JavaEra.com

A Perfect Place for All Java Resources

Core Java | Servlet | JSP | JDBC | Struts | Hibernate | Spring

Java Projects | FAQ's | Interview Questions | Sample Programs

Certification Stuff | eBooks | Interview Tips | Forums | Java Discussions

For More Java Stuff Visit

www.JavaEra.com

A Perfect Place for All Java Resources

Hi Friends!

Welcome to JavaEra.com.

As an admin, let me introduce myself to all of you.

I am Anil Reddy from Hyderabad. I'm a normal guy who lives life to the fullest, loves programming and lives a happy-go-lucky kind of life.

Why I started JavaEra.com?

I was a job seeker like you all but I was not sure about how to attempt an interview or various companies selection process. I had searched many Portals on the web & joined many job groups. From all those sources I was getting the information regarding latest job openings only but not any resources to chase job selection process by improving my skills.

Why we need those skills? Just go through this small story...

Once upon a time a very strong woodcutter asks for a job in a timber merchant, and he got it. The paid was really good and so were the work conditions. For that reason, the woodcutter was determined to do his best. His boss gave him an axe and showed him the area where he was supposed to work. The first day, the woodcutter brought 18 trees "Congratulations," the boss said. "Go on that way!" Very motivated for the boss' words, the woodcutter try harder the next day, but he only could bring 15 trees. The third day he try even harder, but he only could bring 10 trees. Day after day he was bringing less and less trees. "I must be losing my strength", the woodcutter thought. He went to the boss and apologized, saying that he could not understand what was going on.

"When was the last time you sharpened your axe?" the boss asked. "Sharpen? I had no time to sharpen my axe. I have been very busy trying to cut trees.

If we are just busy in applying for jobs, when we will sharpen our skills to chase the job selection process?

I don't have many friends who can suggest me or who can share some tips. I got irritated like anything. I thought that like me, there may be millions of my brothers and sisters across the nation facing same kind of problem.

So I strongly committed to start a special zone for all job seekers & IT professionals where everyone can sharpen their skills to find their dream job & in their dream company.

As a result in 10 months nearly 15000+ job seekers joined our family. Thousands of people got benefited by utilizing our resources.

This is just a start, what we have achieved till now is just 0.1% in our vision which is "*Creating a perfect place to share **Java** knowledge*".

Now I am not alone in chasing this challenge. We will work together by sharing knowledge & giving a ray of hope to millions of our brothers and sisters across the nation.

My Technical Skills:

Google SEO (Search Engine Optimization) Certified.

JSE (Certified), Servlets, JSP, Struts, Hibernate, Spring, PHP, Zend Framework.

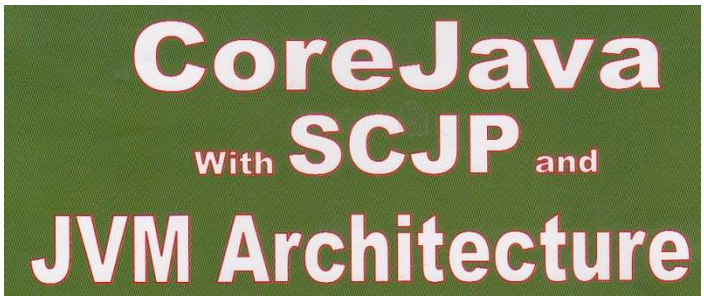
HTML, JavaScript, Jquery, AJAX, CSS, Adobe Photoshop CS6, Adobe Illustrator CS5, Adobe Flash.

My aim is to provide good and quality articles and content to readers and visitors to understand easily. All articlesare written and practically tested by me before publishing online. If you have any query or questions regarding any article feel free to leave a comment or You can get in touch with me On **anilreddy@JavaEra.com**, ***www.facebook.com*/JavaAnil**

Regards

Anil Reddy
Founder & Administrator
JavaEra.com
Email :
anilreddy77466@gmail.com,
anilreddy@JavaEra.com.
www.facebook.com/JavaAnil

Volume-2



www.JavaEra.com

Chapter 24

API & API Documentation

- In this chapter, You will learn
  - The difference between API and API Documentation
  - API and API Documentation file formats
  - Need of *javadoc* tool.
  - Working with API Documentation.
- By the end of this chapter- you will be comfortable in using Java API Documentation to develop Java Applications using predefined API.

Interview Questions

By the end of this chapter you answer all below interview questions

- Definition of API and API Documentation
- Differences between API and API Documentation
- For whom sake API and API documentation is generated?
- How API is distributed?
- How API Documentation is distributed?
- What files should be update in Classpath environment variable?
- Can we generate API documentation for our user defined classes?
- Java SE API Documentation folder structure

In this chapter we learn how we develop real time projects by using SUN given predefined classes and other team members given classes.

Definition and differences between API and API Documentation

- API stands for Application Programming Interface.
- The class that is used for developing a new class is called API. This new class again becomes API for other class developers. So, set of ".class" files is called API.
- The documentation given for that API is called API Documentation. And it is developed by using the documentation comment "/** */"
- From every .java file we develop two files ".class" and ".html".
- API is available in the format of bytecodes, means .class file
- API documentation is available in the format of HTML, means .html file
- .class files are generated by using "javac" command from java file.
- .html files are generated by using "javadoc" command from the same java file.

```
>javac Additon.java
        |-> Addition.class

>javadoc Addition.java
        |-> Addition.html
```

- .class files have only class, methods, and variables definitions.
- .html files have the comments written in java file for the classes, methods and variables plus their declarations (only prototypes)
- From the below program, after compilation both "add" methods are placed in Addition.class but where as javadoc command places class documentation, and only protected and public methods documentation with its prototype in .html file as shown below.

```
/**
This class is written to show API and API Documentation
call this program as Addition.java
*/
package ao;

public class Addition{
    /**
        This method takes two int numbers and print their addition result
    */
    public void add(int a, int b){
        System.out.println(a + b);
    }

    /**
        This method takes two Strings and print their concatenation result
    */
    void add(String s1, String s2){
        System.out.println(s1 + s2);
    }
}
```

javac →

Addition.class

```
class Addition
{
  p v add(int a, int b){ ---}
  v add(String a, String b)
  {----------}
}
```

javadoc →

Addition.html

```
documentation
class Addition
 p v add(int a, int b)
```

Note: In .html file we can only find *protected* and *public* members.

For whom API and API documentation is required?

- API documentation (.html files) is required for developer for developing new classes by using existing classes.
- API (.class files) is required for compiler and JVM to compile and execute those new classes.

Answer below question?

Q) How another class developer uses your class methods? Or how do you use methods from SUN given predefined classes? How do you know what is the method name, parameters, return type of that method?

A) You must get this information from API documentation. Every java source file developer must generate and distribute API documentation files (.html files) also.

How API is distributed?

API is distributed as jar file, and is available in our system by installing the software.

How API Documentation is distributed?

API documentation is distributed as zip file. It may or may not be distribute with software installer (.exe, .bin). If it is not distributed with that software, we must download it from that software vendor home site.

For example we get API for java software, but we do not get its API documentation with java software. We must download it from *oracle.com* (or you can get it from my DVD, "*03 SUN API Docs*" folder).

Updating Classpath environment variable

- We must update *API jar* file in Classpath to be used by Compiler and JVM.
- And we not need to update API documentation zip file in Classpath as it is used by developer.

Can we generate API documentation for our user defined classes?

Yes we can develop.

Procedure

1. Open cmd prompt
2. Change to Java file saved directory
3. Execute below command
   - javadoc <filename>

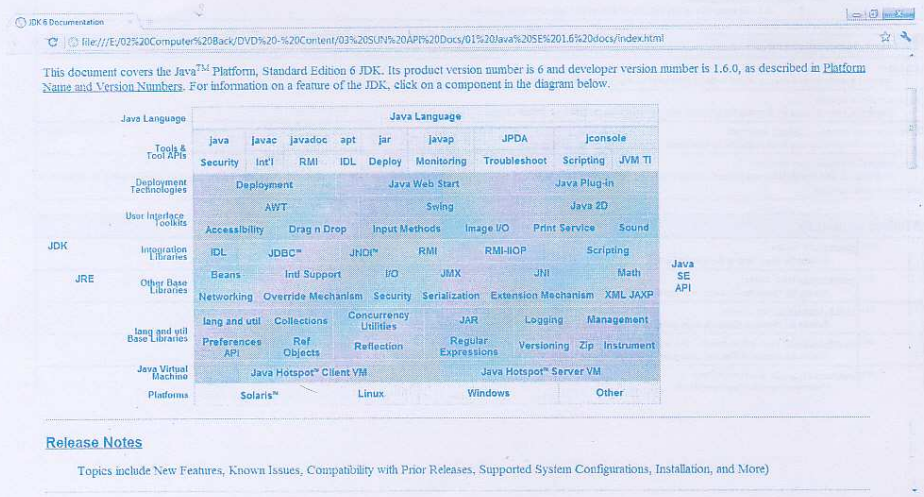
   For example
   - javadoc Example.java
     i. In the current directory you will find many html files and directories. Among them open "index.html" file to find your class html file.

Java SE API Documentation folder structure

You must download "Java SE" API documentation zip file form Oracle.com
After downloading Java SE documentation, extract it.
After extracting

- Open "Java SE 1.6 docs" folder
- You will find below folders
  - api
  - Images
  - jdk
  - jre
  - legal
  - platform
  - technotes
  - index.html

To get quick access on API documentation open "index.html" file, and you will find a big diagram with all concepts names as hyperlinks.



Click on the hyperlink of the concept to learn about it

The other way is

1. Open "api" folder
2. You will find below folders "*java* and *javax*"and a file "index.html"
3. Open index.html, you will see all Java SE packages as a list.
4. Click on the required package hyperlink. You will find all interfaces, classes, Exceptions, Errors, Enum and Annotations defined in that package.

Procedure to Open individual classes' API documentation file

+ If you want to open individual class's html file, open "java" folder, you will find its sub packages. Inside those sub packages you will find individual html file of every Java class.

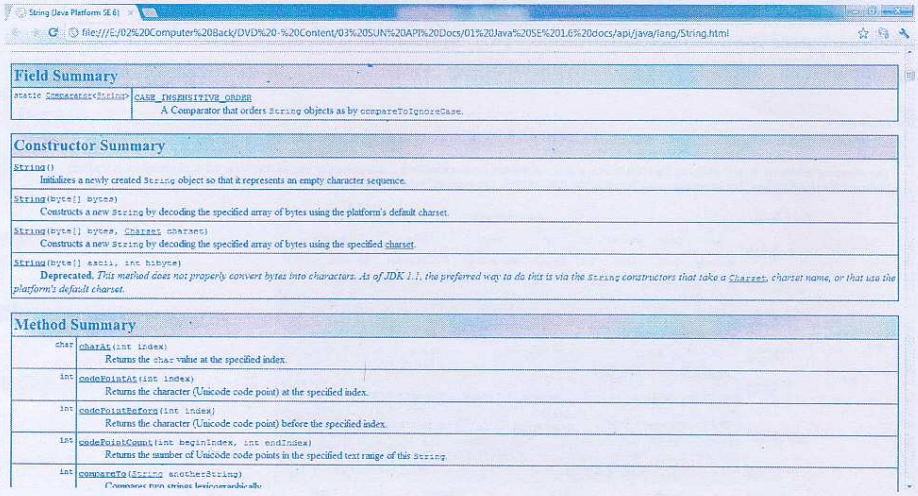
For example to open "String" class's html file open String.html from below path

"Java SE 6 Docs\api\java\lang\String.html"

+ In that class's html file you will find its protected and public members in the below order

1. static final variables - these are placed under Fields section
2. constructors - these are placed under Constructors section
3. methods - these are placed under Methods section.

In every section all members are placed in alphabetical order, below is the String class file



Field Summary

static Comparator<String>	CASE_INSENSITIVE_ORDER A Comparator that orders String objects as by compareToIgnoreCase.

Constructor Summary

String() Initializes a newly created String object so that it represents an empty character sequence.
String(byte[] bytes) Constructs a new String by decoding the specified array of bytes using the platform's default charset.
String(byte[] bytes, Charset charset) Constructs a new String by decoding the specified array of bytes using the specified charset.
String(byte[] ascii, int hibyte) **Deprecated.** *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a Charset, charset name, or that use the platform's default charset.*

Method Summary

char	charAt(int index) Returns the char value at the specified index.
int	codePointAt(int index) Returns the character (Unicode code point) at the specified index.
int	codePointBefore(int index) Returns the character (Unicode code point) before the specified index.
int	codePointCount(int beginIndex, int endIndex) Returns the number of Unicode code points in the specified text range of this String.
int	compareTo(String anotherString) Compares two strings lexicographically.

Chapter 25

Fundamental Classes

- In this chapter, You will learn
  - Overview on *java.lang* package and its classes hierarchy
  - Need of java.lang.Object class
  - Need of java.lang.Class class.
  - objects comparison
  - retrieving object hashcode
  - customizing printing object information
  - retrieving object class name
  - objects cloning
  - objects finalization
  - Locking and unlocking objects
- By the end of this chapter- you will be comfortable in using all 11 methods of Object class.

i

Interview Questions

By the end of this chapter you answer all below interview questions

- Overview of the *java.lang* package

The **Object** class

- Why Object class is the super class for all java classes
- Why its name is Object?
- What are the operations defined in Object class common for all subclasses?
- Explain all 11 methods of Object class.
- Why equals() method is defined when we have == operator?
- Difference between "==" operator and equals() method.
- Contract between equals() and hashCode() methods overriding.
- If object state is changed will object hash code also be changed?
- Introduction to ***java.lang.Class***
- How can an object's class name is known?
- When sub class must override below methods
  - equals()
  - hashCode()
  - toString()
- What is a marker interface?
- Introduction to a marker interface ***java.lang.Cloneable***
- Is clone() method is defined in Cloneable interface?
- Cloning object, rule on object cloning?
- What are the 2 Types of cloning?
- How can you execute some logic before object is destroying?
- Why clone() and finalize() methods are declared as protected?
- Why wait(), notify(), notifyAll() methods are given in Object class instead of in Thread class?

Fundamental Classes

Overview of the *java.lang* package

This package is called default package, because JVM loads all classes of this package automatically. So to use this package classes we no need write import statement.

This package provides classes that are fundamental to the design of the Java programming language. The most important classes are *Object*, which is the root of the class hierarchy, and *Class*, instances of which represent classes at run time.

This package provides classes to perform operations on character stings, they are *Sting, StringBuffer, and StringBuilder* and also provide classes that used to represent primitive types as if it were objects, and these classes are called *Wrapper classes*.

This package provides some other useful classes to signal the logical mistakes in the program and to terminate the program execution if the desired condition is not met; those classes are *Throwable, Error and Exception*.

This package also has classes to create custom threads to have concurrent execution using Runnable, Thread and ThreadGroup classes.

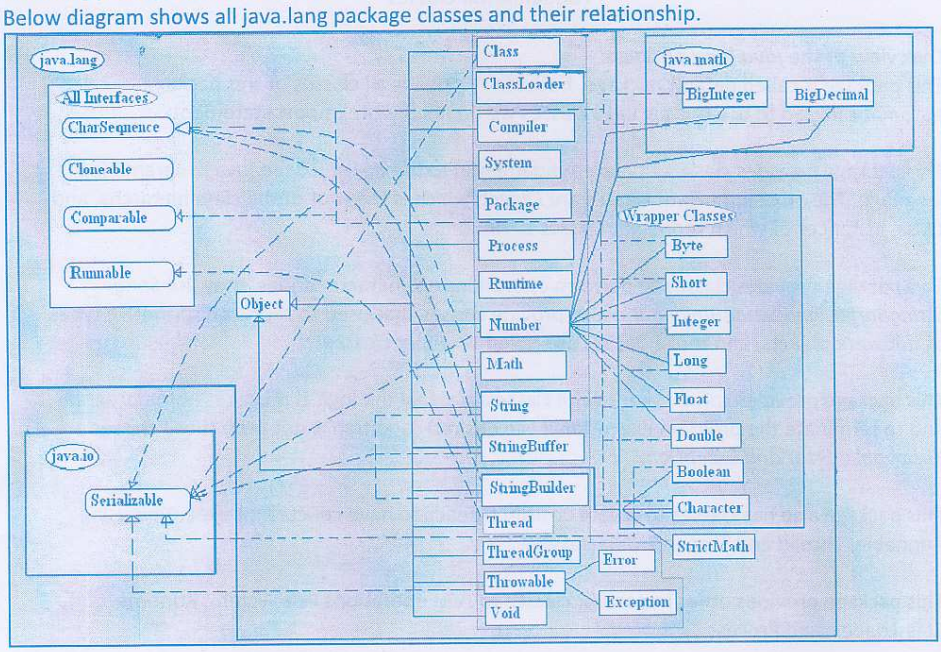
This package provides other important classes to JVM operations like System, Runtime, ClassLoader, and Process.

+ The fundamental classes such as

1. ***Object*** - it is root class of all Java user defined and predefined classes.
2. ***Class*** - JVM creates this class object to store the all Java classes' byte codes in JVM's method area.
3. ***String, StringBuffer, StringBuilder*** - these classes are responsible to represent character strings.
4. ***wrapper classes (Byte, Short, Integer, Long, Float, Double, Character, Boolean)*** – are responsible to convert primitive type to object and vice versa.
5. ***Throwable, Exception, Error*** - responsible to handle runtime errors.
6. ***Thread, ThreadGroup*** - are responsible to create and manage user defined threads..
7. ***System, Runtime, ClassLoader, Process, SecurityManger*** - These classes are responsible in providing "system operations" that manage the dynamic loading of classes, creation of external processes, host environment inquiries such as the time of day, and enforcement of security policies.

Java.lang package class hierarchy

Below diagram shows all java.lang package classes and their relationship.



The Object class

Object is the root or super class of the class hierarchy. Every predefined and user defined classes are sub classed from Object class.

Why Object class is the super class for all java classes?

Because of below 2 reasons Object class is developed as super class for every class

1. **Reusability** - every object has 11 common behaviours. These behaviours must be implemented by every class developer. So to reduce burden on developer SUN developed a class called Object by implementing all these 11 behaviours with 11 methods. All these 11 methods have generic logic common for all subclasses. If this logic is not satisfying to subclass requirement then subclass should override it.

2. **To achieve runtime polymorphism**, So that we can write single method to receive and send any type of class object as argument and as return type.

Why this class name is chosen as Object?

As per coding standards, class name must convey the operations doing by that class. So this class name is chosen as Object as it has all operations related to object. These operations are chosen based on real world object's behaviour.

What are the common functionalities of every class object?
Every object contains 11 common operations - list is given below.

1. **Comparing two objects**
   public boolean equals(Object obj)

2. **Retrieving hashcode i.e; object identity**
   public *native* int hashCode()

3. **Retrieving object information in String format for printing purpose**
   public String toString()

4. **Retrieving the runtime class object reference**
   public final native Class getClass()

5. **Cloning object**
   protected *native* Object clone() throws CloneNotSupportedException

6. **executing object clean-up code / resources releasing code just before object is being destroyed**
   protected void finalize() throws Throwable

7. 8. 9. **Releasing object lock and sending thread to waiting state**
   public final void wait() throws InterruptedException
   public final *native* void wait(long mills) throws InterruptedException
   public final void wait(long mills, int nanos) throws InterruptedException

10. 11. **Notify about object lock availability to waiting threads**
   public final *native* void notify()
   public final *native* void notifyAll()

All above methods are implemented with generic logic that is common for all subclass objects. So that developer can avoid implementing these operation in every class, also SUN can ensure that all these operations are overridden by developers with the same method prototype. This feature provides runtime polymorphism.

Take *toString()* method as an example, it is called in PrintStream class in print and println methods on Object class referenced variable, but at runtime this method is executed from the subclass class based on the object we are printing.

If SUN given logic in these methods is not satisfying our class business requirements we can override them. Basically SUN implemented all these 11 methods with object's reference.

Developer should override these methods with object state.

Q) What are the methods we can override in subclass from object class?
We can override below five methods as they are not final methods.

1. equals
2. hashcode
3. toString
4. clone
5. finalize

Q) When should we override equals and hashcode methods in subclass?
To use a subclass object as key to add entry to Map objects and also to add its objects as element to Set collection, that class must override *equals* and *hashCode* methods. If these two methods are not overridden no CE, no RE, but those objects are not found for retrieving or removing because these methods are executed from java.lang Object class.

Q) When should we override toString() method?
To print object data we must override toString() method by returning object's non-static variables value (state).

Overriding above methods in subclass
It is important to know about each method clearly.

Let us start our discussion with ***equals()*** method

Operation#1: Comparing two objects (understanding equals() method)
equals() method is given to compare two objects for their *equality*.
In Java we can compare objects in two ways either

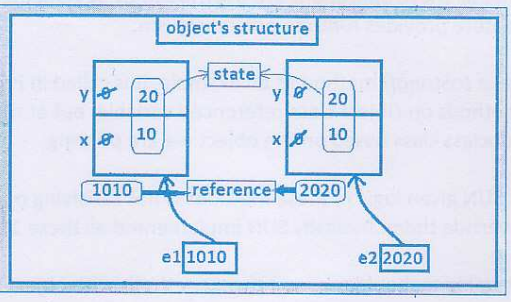
1. By using **objects reference** or
2. By using **objects state**

If two objects of a class are said to be equal only if they have same *reference* or *state*.
So we have two ways to compare objects either by

1. Using **"=="** operator – It always compares objects with their references.
2. Using **equals** method - It compares objects based on its implementation.

Let us assume a class called "Example" it has state x and y variables. Below are the two objects created from this class, find out their state and references?

```
class Example
{
        int x = 10;
        int y = 20;

        public static void main(String[] args)
        {
                Example e1 = new Exammple();
                Example e2 = new Exammple();
        }
}
```



The above two objects e1 and e2

- are equal based on their state and
- are not equal based on their reference

equals() method implementation (logic) in Object class

In Object class this method is implemented to compare two objects with their references, as shown below.

```
public boolean equals(Object obj){
        return (this == obj);
}
```

This method returns true if the current object reference is equal to argument object reference, else it returns false.

If we want to compare two objects with state we must override this method in subclass.

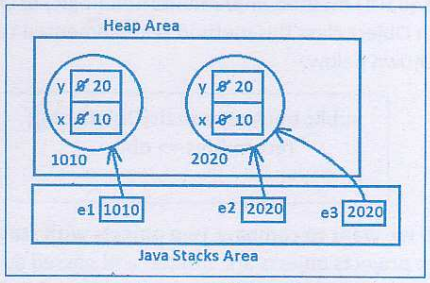
In projects objects are created and passed as method arguments at runtime. So, to know the given object is expected one or not, we should compare that object with our own created object for their equality.

```
/*
Below applications shows comparing two primitive values and objects
        using   "==" operator and "equals" method
Call this program as Comparision.java
*/

class Test{
}

class Example{
        int x = 10;
        int y = 20;
}

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

                // Comparing primitive values and objects using "==" operator
                int x = 10;
                int y = 20;
                int z = 10;

                System.out.println(x == y);
                System.out.println(x == z);

                //Rule: we cannot use "==" operator to compare incompatible type values
                boolean bo = true;
                //System.out.println(x == bo); // CE: incomparable types: int and boolean
```

```
//== operator compares object's references, not state (values)
//It returns true if both variables store same object reference, else return false

//objects creation
Example e1 = new Example();
Example e2 = new Example();
Example e3 = e2;

System.out.println(e1 == e2);
System.out.println(e2 == e3);
```

Focus: if "==" returns true then we can say both referenced variables are pointing to same object

JVM architecture




```
//Rule: we cannot use "==" operator to compare incompatible referenced types

Test t1 = new Test();
//System.out.println(e1 == t1); //CE: incomparable types: Example and Test

//comparing objects using Object class equals method

/*
In Object class we have equals() method to compare two objects.
It also compares two objects with their references like == operator.
Inside this method logic == operator is used as shown below

        public boolean equals(Object obj){
                return this == obj;
        }
*/

System.out.println(e1.equals(e2)); // false
System.out.println(e2.equals(e3)); // true
System.out.println(e1.equals(t1)); //false  (No compile time error)

/*
Note: equals() method returns false for incompatible types comparison.
It does not throw CE, because compiler checks only equals() method can be
invoked on e1 object by passing t1 object, since it is possible so no CE.

JVM returns false because it only checks values not types. The object reference
stored in e1 referenced variable is different from t1 reference value, so ==
operator returns false from equals() method.
*/
```

```
        //comparing two nulls using == operator

        //comparing null with other null returns true
        System.out.println(null == null); // true

        //comparing null with null reference variable returns true
        Example e4 = null;
        System.out.println(null == e4); // true

        //comparing null with object reference variable returns false
        Example e5 = new Example();
        System.out.println(null == e5); // false

        //comparing null with object also returns false
        System.out.println(null == new Example()); // false

        //comparing two nulls using equals() method

        //we cannot invoke method using null leads to compile time error
        //System.out.println(null.equals(null)); // <nulltype> cannot be dereferenced

        //we cannot invoke method using null referenced variable leads to exception
        Example e6 = null;
        //System.out.println(e6.equals(null)); // RE: NPE

        //we can invoke equals() method by passing null
        Example e7 = new Example();
        System.out.println(e7.equals(null)); // false

    }
}
```

Q) Why *equals()* method is given when == operator is already available to compare objects?
"==" operator compares only reference of the objects, so equals() method is given to compare objects with state. This method is by default implemented with reference comparision, so we must override it in subclass to compare its instances with state.

***Q)* In Object class, why equals() method is defined to compare two objects with reference *not with state*?** It is because of below two reasons

- Every object may not contain state and also Object class developer does not know the sub class required object state comparison logic.

What are the differences between "==" operator and equals() method?

== operator	equals() method
1. It always compares objects with references.	1. It compares objects either with reference or with state based on its implementation. ➢ In Object class it compares objects with reference ➢ In subclass it compares objects with state
2. We cannot compare incompatible objects, compiler throws CE	2. We can compare incompatible objects using equals() it returns "false".
3. We can use it also for comparing primitive values and two nulls directly.	3. We cannot use it for comparing primitive values and two nulls directly it leads to CE, because a method cannot be invoked by using *primitive types* and *null* value

```
/*
Below application shows overriding equals method for comparing Student objects with their
state. As it is a real world object, its objects must be compared with state not with reference.

Contract: as per the above default implementation overriding equals method also should
return false if the argument object is incompatible or null.

call this program as Student.java
*/
public class Student{
        private int     sno;
        private String  sname;
        private int     whichClass;

        //defining constructor to initialize object with user given values
        public Student(int sno, String sname, int whichClass){
                this.sno        = sno;
                this.sname      = sname;
                this.whichClass = whichClass;
        }
        //overriding equals() method
        public boolean equals(Object obj){

                if(this == obj){
                        return true;
                }
                else{
                        if (obj instanceof Student){
                                Student  s = (Student)obj;

                                return  this.sno == s.sno &&
                                        this.sname.equals(s.sname) &&
                                        this.whichClass == s.whichClass;
                        }
                        else{
                                return false;
                        }
                }
        }
}

//Address.java
class Address{}
```

//comparing both objects reference, if references in variables are same return true, because same object is passed, else compare state

//comparing state, checking passed object IS-A Student or not, and casting passed object into Student, to call variables

//comparing state of current object and passed object

//As per equals method contract, returning false if objects are incompatible.

One dummy incompatible class for comparing its object with Student class object using equals method

```
//Test.java
class Test{
        public static void main(String[] args) {
                Student s1 = new Student(1, "Hari", 12);
                Student s2 = new Student(2, "Krishna", 12);
                Student s3 = new Student(1, "Hari" , 12);
                Student s4 = s2;

                System.out.println(s1 == s2); //false, different references
                System.out.println(s1.equals(s2));//false, different state
                System.out.println();

                System.out.println(s1 == s3); // false, different references
                System.out.println(s1.equals(s3));// true, same state => but objects are different

                System.out.println(s2 == s4); // true
                System.out.println(s2.equals(s4)); //true, both reference variables has same
                                                        object reference

                Address add = new Address();

                //System.out.println(s1 == add); //CE: incomparable types: Address and Student
                System.out.println(s1.equals(add)); //false, incompatible objects comparison, it
                System.out.println(add.equals(s1); //false                          returns false

                //comparing two nulls
                System.out.println(null == null); //true
                //System.out.println(null.equals(null)); //CE: <nulltype> cannot be dereferenced

                Address a1 = null;
                Address a2 = null;

                //comparing null with null using == operator always returns true,
                //but equals() throws NPE
                System.out.println(a1 == a2); //true
                //System.out.println(a1.equals(a2)); //NPE

                //comparing null with object always returns false
                Address a3 = new Address();
                System.out.println(a3 == a2); //false
                System.out.println(a3.equals(a2)); //false

                System.out.println(s2.equals(a2)); //false
        }
}
```

Operation#2: Retrieving hashcode of an object (understanding hashCode() method)

Hashcode is an identity of an object. It is used to differentiate one object from another object. Every object has its own unique hashcode. As per javadocs defined "hashcode is typically implemented by converting the internal address of the object into an integer number".

Q) When hashcode of an object is used?

A) It is used by hashtable data structure to store, retrieve, remove, and search the object in Set and Map collection objects.

Q) How can we retrieve hashcode of an object?

By calling hashCode() method, it is defined in java.lang.Object class.

It is a native method and its default implementation is returning the reference of the object in integer form. The prototype of this method is:

public native int hashCode()

Q) Can we generate custom hashcode for our object?

Yes, we can also generate hashCode of the object by using its state. In this case we must override hashCode() method in subclass by returning state of the object by developing some hashing algorithm.

Q) So in how many ways we can generate hashcode of an object?

A) In two ways

1. By using its reference
   It is the default implementation of JVM and that hashcode number is returned through hashCode() method of java.lang.Object class.
2. By using its state
   It is the overriding implementation developed by subclass developer by overriding hashCode() method in subclass.

Q) If we change state of the object is its hashcode changed?

It is depending on hashCode() method implementation:

- If hashCode is generated by using object reference then its hashCode is not changed when object state is changed.
- If hashCode() method is overridden in subclass for generating hashcode by using object's state then object hashCode is changed when its state is changed.

Q) Can two objects have same hashcode?

Yes there is a possibility, it is depending on hashCode() method implementation:

- JVM always generates different hashcode for objects because objects always have different references.
- But developer overriding hashCode() method may gives same hashcode for multiple objects because objects of same class can have same state.

```
/*
Below applications shows
  ▪ working with hashcode() method
  ▪ contract between hashCode() and equals() method

  If equals() method is overridden then hashcode() method must be overridden with below
  contract,
  ➢ If equals() method returns true by comparing two objects, then hashcode of the both
    objects must be same.
  ➢ If it returns false, the hashCode of the both objects may or may not be same.

call this program as HashCodeDemo.java
*/

class Example{}

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

                Example e1 = new Example();
                Example e2 = new Example();

                System.out.println(e1.hashCode()); //1671711
                System.out.println(e2.hashCode()); //11394033

                /* Checking contract between equals() and hashCode() methods
                If equals method returns true, hashcode of both objects must be same */

                System.out.println(e1 == e2); //false
                System.out.println(e1.equals(e2)); //false
                System.out.println(e1.hashCode() == e2.hashCode()); //false

                Example e3 = e2;
                System.out.println(e2 == e3); //true
                System.out.println(e2.equals(e3)); //true
                System.out.println(e2.hashCode() == e3.hashCode()); //true

                /* In the below case the above contract is failed, as equals() method in Student
                class compares state of the objects */

                Student s1 = new Student(1, "Hari", 12);
                Student s2 = new Student(1, "Hari", 12);
                System.out.println(s1.equals(s2)); //true
                System.out.println(s1.hashCode() == s2.hashCode()); //false
```

```
        /* Conclusion
        So to satisfy the contract of equals and hashCode() we should always override
        hashCode() method when we override equals() method. If we do not override
        hashCode() it does not leads to CE or RE, you face business problems. i.e the
        objects you added to Set and Map collection objects they are not found further.
        You will understand it more in collection framework chapter */
    }
}

/*
Below program shows overriding hashcode method with object state
It is always recommended to use the non-static variables in hashcode method those are used
in equals method. This approach gives better results; you can also use some other variables
but should ensure the contract is satisfying.
*/

//Student.java
class Student{
    int sno;        String sname;        int whichStd;

    Student(int sno, String sname, int whichStd){
        this.sno        = sno;
        this.sname      = sname;
        this. whichStd  = whichStd;
    }
    public boolean equals(Object obj){
        if (this == obj){ return true; }
        else if (obj instanceof Student){
            Student s1 = (Student)obj;
            return ( this.sno == s1.sno &&
                this.sname.equals(s1.sname) &&
                this.whichStd == s1.whichStd);
        }
        else {
            return false;
        }
    }
    public int hashCode(){
        return (sno + sname.length() + whichClass);
    }
}
```

Use some hashing algorithm as per your project need, here I just adding all three variables to generate hashcode based on state

```
//Test.java
class Test{
        public static void main(String[] args){

                Student s1 = new Student(1, "HariKrishna", 9);
                Student s2 = new Student(1, "HariKrishna", 9);

                System.out.println(s1.equals(s2)); //true
                System.out.println(s1.hashCode() == s2.hashCode()); //true

        }
}
```

Now equals() & hashCode() method's contract is satisfied correctly

Q) When we override hashCode() method in subclass, is subclass object contains JVM generated hashcode number?

A) Yes, every object compulsory has JVM generated hashcode even though we overriding hashCode() method in subclass. In this case we have two hashcodes to object, and always we get the hashcode number that is returned by overriding hashCode() from subclass.

Q) How can we get JVM generated hashcode of the subclass object?

There are two ways

1. By calling *System.identityHashCode()* method. It is a static native method defined in System class to return JVM generated hashcode of the given object. It prototype is:
   public static native int identityHashCode(Object obj)
2. By calling super.hashCode() method from a new method that is defined in the subclass.

Below program shows retrieving JVM generated hashcode in the above two approaches.

```
class A {
    int x;
    A(int x){
        this.x = x;
    }
    public int hashCode() {
        return x;
    }
    public int JVMHC(){
        return super.hashCode();
    }
}
```

```
class IdentityHashCodeTest{
    public static void main(String[] args){
        A a1 = new A(5);
        A a2 = new A(5);
        A a3 = new A(6);

        System.out.println( a1.hashCode() );
        System.out.println( a2.hashCode() );
        System.out.println( a3.hashCode() );

        System.out.println( System.identityHashCode( a1 ) );
        System.out.println( System.identityHashCode( a2 ) );
        System.out.println( System.identityHashCode( a3 ) );

        System.out.println( a1.JVMHC() );
        System.out.println( a2.JVMHC() );
        System.out.println( a3.JVMHC() );
```

Output:

Operation# 3: retrieving object information in string format (understanding toString())

toString method is used to retrieve object information in string form.

Every object has below information

1. Its class name
2. Its hashcode and
3. Its state

In Object class toString() method is implemented for returning class name and hashcode of its current object in the format: *classname@hashcode in hex string format*

The prototype of this method is

public String toString()

If we want to return object's state from this method we must override it in subclass.

toString method implementation logic in Object class:

```
public String toString(){
        return  getClass().getName()  + "@" + Integer.toHexString(hashCode()) ;
}
```

Focus: toString() method is internally called from println() method when we print object.

What is the output in the below cases?

Case# 1: toString() method is not overridden in A class

```
class A{
        int x;
        A(int x){
                this.x = x;
        }
}
```

```
class Test{
        public static void main(String[] args){
                A a1 = new A(5);
                A a2 = new A(6);

                System.out.println(a1);
                System.out.println(a2);
        }
}
```

Case# 2: toString() method is overridden in print A class objects state(data)

```
class A{
        int x;
        A(int x){
                this.x = x;
        }
        public String toString(){
           return ""+x;
        }
}
```

```
class Test{
        public static void main(String[] args){
                A a1 = new A(5);
                A a2 = new A(6);

                System.out.println(a1);
                System.out.println(a2);
        }
}
```

What is the output you get if Student class objects are printed, Check Student class definition in previous pages which we developed for overriding equals() and hashCode() methods?

```
//College.java
class College{
        public static void main(String[] args){

                Student s1 = new Student(1, "HariKrishna", 9);
                Student s2 = new Student(1, "HariKrishna", 9);

                System.out.println(s1); //Student@addbf1
                System.out.println(s1); //Student@19821f
        }
}
```

Q) Is the above output meaningful?

No.

To get meaningful output - means Student object state, **override toString()** method in Student class to return current Student object state in String format

Place below toString method in Student class

```
public String toString() {
        return "sno: "+sno + "\n" +
                "name:" + sname +"\n" +
                "class: "+ whichStd +"\n" ;
}
```

Now compile and execute College.java again you will get below output.

s1 object:	**s2 object:**
sno: 1	sno: 1
name: HariKrishna	name: HariKrishna
class: 9	class: 9

Conclusion

Q) When should we override equals(), hashCode() and toString() methods in subclass?

We should override

equals()	- to compare objects with state
hashCode()	- to generate objects hashCode using state, for satisfying contract
toString()	- to return object state in String format

Point to be remembered

We must override equals() and hashCode() methods in subclass to add subclass objects as element/key to Set and Map collection objects and further to search this element with new object. If we do not override these methods this element is not found.

Operation# 4: retrieving current objects runtime class object's reference (Understanding getClass() method)

Runtime class: The class that is loaded into JVM at execution time is called runtime class.
Runtime class object: Every class bytecodes are stored using java.lang.Class object. This java.lang.Class object is called runtime class object.

So to retrieve the class name of the current object we must first get this object runtime class object's reference. For this purpose we must use *getClass()* method that is defined in java.lang.Object class. Its prototype is:

public final native Class getClass()

Q) How can we get the class name of a given object?
A) We must call a method getName() on this object's runtime class object. This getName() method is defined in java.lang.Class class. Its prototype is:

public String getName()

Below code shows retrieving a1 object's class name:

```
A a1 = new A();

Class  cls = a1.getClass();
String name = cls.getName();
```

Above lines of code we can also write in single line as shown below:

```
String name = a1.getClass().getName();
```

Q) Develop a class to accept all types of object as argument. Then print that object's class name from that class.

```
//A.java
class A{
        static void m1(Object obj){

            String name = obj.getClass().getName();

            System.out.println("
                The passed object is of type: "+ name);
        }
}
```

```
//B.java
class B{}
```

```
//C.java
class C extends B{}
```

```
//Test.java
class Test{
    public static void main(String[] args){
                String s1 = "abc";
                Integer io = 50;
                Object obj = new A();
                B b1 = new B();
                B b2 = new C();

                A.m1(s1);
                A.m1(io);
                A.m1(obj);
                A.m1(b1);
                A.m1(b1);
        }
}
```

Operation #5: Object cloning (understanding clone() method)
Cloning object means creating duplicate copy with current object state is called object cloning.
To perform cloning we must call ***Object class*** **clone()** method.

Below is the prototype of clone() method
protected native Object clone() throws CloneNotSupportedException

Rule: To *execute* clone() method on an object its class must be subclass of ***java.lang.Cloneable*** interface, else this method throws exception ***"java.lang.CloneNotSupportedException"***

What is Cloneable interface?
Cloneable is a marker interface which is an empty interface. It provides permission to execute clone() method to clone the current object.

Why are we implementing Cloneable interface when we do not implement any method?
To provide permission to execute clone() method logic on this class instance.

Dose cloned object has same original object reference and hashcode?
No, both objects original object and cloned objects have different reference and hashcode because object cloning creates new object but both objects have same state. Since they are different objects the modification performed on one object is not effected to another object.

Programming rule in calling clone() method
We must follow below four rules in calling clone() method

1. clone() method can be called on a class object only inside that class because it is protected. If we call it in other classes including in subclass it leads to CE. To call it from other classes we must override it in that subclass with public keyword.
2. We must cast the clone() method retuned object to its current objects class, because it is returning that object as java.lang.Object.
3. clone() method's calling method should handle CloneNotSupportedException either by catching it using try/catch or by reporting it using throws keyword.
4. To execute clone() method the current object should be Cloneable type else it leads to exception CloneNotSupportedException.

Consider above points and find out
Is below program compiled and executed fine to clone current object?

```
class Example{
        int x = 10, y = 20;

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

        Example e1 = new Example();
        Example e2 = (Example)e1.clone();
    }
}
```

```
class Example implements Cloneable{
        int x = 10, y = 20;

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

        Example e1 = new Example();
        Example e2 = (Example)e1.clone();
    }
}
```

What is the output from below program?

```
class Example implements Cloneable{
        int x = 10, y = 20;

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

        Example e1 = new Example();
        e1.x = 5;
        e1.y = 6;

        Example e2 = (Example)e1.clone();

        System.out.println(e1 == e2);
        System.out.println(e1.hashCode() == e2.hashCode());

        System.out.println(e1.x +"..."+ e1.y);
        System.out.println(e2.x +"..."+ e2.y);

        e2.x = 8;
        e2.y = 9;

        System.out.println();
        System.out.println(e1.x +"..."+ e1.y);
        System.out.println(e2.x +"..."+ e2.y);
  }
}
```

Are below programs compiled?

```
class Test{
  public static void main(String[] args)
        throws CloneNotSupportedException{

        Example e1 = new Example();
        Example e2 = (Example)e1.clone();
  }
}
```

```
class Test extends Example{
  public static void main(String[] args)
        throws CloneNotSupportedException{

        Example e1 = new Example();
        Example e2 = (Example)e1.clone();
  }
}
```

Q) How can we call clone() method on Example class objects from its user classes?

A) We must override clone() method in Example class with *public* accessibility modifier.

Procedure:

1. In this overriding method we must call Object class clone method because we are overriding clone() method only for changing accessibility modifier not for changing logic, so we should return the cloned object that is returned by super.clone()
2. We can implement covariant returns in overriding clone method, means we can override clone() method with *Example* as return type. So that user class developer no need to downcast the cloned object to Example type. But in overriding clone() method we must downcast it to Example type before returning.

Below program shows overriding clone() method in Example to clone its objects in user classes

```
class Example implements Cloneable{
        int x = 10, y = 20;

        public Example clone()
                throws CloneNotSupportedException{
                        return (Example) super.clone();
        }
}
```

Now Test class can be compiled and executed without errors and more over we no need to downcast the cloned object to Example since we implemented covariant returns.

```
class Test{
  public static void main(String[] args) throws CloneNotSupportedException{

        Example e1 = new Example();
        Example e2 = e1.clone();

        System.out.println(e1);
        System.out.println(e2);
  }
}
```

Cloning with HAS-A relation:

Q) When an object is cloned is its internal object also cloned?

A) No, clone() method does not clone internal objects. It only clones current object's non-static variables memory. So if current object contains referenced variables only these referenced variables memory is cloned with the same reference but not the object pointing by this referenced variable. Then this internal object is pointed from both current object and cloned object referenced variables.

What is the output from the below program?

```
class Example implements Cloneable{
        A a = new A();

        public static void main(String[] args){

                Example e1 = new Example();
                Example e2 = (Example)e1.clone();

                System.out.println( e1.a == e2.a);
        }
}
```

2 types of cloning

Q1) What are the different types of clonings in Java?

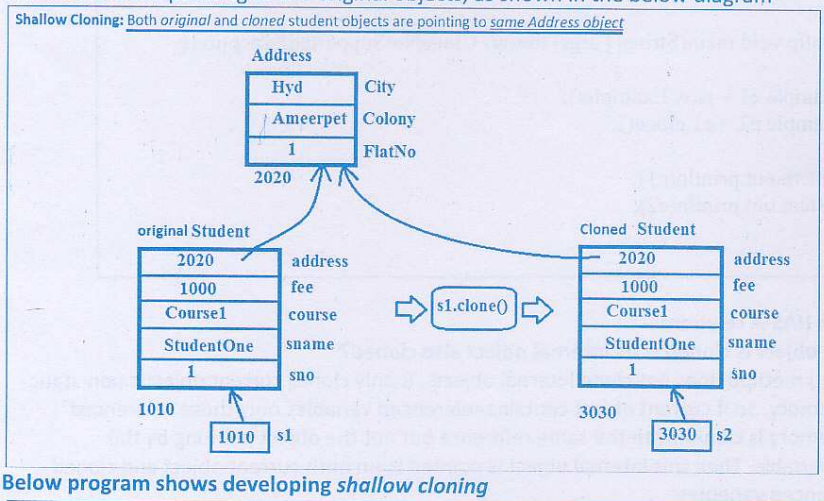
Java supports two type of clonings:-

1. **Shallow cloning** and
2. **Deep cloning**.

Object class method clone() does shallow cloning by default.

Q2) What is Shallow cloning?

In shallow cloning the object is copied without its contained objects.
Shallow clone only copies the top level structure of the object not the lower levels.
It is an exact bit copy of all the attributes. So in shallow cloning, the cloned objects referenced variables are still pointing to the original objects, as shown in the below diagram



Below program shows developing *shallow cloning*

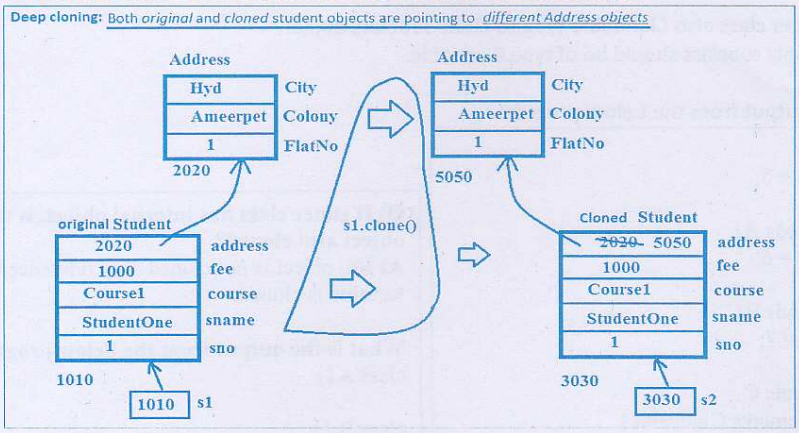
```
public class Address{
  int     flatNo      = 1;
  String colony       = "Ameerpet";
  String city         = "Hyd";
}
```

```
public class Student{
  int     sno     = 1;
  String sname  = "StudentOne";
  String course = "Course1";
  double fee    = 1000;
  Address add = new Address();

  public static void main(String[] args)
          throws CloneNotSupportedException{
    Student s1 = new Student();
    Student s2 = (Student) s1.clone();
    System.out.println(s1.add == s2.add);  //true
  }
}
```

add variable is pointing to the same Address object from s1 and s2 objects, since s1.clone() method does shallow cloning. So == operator returns true.

Q3) What is deep cloning and how it can be achieved?

In deep cloning the object is copied along with its internal objects. Deep clone copies all the levels of the object from top to the bottom recursively. *Developer must develop deep cloning by overriding clone() method.* Below diagram shows deep cloning.



Below program shows developing student objects deep cloning.

Procedure:

Step #1: Derive Address class from Cloneable

Step #2: Override clone method in Address class and Student class as public

Step #3: Call Address class clone method on *s1.add* object in Student class clone method and

Step #4: Store that cloned object reference in *s2.add*

```
public class Address implements Cloneable{
  int    flatNo    = 1;
  String colony    = "Ameerpet";
  String city      = "Hyd";

  public Address clone()
      throws CloneNotSupportedException{

    return (Address) super.clone();
  }
}
```

```
public class Student{
  int    sno    = 1;
  String sname  = "StudentOne";
  String course = "Course1";
  double fee    = 1000;
  Address add = new Address();

  public Student clone()
          throws CloneNotSupportedException {
    Student s = (Student) super.clone();
    s.add     = this.add.clone();
    return s;
  }
  public static void main(String[] args)
          throws CloneNotSupportedException{
    Student s1 = new Student();
    Student s2 = s1.clone();
    System.out.println(s1.add == s2.add);  //false
  }
}
```

Cloning with IS-A relation:

Q) When an object is cloned is its super class non-static variables are also cloned?

A) Yes, clone() method clones the object inheritance graphs stats from root super class to current cloning subclass, i.e.. from java.lang.Object to D class as per below example.

Q) Should super class also Cloneable type to clone subclass object?

A) No need, only subclass should be of type Cloneable.

What is the output from the below program?

```
class A{
        int p = 5;
}
class B extends A{
        int q = 6;
}
class C extends B{
        int r = 7;
}
class D extends C
        implements Cloneable{

        int s = 8;

        public String toString(){
                return  "p: "+p +"\n" +
                        "q: "+q +"\n" +
                        "r: "+r +"\n" +
                        "s: "+s;
        }
        public static void main(String[] args){

                D d1 = new D();
                D d2 = (D)d1.clone();

                Sopln( "d1 object data\n"+ d1);
                Sopln ( "d2 object data\n"+ d2);

                d2.p = 12; d2.q = 13;
                d2.r = 14; d2.s = 15;

                Sopln("\nAfter modification");
                Sopln( "d1 object data\n"+ d1);
                Sopln( "d2 object data\n"+ d2);
        }
}
```

Q) If super class has internal object, is that object also cloned?

A) No, object is not cloned only referenced variable is cloned.

What is the output from the below program?

```
class A{}

class B {}

class C {
        A a1 = new A();
}
class D extends C
        implements Cloneable{

        B b1 = new B();

        public static void main(String[] args){

                D d1 = new D();
                D d2 = (D)d1.clone();

                Sopln( d1.a == d2.a);
                Sopln ( d1.b == d2.b);

                d1.a = new A();
                d1.b = new B();

                Sopln( d1.a == d2.a);
                Sopln ( d1.b == d2.b);
        }
}
```

Project Requirement of object cloning

If we want to create second object with first object current modified state, we must use cloning. If we create objects with "*new keyword* and *constructor*" always objects are created with default initial state.

Q) *For example* if we want to create 100 instances of an object and these 100 instances want to be initialized with same state after performing many validations and calculations what is the correct approach in creating these 100 instances?

A) Develop cloning operation

Create one instance with new keyword and constructor and initialize it with the state that is generated after performing all validations and calculations. Then clone this object 99 times.

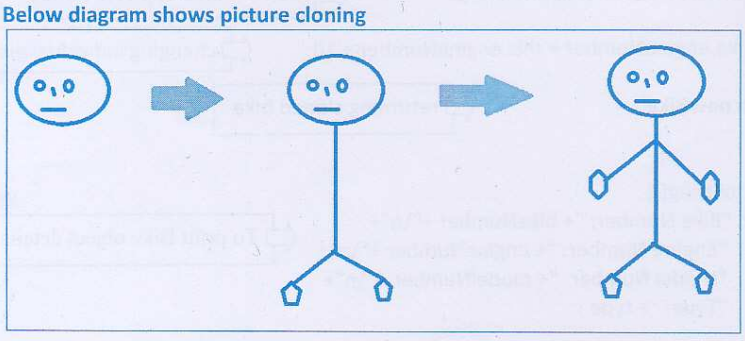
Then you have all 100 instances of this object with same state by executing validations and calculations only once. Hence you get high performance.

For example consider below real world example.

In the below diagram our target is creating one beautiful picture.

Step #1: First we create head part
Step #2: We create duplicate file and we do modifications in the duplicate file
Step #3: We follow the same procedure till we finish the complete picture.

Below diagram shows picture cloning



Another real world example - Bike

Consider a *bikes* manufacturing factory, here all bikes has same state except few properties like engine number, model number etc...

To develop this factory application it is recommended to use cloning approach, because all bike objects have same state. So once we create one bike object, we can clone that object and will change only the specific properties.

Below application shows performing cloning Bike object instances

```
//Bike.java
public class Bike implements Cloneable{

        private int engineNumber;     <= should be initialized by manufacturing company
        private int modelNumber;      <= should be initialized by manufacturing company
        private String type;          <= should be initialized by manufacturing company
        private int bikeNumber;       <= should be initialized after bike sales by RT office

        public Bike(int engineNumber, int modelNumber, String type){
                this.engineNumber   = engineNumber;
                this.modelNumber    = modelNumber;
                this.type           = type;
        }

        public void setBikeNumber(int bikeNumber){
                this.bikeNumber = bikeNumber;     <= RT office calls this method to set bike number.
        }

        //overriding clone method to develop above design
        public Bike  clone() throws CloneNotSupportedException{

                Bike newBike = (Bike)super.clone();          <= current bike object is cloned

                newBike.engineNumber = this.engineNumber + 10;     <= changing individual property

                return newBike;                              <= returning cloned bike
        }

        public String toString(){
                return "Bike Number: "+ bikeNumber +"\n"+
                       "Engine Number: "+ engineNumber +"\n"+
                       "Model Number: "+ modelNumber +"\n"+
                       "Type: "+ type ;
        }
}
```

<= To print Bike object details

```
//Factory.java
public class  Factory{
        public static void main(String[] args) throws CloneNotSupportedException{

                Byke b1 = new Byke(12345, 2012, "Pulsar 180CC");

                Bike b2 = (Bike) b1.clone();

                System.out.println(b1 == b2);


                b1.setBikeNumber(8192);
                b2.setBikeNumber(8193);

                System.out.println("b1 object details");
                System.out.println(b1);

                System.out.println();
                System.out.println("b2 object details");
                System.out.println(b2);
        }
}
```

First bike object

Cloning first bike object

clone() method creates new object, so == returns false

Setting bike number
This method is called in RT office

Operation# 6: Object finalization (understanding finalize() method)
Executing object's clean up code or releasing object's resources is called object finalization. Resource means internal objects of subclass object. *Unreferencing internal objects is called resource releasing logic*. We should override *finalize()* method to place this clean up code.

Its prototype is:

protected void finalize() throws Exception

What is the logic of finalize method in Object class?
It is defined as empty method in Object class, because object's resources releasing logic is specific to subclass. So subclass developer should override finalize method in subclass with that class object's resource releasing logic.

Below program explains overriding finalize method with required resource releasing logic.

```
//Example.java
class Building{
        Furniture f = new Furniture();

        Building (Furniture f)  {
                this.f = f;
        }
        void display(){
                System.out.println( f );
        }
        public void finalize(){
                //Furniture is the internal object of Building object.
                //Unreferencing this object is called resource releasing logic
                f = null;
        }
}
```

Q) When finalize method is executed?

Finalize method is automatically called and executed by garbage collector just before object is destroying. After finalize method execution is finished current object is destroyed if it is still unreferenced.

Q) Can we call finalize method, if so then current object is destroyed?

Yes we can call, but we should not call it explicitly because without destroying the current object its internal objects are unreferenced. Then it leads to NullPointerException when this internal object is used. *Below applications shows executing finalize method by gc:*

```
//FinalizeDemo.java
class FinalizeDemo{
        public static void main(String[] args) throws Exception{

                //creating unreferenced objects
                for (int i = 1; i <= 40; i++){
                        new Building ( new Furniture() );
                }

                //requesting GC thread to destroy all unreferenced objects
                System.gc();

                //it will halt main thread for 1000 milliseconds, mean while garbage collector
                destroys all unreferenced objects.
                Thread.sleep(1000);
        }
}
```

Q) Can an unreachable Java object become reachable again?
Yes. It can happen when the Java object's finalize() method is invoked and the Java object performs an operation which causes it to become accessible to reachable objects.

Check below program

```
class Example{
        int x;
        static Example e;

        Example(int x){
                this.x = x;
        }

         public void finalize(){
                System.out.println("In finalize");

                //converting unreferenced object as referenced object
                 e = this;

                //now GC cannot destroy this object.
        }

        public static void main(String[] args) throws Exception{
                Example e1 = new Example(10);

                //unreferencing object
                e1 = null;

                //requesting GC
                System.gc();

                //pausing main thread to allow GC to execute
                Thread.sleep(100);
        }
```

} Please read ***Garbage Collection*** **chapter** now again for complete interview questions on *finalize()* method

Q) Why clone() and finalize() methods are declared as protected?
A) To make sure they are only called on subclass object, rather than on Object class object. Because Object class does not has state to clone and does not has internal objects to unreference.

Q) Why wait(), notify(), and notifyAll() methods are given in Object class instead in Thread class? Check it in Multithreading chapter.

Chapter 26

Multithreading

with *JVM Architecture*

- In this chapter, You will learn
  - Multitasking and Multithreading
  - Sequential, Parallel, and Concurrent execution flow
  - Thread and StackFrame architecture
  - Creating custom threads
  - Interview questions in creating and executing custom threads
  - In how many ways can we create custom threads execution
  - Advantages of multithread programming model
  - Thread execution procedure/algorithms
  - Setting and getting thread name and priority
  - Types of threads - Non-daemon and daemon threads
  - Controlling thread execution using Thread class methods
  - Joining a thread execution to another thread execution
  - Synchronization, synchronized methods and blocks
  - Deadlock
  - Inter thread communication -> wait, notify & notifyAll
  - Inline thread
  - Grouping threads
- By the end of this chapter- you will be in a position to develop multithreading program with different cases.

i

Interview Questions

By the end of this chapter you answer all below interview questions

- Multitasking and types of multitasking.
- Sequential, parallel and concurrent execution flow
- Need of Multitasking
- Difference between multitasking and Multithreading
- Definition of Thread and multithreading?
- Overview of Threads - Main thread, garbage collector thread
- Thread Architecture, StackFrame Architecture
- Two ways to create custom threads in java
    a. Implementing java.lang.Runnable
    b. Extending java.lang.Thread
- *JVM Architecture* with multiple threads
- Multiple threads creation to execute same logic concurrently with different state
- Multiple threads creation to execute different logic concurrently
- Difference in single thread program execution and program execution with multiple threads.
- Thread Transition diagram explanation
- Threads execution process
    a. Thread Priority
    b. Thread Scheduling
- Creating different types of thread using setDaemon()
    a. Non-daemon
    b. Daemon
- Controlling Thread execution by using Thread class methods
    a. yield()
    b. sleep()
    c. join()
    d. suspend()
- Synchronization
    a. Importance of "synchronized" keyword
    b. Locks and monitor
    c. Synchronized methods
    d. Synchronized blocks
- Deadlocks
- Inter thread communication:
    a. wait(),
    b. notify()
    c. notifyAll()
- Thread Groups
- How can we create Thread without extending from Thread class or implementing from Runnable interface? What is inline thread?

So far you have learnt about a single thread. Lets us learn about the concept of multithreading and its development. Before understanding multithreading let us quickly review multitasking.

Multitasking and types of multitasking

Executing multiple tasks at a time is called multitasking.

Ex: while typing a program we can download a file, we can listen to music. It is called multi tasking.

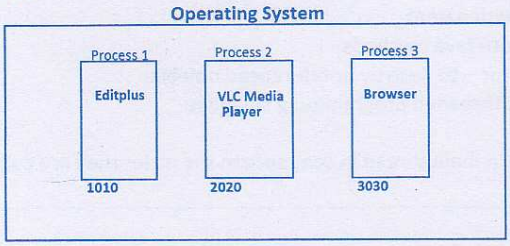
Multitasking is of two types

1. Process based multitasking
2. Thread based multitasking

Process based multitasking

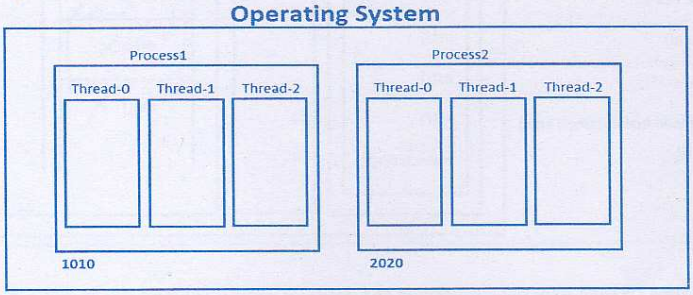
Executing multiple tasks *simultaneously* is called process based multitasking here each task is a separate independent process.

For example while typing a java program we can listen to a song and at the same time we can download a file from net, all these tasks are executing simultaneously and there is no relationship between these task. All these tasks have their own independent address space as shown below. This type of multitasking is developed at *OS level*.



Thread based multitasking

Executing multiple tasks *concurrently* is called thread based multitasking here each task is a separate independent part of a single process. That part is called *thread*. This type of multitasking is developed at *programmatic level*.



Advantage of multitasking
Either it is a process based or thread based multitasking the advantage of multitasking is to improve the performance of the system by decreasing the response time.

Note: In general, process based multitasking is called just multitasking and thread based multitasking is called multithreading.

The differences between multitasking and multithreading is
Multitasking is *heavy weight* because switching between contexts is slow because each process is stored at separate address as shown in Figure 1.

Multithreading is *light weight* because switching between contexts is fast because each thread is stored in same address as shown in Figure 2.

How can we create multiple threads in Java program?
We can develop multithread program very easily in Java, because Java provides in-build support for creating custom threads by providing API – Runnable, Thread, ThreadGroup.

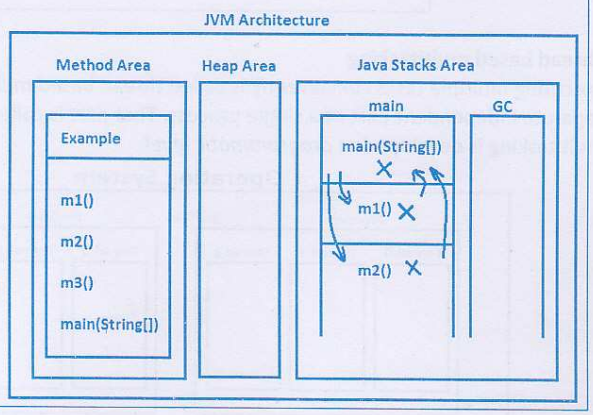
Overview on Java threads
We can consider JVM is also a process, when JVM is created in its Java stacks area by default two threads are created with names

1. main - to execute Java methods
2. garbage collector – to destroy unreferenced objects.

So, by default Java is multithreaded programming language

All methods are executed in main thread in sequence in the order they are called from main method as shown below

```
class Example
{
    static void m1()
    {
        Sopln("m1");
    }
    static void m2()
    {
        Sopln("m1");
    }
    static void m3()
    {
        Sopln("m1");
    }
    public static void main(String[] args)
    {
        m1();
        m2();
    }
}
```



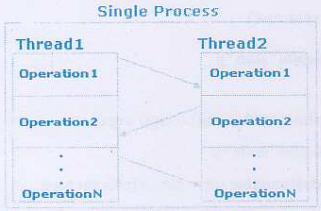
By using single thread JVM executes methods sequentially one after one.

Sequential execution vs **Concurrent execution**

Sequential execution – means single thread execution - takes *more time* to complete all methods execution. Whereas *concurrent execution* - means multithreading execution - takes *less time* to complete all methods execution. To have concurrent execution developer must create user defined thread in the program.

Meaning of concurrent execution

Executing multiple tasks in "start – suspend – resume – end" fashion is called concurrent execution. Means both tasks are started at different point of time and one task is paused while other task is executing. In Java, the Java Virtual Machine (JVM) allows an application to have multiple threads of execution running concurrently. When a program contains multiple threads then the CPU can switch between the two threads to execute them at the same time as shown in the below diagram.



Important point to be remembered at a single instance of time JVM cannot execute two tasks at a time.

As show in this diagram both tasks execution is started and are executed in *suspend – resume* fashion.

Definition of Thread

- A thread is an independent sequential flow of execution / path.
- A thread is a stack created in Java Stacks Area.
- It executes methods in sequence one after one.

Definition of Multithreading

It is the process of creating multiple threads in JSA for executing multiple tasks concurrently to finish their execution in short time by using Processer ideal time effectively.

When multithreading programming is suitable – means Need of multithreading?

To complete independent multiple tasks execution in short time we should develop multithreading. In multithreading based programming CPU Ideal time is utilized effectively.

How can we create user defined thread in JVM?

In JVM user defined thread is created and can start its execution - by creating *Thread class object* and by calling its method *start*.

How can we execute method logic in "*user defined*" thread?

We must call that logic method from ***run*** *method*.

Introduction to *run* method

1. It is the *initial point* of user defined thread execution.
2. It is actually defined in Runnable interface and is implemented in Thread class.
3. It is implemented as empty method in Thread class.
4. To execute our logic in user defined thread we must *override run* method.

Different ways to create custom threads in java

In Java we can create user defined threads in two ways

1. Implementing Runnable interface
2. Extending Thread class

In the both approaches we should override run() method in sub class with the logic that should be executed in user defined thread concurrently, and should call start() method on Thread class object to create thread of execution in Java Stacks Area.

Below diagram shows developing custom thread in above two ways

Developing custom thread extending from Thread

```
class MyThread extends Thread
{
    public void run()
    {
        Sopln("Run");
    }

    public static void main(String[] args)
    {
        Sopln("main");

        MyThread mt = new MyThread();
        mt.start();
    }

}
```

Developing custom thread implementing from Runnable

```
class MyRunnable implements Runnable
{
    public void run()
    {
        Sopln("Run");
    }

    public static void main(String[] args)
    {
        Sopln("main");

        MyRunnable mr = new MyRunnable()

        Thread th = new Thread(mr);
        th.start();
    }
}
```

In the first approach when we create subclass object, Thread class object is also created by using its no-arg constructor. Then when *start* method is called using subclass object custom thread is created in Java Stacks Area, and its execution is *started by executing run() method from subclass based on processor busy*.

In the second approach when we create subclass object Thread class object is not created, because it is not a subclass of Thread. So to call *start* method we should create Thread class object explicitly by using *Runnable parameter constructor,* then using this Thread class object we should call *start* method. Then custom thread is created in Java Stacks Area, and its execution *is started by executing run() method from Runnable interface subclass based on processor busy*.

Objects structure for above two programs

Thread class constructor

Below constructors are used when thread is created by extending from Thread class	Below two constructors are used when thread is created implementing from Runnable interface
Thread() - Creates thread with default name Thread-<index> - It is called using super(); - it executes run method from current object of start() method	Thread(Runnable target) - Creates thread with default name Thread-<index> - Thread class object is created explicitly using this constructor - it executes run method from the passed argument Runnable object
Thread(String name) Creates thread with given name Called it via super(name)	Thread(Runanble target, String name) Creates thread with given name

Thread class important methods

1. **public synchronized void start()**
   Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread. The result is that two threads are running concurrently: the current thread (which returns from the call to the start method) and the other thread (which executes its run method).

   Rule: It is never legal to call start method more than once on a same thread object. In practical, a thread may not be restarted once it has completed execution. It leads to exception *"java.lang.IllegalThreadStateException"*.

2. **public void run()**
   It is the initial point of custom thread execution.

3. **public static native void *sleep*(long millis) throws InterruptedException**
   public static native void *sleep*(long millis, int nanos) throws InterruptedException
   Causes the currently executing thread to sleep (cease execution) for the specified number of milliseconds plus the specified number of nanoseconds.

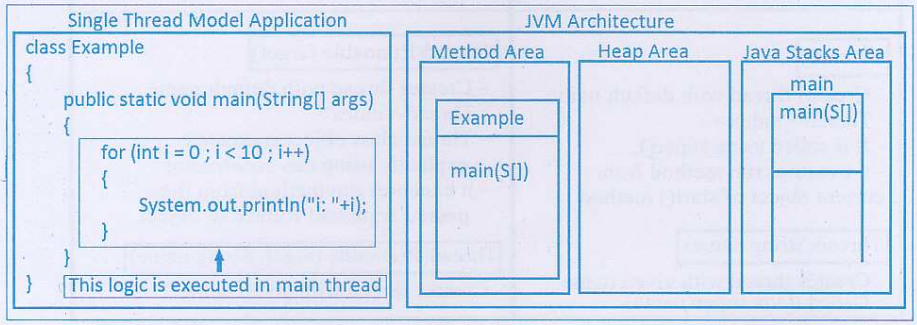
   Rule #1: The value of millis should not be negative or the value of nanos is should be in the range 0-999999 else it leads to exception "java.lang.IllegalArgumentException".

   Rule #2: InterruptedException is a checked exception so the sleep method caller should handle or report this exception else it leads to CE: unreported exception must be caught or declared to be thrown

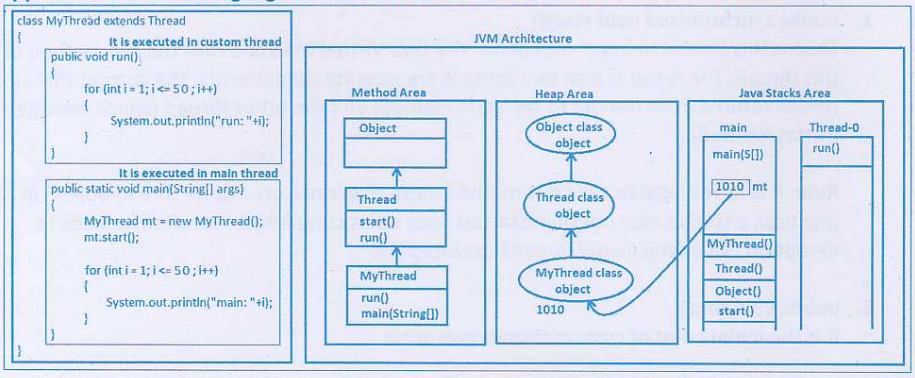
   Remaining methods are explained in the respective programs

Programs

Application #1: Executing logic using single thread in *main* thread



Application #2: Executing logic with two threads - *main thread* and *custom thread*



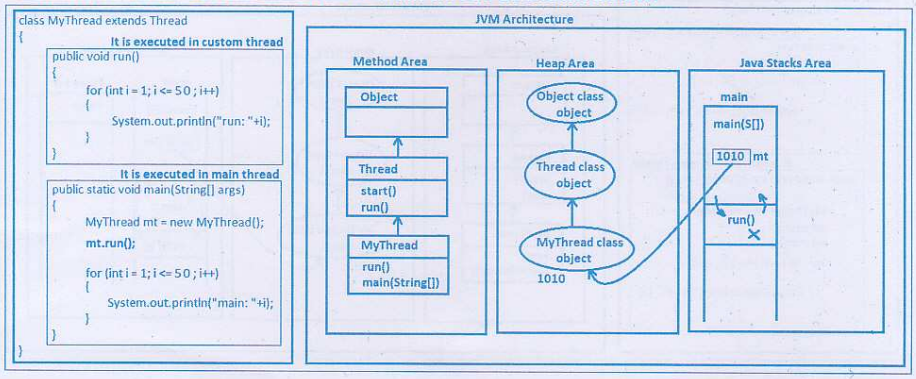
Output: "*main method for loop*" and "*run method logic*" are executed concurrently.

Point to be remembered like other methods *start method call* does not pause main method execution. By calling start method we just requesting JVM for user defined thread creation in Java Stacks Area and start its execution by calling run method from thread object on which start method is called. Then JVM starts its execution based on processor busy.

Answer below questions

Q) Can we call run method directly from main method?

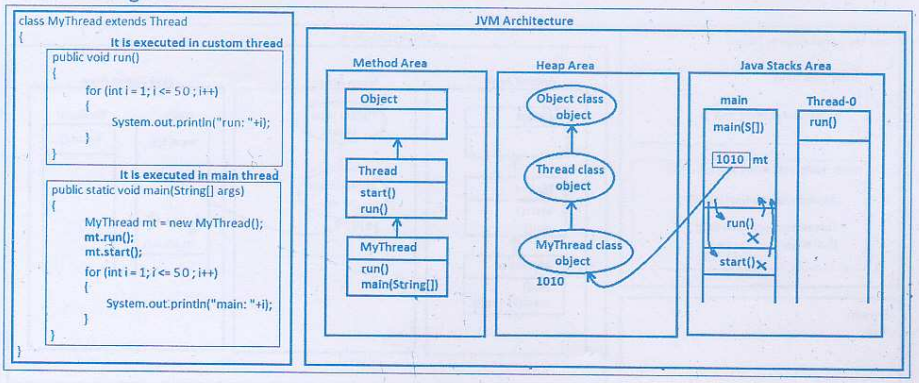
Yes, we can call *run* method directly. If we call it directly, user defined thread in Java Stacks Area is not be created. It is executed in *main thread* as shown below.



Output: *main* and *run* methods are executed in same *main thread* sequentially. Since run method is called before main method *for loop*, first *run method for loop* output is printed then later *main method for loop* output is printed.

Q) What is the output in the above program if we call both *run* and *start* method?

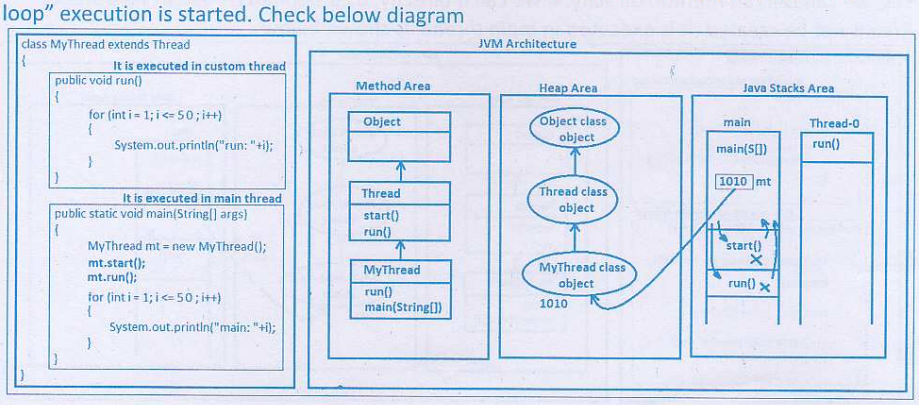
run method logic is executed two times. It is executed in *main thread* since it is called directly from main method, also it is executed in *custom thread* due to *start* method call, as shown in the below diagram.



Output:

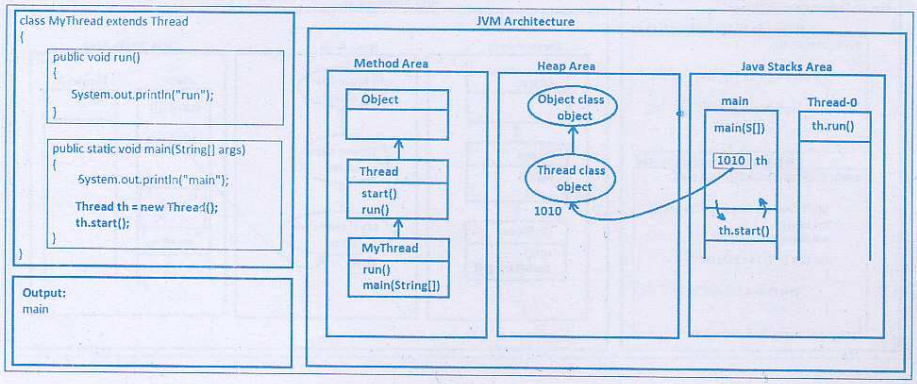
First *run method* in main thread completes its execution, then due to start method call user defined thread is created and run method is called again. Now "*run method* in *Thread-0*" and "*main method* for loop in *main thread*" are executed concurrently.

Q) What is the output from the above program if we call start() before run() method?
Output: run methods logic is executed concurrently, one is from Thread-0 and another is from main thread. After completion of run method execution in main thread main method "for loop" execution is started. Check below diagram

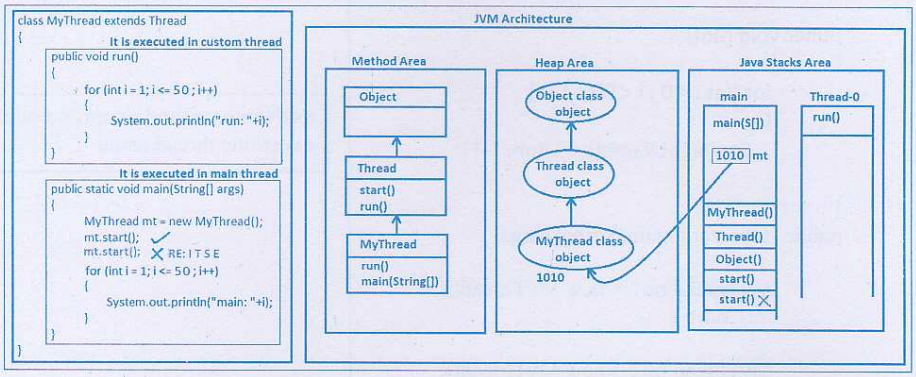


What is the output in the below case?
Q) If start method is called on Thread class object directly, is subclass run() method executed?
No, run method is executed from Thread class.

The basic point to be remembered is run method is executed from the current thread object of start method. So in this case Thread object is the current object, so run method is executed from Thread class. Check below diagram



Q) Can we call *start* method more than once on the same thread object?
No, it leads to exception *java.lang.IllegalThreadStateException*. The current thread – the thread that calls start method – execution is terminated abnormally. Check below program



Output: main method execution is terminated means for loop will not be executed. But run method execution in Thread-0 will be continued. Means even though *main thread* execution is terminated *custom thread* execution is continued.

Then how can we create multiple user defined threads in Java Stacks Area?
There are two ways to created more than one user defined threads in Java Stacks Area

1. Create *multiple thread subclass objects* and call start method on each thread object.
2. Create *multiple subclasses from Thread class,* create its object and call start method.

In first approach, all threads execute same run method logic, because all its thread objects are created from same class. This approach is recommended only to execute *same logic concurrently with different object state,* check below application

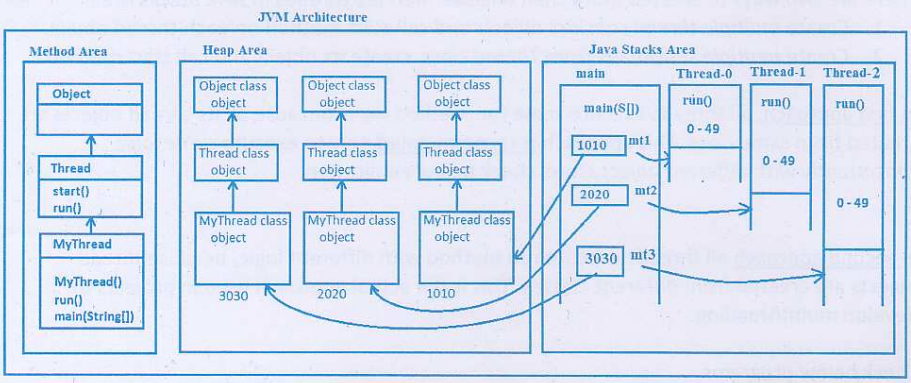
In second approach all threads execute run method with different logic, because thread objects are created from different classes. This is the actual approach used in projects to develop multithreading.

Check below programs.

Application #3: This application shows creating multiple user defined threads to execute same logic from all threads.

```
class MyThread extends Thread
{
        public void run()
        {
                for (int i = 0 ; i < 50 ;  i ++)
                {
                   SopIn(getName() + " Run: "+i );
                }
        }
        public static void main(String[] args)
        {
                MyThread mt1 = new MyThread();
                mt1.start();

                MyThread mt2 = new MyThread();
                mt2.start();

                MyThread mt3 = new MyThread();
                mt3.start();
        }
}
```

getName method returns currently executing thread name.



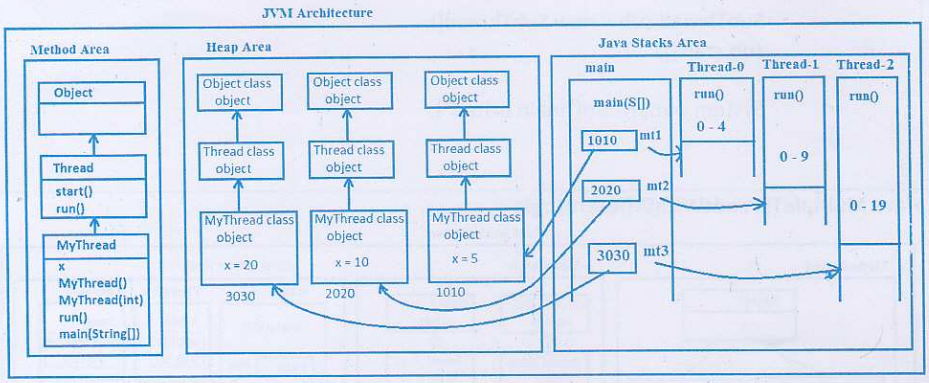
As you noticed, three *thread objects* are created in heap area and for each thread object separate *thread of executions* are created in Java Stacks Area.

Same for loop is executed from all threads with 50 iterations.

Application #4: This application shows creating multiple user defined threads to execute same logic concurrently with different state.

```
class MyThread extends Thread
{
        int x;

        MyThread(){
                x = 5;
        }

        MyThread(int x){
                this.x = x;
        }

        public void run()
        {
                for (int i = 0 ; i < this.x ;  i ++)
                {
                  Sopln(getName() + " Run: "+i );
                }
        }
}
```

```
class MultipleThreadsWithSameLogic
{
    public static void main(String[] args)
    {
        MyThread mt1 = new MyThread();
        mt1.start();

        MyThread mt2 = new MyThread(10);
        mt2.start();

        MyThread mt3 = new MyThread(20);
        mt3.start();


        for (int i = 0 ; i < 20;  i ++)
        {
                Sopln("main: "+i);
        }
    }
}
```



As you noticed, from all three threads same for loop is executing but with different **x** value.

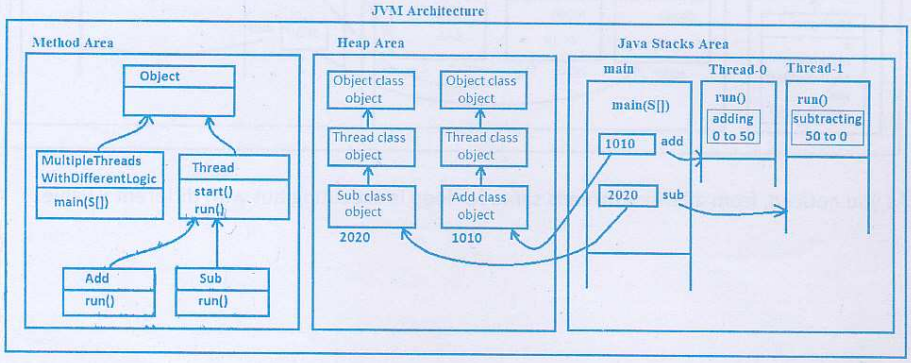
Application #5: This application shows creating multiple user defined threads to execute different logic. This program shows printing summation of numbers "0to50" and subtraction of numbers "50to0" concurrently.

```
//AddThread.java
class AddThread extends Thread {
    int sum = 0;
    public void run()    {
        for (int i = 0 ; i <= 50 ; i++)
        {
            sum += i;
            Sopln("The Summation: "+sum);
        }
    }
}
```

```
//SubThread.java
class SubThread extends Thread {
    int diff  = 0;
    public void run()    {
        for (int i = 50 ; i >= 0 ; i--)
        {
            diff  -= i;
            Sopln("The Substraction: "+diff);
        }
    }
}
```

```
//MultipleThreadsWithDifferentLogic.java
class MultipleThreadsWithDifferentLogic{
        public static void main(String[] args)  {
                System.out.println("main started");

                AddThread add = new AddThread();
                add.start();

                SubThread sub = new SubThread();
                sub.start();

                System.out.println("main exited");
        }
}
```

>java MultipleThreadsWithDifferentLogic



Q) Can we override start() method?
Yes we can override as it is a non-final method. Below is the valid code No CE, and No RE.

```
class MyThread extends Thread{
        public void run(){
                System.out.println("run");
        }
        public void start(){
                System.out.println("start");
        }
}
```

Q) If we override start() method is custom thread created?
No, custom thread is not created.

Q) Then why Thread class developer not declared start() method as final?
It is project requirement: before starting this thread execution if we want do some validations and calculations to update current custom thread object state, we should override start method in subclass with this validation logic and then we should start custom thread.

This is the reason Thread class developer leaving start as non-final method

Q) How can we start custom thread from overriding method?
We must place *super.start()* at end of overriding start() method.

Given:

```
class MyThread extends Thread{
        public void run(){
                System.out.println("run");
        }
        public void start(){
                System.out.println("start");
        }
        public static void main(String[] args){
                MyThread mt = new MyThread();
                mt.start();
                System.out.println("main");
        }
}
```

Q1) From above program is custom thread created and what is the output?
A) Custom thread is not created. So run() is not executed.
Output is:

```
start
main
```

Q2) If we call run() from overriding start(), in which thread run() is executed and what is the output?

add below start() method in above code

```
public void start(){
        System.out.println("start");
        run();
}
```

A) custom thread is not created, run() is executed in main thread as start() method is executed in main thread.

Output:

```
start
run
main
```

Q3) If we call super.start() in the above updated program, is custom thread created, where run() method is executed, how many times?

add below start method in above code

```
public void start(){
        System.out.println("start");
        run();
        super.start();
}
```

A)

Yes custom is created with name Thread-0.

run() is executed 2 times

1. in main thread because we call it explicity from overriding start method
2. in custom thread because it is implicitly called by JVM becuase of super.start() call

Output:

```
start
run
main
run
```

Q4) When should we override start() method in projects as it is given as non-final method?

A) If we want do some validation and calculations to update current custom thread object state before starting this thread execution, we should override start method in subclass with this validation logic and at end of this overriding start() method we must place super.start() to create custom thread.

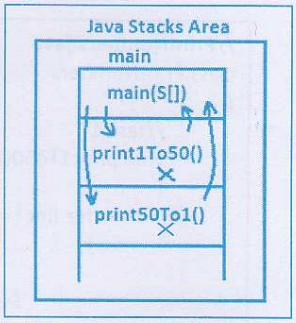
Write a program to show time difference between single and multiple threads execution

Write a Java program with two methods to print numbers "1 to 50" and "50 to 1" in sequence. This application uses Thread.sleep(100) method to delay printing every number for 100 milliseconds.

```
//PrintNumbers.java
class PrintNumbers
{
        //task 1
        void print1To50()
        {
                for (int i = 1; i <= 50 ; i++)
                {

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

                        try{ Thread.sleep(100); }
                        catch(InterruptedException ie){ ie.printStackTrace(); }

                }
        }

        //task 2
        void print50To1()
        {
                for (int i = 50; i >= 1 ; i--)
                {

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

                        try{ Thread.sleep(100); }
                        catch(InterruptedException ie){ ie.printStackTrace(); }

                }
        }
}
```

Application #6: This application shows executing above two tasks sequentially using main thread. Also it prints the time taken to complete both tasks

```
//SingleThreadModelApplication.java

class SingleThreadModelApplication
{
    public static void main(String[] args)
    {
        PrintNumbers pn = new PrintNumbers();

        long time1 = System.currentTimeMillis();

            pn.print1To50();

        System.out.println();

            pn.print50To1();

        long time2 = System.currentTimeMillis();

        SopIn("Time taken to complete both tasks: "+((time2- time1) / 1000) +" secs");
    }
}
```

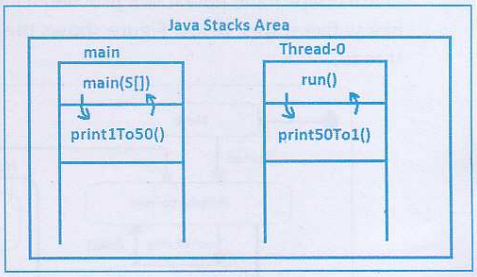


Output: Sequential execution – takes 10 secs to complete two tasks

```
C:\Program Files\EditPlus 2\launcher.exe
1       2       3       4       5       6       7       8       9       10
11      12      13      14      15      16      17      18      19      20
21      22      23      24      25      26      27      28      29      30
31      32      33      34      35      36      37      38      39      40
41      42      43      44      45      46      47      48      49      50

50      49      48      47      46      45      44      43      42      41
40      39      38      37      36      35      34      33      32      31
30      29      28      27      26      25      24      23      22      21
20      19      18      17      16      15      14      13      12      11
10      9       8       7       6       5       4       3       2       1
Time taken to complete both tasks: 10 secs
```

Application #7: This application shows executing above tasks concurrently. Also it shows multithreading program takes less time than single thread model application.

```
//MultiThreadModelApplication.java
class MultiThreadModelApplication extends Thread
{
        static PrintNumbers pn = new PrintNumbers();

        public void run()
        {
                pn.print50To1();
        }

        public static void main(String[] args)
        {
                MultiThreadModelApplication mt = new MultiThreadModelApplication();

                long time1 = System.currentTimeMillis();
                mt.start();

                        pn.print1To50();

                long time2 = System.currentTimeMillis();

                SopIn("Time taken to complete both tasks: "+((time2- time1) / 1000) +" secs");
        }
}
```



Output: Concurrent execution – takes 5 secs to complete the same two tasks

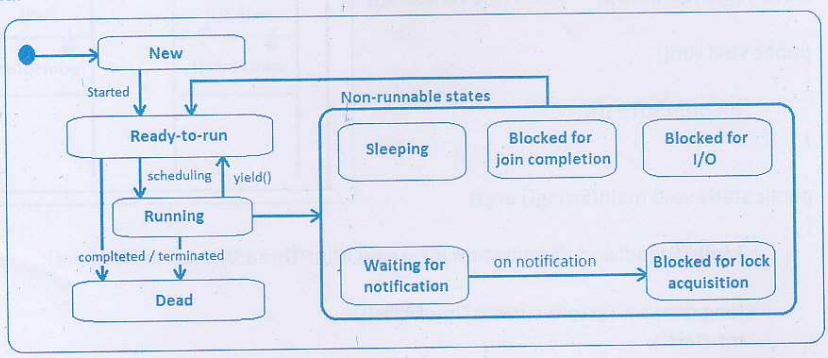
```
C:\Program Files\EditPlus 2\launcher.exe
1       50      49      2       48      3       4       47      5       46
6       45      44      7       8       43      42      9       41      10
40      11      39      12      13      38      37      14      36      15
35      16      34      17      18      33      32      19      31      20
21      30      29      22      23      28      27      24      25      26
25      26      24      27      23      28      29      22      21      30
31      20      32      19      18      33      34      17      16      35
15      36      14      37      38      13      39      12      11      40
10      41      9       42      8       43      44      7       6       45
5       46      4       47      3       48      2       49      1       50
Time taken to complete both tasks: 5 secs
```

Note: Run this application more than once, you will observe the output printed will be different in every turn. Because Multithreading program execution is depends on processor.

So, we can conclude that multithreading program output we cannot guarantee.

Threads transition diagram

Understanding the life cycle of a thread is valuable when programming with threads. Threads can exist in different states. Just because of a thread's start() method has been called, it does not mean that the thread has access to the CPU and can start executing straight away. Several factors determine how it will proceed. Once thread is created it is available in any one the below five states. Below Figure shows the states and the transitions in the life cycle of a thread.



- **New state**
  A thread has been created, but it has not yet started. A thread is started by calling its start() method.
- **Ready-to-run state**
  This start is also called Runnable state, also called Queue. A thread starts life in the Ready-to-run state by calling start method and waits for its turn. The thread scheduler decides which thread runs and for how long.
- **Running state**
  If a thread is in the Running state, it means that the thread is currently executing
- **Dead state**
  Once in this state, the thread cannot ever run again.

- **Non-runnable states**
  A running thread can transit to one of the non-runnable states, depending on the circumstances. A thread remains in a non-runnable state until a special transition occurs. A thread does not go directly to the Running state from a non-runnable state, but transits first to the Ready-to-run state.

  The non-runnable states can be characterized as follows:
  - *Sleeping*: The thread sleeps for a specified amount of time.
  - *Blocked for I/O*: The thread waits for a blocking operation to complete
  - *Blocked for join completion*: The thread awaits completion of another thread
  - *Waiting for notification*: The thread awaits notification from another thread
  - *Blocked for lock acquisition*: The thread waits to acquire the lock of an object

Threads execution procedure -> How threads are executed in JVM?
JVM executes Threads based on their *priority* and *scheduling*.

Thread Scheduler
Schedulers in JVM implementations usually employ one of the two following strategies:

- Preemptive scheduling.
  If a thread with a higher priority than the current running thread moves to the Ready-to-run state, the current running thread can be preempted (moved to the Ready-to-run state) to let the higher priority thread execute.

- Time-Sliced or Round-Robin scheduling.
  A running thread is allowed to execute for a fixed length of time, after which it moves to the Ready-to-run state to await its turn to run again.

Thread schedulers are implementation and platform dependent; therefore, how threads will be scheduled is unpredictable.

Thread Priority
Every thread created in JVM is assigned with a priority. The priority range is between 1 and 10.

- 1 is called minimum priority
- 5 is called normal priority
- 10 is called maximum priority.

In Thread class below three variables are defined to represent above three values.

- public static final int **MIN_PRIORITY**;
- public static final int **NORM_PRIORITY**;
- public static final int **MAX_PRIORITY**;

Threads are assigned priorities, based on that the thread scheduler can use to determine how the threads will be scheduled. The thread scheduler can use thread priorities to determine which thread gets to run. The thread scheduler favors giving CPU time to the thread with the highest priority in the Ready-to-run state. This is not necessarily the thread that has been the longest time in the Ready-to-run state.

A thread inherits the priority from its parent thread. The default priority of every thread is normal priority **5**, because *main thread* priority is 5.

The priority of a thread can be set using the setPriority() method and read using the getPriority() method, both of which are defined in the Thread class with the below prototype.

```
public final void setPriority(int newPriority)
public final int getPriority()
```

Rule: newPriority value range should between 1 to 10, else it leads to exception *java.lang.IllegalArgumentException*

Thread Name

User defined thread is created with the default name *"Thread-"+<index>*, where index is the integer number starts with 0. So the first user defined thread name will be Thread-0, second thread name is Thread-1, etc...

The name of a thread can be set using the setName() method and read using the getName() method, both of which are defined in the Thread class with the below prototype.

```
public final void setName(String name)
public final String getName()
```

So the default thread name can be changed by using either

- at time of thread object creation using String parameterized constructor or
- after object creation using above set method

Application #8:

This application shows creating custom thread with user defined name and priority.

```
// ThreadNameAndPriority.java
class MyThread extends Thread
{
        MyThread()
        {
                super();
        }

        MyThread(String name)
        {
                super(name);
        }
        public void run()
        {
                for (int i = 0; i < 10 ; i++)
                {
                        System.out.println(getName() + " i: "+i);
                }
        }
}
```

```
class ThreadNameAndPriority
{
        public static void main(String[] args)
        {
                MyThread mt1 = new MyThread();
                MyThread mt2 = new MyThread("child2");

                System.out.println("mt1 Thread's initial name and priority");
                System.out.println("mt1 name: "+mt1.getName());
                System.out.println("mt1 priority: "+mt1.getPriority());

                System.out.println();

                System.out.println("mt2 Thread's initial name and priority");
                System.out.println("mt2 name: "+mt2.getName());
                System.out.println("mt2 priority: "+mt2.getPriority());

                mt1.setName("child1");

                mt1.setPriority(6);
                mt2.setPriority(9);

                System.out.println("mt1 Thread's changed name and priority");
                System.out.println("mt1 name: "+mt1.getName());
                System.out.println("mt1 priority: "+mt1.getPriority());

                System.out.println();

                System.out.println("mt2 Thread's changed name and priority");
                System.out.println("mt2 name: "+mt2.getName());
                System.out.println("mt2 priority: "+mt2.getPriority());

                mt1.start();
                mt2.start();

                for (int i = 0; i < 10 ; i++)
                {
                        System.out.println("main i :"+i);
                }
        }
}
```

Retrieving currently executing thread object reference

Below method is used to retrieve reference of the currently executing thread object.

public static native Thread currentThread()

This method is useful to perform some operations on a thread object when its reference is not stored in our logic.

Application #9: This application shows changing main thread name and priority. Also this application shows static block is executed in main thread.

```
// CurrentThreadDemo.java
class CurrentThreadDemo
{
        static
        {
                System.out.println("In SB");

                //retrieving currently executing thread reference
                Thread th = Thread.currentThread();
                System.out.println("SB is  exeucting in \""+ th.getName() + "\" thread\n");
        }

        public static void main(String[] args)
        {
                System.out.println("\nIn main method");

                //retrieving currently executing thread reference
                Thread th = Thread.currentThread();

                System.out.println("Original name and priority of main thread");
                System.out.println("current thread name: "+th.getName());
                System.out.println("current thread priority: "+th.getPriority());

                th.setName("xxyy");
                th.setPriority(7);

                System.out.println("\nmodified name and priority of main thread");
                System.out.println("current thread name: "+th.getName());
                System.out.println("current thread priority: "+th.getPriority());
        }
}
```

ThreadGroup

Every thread is created with a thread group. The default thread group name is "***main***". We can also create user defined thread groups using *ThreadGroup* class.

In Thread class we have below method to retrieve current thread's ThreadGroup object reference

public final ThreadGroup getThreadGroup()

In ThreadGroup class we have below method to retrieve the ThreadGroup name

public final String getName()

So in the program we must write below statement to get thread's ThreadGroup name

String groupName = th.getThreadGroup().getName();

For more details on creating ThreadGroup object and placing threads in user defined thread groups check *Application #16*.

---

toString() method in Thread class

In Thread class toString() method is overridden to return thread object information as like below: *Thread[thread name, thread priority, thread group name]*

So its logic in Thread class should be as like below

```
public String toString(){
        return "Thread["+getName() +"," +getPriority() +","+getThreadGroup().getName() +"]";
}
```

Application #10: This application shows Thread class's toString() method functionality.

```
// ToStringDemo.java
class ToStringDemo{
    public static void main(String[] args){
            Thread th1 = new Thread();
            System.out.println(th1);

            Thread th2 = new Thread("child1");
            System.out.println(th2);

            Thread th3 = Thread.currentThread();
            System.out.println(th3);

            th3.setPriority(7);

            Thread th4 = new Thread();
            System.out.println(th4);
        }
}
```

Types of threads:

Java allows us to create two types of threads they are

1. Non-Daemon threads
2. Daemon threads

- A thread that executes main logic of the project is called non-daemon thread.
- A thread that is running in background to provide services to non-daemon threads is called daemon thread. So we can say *daemon threads* are *service threads*.

Since daemon threads are service threads, its execution is terminated if all non-daemon threads execution is completed.

Every user defined thread is created as non-daemon thread by default, because main thread is a non-daemon thread and daemon property is also inherited from parent thread.

Garbage collector is a daemon thread. Since garbage collector provides service - destroying unreferenced objects – it is created as daemon thread. It is a low priority thread so we cannot guarantee its execution, and also it is terminated if all non-daemon threads execution is completed.

Daemon thread creation

To create user defined thread as daemon thread, Thread class has below method

public final void setDaemon(boolean on);

if **on** value is true – thread is created as daemon

else it is created as non-daemon thread. So the daemon property default value is false.

To check thread is daemon or non-daemon, Thread class has below method

public final boolean isDaemon()

returns *true* if thread is *daemon*, else returns false.

Rule: setDaemon method cannot be called after start() method call, it leads to RE: *java.lang.IllegalThreadStateException*. because once thread is created as non-daemon thread, it cannot be converted as daemon. Check below programs

```
class MyThread extends Thread
{
    public static void main(String[] args)
    {
        MyThread mt = new MyThread();

        mt.setDaemon(true);   ✓
        mt.start();
    }
}
```

```
class MyThread extends Thread
{
    public static void main(String[] args)
    {
        MyThread mt = new MyThread();

        mt.start();
        mt.setDaemon(true);   ✗ RE: I T S E
    }
}
```

Application #11: This application shows creating threads as daemon threads and their execution. In this application threads are created by implementing Runnable interface.

```
// DaemonDemo.java
class  DaemonDemo implements Runnable
{
        Thread th;

        DaemonDemo()
        {
                th = new Thread(this);

                th.setDaemon(true);
                th.start();

                //th.setDaemon(true);
        }

        public void run()
        {
                System.out.println("Run: "+th.isDaemon());

                for (int i = 1; i <= 100 ; i++)
                {
                        System.out.println("Run: "+i);

                }
        }

        public static void main(String[] args)
        {
                DaemonDemo dd1 = new DaemonDemo();

                System.out.println("Baba countdown starts");

                for (int i = 1; i <= 5 ; i++)
                {
                        System.out.println("main: "+i);
                }
        }
}
```

Output:
Daemon thread execution is terminated in middle, all iterations output will not be printed.

Controlling Threads execution

Threads execution can be controlled in three ways

1. Pausing thread execution for a given period of time using *sleep* method
2. Pausing thread execution until other thread execution is completed using *join* method
3. Executing threads sequentially using synchronized keyword – synchronization concept.

Joining a thread execution to other thread

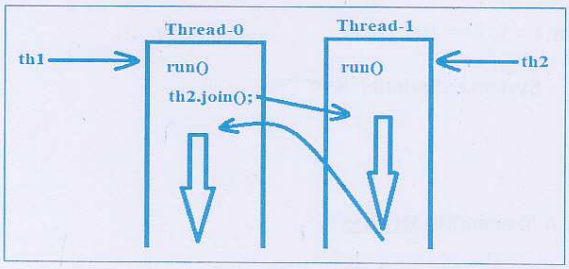
Pausing thread execution until other thread execution is completed is called *thread join*.
Thread class has below overloaded join methods to perform this task.

```
public final void join() throws InterruptedException
public final synchronized void join(long millis) throws InterruptedException
public final synchronized void join(long millis, int nanos) throws InterruptedException
```

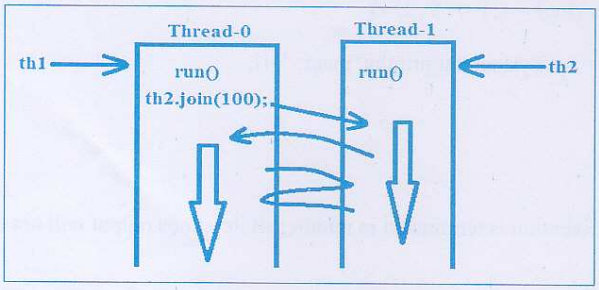
Difference between *no-arg join* and *parameterized join* methods

No-arg join method pauses thread execution until completion of other thread execution. If other thread execution is blocked forever, this thread execution is also blocked forever. Check below diagram,

As per below diagram **Thread-0** is blocked until **Thread-1** execution is completed fully.



Whereas parameterized join method does not block thread execution until completion of other thread execution. Its execution is resumed after completion of given time.



Difference between join(long) and sleep(long)?

sleep(long) method pauses thread execution *independent of other threads execution* for the given period of time completely. It does not allow thread to run until the given time is completed.

join(long) method pauses thread execution dependent on other thread's execution for the given period of time. It pauses thread execution only for the given period of time or if that other thread execution is completed before the given time, current thread execution is resumed immediately even through given paused time is not finished .

Application #12: This application shows joining a thread execution with another thread execution using join() method.

```
// JoinDemo.java
public class JoinDemo extends Thread{
        public void run(){

                for (int i = 0; i < 20 ; i++){
                        System.out.println(getName() +" : "+i);

                        if(i == 5 && getName().equals("Child2")){
                                try {    Thread.sleep(500);    }
                                catch(Exception e){    e.printStackTrace();   }
                        }
                }
        }
        public static void main(String[] args) throws InterruptedException{
                System.out.println("main is started");

                JoinDemo jd1 = new JoinDemo();
                jd1.setName("Child1");
                jd1.start();

                JoinDemo jd2 = new JoinDemo();
                jd2.setName("Child2");
                jd2.start();

                jd1.join();
                jd2.join();

                System.out.println("main is exited");
        }
}
```

Output: Always main thread is the last terminated thread. Means the output "*main is exited*" always will be printed after two threads output.

Synchronization

The process of allowing multiple threads to modify an object in sequence is called synchronization. We can allow multiple threads modifying the object sequentially only by executing that object's mutator methods logic in sequence from multiple threads. This is possible by using object locking concept.

In Java Synchronization is implemented by using *synchronized* keyword.
Synchronized keyword is applied to methods and local blocks.
So synchronization is developed by using

1. Synchronized methods
2. Synchronized blocks

Below diagram shows defining synchronized method and block

```
public class Bank
{
    private double balance;

    public synchronized void withdraw(int amt)
    {
        System.out.println("balance before withdraw: "+ balance);
        balance = balance - amt;
        System.out.println("balance after withdraw: "+ balance);
    }
}
```

```
public class Bank
{
    private dobule balance;

    public void withdraw(int amt)
    {
        System.out.println("balance before withdraw: "+ balance);
        synchronized (this)
        {
            balance = balance - amt;
        }
        System.out.println("balance after withdraw: "+ balance);
    }
}
```

Synchronization process:

Q) What does happen when we declare method as synchronized? Or
Q) When JVM lock the object?

When we call synchronized method the current object of this method is locked by this current thread. So that other threads cannot use this locked object for executing either the same synchronized method or other synchronized methods. But it is possible to access non-synchronized methods by using the locked object, because to execute non-synchronized methods object is no need to be locked by this thread. The thread that locks the object is called *monitor*. This current object is unlocked only after completion of that synchronized method execution either normally or abnormally.

What is the difference between synchronized methods and blocks?

Difference #1:

- If method is declared as synchronized that method's complete logic is executed in sequence from multiple threads by using same object.
- If we declare block as synchronized, only the statements written inside that block are executed sequentially not complete method logic.

Difference #2:

- Using Synchronized method we can only *lock* current object of the method
- Using Synchronized block we can *lock* either current object *or* argument object of the method. We must pass the object's referenced variable to synchronized block, as shown in the below program.

Locking current object

```
class Example{
        void m1(){
                synchronized(this){ }
        }
}
```

Locking argument object

```
class Example{
        void m1(Sample s){
                synchronized(s){ }
        }
}
```

Q) How many synchronized blocks can we develop in a method?
A) We can develop more than one synchronized block

Q) When should we develop multiple synchronized blocks instead of declaring complete method as synchronized?
A) In below two cases we develop multiple synchronized blocks

1. For executing parts of method logic sequentially for doing object modification
2. For executing one of part of method logic by locking current object and other part of method by locking argument object.

Below example shows developing multiple synchronized blocks

```
class Example{
        void m1(Sample s){
                synchronized(s){ }
                synchronized(this){ }
        }
}
```

Q) What is the difference in declaring NSM and SM as synchronized?

- If we declare NSM as synchronized its current object is locked, so that in this method NSVs of this object are modified sequentially by multiple threads.
- If we declare SM as synchronized its class's java.lang.Class object is locked, so that in this method SVs are modified sequentially by multiple threads.

Focus: in a static method by using synchronized block we can only lock argument object but not current object as this keyword can exist in static method.

Need of synchronization or when should we develop synchronization?
We must develop synchronization to get correct results in modifying object's data from multiple threads. We get the data corruption problem only when multiple threads accessing same object. If different object is using by multiple threads then no need to implement synchronization.

Application #12: Below application shows developing synchronization by using synchronized method. This application shows data corruption problem and solution with synchronization.

```
//Add.java
class Add{
        int x, y;

        void add(int x, int y){
        //synchronized void add(int x, int y){
                this.x = x;   this.y = y;

                try{ Thread.sleep(1000); }
                catch(Exception e){e.printStackTrace();}

                int res = this.x + this.y;

                System.out.println("In "+ Thread.currentThread().getName() +" Result: "+ res);
        }
}
```

Execute add method with and without synchronized keyword to find output difference.

```
//Thread1.java
class Thread1 extends Thread {
        Add a;
        Thread1(Add a){ this.a = a; }

        public void run(){
                a.add(50, 60);
        }
}
```

```
//Thread2.java
class Thread2 extends Thread {
        Add a;
        Thread2(Add a){ this.a = a; }

        public void run(){
                a.add(70, 80);
        }
}
```

```
//Test.java
class Test {
        public static void main(String[] args) {
                Add a = new Add();

                new Thread1(a).start();
                new Thread2(a).start();
        }
}
```

What is the Output for the below cases:

Case #1: add method *is not* synchronized

Case #2: add method is synchronized

Note: Local variables are not sharable by multiple threads. So a thread cannot change other thread's local variables values, hence we get desired result.

Case #3: do not declare add method as synchronized and in add method use parameters in addition operation.

Another example on synchronization

```
//PrintMessage.java
class PrintMessage{
        //void printMsg(String msg)
        synchronized void printMsg(String msg)      {
                System.out.print("["+msg);

                try {  Thread.sleep(1000); }
                catch(Exception e) { e. printStackTrace(); }

                System.out.println("]");
        }
}

//MessagePrinterThread.java
class MessagePrinterThread implements Runnable{
        String msg;
        PrintMessage pm;
        Thread th;

        public MessagePrinterThread(PrintMessage pm, String msg)  {
                this.pm         = pm;
                this.msg        = msg;

                th              = new Thread(this);
                th.start();
        }
        public void run() {
                pm.printMsg (msg);
        }
}

//MessagePrinterThreadUser.java
public class MessagePrinterThreadUser{
        public static void main(String args[]) {
                PrintMessage pm      = new PrintMessage();

                MessagePrinterThread  ob1   = new MessagePrinterThread(pm,"Abc");
                MessagePrinterThread  ob2   = new MessagePrinterThread(pm,"Bbc");
                MessagePrinterThread  ob3   = new MessagePrinterThread(pm,"Cbc");

        }
}
```

Output:
printMsg method is not synchronized

printMsg method is synchronized

Real time application for Synchronization:
In Bank application we must declare deposit and withdraw operation methods as synchronized. Otherwise balance is modified concurrently by multiple threads (persons). In this case either Bank or customer loses money due to concurrent modification on balance variable.

For example assume withdraw method is not declared as synchronized, so multiple threads (persons) withdrawing money at a time. In this case other thread is allowed to withdraw money before updating balance amount. Finally Bank loses money.

Check below diagram:

Coding design in Servlet and JSP program development

Servlet and JSP objects are not thread safe objects, so in Servlet and JSP program we must avoid declaring instance variables if they are meant for updating in different requests. In this case they must be created as local variables in service method to avoid data corruption.

The architecture for executing the Servlet is: "*single instance - multiple threads*"

Means for every new client (browser) ServletContainer creates new thread and process that client request with same servlet instance.

So if we create instance variables in servlet the modifications done in servlet object in one client request are affected to another client request and that client get wrong results as response.

Solution: Create those variables as local variables in service method. It is wrong idea declaring service() method as synchronized because requests are processed sequentially.

If we create them as local variables in service method in every request local variables are created separately in service method StackFrame for processing that request. So there is no data corruption and hence wrong results are not generated.

Focus: if a variable is accessing by multiple thread only for reading its value but not for updating then in this case it is recommended to create it as non-static variable in servlet class to improve performance and also to save memory because this variable is created only once as servlet object is singleton.

For example assume we are inserting Student records in DB. In this example we must create PreparedStatement object in service() method else PStmt object is modified asynchronously from multiple threads and finally we get same record inserted multiple times as shown in the below diagram. We should create Connection object at Servlet class level, because we are not updating it so that object is created only once.

For the above example you can get source code from DVD:
Folder Path is: DVD\06 Class Programs\02 Adv Java\02 Servlets\12Servlet-Jdbc-Insert\src
To check above problem in this servlet class add "Thread.sleep(10000);" before "psmt.executeUpdate()" method call. For more details check your advanced java notes.

Deadlock
Deadlock is a situation where if two threads are waiting on each other to complete their execution. Deadlock is occurred due to wrong usage of synchronized keyword. If first thread is calling a synchronized method by using a locked object of second thread, and second thread calling a synchronized method by using a locked object of first thread then deadlock is occur.

Application #13:
Below application implements above condition to show deadlock situation.

```
//FirstClass.java
class FirstClass {
        synchronized void firstClassMethod(SecondClass sc){

                String name= Thread.currentThread().getName();
                System.out:println(name +" entered into FC.firstClassMethod()");

                try{ Thread.sleep(1000);}
                catch(Exception e){ e.printStackTrace(); }

                System.out.println(name +" is  trying to call sc.lastMethod()");
                sc.lastMethod();
        }
        synchronized void lastMethod(){
                System.out.println("Inside FC.lastMethod()");
        }
}
```

```
// SecondClass.java
class SecondClass{
        synchronized void secondClassMethod(FirstClass fc){

                String name= Thread.currentThread().getName();
                System.out.println(name +" is entered into sc.secondClassMethod()");

                try{ Thread.sleep(1000); }
                catch(Exception e){ e.printStackTrace(); }

                System.out.println(name +" is trying to call fc.lastMethod()");
                fc.lastMethod();
        }
        synchronized void lastMethod(){
                System.out.println("Inside sc.lastMethod()");
        }
}

//DeadLockDemo.java
public class DeadLockDemo implements Runnable{

        FirstClass fc           = new FirstClass();
        SecondClass sc          = new SecondClass();

        DeadLockDemo(){

                Thread th       = new Thread(this, "Racing Thread");
                th.start();

                fc.firstClassMethod(sc);//main thread Locked fc object.
                System.out.println("Back in Main Thread");
        }

        public void run(){
                sc.secondClassMethod(fc); //Racing Thread locked sc object.
                System.out.println("Back in other thread");
        }

        public static void main(String[] args) {
                new DeadLockDemo();
        }
}
```

Inter-thread communication:
The process of executing threads in sequence in loop with communication is called inter thread communication. We develop this concept when two different dependent tasks want to be executed continuously in sequence by two different threads.

The best example for the inter thread communication application is "Producer and Consumer" application. Producer should produce goods only when goods are consumed, and consumer can consume goods only when goods are produced.

To develop inter thread communication application we must use below three methods

1. wait()
2. notify()
3. notifyAll()

All these three methods are defined in *java.lang.Object* class with below prototypes.

```
public final void wait()  throws InterruptedException
public final native void wait(long milliSecs)  throws InterruptedException
public final void wait(long milliSecs, int nanoSecs)  throws InterruptedException

public final native void notify()
public final native void notifyAll()
```

The functionality of ***wait()*** method is – block the currently executing thread by releasing lock on the current object of the currently executing method. So that the other waiting threads can use this current object to execute other dependent synchronized method.

The functionality of ***notify()*** method is – notifying to waiting thread about the lock availability of the current object. Then the waiting thread is moved from "wait" state to "wait for lock acquisition" state. Once the lock is obtained then that waiting thread is moved to Runnable state for its turn to execute.

notifyAll() method is also used for the same above purpose but if there are multiple waiting thread are available.

Q) Why wait, notify and notifyAll methods are defined in Object class why not in Thread class?
A) These three methods are not only working on Thread objects they are also working on normal objects for unlocking object from the thread and for notifying above the lock availability on that object. So these three methods must be defined in Object class to work on normal objects also.

Q) What will happen if we do not call notify() or notifyAll() methods on waiting threads?
A) That thread is in the blocked state forever.

Q) What is the difference between sleep(100), join(100), wait(100) method calls?

- **sleep(100)** blocks thread execution independent of other thread for 100 milliseconds
- **join(100)** blocks thread execution by depending on either other thread till *it is completed* or till *100 milliseconds completed*, whichever coming first thread resume its execution immediately.
- **wait(100)** blocks thread execution by depending on either other thread till it is called *notify()* or *till 100 milliseconds completed*, whichever coming first thread resume its execution immediately.

Q) What is the rule in using these three methods or in what type of methods these three methods are allowed?

A) These three methods are allowed only in synchronized methods. If we call them in non-synchronized methods JVM throws an exception "*java.lang.IllegalMonitorStateException*". Because to release the lock that thread should the monitor of that object.

Application #15:

This application shows developing Inter thread communication using wait() and notify() for producer and consumer application.

```
//Factory.java
class Factory{

    int items;
    boolean itemsInFactory;

    synchronized void produce(int items){

        if( itemsInFactory ){

            try{
                wait();
            }
            catch(InterruptedException ie){
                ie.printStackTrace();
            }
        }

        this.items          = items;
        itemsInFactory      = true;

        Sopln("produced items: " + items);

        notify();

    }//produce

    synchronized int consume(){

        if( !itemsInFactory ){

            try{
                wait();
            }
            catch(InterruptedException ie){
                ie.printStackTrace();
            }
        }

        Sopln("items comsumed:"+items);
        itemsInFactory      = false;

        notifyAll();

        return items;
    }
}
```

```
//Producer.java
class Producer implements Runnable
{
        Factory fa;

        Producer(Factory fa)
        {
                this.fa  = fa;
                new Thread(this,"Producer").start();
        }

        public void run()
        {
                int i = 1;

                while( i <= 10)
                {
                        fa.produce( i++ );
                }
        }
}
```

```
//Consumer.java
class Consumer implements Runnable
{
        Factory fa;

        Consumer(Factory fa)
        {
            this.fa = fa;
            new Thread(this , "Consumer").start();
        }

        public void run()
        {
                int i = 1;

                while(i <= 10)
                {
                        fa.consume();
                        i++;
                }
        }
}
```

```
//ITCWithWaitNotify.java
public class ITCWithWaitNotify
{
        public static void main(String[] args)
        {
                Factory bajaj = new Factory();

                new Producer(bajaj);
                new Consumer(bajaj);

        }
}
```

Objects Communication Diagram:

Threads execution Processor Diagram:

What is the output from the below cases

Case #1: *wait* method is called in non-synchronized method without try/catch or throws keywords

A) **It leads to CE: unreported exception java.lang.InterruptedException must be caught or declared be thrown**

```
class WaitTest1 {
        void m1(){
                wait();
        }
}
```

Case #2: *wait* method is called in non-synchronized method in try/catch block

A) **It leads to RE: java.lang.IllegalMonitorStateException**

```
class WaitTest2 {
        void m1() {
                try{ wait(); }
                catch(InterruptedException ie){ ie.printStackTrace(); }
        }

        public static void main(String[] args){
                WaitTest2 wt = new WaitTest2();
                wt.m1();
        }
}
```

Case #3: no-arg *wait* method is called in synchronized method but *nofity()* method is not called from other thread

A) **This thread execution is blocked for ever**

```
class WaitTest3 {
        synchronized void m1() {
                try{
                        System.out.println( "Hi" );
                        wait();
                        System.out.println( "Hello" );
                }
                catch(InterruptedException ie){ ie.printStackTrace(); }
        }

        public static void main(String[] args){
                WaitTest3 wt = new WaitTest3();
                wt.m1();
        }
}
```

Case #4: long-parameter *wait* method is called in synchronized method but *nofity()* method is not called from other thread

A) This thread execution is resumed after the given time is elapsed

```
class WaitTest4 {
        synchronized void m1() {
                try{
                        System.out.println( "Hi" );
                        wait(6000);
                        System.out.println( "Hello" );
                }
                catch(InterruptedException ie){ ie.printStackTrace(); }
        }

        public static void main(String[] args){
                WaitTest3 wt = new WaitTest3();
                wt.m1();
        }
}
```

Application #16:

This application shows collecting threads into groups using ThreadGroup class.

```
//ThreadGroupImpDemo.java
class ThreadGroupImp implements Runnable{
        String name;   // Name of thread

        ThreadGroupImp(ThreadGroup thg, String threadName) {

                name = threadName;
                new Thread(thg, this).start();
        }
        public void run() {
                try {
                        for( int i = 1; i <= 10; i++ ) {
                                System.out.println( name + ": " + i);
                                Thread.sleep(1000);
                        }
                }
                catch(InterruptedException e){
                        System.out.println(name + " interrupted...");
                }

                System.out.println(name + " exiting...");
        }
}
```

```
public class ThreadGroupImpDemo {
        public static void main(String args[]) throws InterruptedException {

                ThreadGroup grpA = new ThreadGroup("Group A");
                ThreadGroup grpB = new ThreadGroup("Group B");

                ThreadGroupImp th1 = new ThreadGroupImp(grpA, "Thread1");
                ThreadGroupImp th2 = new ThreadGroupImp(grpA, "Thread2");
                ThreadGroupImp th3 = new ThreadGroupImp(grpB, "Thread3");
                ThreadGroupImp th4 = new ThreadGroupImp(grpB, "Thread4");


                grpA.list();
                grpB.list();

                System.out.println("Suspending GroupA...");
                grpA.suspend();

                Thread.sleep(2000);

                System.out.println("Resuming GroupA...");
                grpA.resume();

                // Waiting for the threads to end
                th1.t.join();
                th2.t.join();
                th3.t.join();
                th4.t.join();

                System.out.println("Main thread exiting...");
        }

}
```

Two different models of applications

Based on the number of threads used by an application, applications are divided into two types

1. Single thread model application- An application that uses single thread to execute multiple tasks is called single thread model application. It executes all tasks sequentially.

2. Multithread model application- An application that uses more than one thread to execute all tasks is called multithread model application. It executes all tasks concurrently. This application model is recommended only if all tasks are independent to each other.

Q) What is an "inline" thread?
A) A thread that is created with anonymous thread object is called inline thread. Anonymous thread object means a thread that is created without extending from Thread class or implementing from Runnable interface explicitly is call "anonymous" thread object. The other name for anonymous thread object is "inline" thread.

Note: It is not the official word given by SUN Microsystems. It is invented by our textbook authors or software industry experts.

How can we create Thread without extending from Thread class or implementing from Runnable interface explicitly?
Using anonymous inner class

```
//AnonymousThread.java – creates inline thread with Thread object
class AnonymousThread {
        public static void main(String[] args) {

                ( new Thread()
                  {
                        public void run(){
                                for (int i = 1; i<= 10; i++){
                                        System.out.println("run: "+i);
                                }
                        }
                  }
                ).start();
        }
}
//AnonymousRunnable.java – create inline thread with Runnable object
class AnonymousRunnable {
        public static void main(String[] args) {

                ( new Thread(
                        new Runnable()
                        {
                                public void run(){
                                        for (int i = 1; i<= 10; i++){
                                                System.out.println("run: "+i);
                                        }
                                }
                        }
                  )
                ).start();
        }
}
```

What is multithreading?
The process of creating multiple threads in a single application is called multithreading.

Multithreading is a technique by which a single set of code can be used by several threads at different stages of execution.

Unlike most other computer languages, Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. This is especially important for the interactive, networked environment in which Java operates, because idle time is common. For example, the transmission rate of data over a network is much slower than the rate at which the computer can process it. Even local file system resources are read and written at a much slower pace than they can be processed by the CPU. And, of course, user input is much slower than the computer. In a traditional, single-threaded environment, your program has to wait for each of these tasks to finish before it can proceed to the next one—even though the CPU is sitting idle most of the time. Multithreading lets you gain access to this idle time and put it to good use.

Difference between multitasking and Multithreading
Multitasking:
Multitasking allow to execute more than one tasks at the same time, a task being a program. In multitasking only one CPU is involved but it can switches from one program to another program so quickly that's why it gives the appearance of executing all of the programs at the same time. Multitasking allow processes (i.e. programs) to run concurrently on the program. For Example running the spreadsheet program and you are working with word processor also. Multitasking is running heavyweight processes by a single OS.

Multithreading:
Multithreading is a technique that allows a program or a process to execute many tasks concurrently (at the same time and parallel). It allows a process to run its tasks in parallel mode on a single processor system

In the multithreading concept, several multiple lightweight processes are run in a single process/task or program by a single processor. For Example, when you use a word processor you perform a many different tasks such as printing, spell checking and so on. Multithreaded software treats each process as a separate program.

Advantages of multithreading over multitasking:

- Reduces the computation time.
- Improves performance of an application.
- Threads share the same address space so it saves the memory.
- Context switching between threads is usually less expensive than between processes.
- Cost of communication between threads is relatively low.

Chapter 27

String Handling

- In this chapter, You will learn
  - Need of String, StringBuffer, StringBuilder
  - Differences between String, StringBuffer and StringBuilder
  - String Pooling
  - Why String class is immutable?
  - Why StringBuffer is mutable?
  - Why StringBuilder class is given?
  - Limitation of String?
  - Special operations we can perform on StringBuffer
  - Controlling StringBuffer capacity
  - Converting String to StringBuffer and vice versa.
- By the end of this chapter- you will be comfortable in working with string data by using all above three classes.

Interview Questions

By the end of this chapter you answer all below interview questions

The **String** class

- Define string?
- In how many ways we can store string data?
- Why String class is given when char array is already available?
- What are the different ways to construct Sting Object?
- What is the difference in String object creation from the String literal and the constructor?
- Explanation of String Pooling.
- What is immutable and why String object is immutable?
- What do you mean by immutable and mutable objects?
- Can we develop immutable object?
- Why StringBuffer class is given when we have String class to store string data?
- Why StringBuilder is given when we have StringBuffer?
- Is String thread safe?
- Definition of String, StringBuffer, and StringBuilder
- Can we assign String literal directly to StringBuffer or StringBuilder type variables?

- Printing String object
- Finding is string empty or not
- How can we find length of the String?
- What is the difference between length & length()?

- The limitation of String class.
- What is meant by concatenation, how Strings can be concatenated?
- How can we convert string case in the String?
- How can we replace a character or substring in this String?
- How can we trim string leading and trailing spaces?

- How String Objects must be compared for equality?
- What is meant by comparing string objects lexicographically, how it can be done?
- How can we read characters from the String?
- How can we find the character case in the String object?
- How can we find the position of a character or sub String?
- Searching for a character or a substring?
- Finding string startsWith or endsWith
- How can we retrieve substring from the String?
- How can we convert primitive values and objects to Strings?
- Splitting string into tokens

- What are the operations we cannot perform on string using String object?

The **StringBuffer, StringBuilder** classes

- Define StringBuffer.
- When StringBuffer class must be chosen?
- What are the special operations can be performed on StringBuffer; those cannot be applied on String?
- Ways of creating StringBuffer object creation?
- Appending, Inserting, Deleting, reversing, and overriding characters in the StringBuffer.
- Finding StringBuffer capacity and length
- Controlling StringBuffer capacity
- Difference between String and StringBuffer, and StringBuffer and StringBuilder.
- Ways of converting String to StringBuffer and vice versa.

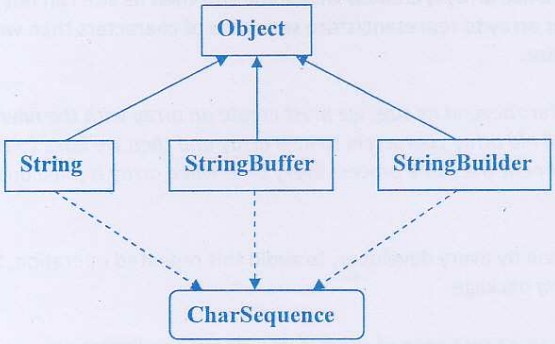
string is a sequence of characters placed in double quote (" ").
Performing different operations on string data is called String Handling.

In *java.lang* package we have below three classes to store and perform different operations on string of characters.

1. String
2. StringBuffer
3. StringBuilder

StringBuilder class was introduced in Java 5, as a *non-thread safe* class. Except this all its operations are exactly same as StringBuffer.

All these three classes are siblings, and they are subclasses of ***CharSequence*** interface. This interface also was introduced in Java 5. Below diagram shows the inheritance relation of these three classes,



Note:

- All these three classes are ***final*** classes
- These three classes are also subclasses of *java.io.Serializable* interface
- String class is a subclass of *Comparable* interface, but StringBuffer and StringBuilder are not

Any data that is place in " " is treated by compiler and JVM as an instance of java.lang.String
For example:
"abc" is an object of type java.lang.String

Find CE in the below list

```
String        s1    = "abc";   ✓
StringBuffer  sb1   = "abc";   X
StringBuilder sb2   = "abc";   X
```

We cannot assign string literal directly to StringBuffer and StringBuilder variables, it leads to CE: **in compatible types**

Q) How can we store string data in StringBuffer & StringBuilder class objects?
A) Through constructors and methods of these two classes. These constructors and methods perform copying operation. It means they store given string characters in this SB/SB object.

The **String** class

The ***String*** class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

Definition
In general String can be defined as it is a sequence of characters. Strings are constant; their values cannot be changed in the same memory after they are created,

So String is defined as it is an ***immutable sequence of characters***.

Why String class is given when *char array* is available to represent sequence of characters?
String class is given to store characters dynamically without size limitation and also to perform different common operations of string data.
Explanation:
The limitation on arrays is size. Once array is created with some size then its size can not be modified. So if we use character array to represent/store sequence of characters then we can only store characters up to its size.

If we want to store new characters beyond its size, we must create an array with the new required size and should copy all old array characters to new array and then we have to store new values at end. We should repeat the same process every time when array is filled and want to add new characters.

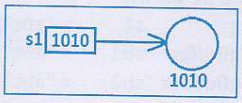
Since this operation must be done by every developer, to avoid this repeated operation, SUN has given String class in *java.lang* package.

Hence using String class we can store sequence of characters without size limitation.

What are the possible ways to create String object?
String object can be created in two ways
1. By assigning string literal directly => String s = "abc";
2. By using any one of the string class constructors

String class has totally 15 constructors among them below are the 8 important constructors, check *API documentation* for more details

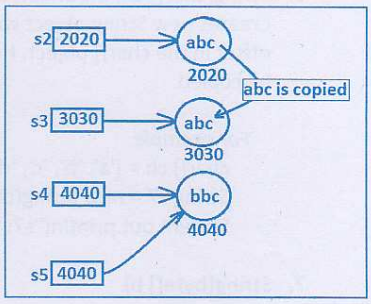
1. **String()**
   Creates empty string object, not null string object
   For example:
   ```
   String s1 = new String();
   System.out.println(s1); //no output
   ```
   string object memory structure



2. **String(String value)**
   Creates String object with given String object characters.
   It performs String copy operation

For example:

```
String s2 = "abc";

//string copy
String s3 = new String(s2);

//creating string with direct string literal
String s4 = new String("bbc");

//String assignment
String s5 = s4;

System.out.println("s2: "+s2);//abc
System.out.println("s3: "+s3);//abc
System.out.println("s4: "+s4);//bbc

System.out.println(s2 == s3);//false
System.out.println(s4 == s5);//true
```



Note:

- **In String copy** two different String objects are created with same data, it is like a file copy operations in OS.
- **In String assignment** current object reference is copied to destination variable, new object is not created.

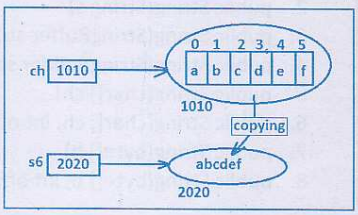
3. **String(StringBuffer sb)**
   Creates new String object with the given StringBuffer object' data.
   Performs string copy operation from StringBuffer object to String object

4. **String(StringBuilder sb)**
   Creates new String object with the given StringBuilder object content
   Performs string copy operation from StringBuilder object to String object

5. **String(char[] ch)**
   Creates String object with the given char array values
   Performs string copy operation from char[] object to String object

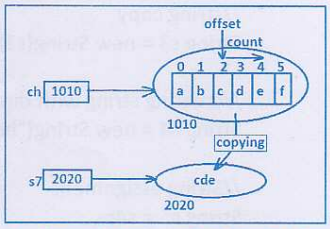
For example:

```
char[] ch = {'a', 'b', 'c', 'd', 'e', 'f'};
String s6 = new String(ch);
System.out.println("s6: "+s6);//abcdef
```



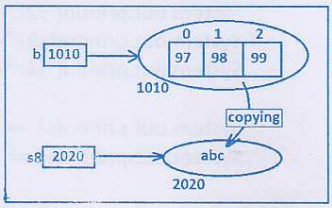
6. **String(char[] ch, int offset, int count)**
   Creates new String object with the given count number of characters from the given offset in the char[] object. Here offset is the starting index from which characters must be copied.

   For example:
   ```
   char[] ch = {'a', 'b', 'c', 'd', 'e', 'f'};
   String s7 = new String(ch, 2, 3);
   System.out.println("s7: "+s7);//cde
   ```



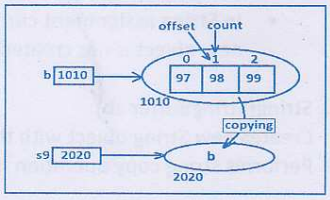
7. **String(byte[] b)**
   Creates new String object by copying the given byte[] numbers by converting them into their ASCII characters.

   For example:
   ```
   byte[] b = {97, 98, 99};
   String s8 = new String(b);
   System.out.println(b);//abc
   ```



8. **String(byte[] b, int offset, int count)**
   Creates new String object with the given count number of bytes from the given offset in the byte[]. All bytes are stored in their ASCII character form

   For example:
   ```
   byte[] b = {97, 98, 99};
   String s9 = new String(b, 1, 1);
   System.out.println("s9: "+s9);//b
   ```



Note:
Write all above object creation statements in main method and compile and execute.

3 important rules
We have there 3 important rules on String constructors, so recollect once all above 8 constructors

1. public String()
2. public String(String s)
3. public String(StringBuffer sb)
4. public String(StringBuilder sb)
5. public String(char[] ch)
6. public String(char[] ch, int offset, int count)
7. public String(byte[] b)
8. public String(byte[] b, int offset, int count)

All these constructors are overloaded constructors with siblings.

Rule #1:
For char[] and byte[] parameter constructors the give offset and count argument value must be within the range of [0 to string length -1], else it leads to
RE: ***java.lang.StringIndexOutOfBoundsException***

```
byte[] b = {97, 98, 99}
String s10 = new String(b, 1, 4);   × RE: SIOBE: 4
        //we do not have 4 bytes in array
```

Rule #2:
We cannot pass ***null*** as argument directly to constructor, it leads CE: ***ambiguous error*** because it is matched with all parameters of String class constructors.

```
String s11 = new String(null); × CE: ambiguous error
```

Rule #3:
We cannot create String object with ***null*** it leads to RE: ***java.lang.NullPointerException***

```
String s1 = null;
String s2 = new String(s1);   × RE: NPE
String s3 = new String( (StringBuffer) null);  ×  RE: NPE
```

Identify the operation I am doing in the below statements

```
String s4 = "";
String s5 = new String();
String s6 = new String("");
```
⇐ **Empty string object creation**

```
String s7 = null;
```
⇐ ***null*** **string referenced variable creation**

```
//String s8 = new String(null); CE: ambiguous error
//String s8 = new String((String)null); RE: NPE
//String s8 = new String(s7); RE: NPE
```
⇐ **We cannot create *null* String object**

```
String s = new String("null");
```
⇐ **"null" is not *null* value. It is a string with characters *n u l l*. So string object is created with characters "null"**

Note: We can create null String referenced variable (ex: s7), but we cannot create String object with null. It leads to NPE

```
String s8 = new String("A");
String s9 = new String(s8);
```
⇐ **String copy - Creating String object from another String object**

```
String s10 = new String("B");
String s11 = s10;
```
⇐ **assigning one string object reference to another String reference variable**

Why String object is immutable?
Because of String pooling concept

String pooling
Q) What is the difference in creating String objects *using constructor* and by *assigning literal*?
A) String pooling

String pooling means pooling strings, it means grouping string objects.

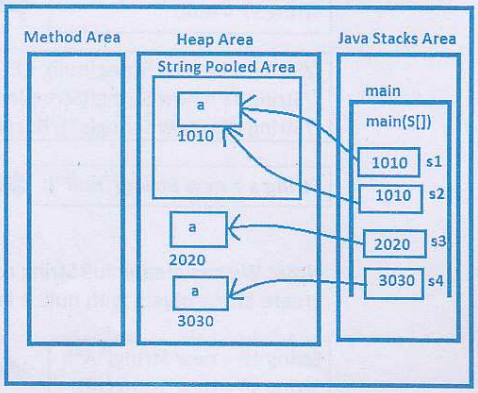
If we create String objects by assigning same string literal, only one object is created and all referenced variables are initialized with that same String object reference. Then all referenced variables are pointing to same String object. This is reason why String object become immutable. It means string data modification is not stored in same memory.

Since a string object is pointing by multiple referenced variables if we change that String value using one referenced variable, all other referenced variables those are pointing to that objects are also affected. To solve this problem SUN decided to create String as immutable.

This String pooling concept is not applicable for the String objects those are created using "*new keyword and constructor*". They have separate object and those string objects are created in heap area directly. The string objects created with expression (+ operator), method returned String objects are also does not come under string pooling. So, all these objects are created in heap area, not in string constant pooled area.

Below diagram shows String Pooling with JVM architecture

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

        String s1 = "a";
        String s2 = "a";

        System.out.println(s1 == s2);          //true
        System.out.println(s1.equals(s2));     //true


        String s3 = new String("a");
        String s4 = new String("a");

        System.out.println(s3 == s4);          //false
        System.out.println(s3.equals(s3)); //true
    }
}
```



Q) Why this String pooling?
A) To improve performance by saving memory.

String expression calculation

If an expression has only String literals that expression is calculated by compiler and places result as literal. So JVM creates that String object in stringpooled area. If expression has a variable compiler does not calculate it, it is calculated by JVM and creates that object in heap area.

Q) In below code how many string objects are created?

```
String s1 = "a";        String s2 = "a";
String s3 = "a";        String s4 = "b";

String s5 = new String("a");
String s6 = new String("a");

String s7 = "ab";
String s8 = s1 + "b";
String s9 = "a"+"b";

System.out.println(s1 == s2);// true
System.out.println(s5 == s6);// false
System.out.println(s7 == s8);// false
System.out.println(s7 == s9);// true
```

A) 6 objects, they are s1, s4, s5, s6, s7, s8

String Common Operations:-

1. Checking String is empty or not
2. Finding length of the String
3. Printing String objects
4. Compare String objects normally
5. Comparing String objects lexicographically
6. Reading character at the given index
7. Finding index of given character
8. Searching for the given String
9. Checking the String starts with
10. Checking the String ends with
11. Retrieving substring
12. String objects concatenation
13. Removing trailing and leading spaces
14. Changing all string characters to lowercase
15. Changing all string characters to uppercase
16. Replacing old character with new character
17. Replacing old String with new String
18. Converting String into character array and byte array
19. Preparing String object from a given character String
20. Split the String into tokens
21. Converting any data type, including primitive types, into String type.

Below program shows using all String class methods

```
class Example{}

class StringOperations {

        public static void main(String[] args) {

                String s = "Java Programming Language";

                //finding string is empty or not
                System.out.println(s.isEmpty());

                //finding length of the string
                System.out.println(s.length());

                String s1 = "";
                String s2 = " ";
                String s3 = new String("");
                System.out.println(s1.isEmpty());
                System.out.println(s2.isEmpty());
                System.out.println(s3.isEmpty());

                System.out.println(s1.length());
                System.out.println(s2.length());
                System.out.println(s3.length());

                //printing string object
                System.out.println(s);

                //comparing two strings
                String s1 = new String("abc");
                String s2 = new String("abc");
                String s3 = new String("Abc");

                System.out.println(s1 == s2);        //false    => different references
                System.out.println(s1.equals(s2)); //true     => both objects has same state

                System.out.println(s2 == s3);        //false => different references
                System.out.println(s2.equals(s3)); //false => both objects has different state

                //comaring strings without case, below method is defined in String class
                //public boolean equalsIgnoreCase(String s)

                String s4 = new String("a");
                String s5 = new String("A");
```

```
System.out.println(s4.equals(s5));              //false => compares with case
System.out.println(s4.equalsIgnoreCase(s5)); //true => compares without case

//comparing strings lexicographically, means after comparison method should
//return difference between string content,
//public int compareTo(String s)              <= compares Strings with case
//public int compareToIgnoreCase(String s)  <= compares Strings without case

String s6 = new String("a");
String s7 = new String("A");
System.out.println(s6.compareTo(s7)); //32              => a - A => 97 - 65   =>32
System.out.println(s6.compareToIgnoreCase(s7));//0  => a - a => 97 - 97   =>0

String s8 = "abcdef";
String s9 = "abc";
System.out.println(s8.compareTo(s9)); //3   =>s8.length() – s9.length() => 3

String s10 = “abcdef”;
System.out.println(s9.compareTo(s10)); //-3 =>s9.length() – s10.length() => -3

String s11 = “adc”;
System.out.println(s11.compareTo(s9)); //2  => d – b => 100 – 98 => 2

//startsWith() / endsWith()
String s = "Java Programming Language";
System.out.println(s.startsWith("Java"));
System.out.println(s.startsWith("java"));
System.out.println(s.startsWith("hari"));

System.out.println(s.endsWith("hari"));
System.out.println(s.endsWith("Language"));

//print character of the given index, string index starts from ZERO, because its
internal object is char array, should use below method.
//public char chatAt(int index)

String s1 = new String("Java Programming Language");
System.out.println("character at 10th index in s1 string: "+ s1.charAt(10));

//print all characters in given string with index, we should write our own logic
with charAt() and length() methods
for (int i = 0; i < size ; i++ ){
        System.out.println("character at index "+ i + "is : "+ s1.charAt(i));
}
```

```
//Note: If we pass index out of range (index < 0 || index >= string.length()) to
charAt() method it leads StringIndexOutOfBoundsException
//In the above for loop if we keep condition i <= size, after printing all 25
characters JVM throws SIOBE for 25th index.

System.out.println(s1.charAt(0)); //=> J
System.out.println(s1.charAt(10)); //=> a
//System.out.println(s1.charAt(-10)); //=> SIOBE
//System.out.println(s1.charAt(26)); //=> SIOBE

//Finding index of given character of String in the given string,
use below methods

//below methods return first occurrence of given character or String
        //public int indexOf(char ch)
        //public int indexOf(String str)

//below methods return last occurrence of given character or String
        //public int lastIndexOf(char ch)
        //public int lastIndexOf(String str)

//below methods return first occurrence of given character or String by
searching it from the "givenIndex" not from ZERO index.
        //public int indexOf(char ch, int fromIndex)
        //public int indexOf(String str, int fromIndex)

//below methods return first occurrence of given character or String by
searching it from the "givenIndex" not from END index.
        //public int lastIndexOf(char ch, int fromIndex)
        //public int lastIndexOf(String str, int fromIndex)

System.out.println(s1.indexOf('a')); //=> 1
System.out.println(s1.lastIndexOf('a')); //=> 22
System.out.println(s1.indexOf('a', 4)); //=> 10
System.out.println(s1.lastIndexOf('a', 22)); //=> 22
System.out.println(s1.lastIndexOf('a', 21)); //=> 18

System.out.println(s1.indexOf("Programming")); //=> 5

//Focus: the given char or string is not available in the current string, above
methods return -1.
System.out.println(s1.indexOf("hari")); //=> -1
System.out.println(s1.indexOf('k')); //=> -1
```

```
//write a program to find the given string or char available or not
//until 1.4 version, we should write a condition using indexOf() method to find
string is available or not
//In 1.5 version, SUN introduced a new method called "contains(String s)" to
find string is available or not
//It returns true if substring is available in current string , else returns false.

//check below methods definition given at end of the program
findWithIndexOf("abcdef");
findWithContains("abcdef");

findWithIndexOf("asdfasdfsadf hari krishna asdfasdfsadf");
findWithContains("asdfasdfsadf hari krishna asdfasdfsadf");

//substring()
String s = "Java Programming Language";
//Note: if (begin index == end index) this method returns empty string

System.out.println(s1.substring(5));
System.out.println(s1.substring(5, 15));
System.out.println(s1.substring(5, 16));
//      System.out.println(s1.substring(-6));
System.out.println(s1.substring(5, 25));
//      System.out.println(s1.substring(5, 26));
System.out.println(s1.substring(24, 24));
System.out.println(s1.substring(24, 25));
System.out.println(s1.substring(23, 24));
System.out.println(s.substring(5,16));
System.out.println(s.substring(s.indexOf("P"),s.indexOf(" L")));
System.out.println(s.substring(s.indexOf("Language")));
System.out.println(s.substring(16,5));
System.out.println(s.substring(-1,5));

//concatenating new string,
//adding given string characters at end of current string characters and placing
//the result in new object is called concatenation
//We can do it in two ways
        //1. using + operator
        //2. using concat() method
                //public String concat(String newString)

//concatenation using + operator
String s1 = "a";
String s2 = s1 + "b";
```

```
System.out.println(s1);//a
System.out.println(s2);//ab
System.out.println(s1 == s2); //false

//concatenation using concat() method
String s3 = "a";
String s4 = s3.concat("b");

System.out.println(s3);//a
System.out.println(s4);//ab
System.out.println(s3 == s4); //false

//String limitation
//If we perform modification on string using String class methods, and if that
object is changed as a result of that method call, new String object is created
with that result. If we store that new object reference in reference variable, it is
reachable else it is garbage collected.

//In the below statement string is modified => so new String object is created =>
but its reference is not stored => hence s4 state is still ab only

s4.concat("c");
System.out.println(s4);//ab

// Due to a String method call, if String object is not changed, then the current
//String object reference is returned

//What is the output from the below program?
String s5 = s4.concat("");
System.out.println(s4); // ab
System.out.println(s5); // ab
System.out.println(s4 == s5); //true

Note: '+' operator always creates and returns new string object irrespective of
the object is changed or not.

String s6 = "a";
String s7 = s6 + "";
System.out.println(s6 == s7); //false

//converting all characters in string to upper or lower case
//public String toUpperCase()
//public String toLowerCase()

String s1 = "Hari Krishna";
```

```
System.out.println(s1.toUpperCase()); //HARI KRISHNA
System.out.println(s1.toLowerCase()); //hari krishna
System.out.println(s1); //Hari Krishna

String s1 = "hari";
String s2 = s1.toLowerCase();
String s3 = s1.toUpperCase();
System.out.println(s1); //hari
System.out.println(s2); //hari
System.out.println(s3); //HARI
System.out.println(s1 == s2); //true
System.out.println(s1 == s3); //false

//replacing a character with new character, use below methods of String class
//public String replace(char oldChar, char newChar)
//public String replace(CharSequence oldString, CharSequence newString)
//public String replaceAll(String oldString, String newString)
//public String replaceFirst(String oldString, String newString)

String s1 = "Java Programming Language";
String s2 = s1.replace('J', 'K');

System.out.println("s1 String: "+s1);
System.out.println("s2 String: "+s2);

String s3 = s1.replace('a', 'A');
System.out.println("s3 String: "+s3);

String s4 = s1.replace("Programming", "Object-Oriented Programming");
System.out.println("s4 String: "+s4);

String s5 = "Ha Ha Ha";
String s6 = s5.replace("Ha", "Hello");
System.out.println("s5 String: "+s5);
System.out.println("s6 String: "+s6);

String s7 = s6.replace("hello", "Hi");
System.out.println("s7 String: "+s7);

//due to replace method call if there is no change in current string, it returns
same current string object reference.
System.out.println(s6 == s7);//true
```

```
        //deleting trailing and leading spaces of a String
        //public String trim()  => it will not remove middle spaces

        String s1 = new String(" hari krishna ");
        String s2 = s1.trim();
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s1 == s2);

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

        //What is the output from below program?
        String s3 = new String("Naresh Technologies");
        String s4 = s3.trim();

        System.out.println(s3);
        System.out.println(s4);
        System.out.println(s3 == s4);

        //split()
        String[] sarray = s.split(" ");
        int size = sarray.length;
        for (int i = 0; i < size ; i++ ){
                System.out.println(i + " tocken is "+ sarray[i]);
        }
    }
    static void findWithIndexOf(String originalString){

        if(originalString.indexOf("hari") != -1){
                System.out.println("The String hari is available ");
        }
        else{
                System.out.println("The String hari is not available ");
        }
    }
    static void findWithContains(String originalString){
        if(originalString.contains("hari")){
                System.out.println("The String hari is available ");
        }
        else{
                System.out.println("The String hari is not available ");
        }
    }
}
```

Q) How can we remove middle spaces?
A) We must use replace method. Remove not middle spaces also begin and end spaces.
For example:

```
String s4 = " abc abc  ";
String s5 = s4.replace( " ", "" );
System.out.println(s4);
System.out.println(s5);
System.out.println(s4.length());
System.out.println(s5.length());
```

Assignment

Write a program to reverse the words in the String objects
InputString: How are you?
Ouput String: you? are How

```
//StringWordsReverse.java
public class StringWordsReverse{

        public static String reverseStringWords(String s){

                String[] stringWords = s.split(" ");

                String resultString = "";

                for (int i = stringWords.length-1; i >= 0; i--){

                        resultString += stringWords[i] + " ";

                }

                return resultString;
        }
}
```

Just you count how many objects are created in the above program.

It leads to big performance issue. we should use *StringBuffer* or *StringBuilder* class to import performance by reducing memory usage.

```
//Test.java
import java.util.*;

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

                Scanner s = new Scanner(System.in);
                System.out.print("Enter string: ");

                String str = s.nextLine();
                String result = StringWordsReverse.reverseStringWords (str);

                System.out.println("original string: "+str);
                System.out.println("reverse string: "+result);
        }
}
```

The **StringBuffer** class

Definition
It is a thread-safe, mutable sequence of characters. A string buffer is like a String, but can be modified in the same memory location.

How StringBuffer object can be created?
StringBuffer object can be created only in one way
1. By using its available constructor

In StringBuffer class we have below 4 constructors

1. **public StringBuffer()**
   Creates empty StringBuffer object with default capacity 16 buffers, it means it has 16 empty locations

   For example:
   ```
   StringBuffer sb = new StringBuffer();
   System.out.println("sb: "+sb); //sb:
   System.out.println("sb1 capacity: "+sb1.capacity()); //16
   ```

2. **public StringBuffer(int capacity)**
   Creates empty StringBuffer object with the given capacity

   For example:
   ```
   StringBuffer sb2 = new StringBuffer(3);
   System.out.println("sb2: "+sb2);//sb2:
   System.out.println("sb2 capacity: "+sb2.capacity());//3
   ```

   Rule: capacity value should be >=0, if we pass negative number, JVM throws "*java.lang.NegativeArraySizeException*"
   ```
   StringBuffer sb = new StringBuffer(-5);   RE: NASE
   ```

3. **public StringBuffer(String s)**
   Creates StringBuffer object with given String object data. It performs string copy from String to StringBuffer object. The default capcity is [16 + s.length()]

   For example:
   ```
   StringBuffer sb3 = new StringBuffer("abc");
   System.out.println("sb3: "+sb3); //sb3: abc
   System.out.println("sb3 capacity: "+sb3.capacity()); //19
   ```

4. **public StringBuffer(CharSequence cs)**
   Creating new StringBuffer object with the give CS object characters, it also performs string copy from CS object to StringBuffer object. Its capacity is [16 + cs.length()]

For example:

```
StringBuffer sb4 = new StringBuffer("abc");
StringBuffer sb5 = new StringBuffer(sb4);

System.out.println("sb4: "+sb4); //abc
System.out.println("sb5: "+sb5); //abc
System.out.println("sb4 capacity: "+sb4.capacity()); //19
System.out.println("sb5 capacity: "+sb5.capacity()); //19

System.out.println(sb4 == sb5); //false
```

Rule:
We cannot create SB object by passing null. It leads to RE: NPE
In StringBuffer case CE: ambiguous error is not raised for null argument because its constructors are overloaded with super and subclasses CharSequence and String.

For null argument String parameter constructor is called.

```
StringBuffer sb = new StringBuffer(null); RE: NPE
```

StringBuilder constructors
StringBuilder class functionality is exactly same as StringBuffer class. So it also has same StringBuffer parameter constructors as shown below

1. public StringBuilder()
2. public StringBuilder(int capacity)
3. public StringBuilder(String s)
4. public StringBuilder(CharSequence cs)

Q) What is the difference between StringBuffer and StringBuilder?
A) StringBuffer is synchronized and where as StringBuilder is not synchronized

StringBuilder class is given in Java 5 version to develop non synchronized StringBuffer object. Except this difference StringBuffer and StringBuilder operations are exactly same.

So,

- StringBuffer object must be used in Multithread programming model applications
- StringBuilder object must be used in Single Thread programming model applications

Find out valid objects creation in the below list

```
String s1 = "abc";
String s2 = new String("abc");

StringBuffer sb1 = new StringBuffer("abc");
StringBuffer sb2 = "abc";
```

Special operations those we can perform only on StringBuffer and StringBuilder classes

Since StringBuffer is mutable we can perform below 5 operations specially on StringBuffer

1. Append
2. Insert
3. Delete
4. Reverse
5. Replacing character at given index

In StringBuffer class, we have below methods to perform above 5 operations

1. public StringBuffer ***append***(xxx data)
2. public StringBuffer ***insert***(int index, xxx data)
   - where xxx is java data types
   - above methods are overloaded to take all possible java datatype values
3. public StringBuffer ***deleteCharAt***(int index)
   public StringBuffer ***delete***(int start, int end)
4. public StringBuffer ***reverse***()
5. public StringBuffer ***setCharAt***(int index, char ch)

All above methods *return current StringBuffer object reference* after modification is completed

Rule: index value must be with range of ["0" to sb.length() -1], else it leads to "*java.lang.StringIndexOutOfBoundsException*"

Below program shows performing all above operations

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

        //append
        StringBuffer sb1  = new StringBuffer("abc");
        StringBuffer sb2  = sb1.append("d");

        System.out.println(sb1);//abcd
        System.out.println(sb2);//abcd
        System.out.println(sb1 == sb2);//true

        //insert
        StringBuffer sb3 = new StringBuffer("2942011");

        System.out.println(sb3);// 2942011

        sb3.insert(2, '/');
        System.out.println(sb3);// 29/42011

        sb3.insert(4, '/');
        System.out.println(sb3);// 29/4/2011
```

```
        //delete
        StringBuffer sb4 = new StringBuffer("Hari xyz Krishna");
        System.out.println(sb4);//Hari xyz Krishna

        sb4.delete(sb4.indexOf("xyz") , sb4.indexOf("xyz") + 3);
        //sb.delete(5, 5 +3 );   //sb.delete(5, 8);
        System.out.println(sb4); //Hari  Krishna

        sb4.deleteCharAt(4);
        sb4.deleteCharAt(4);
        System.out.println(sb4); //HariKrishna

        //reverse
        StringBuffer sb5 = new StringBuffer("abc");
        System.out.println(sb5); //abc

        sb5.reverse();
        System.out.println(sb5); //cba

        //capacity() and length()
        //public int capacity()
        //public int length()

        StringBuffer sb6 = new StringBuffer();
        System.out.println(sb6);//
        System.out.println(sb6.capacity());//16
        System.out.println(sb6.length());//0

        sb6.insert(0,"abc");
        System.out.println(sb6);//abc
        System.out.println(sb6.capacity());//16
        System.out.println(sb6.length());//3

        StringBuffer sb7 = new StringBuffer("abc");
        System.out.println(sb7);//abc
        System.out.println(sb7.capacity());//19
        System.out.println(sb7.length());//3
    }
}
```

Assignment

Write a program to perform append, and insert operations

Take a string "object language". Change this string as "Object-Oriented Programming Language is secure"

Controlling StringBuffer capacity

We have below three methods to control StringBuffer capacity

1. public void ensureCapacity(int minimumCapacity)
2. public void trimToSize()
3. public void setLength(int newLength)

enusureCapacity

Ensures that the capacity is at least equal to the specified minimum. If the current capacity is less than the argument, then a new internal array is allocated with greater capacity. The new capacity is the larger of:

- The minimumCapacity argument.
- Twice the old capacity, plus 2.

If the minimumCapacity argument is non positive, this method takes no action and simply returns, the capacity remains the same current capacity.

What is the output from below program?

```
StringBuffer sb = new StringBuffer();
System.out.println(sb.capacity());     //16

sb.ensureCapacity(17);                 //16*2+2 => 34 => (34 > 17)
System.out.println(sb.capacity());     //34

sb.ensureCapacity(100);                //34*2+2 => 70 => (70 < 100)
System.out.println(sb.capacity());     //100

sb.ensureCapacity(-3);
System.out.println(sb.capacity());     //100
```

Q) What is the capacity of length of SB object in the below program?

```
StringBuffer sb = new StringBuffer();

for (int i = 1; i<= 17 ; i++){
        sb.append(i);
}

System.out.println(sb.capacity());
System.out.println(sb.length());
```

trimToSize
It reduces SB capacity to its size. The prototype is: *public void trimToSize()*

What is the output from the below program?

```
StringBuffer sb1 = new StringBuffer("abc");
System.out.println(sb1.capacity());//19
System.out.println(sb1.length());//3

sb1.trimToSize();

System.out.println(sb1.capacity());//3
System.out.println(sb1.length());//3
```

setLength
Sets the length of the SB to current given length. Its prototype is:

```
public void setLength(int newLength)
```

Rule: The newLength argument must be greater than or equal to 0.
Else this method throws "*java.lang.IndexOutOfBoundsException*".

Case #1: If the passed length is greater than current length, it places empty spaces
What is the output from below program?

```
StringBuffer sb2 = new StringBuffer("abcdefg");
System.out.println(sb2.capacity()); //23
System.out.println(sb2.length()); //7
System.out.print(sb2); //adbcdefg
System.out.println("hari"); //adbcdefghari

sb2.setLength(9);
System.out.println(sb2.capacity()); //23
System.out.println(sb2.length()); //9
System.out.print(sb2); //adbcdefg
System.out.println("hari"); //adbcdefg  hari
```

Case # 2: If the passed length is less than current length,
it removes all characters > this length

What is the output from below program?

```
sb2.setLength(4);
System.out.println(sb2.capacity()); //23
System.out.println(sb2.length()); //4
System.out.print(sb2); //abcd
```

Diff #2: Creating String, StringBuffer objects
String object can be created in two ways
1. By assigning string literal => String s1 = "a";
2. By using its available constructors

StringBuffer objects are creating only in one way
1. By using its available constructor.

Diff #3: Capacity
String does not have default capacity, the passed string length is its capacity.

StringBuffer has default capacity 16 buffers. It increases its capacity automatically when size reached its capacity: formula is: (current capacity * 2) + 2.
To control string buffer capacity or to create string buffer with user given capacity we have
1. Constructor - to assign capacity at the of creation
2. method - to change its capacity after its creation

Diff #4: Concatenation
Using String object, we can concat new string to the current string in two ways
1. using + operator
2. using concat() method

Using StringBuffer we can perform concat operation only in one way
1. using append() method

Important point to be remembered
Due to concatenation operation by using concat() method, if current string object is modified, this method creates new object with this result and returns that object reference. If it is not modified it returns the same current object reference. But when we use + operator whether current string object is modified or not modified, JVM always creates new String object.

Where as the append() always stores modification result in the current SB object and returns the current SB object

What is the output from the below program?

```
String s1 = new String("a");
String s2 = s1.concat("b");
System.out.println(s1);
System.out.println(s2);
System.out.println(s1 == s2);
s2.concat("c");
System.out.println(s2);
```

```
StringBuffer sb1 = new StringBuffer("a");
StringBuffer sb2 = sb1.append("b");
System.out.println(sb1);
System.out.println(sb2);
System.out.println(sb1 == sb2);
sb2.append("c");
System.out.println(sb2);
```

Diff #5: Special operations those we can only perform on StringBuffer,

1. append
2. insert
3. delete
4. reverse

Since String is immutable we cannot perform these operations on String.

Diff #6: Comparison

Equals method is overridden in String class to compare String objects with state So string objects are compared with reference by using == operator and with sate by using equals() method..

In StringBuffer and StringBuilder classes equals() method is not overridden, so using == operator and equals() method their objects are compared with reference.

In all three class toString() method is overridden to print the object state.

What is the output from below program?

```
String s1 = "a";
String s2 = "a";
System.out.println(s1 == s2);
System.out.println(s1.equals(s2));

String s3 = new String("a");
String s4 = new String("a");

System.out.println(s3 == s4);
System.out.println(s3.equals(s4));

StringBuffer sb1 = new StringBuffer("a");
StringBuffer sb2 = new StringBuffer("a");

System.out.println(sb1 == sb2);
System.out.println(sb1.equals(sb2));
```

How many String objects are created in the below program?

```
String s1 = “a”;
String s2 = “a”;
String s3 = “b”;

String s4 = “ab”;
String s5 = “a” + “b”;
String s6 = s1 + “b”;
String s7 = s1 + s2;
```

Q) What is the best design in defining a method to take and return string data as argument?
A) The parameter and return type should be java.lang.String, not StringBuffer

The problem if we take StringBuffer as parameter or return type is: user should convert that string data as StringBuffer object, because StringBuffer will not accept string data directly.

For Example:

```
void m1(String s){

}

void m1(StringBuffer sb){

}
```

```
m1("abc");

m1(new StringBuffer("abc"));
//It is not the convenient way for user.
```

Project development procedure:

- In projects we store string data by using java.lang.String object.
- To perform modifications on that string data we convert/copy that string into StringBuilder object.
- After modification we convert/copy back the result string to String object and return the result.

Q) What are the different ways for converting String to StringBuffer and vice versa.

- Using *StringBuffer(String)* constructor we can convert String to StringBuffer.
- Using *String(StringBuffer)* constructor and *StringBuffer.toString()* we can convert StringBuffer to String.

Write a program to reverse the string words?
Input: How Are You.
Output: You Are How

```
class  StringWordsReverse{

        public static String reverseStringWords(String s){

                String[]        stringWords    = s.split(" ");
                int             noOfWords      = stringWords.length;

                StringBuilder resultBuffer = new StringBuilder ();

                for (int i = noOfWords -1; i >= 0 ; i--){
                        resultBuffer.append(stringWords[i]);
                        resultBuffer.append(" ");
                }

                //removing last space character
                return resultBuffer.toString().trim();
        }

        public static void main(String[] args){
                System.out.print(reverseStringWords("How Are You"));
        }
}
```

Immutable objects and their creation

Immutable objects are simply objects whose state (the object's data) cannot be changed after construction. Examples of immutable objects from the JDK include String and Integer. Immutable objects greatly simplify your program, since they:

- are simple to construct, test, and use
- are automatically thread-safe and have no synchronization issues
- do not need a copy constructor
- do not need an implementation of clone
- allow hashCode to use lazy initialization, and to cache its return value
- do not need to be copied defensively when used as a field
- make good Map keys and Set elements (these objects must not change state while in the collection)
- have their class invariant established once upon construction, and it never needs to be checked again
- always have "failure atomicity" (a term used by Joshua Bloch) : if an immutable object throws an exception, it's never left in an undesirable or indeterminate state

Immutable objects have a very compelling list of positive qualities. Without question, they are among the simplest and most robust kinds of classes you can possibly build. When you create immutable classes, entire categories of problems simply disappear.

Make a class immutable by following these guidelines :

- *Make the class final, or use static factories and keep constructors private* to ensure the class cannot be overridden
- *Make fields private and final* to ensure the variable is not accessible from outside of class and also to ensure it is not modified unknowingly in the same class from its mutator methods.
- Define parameterized constructor to initialize the state of the object to ensure callers are constructing an object completely in a single step, instead of using a no-argument constructor combined with subsequent calls to mutator methods
- Do not provide mutator methods not to allow changes *or*, if we want to allow changes define mutator method but store the result in new object and return it.

For example

- String is an immutable object with mutator methods that is it allows changes on store string data but result is stored in new String object and returns that object
- Wrapper classes -they are - Integer, Character, Boolean, etc are immutable object without mutator methods that is they do not allow changes even new object.

Chapter 28

IO Streams

- In this chapter, You will learn
  - Storing and reading binary and text data from a flat file
  - Storing and reading Primitive Data from a flat file
  - Storing and reading objects from a flat file
    - Serialization and deserialization
  - Different ways to create objects in Java
  - Logical concatenation of streams
  - Default streams created in JVM
  - Importance of PrintStream class
  - Clear understanding on System.out.println() statement
  - Reading data from key board
  - Introduction to reader and writer classes
  - Importance of BufferedReader class
  - Creating and deleting files & directories from a program
- By the end of this chapter- you will be in a position to perform file based persistence operations.

Interview Questions

By the end of this chapter you answer all below interview questions

- Need of this chapter
- Define Stream
- Types of Streams
- Stream creation in Java

[Binary Streams]

- InputStream, OutputStream class hierarchy
- InputStream and OutputStream class methods
- Reading, writing, copying file data using FileInputStream and FileOutputStream
- Reading and writing data as primitive types using DataInputStream and DataOutputStream
- Reading and writing object using ObjectInputStream and ObjectOutputStream
- Understating Serialization and De-serialization
- Use of serialVersionUID variable, transient keyword in Serialization and Deserialization
- Serialization and Deserialization with IS-A and HAS-A relation objects
- Use of SequenceInputStream, BufferedInputStream, BufferedOutputStream
- Understating PrintStream class
- Default streams created in JVM
- Understanding System.out.println() method call structure
- How can we redirect STDOUTPUT data to a file using Sopln().
- Need of File class

[Character Streams]

- Need of Character Streams
- Reader and Writer classes hierarchy
- Reader and Writer classes methods
- Need of FileReader and FileWriter
- Need of InputStreamReader, and OutputStreamWriter
- Need of BufferedReader, and BufferedWriter
- How can we read one line at a time either from keyboard or from a file?
- How can we read input from end-user

So far you have learnt how to store data in variables. In this chapter you will learn how to store the data in files permanently.

Introduction – Need of this chapter

The basic idea of IOStreams chapter is

- storing and reading data from files and
- also reading data from keyboard

In this chapter, we cover the methods required for handling files and directories and also the methods for writing data to different destinations and reading data from different sources.

This chapter also shows you the *object serialization* mechanism that lets you store objects as easily as you can store text or numeric data in files.

First let us understand some basic terminology

Persistence media
The environment that allows us to store data permanently is called persistent media. We can store data permanently in three places.

- File
- Database
- Remote Computer (Socket)

Persistence
The process of storing data permanently in a persistence media is called persistence.

Persistence Logic
The logic that persist data in a persistence media is called persistence logic.
Ex: IOStreams based logic, JDBC based logic, Networking based logic

Persistence Technologies
The technology that provides API to develop persistence logic is called persistence technology.

Well know persistence technologies are
IOStreams – to persist data in flat files
JDBC, EJB, Hibernate – to persist data in DB
Networking – to persist data in remote computer.

In real-time projects the basic operation that we do using this chapter, is to perform persisting the data in files. Persisting the data means storing the data permanently.

Where can we store data or result permanently?
In persistence medias either in files or in Databases.

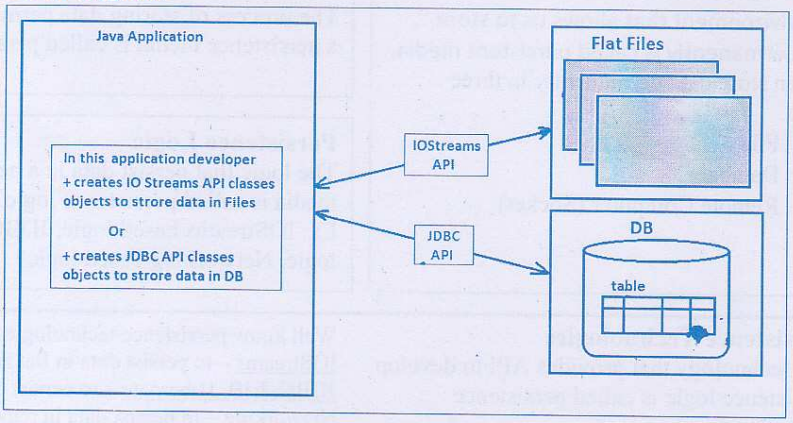
Storing data in variables and arrays is temporary. Data will be lost when a local variable goes out of scope or when the program terminates.

Programmers use files or DBs for long-term storage of large amounts of data. It is available even after termination of the program. We refer to data maintained on files as persistent data, because the data exists beyond the duration of program execution.

To store data in files and DBs SUN has given in-built API. We all need to do is creating the particular class object and calling methods on that object for storing and reading data from that persistence media.

IOStreams API is given to store and read data from files
JDBC API is given to store and read data from DBs.

Check below diagram



In this chapter we will learn storing data in files using IOStreams API.
In Advanced Java in JDBC chapter we will learn storing data in DB using JDBC API.

Note: In either of the approaches we just create object for predefined classes and we will call appropriate methods to store and read data from files and DBs.

So we can conclude IOStreams API and JDBC API is given not for developing Business logic rather they have been given to develop persistence logic.

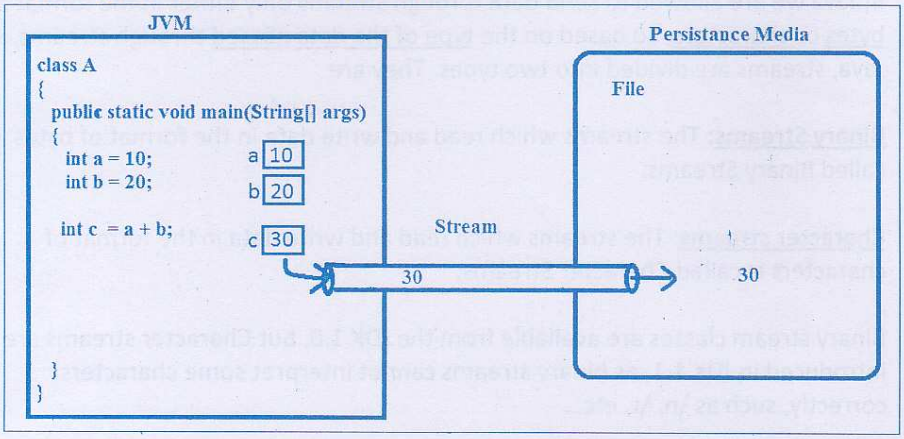
How a Java Application can store or read data from a file?
Using **stream** object.

Introduction to Streams
Stream is a logical connection between Java program and a file.

To store the data in the persistent media there should be a way to connect to persistent media from java application either physically or logically. Stream provides logical connection.

Stream can be defined as "*it is a continuous flow of data* between java program and persistence media".

Below diagram shows stream architecture



This diagram shows storing the result 30 in a file via a stream connection. In this chapter we learn how to create this stream object.

What is the direction of the stream flow?
Stream has a direction and its direction depends on the viewer's view. In Java the viewer is Java Application. If you look from Java application data is sending out from Java application.

Types of streams

Generally streams are divided into two types based on data flow direction.

InputStream: the stream that allows data to *come into* the java application from the persistent media is called Input Stream.

OutputStream: the stream that allows data to *send out* from the java application to be stored into the persistent media is called Output Stream.

Basically *InputStreams* are used to *read* data from a persistence media, and *OutputStreams* are used to *write or store* data in a persistence media from a java application.

Types of Java streams

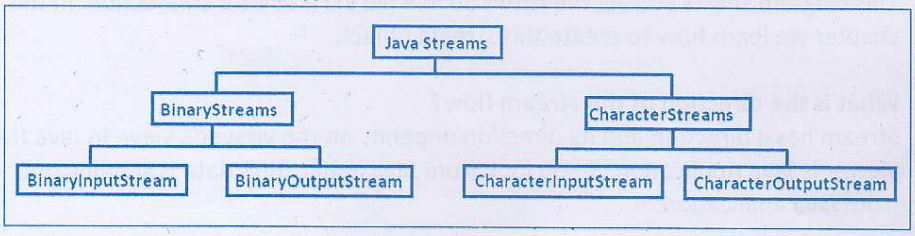
In Java we are allowed to send data through streams only either in the format of bytes or characters. So based on the type of the data passed through streams, in Java, streams are divided into two types. They are

Binary Streams: The streams which read and write data in the format of bytes is called Binary Streams.

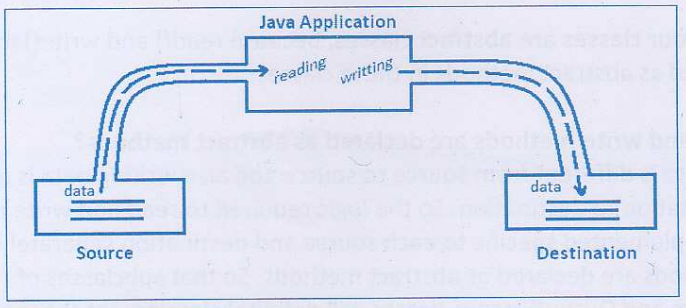
Character streams: The streams which read and write data in the format of characters is called Character Streams.

Binary stream classes are available from the JDK 1.0, but Character streams are introduced in JDK 1.1, as binary streams cannot interpret some characters correctly, such as \n, \t, etc...

Below diagram shows the types of JAVA streams.



What are the different sources and destinations to read and write data?
The data can be read from and written into different sources and destinations. From sources Java application reads data, and it writes data to destinations.



Below are the commonly available sources and destinations:
The **sources** can be a Keyboard, Mouse, File, Database, Socket (Computer), object, Array, String, StringBuffer.

The **destinations** can be a Monitor, File, Database, Socket (Computer), object, Array, String, StringBuffer.

How can we create stream in Java specific to source and destination?
In **java.io** package we have several classes to create input or output streams specific to source and destination.

So the technical definition of stream would be "Stream is a java object that allows reading and writing data to a persistence media".

What is the super class for all types of input and output stream classes?
InputStream is the super class for all *binary input stream* classes.
OutputStream is the super class for all *binary output stream* classes.

Reader is the super class for all *character input stream* classes.
Writer is the super class for all *character input stream* classes.

InputStream and Reader classes has **read()** method to read data from a source. In InputStream class it returns one byte at a time and in Reader class it returns one character at a time.

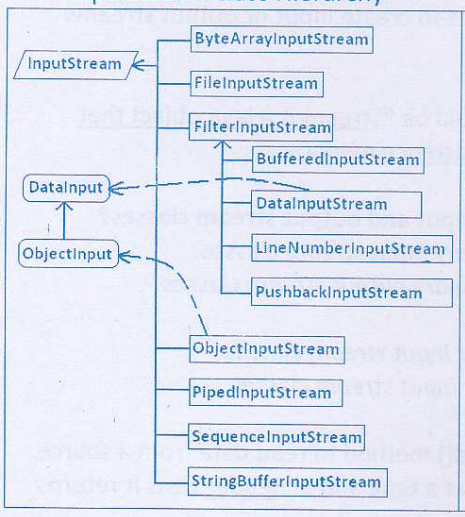
OutputStream and Writer classes has **write()** method to write data to a destination. In OutputStream class it writes one byte at a time and in writer class it writes one character at a time.

All above four classes are abstract classes, because read() and write() methods are declared as abstract methods in these classes.

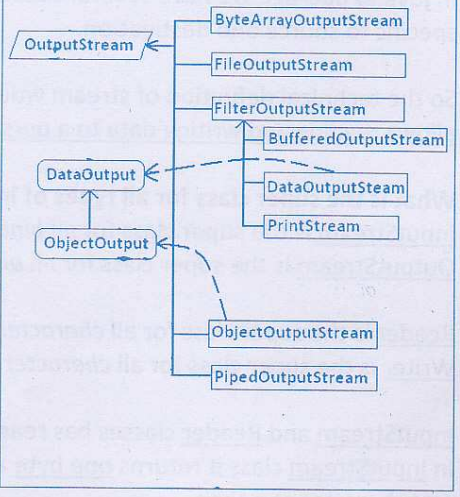
Why read and write methods are declared as abstract methods?
Reading data is different from source to source and also writing data is different from destination to destination. So the logic required to read and write data must be implemented specific to each source and destination separately. Hence these methods are declared as abstract methods. So that subclasses of InputStream and OutputStream classes will definitely implement these two methods.

InputStream and OutputStream class hierarchy:
For each type of source and destination in java.io package sun has given a separate class. Below diagram shows InputStream and OutputStream classes' hierarchy

InputStream class Hierarchy



OutputStream class Hierarchy



Explanation on above important classes

FileInputStream & FileOutputStream classes are used to read and write data as bytes from File. Basically these two streams are used as basic data source and destination for other streams.

DataInputStream & DataOutputStream classes are used to read and write data as primitive data. Basically these two streams are used to add capability to another input stream and output stream to read and write data as primitive types.

ObjectInputStream & ObjectOutputStream classes are used to read and write data as Object. Basically these two streams perform object Serialization and De-serialization.

BufferedInputStream & BufferedOutputStream classes are used to read and write data as buffers. Basically these two streams are used to improve reading and writing performance of other streams.

SequenceInputStream class is used to read data from multiple InputStreams sequentially.

PrintStream class is the most continent output stream class to write ta.

What is the procedure to perform read and write operations using streams?
Below is the basic procedure to perform IO operations

- Create stream class object based on source and destination
- call *read()* method to read data from the source
- call *write()* method to write data to the destination.

For instance, for performing IO operations on File we must create FileInputStream, FileOutputStream or FileReader, FileWriter stream classes object.

InputStream class methods:

Since InputStream is the super class for all binary input stream classes, it has below method with general implementation for all subclasses. As I said before no-arg read method is abstract method, it is implemented in its all subclasses.

InputStream class methods

Book Reading	InputStream Reading
Buy a book	Create stream object using its available constructors
1. checking available pages	Checking avaialbe bytes to read `public int available() throws IOException`
2. reading word by word	reading byte - by - byte `public abstract int read() throws IOException`
3. reading all words	reading all bytes `public int read(byte[] b) throws IOException`
4. reading important words	reading selected range of bytes `public int read(byte[] b, int off, int len) throws IOException`
5. Checking marksupported or not	checking is this stream supports marking `public boolean markSupported()`
6. marking important lines	marking current position in stream `public void mark(int readLimit)`
7. re-reading marking lines	placing the control at marked place `public void reset() throws IOException`
8. reading only required pages	skip reading given range of bytes `public long skip(long n) throws IOException`
9. closing the book	closing the stream `public void close() throws IOException`

What is the difference in read() and read(byte[]) methods return value?

read() method returns the actual byte value, where as read(byte[]) method returns number of bytes read from the stream into byte array.

OutputStream class methods:

Since OutputStream is the super class for all binary output stream classes, it has below method with general implementation for all subclasses. As I said before "int parameter" write method is abstract method, it is implemented in its all subclasses.

OutputStream class Methods

storing data in our mind	writing data to an output stream
writing data word by word	writing one by at time to an outputstream public abstract void write(int i) throws IOException
writing all words	writing stream of bytes to an outputstream public void write(byte[] b) throws IOException
writing important words	writing give range of stream of bytes to an outputstream public void write(byte[] b, int off, int len) throws IOException
store all words in permanent mind	flush all bytes to destination from outputstream public void flush() throws IOException
close mind	close outputstream public void close() throws IOException

Note: flush() method must be called for BufferedOutputStream, and BufferedWriter. Else data will not be stored in destination. For all other streams calling flush method is optional. But it is good practice if we call flush() before stream close.

Let us start developing applications for reading and writing data from file.

Application #1: Reading the data from a file
FileInputStream class must be used to read data from a file.

It is two steps process.

1. Create FileInputStream class object by using its available constructors
2. Then call read() method on this object till control reach end of the file. Means read() method must be called in loop, because it returns only one byte at a time. Hence it must be called in loop for each byte available in the file till it returns "-1". It returns -1 if control reached end of the file.

FileInputStream class has below **constructors** to create its Object.

1. public FileInputStream(String name) throws FileNotFoundException.
2. public FileInputStream(File file) throws FileNotFoundException
3. public FileInputStream(FileDescriptor fd)

Rule: To create FileInputStream class object the file must be existed with the passed argument name, else this constructor throws **java.io.FileNotFoundException.**

FileInputStream class constructor throws this exception in below situations

- If the file is not existed with the passed name.
- Passed file name is a directory rather than is a regular file.
- If file does not have reading permissions.

Below program shows reading data from a file
First create a file with the name **test.txt** and store data **abcd** in this file.

Below **is the sample code** to read data from this file

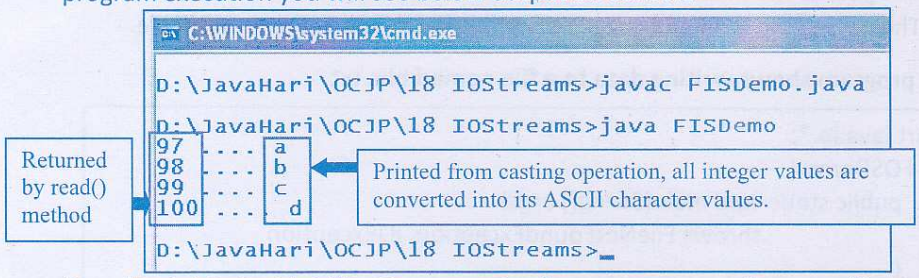
```
//1. Creating stream object connecting to test.txt file
 FileInputStream fis = new FileInputStream("test.txt");

//2. Reading all bytes from the file till control reaches end of the file.
      int i = fis.read() ; <- this statement reads only first character "a".
printing returned character
      System.out.println(i); <- It prints 97, 'a' ASCII integer number.

Converting it to character to print 'a'
      System.out.println((char)i); <- now it prints 'a'
```

Below is the complete program to read complete data from abc.txt file.

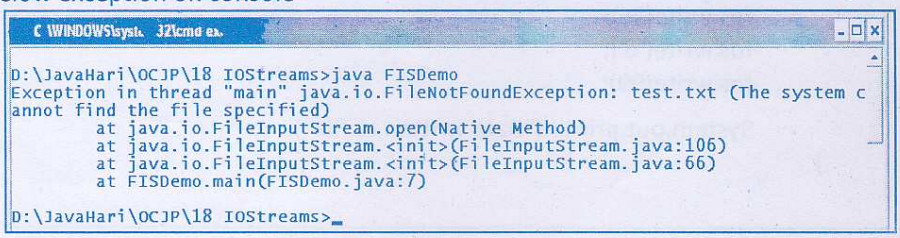
```
import java.io.*;

class FISDemo
{
        public static void main(String[] args) throws FileNotFoundException, IOException
        {
                FileInputStream fis = new FileInputStream("test.txt");

                int i;
                while( ( i = fis.read() ) != -1)
                {
                        System.out.println(i + " .... ");
                        System.out.print((char)i);

                }
        }
}
```

Find the difference in executing above program with and without file

Case #1: Assume **test.txt** file is available with the data **abcd**. After the above program execution you will see below output on console



Case #2: delete **test.txt** file, then execute above program again; you will see below exception on console



Application #2: Writing data to a file
FileOutputStream class must be used to write data to a file.

It is also two steps process

1. Create FileOutputStream object by using its available constructors
2. Then call write() method to write either single or multiple bytes.

FileOutputStream class has below **constructors** to create its Object.

1. public FileOutputStream(String name) throws FileNotFoundException
2. public FileOutputStream(File name) throws FileNotFoundException.
3. public FileOutputStream(String name, boolean append) throws FileNotFoundException.
4. public FileOutputStream(File file, boolean append) throws FileNotFoundException

Note*:** Unlike FileInputStream, FileOutputStream class constructor ***does not throw FileNotFoundException if file is not existed, instead it creates empty file with the given name. Then it writes the given data to that file.

It throws above exception in the below cases.

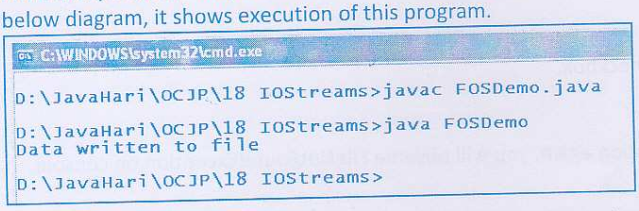
- File doesn't exist but can't be created.
- The passed file name is a directory rather than a regular file.
- The passed file is a Read Only File, writing permissions aren't available.

Below program shows writing data to a file named bbc.txt.

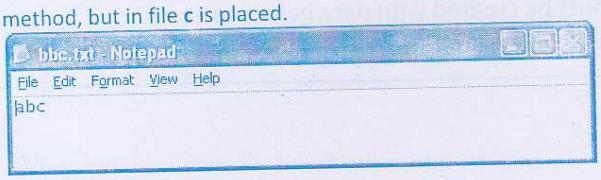
```
import java.io.*;
class FOSDemo {
        public static void main(String[] args)
                        throws FileNotFoundException, IOException
        {
                FileOutputStream fos = new FileOutputStream("bbc.txt");

                fos.write('a');
                fos.write('b');
                fos.write(99);

                System.out.println("Data written to file");
        }
}
```

Follow below test cases to run this application:

Case #1: execute above application without creating **bbc.txt** file. It is executed fine without FNFE exception. FileOutputStream class creates this file when its object is created. Check below diagram, it shows execution of this program.

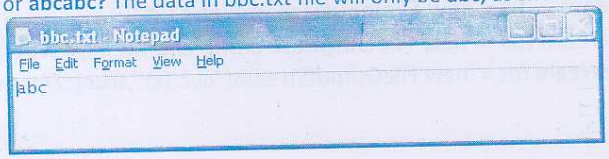


Below is the bbc.txt file with its data. Notice its data, in the application we passed **99** to **write()** method, but in file **c** is placed.



So we can conclude that **write()** method always writes **ASCII character data** and **read()** method always returns **ASCII integer data**.

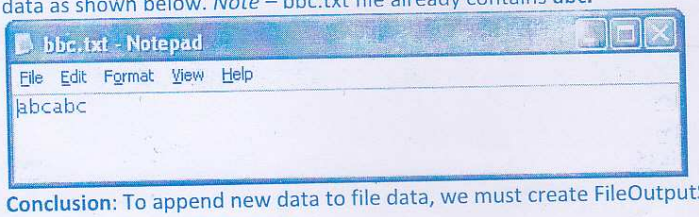
Case #2: Execute this application again, and observe bbc.txt file data. Find out whether it is **abc** or **abcabc?** The data in bbc.txt file will only be **abc,** as shown below



Actually FileOutputStream ***by default overrides the file data*** with new data.

Case #3:
In the above program, replace **FileOutputStream fos = new FileOutStream("bbc.txt");** with
FileOutputStream fos = new FileOutStream("bbc.txt", true);

Then compile and execute the program again. Now observe bbc.txt file, you can find **abcabc** data as shown below. *Note* – bbc.txt file already contains **abc.**



Conclusion: To append new data to file data, we must create FileOutputStream class object with second argument as **true**.

Case #4: Change bbc.txt file permission to read-only.
Setting read-only permission:

1. right click on file
2. click properties
3. Check *Read –only* checkbox.
4. Click Ok

Now execute above application again, you will observe FileNotFoundException on console.

Application #3: File copy
Source file: **abc.txt** file it must be created with data before execution.
Destination file: **cbc.txt** file it will be created by FOS object.

```
import java.io.*;

public class FileCopy
{
	public static void main(String[] args)
		throws FileNotFoundException, IOException
	{
		FileInputStream	fis = new FileInputStream("abc.txt");
		FileOutputStream fos =  new FileOutputStream("cbc.txt"); //overrides
		//FileOutputStream fos =  new FileOutputStream("cbc.txt",true); //appends

		int i;
		while( ( i = fis.read() ) != -1)
		{
			fos.write(i);
		}

		System.out.println("Data has written");
	}
}
```

Below application shows copying file data by reading filenames at runtime. It also shows handling exception properly.

```
//FileCopy.java
import java.io.*;
public class FileCopy
{
        //This method has reusable code, so should not handle exception.
        public static void copyFile(String srcFile, String destFile)
                                throws FileNotFoundException, IOException
        {
                FileInputStream     fis = new FileInputStream(srcFile);
                FileOutputStream    fos =  new FileOutputStream(destFile);

                int i;
                while( ( i = fis.read() ) != -1)
                {
                        fos.write(i);
                }

                System.out.println("Data has written to "+destFile);
        }
}

//It is user application, so it must handle exceptions to print user understandable exception messages
//Test.java
class Test{
        public static void main(String[] args) {
                try{
                        FileCopy.copyFile(args[0], args[1]);
                }
                catch(ArrayIndexOutOfBoundsException aeiob){
                        System.out.println("Error: Please pass source and destination file names");
                        System.out.println("Usage: java Test abc.txt bbc.txt");
                }
                catch(FileNotFoundException fnfe){
                        System.out.println("Error: The given files "+ args[0] +" , " + args[1] + "
                                are not found, make sure they are avilable in the current path");
                }
                catch(IOException ioe)
                {
                        System.out.println("Error: Reading or writting failed");
                        e.printStackTrace();
                }
        }
}
```

Limitation of FIS and FOS

FIS and FOS allow us to read and write data only in the format of bytes. It is not possible to read or write data in the format of Primitive data or objects.

Solution

We should use *filter input stream* and *filter output stream* classes in connection with *FileInputStream* and *FileOutputStream* classes.

FilterInputStream and FilterOutputStream

Filter classes are used to improve the performance or used to add more functionality to underlying InputStream and OutputStream.

Note: From above diagram we can conclude

- The filter connected to InputStream is called FilterInputStream.
- The filter connected to OutputStream is called FilterOutputStream.
- Filters cannot connect to source or destination directly, instead they can only be connected to another *InputStream* or *OutputStream*
- So, all *FilterInputStream* and *FilterOutputStream* classes contain constructor to take other *InputStream* and *OutputStream* object as argument which it uses as its basic source of data for reading and writing.

DataInputStream and **DataOutputStream**

These classes are used to read and write the data as primitive type from the underlying InputStreams and OutputStreams. These classes have special methods to perform reading and writing operations as primitive data type.

DataInputStream

DataInputStream is a sub class of *FilterInputStream* and *DataInput* interface. It implements below methods from the DataInput interface for reading bytes from a binary input stream and convert them in to corresponding java primitive types.

DataOutputStream

DataOutputStream class is a subclass of FilterOutputStream and DataOutput interface. It implements below methods from the DataOutput interface for converting data from any of the java primitive types to a stream of bytes and writing these bytes to a binary output stream.

DataOutputStream methods	DataInputStream methods
• public void writeByte(byte b) • public void writeShort(short s) • public void writeInt(int i) • public void writeLong(long l) • public void writeFloat(float f) • public void writeDouble(double d) • public void writeChar(char ch) • public void writeBoolean(boolean b) • public writeUTF(String s) • public writeBytes(String s)	• public byte readByte() • public short readShort() • public int readInt() • public long readLong() • public float readFloat() • public double readDouble() • public char readChar() • public boolean readBoolean() • public String readUTF() • public String readLine()

Rule #1: To read data as primitive types using above readXxx() methods, the data must be written to the file as primitive types using writeXxx() methods.

Rule #2: readXxx methods must be called in the same order in which the writeXxx methods are called otherwise wrong value will be returned or EOFException is raised.

Application #4: Below application demonstrates writing data as primitive type using DataOutputStream class.

```
// DOSDemo.java
import java.io.*;
class DOSDemo
{
        public static void main(String[] args) throws Exception
        {
                FileOutputStream fos = new FileOutputStream("data.txt");
                DataOutputStream dos = new DataOutputStream(fos);

                dos.writeInt(97);
                dos.writeFloat(3.14f);
                dos.writeChar('a');
                dos.writeBoolean(true);
                dos.writeUTF("Hari");

                System.out.println("Data Written to file");
        }
}
```

After compilation and execution in current working directory you can find file "data.txt". Check data.txt file size.

To check file size - "right click on file -> click Properties".
You can find Size as 17 bytes. It is an addition of all primitive types
(int + float + char + boolean + string) = (4 + 4+ 2 + 1 + 6).

From this program we can prove boolean takes 1 byte, the minimum memory location size in Java is 1 byte. For string data including double quotes ("") JVM provides 1 byte for each character to it in file.

Applicaton #5: Below application demonstrates reading data as primitive type using DataInputStream class.

```
import java.io.*
class DISDemo
{
	public static void main(String[] args) throws Exception
	{
		FileInputStream fis = new FileInputStream("data.txt");
		DataInputStream dis = new DataInputStream(fis);

		int		i1	=	dis.readInt();
		char		f1	=	dis.readFloat();
		char		ch2	=	dis.readChar();
		boolean		bo1	=	dis.readBoolean();
		String		s1	=	dis.readUTF();

		System.out.println(i1);
		System.out.println(f1);
		System.out.println(ch2);
		System.out.println(bo1);
		System.out.println(s1);
	}
}
```

Test case #1: Execute above application by changing readXxx() methods calling order. Then check whether you will get right values or not.

Test case #2: Execute above application by calling different readXxx() methods. Then check whether you will get right values or not.

Actually readXxx() method read number of bytes from file based on the data type. For example if we call readInt() method it reads 4 bytes from file. So if four bytes are available in file program executed fine, else program execution terminates with EOFException.

Limitation of DIS and DOS

Using DOS and DIS classes we cannot read and write objects from persistence media. They have only capability to read data up to only primitive data types.

Solution

To read and write objects we must use ObjectInputStream and ObjectOutputStream.

ObjectInputStream and **ObjectOutputStream**

These two classes are used to store object's state permanently in files or to send to remote computer via network.

ObjectOutputStream class is a subclass of ObjectOutput interface. It implements below method to write object to underline output stream

```
public void writeObject(Object obj) throws IOException
```

ObjectInputStream class is a subclass of ObjectInput interface. It implements below method to read object from underline input stream

```
public Object readObject()throws IOException
```

Rule: To write or send object to external world it must be of type **java.io.Serializable** interface. It means this object's class must be a subclass of java.io.Serializable interface. Else writeObject() method throws **java.io.NotSerializableException.**

Constructors to create above classes objects

Like DIS and DOS, OIS and OOS are also cannot connect to source and destination directly. So their constructors take other input and output stream class objects.

```
public ObjectInputStream(InputStream in)
public ObjectOutputStream(OutputStream out)
```

Applcation #6: Below application shows writing Bank class object to file

```
//Bank.java
public class Bank implements java.io.Serializable
{
	static double minBalance = 5000;

	private long accNo;
	private String accHName;
	private String username;
	private transient String password;
	private transient double balance;

	public void setAccNo(long accNo)
	{
		this.accNo = accNo;
	}
	public long getAccNo()
	{
		return accNo;
	}
	public void setAccHName(String accHName)
	{
		this.accHName = accHName;
	}
	public String getAccHName()
	{
		return accHName;
	}

	public void setUsername(String username)
	{
		this.username = username;
	}
	public String getUsername()
	{
		return username;
	}
```

```
    public void setPassword(String password)
    {
        this.password = password;
    }
    public String getPassword()
    {
        return password;
    }
    public void setBalance(double amt)
    {
        this.balance = this.balance + amt;
    }
    public double getBalance()
    {
        return balance;
    }

    public String toString()
    {
        return "accNo: "+accNo +"\n" +
                    "accHName: "+accHName +"\n" +
                    "username: "+username+"\n" +
                    "password: "+password+"\n"+
                    "balance: "+balance+"\n"+
                    "minBalance: "+minBalance+"\n";
    }
}
```

```
// BankTransaction.java  -> shows object writing
import java.io.*;

class BankTransaction {
	public static void main(String[] args) throws Exception
	{
		//creating bank object
		Bank acc1 = new Bank();

		//setting bank object state
		acc1.setAccNo(1);
		acc1.setAccHName("Hari");
		acc1.setUsername("Hari Krishna");
		acc1.setPassword("Naresh technologies");
		acc1.setBalance(999999999);

		//printing bank object state
		System.out.println(acc1);

		//creating OOS object
		ObjectOutputStream oos = new ObjectOutputStream(
				new FileOutputStream("BankAccountsinfo.ser"));

		//writing bank object state to file
		oos.writeObject(acc1);

		System.out.println("Object written to file");
	}
}
```

```
// BankUser.java -> shows object reading
import java.io.*;

class BankUser
{
	public static void main(String[] args) throws Exception
	{
		//creating OIS object
		ObjectInputStream ois = new ObjectInputStream(
				new FileInputStream("BankAccountsinfo.ser"));

		//casting returned object to Bank type
		Bank accDetails = (Bank) ois.readObject();

		//printing Bank object data
		System.out.println(accDetails.getAccHName() + " details");
		System.out.println(accDetails);
	}
}
```

Explain what is the actual functionality of OOS and OIS?
Performing Serialization and deserialization

Serialization
Serialization is the process of converting objects into stream of bytes and sending them to underlying OutputStream. Using Serialization we can store object state permanently in a destination, for example file or remote computer.

Serialization operation is performed by writeObject () method of ObjectOutputStream.

Deserialization
Deserialization is the process of converting stream of bytes into original object. Deserialization operation is performed by readObject () of ObjectInputStream.

Rule on Serialization

Only java.io.Serializable type objects are serialized. Serializable is a marker interface, it doesn't have any methods. It provides special permission or identity to JVM to serialize object.

If the given object isn't of type Serializable interface then writeObject() will throw an unchecked exception called java.io.NotSerializableException.

The below diagram explains serialization and deserialization operations.

How can readObject() method load class of the serialized object?
It gets that serialized object's class name from serialized file.

Q) What is the functionality of writeObject and readObject methods?
In Serialization writeObject() method stores object state in the file with the below information.

1. Its class name
2. Current .class file's serialVersionUID
3. Its non-transient variable names and their data type
4. Those variables current modified values.

In deserialization readObject() method creates object with the state available in the file as shown below

1. readObject() method first reads class name from serialized file and loads that class by using Class.forName(). While loading the class static variables get memory with originally assigned values and SBs are executed.
2. Then it compares serialVersionUID of current loaded class with the serialVersionUID that is stored in serialized file
   a. If both are not same it terminates deserialization process by throwing exception
   java.io.InvalidClassException
   b. If both values are same, it creates object with the current loaded class non-static variables without using new keyword and constructor.
3. Then readObject method populates values
   a. from serialized file for the non-transient variables and
   b. default values for transient variables and also for the non-static variables those are not avaiable in serialized file.

Rules in deserialization:

" In deserialization the object class's ".class" file must be available, else readObject() method throws java.lang.ClassNotFoundException.

" In deserialization we must use the same version of .class file that is used in serialization, else object is not deserialized.

Let us understand Serialization with *IS-A* and *HAS-A* relations

Serialization with IS-A relation

If super class is deriving from Serializable interface, then all its subclasses are also serializable types. They no need to be sub classing from Serializable interface again explicitly.

In this case, if we serialize subclass object, super class all non-transient instance variables are also serialized. In deserialization all subclass and super class instance variable's values are populated from serialized file.

For Example:

```
//A.java
class A implements java.io.Serializable{
        int x;

        A(){
                x = 50;
                System.out.println("x is initialized with 50");
        }
}
//B.java
class B extends A{
        int y;

        B(){
                y = 60;
                System.out.println("y is initialized with 60");
        }

        public String toString(){
                return "x: "+x + "\ny: "+y;
        }
}
```

```
//OOSDemo.java
import java.io.*;

public class OOSDemo{
        public static void main(String[] args)throws Exception{
                OOS oos = new OOS(new FOS("test.ser"));
```

```
            B b = new B();
            b.x = 70;
            b.y = 80;

            System.out.println("x,y values are changed to 70, 80");

            oos.writeObject(b);
            System.out.println("B object is written to file");
        }
    }
```

Compilation and execution

```
>javac A.java
>javac B.java
>javac OOSDemo.java
>java OOSDemo
    x variable is initialized with 50
    y variable is initialized with 60
    x, y variables are changed to 70,80
    B object is written to file
```

```
//OISDemo.java
import java.io.*;

public class OISDemo{
    public static void main(String[] args)throws Exception{
        OIS ois = new OIS(new FIS("test.ser"));
        Object obj = ois.readObject();
        System.out.println(obj);
    }
}
```

```
>javac OISDemo.java
>java OISDemo
    x: 70
    y: 80
```

As you can observe in the above output, in deserialization B class object is populated with modified x and y variables values.

Q) If we remove "implements java.io.Serializable" from A class declaration, will B object be written to file?
A) No, it leads to exception java.io.NotSerializableException

Q) If we remove "implements java.io.Serializable" from A class declaration, and we added to B class declaration, will B object be written to file?
A) Yes, but only B class non-transient variable's values are stored in serialized file.

In deserialization, super class A's instance variable x is initialized by constructor.
So, if we run OISDemo with above change the output will as shown below

```
> java OISDemo
        x variable is initialized with 50
        x: 50;
        y: 80;
```

FAQ: Is super class must be Serializable to serialize subclass object?
A) No.

Serialization with HAS-A relationship:
Q) To serialize an object, will its internal object also must be Serializable?
A) Yes, else object will not be serialized

For Example:

```
class Student implements java.io.Serializable{
        String name = "Hari Krishna, Naresh Technologies";
        Address add = new Address();
}
class Address{
        String city = "Hyd";
}

------
        Student s = new Student();
        oos.writeObject(s);
-----
```

In the above case, writeObject() method throws java.io.NotSerializableException, because Address is not sub classing from Serializable interface.

Solutions

There are three solutions for the above problem

1. Declare "add" as transient
   In this case, deserialized Student object's add variable will have value null, so Student will not have address (*not good*)
2. Create subclass of Address and use this subclass object
   In this case, deserialized Student object will have Address with null value, but it is not with current Student address value.
3. Customize serialization with below private methods
   private void writeObject(ObjectOutputStream oos)throws IOException
   private void readObject(ObjectInputStream ois)throws IOException

Customizing Serialization

If we want to write Student object with its current address object values we must customize serialization by using below two private methods

```
private void writeObject(ObjectOutputStream oos)throws IOException{
        // here you write address object values to file
        //using writeXxx(xxx) methods
}

private void readObject(ObjectInputStream ois)throws IOException{
        // here you read address object values from file
        // using readXxx() methods and store those values
        // in newly created Address object, and set this
        // Address object to Student's Address referenced variable "add"
}
```

Below is the changed Student class to write address object state

```
class Student implements java.io.Serializable {
    int rollNum;
    String sname;
    transient Address add;
    Student(int rollNum, String sname, Address add){
        this.rollNum    =    rollNum;
        this.sname      =    sname;
        this.add        =    add;
    }
```

```
    private void writeObject(ObjectOutputStream oos){
        try{
            oos.defaultWriteObject();

            //writing current Student's Address
            oos.writeInt(add.hno);
            oos.writeUTF(add.city);
        }
        catch(IOException e){
            e.printStackTrace();
        }
    }
    private void readObject(ObjectInputStream ois){
        try{
            ois.defaultReadObject();

            //reading current Student's Address state
            int    hno = ois.readInt();
            String city = ois.readUTF();

            add = new Address(hno, city);
        }
        catch(IOException e){
            e.printStackTrace();
        }
        catch(ClassNotFoundException e){
            e.printStackTrace();
        }
    }
    public String toString(){
        return      "rollNum:"  + rollNum              + "\n" +
                    " sname:    "      + sname         + "\n" +
                    " hno:"     + add.hno    + "\n" +
                    " city: "           + add.city;
    }
}
```

```
class Address{
        int hno;
        String city;

        Address(int hno, String city){
                this.hno = hno;
                this.city = city;
        }
}
```

Functionality of private writeObject and readObject methods:

- These two methods are automatically called by JVM
  - *private writeObject* method is called from *oos.writeObject()* method by passing current *OOS object* reference as argument. So that in this private writeObject method we can write data to the same underlying file (serialized file).
  - *private readObject* method is called from *ois.readObject()* method by passing current *OIS object* reference as argument. So that in this private readObject method we can read data from the same underlying file (serialized file).
- Inside these two methods we must call "*oos.defaultWriteObject()*"; and "*ois.defaultReadObject()*" methods to perform serialization and deserialization on the given object. Otherwise object's data written to the file is not read instead we get default values of that object's non-static variables.
- next to these two methods call we must call writeXxx() and readXxx() methods to write and read non-Serializable object data to the same file individually

Q) What is a marker interface?

An interface that is used to check / mark / tag the given object is of a specific type to perform a special operations on that object, is called marker interface. Maker interface will not have methods, they are empty interfaces.

Q) What is the need of marker interface?

It is used to check, whether we can perform some special operations on the passed object.

For example,
If class is deriving from java.io.Serializable, then that class object can be Serializable, else it cannot.If class is deriving from java.lang.Cloneable, then that class object can be cloned, else it cannot.

In side those particular methods, that method developer will check that the passed object is of type the given interface or not by using instanceof operator.

For example, in writeObject() method we will find below code

```
public class ObjectOutputStream extends OuputStream{
        public void writeObject(Object obj){
                if (obj instance of Serializable){
                                ------------
                                ------------
                                ------------
                }
                else{
                        throw new
                        NotSerializableException(obj.getClass().getName());
                }
        }
}
```

Q) Can we write custom marker interfaces?
A) Yes. Write an empty interface, and use in the method to check the passed object is of type that interface or not by using "instanceof" operator.

Q) Is marker interface an empty interface?Yes
Q) Is empty interface a marker interface?
Depends, if it is used in instanceof operator condition then it called marker else it is called empty interface.

List of know predefined marker interfaces

1. java.lang.Cloneable
2. java.io.Serializable
3. java.util.RandomAccess
4. java.rmi.Remote
5. javax.servlet.SingleThreadModel
6. java.util.EventListener

SequenceInputStream

This class is used to read data from multiple input streams in sequence from the first one until end of file is reached, whereupon it reads from the second one, and so on, until end of file is reached on the last of the contained input streams.

Basically it represents the logical concatenation of other input streams.
It has below two constructors to create its object

1. public SequenceInputStream(InputStream s1, InputStream s2)
   This constructor allows us to create SequenceInputStream class object only with two InputStreams.

2. public SequenceInputStream(Enumeration<? extends InputStream> e)
   This constructor allows us to create SequenceInputStream class object with 'n' number of InputStreams.

Below applications shows reading data from two InputStreams (two files)

```
//SequenceBufferdIOSDemo.java
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.SequenceInputStream;

/**
 * Demonstrates BufferedInputStream/BufferedOutputStream/SequenceInputStream.
 */
public class SequenceBufferdIOSDemo {
        public static void main(String args[]) throws IOException {

                FileInputStream f1 = new FileInputStream("abc.txt");
                FileInputStream f2 = new FileInputStream("bbc.txt");

                SequenceInputStream sis = new SequenceInputStream(f1, f2);
                int data;

                while( ( data=sis.read() ) != -1){
                        System.out.println((char)data);
                }
                bos.flush();
        }// End main()
}// End class
```

Q) Write a program to read data from three files by using SIS
Clue: In this application development we must use second constructor.

PrintStream class
This class is a filter output stream class used to write data *as it is in the given format*.
That means unlike *FileOutputStream*, this class writes 97 as 97 rather as 'a'.

It is the most convenient class in writing data to another output stream, because of below three reasons

1. Unlike other output stream's *write()* method, this class *write()* methods does not throw IOException
2. Calling *flush()* method is optional, it is called automatically.
3. In addition to default *write()* methods it defined new methods called *print* and *println*. These two methods are overloaded to print all types of java data type values including object *as it is in those format*. Below are the overloaded print and println methods list

```
public void print(int x);
public void print(long x);

public void print(float x);
public void print(double x);

public void print(char x);
public void print(boolean x);

public void print(char[] x);
public void print(String x);

public void print(Object x);
```

```
public void println();

public void println(int x);
public void println(long x);

public void println(float x);
public void println(double x);

public void println(char x);
public void println(boolean x);

public void println(char[] x);
public void println(String x);

public void println(Object x);
```

Note:
PrintStream class object is created in System class using a static variable called **out.** So we can access above methods from our class using the System class's PrintStream object as shown below.

```
System.out.print("abc");      System.out.print(30);
System.out.println("bbc");    System.out.println(40);
```

In the above first two statements we called *String* parameter print and println methods, and in the second two statements we called *int* parameter print and println methods

Q) How these methods are printing the given data *as it is in* the given format?
A) In side these methods the given data is converted into *String* data by calling "*String.valueOf*()" method, then the returned string data is printing on destination.

What is the difference between print and println methods?

print method places cursor in the same line after printing data, so that next coming output will be printed in the same line. But **println method** places cursor in the next line after printing data so that next coming output will be printed in the next line.

What is the output from below program?

```
class Test {
        public static void main(String[] args) {
                System.out.print("A");
                System.out.println("B");
                System.out.println("C");
        }
}
```

O/P
====

Rule #1: We cannot call print method without passing arguments because we do not have no-arg print() method it leads to *CE: cannot find symbol*. But we can call println method without passing arguments because we have no-arg println() method, it prints new line.

What is the output from below program?

```
class Test {
        public static void main(String[] args) {
                System.out.print("A");
                System.out.println();
                System.out.print("B");
                System.out.print();
                System.out.print("C");
        }
}
```

O/P
====

Rule #2: We cannot call print() or println() methods by passing ***null*** literal directly. It leads to CE: Ambiguous error. Because two methods are overloaded with String and Char[] parameters. Since these two parameters are siblings it leads to above compile time error. But we can pass null via referenced variable, then that referenced variable type parameter method is invoked. Object, String parameter methods prints null *where as* Char[] parameter method throws NPE.

What is the output from below program?

```
class Test {
        public static void main(String[] args){
                System.out.print(null);
                System.out.println(null);
        }
}
```

```
class Test {
        public static void main(String[] args) {
                String s = null;
                System.out.print(s);

                char[] ch = null;
                System.out.println(ch);
        }
}
```

What is the difference between println(Object) and writeObject(Object) methods?
writeObject(Object) method serializes the given object and stores its state in underlying output stream. But where as println(Object) method does not perform serialization instead it prints the passed object information that is returned by its *toString* method.

It means println(Object) method internally calls *toString()* on given object to print the given object information. If toString method is not defined in the passed object's class, then it is called from java.lang.Object, it is originally defined in Object class. toString method in Object class returns current object's "classname@hashcode in hexadecimal string format".

What is the output from below programs?

Case 1: toString method is not overridden in Example

```
class Example{
        int x = 10, y = 20;
}


class Test {
        public static void main(String[] args) {
                Example e = new Example();
                System.out.print(e);
        }
}
```

Case 2: toString method is overridden in Example

```
class Example{
        int x = 10, y = 20;
        public String toString(){
                return "Example class object" ;
        }
}
class Test {
        public static void main(String[] args) {
                Example e = new Example();
                System.out.print(e);
        }
}
```

Case 3: toString method is overridden with Example class object state

```
class Example{
        int x = 10, y = 20;
        public String toString(){
                return x + " ... " + y;
        }
}
class Test {
        public static void main(String[] args) {
                Example e = new Example();
                System.out.print(e);
        }
}
```

Case 4: Printing String class object

```
class Example{
        int x = 10, y = 20;
        public String toString(){
                return x +" ... " +y;
        }
}
class Test {
        public static void main(String[] args) {
                Example e = new Example();
                System.out.print(e);

                String s = "abc";
                System.out.print(s);
        }
}
```

Can developer create PrintStream class object explicitly?
Yes, it has below constructors to create its object.

```
public PrintStream(OutputStream out)
public PrintStream(OutputStream out,  boolean autoFlush)

public PrintStream(String fileName)
        throws FileNotFoundException

public PrintStream(File file)
        throws FileNotFoundException
```

These two constructors are given in Java 5 version to pass file name directly without creating FileOutputStream class object explicitly.

How can we print data in file using PrintStream class methods?
Yes, create PrintStream class object connecting to that file by using above constructors

Jdk1.4 based PrintStream object creation to connect to file

```
PrintStream ps = new PrintStream(new FileOutStream("abc.txt"));
```

Jdk1.5 based PrintStream object creation to connect to file

```
PrintStream ps = new PrintStream("abc.txt");
```

Below application shows storing data in file using PrintStream class

```
import java.io.*;
class PrintStreamDemo
{
        public static void main(String[] args) throws FileNotFoundException
        {
                //PrintStream object creation connecting to abc.txt file
                PrintStream ps = new PrintStream("abc.txt");

                //storing data in abc.txt file using explicit PrintStream object
                ps.print("A");
                ps.println("B");
                ps.println("C");

                System.out.print("Data written to abc.txt file");

                //printing data on console using implicit PrintStream object
                System.out.print("A");
                System.out.println("B");
                System.out.println("C");
        }
}
```

Difference between print(int) and write(int) methods?

print or println methods print the given data as it is in either on console or on file, it does not convert given number into *bytes*. But write method converts given number into bytes and writes those *bytes* into file. Then file editor showing those bytes as its corresponding ASCII character. Check below program and output

```
import java.io.*;
class PrintStreamDemo {
        public static void main(String[] args) throws Exception{

                FileOutputStream fos = new FileOutputStream("abc.txt");
                PrintStream ps = new PrintStream("bbc.txt");

                //storing data in abc.txt file using FileOutputStream object
                fos.write(97);
                fos.write(98);
                fos.write(99);
                System.out.print("Data written to abc.txt file");

                //storing data in bbc.txt file using explicit PrintStream object
                ps.print(97);
                ps.print(98);
                ps.print(99);
                System.out.print("Data written to bbc.txt file");

                //printing data on console using implicit PrintStream object
                System.out.println(97);
                System.out.println(98);
                System.out.println(99);
        }
}
```

Written to abc.txt file in same line

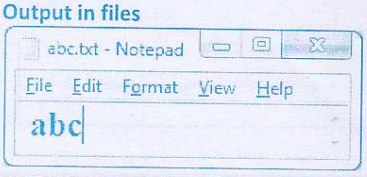
Printed in same line in bbc.txt file

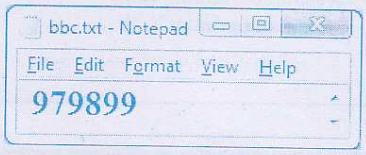
Printed in different lines on console

Output on console

```
d:\Nareshit\JavaHari\OCJP\IOStreams:javac PrintStreamDemo.java

d:\Nareshit\JavaHari\OCJP\IOStreams:java PrintStreamDemo
Data written to bbc.txt file
Data written to bbc.txt file
97
98
99

d:\Nareshit\JavaHari\OCJP\IOStreams:_
```

Output in files





Default Streams created in JVM

In JVM by default three streams are created by the class System to read data from keyboard and to write data to console (monitor). These stream objects are hold by static final referenced variables in System class. They are

- public static final InputStream **in**;
- public static final PrintStream **out**;
- public static final PrintStream **err**;

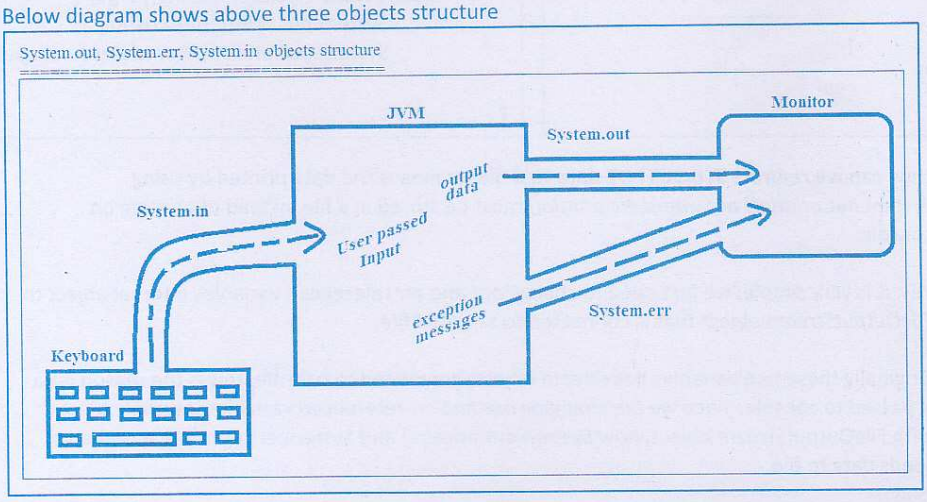
in variable holds BufferedInputStream class object that is connected to keyboard
out and **err** variables holds PrintStream class object that is connected to console.

Since these variables are defined as static variables we can access these objects hold by these variables from our class by using class name System like, **System.in**, **System.out**, **System.err**.

What is the difference between System.out and System.err?

JVM internally uses System.out to write output to console, and it uses System.err to write exceptions to console.

Below diagram shows above three objects structure



System.in object connects to keybord

Means to read data from keyboard we must use below statement

int data = System.in.read();

System.out and **System.err** objects connect to console.

Means to print data on console we must use below statements

System.out.println() or *System.err.println()*

Explain System.out.println() statement

- **System** is a predefined class defined in java.lang package
- **out** is a static referenced variable of type PrintStream class created in System class to hold PrintStream class object that is pointing to console.
- **println()** is a non static method defined in PrintStream class to print data.

So to call println() method from our class using PrintStream object created in System class we must use the statement **System.out.println()**;

Check below diagram, it shows the linking of all System, PrintStream and user class.

```
public class PrintStream
{
        public void print(String msg)
        {
                ----------
        }
        public void println(String msg)
        {
                ----------
        }
}
```

```
public class System{
        public static final InputStream in;
        public static final PrintStream out;
        public static final PrintStream err;
}
```

```
public class Test{
        public static void main(String[] args) {
                System.out.print("HariKrishna");
                System.out.println("Naresh Technologies");
        }
}
```

How can we redirect STDOUTPUT data to a file? It means the data printed by using *System.out.println()* or *System.err.println()* must be stored in a file instead of printing on console.

Yes, it is very simple; we just need to change *out* and *err* referenced variables internal object to *FileOutputStream* object that is connected to targeted file.

Originally these two variables has stream objects connected to console, this is the reason data is passed to console. Since we are changing *out* and *err* referenced variables internal objects with FileOutputStream object, now System.out.println() and System.err.println() statements sends data to file.

To set *out* and *err* referenced variables with new stream objects in System class below two methods are given

```
public static void setOut(PrintStream out)
public static void setErr(PrintStream err)
```

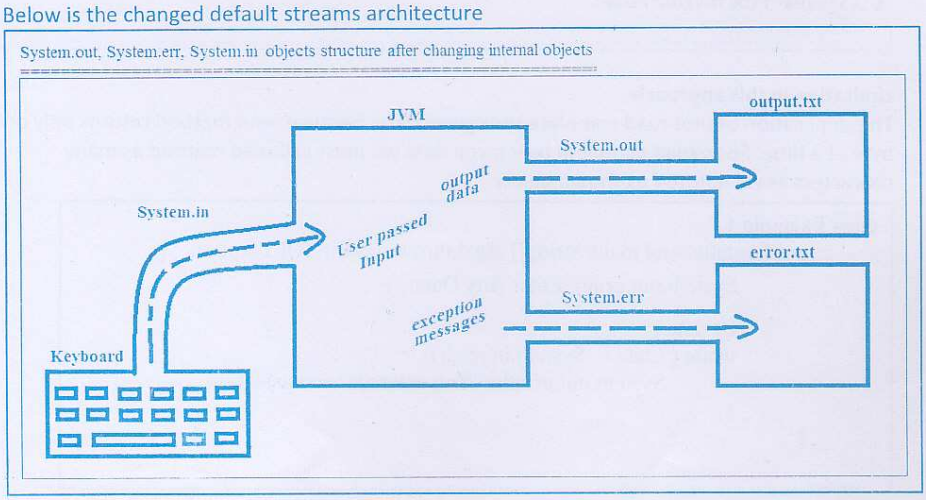
We also have below method to change *in* referenced variable internal object

```
public static void setIn(InputStream in)
```

Below program shows redirecting STDOUTPUT to a file.

```
import java.io.*;

class STDOutDemo
{
        public static void main(String[] args) throws Exception
        {
                System.out.println("Data before setOut() Method"); // printed on console
                System.err.println("Data before setErr() Method"); // printed on console

                //Creating PrintStream objects with targeted files
                PrintStream ps1 = new PrintStream(new FileOutputStream("output.txt"));
                PrintStream ps2 = new PrintStream(new FileOutputStream("error.txt"));

                //changing default streams
                System.setOut(ps1);
                System.setErr(ps2);

                System.out.println("Data after setOut() Method"); // passed to output.txt file
                System.err.println("Data after setErr() Method"); // passed error.txt
                int x = 10/0; //AE exception is passed error.txt
        }
}
```

Below is the changed default streams architecture



Reading Data from keyboard

To read data from keyboard we must use below statements in the program

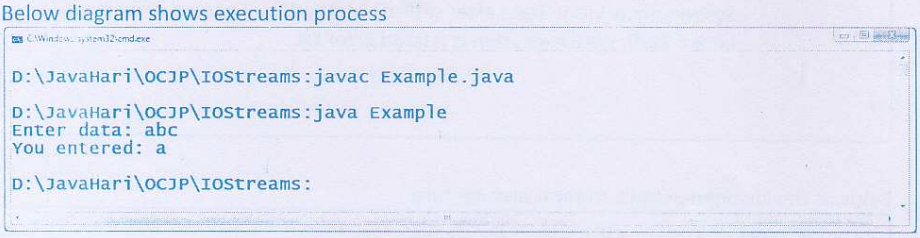
```
int i = System.in.read();
```

Below program shows reading data from keyboard using *System.in.read()*

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

                System.out.print("Enter data: ");

                int data = System.in.read();

                System.out.println("You entered: "+(char)data);
        }
}
```

Execution flow: when you execute this application, on console "Enter data:" message is printed and program execution is paused till user presses "Enter" key on keyboard. Then read method returns the user entered data and stores it in the variable *"data"*.

Below diagram shows execution process

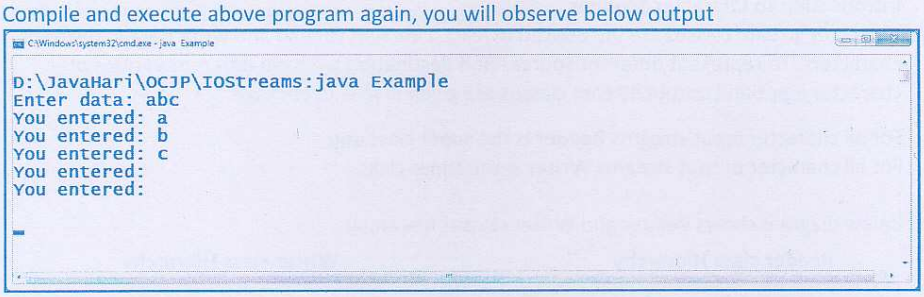


Limitation in this approach

This application cannot read complete user given data, because read method returns only one byte at a time. So to read complete user given data we must call read method as many characters as we entered as shown below

```
class Example {
        public static void main(String[] args) throws java.io.IOException   {
                System.out.print("Enter Any Data: ");

                int data ;
                while ( (data =  System.in.read()) != -1){
                        System.out.println("You entered: "+(char)data);
                }
        }
}
```

Compile and execute above program again, you will observe below output



The problem in this application is some extra unwanted characters are printed and more over application execution is not terminated.

When user presses "Enter" key on keyboard OS appends two more characters to the input data. These two characters are appended based on OS.

In Windows those two characters are **C**arriage **R**eturn (ASCII number 13) and **L**ine **F**eed (ASCII number 10).

So, above application prints user input and these special characters and still executing because while loop condition is not failed, means read() method never returns "-1".

To read only user given input characters, and further to terminate application we must prepare condition with value "**13**" as shown below.

```
class Example {
        public static void main(String[] args) throws java.io.IOException   {
                System.out.print("Enter Any Data: ");

                int data ;
                while ( (data =  System.in.read()) != 13){
                        System.out.println("You entered: "+(char)data);
                }
        }
}
```

This application only works in Windows OS, because other OS returns different value not 13.

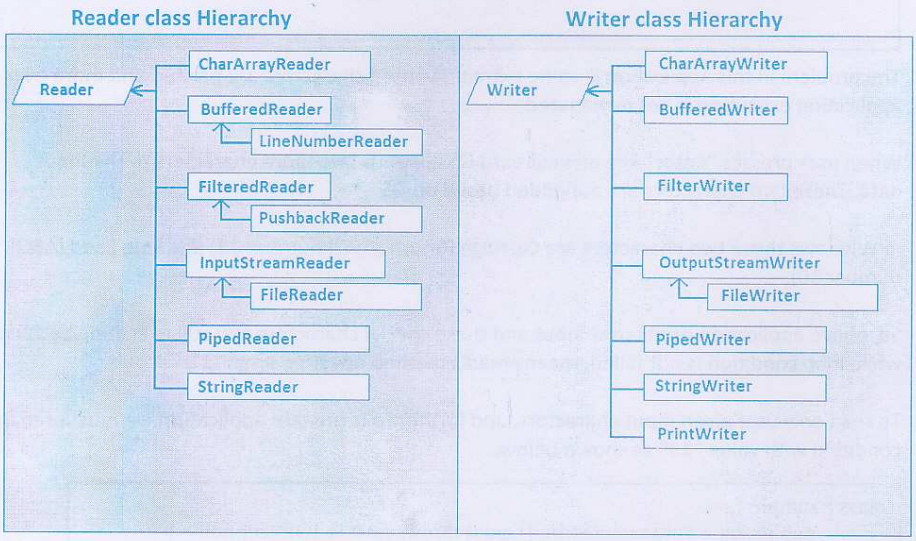
To solve these two problems in every java program developer must write manual validation code that is suitable to all OS. So, to solve this problem Sun has given a class called **BufferedReader** class as part of character streams. It has a method called **readLine** to return complete line as string at time.

Introduction to Character Streams

Character stream classes are introduced in jdk 1.1 version to read and write data in terms of characters. To represent different sources and destination we have different verities of character input and output stream classes are given in java.io package.

For all character input streams **Reader** is the super class and
For all character output streams **Writer** is the super class

Below diagram shows Reader and Writer classes hierarchy



Below are the Reader and Writer class methods

Reader class methods

```
public boolean ready() throws IOException;

public int read() throws IOException;
public int read(char[] ch) throws IOException;
public abstract int read(char[] ch, int off, int len)  throws IOException;

public long skip(long n) throws IOException;

public boolean markSupported();
public void mark(int readAheadLimit) throws IOException;
public void reset() throws IOException;

public abstract void close() throws IOException;
```

Writer class methods

```
==================
public void write(int i) throws IOException;
public void write(char[] ch) throws IOException;
public abstract void write(char[] ch, int off, int len) throws IOException;
public void write(String str) throws IOException;
public void write(String str, int off, int len) throws IOException;

public Writer append(char ch) throws IOException;
public Writer append(CharSequence seq) throws IOException;
public Writer append(CharSequence seq, int off, int len) throws IOException;

public Appendable append(char ch) throws IOException;
public Appendable append(CharSequence seq) throws IOException;
public Appendable append(CharSequence seq, int off, int len) throws IOException;

public abstract void flush() throws IOException;
public abstract void close()  throws IOException;
```

FileReader and FileWriter

These two classes are used to read and write data from a file as characters. It is recommended to use FileReader and FileWriter classes to read data from text file, where as FileInputStream and FileOutputStream classes are only recommended to use reading and writing image files.

Below are constructors available in these classes

```
FileReader(File f) throws FileNotFoundException
FileReader(FileDescriptor fd) throws FileNotFoundException
FileReader(String name) throws FileNotFoundException
```

```
FileWriter(File f) throws FileNotFoundException
FileWriter(FileDescriptor fd) throws FileNotFoundException
FileWriter(String name) throws FileNotFoundException

FileWriter(File f, boolean append) throws FileNotFoundException
FileWriter(String name, boolean append) throws FileNotFoundException
```

Below application shows reading data from a file "abc.txt" using FileReader

```
import java.io.*;

class FRDemo{
        public static void main(String[] args) throws FileNotFoundException,IOException{
                FileReader fr = new FileReader("abc.txt");

                int data;
                while ( (data = fr.read()) != -1)                {
                        System.out.print(data +" ... ");
                        System.out.println((char)data);
                }
        }
}
```

Limitation with FileReader

Using FileReader class we cannot read one line at time from the file. We can only read one character at time. It reduces the reading performance of the application.

To solve this problem and to read one line at a time we must use BufferedReader class.

BufferedReader and BufferedWriter

These two classes are used to improve reading and writing capability of other input and output stream.

Below are the constructors available in these classes to create its objects

```
public BufferedReader(Reader in)
public BufferedReader(Reader in, int size)

public BufferedWriter(Writer out)
public BufferedWriter(Writer out, int size)
```

BufferedReader class has below method to read character data one line at time.

```
public String readLine() throws IOException
```

Below statements shows creating BufferedReader class object for reading one line at time from the file "abc.txt"

```
BufferedReader br = new BufferedReader(new FileReader("abc.txt"));
```

We should call readLine() method as shown below to read one line

```
String line     = br.readLine();
```

It returns only first line, to read all line of text we must call it in while loop. It returns null if cursor reaches end of the file.

Below program shows reading file data one line at a time.

```
import java.io.*;
class BRFRDemo{
        public static void main(String[] args) throws FileNotFoundException,IOException{
                BufferedReader br = new BufferedReader(new FileReader("abc.txt"));

                String line;
                while ( (line = br.readLine()) != null){
                        System.out.println(line);
                }
        }
}
```

How can we read data from keyboard one line at a time?
It is only possible by using BufferedReader class object as this class has *readLine()* method.

We must create *BufferedReader* class object in connection with *System.in* as shown below
BufferedReader br = new BufferedReader(System.in); (***Wrong***)

But this statement leads to CE because BufferedReader class does not have InputStream parameter constructor. It has Reader parameter constructor.

So we need a Reader subclass that has InputStream parameter constructor. That class is *InputStreamReader*.

InputStreamReader and **OutputStreamWriter**
An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset.

It has below constructor to take InputStream type object

```
public InputStreamReader(InputStream in)
```

An OutputStreamWriter is a bridge from character streams to byte streams: Characters written to it are encoded into bytes using a specified charset.

It has below constructor to take OutputStream type object

```
public OutputStreamWriter(OutputStream out)
```

Now let us try to create BufferedReader object connecting with keyboard

- pass *System.in* as argument to *InputStreamReader* object creation
- pass *InputStreamReader* object as argument to *BufferedReader* object creation

Below is the complete object creation statement

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

Below application shows reading data from keyboard

```
import java.io.*;
class KeyboardReader{
        public static void main(String[] args) throws FileNotFoundException,IOException{
                BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

                System.out.print("Enter data:");

                String line = br.readLine();
                System.out.println("You entered: "+line);
        }
}
```

Write a program to add two numbers by reading its values from keyboard?

```
import java.io.*;
public class Addition {
        public static void main(String[] args) {

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

                        System.out.print("Enter first integer value: ");
                        String fno = br.readLine();

                        System.out.print("Enter second integer value: ");
                        String sno = br.readLine();

                        //converring string form to int form
                        int i = Integer.parseInt(fno);
                        int j = Integer.parseInt(sno);

                        int k = i + j;

                        System.out.println("Result: "+k);
                }
                catch(NumberFormatException nfe){
                        System.out.println("Please pass only integer numbers");
                }
                catch(IOException ioe){
                        ioe.printStackTrace();
                }
        }
}
```

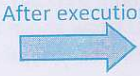
IOStreams real-time project based code

In project development sometimes it is necessary to change a number or a word in a file after our project installation. For example port number of tomcat server in server.xml file.

Below example shows replacing "xyz" with "abc" and storing changing content in the same file.

abc.txt

```
abc bbc xyz bbc abc
bbc xyz bbc xyz bbc
cbc bbc xyz bbc cbc
bbc xyz cbc xyz cbc
```



abc.txt

```
abc bbc abc bbc abc
bbc abc bbc abc bbc
cbc bbc abc bbc cbc
bbc abc cbc abc cbc
```

```
import java.io.*;

public class IOStreamProject {
        public void changeData()throws IOException{
                BufferedReader br = new BufferedReader(new FileReader("abc.txt"));

                //StringBuilder object creation to store file data
                StringBuilder fileDataBuffer = new StringBuilder();

                //reading complete file data and storing it in StringBuilder object
                while(br.ready()) //ready() method checks is stream has data to read or not
                {
                        fileDataBuffer.append( br.readLine() + "\n" );
                }

                //changing StringBuilder to String to call replace() method
                String fileData = fileDataBuffer.toString();

                //replacing "xyz" with "abc"
                fileData = fileData.replace("xyz","abc");

                //writting changed content to file
                FileWriter fw = new FileWriter("abc.txt"));
                fw.write(fileData);

                fw.flush();

                System.out.println("Data changed successfully");

                br.close();
                fw.close();
        }
        public static void main(String[] args) throws IOException  {
                IOStreamProject p = new IOStreamProject();
                p.changeData();
        }
}
```

How FileInputStream and FileOutputStream classes can check whether given file is existed or not? Also how FileOutputStream class can create file if the given file is not existed?
These two classes internally use **File** class.

File class
This class is used to represent files and directory paths (but not data of the file, file data is represented by FileInputStream and FileOutputStream). Basically this class is used to create, delete, rename files and directories and also used to know above the files and directory information like, given file name represents file or directory, type of the file read-only or writable, last modified, etc...

Instances of the File class are immutable; that is, once created, the abstract pathname represented by a File object will never change, because we do not have setter method to change the file name in this File object. But we have getter method to get file name.

File class *constructors*
File class has below constructors to create its object.

public File(String pathname)
Creates a new File instance with the given file name.

Ex:
```
File f = new File("abc.txt"); <- abc.txt file is used from the current working directory
```

public File(String parent, String child)
Creates a new File instance with the given parent file and child path.

Ex:
```
File f = new File("IOStreams", "abc.txt"); <- abc.txt file is used from IOStreams folder from
                                                the current working directory.
```

public File(File parent, String child)
Creates a new File instance with the given parent file and child path. To use this constructor parent file name must be passed as File class object.

Ex:
```
File f1 = new File("IOStreams");
File f2 = new File(f1, "abc.txt"); <- abc.txt file is used from IOStreams folder from the
                                      current working directory
```
Note: If we pass parent as null, abc.txt file is used from current working directory.

public File(URI uri)
Creates a new File instance by converting the given *file:* URI into an abstract pathname.

Ex:
```
File f1 = new File("abc.txt");
File f2 = new File(f1.toURI());
```

Rule: If we pass child argument as *null* all four constructors throws *NullPonterException*

Below program shows creating File class objects and printing those objects

```
import java.io.*;

class FileConstructors
{
        public static void main(String[] args)
        {
                File f1 = new File("abc.txt");
                File f2 = new File("test","abc.txt");
                File f = new File("test");
                File f3 = new File(f,"abc.txt");
                File f4 = new File(f1.toURI());

                System.out.println("f1: "+f1);
                System.out.println("f2: "+f2);
                System.out.println("f: "+f);
                System.out.println("f3: "+f3);
                System.out.println("f4: "+f4);
        }
}
```

Output

```
C:\Windows\system32\cmd.exe
D:\Nareshit\JavaHari\OCJP\IOStreams>javac FileConstructors.java

D:\Nareshit\JavaHari\OCJP\IOStreams>java FileConstructors
f1: abc.txt
f2: test\abc.txt
f:  test
f3: test\abc.txt
f4: D:\Nareshit\JavaHari\OCJP\IOStreams\abc.txt

D:\Nareshit\JavaHari\OCJP\IOStreams>
```

What happened when we create File **class object? Will JVM check whether a file** or **directory existed with given name? When executing above program if the file** abc.txt **and the folder** test **are not existed will JVM throws any exception else it creates files with given names?**
When File class object is created, JVM just creates File class object with the given file name as that object state. It will not check for file or directory with the given name.

While executing the above program, the file *abc.txt* and folder *test* are not existed in the current working directory, even though there is no exception as you can see in the above cmd prompt window, and after this program execution check current working directory you will not find the file *abc.txt* or the directory *test* there.

Conclusion: File class object creation does not create files.

File class *methods*

File class has below methods to perform different operations on regular files and directories including creating, deleting and renaming files and directories

Use below method to find is there any regular file or directory existed with given name

- public boolean exists()
  It checks whether file or directory is existed with given name or not.
  Return *true*, if file is existed else returns *false*

Use below methods to create file or directory with given name.

- public boolean createNewFile() throws IOException
  It creates an empty normal file with the given name in given path.
  Returns *true,* if the file is created with given file name.
- public boolean mkdir() throws IOException
  It creates a directory with given name in the given path.
  Returns *true,* if the directory is created with the given file.
- public boolean mkdirs() throws IOException
  It creates a both parent and child directories with given names in the given path. It creates child directory inside parent directory.
  Returns *true,* if the directory is created with the given file.

Use below methods to know given file name denotes normal file or directory

- public boolean isFile()
  It returns *true,* if the given file name denoted a normal file.
- public boolean isDirectory()
  It returns *true,* if the given file name denoted a directory.

Use below methods to know basic file properties

- public boolean canRead();
  returns *true*, if file is readable
- public boolean canWrite();
  returns *true*, if file is writable
- public boolean isHidden();
  returns *true*, if file is hidden
- public boolean setReadOnly();
  Set file is only readable, after this method call we cannot write data to this file.
- public long lastModified();
  Returns last modified time in milliseconds.
- public long length()
  Returns the length of the file denoted by this abstract pathname. The return value is unspecified if this pathname denotes a directory.

Use below methods to get file name and its path

- public String getName();
  Returns the relative path of the file, means name of the file.
- public String getAbsolutePath();
  Returns the absolute path of the file, means complete path of the file including name, in String object format.
- public File getAbsoluteFile();
  Returns the absolute path of the file in File object format.
- public String getCanonicalPath() throws IOException;
  Returns absolute path in String object format with system depended drive name.
- public File getCanonicalFile() throws IOException;
  Returns absolute path in File object format with system depended drive name.
- public boolean isAbsolute();
  Returns true, if file object is created with absolute path

Use below method to rename file or directory

- public boolean renameTo(File f);
  Renames current file with given name.

Use below methods to delete file or directory that is represented by this file object

- public boolean delete();
  It deletes file immediately
- public void deleteOnExit();
  It deletes file after JVM terminates.

Use below methods to list files and subdirectories of a given directory

- public String[] list();
  It returns all file and subdirectory names as String objects using String array.
- public File[] listFiles();
  It returns all file and subdirectory names as File objects using File array.
- public String[] list(FilenameFilter filter);
  It returns all file and subdirectory names whose names are matched with given filter as String objects using String array.
- public File[] listFiles(FilenameFilter filter);
  It returns all file and subdirectory names whose names are matched with given filter as File objects using File array.

Use below method to print file object state

- public String toString();
  It returns File object state in String format.

Use below method to convert file normal path to URL and URI

- public java.net.URL toURL() throws java.net.MalformedURLException;
- public java.net.URI toURI();

Below applications shows using all above methods

```
import java.io.*;

public class NormalFileCreation
{
	public static void main(String[] args) throws IOException
	{
		File f= new File("xyz.txt"); //just File object is created, xyz.txt file is not existed

		if(!f.exists())
		{
			System.out.println("f.createNewFile():"+f.createNewFile());
			System.out.println("File is Created");
		}
		else
		{
			if(f.isFile())
			{
				System.out.println("in else block");
				System.out.println("File Object is a normal file");
				System.out.println("f.getName():"+f.getName());
				System.out.println("f.length():"+f.length());

				System.out.println("f.canRead():"+f.canRead());
				System.out.println("f.canWrite():"+f.canWrite());
				System.out.println("f.getAbsolutePath():"+f.getAbsolutePath());
				System.out.println("f.getPath():"+f.getPath());
				System.out.println("f.deleteOnExit()");
				f.deleteOnExit();
				System.out.println("f.setReadOnly():"+f.setReadOnly());

				System.out.println("f.canWrite():"+f.canWrite());

			}
		}
	}
}
```

Run this application twice

In first run if block is executed, and file is created. In second run else block is executed.

Now answer below question

How FileInputStream and **FileOutputStream classes know whether file is existed** or **not with the given name?** *By using java.io.File class API*

These two classes in their constructors first they create File class object with given name then they call exist() method on that file object. If exist() method returns *true* means file is existed then they start reading and writing operation. If exist() method returns *false* then FileInputStream throws FileNotFoundException, and FileOutputStream creates file with the given name by using createNewFile() method. Below is the sample code of FIS and FOS

```
public class FileInputStream extends InputStream {
        private File name;
        public FileInputStream(String name) throws FileNotFoundException{
                this(new File(name));
        }
        public FileInputStream(File name) throws FileNotFoundException {
                if (!name.exists()){
                        throw new FileNotFoundException(name + " not existed")
                }

                this.name = name;
        }
}
```

```
public class FileOutputStream extends OutputStream {
        private File name;

        public FileOutputStream(String name) throws FileNotFoundException{
                this(new File(name));
        }
        public FileOutputStream(File name) throws FileNotFoundException        {

                if (name.exists()){
                        if(name.isDirectory()){
                                throw new FileNotFoundException(name + " not existed");
                        }
                }
                else{
                        boolean created = name.createNewFile();
                        if( !created ){
                                throw new FileNotFoundException(name + " not existed");
                        }
                }

                this.name = name;

        }
}
```

What happened in JVM when below statements are executed?

File f = new File("abc.txt");

- Just File class object is created in JVM. The file abc.txt is not created if it is not present.

FileInputStream fis = new FileInputStream("abc.txt");
FileReader fr = new FileReader("abc.txt");

- If abc.txt file is existed FIS or FR stream object is created to read data from that file,
- If it is not exited FIS or FR throws FileNotFoundException

FileOutputStream fos = new FileOutputStream("abc.txt");
FileWriter fw = new FileWriter("abc.txt");

- If abc.txt file is existed FOS stream object is created to write data into that file,
- If it is not exited FOS creates empty file with name abc.txt and then stream object is created.

Conclusion:

File class is only responsible of file creation, deletion and changing its properties
FileInputStream, FileReader are responsible of file data means in reading data from file
FileOutputStream, FileWriter are also responsible of file data means in writing to file

Below program shows creating directory, subdirectory and file inside a directory

```
import java.io.*;

class FileParentDirectoryDemo {
        public static void main(String[] args) throws IOException{
                File f1 = new File("abc.txt");
                f1.createNewFile(); //abc.txt is created as normal file in current working
                                                                        directory
                File f2 = new File("bbc.txt");
                f1.mkdir(); //bbc.txt is created as directory in current working directory.
                                We can create directories with extensions.

                File f3 = new File("xyz");
                f3.mkdir(); //xyz is created as directory in current working directory.

                File f4 = new File(f3, "1.txt");
                f4.createNewFile(); //1.txt is created as normal file in xyz directory.

                File f5 = new File(f3, "abc");
                f5.mkdir(); //abc is created as directory in xyz directory.

                File f6 = new File("pqr", "stv");
                f6.mkdirs(); //pqr is created as directory in current working directory, and stv is
                                                                        created in pqr as subdirectory.
```

```
                System.out.println("f1: "+f1);
                System.out.println("f2: "+f2);
                System.out.println("f3: "+f3);
                System.out.println("f4: "+f4);
                System.out.println("f5: "+f5);
                System.out.println("f6: "+f6);
        }
}
```

O/P:

```
f1: abc.txt
f2: bbc.txt
f3: xyz
f4: xyz\1.txt
f5: abc.txt\abc
f6: pqr\stv
```

```
//List directory files
import java.io.File;
public class ListFiles{
        public static void listFiles(String file)  {
                listFiles (new File(file));
        }
        public static void listFiles(File dir){
                try{
                        if(dir != null){

                                File dirList[] = dir.listFiles(); //File class method not current method

                                if(dirList != null ){
                                        for (int i = 0 ; i < dirList.length ; i++ ){
                                                File file = dirList[i];

                                                if(file.isFile())   {
                                                        System.out.println(file + " is a file");
                                                }
                                                else{
                                                        System.out.println(file + " is a directory");
                                                        listFiles(file);
                                                }
                                        }
                                }
                        }
                        else{
                                System.out.println("directory is null");
                        }
                }
                catch(Exception e){
                        e.printStackTrace();
                }
        }
}

class Test{
        public static void main(String[] args)  {
                //incurrent directory create a directory test with files and sub folders, then pass it
                as argument to this method
                ListFiles.listFiles("test");
        }
}
```

```
// This program demonstrates FileNameFilter.
import java.io.*;

class FileExtention implements FilenameFilter
{
        String extFile;

        public FileExtention(String extFile)
        {
                this.extFile = "." + extFile;
        }

        public boolean accept(File dir, String name)
        {
                return name.endsWith(extFile);
        }
}

public class FilterListFiles
{
        public static void main(String args[])
        {
                String strDir = "../IOStreams";
                File f = new File(strDir);

                FilenameFilter onlyFile = new FileExtention("txt");

                String strFile[] = f.list(onlyFile);

                System.out.println("\nThe JAVA files in the current directory are:\n");
                for(int i=0; i<strFile.length; i++)
                {
                        System.out.println(strFile[i]);
                }
        }
}
```

```
/*
Below application shows deleting normal files and directories
        Before executing this application, create normal empty files "1.txt, 2.txt" and a
        directory "xyz with a file abc.txt"
*/
import java.io.*;

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

                File f1 = new File("1.txt");
                f1.delete();
                if(!f1.exists()){
                        System.out.println("1.txt is deleted");
                }
                else{
                        System.out.println("1.txt is not deleted");
                }

                File f2 = new File("2.txt");
                f2.deleteOnExit();
                if(!f2.exists()){
                        System.out.println("2.txt is deleted");
                }
                else{
                        System.out.println("2.txt is not deleted");
                }

                File f5 = new File("xyz");
                f5.delete();
                if(!f5.exists()){
                        System.out.println("xyz is deleted");
                }
                else{
                        System.out.println("xyz is not deleted");
                }
        }
}
```

```
O/P
===
1.txt is deleted
2.txt is not deleted
xyz is not deleted
```

After this application execution check current working directory; you will discover "2.txt" is also deleted. Because deleteOnExit() method deletes file after JVM terminates.

The directory "xyz" will not be deleted, because it is not an empty directories, delete() method deletes only empty directories.

```
//Deleting directory files
import java.io.File;
public class DirectoryDelete{
	public static void directoryDelete(String file){
		directoryDelete(new File(file));
	}
	public static void directoryDelete(File dir){
		try{
			if(dir != null){

				File dirList[] = dir.listFiles();

				if(dirList != null ){
					for (int i = 0 ; i < dirList.length ; i++ ){
						File file = dirList[i];

						if(file.isFile())  {
							file.delete();
						}
						else{
							directoryDelete(file);
						}
					}
				}
				dir.delete();
			}
			else{
				System.out.println("directory is null");
			}
		}
		catch(Exception e){
			e.printStackTrace();
		}
	}
}

class Test{
	public static void main(String[] args) {
		//incurrent directory create a directory test with files and sub folders, then pass it
		as argument to this method
		DirectoryDelete.directoryDelete("test");
	}
}
```

File object creation with complete file or directory path

Q) How can we create File object by using normal file or directory that is existed in another directory let us say "D:\examples\test\abc.txt"?

A) We must create File class object by passing complete file path as shown below

```
import java.io.*;

class FileSeperators {
        public static void main(String[] args) {
                File f = new File("D:\examples\test\abc.txt");
                System.out.println("f: "+f);
        }
}
```

Compile above program you will get below compile time error

FileSeperators.java:7: illegal escape character

"\" represents escape sequence character.

After "\" the only allowed characters are "

	Character	usage
➢ Back slash	-> \	" \\ "
➢ Double quote	-> "	" \" "
➢ Single quote	-> '	" \' "
➢ Space character	->	" \ "
➢ Tab character	-> t	" \t "
➢ New line character	-> n	" \n "
➢ Carriage return character	-> r	" \r "
➢ Form feed character	-> f	" \f "

So to pass complete path of the file or directory we must use "\\" as file separator as shown below

```
import java.io.*;

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

                File f = new File("D:\\examples\\test\\abc.txt");

                System.out.println("f: "+f);
        }
}
```

File separators

File separators are platform dependent. In windows we must use "\\", and in Linux or Solaris we must use "/". So when we are moving project from Windows to Linux or Solaris or vice versa in all Java files we must change file separators to platform specific separator. Since it is a manual task, it requires lot of testing so it leads lot of maintenance cost.

So to solve this problem we must have a way to retrieve file separators dynamically specific to current platform.

In File class we have been given below two variables to retrieve and place path separators dynamically specific to current platform, they are

```
public static final char separatorChar
public static final String separator
```

Rewrite above application to place file separators dynamically

```
import java.io.*;

class FileSeperators {
        public static void main(String[] args) {
                String fs       = File.seperator;

                File f = new File("D:"+ fs +"examples"+ fs +"test"+ fs +"abc.txt");

                System.out.println("f: "+f);
        }
}
```

O/P

After executing above application we will get output

In windows
D:\examples\test\abc.txt

In Linux or Solaris
D:/examples/test/abc.txt

Chapter 29

Networking

- In this chapter, You will learn
  - Networking basics
  - Basic networking terminology
  - Types of protocols
  - How a request is sent to targeted server
  - How a response is sent back to same client
  - Socket programming
  - Understanding URLs and URLConnection
  - Understanding retrieving IP address and Host name dynamically
- By the end of this chapter- you will be in a position to develop networking based applications.

Interview Questions

By the end of this chapter you answer all below interview questions

- What is a Network?
- Types of Networks?
- Networking terminology
  - Request and response
  - Server and server system
  - Client and Client system
  - IP Address and host name
  - Port number
  - Protocol and Types of Protocols
  - URL, URI
  - Socket and ServerSocket
- How user can send request to only targeted Server when that client computer is connected to multiple other servers?
- And in the same way how server can send response back to same client computer when it is connected to multiple other client computers?
- How can we perform network operations using Java Application?
- How can we read / Update a resource available in remote / server computer?
- How can we get IP Address or host name of a computer dynamically?

Networking

Definition
Networking is the process of connecting multiple remote or local computers together.

Types of Networks
In the world we have below three types of networks

1. LAN
2. MAN
3. WAN

LAN – Local Area Network, used to connect computers within the building

MAN – Metropolitan Area Network, used to connect computers within a city.

WAN – Wide Area Network, used to connect computers throughout world.

Networking Terminology

Request and Response:
An *input data* sending via network to an application that is running in a remote computer is called *request*.

The *output* coming out from that application back to this client program is called *response*.

Client and Client System:
An *application* that allows us to send request is called *client*, and the person who is inputting the required data is called *end-user*, and the person who pays money to develop that entire project is called *customer*.

Client System is a *computer* in which client programs are executed, and on which end-user works.

Server and Server System:
Server is a *set of programs* (software) that takes request via network, process that request, generates response and sends it back to client application.

Server System is also a *computer* that has capability to receive network calls and handovers those calls to appropriate server software those are installed in that computer.

What is an IP Address and host name?
IP Address, Internet Protocol Address, is an identification number of computer in the network.

Rule: In a network two computers should not contain same IP Address. In this case network will prompt an error message **"IP Address conflict"**.

Host Name is the alias name of the computer. As human being we cannot remember computers with numbers. So each computer IP Address is mapped with a string called hostname.

Note:
In projects it is not recommended to use IP Address in sending request, instead we should use its hostname.

In order to use host name in place of IP Address we should configure server system IP Address and hostname in our local system in *hosts* file

This file path is "C:\Windows\System32\drivers\etc"

The mappings placed in this file are called host entries.

- Each entry should be kept on an individual line.
- The IP address should be placed in the first column followed by the corresponding host name with at least one space gap.

Example:

192.168.2.19	NareshIT19
192.168.2.20	NareshIT20
192.168.2.30	NareshIT30

Q) How network can get IP Address of the host name used in the request?
A) From hosts file of the local computer.

The default IP address of a computer is ***127.0.0.1***, and its mapped host name is local host.

Range of the IP Address

0.0.0.0 to 255.255.255.255

This entire range is divided into five classes
Class A, Class B, Class C, Class D, Class E

Class C range IP Address is used to build LAN, the range starts with "192.168.-.-"
Ex: 192.168.2.14

Below are the DOS commands we use to know

IP Address of the computer – **ipconfig**
Hostname of the computer – **hostname**

Sometimes computer cannot be connected to network even though network cable is plugged. In this situation use below command to establish / reestablish the connection with the server system (with network) – "**ipconfig /renew**".

What is a port number?
Port number is an identification number of server software.
It is a 32-bit positive integer number, having range 0 to 65535.

In a computer how many servers can be installed?
N number of servers can be installed and all of them can be start at a time if they have different port number.

Rule: Two servers cannot have same port number (PORT), as they cannot be run at same time. It leads to ambiguous problem for server system in redirecting request.

Note:
We can install more than one server with same port number, but we cannot start / use them at same time. If we start more than one server with same port number or if we start same server again while it is running, we experience an exception java.net.BindException: Address already in use

Note:
The port numbers 0 to 1024 are already registered for public servers. Hence we cannot release new server software with the port number in this range.

What is Protocol?
Protocol is a set of instructions to send request and receive response via network.

Types of Protocols:
There are two types of protocols
1. TCP/IP – Transmission Control Protocol / Internet Protocol
2. UDP – User Datagram Protocol

TCP/IP is a connection-oriented and secured protocol, for every request we will get response.
UDP is a connectionless protocol, and it is non-secured protocol. Using this protocol we cannot guarantee data transfer.
Ex: Mobile and radio communications.

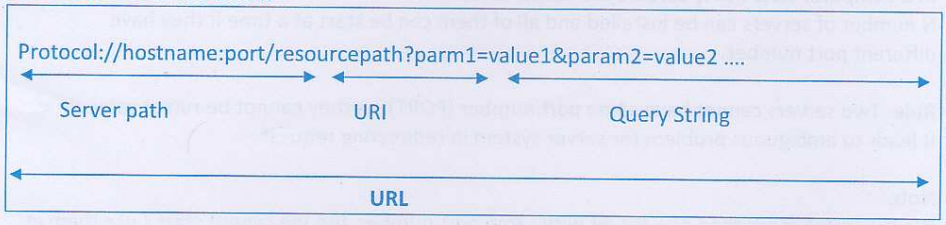
TCP/IP protocol is further divided into four protocols based on type of data we transfer
1. HTTP - Hyper Text Transfer Protocol
2. HTTPS - Hyper Text Transfer Protocol
3. FTP - File Transfer Protocol
4. SMTP - Simple Mail Transfer Protocol

What is a URL, URI?

URL is Uniform Resource Locator, which is an absolute path of remote file, used to locate that file in server.

It contains protocol, IP Address / host name of server system, Port number of the server, resource path and optionally its required input values (called query String).

URL format is as like below



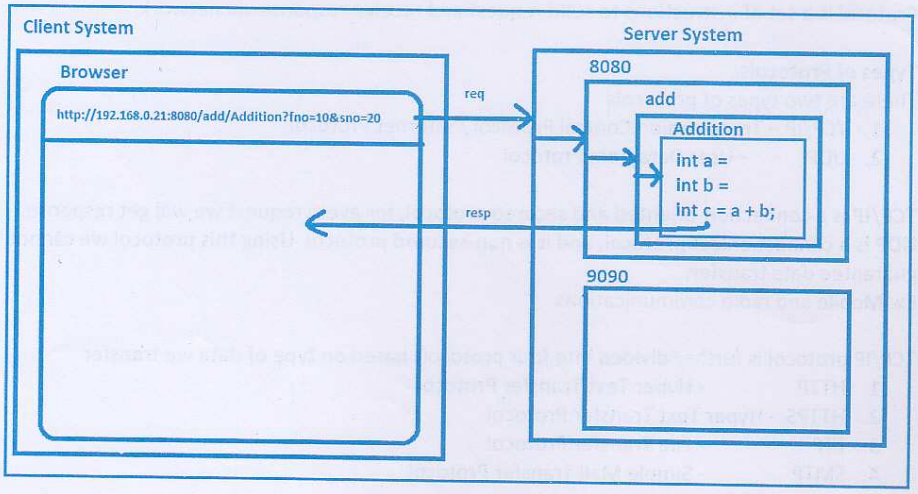
URI- is Uniform Resource Identifier is the path of the resource in server system.

URL – absolute path of remote file
URI – relative path of remote file

Ex:

The URL for sending request to below application is

http://192.168.0.21:8080/add/Addition?fno=10&sno=20

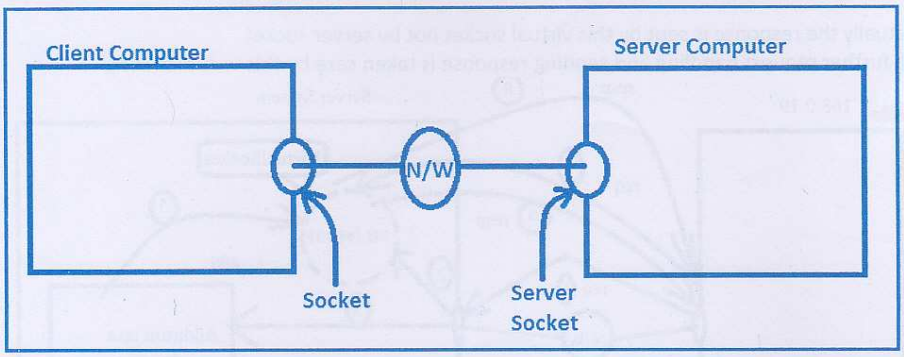


What is Socket, Server Socket?

Socket is a listener through which computer can receive requests and send responses. Using this listener actually computer connected to network and communicate to other computers.

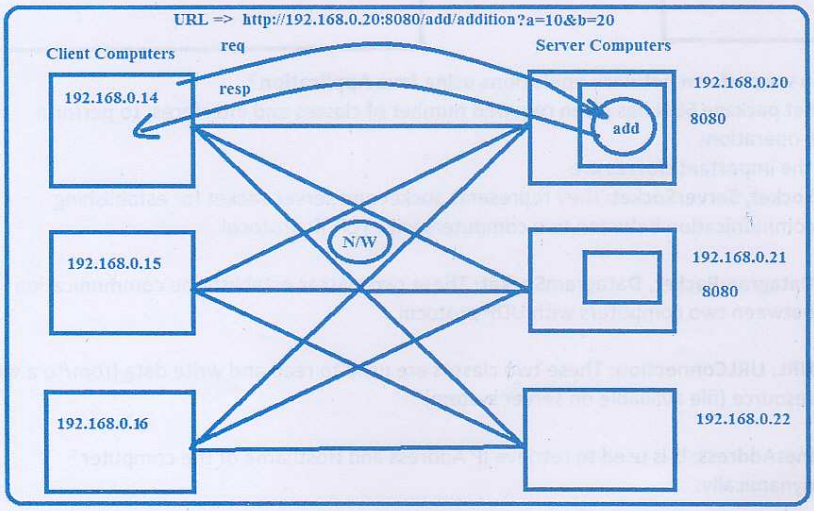
The listener of client system is called socket
The listener of server system is called server socket

It shows in below diagram



How user can send request from client to Server when client computer is connected with multiple servers?

It is possible due to URL format, which contains server IP Address / hostname, port of the server and resource path.

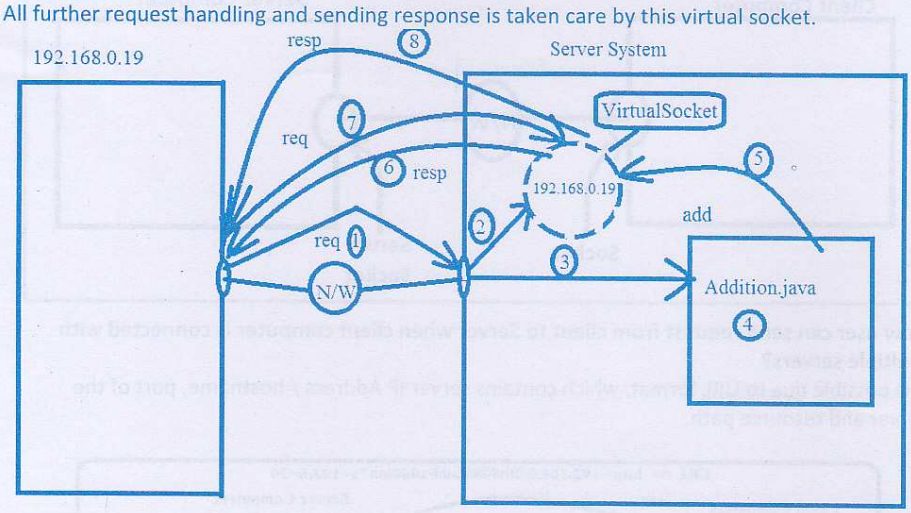


How server can send response back to same client when it is connected to multiple clients?
Due to the virtual socket created in server system for this socket.

When a request is sent from a client, in server system server socket creates a virtual socket (logical socket) for this client and stores this client information such as IP Address of the computer, client application details, etc...

Then server socket sends response back to same client program by using this virtual socket.

Actually the response is sent by this virtual socket not by server socket.
All further request handling and sending response is taken care by this virtual socket.



How can we perform network operations using Java Application?
In java.net package SUN has given required number of classes and interfaces, to perform network operations
Among the important classes are

1. **Socket, ServerSocket**: they represents socket and server socket for establishing communication between two computers with TCP/IP protocol

2. **DatagramPacket, DatagramSocket**: These two classes establish the communication between two computers with UDP protocol.

3. **URL, URLConnection:** These two classes are used to read and write data from/to a web resource (file available on server system)

4. **InetAddress**: It is used to retrieve IP Address and Hostname of the computer dynamically.

Networking programming

```
/*
 Client1.java
This program demonstrates printing a message on to the client, whenever client gets
connected to the server (One-way communication).
 */

import java.io.InputStream;
import java.io.DataInputStream;
import java.net.Socket;

class Client1
{
        public static void main(String args[])
        {
                try
                {
                        Socket s = new Socket("localhost", 4444);

                        InputStream     in = s.getInputStream();
                        DataInputStream dis1 = new DataInputStream(in);

                        String msg1 = dis1.readLine();
                        System.out.println("Server message is: " + msg1);
                        dis1.close();
                        s.close();
                }
                catch(Exception e)
                {
                        System.out.println("Exception: " + e);
                }
        }
}
```

```
/* Server1.java */

import java.io.OutputStream;
import java.io.DataOutputStream;
import java.net.ServerSocket;
import java.net.Socket;

class Server1 {
        public static void main(String args[]){
                try {
                        ServerSocket ss = new ServerSocket(4444);
                        System.out.println("Server is ready...");

                        Socket s = ss.accept();

                        System.out.println("Connection is accepted...");

                        System.out.println("Sent a message to the client...");
                        OutputStream out = s.getOutputStream();
                        DataOutputStream dos1 = new DataOutputStream(out);

                        dos1.writeBytes("Hello");

                        dos1.close();
                        s.close();
                        ss.close();
                }
                catch(Exception e)
                {
                        System.out.println("Exception: " + e);
                }
        }
}
```

Compilation:

Compile above two programs in any order.

Execution:

To execute we need two command prompts, because we are executing two applications at a time. To open new instance of cmd prompt with same folder path we must execute the dos command "start". In first cmd prompt execute server application, it waits for client application to be executed. Then execute client application.

```
>javac *.java
>start  >java Server1  >java Client1
```

```
/** This is an online chatting application. */

//ChatClient.java
import java.io.*;
import java.net.Socket;

public class ChatClient {
        public static void main(String[] args) {
                Socket s1;
                OutputStream os;
                InputStream is;
                DataOutputStream dos;
                DataInputStream dis;
                String sendMsg, receiveMsg;

                try
                {
                        s1      = new Socket("localhost",5555);
                        os      = s1.getOutputStream();
                        is      = s1.getInputStream();
                        dos = new DataOutputStream(os);
                        dis = new DataInputStream(is);

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

                        while(true)
                        {
                                sendMsg = br.readLine();
                                dos.writeUTF(sendMsg);
                                receiveMsg = (String) dis.readUTF();
                                System.out.println(receiveMsg);

                                if(receiveMsg.equals("bye")) {
                                        break;
                                }
                        }
                        dis.close();    dos.close();    os.close(); is.close();  s1.close();
                }
                catch(Exception e) {
                        e.printStackTrace();
                }
        }
}
```

```
//ChatServer.java
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;

public class ChatServer {
        public static void main(String[] args) {
                ServerSocket ss;
                Socket s1;
                OutputStream os;
                InputStream is;
                DataOutputStream dos;
                DataInputStream dis;
                String sendMsg, receiveMsg;

                try
                {
                        ss      = new ServerSocket(5555);
                        System.out.println("This is Online Chatting done on  Java");
                        s1      = ss.accept();
                        os      = s1.getOutputStream();
                        is      = s1.getInputStream();
                        dos     = new DataOutputStream(os);
                        dis = new DataInputStream(is);

                        BufferedReader br = new BufferedReader(new
                                                InputStreamReader(System.in));
                        while(true)
                        {
                                receiveMsg = (String) dis.readUTF();
                                System.out.println(receiveMsg);
                                if(receiveMsg.equals("bye")) {
                                        break;
                                }
                                sendMsg = br.readLine();
                                dos.writeUTF(sendMsg);
                        }
                        dis.close(); dos.close(); os.close(); is.close();  ss.close(); s1.close();
                }
                catch(Exception e){
                        e.printStackTrace();
                }
        }
}
```

```
/*
This program demonstrates displaying the server date on to the client,
using DatagramPacket and DatagramSocket classes.

UDPClient.java
*/

import java.net.DatagramPacket;
import java.net.DatagramSocket;

public class UDPClient
{
        public static void main(String args[])
        {
                try
                {
                        String strDate;
                        DatagramSocket ds     = new DatagramSocket(5555);
                        byte rdata[]                  = new byte[64];
                        DatagramPacket packate = new DatagramPacket(rdata,
                                                                rdata.length);

                        while(true)
                        {
                                ds.receive(packate);
                                strDate = new String(packate.getData());
                                System.out.println("Server Date and Time is: " +
                                                                strDate);
                        }
                }
                catch(Exception e)
                {
                        System.out.println("Exception: " + e);
                }
        }
}



//UDPServer.java
/*
```

```
 This program demonstrates displaying the server date on to the client,
 using DatagramPacket and DatagramSocket classes.
 */

import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.util.Date;

public class UDPServer {
        public static void main(String args[]) {
                try
                {
                        Date date;
                        String strDate;
                        DatagramSocket ds = new DatagramSocket(4444);
                        int i = 0;

                        while(true)
                        {
                                date            = new Date();
                                strDate         = date.toString();
                                byte dbyte[]    = strDate.getBytes();

                                InetAddress address   =
                                                InetAddress.getByName("localhost");

                                DatagramPacket packate      = new
                                DatagramPacket(dbyte, dbyte.length, address, 5555);

                                ds.send(packate);
                                System.out.println((++i)+" packate sent");
                                Thread.sleep(1000);
                        }
                }
                catch(Exception e)
                {
                        System.out.println("Exception: " + e);
                }
        }
}
```

```
//URLDemo.java
/* This program demonstrates URL class.*/
import java.net.URL;
public class URLDemo {
        public static void main(String args[]) {
                try {
                        URL u = new URL("http://system1:1024/NetDemo2.java");

                        System.out.println("\nProtocol: " + u.getProtocol());
                        System.out.println("\nHost: " + u.getHost());
                        System.out.println("\nPort: " + u.getPort());
                        System.out.println("\nFile: " + u.getFile());
                        System.out.println("\nPath: " + u.toExternalForm());
                }
                catch(Exception e){
                        System.out.println("Exception: " + e);
                }
        }
}
//FileURLDemo.java
/*  This program demonstrates URL class.  */
import java.io.*;
import java.net.*;

public class FileURLDemo{
        public static void main(String args[]) {
                try {
                        File file       = new File("URLDemo.java");
                        String filePath = "file:///" + file.getAbsolutePath();
                        URL fileURL             = new URL(filePath);

                        InputStream in          = fileURL.openStream();
                        int data;

                        while((data = in.read()) != -1) {
                                System.out.print((char)data);
                        }
                        in.close();
                }
                catch(Exception e){
                        System.out.println("Exception: " + e);
                }
        }
}
```

```
//URLConnectionDemo.java
/* This program demonstrates URLConnection class. */

import java.io.*;
import java.net.URL;
import java.net.URLConnection;
import java.util.Date;
public class URLConnectionDemo {
        public static void main(String args[]) {
                try {
                        int c;
                        File file               = new File("Example.txt");
                        String filePath         = "file:///" + file.getAbsolutePath();
                        URL fileURL             = new URL(filePath);

                        URLConnection ucon  = fileURL.openConnection();

                        System.out.println("\nDate: " + new Date(ucon.getDate()));
                        System.out.println("Content-Type: " + ucon.getContentType());
                        System.out.println("Expires: " + ucon.getExpiration());
                        System.out.println("Last Modified: " + ucon.getLastModified());

                        int len = ucon.getContentLength();
                        System.out.println("Content Length: " + len + " bytes");

                        if(len > 0) {
                                System.out.println("\n= = = = = CONTENT = = = = = = ");
                                InputStream in = ucon.getInputStream();

                                int i = len;
                                while(((c = in.read()) != -1)) {
                                        System.out.print((char)c);
                                }
                                in.close();
                        }
                        else{
                                System.out.println("No content available.");
                        }
                }
                catch(Exception e)              {
                        System.out.println("Exception: " + e);
                }
        }
}
```

```
//InetAddressDemo.java
/* This program demonstrates InetAddress class. */

import java.net.InetAddress;

public class InetAddressDemo
{
        public static void main(String args[])
        {
                try
                {
                        InetAddress add = InetAddress.getLocalHost();
                        System.out.println("\nLocal Host Details      : " + add);
                        System.out.println("The Host IP Address is   : " +
                                                        add.getHostAddress());
                        System.out.println("The Host name is                 : " +
                                                                add.getHostName());

                }
                catch(Exception e)
                {
                        System.out.println("Exception: " + e);
                }
        }
}
```

Chapter 30

Collections and Generics

- In this chapter, You will learn
  - Definition and Need of collections
  - Problems of array object and its solution
  - Legacy collection classes
  - Collection Framework hierarchy
  - Types of collections
  - Advantage and disadvantage of every collection
  - Storing and retrieving elements from every collection
  - Use of Iterator, ListIterator, and Enumeration.
  - Differences between Iterator and Enumeration
  - Differences between Iterator and ListIterator
  - How collection uses hashCode() and equals() methods
  - Differences between Comparable and Comparator interfaces
  - Adding custom objects as keys to Map objects.
  - Need of utility classes – Collections, Array, BitSet
  - Reading properties from a properties file
  - Internationalization
  - Tokenizing string
  - Working with Date and TimeZone
  - Different timezones
  - Creating Timer and TimerTask (scheduling threads)
  - Real-time project development to show collection need with MVC and LC-RP architectures
- By the end of this chapter- you will understand collecting homogeneous and heterogeneous objects without size limitation.

i

Interview Questions

By the end of this chapter you answer all below interview questions

Collection prerequisites

- In how many ways we can store data in JVM?
- Memory location structure of variable, array object and class object.
- How can we pass single and multiple values and objects as arguments to methods
- Java Bean design pattern - DB and Java Object mapping
- What is effect upon primitive values and objects when they are passed as arguments and if they are modified using that method's parameter?
- Runtime polymorphism
- Use of toString(), equals() and hashCode() methods
- Generics
- Enhanced for-loop
- Autoboxing and unboxing

Introduction to collection framework

- Definition and need of collection
- Introduction to java.util package
- Why these classes are given when we have Object[] to group heterogeneous objects?
- Definition of framework, why this package classes are called Collections Framework?
- In how many formats we can group objects using collection? **Collection** and **Map**
- What are the benefits of collection classes?
- Legacy classes to group objects
- Introduction to Collection framework and its hierarchy.
- Common terminology used in this chapter
- Introduction to core collection interfaces and their inheritance relationship
- Collection
  - |<- List
  - |<- Set <- SortedSet <- NavigableSet (1.6)
  - |<- Queue (1.5)
- Map
  - |<- SortedMap <- NavigableMap (1.6)
- Introduction to **Collection interface class hierarchy**
- Situation force you to use Collection hierarchy classes.
- Similarities and Differences between ArrayList and Vector.
- Similarities and Differences between HashMap and Hashtable.
- Collection and its sub interface's methods.
- Different ways for retrieving elements from the Collection objects

- Introduction to Iterator, ListIterator, and Enumeration
- Their rules and methods usage
- Enhanced *for* loop (1.5)
- Similarities and Differences between Iterator and Enumeration.
- Similarities and differences between Iterator and ListIterator
- Understand the operations performed by addAll(), removeAll(), containsAll() and retainAll() methods in the Collection interface.
- Understanding equals() and hashCode() methods usage in collection objects List and HashSet, LinkedHashSet.
- Storing elements in sorting order using TreeSet
- Use of Comparator interface and difference between Comparable and Comparator interfaces.
- Inserting user defined objects in TreeSet using Custom Comparator
- Introduction to **Map interface class hierarchy**
- Three views of Map.
- Difference between Collection and Map
- Similarities and differences between HashMap and Hashtable.
- Map.Entry
- Storing entries with custom objects as keys
- Identify methods in the **Collections** class those can be used to produce immutable and thread-safe versions of various types of collections also method for searching, sorting, swapping elements in collection.
- How can you sort elements of unsorted collection like ArrayList?
- Identify methods in the **Arrays** class those can be used to perform searching and sorting operations of array elements.
- How can you print all elements of array without using for loop explicitly?
- Introduction to Properties class and the way of loading the properties into properties object from the file and storing the properties from the properties object to a file.
- Tokenizing the given string using **StringTokenizer**
- Implementing I18N application in JAVA using ResourceBundle and Locale
- Use of Currency class.
- Retrieving date and timestamp of the current computer using **Date** and **TimeStamp**.
- Introduction to ***Generics***, need of Generics and rules while using Generics.

Collections Framework and Generics

Definition and Need of Collection

Collection is a Java object that is used to group homogeneous & heterogeneous, duplicate & unique objects without size limitation for carrying multiple objects at a time from one application to another application among multiple layers of MVC architecture as method arguments and return type.

Introduction to java.util package

java.util package contains several classes to group / collect *homogeneous* and *heterogeneous* objects *without size limitation*. These classes are usually called collection framework classes.

Why collection classes are given when we have Object[] to group heterogeneous objects? What is the problem with Object[] to collect objects?

All array objects has below two problems

1. It allows us to store only same type of elements.
2. It is fixed in size

The first problem can be solved using java.lang.Object[]. Using Object[] array we can collect all types of objects. But the second problem cannot be solved automatically. We should develop below code to solve this problem

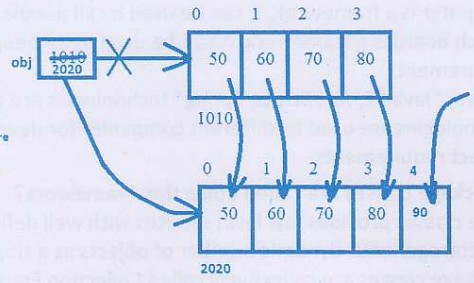
- Create array object with initial size and store elements
- Once array size is reached its capacity, execute below steps to store new element.
  - *Step #1:* Create another temporary array with required size
  - *Step #2:* Copy old array values to new array
  - *Step #3:* Add new element to new array at the end of all its elements
  - *Step #4:* Assign new array object reference to old array referenced variable

Check below program for more clarity

Program

```
Array object creation
Object[] obj = new Object[4];

Storing elements
obj[0] = "50";
obj[1] = "60";
obj[2] = "70";
obj[3] = "80";

Size reached its Capacity so further we can not store
any more elements,
1. So creating new array object with required size
   Object[] tempObj = new Object[10];
2. Copying old array elements to new array
   int oldArraySize = obj.length;
   int i = 0;
   for( ; i < oldArraySize; i++)
   {
       tempObj[i] = obj[i];
   }
3. storing new element at end of new array object
   tempObj[i] = "90";

4. Assinging new array object reference to old array
   referenced variable
   obj = tempObj;
```

Memory Location Structure



The above array object creation logic should be developed in every java project, which is treated as code redundancy because in addition to business logic development in every project and in every company developer must spent time to develop this repeated code. It leads to lot of code maintains problems and also takes more time to complete project development.

So, to solve all above problems SUN decided to implement collection classes common to every java project with high performance.

All collection classes are defined in java.util package.

Sun developed many collection classes among them 15 are important, they are

- LinkedList
- ArrayList
- Vector
- Stack
- PriorityQueue
- HashSet
- LinkedHashSet
- TreeSet
- HashMap
- LinkedHashMap
- TreeMap
- Hashtable
- Properties

In addition to collection classes SUN also defined some helper classes
They are

- Collections
- Arrays
- Random
- Scanner
- StringTockenizer
- Locale
- ResourceBundle
- Date
- Calendar
- GregorianCalendar

Why the name collection framework?

Let us first understand what is a framework?

Framework is a semi finished reusable application which provides some common low level services for solving reoccurring problems and that can be customized according to our requirement.

For example

- Computer is a framework, it can be used by all people according to their requirement
- Switch board is a framework, it can be used by all people according to their requirement
- In Java, "Java EE, JSF, Struts, Spring" technologies are frameworks, all these technologies are used by different companies for developing projects according to the project requirements.

Why this package classes are called Collection Framework?

This package classes provides low level services with well defined data structures to solve collecting heterogeneous dynamic number of objects as a single entity. Due to these reason java.util package classes are collectively called Collection Framework.

All collection classes are implementing from *java.io.Serializable* so the collection object along with all its internal objects can be stored in local file system or can be sent across network to remote computer. Here **Rule is** *objects stored in collection are also should be Serializable types to store collection in file*.

Need of Collection framework classes

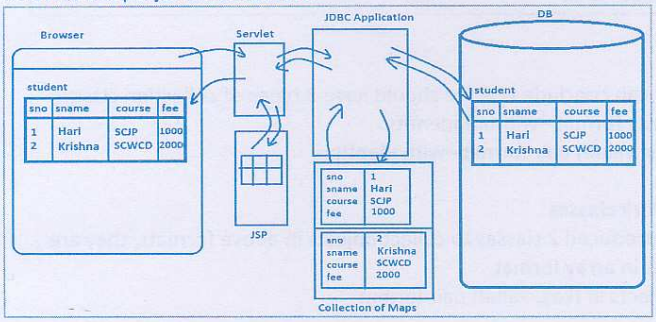
In Java projects Collection Framework classes are used to store and transport objects are of same and different types without *size limitation*.

Project Code design with collection classes

As you know, every project has three layers Model, View, Controller.

- ➔ Model layer application collects data from DB using ResultSet and store it collection object, and return this collection object to Controller layer application.
- ➔ Then Controller layer application read data from collection object and fill it in View layer required code using HTML components
- ➔ Finally this HTML code is passed to client then browser displays the result to end-user.

Below is the project architecture



In this project

- ➔ First we retrieve student's records using JDBC API
- ➔ Then we read all records from ResultSet and store them in Collection object
- ➔ Finally we send that collection object to Servlet
- ➔ Then Servlet by using JSP displays student's records on browser.

Check *Projects* section for the complete code of this project.

Collection classes are Java data structures

Basically collection framework classes are Java data structures. These classes internally use several standard data structures to collect objects such as growable array, vector, stack, queue, linked list, doubly linked list, hash table, tree etc. So, collection object size is automatically incremented and decremented.

The primary advantages of collections framework are that it

- **Reduces programming effort** by providing useful data structures and algorithms so you don't have to write them yourself.
- **Increases performance** by providing high-performance implementations of useful data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be easily tuned by switching implementations.
- **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth.
- **Reduces the effort required to learn APIs** by eliminating the need to learn multiple ad hoc collection APIs.
- **Reduces the effort required to design and implement APIs** by eliminating the need to produce ad hoc collections APIs.
- **Fosters software reuse** by providing a standard interface for collections and algorithms to manipulate them.

In how many formats can we collect objects?

We can collect objects in 2 ways

1. In **array format** – in this format object does not have identity
2. In **(key, value) pair format** - in this format object has identity

Storing Employee data in different format

Array format

7279	Hari Krishna	Naresh Technologies	Java

Key, Value pair format

eno	7279
ename	Hari Krishna
working with	Naresh Technologies
teaches	Java

In the first format data does not have identity, so user may interpret it wrongly. In the second format data has identity, so it is very clear for the user to understand.

Types of collection classes

Based on above discussion we can conclude that we should have 2 types of collection classes

1. To collect objects in array format –without identity.
2. To collect object in (key, value) pair format –with identity.

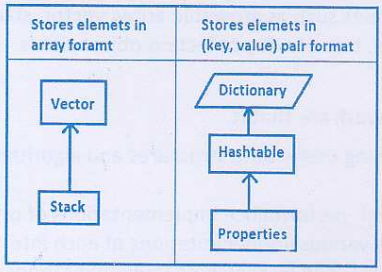
Legacy and collection framework classes

In Java 1.0 version SUN only introduced 2 classes to collect objects in above formats, they are

a. **Vector**: It stores objects in array format
b. **Hashtable**: It stores objects in (key, value) pair format

In addition to above 2 classes we have one more class called **Stack,** it is the subclass of Vector class. It is used to store objects and to retrieve them in **L**ast **I**n **F**irst **O**ut (**LIFO**) fashion.

Below diagram shows relation between these classes.



The drawback of Vector and Hashtable classes

These two classes were created as thread-safe classes, means all the methods in these two classes are declared as synchronized. Hence for every method call on these two objects either for adding or removing elements, JVM locks and unlocks these two objects. So in Single thread model application it leads to performance issue. In Single thread model application execution is sequential so there is no data corruption, hence no need locking. To solve this performance issue problem in Java 1.2 SUN introduced non-thread safe classes as an alternative to above two classes.

Collection Framework

In *Java 1.2 version* more number of collection classes are added. These classes are added to include more features and also to solve the limitations of the above classes – Vector and Hashtable. So in Java 1.2 version new classes were added as alternative to these two classes they are – ArrayList and HashMap respectively.

Note:

From Java 1.2 version all collection classes are collectively called as collection framework. And the collection classes available from Java 1.0 version are called as legacy classes.

Types of Collection Framework Hierarchy

The collection framework is divided into two hierarchies to store objects in array format and (key, value) pair format, they are:

1. **Collection** hierarchy
2. **Map** hierarchy

Collection hierarchy classes collect object in **array format**, it is the root interface of all those classes. **Map** hierarchy classes collect object in **(key, value) pair format**, it is the root interface of all those classes.

Common terminology used in this chapter

Before learning more points on collection framework let us first learn common terminology used in this chapter

collection of objects

The collection object that contains other normal class objects is called collection of objects.

collection of collections

The collection object that contains other collection objects is called collection of collections.

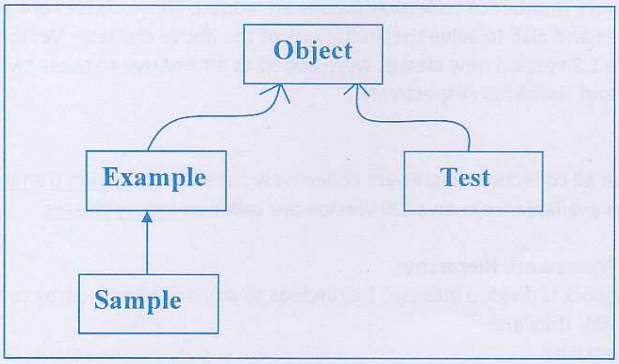
collection of maps

The collection object that contains map objects is called collection of maps.

Element	- means **object**
Entry	- means **(key, value) pair**
Homogeneous elements	- same class objects
Heterogeneous elements	- different class objects
Unique elements	- different class objects or same class objects with different state / reference
Duplicate elements	- same class objects with same state /reference

Note: Unique and duplicate elements is actually decided by "==" operator and equals() method returned value. If equals method returns true, those two objects are considered as duplicate (same objects), else they are considered as unique (different objects).

Consider below hierarchy



In the below list find out all above four types of objects
+ Same type of objects are called homogeneous objects / elements
+ Different type of objects are called heterogeneous objects / elements.

Homogeneous objects

```
Example e1 = new Example();
Example e2 = new Example();
Sample  s1 = new Sample();
Sample  s2 = new Sample();
```

Heterogeneous objects

```
Example e = new Example();
Test    t = new Test();
String  s = new String();
Integer i = new Integer();
```

+ If objects have same state/reference, they are considered as duplicate objects.
+ Else they are considered as unique objects.
+ They can be homogeneous or heterogeneous.

Based on *state* you decide the objects type

Homogeneous **duplicate** objects

```
Example e1 = new Example();
Example e2 = new Example();
```

Homogeneous **unique** objects

```
Example e1 = new Example(5, 6);
Example e2 = new Example(7, 8);
```

Heterogeneous **unique** objects

```
Example e = new Example();
Test    t = new Test();
```

In the below list, find out the above type of objects

```
new String("a");
new String("a");
new String("a");
```

```
new String("a");
new String("b");
new String("c");
```

```
new Integer(10);
new Double(10);
new String("a");
```

In the next sections you will learn Collection **and** Map **hierarchy classes**

Collection hierarchy

Collection hierarchy classes are divided into three categories - **Set, List,** and **Queue.**

Using Collection hierarchy classes we can collect **unique** and **duplicate objects** in array format.

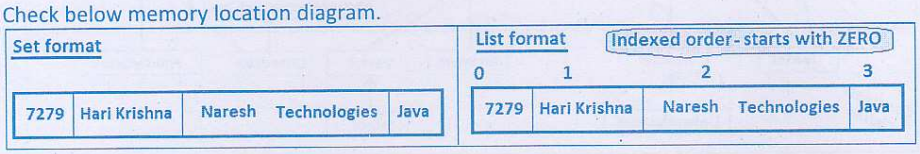
Set is a **unique collection,** it does ***not*** allow duplicate elements.
Set is the super interface for all unique collection classes.

List is a **duplicate collection,** it allows duplicate elements.
List is the super interface for all duplicate collection classes.

Unique collection – Set – is an **unsorted** and **unordered**, means it stores elements without index, where as

Duplicate collection – List – is **unsorted** and **indexed ordered,** means it stores each element with index exactly same as array in insertion order.

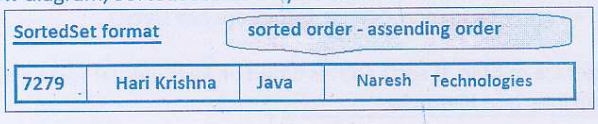
Check below memory location diagram.



As you can noticed in the above diagram, Set does not store elements in indexed or sorted order, whereas List stores elements in indexed order but not in sorted order.

SortedSet is a sub interface of Set that **stores elements in sorted order** either in ascending or descending order based on the object's natural sorting order.

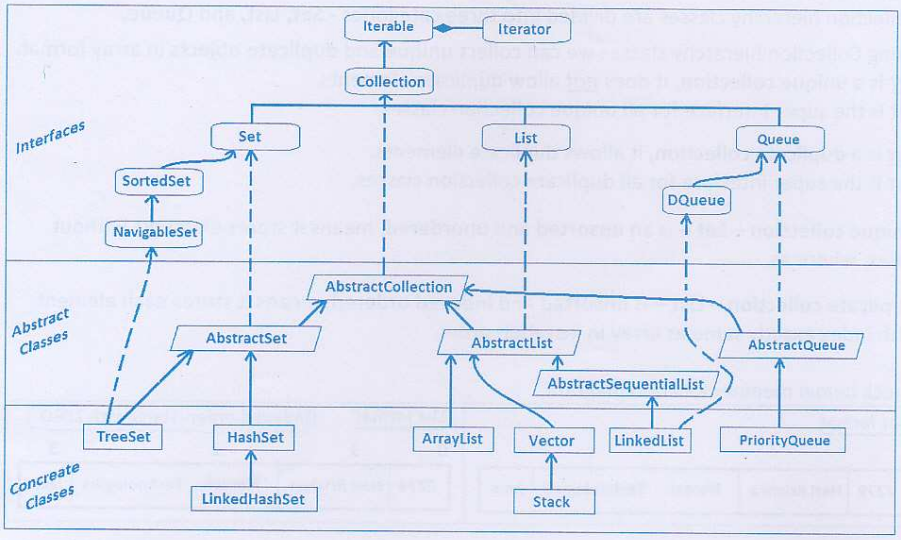
Check below diagram, SortedSet memory location format



NavigableSet is a sub interface of SortedSet. It is added in Java 6 to add more navigation methods to sorted set. The methods are *lower, floor, ceiling*, and *higher* return elements respectively less than, less than or equal, greater than or equal, and greater than a given element, returning *null* if there is no such element.

Queue is a root interface of all types of queues. Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations. Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. We can create different types of queues like **1. *Priority Queues***, which order elements according to a supplied comparator, or the elements' natural ordering, and **2. *LIFO queues*** (or stacks) which order the elements LIFO (last-in-first-out). In a **3. FIFO queue**, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules.
Every Queue implementation must specify its ordering properties.

Collection interface classes' hierarchy

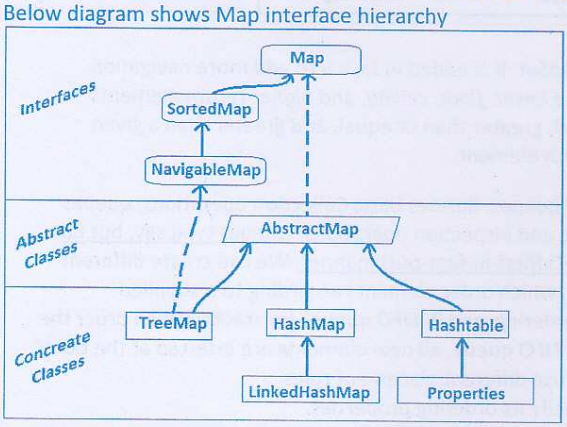


In the above hierarchy, *Vector and Stack* classes are available since Java 1.0, *LinkedHashSet* class is available since Java 1.4, *Queue* is available since Java 5, and *NavigableSet* is available since Java 6, and all other remaining classes and interfaces are available since Java 1.2 version.

Map hierarchy

Map hierarchy classes are used to collect elements in (key, value) pair format.
In a map, *keys* should be unique and *values* can be duplicated.
Each (*key, value*) is called an entry. In map by default entries are not sorted.

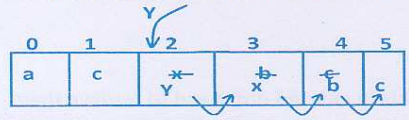
SortedMap is the sub interface of Map. It sorts entries based on keys natural sorting order.
Below diagram shows Map interface hierarchy



In this hierarchy, *Hashtable and Properties* classes are available since Java 1.0, *LinkedHashMap* class is available since Java 1.4, and *NavigableMap* is available since Java 6, and all other remaining classes and interfaces are available since Java 1.2 version.

Different project Scenarios- we choose above collection classes

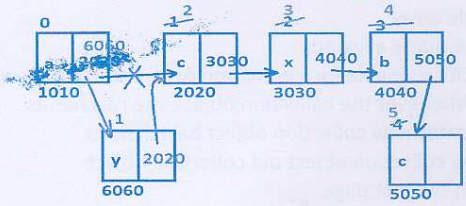
- To collect objects in array format we choose *Collection hierarchy* classes.
- To collect objects in (key, value) pair format we should choose Map hierarchy classes.
- To collect unique elements we must choose *Set implemented* classes.
- To collect unique and/or duplicate elements in indexed order we must choose *List implemented* classes.
- To retrieve elements in FIFO manner we choose *Queue implemented* classes.
- To store heterogeneous unique elements with no order we must choose *HashSet*.
- To store heterogeneous unique elements in insertion order we must choose *LinedHashSet*.
- To store elements in their natural sorting order we must choose *TreeSet*. TreeSet allows only Homogeneous Comparable unique elements.
- To store and retrieve duplicate elements in random access we must choose *ArrayList* or *Vector*. These two classes functionality is same except, Vector is thread-safe. So, in Single Thread model application we should choose ArrayList, in multithreading model application if list object is modifying concurrently we should choose Vector.
- The ***drawback of ArrayList and Vector*** is "They give less performance if we perform insertions and deletions in middle", because for every insertion of new element, all remaining elements must be moved right hand side, and for every element deletion, elements must be moved to left hand side. As shown in the below diagram.



ArrayList or Vector object

- To solve this performance issue we must choose LinkedList. It gives high performance in inserting or deleting elements in middle. It internally uses linked list data structure, so there will not be any data movements, instead we will have only changing links, and the indexes.

  Check below diagram for its working functionality



- To store and retrieve elements in FILO manner we should choose *Stack*.

- To store unique entries, we must choose *HashMap*.
- To store unique entries in insertion order, we must choose *LinkedHashMap*.
- To store unique entries in sorted order, we must choose *TreeMap*. It only allows homogeneous unique entries with Comparable type keys.

- Hashtable and HashMap both works similar, the only difference is Hashtable is thread-safe.

- Properties class is used to store properties permanently.

Let us understand each class separately also let us understand important interview questions
Interview Questions you will face from this chapter

1. What is the implemented data structure?
2. What is the default capacity, and incremental capacity?
3. What type of elements allowed to add?
4. In which order elements are preserved and retrieved?
5. Is null allowed, if so how many?

List implemented classes
We should choose list implemented classes to store heterogeneous unique or duplicate elements in indexed order. It has four concrete classes, they are
ArrayList, Vector, Stack, LinkedList.

To store duplicate objects Vector class is already available, then why ArrayList and LinkedList classes are given?
Read below points.

ArrayList, Vector
We should choose these two classes to store elements in indexed order and to retrieve them randomly, it means retrieving nth element directly without retrieving (n-1) elements, because these two classes are sub classes of ***RandomAccess*** interface.

Specific Functionalities of these two classes

1. duplicate objects are allowed in indexed order
2. heterogeneous objects are allowed
3. insertion order is preserved
4. implemented data structure is Growable array
5. null insertion is possible, more than one null is allowed.
6. Initial capacity is 10, incremental capacity is double for Vector and ArrayList uses below formula (currentCapacity * 3 / 2 + 1). Whenever the collection object size reaches its max capacity, that class internal API creates new collection object based on its incremental capacity value. Then in new collection object old collection object elements are copied, as we discussed in the first page.

The **main difference** between these two classes is:

Vector object is thread-safe it means synchronized object – so multiple threads cannot modify this object concurrently. It is best suitable in multithreaded model application to get desired result. But in single thread model application it gives poor performance, because for every operation this object should be locked and locked, which useless activity in single thread model application. To solve this performance issue in collection framework ArrayList class is given.

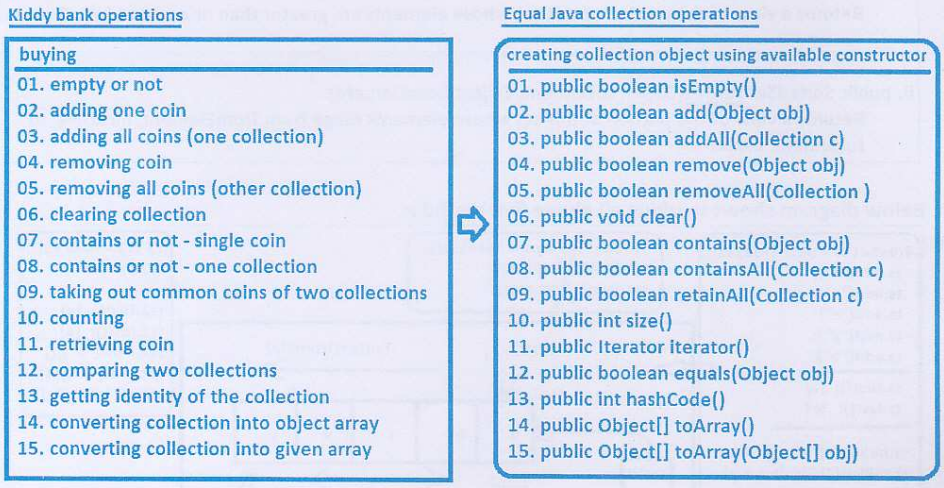
ArrayList object is not thread-safe it means it is not synchronized object. It is best suitable in multithreaded model application, also in multithreaded model application if you ensure there is not data corruption.

Limitation of both classes

Inserting and removing elements in middle is costlier operation. Because for every insert operation elements must move to right and for every delete operation elements must move to left from that location.

Collection interface methods - 15:

In collection interface below methods are defined commonly for all collections to perform different operations



Set interface methods - 15:

Why add(Object) method return type is boolean?

In Collection interface **add()** method is defined commonly for both **Set** and **List.** So its return type should be boolean return **false** if duplicate element is added.

Answer my question, is Set interface has any specific methods?

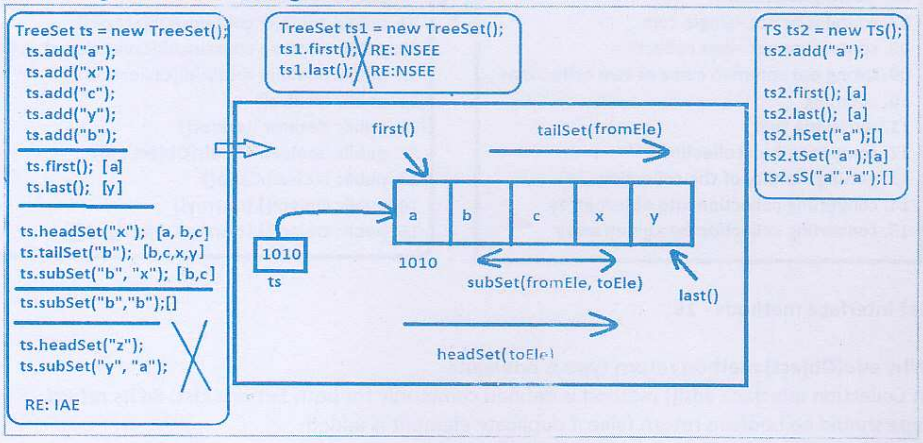
No, Set interface have same above 15 methods because its specific functionality – maintaining unique elements – is implemented with the same add(Object) method as explained above.

SortedSet interface methods -21:

In addition to above 15 methods this interface has specific 6 methods based on sorting functionality, they are:

1. public Comparator comparator()
   Returns the comparator used to order the elements in this set, or null if this set uses the natural ordering of its elements
2. public Object first()
   retruns first (lowest) element
3. public Object last()
   retruns last (highest) element
4. public SortedSet headSet(Object toElement)
   Returns a view of the portion of this set whose elements are strictly less than toElement
5. public SortedSet tailSet(Object fromElement)
   Returns a view of the portion of this set whose elements are greater than or equal to fromElement.
6. public SortedSet subSet(Object toElement, Object fromElement)
   Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.

Below diagram shows working all above five methdos



List interface methods – 25

In addition to Collection interface 15 methods List interface has 10 more specific methods based in indexed order.

```
1. public void add(int index,Object element)      => inserts element at given index
2. public boolen addAll(int index,Collection c)   => inserts Collection at given index
3. public Object get(int index)                   => returns element at given index, it will not remove that element
                                                     from collection
4. public Object set(int index,Object element)    => replaces element at given index, and return old element
5. public Object remove(int index)                => deletes element at given index, and return old element

6. public int indexOf(Object element)             => retruns first occurence index of given object
7. public int lastIndexOf(Object element)         => retruns last occurence index of given object

8. public List subList(int start, int end)        => retruns sub list in the given range

9. public ListIterator listIterator()             => retruns ListIterator on this collection
10. public ListIterator listIterator(int index)   => retruns ListIterator on this collection from given index
```

The more specific operations we can do on List object are: insert, retrieve, replace, and remove at the given index.

ArrayList class Constructors

ArrayList class has below constructors to create its object

```
public class ArrayList extends AbstractList implements List RandomAccess, Cloneable, Serializable
{
  public ArrayList();
      Constructs an empty list with an initial capacity of ten.
  public ArrayList(int capacity)
      Constructs an empty list with the specified initial capacity.
  public ArrayList(Collection c)
      Constructs a list containing the elements of the specified collection, in the order they are returned by the
      collection's iterator.

  //Specific methods
  public void trimToSize()
      Trims the capacity of this ArrayList instance to be the list's current size. An application can use this operation to
      minimize the storage of an ArrayList instance.
  public void ensureCapacity(int minCapacity)
      Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of
      elements specified by the minimum capacity argument.
  protected void removeRange(int fromIndex, int toIndex)
      Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
      Shifts any succeeding elements to the left (reduces their index). This call shortens the list by (toIndex - fromIndex)
      elements.
}
```

Below program shows adding, retrieving and removing elements from ArrayList.

```
//ArrayListDemo.java
import java.util.ArrayList;

public class ArrayListDemo
{
        public static void main(String args[])
        {
                // Creating an ArrayList
                ArrayList al = new ArrayList();
                System.out.println("Initial size of ArrayList: " + al.size());

                //Add elements to the ArrayList
                al.add("Red");
                al.add("Green");
                al.add("Blue");
                al.add("Pink");
                al.add("Orange");

                System.out.println("\nSize of ArrayList after additions: " + al.size());

                //Display the ArrayList
                System.out.println("\nContents of ArrayList After add: " + al);

                // Remove 4th index element from ArrayList
                al.remove(4);
                System.out.println("\nContents of ArrayList after remove index: " + al);

                // Remove "Pink" element from ArrayList
                al.remove("Pink");
                System.out.println("\nContents of ArrayList after remove object: " + al);
                System.out.println("\nSize of ArrayList after deletions: " + al.size());

                //retrieving 1st index element
                String alElement        = (String)(al.get(1));
                System.out.println("alElement :"+alElement);

                //inserting at 2nd index
                al.add(2, alElement +" rose" );
                System.out.println(al);
        }
}
```

Vector class constructors and methods

As being a Legacy class Vector class has below specific methods and constructors

```
public class Vector extends AbstractList implements List, RandomAccess, Cloneable,
Serializable
{
        public Vector()
                Constructs an empty vector so that its internal data array has size 10.

        public Vector(Collection c)
                Constructs a vector containing the elements of the specified collection, in the
                order they are returned by the collection's iterator.

        public Vector(int initialCapacity)
                Constructs an empty vector with the specified initial capacity.

        public Vector(int initialCapacity, int incrementalCapacity)
                Constructs an empty vector with the specified initial capacity and capacity
                increment.

        //specific methods
        public synchronized boolean isEmpty()
        public synchronized void addElement(Object o)
        public synchronized Object elementAt(int index)
        public synchronized Object firstElement()
        public synchronized Object lastElement()
        public synchronized void insertElementAt(Object o, int index)
        public synchronized void setElementAt(Object o, int index)
        public synchronized boolean removeElement(Object o)
        public synchronized void removeAllElements()
        public synchronized void removeElementAt(int index)
        protected synchronized void removeRange(int index, int index)
        public synchronized int size()
        public synchronized int capacity()
        public synchronized void ensureCapacity(int capacity)
        public synchronized void trimToSize()
        public synchronized void setSize(int size)

        public Enumeration elements()

        public synchronized void copyInto(Object[])

        public synchronized Object clone()
}
```

Below program shows storing and removing elements from Vector class

```
//VectorDemo.java
import java.util.Vector;
public class VectorDemo{
        public static void main(String args[]){
                Vector v = new Vector();
                for(int i = 0 ; i <= 9 ; i++){
                        v.addElement(new Integer(i*10));
                }

                v.removeElementAt(0);
                v.removeElementAt(1);

                for(int i = 0 ; i < v.size() ; i++){
                        System.out.println(v.elementAt(i));
                }
        }
}
```

Below program shows changing Vector capacity

```
//Address.java
class Address {

        String hno, street, city, ph;

        Address(String hno, String street, String city, String ph) {

                this.hno = hno;
                this.street = street;
                this.city = city;
                this.ph = ph;
        }
}
//Customer.java
class Customer {

        String name;   int age;   Address address;

        Customer(String name, int age, Address address)    {
                this.name       = name;
                this.age        = age;
                this.address    = address;
        }
```

```
        public String toString(){
                return (        "Name: " + name + "\nAge: " + age +
                                "\nH.No: " + address.hno + "\nStreet: " + address.street +
                                "\nCity: " + address.city+"\nph"+address.ph
                        );
        }
}

//VectorCapacitySizeDemo.java
import java.util.Enumeration;
import java.util.Vector;

public class VectorCapacitySizeDemo {
        public static void main(String[] args)  {
                Vector v = new Vector(3);

                System.out.println("Intial Capacity and Size of Vector is..");
                System.out.println("Capacity: " + v.capacity());
                System.out.println("Size       : " + v.size());

                System.out.println();

                Customer c1 = new Customer("Newton", 30 ,
                                                new Address("2-3-102/1",
                                                "Ameerpet","Hyderabad","12345"));
                Customer c2 = new Customer("Einstene",23,
                                                new Address("1-10-1022/3",
                                                "Kukatpally","Hyderabad","23345"));
                Customer c3 = new Customer("Clinton",31,
                                                new Address("1-3-32",
                                                        "Amberpet","SecBad","33345"));
                Customer c4   =new Customer("Ken Thomposn",31,
                                                new Address("1-3-32",
                                                        "Amberpet","SecBad","33345"));
                Customer c5 = new Customer("Edward Shepered",31,
                                                new Address("1-3-32",
                                                        "Amberpet","SecBad","33345"));
                Customer c6 = new Customer("Michael Slater",31,
                                                new Address("1-3-32",
                                                        "Amberpet","SecBad","33345"));
                Customer c7 = new Customer("Justin Langer",31,
                                                new Address("1-3-32",
                                                        "Amberpet","SecBad","33345"));
```

```
                v.add(c1);
                v.add(c2);

                System.out.println("Reached  its Capacity....Or not?");
                System.out.println("Capacity: " + v.capacity());
                System.out.println("Size       : " + v.size());
                System.out.println();

                v.add(c3);
                System.out.println("Reached  its Capacity....Or not?");
                System.out.println("Capacity: " + v.capacity());
                System.out.println("Size       : " + v.size());
                System.out.println();

                v.add(c4);
                System.out.println("Reached  its Capacity....Or not?");
                System.out.println("Capacity: " + v.capacity());
                System.out.println("Size       : " + v.size());
                System.out.println();

                v.add(c5);
                v.add(c6);
                System.out.println("Reached  its Capacity....Or not?");
                System.out.println("Capacity: " + v.capacity());
                System.out.println("Size       : " + v.size());
                System.out.println();

                v.add(c7);
                System.out.println("Capacity Increased...and Size of Vector Altered..");
                System.out.println("Capacity: " + v.capacity());
                System.out.println("Size       : " + v.size());
                System.out.println();

                System.out.println("The customer labels are: ");
                System.out.println();

                Enumeration e = v.elements();
                while(e.hasMoreElements())
                {
                        System.out.println(e.nextElement());
                        System.out.println();
                }
        }
}
```

Stack class constructors and methods

The Stack class represents a last-in-first-out (LIFO) stack of objects. It extends class Vector with five operations that allow a vector to be treated as a stack. The usual *push* and *pop* operations are provided, as well as a method to *peek* at the top item on the stack. When a stack is first created, it contains no items.

public class Stack extends Vector{

public Stack()
Creates an empty Stack.

public boolean empty()
Tests if this stack is empty.

public Object push(Object o)
Pushes an item onto the top of this stack.

public synchronized Object pop()
Removes the object at the top of this stack and returns that object as the value of this method.

public synchronized Object peek()
returns the object at the top of this stack without removing it from the stack.

public synchronized int search(Object o)
Returns the 1-based position where an object is on this stack. If the object o occurs as an item in this stack, this method returns the distance from the top of the stack of the occurrence nearest the top of the stack; the topmost item on the stack is considered to be at distance 1. The equals method is used to compare o to the items in this stack.

}

Below program shows java based stack operations

```
//StackDemo.java
import java.util.Stack;
public class StackDemo{
    public static void main(String args[]){
        Stack st        = new Stack();
        st.push(new Integer(10));          st.push(new Integer(20));
        st.push(new Integer(30));          st.push(new Integer(40));

        System.out.println(st);
        System.out.println(st.pop());
        System.out.println(st);
        System.out.println(st.peek());
        System.out.println(st);
        System.out.println(st.search(new Integer(10)));
    }
}
```

Set Implemented Classes:

public class HashSet extends AbstractSet implements Set, Cloneable, Serializable{

public HashSet()
Constructs a new, empty set; the backing HashMap instance has default initial capacity (16) and load factor (0.75).

public HashSet(Collection c)
Constructs a new set containing the elements in the specified collection. The HashMap is created with default load factor (0.75) and an initial capacity sufficient to contain the elements in the specified collection.

public HashSet(int initialCapacity)
Constructs a new, empty set; the backing HashMap instance has the specified initial capacity and default load factor (0.75).

public HashSet(int initialCapacity, float loadFactor)
Constructs a new, empty set; the backing HashMap instance has the specified initial capacity and the specified load factor.

}

public class LinkedHashSet extends HashSet implements Set, Cloneable, Serializable{

public LinkedHashSet()
public LinkedHashSet(Collection c)
public LinkedHashSet(int initialCapacity)
public LinkedHashSet(int initialCapacity, float loadFactor)

}

public class TreeSet extends AbstractSet implements SortedSet, Cloneable, Serializable{

public TreeSet()
Constructs a new, empty tree set, sorted according to the natural ordering of its elements. All elements inserted into the set must implement the Comparable interface.

public TreeSet(Collection c)
Constructs a new tree set containing the elements in the specified collection, sorted according to the natural ordering of its elements

public TreeSet(SortedSet s)
Constructs a new tree set containing the same elements and using the same ordering as the specified sorted set.

public TreeSet(Comparator comparator)
Constructs a new, empty tree set, sorted according to the specified comparator. It should be used to supply custom sorting order for comparable objects and to store non-comparable objects.

}

```
// Below program explains all Set implemented concrete classes, and their functionality

//it is a dummy class, created to store its objects in collection
//Example.java
public class Example{
        int x = 10;
        int y = 20;
}

//SetClassesDemo.java
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.TreeSet;

public class SetClassesDemo  {
        public static void main(String[] args) {
                //set objects creation
                HashSet         hs      = new HashSet();
                LinkedHashSet   lhs     = new LinkedHashSet();
                TreeSet         ts      = new TreeSet();
                //printing initial size of collection objects
                System.out.println("hs      length: "+hs.size());
                System.out.println("lhs     length: "+lhs.size());
                System.out.println("ts      length: "+ts.size());
                System.out.println();

                //adding elements to hs
                System.out.println("is 20 added: "+  hs.add( Integer.toString(20) ) );
                System.out.println("is a added: "+   hs.add( new Character('a') )   );
                System.out.println("is b added: "+   hs.add( new Character('b') )   );
                System.out.println("is abc added: "+ hs.add( "abc"  )               );
                System.out.println("is Abc added: "+ hs.add( "Abc" )                );
                System.out.println("is abc added: "+ hs.add( "abc" )                );
                System.out.println("is abc added: "+ hs.add( new String("abc") )    );
                System.out.println("is Example added: "+hs.add( new Example() ) );
                System.out.println("is Example added: "+hs.add( new Example() ) );

                //adding nulls to hs
                System.out.println("is null added: "+hs.add(null));
                System.out.println("is null added: "+hs.add(null));
                System.out.println();

                //printing hs modified size and elements
                System.out.println(hs.size());
                System.out. println("hs : " + hs);
```

```
        //adding elements to lhs
        System.out.println("is 20 added: "+lhs.add( Integer.toString(20)));
        System.out.println("is a added: "+lhs.add(new Character('a')));
        System.out.println("is b added: "+lhs.add(new Character('b')));
        System.out.println("is abc added: "+lhs.add("abc"));
        System.out.println("is Abc added: "+lhs.add("Abc"));
        System.out.println("is abc added: "+lhs.add("abc"));
        System.out.println("is abc added: "+lhs.add(new String("abc")));
        System.out.println("is Example added: "+lhs.add(new Example()));
        System.out.println("is Example added: "+lhs.add(new Example()));

        //adding nulls to lhs
        System.out.println("is null added: "+lhs.add(null));
        System.out.println("is null added: "+lhs.add(null));
        System.out.println();

        //printing lhs modified size and elements
        System.out.println(lhs.size());
        System.out. println("lhs : " + lhs);

        //adding homogeneous elements to ts
        System.out.println("is abc added: "+ts.add("abc"));
        System.out.println("is xyz added: "+ts.add("xyz"));
        System.out.println("is bbc added: "+ts.add("bbc"));
        System.out.println("is pqr added: "+ts.add(new String("pqr")));

        //adding heterogeneous element
        System.out.println("is Integer added: "+ts.add(new Integer(10)));

        //adding null to ts
        System.out.println("is null added: "+ts.add(null));
        System.out.println("is null added: "+ts.add(null));

        //printing ts modified size and elements
        System.out.println(ts.size());
        System.out. println("ts : " + ts);

        //Q) when can we add null to TreeSet?
        //If it is solo TreeSet we can add null, check the below code
        TreeSet solots = new TreeSet();
        solots.add(null);

        System.out.println(solots.size());
        System.out. println("solots : " + solots);
    }
}
```

Note: From Java 7 onwards *we cannot* add *single null also.*

This solots code leads to NPE.

Three types of cursors to retrieve elements from all types of collection objects

We have three types of cursor objects for retrieving elements they are:

1. Enumeration
2. Iterator
3. ListIterator

All above three are interfaces. Their subclasses are created by SUN as inner classes inside collection classes. These subclasses objects are returned via factory methods. These factory methods are also defined inside collection classes. So these factory methods must be called with collection object from which we want to retrieve elements.

Those Factory methods are:

- to retrieve Enumeration object
  public Enumeration ***elements()***
  - This method is defined inside both Vector and Hashtable classes

- to retrieve Iterator object
  public Iterator ***iterator()***
  - It is defined in Collection interface. So it can be called on all collection objects

- to retrieve ListIterator object
  public ListIterator ***listIterator()***
  public ListIterator listIterator(int index)
  - These methods are defined in List interface. So these methods can be called only List type collection objects.

Q) What are the operations we must perform using above three cursor objects?

1. Checking is element available in the underlying collection
2. If element available retrieve that element reference.

So to perform above two tasks these three cursor objects must have minimum two methods

1. For checking element available or not
2. For returning the element reference from collection and moving cursor to next location

All above cursor objects cursor is initially pointing to before first location of the collection. Below is the cursor object structure with given collection object "c".

We must call nextElement()/next() methods till cursor reaches last element.

Understanding *Enumeration* interface

Enumeration is a legacy interface defined in Java 1.0 to retrieve elements from legacy collections Vector and Hashtable. It has below two methods to retrieve elements:

```
public interface Enumeration{
        public boolean hashMoreElements()
          - It checks whether elements exist or not. If not existed returns false.

        public Object nextElement()
          - It returns the next element. If no element is available it throws an exception
                                                    "java.util.NoSuchElementException"
}
```

Q) Who implement above two methods?

SUN in Vector and Hashtable classes as inner class.

Q) How can we obtain Enumeration object on Vector and Hashtable?

- Vector class has below factory method
  public Enumeration **elements**()
- Hashtable class has below two factory methods
  1. public Enumeration **keys**()
     - it returns enumeration on all keys
  2. public Enumeration **elements**()
     - it returns enumeration on all values

Q) How can we obtain Enumeration on collection framework classes?

A) We cannot obtain Enumeration on collection framework classes directly, we must use "*Collections*" class static method "*enumeration()*". Its prototype is:

public static Enumeration **enumeration(Collection c)**

Q) Write a program to retrieve elements from *Vector* object by using Enumeration.

```
Vector v = new Vector();
v.add(10);
v.add("a");

Enumeration e = v.elements();

while( e.hasMoreElements() ){
     Object obj = e.nextElement();
     System.out.println(obj);
}
```

Q) Write a program to retrieve elements from *ArrayList* object by using Enumeration.

```
ArrayList  al = new ArrayList();
al.add(10);
al.add("a");

Enumeration e = Collections.enumeration( al );

while( e.hasMoreElements() ){
     Object obj = e.nextElement();
     System.out.println(obj);
}
```

Execution flow of above two programs

- Enumeration object is created, stored in e variable, it is pointing to before first element.
- While loop is iterated till *e.hasMoreElements()* returns false. It returns false when cursor reaches end of the collection elements.
- In every iteration *e.nextElement()* returns the next element in the collection and moves the cursor to this next element location..

Understanding *Iterator* interface

Iterator is a Collection Framework interface defined in Java 1.2. It is a cursor object used to retrieve elements from all Collection type objects List, Set and Queue including Vector. It is an alternative of Enumeration object and replacement of "for" loop on collection objects.

List objects has index so we can retrieve elements by using for loop as shown below

Retrieving elements from List object

```
ArrayList al = new ArrayList();

al.add("a");
al.add("b");
al.add("c");
al.add("a");
```

Since ArrayList stores objects with **index** we can retrieve elements using **for loop** as shown below,

```
for(int i = 0 ; i < al.size() ; i++){
        SopIn(al.get(i));
}
```

But how can we retrieve elements from Set objects since they do not have index?
We do not have *get* method in *Set*, check below code

Retrieving elements from Set object

```
LinkedHashSet lhs = new LinkedHashSet();

lhs.add("a");
lhs.add("b");
lhs.add("c");
lhs.add("a");
```

Size() method is available but get() method is not available, then how can we retrieve elements from Set

```
for(int i = 0 ; i < lhs.size() ; i++){
        SopIn( ? );
}
```

Iterator is only the option we have to retrieve elements from Set objects

Iterator interface provides below three methods for retrieving elements from collection

```
public interface Iterator{

  public boolean hasNext()
        Returns true if the iteration has more elements.

  public Object next()
        Returns the next element in the iteration, and moves cursor to next location.

  public void remove()
        Removes from the underlying collection the last element returned by the Iterator.
}
```

Q) Who does implement above three methods?
SUN in all Collection subclasses as inner class.

Q) How can we obtain Iterator object?
In Collection interface we have below method to obtain Iterator object

public Iterator iterator();

Below statement we should use to obtain Iterator object on a given collection object **c**.

Iterator itr = c.iterator();

Q) Write a program to retrieve elements from *ArrayList* object by using Iterator.

```
ArrayList al = new ArrayList();

al.add("abc");
al.add("bbc");
al.add("cbc");

Iterator itr = al.iterator();

while(itr.hasNext()){
      Object ele = itr.next();
      System.out.println(ele);
}
```

Q) Write a program to retrieve elements from *LinkedHashSet* object by using Iterator.

```
LinkedHashSet lhs = new LinkedHashSet();

lhs.add("abc");
lhs.add("bbc");
lhs.add("cbc");

Iterator itr = lhs.iterator();

while(itr.hasNext()){
      Object ele = itr.next();
      System.out.println(ele);
}
```

ClassCastException while using collection elements:
next() method returns every element as java.lang.Object type from collection. So we must cast the returned objects to their own type by using cast operator to call that returned object's specific members. So in this code we should use **instanceof** operator to avoid **CCE**.

Below program shows retrieving elements from Set implemented classes using Iterator.
This program shows adding string and integer objects to LHS and retrieving and printing all string objects in uppercase and remaining objects as it is.

```
//IteratorDemo.java
import java.util.LinkedHashSet;
import java.util.Iterator;

public class IteratorDemo  {

        public static void main(String[] args) {

                LinkedHashSet lhs = new LinkedHashSet();

                lhs.add("abc");
                lhs.add(10);
                lhs.add(20);
                lhs.add(40);
                lhs.add("bbc");

                Iterator lhsIterator = lhs.iterator();

                while(lhsIterator.hasNext()) {

                        Object obj = lhsIterator.next();

                        if(obj instanceof String)  {

                                String str = ( (String)obj ).toUpperCase();
                                System.out.println("Modified String :"+ str );

                        }
                        else{
                                System.out.println("Object :"+obj);
                        }

                }//while closed
        }
}
```

Iterator rules

Rule #1: While retrieving elements, in while loop **collection object structure should not be modified** by either adding element or by removing element using collection methods. It leads to **java.util.ConcurrentModificationException.** This behavior of Iterator is called ***fail-fast***.

Rule #2: next() method should not be called on empty collection or empty location, it leads RE: **java.util.NoSuchElementException**

Rule #3: remove() method must be called only after next() method call and also only once, violation leads to **java.lang.IllegalStateException**

Below program shows Iterator rules

```
//IteratorMethodsRulesDemo.java
import java.util.LinkedHashSet;
import java.util.Iterator;

public class IteratorMethodsRulesDemo {
        public static void main(String[] args) {
                LinkedHashSet lhs = new LinkedHashSet();
                lhs.add("a");
                lhs.add("b");
                lhs.add("c");

                Iterator itr = lhs.iterator();

                //itr.remove();  RE: java.lang.IllegalStateException

                Object o1 = itr.next();
                System.out.println(o1);//a

                itr.remove(); //a is removed

                //lhs.add("d");  // this statement causes CME when next() is called

                Object o2 = itr.next(); // RE: java.util.ConcurrentModificationException
                System.out.println(o2);//b

                Object o3 = itr.next();
                System.out.println(o3);//c

                //Object o4 = itr.next(); //RE: java.util.NoSuchElementException

                itr.remove();//c is removed

                lhs.add("d"); //it is correct and d is added

                System.out.println(lhs);//[b, d]
        }
}
```

Q) What are the differences between *Enumeration* and *Iterator*?

Iterator takes the place of Enumeration in the Java collections framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

Below is the complete list of differences

Diff #1:

- Enumeration is legacy interface.
- Iterator is collection framework interface

Diff #2:

- Enumeration method names are big
- Iterator method names are short and it has an additional method remove()

Diff #3:

- Enumeration cannot remove elements
- Iterator can remove elements with well defined semantics

Diff #4:

- Enumeration is not fail-fast, so we can modify collection while enumerating.
- Iterator is fail-fast, so we cannot modify collection while iterating.

Diff #5:

- Enumeration cannot be applied on collection framework classes directly
- Iterator can be applied on all types of collection classes including legacy classes as these classes are also subclass of Collection and Map.

ListIterator

It is a bidirectional cursor object used to retrieve elements only from List implemented collection objects- ArrayList, Vector, Stack, LinkedList. It is a sub interface of Iterator.

Obtaining its object

In "List" interface we have below two methods to obtain its object

- public ListIterator listIterator()
- public ListIterator listIterator(int index)

➔ first method iterates starts from begin to end
➔ second method iterates starts from given index to end

Q) What are the differences between Iterator and ListIterator?

The main difference is:

Diff #1:

- Iterator is unidirectional – means elements are retrieved only in forward direction.
- ListIterator is bidirectional – means elements are retrieved in both directions.

The other differences are:

Diff #2:

- Iterator is a super interface
- ListIterator is a sub interface of Iterator

Diff #3:

- Iterator can be applied on both Set and List implemented classes
- ListIterator can only be applied on List implemented classes, because it works with index.

Diff #4:

- Iterator can only allow us to retrieve and remove elements
- ListIterator can allow us to retrieve, remove, insert, and also replace elements.

ListIterator methods

It has below special methods to perform above operations:

It has below methods to retrieve elements in forward direction.
These methods are inherited from Iterator interface

1. public boolean **hasNext**()
2. public Object **next**()

It has below methods to retrieve elements in backward direction

3. public boolean **hasPrevious**()
4. public Object **previous**()

For removing current returned element

5. public void **remove**()

For inserting element

6. public void **add**(Object obj)
   - It inserts the given element in the next location of current returned element.

For replacing current element

7. public void **set**(Object obj)
   - It replaces current returned element with given element.

Write a program to create ArrayList object with
- **3 String objects and 3 Integer objects**
- **while iterating replace all String objects with their uppercase**
- **Insert the number 20 after the first Integer object**

```
//ListIteratorDemo.java
import java.util.*;

class ListIteratorDemo {
        public static void main(String[] args) {
                ArrayList al = new ArrayList();
                al.add("a");
                al.add("b");
                al.add("c");
                al.add(5);
                al.add(6);
                al.add(7);

                System.out.println("AL elements before iteration: "+ al);
                int count = 1;

                ListIterator litr = al.listIterator();
                while (litr.hasNext()){
                        Object obj = litr.next();

                        if (obj instanceof String){
                                String s = (String)obj;
                                litr.set(s.toUpperCase());
                        }
                        else if (obj instanceof Integer){
                                if (count == 1){
                                        litr.add(20);
                                        count++;
                                }
                        }//else if
                }//while

                System.out.println("AL elements after iteration: "+ al);

                //retrieving elements in reverse order
                while (litr.hasPrevious()){
                        Object obj = litr.previous();
                        System.out.println(obj);
                }
        }//main
}//class
```

Understanding the operations performed by addAll(), removeAll(), containsAll() and retainAll() methods in the Collection interface.

- addAll() - performs Union operation
- containsAll() - performs subset operation
- removeAll() - performs subtraction operation
- retainAll() - performs intersection operation

Assume we have two collections c1 and c2
If we call

➔ **c1.addAll(c2)** - all elements of c2 are added to c1.

If c1 is **Set** collection duplicates are *not* added.
If c1 is **List** collection duplicates are also added.

➔ **c1.containsAll(c2)** - It returns true if all elements of c2 available in c1. else it returns false.

➔ **c1.removeAll(c2)** - It removes all elements of c2 from c1. Duplicate elements are also removed.

➔ **c1.retainAll(c2)** - It removes all elements of c1 except c2 elements. Duplicate elements are also retained.

Conclusion:
addAll(), retainAll(), removeAll() methods modifies current collection object based on argument collection object elements.

Difference between Comparable and Comparator interfaces
Comparable and Comparator both are interfaces. They are used to specify elements sorting order in TreeSet and keys in TreeMap.

The difference between these two interfaces
Comparable used to specify natural sorting order of the object.
Comparator is used to specify custom sorting order of the object.

To specify the sorting order by developer these two interfaces have below methods

Comparable method
```
public int compareTo(Object o1)
```

Comparator method
```
public int compare(Object o1, Object o2)
```

Adding custom objects to TreeSet

- TreeSet internally uses Comparable interface method *compareTo()* to compare and sort objects.
- So we can only add Comparable objects to TreeSet.
- Compiler allows to add non-comparable objects to TreeSet because add () method parameter is Object, but JVM throws ClassCastException. In TreeSet class add() method the argument object is cast to Comparable to call compareTo() method on that object.

  So to add custom object we must follow below procedure
  - its class must be subclass of Comparable interface
  - then should override compareTo() method with required sorting order logic. This logic is used as default sorting order to sort that class objects.

TreeSet class sample add() method logic

```
public boolean add(Object obj)
{
        if (! (this.isEmpty()) )
        {
                Comparable c = (Comparable) obj;
                c.compateTo(tsItr.next()); //passing TreeSet elements.
        }
}
```

For more details check running notes.

Below program shows adding Example objects to TreeSet

Case #1: Example class is not deriving from Comparable

```
class  Example {
        int x;
        Example(){}
        Example(int x){this.x = x;}
}

class AddingCustomObjects{
        public static void main(String[] args){
                TreeSet ts = new TreeSet();
                ts.add(new Example());
                //ts.add(new Example());       RE: ClassCastException: Example cannot
                                                    be  cast to java.lang.Comparable
        }
}
```

Case #2: Example class is deriving from Comparable and overriding compareTo().

```
class  Example implements Comparable {

        int x;
        Example(){}
        Example(int x){this.x = x;}

        public int compareTo(Object obj) {
                Example e = (Example)obj;
                return e.x - this.x;
        }

        public String toString(){
                return ""+x;
        }
}

class AddingCustomObjects{

        public static void main(String[] args){

                TreeSet ts = new TreeSet();

                ts.add(new Example());
                ts.add(new Example(20));
                ts.add(new Example());
                ts.add(new Example(5));
                ts.add(new Example(30));
                ts.add(new Example(20));
                ts.add(new Example());

                System.out.println(ts.size());
                System.out.println(ts);
        }
}
```

O/P

Changing default sorting order of objects

```
/*
 * This program demonstrates developing a custom Comparator to show you
 * changing natural order of predefined objects like String and wrapper classes.
 * call this ComparatorDemo.java
 */
import java.util.Comparator;
```

```
import java.util.TreeSet;

class StringComparator implements Comparator {
        public int compare(Object a, Object b) {
                String aStr, bStr;
                aStr = (String) a;
                bStr = (String) b;

                // Reverse the comparison
                return bStr.compareTo(aStr);
        }
}

public class ComparatorDemo {
        public static void main(String args[]) {
                /*
                  - Creating a TreeSet object using no-arg constuctor,
                  - So, elements are sorted according to natural sorting order of adding object
                 */
                TreeSet ts = new TreeSet();
                // Add elements to the TreeSet
                ts.add("C");
                ts.add("A");
                ts.add("B");
                ts.add("E");
                ts.add("F");
                ts.add("D");

                System.out.println("ts object with default comparator :"+ts);

                /*
                  - Creating a TreeSet object using Comparator parameter constructor,
                  - to sort elements according to custom comparator sorting order
                 */

                TreeSet tsc = new TreeSet(new StringComparator());
                tsc.add("C");
                tsc.add("A");
                tsc.add("B");
                tsc.add("E");
                tsc.add("F");
                tsc.add("D");

                System.out.println("tsc object with custom comparator :"+tsc);
        }
```

```
}
```

Adding *non-Comparable* objects to TreeSet by using custom Comparator. StringBuffer is a non-comparable object. Below program shows adding StringBuffer objects to TreeSet.

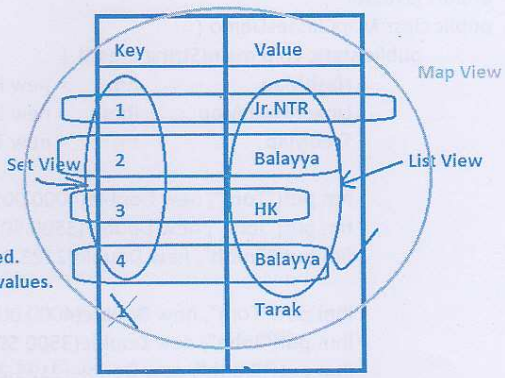
```
//SBComparator.java
class SBComparator implements java.util.Comparator {

        public int compare(Object obj1, Object obj2){
                StringBuffer sb1 = (StringBuffer)obj1;
                StringBuffer sb2 = (StringBuffer)obj2;

                String s1 = sb1.toString();
                String s2 = sb2.toString();

                return s2.compareTo(s1);
        }
}

//AddingSBtoTS.java
import java.util.*;

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

                TreeSet ts = new TreeSet(new SBComparator());
                ts.add(new StringBuffer("a"));
                ts.add(new StringBuffer("b"));
                ts.add(new StringBuffer("x"));
                ts.add(new StringBuffer("c"));
                ts.add(new StringBuffer("y"));

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

Map architecture

Map object has three views
- Set view - collection of keys
- List view - collection of values
- Map view - collection of entries

Check the diagram.

we can retrive value by passing key its associated.
So key should be unique, but we can duplicate values.



Map interface methods

```
public interface Map{

    public abstract boolean isEmpty();

    public abstract Object put(Object key, Object value);
    public abstract void putAll(Map m);

    public abstract Object get(Object key);

    public abstract Object remove(Object key);
    public abstract void clear();

    public abstract boolean containsKey(Object key);
    public abstract boolean containsValue(Object value);

    public int size();

    public abstract Set keySet();
    public abstract Collection values();
    public abstract Set entrySet();

    public abstract boolean equals(Object obj);
    public abstract int hashCode();
}
```

```
//MapClassesDemo.java
import java.util.*;
public class MapClassesDemo {
        public static void main(String args[]) {
                HashMap         hm      = new HashMap();
                LinkedHashMap   lhm     = new LinkedHashMap();
                TreeMap         tm      = new TreeMap();

                hm.put("Tom", new Double(4000.00));
                hm.put("John", new Double(3500.50));
                hm.put("Smith", new Double(2125.25));

                lhm.put("Tom", new Double(4000.00));
                lhm.put("John", new Double(3500.50));
                lhm.put("Smith", new Double(2125.25));

                tm.put("Abc", new Double(4000.00));
                tm.put("Cbc", new Double(3500.50));
                tm.put("Bbc", new Double(2125.25));

                System.out.println("hm  elements : "+hm);
                System.out.println("lhm elements : "+lhm);
                System.out.println("tm  elements : "+tm);

                //Getting the set of keys and getting the iterator
                Set             set             = hm.keySet();
                Iterator        hmitr           = set.iterator();

                System.out.println("\nThe Account balance hm Account holders:");
                while(hmitr.hasNext()) {
                        Object key      = hmitr.next();
                        Object value    = hm.get(key);
                        System.out.println(key +" : "+ value);
                }
                System.out.println();

                System.out.println("\nThe Account balance of lhm Account holders:");
                //Getting the collection of values and getting Iterator
                Collection lhmcol       = lhm.values();
                Iterator   lhmitr       = lhmcol.iterator();

                while(lhmitr.hasNext()) {
                        System.out.println(lhmitr.next());
                }
```

```
                System.out.println();
                System.out.println("\nThe Account balance of tm Account holders:");

                //Getting the set of Entries and obtaining Iterator
                Set tmset       = tm.entrySet();
                Iterator tmitr  = tmset.iterator();
                while(tmitr.hasNext()) {
                        Map.Entry me = (Map.Entry)tmitr.next();

                        System.out.print(me.getKey() + ": ");
                        System.out.println(me.getValue());
                }
                double balance = ((Double)hm.get("John")).doubleValue();
                hm.put("John", new Double(balance + 1000));

                System.out.println("John's new balance: " + hm.get("John"));
        }
}

//HashTableDemo.java
import java.util.*;
public class HashTableDemo {
        public static void main(String[] args) {
                Hashtable ht = new Hashtable();

                ht.put("Tom",new Double(10.50));
                ht.put("Henery",new Double(12.25));
                ht.put("Scott",new Double(8.75));

                System.out.println("ht elements :"+ht);

                Enumeration e = ht.keys();
                System.out.println("Employee Name \t\t Employee Sal");

                while(e.hasMoreElements()) {
                        String key = (String) e.nextElement();
                        System.out.println(key + "\t\t" + ht.get(key));
                }
                System.out.println();

                double sal = ((Double)ht.get("Scott")).doubleValue();
                sal += 11.25;

                ht.put("Kean",new Double(sal));
                ht.put("Scott",new Double(sal));

                e = ht.keys();
                System.out.println();
```

```
                System.out.println("The changed values are: ");
                while(e.hasMoreElements()) {
                        String key = (String) e.nextElement();
                        System.out.println(key + "\t\t" + ht.get(key));
                }
        }
}

//Understanding hashcode and equals() methods usage with collection objects
public class Student implements java.io.Serializable{
        int sno;
        String name;
        int whichClass;

        public Student(int sno, String name, int whichClass) {
                this.sno        = sno;
                this.name       = name;
                this.whichClass = whichClass;
        }

        public int hashCode() {
                return whichClass;
        }

        public boolean equals(Object obj){
                //below condition should be presented; else there is a chance of getting
                                                        ClassCastException
                if(obj instanceof Student)
                {
                        Student s = (Student)obj;
                        if (    this.sno == s.sno               &&
                                this.name.equals(s.name)        &&
                                this.whichClass == s.whichClass
                          )
                        {
                                return true;
                        }
                }
                return false;
        }
        public String toString(){
                return whichClass+": "+name;
        }
}
```

```
//School.java – storing and removing custom elements from Set
import java.util.*;
class School {
        public static void main(String[] args) {
                HashSet hs = new HashSet();

                hs.add(new Student(1, "Balayya", 1));
                hs.add(new Student(1, "Harikrishna", 1));

                hs.add(new Student(2, "Balayya", 2));
                hs.add(new Student(2, "Harikrishna", 2));
                hs.add(new Student(2, "Jr.NTR", 1));
                hs.add(new Student(2, "Jr.NTR", 1));

                hs.add(new Integer(8));
                hs.add(new Integer(10));
                hs.add(new String("a"));

                System.out.println(hs);

                hs.remove(new Student(2, "Jr.NTR", 1));
                System.out.println("\nafter removing");
                System.out.println(hs);
        }
}


// – using custom elements as keys to store, retrieve and remove values from Map
//Address.java
class Address implements java.io.Serializable{
        int hno;
        int streetNo;
        String city;
        Address(int hno, int streetNo, String city){
                this.hno        = hno;
                this.streetNo   = streetNo;
                this.city       = city;
        }
        public String toString(){
                return  "hno: "+ hno +"\n" +
                        "streetNo: "+ streetNo +"\n" +
                        "city: "+ city;
        }
}
```

```
//College.java
import java.util.*;
import java.io.*;
public class College {
        public static void main(String[] args) throws Exception       {
                LinkedHashMap       lhm     = new LinkedHashMap();

                lhm.put(new Student(11, "Rama", 11), new Address(1, 2, "Hyd"));
                lhm.put(new Student(11, "Raju", 12), new Address(1, 3, "Hyd"));

                lhm.put(new Student(12, "Sita", 11), new Address(12, 30, "Sec"));
                lhm.put(new Student(12, "Rani", 12), new Address(14, 30, "Sec"));

                //storing students information in file
                FileOutputStream      fos      = new FileOutputStream("studentinfo.ser");
                ObjectOutputStream oos      = new ObjectOutputStream(fos);

                oos.writeObject(lhm);
                System.out.println("\n Student info is saved ");
        }
}

//CollegeWebSite.java
import java.io.*;
import java.util.*;
public class CollegeWebSite {
        public static void main(String[] args) throws Exception{
                //reading students information from file
                FileInputStream       fis     = new FileInputStream("studentinfo.ser");
                ObjectInputStream     ois     = new ObjectInputStream(fis);

                LinkedHashMap       lhm     = (LinkedHashMap)ois.readObject();
                Set keySet                =       lhm.keySet();
                Iterator keyItr           =       keySet.iterator();

                while (keyItr.hasNext()){
                        Object key      = keyItr.next();
                        Object value    = lhm.get(key);

                        System.out.println(key + " student Address");
                        System.out.println(value);
                        System.out.println();
                }
        }
}
```

Collections class

This class consists exclusively of static methods that operate on or return collection objects. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.

This class is a member of the Java Collections Framework available since 1.2

This class has methods for performing below different operations on collection objects

1. searching element - only work for List
2. sorting elements - only work for List
3. swapping two elements - only work for List
4. shuffling elements - only work for List
5. reverse the order - only work for List
6. rotating elements at given distance - only work for List
7. copy elements - only work for List
8. checking whether two collections are disjoint - work for Set, List and Queue
9. Finding max, min elements - work for Set, List and Queue
10. Finding number of frequencies of a given element - work for Set, List and Queue
11. Obtaining Enumeration object - work for Set, List and Queue

12. Obtaining *synchronized* Set, List, Queue, Map, SortedSet, SortedMap collections
13. Obtaining *unmodifiable* Set, List, Queue, Map, SortedSet, SortedMap collections
14. Creating *immutable* empty Set, List, Map collections
15. Creating *singleton* Set, List, Map
16. Obtaining *reverse natural sorting* Comparable object

Very important methods for 1.5 collection API enhancement with Generics

Q) How can we convert Non-generics collections to Generics based collections?

A) We should use below methods.

17. Creating *checked* Set, List, Queue, Map, SortedSet, SortedMap

Arrays class

This class contains various methods for manipulating arrays (such as sorting and searching). This class also contains a static factory that allows arrays to be viewed as lists.

The methods in this class all throw a NullPointerException if the specified array reference is null.

Q) How can we print given array data directly without using for loop?

A) *Arrays* class has static parameterized *toString* method to return array content in String format.

For method prototypes and more details on above two classes please refer its API documentation. By following API documentation try to develop your own programs to check all above operations.

Properties class:
This class is used to store properties permanently in a file.
Property means (key, value) in string form.

Rule: To store properties in a file both key and value type should be java.lang.String type. Because file can only display string data, and more over any data that we type in file is of type string.

Procedure to create Properties file

- The file that contains properties is called properties file, and as per coding standards its extension should be ".properties".
- Every property's name and value must be separated by "=" or ":"
- Every property must be placed in new line.
- The properties file of Employee is look like as below

Employee.properties

```
eid=7279
ename=HariKrishna
desig=JavaFaculty
company=NareshTechnologies
```

It has below methods to read and store properties from a properties file

1. public void **load**(InputStream in)
2. public String **getProperty**(String name)
   If the given name is not existed in the file it returns null
3. public String **getProperty**(String name, String msg)
   If the given name is not existed in the file it returns the given message not *null*
4. public Enumeration **propertyNames**()
   Returns all property names as Enumeration object
5. public Object **setProperty**(String name, String value)
   It sets this new property to Properties object
   It returns Object because it internally calls "put" method to add this property
6. public void **list**(PrintStream out)
7. public void **store**(OutputStream out, String msg)
   Both methods are used to store current properties object to file
   "store" method allows to store properties with given message comment, where as it is not possible with "list" method.

```
//PropertiesDemo.java
import java.io.*;
import java.util.*;

public class PropertiesDemo
{
        public static void main(String[] args)  throws Exception
        {
                Properties p = new Properties();
                p.load(new FileInputStream(args[0]));

                System.out.println("name:"+p.getProperty("eid"));
                System.out.println("sal:"+p.getProperty("ename"));
                System.out.println("desg:"+p.getProperty("desig"));
                System.out.println("company:"+p.getProperty("company"));

                System.out.println("abc:"+p.getProperty("abc"));
                System.out.println("abc:"+p.getProperty("abc","abc key is not found"));

                p.setProperty("exp","Since 2004");
                Enumeration e          = p.propertyNames();

                while(e.hasMoreElements()){
                        System.out.println(p.getProperty((String)e.nextElement()));
                }

                p.list(new PrintStream(new FileOutputStream("
                                                        List Result.properties")));
                p.store(new PrintStream(new FileOutputStream("Store
                Result.properties")),"Emp details");

        }
}

emp.properties
eid=7279
ename=Harikrishna
desig=JavaSCJPFaculty
company=NareshiTechnologies
```

//Internationalization – I18N

+ To execute the Java application in different countries with that country specific language we must develop I18N.

+ It means if we want to display a GUI components names - for example, Username, Password, SignIn - in a country specific language we must develop this feature.

+ In Java we have two classes given in java.util package to develop I18N.
They are:
1. Locale
2. ResourceBundle

+ Locale represents country
+ ResourceBundle loads the country's properties file dynamically based on the Locale object.

Procedure to develop I18N application:
1. We must develop a properties file one per each country with that country lanaguage. For all these properties file we must create a base file in the default language in which we want to display names.

2. Then we must create ResourceBundle object by using its factory method getBundle() by passing the properties file base name with out extension.

3. ResourceBundle loads the current country properties file by appending its locale to the base name.

4. ResourceBundle gets the current country locale from Locale object, it gets that current country locale from the OS from its environment variable "lang" and "country".

Methods:
public ResourceBundle getBundle(String fileName);
public ResourceBundle getBundle(String fileName, Locale locale);
- for creating resource bundle object
- If given properties file is not existed it throws java.util.MissingResourceException

public String getString(String pn)
- for retriving property value
- If pn is not found in the properties file it also throws MRE
- If pn is null it throws NPE
- If pn value is not String it throws CCE

Locale class contains many fields to represent each country language and country code.
For US
language code is: en

counrty code is: US

Below are the variables defined in Locale class for above US codes
public static final Locale ENGLISH
public static final Locale US

Refer Locale API for more details

Q) How can we get System properties?
A) By calling System.getProperties() method.

```
import java.util.*;

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

                //loaded properties file based on current locale
                ResourceBundle rb     = ResourceBundle.getBundle("wish");
                //ResourceBundle rb  = ResourceBundle.getBundle("wish",
                                                        Locale.getDefault());

                //loaded properties file based on passed Locale
                //ResourceBundle rb  = ResourceBundle.getBundle("wish",
                                                new Locale(Locale.FRENCH.toString(),
                                                        Locale.FRENCH.toString()));

                String wish      = rb.getString("hellowish");

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

```
#wish.properties        => it is the default properties file for all languages and countries
hellowish=hello
```

```
#wish_fr_FR.properties   => it contains France country language wish
hellowish=hello in French
```

```
#wish_de_DE.properties   => it contains German country language wish
hellowish=hello in German
```

StringTokenizer:
This class allows us to break a string into tokens. To break the string we must pass a delimiter, it is a character or substring available in the tokenizing string.

The default delimiters are
1. the space character,
2. the tab character (\t),
3. the newline character (\n),
4. the carriage-return character (\r), and
5. the form-feed character (\f).

In constructing StringTokenizer object if we do not pass delimiter, it uses above delimiters to break the string. From Java 1.4 onwards it is recommended to use String class "*split*" method rather StringTokenizer class.

Constructors:
It has three constructors

a. **public StringTokenizer(String str)**
It breaks given string with above default delimiters

For example

```
String s = "Hari Krishna";

StringTokenizer st = new StringTokenizer (s);
```

It returns two tokens
1. Hari
2. Krishna

b. **public StringTokenizer(String str, String delim)**
It breaks given string with given delimiter

For example

```
String s = "Hari Krishna";

StringTokenizer st = new StringTokenizer (s, "r");
```

It returns 3 tokens
1. Ha
2. i K
3. ishna

c. **public StringTokenizer(String str, String delim, boolean returnDelims)**
It breaks given string with given delimiter and also includes delimiter as token

For example

```
String s = "Hari Krishna";
```

StringTokenizer st = new **StringTokenizer** (s, "r", true);
It returns 5 tokens

1. Ha
2. **r**
3. i K
4. **r**
5. ishna

Delimiter "r" is also included in the list of tockens

Q) What is the direct subclass of Enumeration interface?
StringTokenizer

Methods:
It has below 5 methods

1. **To get tokens count**
   public int countToken()

2. **To check is token available or not**
   public boolean hasMoreElements()
   public boolean hasNextToken()

3. **To retrieve token**
   public Object nextElement()
   public String nextToken()

//StringTokenizerDemo.java

```
import java.util.StringTokenizer;

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

                String str              = "Hari Krishna, Naresh i Technologies";

                //StringTokenizer st    = new StringTokenizer(str);

                //StringTokenizer st    = new StringTokenizer(str, ",");

                StringTokenizer st      = new StringTokenizer(str,",",true);

                System.out.println("Number of tockens: "+st.countTokens());
                while(st.hasMoreElements()) {
                        System.out.println(st.nextElement());
                }
        }
}
```

Introduction to Generics

This feature is introduced in Java 5 to provide the type for method's parameters and return types dynamically by achieving compile-time type checking for solving ClassCastException.

It means this feature allows us to pass the type for method parameter and return type at runtime when this class object is created.

If we define "class / interface / enum" with generic type that

- class is called parameterized type or generic type.
- method is called generic type method
- constructor is called generic type constructor

Syntax to create a class with generic parameter:

```
class Example<T>{

    void m1(T t){}
    <T> m2(){ --- }

    Example<T> (){}
    Example<T> (T t){}
}
```

object creation of above class

We must pass class name as argument for creating object of this class

```
Example<String> e1 =  new Example<String>();
```

Rule: Then using e1 we should call m1() method only by passing "String" objects, because m1 method parameter for e1 object is String. If we pass Integer objects it leads to CE

```
e1.m1("a");  ✓
e1.m1(10);   ✗
```

Example class object to accept only Integer objects in calling m1() method

```
Example<Integer> e2 = new Example<Integer>();
e2.m1("a");  ✗
e2.m1(10);   ✓
```

The main reason of introducing Generics

Basically Generic Types feature provides compile-time type safety for calling methods as shown above. The main reason of introducing this feature is to solve collection coding problem *ClassCastException*. To solve this exception compiler should only allow homogeneous objects to add to collection object. So generics feature is introduced to create homogeneous collection that means a collection with only same type objects.

So by using this feature we can tell to compiler what type of elements only must be added to a collection object, as shown below

```
ArrayList<String> al = new ArrayList<String>();
```

From the above statements compiler allows us to add only String objects to this collection object. If we add Integer objects compiler throws compile time error, because add method parameter is String.

```
al.add("a");
al.add(10);
```

Below are the collection classes' definitions as per Java 1.4 and Java 5 with generics

Java 1.4 ArrayList and Iterator

```
public class ArrayList{

   public boolean add(Object obj){}
   public Object get(int index)

   Iterator iterator(){}

}
```

```
public interface Iterator{

   public boolean hasNext();
   public Object next();
   public void remove();
}
```

Java 5 ArrayList and Iterator

```
public class ArrayList<E>{

   public boolean add(E e){}
   public E get(int index)

   Iterator<E> iterator(){}

}
```

```
public interface Iterator<E>{

   public boolean hasNext();
   public E next();
   public void remove();
}
```

Java 1.4 LinkedHashMap

```
public class LinkedHashMap{

   public Object put(Object key, Object value){}
   public Object get(Object key){}

   public Set keySet(){}
   public Collection values(){}
   public Set entrySet(){}

}
```

Java 5 LinkedHashMap

```
public class LinkedHashMap<K, V>{

   public V put(K key, V value){}
   public V get(K key){}

   public Set<K> keySet(){}
   public Collection<V> values(){}
   public Set<Map.Entry<K,V>> entrySet(){}

}
```

Q) Write a program to collect only String objects using ArrayList. Retrieve and print all elements by using get, iterator and for-each loops.

Java 1.4 based code

```
import java.util.*;

class NonGenericAL{

    public static void main(String[] args){

        ArrayList al = new ArrayList();

        al.add("a");
        al.add("b");
        al.add("c");

        Iterator itr = al.iterator();

        while( itr.hasNext() ){

            Object obj = itr.next();
            String s = (String)obj;
            System.out.println(
                    s.toUpperCase()
            );

        }

    }
}
```

Java 5 based code

```
import java.util.*;

class GenericAL{

    public static void main(String[] args){

        ArrayList<String> al =
                new ArrayList<String>();

        al.add("a");
        al.add("b");
        al.add("c");

        Iterator<String> itr = al.iterator();

        while( itr.hasNext() ){

            String s = itr.next();
            System.out.println(
                    s.toUpperCase()
            );

        }

//if we do not add <String> parameter to
Iterator then next() method returns elements
as java.lang.Object type, then again we should
downcast it String class as shown below

        Iterator itr2 = al.iterator();

        while( itr.hasNext() ){

            Object obj = itr.next();
            String s = (String)obj;
            System.out.println(
                    s.toUpperCase()
            );

        }
    }
}
```

SCJP Questions – Generics Rules

Rule #1: we can create generic class object without passing parameter, for that object the implicit parameter is java.lang.Object.

```
ArrayList al = new ArrayList();

//for this al object add method parameter is java.lang.Object.
//So we can add all heterogeneous objects

al.add(10);
al.add("a");
```

Note: for these two methods call compiler shows a "Note" saying unchecked operation. Note is not a CE, so for the current class ".class" file is generated.

Q) How can we create generic type collection object to collect heterogeneous elements.

A) The parameter type should be java.lang.Object

```
ArrayList<Object> al = new ArrayList<Object>();
```

Rule #2: Upcasting is not possible for generic type, it means "List<String>" is not a subtype of "List<Object>"

Find CEs

```
List<String> ls = new ArrayList<String>();
List<Object> ls = new ArrayList<String>();
```

Why not allowed?

Because using "ls" we can call add method by passing heterogeneous objects, which is illegal.

```
Ex: ls.add(10);
```

Q) Is below method definition and its call allowed?

```
interface Shape{}
class Rectangle implements Shape{}
class Test{
        static void m1(List<Shape> ls){

        }
        pubic static void main(String[] args){
                m1(new ArrayList<Rectangle>())
        }
}
```

Method call leads to CE, because ArrayList<Rectangle> is not a subclass of List<Shape>

Rule #3: to solve above problem we should use wildcard character (?).

- ? means unknown type, it creates super type for generics.

- Collection<Object> is not a super generic type of all collection objects.
- Collection<?> is a super type of all collection objects.

```
//List<Object> ls1 = new ArrayList<String>();

List<?> ls1 = new ArrayList<String>();
List<?> ls2 = new ArrayList<Integer>();
```

Problem #1 with wildcard: method parameters are not replaced with type, but return types are replaced with java.lang.Object. Due to this we cannot call parameterized methods (mutator methods) but we can call return type method (accessor methods)
So, with this ? syntax we cannot call add method to add elements to collection object because compiler does not know add() method parameter.

```
ls1.add("a"); CE:
```

but we can call get() method on this unknown type, because every added object is of type java.lang.Object.

Find out CE in the below lines

```
Object obj = ls1.get(0);
String s = ls1.get(0);
```

Q) In which case should we use wildcard character syntax based generic type?
A) To develop runtime polymorphism based method to take all generic type objects, we must use wildcard generic type parameter. But in this method we can only read elements but we can add, or update or remove elements.

```
void m1(List<?> ls){
        for(Object obj : ls){
                System.out.println(obj);
        }
}

m1(new ArrayList<String>());
m1(new ArrayList<Integer>());
m1(new ArrayList<Object>());
m1(new ArrayList<Rectangle>());
```

Problem #2 with ? generic type
Wild card generic type parameterized method allows all types of generic class objects. We need new syntax to pass specific category subclasses generic types.

For example

ArrayList<Rectangle>, ArrayList<Square>, ArrayList<Circle>

Solution: We must define boundaries for "?"

We can define two boundaries

1. subclasses boundary (? extends Shape)
2. superclasses boundary (? super Shape)

- ArrayList<? extends Shape> means the argument collection object should contain only Shape subclasses objects.
- ArrayList<? super Shape> means the argument collection object should contain only Shape superclasses objects.

For example:

```
void m1(List<? extends Shape>){

}

m1(new ArrayList<String>());
m1(new ArrayList<Integer>());
m1(new ArrayList<Rectangle>());
m1(new ArrayList<Square>());
m1(new ArrayList<Circle>());
m1(new ArrayList<Shape>());
m1(new ArrayList<Object>());

void m1(List<? super Shape>){

}

m1(new ArrayList<String>());
m1(new ArrayList<Integer>());
m1(new ArrayList<Rectangle>());
m1(new ArrayList<Square>());
m1(new ArrayList<Circle>());
m1(new ArrayList<Shape>());
m1(new ArrayList<Object>());
```

In the above two m1() methods also we are not allowed to call "add" method because of unknown type (?) but we can call "get" with destination variable type "Object".

I would try to update our site JavaEra.com everyday with various interesting facts, scenarios and interview questions. Keep visiting regularly.....

Thanks and I wish all the readers all the best in the interviews.

www.JavaEra.com

A Perfect Place for All Java Resources