

Core Java

Core Java

Core Java

Java technology is widely used currently. Let's start learning of java from basic questions like what is java, where it is used, what type of applications are created in java and why use java?

What is Java?

Java is a **programming language** and a **platform**.

Platform Any hardware or software environment in which a program runs, 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:

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.

Core 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.

Difference between JDK,JRE and JVM

Understanding the difference between JDK, JRE and JVM is important in Java. We will have brief overview of JVM here. If you want to gain the detailed knowledge of JVM, 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 (i.e.JVM is platform dependent).

The JVM performs four main tasks:

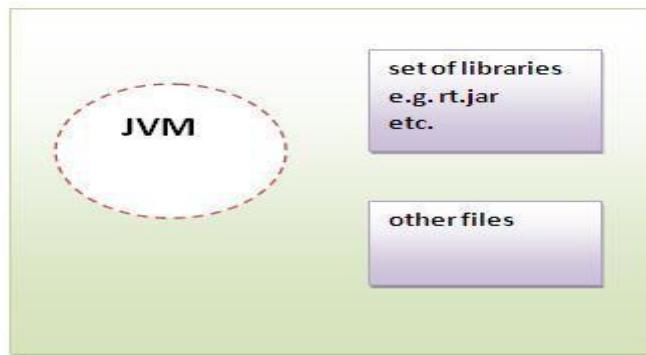
- Loads code
- Verifies code
- Executes code
- Provides runtime environment

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.

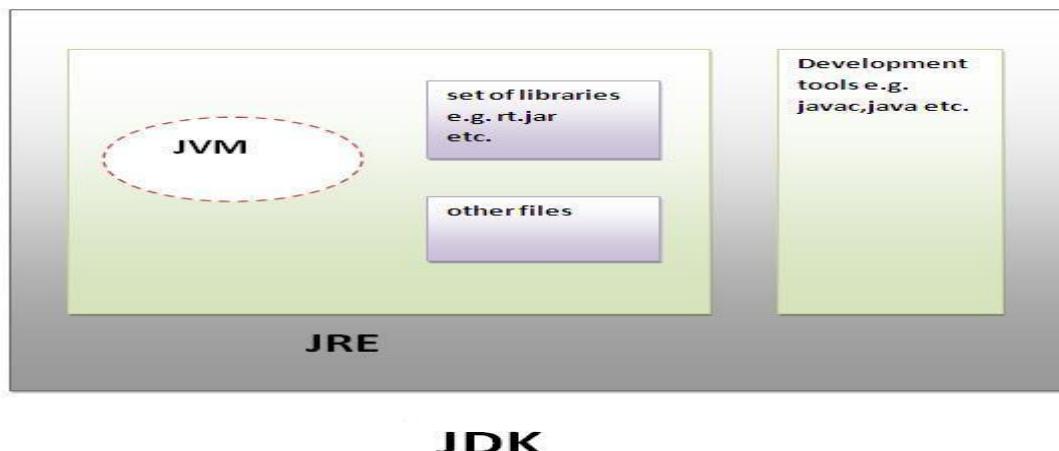
Core Java



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 platform dependent).

The JVM performs four main tasks:

- Loads code
- Verifies code
- Executes code

Core Java

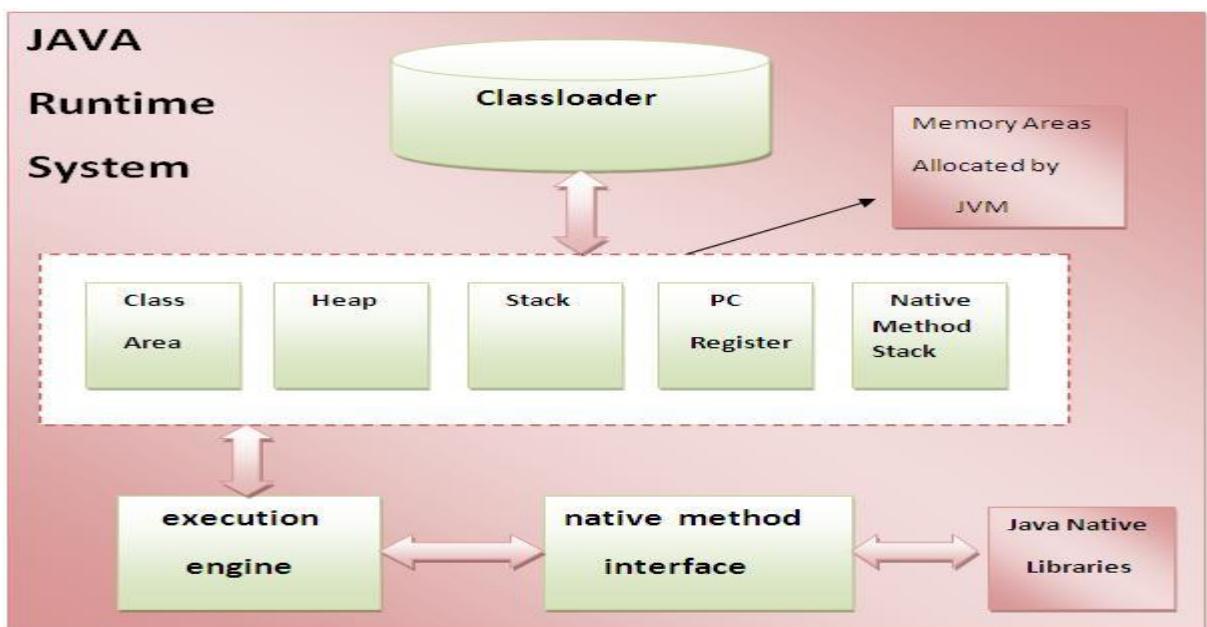
- Provides runtime environment

JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

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.

Core Java

4) Stack:

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

Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

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.

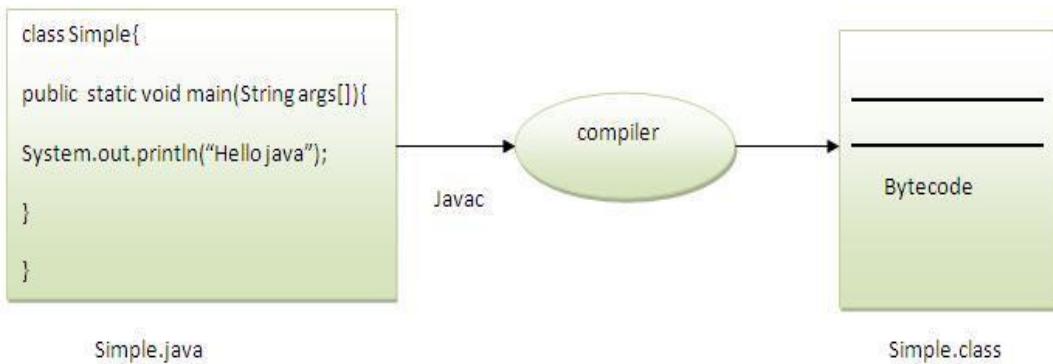
Internal Details of Hello Java Program

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 questions based on the first program.

What happens at compile time?

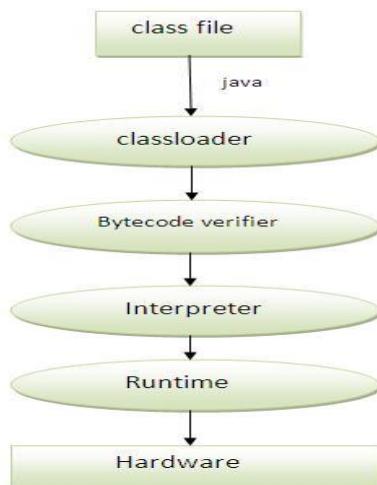
At compile time, java file is compiled by Java Compiler (It does not interact with OS) and converts the java code into bytecode.

Core Java



What happens at runtime?

At runtime, following steps are performed:



Classloader: is the subsystem of JVM that is used to load class files.

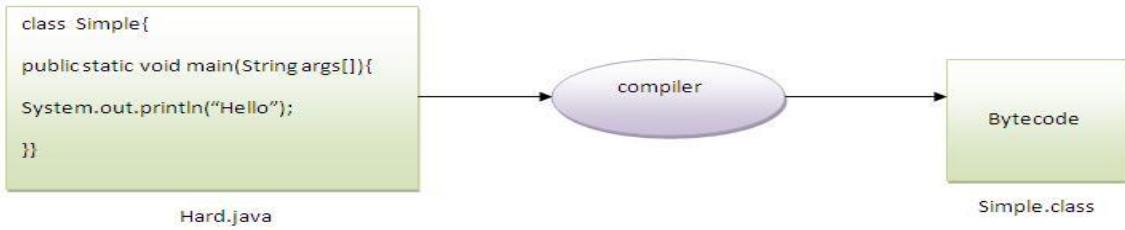
Bytecode Verifier: checks the code fragments for illegal code that can violate accesss right to objects.

Interpreter: read bytecode stream then execute the instructions.

Q)Can you save a java source file by other name than the class name?

Yes, like the figure given below illustrates:

Core Java

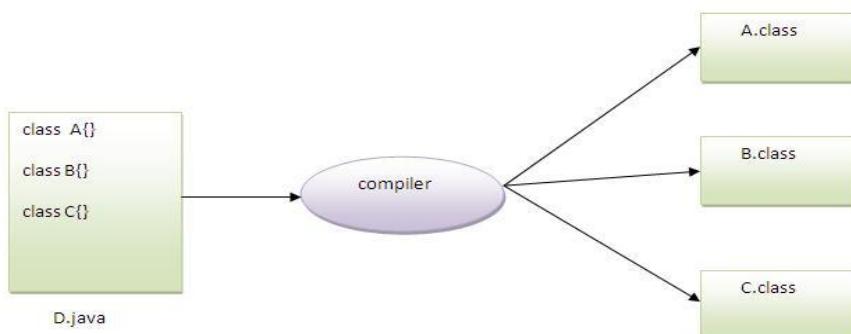


To compile: javac Hard.java

To execute: java Simple

Q) Can you have multiple classes in a java source file?

Yes, like the figure given below illustrates:



Unicode System

Unicode is a universal international standard character encoding that is capable of representing most of the world's written languages.

Why java uses Unicode System?

Before Unicode, there were many language standards:

- **ASCII** (American Standard Code for Information Interchange) for the United States.
- **ISO 8859-1** for Western European Language.
- **KOI-8** for Russian.

Core Java

- **GB18030 and BIG-5** for chinese, and so on.

This caused two problems:

1. A particular code value corresponds to different letters in the various language standards.
2. The encodings for languages with large character sets have variable length. Some common characters are encoded as single bytes, others require two or more bytes.

To solve these problems, a new language standard was developed i.e. Unicode System.

In unicode, character holds 2 byte, so java also uses 2 byte for characters.

lowest value:\u0000

highest value:\uFFFF

History of Java

Java history is interesting to know. 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.

At that time, it was an advanced concept for the green team. But, it was good for internet programming. Later, Netscape Navigator incorporated Java technology.

Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc. Let's see the major points that describe 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.
- 5) **Why Oak?** Oak is a symbol of strength and chosen 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.
- 7) **Why they chose java name for java language?** The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA" etc. They

Core Java

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.

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)

Features of Java

There is given many features of java. They are also called java buzzwords.

- 1.Simple
- 2.Object-oriented
- 3.Platform independent
- 4.Secured
- 5.Robust

Core Java

6. Architecture neutral

7. Portable

8. Dynamic

9. Interpreted

10. High Performance

11. Multithreaded

12. Distributed

Simple

Java is simple in the sense that:

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

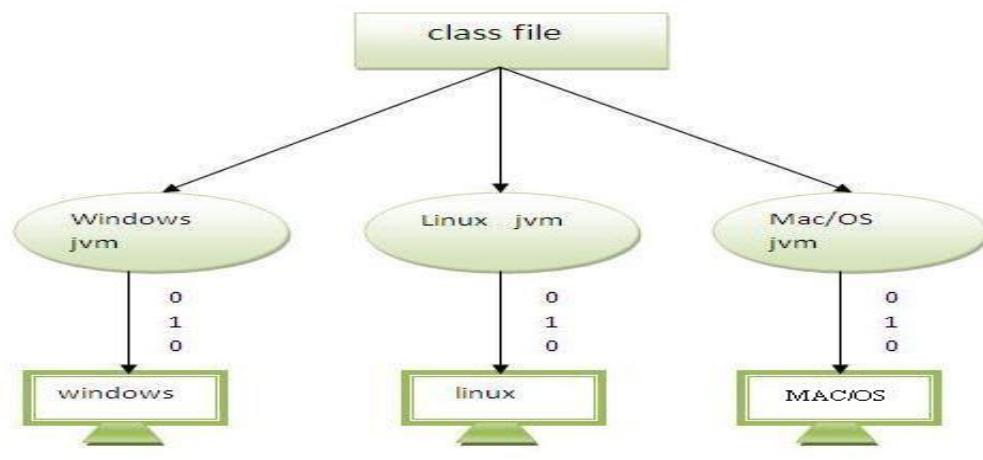
Core Java

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



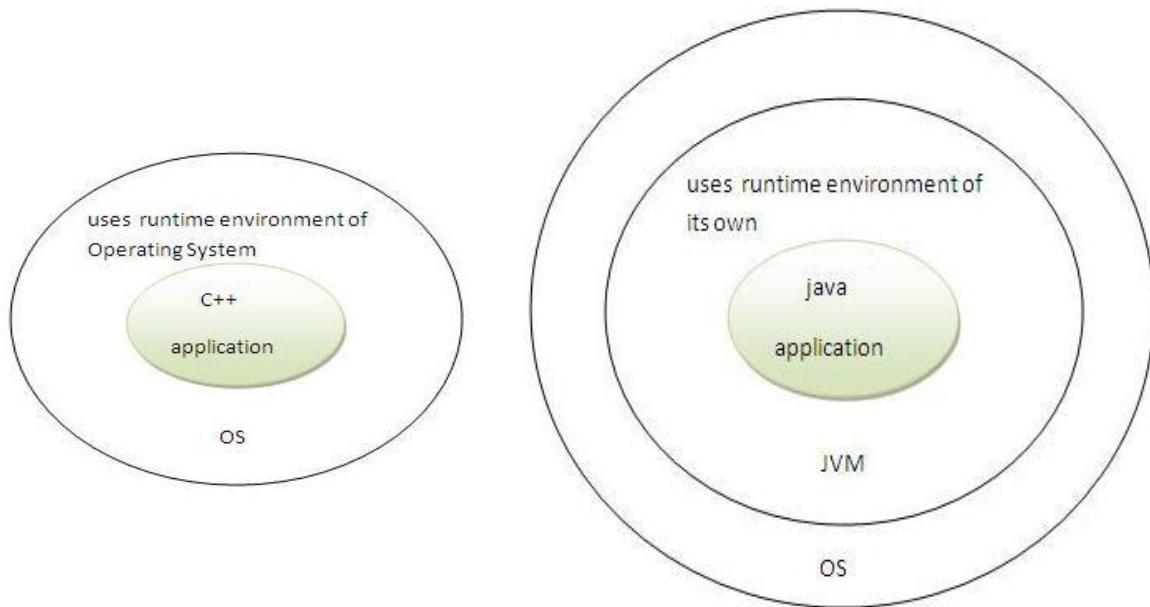
compiled by the compiler and converted into bytecode. This bytecode is a platform independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere(WORA).

Secured

Java is secured because:

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

Core Java



- **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 accesss 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 sucurity can also be provided by application developer through SSL,JAAS,cryptography 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

Core Java

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.

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.

Simple Program of Java

In this page, we will learn how to write the hello java program. Creating hello java example is too easy. Here, we have created a class named Simple that contains only main method and prints a message hello java. It is the simple program of java.

Requirement for Hello Java Example

For executing any java program, you need to

- create the java program.
- install the JDK if you don't have installed it, [download the JDK](#) and install it.
- set path of the bin directory under jdk.
- compile and execute the program.

Creating hello java example

Let's create the hello java program:

```
1. class Simple{  
2.     public static void main(String args[]){  
3.         System.out.println("Hello Java")  
4.     }  
5. }
```

save this file as Simple.java

Core Java

To compile: javac Simple.java

To execute: java Simple

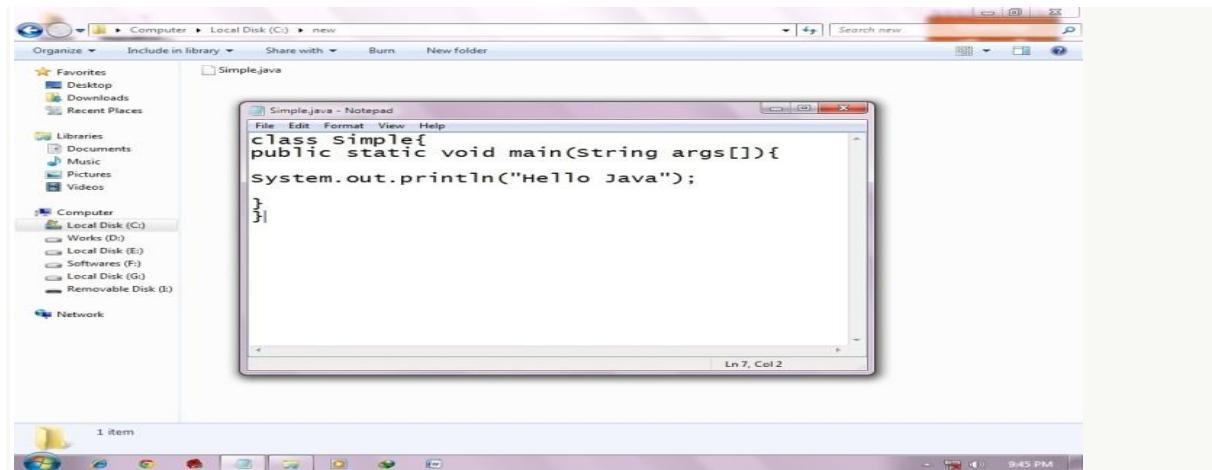
Output:Hello Java

Understanding first java program

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

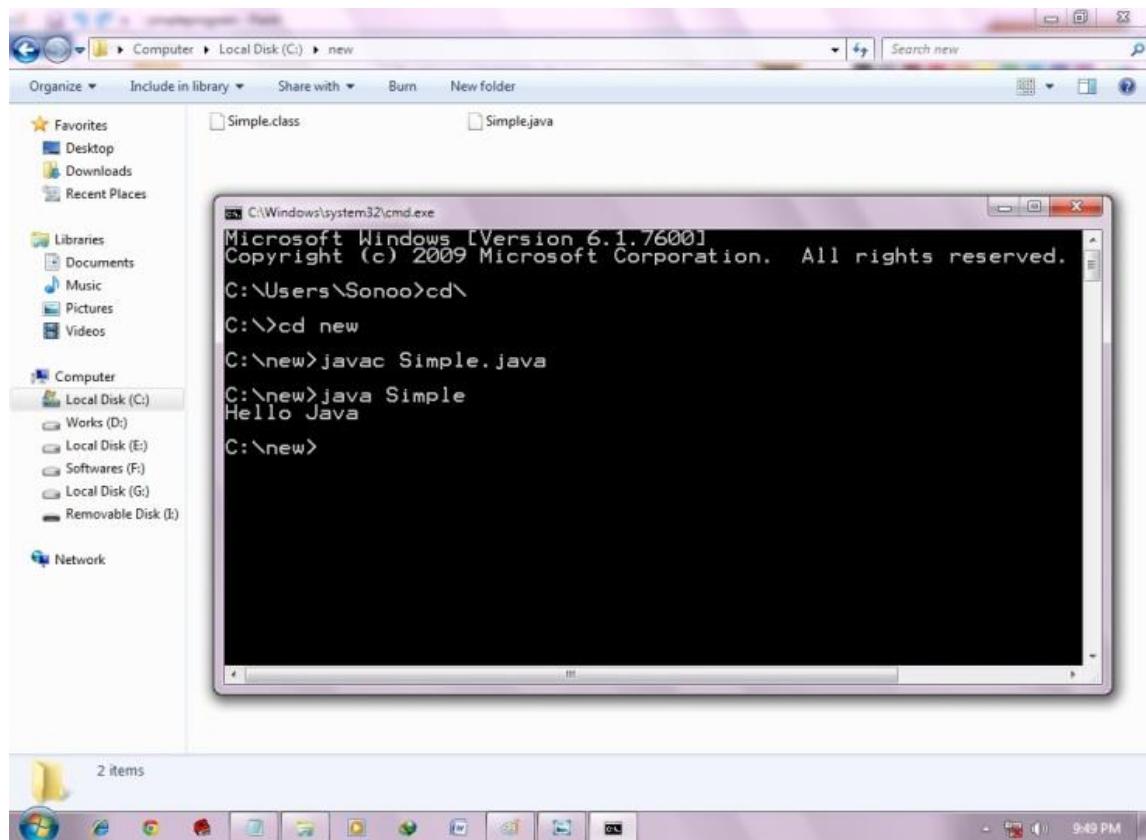
- **class** is used to declare a class in java.
- **public** is an access modifier which represents visibility, it means it is visible to all.
- **static** is a keyword, if we declare any method as static, it is known as static method. The core advantage of static method is that there is no need to create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

To write the simple program, open notepad and write simple program as displayed below:



As displayed in the above diagram, write the simple program of java in notepad and saved it as Simple.java. To compile and run this program, you need to open command prompt by start -> All Programs -> Accessories -> command prompt.

Core Java

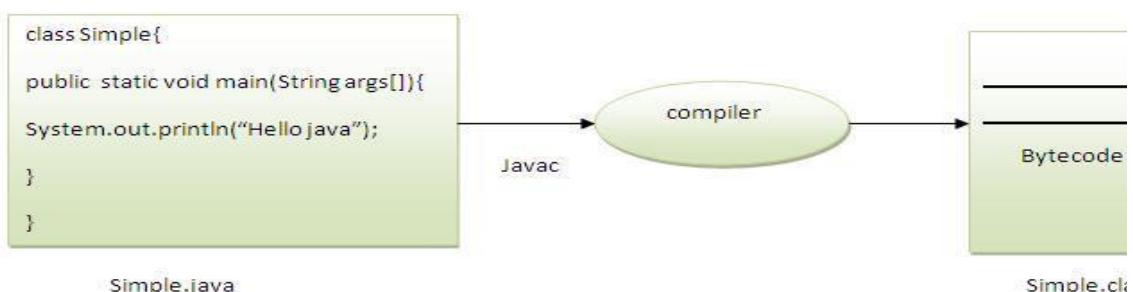


Internal Details of Hello Java Program

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 questions based on the first program.

What happens at compile time?

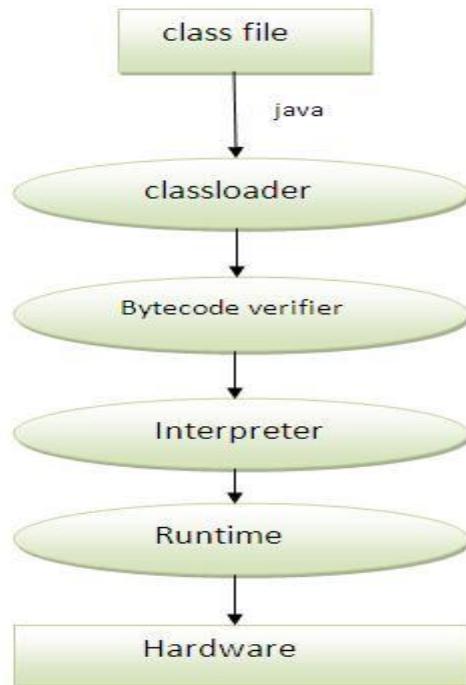
At compile time, java file is compiled by Java Compiler (It does not interact with OS) and converts the java code into bytecode.



Core Java

What happens at runtime?

At runtime, following steps are performed:



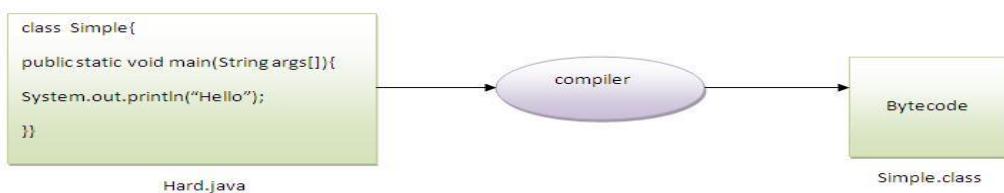
Classloader: is the subsystem of JVM that is used to load class files.

Bytecode Verifier: checks the code fragments for illegal code that can violate accesss right to objects.

Interpreter: read bytecode stream then execute the instructions.

Q)Can you save a java source file by other name than the class name?

Yes, like the figure given below illustrates:



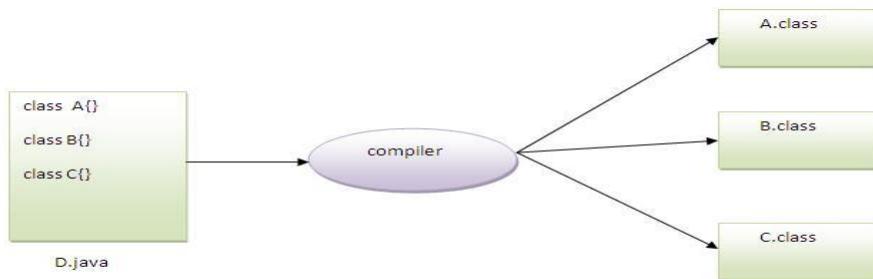
Core Java

To compile: javac Hard.java

To execute: java Simple

Q)Can you have multiple classes in a java source file?

Yes, like the figure given below illustrates:



How to set path of JDK in Windows:

Path is required for using tools such as javac, java etc. If you are saving the java file in jdk/bin folder, path is not required. But If you are having your java file outside the jdk/bin folder, it is necessary to set path of JDK. There are two ways to set path of JDK:

1. temporary
2. permanent

1)Setting temporary Path of JDK in Windows:

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

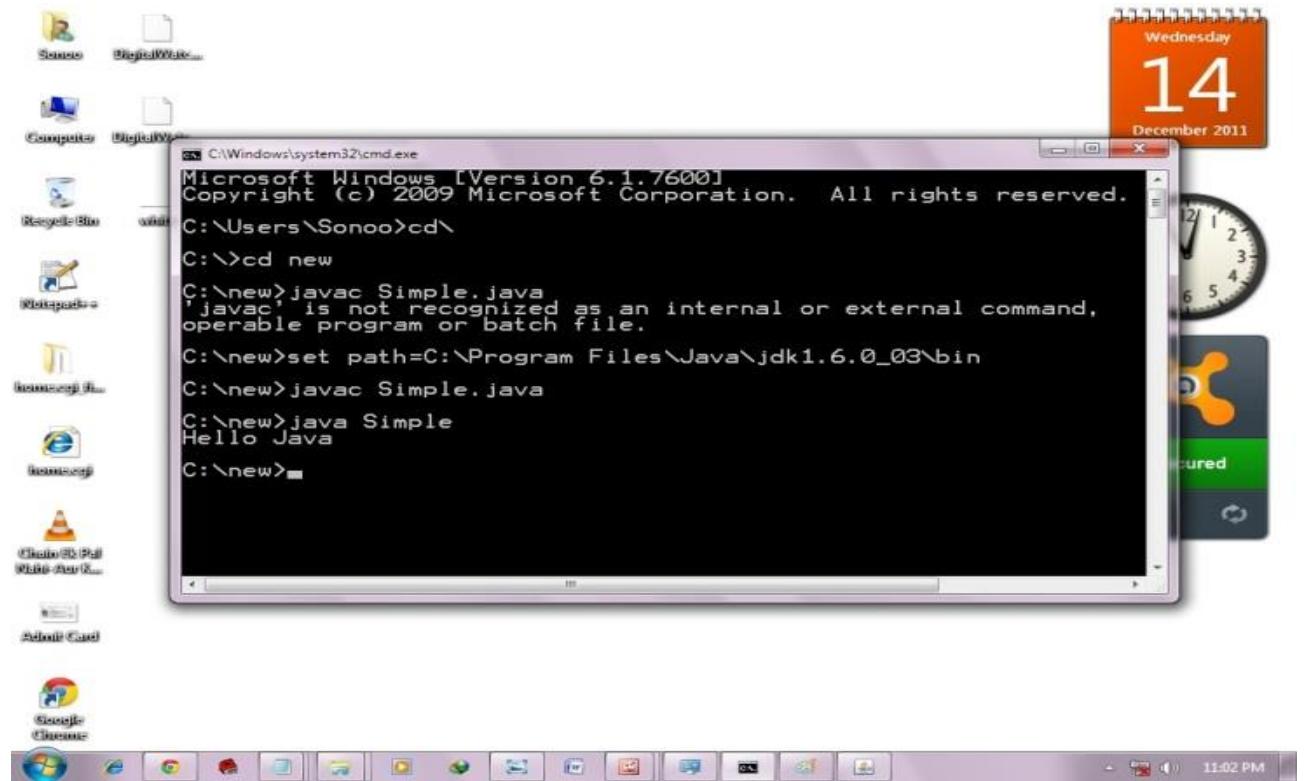
- Open command prompt
- copy the path of bin folder
- write in command prompt: set path=copiedpath

For Example:

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

Let's see it in the figure given below:

Core Java



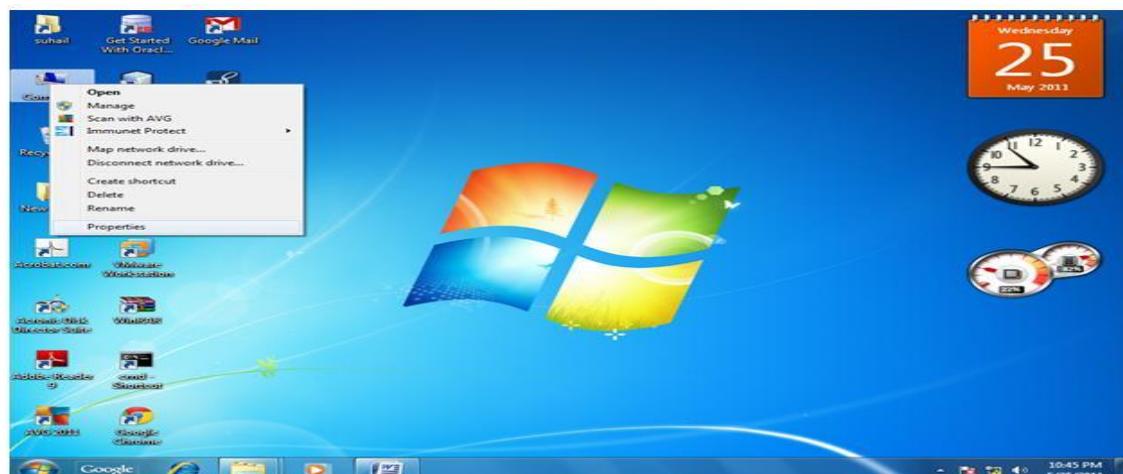
2)Setting Permanent Path of JDK in Windows:

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

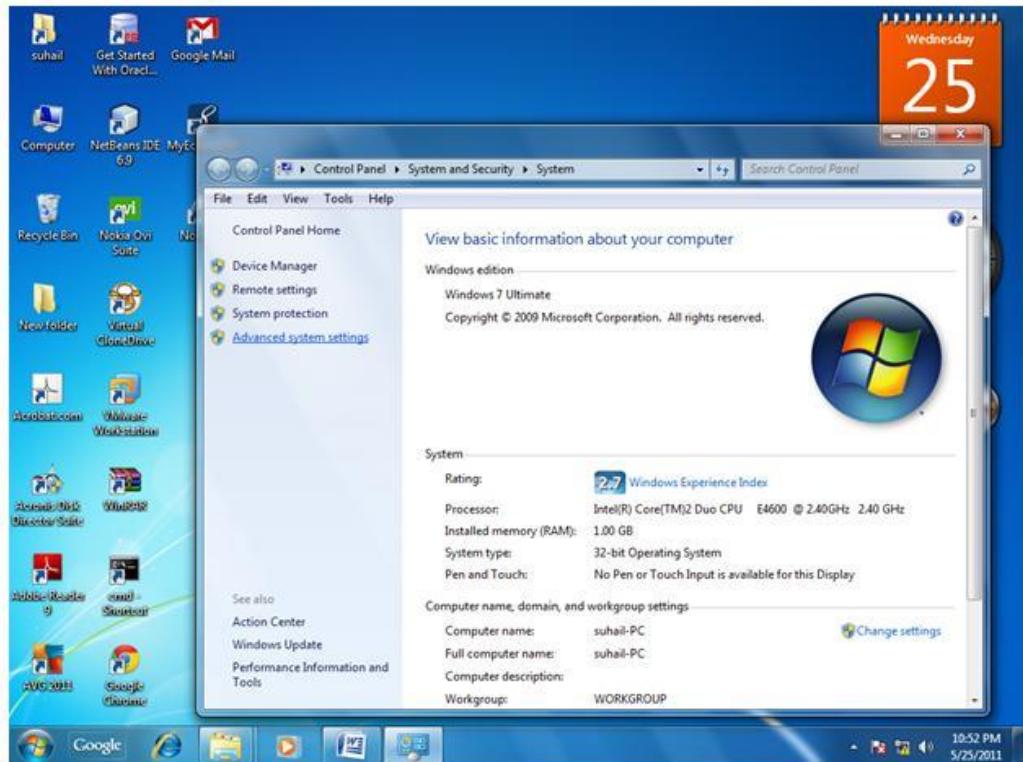
For Example:

1)Go to MyComputer properties

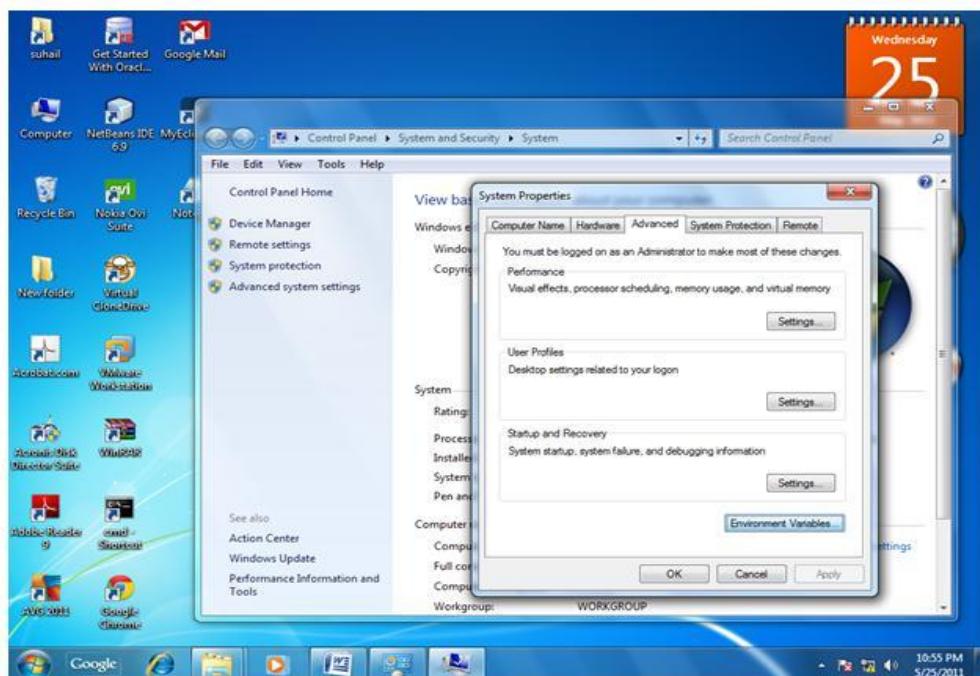


2)click on advanced tab

Core Java

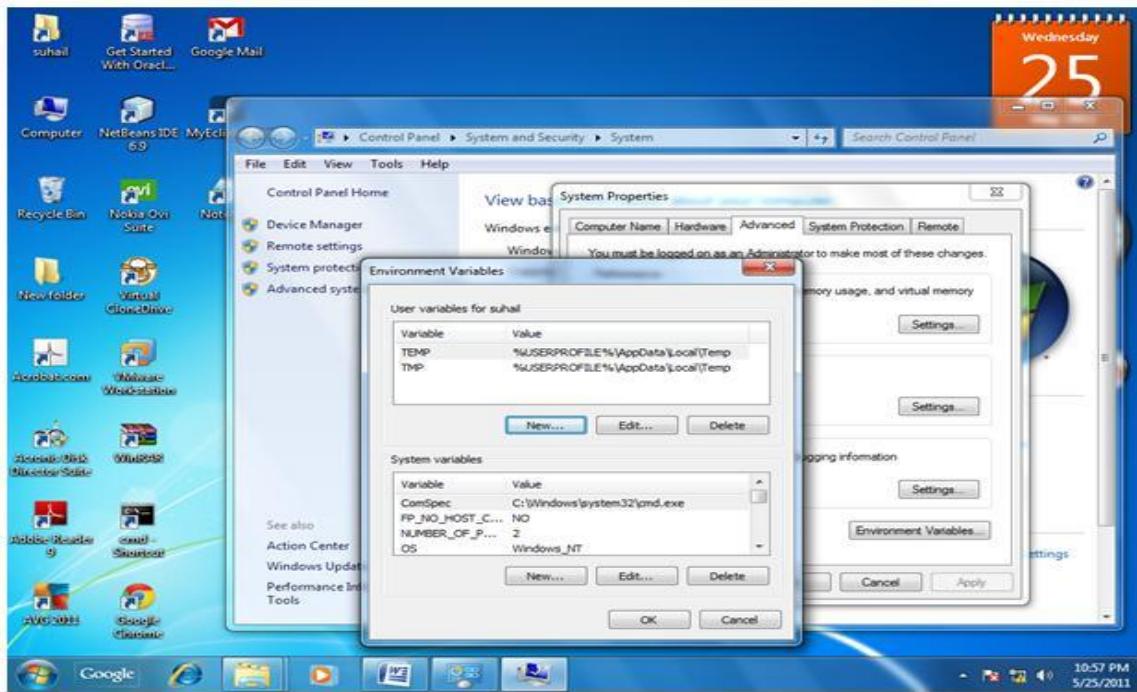


3)click on environment variables

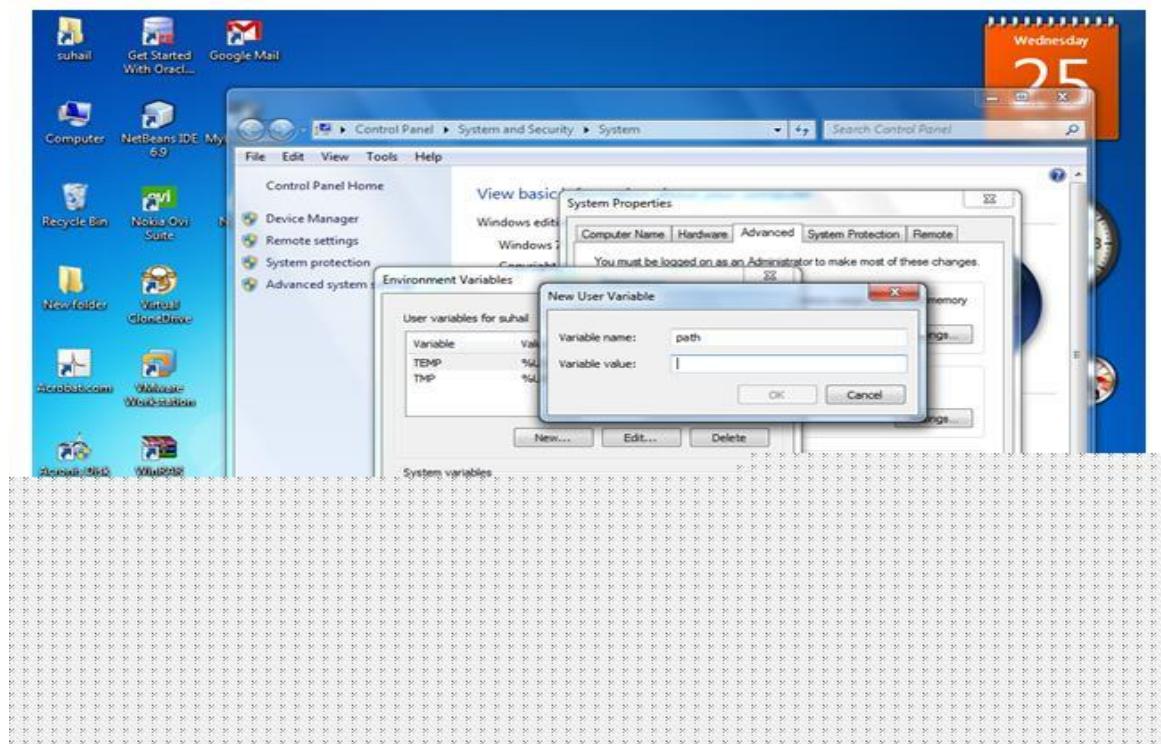


4)click on new tab of user variables

Core Java

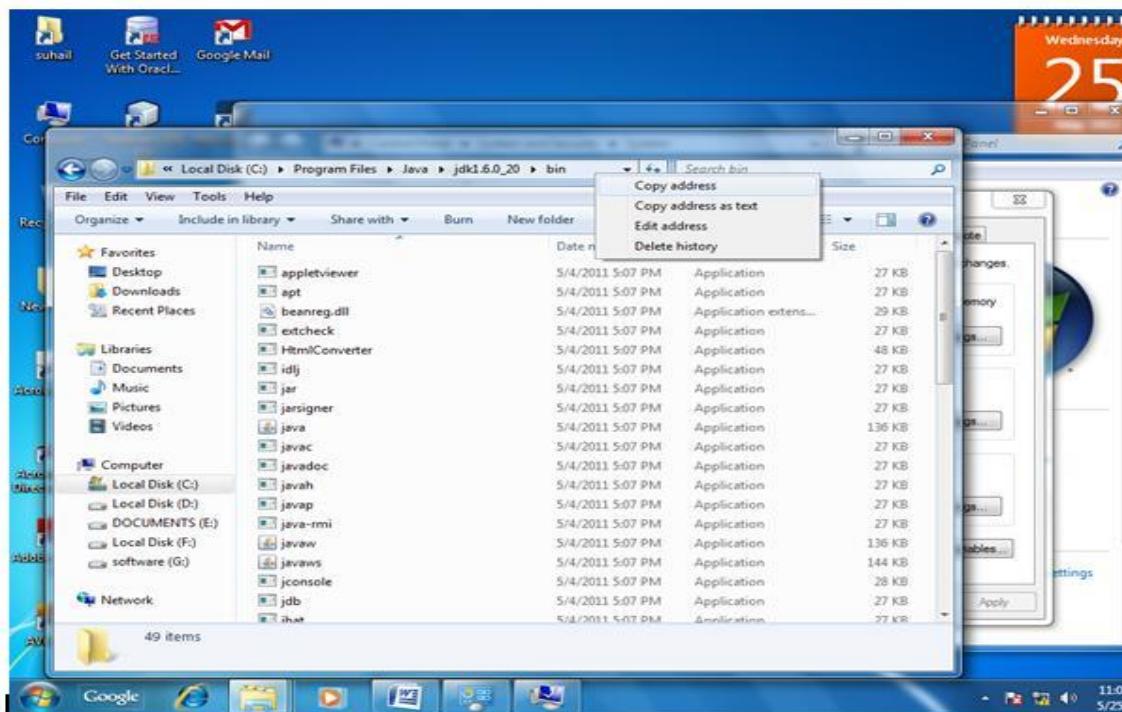


5) write path in variable name

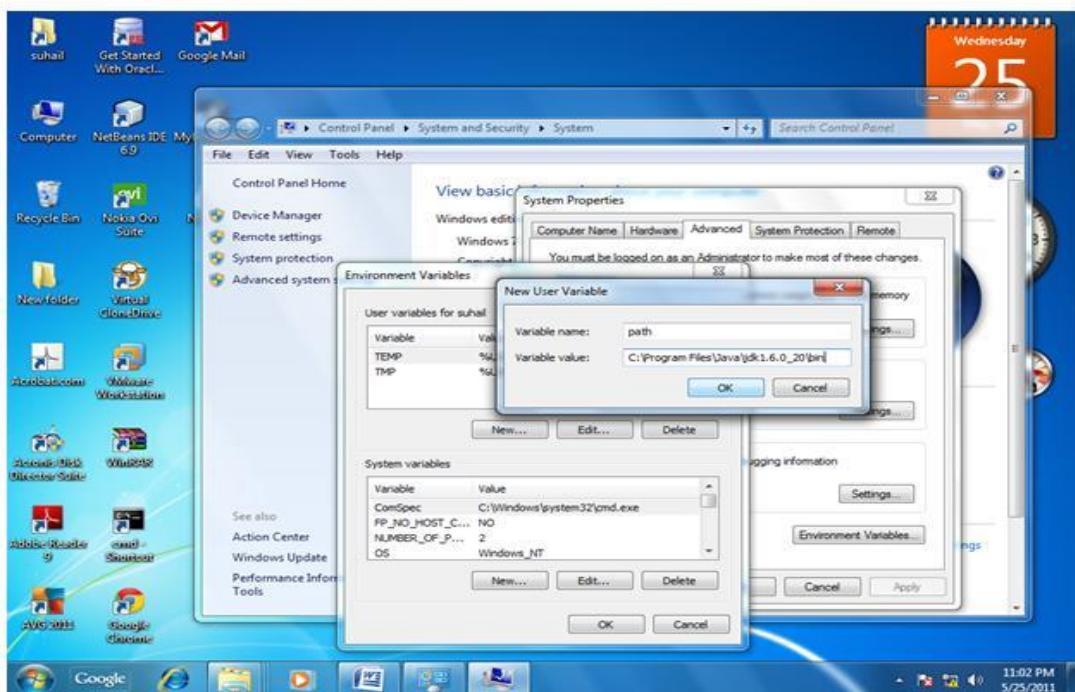


6) Copy the path of bin folder

Core Java

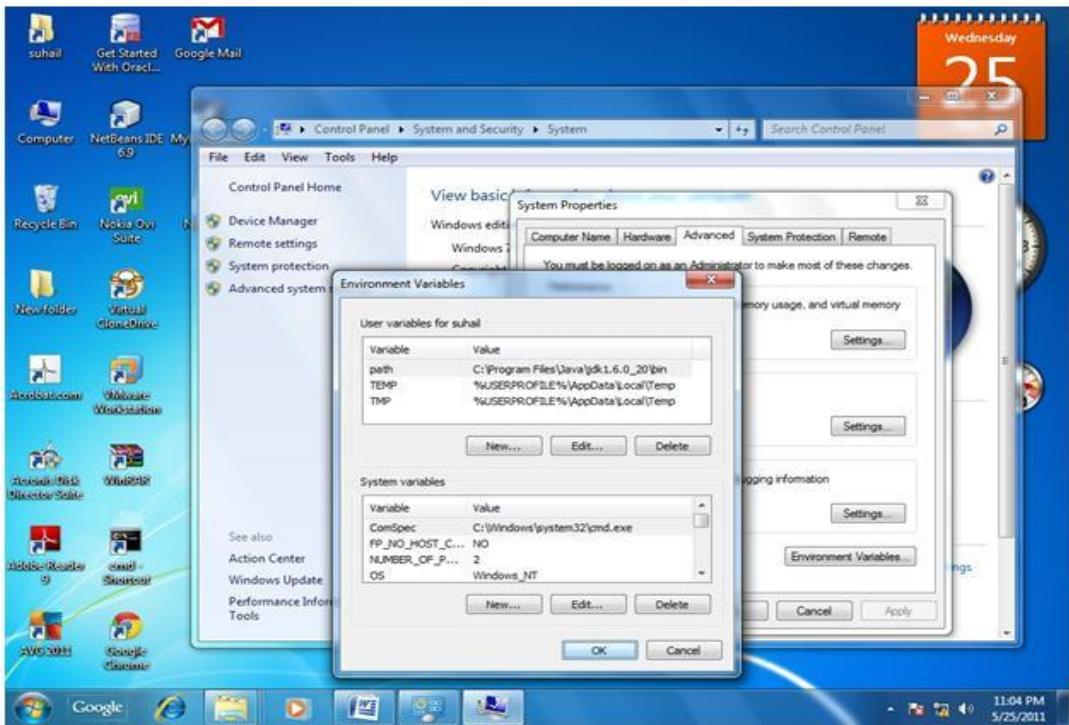


7) paste path of bin folder in variable value

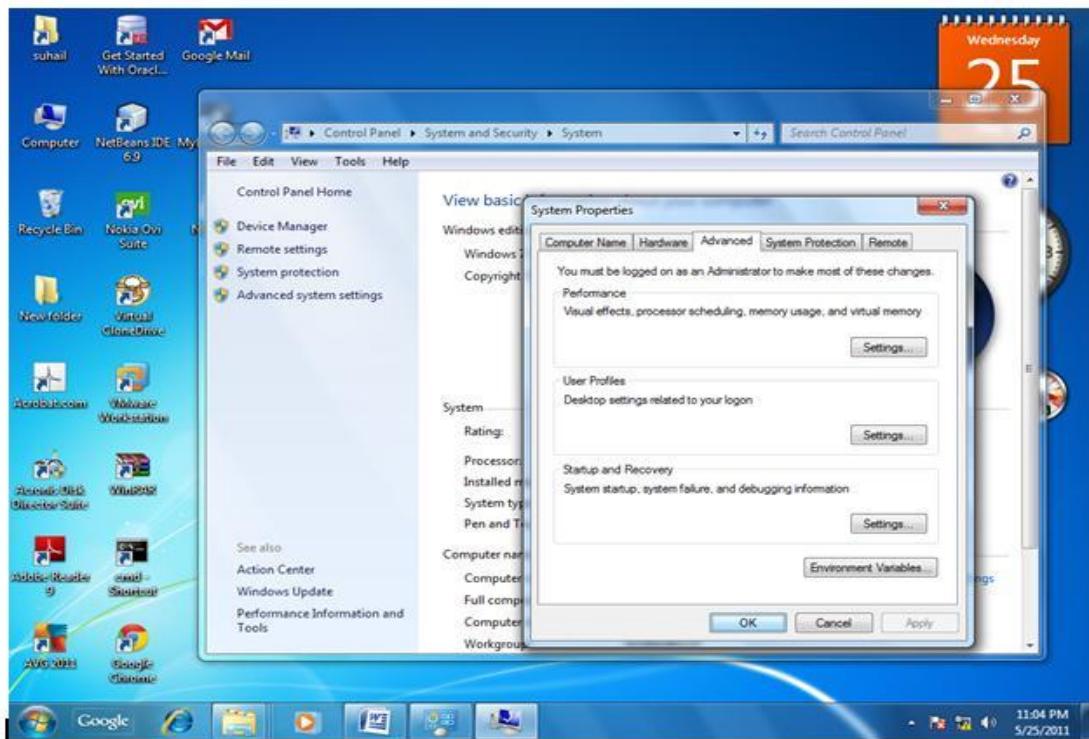


8) click on ok button

Core Java



9)click on ok button



Now your permanent path is set. You can now execute any program of java from any drive.

Setting Path in Linux OS

Core Java

Setting the path in Linux OS is same as setting the path in the Windows OS. But here we use export tool rather than set. Let's see how to set path in Linux OS:

```
export PATH=$PATH:/home/jdk1.6.01/bin/
```

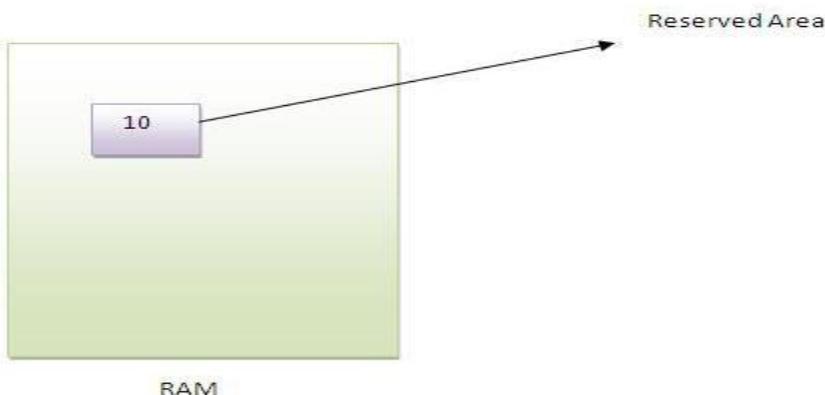
Here, we have installed the JDK in the home directory under Root (/home).

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.



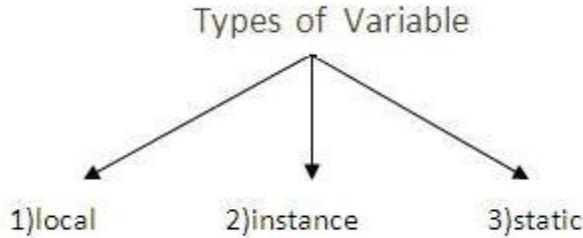
```
int data=50;//Here data is variable
```

Types of Variable

There are three types of variables in java

- local variable
- instance variable
- static variable

Core Java



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

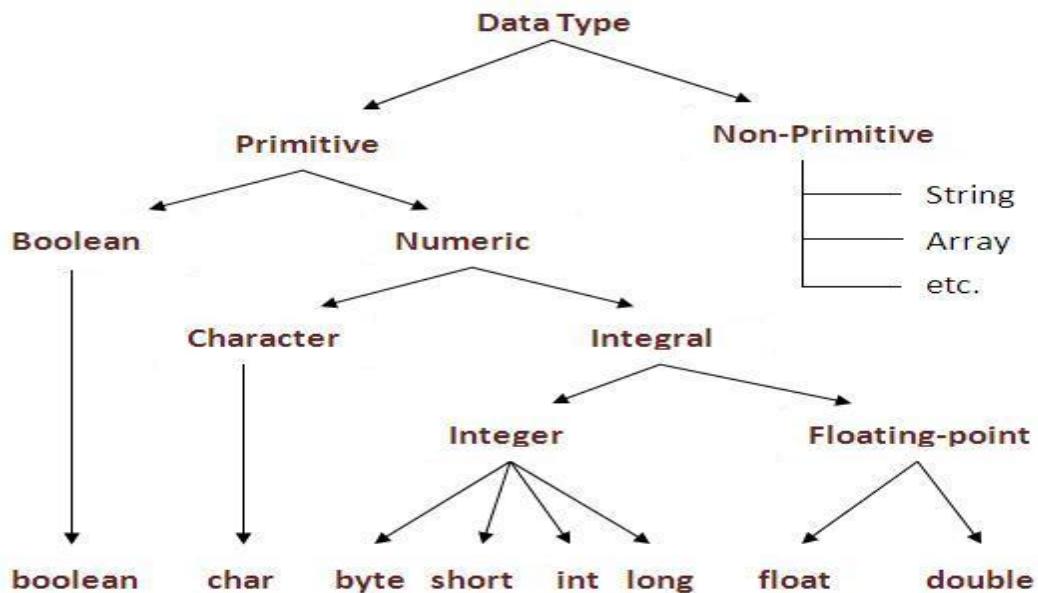
```
class A{  
  
    int data=50;//instance variable  
  
    static int m=100;//static variable  
  
    void method(){  
        int n=90;//local variable  
    }  
  
}//end of class
```

Core Java

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

Core Java

Why char uses 2 byte in java and what is \u0000 ?

because java uses unicode system rather than ASCII code system. \u0000 is the lowest range of unicode system. To get detail about Unicode see below.

Unicode System

Unicode is a universal international standard character encoding that is capable of representing most of the world's written languages.

Why java uses Unicode System?

Before Unicode, there were many language standards:

- **ASCII** (American Standard Code for Information Interchange) for the United States.
- **ISO 8859-1** for Western European Language.
- **KOI-8** for Russian.
- **GB18030 and BIG-5** for Chinese, and so on.

Precedence of Operators

Operators	Precedence
postfix	<i>expr++ expr--</i>
unary	<i>++expr --expr +expr -expr ~ !</i>
multiplicative	<i>* / %</i>
additive	<i>+ -</i>
shift	<i><< >> >>></i>
relational	<i>< > <= >= instanceof</i>

Core Java

equality	<code>== !=</code>
bitwise AND	<code>&</code>
bitwise exclusive OR	<code>^</code>
bitwise inclusive OR	<code> </code>
logical AND	<code>&&</code>
logical OR	<code> </code>
ternary	<code>? :</code>
assignment	<code>= += -= *= /= %= &= ^= = <<= >>= >>>=</code>

This caused two problems:

1. A particular code value corresponds to different letters in the various language standards.
2. The encodings for languages with large character sets have variable length. Some common characters are encoded as single bytes, others require two or more bytes.

To solve these problems, a new language standard was developed i.e. Unicode System.

In Unicode, a character holds 2 bytes, so Java also uses 2 bytes for characters.

lowest value: \u0000

highest value: \uFFFF

Operators in Java

Operator is a special 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.

Useful Programs:

There are some useful programs such as factorial number, prime number, Fibonacci series etc.

Core Java

It is better for the freshers to skip this topic and come to it after OOPs concepts.

1) Program of factorial number.

```
class Operation{  
  
    static int fact(int number){  
        int f=1;  
        for(int i=1;i<=number;i++){  
            f=f*i;  
        }  
        return f;  
    }  
  
    public static void main(String args[]){  
        int result=fact(5);  
        System.out.println("Factorial of 5="+result);  
    }  
}
```

2) Program of fibonacci series.

```
class Fabnoci{  
  
    public static void main(String...args)  
    {  
        int n=10,i,f0=1,f1=1,f2=0;  
        for(i=1;i<=n;i++)  
        {  
            f2=f0+f1;  
            f0=f1;  
            f1=f2;  
            f2=f0;  
            System.out.println(f2);  
        }  
    }  
}
```

Core Java

3) Program of armstrong number.

```
class ArmStrong{
    public static void main(String...args)
    {
        int n=153,c=0,a,d;
        d=n;
        while(n>0)
        {
            a=n%10;
            n=n/10;
            c=c+(a*a*a);
        }
        if(d==c)
            System.out.println("armstrong number");
        else
            System.out.println("it is not an armstrong number");

    }
}
```

4) Program of checking palindrome number.

```
class Palindrome
{
    public static void main( String...args)
    {
        int a=242;
        int n=a,b=a,rev=0;
        while(n>0)
        {
            a=n%10;
            rev=rev*10+a;
            n=n/10;
        }
        if(rev==b)
            System.out.println("it is Palindrome");
        else
            System.out.println("it is not palinedrome");
```

Core Java

```
}
```

5) Program of swapping two numbers without using third variable.

```
class SwapTwoNumbers{
    public static void main(String args[]){
        int a=40,b=5;
        a=a*b;
        b=a/b;
        a=a/b;

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

    }
}
```

6) Program of factorial number by recursion

```
class FactRecursion{

    static int fact(int n){
        if(n==1)
            return 1;

        return n*=fact(n-1);
    }

    public static void main(String args[]){
        int f=fact(5);
        System.out.println(f);
    }
}
```

Java OOPs Concepts

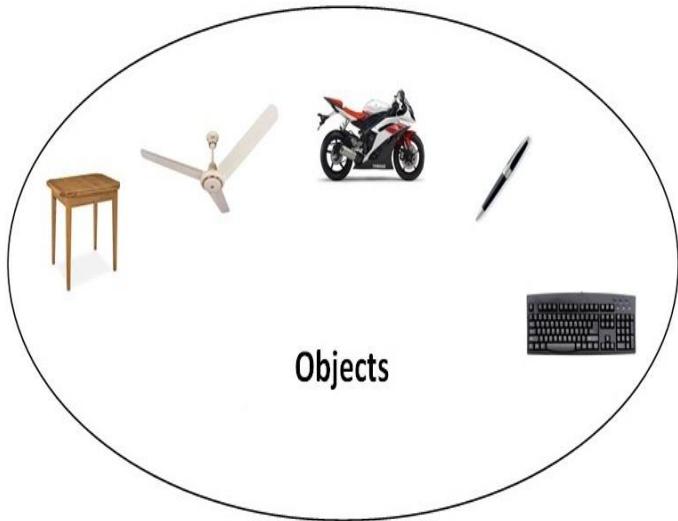
Java OOPs Concepts

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 world 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.

Core Java

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 converse 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.



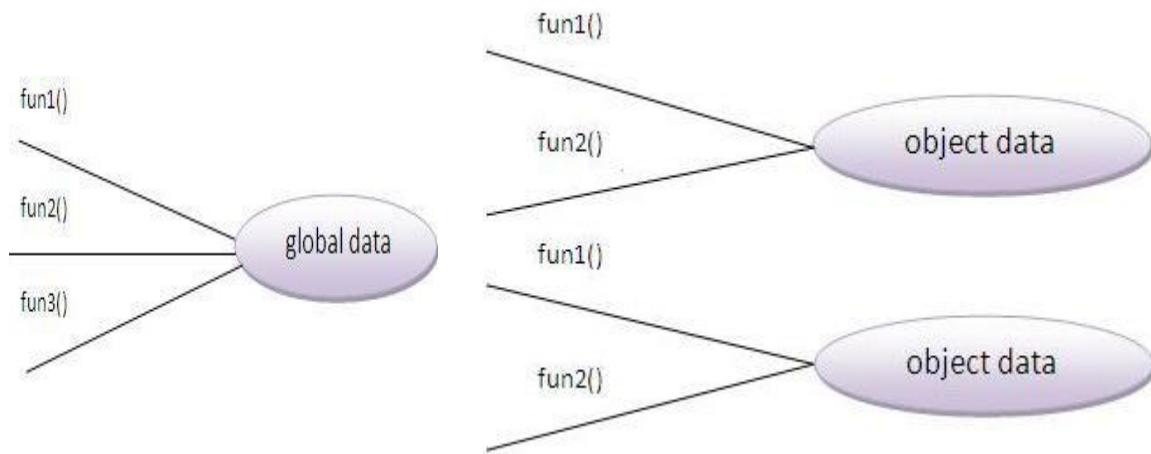
Encapsulation

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.

Advantage of OOPs over Procedure-oriented programming language

- 1)OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.
- 2)OOPs provides data hiding whereas in Procedure-oriented prgramming language a global data can be accessed from anywhere.
- 3)OOPs provides ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.



What is difference between object-oriented programming language and object-based programming language?

Object based programming language follows all the features of OOPs except Inheritance. JavaScript and VBScript are examples of object based programming languages.

- Advantage of OOPs
- Naming Convention
- Object and class
- Method overloading
- Constructor

Core Java

- static keyword
- this keyword with 6 usage
- Inheritance
- Aggregation
- Method Overriding
- Covariant Return Type
- super keyword
- Instance Initializer block
- final keyword
- Abstract class
- Interface
- Runtime Polymorphism
- Static and Dynamic Binding
- Downcasting with instanceof operator
- Package
- Access Modifiers
- Encapsulation
- Object Cloning

Java Naming convention

A **naming convention** is a rule to follow as you decide what to name your identifiers (e.g. class, package, variable, method, etc.), but it is not mandatory to follow that is why it is known as convention not rule.

Advantage of java naming convention

By using standard Java naming conventions they make their code easier to read for themselves and for other programmers. Readability of Java code is important because it means less time is spent trying to figure out what the code does.

Name	Convention
class name	should begin with uppercase letter and be a noun e.g.String, System, Thread etc.
Interface name	should begin with uppercase letter and be an adjective (wherever possible). e.g. Runnable, ActionListener etc.
method name	should begin with lowercase letter and be a verb. e.g. main(), print(), println(), actionPerformed() etc.

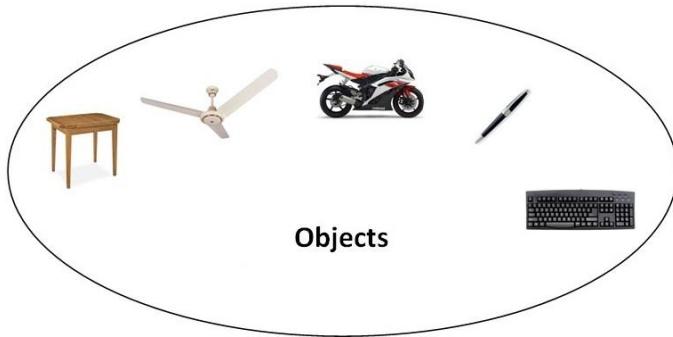
Core Java

variable name	should begin 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.

Object and Class in Java

In this page, we will learn about the objects and classes. In object-oriented programming, we design a program using objects and classes. Object is the physical entity whereas class is the logical entity. A class works as a template from which we create the objects.

Object



A runtime entity that has state and behaviour is known as an object. For example: chair, table, pen etc. It can be tangible or intangible (physical or logical).

An object has three characteristics:

- **state:**represents the data of an object.
- **behaviour:**represents the behaviour of an object.
- **identity:**Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user, but 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 behaviour.

Core Java

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

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

A class in java can contain:

- **data member**
- **method**
- **constructor**
- **block**

Syntax to declare a class:

```
1. class <class_name>{  
2.     data member;  
3.     method;  
4. }
```

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.

```
1. class Student{  
2.     int id;//data member (also instance variable)  
3.     String name;//data member(also instance variable)  
4.  
5.     public static void main(String args[]){  
6.         Student s1=new Student();//creating an object of Student  
7.         System.out.println(s1.id+" "+s1.name);  
8.  
9.     }  
10. }
```

Output:0 null

Instance variable

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.

Core Java

Method

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.

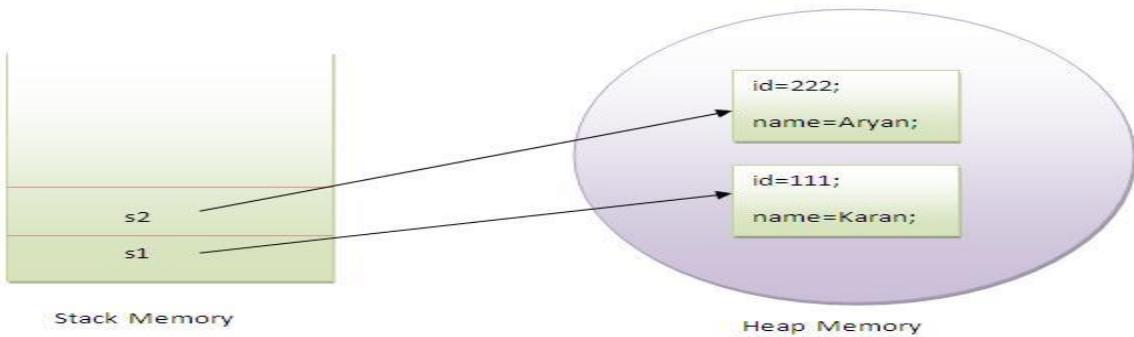
Example of Object and class that maintains the records of students

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.

```
1. class Student{  
2.     int rollno;  
3.     String name;  
4.  
5.     void insertRecord(int r, String n){ //method  
6.         rollno=r;  
7.         name=n;  
8.     }  
9.  
10.    void displayInformation(){System.out.println(rollno+" "+name);} //method  
11.  
12.    public static void main(String args[]){  
13.        Student s1=new Student();  
14.        Student s2=new Student();  
15.  
16.        s1.insertRecord(111,"Karan");  
17.        s2.insertRecord(222,"Aryan");  
18.  
19.        s1.displayInformation();  
20.        s2.displayInformation();  
21.  
22.    }  
23. }
```

Output:111 Karan
222 Aryan

Core Java



As you see in the above figure, object gets the memory in Heap area and reference variable refers to the object allocated in the Heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

Another Example of Object and Class

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

```
1. class Rectangle{  
2.     int length;  
3.     int width;  
4.  
5.     void insert(int l,int w){  
6.         length=l;  
7.         width=w;  
8.     }  
9.  
10.    void calculateArea(){System.out.println(length*width);}  
11.  
12.    public static void main(String args[]){  
13.        Rectangle r1=new Rectangle();  
14.        Rectangle r2=new Rectangle();  
15.  
16.        r1.insert(11,5);  
17.        r2.insert(3,15);  
18.  
19.        r1.calculateArea();  
20.        r2.calculateArea();  
21.    }  
22.}
```

Output:55

45

Core Java

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.

Anonymous object

Anonymous simply means nameless. An object that have no reference is known as anonymous object.

If you have to use an object only once, anonymous object is a good approach.

```
1. class Calculation{  
2.  
3.     void fact(int n){  
4.         int fact=1;  
5.         for(int i=1;i<=n;i++){  
6.             fact=fact*i;  
7.         }  
8.         System.out.println("factorial is "+fact);  
9.     }  
10.  
11.    public static void main(String args[]){  
12.        new Calculation().fact(5);//calling method with anonymous object  
13.    }  
14. }
```

Output:Factorial is 120

Creating multiple objects by one type only

We can create multiple objects by one type only as we do in case of primitives.

```
1. Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
```

Let's see the example:

```
1. class Rectangle{  
2.     int length;  
3.     int width;  
4.  
5.     void insert(int l,int w){  
6.         length=l;  
7.         width=w;  
8.     }  
9. }
```

Core Java

```
10. void calculateArea(){System.out.println(length*width);}
11.
12. public static void main(String args[]){
13. Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
14.
15. r1.insert(11,5);
16. r2.insert(3,15);
17.
18. r1.calculateArea();
19. r2.calculateArea();
20. }
21. }
```

Output:55

45

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

Core Java

In java, Method 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.

```
1. class Calculation{  
2.     void sum(int a,int b){System.out.println(a+b);}  
3.     void sum(int a,int b,int c){System.out.println(a+b+c);}  
4.  
5.     public static void main(String args[]){  
6.         Calculation obj=new Calculation();  
7.         obj.sum(10,10,10);  
8.         obj.sum(20,20);  
9.  
10.    }  
11. }
```

Output:30

40

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 Calculation{  
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.  
5.     public static void main(String args[]){  
6.         Calculation obj=new Calculation();  
7.         obj.sum(10.5,10.5);  
8.         obj.sum(20,20);  
9.  
10.    }  
11. }
```

Output:21.0

40

Que) Why Method Overloading is not possible by changing the return type of method?

In java, method overloading is not possible by changing the return type of the method

Core Java

because there may occur ambiguity. Let's see how ambiguity may occur:

because there was problem:

```
1. class Calculation{
2.     int sum(int a,int b){System.out.println(a+b);}
3.     double sum(int a,int b){System.out.println(a+b);}
4.
5.     public static void main(String args[]){
6.         Calculation obj=new Calculation();
7.         int result=obj.sum(20,20); //Compile Time Error
8.
9.     }
10. }
```

int result=obj.sum(20,20); //Here how can java determine which sum() method should be called

Can we overload main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. Let's see the simple example:

```
1. class Simple{
2.     public static void main(int a){
3.         System.out.println(a);
4.     }
5.
6.     public static void main(String args[]){
7.         System.out.println("main() method invoked");
8.         main(10);
9.     }
10. }
```

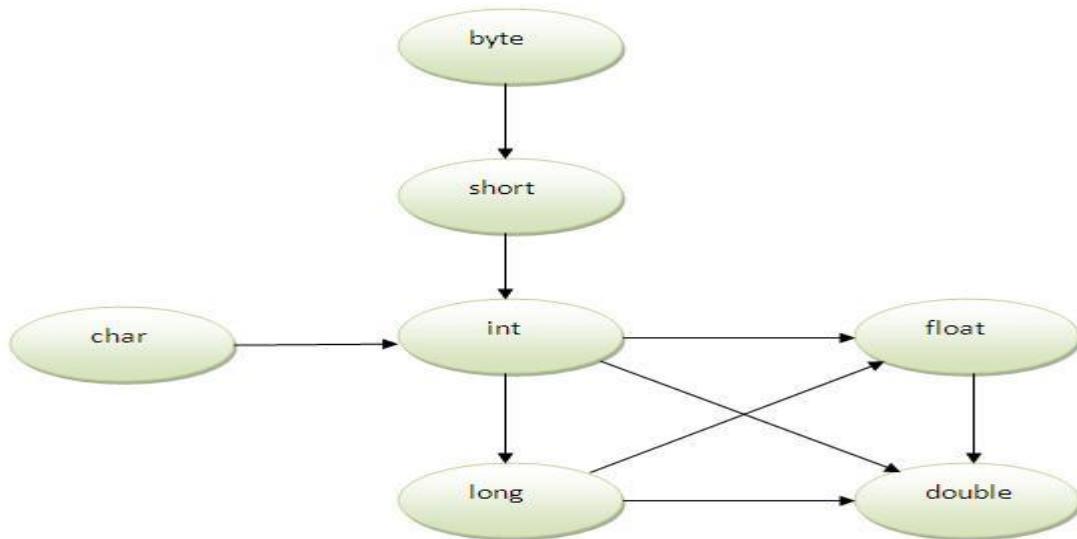
Output:main() method invoked

10

Method Overloading and TypePromotion

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:

Core Java



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int,long,float or double. The char datatype can be promoted to int,long,float or double and so on.

Example of Method Overloading with TypePromotion

```
1. class Calculation{  
2.     void sum(int a,long b){System.out.println(a+b);}  
3.     void sum(int a,int b,int c){System.out.println(a+b+c);}  
4.  
5.     public static void main(String args[]){  
6.         Calculation obj=new Calculation();  
7.         obj.sum(20,20); //now second int literal will be promoted to long  
8.         obj.sum(20,20,20);  
9.  
10.    }  
11. }
```

Output:40

60

Example of Method Overloading with TypePromotion if matching found

If there are matching type arguments in the method, type promotion is not performed.

```
1. class Calculation{  
2.     void sum(int a,int b){System.out.println("int arg method invoked");}  
3.     void sum(long a,long b){System.out.println("long arg method invoked");}  
4.  
5.     public static void main(String args[]){  
6.         Calculation obj=new Calculation();
```

Core Java

```
7. obj.sum(20,20);//now int arg sum() method gets invoked
8. }
9. }
```

Output:int arg method invoked

Example of Method Overloading with TypePromotion in case ambiguity

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

```
1. class Calculation{
2.     void sum(int a,long b){System.out.println("a method invoked");}
3.     void sum(long a,int b){System.out.println("b method invoked");}
4.
5.     public static void main(String args[]){
6.         Calculation obj=new Calculation();
7.         obj.sum(20,20);//now ambiguity
8.     }
9. }
```

Output:Compile Time Error

Constructor in Java

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

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 constructor

There are basically two rules defined for the constructor.

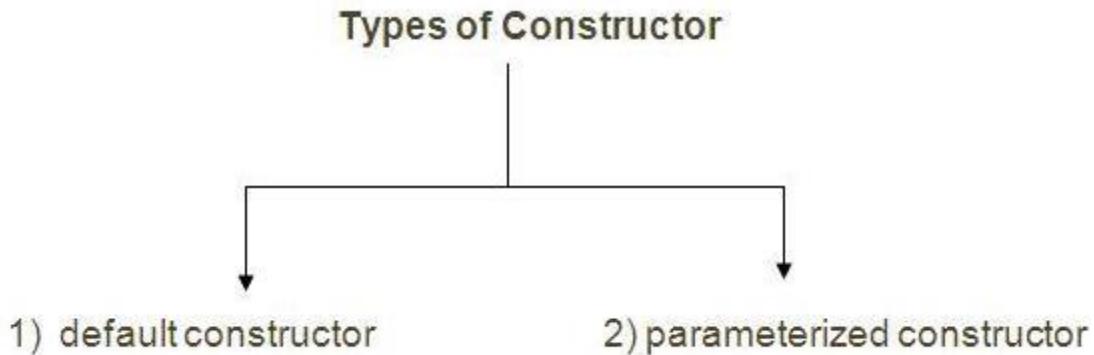
1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

Types of constructors

There are two types of constructors:

1. default constructor (no-arg constructor)
2. parameterized constructor

Core Java



1) Default Constructor

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

Syntax of default constructor:

1. `<class_name>(){}`

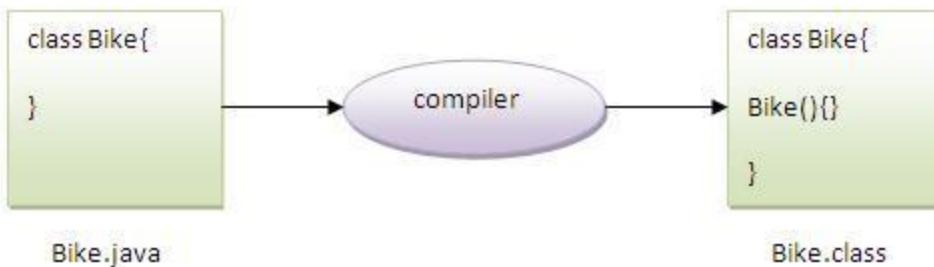
Example of default constructor

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

```
1. class Bike{  
2.  
3.     Bike(){System.out.println("Bike is created");}  
4.  
5.     public static void main(String args[]){  
6.         Bike b=new Bike();  
7.     }  
8. }
```

Output: Bike is created

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



Core Java

Que)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 Student{  
2.     int id;  
3.     String name;  
4.  
5.     void display(){System.out.println(id+" "+name);}  
6.  
7.     public static void main(String args[]){  
8.         Student s1=new Student();  
9.         Student s2=new Student();  
10.        s1.display();  
11.        s2.display();  
12.    }  
13. }  
1. Output:0 null  
2.      0 null
```

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.

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.

```
1. class Student{  
2.     int id;  
3.     String name;  
4.  
5.     Student(int i,String n){  
6.         id = i;  
7.         name = n;  
8.     }  
9.     void display(){System.out.println(id+" "+name);} 
```

Core Java

```
10.  
11. public static void main(String args[]){  
12.     Student s1 = new Student(111,"Karan");  
13.     Student s2 = new Student(222,"Aryan");  
14.     s1.display();  
15.     s2.display();  
16. }  
17. }
```

Output:111 Karan
222 Aryan

Constructor Overloading

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

Example of Constructor Overloading

```
1. class Student{  
2.     int id;  
3.     String name;  
4.     int age;  
5.     Student(int i,String n){  
6.         id = i;  
7.         name = n;  
8.     }  
9.     Student(int i,String n,int a){  
10.        id = i;  
11.        name = n;  
12.        age=a;  
13.    }  
14.    void display(){System.out.println(id+" "+name+" "+age);}  
15.  
16.    public static void main(String args[]){  
17.        Student s1 = new Student(111,"Karan");  
18.        Student s2 = new Student(222,"Aryan",25);  
19.        s1.display();  
20.        s2.display();  
21.    }  
22. }
```

Output:111 Karan 0
222 Aryan 25

What is the difference between constructor and method ?

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

Core Java

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

Copying the values of one object to another like copy constructor in C++

There are many ways to copy the values of one object into another. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

In this example, we are going to copy the values of one object into another using constructor.

```

1. class Student{
2.     int id;
3.     String name;
4.     Student(int i,String n){
5.         id = i;
6.         name = n;
7.     }
8.
9.     Student(Student s){
10.        id = s.id;
11.        name = s.name;
12.    }
13.    void display(){System.out.println(id+" "+name);}
14.
15.    public static void main(String args[]){
16.        Student s1 = new Student(111,"Karan");

```

Core Java

```
17. Student s2 = new Student(s1);
18. s1.display();
19. s2.display();
20. }
21. }
```

Output:111 Karan

111 Karan

Copying the values of one object to another without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```
1. class Student{
2.     int id;
3.     String name;
4.     Student(int i,String n){
5.         id = i;
6.         name = n;
7.     }
8.     Student(){}
9.     void display(){System.out.println(id+" "+name);}
10.
11.    public static void main(String args[]){
12.        Student s1 = new Student(111,"Karan");
13.        Student s2 = new Student();
14.        s2.id=s1.id;
15.        s2.name=s1.name;
16.        s1.display();
17.        s2.display();
18.    }
19. }
```

Output:111 Karan

111 Karan

Que) Does constructor return any value?

Ans: yes, that is current class instance (You cannot use return type yet it returns a value).

Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, calling method etc. You can perform any operation in the constructor as you perform in the method.

static keyword

Core Java

The **static keyword** is used in java mainly for memory management. We may apply 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) 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.

static property is shared to all objects.

Example of static variable

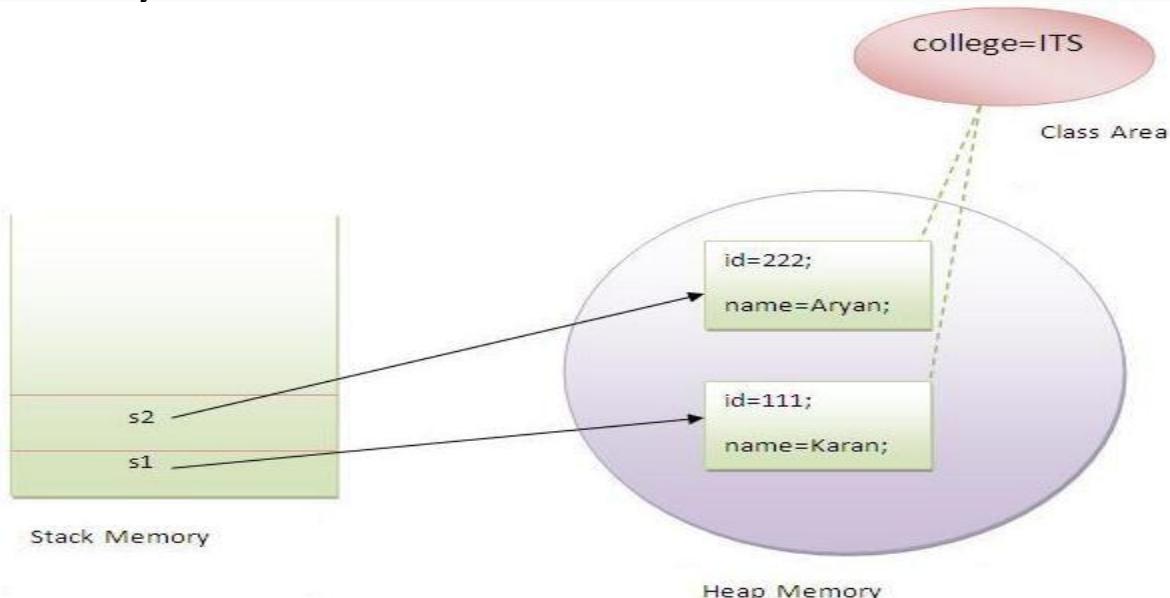
```
1. //Program of static variable  
2.  
3. class Student{  
4.     int rollno;  
5.     String name;  
6.     static String college ="ITS";  
7.  
8.     Student(int r, String n){
```

Core Java

```
9.     rollno = r;
10.    name = n;
11. }
12. void display (){System.out.println(rollno+" "+name+" "+college);}
13.
14. public static void main(String args[]){
15. Student s1 = new Student (111,"Karan");
16. Student s2 = new Student (222,"Aryan");
17.
18. s1.display();
19. s2.display();
20. }
21. }
```

Output:111 Karan ITS

222 Aryan ITS



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.
4. Counter(){
5. count++;
6. System.out.println(count);
7. }
8.
9. public static void main(String args[]){
10. }
```

Core Java

```
11. Counter c1=new Counter();
12. Counter c2=new Counter();
13. Counter c3=new Counter();
14.
15. }}
```

Output:1

```
1
1
```

Program of counter by static variable

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```
1. class Counter{
2. static int count=0;//will get memory only once and retain its value
3.
4. Counter(){
5. count++;
6. System.out.println(count);
7. }
8.
9. public static void main(String args[]){
10.
11. Counter c1=new Counter();
12. Counter c2=new Counter();
13. Counter c3=new Counter();
14.
15. }}
```

Output:1

```
2
3
```

2) 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.
3. class Student{
4.     int rollno;
5.     String name;
6.     static String college = "ITS";
7.
```

Core Java

```
8. static void change(){
9.     college = "BBDIT";
10.    }
11.
12. Student(int r, String n){
13.     rollno = r;
14.     name = n;
15.    }
16.
17. void display (){System.out.println(rollno+" "+name+" "+college);}
18.
19. public static void main(String args[]){
20.     Student.change();
21.
22.     Student s1 = new Student (111,"Karan");
23.     Student s2 = new Student (222,"Aryan");
24.     Student s3 = new Student (333,"Sonoo");
25.
26.     s1.display();
27.     s2.display();
28.     s3.display();
29.    }
30. }
```

Output:111 Karan BBDIT

222 Aryan BBDIT

333 Sonoo BBDIT

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. }
```

Output:125

Restrictions for static method

There are two main restrictions for the static method. They are:

Core Java

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

```
1. class A{  
2.     int a=40;//non static  
3.  
4.     public static void main(String args[]){  
5.         System.out.println(a);  
6.     }  
7. }
```

Output:Compile Time Error

Que)why main method is static?

Ans) because object is not required to call static method if it were non-static method, jvm creates object first then call main() method that will lead the problem of extra memory allocation.

3)static block

- Is used to initialize the static data member.
- It is executed before main method at the time of classloading.

Example of static block

```
1. class A{  
2.  
3.     static{System.out.println("static block is invoked");}  
4.  
5.     public static void main(String args[]){  
6.         System.out.println("Hello main");  
7.     }  
8. }
```

Output:static block is invoked

 Hello main

Que)Can we execute a program without main() method?

Ans)Yes, one of the way is static block but in previous version of JDK not in JDK 1.7.

```
1. class A{  
2.     static{  
3.         System.out.println("static block is invoked");  
4.         System.exit(0);  
5.     }  
6. }
```

Output:static block is invoked (if not JDK7)

this keyword

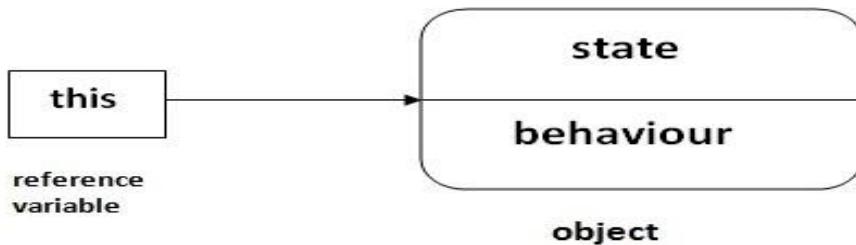
There can be a lot of usage of **this keyword**. In java, this is a **reference variable** that refers to the current object.

Usage of this keyword

Here is given the 6 usage of 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 student{  
2.     int id;  
3.     String name;  
4.  
5.     student(int id,String name){  
6.         id = id;  
7.         name = name;  
8.     }  
9.     void display(){System.out.println(id+" "+name);}  
10.
```

Core Java

```
11. public static void main(String args[]){
12.     student s1 = new student(111,"Karan");
13.     student s2 = new student(321,"Aryan");
14.     s1.display();
15.     s2.display();
16. }
17. }
```

Output:0 null

0 null

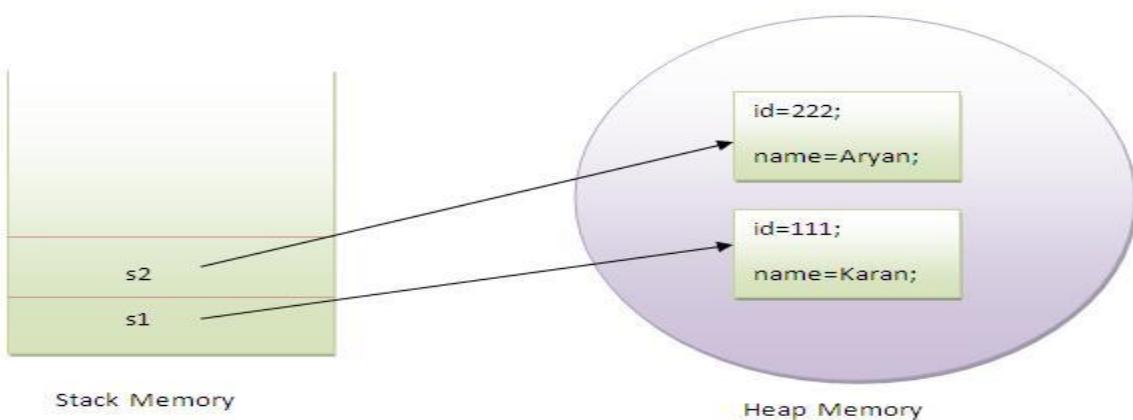
In the above example, parameter (formal arguments) and instance variables are same that is why we are using this keyword to distinguish between local variable and instance variable.

Solution of the above problem by this keyword

```
1. //example of this keyword
2. class Student{
3.     int id;
4.     String name;
5.
6.     student(int id,String name){
7.         this.id = id;
8.         this.name = name;
9.     }
10.    void display(){System.out.println(id+" "+name);}
11.    public static void main(String args[]){
12.        Student s1 = new Student(111,"Karan");
13.        Student s2 = new Student(222,"Aryan");
14.        s1.display();
15.        s2.display();
16.    }
17. }
```

Output:111 Karan

222 Aryan



Core Java

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 Student{  
2.     int id;  
3.     String name;  
4.  
5.     student(int i,String n){  
6.         id = i;  
7.         name = n;  
8.     }  
9.     void display(){System.out.println(id+" "+name);}  
10.    public static void main(String args[]){  
11.        Student e1 = new Student(111,"karan");  
12.        Student e2 = new Student(222,"Aryan");  
13.        e1.display();  
14.        e2.display();  
15.    }  
16. }
```

Output:111 Karan

222 Aryan

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.  
3. class Student{  
4.     int id;  
5.     String name;  
6.     Student (){System.out.println("default constructor is invoked");}  
7.  
8.     Student(int id,String name){  
9.         this ();//it is used to invoked current class constructor.  
10.        this.id = id;  
11.        this.name = name;  
12.    }  
13.    void display(){System.out.println(id+" "+name);}  
14.  
15.    public static void main(String args[]){  
16.        Student e1 = new Student(111,"karan");  
17.        Student e2 = new Student(222,"Aryan");  
18.        e1.display();  
19.        e2.display();
```

Core Java

```
20. }
21. }
```

Output:

```
default constructor is invoked
default constructor is invoked
111 Karan
222 Aryan
```

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 Student{
2.     int id;
3.     String name;
4.     String city;
5.
6.     Student(int id,String name){
7.         this.id = id;
8.         this.name = name;
9.     }
10.    Student(int id,String name,String city){
11.        this(id,name); //now no need to initialize id and name
12.        this.city=city;
13.    }
14.    void display(){System.out.println(id+" "+name+" "+city);}
15.
16.    public static void main(String args[]){
17.        Student e1 = new Student(111,"karan");
18.        Student e2 = new Student(222,"Aryan","delhi");
19.        e1.display();
20.        e2.display();
21.    }
22. }
```

Output:

```
111 Karan null
222 Aryan delhi
```

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

```
1. class Student{
2.     int id;
3.     String name;
4.     Student (){System.out.println("default constructor is invoked");}
5.
6.     Student(int id,String name){
7.         id = id;
8.         name = name;
9.         this ();//must be the first statement
```

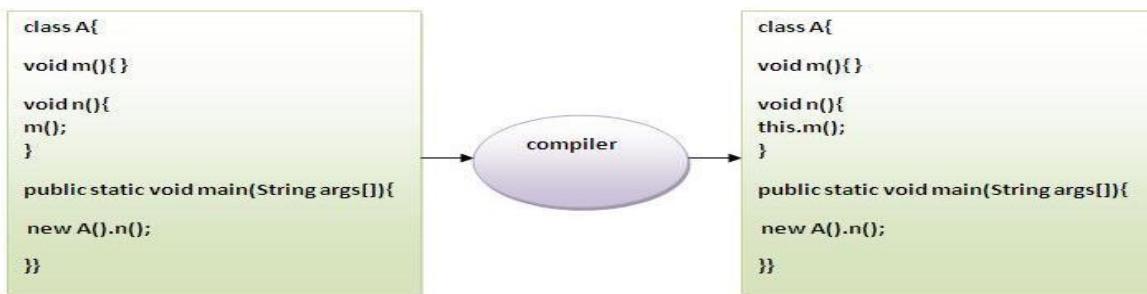
Core Java

```
10. }
11. void display(){System.out.println(id+" "+name);}
12.
13. public static void main(String args[]){
14.     Student e1 = new Student(111,"karan");
15.     Student e2 = new Student(222,"Aryan");
16.     e1.display();
17.     e2.display();
18. }
19. }
```

Output:Compile Time Error

3)The this keyword can be used to invoke current class method (implicitly).

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example



```
1. class S{
2.     void m(){
3.         System.out.println("method is invoked");
4.     }
5.     void n(){
6.         this.m(); //no need because compiler does it for you.
7.     }
8.     void p(){
9.         n(); //compiler will add this to invoke n() method as this.n()
10.    }
11.    public static void main(String args[]){
12.        S s1 = new S();
13.        s1.p();
14.    }
15. }
```

Output:method is invoked

4) this keyword can be passed as an argument in the method.

The this keyword can also be passed as an argument in the method. It is mainly used in the

Core Java

event handling. Let's see the example:

```
1. class S{  
2.     void m(S obj){  
3.         System.out.println("method is invoked");  
4.     }  
5.     void p(){  
6.         m(this);  
7.     }  
8.  
9.     public static void main(String args[]){  
10.        S s1 = new S();  
11.        s1.p();  
12.    }  
13. }
```

Output:method is invoked

Application of this that can be passed as an argument:

In event handling (or) in a situation where we have to provide reference of a class to another one.

5) The this keyword can be passed as argument in the constructor call.

We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

```
1. class B{  
2.     A obj;  
3.     B(A obj){  
4.         this.obj=obj;  
5.     }  
6.     void display(){  
7.         System.out.println(obj.data);//using data member of A class  
8.     }  
9. }  
10.  
11. class A{  
12.     int data=10;  
13.     A(){  
14.         B b=new B(this);  
15.         b.display();  
16.     }  
17.     public static void main(String args[]){  
18.         A a=new A();  
19.     }  
20. }
```

Output:10

Core Java

6) The this keyword can be used to return current class instance.

We can return the this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

Syntax of this that can be returned as a statement

```
1. return_type method_name(){  
2.     return this;  
3. }
```

Example of this keyword that you return as a statement from the method

```
1. class A{  
2.     A getA(){  
3.         return this;  
4.     }  
5.     void msg(){System.out.println("Hello java");}  
6. }  
7.  
8. class Test{  
9.     public static void main(String args[]){  
10.        new A().getA().msg();  
11.    }  
12. }
```

Output:Hello java

Proving this keyword

Let's prove that this keyword refers to the current class instance variable. In this program, we are printing the reference variable and this, output of both variables are same.

```
1. class A{  
2.     void m(){  
3.         System.out.println(this);//prints same reference ID  
4.     }  
5.  
6.     public static void main(String args[]){  
7.         A obj=new A();  
8.         System.out.println(obj);//prints the reference ID  
9.  
10.        obj.m();  
11.    }  
12. }
```

Output:A@13d9c02
A@13d9c02

Inheritance in Java

Core Java

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

The idea behind inheritance is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you reuse (or inherit) methods and fields, and you add new methods and fields to adapt your new class to new situations.

Inheritance represents the **IS-A relationship**.

Why use Inheritance?

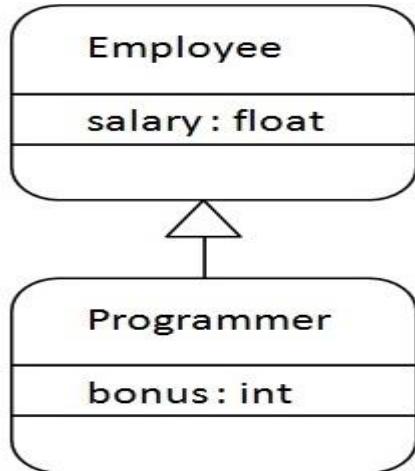
- For Method Overriding (So Runtime Polymorphism).
- For Code Reusability.

Syntax of Inheritance

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

The keyword `extends` 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 superclass. 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.  
5. class Programmer extends Employee{
```

Core Java

```
6. int bonus=10000;  
7.  
8. public static void main(String args[]){  
9.     Programmer p=new Programmer();  
10.    System.out.println("Programmer salary is:"+p.salary);  
11.    System.out.println("Bonus of Programmer is:"+p.bonus);  
12. }  
13. }
```

Output:Programmer salary is:40000.0

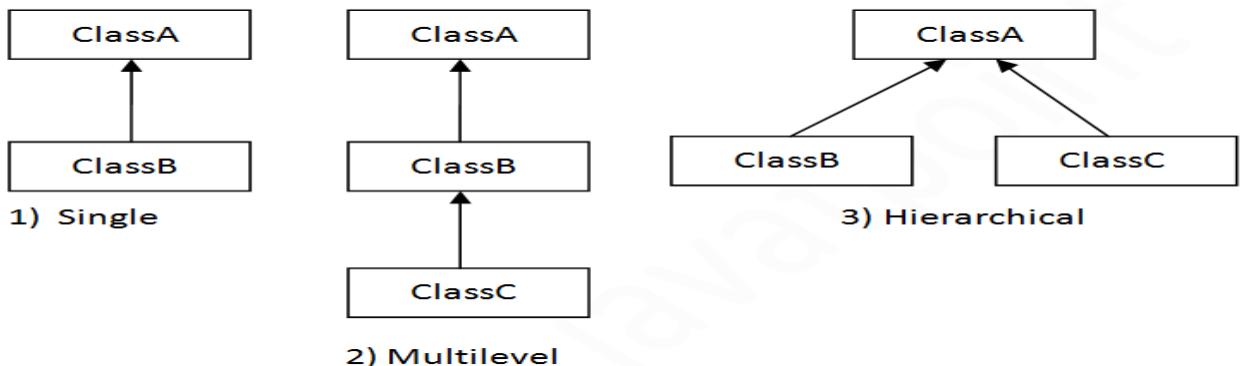
Bonus of programmer is:10000

In the above example,Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

Types of Inheritance

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

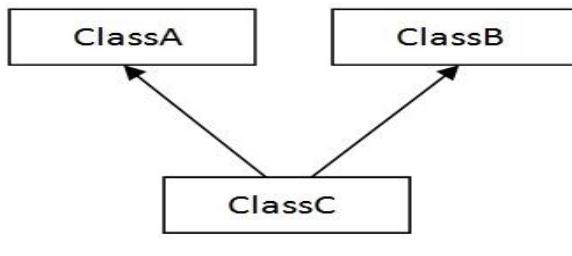
Multiple and Hybrid is supported through interface only. We will learn about interfaces later.



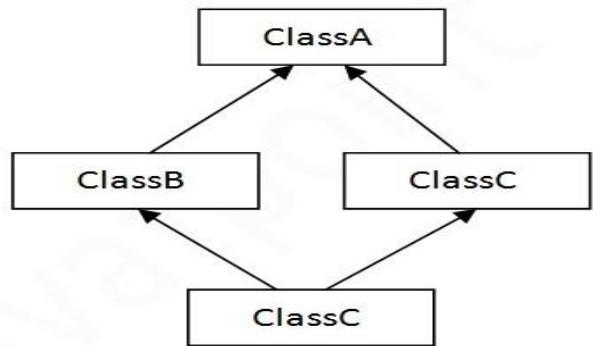
Multiple inheritance is not supported in java in case of class.

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

Core Java



4) Multiple



5) Hybrid

Que) Why multiple inheritance is not supported in java?

- To reduce the complexity and simplify the language, multiple inheritance is not supported in java. For example:

```
1. class A{  
2.     void msg(){System.out.println("Hello");}  
3. }  
4.  
5. class B{  
6.     void msg(){System.out.println("Welcome");}  
7. }  
8.  
9. class C extends A,B{//suppose if it were  
10.  
11. Public Static void main(String args[]){  
12.     C obj=new C();  
13.     obj.msg();//Now which msg() method would be invoked?  
14. }  
15. }
```

Aggregation in Java

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

```
1. class Employee{  
2.     int id;  
3.     String name;  
4.     Address address;//Address is a class  
5. ...
```

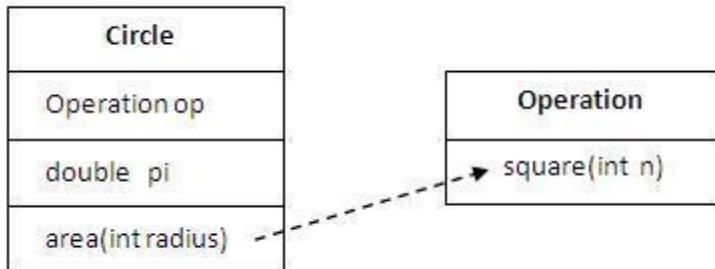
6. }

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

Why use Aggregation?

- For Code Reusability.

Simple Example of Aggregation



In this example, we have created the reference of Operation class in the Circle class.

```
1. class Operation{
2.     int square(int n){
3.         return n*n;
4.     }
5. }
6.
7. class Circle{
8.     Operation op;//aggregation
9.     double pi=3.14;
10.
11.    double area(int radius){
12.        op=new Operation();
13.        int rsquare=op.square(radius);//code reusability (i.e. delegates the method call).
14.        return pi*rsquare;
15.    }
16.
17.
18.
19. public static void main(String args[]){
20.     Circle c=new Circle();
21.     double result=c.area(5);
22.     System.out.println(result);
23. }
24. }
```

Output:78.5

Core Java

When use Aggregation?

- Code reuse is also best achieved by aggregation when there is no is-a relationship.
- Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

Understanding meaningful example of Aggregation

In this example, Employee has an object of Address, address object contains its own informations such as city, state, country etc. In such case relationship is Employee HAS-A address.

Address.java

```
1. public class Address {  
2.     String city,state,country;  
3.  
4.     public Address(String city, String state, String country) {  
5.         this.city = city;  
6.         this.state = state;  
7.         this.country = country;  
8.     }  
9.  
10. }
```

Emp.java

```
1. public class Emp {  
2.     int id;  
3.     String name;  
4.     Address address;  
5.  
6.     public Emp(int id, String name,Address address) {  
7.         this.id = id;  
8.         this.name = name;  
9.         this.address=address;  
10.    }  
11.  
12.    void display(){  
13.        System.out.println(id+" "+name);  
14.        System.out.println(address.city+" "+address.state+" "+address.country);  
15.    }  
16.  
17.    public static void main(String[] args) {  
18.        Address address1=new Address("gzb","UP","india");  
19.        Address address2=new Address("gno","UP","india");  
20.  
21.        Emp e=new Emp(111,"varun",address1);  
22.        Emp e2=new Emp(112,"arun",address2);  
23.  
24.        e.display();
```

Core Java

```
25. e2.display();
26.
27. }
28. }
```

Output:111 varun
gzb UP india
112 arun
gno UP india

Method Overriding in Java

Having the same method in the subclass as declared in the parent class is known as **method overriding**.

In other words, If subclass provides the specific implementation of the method i.e. already provided by its parent class, it is known as Method Overriding.

Advantage of 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 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 inheritance (IS-A) relationship.

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

Output:Vehicle is running

Core Java

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

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 Bike extends Vehicle{  
5.     void run(){System.out.println("Bike is running safely");}  
6. }  
7. public static void main(String args[]){  
8.     Bike obj = new Bike();  
9.     obj.run();  
10. }
```

Output:Bike is running safely

Can we override static method?

No, static method cannot be overridden. It can be proved by runtime polymorphism so we will learn it later.

Why we cannot override static method?

because static method is bound with class whereas instance method is bound with object.

Static belongs to class area and instance belongs to heap area.

What is the difference between method Overloading and Method Overriding?

There are three basic differences between the method overloading and method overriding.

They are as follows:

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 a class.	Method overriding occurs in two classes that have IS-A relationship.

Core Java

3) In case of method overloading parameter must be different.

In case of method overriding parameter must be same.

More topics on Method Overriding (Not For Fresher)

Method Overriding with Access Modifier

Let's see the concept of method overriding with access modifier.

Access Modifiers

There are two types of modifiers in java: **access modifier** and **non-access modifier**. The access modifiers specifies accessibility (scope) of a datamember, method, constructor or class.

There are 4 types of 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

The private access modifier is accessible only within class.

Simple example of private access modifer

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.  
6. public class Simple{  
7.     public static void main(String args[]){  
8.         A obj=new A();
```

Core Java

```
9. System.out.println(obj.data);//Compile Time Error  
10. obj.msg();//Compile Time Error  
11. }  
12. }
```

Role of Private Constructor:

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

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

2) default

If you don't use any modifier, it is treated as **default** by default. 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.  
3. package pack;  
4. class A{  
5. void msg(){System.out.println("Hello");}  
6. }  
1. //save by B.java  
2.  
3. package mypack;  
4. import pack.*;  
5.  
6. class B{  
7. public static void main(String args[]){  
8. A obj = new A();//Compile Time Error  
9. obj.msg();//Compile Time Error
```

Core Java

```
10. }
11. }
```

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

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

Output:Hello

4) public

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

Example of public access modifier

```
1. //save by A.java
```

Core Java

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

Output:Hello

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

Core Java

Applying access modifier with method overriding

If you are overriding any method, overriden 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.  
5. public class Simple extends A{  
6.     void msg(){System.out.println("Hello java");}//C.T.Error  
7.     public static void main(String args[]){  
8.         Simple obj=new Simple();  
9.         obj.msg();  
10.    }  
11. }
```

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

Exception Handling with Method Overriding

Let's see the concept of method overriding with exception handling.

ExceptionHandling with MethodOverriding

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.

Core Java

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.
6. class Child extends Parent{
7.     void msg()throws IOException{
8.         System.out.println("child");
9.     }
10. public static void main(String args[]){
11.     Parent p=new Child();
12.     p.msg();
13. }
14. }
```

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.
6. class Child extends Parent{
7.     void msg()throws ArithmeticException{
8.         System.out.println("child");
9.     }
10. public static void main(String args[]){
11.     Parent p=new Child();
12.     p.msg();
13. }
14. }
```

Output:child

Core Java

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.
6. class Child extends Parent{
7.     void msg()throws Exception{System.out.println("child");}
8.
9. public static void main(String args[]){
10.    Parent p=new Child();
11.    try{
12.        p.msg();
13.    }catch(Exception e){}
14. }
15. }
```

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.
6. class Child extends Parent{
7.     void msg()throws Exception{System.out.println("child");}
8.
9. public static void main(String args[]){
10.    Parent p=new Child();
11.    try{
12.        p.msg();
13.    }catch(Exception e){}
14. }
15. }
```

Output:child

Example in case subclass overridden method declares subclass exception

```
1. import java.io.*;
```

Core Java

```
2. class Parent{  
3.     void msg()throws Exception{System.out.println("parent");}  
4. }  
5.  
6. class Child extends Parent{  
7.     void msg()throws ArithmeticException{System.out.println("child");}  
8.  
9.     public static void main(String args[]){  
10.        Parent p=new Child();  
11.        try{  
12.            p.msg();  
13.        }catch(Exception e){ }  
14.    }  
15. }
```

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.  
6. class Child extends Parent{  
7.     void msg(){System.out.println("child");}  
8.  
9.     public static void main(String args[]){  
10.        Parent p=new Child();  
11.        try{  
12.            p.msg();  
13.        }catch(Exception e){ }  
14.    }  
15. }
```

Output:child

Covariant Return Type

The covariant return type specifies that the return type may vary in the same direction as the subclass.

Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type. Let's take a simple example:

Note: If you are beginner to java, skip this topic and return to it after OOPs concepts.

Simple example of Covariant Return Type

```
1. class A{  
2.     A get(){return this;}  
3. }  
4.  
5. class B extends A{  
6.     B get(){return this;}  
7.     void message(){System.out.println("welcome to covariant return type");}  
8.  
9. public static void main(String args[]){  
10.    new B().get().message();  
11. }  
12. }
```

Output:welcome to covariant return type

As you can see in the above example, the return type of the get() method of A class is A but the return type of the get() method of B class is B. Both methods have different return type but it is method overriding. This is known as covariant return type.

super keyword

The **super** 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 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.  
5. class Bike extends Vehicle{  
6.     int speed=100;  
7.  
8.     void display(){  
9.         System.out.println(speed);//will print speed of Bike  
10.    }  
11.    public static void main(String args[]){
```

Core Java

```
12. Bike b=new Bike();
13. b.display();
14.
15. }
16. }
```

Output:100

In the above example Vehicle and Bike both class have a common property speed. Instance variable of current class is referred by instance by default, but I have to refer parent class instance variable that is why we use super keyword to distinguish between parent class instance variable and current class instance variable.

Solution by super keyword

```
1. //example of super keyword
2.
3. class Vehicle{
4.     int speed=50;
5. }
6.
7. class Bike extends Vehicle{
8.     int speed=100;
9.
10. void display(){
11.     System.out.println(super.speed);//will print speed of Vehicle now
12. }
13. public static void main(String args[]){
14.     Bike b=new Bike();
15.     b.display();
16.
17. }
18. }
```

Output:50

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:

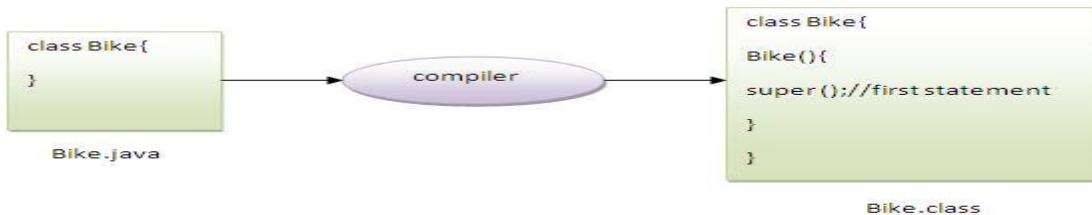
```
1. class Vehicle{
2.     Vehicle(){System.out.println("Vehicle is created");}
3. }
4.
5. class Bike extends Vehicle{
6.     Bike(){
7.         super();//will invoke parent class constructor
8.         System.out.println("Bike is created");
9.     }
10. public static void main(String args[]){
11.     Bike b=new Bike();
12. }
```

Core Java

```
13. }  
14. }
```

Output:Vehicle is created
Bike is created

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.

Another example of super keyword where super() is provided by the compiler implicitly.

```
1. class Vehicle{  
2.   Vehicle(){System.out.println("Vehicle is created");}  
3. }  
4.  
5. class Bike extends Vehicle{  
6.   int speed;  
7.   Bike(int speed){  
8.     this.speed=speed;  
9.     System.out.println(speed);  
10. }  
11. public static void main(String args[]){  
12.   Bike b=new Bike(10);  
13. }  
14. }
```

Output:Vehicle is created

10

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. }
```

Core Java

```
4.  
5. class Student extends Person{  
6.     void message(){System.out.println("welcome to java");}  
7.  
8.     void display(){  
9.         message();//will invoke current class message() method  
10.        super.message();//will invoke parent class message() method  
11.    }  
12.  
13. public static void main(String args[]){  
14.     Student s=new Student();  
15.     s.display();  
16. }  
17. }
```

Output:welcome to java

welcome

In the above example Student and Person both classes have message() method if we call message() method from Student class, it will call the message() method of Student class not of Person class because priority is given to local.

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.  
5. class Student extends Person{  
6.  
7.     void display(){  
8.         message();//will invoke parent class message() method  
9.     }  
10.  
11. public static void main(String args[]){  
12.     Student s=new Student();  
13.     s.display();  
14. }  
15. }
```

Output:welcome

Instance initializer block:

Instance Initializer block is used to initialize the instance data member. It run each time

Core Java

when object of the class is created.

The initialization of the instance variable can be directly but there can be performed extra operations while initializing the instance variable in the instance initializer block.

Que) What is the use of instance initializer block while we can directly assign a value in instance data member? For example:

```
1. class Bike{  
2.     int speed=100;  
3. }
```

Why use instance initializer block?

Suppose I have to perform some operations while assigning value to instance data member e.g. a for loop to fill a complex array or error handling etc.

Example of Instance initializer block

Let's see the simple example of instance initializer block the performs initialization.

```
1. class Bike{  
2.     int speed;  
3.  
4.     Bike(){System.out.println("speed is "+speed);}  
5.  
6.     {speed=100;}  
7.  
8.     public static void main(String args[]){  
9.         Bike b1=new Bike();  
10.        Bike b2=new Bike();  
11.    }  
12. }
```

Output:
speed is 100
speed is 100

There are three places in java where you can perform operations:

1. method
2. constructor
3. block

What is invoked firstly instance initializer block or constructor?

```
1. class Bike{  
2.     int speed;
```

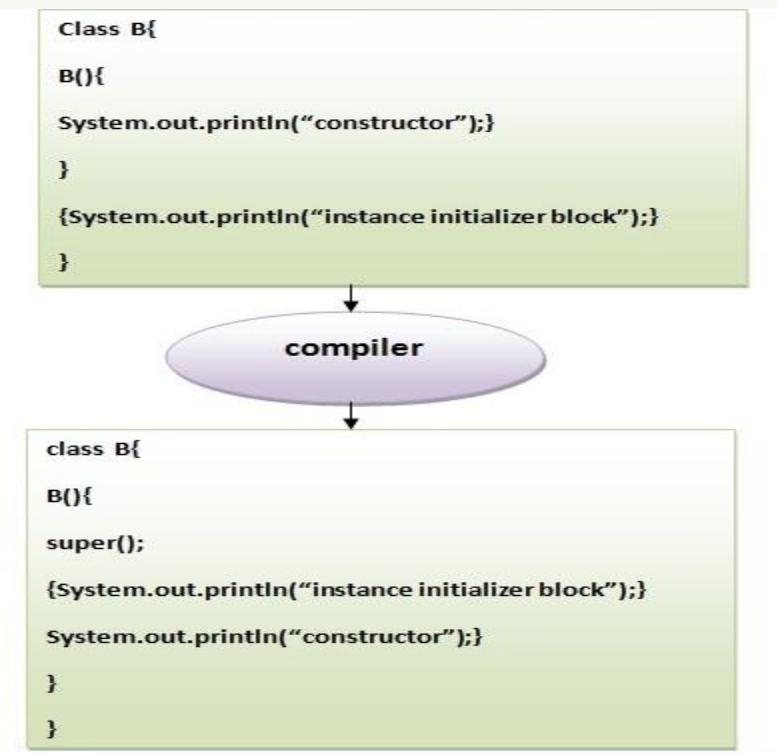
Core Java

```
3.  
4. Bike(){System.out.println("constructor is invoked");}  
5.  
6. {System.out.println("instance initializer block invoked");}  
7.  
8. public static void main(String args[]){  
9.     Bike b1=new Bike();  
10.    Bike b2=new Bike();  
11. }  
12. }
```

Output:
instance initializer block invoked
constructor is invoked
instance initializer block invoked
constructor is invoked

In the above example, it seems that instance initializer block is firstly invoked but NO. Instance intializer block is invoked at the time of object creation. The java compiler copies the instance initializer block in the costructor after the first statement super(). So firstly, constructor is invoked. Let's understand it by the figure given below:

Note: The java compiler copies the code of instance initializer block in every constructor.



Rules for instance initializer block :

There are mainly three rules for the instance initializer block. They are as follows:

1. The instance initializer block is created when instance of the class is created.

Core Java

2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
3. The instance initializer block comes in the order in which they appear.

Program of instance initializer block that is invoked after super()

```
1. class A{  
2. A(){  
3. System.out.println("parent class constructor invoked");  
4. }  
5. }  
6. class B extends A{  
7. B(){  
8. super();  
9. System.out.println("child class constructor invoked");  
10. }  
11.  
12. {System.out.println("instance initializer block is invoked");}  
13.  
14. public static void main(String args[]){  
15. B b=new B();  
16. }  
17. }
```

Output:
parent class constructor invoked
instance initializer block is invoked
child class constructor invoked

Another example of instance block

```
1. class A{  
2. A(){  
3. System.out.println("parent class constructor invoked");  
4. }  
5. }  
6.  
7. class B extends A{  
8. B(){  
9. super();  
10. System.out.println("child class constructor invoked");  
11. }  
12.  
13. B(int a){  
14. super();  
15. System.out.println("child class constructor invoked "+a);  
16. }  
17.  
18. {System.out.println("instance initializer block is invoked");}  
19.  
20. public static void main(String args[]){
```

```
21. B b1=new B();  
22. B b2=new B(10);  
23. }  
24. }
```

Output:parent class constructor invoked
instance initializer block is invoked
child class constructor invoked
parent class constructor invoked
instance initializer block is invoked
child class constructor invoked 10

Final Keyword in Java

The **final keyword** in java is used to restrict the user. The 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.

Java Final Keyword

- ⇒ Stop Value Change
- ⇒ Stop Method Overriding
- ⇒ Stop Inheritance

javatpoint.com

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

Output:Compile Time Error

2) 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.  
5. class Honda extends Bike{  
6.     void run(){System.out.println("running safely with 100kmph");}  
7.  
8.     public static void main(String args[]){  
9.         Honda honda= new Honda();  
10.        honda.run();  
11.    }  
12. }
```

Output:Compile Time Error

3) final class

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

Example of final class

```
1. final class Bike{ }  
2.  
3. class Honda extends Bike{  
4.     void run(){System.out.println("running safely with 100kmph");}  
5.
```

Core Java

```
6. public static void main(String args[]){
7.     Honda honda= new Honda();
8.     honda.run();
9. }
10. }
```

Output:Compile Time Error

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

Output:running...

Q) What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

Example of blank final variable

```
1. class Student{
2.     int id;
3.     String name;
4.     final String PAN_CARD_NUMBER;
5.     ...
6. }
```

Que) Can we initialize blank final variable?

Yes, but only in constructor. For example:

```
1. class Bike{
2.     final int speedlimit;//blank final variable
3.
4.     Bike(){
5.         speedlimit=70;
```

Core Java

```
6. System.out.println(speedlimit);
7. }
8.
9. public static void main(String args[]){
10.   new Bike();
11. }
12. }
```

Output:70

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

```
1. class A{
2.   static final int data;//static blank final variable
3.   static{ data=50; }
4.   public static void main(String args[]){
5.     System.out.println(A.data);
6.   }
7. }
```

Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

```
1. class Bike{
2.   int cube(final int n){
3.     n=n+2;//can't be changed as n is final
4.     n*n*n;
5.   }
6.   public static void main(String args[]){
7.     Bike b=new Bike();
8.     b.cube(5);
9.   }
10. }
```

Output:Compile Time Error

Q) Can we declare a constructor final?

No, because constructor is never inherited.

Runtime Polymorphism

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.

Core Java

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 Runtime Polymorphism

In this example, we are creating two classes Bike and Splender. Splender 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.
7. `public static void main(String args[]){`
8. `Bike b = new Splender();//upcasting`
9. `b.run();`
10. `}`
11. `}`

Output: running safely with 60km.

Core Java

Runtime Polymorphism with data member

Method is overriden 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 Honda extends Bike{  
5.     int speedlimit=150;  
6. }  
7. public static void main(String args[]){  
8.     Bike obj=new Honda();  
9.     System.out.println(obj.speedlimit); //90  
10. }
```

Output: 90

Runtime Polymorphism with Multilevel Inheritance

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

```
1. class Animal{  
2.     void eat(){System.out.println("eating");}  
3. }  
4.  
5. class Dog extends Animal{  
6.     void eat(){System.out.println("eating fruits");}  
7. }  
8.  
9. class BabyDog extends Dog{  
10.    void eat(){System.out.println("drinking milk");}  
11. }  
12. public static void main(String args[]){  
13.     Animal a1,a2,a3;  
14.     a1=new Animal();  
15.     a2=new Dog();  
16.     a3=new BabyDog();  
17.  
18.     a1.eat();  
19.     a2.eat();  
20.     a3.eat();  
21. }
```

Core Java

22. }

Output: eating
eating fruits
drinking Milk

Try for Output

```
1. class Animal{  
2. void eat(){System.out.println("animal is eating...");}  
3. }  
4.  
5. class Dog extends Animal{  
6. void eat(){System.out.println("dog is eating...");}  
7. }  
8.  
9. class BabyDog extends Dog{  
10. public static void main(String args[]){  
11. Animal a=new BabyDog();  
12. a.eat();  
13. }}
```

Output: Dog is eating

Since, BabyDog is not overriding the eat() method, so eat() method of Dog class is invoked.

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[]){
```

Core Java

```
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.  
4. public static void main(String args[]){  
5. Dog d1=new Dog();  
6. d1.eat();  
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.  
5. class Dog extends Animal{  
6. void eat(){System.out.println("dog is eating...");}  
7.  
8. public static void main(String args[]){
```

Core Java

```
9. Animal a=new Dog();
10. a.eat();
11. }
12. }
```

Output:dog is eating...

instanceof operator

The **instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof operator is also known as type comparison operator because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that have null value, it returns false.

Simple example of instanceof operator

Let's see the simple example of instance operator where it tests the current class.

```
1. class Simple{
2. public static void main(String args[]){
3. Simple s=new Simple();
4. System.out.println(s instanceof Simple);//true
5. }
6. }
```

Output:true

An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.

Another example of instanceof operator

```
1. class Animal{}
2. class Dog extends Animal{//Dog inherits Animal
3.
4. public static void main(String args[]){
5. Dog d=new Dog();
6. System.out.println(d instanceof Animal);//true
7. }
8. }
```

Output:true

Core Java

instanceof operator with a variable that have null value

If we apply instanceof operator with a variable that have null value, it returns false. Let's see the example given below where we apply instanceof operator with the variable that have null value.

```
1. class Dog{  
2.     public static void main(String args[]){  
3.         Dog d=null;  
4.         System.out.println(d instanceof Dog);//false  
5.     }  
6. }
```

Output:false

Downcasting with instanceof operator

When Subclass type refers to the object of Parent class, it is known as downcasting. If we perform it directly, compiler gives Compilation error. If you perform it by typecasting, ClassCastException is thrown at runtime. But if we use instanceof operator, downcasting is possible.

```
1. Dog d=new Animal();//Compilation error
```

If we perform downcasting by typecasting, ClassCastException is thrown at runtime.

```
1. Dog d=(Dog)new Animal();  
2. //Compiles successfully but ClassCastException is thrown at runtime
```

Possibility of downcasting with instanceof operator

Let's see the example, where downcasting is possible by instanceof operator.

```
1. class Animal { }  
2.  
3. class Dog extends Animal {  
4.     static void method(Animal a) {  
5.         if(a instanceof Dog){  
6.             Dog d=(Dog)a;//downcasting  
7.             System.out.println("ok downcasting performed");  
8.         }  
9.     }  
10.  
11. public static void main (String [] args) {  
12.     Animal a=new Dog();  
13.     Dog.method(a);  
14. }  
15.  
16. }
```

Core Java

Output:ok downcasting performed

Downcasting without the use of instanceof operator

Downcasting can also be performed without the use of instanceof operator as displayed in the following example:

```
1. class Animal { }
2. class Dog extends Animal {
3.     static void method(Animal a) {
4.         Dog d=(Dog)a;//downcasting
5.         System.out.println("ok downcasting performed");
6.     }
7.     public static void main (String [] args) {
8.         Animal a=new Dog();
9.         Dog.method(a);
10.    }
11. }
```

Output:ok downcasting performed

Let's take closer look at this, actual object that is referred by a, is an object of Dog class. So if we downcast it, it is fine. But what will happen if we write:

```
1. Animal a=new Animal();
2. Dog.method(a);
3. //Now ClassCastException but not in case of instanceof operator
```

Understanding Real use of instanceof operator

Let's see the real use of instanceof keyword by the example given below.

```
1. interface Printable{}
2. class A implements Printable{
3.     public void a(){System.out.println("a method");}
4. }
5. class B implements Printable{
6.     public void b(){System.out.println("b method");}
7. }
8.
9. class Call{
10.     void invoke(Printable p){//upcasting
11.         if(p instanceof A){
12.             A a=(A)p;//Downcasting
13.             a.a();
14.         }
15.         if(p instanceof B){
16.             B b=(B)p;//Downcasting
17.             b.b();
18.         }
19.     }
20. }
```

Core Java

```
19.  
20. }  
21. }//end of Call class  
22.  
23. class Test{  
24. public static void main(String args[]){  
25. Printable p=new B();  
26. Call c=new Call();  
27. c.invoke(p);  
28. }  
29. }
```

Output: b method

Abstract class in Java

A class that is declared with abstract keyword, is known as abstract class. Before learning abstract class, let's understand the abstraction first.

Abstraction

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

Core Java

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 Abstraction

There are two ways to achieve abstraction in java

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

Abstract class

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.

Syntax to declare the abstract class

1. **abstract class** <class_name>{ }

abstract method

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

Syntax to define the abstract method

1. **abstract** return_type <method_name>();//no braces{ }

Example of abstract class that have abstract method

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

```
1. abstract class Bike{  
2.   abstract void run();  
3. }  
4.  
5. class Honda extends Bike{  
6.   void run(){System.out.println("running safely..");}  
7.  
8.   public static void main(String args[]){  
9.     Bike obj = new Honda();  
10.    obj.run();  
11.  }  
12. }  
Output:running safely..
```

Core Java

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.

```
1. abstract class Shape{  
2.   abstract void draw();  
3. }  
4.  
5. class Rectangle extends Shape{  
6.   void draw(){System.out.println("drawing rectangle");}  
7. }  
8.  
9. class Circle extends Shape{  
10.  void draw(){System.out.println("drawing circle");}  
11. }  
12.  
13. class Test{  
14.   public static void main(String args[]){  
15.     Shape s=new Circle();  
16.     //In real scenario, Object is provided through factory method  
17.     s.draw();  
18.   }  
19. }
```

Output:drawing circle

Abstract class having constructor, data member, methods etc.

Note: An abstract class can have data member, abstract method, method body, constructor and even main() method.

```
1. //example of abstract class that have method body  
2. abstract class Bike{  
3.   abstract void run();  
4.   void changeGear(){System.out.println("gear changed");}  
5. }  
6.  
7. class Honda extends Bike{  
8.   void run(){System.out.println("running safely..");}  
9. }  
10. public static void main(String args[]){
```

Core Java

```
11. Bike obj = new Honda();  
12. obj.run();  
13. obj.changeGear();  
14. }  
15. }
```

Output:running safely..

 gear changed

```
1. //example of abstract class having constructor, field and method  
2. abstract class Bike  
3. {  
4.     int limit=30;  
5.     Bike(){System.out.println("constructor is invoked");}  
6.     void getDetails(){System.out.println("it has two wheels");}  
7.     abstract void run();  
8. }  
9.  
10. class Honda extends Bike{  
11.     void run(){System.out.println("running safely..");}  
12. }  
13. public static void main(String args[]){  
14.     Bike obj = new Honda();  
15.     obj.run();  
16.     obj.getDetails();  
17.     System.out.println(obj.limit);  
18. }  
19. }
```

Output:constructor is invoked

running safely..

it has two wheels

30

Rule: If there is any abstract method in a class, that class must be abstract.

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

Output:compile time error

Rule: If you are extending any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract.

Another real scenario of abstract class

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

Note: If you are beginner to java, learn interface first and skip this example.

```
1. interface A{
```

Core Java

```
2. void a();  
3. void b();  
4. void c();  
5. void d();  
6. }  
7.  
8. abstract class B implements A{  
9.     public void c(){System.out.println("I am C");}  
10. }  
11.  
12. class M extends B{  
13.     public void a(){System.out.println("I am a");}  
14.     public void b(){System.out.println("I am b");}  
15.     public void d(){System.out.println("I am d");}  
16. }  
17.  
18. class Test{  
19.     public static void main(String args[]){  
20.         A a=new M();  
21.         a.a();  
22.         a.b();  
23.         a.c();  
24.         a.d();  
25.     }}  
Output:I am a  
I am b  
I am c  
I am d
```

Interface in Java

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

The interface is **a mechanism to achieve fully abstraction** in java. There can be only abstract methods in the interface. It is used to achieve fully abstraction and multiple inheritance in Java.

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

It cannot be instantiated just like abstract class.

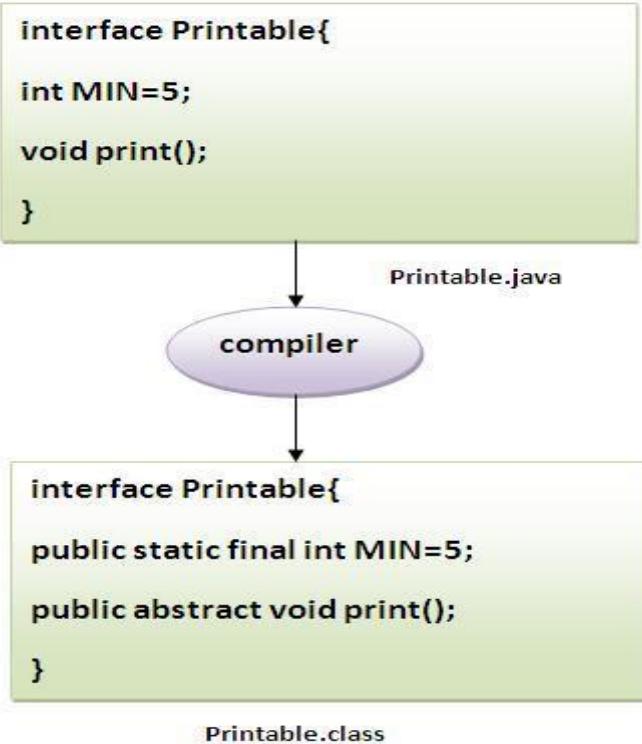
Why use Interface?

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.

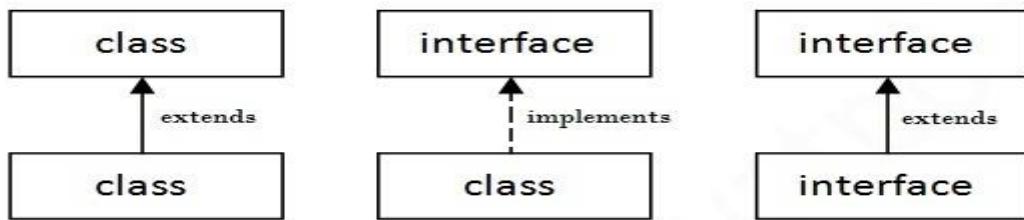
In other words, Interface fields are public, static and final by default, and methods are public and abstract.



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**.

Core Java



Simple example of Interface

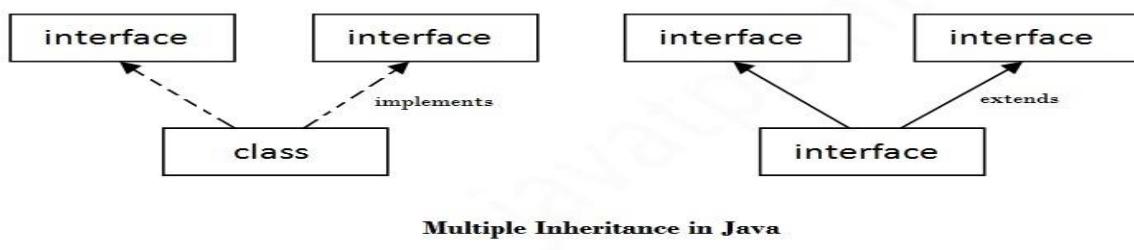
In this example, Printable interface have only one method, its implementation is provided in the A class.

```
1. interface printable{
2.     void print();
3. }
4.
5. class A implements printable{
6.     public void print(){System.out.println("Hello");}
7.
8. public static void main(String args[]){
9.     A obj = new A();
10.    obj.print();
11. }
12. }
```

Output:Hello

Multiple inheritance in Java by interface

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



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

Core Java

```
7. }
8.
9. class A implements Printable,Showable{
10.
11. public void print(){System.out.println("Hello");}
12. public void show(){System.out.println("Welcome");}
13.
14. public static void main(String args[]){
15. A obj = new A();
16. obj.print();
17. obj.show();
18. }
19. }
```

Output:Hello

Welcome

Q) Multiple inheritance is not supported in case of class but it is supported in case of 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.
5. interface Showable{
6. void print();
7. }
8.
9. class A implements Printable,Showable{
10.
11. public void print(){System.out.println("Hello");}
12.
13. public static void main(String args[]){
14. A obj = new A();
15. obj.print();
16. }
17. }
```

Output:Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class A, so there is no ambiguity.

Note: A class implements interface but One interface extends another interface .

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

Core Java

```
4.  
5. interface Showable extends Printable{  
6.     void show();  
7. }  
8.  
9. class A implements Showable{  
10.  
11.     public void print(){System.out.println("Hello");}  
12.     public void show(){System.out.println("Welcome");}  
13.  
14.     public static void main(String args[]){  
15.         A obj = new A();  
16.         obj.print();  
17.         obj.show();  
18.     }  
19. }
```

Output:Hello

Welcome

Que) What is marker or tagged interface ?

An interface that have no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

```
1. //How Serializable interface is written?  
2.  
3. public interface Serializable{  
4. }
```

Nested Interface

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. }
```

Nested Interface

Core Java

An interface which is declared within another interface or class is known as nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred by the outer interface or class. It can't be accessed directly.

Points to remember for nested interfaces

There are given some points that should be remembered by the java programmer.

- Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.
- Nested interfaces are declared static implicitly.

Syntax of nested interface which is declared within the interface

```
1. interface interface_name{  
2. ...  
3. interface nested_interface_name{  
4. ...  
5. }  
6. }  
7.
```

Syntax of nested interface which is declared within the class

```
1. class class_name{  
2. ...  
3. interface nested_interface_name{  
4. ...  
5. }  
6. }  
7.
```

Example of nested interface which is declared within the interface

In this example, we are going to learn how to declare the nested interface and how we can access it.

```
1. interface Showable{  
2.   void show();  
3.   interface Message{  
4.     void msg();  
5.   }  
6. }  
7.  
8. class Test implements Showable.Message{  
9.   public void msg(){System.out.println("Hello nested interface");}  
10.  
11. public static void main(String args[]){  
12.   Showable.Message message=new Test()://upcasting here
```

Core Java

```
13. message.msg();  
14. }  
15. }
```

Output:hello nested interface

As you can see in the above example, we are accessing the Message interface by its outer interface Showable because it cannot be accessed directly. It is just like almirah inside the room, we cannot access the almirah directly because we must enter the room first. In collection framework, sun microsystem has provided a nested interface Entry. Entry is the subinterface of Map i.e. accessed by Map.Entry.

Internal code generated by the java compiler for nested interface Message

The java compiler internally creates public and static interface as displayed below::

```
1. public static interface Showable$Message  
2. {  
3.   public abstract void msg();  
4. }
```

Example of nested interface which is declared within the class

Let's see how can we define an interface inside the class and how can we access it.

```
1. class A{  
2.   interface Message{  
3.     void msg();  
4.   }  
5. }  
6.  
7. class Test implements A.Message{  
8.   public void msg(){System.out.println("Hello nested interface");}  
9.  
10. public static void main(String args[]){  
11.   A.Message message=new Test();//upcasting here  
12.   message.msg();  
13. }  
14. }
```

Output:hello nested interface

Can we define a class inside the interface ?

Yes, Ofcourse! If we define a class inside the interface, java compiler creates a static nested class. Let's see how can we define a class within the interface:

```
1. interface M{  
2.   class A{}  
3. }
```

Package in Java

Core Java

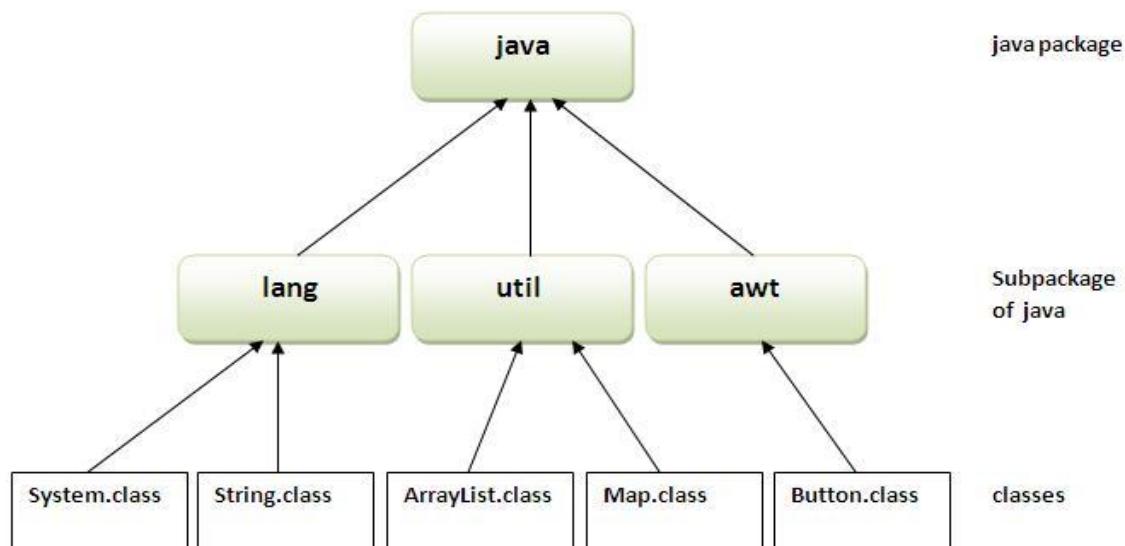
A **package** is a group of similar types of classes, interfaces and sub-packages.

Package can be categorized in two form, built-in package and user-defined package. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Package

- Package is used to categorize the classes and interfaces so that they can be easily maintained.
- Package provides access protection.
- Package removes naming collision.



Simple example of package

The **package keyword** is used to create a package.

```
1. //save as Simple.java
2.
3. package mypack;
4. public class Simple{
5.   public static void main(String args[]){
6.     System.out.println("Welcome to package");
7.   }
8. }
```

Core Java

How to compile the Package (if not using IDE)

If you are not using any IDE, you need to follow the **syntax** given below:

1. `javac -d directory javafilename`

For example

1. `javac -d . Simple.java`

The `-d` switch specifies the destination where to put the generated class file. You can use any directory name like `/home` (in case of Linux), `d:/abc` (in case of windows) etc. If you want to keep the package within the same directory, you can use `.` (dot).

How to run the Package (if not using IDE)

You need to use fully qualified name e.g. `mypack.Simple` etc to run the class.

To Compile: `javac -d . Simple.java`

To Run: `java mypack.Simple`

Output: Welcome to package

The `-d` is a switch that tells the compiler where to put the class file i.e. it represents destination. The `.` represents the current folder.

How to access package from another package?

There are three ways to access the package from outside the package.

1. `import package.*;`
2. `import package.classname;`
3. `fully qualified name.`

Using packagename.*

If you use `package.*` then all the classes and interfaces of this package will be accessible but not subpackages.

The `import` keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

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

Output:Hello

Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

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

Output:Hello

Core Java

Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

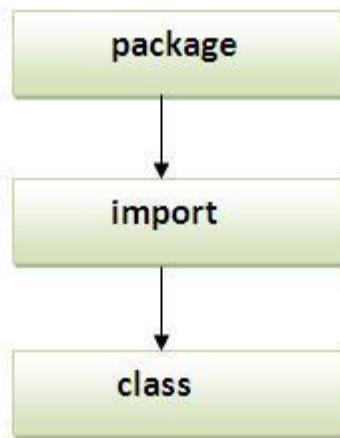
```
1. //save by A.java
2.
3. package pack;
4. public class A{
5.     public void msg(){System.out.println("Hello");}
6. }
1. //save by B.java
2.
3. package mypack;
4. class B{
5.     public static void main(String args[]){
6.         pack.A obj = new pack.A(); //using fully qualified name
7.         obj.msg();
8.     }
9. }
```

Output:Hello

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Note: Sequence of the program must be package then import then class.



Subpackage

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has defined a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.

Example of Subpackage

```
1. package com.javatpoint.core;
2. class Simple{
3.     public static void main(String args[]){
4.         System.out.println("Hello subpackage");
5.     }
6. }
```

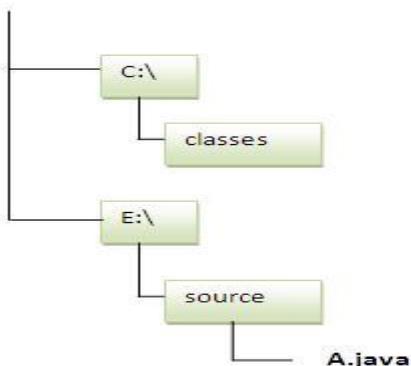
To Compile: javac -d . Simple.java

To Run: java com.javatpoint.core.Simple

Output:Hello subpackage

How to send the class file to another directory or drive?

There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:



Core Java

```
1. //save as Simple.java
2.
3. package mypack;
4. public class Simple{
5.     public static void main(String args[]){
6.         System.out.println("Welcome to package");
7.     }
8. }
```

To Compile:

```
e:\sources> javac -d c:\classes Simple.java
```

To Run:

To run this program from e:\source directory, you need to set classpath of the directory where the class file resides.

```
e:\sources> set classpath=c:\classes;;
```

```
e:\sources> java mypack.Simple
```

Another way to run this program by -classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

```
e:\sources> java -classpath c:\classes mypack.Simple
```

Output:Welcome to package

Ways to load the class files or jar files

There are two ways to load the class files temporary and permanent.

- Temporary
 - By setting the classpath in the command prompt
 - By -classpath switch
- Permanent
 - By setting the classpath in the environment variables
 - By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

Rule: There can be only one public class in a java source file and it must be saved by the public class name.

Core Java

1. //save as C.java otherwise Compile Time Error
- 2.
3. **class** A{}
4. **class** B{}
5. **public class** C{}

How to put two public classes in a package?

If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example:

1. //save as A.java
- 2.
3. **package** javatpoint;
4. **public class** A{}
1. //save as B.java
- 2.
3. **package** javatpoint;
4. **public class** B{}

Access Modifiers

There are two types of modifiers in java: **access modifier** and **non-access modifier**. The access modifiers specifies accessibility (scope) of a datamember, method, constructor or class.

There are 4 types of 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

The private access modifier is accessible only within class.

Simple example of private access modifer

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{

Core Java

```
2. private int data=40;  
3. private void msg(){System.out.println("Hello java");}  
4. }  
5.  
6. public class Simple{  
7.   public static void main(String args[]){  
8.     A obj=new A();  
9.     System.out.println(obj.data);//Compile Time Error  
10.    obj.msg();//Compile Time Error  
11.  }  
12. }
```

Role of Private Constructor:

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

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

2) default

If you don't use any modifier, it is treated as **default** by default. 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.  
3. package pack;  
4. class A{  
5.   void msg(){System.out.println("Hello");}  
6. }  
1. //save by B.java  
2.
```

Core Java

```
3. package mypack;  
4. import pack.*;  
5.  
6. class B{  
7.     public static void main(String args[]){  
8.         A obj = new A(); //Compile Time Error  
9.         obj.msg(); //Compile Time Error  
10.    }  
11. }
```

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

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

Output:Hello

Core Java

4) public

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

Example of public access modifier

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

Output:Hello

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

Core Java

Applying access modifier with method overriding

If you are overriding any method, overriden 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.  
5. public class Simple extends A{  
6.     void msg(){System.out.println("Hello java");}//C.T.Error  
7.     public static void main(String args[]){  
8.         Simple obj=new Simple();  
9.         obj.msg();  
10.    }  
11. }
```

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

Encapsulation in Java

Encapsulation is a process of wrapping code and data together into a single unit e.g. capsule i.e mixed of several medicines.



Capsule

We can create a fully encapsulated class 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.

Java Bean is the example of fully encapsulated class.

Advantage of Encapsulation

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
```

Core Java

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

Compile By: javac -d . Test.java

Run By: java com.javatpoint.Test

Output: vijay

Object class in Java

The **Object class** is the parent class of all the classes in java bydefault. In other words, it is the topmost class of java.

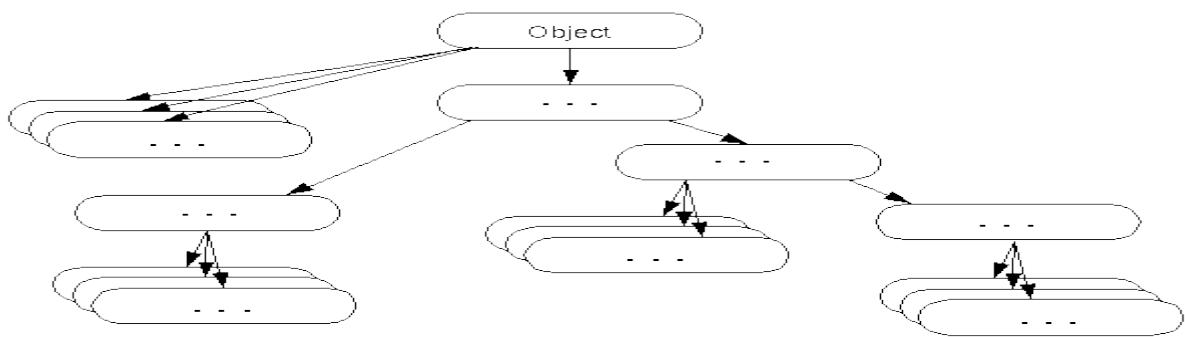
The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.

Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee,Student etc, we can use Object class reference to refer that object. For example:

1. Object obj=getObject();//we don't what object would be returned from this method

The Object class provides some common behaviours to all the objects such as object can be compared, object can be cloned, object can be notified etc.

Core Java



Methods of Object class

The Object class provides many methods. They are as follows:

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout) throws InterruptedException	causes the current thread to wait for the specified miliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos) throws InterruptedException	causes the current thread to wait for the specified miliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).

Core Java

public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

We will have the detailed learning of these methods in next chapters.

Object Cloning in Java



The **object cloning** is a way to create exact copy of an object. For this purpose, `clone()` method of Object class is used to clone an object.

The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, `clone()` method generates **CloneNotSupportedException**.

The **clone()** method is defined in the Object class. Syntax of the `clone()` method is as follows:

1. **protected Object clone() throws CloneNotSupportedException**

Why use `clone()` method ?

The **clone()** method saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing to be performed that is why we use object cloning.

Advantage of Object cloning

Less processing task.

Example of clone() method (Object cloning)

Let's see the simple example of object cloning

```
1. class Student implements Cloneable{
2.     int rollno;
3.     String name;
4.
5.     Student(int rollno,String name){
6.         this.rollno=rollno;
7.         this.name=name;
8.     }
9.
10.    public Object clone()throws CloneNotSupportedException{
11.        return super.clone();
12.    }
13.
14.    public static void main(String args[]){
15.        try{
16.            Student s1=new Student(101,"amit");
17.
18.            Student s2=(Student)s1.clone();
19.
20.            System.out.println(s1.rollno+" "+s1.name);
21.            System.out.println(s2.rollno+" "+s2.name);
22.
23.        }catch(CloneNotSupportedException c){}
24.
25.    }
26. }
```

Output:101 amit

101 amit

As you can see in the above example, both reference variables have the same value. Thus, the clone() copies the values of an object to another. So we don't need to write explicit code to copy the value of an object to another.

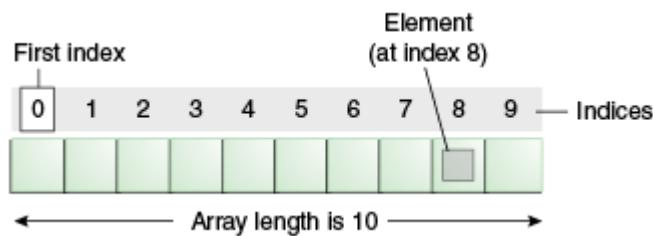
If we create another object by new keyword and assign the values of another object to this one, it will require a lot of processing on this object. So to save the extra processing task we use clone() method.

Array in Java

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

In java, array is an object that contains elements of similar data type. It is a data structure where we store similar elements. We can store only fixed elements in an array.

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



Advantage of 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 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

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array

Syntax to Declare an Array in java

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

Instantiation of an Array in java

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

Core Java

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 B{  
2.     public static void main(String args[]){  
3.           
4.             int a[]={new int[5]);//declaration and instantiation  
5.             a[0]=10;//initialization  
6.             a[1]=20;  
7.             a[2]=70;  
8.             a[3]=40;  
9.             a[4]=50;  
10.              
11.            //printing array  
12.            for(int i=0;i<a.length;i++)//length is the property of array  
13.                System.out.println(a[i]);  
14.              
15.        }  
Output: 10  
20  
70  
40  
50
```

Declaration, Instantiation and Initialization of Java Array

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

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

Let's see the simple example to print this array.

```
1. class B{  
2.     public static void main(String args[]){  
3.           
4.             int a[]={3,3,4,5};//declaration, instantiation and initialization  
5.               
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.               
10.              
Output:33  
3  
4  
5
```

Passing Java Array in the method

We can pass the array in the 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 B{  
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.  
10.  
11.    public static void main(String args[]){  
12.  
13.        int a[]={3,3,4,5};  
14.        min(a);//passing array in the method  
15.  
16.    }  
Output:3
```

Multidimensional array

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[];

Example to initantiate Multidimensional Array in java

1. **int[][] arr=new int[3][3];//3 row and 3 column**

Example to initialize Multidimensional Array in java

1. arr[0][0]=1;
2. arr[0][1]=2;
3. arr[0][2]=3;
4. arr[1][0]=4;
5. arr[1][1]=5;
6. arr[1][2]=6;
7. arr[2][0]=7;
8. arr[2][1]=8;
9. arr[2][2]=9;

Core Java

Example of Multidimensional java array

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

```
1. class B{  
2.     public static void main(String args[]){  
3.           
4.             //declaring and initializing 2D array  
5.             int arr[][]={{1,2,3},{2,4,5},{4,4,5}};  
6.           
7.             //printing 2D array  
8.             for(int i=0;i<3;i++){  
9.                 for(int j=0;j<3;j++){  
10.                     System.out.print(arr[i][j]+" ");  
11.                 }  
12.             System.out.println();  
13.         }  
14.     }  
15. }
```

Output:
1 2 3
2 4 5
4 4 5

What is class name of java array?

In java, array is an object. For array object, an proxy class is created whose name can be obtained by getClass().getName() method on the object.

```
1. class B{  
2.     public static void main(String args[]){  
3.           
4.             int arr[]={4,4,5};  
5.           
6.             Class c=arr.getClass();  
7.             String name=c.getName();  
8.           
9.             System.out.println(name);  
10.        }  
11.    }
```

Output:[I

Copying an array

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

Core Java

Syntax of arraycopy method

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

Example of arraycopy method

```
1. class ArrayCopyDemo {  
2.     public static void main(String[] args) {  
3.         char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',  
4.             'i', 'n', 'a', 't', 'e', 'd' };  
5.         char[] copyTo = new char[7];  
6.  
7.         System.arraycopy(copyFrom, 2, copyTo, 0, 7);  
8.         System.out.println(new String(copyTo));  
9.     }  
10.}
```

Output:caffein

Addition 2 matrices

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

```
1. class AE{  
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.  
7.         //creating another matrix to store the sum of two matrices  
8.         int c[][]=new int[2][3];  
9.  
10.        //adding and printing addition of 2 matrices  
11.        for(int i=0;i<2;i++){  
12.            for(int j=0;j<3;j++){  
13.                c[i][j]=a[i][j]+b[i][j];  
14.                System.out.print(c[i][j]+" ");  
15.            }  
16.            System.out.println();//new line  
17.        }  
18.  
19.    }}
```

Output:2 6 8

6 8 10

Call by Value and Call by Reference in Java

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

Core Java

Example of call by value in java

In case of call by value original value is not changed. Let's take a simple example:

```
1. class Operation{
2.     int data=50;
3.
4.     void change(int data){
5.         data=data+100;//changes will be in the local variable only
6.     }
7.
8.     public static void main(String args[]){
9.         Operation op=new Operation();
10.
11.        System.out.println("before change "+op.data);
12.        op.change(500);
13.        System.out.println("after change "+op.data);
14.
15.    }
16. }
```

output: before change 50
after change 50

Another Example of call by value in java

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

```
1. class Operation2{
2.     int data=50;
3.
4.     void change(Operation2 op){
5.         op.data=op.data+100;//changes will be in the instance variable
6.     }
7.
8.
9.     public static void main(String args[]){
10.        Operation2 op=new Operation2();
11.
12.        System.out.println("before change "+op.data);
13.        op.change(op);//passing object
14.        System.out.println("after change "+op.data);
15.
16.    }
17. }
```

Output: before change 50
after change 150

strictfp keyword

The strictfp keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable. The precision may differ from platform to platform that is why java programming language have provided the strictfp keyword, so that you get same result on every platform. So, now you have better control over the floating-point arithmetic.

Legal code for strictfp keyword

The strictfp keyword can be applied on methods, classes and interfaces.

1. **strictfp class** A{}//strictfp applied on class
1. **strictfp interface** M{}//strictfp applied on interface
1. **class** A{
2. **void** m(){}}//strictfp applied on method
3. }

Illegal code for strictfp keyword

The strictfp keyword can be applied on abstract methods, variables or constructors.

1. **class** B{
2. **strictfp abstract void** m();//Illegal combination of modifiers
3. }
1. **class** B{
2. **strictfp int** data=10;//modifier strictfp not allowed here
3. }
1. **class** B{
2. **strictfp** B(){}}//modifier strictfp not allowed here
3. }

Creating API Document | javadoc tool

We can create document api in java by the help of **javadoc** tool. In the java file, we must use the documentation comment `/**... */` to post information for the class, method, constructor, fields etc.

Let's see the simple class that contains documentation comment.

1. **package** com.abc;
2. `/** This class is a user-defined class that contains one methods cube.*/`
3. **public class** M{
- 4.
5. `/** The cube method prints cube of the given number */`
6. **public static void** cube(**int** n){System.out.println(n*n*n);}
7. }

Core Java

To create the document API, you need to use the javadoc tool followed by java file name. There is no need to compile the javafile.

On the command prompt, you need to write:

```
javadoc M.java
```

to generate the document api. Now, there will be created a lot of html files. Open the index.html file to get the information about the classes.

Command Line Argument

The command-line argument is an argument passed at the time of running the java program. The argument can be received in the program and used as an input.

So, it provides a convenient way to check out the behavior of the program on different values. You can pass **N** numbers of arguments from the command prompt.

Simple example of command-line argument

In this example, we are receiving only one argument and printing it. For running this program, you must pass at least one argument from the command prompt.

```
1. class A{  
2. public static void main(String args[]){  
3.  
4. System.out.println("first argument is: "+args[0]);  
5.  
6. }  
7. }  
1. compile by > javac A.java  
2. run by > java A sonoo  
Output: first argument is: sonoo
```

Example of command-line argument that prints all the values

In this example, we are printing all the arguments passed from the command-line. For this purpose, we have traversed the array using for loop.

```
1. class A{  
2. public static void main(String args[]){  
3.  
4. for(int i=0;i<args.length;i++)  
5. System.out.println(args[i]);  
6.  
7. }  
8. }  
1. compile by > javac A.java  
2. run by > java A sonoo jaiswal 1 3 abc  
Output: sonoo  
        jaiswal  
        1  
        3  
        abc
```

String Handling in Java

String Handling provides a lot of concepts that can be performed on a string such as concatenating string, comparing string, substring etc.

In java, string is basically an immutable object. We will discuss about immutable string later. Let's first understand what is string and how we can create the string object.

String

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

- Why String objects are immutable ?
- How to create an immutable class ?
- What is string constant pool ?
- What code is written by the compiler if you concat any string by + (string concatenation operator) ?
- What is the difference between StringBuffer and StringBuilder class ?

How to create String object?

There are two ways to create String object:

1. By string literal
2. By new keyword

1) String literal

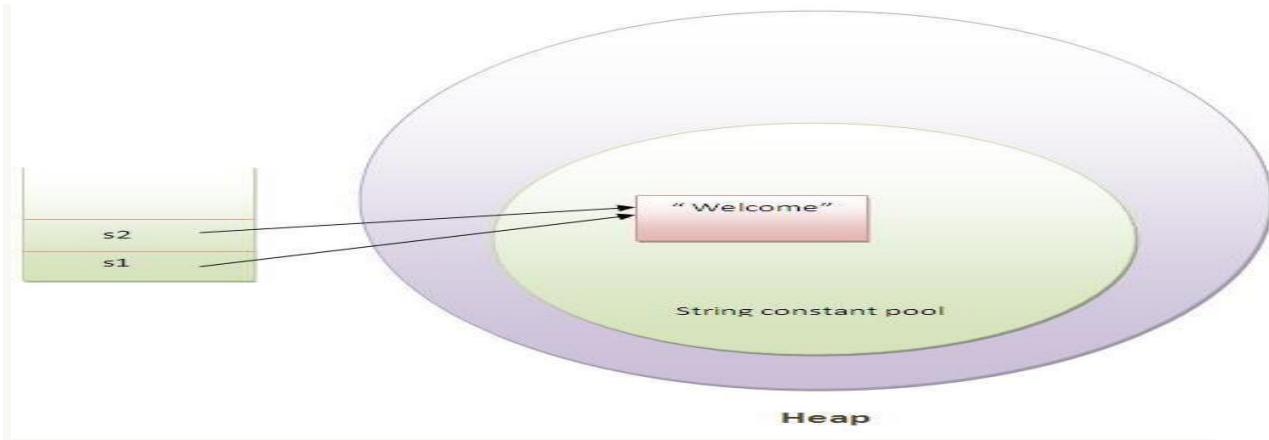
String literal is created by double quote. For Example:

1. String s="Hello";

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 returns. If the string does not exist in the pool, a new String object instantiates, then is placed in the pool. For example:

1. String s1="Welcome";
2. String s2="Welcome";//no new object will be created

Core Java



In the above example only one object will be created. First time JVM will find no string object with the name "Welcome" in string constant pool, so it will create a new object. Second time it will find the string with the name "Welcome" in string constant pool, so it will not create new object whether will return the reference to the same instance.

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

Why java uses concept of string literal?

To make Java more memory efficient (because no new objects are created if it exists already in string constant pool).

2) By new keyword

String s=new String("Welcome");//creates two objects and one reference variable

In such case, JVM will create a new String object in normal(nonpool) Heap memory and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in Heap(nonpool).

What we will learn in String Handling ?

- Concept of String
- Immutable String
- String Comparison
- String Concatenation
- Concept of Substring
- String class methods and its usage
- StringBuffer class
- StringBuilder class
- Creating Immutable class
- `toString()` method
- StringTokenizer class

Immutable String in Java

In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.

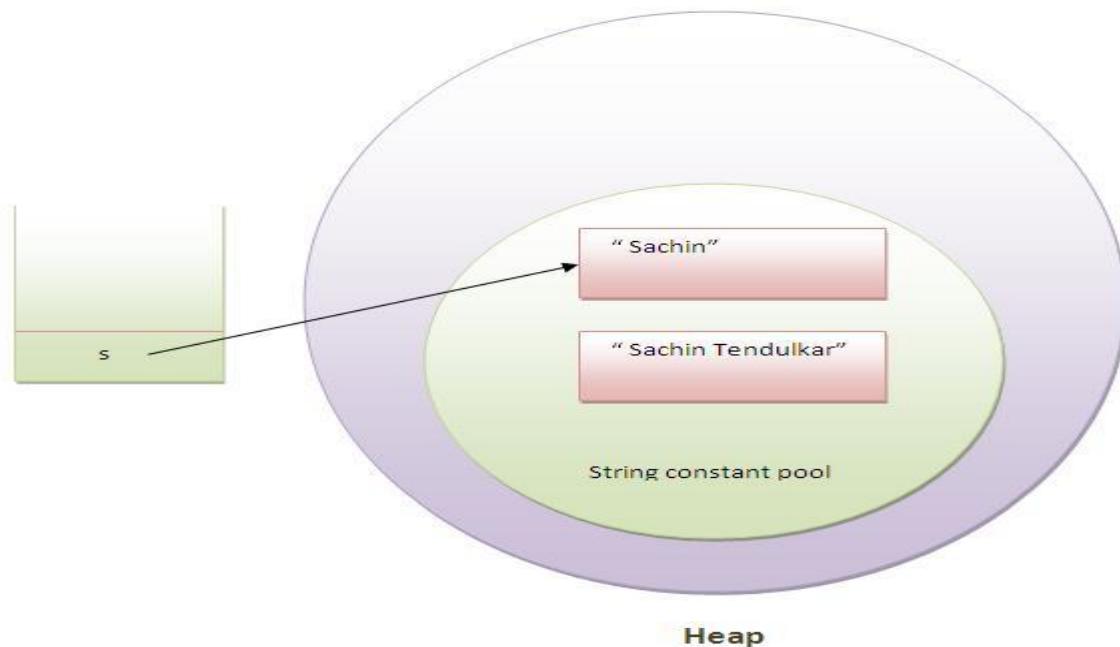
Once string object is created its data or state can't be changed but a new string object is created.

Let's try to understand the immutability concept by the example given below:

```
1. class Simple{  
2.     public static void main(String args[]){  
3.         String s="Sachin";  
4.         s.concat(" Tendulkar");//concat() method appends the string at the end  
5.         System.out.println(s);//will print Sachin because strings are immutable objects  
6.     }  
7. }
```

Output:Sachin

Now it can be understood by the diagram given below. Here Sachin is not changed but a new object is created with sachintendulkar. That is why string is known as immutable.



As you can see in the above figure that two objects are created but `s` reference variable still refers to "Sachin" not to "Sachin Tendulkar".

But if we explicitly assign it to the reference variable, it will refer to "Sachin Tendulkar" object. For example:

```
1. class Simple{  
2.     public static void main(String args[]){  
3.         String s="Sachin";  
4.         s=s.concat(" Tendulkar");  
5.         System.out.println(s);  
6.     }  
7. }
```

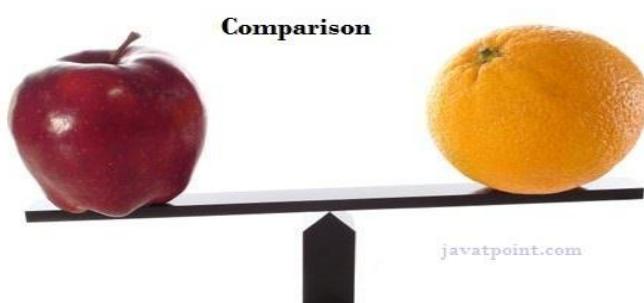
Output:Sachin Tendulkar

In such case, s points to the "Sachin Tendulkar". Please notice that still sachin object is not modified.

Why string objects are immutable in java?

Because java uses the concept of string literal. Suppose there are 5 reference variables, all refers to one object "sachin". If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java.

String comparison in Java



We can compare two given strings on the basis of content and reference.

It is used in **authentication** (by equals() method), **sorting** (by compareTo() method), **reference matching** (by == operator) etc.

There are three ways to compare String objects:

1. By equals() method
2. By == operator
3. By compareTo() method

1) By equals() method

equals() method compares the original content of the string. It compares values of string for equality. String class provides two methods:

Core Java

- **public boolean equals(Object another){}** compares this string to the specified object.
- **public boolean equalsIgnoreCase(String another){}** compares this String to another String, ignoring case.

```
1. class Simple{  
2.     public static void main(String args[]){  
3.           
4.         String s1="Sachin";  
5.         String s2="Sachin";  
6.         String s3=new String("Sachin");  
7.         String s4="Saurav";  
8.           
9.         System.out.println(s1.equals(s2));//true  
10.        System.out.println(s1.equals(s3));//true  
11.        System.out.println(s1.equals(s4));//false  
12.    }  
13. }
```

Output:true

```
    true  
    false
```

```
1. //Example of equalsIgnoreCase(String) method  
2. class Simple{  
3.     public static void main(String args[]){  
4.           
5.         String s1="Sachin";  
6.         String s2="SACHIN";  
7.           
8.         System.out.println(s1.equals(s2));//false  
9.         System.out.println(s1.equalsIgnoreCase(s3));//true  
10.    }  
11. }
```

Output:false

```
    true
```

2) By == operator

The == operator compares references not values.

```
1. //<b><i>Example of == operator</i></b>  
2.   
3. class Simple{  
4.     public static void main(String args[]){  
5.           
6.         String s1="Sachin";  
7.         String s2="Sachin";  
8.         String s3=new String("Sachin");  
9.           
10.        System.out.println(s1==s2);//true (because both refer to same instance)  
11.        System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)
```

Core Java

```
12. }
13. }
```

Output:
true
false

3) By compareTo() method:

compareTo() method compares values and returns an int which tells if the values compare less than, equal, or greater than.

Suppose s1 and s2 are two string variables.If:

- **s1 == s2** :0
- **s1 > s2** :positive value
- **s1 < s2** :negative value

```
1. //<b><i>Example of compareTo() method:</i></b>
2.
3. class Simple{
4. public static void main(String args[]){
5.
6.     String s1="Sachin";
7.     String s2="Sachin";
8.     String s3="Ratan";
9.
10.    System.out.println(s1.compareTo(s2));//0
11.    System.out.println(s1.compareTo(s3));//1(because s1>s3)
12.    System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
13. }
14. }
```

Output:
0
1
-1

String Concatenation in Java

Concating strings form a new string i.e. the combination of multiple strings.

There are two ways to concat string objects:

1. By + (string concatenation) operator
2. By concat() method

1) By + (string concatenation) operator

String concatenation operator is used to add strings.For Example:

```
1. //Example of string concatenation operator
2.
```

Core Java

```
3. class Simple{  
4.     public static void main(String args[]){  
5.         String s="Sachin"+" Tendulkar";  
6.         System.out.println(s); //Sachin Tendulkar  
7.     }  
8. }  
9. }
```

Output:Sachin Tendulkar

The compiler transforms this to:

```
1. String s=(new StringBuilder()).append("Sachin").append(" Tendulkar").toString();
```

String concatenation is implemented through the `StringBuilder`(or `StringBuffer`) class and its `append` method. String concatenation operator produces a new string by appending the second operand onto the end of the first operand. The string concatenation operator can concat not only string but primitive values also. For Example:

```
1. class Simple{  
2.     public static void main(String args[]){  
3.         String s=50+30+"Sachin"+40+40;  
4.         System.out.println(s); //80Sachin4040  
5.     }  
6. }  
7. }
```

Output:80Sachin4040

Note:If either operand is a string, the resulting operation will be string concatenation. If both operands are numbers, the operator will perform an addition.

2) By concat() method

`concat()` method concatenates the specified string to the end of current string.

Syntax:`public String concat(String another){}`

```
1. //<b><i>Example of concat(String) method</i></b>  
2.  
3. class Simple{  
4.     public static void main(String args[]){  
5.  
6.         String s1="Sachin ";  
7.         String s2="Tendulkar";  
8.  
9.         String s3=s1.concat(s2);  
10.  
11.        System.out.println(s3); //Sachin Tendulkar  
12.    }
```

Substring in Java



A part of string is called **substring**. In other words, substring is a subset of another string.

In case of substring startIndex starts from 0 and endIndex starts from 1 or startIndex is inclusive and endIndex is exclusive.

You can get substring from the given String object by one of the two methods:

1. **public String substring(int startIndex):** This method returns new String object containing the substring of the given string from specified startIndex (inclusive).
2. **public String substring(int startIndex,int endIndex):** This method returns new String object containing the substring of the given string from specified startIndex to endIndex.

In case of string:

- **startIndex:** starts from index 0(inclusive).
- **endIndex:** starts from index 1(exclusive).

Example of java substring

```
1. //Example of substring() method
2.
3. class Simple{
4.     public static void main(String args[]){
5.
6.         String s="Sachin Tendulkar";
7.         System.out.println(s.substring(6));//Tendulkar
8.         System.out.println(s.substring(0,6));//Sachin
9.     }
10. }
```

Output:Tendulkar
Sachin

Core Java

Methods of String class

java.lang.String class provides a lot of methods to work on string. Let's see the commonly used methods of String class.

Method	Description
1)public boolean equals(Object anObject)	Compares this string to the specified object.
2)public boolean equalsIgnoreCase(String another)	Compares this String to another String, ignoring case.
3)public String concat(String str)	Concatenates the specified string to the end of this string.
4)public int compareTo(String str)	Compares two strings and returns int
5)public int compareToIgnoreCase(String str)	Compares two strings, ignoring case differences.
6)public String substring(int beginIndex)	Returns a new string that is a substring of this string.
7)public String substring(int beginIndex,int endIndex)	Returns a new string that is a substring of this string.
8)public String toUpperCase()	Converts all of the characters in this String to upper case
9)public String toLowerCase()	Converts all of the characters in this String to lower case.
10)public String trim()	Returns a copy of the string, with leading and trailing whitespace omitted.
11)public boolean startsWith(String prefix)	Tests if this string starts with the specified prefix.
12)public boolean endsWith(String suffix)	Tests if this string ends with the specified suffix.
13)public char charAt(int index)	Returns the char value at the specified index.

Core Java

14)public int length()	Returns the length of this string.
15)public String intern()	Returns a canonical representation for the string object.

First seven methods have already been discussed. Now Let's take the example of other methods:

toUpperCase() and toLowerCase() method

```
1. //<b><i>Example of toUpperCase() and toLowerCase() method</i></b>
2.
3. class Simple{
4. public static void main(String args[]){
5.
6.     String s="Sachin";
7.     System.out.println(s.toUpperCase());//SACHIN
8.     System.out.println(s.toLowerCase());//sachin
9.     System.out.println(s);//Sachin(no change in original)
10. }
11. }
```

Output:SACHIN

```
sachin
Sachin
```

trim() method

```
1. //<b><i>Example of trim() method</i></b>
2.
3. class Simple{
4. public static void main(String args[]){
5.
6.     String s=" Sachin ";
7.     System.out.println(s);// Sachin
8.     System.out.println(s.trim());//Sachin
9. }
10. }
```

Output: Sachin

```
Sachin
```

startsWith() and endsWith() method

```
1. //<b><i>Example of startsWith() and endsWith() method</i></b>
2.
3. class Simple{
4. public static void main(String args[]){
5.
6.     String s="Sachin";
7.     System.out.println(s.startsWith("Sa"));//true
```

Core Java

```
8.     System.out.println(s.startsWith("n")); //true
9. }
10. }
```

Output:true

 true

charAt() method

```
1. //<b><i>Example of charAt() method</i></b>
2.
3. class Simple{
4. public static void main(String args[]){
5.
6.     String s="Sachin";
7.     System.out.println(s.charAt(0)); //S
8.     System.out.println(s.charAt(3)); //h
9. }
10. }
```

Output:S

 h

length() method

```
1. //<b><i>Example of length() method</i></b>
2.
3. class Simple{
4. public static void main(String args[]){
5.
6.     String s="Sachin";
7.     System.out.println(s.length()); //6
8. }
9. }
```

Output:6

intern() method

A pool of strings, initially empty, is maintained privately by the class String.

When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

```
1. //<b><i>Example of length() method</i></b>
2.
3. class Simple{
4. public static void main(String args[]){
5.
6.     String s=new String("Sachin");
7.     String s2=s.intern();
8.     System.out.println(s2); //Sachin
```

Core Java

```
9.  }
10. }
```

Output:Sachin

StringBuffer class:

The StringBuffer class is used to created mutable (modifiable) string. The StringBuffer class is same as String except it is mutable i.e. it can be changed.

Note: StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously .So it is safe and will result in an order.

Commonly used Constructors of StringBuffer class:

1. **StringBuffer():** creates an empty string buffer with the initial capacity of 16.
2. **StringBuffer(String str):** creates a string buffer with the specified string.
3. **StringBuffer(int capacity):** creates an empty string buffer with the specified capacity as length.

Commonly used methods of StringBuffer class:

1. **public synchronized StringBuffer append(String s):** is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
2. **public synchronized StringBuffer insert(int offset, String s):** is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
3. **public synchronized StringBuffer replace(int startIndex, int endIndex, String str):** is used to replace the string from specified startIndex and endIndex.
4. **public synchronized StringBuffer delete(int startIndex, int endIndex):** is used to delete the string from specified startIndex and endIndex.
5. **public synchronized StringBuffer reverse():** is used to reverse the string.
6. **public int capacity():** is used to return the current capacity.
7. **public void ensureCapacity(int minimumCapacity):** is used to ensure the capacity at least equal to the given minimum.
8. **public char charAt(int index):** is used to return the character at the specified position.
9. **public int length():** is used to return the length of the string i.e. total number of characters.
10. **public String substring(int beginIndex):** is used to return the substring from the specified beginIndex.
11. **public String substring(int beginIndex, int endIndex):** is used to return the

Core Java

substring from the specified beginIndex and endIndex.

What is mutable string?

A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.

simple example of StringBuffer class by append() method

The append() method concatenates the given argument with this string.

```
1. class A{  
2. public static void main(String args[]){  
3.  
4. StringBuffer sb=new StringBuffer("Hello ");  
5. sb.append("Java");//now original string is changed  
6.  
7. System.out.println(sb);//prints Hello Java  
8. }  
9. }
```

Example of insert() method of StringBuffer class

The insert() method inserts the given string with this string at the given position.

```
1. class A{  
2. public static void main(String args[]){  
3.  
4. StringBuffer sb=new StringBuffer("Hello ");  
5. sb.insert(1,"Java");//now original string is changed  
6.  
7. System.out.println(sb);//prints HJavaello  
8. }  
9. }
```

Example of replace() method of StringBuffer class

The replace() method replaces the given string from the specified beginIndex and endIndex.

```
1. class A{  
2. public static void main(String args[]){  
3.  
4. StringBuffer sb=new StringBuffer("Hello");  
5. sb.replace(1,3,"Java");  
6.  
7. System.out.println(sb);//prints HJava  
8. }  
9. }
```

Core Java

Example of delete() method of StringBuffer class

The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

```
1. class A{
2. public static void main(String args[]){
3.
4. StringBuffer sb=new StringBuffer("Hello");
5. sb.delete(1,3);
6.
7. System.out.println(sb);//prints Hlo
8. }
9. }
```

Example of reverse() method of StringBuffer class

The reverse() method of StringBuilder class reverses the current string.

```
1. class A{
2. public static void main(String args[]){
3.
4. StringBuffer sb=new StringBuffer("Hello");
5. sb.reverse();
6.
7. System.out.println(sb);//prints olleH
8. }
9. }
```

Example of capacity() method of StringBuffer class

The capacity() method of StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by $(\text{oldcapacity} * 2) + 2$. For example if your current capacity is 16, it will be $(16 * 2) + 2 = 34$.

```
1. class A{
2. public static void main(String args[]){
3.
4. StringBuffer sb=new StringBuffer();
5. System.out.println(sb.capacity());//default 16
6.
7. sb.append("Hello");
8. System.out.println(sb.capacity());//now 16
9.
10. sb.append("java is my favourite language");
11. System.out.println(sb.capacity());//now  $(16 * 2) + 2 = 34$  i.e  $(\text{oldcapacity} * 2) + 2$ 
12. }
13. }
```

Core Java

Example of ensureCapacity() method of StringBuffer class

The ensureCapacity() method of StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by $(\text{oldcapacity} \times 2) + 2$. For example if your current capacity is 16, it will be $(16 \times 2) + 2 = 34$.

```
1. class A{
2.     public static void main(String args[]){
3.
4.         StringBuffer sb=new StringBuffer();
5.         System.out.println(sb.capacity()); //default 16
6.
7.         sb.append("Hello");
8.         System.out.println(sb.capacity()); //now 16
9.
10.        sb.append("java is my favourite language");
11.       System.out.println(sb.capacity()); //now  $(16 \times 2) + 2 = 34$  i.e  $(\text{oldcapacity} \times 2) + 2$ 
12.
13.        sb.ensureCapacity(10); //now no change
14.       System.out.println(sb.capacity()); //now 34
15.
16.        sb.ensureCapacity(50); //now  $(34 \times 2) + 2$ 
17.       System.out.println(sb.capacity()); //now 70
18.
19.    }
20. }
```

StringBuilder class:

The StringBuilder class is used to create mutable (modifiable) string. The StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK1.5.

Commonly used Constructors of StringBuilder class:

1. **StringBuilder():** creates an empty string Builder with the initial capacity of 16.
2. **StringBuilder(String str):** creates a string Builder with the specified string.
3. **StringBuilder(int length):** creates an empty string Builder with the specified capacity as length.

Core Java

Commonly used methods of StringBuilder class:

1. **public StringBuilder append(String s):** is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
2. **public StringBuilder insert(int offset, String s):** is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
3. **public StringBuilder replace(int startIndex, int endIndex, String str):** is used to replace the string from specified startIndex and endIndex.
4. **public StringBuilder delete(int startIndex, int endIndex):** is used to delete the string from specified startIndex and endIndex.
5. **public StringBuilder reverse():** is used to reverse the string.
6. **public int capacity():** is used to return the current capacity.
7. **public void ensureCapacity(int minimumCapacity):** is used to ensure the capacity at least equal to the given minimum.
8. **public char charAt(int index):** is used to return the character at the specified position.
9. **public int length():** is used to return the length of the string i.e. total number of characters.
10. **public String substring(int beginIndex):** is used to return the substring from the specified beginIndex.
11. **public String substring(int beginIndex, int endIndex):** is used to return the substring from the specified beginIndex and endIndex.

simple program of StringBuilder class by append() method

The append() method concatenates the given argument with this string.

```
1. class A{  
2. public static void main(String args[]){  
3.  
4. StringBuilder sb=new StringBuilder("Hello ");  
5. sb.append("Java");//now original string is changed  
6.  
7. System.out.println(sb);//prints Hello Java  
8. }  
9. }
```

Example of insert() method of StringBuilder class

The insert() method inserts the given string with this string at the given position.

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

Core Java

```
3.  
4. StringBuilder sb=new StringBuilder("Hello ");  
5. sb.insert(1,"Java");//now original string is changed  
6.  
7. System.out.println(sb);//prints HJavaello  
8. }  
9. }
```

Example of replace() method of StringBuilder class

The replace() method replaces the given string from the specified beginIndex and endIndex.

```
1. class A{  
2. public static void main(String args[]){  
3.  
4. StringBuilder sb=new StringBuilder("Hello ");  
5. sb.replace(1,3,"Java");  
6.  
7. System.out.println(sb);//prints HJavaello  
8. }  
9. }
```

Example of delete() method of StringBuilder class

The delete() method of StringBuilder class deletes the string from the specified beginIndex to endIndex.

```
1. class A{  
2. public static void main(String args[]){  
3.  
4. StringBuilder sb=new StringBuilder("Hello ");  
5. sb.delete(1,3);  
6.  
7. System.out.println(sb);//prints Hlo  
8. }  
9. }
```

Example of reverse() method of StringBuilder class

The reverse() method of StringBuilder class reverses the current string.

```
1. class A{  
2. public static void main(String args[]){  
3.  
4. StringBuilder sb=new StringBuilder("Hello ");  
5. sb.reverse();  
6.  
7. System.out.println(sb);//prints olleH  
8. }  
9. }
```

Core Java

Example of capacity() method of StringBuilder class

The capacity() method of StringBuilder class returns the current capacity of the Builder.

The default capacity of the Builder is 16. If the number of character increases from its current capacity, it increases the capacity by $(\text{oldcapacity} * 2) + 2$. For example if your current capacity is 16, it will be $(16 * 2) + 2 = 34$.

```
1. class A{
2. public static void main(String args[]){
3.
4.     StringBuilder sb=new StringBuilder();
5.     System.out.println(sb.capacity()); //default 16
6.
7.     sb.append("Hello");
8.     System.out.println(sb.capacity()); //now 16
9.
10.    sb.append("java is my favourite language");
11.    System.out.println(sb.capacity()); //now  $(16 * 2) + 2 = 34$  i.e  $(\text{oldcapacity} * 2) + 2$ 
12. }
13. }
```

Example of ensureCapacity() method of StringBuilder class

The ensureCapacity() method of StringBuilder class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by $(\text{oldcapacity} * 2) + 2$. For example if your current capacity is 16, it will be $(16 * 2) + 2 = 34$.

```
1. class A{
2. public static void main(String args[]){
3.
4.     StringBuilder sb=new StringBuilder();
5.     System.out.println(sb.capacity()); //default 16
6.
7.     sb.append("Hello");
8.     System.out.println(sb.capacity()); //now 16
9.
10.    sb.append("java is my favourite language");
11.    System.out.println(sb.capacity()); //now  $(16 * 2) + 2 = 34$  i.e  $(\text{oldcapacity} * 2) + 2$ 
12.
13.    sb.ensureCapacity(10); //now no change
14.    System.out.println(sb.capacity()); //now 34
15.
16.    sb.ensureCapacity(50); //now  $(34 * 2) + 2$ 
17.    System.out.println(sb.capacity()); //now 70
18.
19. }
20. }
```

How to create Immutable class?

There are many immutable classes like String, Boolean, Byte, Short, Integer, Long, Float, Double etc. In short, all the wrapper classes and String class is immutable. We can also create immutable class by creating final class that have final data members as the example given below:

Example to create Immutable class

In this example, we have created a final class named Employee. It have one final datamember, a parameterized constructor and getter method.

```
1. public final class Employee{  
2.     final String pancardNumber;  
3.  
4.     public Employee(String pancardNumber){  
5.         this.pancardNumber=pancardNumber;  
6.     }  
7.  
8.     public String getPancardNumber(){  
9.         return pancardNumber;  
10.    }  
11.  
12. }
```

The above class is immutable because:

- The instance variable of the class is final i.e. we cannot change the value of it after creating an object.
- The class is final so we cannot create the subclass.
- There is no setter methods i.e. we have no option to change the value of the instance variable.

These points makes this class as immutable.

Understanding **toString()** method

If you want to represent any object as a string, **toString() method** comes into existence.

The **toString()** method returns the string representation of the object.

If you print any object, java compiler internally invokes the **toString()** method on the object. So overriding the **toString()** method, returns the desired output, it can be the state of an object etc. depends on your implementation.

Core Java

Advantage of the `toString()` method

By overriding the `toString()` method of the `Object` class, we can return values of the object, so we don't need to write much code.

Understanding problem without `toString()` method

Let's see the simple code that prints reference.

```
1. class Student{  
2.     int rollno;  
3.     String name;  
4.     String city;  
5.  
6.     Student(int rollno, String name, String city){  
7.         this.rollno=rollno;  
8.         this.name=name;  
9.         this.city=city;  
10.    }  
11.  
12.   public static void main(String args[]){  
13.       Student s1=new Student(101,"Raj","lucknow");  
14.       Student s2=new Student(102,"Vijay","ghaziabad");  
15.  
16.       System.out.println(s1); //compiler writes here s1.toString()  
17.       System.out.println(s2); //compiler writes here s2.toString()  
18.   }  
19. }
```

Output:Student@1fee6fc

Student@1eed786

As you can see in the above example, printing `s1` and `s2` prints the hashcode values of the objects but I want to print the values of these objects. Since java compiler internally calls `toString()` method, overriding this method will return the specified values. Let's understand it with the example given below:

Example of `toString()` method

Now let's see the real example of `toString()` method.

```
1. class Student{  
2.     int rollno;  
3.     String name;  
4.     String city;  
5.  
6.     Student(int rollno, String name, String city){
```

Core Java

```
7. this.rollno=rollno;
8. this.name=name;
9. this.city=city;
10. }
11.
12. public String toString(){//overriding the toString() method
13.     return rollno+" "+name+" "+city;
14. }
15. public static void main(String args[]){
16.     Student s1=new Student(101,"Raj","lucknow");
17.     Student s2=new Student(102,"Vijay","ghaziabad");
18.
19.     System.out.println(s1);//compiler writes here s1.toString()
20.     System.out.println(s2);//compiler writes here s2.toString()
21. }
22. }
```

Output:101 Raj lucknow
102 Vijay ghaziabad

StringTokenizer in Java

The **java.util.StringTokenizer** class allows you to break a string into tokens. It is simple way to break string.

It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc. like StreamTokenizer class. We will discuss about the StreamTokenizer class in I/O chapter.

Constructors of StringTokenizer class

There are 3 constructors defined in the StringTokenizer class.

Constructor	Description
StringTokenizer(String str)	creates StringTokenizer with specified string.
StringTokenizer(String str, String delim)	creates StringTokenizer with specified string and delimiter.
StringTokenizer(String str, String delim, boolean returnValue)	creates StringTokenizer with specified string, delimiter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

Core Java

Methods of StringTokenizer class

The 6 useful methods of StringTokenizer class are as follows:

Public method	Description
boolean hasMoreTokens()	checks if there is more tokens available.
String nextToken()	returns the next token from the StringTokenizer object.
String nextToken(String delim)	returns the next token based on the delimiter.
boolean hasMoreElements()	same as hasMoreTokens() method.
Object nextElement()	same as nextToken() but its return type is Object.
int countTokens()	returns the total number of tokens.

Simple example of StringTokenizer class

Let's see the simple example of StringTokenizer class that tokenizes a string "my name is khan" on the basis of whitespace.

```
1. import java.util.StringTokenizer;
2. public class Simple{
3.     public static void main(String args[]){
4.         StringTokenizer st = new StringTokenizer("my name is khan","");
5.         while (st.hasMoreTokens()) {
6.             System.out.println(st.nextToken());
7.         }
8.     }
9. }
```

Output:
my
name
is
khan

Core Java

Example of nextToken(String delim) method of StringTokenizer class

```
1. import java.util.*;
2.
3. public class Test {
4.     public static void main(String[] args) {
5.         StringTokenizer st = new StringTokenizer("my,name,is,khan");
6.
7.         // printing next token
8.         System.out.println("Next token is : " + st.nextToken(","));
9.     }
10. }
```

Output:Next token is : my

Exception Handling in Java

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

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

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.

Exception Handling

Exception Handling is a mechanism to handle runtime errors.

Advantage of Exception Handling

The core advantage of exception handling is that normal flow of the application is maintained. 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;
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

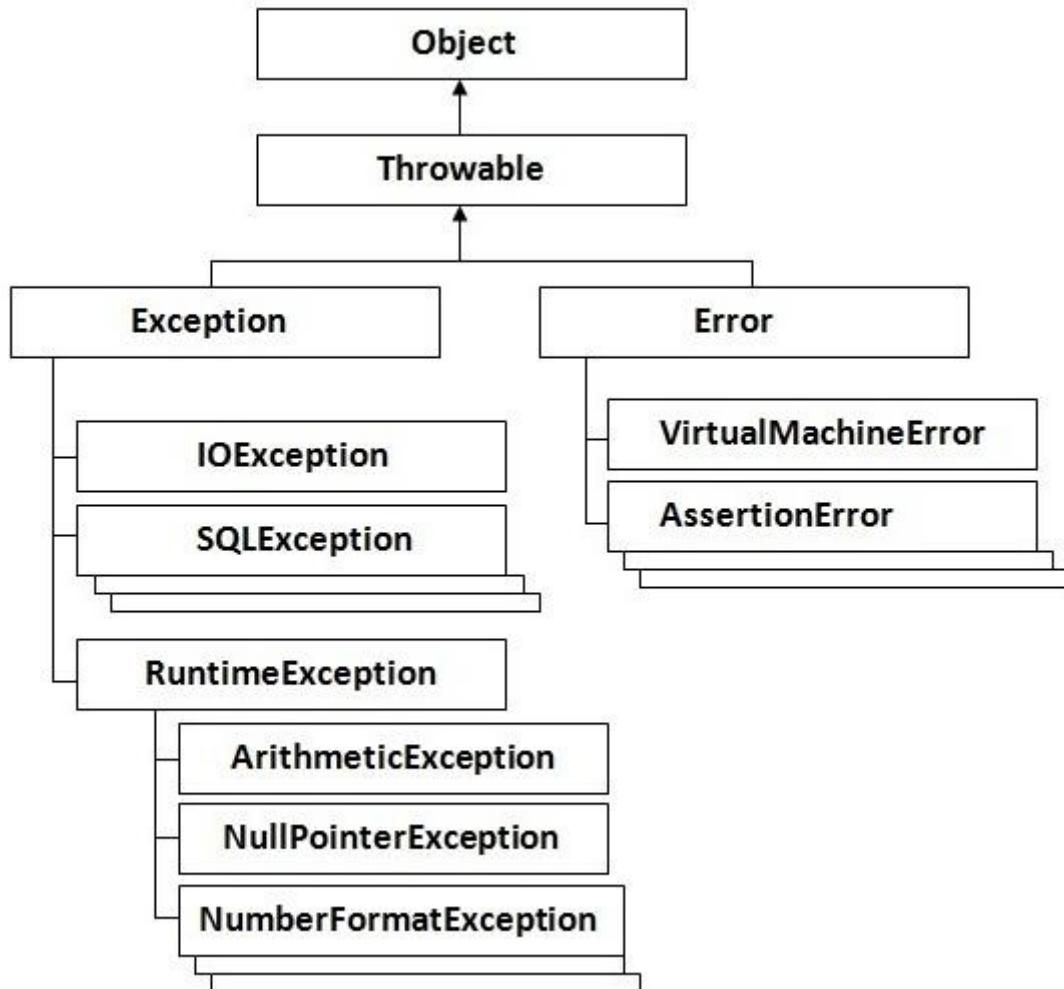
Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the exception will be executed. That is why we use exception handling.

- ***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 ?***

Core Java

- *What are the 4 rules for using exception handling with method overriding ?*

Hierarchy of Exception classes



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

Core Java

What is the 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 of Exception Handling 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;//ArithmaticException`

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

Core Java

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[]={};
2. a[10]=50; //ArrayIndexOutOfBoundsException

Use of try-catch block in Exception handling:

Five keywords used in Exception handling:

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

try block

Enclose the code that might throw an exception in try block. It must be used within the method and must be followed by either catch or finally block.

Syntax of try with catch block

1. try{
2. ...
3. }catch(Exception_class_Name reference){ }

Syntax of try with finally block

1. try{
2. ...
3. }finally{ }

Core Java

catch block

Catch block is used to handle the Exception. It must be used after the try block.

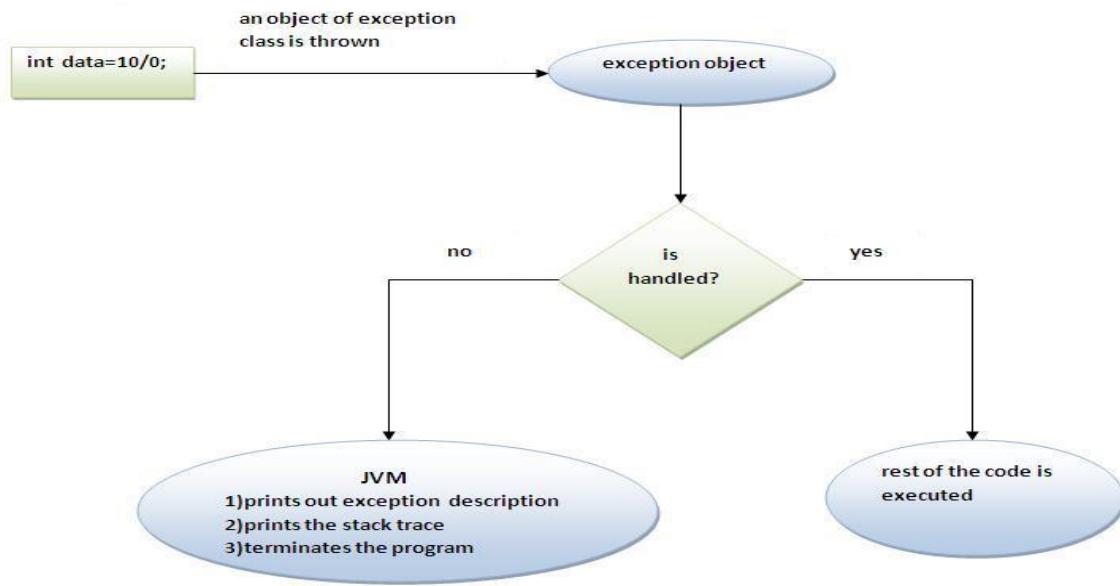
Problem without exception handling

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

Output:Exception in thread main java.lang.ArithmaticException:/ by zero

As displayed in the above example, rest of the code is not executed i.e. rest of the code... statement is not printed. Let's see what happens behind the scene:

What happens behind the code int a=50/0;



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.

Core Java

Solution by exception handling

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

```
9. }
```

```
10. }
```

Output:Exception in thread main java.lang.ArithmaticException:/ 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.

Multiple catch block:

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

```
1. <b><i>Example of multiple <b>catch</b> block</i></b>
2.
3. class Excep4{
4.   public static void main(String args[]){
5.     try{
6.       int a[] = new int[5];
7.       a[5] = 30/0;
8.     }
9.     catch(ArithmaticException e){System.out.println("task1 is completed");}
10.    catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
11.    catch(Exception e){System.out.println("common task completed");}
12.
13.    System.out.println("rest of the code...");
```

```
14. }
```

```
15. }
```

Output:task1 completed
rest of the code...

Rule:At a time only one Exception is occurred 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 ArithmaticException must come before catch for Exception .

```
1. class Excep4{
2.   public static void main(String args[]){
3.     try{
4.       int a[] = new int[5];
```

Core Java

```
5.     a[5]=30/0;
6.   }
7.   catch(Exception e){System.out.println("common task completed");}
8.   catch(ArithmaticException e){System.out.println("task1 is completed");}
9.   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
10.
11. System.out.println("rest of the code...");
```

```
12. }
13. }
```

Output:Compile-time error

Nested try block:

try block within a try block is known as nested try block.

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:

```
1. ....
2. try
3. {
4.   statement 1;
5.   statement 2;
6.   try
7.   {
8.     statement 1;
9.     statement 2;
10.  }
11.  catch(Exception e)
12.  {
13.  }
14. }
15. catch(Exception e)
16. {
17. }
18. ....
```

Example:

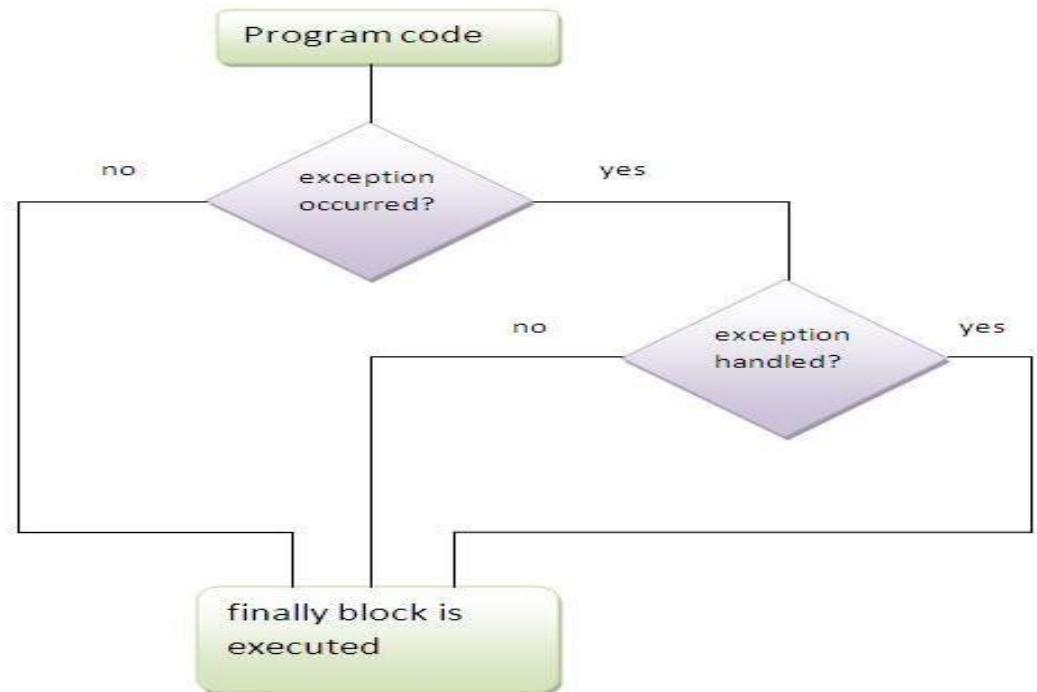
```
1. <b><i>Example of nested try block</i></b>
2.
3. class Excep6{
```

Core Java

```
4. public static void main(String args[]){
5.     try{
6.         try{
7.             System.out.println("going to divide");
8.             int b =39/0;
9.         }catch(ArithmeticException e){System.out.println(e);}
10.
11.        try{
12.            int a[] =new int[5];
13.            a[5]=4;
14.        }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}
15.
16.        System.out.println("other statement");
17.    }catch(Exception e){System.out.println("handled");}
18.
19.    System.out.println("normal flow..");
20. }
21. }
```

finally block

The finally block is a block that is always executed. It is mainly used to perform some important tasks such as closing connection, stream etc.



Note: Before terminating the program, JVM executes finally block(if any).

Note: finally must be followed by try or catch block.

Core Java

Why use finally block?

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

case 1

Program in case exception does not occur

```
1. class Simple{  
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.  
9.         finally{System.out.println("finally block is always executed");}  
10.  
11.        System.out.println("rest of the code...");  
12.    }  
13. }
```

Output:5

finally block is always executed
rest of the code...

case 2

Program in case exception occurred but not handled

```
1. class Simple{  
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.  
9.         finally{System.out.println("finally block is always executed");}  
10.  
11.        System.out.println("rest of the code...");  
12.    }  
13. }
```

Output:finally block is always executed

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

case 3

Program in case exception occurred and handled

```
1. class Simple{
```

Core Java

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

Output:Exception in thread main java.lang.ArithmaticException:/ by zero
finally block is always executed
rest of the code...

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

throw keyword

The throw keyword is used to explicitly throw an exception.

We can throw either checked or unchecked exception. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

Example of throw keyword

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 ArithmaticException otherwise print a message welcome to vote.

```
1. class Excep13{
2.
3.     static void validate(int age){
4.         if(age<18)
5.             throw new ArithmaticException("not valid");
6.         else
7.             System.out.println("welcome to vote");
8.     }
9.
10.    public static void main(String args[]){
11.        validate(13);
12.        System.out.println("rest of the code...");
```

Core Java

```
13. }
14. }
```

Output:Exception in thread main java.lang.ArithmaticException:not valid

Exception propagation:

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method. If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

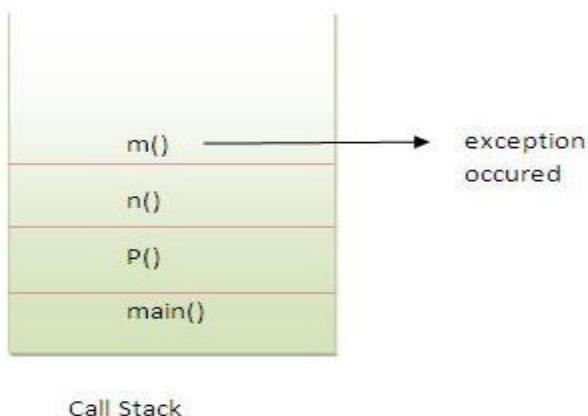
Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).

Program of Exception Propagation

```
1. class Simple{
2.     void m(){
3.         int data=50/0;
4.     }
5.     void n(){
6.         m();
7.     }
8.     void p(){
9.         try{
10.             n();
11.         }catch(Exception e){System.out.println("exception handled");}
12.     }
13.     public static void main(String args[]){
14.         Simple obj=new Simple();
15.         obj.p();
16.         System.out.println("normal flow...");
17.     }
18. }
```

Output:exception handled

normal flow...



Core Java

In the above example exception occurs in m() method where it is not handled, so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled.

Exception can be handled in any method in call stack either in main() method, p() method, n() method or m() method.

Rule: By default, Checked Exceptions are not forwarded in calling chain (propagated).

Program which describes that checked exceptions are not propagated

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

Output: Compile Time Error

throws keyword

The **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 throws keyword:

```
1. void method_name() throws exception_class_name{  
2.     ...  
3. }
```

Core Java

Que) Which exception should we declare?

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 throws keyword:

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

Program which describes that checked exceptions can be propagated by throws keyword.

```
1. import java.io.IOException;
2. class Simple{
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.        Simple obj=new Simple();
16.        obj.p();
17.        System.out.println("normal flow...");
18.    }
19. }
```

Output:exception handled

normal flow...

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.

Core Java

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.
8.
9. class Test{
10.    public static void main(String args[]){
11.        try{
12.            Test t=new Test();
13.            t.method();
14.        }catch(Exception e){System.out.println("exception handled");}
15.
16.        System.out.println("normal flow...");
```

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 occurs, 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.
8.
9. class Test{
10.    public static void main(String args[])throws IOException{//declare exception
11.        Test t=new Test();
12.        t.method();
13.
14.        System.out.println("normal flow...");
```

Core Java

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.
8.
9. class Test{
10.    public static void main(String args[])throws IOException{//declare exception
11.        Test t=new Test();
12.        t.method();
13.
14.        System.out.println("normal flow...");
15.    }
16. }
```

Output:Runtime Exception

Difference between throw and throws:

throw keyword	throws keyword
1)throw is used to explicitly throw an exception.	throws is used to declare an exception.
2)checked exception can not be propagated without throws.	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 exception	You can declare multiple exception e.g. public void method()throws IOException,SQLException.

Que) Can we rethrow an exception?

Yes by throwing same exception in catch block.

ExceptionHandling with MethodOverriding

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.  
6. class Child extends Parent{  
7.     void msg()throws IOException{  
8.         System.out.println("child");  
9.     }  
10.    public static void main(String args[]){  
11.        Parent p=new Child();  
12.        p.msg();  
13.    }  
14. }
```

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.  
6. class Child extends Parent{  
7.     void msg()throws ArithmeticException{  
8.         System.out.println("child");  
9.     }  
10.    public static void main(String args[]){
```

Core Java

```
11. Parent p=new Child();  
12. p.msg();  
13. }  
14. }
```

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.  
6. class Child extends Parent{  
7.     void msg()throws Exception{System.out.println("child");}  
8.  
9. public static void main(String args[]){  
10. Parent p=new Child();  
11. try{  
12.     p.msg();  
13. }catch(Exception e){}  
14. }  
15. }
```

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

Core Java

```
14. }
15. }
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.
6. class Child extends Parent{
7.     void msg()throws ArithmeticException{System.out.println("child");}
8.
9.     public static void main(String args[]){
10.     Parent p=new Child();
11.     try{
12.         p.msg();
13.     }catch(Exception e){}
14. }
15. }
```

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.
6. class Child extends Parent{
7.     void msg(){System.out.println("child");}
8.
9.     public static void main(String args[]){
10.     Parent p=new Child();
11.     try{
12.         p.msg();
13.     }catch(Exception e){}
14. }
15. }
```

Output:child

Custom Exception

If you are creating your own Exception that is known as custom exception or user-defined exception.

```
1. class InvalidAgeException extends Exception{
2.     InvalidAgeException(String s){
```

Core Java

```
3.     super(s);
4. }
5. }

1. class Excep13{
2.
3.     static void validate(int age) throws InvalidAgeException{
4.         if(age<18)
5.             throw new InvalidAgeException("not valid");
6.         else
7.             System.out.println("welcome to vote");
8.     }
9.

10.    public static void main(String args[]){
11.        try{
12.            validate(13);
13.        }catch(Exception m){System.out.println("Exception occurred: "+m);}
14.
15.        System.out.println("rest of the code...");
16.    }
17. }
```

Output:Exception occurred: InvalidAgeException:not valid
rest of the code...

Nested classes in Java

Nested classes in Java

A class declared inside a class is known as nested class. We use nested classes to logically group classes in one place so that it can be more readable and maintainable code. Moreover, it can access all the members of outer class including private members.

Syntax of Nested class

```
1. class Outer_class_Name{  
2. ...  
3. class Nested_class_Name{  
4. ...  
5. }  
6. ...  
7. }
```

Advantage of nested classes

There are basically three advantages of nested classes. They are

- Nested classes represent a special type of relationship that is it can access all the members (data members and methods) of outer class including private.
- Nested classes can lead to more readable and maintainable code because it logically group classes in one place only.
- Code Optimization as we need less code to write.

Do You Know ?

- *What is the internal code generated by the compiler for member inner class ?*
- *What are the two ways to create anonymous inner class ?*
- *Can we access the non-final local variable inside the local inner class ?*
- *How to access the static nested class ?*
- *Can we define an interface within the class ?*
- *Can we define a class within the interface ?*

Que) What is the difference between nested class and inner class?

Inner class is a part of nested class. Non-static nested classes are known as nested classes.

Core Java

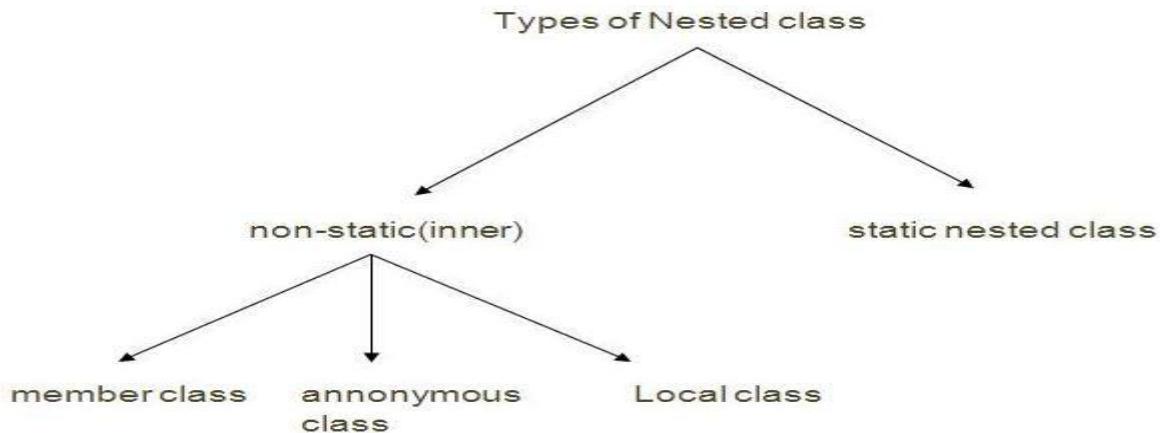
Types of Nested class:

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

1. non-static nested class(inner class)

- o a)Member inner class
- o b)Anonymous inner class
- o c)Local inner class

2. static nested class



What we will learn in Nested classes ?

- Nested class
- Member inner class
- Anonymous inner class
- Local Inner class
- static nested class
- Nested interface

1)Member inner class

A class that is declared inside a class but outside a method is known as member inner class.

Invocation of Member Inner class

- 1. From within the class
- 2. From outside the class

Core Java

Example of member inner class that is invoked inside a class

In this example, we are invoking the method of member inner class from the display method of Outer class.

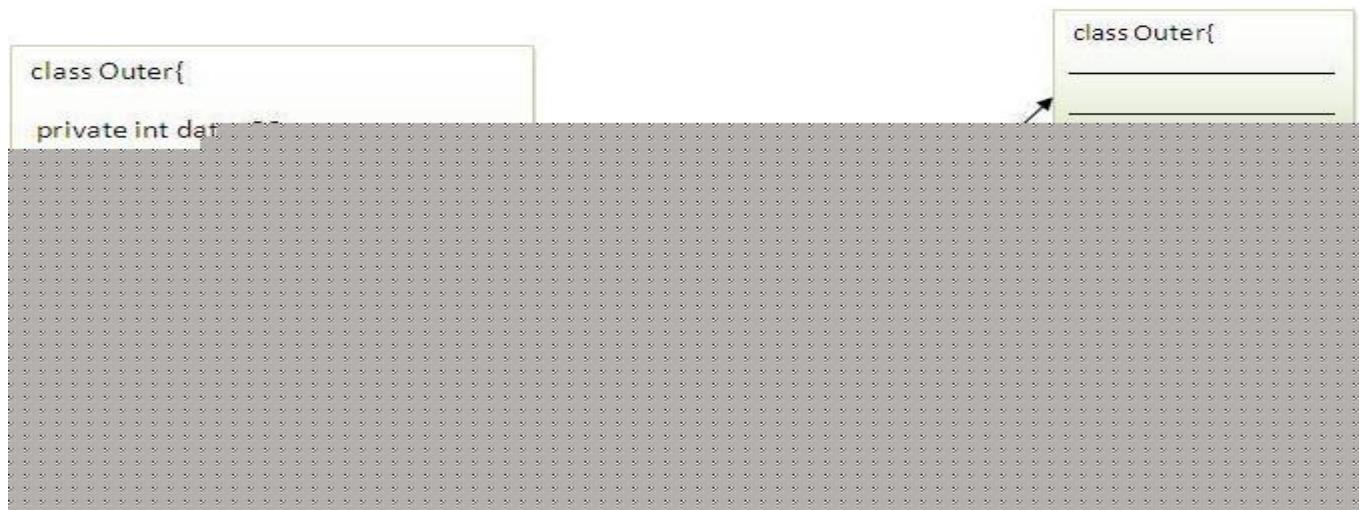
```
1. class Outer{  
2.     private int data=30;  
3.     class Inner{  
4.         void msg(){System.out.println("data is "+data);}  
5.     }  
6.  
7.     void display(){  
8.         Inner in=new Inner();  
9.         in.msg();  
10.    }  
11.   public static void main(String args[]){  
12.       Outer obj=new Outer();  
13.       obj.display();  
14.   }  
15. }
```

Output:data is 30

Internal code generated by the compiler for member inner class:

The java compiler creates a class file named Outer\$Inner in this case. The Member inner class have the reference of Outer class that is why it can access all the data members of Outer class including private.

```
1. import java.io.PrintStream;  
2.  
3. class Outer$Inner  
4. {  
5.     final Outer this$0;  
6.     Outer$Inner()  
7.     { super();  
8.         this$0 = Outer.this;  
9.     }  
10.  
11.    void msg()  
12.    {  
13.        System.out.println((new StringBuilder()).append("data is ")  
14.                      .append(Outer.access$000(Outer.this)).toString());  
15.    }  
16.  
17. }
```



Example of member inner class that is invoked outside a class

In this example, we are invoking the msg() method of Inner class from outside the outer class i.e. Test class.

1. <i>//Program of member inner class that is invoked outside a class</i>
- 2.
3. **class** Outer{
4. **private int** data=30;
5. **class** Inner{
6. **void** msg(){System.out.println("data is"+data);}
7. }
8. }
- 9.
10. **class** Test{
11. **public static void** main(String args[]){
12. Outer obj=**new** Outer();
13. Outer.Inner in=obj.**new** Inner();
14. in.msg();
15. }
16. }

Output:data is 30

2)Anonymous inner class

A class that have no name is known as anonymous inner class.

Anonymous class can be created by:

1. Class (may be abstract class also).
2. Interface

Core Java

Program of anonymous inner class by abstract class

```
1. abstract class Person{  
2.   abstract void eat();  
3. }  
4.  
5. class Emp{  
6.   public static void main(String args[]){  
7.     Person p=new Person(){  
8.       void eat(){System.out.println("nice fruits");}  
9.     };  
10.  
11.   p.eat();  
12. }  
13. }
```

Output:nice fruits

What happens behind this code?

```
1. Person p=new Person(){  
2.   void eat(){System.out.println("nice fruits");}  
3. }  
4.  
5. }
```

1. A class is created but its name is decided by the compiler which extends the Person class and provides the implementation of the eat() method.
2. An object of Anonymous class is created that is referred by p reference variable of Person type. As you know well that Parent class reference variable can refer the object of Child class.

The internal code generated by the compiler for anonymous inner class

```
1. import java.io.PrintStream;  
2. static class Emp$1 extends Person  
3. {  
4.   Emp$1(){  
5.     void eat()  
6.     {  
7.       System.out.println("nice fruits");  
8.     }  
9.   }  
10. }
```

Program of anonymous inner class by interface

```
1. interface Eatable{  
2.   void eat();
```

Core Java

```
3. }
4.
5. class Emp{
6.     public static void main(String args[]){
7.
8.     Eatable e=new Eatable(){
9.         public void eat(){System.out.println("nice fruits");}
10.    };
11.    e.eat();
12. }
13. }
```

Output:nice fruits

What does the compiler for anonymous inner class created by interface

It performs two main tasks behind this code:

```
1. Eatable p=new Eatable(){
2.     void eat(){System.out.println("nice fruits");}
3. };
4.
5. }
6. }
```

1. A class is created but its name is decided by the compiler which implements the Eatable interface and provides the implementation of the eat() method.
2. An object of Anonymous class is created that is referred by p reference variable of Eatable type. As you know well that Parent class reference variable can refer the object of Child class.

The internal code generated by the compiler for anonymous inner class created by interface

```
1. import java.io.PrintStream;
2. static class Emp$1 implements Eatable
3. {
4.     Emp$1(){}
5.
6.     void eat(){System.out.println("nice fruits");}
7. }
```

3)Local inner class

A class that is created inside a method is known as local inner class. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

Program of local inner class

```
1. class Simple{
2.     private int data=30;//instance variable
```

Core Java

```
3. void display(){  
4. class Local{  
5.     void msg(){System.out.println(data);}  
6. }  
7. Local l=new Local();  
8. l.msg();  
9. }  
10. public static void main(String args[]){  
11. Simple obj=new Simple();  
12. obj.display();  
13. }  
14. }
```

Output:30

Internal code generated by the compiler for local inner class

In such case, compiler creates a class named Simple\$1Local that have the reference of the outer class.

```
1. import java.io.PrintStream;  
2. class Simple$1Local  
3. {  
4.     final Simple this$0;  
5.  
6.     Simple$1Local()  
7.     {  
8.         super();  
9.         this$0 = Simple.this;  
10.    }  
11.    void msg()  
12.    {  
13.        System.out.println(Simple.access$000(Simple.this));  
14.    }  
15.  
16. }
```

Rule: Local variable can't be private, public or protected.

Rules for Local Inner class

1) Local inner class cannot be invoked from outside the method.

2) Local inner class cannot access non-final local variable.

Program of accessing non-final local variable in local inner class

```
1. class Simple{  
2.     private int data=30;//instance variable  
3.     void display(){  
4.         int value=50;//local variable must be final
```

Core Java

```
5. class Local{  
6.     void msg(){System.out.println(value);}//C.T.Error  
7. }  
8. Local l=new Local();  
9. l.msg();  
10. }  
11. public static void main(String args[]){  
12.     Simple obj=new Simple();  
13.     obj.display();  
14. }  
15. }
```

Output:Compile Time Error

Program of accessing final local variable in local inner class

```
1. class Simple{  
2.     private int data=30;//instance variable  
3.     void display(){  
4.         final int value=50;//local variable must be final  
5.         class Local{  
6.             void msg(){System.out.println(data+" "+value);}//ok  
7.         }  
8.         Local l=new Local();  
9.         l.msg();  
10.    }  
11.    public static void main(String args[]){  
12.        Simple obj=new Simple();  
13.        obj.display();  
14.    }  
15. }
```

Output:30 50

4)static nested class

A static class that is created inside a class is known as static nested class. It cannot access the non-static members.

- It can access static data members of outer class including private.
- static nested class cannot access non-static (instance) data member or method.

Program of static nested class that have instance method

```
1. class Outer{  
2.     static int data=30;  
3.     static class Inner{  
4.         void msg(){System.out.println("data is "+data);}  
5.     }  
6.     public static void main(String args[]){  
7. }
```

Core Java

```
9. Outer.Inner obj=new Outer.Inner();
10. obj.msg();
11. }
12. }
```

Output:data is 30

In this example, you need to create the instance of static nested class because it has instance method msg(). But you don't need to create the object of Outer class because nested class is static and static properties, methods or classes can be accessed without object.

Internal code generated by the compiler for static nested class

```
1. import java.io.PrintStream;
2.
3. static class Outer$Inner
4. {
5. Outer$Inner(){}
6.
7. void msg(){
8. System.out.println(new StringBuilder().append("data is "))
9. .append(Outer.data).toString());
10. }
11.
12. }
```

Program of static nested class that have static method

```
1. class Outer{
2. static int data=30;
3.
4. static class Inner{
5. static void msg(){System.out.println("data is "+data);}
6. }
7.
8. public static void main(String args[]){
9. Outer.Inner.msg();//no need to create the instance of static nested class
10. }
11. }
```

Output:data is 30

Nested Interface

An interface which is declared within another interface or class is known as nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred by the outer interface or class. It can't be accessed directly.

Core Java

Points to remember for nested interfaces

There are given some points that should be remembered by the java programmer.

- Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.
- Nested interfaces are declared static implicitly.

Syntax of nested interface which is declared within the interface

```
1. interface interface_name{  
2. ...  
3. interface nested_interface_name{  
4. ...  
5. }  
6. }  
7.
```

Syntax of nested interface which is declared within the class

```
1. class class_name{  
2. ...  
3. interface nested_interface_name{  
4. ...  
5. }  
6. }  
7.
```

Example of nested interface which is declared within the interface

In this example, we are going to learn how to declare the nested interface and how we can access it.

```
1. interface Showable{  
2.   void show();  
3.   interface Message{  
4.     void msg();  
5.   }  
6. }  
7.  
8. class Test implements Showable.Message{  
9.   public void msg(){System.out.println("Hello nested interface");}  
10.  
11. public static void main(String args[]){  
12.   Showable.Message message=new Test();//upcasting here  
13.   message.msg();  
14. }  
15. }
```

Output:hello nested interface

As you can see in the above example, we are accessing the Message interface by its outer

Core Java

interface Showable because it cannot be accessed directly. It is just like almirah inside the room, we cannot access the almirah directly because we must enter the room first. In collection framework, sun microsystem has provided a nested interface Entry. Entry is the subinterface of Map i.e. accessed by Map.Entry.

Internal code generated by the java compiler for nested interface Message

The java compiler internally creates public and static interface as displayed below:.

```
1. public static interface Showable$Message
2. {
3.   public abstract void msg();
4. }
```

Example of nested interface which is declared within the class

Let's see how can we define an interface inside the class and how can we access it.

```
1. class A{
2.   interface Message{
3.     void msg();
4.   }
5. }
6.
7. class Test implements A.Message{
8.   public void msg(){System.out.println("Hello nested interface");}
9.
10. public static void main(String args[]){
11.   A.Message message=new Test();//upcasting here
12.   message.msg();
13. }
14. }
```

Output:hello nested interface

Can we define a class inside the interface ?

Yes, Ofcourse! If we define a class inside the interface, java compiler creates a static nested class. Let's see how can we define a class within the interface:

```
1. interface M{
2.   class A{}
3. }
```

Multithreading in Java

Multithreading is a process of executing multiple threads simultaneously.

Thread is basically a lightweight subprocess, a smallest unit of processing.

Multiprocessing and multithreading, both are used to achieve multitasking. But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so save memory, and context-switching between the threads takes less time than processes.

Multithreading is mostly used in games, animation etc.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

- Process-based Multitasking(Multiprocessing)
- Thread-based Multitasking(Multithreading)

1)Process-based Multitasking (Multiprocessing)

- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

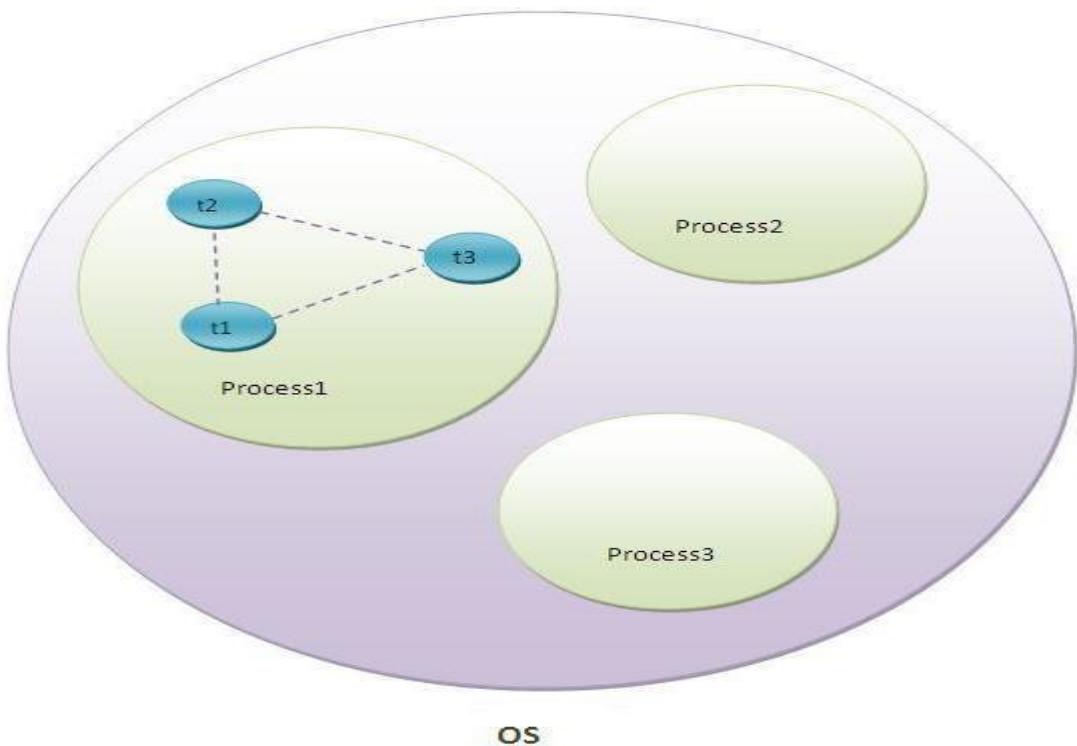
2)Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the thread is low.
- **Note:**At least one process is required for each thread.

What is Thread?

A thread is a lightweight subprocess, a smallest unit of processing. It is a separate path of execution. It shares the memory area of process.

Core Java



As shown in the above figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

Note: At a time only one thread is executed.

What we will learn in Multithreading ?

- Multithreading
- Life Cycle of a Thread
- Two ways to create a Thread
- How to perform multiple tasks by multiple threads
- Thread Scheduler
- Sleeping a thread
- Can we start a thread twice ?
- What happens if we call the run() method instead of start() method ?
- Joining a thread
- Naming a thread
- Priority of a thread
- Daemon Thread
- ShutdownHook
- Garbage collection
- Synchronization with synchronized method
- Synchronized block

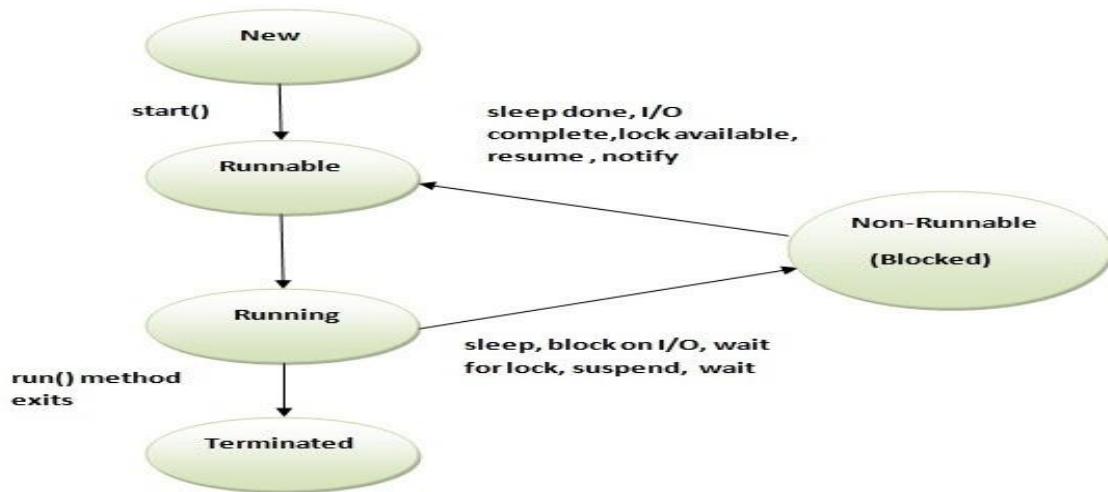
Core Java

- Static synchronization
- Deadlock
- Inter-thread communication

Life cycle of a Thread (Thread States)

A thread can be in one of the five states in the thread. According to sun, there is only 4 states new, runnable, non-runnable and terminated. There is no running state. But for better understanding the threads, we are explaining it in the 5 states. The life cycle of the thread is controlled by JVM. The thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



1)New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2)Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3)Running

The thread is in running state if the thread scheduler has selected it.

Core Java

4)Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5)Terminated

A thread is in terminated or dead state when its run() method exits.

How to create thread:

There are two ways to create a thread:

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

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

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 milliseconds.
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.

Core Java

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(deprecated).
16. **public void resume():** is used to resume the suspended thread(deprecated).
17. **public void stop():** is used to stop the thread(deprecated).
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:

start() method of Thread class is used to start a newly created thread. It performs 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)By extending Thread class:

```
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...

Core Java

Who makes your class object as thread object?

Thread class constructor allocates a new thread object. When you create object of Multi class, your class constructor is invoked (provided by Compiler) from where Thread class constructor is invoked (by super() as first statement). So your Multi class object is thread object now.

2) By implementing the Runnable interface:

```
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);  
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 Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

The Thread Scheduler:

- The thread scheduler is the part of the JVM that decides which thread should run.
- There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.
- Only one thread at a time can run in a single process.
- The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

What is the difference between preemptive scheduling and time slicing?

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

Sleeping a thread (sleep() method):

The sleep() method of Thread class is used to sleep a thread for the specified time. Syntax:

Syntax of sleep() method:

The Thread class provides two methods for sleeping a thread:

- public static void sleep(long milliseconds) throws InterruptedException
- public static void sleep(long milliseconds, int nanos) throws InterruptedException

```
1. //<b><i>Program of sleep() method</i></b>
2.
3. class Multi extends Thread{
4.     public void run(){
5.         for(int i=1;i<5;i++){
6.             try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
7.             System.out.println(i);
8.         }
9.     }
10.    public static void main(String args[]){
11.        Multi t1=new Multi();
12.        Multi t2=new Multi();
13.
14.        t1.start();
15.        t2.start();
16.    }
17. }
```

1. Output:

2. 1
3. 2
4. 2
5. 3
6. 3
7. 4
8. 4
9. 5
10. 5

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time, the thread scheduler picks up another thread and so on.

Can we start a thread twice?

No. After starting a thread, it can never be started again. If you do so, an IllegalThreadStateException is thrown. For Example:

```
1. class Multi extends Thread{
2.     public void run(){
```

Core Java

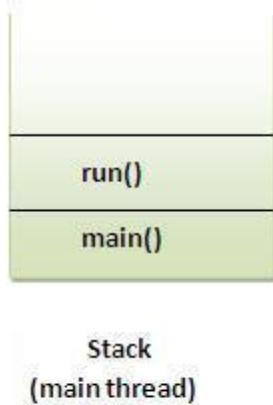
```
3.     System.out.println("running...");  
4. }  
5. public static void main(String args[]){  
6.     Multi t1=new Multi();  
7.     t1.start();  
8.     t1.start();  
9. }  
10.  
1. <strong>Output:</strong>running  
2.     Exception in thread "main" java.lang.IllegalThreadStateException
```

What if we call run() method directly instead start() method?

- Each thread starts in a separate call stack.
- Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

```
1. class Multi extends Thread{  
2.     public void run(){  
3.         System.out.println("running...");  
4.     }  
5.     public static void main(String args[]){  
6.         Multi t1=new Multi();  
7.         t1.run();//fine, but does not start a separate call stack  
8.     }  
9. }
```

1. Output:running...



```
1. //<b><i>Problem if you direct call run() method</i></b>  
2.  
3. class Multi extends Thread{  
4.     public void run(){  
5.         for(int i=1;i<5;i++){  
6.             try{Thread.sleep(500);} catch(InterruptedException e){System.out.println(e);}  
7.             System.out.println(i);
```

Core Java

```
8. }
9. }
10. public static void main(String args[]){
11.     Multi t1=new Multi();
12.     Multi t2=new Multi();
13.
14.     t1.run();
15.     t2.run();
16. }
17. }
```

Output:1

```
2
3
4
5
1
2
3
4
5
```

As you can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object.

The join() method:

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

Syntax:

```
public void join()throws InterruptedException
```

```
public void join(long miliseconds)throws InterruptedException
```

```
1. //<b><i>Example of join() method</i></b>
2.
3. class Multi extends Thread{
4.     public void run(){
5.         for(int i=1;i<=5;i++){
6.             try{
7.                 Thread.sleep(500);
8.             } catch(Exception e){System.out.println(e);}
9.             System.out.println(i);
10.        }
11.    }
12. public static void main(String args[]){
```

Core Java

```
13. Multi t1=new Multi();
14. Multi t2=new Multi();
15. Multi t3=new Multi();
16. t1.start();
17. try{
18.   t1.join();
19. }catch(Exception e){System.out.println(e);}
20.
21. t2.start();
22. t3.start();
23. }
24. }
```

Output:1

```
2
3
4
5
1
1
2
2
3
3
4
4
5
5
```

As you can see in the above example, when t1 completes its task then t2 and t3 starts executing.

```
1. //<b><i>Example of join(long miliseconds) method</i></b>
2.
3. class Multi extends Thread{
4.   public void run(){
5.     for(int i=1;i<=5;i++){
6.       try{
7.         Thread.sleep(500);
8.       }catch(Exception e){System.out.println(e);}
9.       System.out.println(i);
10.    }
11. }
12. public static void main(String args[]){
13.   Multi t1=new Multi();
14.   Multi t2=new Multi();
15.   Multi t3=new Multi();
16.   t1.start();
```

Core Java

```
17. try{  
18. t1.join(1500);  
19. }catch(Exception e){System.out.println(e);}  
20.  
21. t2.start();  
22. t3.start();  
23. }  
24. }
```

Output:1

```
2  
3  
1  
4  
1  
2  
5  
2  
3  
3  
4  
4  
5  
5
```

In the above example, when t1 is completes its task for 1500 miliseconds(3 times) then t2 and t3 starts executing.

getName(), setName(String) and getId() method:

```
public String getName()  
  
public void setName(String name)  
  
public long getId()
```

```
1. class Multi6 extends Thread{  
2.     public void run(){  
3.         System.out.println("running...");  
4.     }  
5.     public static void main(String args[]){  
6.         Multi6 t1=new Multi6();  
7.         Multi6 t2=new Multi6();  
8.         System.out.println("Name of t1:"+t1.getName());  
9.         System.out.println("Name of t2:"+t2.getName());  
10.        System.out.println("id of t1:"+t1.getId());  
11.    }  
12.    t1.start();
```

Core Java

```
13. t2.start();
14.
15. t1.setName("Sonoo Jaiswal");
16. System.out.println("After changing name of t1:"+t1.getName());
17. }
18. }
```

Output:Name of t1:Thread-0

```
    Name of t2:Thread-1
    id of t1:8
    running...
    After changling name of t1:Sonoo Jaiswal
    running...
```

The currentThread() method:

The currentThread() method returns a reference to the currently executing thread object.

Syntax:

```
public static Thread currentThread()
```

```
1. //<b><i>Example of currentThread() method</i></b>
2.
3. class Multi6 extends Thread{
4.     public void run(){
5.         System.out.println(Thread.currentThread().getName());
6.     }
7. }
8. public static void main(String args[]){
9.     Multi6 t1=new Multi6();
10.    Multi6 t2=new Multi6();
11.
12.    t1.start();
13.    t2.start();
14. }
15. }
```

Output:Thread-0

```
    Thread-1
```

Naming a thread:

The Thread class provides methods to change and get the name of a thread.

1. **public String getName():** is used to return the name of a thread.
2. **public void setName(String name):** is used to change the name of a thread.

Example of naming a thread:

```
1. class Multi6 extends Thread{  
2.     public void run(){  
3.         System.out.println("running...");  
4.     }  
5.     public static void main(String args[]){  
6.         Multi6 t1=new Multi6();  
7.         Multi6 t2=new Multi6();  
8.         System.out.println("Name of t1:"+t1.getName());  
9.         System.out.println("Name of t2:"+t2.getName());  
10.    }  
11.    t1.start();  
12.    t2.start();  
13.    t1.setName("Sonoo Jaiswal");  
14.    System.out.println("After changing name of t1:"+t1.getName());  
15.    }  
16. }  
17. }
```

Output: Name of t1:Thread-0

Name of t2:Thread-1
id of t1:8
running...
After changling name of t1:Sonoo Jaiswal
running...

The currentThread() method:

The currentThread() method returns a reference to the currently executing thread object.

Syntax of currentThread() method:

- **public static Thread currentThread():** returns the reference of currently running thread.

Example of currentThread() method:

```
1. class Multi6 extends Thread{  
2.     public void run(){  
3.         System.out.println(Thread.currentThread().getName());  
4.     }  
5. }  
6. public static void main(String args[]){
```

```
7. Multi6 t1=new Multi6();
8. Multi6 t2=new Multi6();
9.
10. t1.start();
11. t2.start();
12. }
13. }
```

Output:Thread-0

 Thread-1

Priority of a Thread (Thread Priority):

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

```
1. class Multi10 extends Thread{
2.     public void run(){
3.         System.out.println("running thread name is:"+Thread.currentThread().getName());
4.         System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
5.
6.     }
7.     public static void main(String args[]){
8.         Multi10 m1=new Multi10();
9.         Multi10 m2=new Multi10();
10.        m1.setPriority(Thread.MIN_PRIORITY);
11.        m2.setPriority(Thread.MAX_PRIORITY);
12.        m1.start();
13.        m2.start();
14.
15.    }
16. }
```

Output:running thread name is:Thread-0

 running thread priority is:10

```
running thread name is:Thread-1  
running thread priority is:1
```

Daemon Thread

There are two types of threads user thread and daemon thread. The daemon thread is a service provider thread. It provides services to the user thread. Its life depends on the user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

Points to remember for Daemon Thread:

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.

Why JVM terminates the daemon thread if there is no user thread remaining?

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

Methods for Daemon thread:

The `java.lang.Thread` class provides two methods related to daemon thread

- **`public void setDaemon(boolean status)`:** is used to mark the current thread as daemon thread or user thread.
- **`public boolean isDaemon()`:** is used to check that current is daemon.

Simple example of Daemon thread

```
1. class MyThread extends Thread{  
2.     public void run(){  
3.         System.out.println("Name: "+Thread.currentThread().getName());  
4.         System.out.println("Daemon: "+Thread.currentThread().isDaemon());  
5.     }  
6.  
7.     public static void main(String[] args){  
8.         MyThread t1=new MyThread();  
9.         MyThread t2=new MyThread();  
10.        t1.setDaemon(true);
```

Core Java

```
11.  
12. t1.start();  
13. t2.start();  
14. }  
15. }
```

Output:Name: thread-0

 Daemon: true

 Name: thread-1

 Daemon: false

Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.

```
1. class MyThread extends Thread{  
2.     public void run(){  
3.         System.out.println("Name: "+Thread.currentThread().getName());  
4.         System.out.println("Daemon: "+Thread.currentThread().isDaemon());  
5.     }  
6.  
7.     public static void main(String[] args){  
8.         MyThread t1=new MyThread();  
9.         MyThread t2=new MyThread();  
10.        t1.start();  
11.        t1.setDaemon(true); //will throw exception here  
12.        t2.start();  
13.    }  
14. }
```

Output:exception in thread main: java.lang.IllegalThreadStateException

Thread Pooling in Java

Thread pool represents a group of worker threads that are waiting for the job. Here, threads are executed whenever they get the job.

In case of thread pool, a group of fixed size threads are created. A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, thread is contained in the thread pool again.

Advantage of Thread Pool

Better performance It saves time because there is no need to create new thread.

Where is it used?

It is used in Servlet and JSP where container creates a thread pool to process the request.

Core Java

Example of Java Thread Pool

Let's see a simple example of java thread pool using executors and ThreadPoolExecutor.

```
1. import java.util.concurrent.ExecutorService;
2. import java.util.concurrent.Executors;
3. class WorkerThread implements Runnable {
4.     private String message;
5.     public WorkerThread(String s){
6.         this.message=s;
7.     }
8.
9.     public void run() {
10.         System.out.println(Thread.currentThread().getName()+" (Start) message = "+message);
11.         processmessage();
12.         System.out.println(Thread.currentThread().getName()+" (End)");
13.     }
14.
15.     private void processmessage() {
16.         try { Thread.sleep(2000); } catch (InterruptedException e) { e.printStackTrace(); }
17.     }
18. }
19.
20. public class SimpleThreadPool {
21.     public static void main(String[] args) {
22.         ExecutorService executor = Executors.newFixedThreadPool(5);
23.         for (int i = 0; i < 10; i++) {
24.             Runnable worker = new WorkerThread(""+ i);
25.             executor.execute(worker);
26.         }
27.         executor.shutdown();
28.         while (!executor.isTerminated()) { }
29.
30.         System.out.println("Finished all threads");
31.     }
32.
33. }
```

Output:

1. pool-1-thread-1 (Start) message = 0
2. pool-1-thread-2 (Start) message = 1
3. pool-1-thread-3 (Start) message = 2
4. pool-1-thread-5 (Start) message = 4
5. pool-1-thread-4 (Start) message = 3
6. pool-1-thread-2 (End)
7. pool-1-thread-2 (Start) message = 5
8. pool-1-thread-1 (End)
9. pool-1-thread-1 (Start) message = 6
10. pool-1-thread-3 (End)
11. pool-1-thread-3 (Start) message = 7

Core Java

```
12. pool-1-thread-4 (End)
13. pool-1-thread-4 (Start) message = 8
14. pool-1-thread-5 (End)
15. pool-1-thread-5 (Start) message = 9
16. pool-1-thread-2 (End)
17. pool-1-thread-1 (End)
18. pool-1-thread-4 (End)
19. pool-1-thread-3 (End)
20. pool-1-thread-5 (End)
21. Finished all threads
```

Shutdown Hook

The shutdown hook can be used to perform cleanup resource or save the state when JVM shuts down normally or abruptly. Performing clean resource means closing log file, sending some alerts or something else. So if you want to execute some code before JVM shuts down, use shutdown hook.

When does the JVM shut down?

The JVM shuts down when:

- user presses ctrl+c on the command prompt
- System.exit(int) method is invoked
- user logoff
- user shutdown etc.

The addShutdownHook(Runnable r) method

The addShutdownHook() method of Runtime class is used to register the thread with the Virtual Machine. Syntax:

```
1. public void addShutdownHook(Runnable r){ }
```

The object of Runtime class can be obtained by calling the static factory method getRuntime(). For example:

```
Runtime r = Runtime.getRuntime();
```

Factory method

The method that returns the instance of a class is known as factory method.

Simple example of Shutdown Hook

```
1. class MyThread extends Thread{
```

Core Java

```
2. public void run(){
3.     System.out.println("shut down hook task completed..");
4. }
5.
6.
7. public class Shutdown {
8.     public static void main(String[] args) throws Exception {
9.
10.    Runtime r=Runtime.getRuntime();
11.    r.addShutdownHook(new MyThread());
12.
13.    System.out.println("Now main sleeping... press ctrl+c to exit");
14.    try{Thread.sleep(3000);}catch (Exception e) {}
15. }
16. }
```

Output:Now main sleeping... press ctrl+c to exit
shut down hook task completed..

Note: The shutdown sequence can be stopped by invoking the halt(int) method of Runtime class.

Same example of Shutdown Hook by anonymous class:

```
1. public class Shutdown {
2.     public static void main(String[] args) throws Exception {
3.
4.    Runtime r=Runtime.getRuntime();
5.
6.    r.addShutdownHook(new Runnable(){
7.        public void run(){
8.            System.out.println("shut down hook task completed..");
9.        }
10.    });
11. );
12.
13. System.out.println("Now main sleeping... press ctrl+c to exit");
14. try{Thread.sleep(3000);}catch (Exception e) {}
15. }
16. }
```

Output:Now main sleeping... press ctrl+c to exit
shut down hook task completed..

How to perform single task by multiple threads?

If you have to perform single task by many threads, have only one run() method. For

Core Java

example:

```
1. //<b><i>Program of performing single task by multiple threads</i></b>
2.
3. class Multi extends Thread{
4.     public void run(){
5.         System.out.println("task one");
6.     }
7.     public static void main(String args[]){
8.         Multi t1=new Multi();
9.         Multi t2=new Multi();
10.        Multi t3=new Multi();
11.
12.        t1.start();
13.        t2.start();
14.        t3.start();
15.    }
16. }
```

Output:task one

task one
task one

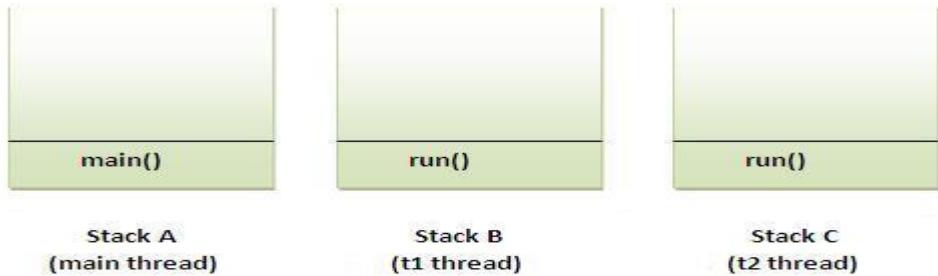
```
1. //<b><i>Program of performing single task by multiple threads</i></b>
2.
3. class Multi3 implements Runnable{
4.     public void run(){
5.         System.out.println("task one");
6.     }
7.
8.     public static void main(String args[]){
9.         Thread t1 =new Thread(new Multi3());//passing anonymous object of Multi3 class
10.        Thread t2 =new Thread(new Multi3());
11.
12.        t1.start();
13.        t2.start();
14.
15.    }
16. }
```

Output:task one

task one

Core Java

Note: Each thread run in a separate callstack.



How to perform multiple tasks by multiple threads (multitasking in multithreading)?

If you have to perform multiple tasks by multiple threads, have multiple `run()` methods. For example:

```
1. //<b><i>Program of performing two tasks by two threads</i></b>
2.
3. class Simple1 extends Thread{
4.     public void run(){
5.         System.out.println("task one");
6.     }
7. }
8.
9. class Simple2 extends Thread{
10.    public void run(){
11.        System.out.println("task two");
12.    }
13. }
14.
15. class Test{
16.     public static void main(String args[]){
17.         Simple1 t1=new Simple1();
18.         Simple2 t2=new Simple2();
19.
20.         t1.start();
21.         t2.start();
22.     }
23. }
```

Output:
task one
task two

Core Java

Same example as above by anonymous class that extends Thread class:

```
1. //<b><i>Program of performing two tasks by two threads</i></b>
2.
3. class Test{
4. public static void main(String args[]){
5.     Thread t1=new Thread(){
6.         public void run(){
7.             System.out.println("task one");
8.         }
9.     };
10.    Thread t2=new Thread(){
11.        public void run(){
12.            System.out.println("task two");
13.        }
14.    };
15.
16.
17.    t1.start();
18.    t2.start();
19. }
20. }
```

Output:task one

task two

Same example as above by anonymous class that implements Runnable interface:

```
1. //<b><i>Program of performing two tasks by two threads</i></b>
2.
3. class Test{
4. public static void main(String args[]){
5.     Runnable r1=new Runnable(){
6.         public void run(){
7.             System.out.println("task one");
8.         }
9.     };
10.
11.    Runnable r2=new Runnable(){
12.        public void run(){
13.            System.out.println("task two");
14.        }
15.    };
16.
17.    Thread t1=new Thread(r1);
18.    Thread t2=new Thread(r1);
19.
20.    t1.start();
21.    t2.start();
```

```
22. }  
23. }
```

Output:task one
task two

Garbage Collection:

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically.

Advantage of Garbage Collection:

- It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
- It is automatically done by the garbage collector so we don't need to make extra efforts.

How can an object be unreferenced?

There are many ways:

- By nulling the reference
- By assigning a reference to another
- By anonymous object etc.

1) By nulling a reference:

1. Employee e=**new** Employee();
2. e=**null**;

2) By assigning a reference to another:

1. Employee e1=**new** Employee();
2. Employee e2=**new** Employee();
- 3.
4. e1=e2;**//now the first object referred by e1 is available for garbage collection**

3) By anonymous object:

1. **new** Employee();

finalize() method:

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in System class as:

Core Java

1. **protected void** finalize(){}

Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

gc() method:

The gc() method is used to invoke the garbage collector to perform cleanup processing.
The gc() is found in System and Runtime classes.

1. **public static void** gc(){}

Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

Simple Example of garbage collection:

```
1. class Simple{  
2.       
3.     public void finalize(){System.out.println("object is garbage collected");}  
4.       
5.     public static void main(String args[]){  
6.         Simple s1=new Simple();  
7.         Simple s2=new Simple();  
8.         s1=null;  
9.         s2=null;  
10.        System.gc();  
11.    }  
12. }
```

Output:object is garbage collected
object is garbage collected

Note: Neither finalization nor garbage collection is guaranteed.

Synchronization

Synchronization is the capability of control the access of multiple threads to any shared resource. Synchronization is better in case we want only one thread can access the shared resource at a time.

Why use Synchronization?

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

- Mutual Exclusive
 - 1. Synchronized method.
 - 2. Synchronized block.
 - 3. static synchronization.
- Cooperation (Inter-thread communication)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

1. by synchronized method
2. by synchronized block
3. by static synchronization

Understanding the concept of Lock

Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package `java.util.concurrent.locks` contains several lock implementations.

Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

```
1. Class Table{  
2.  
3.     void printTable(int n){//method not synchronized  
4.         for(int i=1;i<=5;i++){  
5.             System.out.println(n*i);  
6.             try{  
7.                 Thread.sleep(400);  
8.             }catch(Exception e){System.out.println(e);}  
9.         }  
10.  
11.    }  
12. }  
13.  
14. class MyThread1 extends Thread{  
15.     Table t;  
16.     MyThread1(Table t){  
17.         this.t=t;  
18.     }  
19.     public void run(){  
20.         t.printTable(5);  
21.     }  
22.  
23. }  
24. class MyThread2 extends Thread{  
25.     Table t;  
26.     MyThread2(Table t){  
27.         this.t=t;  
28.     }  
29.     public void run(){  
30.         t.printTable(100);  
31.     }  
32. }  
33.
```

Core Java

```
34. class Use{  
35.     public static void main(String args[]){  
36.         Table obj = new Table(); //only one object  
37.         MyThread1 t1=new MyThread1(obj);  
38.         MyThread2 t2=new MyThread2(obj);  
39.         t1.start();  
40.         t2.start();  
41.     }  
42. }
```

Output: 5

```
    100  
    10  
    200  
    15  
    300  
    20  
    400  
    25  
    500
```

Solution by synchronized method

- If you declare any method as synchronized, it is known as synchronized method.
- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the method returns.

```
1. <b><i>//Program of synchronized method</i></b>  
2.  
3. Class Table{  
4.  
5.     synchronized void printTable(int n){ //synchronized method  
6.         for(int i=1;i<=5;i++){  
7.             System.out.println(n*i);  
8.             try{  
9.                 Thread.sleep(400);  
10.            }catch(Exception e){System.out.println(e);}  
11.        }  
12.  
13.    }  
14. }  
15.  
16. class MyThread1 extends Thread{  
17.     Table t;  
18.     MyThread1(Table t){  
19.         this.t=t;  
20.     }  
21.     public void run(){
```

Core Java

```
22. t.printTable(5);
23. }
24.
25. }
26. class MyThread2 extends Thread{
27. Table t;
28. MyThread2(Table t){
29. this.t=t;
30. }
31. public void run(){
32. t.printTable(100);
33. }
34. }
35.
36. class Use{
37. public static void main(String args[]){
38. Table obj = new Table();//only one object
39. MyThread1 t1=new MyThread1(obj);
40. MyThread2 t2=new MyThread2(obj);
41. t1.start();
42. t2.start();
43. }
44. }
```

Output: 5

```
10
15
20
25
100
200
300
400
500
```

Same Example of synchronized method by using anonymous class

In this program, we have created the two threads by anonymous class, so less coding is required.

```
1. <b><i>//Program of synchronized method by using anonymous class</i></b>
2.
3. Class Table{
4.
5. synchronized void printTable(int n){//synchronized method
6. for(int i=1;i<=5;i++){
7. System.out.println(n*i);
8. try{
9. Thread.sleep(400);
```

Core Java

```
10. }catch(Exception e){System.out.println(e);}
11. }
12.
13. }
14. }
15.
16. class Use{
17. public static void main(String args[]){
18. final Table obj = new Table();//only one object
19.
20. MyThread1 t1=new MyThread1(){
21. public void run(){
22. obj.printTable(5);
23. }
24. };
25. MyThread1 t2=new MyThread1(){
26. public void run(){
27. obj.printTable(100);
28. }
29. };
30.
31. t1.start();
32. t2.start();
33. }
34. }
```

Output: 5

```
10
15
20
25
100
200
300
400
500
```

Synchronized block

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to remember for Synchronized block

Core Java

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.

Syntax to use synchronized block

```
1. synchronized (object reference expression) {  
2.   //code block  
3. }
```

Example of synchronized block

Let's see the simple example of synchronized block.

```
1. <b><i>//Program of synchronized block</i></b>  
2.  
3. class Table{  
4.  
5.   void printTable(int n){  
6.     synchronized(this){//synchronized block  
7.       for(int i=1;i<=5;i++){  
8.         System.out.println(n*i);  
9.         try{  
10.           Thread.sleep(400);  
11.         }catch(Exception e){System.out.println(e);}  
12.       }  
13.     }  
14.   }//end of the method  
15. }  
16.  
17. class MyThread1 extends Thread{  
18.   Table t;  
19.   MyThread1(Table t){  
20.     this.t=t;  
21.   }  
22.   public void run(){  
23.     t.printTable(5);  
24.   }  
25.  
26. }  
27. class MyThread2 extends Thread{  
28.   Table t;  
29.   MyThread2(Table t){  
30.     this.t=t;  
31.   }  
32.   public void run(){  
33.     t.printTable(100);  
34.   }  
35. }  
36.
```

Core Java

```
37. class Use{  
38.     public static void main(String args[]){  
39.         Table obj = new Table(); //only one object  
40.         MyThread1 t1=new MyThread1(obj);  
41.         MyThread2 t2=new MyThread2(obj);  
42.         t1.start();  
43.         t2.start();  
44.     }  
45. }
```

Output:5

```
10  
15  
20  
25  
100  
200  
300  
400  
500
```

Same Example of synchronized block by using anonymous class:

```
1. <b><i> //Program of synchronized block by using anonymous class</i></b>  
2.  
3. class Table{  
4.  
5.     void printTable(int n){  
6.         synchronized(this){ //synchronized block  
7.             for(int i=1;i<=5;i++){  
8.                 System.out.println(n*i);  
9.             try{  
10.                 Thread.sleep(400);  
11.             }catch(Exception e){System.out.println(e);}  
12.         }  
13.     }  
14. } //end of the method  
15. }  
16.  
17. class Use{  
18.     public static void main(String args[]){  
19.         final Table obj = new Table(); //only one object  
20.  
21.         Thread t1=new Thread(){  
22.             public void run(){  
23.                 obj.printTable(5);  
24.             }  
25.         };
```

Core Java

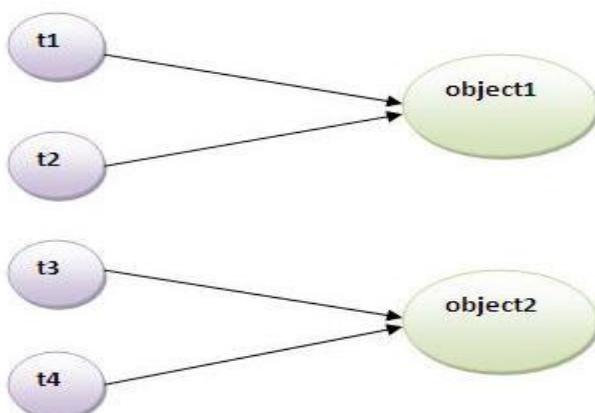
```
26. Thread t2=new Thread(){  
27.     public void run(){  
28.         obj.printTable(100);  
29.     }  
30. };  
31.  
32. t1.start();  
33. t2.start();  
34. }  
35. }
```

Output:5

```
10  
15  
20  
25  
100  
200  
300  
400  
500
```

Static synchronization

If you make any static method as synchronized, the lock will be on the class not on object.



Problem without static synchronization

Suppose there are two objects of a shared class(e.g. Table) named object1 and object2.In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock.But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock.I want no interference between t1 and t3 or t2 and t4.Static synchronization solves this problem.

Core Java

Example of static synchronization

In this example we are applying synchronized keyword on the static method to perform static synchronization.

```
1. class Table{  
2.  
3.     synchronized static void printTable(int n){  
4.         for(int i=1;i<=10;i++){  
5.             System.out.println(n*i);  
6.             try{  
7.                 Thread.sleep(400);  
8.             }catch(Exception e){ }  
9.         }  
10.    }  
11. }  
12.  
13. class MyThread1 extends Thread{  
14.     public void run(){  
15.         Table.printTable(1);  
16.     }  
17. }  
18.  
19. class MyThread2 extends Thread{  
20.     public void run(){  
21.         Table.printTable(10);  
22.     }  
23. }  
24.  
25. class MyThread3 extends Thread{  
26.     public void run(){  
27.         Table.printTable(100);  
28.     }  
29. }  
30.  
31.  
32.  
33.  
34. class MyThread4 extends Thread{  
35.     public void run(){  
36.         Table.printTable(1000);  
37.     }  
38. }  
39.  
40. class Use{  
41.     public static void main(String t[]){  
42.         MyThread1 t1=new MyThread1();  
43.         MyThread2 t2=new MyThread2();  
44.         MyThread3 t3=new MyThread3();
```

Core Java

```
45. MyThread4 t4=new MyThread4();
46. t1.start();
47. t2.start();
48. t3.start();
49. t4.start();
50. }
51. }
```

Output: 1

```
2
3
4
5
6
7
8
9
10
10
20
30
40
50
60
70
80
90
100
100
200
300
400
500
600
700
800
900
1000
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
```

Same example of static synchronization by anonymous class

In this example, we are using anonymous class to create the threads.

```
1. class Table{  
2.  
3.     synchronized static void printTable(int n){  
4.         for(int i=1;i<=10;i++){  
5.             System.out.println(n*i);  
6.             try{  
7.                 Thread.sleep(400);  
8.             }catch(Exception e){ }  
9.         }  
10.    }  
11. }  
12.  
13. public class Test {  
14.     public static void main(String[] args) {  
15.  
16.         Thread t1=new Thread(){  
17.             public void run(){  
18.                 Table.printTable(1);  
19.             }  
20.         };  
21.  
22.         Thread t2=new Thread(){  
23.             public void run(){  
24.                 Table.printTable(10);  
25.             }  
26.         };  
27.  
28.         Thread t3=new Thread(){  
29.             public void run(){  
30.                 Table.printTable(100);  
31.             }  
32.         };  
33.  
34.         Thread t4=new Thread(){  
35.             public void run(){  
36.                 Table.printTable(1000);  
37.             }  
38.         };  
39.         t1.start();  
40.         t2.start();  
41.         t3.start();  
42.         t4.start();  
43.  
44.     }
```

Core Java

45. }

Output: 1

```
2
3
4
5
6
7
8
9
10
10
20
30
40
50
60
70
80
90
100
100
200
300
400
500
600
700
800
900
1000
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
```

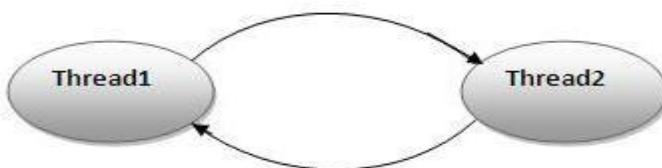
Synchronized block on a class lock:

The block synchronizes on the lock of the object denoted by the reference `.class`. A static synchronized method `printTable(int n)` in class `Table` is equivalent to the following declaration:

```
1. static void printTable(int n) {  
2.     synchronized (Table.class) { // Synchronized block on class A  
3.         // ...  
4.     }  
5. }
```

Deadlock:

Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



Example of Deadlock in java:

```
1. public class DeadlockExample {  
2.     public static void main(String[] args) {  
3.         final String resource1 = "ratan jaiswal";  
4.         final String resource2 = "vimal jaiswal";  
5.         // t1 tries to lock resource1 then resource2  
6.         Thread t1 = new Thread() {  
7.             public void run() {  
8.                 synchronized (resource1) {  
9.                     System.out.println("Thread 1: locked resource 1");  
10.                try { Thread.sleep(100); } catch (Exception e) {}  
11.                synchronized (resource2) {  
12.                    System.out.println("Thread 1: locked resource 2");  
13.                }  
14.            }  
15.        }  
16.    }  
17. }
```

Core Java

```
18.    };
19.
20. // t2 tries to lock resource2 then resource1
21. Thread t2 = new Thread() {
22.     public void run() {
23.         synchronized (resource2) {
24.             System.out.println("Thread 2: locked resource 2");
25.
26.             try { Thread.sleep(100); } catch (Exception e) {} }
27.
28.         synchronized (resource1) {
29.             System.out.println("Thread 2: locked resource 1");
30.         }
31.     }
32. }
33. };
34.
35.
36. t1.start();
37. t2.start();
38. }
39. }
40.
```

Output: Thread 1: locked resource 1

Thread 2: locked resource 2

Inter-thread communication in Java

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of **Object class**:

- wait()
- notify()
- notifyAll()

1) wait() method

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Core Java

Method	Description
public final void wait() throws InterruptedException	waits until object is notified.
public final void wait(long timeout) throws InterruptedException	waits for the specified amount of time.

2) notify() method

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. Syntax:

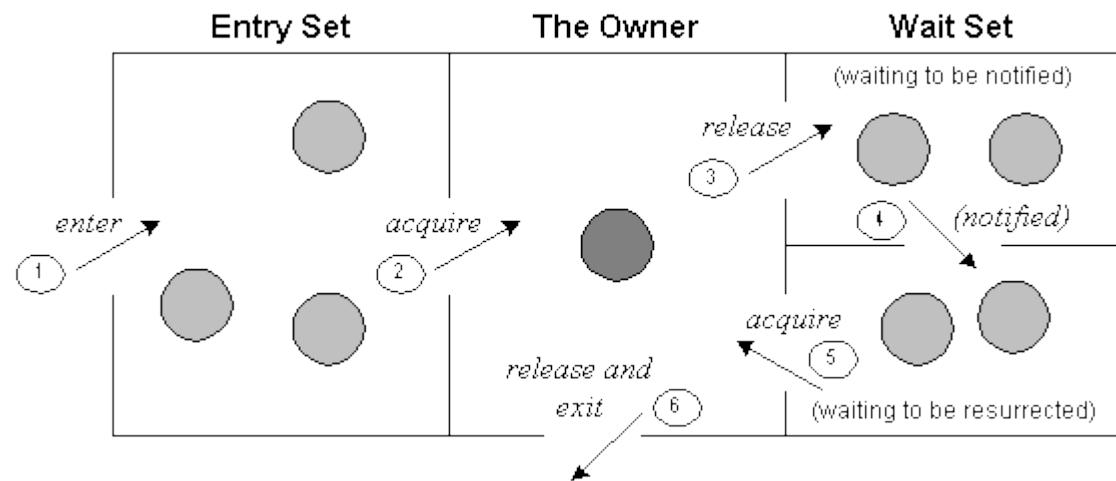
```
public final void notify()
```

3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor. Syntax:

```
public final void notifyAll()
```

Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by one thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.

Core Java

6. After completion of the task, thread releases the lock and exits the monitor state of the object.

Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

It is because they are related to lock and object has a lock.

Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

wait()	sleep()
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method
is the non-static method	is the static method
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.

Example of inter thread communication in java

Let's see the simple example of inter thread communication.

```
1. class Customer{  
2.     int amount=10000;  
3.  
4.     synchronized void withdraw(int amount){  
5.         System.out.println("going to withdraw...");  
6.  
7.         if(this.amount<amount){  
8.             System.out.println("Less balance; waiting for deposit...");  
9.             try{wait();}catch(Exception e){}  
10.        }  
11.        this.amount-=amount;  
12.        System.out.println("withdraw completed...");  
13.    }
```

```
14.  
15. synchronized void deposit(int amount){  
16. System.out.println("going to deposit...");  
17. this.amount+=amount;  
18. System.out.println("deposit completed... ");  
19. notify();  
20. }  
21. }  
22.  
23. class Test{  
24. public static void main(String args[]){  
25. final Customer c=new Customer();  
26. new Thread(){  
27. public void run(){c.withdraw(15000);}  
28. }.start();  
29. new Thread(){  
30. public void run(){c.deposit(10000);}  
31. }.start();  
32.  
33. }}
```

Output: going to withdraw...

Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed

Interrupting a Thread:

If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException. If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true. Let's first see the methods provided by the Thread class for thread interruption.

The 3 methods provided by the Thread class for interrupting a thread

- **public void interrupt()**
- **public static boolean interrupted()**
- **public boolean isInterrupted()**

Example of interrupting a thread that stops working

In this example, after interrupting the thread, we are propagating it, so it will stop working. If we don't want to stop the thread, we can handle it where sleep() or wait() method is invoked. Let's first see the example where we are propagating the exception.

```
1. class A extends Thread{  
2.     public void run(){  
3.         try{  
4.             Thread.sleep(1000);  
5.             System.out.println("task");  
6.         }catch(InterruptedException e){  
7.             throw new RuntimeException("Thread interrupted..."+e);  
8.         }  
9.  
10.    }  
11.  
12.    public static void main(String args[]){  
13.        A t1=new A();  
14.        t1.start();  
15.        try{  
16.            t1.interrupt();  
17.        }catch(Exception e){System.out.println("Exception handled "+e);}  
18.  
19.    }  
20. }
```

1. Output:Exception in thread-0
2. java.lang.RuntimeException: Thread interrupted...
3. java.lang.InterruptedException: sleep interrupted
4. at A.run(A.java:7)

Example of interrupting a thread that doesn't stop working

In this example, after interrupting the thread, we handle the exception, so it will break out the sleeping but will not stop working.

```
1. class A extends Thread{  
2.     public void run(){  
3.         try{  
4.             Thread.sleep(1000);  
5.             System.out.println("task");  
6.         }catch(InterruptedException e){  
7.             System.out.println("Exception handled "+e);  
8.         }  
9.         System.out.println("thread is running...");  
10.    }  
11.
```

Core Java

```
12. public static void main(String args[]){
13.     A t1=new A();
14.     t1.start();
15.
16.     t1.interrupt();
17.
18. }
19. }

1. <strong>Output:</strong>Exception handled
2.     java.lang.InterruptedException: sleep interrupted
3.     thread is running...
```

Example of interrupting thread that behaves normally

If thread is not in sleeping or waiting state, calling the interrupt() method sets the interrupted flag to true that can be used to stop the thread by the java programmer later.

```
1. class A extends Thread{
2.
3.     public void run(){
4.         for(int i=1;i<=5;i++)
5.             System.out.println(i);
6.     }
7.
8.     public static void main(String args[]){
9.         A t1=new A();
10.        t1.start();
11.
12.        t1.interrupt();
13.
14.    }
15. }

1. <strong>Output:</strong>1
2.     2
3.     3
4.     4
5.     5
```

What about isInterrupted and interrupted method?

The isInterrupted() method returns the interrupted flag either true or false. The static interrupted() method returns the interrupted flag after that it sets the flag to false if it is true.

```
1. class InterruptedDemo extends Thread{
2.
3.     public void run(){
4.         for(int i=1;i<=2;i++){
5.             if(Thread.interrupted()){
6.                 System.out.println("code for interrupted thread");
```

Core Java

```
7. }
8. else{
9. System.out.println("code for normal thread");
10. }
11.
12. }//end of for loop
13. }
14.
15. public static void main(String args[]){
16.
17. InterruptedDemo t1=new InterruptedDemo();
18. InterruptedDemo t2=new InterruptedDemo();
19.
20. t1.start();
21. t1.interrupt();
22.
23. t2.start();
24.
25. }
26. }

1. <strong>Output:</strong>Code for interrupted thread
2.     code for normal thread
3.     code for normal thread
4.     code for normal thread
5.
```

Reentrant Monitor in Java

According to Sun Microsystems, **Java monitors are reentrant** means java thread can reuse the same monitor for different synchronized methods if method is called from the method.

Advantage of Reentrant Monitor

It eliminates the possibility of single thread deadlocking

Let's understand the java reentrant monitor by the example given below:

```
1. class Reentrant {
2.     public synchronized void m() {
3.         n();
4.         System.out.println("this is m() method");
5.     }
6.     public synchronized void n() {
7.         System.out.println("this is n() method");
8.     }
9. }
```

In this class, m and n are the synchronized methods. The m() method internally calls the n() method.

Core Java

Now let's call the m() method on a thread. In the class given below, we are creating thread using anonymous class.

```
1. class ReentrantExample{  
2.     public static void main(String args[]){  
3.         final Reentrant re=new Reentrant();  
4.  
5.         Thread t1=new Thread(){  
6.             public void run(){  
7.                 re.m();//calling method of Reentrant class  
8.             }  
9.         };  
10.        t1.start();  
11.    }}
```

Output: this is n() method
this is m() method

Input and Output in Java

Input and Output (I/O) is used to process the input and produce the output based on the input. Java uses the concept of stream to make I/O operations fast. `java.io` package contains all the classes required for input and output operations.

Stream

A stream is a sequence of data. In Java a stream is composed of bytes. It's called a stream because it's like a stream of water that continues to flow.

Three streams are created for us automatically:

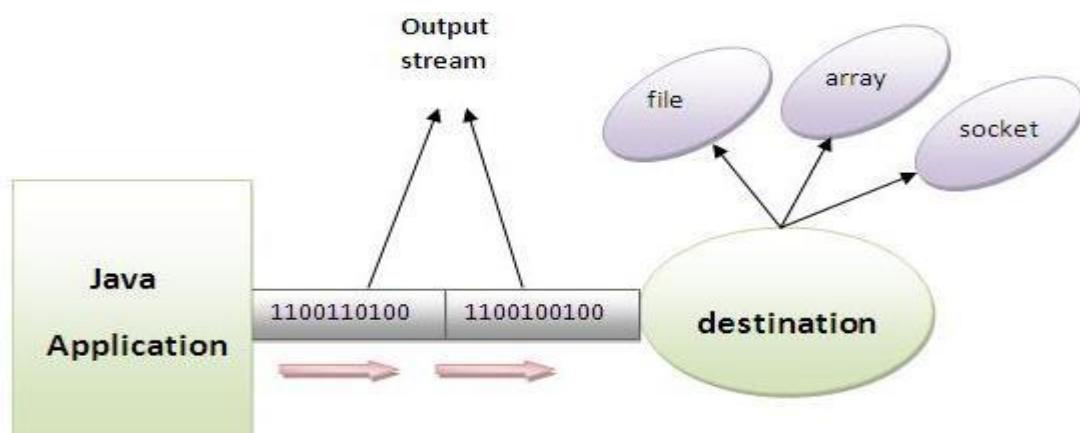
- 1) `System.out`: standard output stream
- 2) `System.in`: standard input stream
- 3) `System.err`: standard error

Do You Know ?

- How to write a common data to multiple files using single stream only ?
- How can we access multiple files by single stream ?
- How can we improve the performance of Input and Output operation ?
- How many ways can we read data from the keyboard?
- What is console class ?
- How to compress and uncompress the data of a file?

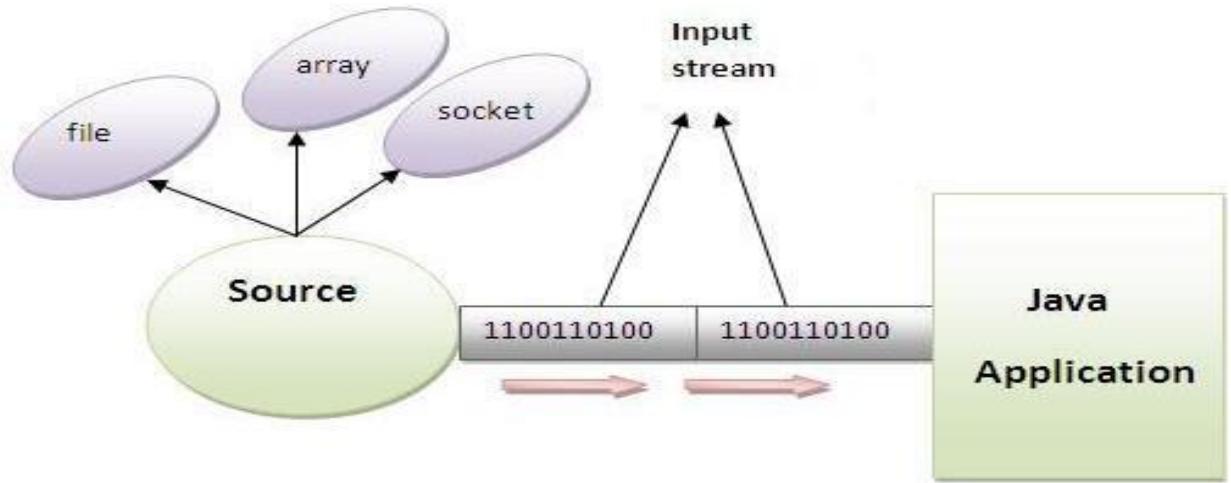
OutputStream

Java application uses an output stream to write data to a destination, it may be a file, an array, peripheral device or socket.



InputStream

Java application uses an input stream to read data from a source, it may be a file, an array, peripheral device or socket.



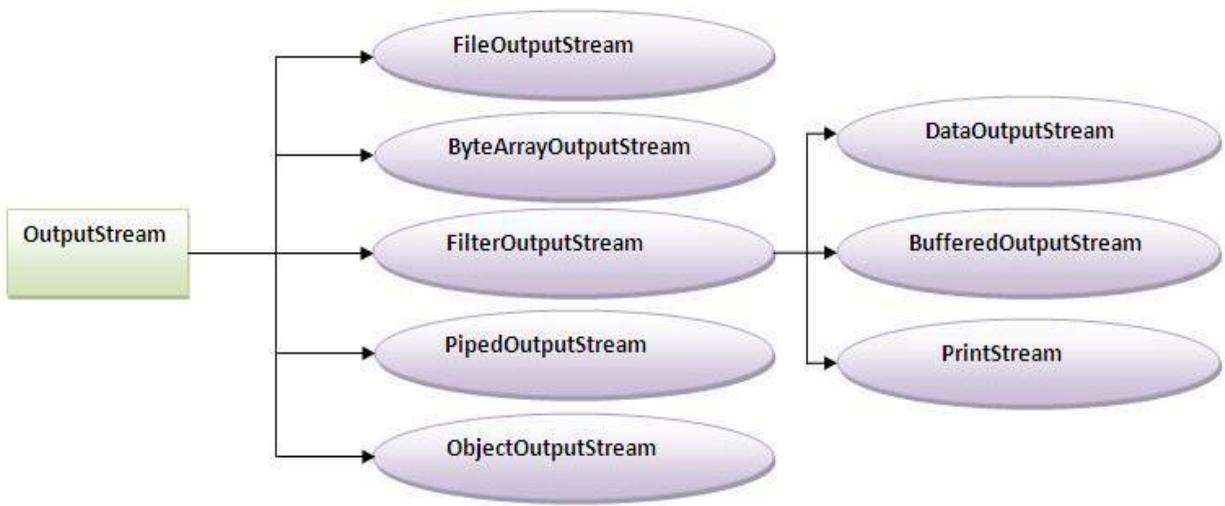
OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Commonly used methods of OutputStream class

Method	Description
1) public void write(int) throws IOException:	is used to write a byte to the current output stream.
2) public void write(byte[]) throws IOException:	is used to write an array of byte to the current output stream.
3) public void flush() throws IOException:	flushes the current output stream.
4) public void close() throws IOException:	is used to close the current output stream.

Core Java

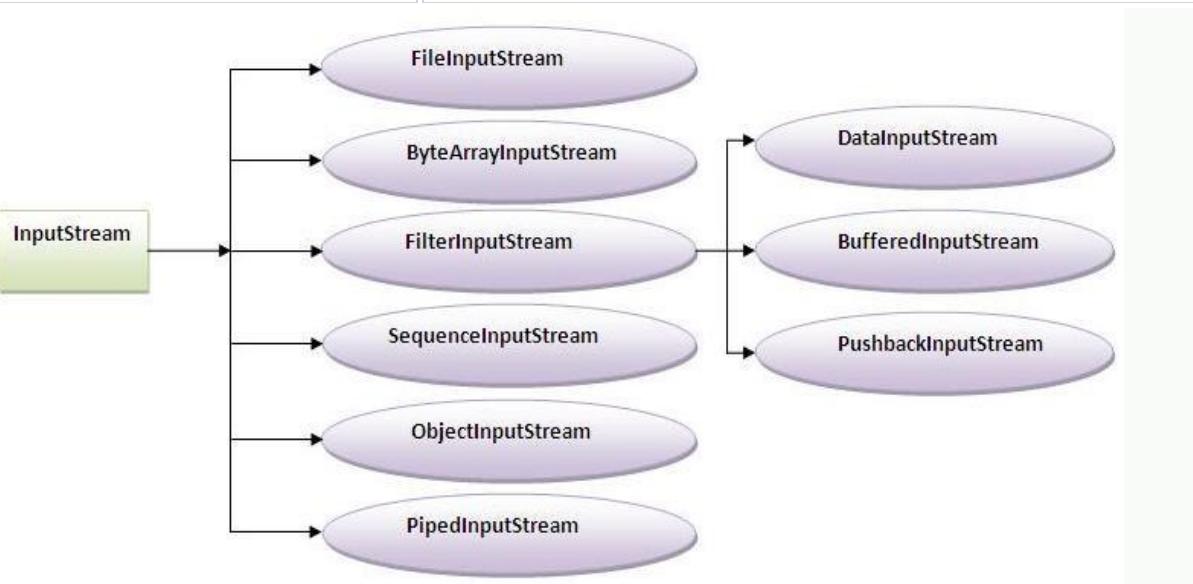


InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Commonly used methods of InputStream class

Method	Description
1) public abstract int read()throws IOException:	reads the next byte of data from the input stream. It returns -1 at the end of file.
2) public int available()throws IOException:	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException:	is used to close the current input stream.



Core Java

FileInputStream and FileOutputStream (File Handling):

FileInputStream and FileOutputStream classes are used to read and write data in file. In another words, they are used for file handling in java.

FileOutputStream class:

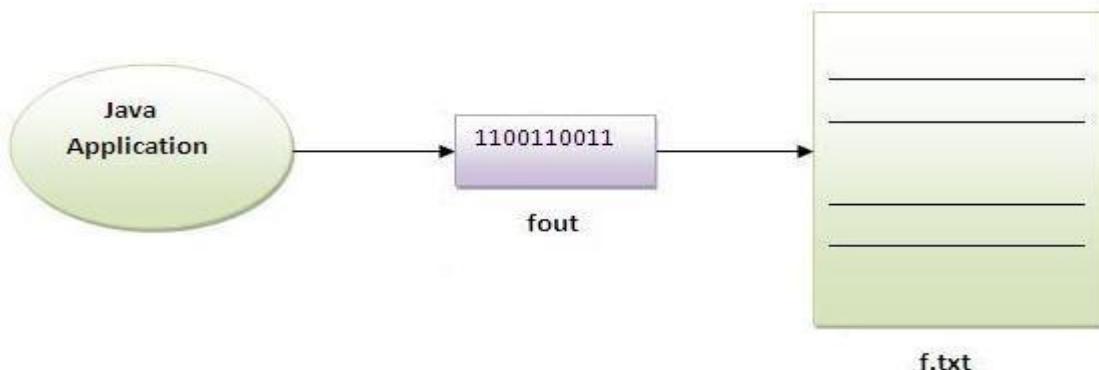
A FileOutputStream is an output stream for writing data to a file.

If you have to write primitive values then use FileOutputStream.Instead, for character-oriented data, prefer FileWriter.But you can write byte-oriented as well as character-oriented data.

Example of FileOutputStream class:

```
1. //<b><i>Simple program of writing data into the file</i></b>
2.
3.
4. import java.io.*;
5. class Test{
6. public static void main(String args[]){
7. try{
8. FileOutputStream fout=new FileOutputStream("abc.txt");
9. String s="Sachin Tendulkar is my favourite player";
10.
11. byte b[]=s.getBytes();
12. fout.write(b);
13.
14. fout.close();
15.
16. System.out.println("success...");
17. }catch(Exception e){System.out.println(e);}
18. }
19. }
```

Output:success...



Core Java

FileInputStream class:

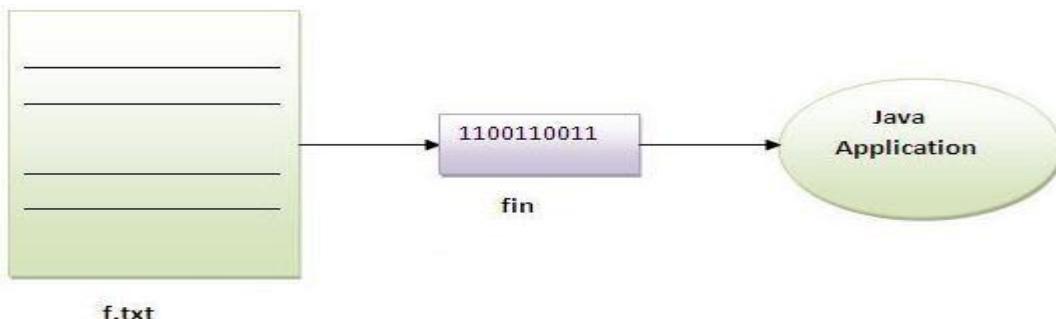
A FileInputStream obtains input bytes from a file. It is used for reading streams of raw bytes such as image data. For reading streams of characters, consider using FileReader.

It should be used to read byte-oriented data. For example, to read image etc.

Example of FileInputStream class:

```
1. //<b><i>Simple program of reading data from the file</i></b>
2.
3. import java.io.*;
4. class SimpleRead{
5. public static void main(String args[]){
6. try{
7. FileInputStream fin=new FileInputStream("abc.txt");
8. int i;
9. while((i=fr.read())!=-1)
10. System.out.println((char)i);
11.
12. fin.close();
13. }catch(Exception e){system.out.println(e);}
14. }
15. }
```

1. Output: Sachin is my favourite player.



Example of Reading the data of current java file and writing it into another file

We can read the data of any file using the FileInputStream class whether it is java file, image file, video file etc. In this example, we are reading the data of C.java file and writing it into another file M.java.

```
1. import java.io.*;
2.
3. class C{
4. public static void main(String args[])throws Exception{
5.
6. FileInputStream fin=new FileInputStream("C.java");
7. FileOutputStream fout=new FileOutputStream("M.java");
```

Core Java

```
8.  
9. int i=0;  
10. while((i=fin.read())!=-1){  
11. fout.write((byte)i);  
12. }  
13.  
14. fin.close();  
15. }  
16. }
```

ByteArrayOutputStream class:

In this stream, the data is written into a byte array. The buffer automatically grows as data is written to it.

Closing a ByteArrayOutputStream has no effect.

Commonly used Constructors of ByteArrayOutputStream class:

- 1) **ByteArrayOutputStream():**creates a new byte array output stream with the initial capacity of 32 bytes, though its size increases if necessary.
- 2) **ByteArrayOutputStream(int size):**creates a new byte array output stream, with a buffer capacity of the specified size, in bytes.

Commonly used Methods of ByteArrayOutputStream class:

- 1) **public synchronized void writeTo(OutputStream out) throws IOException:** writes the complete contents of this byte array output stream to the specified output stream.

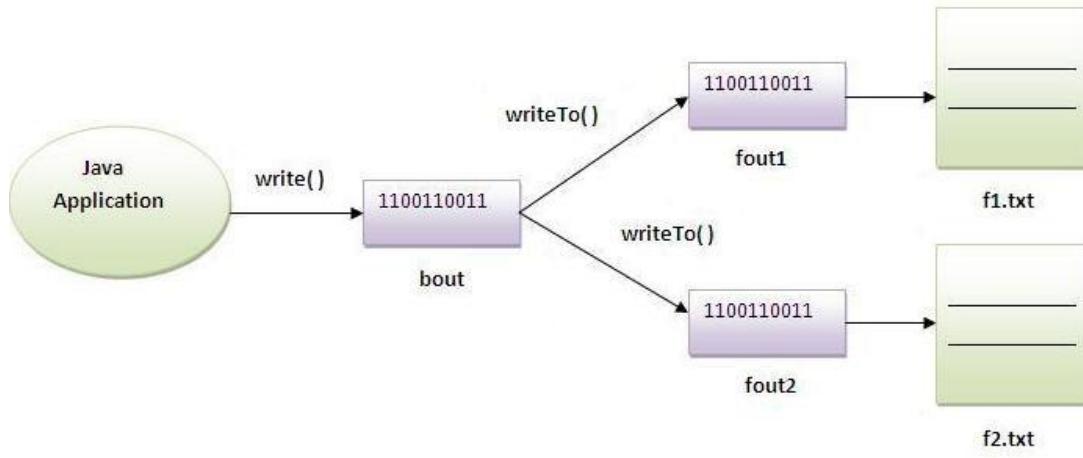
Example of ByteArrayOutputStream class:

```
1. //<b><i>Simple program of writing data by ByteArrayOutputStream class</i></b>  
2.  
3.  
4. import java.io.*;  
5. class S{  
6. public static void main(String args[])throws Exception{  
7.  
8. FileOutputStream fout1=new FileOutputStream("f1.txt");  
9. FileOutputStream fout2=new FileOutputStream("f2.txt");  
10.  
11. ByteArrayOutputStream bout=new ByteArrayOutputStream();  
12. bout.write(239);  
13.  
14. bout.writeTo(fout1);  
15. bout.writeTo(fout2);
```

Core Java

```
16.  
17. bout.flush();  
18.  
19. bout.close();//has no effect  
20. System.out.println("success...");  
21. }  
22. }
```

Output:success...



SequenceInputStream class:

SequenceInputStream class is used to read data from multiple streams.

Constructors of SequenceInputStream class:

- 1) **SequenceInputStream(InputStream s1, InputStream s2):** creates a new input stream by reading the data of two input stream in order, first s1 and then s2.
- 2) **SequenceInputStream(Enumeration e):** creates a new input stream by reading the data of an enumeration whose type is InputStream.

Simple example of SequenceInputStream class

In this example, we are printing the data of two files f1.txt and f2.txt.

```
1. //Program of SequenceInputStream that reads data of 2 files  
2.  
3. import java.io.*;  
4. class Simple{  
5.     public static void main(String args[])throws Exception{  
6.  
7.         FileinputStream fin1=new FileinputStream("f1.txt");
```

Core Java

```
8. FileInputStream fin2=new FileInputStream("f2.txt");
9.
10. SequenceInputStream sis=new SequenceInputStream(fin1,fin2);
11. int i;
12. while((i=sis.read())!=-1)
13. {
14.     System.out.println((char)i);
15. }
16. }
17. }
```

Example of SequenceInputStream class that reads the data from two files and write it into another

In this example, we are writing the data of two files f1.txt and f2.txt into another file named f3.txt.

```
1. //reading data of 2 files and writing it into one file
2.
3. import java.io.*;
4. class Simple{
5.     public static void main(String args[])throws Exception{
6.
7.         FileInputStream fin1=new FileInputStream("f1.txt");
8.         FileInputStream fin2=new FileInputStream("f2.txt");
9.
10.        FileOutputStream fout=new FileOutputStream("f3.txt");
11.
12.        SequenceInputStream sis=new SequenceInputStream(fin1,fin2);
13.        int i;
14.        while((i=sis.read())!=-1)
15.        {
16.            fout.write(i);
17.        }
18.        sis.close();
19.        fout.close();
20.        fin1.close();
21.        fin2.close();
22.
23.    }
24. }
```

Example of SequenceInputStream class that reads the data from multiple files using enumeration

If we need to read the data from more than two files, we need to have these information in the Enumeration object. Enumeration object can be get by calling elements method of the Vector class. Let's see the simple example where we are reading the data from the 4 files.

Core Java

```
1. import java.io.*;
2. import java.util.*;
3.
4. class B{
5. public static void main(String args[])throws IOException{
6.
7. //creating the FileInputStream objects for all the files
8. FileInputStream fin=new FileInputStream("A.java");
9. FileInputStream fin2=new FileInputStream("abc2.txt");
10. FileInputStream fin3=new FileInputStream("abc.txt");
11. FileInputStream fin4=new FileInputStream("B.java");
12.
13. //creating Vector object to all the stream
14. Vector v=new Vector();
15. v.add(fin);
16. v.add(fin2);
17. v.add(fin3);
18. v.add(fin4);
19.
20. //creating enumeration object by calling the elements method
21. Enumeration e=v.elements();
22.
23. //passing the enumeration object in the constructor
24. SequenceInputStream bin=new SequenceInputStream(e);
25. int i=0;
26.
27. while((i=bin.read())!=-1){
28. System.out.print((char)i);
29. }
30.
31. bin.close();
32. fin.close();
33. fin2.close();
34. }
35. }
```

BufferedOutputStream class:

BufferedOutputStream used an internal buffer. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

Example of BufferedOutputStream class:

In this example, we are writing the textual information in the BufferedOutputStream object which is connected to the FileOutputStream object. The flush() flushes the data of one stream and send it into another. It is required if you have connected the one stream with another.

Core Java

```
1. import java.io.*;
2. class Test{
3.     public static void main(String args[])throws Exception{
4.
5.         FileOutputStream fout=new FileOutputStream("f1.txt");
6.         BufferedOutputStream bout=new BufferedOutputStream(fout);
7.
8.         String s="Sachin is my favourite player";
9.         byte b[]={s.getBytes()};
10.        bout.write(b);
11.
12.        bout.flush();
13.        bout.close();
14.        System.out.println("success");
15.    }
16. }
```

Output:success...

Example of BufferedInputStream class:

```
1. //<b><i>Simple program of reading data from the file using buffer</i></b>
2.
3. import java.io.*;
4. class SimpleRead{
5.     public static void main(String args[]){
6.         try{
7.
8.             FileInputStream fin=new FileInputStream("f1.txt");
9.             BufferedInputStream bin=new BufferedInputStream(fin);
10.            int i;
11.            while((i=bin.read())!=-1)
12.                System.out.println((char)i);
13.
14.            fin.close();
15.        }catch(Exception e){system.out.println(e);}
16.    }
17. }
```

1. Output:Sachin is my favourite player

FileWriter class:

FileWriter class is used to write character-oriented data to the file. Sun Microsystem has suggested not to use the FileInputStream and FileOutputStream classes if you have to read and write the textual information.

Example of FileWriter class:

In this example, we are writing the data in the file abc.txt.

```
1. import java.io.*;
2. class Simple{
3. public static void main(String args[]){
4. try{
5. FileWriter fw=new FileWriter("abc.txt");
6. fw.write("my name is sachin");
7. fw.flush();
8.
9. fw.close();
10. }catch(Exception e){System.out.println(e);}
11. System.out.println("success");
12. }
13. }
```

Output:success...

FileReader class:

FileReader class is used to read data from the file.

Example of FileReader class:

In this example, we are reading the data from the file abc.txt file.

```
1. import java.io.*;
2. class Simple{
3. public static void main(String args[])throws Exception{
4.
5. FileReader fr=new FileReader("abc.txt");
6. int i;
7. while((i=fr.read())!=-1)
8. System.out.println((char)i);
9.
10. fr.close();
11. }
12. }
```

Output:my name is sachin

CharArrayWriter class:

The CharArrayWriter class can be used to write data to multiple files. This class implements the Appendable interface. Its buffer automatically grows when data is written in this stream. Calling the close() method on this object has no effect.

Core Java

Example of CharArrayWriter class:

In this example, we are writing a common data to 4 files a.txt, b.txt, c.txt and d.txt.

```
1. import java.io.*;
2. class Simple{
3.     public static void main(String args[])throws Exception{
4.
5.     CharArrayWriter out=new CharArrayWriter();
6.     out.write("my name is");
7.
8.     FileWriter f1=new FileWriter("a.txt");
9.     FileWriter f2=new FileWriter("b.txt");
10.    FileWriter f3=new FileWriter("c.txt");
11.    FileWriter f4=new FileWriter("d.txt");
12.
13.    out.writeTo(f1);
14.    out.writeTo(f2);
15.    out.writeTo(f3);
16.    out.writeTo(f4);
17.
18.
19.    f1.close();
20.    f2.close();
21.    f3.close();
22.    f4.close();
23. }
24. }
```

Reading data from keyboard:

There are many ways to read data from the keyboard. For example:

- InputStreamReader
- Console
- Scanner
- DataInputStream etc.

InputStreamReader class:

InputStreamReader class can be used to read data from keyboard. It performs two tasks:

- connects to input stream of keyboard
- converts the byte-oriented stream into character-oriented stream

BufferedReader class:

BufferedReader class can be used to read data line by line by readLine() method.

Core Java

Example of reading data from keyboard by InputStreamReader and BufferedReader class:

In this example, we are connecting the BufferedReader stream with the InputStreamReader stream for reading the line by line data from the keyboard.

//Program of reading data

```
import java.io.*;
class G5{
public static void main(String args[])throws Exception{

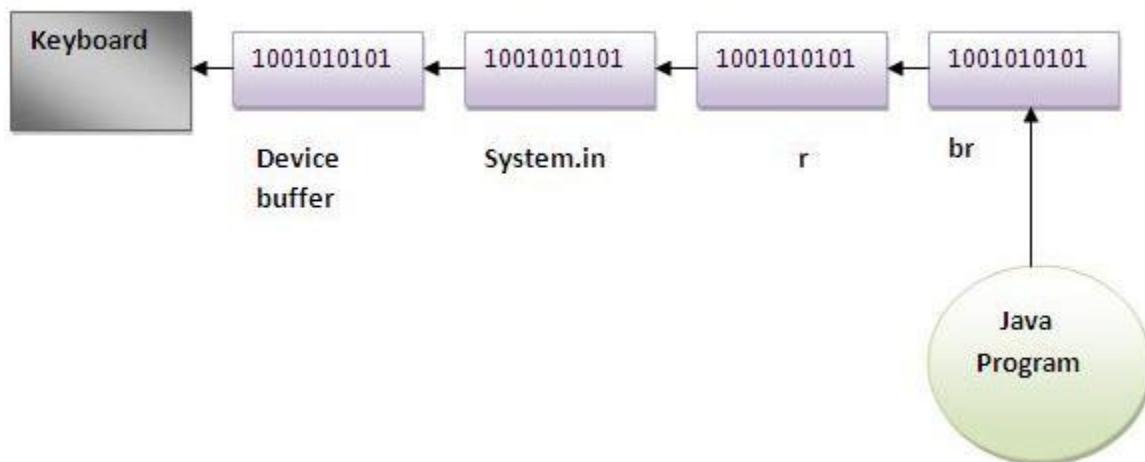
InputStreamReader r=new InputStreamReader(System.in);
BufferedReader br=new BufferedReader(r);

System.out.println("Enter ur name");
String name=br.readLine();
System.out.println("Welcome "+name);
}
}
```

Output:Enter ur name

Amit

Welcome Amit



Another Example of reading data from keyboard by InputStreamReader and BufferedReader class until the user writes stop

In this example, we are reading and printing the data until the user prints stop.

```
import java.io.*;
class G5{
public static void main(String args[])throws Exception{

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

Core Java

```
BufferedReader br=new BufferedReader(r);
```

```
String name="";
```

```
while(name.equals("stop")){
    System.out.println("Enter data: ");
    name=br.readLine();
    System.out.println("data is: "+name);
}
```

```
br.close();
```

```
r.close();
```

```
}
```

```
}
```

Output: Enter data: Amit

```
    data is: Amit
```

```
    Enter data: 10
```

```
    data is: 10
```

```
    Enter data: stop
```

```
    data is: stop
```

Console class (I/O)

The Console class can be used to get input from the keyboard.

How to get the object of Console class?

System class provides a static method named `console()` that returns the unique instance of Console class.

Syntax:

```
1. public static Console console(){ }
```

Commonly used methods of Console class:

1) public String readLine(): is used to read a single line of text from the console.

2) public String readLine(String fmt, Object... args): it provides a formatted prompt then reads the single line of text from the console.

3) public char[] readPassword(): is used to read password that is not being displayed on the console.

4) public char[] readPassword(String fmt, Object... args): it provides a formatted

Core Java

prompt then reads the password that is not being displayed on the console.

Example of Console class that reads name of user:

```
1. import java.io.*;
2. class A{
3. public static void main(String args[]){
4.
5.     Console c=System.console();
6.
7.     System.out.println("Enter ur name");
8.     String n=c.readLine();
9.     System.out.println("Welcome "+n);
10.
11. }
12. }
```

Example of Console class that reads password:

```
1. import java.io.*;
2. class A{
3. public static void main(String args[]){
4.
5.     Console c=System.console();
6.
7.     System.out.println("Enter password");
8.     char[] ch=c.readPassword();
9.
10.    System.out.println("Password is");
11.    for(char ch2:ch)
12.        System.out.print(ch2);
13.
14. }
15. }
```

java.util.Scanner class:

There are various ways to read input from the keyboard, the `java.util.Scanner` class is one of them. The `Scanner` class breaks the input into tokens using a delimiter which is whitespace by default. It provides many methods to read and parse various primitive values.

Commonly used methods of Scanner class:

There is a list of commonly used `Scanner` class methods:

- `public String next()`: it returns the next token from the scanner.
- `public String nextLine()`: it moves the scanner position to the next line and

Core Java

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.
- **public double nextDouble()**: it scans the next token as a double value.

Example of java.util.Scanner class:

Let's see the simple example of the 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.
5. Scanner sc=new Scanner(System.in);
6.
7. System.out.println("Enter your rollno");
8. int rollno=sc.nextInt();
9. System.out.println("Enter your name");
10. String name=sc.next();
11. System.out.println("Enter your fee");
12. double fee=sc.nextDouble();
13.
14. System.out.println("Rollno:"+rollno+" name:"+name+" fee:"+fee);
15.
16. }
17. }
```

Output:Enter your rollno

```
111
Enter your name
Ratan
Enter
450000
Rollno:111 name:Ratan fee:450000
```

Core Java

java.io.PrintStream class:

The PrintStream class provides methods to write data to another stream. The PrintStream class automatically flushes the data so there is no need to call flush() method. Moreover, its methods don't throw IOException.

Commonly used methods of PrintStream class:

There are many methods in PrintStream class. Let's see commonly used methods of PrintStream class:

- **public void print(boolean b):** it prints the specified boolean value.
- **public void print(char c):** it prints the specified char value.
- **public void print(char[] c):** it prints the specified character array values.
- **public void print(int i):** it prints the specified int value.
- **public void print(long l):** it prints the specified long value.
- **public void print(float f):** it prints the specified float value.
- **public void print(double d):** it prints the specified double value.
- **public void print(String s):** it prints the specified string value.
- **public void print(Object obj):** it prints the specified object value.
- **public void println(boolean b):** it prints the specified boolean value and terminates the line.
- **public void println(char c):** it prints the specified char value and terminates the line.
- **public void println(char[] c):** it prints the specified character array values and terminates the line.
- **public void println(int i):** it prints the specified int value and terminates the line.
- **public void println(long l):** it prints the specified long value and terminates the line.
- **public void println(float f):** it prints the specified float value and terminates the line.
- **public void println(double d):** it prints the specified double value and terminates the line.
- **public void println(String s):** it prints the specified string value and terminates the line.
- **public void println(Object obj):** it prints the specified object value and terminates the line.
- **public void println():** it terminates the line only.
- **public void printf(Object format, Object... args):** it writes the formatted string to the current stream.
- **public void printf(Locale l, Object format, Object... args):** it writes the formatted string to the current stream.
- **public void format(Object format, Object... args):** it writes the formatted string to the current stream using specified format.
- **public void format(Locale l, Object format, Object... args):** it writes the

Core Java

formatted string to the current stream using specified format.

Example of java.io.PrintStream class:

In this example, we are simply printing integer and string values.

```
1. import java.io.*;
2. class PrintStreamTest{
3.     public static void main(String args[])throws Exception{
4.
5.     FileOutputStream fout=new FileOutputStream("mfile.txt");
6.     PrintStream pout=new PrintStream(fout);
7.     pout.println(1900);
8.     pout.println("Hello Java");
9.     pout.println("Welcome to Java");
10.    pout.close();
11.    fout.close();
12.
13. }
14. }
```

Example of printf() method of java.io.PrintStream class:

Let's see the simple example of printing integer value by format specifier.

```
1. class PrintStreamTest{
2.     public static void main(String args[]){
3.         int a=10;
4.         System.out.printf("%d",a);//Note, out is the object of PrintStream class
5.
6.     }
7. }
```

Output:10

Compressing and Uncompressing File

The DeflaterOutputStream and InflaterInputStream classes provide mechanism to compress and uncompress the data in the **deflate compression format**.

DeflaterOutputStream class:

The DeflaterOutputStream class is used to compress the data in the deflate compression format. It provides facility to the other compression filters, such as GZIPOutputStream.

Core Java

Example of Compressing file using DeflaterOutputStream class

In this example, we are reading data of a file and compressing it into another file using DeflaterOutputStream class. You can compress any file, here we are compressing the Deflater.java file.

```
1. import java.io.*;
2. import java.util.zip.*;
3.
4. class Compress{
5. public static void main(String args[]){
6.
7. try{
8. FileInputStream fin=new FileInputStream("Deflater.java");
9.
10. FileOutputStream fout=new FileOutputStream("def.txt");
11. DeflaterOutputStream out=new DeflaterOutputStream(fout);
12.
13. int i;
14. while((i=fin.read())!=-1){
15. out.write((byte)i);
16. out.flush();
17. }
18.
19. fin.close();
20. out.close();
21.
22. }catch(Exception e){System.out.println(e);}
23. System.out.println("rest of the code");
24. }
25. }
```

InflaterInputStream class:

The InflaterInputStream class is used to uncompress the file in the deflate compression format. It provides facility to the other uncompression filters, such as GZIPInputStream class.

Example of uncompressing file using InflaterInputStream class

In this example, we are decompressing the compressed file def.txt into D.java .

```
1. import java.io.*;
2. import java.util.zip.*;
3.
4. class UnCompress{
5. public static void main(String args[]{
6.
7. }
```

Core Java

```
6.  
7. try{  
8. FileInputStream fin=new FileInputStream("def.txt");  
9. InflaterInputStream in=new InflaterInputStream(fin);  
10.  
11. FileOutputStream fout=new FileOutputStream("D.java");  
12.  
13. int i;  
14. while((i=in.read())!=-1){  
15.     fout.write((byte)i);  
16.     fout.flush();  
17. }  
18.  
19. fin.close();  
20. fout.close();  
21. in.close();  
22.  
23. }catch(Exception e){System.out.println(e);}  
24. System.out.println("rest of the code");  
25. }  
26. }
```

PipedInputStream and PipedOutputStream classes

The PipedInputStream and PipedOutputStream classes can be used to read and write data simultaneously. Both streams are connected with each other using the connect() method of the PipedOutputStream class.

Example of PipedInputStream and PipedOutputStream classes using threads

Here, we have created two threads t1 and t2. The **t1** thread writes the data using the PipedOutputStream object and the **t2** thread reads the data from that pipe using the PipedInputStream object. Both the piped stream object are connected with each other.

```
1. import java.io.*;  
2.  
3. class PipedWR{  
4. public static void main(String args[])throws Exception{  
5.  
6.     final PipedOutputStream pout=new PipedOutputStream();  
7.     final PipedInputStream pin=new PipedInputStream();  
8.  
9.     pout.connect(pin);//connecting the streams  
10.  
11. //creating one thread t1 which writes the data  
12. Thread t1=new Thread(){
```

Core Java

```
13. public void run(){
14.     for(int i=65;i<=90;i++){
15.         try{
16.
17.             pout.write(i);
18.             Thread.sleep(1000);
19.
20.         }catch(Exception e){}
21.     }
22. }
23. ;
24.
25. //creating another thread t2 which reads the data
26. Thread t2=new Thread(){
27.     public void run(){
28.         try{
29.             for(int i=65;i<=90;i++)
30.                 System.out.println(pin.read());
31.         }catch(Exception e){}
32.     }
33. ;
34.
35. //starting both threads
36. t1.start();
37. t2.start();
38.
39. }}
```

Serialization

Serialization is a mechanism of writing the state of an object into a byte stream. It is mainly used in Hibernate, JPA, EJB etc. The reverse operation of the serialization is called deserialization. The String class and all the wrapper classes implements Serializable interface by default.

Advantage of Serialization

It is mainly used to travel object's state on the network.

About Serializable interface

Serializable is a marker interface(have no body). It is just used to "mark" Java classes which support a certain capability. It must be implemented by the class whose object you want to persist. Let's see the example given below:

```
1. import java.io.Serializable;
2.
3. public class Student implements Serializable{
4.     int id;
5.     String name;
6.     public Student(int id, String name) {
7.         this.id = id;
8.         this.name = name;
9.     }
10. }
```

ObjectOutputStream class:

An ObjectOutputStream is used to write primitive data types and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams.

Commonly used Constructors:

1) **public ObjectOutputStream(OutputStream out) throws IOException {}** creates an ObjectOutputStream that writes to the specified OutputStream.

Commonly used Methods:

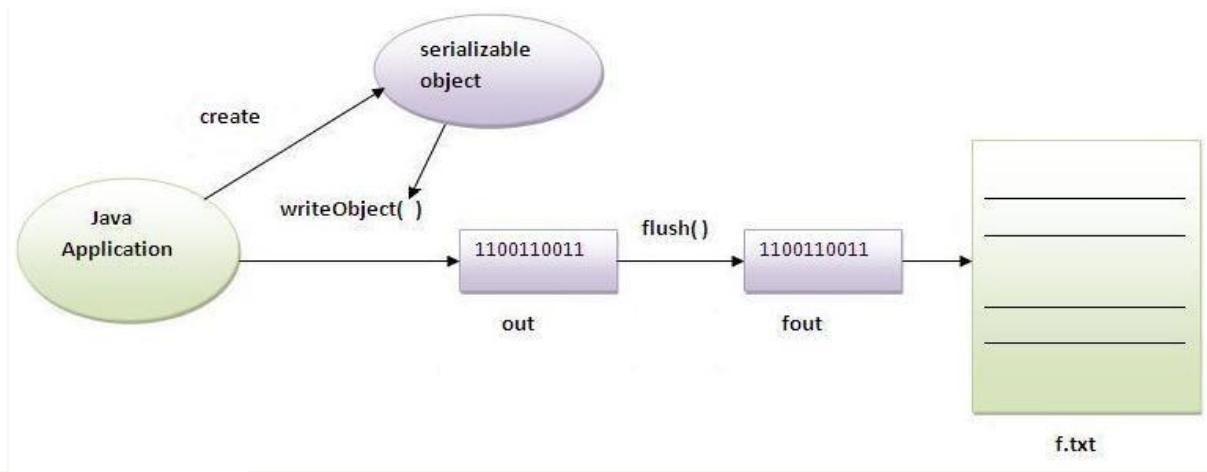
1) **public final void writeObject(Object obj) throws IOException {}** write the specified object to the ObjectOutputStream.

2) **public void flush() throws IOException {}** flushes the current output stream.

Core Java

Example of Serialization

In this example, we are going to serialize the object of Student class. The writeObject() method of ObjectOutputStream class provides the functionality to serialize the object. We are saving the state of the object in the file named f.txt.



```
1. import java.io.*;
2. class Persist{
3.     public static void main(String args[])throws Exception{
4.         Student s1 =new Student(211,"ravi");
5.
6.         FileOutputStream fout=new FileOutputStream("f.txt");
7.         ObjectOutputStream out=new ObjectOutputStream(fout);
8.
9.         out.writeObject(s1);
10.        out.flush();
11.
12.        System.out.println("success");
13.    }
14. }
```

Deserialization:

Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization.

ObjectInputStream class:

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

Commonly used Constructors:

- 1) **public ObjectInputStream(InputStream in) throws IOException {}** creates an ObjectInputStream that reads from the specified InputStream.

Core Java

Commonly used Methods:

- 1) **public final Object readObject() throws IOException,**
ClassNotFoundException{} reads an object from the input stream.

Example of Deserialization:

```
1. import java.io.*;
2. class Depersist{
3. public static void main(String args[])throws Exception{
4.
5. ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));
6. Student s=(Student)in.readObject();
7. System.out.println(s.id+" "+s.name);
8.
9. in.close();
10. }
11. }
1. <strong>Output:</strong>211 ravi
```

Serialization with Inheritance

If a class implements Serializable then all its subclasses will also be serializable. Let's see the example given below:

```
1. import java.io.Serializable;
2.
3. class Person implements Serializable{
4. int id;
5. String name;
6. Person(int id, String name) {
7. this.id = id;
8. this.name = name;
9. }
10. }
1. class Student extends Person{
2. String course;
3. int fee;
4. public Student(int id, String name, String course, int fee) {
5. super(id,name);
6. this.course=course;
7. this.fee=fee;
8. }
9. }
```

Now you can serialize the Student class object that extends the Person class which is Serializable. Parent class properties are inherited to subclasses so if parent class is Serializable, subclass would also be.

Core Java

Externalizable interface:

The Externalizable interface provides the facility of writing the state of an object into a byte stream in compress format. It is not a marker interface.

The Externalizable interface provides two methods:

- **public void writeExternal(ObjectOutput out) throws IOException**
- **public void readExternal(ObjectInput in) throws IOException**

Serialization with Static datamember

Note: If there is any static data member in a class, it will not be serialized because static is related to class not to instance.

```
1. class Employee implements Serializable{  
2.     int id;  
3.     String name;  
4.     static String companyName="IBM";//it won't be serialized  
5.     public Student(int id, String name) {  
6.         this.id = id;  
7.         this.name = name;  
8.     }  
9. }
```

Rule: In case of array or collection, all the objects of array or collection must be serializable, if any object is not serializable then serialization will be failed.

The transient keyword

The transient keyword is used in serialization. If you define any data member as transient, it will not be serialized. Let's take an example, I have declared a class as Student, it has three data members id, name and age. If you serialize the object, all the values will be serialized but I don't want to serialize one value, e.g. age then we can declare the age datamember as transient.

Example of transient keyword

In this example, we have created the two classes Student and Persist. One data member of the Student class is declared as transient, its value will not be serialized. If you deserialize the object, it will return the default value for transient variable.

```
1. import java.io.Serializable;  
2.  
3. public class Student implements Serializable{
```

Core Java

```
4. int id;
5. String name;
6. transient int age; //Now it will not be serialized
7. public Student(int id, String name,int age) {
8.     this.id = id;
9.     this.name = name;
10.    this.age=age;
11. }
12. }
1. import java.io.*;
2. class Persist{
3.     public static void main(String args[])throws Exception{
4.         Student s1 =new Student(211,"ravi",22);
5.
6.         FileOutputStream f=new FileOutputStream("f.txt");
7.         ObjectOutputStream out=new ObjectOutputStream(f);
8.
9.         out.writeObject(s1);
10.        out.flush();
11.
12.        System.out.println("success");
13.    }
14. }
1. <strong>Output:</strong>succss
```

Java Collection Framework

Collection Framework provides an architecture to store and manipulate the group of objects. All the operations that you perform on a data such as searching, sorting, insertion, deletion etc. can be performed by **Java Collection Framework**.

Collection simply means a single unit of objects. Collection framework provides many interfaces (**Set, List, Queue, Deque etc.**) and classes (**ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet etc**).

What is Collection>

Collection represents a single unit of objects i.e. a group.

What is framework?

- provides readymade architecture.
- represents set of classes and interface.
- is optional.

Collection framework

Collection framework represents a unified architecture for storing and manipulating group of object. It has:

1. Interfaces and its implementations i.e. classes
2. Algorithm

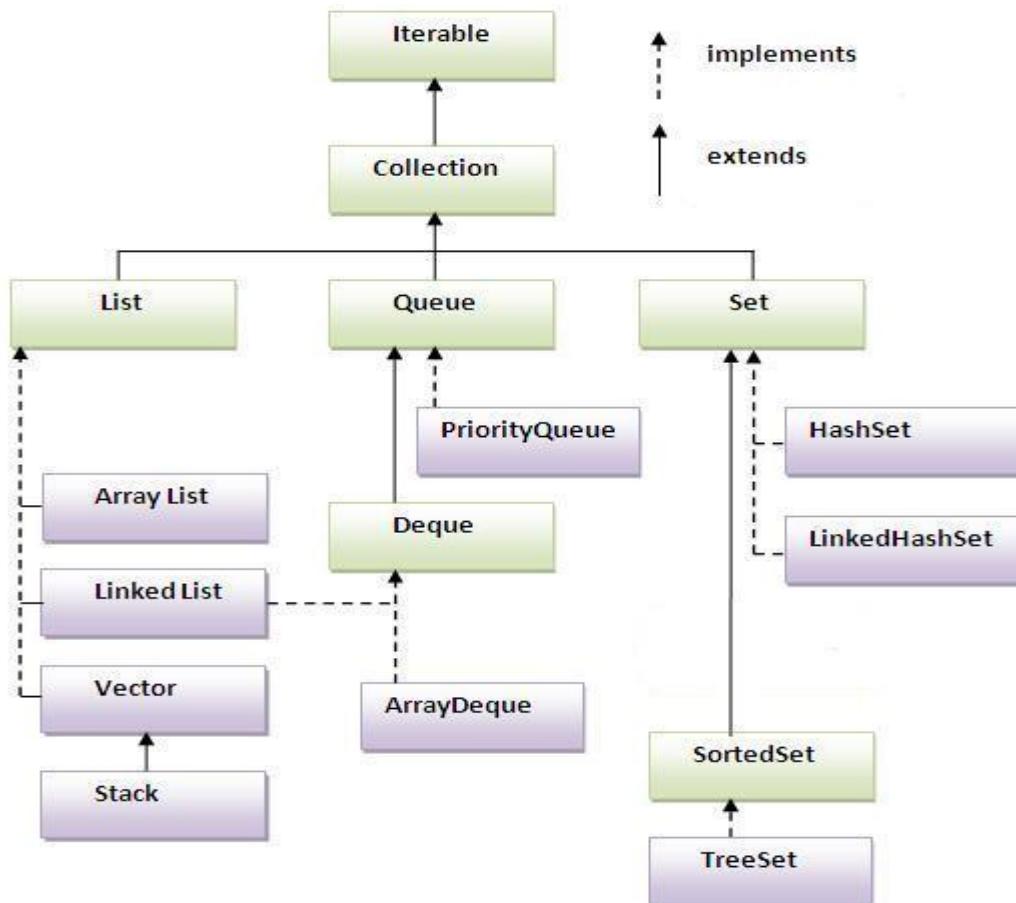
Do You Know ?

- *What are the two ways to iterate the elements of a collection ?*
- *What is the difference between ArrayList and LinkedList classes in collection framework ?*
- *What is the difference between ArrayList and Vector classes in collection framework ?*
- *What is the difference between HashSet and HashMap classes in collection framework ?*
- *What is the difference between HashMap and Hashtable class ?*
- *What is the difference between Iterator and Enumeration interface in collection framework ?*
- *How can we sort the elements of an object. What is the difference between Comparable and Comparator interfaces ?*
- *What does the hashCode() method ?*

Core Java

Hierarchy of Collection Framework

Let us see the hierarchy of collection framework. The **java.util** package contains all the classes and interfaces for Collection framework.



Commonly used methods of Collection interface

There are many methods declared in the Collection interface. They are as follows:

Method	Description
public boolean add(object element)	is used to insert an element in this collection.
public boolean addAll(collection c)	is used to insert the specified collection elements in the invoking collection.
public boolean remove(object element)	is used to delete an element from this collection.

Core Java

public boolean removeAll(Collection c)	is used to delete all the elements of specified collection from the invoking collection.
public boolean retainAll(Collection c)	is used to delete all the elements of invoking collection except the specified collection.
public int size()	return the total number of elements in the collection.
public void clear()	removes the total no of element from the collection.
public boolean contains(object element)	is used to search an element.
public boolean containsAll(collection c)	is used to search the specified collection in this collection.
public Iterator iterator()	returns an iterator.

Iterator interface

Iterator interface provides the facility of iterating the elements in forward direction only.

Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

1. **public boolean hasNext()** it returns true if iterator has more elements.
2. **public object next()** it returns the element and moves the cursor pointer to the next element.
3. **public void remove()** it removes the last elements returned by the iterator. It is rarely used.

What we will learn in Collection Framework ?

- **ArrayList class**
- **LinkedList class**
- **ListIterator interface**
- **HashSet class**
- **LinkedHashSet class**

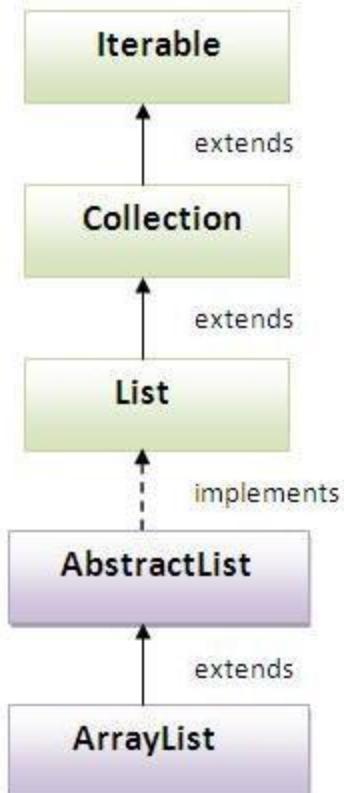
Core Java

- **TreeSet class**
- **PriorityQueue class**
- **Map interface**
- **HashMap class**
- **LinkedHashMap class**
- **TreeMap class**
- **Hashtable class**
- **Sorting**
- **Comparable interface**
- **Comparator interface**

ArrayList class:

- uses a dynamic array for storing the elements. It extends AbstractList class and implements List interface.
- can contain duplicate elements.
- maintains insertion order.
- not synchronized.
- random access because array works at the index basis.
- manipulation slow because a lot of shifting needs to be occurred.

Hierarchy of ArrayList class:



Core Java

Example of ArrayList:

```
1. import java.util.*;
2. class Simple{
3.     public static void main(String args[]){
4.
5.         ArrayList al=new ArrayList();
6.         al.add("Ravi");
7.         al.add("Vijay");
8.         al.add("Ravi");
9.         al.add("Ajay");
10.
11.        Iterator itr=al.iterator();
12.        while(itr.hasNext()){
13.            System.out.println(itr.next());
14.        }
15.    }
16. }
```

Output:Ravi

Vijay
Ravi
Ajay

Two ways to iterate the elements of collection:

1. By Iterator interface.
2. By for-each loop.

Iterating the elements of Collection by for-each loop:

```
1. import java.util.*;
2. class Simple{
3.     public static void main(String args[]){
4.
5.         ArrayList al=new ArrayList();
6.         al.add("Ravi");
7.         al.add("Vijay");
8.         al.add("Ravi");
9.         al.add("Ajay");
10.
11.        for(Object obj:al)
12.            System.out.println(obj);
13.    }
14. }
```

Output:Ravi

Vijay
Ravi
Ajay

Core Java

Storing user-defined class objects:

```
1. class Student{  
2.     int rollno;  
3.     String name;  
4.     int age;  
5.     Student(int rollno,String name,int age){  
6.         this.rollno=rollno;  
7.         this.name=name;  
8.         this.age=age;  
9.     }  
10. }  
1. import java.util.*;  
2. class Simple{  
3.     public static void main(String args[]){  
4.           
5.         Student s1=new Student(101,"Sonoo",23);  
6.         Student s2=new Student(102,"Ravi",21);  
7.         Student s3=new Student(103,"Hanumat",25);  
8.           
9.         ArrayList al=new ArrayList();  
10.        al.add(s1);  
11.        al.add(s2);  
12.        al.add(s3);  
13.          
14.        Iterator itr=al.iterator();  
15.        while(itr.hasNext()){  
16.            Student st=(Student)itr.next();  
17.            System.out.println(st.rollno+" "+st.name+" "+st.age);  
18.        }  
19.    }  
20. }
```

Output:101 Sonoo 23

102 Ravi 21

103 Hanumat 25

Example of addAll(Collection c) method:

```
1. import java.util.*;  
2. class Simple{  
3.     public static void main(String args[]){  
4.           
5.         ArrayList al=new ArrayList();  
6.         al.add("Ravi");  
7.         al.add("Vijay");  
8.         al.add("Ajay");  
9.           
10.        ArrayList al2=new ArrayList();  
11.        al2.add("Sonoo");  
12.        al2.add("Hanumat");  
13.    }
```

Core Java

```
14. al.addAll(al2);
15.
16. Iterator itr=al.iterator();
17. while(itr.hasNext()){
18. System.out.println(itr.next());
19. }
20. }
21. }
```

Output:Ravi

Vijay
Ajay
Sonoo
Hanumat

Example of removeAll() method:

```
1. import java.util.*;
2. class Simple{
3. public static void main(String args[]){
4.
5. ArrayList al=new ArrayList();
6. al.add("Ravi");
7. al.add("Vijay");
8. al.add("Ajay");
9.
10. ArrayList al2=new ArrayList();
11. al2.add("Ravi");
12. al2.add("Hanumat");
13.
14. al.removeAll(al2);
15.
16. System.out.println("iterating the elements after removing the elements of al2... ");
17. Iterator itr=al.iterator();
18. while(itr.hasNext()){
19. System.out.println(itr.next());
20. }
21.
22. }
23. }
```

Output:iterating the elements after removing the elements of al2...

Vijay
Ajay

Example of retainAll() method:

```
1. import java.util.*;
2. class Simple{
3. public static void main(String args[]){
4.
5. ArrayList al=new ArrayList();
```

Core Java

```
6. al.add("Ravi");
7. al.add("Vijay");
8. al.add("Ajay");
9.
10. ArrayList al2=new ArrayList();
11. al2.add("Ravi");
12. al2.add("Hanumat");
13.
14. al.retainAll(al2);
15.
16. System.out.println("iterating the elements after retaining the elements of al2...");
17. Iterator itr=al.iterator();
18. while(itr.hasNext()){
19. System.out.println(itr.next());
20. }
21. }
22. }
```

Output: iterating the elements after retaining the elements of al2...

Ravi

LinkedList class:

- uses doubly linked list to store the elements. It extends the AbstractList class and implements List and Deque interfaces.
- can contain duplicate elements.
- maintains insertion order.
- not synchronized.
- No random access.
- manipulation fast because no shifting needs to be occurred.
- can be used as list, stack or queue.



fig- doubly linked list

Example of LinkedList:

```
1. import java.util.*;
2. class Simple{
3. public static void main(String args[]){
4.
5. LinkedList al=new LinkedList();
6. al.add("Ravi");
```

Core Java

```
7. al.add("Vijay");
8. al.add("Ravi");
9. al.add("Ajay");
10.
11. Iterator itr=al.iterator();
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }
```

Output:Ravi

```
Vijay
Ravi
Ajay
```

List Interface:

List Interface is the subinterface of Collection. It contains methods to insert and delete elements in index basis. It is a factory of ListIterator interface.

Commonly used methods of List Interface:

1. public void add(int index, Object element);
2. public boolean addAll(int index, Collection c);
3. public Object get(int Index position);
4. public Object set(int index, Object element);
5. public Object remove(int index);
6. public ListIterator listIterator();
7. public ListIterator listIterator(int i);

ListIterator Interface:

ListIterator Interface is used to traverse the element in backward and forward direction.

Commonly used methods of ListIterator Interface:

1. public boolean hasNext();
2. public Object next();
3. public boolean hasPrevious();
4. public Object previous();

Core Java

Example of ListIterator Interface:

```
1. import java.util.*;
2. class Simple5{
3.     public static void main(String args[]){
4.
5.     ArrayList al=new ArrayList();
6.     al.add("Amit");
7.     al.add("Vijay");
8.     al.add("Kumar");
9.     al.add(1,"Sachin");
10.
11. System.out.println("element at 2nd position: "+al.get(2));
12.
13. ListIterator itr=al.listIterator();
14.
15. System.out.println("traversing elements in forward direction...");
16. while(itr.hasNext()){
17.     System.out.println(itr.next());
18. }
19.
20.
21. System.out.println("traversing elements in backward direction...");
22. while(itr.hasPrevious()){
23.     System.out.println(itr.previous());
24. }
25. }
26. }
```

Output:element at 2nd position: Vijay
traversing elements in forward direction...
Amit
Sachin
Vijay
Kumar
traversing elements in backward direction...
Kumar
Vijay
Sachin
Amit

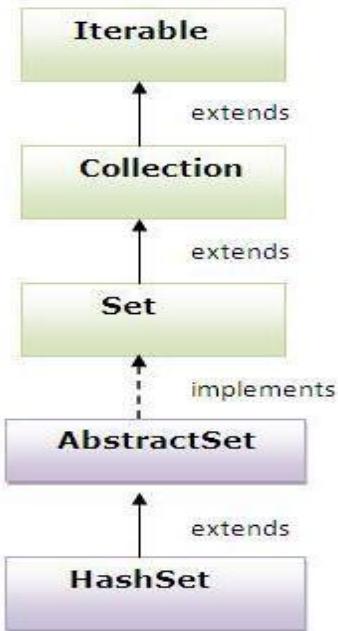
Difference between List and Set:

List can contain duplicate elements whereas Set contains unique elements only.

HashSet class:

- uses hashtable to store the elements. It extends AbstractSet class and implements Set interface.
- contains unique elements only.

Hierarchy of HashSet class:



Example of HashSet class:

```
1. import java.util.*;
2. class Simple{
3.     public static void main(String args[]){
4.
5.         HashSet al=new HashSet();
6.         al.add("Ravi");
7.         al.add("Vijay");
8.         al.add("Ravi");
9.         al.add("Ajay");
10.
11.        Iterator itr=al.iterator();
12.        while(itr.hasNext()){
13.            System.out.println(itr.next());
14.        }
15.    }
16. }
```

Output:Ajay

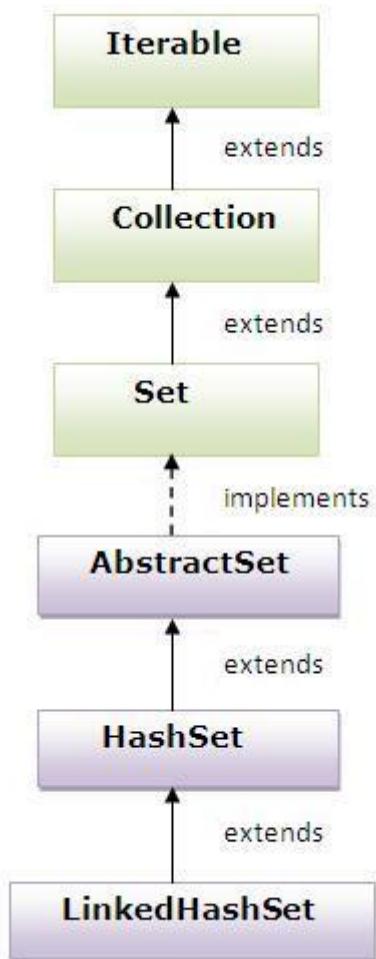
Vijay

Ravi

LinkedHashSet class:

- contains unique elements only like HashSet. It extends HashSet class and implements Set interface.
- maintains insertion order.

Hierarchy of LinkedHashSet class:



Example of LinkedHashSet class:

```
1. import java.util.*;
2. class Simple{
3.     public static void main(String args[]){
4.
5.         LinkedHashSet al=new LinkedHashSet();
6.         al.add("Ravi");
7.         al.add("Vijay");
8.         al.add("Ravi");
9.         al.add("Ajay");
10.
11.        Iterator itr=al.iterator();
```

```
12. while(itr.hasNext()){  
13.     System.out.println(itr.next());  
14. }  
15. }  
16. }
```

Output:Ravi

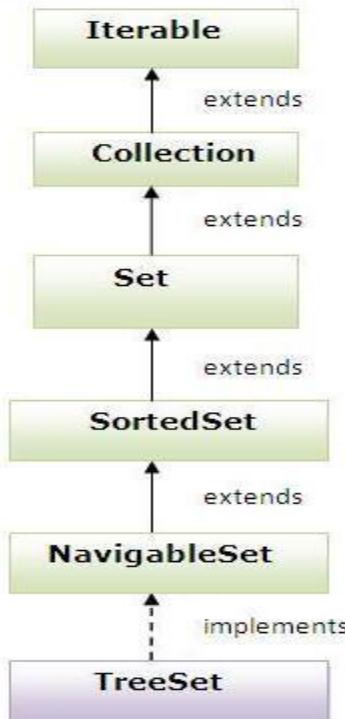
Vijay

Ajay

TreeSet class:

- contains unique elements only like HashSet. The TreeSet class implements NavigableSet interface that extends the SortedSet interface.
- maintains ascending order.

Hierarchy of TreeSet class:



Example of TreeSet class:

```
1. import java.util.*;  
2. class Simple{  
3.     public static void main(String args[]){  
4.           
5.         TreeSet al=new TreeSet();  
6.         al.add("Ravi");
```

Core Java

```
7. al.add("Vijay");
8. al.add("Ravi");
9. al.add("Ajay");
10.
11. Iterator itr=al.iterator();
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }
```

Output:Ajay

Ravi

Vijay

Queue Interface:

The Queue interface basically orders the element in FIFO(First In First Out)manner.

Methods of Queue Interface :

1. public boolean add(object);
2. public boolean offer(object);
3. public remove();
4. public poll();
5. public element();
6. public peek();

PriorityQueue class:

The PriorityQueue class provides the facility of using queue. But it does not orders the elements in FIFO manner.

Example of PriorityQueue:

```
1. import java.util.*;
2. class Simple8{
3. public static void main(String args[]){
4.
5. PriorityQueue queue=new PriorityQueue();
6. queue.add("Amit");
7. queue.add("Vijay");
8. queue.add("Karan");
9. queue.add("Jai");
```

Core Java

```
10. queue.add("Rahul");
11.
12. System.out.println("head:"+queue.element());
13. System.out.println("head:"+queue.peek());
14.
15. System.out.println("iterating the queue elements:");
16. Iterator itr=queue.iterator();
17. while(itr.hasNext()){
18. System.out.println(itr.next());
19. }
20.
21. queue.remove();
22. queue.poll();
23.
24. System.out.println("after removing two elements:");
25. Iterator itr2=queue.iterator();
26. while(itr2.hasNext()){
27. System.out.println(itr2.next());
28. }
29.
30. }
31. }
```

Output:head:Amit

```
head:Amit
iterating the queue elements:
Amit
Jai
Karan
Vijay
Rahul
after removing two elements:
Karan
Rahul
Vijay
```

Map Interface

A map contains values based on the key i.e. key and value pair. Each pair is known as an entry. Map contains only unique elements.

Commonly used methods of Map interface:

1. **public Object put(object key, Object value):** is used to insert an entry in this map.
2. **public void putAll(Map map):** is used to insert the specified map in this map.

Core Java

3. **public Object remove(Object key):** is used to delete an entry for the specified key.
4. **public Object get(Object key):** is used to return the value for the specified key.
5. **public boolean containsKey(Object key):** is used to search the specified key from this map.
6. **public boolean containsValue(Object value):** is used to search the specified value from this map.
7. **public Set keySet():** returns the Set view containing all the keys.
8. **public Set entrySet():** returns the Set view containing all the keys and values.

Entry

Entry is the subinterface of Map. So we will access it by Map.Entry name. It provides methods to get key and value.

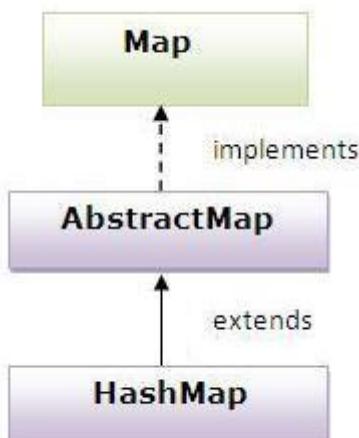
Methods of Entry interface:

1. **public Object getKey():** is used to obtain key.
2. **public Object getValue():** is used to obtain value.

HashMap class:

- A HashMap contains values based on the key. It implements the Map interface and extends AbstractMap class.
- It contains only unique elements.
- It may have one null key and multiple null values.
- It maintains no order.

Hierarchy of HashMap class:



Example of HashMap class:

```
1. import java.util.*;
2. class Simple{
3.     public static void main(String args[]){
4.
5.     HashMap hm=new HashMap();
6.
7.     hm.put(100,"Amit");
8.     hm.put(101,"Vijay");
9.     hm.put(102,"Rahul");
10.
11.    Set set=hm.entrySet();
12.    Iterator itr=set.iterator();
13.
14.    while(itr.hasNext()){
15.        Map.Entry m=(Map.Entry)itr.next();
16.        System.out.println(m.getKey()+" "+m.getValue());
17.    }
18. }
19. }
```

Output:102 Rahul

 100 Amit

 101 Vijay

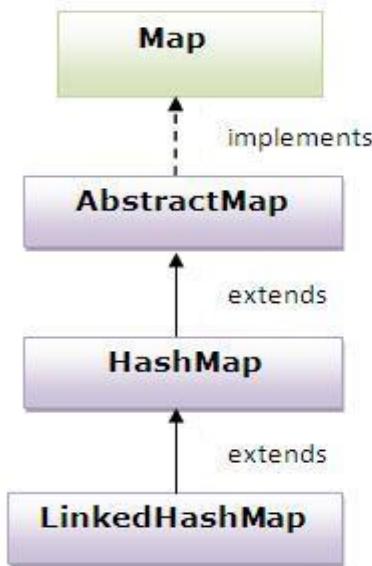
What is difference between HashSet and HashMap?

HashSet contains only values whereas HashMap contains entry(key and value).

LinkedHashMap class:

- A LinkedHashMap contains values based on the key. It implements the Map interface and extends HashMap class.
- It contains only unique elements.
- It may have one null key and multiple null values.
- It is same as HashMap instead maintains insertion order.

Hierarchy of LinkedHashMap class:



Example of LinkedHashMap class:

```
1. import java.util.*;
2. class Simple{
3. public static void main(String args[]){
4.
5.     LinkedHashMap hm=new LinkedHashMap();
6.
7.     hm.put(100,"Amit");
8.     hm.put(101,"Vijay");
9.     hm.put(102,"Rahul");
10.
11.    Set set=hm.entrySet();
12.    Iterator itr=set.iterator();
13.
14.    while(itr.hasNext()){
15.        Map.Entry m=(Map.Entry)itr.next();
16.        System.out.println(m.getKey()+" "+m.getValue());
17.    }
18. }
19. }
```

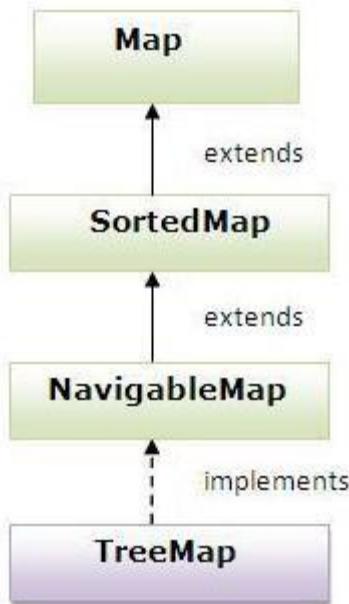
Output:100 Amit

101 Vijay
103 Rahul

TreeMap class

- A TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
- It contains only unique elements.
- It cannot have null key but can have multiple null values.
- It is same as HashMap instead maintains ascending order.

Hierarchy of TreeMap class:



Example of TreeMap class:

```
1. import java.util.*;
2. class Simple{
3. public static void main(String args[]){
4.
5.     TreeMap hm=new TreeMap();
6.
7.     hm.put(100,"Amit");
8.     hm.put(102,"Ravi");
9.     hm.put(101,"Vijay");
10.    hm.put(103,"Rahul");
11.
12.    Set set=hm.entrySet();
13.    Iterator itr=set.iterator();
14.
15.    while(itr.hasNext()){
16.        Map.Entry m=(Map.Entry)itr.next();
17.        System.out.println(m.getKey()+" "+m.getValue());
18.    }
```

Core Java

```
19. }
20. }
```

Output:100 Amit

101 Vijay
102 Ravi
103 Rahul

What is difference between HashMap and TreeMap?

1) HashMap is can contain one null key.	TreeMap connot contain any null key.
2) HashMap maintains no order.	TreeMap maintains ascending order.

Hashtable

- A Hashtable is an array of list. Each list is known as a bucket. The position of bucket is identified by calling the hashCode() method. A Hashtable contains values based on the key. It implements the Map interface and extends Dictionary class.
- It contains only unique elements.
- It may have not have any null key or value.
- It is synchronized.

Example of Hashtable:

```
1. import java.util.*;
2. class Simple{
3. public static void main(String args[]){
4.
5.     Hashtable hm=new Hashtable();
6.
7.     hm.put(100,"Amit");
8.     hm.put(102,"Ravi");
9.     hm.put(101,"Vijay");
10.    hm.put(103,"Rahul");
11.
12.    Set set=hm.entrySet();
13.    Iterator itr=set.iterator();
14.
15.    while(itr.hasNext()){
16.        Map.Entry m=(Map.Entry)itr.next();
17.        System.out.println(m.getKey()+" "+m.getValue());
18.    }
19. }
20. }
```

Core Java

Output:103 Rahul

102 Ravi

101 Vijay

100 Amit

What is difference between HashMap and Hashtable?

1) HashMap is not synchronized.	Hashtable is synchronized.
2) HashMap can contain one null key and multiple null values.	Hashtable cannot contain any null key nor value.

Sorting

We can sort the elements of:

1. String objects
2. Wrapper class objects
3. User-defined class objects

Collections class provides static methods for sorting the elements of collection. If collection elements are of Set type, we can use TreeSet. But We cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements.

Method of Collections class for sorting List elements

public void sort(List list): is used to sort the elements of List. List elements must be of Comparable type.

Note: String class and Wrapper classes implements the Comparable interface. So if you store the objects of string or wrapper classes, it will be Comparable.

Example of Sorting the elements of List that contains string objects

```
1. import java.util.*;
2. class Simple12{
3. public static void main(String args[]){
4.
5. ArrayList al=new ArrayList();
6. al.add("Viru");
7. al.add("Saurav");
8. al.add("Mukesh");
```

Core Java

```
9. al.add("Tahir");
10.
11. Collections.sort(al);
12. Iterator itr=al.iterator();
13. while(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. }
17. }
```

Output:Mukesh

Saurav

Tahir

Viru

Example of Sorting the elements of List that contains Wrapper class objects

```
1. import java.util.*;
2. class Simple12{
3. public static void main(String args[]){
4.
5. ArrayList al=new ArrayList();
6. al.add(Integer.valueOf(201));
7. al.add(Integer.valueOf(101));
8. al.add(230);//internally will be converted into objects as Integer.valueOf(230)
9.
10. Collections.sort(al);
11.
12. Iterator itr=al.iterator();
13. while(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. }
17. }
```

Output:101

201

230

Comparable interface

Comparable interface is used to order the objects of user-defined class. This interface is found in `java.lang` package and contains only one method named `compareTo(Object)`. It provides only single sorting sequence i.e. you can sort the elements based on a single data member only. For instance it may be either rollno, name, age or anything else.

Core Java

Syntax:

public int compareTo(Object obj): is used to compare the current object with the specified object.

We can sort the elements of:

1. String objects
2. Wrapper class objects
3. User-defined class objects

Collections class provides static methods for sorting the elements of collection. If collection elements are of Set type, we can use TreeSet. But We cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements.

Method of Collections class for sorting List elements

public void sort(List list): is used to sort the elements of List. List elements must be of Comparable type.

Note: String class and Wrapper classes implements the Comparable interface. So if you store the objects of string or wrapper classes, it will be Comparable.

Example of Sorting the elements of List that contains user-defined class objects on age basis

Student.java

```
1. class Student implements Comparable{  
2.     int rollno;  
3.     String name;  
4.     int age;  
5.     Student(int rollno, String name, int age){  
6.         this.rollno=rollno;  
7.         this.name=name;  
8.         this.age=age;  
9.     }  
10.  
11.    public int compareTo(Object obj){  
12.        Student st=(Student)obj;  
13.        if(age==st.age)  
14.            return 0;  
15.        else if(age>st.age)  
16.            return 1;  
17.        else  
18.            return -1;  
19.    }
```

20.
21. }

Simple.java

```
1. import java.util.*;
2. import java.io.*;
3.
4. class Simple{
5. public static void main(String args[]){
6.
7. ArrayList al=new ArrayList();
8. al.add(new Student(101,"Vijay",23));
9. al.add(new Student(106,"Ajay",27));
10. al.add(new Student(105,"Jai",21));
11.
12. Collections.sort(al);
13. Iterator itr=al.iterator();
14. while(itr.hasNext()){
15. Student st=(Student)itr.next();
16. System.out.println(st.rollno+" "+st.name+" "+st.age);
17. }
18. }
19. }
```

Output:105 Jai 21

101 Vijay 23

106 Ajay 27

Comparator interface

Comparator interface is used to order the objects of user-defined class.

This interface is found in `java.util` package and contains 2 methods `compare(Object obj1, Object obj2)` and `equals(Object element)`.

It provides multiple sorting sequence i.e. you can sort the elements based on any data member. For instance it may be on rollno, name, age or anything else.

Syntax of compare method

public int compare(Object obj1, Object obj2): compares the first object with second object.

Collections class provides static methods for sorting the elements of collection. If collection elements are of Set type, we can use `TreeSet`. But We cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements.

Core Java

Method of Collections class for sorting List elements

public void sort(List list,Comparator c): is used to sort the elements of List by the given comparator.

Example of sorting the elements of List that contains user-defined class objects on the basis of age and name

In this example, we have created 4 java classes:

- Student.java
- AgeComparator.java
- NameComparator.java
- Simple.java

Student.java

This class contains three fields rollno, name and age and a parameterized constructor.

```
1. class Student{  
2.     int rollno;  
3.     String name;  
4.     int age;  
5.     Student(int rollno,String name,int age){  
6.         this.rollno=rollno;  
7.         this.name=name;  
8.         this.age=age;  
9.     }  
10. }
```

AgeComparator.java

This class defines comparison logic based on the age. If age of first object is greater than the second, we are returning positive value, it can be any one such as 1, 2 , 10 etc. If age of first object is less than the second object, we are returning negative value, it can be any negative value and if age of both objects are equal, we are returning 0.

```
1. import java.util.*;  
2. class AgeComparator implements Comparator{  
3.     public int Compare(Object o1,Object o2){  
4.         Student s1=(Student)o1;  
5.         Student s2=(Student)o2;  
6.         if(s1.age==s2.age)  
7.             return 0;  
8.         else if(s1.age>s2.age)
```

Core Java

```
10. return 1;
11. else
12. return -1;
13. }
14. }
```

NameComparator.java

This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.

```
1. import java.util.*;
2. class NameComparator implements Comparator{
3. public int Compare(Object o1, Object o2){
4. Student s1=(Student)o1;
5. Student s2=(Student)o2;
6.
7. return s1.name.compareTo(s2.name);
8. }
9. }
```

Simple.java

In this class, we are printing the objects values by sorting on the basis of name and age.

```
1. import java.util.*;
2. import java.io.*;
3.
4. class Simple{
5. public static void main(String args[]){
6.
7. ArrayList al=new ArrayList();
8. al.add(new Student(101,"Vijay",23));
9. al.add(new Student(106,"Ajay",27));
10. al.add(new Student(105,"Jai",21));
11.
12. System.out.println("Sorting by Name...");
13.
14. Collections.sort(al,new NameComparator());
15. Iterator itr=al.iterator();
16. while(itr.hasNext()){
17. Student st=(Student)itr.next();
18. System.out.println(st.rollno+" "+st.name+" "+st.age);
19. }
20.
21. System.out.println("sorting by age...");
22.
23. Collections.sort(al,new AgeComparator());
24. Iterator itr2=al.iterator();
25. while(itr2.hasNext()){
26. Student st=(Student)itr2.next();
```

Core Java

```
27. System.out.println(st.rollno+" "+st.name+" "+st.age);
28. }
29.
30.
31. }
32. }
```

Output:Sorting by Name...

```
106 Ajay 27
105 Jai 21
101 Vijay 23
Sorting by age...
105 Jai 21
101 Vijay 23
106 Ajay 27
```

Properties class in Java

The **properties** object contains key and value pair both as a string. It is the subclass of Hashtable.

It can be used to get property value based on the property key. The Properties class provides methods to get data from properties file and store data into properties file. Moreover, it can be used to get properties of system.

Advantage of properties file

Easy Maintenance: If any information is changed from the properties file, you don't need to recompile the java class. It is mainly used to contain variable information i.e. to be changed.

Methods of Properties class

The commonly used methods of Properties class are given below.

Method	Description
public void load(Reader r)	loads data from the Reader object.
public void load(InputStream is)	loads data from the InputStream object
public String getProperty(String key)	returns value based on the key.

Core Java

public void setProperty(String key, String value)	sets the property in the properties object.
public void store(Writer w, String comment)	writers the properties in the writer object.
public void store(OutputStream os, String comment)	writes the properties in the OutputStream object.
storeToXML(OutputStream os, String comment)	writers the properties in the writer object for generating xml document.
public void storeToXML(Writer w, String comment, String encoding)	writers the properties in the writer object for generating xml document with specified encoding.

Example of Properties class to get information from properties file

To get information from the properties file, create the properties file first.

db.properties

1. user=system
2. password=oracle

Now, lets create the java class to read the data from the properties file.

Test.java

```
1. import java.util.*;
2. import java.io.*;
3. public class Test {
4.     public static void main(String[] args) throws Exception{
5.         FileReader reader=new FileReader("db.properties");
6.
7.         Properties p=new Properties();
8.         p.load(reader);
9.
10.        System.out.println(p.getProperty("user"));
11.        System.out.println(p.getProperty("password"));
12.    }
13. }
```

Output:
system
oracle

Core Java

Now if you change the value of the properties file, you don't need to compile the java class again. That means no maintenance problem.

Example of Properties class to get all the system properties

By System.getProperties() method we can get all the properties of system. Let's create the class that gets information from the system properties.

Test.java

```
1. import java.util.*;
2. import java.io.*;
3. public class Test {
4.     public static void main(String[] args) throws Exception{
5.
6.     Properties p=System.getProperties();
7.     Set set=p.entrySet();
8.
9.     Iterator itr=set.iterator();
10.    while(itr.hasNext()){
11.        Map.Entry entry=(Map.Entry)itr.next();
12.        System.out.println(entry.getKey()+" = "+entry.getValue());
13.    }
14.
15. }
16. }
```

Output:

```
java.runtime.name = Java(TM) SE Runtime Environment
sun.boot.library.path = C:\Program Files\Java\jdk1.7.0_01\jre\bin
java.vm.version = 21.1-b02
java.vm.vendor = Oracle Corporation
java.vendor.url = http://java.oracle.com/
path.separator = ;
java.vm.name = Java HotSpot(TM) Client VM
file.encoding.pkg = sun.io
user.country = US
user.script =
sun.java.launcher = SUN_STANDARD
.....
```

Example of Properties class to create properties file

Now lets write the code to create the properties file.

Test.java

```
1. import java.util.*;
2. import java.io.*;
```

Core Java

```
3. public class Test {  
4.   public static void main(String[] args)throws Exception{  
5.  
6.   Properties p=new Properties();  
7.   p.setProperty("name","Sonoo Jaiswal");  
8.   p.setProperty("email","sonoojaiswal@javatpoint.com");  
9.  
10.  p.store(new FileWriter("info.properties"),"Javatpoint Properties Example");  
11.  
12. }  
13. }
```

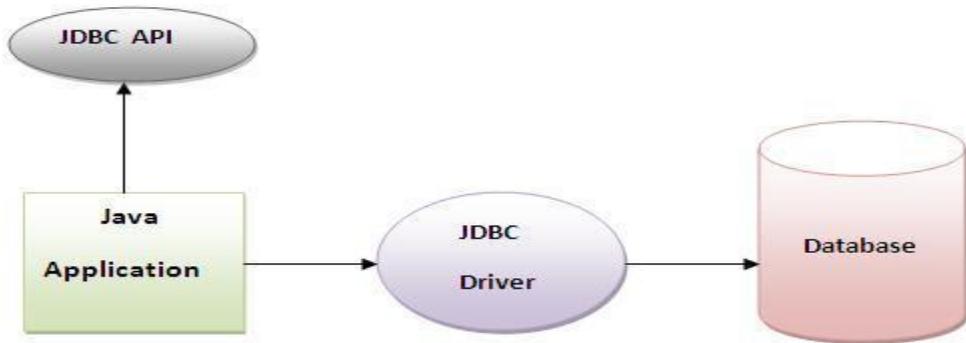
Let's see the generated properties file.

info.properties

```
1. #Javatpoint Properties Example  
2. #Thu Oct 03 22:35:53 IST 2013  
3. email=sonoojaiswal@javatpoint.com  
4. name=Sonoo Jaiswal
```

JDBC

This **JDBC** tutorial covers all the topics of JDBC with the simple examples. JDBC is a Java API that is used to connect and execute query to the database. JDBC API uses jdbc drivers to connects to the database.



Why use JDBC?

Before JDBC, ODBC API was used to connect and execute query to the database. But ODBC API uses ODBC driver that is written in C language which is platform dependent and unsecured. That is why Sun Microsystem has defined its own API (JDBC API) that uses JDBC driver written in Java language.

Do You Know ?

- How to connect Java application with Oracle and Mysql database using JDBC?
- What is the difference between Statement and PreparedStatement interface?
- How to print total numbers of tables and views of a database using JDBC ?
- How to store and retrieve images from Oracle database using JDBC?
- How to store and retrieve files from Oracle database using JDBC?

What is API?

API (Application programming interface) is a document that contains description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc

Upcoming topics in JDBC Tutorial

2) JDBC Drivers

In this JDBC tutorial, we will learn 4 types of JDBC drivers, their advantages and disadvantages.

3) 5 Steps to connect to the database

In this JDBC tutorial, we will see the 5 steps to connect to the database in java using JDBC.

4) Connectivity with Oracle using JDBC

In this JDBC tutorial, we will connect a simple java program with the oracle database.

5) Connectivity with MySQL using JDBC

In this JDBC tutorial, we will connect a simple java program with the mysql database.

6) Connectivity with Access without DSN

Let's connect java application with access database with and without DSN.

7) DriverManager class

In this JDBC tutorial, we will learn what does the DriverManager class and what are its methods.

8) Connection interface

In this JDBC tutorial, we will learn what is Connection interface and what are its methods.

Core Java

9) Statement interface

In this JDBC tutorial, we will learn what is Statement interface and what are its methods.

10) ResultSet interface

In this JDBC tutorial, we will learn what is ResultSet interface and what are its methods. Moreover, we will learn how we can make the ResultSet scrollable.

11) PreparedStatement Interface

In this JDBC tutorial, we will learn what is benefit of PreparedStatement over Statement interface. We will see examples to insert, update or delete records using the PreparedStatement interface.

12) ResultSetMetaData interface

In this JDBC tutorial, we will learn how we can get the metadata of a table.

13) DatabaseMetaData interface

In this JDBC tutorial, we will learn how we can get the metadata of a database.

14) Storing image in Oracle

Let's learn how to store image in the oracle database using JDBC.

15) Retrieving image from Oracle

Let's see the simple example to retrieve image from the oracle database using JDBC.

16) Storing file in Oracle

Let's see the simple example to store file in the oracle database using JDBC.

Core Java

17) Retrieving file from Oracle

Let's see the simple example to retrive file from the oracle database using JDBC.

JDBC Driver

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

1) JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.

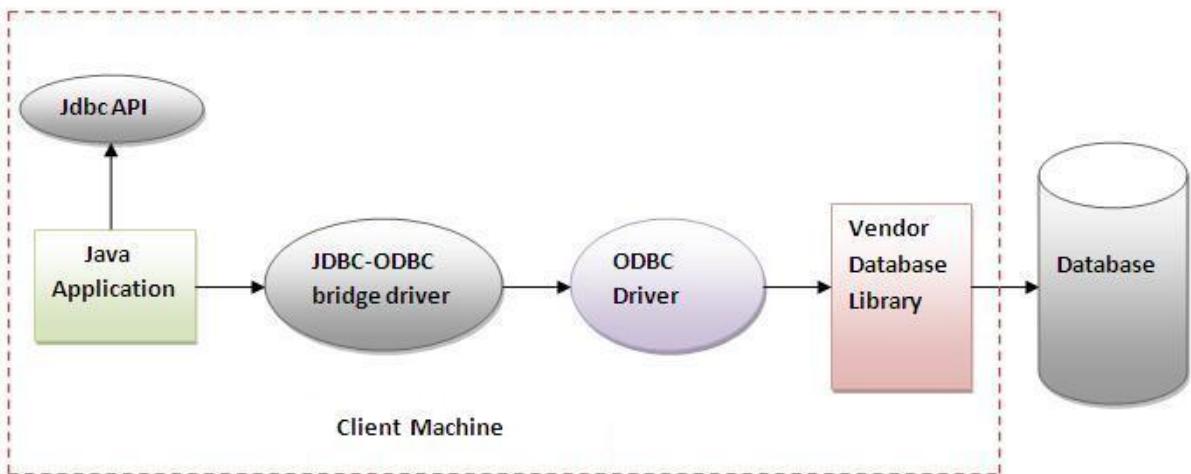


Figure- JDBC-ODBC Bridge Driver

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

Core Java

2) Native-API driver

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.

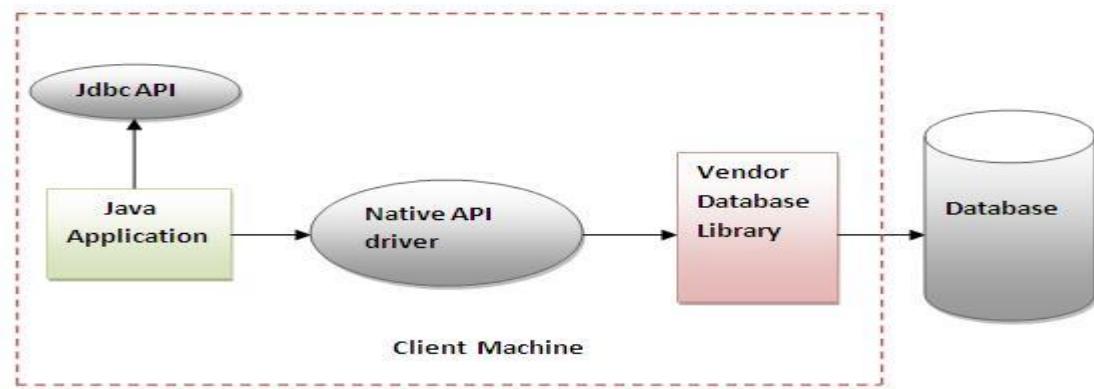


Figure- Native API Driver

Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

3) Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

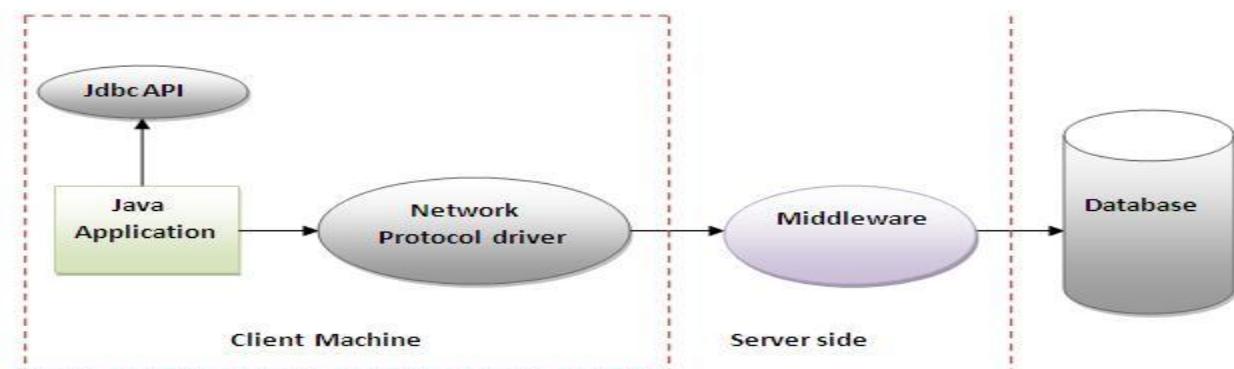


Figure- Network Protocol Driver

Core Java

Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

4) Thin driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

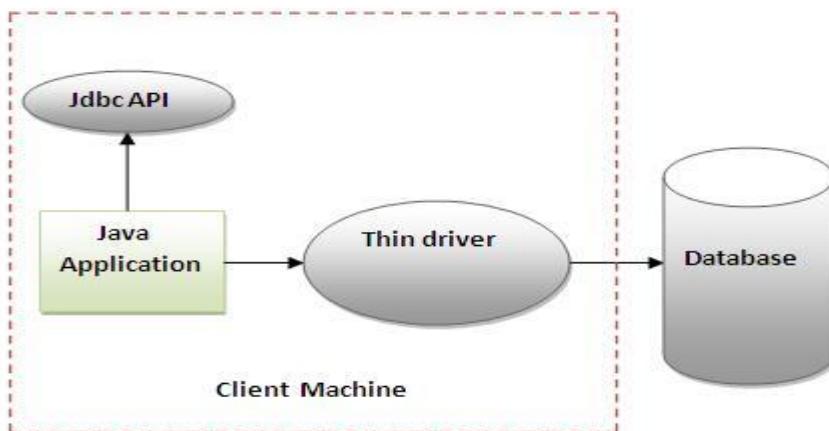


Figure- Thin Driver

Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage:

- Drivers depends on the Database.

5 Steps to connect to the database in java

1) Register the driver class

The `forName()` method of `Class` class is used to register the driver class. This method is used to dynamically load the driver class.

Syntax of `forName()` method

1. `public static void forName(String className) throws ClassNotFoundException`

Example to register the `OracleDriver` class

1. `Class.forName("oracle.jdbc.driver.OracleDriver");`

2) Create the connection object

The `getConnection()` method of `DriverManager` class is used to establish connection with the database.

Syntax of `getConnection()` method

- 1) `public static Connection getConnection(String url) throws SQLException`
- 2) `public static Connection getConnection(String url, String name, String password)`
3. `throws SQLException`

Example to establish connection with the Oracle database

1. `Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","password");`

3) Create the Statement object

The `createStatement()` method of `Connection` interface is used to create statement. The object of statement is responsible to execute queries with the database.

Syntax of `createStatement()` method

1. `public Statement createStatement() throws SQLException`

Example to create the statement object

1. `Statement stmt=con.createStatement();`

4) Execute the query

The `executeQuery()` method of `Statement` interface is used to execute queries to the database. This method returns the object of `ResultSet` that can be used to get all the

Core Java

records of a table.

Syntax of executeQuery() method

1. `public ResultSet executeQuery(String sql) throws SQLException`

Example to execute query

1. `ResultSet rs=stmt.executeQuery("select * from emp");`
- 2.
3. `while(rs.next()){`
4. `System.out.println(rs.getInt(1)+" "+rs.getString(2));`
5. }

5) Close the connection object

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

Syntax of close() method

1. `public void close() throws SQLException`

Example to close connection

1. `con.close();`

Example to connect to the Oracle database

For connecting java application with the oracle database, you need to follow 5 steps to perform database connectivity. In this example we are using Oracle10g as the database. So we need to know following informations for the oracle database:

1. **Driver class:** The driver class for the oracle database is `oracle.jdbc.driver.OracleDriver`.
2. **Connection URL:** The connection URL for the oracle10G database is `jdbc:oracle:thin:@localhost:1521:xe` where jdbc is the API, oracle is the database, thin is the driver, localhost is the server name on which oracle is running, we may also use IP address, 1521 is the port number and XE is the Oracle service name. You may get all these informations from the tnsnames.ora file.
3. **Username:** The default username for the oracle database is `system`.
4. **Password:** Password is given by the user at the time of installing the oracle database.

Core Java

Let's first create a table in oracle database.

1. create table emp(id number(10),name varchar2(40),age number(3));

Example to Connect Java Application with Oracle database

In this example, system is the username and oracle is the password of the Oracle database.

```
1. import java.sql.*;
2. class OracleCon{
3. public static void main(String args[]){
4. try{
5. //step1 load the driver class
6. Class.forName("oracle.jdbc.driver.OracleDriver");
7.
8. //step2 create the connection object
9. Connection con=DriverManager.getConnection(
10. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
11.
12. //step3 create the statement object
13. Statement stmt=con.createStatement();
14.
15. //step4 execute query
16. ResultSet rs=stmt.executeQuery("select * from emp");
17. while(rs.next())
18. System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3));
19.
20. //step5 close the connection object
21. con.close();
22.
23. }catch(Exception e){ System.out.println(e); }
24.
25. }
26. }
```

The above example will fetch all the records of emp table.

To connect java application with the Oracle database ojdbc14.jar file is required to be loaded.

Two ways to load the jar file:

1. paste the ojdbc14.jar file in jre/lib/ext folder
2. set classpath

Core Java

1) paste the ojdbc14.jar file in JRE/lib/ext folder:

Firstly, search the ojdbc14.jar file then go to JRE/lib/ext folder and paste the jar file here.

2) set classpath:

There are two ways to set the classpath:

- temporary
- permanent

How to set the temporary classpath:

Firstly, search the ojdbc14.jar file then open command prompt and write:

1. C:>set classpath=c:\folder\ojdbc14.jar;;

How to set the permanent classpath:

Go to environment variable then click on new tab. In variable name write **classpath** and in variable value paste the path to ojdbc14.jar by appending ojdbc14.jar;; as C:\oraclexe\app\oracle\product\10.2.0\server\jdbc\lib\ojdbc14.jar;;

To see the slides of setting permanent path.

Example to connect to the mysql database

For connecting java application with the mysql database, you need to follow 5 steps to perform database connectivity.

In this example we are using MySql as the database. So we need to know following informations for the mysql database:

1. **Driver class:** The driver class for the mysql database is **com.mysql.jdbc.Driver**.
2. **Connection URL:** The connection URL for the mysql database is **jdbc:mysql://localhost:3306/sonoo** where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and sonoo is the database name. We may use any database, in such case, you need to replace the sonoo with your database name.
3. **Username:** The default username for the mysql database is **root**.
4. **Password:** Password is given by the user at the time of installing the mysql database. In this example, we are going to use root as the password.

Let's first create a table in the mysql database, but before creating table, we need to create database first.

Core Java

1. create database sonoo;
2. use sonoo;
3. create table emp(id int(10),name varchar(40),age int(3));

Example to Connect Java Application with mysql database

In this example, sonoo is the database name, root is the username and password.

```
1. import java.sql.*;  
2. class MysqlCon{  
3. public static void main(String args[]){  
4. try{  
5. Class.forName("com.mysql.jdbc.Driver");  
6.  
7. Connection con=DriverManager.getConnection(  
8. "jdbc:mysql://localhost:3306/sonoo","root","root");  
9.  
10. //here sonoo is database name, root is username and password  
11.  
12. Statement stmt=con.createStatement();  
13.  
14. ResultSet rs=stmt.executeQuery("select * from emp");  
15.  
16. while(rs.next())  
17. System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3));  
18.  
19. con.close();  
20.  
21. }catch(Exception e){ System.out.println(e);}  
22.  
23. }  
24. }
```

The above example will fetch all the records of emp table.

To connect java application with the mysql database mysqlconnector.jar file is required to be loaded.

Two ways to load the jar file:

1. paste the mysqlconnector.jar file in jre/lib/ext folder
2. set classpath

Core Java

1) paste the mysqlconnector.jar file in JRE/lib/ext folder:

Download the mysqlconnector.jar file. Go to jre/lib/ext folder and paste the jar file here.

2) set classpath:

There are two ways to set the classpath:

- temporary
- permanent

How to set the temporary classpath

open command prompt and write:

```
1. C:>set classpath=c:\folder\mysql-connector-java-5.0.8-bin.jar;;;
```

How to set the permanent classpath

Go to environment variable then click on new tab. In variable name write **classpath** and in variable value paste the path to the mysqlconnector.jar file by appending mysqlconnector.jar;;; as C:\folder\mysql-connector-java-5.0.8-bin.jar;;;

Connectivity with Access without DSN

There are two ways to connect java application with the access database.

1. Without DSN (Data Source Name)
2. With DSN

Java is mostly used with Oracle, mysql, or DB2 database. So you can learn this topic only for knowledge.

Example to Connect Java Application with access without DSN

In this example, we are going to connect the java program with the access database. In such case, we have created the login table in the access database. There is only one column in the table named name. Let's get all the name of the login table.

```
1. import java.sql.*;
2. class Test{
3. public static void main(String ar[]){
4. try{
5. String database="student.mdb";//Here database exists in the current directory
6.
7. String url="jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};
8. DBQ=" + database + ";DriverID=22;READONLY=true";
```

Core Java

```
9.  
10. Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  
11. Connection c=DriverManager.getConnection(url);  
12. Statement st=c.createStatement();  
13. ResultSet rs=st.executeQuery("select * from login");  
14.  
15. while(rs.next()){  
16.     System.out.println(rs.getString(1));  
17. }  
18.  
19. }catch(Exception ee){System.out.println(ee);}  
20.  
21. {}
```

Example to Connect Java Application with access with DSN

Connectivity with type1 driver is not considered good. To connect java application with type1 driver, create DSN first, here we are assuming your dsn name is mydsn.

```
1. import java.sql.*;  
2. class Test{  
3. public static void main(String ar[]){  
4.     try{  
5.         String url="jdbc:odbc:mydsn";  
6.         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  
7.         Connection c=DriverManager.getConnection(url);  
8.         Statement st=c.createStatement();  
9.         ResultSet rs=st.executeQuery("select * from login");  
10.  
11.     while(rs.next()){  
12.         System.out.println(rs.getString(1));  
13.     }  
14.  
15. }catch(Exception ee){System.out.println(ee);}  
16.  
17. {}}
```

DriverManager class:

The DriverManager class acts as an interface between user and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method DriverManager.registerDriver().

Core Java

Commonly used methods of DriverManager class:

1) public static void registerDriver(Driver driver);	is used to register the given driver with DriverManager.
2) public static void deregisterDriver(Driver driver);	is used to deregister the given driver (drop the driver from the list) with DriverManager.
3) public static Connection getConnection(String url);	is used to establish the connection with the specified url.
4) public static Connection getConnection(String url, String userName, String password);	is used to establish the connection with the specified url, username and password.

Connection interface:

A Connection is the session between java application and database. The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData i.e. object of Connection can be used to get the object of Statement and DatabaseMetaData. The Connection interface provide many methods for transaction management like commit(), rollback() etc.

By default, connection commits the changes after executing queries.

Commonly used methods of Connection interface:

1) public Statement createStatement(): creates a statement object that can be used to execute SQL queries.

2) public Statement createStatement(int resultSetType,int resultSetConcurrency): Creates a Statement object that will generate ResultSet objects with the given type and concurrency.

3) public void setAutoCommit(boolean status): is used to set the commit status. By default it is true.

4) public void commit(): saves the changes made since the previous commit/rollback

Core Java

permanent.

5) public void rollback(): Drops all changes made since the previous commit/rollback.

6) public void close(): closes the connection and Releases a JDBC resources immediately.

Statement interface

The **Statement interface** provides methods to execute queries with the database. The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

Commonly used methods of Statement interface:

The important methods of Statement interface are as follows:

1) public ResultSet executeQuery(String sql): is used to execute SELECT query. It returns the object of ResultSet.

2) public int executeUpdate(String sql): is used to execute specified query, it may be create, drop, insert, update, delete etc.

3) public boolean execute(String sql): is used to execute queries that may return multiple results.

4) public int[] executeBatch(): is used to execute batch of commands.

Example of Statement interface

Let's see the simple example of Statement interface to insert, update and delete the record.

```
1. import java.sql.*;
2. class FetchRecord{
3.     public static void main(String args[])throws Exception{
4.
5.         Class.forName("oracle.jdbc.driver.OracleDriver");
6.         Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
7.         Statement stmt=con.createStatement();
8.
9.         //stmt.executeUpdate("insert into emp765 values(33,'Irfan',50000)");
10.        //int result=stmt.executeUpdate("update emp765 set name='Vimal',salary=10000 where id=3
3");
```

Core Java

```
11. int result=stmt.executeUpdate("delete from emp765 where id=33");
12.
13. System.out.println(result+" records affected");
14. con.close();
15. } }
```

ResultSet interface

The object of `ResultSet` maintains a cursor pointing to a particular row of data. Initially, cursor points to before the first row.

By default, ResultSet object can be moved forward only and it is not updatable.

But we can make this object to move forward and backward direction by passing either TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE in createStatement(int,int) method as well as we can make this object as updatable by:

1) public boolean next():	is used to move the cursor to the one row next from the current position.
2) public boolean previous():	is used to move the cursor to the one row previous from the current position.
3) public boolean first():	is used to move the cursor to the first row in result set object.
4) public boolean last():	is used to move the cursor to the last row in result set object.
5) public boolean absolute(int row):	is used to move the cursor to the specified row number in the ResultSet object.
6) public boolean relative(int row):	is used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative.
7) public int getInt(int columnIndex):	is used to return the data of specified column index of the current row as int.

Core Java

8) public int getInt(String columnName):	is used to return the data of specified column name of the current row as int.
9) public String getString(int columnIndex):	is used to return the data of specified column index of the current row as String.
10) public String getString(String columnName):	is used to return the data of specified column name of the current row as String.

Commonly used methods of ResultSet interface

Example of Scrollable ResultSet

Let's see the simple example of ResultSet interface to retrieve the data of 3rd row.

```
1. import java.sql.*;
2. class FetchRecord{
3.     public static void main(String args[])throws Exception{
4.
5.     Class.forName("oracle.jdbc.driver.OracleDriver");
6.     Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
7.     Statement stmt=con.createStatement	ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);
8.     ResultSet rs=stmt.executeQuery("select * from emp765");
9.
10. //getting the record of 3rd row
11. rs.absolute(3);
12. System.out.println(rs.getString(1)+" "+rs.getString(2)+" "+rs.getString(3));
13.
14. con.close();
15. }}
```

PreparedStatement:

The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.

Why use PreparedStatement?

The performance of the application will be faster if you use PreparedStatement interface because query is compiled only once.

The prepareStatement() method of Connection interface is used to return the object of PreparedStatement.Syntax:

1. **public** PreparedStatement prepareStatement(String query)**throws** SQLException{ }
1. create table emp(id number(10),name varchar2(50));

Example of PreparedStatement interface that inserts the record:

```
1. import java.sql.*;
2. class InsertPrepared{
3. public static void main(String args[]){
4. try{
5. Class.forName("oracle.jdbc.driver.OracleDriver");
6.
7. Connection con=DriverManager.getConnection(
8. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
9.
10. PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");
11. stmt.setInt(1,101);//1 specifies the first parameter in the query
12. stmt.setInt(2,"Ratan");
13.
14. int i=stmt.executeUpdate();
15. System.out.println(i+" records inserted");
16.
17. con.close();
18.
19. }catch(Exception e){ System.out.println(e);}
20.
21. }
22. }
```

Core Java

Example of PreparedStatement interface that updates the record:

```
1. import java.sql.*;
2. class UpdatePrepared{
3. public static void main(String args[]){
4. try{
5. Class.forName("oracle.jdbc.driver.OracleDriver");
6.
7. Connection con=DriverManager.getConnection(
8. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
9.
10. PreparedStatement stmt=con.prepareStatement("update emp set name=? where id=?");
11. stmt.setString(1,"Sonoo");//1 specifies the first parameter in the query i.e. name
12. stmt.setInt(2,101);
13.
14. int i=stmt.executeUpdate();
15. System.out.println(i+" records updated");
16.
17. con.close();
18.
19. }catch(Exception e){ System.out.println(e);}
20.
21. }
22. }
```

Example of PreparedStatement interface that deletes the record:

```
1. import java.sql.*;
2. class DeletePrepared{
3. public static void main(String args[]){
4. try{
5. Class.forName("oracle.jdbc.driver.OracleDriver");
6.
7. Connection con=DriverManager.getConnection(
8. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
9.
10. PreparedStatement stmt=con.prepareStatement("delete from emp where id=?");
11. stmt.setInt(1,101);
12.
13. int i=stmt.executeUpdate();
14. System.out.println(i+" records deleted");
15.
16. con.close();
17.
18. }catch(Exception e){ System.out.println(e);}
19.
20. }
21. }
```

Example of PreparedStatement interface that retrieve the records of a table:

```
1. import java.sql.*;  
2. class RetrievePrepared{  
3. public static void main(String args[]){  
4. try{  
5. Class.forName("oracle.jdbc.driver.OracleDriver");  
6.  
7. Connection con=DriverManager.getConnection(  
8. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");  
9.  
10. PreparedStatement stmt=con.prepareStatement("select * from emp");  
11. ResultSet rs=stmt.executeQuery();  
12. while(rs.next()){  
13. System.out.println(rs.getInt(1)+" "+rs.getString(2));  
14. }  
15.  
16. con.close();  
17.  
18. }catch(Exception e){ System.out.println(e);}  
19.  
20. }  
21. }
```

DatabaseMetaData interface:

DatabaseMetaData interface provides methods to get meta data of a database such as database product name, database product version, driver name, name of total number of tables, name of total number of views etc.

Commonly used methods of DatabaseMetaData interface

- **public String getDriverName()throws SQLException:** it returns the name of the JDBC driver.
- **public String getDriverVersion()throws SQLException:** it returns the version number of the JDBC driver.
- **public String getUserName()throws SQLException:** it returns the username of the database.
- **public String getDatabaseProductName()throws SQLException:** it returns the product name of the database.
- **public String getDatabaseProductVersion()throws SQLException:** it returns the product version of the database.

Core Java

- **public ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types) throws SQLException:** it returns the description of the tables of the specified catalog. The table type can be TABLE, VIEW, ALIAS, SYSTEM TABLE, SYNONYM etc.

How to get the object of DatabaseMetaData:

The getMetaData() method of Connection interface returns the object of DatabaseMetaData. Syntax:

1. **public DatabaseMetaData getMetaData() throws SQLException**

Simple Example of DatabaseMetaData interface :

```
1. import java.sql.*;  
2. class Dbmd{  
3. public static void main(String args[]){  
4. try{  
5. Class.forName("oracle.jdbc.driver.OracleDriver");  
6.  
7. Connection con=DriverManager.getConnection(  
8. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");  
9.  
10. DatabaseMetaData dbmd=con.getMetaData();  
11.  
12. System.out.println("Driver Name: "+dbmd.getDriverName());  
13. System.out.println("Driver Version: "+dbmd.getDriverVersion());  
14. System.out.println("UserName: "+dbmd.getUserName());  
15. System.out.println("Database Product Name: "+dbmd.getDatabaseProductName());  
16. System.out.println("Database Product Version: "+dbmd.getDatabaseProductVersion());  
17.  
18. con.close();  
19.  
20. }catch(Exception e){ System.out.println(e);}  
21.  
22. }  
23. }
```

Output:
Driver Name: Oracle JDBC Driver
Driver Version: 10.2.0.1.0XE
Database Product Name: Oracle
Database Product Version: Oracle Database 10g Express Edition
Release 10.2.0.1.0 -Production

Example of DatabaseMetaData interface that prints total number of tables :

1. **import java.sql.*;**

Core Java

```
2. class Dbmd2{
3.     public static void main(String args[]){
4.         try{
5.             Class.forName("oracle.jdbc.driver.OracleDriver");
6.
7.             Connection con=DriverManager.getConnection(
8.                 "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
9.
10.            DatabaseMetaData dbmd=con.getMetaData();
11.            String table[]={ "TABLE" };
12.            ResultSet rs=dbmd.getTables(null,null,null,table);
13.
14.            while(rs.next()){
15.                System.out.println(rs.getString(3));
16.            }
17.
18.            con.close();
19.
20.        }catch(Exception e){ System.out.println(e); }
21.
22.    }
23. }
```

Example of DatabaseMetaData interface that prints total number of views :

```
1. import java.sql.*;
2. class Dbmd3{
3.     public static void main(String args[]){
4.         try{
5.             Class.forName("oracle.jdbc.driver.OracleDriver");
6.
7.             Connection con=DriverManager.getConnection(
8.                 "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
9.
10.            DatabaseMetaData dbmd=con.getMetaData();
11.            String table[]={ "VIEW" };
12.            ResultSet rs=dbmd.getTables(null,null,null,table);
13.
14.            while(rs.next()){
15.                System.out.println(rs.getString(3));
16.            }
17.
18.            con.close();
19.
20.        }catch(Exception e){ System.out.println(e); }
21.
22.    }
23. }
```

Core Java

Example to store image in Oracle database

You can store images in the database in java by the help of **PreparedStatement** interface.

The **setBinaryStream()** method of PreparedStatement is used to set Binary information into the parameterIndex.

Signature of setBinaryStream method

The syntax of setBinaryStream() method is given below:

1. 1) **public void** setBinaryStream(**int** paramIndex,InputStream stream)
2. **throws** SQLException
3. 2) **public void** setBinaryStream(**int** paramIndex,InputStream stream,**long** length)
4. **throws** SQLException

For storing image into the database, BLOB (Binary Large Object) datatype is used in the table. For example:

1. CREATE TABLE "IMGTABLE"
2. ("NAME" VARCHAR2(4000),
3. "PHOTO" BLOB
4.)
5. /

Let's write the jdbc code to store the image in the database. Here we are using d:\\d.jpg for the location of image. You can change it according to the image location.

1. **import** java.sql.*;
2. **import** java.io.*;
3. **public class** InsertImage {
4. **public static void** main(String[] args) {
5. **try**{
6. Class.forName("oracle.jdbc.driver.OracleDriver");
7. Connection con=DriverManager.getConnection(
8. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
- 9.
10. PreparedStatement ps=con.prepareStatement("insert into imgtable values(?,?)");
11. ps.setString(1,"sonoo");
- 12.
13. FileInputStream fin=**new** FileInputStream("d:\\g.jpg");
14. ps.setBinaryStream(2,fin,fin.available());
15. **int** i=ps.executeUpdate();
16. System.out.println(i+" records affected");
- 17.
18. con.close();
19. }**catch** (Exception e) {e.printStackTrace();}
20. }

Core Java

21. }

If you see the table, record is stored in the database but image will not be shown. To do so, you need to retrieve the image from the database which we are covering in the next page.

Example to retrieve image from Oracle database

By the help of **PreparedStatement** we can retrieve and store the image in the database.

The **getBlob()** method of PreparedStatement is used to get Binary information, it returns the instance of Blob. After calling the **getBytes()** method on the blob object, we can get the array of binary information that can be written into the image file.

Signature of getBlob() method of PreparedStatement

1. **public Blob getBlob()throws SQLException**

Signature of getBytes() method of Blob interface

1. **public byte[] getBytes(long pos, int length) throws SQLException**

We are assuming that image is stored in the imgtable.

```
1. CREATE TABLE "IMGTABLE"  
2. (  "NAME" VARCHAR2(4000),  
3.   "PHOTO" BLOB  
4. )  
5. /
```

Now let's write the code to retrieve the image from the database and write it into the directory so that it can be displayed.

In AWT, it can be displayed by the Toolkit class. In servlet, jsp, or html it can be displayed by the img tag.

```
1. import java.sql.*;  
2. import java.io.*;  
3. public class RetrieveImage {  
4. public static void main(String[] args) {  
5. try{  
6. Class.forName("oracle.jdbc.driver.OracleDriver");  
7. Connection con=DriverManager.getConnection(  
8. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");  
9.  
10. PreparedStatement ps=con.prepareStatement("select * from imgtable");  
11. ResultSet rs=ps.executeQuery();  
12. if(rs.next()){//now on 1st row  
13.  
14. Blob b=rs.getBlob(2);//2 means 2nd column data  
15. byte barr[]={};b.getBytes(1,(int)b.length());//1 means first image  
16.
```

Core Java

```
17. FileOutputStream fout=new FileOutputStream("d:\\sonoo.jpg");
18. fout.write(barr);
19.
20. fout.close();
21. }//end of if
22. System.out.println("ok");
23.
24. con.close();
25. }catch (Exception e) {e.printStackTrace(); }
26. }
27. }
```

Now if you see the d drive, sonoo.jpg image is created.

Example to store file in Oracle database:

The setCharacterStream() method of PreparedStatement is used to set character information into the parameterIndex.

Syntax:

- 1) public void setBinaryStream(int paramInt,InputStream stream)throws SQLException
- 2) public void setBinaryStream(int paramInt,InputStream stream,long length)throws SQLException

For storing file into the database, CLOB (Character Large Object) datatype is used in the table. For example:

```
1. CREATE TABLE "FILETABLE"
2. (
3.   "ID" NUMBER,
4.   "NAME" CLOB
5. )
6. /
7. import java.io.*;
8. import java.sql.*;
9.
10.
11. public class StoreFile {
12.   public static void main(String[] args) {
13.     try{
14.       Class.forName("oracle.jdbc.driver.OracleDriver");
15.       Connection con=DriverManager.getConnection(
16.         "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
17.
18.       PreparedStatement ps=con.prepareStatement(
19.         "insert into filetable values(?,?)");
20.
21.       File f=new File("d:\\myfile.txt");
22.
23.       FileInputStream fin=new FileInputStream(f);
24.       ps.setCharacterStream(2,fin);
25.       ps.executeUpdate();
26.     }
27.   }
28. }
```

Core Java

```
15. FileReader fr=new FileReader(f);
16.
17. ps.setInt(1,101);
18. ps.setCharacterStream(2,fr,(int)f.length());
19. int i=ps.executeUpdate();
20. System.out.println(i+" records affected");
21.
22. con.close();
23.
24. }catch (Exception e) {e.printStackTrace();}
25. }
26. }
```

Example to retrieve file from Oracle database:

The getClob() method of PreparedStatement is used to get file information from the database.

Syntax of getClob method

```
1. public Blob getClob(int columnIndex){}
```

Let's see the table structure of this example to retrieve the file.

```
1. CREATE TABLE "FILETABLE"
2. (
3.   "ID" NUMBER,
4.   "NAME" CLOB
5. )
```

The example to retrieve the file from the Oracle database is given below.

```
1. import java.io.*;
2. import java.sql.*;
3.
4. public class RetrieveFile {
5.   public static void main(String[] args) {
6.     try{
7.       Class.forName("oracle.jdbc.driver.OracleDriver");
8.       Connection con=DriverManager.getConnection(
9.         "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
10.
11.      PreparedStatement ps=con.prepareStatement("select * from filetable");
12.      ResultSet rs=ps.executeQuery();
13.      rs.next();//now on 1st row
14.
15.      Blob c=rs.getBlob(2);
```

Core Java

```
16. Reader r=c.getCharacterStream();
17.
18. FileWriter fw=new FileWriter("d:\\retrivefile.txt");
19.
20. int i;
21. while((i=r.read())!=-1)
22. fw.write((char)i);
23.
24. fw.close();
25. con.close();
26.
27. System.out.println("success");
28. }catch (Exception e) {e.printStackTrace(); }
29.
30. }
```

CallableStatement Interface

To call the **stored procedures and functions**, CallableStatement interface is used.

We can have business logic on the database by the use of stored procedures and functions that will make the performance better because these are precompiled.

Suppose you need the get the age of the employee based on the date of birth, you may create a function that receives date as the input and returns age of the employee as the output.

What is the difference between stored procedures and functions.

Stored Procedure	Function
is used to perform business logic.	is used to perform calculation.
must not have the return type.	must have the return type.
may return 0 or more values.	may return only one values.
We can call functions from the procedure.	Procedure cannot be called from function.
Procedure supports input and output parameters.	Function supports only input parameter.

Core Java

Exception handling using try/catch block can be used in stored procedures.

Exception handling using try/catch can't be used in user defined functions.

The differences between stored procedures and functions are given below:

How to get the instance of CallableStatement?

The prepareCall() method of Connection interface returns the instance of CallableStatement. Syntax is given below:

1. **public** CallableStatement prepareCall("{ call procedurename(?,?...?) }");

The example to get the instance of CallableStatement is given below:

1. CallableStatement stmt=con.prepareCall("{call myprocedure(?,?)}");

It calls the procedure myprocedure that receives 2 arguments.

Full example to call the stored procedure using JDBC

To call the stored procedure, you need to create it in the database. Here, we are assuming that stored procedure looks like this.

1. create or replace procedure "INSERTR"
2. (id IN NUMBER,
3. name IN VARCHAR2)
4. is
5. begin
6. insert into user420 values(id,name);
7. end;
8. /

The table structure is given below:

1. create table user420(id number(10), name varchar2(200));

In this example, we are going to call the stored procedure INSERTR that receives id and name as the parameter and inserts it into the table user420. Note that you need to create the user420 table as well to run this application.

1. **import** java.sql.*;
2. **public class** Proc {
3. **public static void** main(String[] args) **throws** Exception{
- 4.
5. Class.forName("oracle.jdbc.driver.OracleDriver");

Core Java

```
6. Connection con=DriverManager.getConnection(  
7. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");  
8.  
9. CallableStatement stmt=con.prepareCall("{call insertR(?,?)}");  
10. stmt.setInt(1,1011);  
11. stmt.setString(2,"Amit");  
12. stmt.execute();  
13.  
14. System.out.println("success");  
15. }  
16. }
```

Now check the table in the database, value is inserted in the user420 table.

Example to call the function using JDBC

In this example, we are calling the sum4 function that receives two input and returns the sum of the given number. Here, we have used the **registerOutParameter** method of CallableStatement interface, that registers the output parameter with its corresponding type. It provides information to the CallableStatement about the type of result being displayed.

The **Types** class defines many constants such as INTEGER, VARCHAR, FLOAT, DOUBLE, BLOB, CLOB etc.

Let's create the simple function in the database first.

```
1. create or replace function sum4  
2. (n1 in number,n2 in number)  
3. return number  
4. is  
5. temp number(8);  
6. begin  
7. temp :=n1+n2;  
8. return temp;  
9. end;  
10. /
```

Now, let's write the simple program to call the function.

```
1. import java.sql.*;  
2.  
3. public class FuncSum {  
4. public static void main(String[] args) throws Exception{  
5.  
6. Class.forName("oracle.jdbc.driver.OracleDriver");  
7. Connection con=DriverManager.getConnection(  
8. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
```

Core Java

```
9.  
10. CallableStatement stmt=con.prepareCall("{?= call sum4(?,?)}");  
11. stmt.setInt(2,10);  
12. stmt.setInt(3,43);  
13. stmt.registerOutParameter(1,Types.INTEGER);  
14. stmt.execute();  
15.  
16. System.out.println(stmt.getInt(1));  
17.  
18. }  
19. }  
1. Output: 53
```

Transaction Management in JDBC

Transaction represents **a single unit of work**.

The ACID properties describes the transaction management well. ACID stands for Atomicity, Consistency, isolation and durability.

Atomicity means either all successful or none.

Consistency ensures bringing the database from one consistent state to another consistent state.

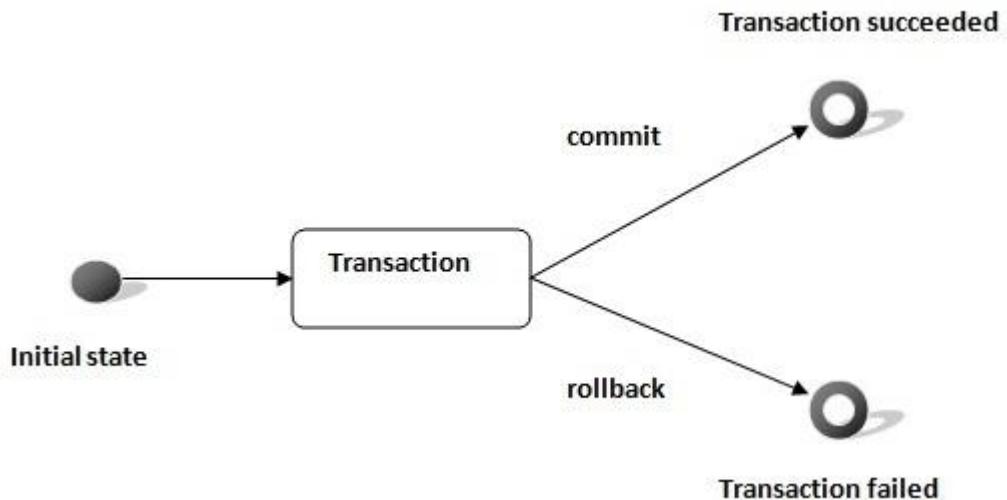
Isolation ensures that transaction is isolated from other transaction.

Durability means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.

Advantage of Transaction Management

fast performance It makes the performance fast because database is hit at the time of commit.

Core Java



In JDBC, **Connection interface** provides methods to manage transaction.

Method	Description
void setAutoCommit(boolean status)	It is true by default means each transaction is committed by default.
void commit()	commits the transaction.
void rollback()	cancels the transaction.

Simple example of transaction management in jdbc using Statement

Let's see the simple example of transaction management using Statement.

```
1. import java.sql.*;
2. class FetchRecords{
3. public static void main(String args[])throws Exception{
4. Class.forName("oracle.jdbc.driver.OracleDriver");
5. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
6. con.setAutoCommit(false);
7.
8. Statement stmt=con.createStatement();
9. stmt.executeUpdate("insert into user420 values(190,'abhi',40000)");
10. stmt.executeUpdate("insert into user420 values(191,'umesh',50000)");
11.
```

Core Java

```
12. con.commit();
13. con.close();
14. }}
```

If you see the table emp400, you will see that 2 records has been added.

Example of transaction management in jdbc using PreparedStatement

Let's see the simple example of transaction management using PreparedStatement.

```
1. import java.sql.*;
2. import java.io.*;
3. class TM{
4. public static void main(String args[]){
5. try{
6.
7. Class.forName("oracle.jdbc.driver.OracleDriver");
8. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
9. con.setAutoCommit(false);
10.
11. PreparedStatement ps=con.prepareStatement("insert into user420 values(?, ?, ?)");
12.
13. BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
14. while(true){
15.
16. System.out.println("enter id");
17. String s1=br.readLine();
18. int id=Integer.parseInt(s1);
19.
20. System.out.println("enter name");
21. String name=br.readLine();
22.
23. System.out.println("enter salary");
24. String s3=br.readLine();
25. int salary=Integer.parseInt(s3);
26.
27. ps.setInt(1,id);
28. ps.setString(2,name);
29. ps.setInt(3,salary);
30. ps.executeUpdate();
31.
32. System.out.println("commit/rollback");
33. String answer=br.readLine();
34. if(answer.equals("commit")){
35. con.commit();
36. }
37. if(answer.equals("rollback")){
38. con.rollback();
```

Core Java

```
39. }
40.
41.
42. System.out.println("Want to add more records y/n");
43. String ans=br.readLine();
44. if(ans.equals("n")){
45. break;
46. }
47.
48. }
49. con.commit();
50. System.out.println("record successfully saved");
51.
52. con.close();//before closing connection commit() is called
53. }catch(Exception e){System.out.println(e);}
54.
55. }}
```

It will ask to add more records until you press n. If you press n, transaction is committed.

Batch Processing in JDBC

Instead of executing a single query, we can execute a batch (group) of queries. It makes the performance fast.

The `java.sql.Statement` and `java.sql.PreparedStatement` interfaces provide methods for batch processing.

Advantage of Batch Processing

Fast Performance

Methods of Statement interface

The required methods for batch processing are given below:

Method	Description
<code>void addBatch(String query)</code>	It adds query into batch.
<code>int[] executeBatch()</code>	It executes the batch of queries.

Core Java

Example of batch processing in jdbc

Let's see the simple example of batch processing in jdbc. It follows following steps:

- Load the driver class
- Create Connection
- Create Statement
- Add query in the batch
- Execute Batch
- Close Connection

```
1. import java.sql.*;
2. class FetchRecords{
3. public static void main(String args[])throws Exception{
4. Class.forName("oracle.jdbc.driver.OracleDriver");
5. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
6. con.setAutoCommit(false);
7.
8. Statement stmt=con.createStatement();
9. stmt.addBatch("insert into user420 values(190,'abhi',40000)");
10. stmt.addBatch("insert into user420 values(191,'umesh',50000)");
11.
12. stmt.executeBatch();//executing the batch
13.
14. con.commit();
15. con.close();
16. }}
```

If you see the table user420, two records has been added.

Example of batch processing using PreparedStatement

```
1. import java.sql.*;
2. import java.io.*;
3. class BP{
4. public static void main(String args[]){
5. try{
6.
7. Class.forName("oracle.jdbc.driver.OracleDriver");
8. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
9.
10. PreparedStatement ps=con.prepareStatement("insert into user420 values(?, ?, ?)");
11.
12. BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
13. while(true){
14.
15. System.out.println("enter id");
```

Core Java

```
16. String s1=br.readLine();
17. int id=Integer.parseInt(s1);
18.
19. System.out.println("enter name");
20. String name=br.readLine();
21.
22. System.out.println("enter salary");
23. String s3=br.readLine();
24. int salary=Integer.parseInt(s3);
25.
26. ps.setInt(1,id);
27. ps.setString(2,name);
28. ps.setInt(3,salary);
29.
30. ps.addBatch();
31. System.out.println("Want to add more records y/n");
32. String ans=br.readLine();
33. if(ans.equals("n")){
34. break;
35. }
36.
37. }
38. ps.executeBatch();
39.
40. System.out.println("record successfully saved");
41.
42. con.close();
43. }catch(Exception e){System.out.println(e);}
44.
45. }
```

It will add the queries into the batch until user press n. Finally it executes the batch. Thus all the added queries will be fired.

JDBC RowSet

The instance of **RowSet** is the java bean component because it has properties and java bean notification mechanism. It is introduced since JDK 5.

It is the wrapper of ResultSet. It holds tabular data like ResultSet but it is easy and flexible to use.

The implementation classes of RowSet interface are as follows:

- JdbcRowSet
- CachedRowSet
- WebRowSet
- JoinRowSet

Core Java

- FilteredRowSet

Let's see how to create and execute RowSet.

```
1. JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();
2. rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
3. rowSet.setUsername("system");
4. rowSet.setPassword("oracle");
5.
6. rowSet.setCommand("select * from emp400");
7. rowSet.execute();
```

It is the new way to get the instance of JdbcRowSet since JDK 7.

Advantage of RowSet

The advantages of using RowSet are given below:

1. It is easy and flexible to use
2. It is Scrollable and Updatable by default

Simple example of JdbcRowSet

Let's see the simple example of JdbcRowSet without event handling code.

```
1. import java.sql.Connection;
2. import java.sql.DriverManager;
3. import java.sql.ResultSet;
4. import java.sql.Statement;
5. import javax.sql.RowSetEvent;
6. import javax.sql.RowSetListener;
7. import javax.sql.rowset.JdbcRowSet;
8. import javax.sql.rowset.RowSetProvider;
9.
10. public class RowSetExample {
11.     public static void main(String[] args) throws Exception {
12.         Class.forName("oracle.jdbc.driver.OracleDriver");
13.
14.         //Creating and Executing RowSet
15.         JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();
16.         rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
17.         rowSet.setUsername("system");
18.         rowSet.setPassword("oracle");
19.
20.         rowSet.setCommand("select * from emp400");
21.         rowSet.execute();
22.     }
}
```

Core Java

```
23. while (rowSet.next()) {  
24.         // Generating cursor Moved event  
25.         System.out.println("Id: " + rowSet.getString(1));  
26.         System.out.println("Name: " + rowSet.getString(2));  
27.         System.out.println("Salary: " + rowSet.getString(3));  
28.     }  
29.  
30. }  
31. }
```

The output is given below:

1. Id: **55**
2. Name: Om Bhim
3. Salary: **70000**
4. Id: **190**
5. Name: abhi
6. Salary: **40000**
7. Id: **191**
8. Name: umesh
9. Salary: **50000**

Full example of Jdbc RowSet with event handling

To perform event handling with JdbcRowSet, you need to add the instance of **RowSetListener** in the addRowSetListener method of JdbcRowSet.

The RowSetListener interface provides 3 method that must be implemented. They are as follows:

1. **1) public void** cursorMoved(RowSetEvent event);
2. **2) public void** rowChanged(RowSetEvent event);
3. **3) public void** rowSetChanged(RowSetEvent event);

Let's write the code to retrieve the data and perform some additional tasks while cursor is moved, cursor is changed or rowset is changed. The event handling operation can't be performed using ResultSet so it is preferred now.

```
1. import java.sql.Connection;  
2. import java.sql.DriverManager;  
3. import java.sql.ResultSet;  
4. import java.sql.Statement;  
5. import javax.sql.RowSetEvent;  
6. import javax.sql.RowSetListener;  
7. import javax.sql.rowset.JdbcRowSet;  
8. import javax.sql.rowset.RowSetProvider;  
9.  
10. public class RowSetExample {
```

Core Java

```
11. public static void main(String[] args) throws Exception {
12.     Class.forName("oracle.jdbc.driver.OracleDriver");
13.
14. //Creating and Executing RowSet
15. JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();
16. rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
17. rowSet.setUsername("system");
18. rowSet.setPassword("oracle");
19.
20. rowSet.setCommand("select * from emp400");
21. rowSet.execute();
22.
23. //Adding Listener and moving RowSet
24. rowSet.addRowSetListener(new MyListener());
25.
26.     while (rowSet.next()) {
27.         // Generating cursor Moved event
28.         System.out.println("Id: " + rowSet.getString(1));
29.         System.out.println("Name: " + rowSet.getString(2));
30.         System.out.println("Salary: " + rowSet.getString(3));
31.     }
32.
33. }
34. }
35.
36. class MyListener implements RowSetListener {
37.     public void cursorMoved(RowSetEvent event) {
38.         System.out.println("Cursor Moved...");
39.     }
40.     public void rowChanged(RowSetEvent event) {
41.         System.out.println("Cursor Changed...");
42.     }
43.     public void rowSetChanged(RowSetEvent event) {
44.         System.out.println("RowSet changed...");
45.     }
46. }
```

The output is as follows:

1. Cursor Moved...
2. Id: 55
3. Name: Om Bhim
4. Salary: 70000
5. Cursor Moved...
6. Id: 190
7. Name: abhi
8. Salary: 40000
9. Cursor Moved...
10. Id: 191
11. Name: umesh

12. Salary: **50000**
13. Cursor Moved...

Jdbc New Features

The latest version of JDBC is 4.0 currently. Java has updated jdbc api to ease and simplify the coding of database interactivity.

Here, we are going to see the features included in Jdbc 3.0 and Jdbc 4.0.

Jdbc 3.0 Features

The important features of JDBC API 3.0 are as follows:

- **Jdbc RowSet** We have done the great discussion on JdbcRowSet in the previous page.
- **Savepoint in transaction management** Now you are able to create, rollback and release the savepoint by Connection.setSavepoint(), Connection.rollback(Savepoint svpt) and Connection.releaseSavepoint(Savepoint svpt) methods.
- **Statement and ResultSet Caching for Connection Pooling** Now you are able to reuse the statement and result set because jdbc 3 provides you the facility of statement caching and result set caching.
- **Switching between Global and Local Transactions**
- **Retrieval of auto generated keys** Now you are able to get the auto generated keys by the method getGeneratedKeys().

Jdbc 4.0 Features

The important features of JDBC API 4.0 are given below:

- **Automatic Loading of Driver class** You don't need to write Class.forName() now because it is loaded by default since jdbc4.
- **Subclasses of SQLException** Jdbc 4 provides new subclasses of SQLException class for better readability and handling.
- **New methods** There are many new methods introduced in Connection, PreparedStatement, CallableStatement, ResultSet etc.
- **Improved DataSource** Now data source implementation is improved.
- **Event Handling support in Statement for Connection Pooling** Now Connection Pooling can listen statement error and statement closing events.

Core Java

Java Reflection API

Java Reflection API

Reflection is the process of examining or modifying the runtime behaviour of a class at runtime.

The `java.lang.Class` class provides many methods that can be used to get metadata, examine and change the runtime behaviour of a class.

Where is it used?

The Reflection API is mainly used in:

- IDE (Integrated Development Environment) e.g. Eclipse, MyEclipse, NetBeans etc.
- Debugger
- Test Tools etc.

Commonly used methods of `Class` class:

Method	Description
1) public String getName()	returns the class name
2) public static Class forName(String className)throws ClassNotFoundException	loads the class and returns the reference of Class class.
3) public Object newInstance()throws InstantiationException,IllegalAccessException	creates new instance.
4) public boolean isInterface()	checks if it is interface.
5) public boolean isArray()	checks if it is array.
6) public boolean isPrimitive()	checks if it is primitive.
7) public Class getSuperclass()	returns the superclass class reference.
8) public Field[] getDeclaredFields()throws SecurityException	returns the total number of fields of this class.
9) public Method[] getDeclaredMethods()throws SecurityException	returns the total number of

Core Java

	methods of this class.
10) public Constructor[] getDeclaredConstructors()throws SecurityException	returns the total number of constructors of this class.
11) public Method getDeclaredMethod(String name,Class[] parameterTypes)throws NoSuchMethodException,SecurityException	returns the method class instance.

Do You Know ?

- *How many ways we can get the instance of Class class ?*
- *How to create the javap tool ?*
- *How to create the appletviewer tool ?*
- *How to access the private method from outside the class ?*

java.lang.Class class

The java.lang.Class class performs mainly two tasks:

- provides methods to get the metadata of a class at runtime.
- provides methods to examine and change the runtime behaviour of a class.

How to get the object of Class class?

There are 3 ways to get the instance of Class class. They are as follows:

- forName() method of Class class
- getClass() method of Object class
- the .class syntax

1) forName() method of Class class

- is used to load the class dynamically.
- returns the instance of Class class.
- It should be used if you know the fully qualified name of class. This cannot be used for primitive types.

Core Java

Let's see the simple example of `forName()` method.

```
1. class Simple{}  
2.  
3. class Test{  
4.     public static void main(String args[]){  
5.         Class c=Class.forName("Simple");  
6.         System.out.println(c.getName());  
7.     }  
8. }
```

Output:Simple

2) `getClass()` method of Object class

It returns the instance of `Class` class. It should be used if you know the type. Moreover, it can be used with primitives.

```
1. class Simple{}  
2.  
3. class Test{  
4.     void printName(Object obj){  
5.         Class c=obj.getClass();  
6.         System.out.println(c.getName());  
7.     }  
8.     public static void main(String args[]){  
9.         Simple s=new Simple();  
10.  
11.        Test t=new Test();  
12.        t.printName(s);  
13.    }  
14. }  
15.
```

Output:Simple

3) The `.class` syntax

If a type is available but there is no instance then it is possible to obtain a `Class` by appending ".class" to the name of the type. It can be used for primitive data type also.

```
1. class Test{  
2.     public static void main(String args[]){  
3.         Class c = boolean.class;  
4.         System.out.println(c.getName());  
5.  
6.         Class c2 = Test.class;  
7.         System.out.println(c2.getName());  
8.     }  
9. }
```

Output:boolean

Test

Determining the class object

Following methods of Class class is used to determine the class object:

- 1) **public boolean isInterface()**: determines if the specified Class object represents an interface type.
- 2) **public boolean isArray()**: determines if this Class object represents an array class.
- 3) **public boolean isPrimitive()**: determines if the specified Class object represents a primitive type.

Let's see the simple example of reflection api to determine the object type.

```
1. class Simple{}  
2. interface My{}  
3.  
4. class Test{  
5.     public static void main(String args[]){  
6.         try{  
7.             Class c=Class.forName("Simple");  
8.             System.out.println(c.isInterface());  
9.  
10.            Class c2=Class.forName("My");  
11.            System.out.println(c2.isInterface());  
12.  
13.        }catch(Exception e){System.out.println(e);}  
14.  
15.    }  
16. }
```

Output: false

true

newInstance() method

The **newInstance()** method of **Class** class and **Constructor** class is used to create a new instance of the class.

The newInstance() method of Class class can invoke zero-argument constructor whereas newInstance() method of Constructor class can invoke any number of arguments. So Constructor class is preferred over Class class.

Core Java

Syntax of newInstance() method of Class class

public T newInstance()throws InstantiationException,IllegalAccessException

Here T is the generic version. You can think it like Object class. You will learn about generics later.

Example of newInstance() method

Let's see the simple example to use newInstance() method.

```
1. class Simple{  
2.     void message(){System.out.println("Hello Java");}  
3. }  
4.  
5. class Test{  
6.     public static void main(String args[]){  
7.         try{  
8.             Class c=Class.forName("Simple");  
9.             Simple s=(Simple)c.newInstance();  
10.            s.message();  
11.        }  
12.    }catch(Exception e){System.out.println(e);}  
13.    }  
14. }  
15. }
```

Output:Hello java

Creating a program that works as javap tool

Following methods of **java.lang.Class** class can be used to display the metadata of a class.

Method	Description
public Field[] getDeclaredFields()throws SecurityException	returns an array of Field objects reflecting all the fields declared by the class or interface represented by this Class object.
public Constructor[] getDeclaredConstructors()throws SecurityException	returns an array of Constructor objects reflecting all the constructors declared by the class represented by this Class object.
public Method[] getDeclaredMethods()throws	returns an array of Method objects reflecting all the

Core Java

SecurityException

methods declared by the class or interface represented by this Class object.

Example of creating javap tool

Let's create a program that works like javap tool.

```
1. import java.lang.reflect.*;
2.
3. public class MyJavap{
4.     public static void main(String[] args) throws Exception {
5.         Class c=Class.forName(args[0]);
6.
7.         System.out.println("Fields.....");
8.         Field f[]=c.getDeclaredFields();
9.         for(int i=0;i<f.length;i++)
10.             System.out.println(f[i]);
11.
12.         System.out.println("Constructors.....");
13.         Constructor con[]=c.getDeclaredConstructors();
14.         for(int i=0;i<con.length;i++)
15.             System.out.println(con[i]);
16.
17.         System.out.println("Methods.....");
18.         Method m[]=c.getDeclaredMethods();
19.         for(int i=0;i<m.length;i++)
20.             System.out.println(m[i]);
21.     }
22. }
```

At runtime, you can get the details of any class, it may be user-defined or pre-defined class.

Core Java

Output:

```
New Volume (E:)

File Edit View Favorites Tools Help
C:\WINDOWS\system32\cmd.exe

E:\>javac MyJavaP.java
E:\>java MyJavaP java.lang.Object
```

```
New Volume (E:)

File Edit View Favorites Tools Help
C:\WINDOWS\system32\cmd.exe

E:\>javac MyJavaP.java
E:\>java MyJavaP java.lang.Object
Fields.....
Constructors.....
Methods.....
protected void java.lang.Object.finalize() throws java.lang.Throwable
public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
public final void java.lang.Object.wait() throws java.lang.InterruptedException
public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
public boolean java.lang.Object.equals(java.lang.Object)
public java.lang.String java.lang.Object.toString()
public native int java.lang.Object.hashCode()
public final native java.lang.Class java.lang.Object.getClass()
protected native java.lang.Object java.lang.Object.clone() throws java.lang.CloneNotSupportedException
private static native void java.lang.Object.registerNatives()
public final native void java.lang.Object.notify()
public final native void java.lang.Object.notifyAll()

E:\>
```

Creating your own appletviewer

As you know well that appletviewer tool creates a frame and displays the output of applet in the frame. You can also create your frame and display the applet output.

Example that works like appletviewer tool

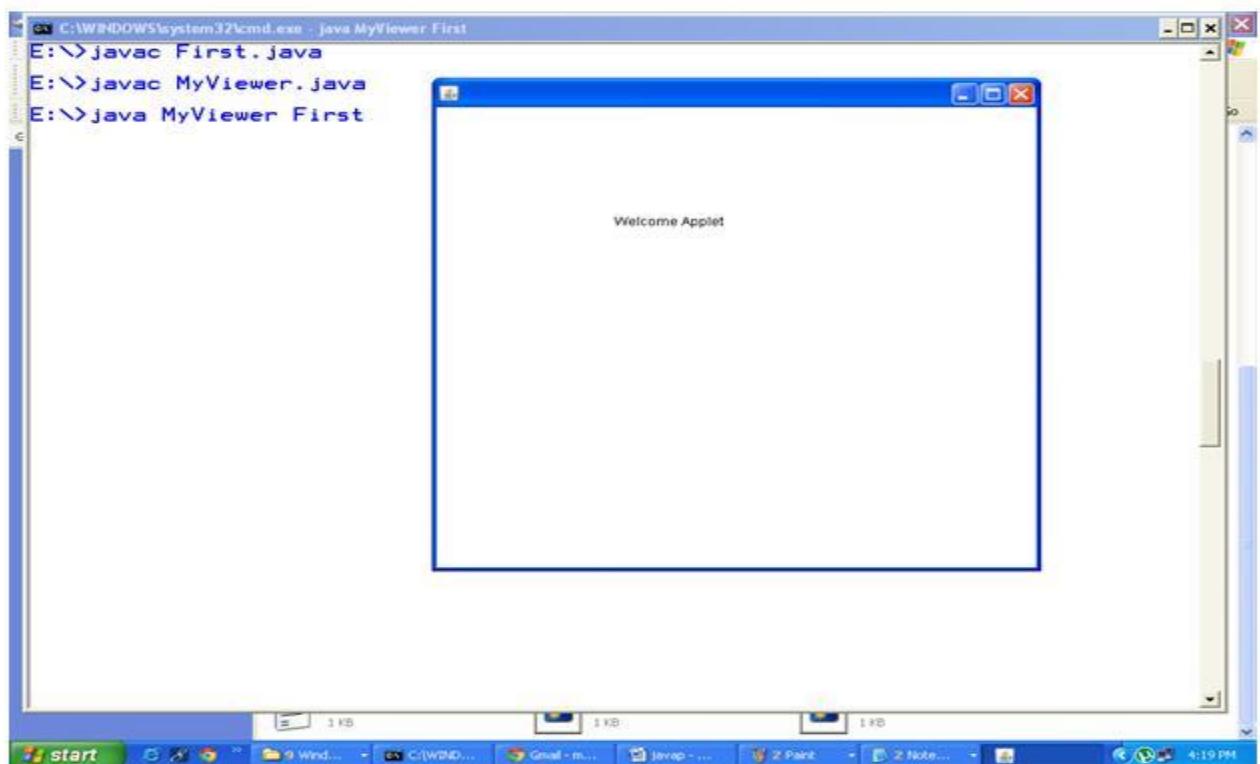
Let's see the simple example that works like appletviewer tool. This example displays applet on the frame.

```
1. import java.applet.Applet;
2. import java.awt.Frame;
3. import java.awt.Graphics;
4.
5. public class MyViewer extends Frame{
6.     public static void main(String[] args) throws Exception{
7.         Class c=Class.forName(args[0]);
8.
9.         MyViewer v=new MyViewer();
10.        v.setSize(400,400);
11.        v.setLayout(null);
12.        v.setVisible(true);
13.
14.        Applet a=(Applet)c.newInstance();
15.        a.start();
16.        Graphics g=v.getGraphics();
17.        a.paint(g);
18.        a.stop();
19.
20.    }
21.
22. }
1. //simple program of applet
2.
3. import java.applet.Applet;
4. import java.awt.Graphics;
5.
6. public class First extends Applet{
7.
8.     public void paint(Graphics g){g.drawString("Welcome",50, 50);}
9. }
```

Core Java

Output:

```
E:\>javac First.java  
E:\>javac MyViewer.java  
E:\>java MyViewer First
```



Core Java

How to call private method from another class in java

You can call the private method from outside the class by changing the runtime behaviour of the class.

By the help of **java.lang.Class** class and **java.lang.reflect.Method** class, we can call private method from any other class.

Required methods of Method class

1) **public void setAccessible(boolean status) throws SecurityException** sets the accessibility of the method.

2) **public Object invoke(Object method, Object... args) throws IllegalAccessException, IllegalArgumentException, InvocationTargetException** is used to invoke the method.

Required method of Class class

1) **public Method getDeclaredMethod(String name,Class[] parameterTypes) throws NoSuchMethodException,SecurityException:** returns a Method object that reflects the specified declared method of the class or interface represented by this Class object.

Example of calling private method from another class

Let's see the simple example to call private method from another class.

File: A.java

```
1. public class A {  
2.     private void message(){System.out.println("hello java");}  
3. }
```

File: MethodCall.java

```
1. import java.lang.reflect.Method;  
2. public class MethodCall{  
3.     public static void main(String[] args) throws Exception{  
4.         Class c = Class.forName("A");  
5.         Object o= c.newInstance();  
6.         Method m =c.getDeclaredMethod("message", null);  
7.         m.setAccessible(true);  
8.         m.invoke(o, null);  
9.     }  
10. }  
11. }
```

Output:hello java

Core Java

Another example to call parameterized private method from another class

Let's see the example to call parameterized private method from another class

File: A.java

```
1. class A{  
2.     private void cube(int n){System.out.println(n*n*n);}  
3. }
```

File: M.java

```
1. import java.lang.reflect.*;  
2. class M{  
3.     public static void main(String args[])throws Exception{  
4.         Class c=A.class;  
5.         Object obj=c.newInstance();  
6.         Method m=c.getDeclaredMethod("cube",new Class[]{int.class});  
7.         m.setAccessible(true);  
8.         m.invoke(obj,4);  
9.     }  
10. }
```

Output:64

Internationalization and Localization in Java

Internationalization is also abbreviated as I18N because there are total 18 characters between the first letter 'I' and the last letter 'N'.

Internationalization is a mechanism to create such an application that can be adapted to different languages and regions.

Internationalization is one of the powerful concept of java if you are developing an application and want to display messages, currencies, date, time etc. according to the specific region or language.

Localization is also abbreviated as I10N because there are total 10 characters between the first letter 'L' and last letter 'N'. Localization is the mechanism to create such an application that can be adapted to a specific language and region by adding locale-specific text and component.

Do You Know ?

- **What is the use of Locale class ?**
- **How can we globalize the messages (or) What is the use of ResourceBundle class?**
- **How can we internationalize the date, time, number, currency and measurements?**

Understanding the culturally dependent data before starting internationalization

Before starting the internationalization, Let's first understand what are the informations that differ from one region to another. There is the list of culturally dependent data:

- Messages
- Dates
- Times
- Numbers
- Currencies
- Measurements
- Phone Numbers
- Postal Addresses
- Labels on GUI components etc.

Importance of Locale class in Internationalization

An object of Locale class represents a geographical or cultural region. This object can be used to get the locale specific information such as country name, language, variant etc.

Fields of Locale class

There are fields of Locale class:

1. public static final Locale ENGLISH
2. public static final Locale FRENCH
3. public static final Locale GERMAN
4. public static final Locale ITALIAN
5. public static final Locale JAPANESE
6. public static final Locale KOREAN
7. public static final Locale CHINESE
8. public static final Locale SIMPLIFIED_CHINESE
9. public static final Locale TRADITIONAL_CHINESE
10. public static final Locale FRANCE
11. public static final Locale GERMANY
12. public static final Locale ITALY
13. public static final Locale JAPAN
14. public static final Locale KOREA
15. public static final Locale CHINA
16. public static final Locale PRC
17. public static final Locale TAIWAN
18. public static final Locale UK
19. public static final Locale US
20. public static final Locale CANADA
21. public static final Locale CANADA_FRENCH
22. public static final Locale ROOT

Constructors of Locale class

There are three constructors of Locale class. They are as follows:

1. Locale(String language)
2. Locale(String language, String country)
3. Locale(String language, String country, String variant)

Core Java

Commonly used methods of Locale class

There are given commonly used methods of Locale class.

1. **public static Locale getDefault()** it returns the instance of current Locale
2. **public static Locale[] getAvailableLocales()** it returns an array of available locales.
3. **public String getDisplayCountry()** it returns the country name of this locale object.
4. **public String getDisplayLanguage()** it returns the language name of this locale object.
5. **public String getDisplayVariant()** it returns the variant code for this locale object.
6. **public String getISO3Country()** it returns the three letter abbreviation for the current locale's country.
7. **public String getISO3Language()** it returns the three letter abbreviation for the current locale's language.

Example of Local class that prints the informations of the default locale

In this example, we are displaying the informations of the default locale. If you want to get the informations about any specific locale, comment the first line statement and uncomment the second line statement in the main method.

```
1. import java.util.*;  
2. public class LocaleExample {  
3.     public static void main(String[] args) {  
4.         Locale locale=Locale.getDefault();  
5. //Locale locale=new Locale("fr","fr");//for the specific locale  
6.  
7.         System.out.println(locale.getDisplayCountry());  
8.         System.out.println(locale.getDisplayLanguage());  
9.         System.out.println(locale.getDisplayName());  
10.        System.out.println(locale.getISO3Country());  
11.        System.out.println(locale.getISO3Language());  
12.        System.out.println(locale.getLanguage());  
13.        System.out.println(locale.getCountry());  
14.  
15.    }  
16. }
```

Output:United States

English
English (United States)
USA
eng

Core Java

en
US

Example of Local class that prints english in different languages

In this example, we are displaying english language in different language. Let's see how english is written in french and spanish languages.

```
1. import java.util.*;  
2. public class LocaleExample2 {  
3.     public static void main(String[] args) {  
4.         Locale enLocale = new Locale("en", "US");  
5.         Locale frLocale = new Locale("fr", "FR");  
6.         Locale esLocale = new Locale("es", "ES");  
7.         System.out.println("English language name (default): " +  
8.                             enLocale.getDisplayLanguage());  
9.  
10.        System.out.println("English language name in French: " +  
11.                            enLocale.getDisplayLanguage(frLocale));  
12.        System.out.println("English language name in spanish: " +  
13.                            enLocale.getDisplayLanguage(esLocale));  
14.    }  
15.  
16. }
```

Output:English language name (default): English
English language name in French: anglais
English language name in spanish: ingl???s

Example of Local class that print display language of many locales

In this example, we are displaying the display lanuage of many locales.

```
1. import java.util.*;  
2. public class LocaleEx {  
3.     public static void main(String[] args) {  
4.         Locale[] locales = { new Locale("en", "US"),  
5.             new Locale("es", "ES"), new Locale("it", "IT") };  
6.  
7.         for (int i=0; i< locales.length; i++) {  
8.             String displayLanguage = locales[i].getDisplayLanguage(locales[i]);  
9.             System.out.println(locales[i].toString() + ": " + displayLanguage);  
10.        }  
11.    }  
12.  
13. }
```

Output:en_US: English
es_ES: espa???ol
it_IT: italiano

Core Java

What we will learn in Internationalization Tutorial ?

- **ResourceBundle class**
- **I18N with Date**
- **I18N with Time**
- **I18N with Number**
- **I18N with Currency**
- **I18N with Measurements**

ResourceBundle class in Java

The **ResourceBundle class** is used to internationalize the messages. In other words, we can say that it provides a mechanism to globalize the messages.

The hardcoded message is not considered good in terms of programming, because it differs from one country to another. So we use the ResourceBundle class to globalize the messages. The ResourceBundle class loads these informations from the properties file that contains the messages.

Conventionally, the name of the properties file should be **filename_languagecode_country** code for example **MyMessage_en_US.properties**.

Commonly used methods of ResourceBundle class

There are many methods in the ResourceBundle class. Let's see the commonly used methods of the ResourceBundle class.

- **public static ResourceBundle getBundle(String basename)** returns the instance of the ResourceBundle class for the default locale.
- **public static ResourceBundle getBundle(String basename, Locale locale)** returns the instance of the ResourceBundle class for the specified locale.
- **public String getString(String key)** returns the value for the corresponding key from this resource bundle.

Example of ResourceBundle class

Let's see the simple example of ResourceBundle class. In this example, we are creating three files:

- **MessageBundle_en_US.properties** file contains the localize message for US country.
- **MessageBundle_in_ID.properties** file contains the localize message for Indonaisa country.

Core Java

- **InternationalizationDemo.java** file that loads these properties file in a bundle and prints the messages.

MessageBundle_en_US.properties

```
greeting=Hello, how are you?
```

MessageBundle_in_ID.properties

```
greeting=Halo, apa kabar?
```

InternationalizationDemo.java

```
1. import java.util.Locale;
2. import java.util.ResourceBundle;
3. public class InternationalizationDemo {
4.     public static void main(String[] args) {
5.
6.         ResourceBundle bundle = ResourceBundle.getBundle("MessageBundle", Locale.US);
7.         System.out.println("Message in "+Locale.US + ":" +bundle.getString("greeting"));
8.
9.         //changing the default locale to indonasan
10.        Locale.setDefault(new Locale("in", "ID"));
11.        bundle = ResourceBundle.getBundle("MessageBundle");
12.        System.out.println("Message in "+Locale.getDefault()+ ":" +bundle.getString("greeting"));
13.
14.    }
15. }
```

Output:Message in en_US : Hello, how r u?

Message in in_ID : halo, apa kabar?

Internationalizing Date (I18N with Date)

The format of the dates differ from one region to another that is why we internationalize the dates.

We can internationalize the date by using the **getDateFormat()** method of the **DateFormat** class. It receives the locale object as a parameter and returns the instance of the DateFormat class.

Core Java

Commonly used methods of DateFormat class for internationalizing date

There are many methods of the DateFormat class. Let's see the two methods of the DateFormat class for internationalizing the dates.

- **public static DateFormat getDateInstance(int style, Locale locale)** returns the instance of the DateFormat class for the specified style and locale. The style can be DEFAULT, SHORT, LONG etc.
- **public String format(Date date)** returns the formatted and localized date as a string.

Example of Internationalizing Date

In this example, we are displaying the date according to the different locale such as UK, US, FRANCE etc. For this purpose we have created the printDate() method that receives Locale object as an instance. The format() method of the DateFormat class receives the Date object and returns the formatted and localized date as a string.

```
1. import java.text.DateFormat;
2. import java.util.*;
3. public class InternationalizationDate {
4.
5.     static void printDate(Locale locale){
6.         DateFormat formatter=DateFormat.getDateInstance(DateFormat.DEFAULT,locale);
7.         Date currentDate=new Date();
8.         String date=formatter.format(currentDate);
9.         System.out.println(date+" "+locale);
10.    }
11.
12.    public static void main(String[] args) {
13.        printDate(Locale.UK);
14.        printDate(Locale.US);
15.        printDate(Locale.FRANCE);
16.    }
17. }
```

Output:01-Mar-2012 en_GB

Mar 1, 2012 en_US

1 mars 2012 fr_FR

Internationalizing Time (I18N with Time)

The display format of the time differs from one region to another, so we need to internationalize the time.

For internationalizing the time, the **DateFormat** class provides some useful methods.

The **getTimeInstance()** method of the DateFormat class returns the instance of the DateFormat class for the specified style and locale.

Syntax of the getTimeInstance() method is given below:

```
public static DateFormat getTimeInstance(int style, Locale locale)
```

Example of Internationalizing Time

In this example, we are displaying the current time for the specified locale. The **format()** method of the DateFormat class receives date object and returns the formatted and localized time as a string. Notice that the object of Date class prints date and time both.

```
1. import java.text.DateFormat;
2. import java.util.*;
3.
4. public class InternationalizingTime {
5.
6.     static void printTime(Locale locale){
7.         DateFormat formatter=DateFormat.getTimeInstance(DateFormat.DEFAULT,locale);
8.         Date currentDate=new Date();
9.         String time=formatter.format(currentDate);
10.        System.out.println(time+" in locale "+locale);
11.    }
12.
13.    public static void main(String[] args) {
14.        printTime(Locale.UK);
15.        printTime(Locale.US);
16.        printTime(Locale.FRANCE);
17.    }
18. }
```

Output:
16:22:49 in locale en_GB
4:22:49 PM in locale en_US
16:22:49 in locale fr_FR

Internationalizing Number (I18N with Number)

The representation of the numbers differ from one locale to another. Internationalizing the numbers is good approach for the application that displays the informations according to the locales.

The **NumberFormat** class is used to format the number according to the specific locale. To get the instance of the NumberFormat class, we need to call either **getInstance()** or **getNumberInstance()** methods.

Syntax of these methods is given below:

1. **public static** NumberFormat getNumberInstance(Locale locale)
2. **public static** NumberFormat getInstance(Locale locale) //same as above

Example of Internationalizing Number

In this example, we are internationalizing the number. The format method of the NumberFormat class formats the double value into the locale specific number.

```
1. import java.text.NumberFormat;
2. import java.util.*;
3.
4. public class InternalizationNumber {
5.
6.     static void printNumber(Locale locale){
7.         double dbl=105000.3245;
8.         NumberFormat formatter=NumberFormat.getNumberInstance(locale);
9.         String number=formatter.format(dbl);
10.        System.out.println(number+" for the locale "+locale);
11.    }
12.
13.    public static void main(String[] args) {
14.        printNumber(Locale.UK);
15.        printNumber(Locale.US);
16.        printNumber(Locale.FRANCE);
17.        printNumber(Locale.JAPAN);
18.
19.    }
20. }
```

Output:105,500.324 for the locale en_GB

105,000.324 for the locale en_US

105,a000,324 for the locale fr_FR

105,000.324 for the locale ja_JP

Internationalizing Currency (I18N with Currency)

As we have internationalized the date, time and numbers, we can internationalize the currency also. The currency differs from one country to another so we need to internationalize the currency.

The **NumberFormat** class provides methods to format the currency according to the locale. The **getCurrencyInstance()** method of the NumberFormat class returns the instance of the NumberFormat class.

The syntax of the **getCurrencyInstance()** method is given below:

1. **public static** NumberFormat getCurrencyInstance(Locale locale)

Example of Internationalizing Currency

In this example, we are internationalizing the currency. The **format** method of the NumberFormat class formats the double value into the locale specific currency.

```
1. import java.text.NumberFormat;
2. import java.util.*;
3. public class InternalizationCurrency {
4.
5.     static void printCurrency(Locale locale){
6.         double dbl=10500.3245;
7.         NumberFormat formatter=NumberFormat.getCurrencyInstance(locale);
8.         String currency=formatter.format(dbl);
9.         System.out.println(currency+" for the locale "+locale);
10.    }
11.
12.    public static void main(String[] args) {
13.        printCurrency(Locale.UK);
14.        printCurrency(Locale.US);
15.        printCurrency(Locale.FRANCE);
16.    }
17. }
```

Output: ?10,500.32 for the locale en_GB

\$10,500.32 for the locale en_US

10 500,32 ? for the locale fr_FR

New Features in Java

There are many new features that have been added in java. There are major enhancement made in Java5, Java6 and Java7 like **auto-boxing, generics, var-args, java annotations, enum, premain method** etc.

Most of the interviewers ask questions from this chapter.

Do You Know ?

1. **How to create generic class and generic method in java ?**
2. **What is annotation and how to create custom annotation ?**
3. **What is the advantage of assertion and where we should not use it ?**
4. **What is variable argument and what rules are defined for variable argument ?**
5. **What is the difference between import and static import ?**
6. **How autoboxing is applied in method overloading. Which concept beats autoboxing ?**
7. **What is enum type and how to specify specific value to the enum constants ?**

J2SE 4 Features

The important feature of J2SE 4 is assertions. It is used for testing.

- **Assertion (Java 4)**

J2SE 5 Features

The important features of J2SE 5 are generics and assertions. Others are auto-boxing, enum, var-args, static import, for-each loop (enhanced for loop etc).

- **For-each loop (Java 5)**
- **Varargs (Java 5)**
- **Static Import (Java 5)**
- **Autoboxing and Unboxing (Java 5)**
- **Enum (Java 5)**
- **Covariant Return Type (Java 5)**
- **Annotation (Java 5)**
- **Generics (Java 5)**

Core Java

JavaSE 6 Features

The important feature of JavaSE 6 is premain method (also known as instrumentation).

- Instrumentation (premain method) (Java 6)

JavaSE 7 Features

The important features of JavaSE 7 are try with resource, catching multiple exceptions etc.

- String in switch statement (Java 7)
- Binary Literals (Java 7)
- The try-with-resources (Java 7)
- Caching Multiple Exceptions by single catch (Java 7)
- Underscores in Numeric Literals (Java 7)

Assertion:

Assertion is a statement in java. It can be used to test your assumptions about the program.

While executing assertion, it is believed to be true. If it fails, JVM will throw an error named `AssertionError`. It is mainly used for testing purpose.

Advantage of Assertion:

It provides an effective way to detect and correct programming errors.

Syntax of using Assertion:

There are two ways to use assertion. First way is:

1. `assert` expression;
and second way is:
1. `assert` expression1 : expression2;

Simple Example of Assertion in java:

1. `import` java.util.Scanner;
- 2.
3. `class` AssertionExample{
4. `public static void` main(String args[]){
- 5.

Core Java

```
6. Scanner scanner = new Scanner( System.in );
7. System.out.print("Enter ur age ");
8.
9. int value = scanner.nextInt();
10. assert value>=18:" Not valid";
11.
12. System.out.println("value is "+value);
13. }
14. }
15.
16.
```

If you use assertion, It will not run simply because assertion is disabled by default. To enable the assertion, **-ea** or **-enableassertions** switch of java must be used.

Compile it by: **javac AssertionExample.java**

Run it by: **java -ea AssertionExample**

Output: Enter ur age 11

Exception in thread "main" java.lang.AssertionError: Not valid

Where not to use Assertion:

There are some situations where assertion should be avoid to use. They are:

1. According to Sun Specification, assertion should not be used to check arguments in the public methods because it should result in appropriate runtime exception e.g. `IllegalArgumentException`, `NullPointerException` etc.
2. Do not use assertion, if you don't want any error in any situation.

For-each loop (Advanced or Enhanced For loop):

The for-each loop introduced in Java5. It is mainly used to traverse array or collection elements. The advantage of for-each loop is that it eliminates the possibility of bugs and makes the code more readable.

Advantage of for-each loop:

- It makes the code more readable.
- It eliminates the possibility of programming errors.

Syntax of for-each loop:

```
1. for(data_type variable : array | collection){ }
```

Simple Example of for-each loop for traversing the array elements:

```
1.  
2. class ForEachExample1{  
3.   public static void main(String args[]){  
4.     int arr[]={12,13,14,44};  
5.  
6.     for(int i:arr){  
7.       System.out.println(i);  
8.     }  
9.  
10.  }  
11. }  
12.
```

Output:12

```
 13  
 14  
 44
```

Simple Example of for-each loop for traversing the collection elements:

```
1. import java.util.*;  
2. class ForEachExample2{  
3.   public static void main(String args[]){  
4.     ArrayList<string> list=new ArrayList<string>();  
5.     list.add("vimal");  
6.     list.add("sonoo");  
7.     list.add("ratan");  
8.  
9.     for(String s:list){  
10.       System.out.println(s);  
11.     }  
12.  
13.   }  
14. }  
15.  
16. </string></string>
```

Output:vimal

```
  sonoo  
  ratan
```

Core Java

Variable Argument (Varargs):

The varargs allows the method to accept zero or multiple arguments. Before varargs either we use overloaded method or take an array as the method parameter but it was not considered good because it leads to the maintenance problem. If we don't know how many argument we will have to pass in the method, varargs is the better approach.

Advantage of Varargs:

We don't have to provide overloaded methods so less code.

Syntax of varargs:

The varargs uses ellipsis i.e. three dots after the data type. Syntax is as follows:

1. return_type method_name(data_type... variableName){ }

Simple Example of Varargs in java:

```
1.
2. class VarargsExample1{
3.
4. static void display(String... values){
5. System.out.println("display method invoked ");
6. }
7.
8. public static void main(String args[]){
9.
10. display();//zero argument
11. display("my","name","is","varargs");//four arguments
12. }
13. }
14.
```

Output:display method invoked
display method invoked

Another Program of Varargs in java:

```
1.
2. class VarargsExample{
3.
4. static void display(String... values){
5. System.out.println("display method invoked ");
6. for(String s:values){
7. System.out.println(s);
8. }
```

Core Java

```
9. }
10.
11. public static void main(String args[]){
12.
13. display();//zero argument
14. display("hello");//one argument
15. display("my", "name", "is", "varargs");//four arguments
16. }
17. }
18.
```

Output:display method invoked

 display method invoked

 hello

 display method invoked

 my

 name

 is

 varargs

Rules for varargs:

While using the varargs, you must follow some rules otherwise program code won't compile. The rules are as follows:

- There can be only one variable argument in the method.
- Variable argument (varargs) must be the last argument.

Examples of varargs that fails to compile:

```
1.
2. void method(String... a, int... b){ }//Compile time error
3.
4. void method(int... a, String b){ }//Compile time error
5.
```

Example of Varargs that is the last argument in the method:

```
1.
2. class VarargsExample3{
3.
4.     static void display(int num, String... values){
5.         System.out.println("number is "+num);
6.         for(String s:values){
7.             System.out.println(s);
8.         }
9.     }
10.
11.    public static void main(String args[]){
12. }
```

Core Java

```
13. display(500,"hello");//one argument  
14. display(1000,"my","name","is","varargs");//four arguments  
15. }  
16. }  
17.
```

Output:number is 500

```
    hello  
    number is 1000  
    my  
    name  
    is  
    varargs
```

Static Import:

The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.

Advantage of static import:

- Less coding is required if you have access any static member of a class oftenly.

Disadvantage of static import:

- If you overuse the static import feature, it makes the program unreadable and unmaintainable.

Simple Example of static import

```
1. import static java.lang.System.*;  
2. class StaticImportExample{  
3.     public static void main(String args[]){  
4.  
5.         out.println("Hello");//Now no need of System.out  
6.         out.println("Java");  
7.  
8.     }  
9. }
```

Output:Hello

```
    Java
```

Core Java

What is the difference between import and static import?

The import allows the java programmer to access classes of a package without package qualification whereas the static import feature allows to access the static members of a class without the class qualification. The import provides accessibility to classes and interface whereas static import provides accessibility to static members of the class.

Autoboxing and Unboxing:

The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing. This is the new feature of Java5. So java programmer doesn't need to write the conversion code.

Advantage of Autoboxing and Unboxing:

No need of conversion between primitives and Wrappers manually so less coding is required.

Simple Example of Autoboxing in java:

```
1.
2. class BoxingExample1{
3.     public static void main(String args[]){
4.         int a=50;
5.         Integer a2=new Integer(a);//Boxing
6.
7.         Integer a3=5;//Boxing
8.
9.         System.out.println(a2+ " "+a3);
10.    }
11. }
12.
```

Output:50 5

Simple Example of Unboxing in java:

The automatic conversion of wrapper class type into corresponding primitive type, is known as Unboxing. Let's see the example of unboxing:

```
1.
2. class UnboxingExample1{
3.     public static void main(String args[]){
4.         Integer i=new Integer(50);
5.         int a=i;
6.
7.         System.out.println(a);
```

Core Java

```
8. }
9. }
10.
```

Output:50

Autoboxing and Unboxing with comparison operators

Autoboxing can be performed with comparison operators. Let's see the example of boxing with comparison operator:

```
1.
2. class UnboxingExample1{
3.     public static void main(String args[]){
4.         Integer i=new Integer(50);
5.
6.         if(i<100){          //unboxing internally
7.             System.out.println(i);
8.         }
9.     }
10. }
11.
```

Output:50

Autoboxing and Unboxing with method overloading

In method overloading, boxing and unboxing can be performed. There are some rules for method overloading with boxing:

- Widening beats boxing
- Widening beats varargs
- Boxing beats varargs

1) Example of Autoboxing where widening beats boxing

If there is possibility of widening and boxing, widening beats boxing.

```
1.
2. class Boxing1{
3.     static void m(int i){System.out.println("int");}
4.     static void m(Integer i){System.out.println("Integer");}
5.
6.     public static void main(String args[]){
7.         short s=30;
8.         m(s);
9.     }
10. }
11.
```

Output:int

Core Java

2) Example of Autoboxing where widening beats varargs

If there is possibility of widening and varargs, widening beats var-args.

```
1.
2. class Boxing2{
3.     static void m(int i, int i2){System.out.println("int int");}
4.     static void m(Integer... i){System.out.println("Integer...");}
5.
6.     public static void main(String args[]){
7.         short s1=30,s2=40;
8.         m(s1,s2);
9.     }
10. }
```

Output:int int

3) Example of Autoboxing where boxing beats varargs

Let's see the program where boxing beats variable argument:

```
1.
2. class Boxing3{
3.     static void m(Integer i){System.out.println("Integer");}
4.     static void m(Integer... i){System.out.println("Integer...");}
5.
6.     public static void main(String args[]){
7.         int a=30;
8.         m(a);
9.     }
10. }
```

Output:Integer

Method overloading with Widening and Boxing

Widening and Boxing can't be performed as given below:

```
1.
2. class Boxing3{
3.     static void m(Long l){System.out.println("Long");}
4.
5.     public static void main(String args[]){
6.         int a=30;
7.         m(a);
8.     }
9. }
```

Output:Compile Time Error

Core Java

Enum

An enum is a data type which contains fixed set of constants. It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY) , directions (NORTH, SOUTH, EAST and WEST) etc. The enum constants are static and final implicitly. It is available from Java 5. Enums can be thought of as classes that have fixed set of constants.

Points to remember for Enum:

1. enum improves type safety
2. enum can be easily used in switch
3. enum can be traversed
4. enum can have fields, constructors and methods
5. enum may implement many interfaces but cannot extend any class because it internally extends Enum class

Simple Example of enum in java:

```
1.  
2. class EnumExample1{  
3.  
4. public enum Season { WINTER, SPRING, SUMMER, FALL }  
5.  
6. public static void main(String[] args) {  
7. for (Season s : Season.values())  
8. System.out.println(s);  
9.  
10. }  
11.
```

Output:
WINTER
SPRING
SUMMER
FALL

What is the purpose of values() method in enum?

The java compiler internally adds the values() method when it creates an enum. The values() method returns an array containing all the values of the enum.

Core Java

Internal code generated by the compiler for the above example of enum type

The java compiler internally creates a static and final class that extends the Enum class as shown in the below example:

```
1. public static final class EnumExample1$Season extends Enum
2. {
3.     private EnumExample1$Season(String s, int i)
4.     {
5.         super(s, i);
6.     }
7.
8.     public static EnumExample1$Season[] values()
9.     {
10.        return (EnumExample1$Season[])$VALUES.clone();
11.    }
12.
13.    public static EnumExample1$Season valueOf(String s)
14.    {
15.        return (EnumExample1$Season)Enum.valueOf(EnumExample1$Season, s);
16.    }
17.
18.    public static final EnumExample1$Season WINTER;
19.    public static final EnumExample1$Season SPRING;
20.    public static final EnumExample1$Season SUMMER;
21.    public static final EnumExample1$Season FALL;
22.    private static final EnumExample1$Season $VALUES[];
23.
24.    static
25.    {
26.        WINTER = new EnumExample1$Season("WINTER", 0);
27.        SPRING = new EnumExample1$Season("SPRING", 1);
28.        SUMMER = new EnumExample1$Season("SUMMER", 2);
29.        FALL = new EnumExample1$Season("FALL", 3);
30.        $VALUES = (new EnumExample1$Season[] {
31.            WINTER, SPRING, SUMMER, FALL
32.        });
33.    }
34.
35. }
```

Defining enum:

The enum can be defined within or outside the class because it is similar to a class.

Core Java

Example of enum that is defined outside the class:

```
1. 
2. enum Season { WINTER, SPRING, SUMMER, FALL }
3. 
4. class EnumExample2{
5.   public static void main(String[] args) {
6. 
7.     Season s=Season.WINTER;
8.     System.out.println(s);
9. 
10.  }
11.
```

Output:WINTER

Example of enum that is defined within the class:

```
1. class EnumExample2{
2.   enum Season { WINTER, SPRING, SUMMER, FALL; }//semicolon(;) is optional here
3. 
4.   public static void main(String[] args) {
5.     Season s=Season.WINTER;//enum type is required to access WINTER
6.     System.out.println(s);
7. 
8.   }
9.
```

Output:WINTER

Initializing specific value to the enum constants:

The enum constants have initial value that starts from 0, 1, 2, 3 and so on. But we can initialize the specific value to the enum constants by defining fields and constructors. As specified earlier, Enum can have fields, constructors and methods.

Example of specifying initial value to the enum constants

```
1. class EnumExample4{
2.   enum Season{
3.     WINTER(5), SPRING(10), SUMMER(15), FALL(20);
4. 
5.   private int value;
6.   private Season(int value){
7.     this.value=value;
8.   }
9. }
10. public static void main(String args[]){
11.   for (Season s : Season.values())
12.     System.out.println(s+" "+s.value);
```

Core Java

```
13.  
14. }  
15. }
```

Output:WINTER 5

SPRING 10
SUMMER 15
FALL 20

Constructor of enum type is private if you don't declare private compiler internally have private constructor

```
1. enum Season{  
2. WINTER(10),SUMMER(20);  
3. private int value;  
4. Season(int value){  
5.     this.value=value;  
6. }  
7. }  
8.
```

Internal code generated by the compiler for the above example of enum type

```
1. final class Season extends Enum  
2. {  
3.  
4.     public static Season[] values()  
5.     {  
6.         return (Season[])$VALUES.clone();  
7.     }  
8.  
9.     public static Season valueOf(String s)  
10.    {  
11.        return (Season)Enum.valueOf(Season, s);  
12.    }  
13.  
14.     private Season(String s, int i, int j)  
15.    {  
16.        super(s, i);  
17.        value = j;  
18.    }  
19.  
20.     public static final Season WINTER;  
21.     public static final Season SUMMER;  
22.     private int value;  
23.     private static final Season $VALUES[];  
24.  
25.     static
```

Core Java

```
26.  {
27.      WINTER = new Season("WINTER", 0, 10);
28.      SUMMER = new Season("SUMMER", 1, 20);
29.      $VALUES = (new Season[] {
30.          WINTER, SUMMER
31.      });
32.  }
33. }
```

Can we create the instance of enum by new keyword?

No, because it contains private constructors only.

Can we have abstract method in enum?

Yes, ofcourse! we can have abstract methods and can provide the implementation of these methods.

Applying enum on switch statement

We can apply enum on switch statement as in the given example:

Example of applying enum on switch statement

```
1. class EnumExample5{
2.     enum Day{ SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, S
ATURDAY}
3.
4.     public static void main(String args[]){
5.
6.         Day day=Day.MONDAY;
7.
8.         switch(day){
9.             case SUNDAY:
10.                 System.out.println("sunday");
11.                 break;
12.             case MONDAY:
13.                 System.out.println("monday");
14.                 break;
15.             default:
16.                 System.out.println("other day");
17.         }
18.
19.     }
20. }
```

Output:monday

Java Annotation

Annotation is a tag that represents the metadata. It is attached with class, interface, methods or fields to indicate some additional information that can be used by java compiler and JVM.

Built-In Annotations

There are several built-in annotations. Some annotations are applied to java code and some to other annotations.

Built-In Annotations that are applied to java code

- @Override
- @SuppressWarnings
- @Deprecated

Built-In Annotations that are applied to other annotations

- @Target
- @Retention
- @Inherited
- @Documented

Understanding Built-In Annotations that are applied to java code

Let's understand the built-in annotations first.

@Override

@Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.

Sometimes, we do the silly mistake such as spelling mistakes etc. So, it is better to mark @Override annotation that provides assurance that method is overridden.

```
1. class Animal{  
2.     void eatSomething(){System.out.println("eating something");}  
3. }  
4.  
5. class Dog extends Animal{  
6.     @Override
```

Core Java

```
7. void eatsomething(){System.out.println("eating foods");}//should be eatSomething  
8. }  
9.  
10. class Test{  
11. public static void main(String args[]){  
12. Animal a=new Dog();  
13. a.eatSomething();  
14. }}  
Output: Comple Time Error
```

@SuppressWarnings

@SuppressWarnings annotation: is used to suppress warnings issued by the compiler.

```
1. import java.util.*;  
2. class Test{  
3. @SuppressWarnings("unchecked")  
4. public static void main(String args[]){  
5.  
6. ArrayList list=new ArrayList();  
7. list.add("sonoo");  
8. list.add("vimal");  
9. list.add("ratan");  
10.  
11. for(Object obj:list)  
12. System.out.println(obj);  
13.  
14. }}
```

Now no warning at compile time.

If you remove the @SuppressWarnings("unchecked") annotation, it will show warning at compile time because we are using non-generic collection.

@Deprecated

@Deprecated annoation marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions. So, it is better not to use such methods.

```
1. class A{  
2. void m(){System.out.println("hello m");}  
3.  
4. @Deprecated  
5. void n(){System.out.println("hello n");}  
6. }
```

Core Java

```
7.  
8. class Test{  
9. public static void main(String args[]){  
10.  
11. A a=new A();  
12. a.n();  
13. }}
```

At Compile Time:

Note: Test.java uses or overrides a deprecated API.

Note: Recompile with -Xlint:deprecation for details.

At Runtime:

Hello n

Custom Annotation

Creating, Applying and Accessing Custom Annotation

Custom Annotation

We can create the user-defined annotations also. The @interface element is used to declare an annotation. For example:

```
1. @interface MyAnnotation{ }
```

Points to remember for annotation signature

There are few points that should be remembered by the programmer.

1. Method should not have any throws clauses
2. Method should return one of the following: primitive data types, String, Class, enum or array of these data types.
3. Method should not have any parameter.
4. We should attach @ just before interface keyword to define annotation.
5. It may assign a default value to the method.

Types of Annotation

There are three types of annotations.

1. Marker Annotation
2. Single-Value Annotation
3. Multi-Value Annotation

1) Marker Annotation

An annotation that has no method, is called marker annotation. For example:

```
1. @interface MyAnnotation{ }
```

The @Override and @Deprecated are marker annotations.

2) Single-Value Annotation

An annotation that has one method, is called Single-Value annotation. For example:

```
1. @interface MyAnnotation{  
2. int value();  
3. }
```

We can provide the default value also. For example:

```
1. @interface MyAnnotation{  
2. int value() default 0;  
3. }
```

How to apply Single-Value Annotation

Let's see the code to apply the single value annotation.

```
1. @MyAnnotation(value=10)
```

The value can be anything.

3) Multi-Value Annotation

An annotation that has more than one method, is called Multi-Value annotation. For example:

```
1. @interface MyAnnotation{  
2. int value1();  
3. String value2();  
4. String value3();  
5. }  
6. }
```

We can provide the default value also. For example:

```
1. @interface MyAnnotation{  
2. int value1() default 1;  
3. String value2() default "";  
4. String value3() default "xyz";
```

5. }

How to apply Multi-Value Annotation

Let's see the code to apply the multi-value annotation.

1. `@MyAnnotation(value1=10,value2="Arun Kumar",value3="Ghaziabad")`

Built-in Annotations applied to other annotations (custom annotations)

- `@Target`
- `@Retention`
- `@Inherited`
- `@Documented`

`@Target`

`@Target` tag is used to specify at which type, the annotation is used.

The `java.lang.annotation.ElementType` enum declares many constants to specify the type of element where annotation is to be applied such as TYPE, METHOD, FIELD etc. Let's see the constants of ElementType enum:

Element Types	Where the annotation can be applied
TYPE	class, interface or enumeration
FIELD	fields
METHOD	methods
CONSTRUCTOR	constructors
LOCAL_VARIABLE	local variables
ANNOTATION_TYPE	annotation type
PARAMETER	parameter

Core Java

Example to specify annotation for a class

1. `@Target(ElementType.TYPE)`
2. `@interface MyAnnotation{`
3. `int value1();`
4. `String value2();`
5. `}`

Example to specify annotation for a class, methods or fields

1. `@Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD})`
2. `@interface MyAnnotation{`
3. `int value1();`
4. `String value2();`
5. `}`

@Retention

@Retention annotation is used to specify to what level annotation will be available.

RetentionPolicy	Availability
<code>RetentionPolicy.SOURCE</code>	refers to the source code, discarded during compilation. It will not be available in the compiled class.
<code>RetentionPolicy.CLASS</code>	refers to the .class file, available to java compiler but not to JVM . It is included in the class file.
<code>RetentionPolicy.RUNTIME</code>	refers to the runtime, available to java compiler and JVM .

Example to specify the RetentionPolicy

1. `@Retention(RetentionPolicy.RUNTIME)`
2. `@Target(ElementType.TYPE)`
3. `@interface MyAnnotation{`
4. `int value1();`
5. `String value2();`
6. `}`

Example of custom annotation: creating, applying and accessing annotation

Let's see the simple example of creating, applying and accessing annotation.

Core Java

```
1. //Creating annotation
2. import java.lang.annotation.*;
3. import java.lang.reflect.*;
4.
5. @Retention(RetentionPolicy.RUNTIME)
6. @Target(ElementType.METHOD)
7. @interface MyAnnotation{
8. int value();
9. }
10.
11. //Applying annotation
12. class Hello{
13. @MyAnnotation(value=10)
14. public void sayHello(){System.out.println("hello annotation");}
15. }
16.
17. //Accessing annotation
18. class Test{
19. public static void main(String args[])throws Exception{
20.
21. Hello h=new Hello();
22. Method m=h.getClass().getMethod("sayHello");
23.
24. MyAnnotation manno=m.getAnnotation(MyAnnotation.class);
25. System.out.println("value is: "+manno.value());
26. }}
```

Output: value is: 10

How built-in annotations are used in real scenario?

In real scenario, java programmer only need to apply annotation. He/She doesn't need to create and access annotation. Creating and Accessing annotation is performed by the implementation provider. On behalf of the annotation, java compiler or JVM performs some additional operations.

@Inherited

By default, annotations are not inherited to subclasses. The @Inherited annotation marks the annotation to be inherited to subclasses.

```
1. @Inherited
2. @interface ForEveryone { } //Now it will be available to subclass also
3.
4. @interface ForEveryone { }
5. class Superclass{}
6.
7. class Subclass extends Superclass{}
```

Core Java

@Documented

The @Documented Marks the annotation for inclusion in the documentation.

Generics in Java

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects.

Before generics, we can store any type of objects in collection i.e. non-generic. Now generics, forces the java programmer to store specific type of objects.

Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

- 1) **Type-safety** : We can hold only a single type of objects in generics. It doesn't allow to store other objects.
- 2) **Type casting is not required:** There is no need to typecast the object.

Before Generics, we need to type cast.

1. List list = **new** ArrayList();
2. list.add("hello");
3. String s = (String) list.get(0);**//typecasting**

After Generics, we don't need to typecast the object.

1. List<String> list = **new** ArrayList<String>();
2. list.add("hello");
3. String s = list.get(0);

3) Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

1. List<String> list = **new** ArrayList<String>();
2. list.add("hello");
3. list.add(32);**//Compile Time Error**

Syntax to use generic collection

1. ClassOrInterface<Type>
Simple example to use Generics

1. ArrayList<String>

Core Java

Full Example of Java Generics

Here, we are using the ArrayList class, but you can use any collection class such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, Comparator etc.

```
1. import java.util.*;
2. class Simple{
3. public static void main(String args[]){
4. ArrayList<String> list=new ArrayList<String>();
5. list.add("rahul");
6. list.add("jai");
7. //list.add(32);//compile time error
8.
9. String s=list.get(1);//type casting is not required
10. System.out.println("element is: "+s);
11.
12. Iterator<String> itr=list.iterator();
13. while(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. }
17. }
```

Output:element is: jai

```
    rahul
    jai
```

Example of Java Generics using Map

Now we are going to use map elements using generics. Here, we need to pass key and value. Let us understand it by a simple example:

```
1. import java.util.*;
2. class Test{
3. public static void main(String args[]){
4. Map<Integer,String> map=new HashMap<Integer,String>();
5. map.put(1,"vijay");
6. map.put(4,"umesh");
7. map.put(2,"ankit");
8.
9. //Now use Map.Entry for Set and Iterator
10. Set<Map.Entry<Integer,String>> set=map.entrySet();
11.
12. Iterator<Map.Entry<Integer,String>> itr=set.iterator();
13. while(itr.hasNext()){
14. Map.Entry e=itr.next();//no need to typecast
15. System.out.println(e.getKey()+" "+e.getValue());
16. }
```

Core Java

```
17.  
18. })
```

Output:1 vijay

2 ankit

4 umesh

Generic class

A class that can refer to any type is known as generic class. Here, we are using **T** type parameter to create the generic class of specific type.

Let's see the simple example to create and use the generic class.

Creating generic class:

```
1. class MyGen<T>{  
2. T obj;  
3. void add(T obj){this.obj=obj;}  
4. T get(){return obj;}  
5. }
```

The **T** type indicates that it can refer to any type (like String, Integer, Employee etc.). The type you specify for the class, will be used to store and retrieve the data.

Using generic class:

Let's see the code to use the generic class.

```
1. class UseGen{  
2. public static void main(String args[]){  
3. MyGen<Integer> m=new MyGen<Integer>();  
4. m.add(2);  
5. //m.add("vivek");//Compile time error  
6. System.out.println(m.get());  
7. }}
```

Output:2

Understanding Type Parameters

The type parameters naming conventions are important to learn generics thoroughly. The commonly type parameters are as follows:

1. T - Type
2. E - Element
3. K - Key

Core Java

4. N - Number
5. V - Value
6. S,U,V etc. - 2nd, 3rd, 4th types

Generic Method

Like generic class, we can create generic method that can accept any type of argument.

Let's see a simple example of java generic method to print array elements. We are using here **E** to denote the element.

```
1. public class GenericMethodDemo{  
2.  
3.     public static < E > void printArray(E[] elements) {  
4.         for ( E element : elements){  
5.             System.out.println(element );  
6.         }  
7.         System.out.println();  
8.     }  
9.     public static void main( String args[] ) {  
10.        Integer[] intArray = { 10, 20, 30, 40, 50 };  
11.        Character[] charArray = { 'J', 'A', 'V', 'A', 'T','P','O','I','N','T' };  
12.  
13.        System.out.println( "Printing Integer Array" );  
14.        printArray( intArray );  
15.  
16.        System.out.println( "Printing Character Array" );  
17.        printArray( charArray );  
18.    }  
19. }
```

Output:Printing Integer Array

```
10  
20  
30  
40  
50
```

Printing Character Array

```
J  
A  
V  
A  
T  
P  
O  
I  
N  
T
```