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.

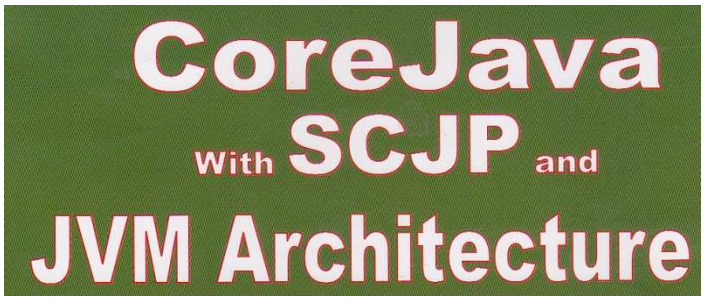
HTML, JavaScript, Jquery, AJAX, CSS, Adobe Photoshop CS6, Adobe Illustrator CS5, Adobe Flash.

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>

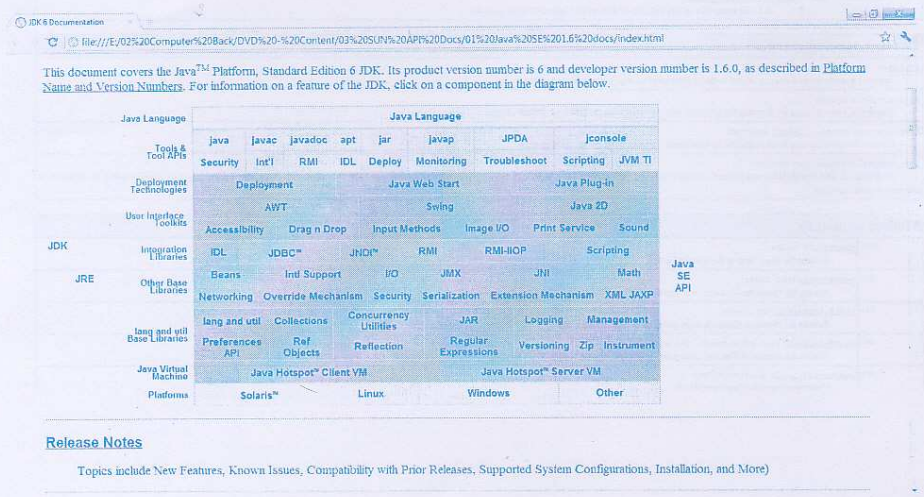
   For example
   - javadoc Example.java
     i. In the current directory you will find many html files and directories. Among them open "index.html" file to find your class html file.

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.

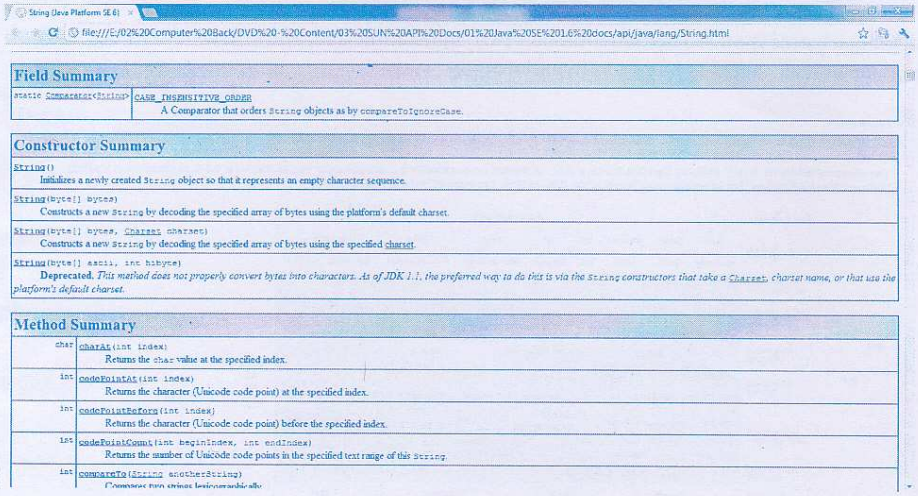
For example to open "String" class's html file open String.html from below path

"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.

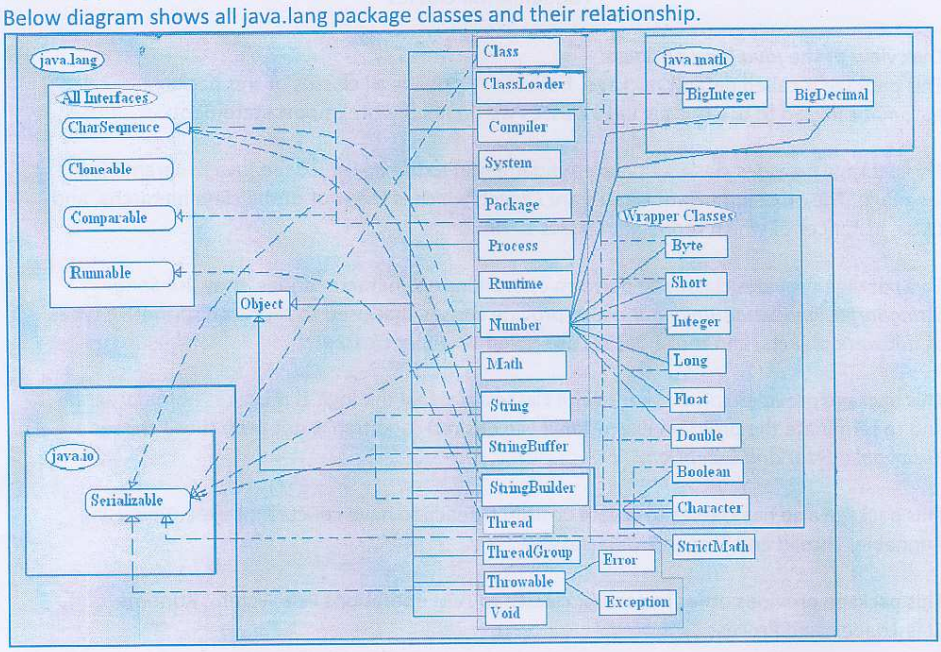
This package provides other important classes to JVM operations like System, Runtime, ClassLoader, and Process.

+ 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

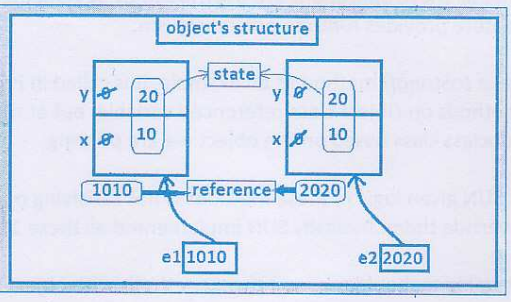
1. By using **objects reference** or
2. By using **objects state**

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.

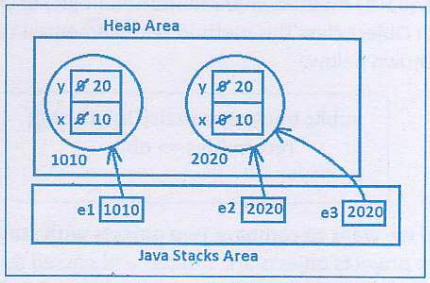
In projects objects are created and passed as method arguments at runtime. So, to know the given object is expected one or not, we should compare that object with our own created object for their equality.

```
/*
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?

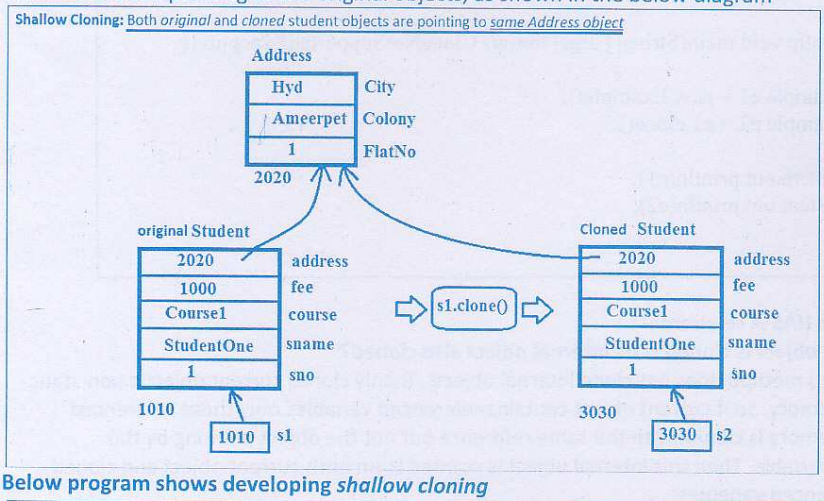
Java supports two type of clonings:-

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*

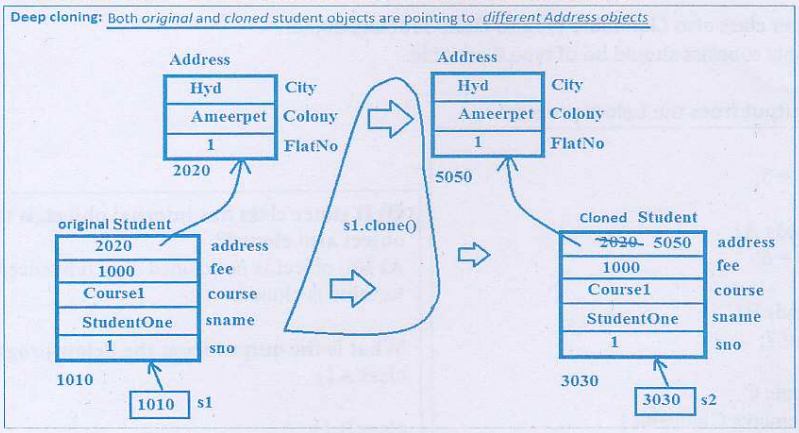
```
public class Address{
  int     flatNo      = 1;
  String colony       = "Ameerpet";
  String city         = "Hyd";
}
```

```
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.

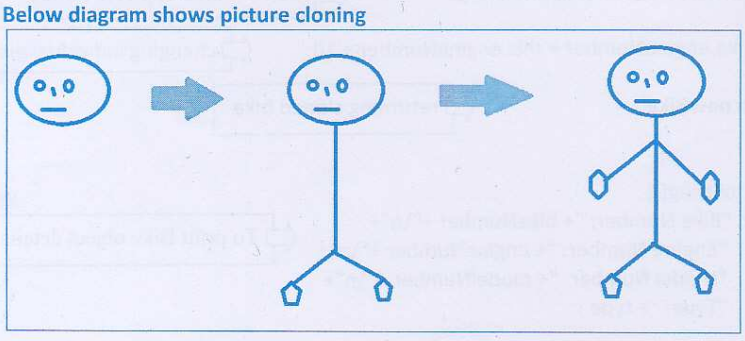
Then you have all 100 instances of this object with same state by executing validations and calculations only once. Hence you get high performance.

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.

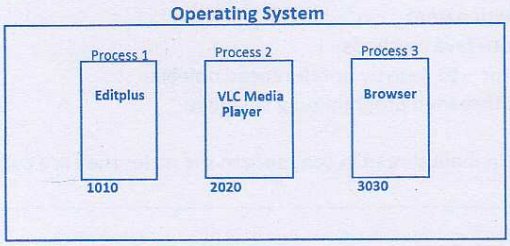
Multitasking is of two types

1. Process based multitasking
2. Thread based multitasking

Process based multitasking

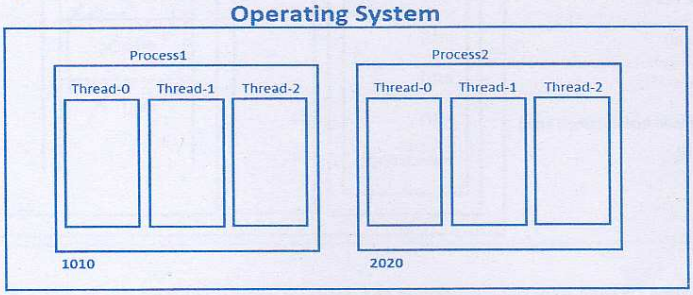
Executing multiple tasks *simultaneously* is called process based multitasking here each task is a separate independent process.

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.

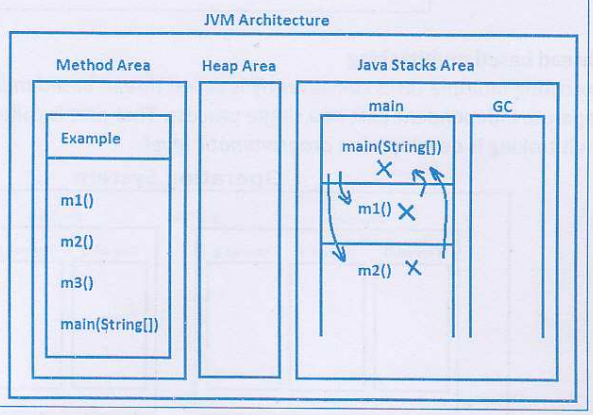
Overview on Java threads
We can consider JVM is also a process, when JVM is created in its Java stacks area by default two threads are created with names

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();
    }
}
```



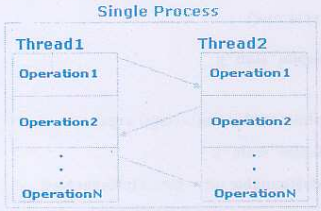
By using single thread JVM executes methods sequentially one after one.

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.

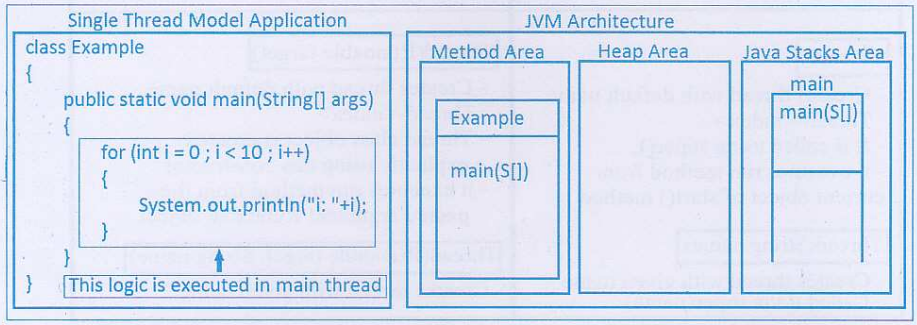
   Rule #1: The value of millis should not be negative or the value of nanos is should be in the range 0-999999 else it leads to exception "java.lang.IllegalArgumentException".

   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

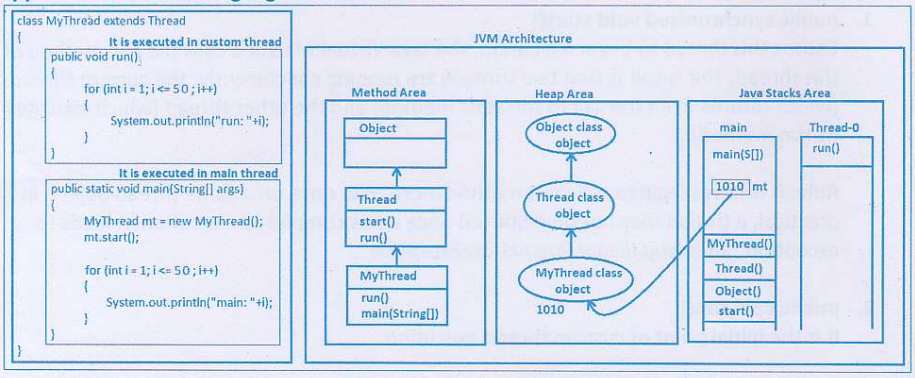
   Remaining methods are explained in the respective programs

Programs

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



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



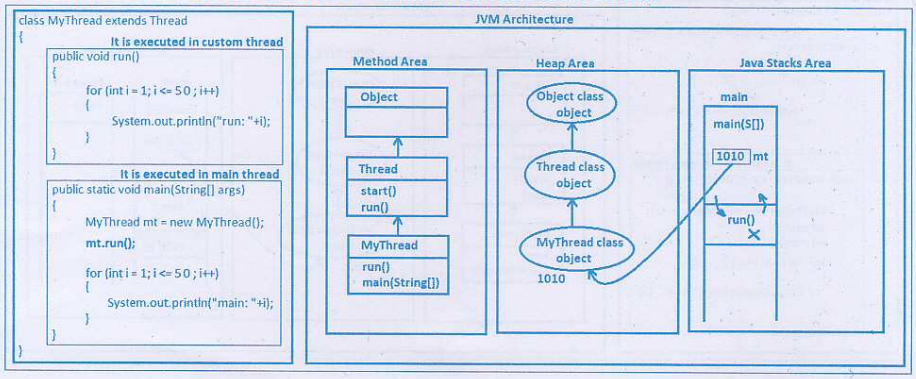
Output: "*main method for loop*" and "*run method logic*" are executed concurrently.

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?

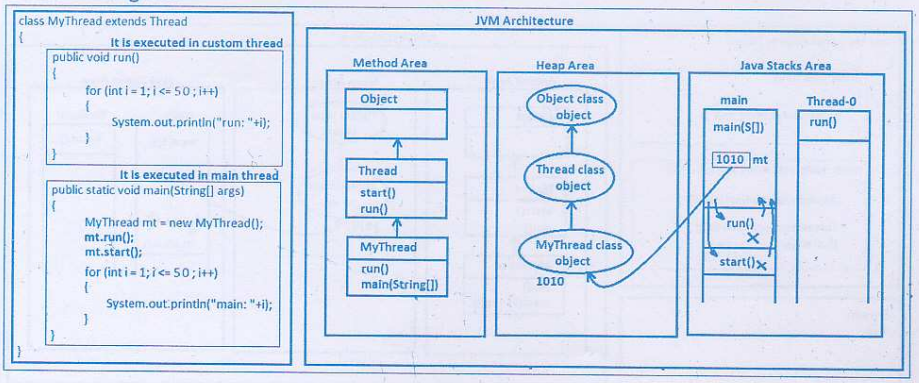
Yes, we can call *run* method directly. If we call it directly, user defined thread in Java Stacks Area is not be created. It is executed in *main thread* as shown below.



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?

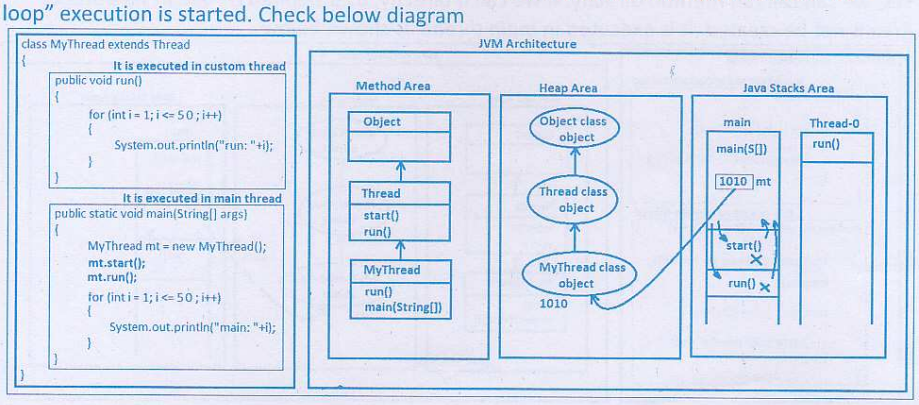
run method logic is executed two times. It is executed in *main thread* since it is called directly from main method, also it is executed in *custom thread* due to *start* method call, as shown in the below diagram.



Output:

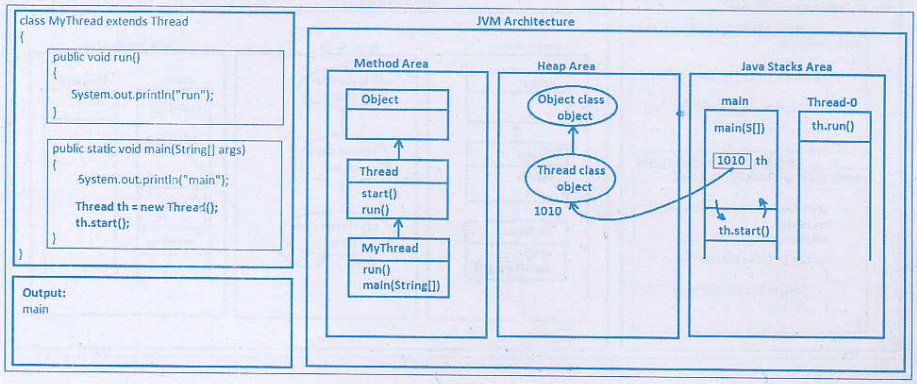
First *run method* in main thread completes its execution, then due to start method call user defined thread is created and run method is called again. Now "*run method* in *Thread-0*" and "*main method* for loop in *main thread*" are executed concurrently.

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

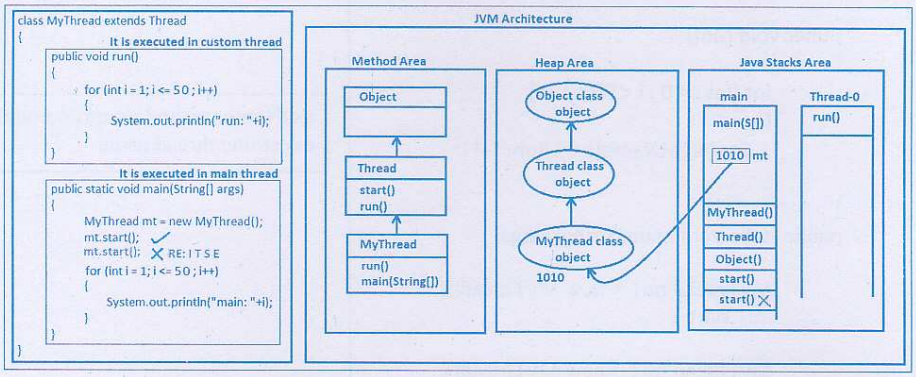


What is the output in the below case?
Q) If start method is called on Thread class object directly, is subclass run() method executed?
No, run method is executed from Thread class.

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

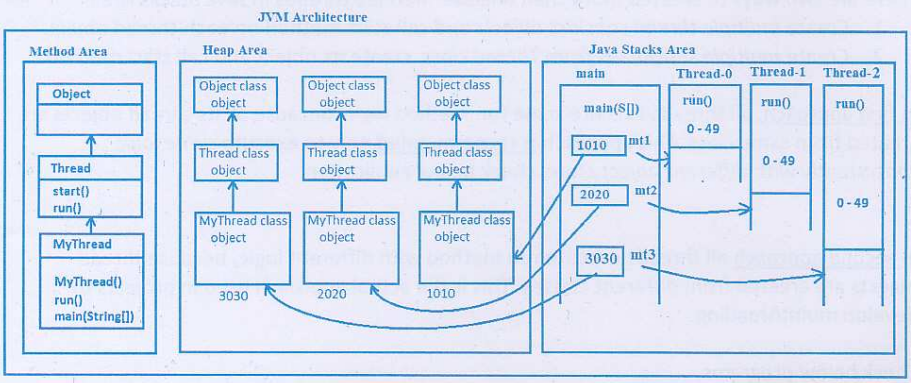
In second approach all threads execute run method with different logic, because thread objects are created from different classes. This is the actual approach used in projects to develop multithreading.

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.



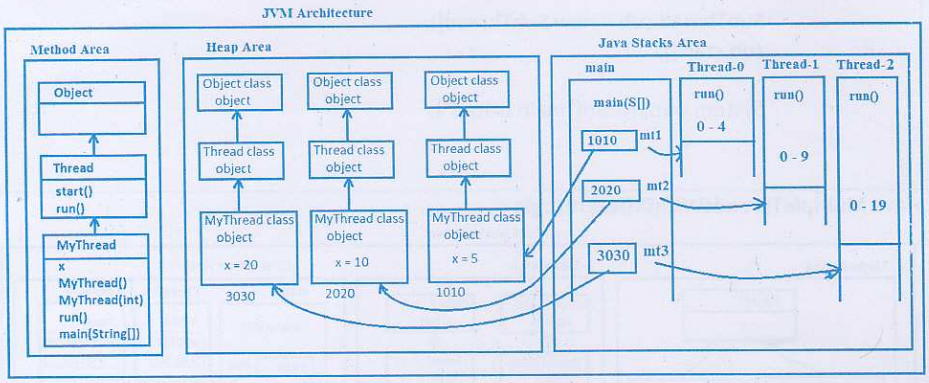
As you noticed, three *thread objects* are created in heap area and for each thread object separate *thread of executions* are created in Java Stacks Area.

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.

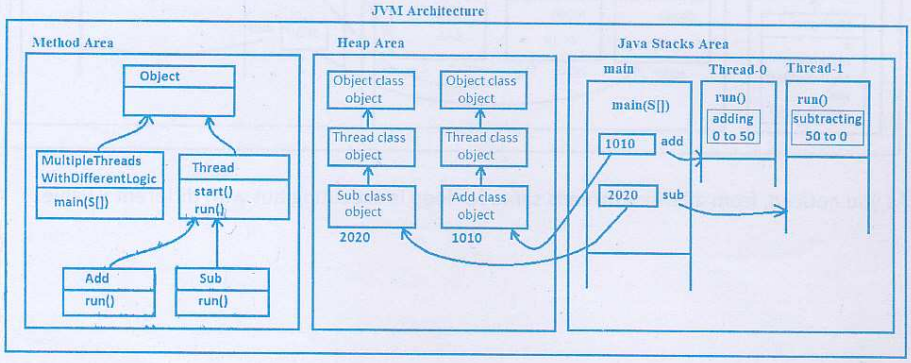
Application #5: This application shows creating multiple user defined threads to execute different logic. This program shows printing summation of numbers "0to50" and subtraction of numbers "50to0" concurrently.

```
//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.

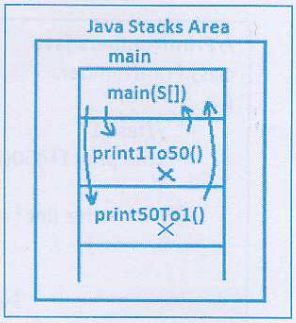
Write a program to show time difference between single and multiple threads execution

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");
    }
}
```

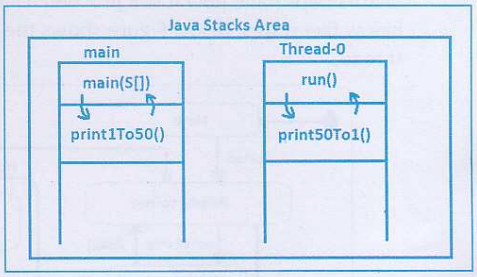


Output: Sequential execution – takes 10 secs to complete two tasks

```
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

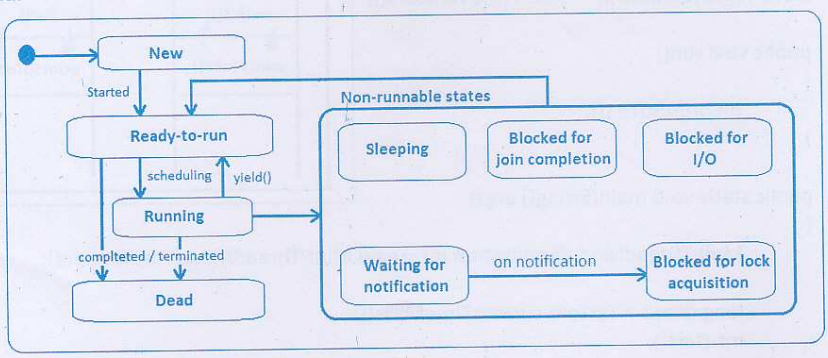
```
C:\Program Files\EditPlus 2\launcher.exe
1       50      49      2       48      3       4       47      5       46
6       45      44      7       8       43      42      9       41      10
40      11      39      12      13      38      37      14      36      15
35      16      34      17      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

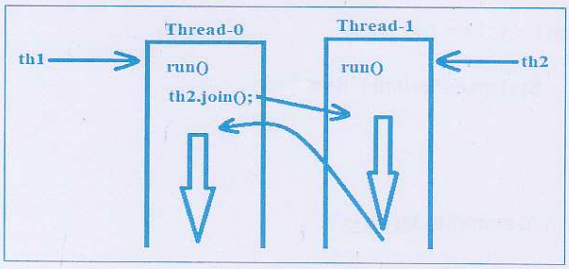
Pausing thread execution until other thread execution is completed is called *thread join*.
Thread class has below overloaded join methods to perform this task.

```
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
```

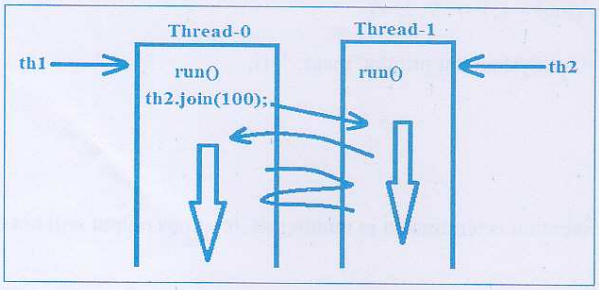
Difference between *no-arg join* and *parameterized join* methods

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...");
        }
}
```