1. What is the output of following code snippet?

**class** Parent {

**protected** HashSet<Integer>mySet = **new** HashSet<Integer>();

**public** Parent() {

mySet.add( 10 );

}

**public** Set<Integer>getSet() {

**return**mySet;

}

}

**public class** Child **extends** Parent {

**public** Child( Integer val ) {

mySet.add( 10 );

}

**Public** HashSet<Integer>getSet() {

**Return** mySet;

}

**Public static void** main( String[] args ) **throws** Exception {

Parent obj = **new** Child( 20 );

System.***out***.println( obj.getSet() );

}

}

1. [10,20]
2. Runtime Error
3. [10]
4. Compile time Error

Ans – C

1. Replace the Return Type with correct option.

**Public** ReturnType getFactor(**byte** x, **double** y) {

**return** (**long**)x / y \* 3;

}

1. long
2. int
3. double
4. float

Ans – double

1. Which collection class allows you to grow or shrink its size and provides indexed access to its elements, but whose methods are not synchronized?
2. java.util.HashSet
3. java.util.LinkedHashSet
4. java.util.List
5. java.util.ArrayList

Ans : ArrayLists

1. Choose all correct statements in the following.
2. Varargs enables the compiler to assemble the java.util.List for you based on the argument list you pass to a method. The three periods next to the parameter type (e.g. public void myMethod(Object … args)) denotes varargs.
3. Static imports let you avoid qualifying static members with class names. Once the static member is imported then you can use it in your code without the class name prefix.
4. Enumerated types are type safe and force users of your class to use one of the acceptable values.
5. Enhanced for loop eliminates error-proneness of iterators and does not require any index variables. Also known as a “foreach” loop.
6. Auto boxing/unboxing makes a programmer’s life easier by not having to write manual code for only conversion from wrapper types Integer, Float etc into primitive types such as int, float etc.
7. Scanner API provide a more robust mechanism for reading in data types rather than simply parsing strings from buffered System.in calls.

Ans B, C, D, E, F

1. What is the output of following code:

**Final class** MyKey {

**int**x;

**int**y;

**public**MyKey(**int**x, **int**y) {

**this**.x = x;

**this**.y = y;

}

}

**final**TreeMap<MyKey, String>myTreeMap = **new**TreeMap<MyKey, String>();

myTreeMap.put( **new** MyKey(10, 20), "10-20" );

myTreeMap.put( **new** MyKey(10, 20), "10-20" );

myTreeMap.put( **new** MyKey(1, 2), "1-2" );

System.***out***.println(myTreeMap.get( **new** MyKey(1, 2) ));

1. null
2. 1-2
3. Runtime Error
4. Compile time Error

Ans. Runtime error, MyKey class must implement compareTo() of Comparable interface as TreeMap store (K,V) pair in sorted order

1. Which is true statement in the following?
2. If only one thread is blocked in the wait method of an object, and another thread executes the notify on that same object, then the first thread immediately resumes execution.
3. If two threads are blocked in the wait method of one object, and another thread executes the notify method on the same object, then the first thread that executed the wait call first definitely resumes execution as a direct and sole consequence of the notify call.
4. If a thread is blocked in the wait method of an object, and another thread executes the notify method on the same object, it is still possible that the first thread might never resume execution.
5. If a thread is blocked in the wait method of an object, and another thread executes the notify method on the same object, then the first thread definitely resumes execution as a direct and sole consequence of the notify call.

Ans. C is correct

1. What is the output of the following code:

**Class** ClassA {

**Private int** fieldX;

**Private int** fieldY;

**private** Map<String, String>keyValues;

**public** ClassA( **int**fieldX, **int**fieldY, Map<String, String>keyValues ) {

**super**();

**this**.fieldX = fieldX;

**this**.fieldY = fieldY;

**this**.keyValues = keyValues;

}

@Override

**public** String toString() {

**return**"ClassA [fieldX=" + fieldX + ", fieldY=" + fieldY + ", keyValues=" + keyValues + "]";

}

}

**Class** ClassB **implements** Cloneable {

**Private** ClassA classA;

**Public** ClassB( ClassA classA ) {

**super**();

**this**.classA = classA;

}

@Override

**public** String toString() {

**return**"ClassB [classA=" + classA + "]";

}

@Override

**Protected** ClassB clone() **throws** CloneNotSupportedException {

**return** (ClassB) **super**.clone();

}

}

**class** Test {

**public static void** main( String[] args ) **throws** Exception {

Map<String, String> keyValues = **new** HashMap<String, String>();

keyValues.put( "X", "20" );

keyValues.put( "Y", "30" );

ClassB classB = **new** ClassB( **new**ClassA( 20, 30, keyValues ) );

ClassB cloneClassB = classB.clone();

System.***out***.println( cloneClassB );

}

}

1. ClassB [classA=ClassA [fieldX=20, fieldY=30, keyValues={Y=30, X=20}]]
2. ClassB [classA=ClassA [fieldX=0, fieldY=0, keyValues={}]]
3. ClassB [classA=ClassA [fieldX=20, fieldY=30, keyValues={}]]
4. ClassB [classA=null]]
5. System will throw CloneNotSupportedException

Ans A

1. Which are the following statements are correct to prevent ConcurrentModificationException when using an iterator in the below code snippet?

List<String> myList = **new** ArrayList<String>( 10 );

myList.add( "Pune" );

myList.add( "Vizag" );

myList.add( "Hyderabad" );

**for** ( Iterator it = myList.iterator(); it.hasNext(); ) {

String myObject = (String) it.next();

System.***out***.println( myObject );

**if** ( "Hyderabad".equals( myObject ) ) {

myList.remove( myObject );

//can throw ConcurrentModificationException in single as

//well as multi-thread access situations.

}

}

1. You can convert your list to an array with myList.toArray() and iterate on the array.
2. You can use the ConcurrentHashMap and CopyOnWriteArrayList classes, which provide much better scalability and the iterator returned by ConcurrentHashMap.iterator() will not throw ConcurrentModificationException while preserving thread-safety.
3. You can use it.remove(); method instead of myList.remove(myObject);
4. All of above

Ans D

1. What is the output of following code snippet?

**public class** MyTest {

**public static void** main( String[] args ) **throws** Exception {

**final** List<String> myList1 = **new** ArrayList<String>();

**final** List<String> myList2 = **new** ArrayList<String>();

Thread threadA = **new** Thread("A")

{

**publicvoid** run()

{

myList1.add( "Thread A - 1" );

**synchronized**(myList2) {

myList2.add( "Thread A - 2" );

}

}

};

Thread threadB = **new** Thread("B")

{

**publicvoid** run()

{

myList2.add( "Thread B - 1" );

**synchronized**(myList1)

{

myList1.add( "Thread B - 2" );

}

}

};

threadA.start();

threadB.start();

threadA.join();

threadB.join();

myList1.addAll( myList2 );

System.***out***.println(myList1);

}

}

1. [Thread A - 1, Thread A - 2, Thread B - 2, Thread B - 1]
2. [Thread A - 1, Thread B - 2, Thread A - 2, Thread B - 1]
3. [Thread A - 1, Thread A - 2, Thread B - 1, Thread B - 2]
4. [Thread B - 2, Thread A - 1, Thread B - 1, Thread A - 2]
5. Dead Lock Error

Ans: A,B,C,D

1. Choose all correct statements in the following.
2. A soft reference will only get removed if memory is low by Garbage Collector. So it is useful for implementing caches while avoiding memory leaks.
3. Each time an object is created in Java it goes into the area of memory known as stack. The primitive variables like int and double are allocated in the heap.
4. Security hole is when you access private data members of the outer class, the JDK compiler creates package-access member functions in the outer class for the inner class to access the private members.
5. Thread class can created by implementing java.lang.Thread class and extending java.lang.Runnable.
6. A monitor is basically a guardian who watches over a sequence of synchronized code and making sure only one thread at a time executes a synchronized piece of code. Each monitor is associated with an object reference.

Ans: C, E