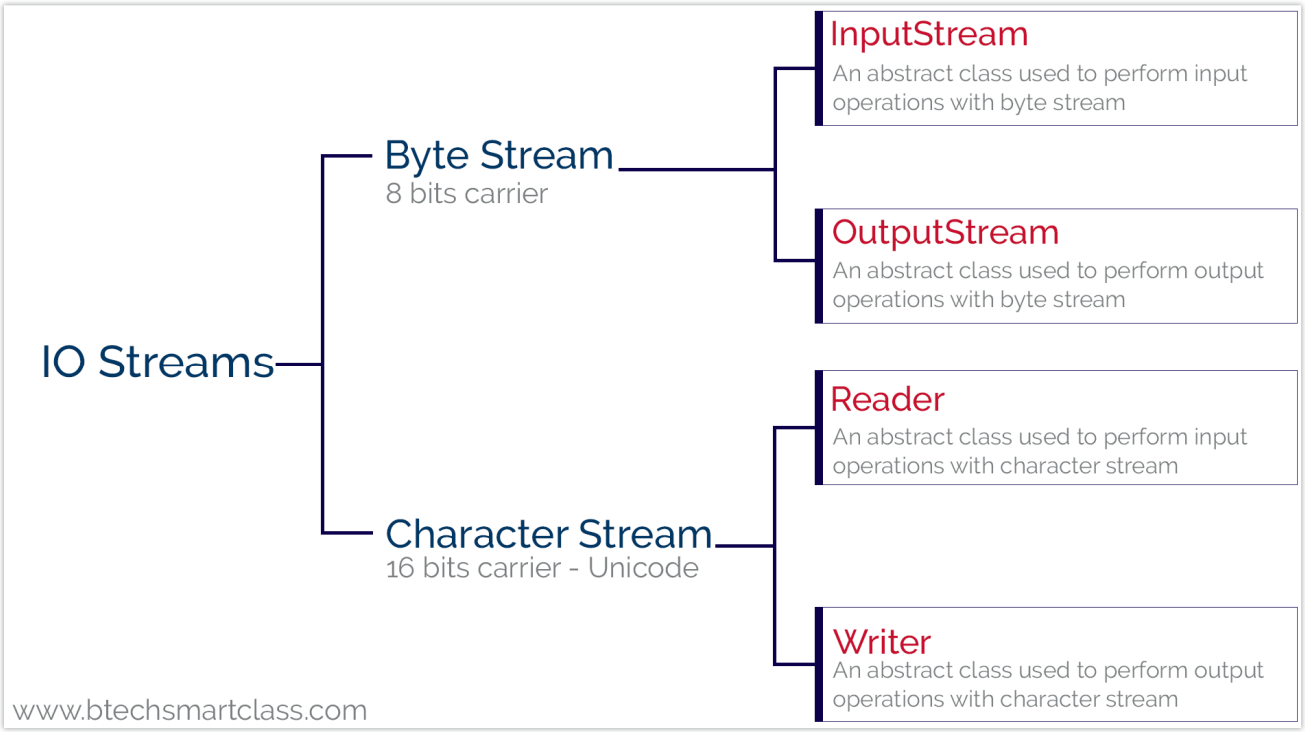
The write() method take integer as argument, and writes its ASCII equalent character on to the console, it also acept escape sequences.

Let's look at the following code to illustrate write() method.

Example

```
public class WritingDemo {

	public static void main(String[] args) {

		int[] list = new int[26];

		for(int i = 0; i < 26; i++) {
```

```
                        list[i] = i + 65;
                }

                for(int i:list) {
                        System.out.write(i);
                        System.out.write('\n');
                }
        }
}
```



Both character and byte streams essentially provides a convenient and efficient way to handle data streams in Java.

In the next tutorial, we will explore different types of streams in java.

File class in Java

The File is a built-in class in Java. In java, the File class has been defined in the java.io package. The File class represents a reference to a file or directory. The File class has various methods to perform operations like creating a file or directory, reading from a file, updating file content, and deleting a file or directory.

The File class in java has the following constructors.

S.No.	Constructor with Description
1	**File(String pathname)** It creates a new File instance by converting the givenpathname string into an abstract pathname. If the given string isthe empty string, then the result is the empty abstract pathname.
2	**File(String parent, String child)** It Creates a new File instance from a parent abstractpathname and a child pathname string. If parent is null then the new File instance is created as if by invoking thesingle-argument File constructor on the given child pathname string.
3	**File(File parent, String child)** It creates a new File instance from a parent abstractpathname and a child pathname string. If parent is null then the new File instance is created as if by invoking thesingle-argument File constructor on the given child pathname string.
4	**File(URI uri)** It creates a new File instance by converting the given file: URI into an abstract pathname.

The File class in java has the following methods.

S.No.	Methods with Description
1	**String getName()** It returns the name of the file or directory that referenced by the current File object.
2	**String getParent()** It returns the pathname of the pathname's parent, or null if the pathname does not name a parent directory.
3	**String getPath()**

S.No.	Methods with Description
	It returns the path of curent File.
4	**File getParentFile()** It returns the path of the current file's parent; or null if it does not exist.
5	**String getAbsolutePath()** It returns the current file or directory path from the root.
6	**boolean isAbsolute()** It returns true if the current file is absolute, false otherwise.
7	**boolean isDirectory()** It returns true, if the current file is a directory; otherwise returns false.
8	**boolean isFile()** It returns true, if the current file is a file; otherwise returns false.
9	**boolean exists()** It returns true if the current file or directory exist; otherwise returns false.
10	**boolean canRead()** It returns true if and only if the file specified exists and can be read by the application; false otherwise.
11	**boolean canWrite()** It returns true if and only if the file specified exists and the application is allowed to write to the file; false otherwise.
12	**long length()** It returns the length of the current file.
13	**long lastModified()** It returns the time that specifies the file was last modified.
14	**boolean createNewFile()** It returns true if the named file does not exist and was successfully created; false if the named file already exists.
15	**boolean delete()** It deletes the file or directory. And returns true if and only if the file or directory is successfully deleted; false otherwise.

S.No.	Methods with Description
16	**void deleteOnExit()** It sends a requests that the file or directory needs be deleted when the virtual machine terminates.
17	**boolean mkdir()** It returns true if and only if the directory was created; false otherwise.
18	**boolean mkdirs()** It returns true if and only if the directory was created, along with all necessary parent directories; false otherwise.
19	**boolean renameTo(File dest)** It renames the current file. And returns true if and only if the renaming succeeded; false otherwise.
20	**boolean setLastModified(long time)** It sets the last-modified time of the file or directory. And returns true if and only if the operation succeeded; false otherwise.
21	**boolean setReadOnly()** It sets the file permission to only read operations; Returns true if and only if the operation succeeded; false otherwise.
22	**String[] list()** It returns an array of strings containing names of all the files and directories in the current directory.
23	**String[] list(FilenameFilter filter)** It returns an array of strings containing names of all the files and directories in the current directory that satisfy the specified filter.
24	**File[] listFiles()** It returns an array of file references containing names of all the files and directories in the current directory.
25	**File[] listFiles(FileFilter filter)** It returns an array of file references containing names of all the files and directories in the current directory that satisfy the specified filter.
26	**boolean equals(Object obj)** It returns true if and only if the argument is not null and is an abstract pathname that denotes the same file or directory as this abstract pathname.

S.No.	Methods with Description
	int compareTo(File pathname)
27	It Compares two abstract pathnames lexicographically. It returns zero if the argument is equal to this abstract pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument.
	int compareTo(File pathname)
28	Compares this abstract pathname to another object. Returns zero if the argument is equal to this abstract pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument.

Let's look at the following code to illustrate file operations.

Example

```
import java.io.*;
public class FileClassTest {

        public static void main(String args[]) {
                File f = new File("C:\\Raja\\datFile.txt");

                System.out.println("Executable File : " + f.canExecute());
                System.out.println("Name of the file : " + f.getName());
                System.out.println("Path of the file : " + f.getAbsolutePath());
                System.out.println("Parent name :  " + f.getParent());
                System.out.println("Write mode :  " + f.canWrite());
                System.out.println("Read mode :  " + f.canRead());
                System.out.println("Existance :  " + f.exists());
                System.out.println("Last Modified :  " + f.lastModified());
                System.out.println("Length :  " + f.length());
                //f.createNewFile()
                //f.delete();
                //f.setReadOnly()
        }

}
```

Let's look at the following java code to list all the files in a directory including the files present in all its subdirectories.

Example

```
import java.util.Scanner;
import java.io.*;

public class ListingFiles {
```

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

		String path = null;
		Scanner read = new Scanner(System.in);
		System.out.print("Enter the root directory name: ");
		path = read.next() + ":\\";
		File f_ref = new File(path);
		if (!f_ref.exists()) {
			printLine();
			System.out.println("Root directory does not exists!");
			printLine();
		} else {
			String ch = "y";
			while (ch.equalsIgnoreCase("y")) {
				printFiles(path);
				System.out.print("Do you want to open any sub-
directory (Y/N):  ");
				ch = read.next().toLowerCase();
				if (ch.equalsIgnoreCase("y")) {
					System.out.print("Enter the sub-directory
name: ");
					path = path + "\\\\" + read.next();
					File f_ref_2 = new File(path);
					if (!f_ref_2.exists()) {
						printLine();
						System.out.println("The sub-directory
does not exists!");
						printLine();
						int lastIndex =
path.lastIndexOf("\\");
						path = path.substring(0, lastIndex);
					}
				}
			}
		}
		System.out.println("***** Program Closed *****");
	}

	public static void printFiles(String path) {
		System.out.println("Current Location: " + path);
		File f_ref = new File(path);
		File[] filesList = f_ref.listFiles();
		for (File file : filesList) {
			if (file.isFile())
				System.out.println("- " + file.getName());
			else
				System.out.println("> " + file.getName());
		}
	}

	public static void printLine() {
		System.out.println("---------------------------------------");
	}
}
```

File Reading & Writing in Java

In java, there multiple ways to read data from a file and to write data to a file. The most commonly used ways are as follows.

- **Using Byte Stream (FileInputStream and FileOutputStream)**
- **Using Character Stream (FileReader and FileWriter)**

Let's look each of these ways.

File Handling using Byte Stream

In java, we can use a byte stream to handle files. The byte stream has the following built-in classes to perform various operations on a file.

- FileInputStream - It is a built-in class in java that allows reading data from a file. This class has implemented based on the byte stream. The FileInputStream class provides a method read() to read data from a file byte by byte.
- FileOutputStream - It is a built-in class in java that allows writing data to a file. This class has implemented based on the byte stream. The FileOutputStream class provides a method write() to write data to a file byte by byte.

Let's look at the following example program that reads data from a file and writes the same to another file using FileInoutStream and FileOutputStream classes.

Example

```
import java.io.*;
public class FileReadingTest {

   public static void main(String args[]) throws IOException {
      FileInputStream in = null;
      FileOutputStream out = null;

      try {
         in = new FileInputStream("C:\\Raja\\Input-File.txt");
         out = new FileOutputStream("C:\\Raja\\Output-File.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
         System.out.println("Reading and Writing has been success!!!");
      }
      catch(Exception e){
          System.out.println(e);
      }finally {
         if (in != null) {
            in.close();
```

```
         }
         if (out != null) {
            out.close();
         }
      }   }}
```

File Handling using Character Stream

In java, we can use a character stream to handle files. The character stream has the following built-in classes to perform various operations on a file.

- FileReader - It is a built-in class in java that allows reading data from a file. This class has implemented based on the character stream. The FileReader class provides a method read() to read data from a file character by character.
- FileWriter - It is a built-in class in java that allows writing data to a file. This class has implemented based on the character stream. The FileWriter class provides a method write() to write data to a file character by character.

Let's look at the following example program that reads data from a file and writes the same to another file using FIleReader and FileWriter classes.

Example

```
import java.io.*;
public class FileIO {

   public static void main(String args[]) throws IOException {
          FileReader in = null;
      FileWriter out = null;

      try {
         in = new FileReader("C:\\Raja\\Input-File.txt");
         out = new FileWriter("C:\\Raja\\Output-File.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
         System.out.println("Reading and Writing in a file is done!!!");
      }
      catch(Exception e) {
          System.out.println(e);
      }
      finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

When we run the above program, it produce the following output.

RandomAccessFile in Java

In java, the java.io package has a built-in class RandomAccessFile that enables a file to be accessed randomly. The RandomAccessFile class has several methods used to move the cursor position in a file.

A random access file behaves like a large array of bytes stored in a file.

RandomAccessFile Constructors

The RandomAccessFile class in java has the following constructors.

S.No.	Constructor with Description
1	**RandomAccessFile(File fileName, String mode)** It creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.
2	**RandomAccessFile(String fileName, String mode)** It creates a random access file stream to read from, and optionally to write to, a file with the specified fileName.

Access Modes

Using the RandomAccessFile, a file may created in th following modes.

- r - Creates the file with read mode; Calling write methods will result in an IOException.
- rw - Creates the file with read and write mode.
- rwd - Creates the file with read and write mode - synchronously. All updates to file content is written to the disk synchronously.
- rws - Creates the file with read and write mode - synchronously. All updates to file content or meta data is written to the disk synchronously.

RandomAccessFile methods

The RandomAccessFile class in java has the following methods.

S.No.	Methods with Description
1	**int read()** It reads byte of data from a file. The byte is returned as an integer in the range 0-255.
2	**int read(byte[] b)** It reads byte of data from file upto b.length, -1 if end of file is reached.
3	**int read(byte[] b, int offset, int len)** It reads bytes initialising from offset position upto b.length from the buffer.
4	**boolean readBoolean()** It reads a boolean value from from the file.
5	**byte readByte()** It reads signed eight-bit value from file.
6	**char readChar()** It reads a character value from file.
7	**double readDouble()** It reads a double value from file.
8	**float readFloat()** It reads a float value from file.
9	**long readLong()** It reads a long value from file.
10	**int readInt()** It reads a integer value from file.
11	**void readFully(byte[] b)** It reads bytes initialising from offset position upto b.length from the buffer.
12	**void readFully(byte[] b, int offset, int len)** It reads bytes initialising from offset position upto b.length from the buffer.
13	**String readUTF()**

S.No.	Methods with Description
	t reads in a string from the file.
	void seek(long pos)
14	It sets the file-pointer(cursor) measured from the beginning of the file, at which the next read or write occurs.
	long length()
15	It returns the length of the file.
	void write(int b)
16	It writes the specified byte to the file from the current cursor position.
	void writeFloat(float v)
17	It converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the file as a four-byte quantity, high byte first.
	void writeDouble(double v)
18	It converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the file as an eight-byte quantity, high byte first.

Let's look at the following example program.

Example

```
import java.io.*;

public class RandomAccessFileDemo
{
    public static void main(String[] args)
    {
        try
        {
            double d = 1.5;
            float f = 14.56f;

            // Creating a new RandomAccessFile - "F2"
            RandomAccessFile f ref = new RandomAccessFile("C:\\Raja\\Input-
File.txt", "rw");

            // Writing to file
            f ref.writeUTF("Hello, Good Morning!");

            // File Pointer at index position - 0
            f_ref.seek(0);

            // read() method :
```

```
            System.out.println("Use of read() method : " + f_ref.read());

            f ref.seek(0);

            byte[] b = {1, 2, 3};

            // Use of .read(byte[] b) method :
            System.out.println("Use of .read(byte[] b) : " + f ref.read(b));

            // readBoolean() method :
            System.out.println("Use of readBoolean() : " + f_ref.readBoolean());

            // readByte() method :
            System.out.println("Use of readByte() : " + f ref.readByte());

            f_ref.writeChar('c');
            f_ref.seek(0);

            // readChar() :
            System.out.println("Use of readChar() : " + f_ref.readChar());

            f ref.seek(0);
            f_ref.writeDouble(d);
            f_ref.seek(0);

            // read double
            System.out.println("Use of readDouble() : " + f ref.readDouble());

            f ref.seek(0);
            f_ref.writeFloat(f);
            f_ref.seek(0);

            // readFloat() :
            System.out.println("Use of readFloat() : " + f_ref.readFloat());

            f_ref.seek(0);
            // Create array upto geek.length
            byte[] arr = new byte[(int) f_ref.length()];
            // readFully() :
            f_ref.readFully(arr);

            String str1 = new String(arr);
            System.out.println("Use of readFully() : " + str1);

            f_ref.seek(0);

            // readFully(byte[] b, int off, int len) :
            f_ref.readFully(arr, 0, 8);

            String str2 = new String(arr);
            System.out.println("Use of readFully(byte[] b, int off, int len) : " +
str2);
        }
        catch (IOException ex)
        {
            System.out.println("Something went Wrong");
            ex.printStackTrace();
```

```
        }
    }
}
```

Java Scanner class

There are various ways to read input from the keyboard, the java.util.Scanner class is one of them. The **Java Scanner** class breaks the input into tokens using a delimiter that is whitespace bydefault. It provides many methods to read and parse various primitive values.

Java Scanner class is widely used to parse text for string and primitive types using regular expression.

Java Scanner class extends Object class and implements Iterator and Closeable interfaces.

Commonly used methods of Scanner class

There is a list of commonly used Scanner class methods:

Method	Description
public String next()	it returns the next token from the scanner.
public String nextLine()	it moves the scanner position to the next line and returns the value as a string.
public byte nextByte()	it scans the next token as a byte.
public short nextShort()	it scans the next token as a short value.
public int nextInt()	it scans the next token as an int value.
public long nextLong()	it scans the next token as a long value.
public float nextFloat()	it scans the next token as a float value.

Java Scanner Example to get input from console

Let's see the simple example of the Java Scanner class which reads the int, string and double value as an input:

```
1. import java.util.Scanner;
2. class ScannerTest{
3. public static void main(String args[]){
4. Scanner sc=new Scanner(System.in);

5. System.out.println("Enter your rollno");
```

```
6.  int rollno=sc.nextInt();
7.  System.out.println("Enter your name");
8.  String name=sc.next();
9.  System.out.println("Enter your fee");
10. double fee=sc.nextDouble();
11. System.out.println("Rollno:"+rollno+" name:"+name+" fee:"+fee);
12. sc.close();
13. } }
```

Chapter 8: Event Programming /Java AWT

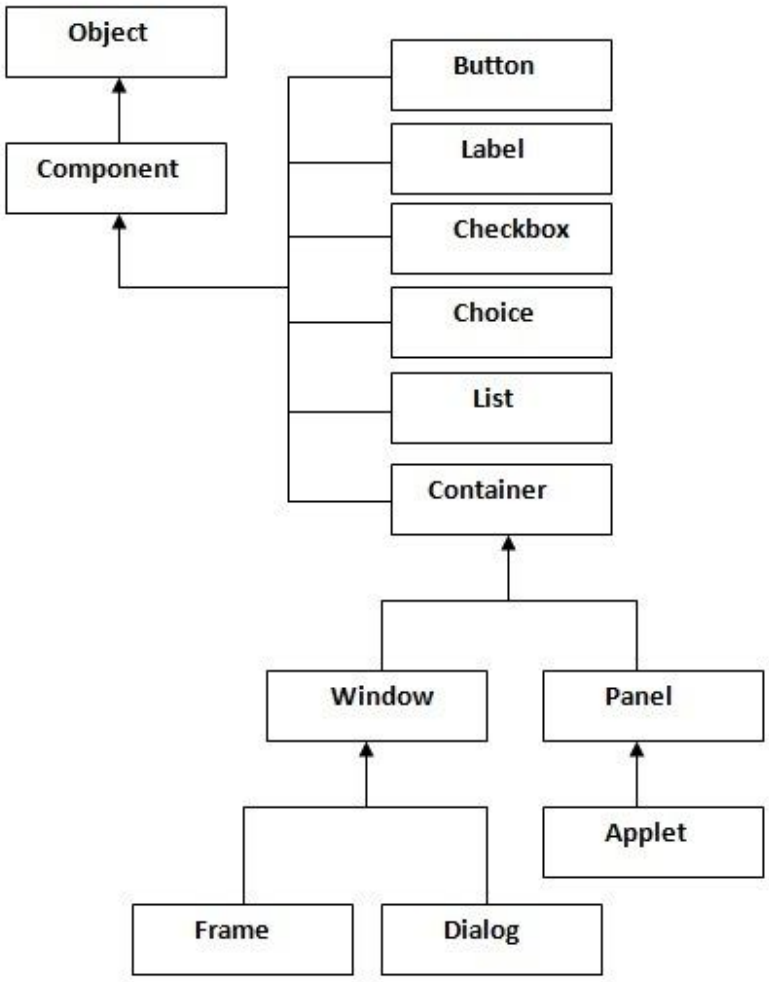
Java AWT (Abstract Windowing Toolkit) is *an API to develop GUI or window-based application in java*.

Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components uses the resources of system.

The java.awt package provides classes for AWT api such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

Java AWT Hierarchy

The hierarchy of Java AWT classes are given below.



Container

The Container is a component in AWT that can contain another components like buttons, textfields, labels etc. The classes that extends Container class are known as container such as Frame, Dialog and Panel.

Window

The window is the container that have no borders and menu bars. You must use frame, dialog or another window for creating a window.

Panel

The Panel is the container that doesn't contain title bar and menu bars. It can have other components like button, textfield etc.

Frame

The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.

Useful Methods of Component class

Method	Description
public void add(Component c)	inserts a component on this component.
public void setSize(int width,int height)	sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.

Java AWT Example

To create simple awt example, you need a frame. There are two ways to create a frame in AWT.

- By extending Frame class (inheritance)
- By creating the object of Frame class (association)

Simple example of AWT by inheritance

```
1. import java.awt.*;
```

```
2.  class First extends Frame{
3.  First(){
4.  Button b=new Button("click me");
5.  b.setBounds(30,100,80,30);// setting button position

6.  add(b);//adding button into frame
7.  setSize(300,300);//frame size 300 width and 300 height
8.  setLayout(null);//no layout manager
9.  setVisible(true);//now frame will be visible, by default not visible
10. }
11. public static void main(String args[]){
12. First f=new First();
13. }}
```

download this example

The setBounds(int xaxis, int yaxis, int width, int height) method is used in the above example that sets the position of the awt button.

Simple example of AWT by association

```
1.  import java.awt.*;
2.  class First2{
3.  First2(){
4.  Frame f=new Frame();

5.  Button b=new Button("click me");
6.  b.setBounds(30,50,80,30);

7.  f.add(b);
8.  f.setSize(300,300);
9.  f.setLayout(null);
10. f.setVisible(true);
```

```
11.     }
12.     public static void main(String args[]){
13.     First2 f=new First2();
14.     }}
```

Java Swing Tutorial

Java Swing tutorial is a part of Java Foundation Classes (JFC) that is *used to create window-based applications*. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

Unlike AWT, Java Swing provides platform-independent and lightweight components.

The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

Difference between AWT and Swing

There are many differences between java awt and swing that are given below.

No.	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are**platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are**lightweight**.
3)	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

What is JFC

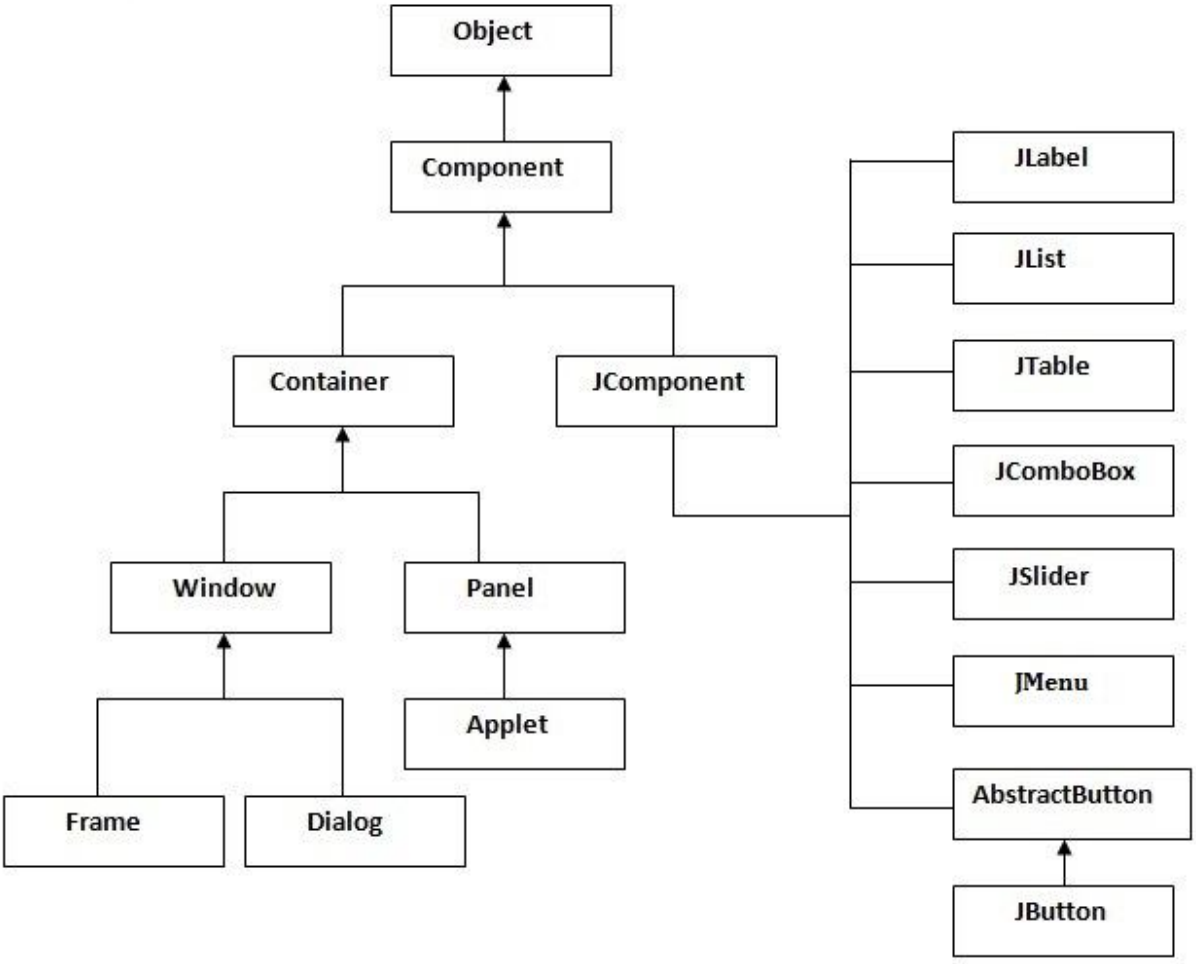
The Java Foundation Classes (JFC) are a set of GUI components which simplify the development of desktop applications.

Do You Know

- How to create runnable jar file in java?
- How to display image on a button in swing?
- How to change the component color by choosing a color from ColorChooser ?
- How to display the digital watch in swing tutorial ?
- How to create a notepad in swing?
- How to create puzzle game and pic puzzle game in swing ?
- How to create tic tac toe game in swing ?

Hierarchy of Java Swing classes

The hierarchy of java swing API is given below.



Commonly used Methods of Component class

The methods of Component class are widely used in java swing that are given below.

Method	Description
public void add(Component c)	add a component on another component.
public void setSize(int width,int height)	sets size of the component.
public void setLayout(LayoutManager m)	sets the layout manager for the component.
public void setVisible(boolean b)	sets the visibility of the component. It is by default false.

Java Swing Examples

There are two ways to create a frame:

- By creating the object of Frame class (association)
- By extending Frame class (inheritance)

We can write the code of swing inside the main(), constructor or any other method.

Simple Java Swing Example

Let's see a simple swing example where we are creating one button and adding it on the JFrame object inside the main() method.

File: FirstSwingExample.java

```
1.  import javax.swing.*;
2.  public class FirstSwingExample {
3.  public static void main(String[] args) {
4.  JFrame f=new JFrame();//creating instance of JFrame

5.  JButton b=new JButton("click");//creating instance of JButton
6.  b.setBounds(130,100,100, 40);//x axis, y axis, width, height

7.  f.add(b);//adding button in JFrame

8.  f.setSize(400,500);//400 width and 500 height
9.  f.setLayout(null);//using no layout managers
10. f.setVisible(true);//making the frame visible
11. }
12. }
```

Example of Swing by Association inside constructor

We can also write all the codes of creating JFrame, JButton and method call inside the java constructor.

File: Simple.java

```
1.  import javax.swing.*;
2.  public class Simple {
3.  JFrame f;
4.  Simple(){
5.  f=new JFrame();//creating instance of JFrame

6.  JButton b=new JButton("click");//creating instance of JButton
7.  b.setBounds(130,100,100, 40);

8.  f.add(b);//adding button in JFrame

9.  f.setSize(400,500);//400 width and 500 height
10. f.setLayout(null);//using no layout managers
11. f.setVisible(true);//making the frame visible
12. }

13. public static void main(String[] args) {
14. new Simple();
15. }
16. }
```

The setBounds(int xaxis, int yaxis, int width, int height)is used in the above example that sets the position of the button.

Simple example of Swing by inheritance

We can also inherit the JFrame class, so there is no need to create the instance of JFrame class explicitly.

File: Simple2.java

```
1.  import javax.swing.*;
2.  public class Simple2 extends JFrame{//inheriting JFrame
3.  JFrame f;
4.  Simple2(){
5.  JButton b=new JButton("click");//create button
6.  b.setBounds(130,100,100, 40);

7.  add(b);//adding button on frame
8.  setSize(400,500);
9.  setLayout(null);
10. setVisible(true);
11. }
12. public static void main(String[] args) {
13. new Simple2();
14. }}
```

What we will learn in Swing Tutorial

- JButton class
- JRadioButton class
- JTextArea class
- JComboBox class
- JTable class
- JColorChooser class
- JProgressBar class
- JSlider class
- Digital Watch
- Graphics in swing
- Displaying image
- Edit menu code for Notepad
- OpenDialog Box
- Notepad
- Puzzle Game
- Pic Puzzle Game
- Tic Tac Toe Game
- BorderLayout
- GridLayout
- FlowLayout
- CardLayout

JButton class:

The JButton class is used to create a button that have plateform-independent implementation.

Commonly used Constructors:

- **JButton():** creates a button with no text and icon.
- **JButton(String s):** creates a button with the specified text.
- **JButton(Icon i):** creates a button with the specified icon object.

Commonly used Methods of AbstractButton class:

1) public void setText(String s): is used to set specified text on button.

2) public String getText(): is used to return the text of the button.

3) public void setEnabled(boolean b): is used to enable or disable the button.

4) public void setIcon(Icon b): is used to set the specified Icon on the button.

5) public Icon getIcon(): is used to get the Icon of the button.

6) public void setMnemonic(int a): is used to set the mnemonic on the button.

7) public void addActionListener(ActionListener a): is used to add the action listener to this object.

Note: The JButton class extends AbstractButton class.

Example of displaying image on the button:

```
1.  import java.awt.event.*;
2.  import javax.swing.*;

3.  public class ImageButton{
4.  ImageButton(){
5.  JFrame f=new JFrame();
6.  JButton b=new JButton(new ImageIcon("b.jpg"));
7.  b.setBounds(130,100,100, 40);
8.  f.add(b);
9.  f.setSize(300,400);
10. f.setLayout(null);
11. f.setVisible(true);
12. f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

13. }
14. public static void main(String[] args) {
15. new ImageButton();
16. }
17. }
```

JRadioButton class

The JRadioButton class is used to create a radio button. It is used to choose one option from multiple options. It is widely used in exam systems or quiz.

It should be added in ButtonGroup to select one radio button only.

Commonly used Constructors of JRadioButton class:

- **JRadioButton():** creates an unselected radio button with no text.
- **JRadioButton(String s):** creates an unselected radio button with specified text.
- **JRadioButton(String s, boolean selected):** creates a radio button with the specified text and selected status.

Commonly used Methods of AbstractButton class:

1) public void setText(String s): is used to set specified text on button.

2) public String getText(): is used to return the text of the button.

3) public void setEnabled(boolean b): is used to enable or disable the button.

4) public void setIcon(Icon b): is used to set the specified Icon on the button.

5) public Icon getIcon(): is used to get the Icon of the button.

6) public void setMnemonic(int a): is used to set the mnemonic on the button.

7) public void addActionListener(ActionListener a): is used to add the action listener to this object.

Note: The JRadioButton class extends the JToggleButton class that extends AbstractButton class.

Example of JRadioButton class:

```
1.  import javax.swing.*;
2.  public class Radio {
3.  JFrame f;
4.  Radio(){
5.  f=new JFrame();
6.  JRadioButton r1=new JRadioButton("A) Male");
7.  JRadioButton r2=new JRadioButton("B) FeMale");
8.  r1.setBounds(50,100,70,30);
9.  r2.setBounds(50,150,70,30);

10. ButtonGroup bg=new ButtonGroup();
11. bg.add(r1);bg.add(r2);

12. f.add(r1);f.add(r2);
13. f.setSize(300,300);
14. f.setLayout(null);
15. f.setVisible(true);
16. }
17. public static void main(String[] args) {
18. new Radio();
19. } }
```

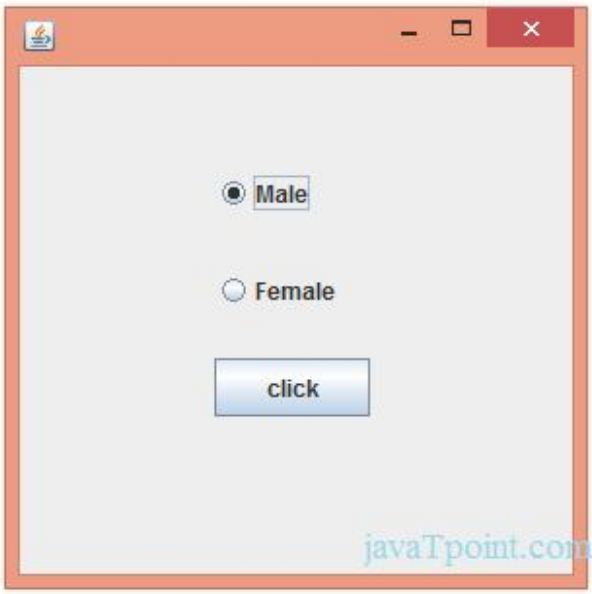
ButtonGroup class:

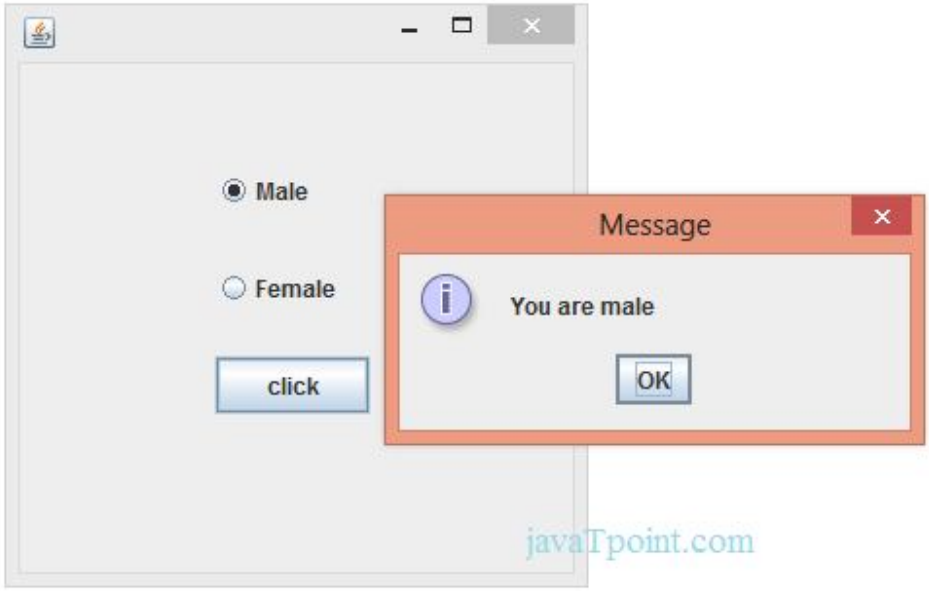
The ButtonGroup class can be used to group multiple buttons so that at a time only one button can be selected.

JRadioButton example with event handling

```
1.  import javax.swing.*;
2.  import java.awt.event.*;
3.  class RadioExample extends JFrame implements ActionListener{
4.  JRadioButton rb1,rb2;
5.  JButton b;
6.  RadioExample(){
7.
8.  rb1=new JRadioButton("Male");
9.  rb1.setBounds(100,50,100,30);
10.
11. rb2=new JRadioButton("Female");
12. rb2.setBounds(100,100,100,30);
13.
14. ButtonGroup bg=new ButtonGroup();
15. bg.add(rb1);bg.add(rb2);
16.
17. b=new JButton("click");
18. b.setBounds(100,150,80,30);
19. b.addActionListener(this);
20.
21. add(rb1);add(rb2);add(b);
22.
23. setSize(300,300);
24. setLayout(null);
25. setVisible(true);
26. }
27. public void actionPerformed(ActionEvent e){
28. if(rb1.isSelected()){
29. JOptionPane.showMessageDialog(this,"You are male");
30. }
31. if(rb2.isSelected()){
32. JOptionPane.showMessageDialog(this,"You are female");
33. }
34. }
35. public static void main(String args[]){
36. new RadioExample();
37. }}
```

38.





JTextArea-class

JTextArea class (Swing Tutorial):

The JTextArea class is used to create a text area. It is a multiline area that displays the plain text only.

Commonly used Constructors:

- **JTextArea():** creates a text area that displays no text initially.
- **JTextArea(String s):** creates a text area that displays specified text initially.
- **JTextArea(int row, int column):** creates a text area with the specified number of rows and columns that displays no text initially..

- **JTextArea(String s, int row, int column):** creates a text area with the specified number of rows and columns that displays specified text.

Commonly used methods of JTextArea class:

1) public void setRows(int rows): is used to set specified number of rows.

2) public void setColumns(int cols):: is used to set specified number of columns.

3) public void setFont(Font f): is used to set the specified font.

4) public void insert(String s, int position): is used to insert the specified text on the specified position.

5) public void append(String s): is used to append the given text to the end of the document.

Example of JTextField class:

```
1.  import java.awt.Color;
2.  import javax.swing.*;

3.  public class TArea {
4.  JTextArea area;
5.  JFrame f;
6.  TArea(){
7.  f=new JFrame();

8.  area=new JTextArea(300,300);
9.  area.setBounds(10,30,300,300);

10. area.setBackground(Color.black);
11. area.setForeground(Color.white);

12. f.add(area);

13. f.setSize(400,400);
14. f.setLayout(null);
15. f.setVisible(true);
16. }
17. public static void main(String[] args) {
18. new TArea();
19. }
20. }
```

JComboBox class:

The JComboBox class is used to create the combobox (drop-down list). At a time only one item can be selected from the item list.

Commonly used Constructors of JComboBox class:

JComboBox()

JComboBox(Object[] items)

JComboBox(Vector<?> items)

Commonly used methods of JComboBox class:

1) public void addItem(Object anObject): is used to add an item to the item list.

2) public void removeItem(Object anObject): is used to delete an item to the item list.

3) public void removeAllItems(): is used to remove all the items from the list.

4) public void setEditable(boolean b): is used to determine whether the JComboBox is editable.

5) public void addActionListener(ActionListener a): is used to add the ActionListener.

6) public void addItemListener(ItemListener i): is used to add the ItemListener.

Example of JComboBox class:

```
1.  import javax.swing.*;
2.  public class Combo {
3.  JFrame f;
4.  Combo(){
5.  f=new JFrame("Combo ex");

6.  String country[]={"India","Aus","U.S.A","England","Newzeland"};

7.  JComboBox cb=new JComboBox(country);
8.  cb.setBounds(50, 50,90,20);
9.  f.add(cb);

10. f.setLayout(null);
11. f.setSize(400,500);
12. f.setVisible(true);

13. }
14. public static void main(String[] args) {
15. new Combo();

16. }
17. }
```

JTable class (Swing Tutorial):

The JTable class is used to display the data on two dimensional tables of cells.

Commonly used Constructors of JTable class:

- **JTable():** creates a table with empty cells.
- **JTable(Object[][] rows, Object[] columns):** creates a table with the specified data.

Example of JTable class:

```
1.  import javax.swing.*;
2.  public class MyTable {
3.  JFrame f;
4.  MyTable(){
5.  f=new JFrame();

6.  String data[][]={ {"101","Amit","670000"},
    a.  {"102","Jai","780000"},
        i.  {"101","Sachin","700000"}};
7.  String column[]={"ID","NAME","SALARY"};

8.  JTable jt=new JTable(data,column);
9.  jt.setBounds(30,40,200,300);

10. JScrollPane sp=new JScrollPane(jt);
11. f.add(sp);

12. f.setSize(300,400);
13. //  f.setLayout(null);
14. f.setVisible(true);
15. }
16. public static void main(String[] args) {
17. new MyTable();
18. }
19. }
```

JColorChooser class:

The JColorChooser class is used to create a color chooser dialog box so that user can select any color.

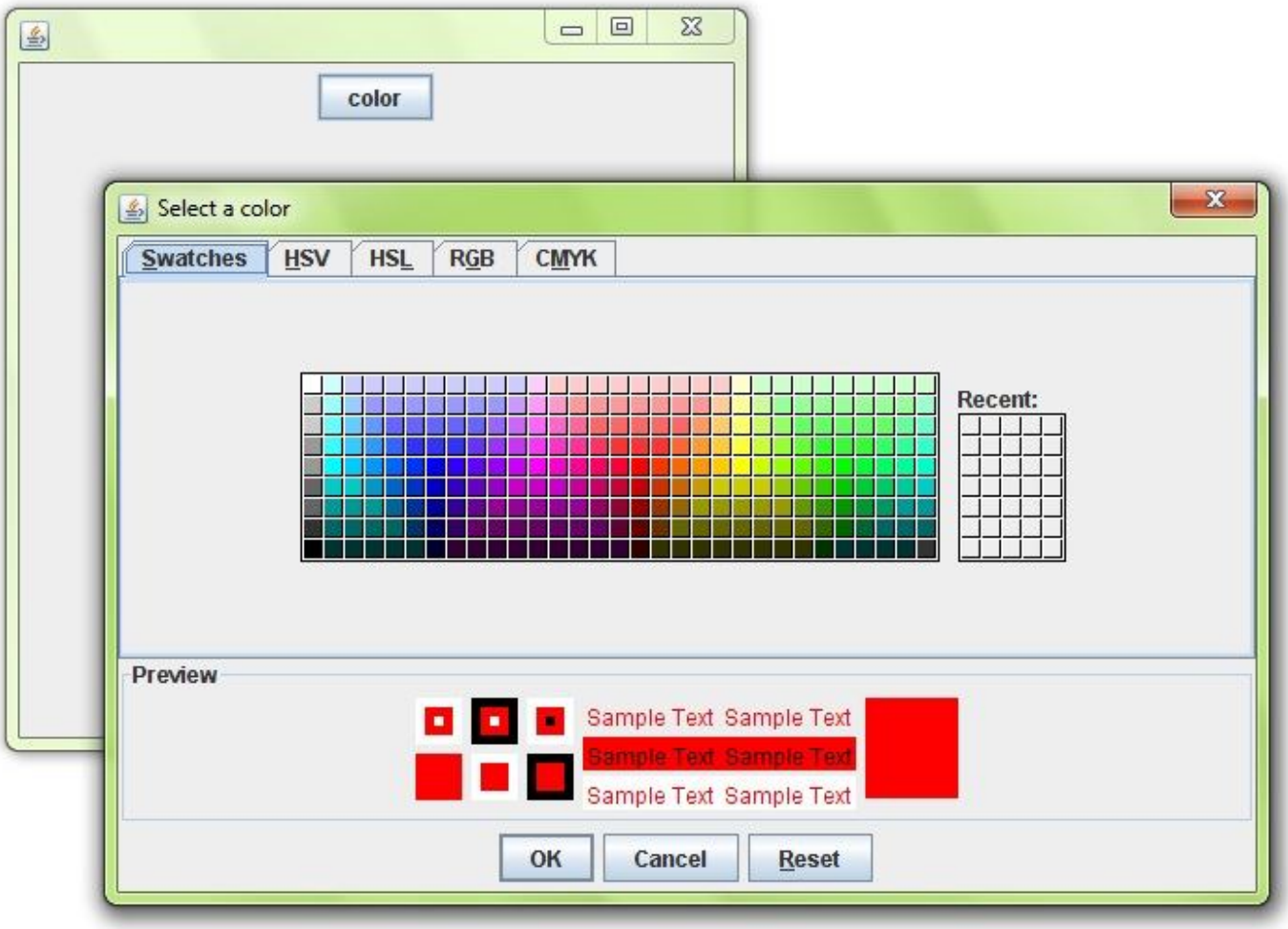
Commonly used Constructors of JColorChooser class:

- **JColorChooser():** is used to create a color chooser pane with white color initially.
- **JColorChooser(Color initialColor):** is used to create a color chooser pane with the specified color initially.

Commonly used methods of JColorChooser class:

public static Color showDialog(Component c, String title, Color initialColor): is used to show the color-chooser dialog box.

Example of JColorChooser class:




```
1.  import java.awt.event.*;
2.  import java.awt.*;
3.  import javax.swing.*;

4.  public class JColorChooserExample extends JFrame implements ActionLis
    tener{
5.  JButton b;
6.  Container c;

7.  JColorChooserExample(){
8.  c=getContentPane();
9.  c.setLayout(new FlowLayout());

10. b=new JButton("color");
11. b.addActionListener(this);
```

```
12.     c.add(b);
13.     }

14.     public void actionPerformed(ActionEvent e) {
15.     Color initialcolor=Color.RED;
16.     Color color=JColorChooser.showDialog(this,"Select a color",initialcolor);
17.     c.setBackground(color);
18.     }

19.     public static void main(String[] args) {
20.     JColorChooserExample ch=new JColorChooserExample();
21.     ch.setSize(400,400);
22.     ch.setVisible(true);
23.     ch.setDefaultCloseOperation(EXIT_ON_CLOSE);
24.     }
25.     }
```

Chapter 9: JDBC/Oracle DataBase

What is JDBC?

JDBC is Java application programming interface that allows the Java programmers to access database management system from Java code. It was developed by JavaSoft, a subsidiary of Sun Microsystems.

Introduction

Java Database Connectivity in short called as JDBC. It is a java API which enables the java programs to execute SQL statements. It is an application programming interface that defines how a java programmer can access the database in tabular format from Java code using a set of standard interfaces and classes written in the Java programming language.

JDBC has been developed under the Java Community Process that allows multiple implementations to exist and be used by the same application. JDBC provides methods for querying and updating the data in Relational Database Management system such as SQL, Oracle etc.

The Java application programming interface provides a mechanism for dynamically loading the correct Java packages and drivers and registering them with the JDBC Driver Manager that is used as a connection factory for creating JDBC connections which supports creating and executing statements such as SQL INSERT, UPDATE and DELETE. Driver Manager is the backbone of the jdbc architecture.

Generally all Relational Database Management System supports SQL and we all know that Java is platform independent, so JDBC makes it possible to write a single database application that can run on different platforms and interact with different Database Management Systems.

Java Database Connectivity (JDBC) is similar to Open Database Connectivity (ODBC) which is used for accessing and managing database, but the difference is that JDBC is designed specifically for Java programs, whereas ODBC is not depended upon any language.

In short JDBC helps the programmers to write java applications that manage these three programming activities:

1. It helps us to connect to a data source, like a database.
2. It helps us in sending queries and updating statements to the database and
3. Retrieving and processing the results received from the database in terms of answering to your query.

JDBC Architecture .

JDBC is Sun's attempt to create a platform-neutral interface between databases and Java. With JDBC, you can count on a standard set of database access features and (usually) a particular subset of SQL, SQL-92. The JDBC API defines a set of interfaces that encapsulate major database functionality, including running queries, processing results, and determining configuration information. A database vendor or third-party developer writes a JDBC *driver*, which is a set of classes that implements these interfaces for a particular database system. An application can use a number of drivers interchangeably. Figure 2-1 shows how an application uses JDBC to interact with one or more databases without knowing about the underlying driver implementations.

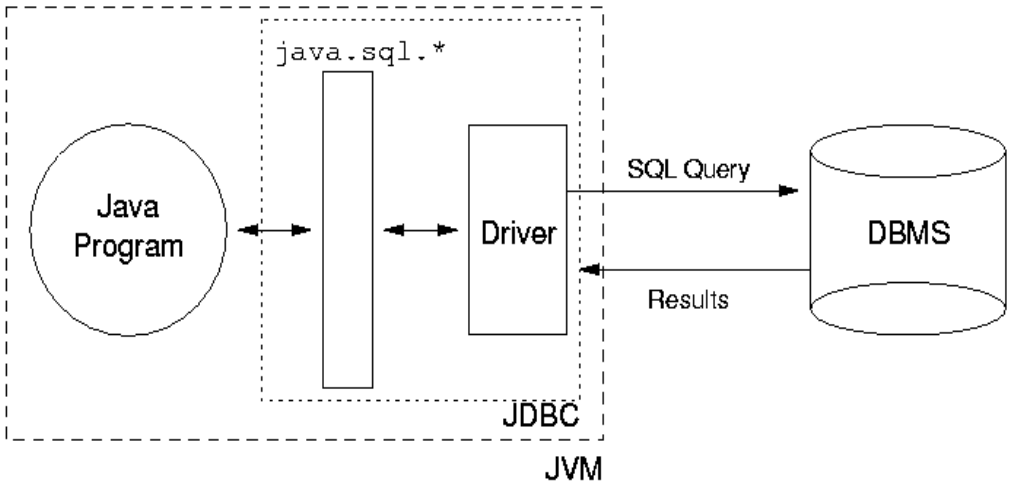


Figure 2-1. JDBC-database interaction

Two-tier Models

The JDBC API supports both two-tier and three-tier models for database access.

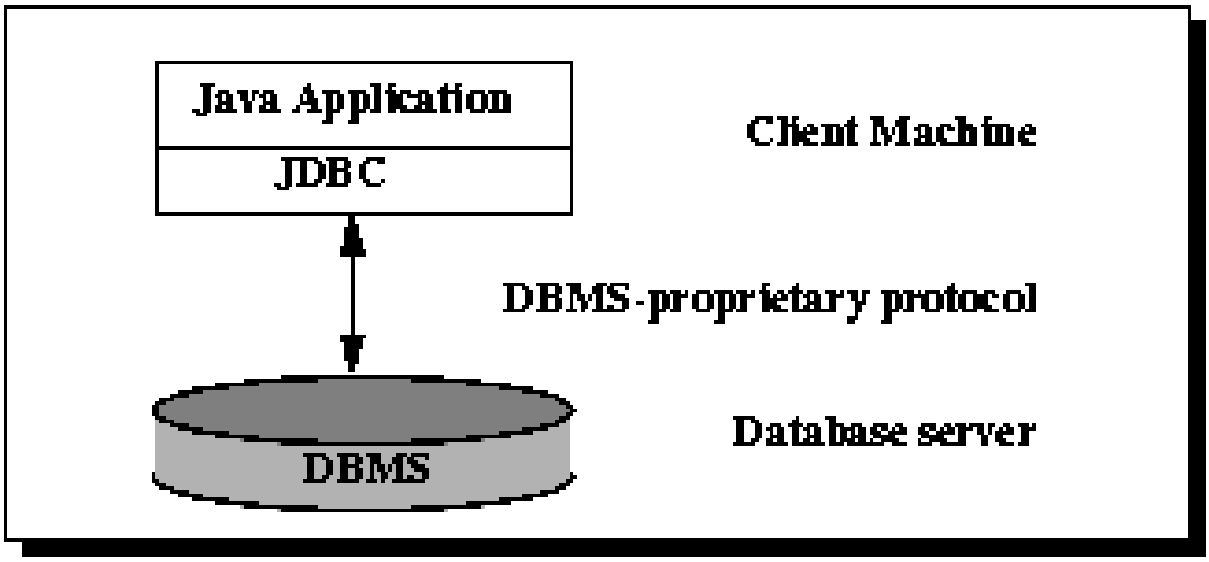


Figure 1.1: illustrates a two-tier architecture for data access.

In the two-tier model, a Java applet or application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

Three-tier Models In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user. MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.

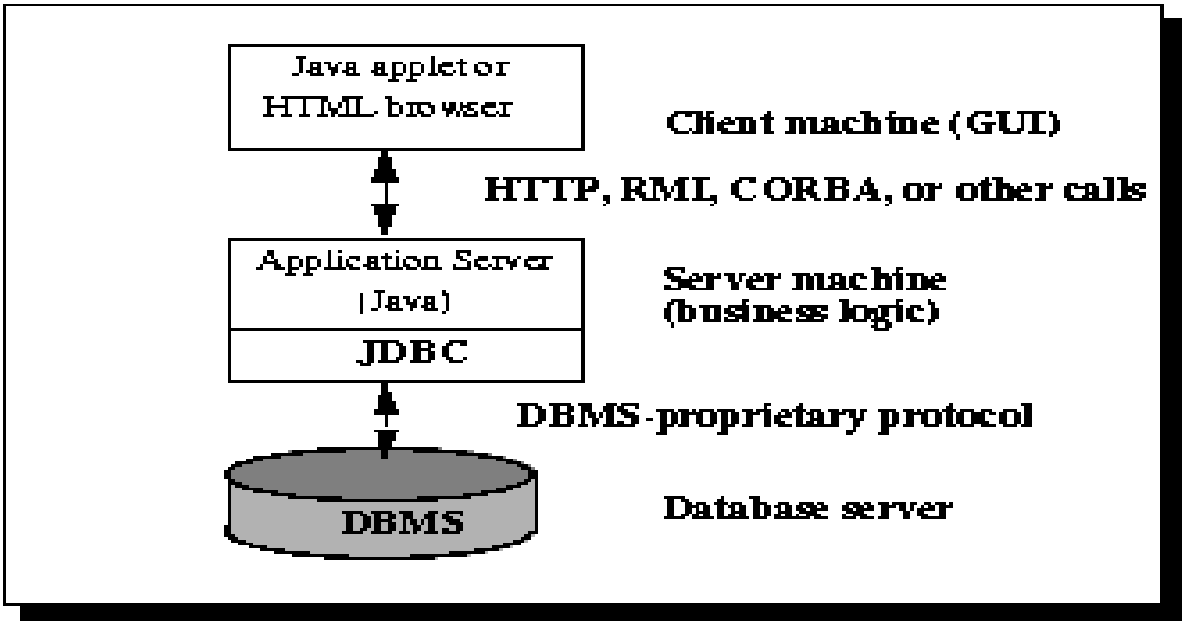


Figure 1.2: illustrates a three-tier architecture for database access.

Until recently, the middle tier has typically been written in languages such as C or C++, which offer fast performance. However, with the introduction of optimizing compilers that translate Java bytecode into efficient machine-specific code and technologies such as Enterprise JavaBeanstm, the Java platform is fast becoming the standard platform for middle-tier development. This is a big plus, making it possible to take advantage of Java's robustness, multithreading, and security features.

JDBC Basics

Before we discuss all of the individual components of JDBC, let's look at a simple example that incorporates most of the major pieces of JDBC functionality. Example 2-1 loads a driver, connects to the database, executes some SQL, and retrieves the results. It also keeps an eye out for any database-related errors.

Example 2-1: A Simple JDBC Example

```
import java.sql.*;

public class JDBCSample {

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

   // This is where we load the driver
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

```
    Connection con = DriverManager.getConnection("jdbc:odbc:sunildb", "", "");

    // Create and execute an SQL Statement
    Statement stmt = con.createStatement( );

    ResultSet rs = stmt.executeQuery("SELECT FIRST_NAME FROM EMPLOYEES");

    // Display the SQL Results
    while(rs.next( )) {
      System.out.println(rs.getString("FIRST_NAME"));
    }

    // Make sure our database resources are released
    rs.close( );
   con.close( );

  }
}
```

Example : starts out by loading a JDBC driver class (in this case, Sun's JDBC-ODBC Bridge). Then it creates a database connection, represented by a Connection object, using that driver. With the database connection, we can create a Statement object to represent an SQL statement. Executing an SQL statement produces a ResultSet that contains the results of a query. The program displays the results and then cleans up the resources it has used. If an error occurs, a SQLException is thrown, so our program traps that exception and displays some of the information it encapsulates.

Clearly, there is a lot going on in this simple program. Every Java application that uses JDBC follows these basic steps, so the following sections discuss each step in much more detail.

JDBC has four Components:

1. The JDBC API.

2. The JDBC Driver Manager.

3. The JDBC Test Suite.

4. The JDBC-ODBC Bridge.

1. The JDBC API:

The JDBC application programming interface provides the facility for accessing the relational database from the Java programming language. The API technology provides the industrial standard for independently connecting Java programming language and a wide range of databases. The user not only execute the SQL statements, retrieve results, and update the data

but can also access it anywhere within a network because of it's "Write Once, Run Anywhere" (WORA) capabilities.

Due to JDBC API technology, user can also access other tabular data sources like spreadsheets or flat files even in the a heterogeneous environment. JDBC application programmming interface is a part of the Java platform that have included Java Standard Edition (Java SE) and the Java Enterprise Edition (Java EE) in itself.

The JDBC API has four main interface:

The latest version of JDBC 4.0 application programming interface is divided into two packages

i-) java.sql
ii-) javax.sql.

Java SE and Java EE platforms are included in both the packages.

2. The JDBC Driver Manager.

The JDBC Driver Manager is a very important class that defines objects which connect Java applications to a JDBC driver. Usually Driver Manager is the backbone of the JDBC architecture. It's very simple and small that is used to provide a means of managing the different types of JDBC database driver running on an application. The main responsibility of JDBC database driver is to load all the drivers found in the system properly as well as to select the most appropriate driver from opening a connection to a database. The Driver Manager also helps to select the most appropriate driver from the previously loaded drivers when a new open database is connected.

3. The JDBC Test Suite.

The function of JDBC driver test suite is to make ensure that the JDBC drivers will run user's program or not . The test suite of JDBC application program interface is very useful for testing a driver based on JDBC technology during testing period. It ensures the requirement of Java Platform Enterprise Edition (J2EE).

4. The JDBC-ODBC Bridge.

The JDBC-ODBC bridge, also known as JDBC type 1 driver is a database driver that utilize the ODBC driver to connect the database. This driver translates JDBC method calls into ODBC function calls. The Bridge implements Jdbc for any database for which an Odbc driver is available.

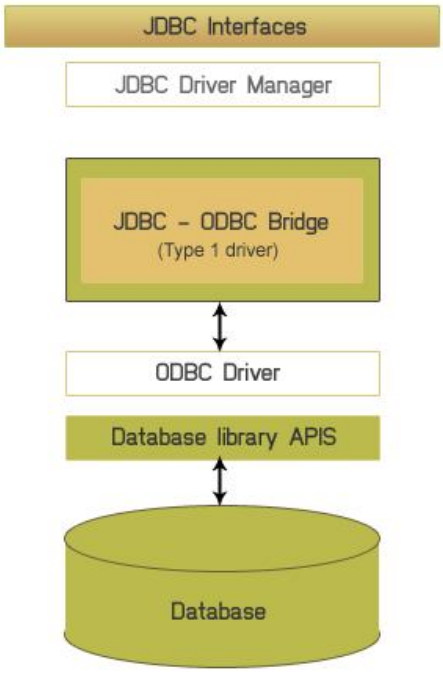
The Bridge is always implemented as the sun.jdbc.odbc Java package and it contains a native library used to access ODBC.

Types of Java Drivers

1. DBC-ODBC bridge plus ODBC driver: The Java Software bridge product provides JDBC access via ODBC drivers. Note that ODBC binary code, and in many cases database client code, must be loaded on each client machine that uses this driver. As a result, this kind of driver is most appropriate on a corporate network where client installations are not a major problem, or for application server code written in Java in a three-tier architecture.

Features

1. Convert the query of JDBC Driver into the ODBC query, which in return pass the data.
2. JDBC-ODBC is native code not written in java.
3. The connection occurs as follows -- Client -> JDBC Driver -> ODBC Driver -> Database .



Pros-->

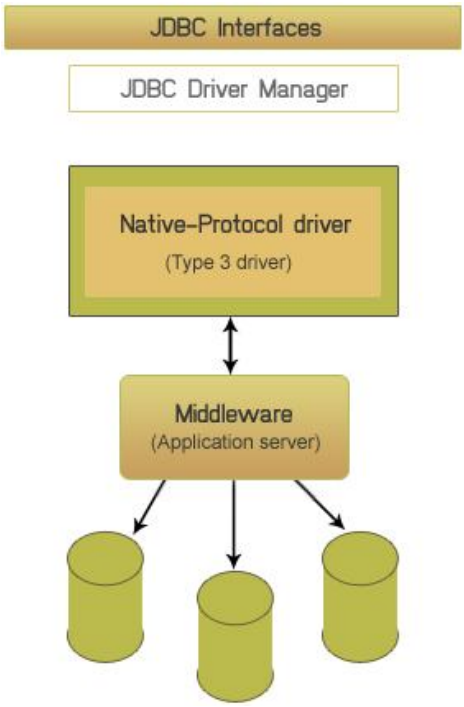
A type-1 driver is easy to install and handle

Cons-->

1. Extra channels in between database and application made performance overhead.
2. Needs to be installed on client machine.
3. Not suitable for applet , due to the installation at clients end.

2. Native-API partly-Java driver:

This kind of driver converts JDBC calls into calls on the client API for Oracle, Sybase, Informix, IBM DB2, or other DBMSs. Note that, like the bridge driver, this style of driver requires that some operating system-specific binary code be loaded on each client machine.



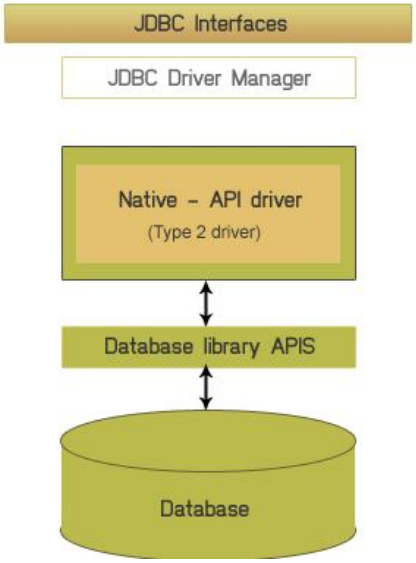
Pros--

1.Type 2 driver has additional functionality and better performance than Type 1.
2.Has faster performance than type 1,3 &4,since it has separate code for native APIS.

Cons--

1.library needs to be installed on the client machine .
2.Due to the client side software demand, it can't be used for web based application.
3.platform dependent.
4.It doesn't support "Applets".

3. JDBC-Net pure Java driver: This driver translates JDBC calls into a DBMS-independent net protocol, which is then translated to a DBMS protocol by a server. This net server middleware is able to connect its pure Java clients to many different databases. The specific protocol used depends on the vendor. In general, this is the most flexible JDBC alternative. It is likely that all vendors of this solution will provide products suitable for intranet use. In order for these products to support Internet access as well, they must handle the additional requirements for security, access through firewalls, and so forth, that the Web imposes.



Pros--

1.The client driver to middleware communication is database independent.

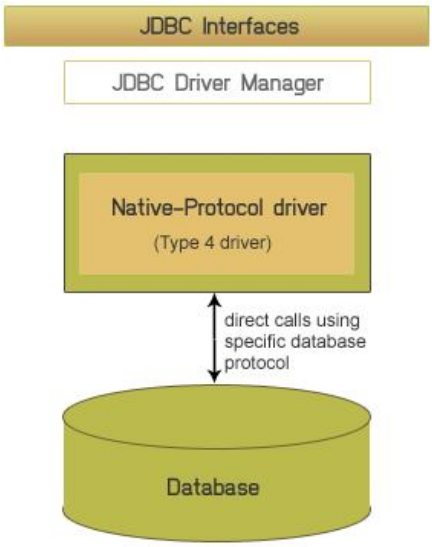
2.Can be used in internet since there is no client side software needed.

Cons--

1.Needs specific coding for different database at middleware.

2.Due to extra layer in middle can result in time-delay.

4. Native-protocol pure Java driver: This kind of driver converts JDBC calls directly into the network protocol used by DBMSs. This allows a direct call from the client machine to the DBMS server and is an excellent solution for intranet access. Since many of these protocols are proprietary, the database vendors themselves are the primary source. Several that are now available include Oracle, Sybase, Informix, IBM DB2, Inprise InterBase, and Microsoft SQL Server.



Pros--

1. Improved performance because no intermediate translator like JDBC or middleware server.
2. All the connection is managed by JVM, so debugging is easier.

JDBC Drivers

Before you can use a driver, it must be registered with the JDBC DriverManager. This is typically done by loading the driver class using the Class.forName( ) method:

```
// For MS Access Database Connectivity
try {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); //
```

```
            Connection Conn=DriverManager.getConnection("jdbc:odbc:bca2")
}
catch (ClassNotFoundException e)
{
            /* Handle Exception */
}

// For Oracle Database Connectivity
try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection Conn=DriverManager.getConnection("jdbc:odbc:bca2","scott","tiger")
}
catch (ClassNotFoundException e)
{
            /* Handle Exception */
}


// For MySql Database Connectivity
try {
            Class.forName("com.mysql.jdbc.Driver"); //
            Connection Conn=DriverManager.getConnection("jdbc:mysql://localhost/EMP")

}
catch (ClassNotFoundException e)
{
            /* Handle Exception */
}
```

Following table lists down popular JDBC driver names and database URL.

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	**jdbc:mysql://**hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	**jdbc:oracle:thin:@**hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	**jdbc:db2:**hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	**jdbc:sybase:Tds:**hostname: port Number/databaseName

All the highlighted part in URL format is static and you need to change only remaining part as per your database setup.

One reason most programs call Class.forName( ) is that this method accepts a String argument, meaning that the program can store driver selection information dynamically (e.g., in a properties file).

JDBC drivers are available for most database platforms, from a number of vendors and in a number of different flavors. There are four categories of drivers:

What Is the JDBC API?

The JDBC API is a Java API for accessing virtually any kind of tabular data. (As a point of interest, JDBC is the trademarked name and is not an acronym; nevertheless, JDBC is often thought of as standing for "Java Database Connectivity.") The JDBC API consists of a set of classes and

interfaces written in the Java programming language that provide a standard API for tool/database developers and makes it possible to write industrial strength database applications using an all-Java API.

The value of the JDBC API is that an application can access virtually any data source and run on any platform with a Java Virtual Machine. In other words, with the JDBC API, it isn't necessary to write one program to access a Sybase database, another program to access an Oracle database, another program to access an IBM DB2 database, and so on. One can write a single program using the JDBC API, and the program will be able to send SQL or other statements to the appropriate data source. And, with an application written in the Java programming language, one doesn't have to worry about writing different applications to run on different platforms. The combination of the Java platform and the JDBC API lets a programmer write once and run anywhere.

The Java programming language, being robust, secure, easy to use, easy to understand, and automatically downloadable on a network, is an excellent language basis for database applications. What is needed is a way for Java applications to talk to a variety of different data sources. JDBC is the mechanism for doing this.

What Does the JDBC API Do?

In simplest terms, a JDBC technology-based driver ("JDBC driver") makes it possible to do three things:

1. Establish a connection with a data source
2. Send queries and update statements to the data source
3. Process the results

The following code fragment gives a simple example of these three steps:

```
Connection con = DriverManager.getConnection(
                                    "jdbc:myDriver:wombat", "myLogin", "myPassword");
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");
while (rs.next()) {
        int x = rs.getInt("a");
        String s = rs.getString("b");
        float f = rs.getFloat("c");
        }
```

Connection

Connection Overview

A Connection object represents a connection with a database. A connection session includes the SQL statements that are executed and the results that are returned over that connection. A single application can have one or more connections with a single database, or it can have connections with many different databases.

A user can get information about a Connection object's database by invoking the Connection.getMetaData method. This method returns a DatabaseMetaData object that contains information about the database's tables, the SQL grammar it supports, its stored procedures, the capabilities of this connection, and so on.

Opening a Connection

The traditional way to establish a connection with a database is to call the method DriverManager.getConnection. This method takes a string containing a URL. The DriverManager class, referred to as the JDBC management layer, attempts to locate a driver that can connect to the database represented by that URL

```
String url = "jdbc:odbc:sunildb";
Connection con = DriverManager.getConnection(url, "", "");
```

Sending SQL Statements

Once a connection is established, it is used to pass SQL statements to its underlying database. The JDBC API does not put any restrictions on the kinds of SQL statements that can be sent; this provides a great deal of flexibility, allowing the use of database-specific statements or even non-SQL statements. It requires, however, that the user be responsible for making sure that the underlying database can process the SQL statements being sent and suffer the consequences if it cannot. For example, an application that tries to send a stored procedure call to a DBMS that does not support stored procedures will be unsuccessful and will generate an exception.

The JDBC API provides three interfaces for sending SQL statements to the database, and corresponding methods in the Connection interface create instances of them. The interfaces for sending SQL statements and the Connection methods that create them are as follows:

1. **Statement**-created by the **Connection.createStatement** methods. A **Statement** object is used for sending SQL statements with no parameters.
2. **PreparedStatement**-created by the **Connection.prepareStatement** methods. A **PreparedStatement** object is used for precompiled SQL statements. These can take one or more parameters as input arguments (IN parameters). **PreparedStatement** has a group of methods that set the value of IN parameters, which are sent to the database when the statement is executed. **PreparedStatement** extends Statement and therefore includes Statement methods. A **PreparedStatement** object has the potential to be more efficient than a Statement object because it has been precompiled and stored for future use. Therefore, in order to improve performance, a **PreparedStatement** object is sometimes used for an SQL statement that is executed many times.
3. **CallableStatement**-created by the **Connection.prepareCall** methods. **CallableStatement** objects are used to execute SQL stored procedures-a group of SQL statements that is called by name, much like invoking a function. A **CallableStatement** object inherits methods for

handling IN parameters from **PreparedStatement**; it adds methods for handling OUT and INOUT parameters.

The following list gives a quick way to determine which Connection method is appropriate for creating different types of SQL statements:

- **createStatement** methods-for a simple SQL statement (no parameters)
- **prepareStatement** methods-for an SQL statement that is executed frequently
- **prepareCall** methods-for a call to a stored procedure

Example Code

```
import java.sql.*;

public class MysqlConnect{
 public static void main(String[] args) {
 System.out.println("MSAccess Connect Example.");
 Connection conn = null;
 String url = "jdbc:odbc:bcadsn";
 String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
 try {
 Class.forName(driver).newInstance();
 conn = DriverManager.getConnection(url+"","");
 System.out.println("Connected to the database");
 conn.close();
 System.out.println("Disconnected from database");
 } catch (Exception e) {
 e.printStackTrace();
 }
 }
}
```

DriverManager Class

DriverManager Overview

The DriverManager class is the traditional management layer of JDBC, working between the user and the drivers. It keeps track of the drivers that are available and handles establishing a

connection between a database and the appropriate driver. In addition, the DriverManager class attends to things like driver login time limits and the printing of log and tracing messages.
For simple applications, the only method in the DriverManager class that a general programmer needs to use directly is DriverManager.getConnection. As its name implies, this method establishes a connection to a database. An application may call the DriverManager methods getDriver, getDrivers, and registerDriver as well as the Driver method connect, but in most cases it is better to let the DriverManager class manage the details of establishing a connection.

```
Class.forName("jdbc.odbc.JdbcOdbcDriver"); //loads the driver
String url = "jdbc:odbc:sunildb";
Connection con = DriverManager.getConnection(       url, "", "");
```

Statement

Statement Overview

A Statement object is used to send SQL statements to a database. There are actually three kinds of Statement objects, all of which act as containers for executing SQL statements on a given connection: Statement, PreparedStatement, which inherits from Statement, and CallableStatement, which inherits from PreparedStatement. They are specialized for sending particular types of SQL statements; a Statement object is used to execute a simple SQL statement with no parameters, a PreparedStatement object is used to execute a precompiled SQL statement with or without IN parameters, and a CallableStatement object is used to execute a call to a database stored procedure.

The Statement interface provides basic methods for executing statements and retrieving results. The PreparedStatement interface adds methods for dealing with IN parameters; the CallableStatement interface adds methods for dealing with OUT parameters.

In the JDBC 2.0 core API, the ResultSet interface has a set of new updateXXX methods and other new related methods that make it possible to update table column values programmatically. This new API also adds methods to the Statement interface (and PreparedStatement and CallableStatement interfaces) so that update statements may be executed as a batch rather than singly.

Creating Statement Objects

Once a connection to a particular database is established, that connection can be used to send SQL statements. A Statement object is created with the Connection method createStatement, as in the following code fragment:

```
Connection con = DriverManager.getConnection(url, "sunny", "");
Statement stmt = con.createStatement();
```

The SQL statement that will be sent to the database is supplied as the argument to one of the execute methods on a Statement object. This is demonstrated in the following example, which uses the method executeQuery:

```
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table2");
```

The variable rs references a result set that cannot be updated and in which the cursor can move only forward, which is the default behavior for ResultSet objects. The JDBC 2.0 core API adds a new

version of the method Connection.createStatement that makes it possible to create Statement objects that produce result sets that are scrollable and/or updatable.

Executing Statements Using Statement Objects

The Statement interface provides three different methods for executing SQL statements: executeQuery, executeUpdate, and execute. The correct method to use is determined by what the SQL statement produces.

The method executeQuery is designed for statements that produce a single result set, such as SELECT statements.

The method executeUpdate is used to execute INSERT, UPDATE, or DELETE statements and also SQL DDL (Data Definition Language) statements like CREATE TABLE, DROP TABLE, and ALTER TABLE. The effect of an INSERT, UPDATE, or DELETE statement is a modification of one or more columns in zero or more rows in a table. The return value of executeUpdate is an integer (referred to as the update count) that indicates the number of rows that were affected. For statements such as CREATE TABLE or DROP TABLE, which do not operate on rows, the return value of executeUpdate is always zero.

Example Code

```
import java.sql.*;
public class MysqlConnect{
 public static void main(String[] args) {
 System.out.println("MSAccess Connect Example.");
 Connection conn = null;
 String url = "jdbc:odbc:bcadsn";
 String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
 try {
 Class.forName(driver).newInstance();
 conn = DriverManager.getConnection(url," ", " ");
 System.out.println("Connected to the database");

String sql="create table stud (id number, rollno number, name varchar(10))";
Statement stmt=conn.createStatement();

Int update=stmt.executeUpdated(sql);
 conn.close();
 System.out.println("Disconnected from database");
 } catch (Exception e) {
 e.printStackTrace();
 }
 }
}
```

Closing Statements

Statement objects will be closed automatically by the Java garbage collector. Nevertheless, it is recommended as good programming practice that they be closed explicitly when they are no

longer needed. This frees DBMS resources immediately and helps avoid potential memory problems.

Batch Updates

The batch update facility provided by the JDBC 2.0 core API allows a Statement object to submit multiple update commands together as a single unit, or batch, to the underlying DBMS. This ability to submit multiple updates as a batch rather than having to send each update individually can improve performance greatly in some situations.

The following code fragment demonstrates how to send a batch update to a database. In this example, a new row is inserted into three different tables in order to add a new employee to a company database. The code fragment starts by turning off the Connection object con's auto-commit mode in order to allow multiple statements to be sent together as a transaction. After creating the Statement object stmt, it adds three SQL INSERT INTO commands to the batch with the method addBatch and then sends the batch to the database with the method executeBatch. The code looks like this:

```
Statement stmt = con.createStatement();
con.setAutoCommit(false);

stmt.addBatch("INSERT INTO employees VALUES (1000, 'Joe Jones')");
stmt.addBatch("INSERT INTO departments VALUES (260, 'Shoe')");
stmt.addBatch("INSERT INTO emp_dept VALUES (1000, '260')");

int [] updateCounts = stmt.executeBatch();
```

PreparedStatement

PreparedStatement Overview

The PreparedStatement interface inherits from Statement and differs from it in two ways:

1. Instances of PreparedStatement contain an SQL statement that has already been compiled. This is what makes a statement "prepared."
2. The SQL statement contained in a PreparedStatement object may have one or more IN parameters. An IN parameter is a parameter whose value is not specified when the SQL statement is created. Instead, the statement has a question mark ("?") as a placeholder for each IN parameter. The "?" is also known as a parameter marker. An application must set a value for each question mark in a prepared statement before executing the prepared statement.

Because PreparedStatement objects are precompiled, their execution can be faster than that of Statement objects. Consequently, an SQL statement that is executed many times is often created as a PreparedStatement object to increase efficiency.

PreparedStatement interface

The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.
Let's see the example of parameterized query:

```
1. String sql="insert into emp values(?,?,?)";
```

As you can see, we are passing parameter (?) for the values. Its value will be set by calling the setter methods of PreparedStatement.

Why use PreparedStatement?

Improves performance: The performance of the application will be faster if you use PreparedStatement interface because query is compiled only once.

How to get the instance of PreparedStatement?

The prepareStatement() method of Connection interface is used to return the object of PreparedStatement. Syntax:

```
1. public PreparedStatement prepareStatement(String query)throws SQLException{}
```

Methods of PreparedStatement interface

The important methods of PreparedStatement interface are given below:

Method	Description
public void setInt(int paramIndex, int value)	sets the integer value to the given parameter index.
public void setString(int paramIndex, String value)	sets the String value to the given parameter index.
public void setFloat(int paramIndex, float value)	sets the float value to the given parameter index.
public void setDouble(int paramIndex, double value)	sets the double value to the given parameter index.
public int executeUpdate()	executes the query. It is used for create, drop, insert, update, delete etc.
public ResultSet executeQuery()	executes the select query. It returns an instance of ResultSet.

Example of PreparedStatement interface that inserts the record

First of all create table as given below:

```
1. create table emp(id number(10),name varchar2(50));
```

Now insert records in this table by the code given below:

```
1. import java.sql.*;
2. class InsertPrepared{
3. public static void main(String args[]){
4. try{
5. Class.forName("oracle.jdbc.driver.OracleDriver");
6.  Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","
   oracle");
7.  PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");
8. stmt.setInt(1,101);//1 specifies the first parameter in the query
9. stmt.setString(2,"Ratan");
10.  int i=stmt.executeUpdate();
11. System.out.println(i+" records inserted");
12.  con.close();
13.  }catch(Exception e){ System.out.println(e);}
14.  } }
```

Example of PreparedStatement interface that updates the record

```
1. PreparedStatement stmt=con.prepareStatement("update emp set name=? where id=?");
2. stmt.setString(1,"Sonoo");//1 specifies the first parameter in the query i.e. name
3. stmt.setInt(2,101);
4.  int i=stmt.executeUpdate();
5. System.out.println(i+" records updated");
```

Example of PreparedStatement interface that deletes the record

```
1. PreparedStatement stmt=con.prepareStatement("delete from emp where id=?");
2. stmt.setInt(1,101);
3.  int i=stmt.executeUpdate();
4. System.out.println(i+" records deleted");
```

Example of PreparedStatement interface that retrieve the records of a table

```
1. PreparedStatement stmt=con.prepareStatement("select * from emp");
2. ResultSet rs=stmt.executeQuery();
3. while(rs.next()){
4. System.out.println(rs.getInt(1)+" "+rs.getString(2));
```

5. }

Example of PreparedStatement to insert records until user press n

```
1. import java.sql.*;
2. import java.io.*;
3. class RS{
4. public static void main(String args[])throws Exception{
5. Class.forName("oracle.jdbc.driver.OracleDriver");
6. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
7. PreparedStatement ps=con.prepareStatement("insert into emp130 values(?,?,?)");
8. BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
9. do{
10. System.out.println("enter id:");
11. int id=Integer.parseInt(br.readLine());
12. System.out.println("enter name:");
13. String name=br.readLine();
14. System.out.println("enter salary:");
15. float salary=Float.parseFloat(br.readLine());

16. ps.setInt(1,id);
17. ps.setString(2,name);
18. ps.setFloat(3,salary);
19. int i=ps.executeUpdate();
20. System.out.println(i+" records affected");

21. System.out.println("Do you want to continue: y/n");
22. String s=br.readLine();
23. if(s.startsWith("n")){
24. break;
25. }
```

```
26. }while(true);
27. con.close(); }}
```

PreparedeStatement Example

```
import java.sql.*;
public class PreparedStatementSetObject{
 public static void main(String[] args) {
 System.out.println("Prepared Statement Set Array Example!");
 Connection con = null;
 try{
 Class.forName("com.mysql.jdbc.Driver");
 con = DriverManager.getConnection("jdbc:odbc:bcadsn"," ","   ");
 PreparedStatement prest = con.prepareStatement("insert emp_sal values(?,?)");
 prest.setObject(1,"Sushil");
 prest.setObject(2,15000);
 int n = prest.executeUpdate();
 System.out.println(n + " Record is added in the table.");
 con.close();
 }
 catch (SQLException s){       System.out.println("SQL statement is not executed!");
 }  }
}
```

CallableStatement

CallableStatement Overview

A CallableStatement object provides a way to call stored procedures in a standard way for all RDBMSs. A stored procedure is stored in a database; the *call* to the stored procedure is what a CallableStatement object contains. This call is written in an escape syntax that may take one of two forms: one form with a result parameter, and the other without one. A result parameter, a kind of OUT parameter, is the return value for the stored procedure. Both forms may have a variable number of parameters used for input (IN parameters), output (OUT parameters), or both (INOUT parameters). A question mark serves as a placeholder for a parameter.

The syntax for invoking a stored procedure using the JDBC API is shown here. Note that the square brackets indicate that what is between them is optional; they are not themselves part of the syntax.

```
{call procedure_name[(?, ?, ...)]}
```

The syntax for a procedure that returns a result parameter is:

```
{? = call procedure_name[(?, ?, ...)]}
```

The syntax for a stored procedure with no parameters would look like this:

```
{call procedure_name}
```

Normally, anyone creating a CallableStatement object would already know that the DBMS being used supports stored procedures and what those procedures are. If one needed to check, however, various DatabaseMetaData methods will supply such information. For instance, the method supportsStoredProcedures will return true if the DBMS supports stored procedure calls, and the method getProcedures will return a description of the stored procedures available.

CallableStatement inherits Statement methods, which deal with SQL statements in general, and it also inherits PreparedStatement methods, which deal with IN parameters. All of the methods defined in CallableStatement deal with OUT parameters or the output aspect of INOUT parameters: registering the JDBC types of the OUT parameters, retrieving values from them, or checking whether a returned value was JDBC NULL. Whereas the getXXX methods defined in ResultSet retrieve values from a result set, the getXXX methods in CallableStatement retrieve values from the OUT parameters and/or return value of a stored procedure.

Creating a CallableStatement Object

CallableStatement objects are created with the Connection method prepareCall. The following example, in which con is an active JDBC Connection object, creates an instance of CallableStatement.

```
CallableStatement cstmt = con.prepareCall(
        "{call getTestData(?, ?)}");
```

The variable cstmt contains a call to the stored procedure getTestData, which has two input parameters and no result parameter. Whether the ? placeholders are IN, OUT, or INOUT parameters depends on the stored procedure getTestData. This instance of a CallableStatement object was created using JDBC 1.0 API; consequently, any query in the stored procedure called by cstmt will produce a default ResultSet object (one that is non-scrollable and non-updatable).

ResultSet

ResultSet Overview

A ResultSet is a Java object that contains the results of executing an SQL query. In other words, it contains the rows that satisfy the conditions of the query. The data stored in a ResultSet object is retrieved through a set of get methods that allows access to the various columns of the current row. The ResultSet.next method is used to move to the next row of the ResultSet, making it the current row.

The general form of a result set is a table with column headings and the corresponding values returned by a query. For example, if your query is SELECT a, b, c FROM Table1, your result set will have the following form:
The following code fragment is an example of executing an SQL statement that will return a collection of rows, with column a as an int, column b as a String, and column c as a float:

Example Code

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");
while (rs.next()) {
        // retrieve and print the values for the current row
        int i = rs.getInt("a");
        String s = rs.getString("b");
        float f = rs.getFloat("c");
        System.out.println("ROW = " + i + " " + s + " " + f);
}
```

Java ResultSetMetaData Interface

The metadata means data about data i.e. we can get further information from the data.
If you have to get metadata of a table like total number of column, column name, column type etc. , ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

Commonly used methods of ResultSetMetaData interface

Method	Description
public int getColumnCount()throws SQLException	it returns the total number of columns in the ResultSet object.
public String getColumnName(int index)throws SQLException	it returns the column name of the specified column index.
public String getColumnTypeName(int index)throws SQLException	it returns the column type name for the specified index.
public String getTableName(int index)throws SQLException	it returns the table name for the specified column index.

How to get the object of ResultSetMetaData:

The getMetaData() method of ResultSet interface returns the object of

ResultSetMetaData. Syntax:

```
1.  public ResultSetMetaData getMetaData()throws SQLException
```

Example of ResultSetMetaData interface :

```
1.  import java.sql.*;
2.  class Rsmd{
3.  public static void main(String args[]){
4.  try{
5.  Class.forName("oracle.jdbc.driver.OracleDriver");
6.  Connection con=DriverManager.getConnection(
7.  "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
8.
9.  PreparedStatement ps=con.prepareStatement("select * from emp");
10. ResultSet rs=ps.executeQuery();
11. ResultSetMetaData rsmd=rs.getMetaData();
12.
13. System.out.println("Total columns: "+rsmd.getColumnCount());
14. System.out.println("Column Name of 1st column: "+rsmd.getColumnName(1));
15. System.out.println("Column Type Name of 1st column: "+rsmd.getColumnTypeName(1));
16.
17. con.close();
18. }catch(Exception e){ System.out.println(e);}
19. }
20. }
```

Cursors

A ResultSet object maintains a cursor, which points to its current row of data. The cursor moves down one row each time the method next is called. When a ResultSet object is first created, the cursor is positioned before the first row, so the first call to the next method puts the cursor on the first row, making it the current row. ResultSet rows can be retrieved in sequence from top to bottom as the cursor moves down one row with each successive call to the method next. This ability to move its cursor only forward is the default behavior for a ResultSet and is the only cursor movement possible with drivers that implement only the JDBC 1.0 API. This kind of result set has the type ResultSet.TYPE_FORWARD_ONLY and is referred to as a forward only result set.

Cursor Movement Examples

As stated in the previous section, the standard cursor movement for forward only result sets is to use the method next to iterate through each row of a result set once from top to bottom. With scrollable result sets, it is possible to revisit a row or to iterate through the result set multiple times. This is possible because the cursor can be moved before the first row at any time (with the method beforeFirst method (ResultSet interface)>beforeFirst). The cursor can begin another iteration through the result set with the method next. The following example positions the cursor before the first row and then iterates forward through the contents of the result set. The methods getString and getFloat retrieve the column values for each row until there are no more rows, at which time the method next returns the value false.

```
rs.beforeFirst();
while (rs.next()) {
        System.out.println(rs.getString("EMP_NO") +
                " " + rs.getFloat("SALARY");
}
```

It is also possible to iterate through a result set backwards, as is shown in the next example. The cursor is first moved to the very end of the result set (with the method afterLast), and then the method previous is invoked within a while loop to iterate through the contents of the result set by moving to the previous row with each iteration. The method previous returns false when there are no more rows, so the loop ends after all the rows have been visited.

```
rs.afterLast();
while (rs.previous()) {
        System.out.println(rs.getString("EMP_NO") +
                " " + rs.getFloat("SALARY");
}
```

The interface ResultSet offers still other ways to iterate through the rows of a scrollable result set. Care should be taken, however, to avoid incorrect alternatives such as the one illustrated in the following example:

```
// incorrect!
while (!rs.isAfterLast()) {
 rs.relative(1);
 System.out.println(
        rs.getString("EMP_NO") + " " + rs.getFloat("SALARY"));
}
```

This example attempts to iterate forward through a scrollable result set and is incorrect for several reasons. One error is that if ResultSet.isAfterLast is called when the result set is empty, it will return a value of false since there is no last row. The loop body will be executed, which is not what is wanted. An additional problem occurs when the cursor is positioned before the first row of a result set that contains data. In this case, calling rs.relative(1) is erroneous because there is no

current row. The method relative moves the cursor the specified number of rows from the current row, and it must be invoked only while the cursor is on the current row.

The following code fragment fixes the problems in the previous example. Here a call to the method ResultSet.first is used to distinguish the case of an empty result set from one that contains data. Because ResultSet.isAfterLast is called only when the result set is non-empty, the loop control works correctly. Since ResultSet.first method (ResultSet interface)>first initially positions the cursor on the first row, the method ResultSet.relative(1) steps through the rows of the result set as expected.

```
if (rs.first()) {
 while (!rs.isAfterLast()) {
  System.out.println(
     rs.getString("EMP_NO") + " " + rs.getFloat("SALARY"));
  rs.relative(1);
  }
}
```

Types of Result Sets

Results sets may have different levels of functionality. For example, they may be scrollable or nonscrollable. A scrollable result set has a cursor that moves both forward and backward and can be moved to a particular row. Also, result sets may be sensitive or insensitive to changes made while they are open; that is, they may or may not reflect changes to column values that are modified in the database. A developer should always keep in mind the fact that adding capabilities to a ResultSet object incurs additional overhead, so it should be done only as necessary.

Based on the capabilities of scrollability and sensitivity to changes, there are three types of result sets available with the JDBC 2.0 core API. The following constants, defined in the ResultSet interface, are used to specify these three types of result sets:

1. **TYPE_FORWARD_ONLY**
   - The result set is nonscrollable; its cursor moves forward only, from top to bottom.
   - The view of the data in the result set depends on whether the DBMS materializes results incrementally.
2. **TYPE_SCROLL_INSENSITIVE**
   - The result set is scrollable: Its cursor can move forward or backward and can be moved to a particular row or to a row whose position is relative to its current position.
   - The result set generally does not show changes to the underlying database that are made while it is open. The membership, order, and column values of rows are typically fixed when the result set is created.
3. **TYPE_SCROLL_SENSITIVE**
   - The result set is scrollable; its cursor can move forward or backward and can be moved to a particular row or to a row whose position is relative to its current position.
   - The result set is sensitive to changes made while it is open. If the underlying column values are modified, the new values are visible, thus providing a dynamic view of the underlying data. The membership and ordering of rows in the result set may be fixed or not, depending on the implementation.