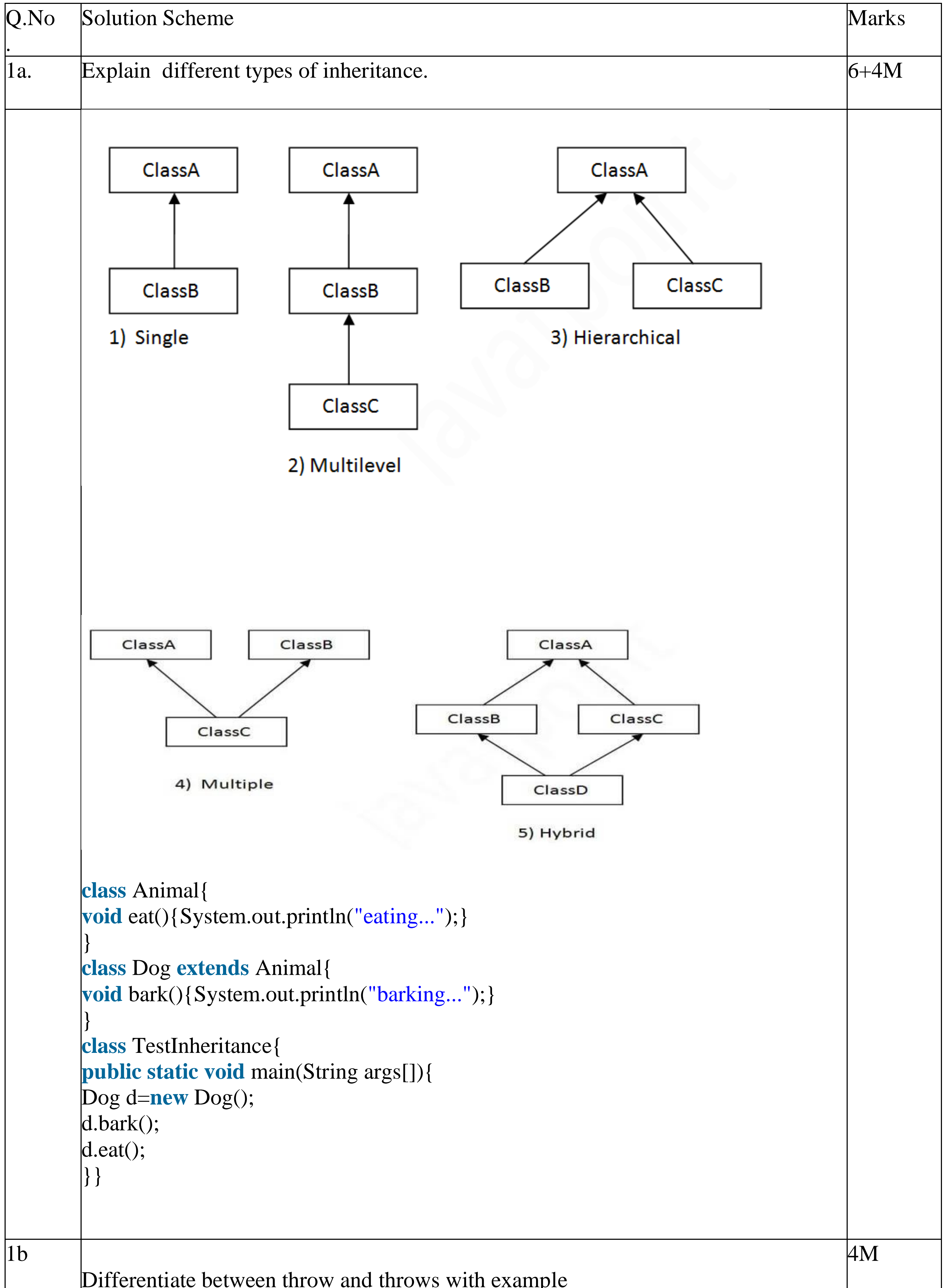
SOLUTIONS & SCHEME IAT3
Date 29th Aug 2022

Course Name: Object oriented programming concepts Course Code: 18CS45

Q.No.	Solution Scheme	Marks
1a.	Explain different types of inheritance.	6+4M




```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```

Q.No.	Solution Scheme	Marks
1b	Differentiate between throw and throws with example	4M

Sr. No.	Key	throw	throws
1	Definition	Throw is a keyword which is used to throw an exception explicitly in the program inside a function or inside a block of code.	Throws is a keyword used in the method signature used to declare an exception which might get thrown by the function while executing the code.
2	Internal implementation	Internally throw is implemented as it is allowed to throw only single exception at a time i.e we cannot throw multiple exception with throw keyword.	On other hand we can declare multiple exceptions with throws keyword that could get thrown by the function where throws keyword is used.
3	Type of exception	With throw keyword we can propagate only unchecked exception i.e checked exception cannot be propagated using throw.	On other hand with throws keyword both checked and unchecked exceptions can be declared and for the propagation checked exception must use throws keyword followed by specific exception class name.
4	Syntax	Syntax wise throw keyword is followed by the instance variable.	On other hand syntax wise throws keyword is followed by exception class names.
5	Declaration	In order to use throw keyword we should know that throw keyword is used within the method.	On other hand throws keyword is used with the method signature.

```
public class JavaTester{
  public void checkAge(int age){
    if(age<18)
      throw new ArithmeticException("Not Eligible for voting");
    else
      System.out.println("Eligible for voting");
  }
  public static void main(String args[]){
    JavaTester obj = new JavaTester();
    obj.checkAge(13);
    System.out.println("End Of Program");
  }
}
```

<table>
<tr><td></td><td><pre>public class JavaTester{
  public int division(int a, int b) throws ArithmeticException{
    int t = a/b;
    return t;
  }
  public static void main(String args[]){
    JavaTester obj = new JavaTester();
    try{
      System.out.println(obj.division(15,0));
    }
    catch(ArithmeticException e){
      System.out.println("You shouldn't divide number by zero");
    }
  }
}</pre></td><td></td></tr>
<tr><td>2a</td><td>What is constructor? list different types of constructors? How is constructor different from member function?
Discuss the following terms with example :
i) super ii) final iii) finalize() method (iv) Garbage Collector</td><td></td></tr>
<tr><td></td><td><pre>/* Here, Box uses a constructor to initialize the
dimensions of a box.
*/
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box() {
System.out.println("Constructing Box");
width = 10;
height = 10;
depth = 10;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo6 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}</pre></td><td></td></tr>
</table>

Parameterized Constructors

```
/* Here, Box uses a parameterized constructor to
initialize the dimensions of a box.
*/
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}

class BoxDemo7 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

The output from this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

2b Super

Garbage Collection

Since objects are dynamically allocated by using the new operator, you might be wondering
how such objects are destroyed and their memory released for later reallocation. In some
languages, such as C++, dynamically allocated objects must be manually released by use of
a delete operator. Java takes a different approach; it handles deallocation for you automatically.

The technique that accomplishes this is called garbage collection. It works like this: when no
references to an object exist, that object is assumed to be no longer needed, and the memory
occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++.
Garbage collection only occurs sporadically (if at all) during the execution of your program.
It will not occur simply because one or more objects exist that are no longer used. Furthermore,
different Java run-time implementations will take varying approaches to garbage collection,
but for the most part, you should not have to think about it while writing your programs.

he finalize( ) Method
Sometimes an object will need to perform some action when it is destroyed. For example, if
an object is holding some non-Java resource such as a file handle or character font, then you
might want to make sure these resources are freed before an object is destroyed. To handle To add a finalizer to a class, you simply define the finalize( ) method. The Java run time
calls that method whenever it is about to recycle an object of that class. Inside the finalize( )
method, you will specify those actions that must be performed before an object is destroyed.
The garbage collector runs periodically, checking for objects that are no longer referenced by
any running state or indirectly through other referenced objects. Right before an asset is freed,
the Java run time calls the finalize( ) method on the object.
The finalize( ) method has this general form:

```
protected void finalize( )
{
// finalization code here
}
```

Here, the keyword protected is a specifier that prevents access to finalize( ) by code defined
outside its class.

Final
The keyword **final** has three uses. First, it can be used to create the equivalent of a named constant. This use was described in the preceding chapter. The other two uses of **final** apply to inheritance.

Using final to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden. The following fragment illustrates **final**:

```
class A {
```

```
    final void meth() {
      System.out.println("This is a final method.");
    }
}
  class B extends A {
    void meth() { // ERROR! Can't override.
      System.out.println("Illegal!");
    }
}
```

Because **meth( )** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result.

Using final to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too. As you might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a **final** class:

```
  final class A {
    // ...
}
  // The following class is illegal.
  class B extends A { // ERROR! Can't subclass A
// ... }
```

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.

Super:

super has two general forms. The first calls the superclass' constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass. Each use is examined here.

Using super to Call Superclass Constructors

A Second Use for super

3	Define a class box with data members : width, height and length and define three overloaded constructions to (i). Pass values for all 3 members (ii). Initialize all values to -1 (iii). Assign same value to all three	

```
// A complete implementation of BoxWeight.
class Box {
  private double width;
  private double height;
  private 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 used when no dimensions specified
```

```
 Box() {
   width = -1;  // use -1 to indicate
   height = -1; // an uninitialized
   depth = -1;  // box
}
 // constructor used when cube is created
 Box(double len) {
   width = height = depth = len;
 }
 // compute and return volume
 double volume() {
   return width * height * depth;
 }
}
// BoxWeight now fully implements all constructors.
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); // call superclass constructor
weight = m; }
 // default constructor
 BoxWeight() {
super();
weight = -1; }
 // constructor used when cube is created
 BoxWeight(double len, double m) {
super(len);
weight = m; }
}
class DemoSuper {
 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 mybox2 is " + mybox2.weight);
   System.out.println();
   vol = mybox3.volume();
   System.out.println("Volume of mybox3 is " + vol);
```

```
    System.out.println("Weight of mybox3 is " + mybox3.weight);
    System.out.println();
    vol = myclone.volume();
    System.out.println("Volume of myclone is " + vol);
    System.out.println("Weight of myclone is " + myclone.weight);
    System.out.println();
    vol = mycube.volume();
    System.out.println("Volume of mycube is " + vol);
    System.out.println("Weight of mycube is " + mycube.weight);
    System.out.println();
} }
```

4a,b

Define package. What are the steps involved in creating user defined package and how to access them with an example.

Defining a Package

To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package.

This is the general form of the **package** statement:

package *pkg*;

Here, *pkg* is the name of the package. For example, the following statement creates a package
called **MyPackage**. package MyPackage;

A Short Package Example

Keeping the preceding discussion in mind, you can try this simple package:

```
// A simple package
package MyPack;
class Balance {
  String name;
  double bal;
  Balance(String n, double b) {
    name = n;
    bal = b;
}
  void show() {
    if(bal<0)
      System.out.print("--> ");
    System.out.println(name + ": $" + bal);
} }
class AccountBalance {
  public static void main(String args[]) {
    Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23);
  current[1] = new Balance("Will Tell", 157.02);
  current[2] = new Balance("Tom Jackson", -12.33);
  for(int i=0; i<3; i++) current[i].show();
}
}
```

Call this file **AccountBalance.java** and put it in a directory called **MyPack**.
Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Then, try executing the **AccountBalance** class, using the following command line: java MyPack.AccountBalance

	Access Protection Subclasses in the same package Non-subclasses in the same package Subclasses in different packages 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. Table 9-1 sums up the interactions.	
4b	Define the concept of multithreading in Java and explain the different phases in the life cycle of thread, with a neat sketch. In Java, Multithreading refers to a process of executing two or more threads simultaneously for maximum utilization of the CPU. A thread in Java is a ***lightweight process*** requiring fewer resources to create and share the process resources. **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. 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. **Terminated (Dead)** − A runnable thread enters the terminated state when it completes its task or otherwise terminates.	
5	Write a Java program that implements a multi-thread application that has three threads. First thread generates a random integer for every 1 second; second thread computes the square of the number andprints; third thread will print the value of cube of the number	
	import java.util.*; class second implements Runnable { public int x; public second (int x)	

```
{
this.x=x;
}
public void run()
{
System.out.println("Second thread:Square of the number is  "+x*x);
}
}
class third implements Runnable
{
public int x;
public third(int x)
{
this.x=x;
}
public void run()
{
System.out.println("third thread:Cube of the number is  "+x*x*x);
}
}
class first extends Thread
{
public void run()
{
int num=0;
Random r=new Random();
try
{
for(int i=0;i<5;i++)
{
num=r.nextInt(100);
System.out.println("first thread generated number is  "+num);

Thread t2=new Thread (new second(num));
t2.start();
Thread t3=new Thread(new third(num));

t3.start();
Thread.sleep(1000);
}
}
catch(Exception e)
{
System.out.println(e.getMessage());
}
}
}
public class multithread
{
public static void main(String args[])
{
first a=new first();
a.start();
}
```

	}	
6	explain with syntax and example isAlive( ) Two ways exist to determine whether a thread has finished. First, you can call **isAlive( )** on the thread. This method is defined by **Thread**, and its general form is shown here: final boolean isAlive( ) The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise. join( ) This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it. Additional forms of **join( )** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.	

```
// 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 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");
    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.");
  }
}
```

O/p:

```
New thread: Thread[One,5,main]
   New thread: Thread[Two,5,main]
   New thread: Thread[Three,5,main]
   Thread One is alive: true
   Thread Two is alive: true
   Thread Three is alive: true
   Waiting for threads to finish.
   One: 5
   Two: 5
   Three: 5
   One: 4
   Two: 4
   Three: 4
   One: 3
   Two: 3
   Three: 3
   One: 2
   Two: 2
   Three: 2
```

wait( ) tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**.
notify( ) wakes up a thread that called **wait( )** on the same object.
notifyAll( ) wakes up all the threads that called **wait( )** on the same object. One of the threads will be granted access.

```
 // A correct implementation of a producer and consumer.
class Q {
  int n;
  boolean valueSet = false;
  synchronized int get() {
    while(!valueSet)
try { wait();
    } catch(InterruptedException e) {
      System.out.println("InterruptedException caught");
}
```

```
    System.out.println("Got: " + n);
    valueSet = false;
    notify();
    return n;
}
 synchronized void put(int n) {
   while(valueSet)
try { wait();
     } catch(InterruptedException e) {
       System.out.println("InterruptedException caught");
}
    this.n = n;
    valueSet = true;
    System.out.println("Put: " + n);
    notify();
} }
class Producer implements Runnable {
 Q q;
Producer(Q q) {
   this.q = q;
   new Thread(this, "Producer").start();
}
 public void run() {
   int i = 0;
   while(true) {
     q.put(i++);
} }
}
class Consumer implements Runnable {
 Q q;
 Consumer(Q q) {
   this.q = q;
   new Thread(this, "Consumer").start();
}
 public void run() {
   while(true) {
q.get(); }
} }
class PCFixed {
 public static void main(String args[]) {
   Q q = new Q();
   new Producer(q);
   new Consumer(q);
   System.out.println("Press Control-C to stop.");
 }
}
Put: 1
  Got: 1
  Put: 2
  Got: 2
  Put: 3
```