OOC IAT-2 April 2019
Solution to IAT 2 Question paper

1 (a)	Define Synchronization? Explain how Synchronization is achieved in JAVA?	[10]	CO4	L3

Synchronization:

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.
- Key to synchronization is the concept of the monitor (also called a semaphore).
- A monitor is an object that is used as a mutually exclusive lock or **mutex**
- only one thread can own a monitor at a give time.
- When a thread acquires a lock, it is said to have entered the monitor.
- All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor.
- A thread that owns a monitor can reenter the same monitor if it so desires.
- As Java implements synchronization through language elements, most of the complexity associated with synchronization has been eliminated.

Two ways of synchronization:

- We can synchronize code in two ways
  1. Using synchronized methods
  2. using synchronized statement

1. Using Synchronized Methods:

- Synchronization is easy in Java, because all objects have their own implicit monitor associated with them.
- To enter an object's monitor, just call a method that has been modified with the synchronized keyword.
- While a thread is inside a synchronized method, all other threads that try to call it on the same instance have to wait.
- To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

Program Explanation:

- The example program below has three classes
- The first one, **Callme** class has a single method named **call()**. The **call()** method takes a String parameter called **msg**. This method tries to print the **msg** string inside of square brackets. But, after call() prints the opening bracket and the msg string, it calls **Thread.sleep(1000),** which pauses the current thread for one second.

- The second class is **Caller**. The constructor of this class takes a reference to an instance of the **Callme** class and a **String**, which are stored in **target** and **msg**, respectively.
- The constructor also creates a new thread that will call this object's **run()** method. The thread started immediately.
- The **run()** method of caller calls the **call()** method on the target instance of **Callme**, passing in the **msg** string.
- Finally, the third class, **Synch** class starts by creating a single instance of **Callme**, and three instances of **Caller**, each with a unique message string. The same instance of **Callme** is passed to each Caller.
- we must **serialize** access to call(). That is, we must restrict its access to only one thread at a time.
- To do this, we have to precede call()'s definition with the keyword **synchronized** as shown
  class Callme {
  synchronized void call (String msg) {
  ….
- This prevents other threads from entering call() while another thread is using it.

Program:

```
// This program is synchronized
class Callme {
    synchronized void call(String msg) {
        System.out.print("[" + msg);
        try {
            Thread.sleep(1000);
        }
        catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}

class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread (this);
        t.start();
    }
```

```
    public void run()  {
       target.call(msg);
    }
}

class Synch  {
    public static void main(String args[])  {
       Callme target = new Callme();
       Caller ob1 = new Caller (target, "Hello");
       Caller ob2 = new Caller(target, "Synchronized");
       Caller ob3 = new Caller (target, "World");

       // wait for the threads to end
       try   {
          ob1.t.join();
          ob2.t.join();
          ob3.t.join();
       }
       catch (InterruptedException e)  {
          System.out.println("Interrupted");
       }
    }
}
```

Output:

```
$ javac Synch.java
$ java Synch
[Hello]
[World]
[Synchronized]
```

2. The synchronized Statement:

- Creating synchronized methods within classes that you create is an easy and effective means of achieving synchronization, but, it will not work in all cases.
- If you want to synchronize access to objects of a class that was not designed for multi-threaded access, the class does not use synchronized methods.
- If the class was not created by you, but by a third party and you do not have access to the source code, you can't add synchronized to the appropriate methods within the class.
- The solution to this problem is you put calls to the methods defined by this class inside a synchronized block.
- The general form of the synchronized statement

  ```
  synchronized (object) {
  // Statements to be synchronized
  }
  ```

- The object is a reference to the object being synchronized.
- A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor.
- Example program which uses synchronized block within run() method

```
// This program uses synchronized block
class Callme  {
    void call(String msg)  {
        System.out.print("[" + msg);
        try  {
            Thread.sleep(1000);
        }
        catch (InterruptedException e)  {
            System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}

class Caller implements Runnable   {
    String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s)  {
        target = targ;
        msg = s;
        t = new Thread (this);
        t.start();
    }

    // Synchronize calls to call()
    public void run()  {
        synchronized (target)  {  // synchronized block
            target.call(msg);
        }
    }
}

class Synch1  {
    public static void main(String args[])  {
        Callme target = new Callme();
        Caller ob1 = new Caller (target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller (target, "World");

        // wait for the threads to end
```

```
        try   {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        }
        catch (InterruptedException e)  {
            System.out.println("Interrupted");
        }
    }
}
```

Output:

```
$ javac Synch1.java
$ java Synch1
[Hello]
[World]
[Synchronized]
```

Program Explanation:

- In the above program, the call() method is not modified by **synchronized**. Instead, the **synchronized** statement is used inside Caller's run() method.

==========XXX==========

2 (a)	**Briefly explain the role of interface in implementing Multiple Inheritance in JAVA.**	**[5]**	**CO3**	**L3**

Multiple Inheritance is a feature of object oriented concept, where a class can inherit properties of more than one parent class.

The problem occurs when there exist methods with same signature in both the super classes and subclass. On calling the method, the compiler cannot determine which class method to be called and even on calling which class method gets the priority.

Therefore, in order to avoid such complications **Java does not support multiple inheritance of classes.** But, a class can implement two or more interfaces.

One significant difference between classes and interfaces is that classes can have fields whereas interfaces cannot. Also, you can instantiate a class to create an object, which you cannot do with interfaces.

Because interfaces do not contain fields, you do not have to worry about problems that result from multiple inheritance of state.

A class can implement more than one interface, which can contain default methods that have the same name. The Java compiler provides some rules to determine which default method a particular class uses.

Example program to demonstrate multiple inheritance using interface:

```
// Program to demonstrate multiple inheritance using interface
// Define the interface I1
interface I1  {
    void showI1() ;
}
// Define the interface I2
interface I2  {
    void showI2();
}
// Define MInheritance that implements both I1 and I2
class MInheritance implements I1, I2   {
   // Implement I1's interface
    public void showI1()  {
        System.out.println("Inside showI1");
   }
   // Implement I2's interface
   public void showI2()  {
       System.out.println("Inside showI2");
   }
 }
 class TestMI  {
      public static void main(String args[])  {
       MInheritance MI = new MInheritance();
       MI.showI1();
       MI.showI2();
   }
}
```

Output:

```
$ javac TestMI.java
$ java TestMI
Inside showI1
Inisde showI2
```

Program Explanation:

- The program defines two interfaces I1 and I2.
- The class Minheritance implements from both interfaces I1 and I2. The method of both the interfaces showI1() and showI2 are implemented in this class.
- The class TestMI creates an instance of Minheritance and calls both the methods showI1() and showI2().

2(b)	**Explain function overloading with example.**	[5]	CO1	L3

Function Overloading:

- C++ allows two or more functions to have the same name, but they must have different signatures.
- **Signature of a function means the number, type, and sequence of formal arguments of the function.**
- In order to distinguish amongst the functions with the same name, the compiler expects their signature to be different.
- Depending upon the type of parameters that are passed to the function call, the compiler decides which of the available definitions will be invoked.
- For this, function prototypes should be provided to the compiler for matching the function calls.
- Accordingly the linker, during the link time, links the function call with the correct function definition.

Example program to demonstrate function overloading:

```
#include<iostream>
using namespace std;

int add(int,int);
int add (int,int,int);

int main()
{
        int x,y;
        x = add(10,20);
        y = add(30, 40, 50);
        cout << x << endl<< y << endl;
}

int add(int a, int b)
{
        return (a+b);
}

int add(int a, int b, int c)
{
        return (a+b+c);
```

}

Output:

```
$ g++ funcOverload.C
$ ./a.out
30
120
```

- Like ordinary functions, the definition of overloaded functions are also put in libraries and function prototypes are placed in header files.
- The two function prototypes at the beginning of the program tell the compiler the two different ways in which the add() function can be called.
- When the compiler encounters the two distinct calls to the add() function, it already has the prototypes to satisfy them both. Hence, the compilation phase is completed successfully.
- During the linking, the linker finds the two necessary definitions of the add() function and hence links successfully to create the executable file.
- The compiler decides which function is to be called based upon the number, type and sequence of parameters that are passed to the function call.
- When the compiler encounters the first function call,
  x = add (10, 20);
  it decides that the function that takes two integers as formal arguments is to be executed.
- Accordingly, the linker then searches for the definition of the add() function where there are two integers as formal arguments.
- Similarly, the second call to the add() function
  y = add(30, 40, 50);
  is also handled by compiler and the linker.
- Since the function prototyping is mandatory in C++, it is possible for the compiler to support function overloading properly.
- Function overloading is also known as function polymorphism.
- Function polymorphism is static in nature because the function definition to be executed is selected by the compiler during compile time itself. Thus an overloaded function is said to exhibit static polymorphism.

==========XXX==========

3 (a)	**Define Custom Exception? Explain How Custom Exception is created with an example.**	**[10]**	**CO3**	**L3**

Custom Exception:

- A user defined exception is known as custom exception.
- Java custom exceptions are used to customize the exception according to user need.
- With the help of custom exception, we can have our own exception and message.
- We can create our own exception types to handle situations specific to our applications.
- To create your own Exception, define a subclass of Exception, which is inturn a subclass of **Throwable**.
- Your subclass need not implement anything. Its existence in the type system allows you to use them as exceptions.

- The **Exception** class does not define any methods of its own. It inherits those methods provided by Throwable.
- Thus, all exceptions, including those that you create, have the methods defined by Throwable available to them.
- To name a few, the methods are

1. Throwable fillInStackTrace()
2. Throwable getCause()
3. String getLocalizedMessage()
4. String getMessage()
5. StackTraceElement [] getStackTrace()
6. Throwable initCause (Throwble causeExc)
7. void printStackTrace()
8. **String toString ()**

- You can override one or more of these methods in exception classes that you create.
- Exception defines four constructors

1. Exception()
2. Exception(String msg)
3. Throwable(Throwable causeExc)
4. Throwable(String msg, Throwable causeExc)

Example Program

Program Explanation:

- The following example declares a new subclass of Exception and then uses that subclass to signal an error condition in a method.
- It overrides the toString() method, allowing a description of the exception to be displayed.

Program:

```
//This program creates a custom exception type
class MyException extends Exception {
 private int detail;

 MyException(int a) {
 detail = a;
 }
 public String toString() {
 return "MyException[" + detail + "]";
}
}


class ExceptionDemo {
 static void compute (int a) throws MyException {
```

```
System.out.println("Called compute (" + a + ")");
if (a > 10)
 throw new MyException(a);
System.out.println("Normal exit");
}
  public static void main (String args[]) {
 try {
 compute(1);
 compute(20);
 }
 catch (MyException e) {
 System.out.println("Caught " + e);
 }
}
}
```

Output:

```
$ javac ExceptionDemo.java
$ java ExceptionDemo
Called compute (1)
Normal exit
Called compute (20)
Caught MyException[20]
```

==========XXX==========

4 (a)	**Define Package? Describe the various levels of access protection for Packages and their implication.**	**[10]**	**CO3**	**L3**

Package:

- Packages are containers for classes that are used to keep the class name space compartmentalized.
- For example, a package allows you to create a class named List, which you can store in your own package without concern that it will collide with some other class named List stored elsewhere.
- Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.
- Packages act as containers for classes and other subordinate packages.
- Classes act as containers for data and code. The class is Java's smallest unit of abstraction.
- **Java addresses four categories of visibility for class members, they are**

1. Subclasses in the same package
2. Non-subclasses in the same package
3. Subclasses in different packages
4. Classes that are neither in the same package nor subclasses

- The three access specifiers, **private, public, and protected,** provide a variety of ways to produce the many levels of access required by these categories.
- Anything declared **public** can be accessed from anywhere.
- Anything declared **private** cannot be seen outside of its class.
- When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the **default** access.
- If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.
- A non-nested class has only two possible access levels: default and public. When a class is declared as public, it is accessible by any other code.
- If a class has default access, then it can only be accessed by other code within its same package.
- When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class.
- The table shows class member access

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

==========XXX==========

5 (a)	**What you mean by Thread? Explain how thread can be created in JAVA?**	**[10]**	**CO4**	**L3**

- A **thread** is a path of execution within a process. A process can contain multiple threads.
- Java provides built-in support for multi threaded programming. A multi-threaded program contains two or more parts that can run concurrently. Each part of such a program is called a **thread**, and each thread defines a separate path of execution. Thus multi threading is a specialized form of multitasking.

Creating a Thread

A thread can be created in two ways
1. we can implement the **Runnable** interface
2. we can extend the **Thread** class

1. Creating a thread by Implementing Runnable interface:

- To create a thread by implementing Runnable interface, create a class that implements the **Runnable** interface.
- Runnable abstracts a unit of executable code.
- To implement Runnable, a class has to implement a single method called run(), which is declared like this -
  public void run()
- Inside run(), you will define the code that constitutes the new thread.
- Run() can call other methods, use other classes and declare variables just like the main thread can.
- The only difference is that run() establishes the entry point for another concurrent thread of execution within your program. This thread will end when run() returns.
- **After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class.** Thread defines several constructors. One such constructor is
  Thread (Runnable threadOb, String threadName)
- In this constructor, threadOb is an instance of a class that implements the Runnable interface. This defines where execution of the thread will begin. The name of the new thread is specified by threadName.
- After the new thread is created, it will not start running until you call its start() method, which is declared within Thread. In essence, start executes a call to run(). The start() method is shown here
  void start()
- Example program that creates a new thread and starts it running:

```
// create a second thread
class NewThread implements Runnable {
   Thread t;
   NewThread() {
      //Create a new, second thread
      t = new Thread(this, "Demo Thread");
      System.out.println("Child thread :" + t);
      t.start();  // start the thread
   }
   // This is the entry point for the second thread
   public void run()  {
      try {
         for (int i=5; i >0;i--)  {
            System.out.println("child Thread:" + i);
            Thread.sleep(500);
         }
```

```
        }
        catch (InterruptedException e)  {
            System.out.println("Child Interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}

class ThreadDemo  {
    public static void main(String args[])  {
        new NewThread();  // create a new thread
        try {
            for(int i=5;i>0;i--)   {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        }
        catch  (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting");
    }
}

$ javac ThreadDemo.java
$ java ThreadDemo
Child thread :Thread[Demo Thread,5,main]
Main Thread: 5
child Thread:5
child Thread:4
Main Thread: 4
child Thread:3
child Thread:2
Main Thread: 3
child Thread:1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting
```

Note:

- In a multi-threaded program, often the main thread must be the last thread to finish running.
- In older JVMs, if the main thread finishes before a child thread has completed, then the Java run-time system may “hang”.

- The above program ensures that the main thread finished last, because the main thread sleeps for 1000 milliseconds between iterations, but the child sleeps for only 500 milliseconds.
- This causes the child thread to terminate earlier than the main thread.

2. Creating a thread by extending Thread class:

- The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class.
- The extending class must override the run() method, which is the entry point for the new thread.
- It must also call start() to begin execution of the new thread.
- The following is the example program -

```
// Create a second thread by extending Thread
class NewThread extends Thread  {
   NewThread ()  {
      // Create a new, second thread
      super("Demo Thread");
      System.out.println("Child thread: " + this);
      start();  // Start the thread
   }

   // This is the entry point for the second thread.
   public void run()  {
      try  {
         for(int i = 5;i >0; i--)  {
            System.out.println("Child thread :" + i);
            Thread.sleep(500);
         }
      }
      catch (InterruptedException e)   {
         System.out.println("Child interrupted.");
      }
      System.out.println("Exiting child thread.");
   }
}

class ExtendThread   {
   public static void main(String args[])   {
      new NewThread();   // Create a new thread
      try   {
         for(int i = 5;i>0;i--)   {
            System.out.println("Main thread : " + i);
            Thread.sleep(1000);
         }
      }
      catch (InterruptedException e) {
```

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

Output:

```
$ javac ExtendThread.java
$ java ExtendThread
Child thread: Thread[Demo Thread,5,main]
Child thread :5
Main thread : 5
Child thread :4
Main thread : 4
Child thread :3
Child thread :2
Main thread : 3
Child thread :1
Exiting child thread.
Main thread : 2
Main thread : 1
Main thread exiting.
```

Program Explanation:

This program generates the same output as the previous version.
The child thread is created by instantiating an object of **NewThread**, which is derived from **Thread**.
Notice the call to **super**() inside **NewThread**. This invokes the following form of the **Thread** constructor:

public Thread (String threadName)
threadName specifies the name of the thread.

==========XXX==========

6 (a)	**Define Reference variable? Explain. Write C++ program to swap two integers. Display the values Before and after swapping using reference variable.**	[5]	CO1	L2

Ans:

- A reference variable can be defined as a reference for an existing variable.
- It shares the memory location with an existing variable.
- The syntax for declaring a reference variable is as follows -
  <data-type> & <ref-var-name> = < existing-var-name>;
- Example:
  int & iRef = x;

- iRef is a reference to x. This means that although iRef and x have separate entries in the OS, their addresses are actually the same.
- Thus a change in the value of x will naturally reflect in iRef and vice versa.
- Reference variables must be initialized at the time of declaration otherwise the compiler will not know what address it has to record for the reference variable.
- After their creation, reference variable function just like any other variable.
- The value of a reference variable can be read in the same way as the value of an ordinary variable is read.
- A reference variable can be a function argument and thus change the value of the parameter that is passed to it in the function call.
- Functions can return by reference also.

C++ Program to swap two integers using reference variable

```
#include <iostream>
using namespace std;
void swap(int &,int &);
int main()
{
        int a,b;
        cout << "Enter Value of A :";
        cin >> a;
        cout << "Enter value of B:";
        cin >> b;
        cout << "Before swapping  Value of A is " << a ;
        cout << "Value of B is " << b << endl;
        swap(a,b);
        cout << "After Swapping " << endl;
        cout << "Value of A is " << a << endl;
        cout << "Value of B is " << b << endl;
}

void swap (int & a, int & b)
{
        int temp;
        temp = a;
        a = b;
        b = temp;
        cout << "Inside swap function after swapping " << endl;
        cout << "Value of A is " << a << endl;
        cout << "Value of B is  " << b << endl;
}
```

Output:

```
$ g++ swap.C
$ ./a.out
Enter Value of A :34
Enter value of B:98
Before swapping  Value of A is 34Value of B is 98
Inside swap function after swapping
Value of A is 98
Value of B is  34
```

After Swapping
Value of A is 98
Value of B is 34

6(b)	**Differentiate between Procedure Oriented Programming and Object Oriented programming.**	**[5]**	**CO1**	**L2**

#	Procedure Oriented Programming	Object Oriented Programming
1	Program is divided into small parts called functions	Program is divided into parts called objects
2	Focus is on procedures. The code is centered around procedures.	Focus is on data. Code is centered around data.
3	Procedures are dissociated from data and are not part of it.	Procedures are bound to the data.
4	Data is not secure. Compilers that implement the procedure-oriented programming system do not prevent unauthorized functions from accessing/manipulating structure variables.	Enables data security by throwing compile time errors against piece of code that violate the prohibition.
5	Data is not initialized	Provides guaranteed initialization of of data. Programmers can ensure a guaranteed initialization of data members of structure variables to the desired values.
6.	Overloading is not supported	Supports overloading of operators and functions
7	Top-down approach	Bottom-up approach
8	Doesn’t support inheritance	Supports inheritance which allows one structure to inherit the characteristics of an existing structure.
9	Doesn’t support polymorphism	Supports polymorphism which allows functions with different set of formal arguments to have the same name.
10	Doesn’t Supports Encapsulation	Supports encapsulation, data and functions that act upon the data are enclosed within a single unit called class.

==========XXX==========

7 (a)	With example, give two uses of super	[5]	CO3	L5

- super is the keyword using which the subclass refers to its immediate superclass.
- super has two general forms
  1. The first calls the superclass' constructor.
  2. The second is used to access a member of the superclass that has been hidden by member of a subclass.

1. Using super to call superclass constructors:

- A subclass can call a constructor defined by its superclass by use of the following form of superclass
  super(arg-list);
  Here, arg-list specifies any arguments needed by the constructor in the superclass.
- super() must always be the first statement executed inside a subclass' constructor.
- Example program to demonstrate the use of super

```
// Example program to demonstrate super
// create a superclass
class Box {
        double width;
        double height;
        double depth;

        // construct clone of an object
        Box(Box ob) { // pass object to constructor
                width = ob.width;
                height = ob.height;
                depth = ob.depth;
        }

        // constructor used when all dimensions specified
        Box (double w, double h, double d) {
                width = w;
                height = h;
                depth = d;
        }

        // Constructor when no dimensions specified
        Box () {
                width = -1;
                height = -1;
                depth = -1;
        }
```

```
        // Constructor used when cube is created
        Box ( double len) {
                width = height = depth =len;
        }

        // Compute and return volume
        double volume() {
                return height * depth * width;
        }
}

// Here, Box is extended to include weight
class BoxWeight extends Box {
        double weight; // weight of Box


        // construct clone of an object
        BoxWeight(BoxWeight ob) { // pass object to constructor
                 super(ob);
                weight = ob.weight;
        }

        // constructor when all parameters are specified
        BoxWeight (double w, double h, double d, double m) {
                super(w,h,d);
                weight = m;
        }

        // default constructor
        BoxWeight() {
                super();
                weight = -1;
        }

        // Constructor used when cube is created
        BoxWeight(double len, double m) {
                super(len);
                weight = m;
        }
}

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

```
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
        BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
        BoxWeight mybox3 = new BoxWeight(); // default
        BoxWeight mycube = new BoxWeight(3, 2);
        BoxWeight myclone = new BoxWeight(mybox1);

        double vol;

        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        System.out.println("Weight of mybox1 is " + mybox1.weight);
        System.out.println();

        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        System.out.println("Weight of mycube is " + mycube.weight);
        System.out.println();
        }
}
```

Output:

```
$ javac Super.java
$ java Super
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3


Volume of mybox2 is 24.0
Weight of mybox2 is 0.076


Volume of mybox3 is -1.0
Weight of mybox3 is -1.0


Volume of myclone is 3000.0
Weight of myclone is 34.3


Volume of mycube is 27.0
Weight of mycube is 2.0
```

Program Explanation:

1. In the above program, BoxWeight() calls super() with the arguments w, h and d.

2. This causes the Box() constructor to be called, which Initializes width, height and depth using these values.
3. BoxWeight no longer initialized these values itself.
4. It only needs to initialize the value unique to it, that is weight.
5. Since constructors can be overloaded, super() can be called using any form defined by the superclass. The constructor executed will be the one that matches the argument.

Key points to remember:

- super() always refers to the superclass immediately above the calling class.
- This is true even in multilevel hierarchy.
- super() must always be the first statement executed inside a subclass constructor.

2. Second use of super to access a member of the superclass:

- The second form of super always refers to the superclass of the subclass in which it is used.
- The general form is
  super.member
  Here, member can be either a method or an instance variable.
- The second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.
- Example program to demonstrate second use of superclass

```
// using super to overcome name hiding
class A {
int i;
}
// create a subclass by extending class A
class B extends A {
int i; // this i hides i in A
B(int a, int b) {
super.i = a; // i in A
i = b;// i in B
}

void show() {
System.out.println("i in superclass: " + super.i);
System.out.println("i in subclass:" + i);
}
}


class UseSuper {
public static void main(String args[]) {
B subOb = new B(1,2);
```

```
subOb.show();
}
}
```

Output:

```
$ javac UseSuper.java
$ java UseSuper
i in superclass: 1
i in subclass:2
```

(b)	**With the syntax explain the use of isAlive() and join().**	**[5]**	**CO4**	**L5**

Using isAlive() and join():

- In a multi-threaded program, often the main thread must be the last thread to finish running.
- This is accomplished by calling sleep() within main(), with a long enough delay to ensure that all child threads terminate prior to the main thread.
- But, this is not the right method as the child might take more than the sleep time.
- If main thread comes to know if all child threads has ended or not then, problem is solved.
- The solution to this problem is there should be a way to know if the thread has ended or not.
- There are two ways to determine whether a thread has finished.

1. isAlive() method:

- First, you can call **isAlive**() on the thread. Its general form is
  final boolean isAlive()
- The isAlive method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.

2. join() method:

- The second method that will be more commonly used to wait for a thread to finish is called **join**(). Its general form is
  final void join() throws InterruptedException
- This method waits until the tread on which it is called terminates.
- **Its name comes from the concept of the called thread waiting until the specified thread joins.**
- Additional forms of join() allows to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Example program to demonstrate the use of join() and isAlive():

```
// program DemoJoin.java
// using join() to wait for threads to finish.
```

```
class NewThread implements Runnable  {
   String name;  // name of thread
   Thread t;

   NewThread (String threadname)  {
      name = threadname;
      t = new Thread (this, name);
      System.out.println("New thread: " + t);
      t.start();  // Start the thread
   }

   // This is the entry point for the thread.
   public void run()  {
      try  {
         for(int i = 5;i >0; i--)  {
            System.out.println(name + ": " + i);
            Thread.sleep(1000);
         }
      }
      catch (InterruptedException e)   {
         System.out.println(name  + " Interrupted");
      }
      System.out.println(name + "exiting.");
   }
}

class DemoJoin   {
   public static void main(String args[])   {
      NewThread ob1 = new NewThread ("One");   // Start threads
      NewThread ob2 = new NewThread ("Two");
      NewThread ob3 = new NewThread ("Three");

      System.out.println("Thread one is alive: " + ob1.t.isAlive());
        System.out.println("Thread Two is alive: " + ob2.t.isAlive());
        System.out.println("Thread Three is alive: " + ob3.t.isAlive());

        // wait for threads to finish
        try   {
           System.out.println("Waiting for threads to finish.");
           ob1.t.join();
           ob2.t.join();
           ob3.t.join();
        }
      catch (InterruptedException e) {
         System.out.println("Main thread Interrupted.");
      }
```

```
        System.out.println("Thread one is alive: " + ob1.t.isAlive());
          System.out.println("Thread Two is alive: " + ob2.t.isAlive());
          System.out.println("Thread Three is alive: " + ob3.t.isAlive());

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

Output:

```
$ javac DemoJoin.java
$ java DemoJoin
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
One: 5
New thread: Thread[Three,5,main]
Two: 5
Three: 5
Thread one is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Oneexiting.
Twoexiting.
Threeexiting.
Thread one is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

Note:

- You can observe that, after the calls to join() return, the threads have stopped executing.

==========XXX==========