4.8 Generic Programming

The term ***generics*** means ***parameterized types***. Parameterized types are important because they enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter.

Using generics, it is possible to create a single class, for example, that automatically works with different types of data. A class, interface, or method that operates on a parameterized type is called *generic*, as in ***generic class* or *generic method*.**

Object is the superclass of all other classes, an **Object** reference can refer to any type object. Thus, in pre-generics code, generalized classes, interfaces, and methods used **Object** references to operate on various types of objects.

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.

Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

4.9 Generic Classes

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

General Form:

```
class class-name<type-param-list>
{
//Body of the class
}
```

Class Reference Declaration:

class-name<type-arg-list> var-name=new class-name<type-arg-list>(cons-arg-list)

4.9.1 Generic class with single type parameters

The following program defines two classes. The first is the generic class Gen and the second class GenDemo, which uses Gen. Here T is a type parameter that will be replaced by a real type when an object of type Gen is created.

Example Program:

```
class Gen<T>
{
T ob; // declare an object of type T
// Pass the constructor a reference to
// an object of type T.
Gen(T o)
{
ob = o;
}
// Return ob.
T getob()
{
return ob;
}
// Show type of T.
void showType()
{
```

```
System.out.println("Type of T is " + ob.getClass().getName()); }
}
// Demonstrate the generic class.
class GenDemo
{
public static void main(String args[])
{
// Create a Gen reference for Integers.
Gen<Integer> iOb;
// Create a Gen<Integer> object and assign its
// reference to iOb. Notice the use of autoboxing
// to encapsulate the value 88 within an Integer object.
iOb = new Gen<Integer>(88);
// Show the type of data used by iOb.
iOb.showType();
// Get the value in iOb. Notice that
// no cast is needed.
int v = iOb.getob();
System.out.println("value: " + v);
System.out.println();
// Create a Gen object for Strings.
Gen<String> strOb = new Gen<String> ("Generics Test");
// Show the type of data used by strOb. strOb.showType();
// Get the value of strOb. Again, notice // that no cast is
needed.
String str = strOb.getob();
System.out.println("value: " + str);
} }
```

Output:

```
Type of T is java.lang.Integer value: 88
Type of T is java.lang.String value: Generics Test
```

4.9.2 Generic class with two type parameters

In a generic type, more than one type parameter can be declared. To specify two or more type parameters, simply use a comma-separated list.

Example Program:

```
// A simple generic class with two type
// parameters: T and V. class TwoGen<T, V> { T ob1;
V ob2;
// Pass the constructor a reference to
// an object of type T and an object of type V.
TwoGen(T o1, V o2) { ob1 = o1;
ob2 = o2; }
// Show types of T and V.

void showTypes()
{
System.out.println("Type of T is " +ob1.getClass().getName());
System.out.println("Type of V is " +ob2.getClass().getName()); }
T getob1()
{
return ob1;
}
V getob2()
{
return ob2;
}
}
// Demonstrate TwoGen.
class SimpGen
{
public static void main(String args[])
{
TwoGen<Integer, String> tgObj =new TwoGen<Integer, String>(88,
"Generics");
// Show the types.
tgObj.showTypes();
// Obtain and show values.
int v = tgObj.getob1();
System.out.println("value: " + v);
String str = tgObj.getob2();
System.out.println("value: " + str);
}
}
```

Output:

```
Type of T is java.lang.Integer
Type of V is java.lang.String
value: 88
value: Generics
```

4.10 Generic Methods

Generic method is a method with type parameters. In this Generic concept , types and methods can be generic.

Syntax:

```
<type-param-list>re-type meth-name(param-list)
{
// Function Body

}
```

where

type-param-list is a list of type parameters separated by commas

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods -

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments

passed to the generic method, which are known as actual type arguments.

- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Example Program1:

```
public class GenericMethodTest
{
// generic method printArray

public static < E > void printArray( E[] inputArray ) {
// Display array elements for(E element : inputArray) {
System.out.printf("%s ", element);
}
System.out.println();
}
public static void main(String args[]) {
// Create arrays of Integer, Double and Character Integer[] intArray =
{ 1, 2, 3, 4, 5 };
Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
System.out.println("Array integerArray contains:");
printArray(intArray); // pass an Integer array
System.out.println("\nArray doubleArray contains:");
printArray(doubleArray); // pass a Double array
System.out.println("\nArray characterArray contains:");
printArray(charArray); // pass a Character array } }
```

Output:

```
Array integerArray contains: 1 2 3 4 5
Array doubleArray contains:
1.1 2.2 3.3 4.4
Array characterArray contains: H E L L O
```

Example Program2:

```
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','P','O','I','N','T' };
System.out.println( "Printing Integer Array" );
printArray( intArray );
System.out.println( "Printing Character Array" );
printArray( charArray );
}
}
```

Output:

```
Printing Integer Array
        10
        20
        30
        40
        50
        Printing Character Array
        J
        A
        V
        A
        T
        P
        O
        I
        N
        T
```

Example Program3:

```
// Demonstrate a simple generic method.
```

```
class GenMethDemo
{
// Determine if an object is in an array.
static <T, V extends T> boolean isIn(T x, V[] y)
{
for(int i=0; i < y.length; i++)
if(x.equals(y[i])) return true;
return false;
}
public static void main(String args[])
{
// Use isIn() on Integers.
Integer nums[] = { 1, 2, 3, 4, 5 };
if(isIn(2, nums))
System.out.println("2 is in nums");
if(!isIn(7, nums))
System.out.println("7 is not in nums");
System.out.println();
// Use isIn() on Strings.
String strs[] = { "one", "two", "three","four", "five" };
if(isIn("two", strs))
System.out.println("two is in strs");
if(!isIn("seven", strs))
System.out.println("seven is not in strs");
// Oops! Won't compile! Types must be compatible.
// if(isIn("two", nums))
// System.out.println("two is in strs");
}
}
```

Output:

```
2 is in nums
7 is not in nums
two is in strs
seven is not in strs
```

4.10.1 Generic Constructors

A generic constructor is a constructor with type parameters.

Example Program:

```
// Use a generic constructor.
class GenCons
{
private double val;
<T extends Number> GenCons(T arg)
{
val = arg.doubleValue();
}
void showval()
{
System.out.println("val: " + val);
}
}
class GenConsDemo
{
public static void main(String args[])
{
GenCons test = new GenCons(100);
GenCons test2 = new GenCons(123.5F);
test.showval();
test2.showval();
}
}
```

Output:

```
val: 100.0
val: 123.5
```

4.11 Bounded Types

Bounded type parameters can be a type parameter with one or more bounds. The bounds restrict the set of types that can be used as type arguments and give access to the methods defined by the bounds.

General Form:

To declare a parameter with bounded type, the list of type parameter names can be followed by the extends keyword.

<T extends superclass>

This specifies that T can be replaced by superclass.Superclass defines an inclusive,upper limit.

Example Program1: public class MaximumTest {
// determines the largest of three Comparable objects public static <T extends Comparable<T>> T maximum(T x, T y, T z) {
T max = x; // assume x is initially the largest if(y.compareTo(max) > 0) { max = y; // y is the largest so far } if(z.compareTo(max) > 0) { max = z; // z is the largest now } return max; // returns the largest object } public static void main(String args[]) {
System.out.printf("Max of %d, %d and %d is %d\n\n",3, 4, 5, maximum(3, 4, 5));
System.out.printf("Max of %.1f,%.1f and %.1f is %.1f\n\n",6.6, 8.8, 7.7, maximum(6.6, 8.8, 7.7));
System.out.printf("Max of %s, %s and %s is %s\n","pear","apple", "orange", maximum("pear", "apple", "orange"));
} }

Output:

```
Max of 3, 4 and 5 is 5
Max of 6.6,8.8 and 7.7 is 8.8
Max of pear, apple and orange is pear
```

Example Program2:

```
class Stats<T extends Number>
{
T[] nums; // array of Number or subclass
// Pass the constructor a reference to
// an array of type Number or subclass.
Stats(T[] o)
{ nums = o;
}
// Return type double in all cases.
double average()
{
double sum = 0.0;
for(int i=0; i < nums.length; i++)
sum += nums[i].doubleValue();
return sum / nums.length;
}
}
// Demonstrate Stats.

class BoundsDemo
{
public static void main(String args[])
{
Integer inums[] = { 1, 2, 3, 4, 5 };
Stats<Integer> iob = new Stats<Integer>(inums);
double v = iob.average();
System.out.println("iob average is " + v);
Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
Stats<Double> dob = new Stats<Double>(dnums);
double w = dob.average();
System.out.println("dob average is " + w);
// This won't compile because String is not a
// subclass of Number.
// String strs[] = { "1", "2", "3", "4", "5" };
// Stats<String> strob = new Stats<String>(strs);
// double x = strob.average();
// System.out.println("strob average is " + v);
}
}
```

Output:

```
iob average is 3.0
dob average is 3.3
```

4.11.1 Wild Card Arguments

A wildcard is a syntactic construct that denotes a family of types. A wildcard describes a family of types. The different types of wildcards are given below

Notation	Meaning
<T>	Concrete type

<?>	The unbounded wildcard. It stands for the family of all types
<?super subclass>	A bounded wildcard supertype of T. It stands for the family of all types that are supertypes of Type,type Type being included
<?extends superclass>	A bounded wildcard subtype of T
<U extends T>	U must be a supertype of T

4.11.2 Unbounded Wildcard

The wildcard "?" matches any type of valid stats object.

Example Program:

```
class Stats<T extends Number> {

T[] nums; // array of Number or subclass
// Pass the constructor a reference to
// an array of type Number or subclass.
Stats(T[] o)
{
nums = o;
}
// Return type double in all cases.
double average()
{
double sum = 0.0;
for(int i=0; i < nums.length; i++)
sum += nums[i].doubleValue();
return sum / nums.length;
}
// Determine if two averages are the same.
// Notice the use of the wildcard.
boolean sameAvg(Stats<?> ob)
{
if(average() == ob.average())
return true;
return false;
}
}
// Demonstrate wildcard.
class WildcardDemo
{
public static void main(String args[])
{
Integer inums[] = { 1, 2, 3, 4, 5 };
Stats<Integer> iob = new Stats<Integer>(inums);
double v = iob.average();
System.out.println("iob average is " + v);
Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
Stats<Double> dob = new Stats<Double>(dnums);
double w = dob.average();
System.out.println("dob average is " + w);
Float fnums[] = { 1.0F, 2.0F, 3.0F, 4.0F, 5.0F };
Stats<Float> fob = new Stats<Float>(fnums);
double x = fob.average();
System.out.println("fob average is " + x);
// See which arrays have same average.
System.out.print("Averages of iob and dob ");
if(iob.sameAvg(dob))
System.out.println("are the same.");
else
System.out.println("differ.");
System.out.print("Averages of iob and fob ");
if(iob.sameAvg(fob))
System.out.println("are the same.");
else
System.out.println("differ.");
}
}
```

Output:

```
iob average is 3.0
dob average is 3.3
fob average is 3.0
```

Averages of iob and dob differ.
Averages of iob and fob are the same.

4.11.3 Bounded Wildcards

Wildcard arguments can be bounded in much the same way that a type parameter can be bounded. A bounded wildcard is especially important when you are creating a generic type that will operate on a class hierarchy.
A bounded wildcard specifies either an upper bound or a lower bound for the type argument. This enables you to restrict the types of objects upon which a method will operate. The most common bounded wildcard is the upper bound, which is created using an **extends** clause in much the same way it is used to create a bounded type.

Example Program:

```
class TwoD
{
int x, y;
TwoD(int a, int b)
{
x = a;
y = b;
}
}
// Three-dimensional coordinates.
class ThreeD extends TwoD
{
int z;
ThreeD(int a, int b, int c)
{
super(a, b);
z = c;
}
}
// Four-dimensional coordinates.
class FourD extends ThreeD
{
int t;
```

```
FourD(int a, int b, int c, int d)
{
super(a, b, c);
t = d;
}
}
// This class holds an array of coordinate objects. class Coords<T
extends TwoD>
{
T[] coords;
Coords(T[] o)
{
coords = o;
}
}
// Demonstrate a bounded wildcard.
class BoundedWildcard
{
static void showXY(Coords<?> c)
{
System.out.println("X Y Coordinates:");
for(int i=0; i < c.coords.length; i++) System.out.println(c.coords[i].x
+ " " + c.coords[i].y);
System.out.println();
}
static void showXYZ(Coords<? extends ThreeD> c) {
System.out.println("X Y Z Coordinates:");
for(int i=0; i < c.coords.length; i++) System.out.println(c.coords[i].x
+ " " + c.coords[i].y + " " +
c.coords[i].z);
System.out.println();
}
static void showAll(Coords<? extends FourD> c)
{
System.out.println("X Y Z T Coordinates:");
for(int i=0; i < c.coords.length; i++) System.out.println(c.coords[i].x
+ " " + c.coords[i].y + " " +
c.coords[i].z + " " +
c.coords[i].t);
System.out.println();
}
public static void main(String args[])
{
TwoD td[] = {
new TwoD(0, 0),
new TwoD(7, 9),
new TwoD(18, 4),
new TwoD(-1, -23)
};
Coords<TwoD> tdlocs = new Coords<TwoD>(td);
System.out.println("Contents of tdlocs.");
showXY(tdlocs); // OK, is a TwoD
// showXYZ(tdlocs); // Error, not a ThreeD
// showAll(tdlocs); // Error, not a FourD
// Now, create some FourD objects.
FourD fd[] = {
new FourD(1, 2, 3, 4),
new FourD(6, 8, 14, 8),
new FourD(22, 9, 4, 9),
new FourD(3, -2, -23, 17)
};
Coords<FourD> fdlocs = new Coords<FourD>(fd);
System.out.println("Contents of fdlocs.");
// These are all OK.
showXY(fdlocs);
showXYZ(fdlocs);
showAll(fdlocs);
}
}
```

Output:

```
Contents of tdlocs.
X Y Coordinates:
0 0
7 9
```

```
18 4
-1 -23

Contents of fdlocs.
X Y Coordinates:
1 2
6 8
22 9
3 -2

X Y Z Coordinates:

1 2 3
6 8 14
22 9 4
3 -2 -23

X Y Z T Coordinates:
1 2 3 4
6 8 14 8
22 9 4 9
3 -2 -23 17
```

4.11.4 Inheritance and Generics

A generic class can act as a superclass or be a subclass.

4.11.4.1 Using a Generic Superclass

in a generic hierarchy, any type arguments needed by a generic superclass must be passed up the hierarchy by all subclasses.

Example Program:

```
// A subclass can add its own type parameters.
class Gen<T> {
T ob; // declare an object of type T
// Pass the constructor a reference to
// an object of type T.
Gen(T o)
{
```

```
ob = o;
}
// Return ob.
T getob()
{
return ob;
}
}
// A subclass of Gen that defines a second
// type parameter, called V.
class Gen2<T, V> extends Gen<T>
{
V  ob2;
Gen2(T o, V o2)
{
super(o);
ob2 = o2;
}
V  getob2()
{
return ob2;
}
}
// Create an object of type Gen2.
class HierDemo
{
public static void main(String args[])
{
// Create a Gen2 object for String and Integer.
Gen2<String, Integer> x =
new Gen2<String, Integer>("Value is: ", 99);
System.out.print(x.getob());
System.out.println(x.getob2());
} }
```

Output:

```
Value is: 99
```

4.11.4.2 A Generic Subclass

A subclass can add its own type parameters,if needed.

Example Program:

```
class NonGen
{
int num;
NonGen(int i)
{
num = i;
}
int getnum()
{
return num;
}
}
// A generic subclass.
class Gen<T> extends NonGen
{
T ob; // declare an object of type T
// Pass the constructor a reference to
// an object of type T.
Gen(T o, int i)
{
super(i);
ob = o;
}
// Return ob.
T getob()
{
return ob;
}
}
// Create a Gen object.
class HierDemo2
{
public static void main(String args[])
{
```

```
// Create a Gen object for String.
Gen<String> w = new Gen<String>("Hello", 47);
System.out.print(w.getob() + " ");
System.out.println(w.getnum());
}
}
```

Output:

```
Hello 47
```

4.11.5 Overriding Methods in a Generic Clas

A method in a generic class can be overridden just like any other method.

Example Program:

```
class Gen<T>
{
T ob; // declare an object of type T
// Pass the constructor a reference to
// an object of type T.
Gen(T o)
{
ob = o;
}
// Return ob.
T getob()
{
System.out.print("Gen's getob(): " );
return ob;
}
}
// A subclass of Gen that overrides getob().
class Gen2<T> extends Gen<T>
{
Gen2(T o)
{
super(o);
}
// Override getob().
T getob()
{
System.out.print("Gen2's getob(): ");
return ob;
}
}
// Demonstrate generic method override.
class OverrideDemo
{
public static void main(String args[])
{
// Create a Gen object for Integers.
Gen<Integer> iOb = new Gen<Integer>(88);
// Create a Gen2 object for Integers.
Gen2<Integer> iOb2 = new Gen2<Integer>(99);
// Create a Gen2 object for Strings.
Gen2<String> strOb2 = new Gen2<String> ("Generics Test");
System.out.println(iOb.getob());
System.out.println(iOb2.getob());
System.out.println(strOb2.getob());
}
}
```

Output:

```
Gen's getob(): 88
Gen2's getob(): 99
Gen2's getob(): Generics Test
```

4.12 Restrictions and Limitations

There are a few restrictions that you need to keep in mind when using generics. They involve creating objects of a type parameter, static members, exceptions, and arrays.

4.12.1 Type Parameters Can’t Be Instantiated

It is not possible to create an instance of a type parameter. For example, consider this class:

Example Program:

```
// Can't create an instance of T.
```

```
class Gen<T>
{
T ob;
Gen()
{
ob = new T(); // Illegal!!!
}
}
```

Here, it is illegal to attempt to create an instance of **T**. The reason should be easy to
understand: because **T** does not exist at run time, how would the compiler know what type
of object to create?

4.12.2 Restrictions on Static Members

No **static** member can use a type parameter declared by the enclosing class. For example,
both of the **static** members of this class are illegal:

Example Program:

```
class Wrong<T>
{
// Wrong, no static variables of type T.
static T ob;
// Wrong, no static method can use T.
static T getob()
{
return ob;
}
}
```

Although you can't declare **static** members that use a type parameter declared by the enclosing class, you can declare **static** generic methods, which define their own type parameters

4.12.3 Generic Array Restrictions

There are two important generics restrictions that apply to arrays. First, you cannot instantiate an array whose element type is a type parameter. Second, you cannot create an array of typespecific generic references.

Example Program:

```
// Generics and arrays.
class Gen<T extends Number>
{
T ob;
T vals[]; // OK

Gen(T o, T[] nums)
{
ob = o;
// This statement is illegal.
// vals = new T[10]; // can't create an array of T
// But, this statement is OK.
vals = nums; // OK to assign reference to existent array
}
}
class GenArrays
{
public static void main(String args[])
{
Integer n[] = { 1, 2, 3, 4, 5 };
Gen<Integer> iOb = new Gen<Integer>(50, n);
// Can't create an array of type-specific generic references.
// Gen<Integer> gens[] = new Gen<Integer>[10]; // Wrong!
// This is OK.
Gen<?> gens[] = new Gen<?>[10]; // OK
}
}
```

4.12.4 Generic Exception Restriction A generic class cannot extend **Throwable**. This means that you cannot create generic exception classes.

2 Marks Questions and Answers

1. Difference between multitasking and multithreading.

Multithreading	**Multitasking**

The system executes multiple threads of the same or different processes at the same time.	The system allows executing multiple programs and tasks at the same time
CPU has to **switch** between **multiple threads** to make it appear that all threads are running simultaneously	**CPU** has to **switch** between **multiple programs** so that it appears that multiple programs are running simultaneously.
Threads belonging to the same process **shares the same memory and resources** as that of the process.	Multitasking allocates **separate memory and resources** for each process/program

2. What is thread?

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.

Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.

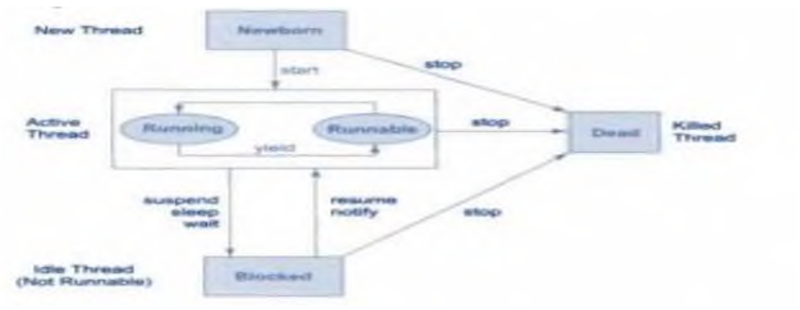
3. Write the states of a thread.

During the life time of a thread, it enters into various states.

The states are

- Newborn State
- Runnable State
- Running State
- Blocked State
- Dead State

4. Draw the life cycle of thread.



5. What is the role of Newborn State?

When we create a thread object, the thread is born and is said to be newborn -state. In this state, we can do the following tasks

- Schedule a thread for running using start() method
- Kill a thread using stop() method

6. What is the role of Runnable State?

The runnable state means that the thread is ready for execution and is waiting for the availability of the processor.The thread is waiting in the queue for its execution.If all threads have equal priority,then they are given time slots for execution in round-robin fashion,that means first- come,first-serve manner.

7. Write down the role of Running State.

Running means that the thread is allotted with the processor for its execution.The thread runs until higher priority thread comes.A running thread may relinquish its control in one of the following situations

8. Write down the role of Blocked State.

A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently the running state. This happens when the thread is suspended, sleeping, or waiting in order to satisfy certain requirements. A blocked thread is considered “not runnable” but not dead and fully qualified to run again.

9. Write down the role of Dead State.

Every thread has a life cycle. A running thread ends its life when it has completed executing its run() method. It is a natural death. We can kill it by sending the stop message to it at any state.

10. What are the ways of creating threads?

A new thread can be created in two ways

- You can implement the **Runnable** interface.
- You can extend the **Thread** class, itself.

11. What are the methods in Thread class?

Method	**Meaning**
getName()	Obtain a thread's name.
getPriority()	Obtain a thread's priority.
isAlive()	Determine if a thread is still running.
join()	Wait for a thread to terminate.
run()	Entry point for the thread.
sleep()	Suspend a thread for a period of time.
start()	Start a thread by calling its run method.

12. What is the difference between yielding and sleeping?

When a task invokes its yield() method, it returns to the ready state. When a task invokes its sleep() method, it returns to the waiting state.

13. Can I implement my own start() method?

The Thread start() method is not marked final, but should not be overridden. This method contains the code that creates a new executable thread and is very specialised. Your threaded application should either pass a Runnable type to a new Thread, or extend Thread and override the run() method.

14. How do you set the priority to a thread?

Each thread is assigned a priority. Based on the priority, the thread

will be scheduled for running. The threads of the same priority are given equal treatment by the Java scheduler, they share the processor on a first come, first serve basis.
The thread is set with the priority, we can use setPriority() method.
Syntax:

```
ThreadName.setPriority(intNumber);
```

The intNumber is an integer value to which the thread's priority is set. The Thread class defines several priority constants:

```
MIN_PRORITY = 1
NORM_PRORITY = 5
MAX_PRORITY = 10
```

15. How to synchronize the threads in Java?

Synchronization in java *is the* capability to control the access of multiple threads to any shared resource*.*
When we want to share the resource with multiple threads, we can use the Java synchronization concept. So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

16. Write the purpose of using synchronization in threads?

- To prevent thread interference
- To prevent consistency problem

17. What are the types of Thread Synchronization?

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. static synchronization.
2. Cooperation (Inter-thread communication in java)