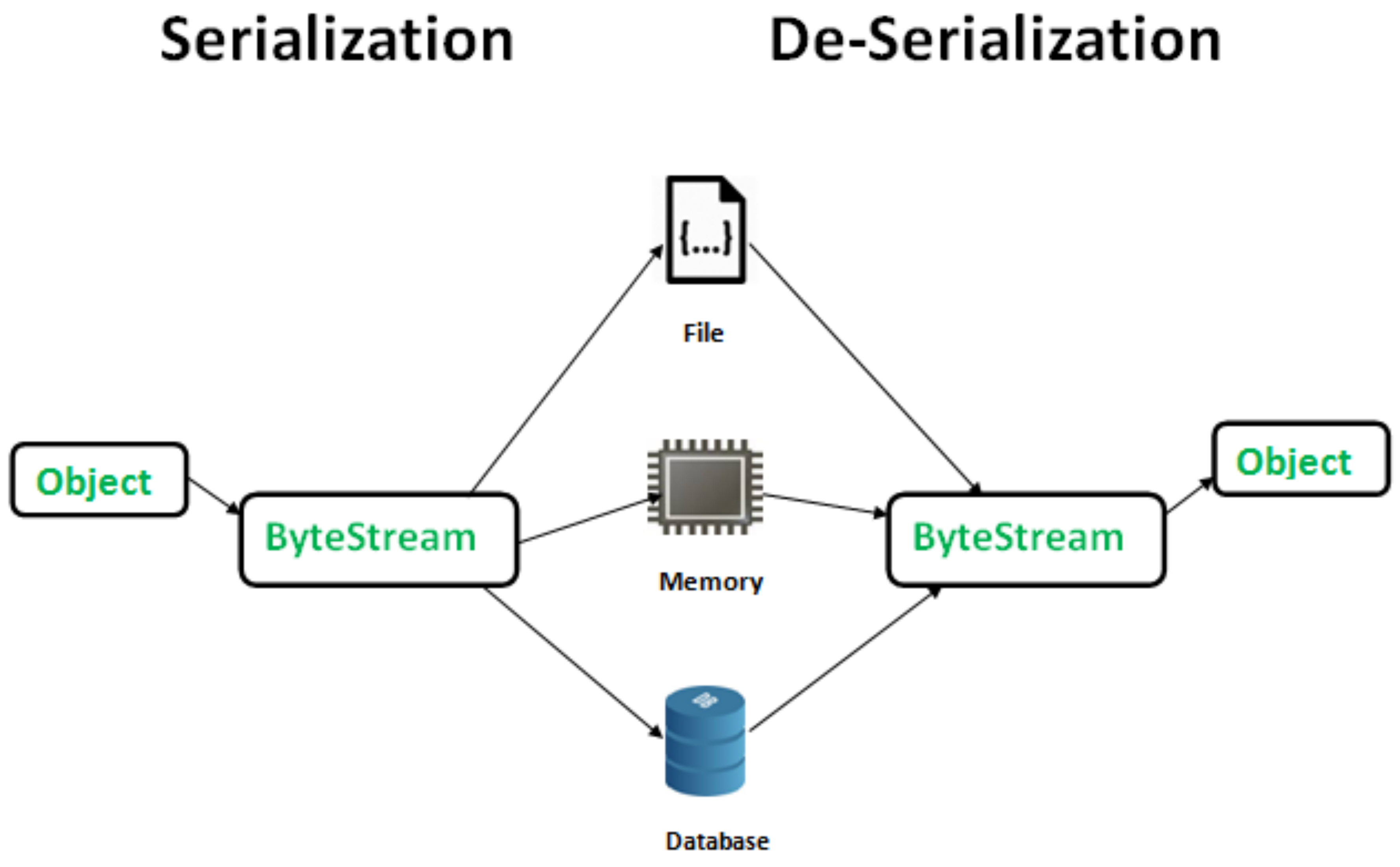
What is Java Serialization?

Serialization is a mechanism to convert an object into stream of bytes so that it can be written into a file, transported through a network or stored into database. De-serialization is just a vice versa.



The advantages of serialization are:

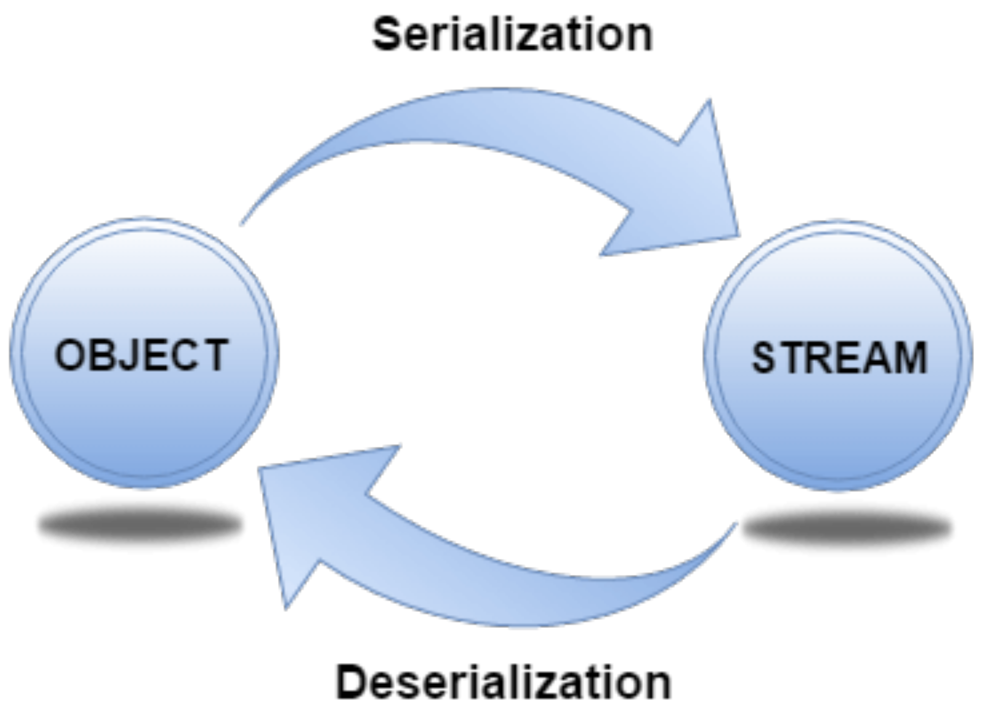
- It is easy to use and can be customized.
- The serialized stream can be encrypted, authenticated and compressed, supporting the needs of secure Java computing.
- Serialized classes can support coherent versioning and are flexible enough to allow gradual evolution of your application's object schema.
- Serialization can also be used as a mechanism for exchanging objects between Java and C++ libraries
- There are simply too many critical technologies that rely upon serialization, including RMI, JavaBeans and EJB.

Serialization:

Object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object. **Serialization in Java** is a mechanism of *writing the state of an object into a byte-stream*. It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.

The reverse operation of serialization is called *deserialization* where byte-stream is converted into an object. The serialization and deserialization process is platform-independent, it means you can serialize an object in a platform and deserialize in different platform.

For serializing the object, we call the **writeObject()** method *ObjectOutputStream*, and for deserialization we call the **readObject()** method of *ObjectInputStream* class.



java.io.Serializable interface

Serializable is a marker interface (has no data member and method). It is used to "mark" Java classes so that the objects of these classes may get a certain capability.

```
import java.io.Serializable;
public class Student implements Serializable{
 int id;
 String name;
 public Student(int id, String name) {
  this.id = id;
  this.name = name;
 }
}
```

ObjectOutputStream class

The ObjectOutputStream class is used to write primitive data types, and Java objects to an OutputStream.

ObjectInputStream class

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

Example of Java Serialization

```
import java.io.*;
class Persist{
 public static void main(String args[]){
  try{
  //Creating the object
  Student s1 =new Student(211,"ravi");
  //Creating stream and writing the object
  FileOutputStream fout=new FileOutputStream("f.txt");
  ObjectOutputStream out=new ObjectOutputStream(fout);
  out.writeObject(s1);
  out.flush();
  //closing the stream
  out.close();
  System.out.println("success");
  }catch(Exception e){System.out.println(e);}
 }
}
```

Example of Java Deserialization

```
import java.io.*;
class Depersist{
 public static void main(String args[]){
  try{
  //Creating stream to read the object
  ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));
  Student s=(Student)in.readObject();
  //printing the data of the serialized object
  System.out.println(s.id+" "+s.name);
  //closing the stream
  in.close();
  }catch(Exception e){System.out.println(e);}
 }
}
```

Java Serialization with Inheritance (IS-A Relationship)

```
import java.io.Serializable;
class Person implements Serializable{
 int id;
 String name;
 Person(int id, String name) {
  this.id = id;
  this.name = name;
 }
}

class Student extends Person{
 String course;
 int fee;
 public Student(int id, String name, String course, int fee) {
  super(id,name);
  this.course=course;
  this.fee=fee;
 }
}
```

Java Serialization with Aggregation (HAS-A Relationship)

```
class Address{
 String addressLine,city,state;
 public Address(String addressLine, String city, String state) {
  this.addressLine=addressLine;
  this.city=city;
  this.state=state;
 }
}

import java.io.Serializable;
public class Student implements Serializable{
 int id;
 String name;
 Address address;//HAS-A
 public Student(int id, String name) {
  this.id = id;
  this.name = name;
 }
}

//Since Address is not Serializable, you can not serialize the instance of Student class.
```

Java Console Class

The Java Console class is be used to get input from console. It provides methods to read texts and passwords.
If you read password using Console class, it will not be displayed to the user.

a simple example to read text from console.

```
String text=System.console().readLine();
System.out.println("Text is: "+text);
```

Java Console class declaration

```
public final class Console extends Object implements Flushable
```

Java Console class methods

Method	Description
Reader reader()	It is used to retrieve the reader object associated with the console
String readLine()	It is used to read a single line of text from the console
Console format(String fmt, Object ….args)	It is used to formatted string to the console output stream
Console printf(String format, Object ….args)	It is used to write a string to the console output stream
PrintWriter write()	It is used to receive the PrintWriter object associated with the console.
void flush()	It is used to flushes the console

Generics

Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods or, with a single class declaration, a set of related types, respectively.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

```
import java.util.*;
class TestGenerics1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();
list.add("rahul");
list.add("jai");
//list.add(32);//compile time error
  String s=list.get(1);//type casting is not required
System.out.println("element is: "+s);
Iterator<String> itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
} } }
```

Java Generics using Map

```
import java.util.*;
class TestGenerics2{
public static void main(String args[]){
Map<Integer,String> map=new HashMap<Integer,String>();
map.put(1,"vijay");
map.put(4,"umesh");
map.put(2,"ankit");

//Now use Map.Entry for Set and Iterator
Set<Map.Entry<Integer,String>> set=map.entrySet();

Iterator<Map.Entry<Integer,String>> itr=set.iterator();
while(itr.hasNext()){
Map.Entry e=itr.next();//no need to typecast
System.out.println(e.getKey()+" "+e.getValue());
}

}}
```

Generic class

A class that can refer to any type is known as a generic class. Here, we are using the T type parameter to create the generic class of specific type.

```
class TestGenerics3{
public static void main(String args[]){
MyGen<Integer> m=new MyGen<Integer>();
m.add(2);
//m.add("vivek");//Compile time error
System.out.println(m.get());
}}
```

Type Parameters

The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:

1. T - Type
2. E - Element
3. K - Key
4. N - Number
5. V - Value

Generic Method

Like the generic class, we can create a generic method that can accept any type of arguments. A simple example of java generic method to print array elements. We are using here **E** to denote the element.

```
public class TestGenerics4{

   public static < E > void printArray(E[] elements) {
        for ( E element : elements){
            System.out.println(element );
         }
         System.out.println();
    }
    public static void main( String args[] ) {
        Integer[] intArray = { 10, 20, 30, 40, 50 };
        Character[] charArray = { 'J', 'A', 'V', 'A', ’T’,’R','A','N','I','N','G' };
        System.out.println( "Printing Integer Array" );
        printArray( intArray  );
       System.out.println( "Printing Character Array" );
        printArray( charArray );
    }
}
```

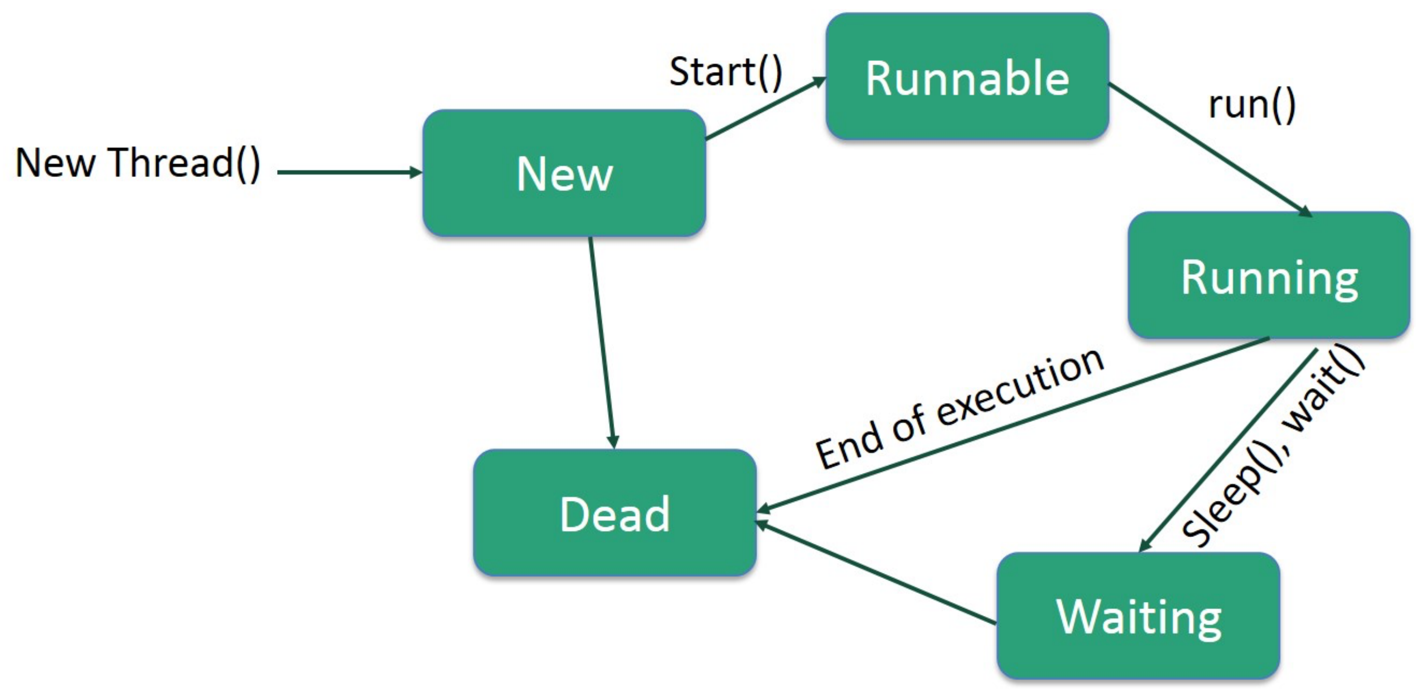
```java
import java.util.*;
abstract class Shape{
abstract void draw();
}
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle extends Shape{
void draw(){System.out.println("drawing circle");}
}
class GenericTest{
//creating a method that accepts only child class of Shape
public static void drawShapes(List<? extends Shape> lists){
for(Shape s:lists){
s.draw();//calling method of Shape class by child class instance
}
}
public static void main(String args[]){
List<Rectangle> list1=new ArrayList<Rectangle>();
list1.add(new Rectangle());

List<Circle> list2=new ArrayList<Circle>();
list2.add(new Circle());
list2.add(new Circle());

drawShapes(list1);
drawShapes(list2);
}}
```

Thread:

Java is a *multi-threaded programming language* which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time

Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application

Life Cycle of a Thread



What is Single Thread?
A single thread is basically a lightweight and the smallest unit of processing. Java uses threads by using a "Thread Class".
There are two types of thread – user thread and daemon thread (daemon threads are used when we want to clean the application and are used in the background).

```
public class SingleThread
{
        public static void main(String[] args) {
                System.out.println("Single Thread");
        }
}
```

Advantages of single thread:

- Reduces overhead in the application as single thread execute in the system
- Also, it reduces the maintenance cost of the application.

•

What is Multithreading in Java?
MULTITHREADING in Java is a process of executing two or more threads simultaneously to maximum utilization of CPU. Multithreaded applications execute two or more threads run concurrently. Hence, it is also known as Concurrency in Java. Each thread runs parallel to each other. Mulitple threads don't allocate separate memory area, hence they save memory. Also, context switching between threads takes less time.

Advantages of multithread:

- The users are not blocked because threads are independent, and we can perform multiple operations at times
- As such the threads are independent, the other threads won't get affected if one thread meets an exception.
-

- **New – A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.**

- **Runnable – After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.**

- **Waiting – Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.**

- **Timed Waiting – A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.**

Method	Description
start()	This method starts the execution of the thread and JVM calls the run() method on the thread
Sleep(int milliseconds)	This method makes the thread sleep hence the thread’s execution will pause for milliseconds provided and after that, again the thread starts executing. This help in synchronisation of the threads.
getName()	It returns the name of the thread.
setPriority(int newpriority)	It changes the priority of the thread.
yield()	It causes current thread on halt and other threads to execute.

Method & Description

public void start()
Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.

public void run()
If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.

public final void setName(String name)
Changes the name of the Thread object. There is also a getName() method for retrieving the name.

public final void setPriority(int priority)
Sets the priority of this Thread object. The possible values are between 1 and 10.

public final void setDaemon(boolean on)
A parameter of true denotes this Thread as a daemon thread.

public final void join(long millisec)
The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.

public void interrupt()
Interrupts this thread, causing it to continue execution if it was blocked for any reason.

public final boolean isAlive()
Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

Create a Thread by Implementing a Runnable Interface

Step 1: public void run( )
Step 2: Thread(Runnable threadObj, String threadName);
Step 3: void start();

```
class RunnableDemo implements Runnable {
   private Thread t;
   private String threadName;

   RunnableDemo( String name) {
      threadName = name;
      System.out.println("Creating " +  threadName );
   }
    public void run() {
      System.out.println("Running " +  threadName );
      try {
         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", " + i);
            // Let the thread sleep for a while.
            Thread.sleep(50);
         }
      } catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + " interrupted.");
      }
      System.out.println("Thread " +  threadName + " exiting.");
   }
```

```java
   public void start () {
      System.out.println("Starting " +  threadName );
      if (t == null) {
         t = new Thread (this, threadName);
         t.start ();
      }
   }
}

public class TestThread {

   public static void main(String args[]) {
      RunnableDemo R1 = new RunnableDemo( "Thread-1");
      R1.start();

      RunnableDemo R2 = new RunnableDemo( "Thread-2");
      R2.start();
   }
}
```

Output:
```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Create a Thread by Extending a Thread Class:

Step 1: public void run( )

Step 2: void start( );

```java
class ThreadDemo extends Thread {
   private Thread t;
   private String threadName;

   ThreadDemo( String name) {
      threadName = name;
      System.out.println("Creating " +  threadName );
   }

   public void run() {
      System.out.println("Running " +  threadName );
      try {
         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", " + i);
            // Let the thread sleep for a while.
            Thread.sleep(50);
         }
      } catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + " interrupted.");
      }
      System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start () {
      System.out.println("Starting " +  threadName );
      if (t == null) {
         t = new Thread (this, threadName);
         t.start ();
      }
   }
}
```

```
public class TestThread {

   public static void main(String args[]) {
      ThreadDemo T1 = new ThreadDemo( "Thread-1");
      T1.start();

      ThreadDemo T2 = new ThreadDemo( "Thread-2");
      T2.start();
   }
}
```

```
Output:
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Thread priorities

- Thread priorities are the integers which decide how one thread should be treated with respect to the others.
- Thread priority decides when to switch from one running thread to another, process is called context switching
- A thread can voluntarily release control and the highest priority thread that is ready to run is given the CPU.
- A thread can be preempted by a higher priority thread no matter what the lower priority thread is doing. Whenever a higher priority thread wants to run it does.
- To set the priority of the thread `setPriority()` method is used which is a method of the class `Thread` Class.
- In place of defining the priority in integers, we can use `MIN_PRIORITY`, `NORM_PRIORITY` or `MAX_PRIORITY`.

notify()
It wakes up one single thread that called `wait()` on the same object. It should be noted that calling `notify()` does not actually give up a lock on a resource. It tells a waiting thread that that thread can wake up. However, the lock is not actually given up until the notifier's synchronized block has completed.
So, if a notifier calls `notify()` on a resource but the notifier still needs to perform 10 seconds of actions on the resource within its synchronized block, the thread that had been waiting will need to wait at least another additional 10 seconds for the notifier to release the lock on the object, even though `notify()` had been called.

```
synchronized(lockObject)
{
    //establish_the_condition;

    lockObject.notify();

    //any additional code if needed
}
```

notifyAll()
It wakes up all the threads that called `wait()` on the same object. The highest priority thread will run first in most of the situation, though not guaranteed. Other things are same as `notify()` method above.

```
synchronized(lockObject)
{
    establish_the_condition;

    lockObject.notifyAll();
}
```

What are wait(), notify() and notifyAll() methods?

wait()
It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls `notify()`. The `wait()` method releases the lock prior to waiting and reacquires the lock prior to returning from the `wait()` method. The `wait()` method is actually tightly integrated with the synchronization lock, using a feature not available directly from the synchronization mechanism.

```
synchronized( lockObject )
{
    while( ! condition )
    {
        lockObject.wait();
    }

    //take the action here;
}
```

Thread Deadlock

Deadlock describes a situation where two or more **threads** are blocked forever, waiting for each other. ... A **Java** multithreaded program may suffer from the **deadlock** condition because the synchronized keyword causes the executing **thread** to block while waiting for the lock, or monitor, associated with the specified object.

Thread Deadlock

Deadlock describes a situation where two or more **threads** are blocked forever, waiting for each other. ... A **Java** multithreaded program may suffer from the **deadlock** condition because the synchronized keyword causes the executing **thread** to block while waiting for the lock, or monitor, associated with the specified object.