```
public PrintStream printf(Locale l,
                          String format,
                          Object... args)
```

Another version does not require locale.

```
System.out.printf("%2$d + %1$d", 123, 456);
```

o/p is: 456 + 123

Format string syntax is as follows:
`%[arg_index$][flags][width][.precision]conversion char`

arg_index An integer followed directly by a `$`, this indicates which argument should be printed in this position.

flags While many flags are available, for the exam you'll need to know:

- "-" Left justify this argument
- "+" Include a sign (+ or -) with this argument
- "0" Pad this argument with zeroes
- "," Use locale-specific grouping separators (i.e., the comma in 123,456)
- "(" Enclose negative numbers in parentheses

width This value indicates the minimum number of characters to print. (If you want nice even columns, you'll use this value extensively.)

precision For the exam you'll only need this when formatting a floating-point number, and in the case of floating point numbers, precision indicates the number of digits to print after the decimal point.

conversion The type of argument you'll be formatting.

- `b` boolean
- `c` char
- `d` integer
- `f` floating point
- `s` string

```
import static java.lang.System.*;

public class FormattingTest
{
   public static void main(String[] args)
   {
      int i1 = -123;
      int i2 = 12345;
      float f1 = 123.4567f;

      out.printf(">%1$(7d< \n", i1);
      out.printf(">%0,7d< \n", i2);
      out.format(">%+-7d< \n", i2);
      out.printf(">%2$b + %1$5d< \n", i1, false);
      out.printf(">%1$-11.3f< \n",f1);
   }
}
```

O/P is:

```
>  (123)<
>012,345<
>+12345 <
>false +  -123<
>123.457    <
```

- The **`format()` and `printf()` methods have identical functionality.**
- Formatting data with `printf()` (or `format()`) is accomplished using *formatting strings* that are associated with primitive or string arguments.
- The `format()` method allows you to mix literals in with your format strings.
- The format string values you should know are
- Flags: `-`, `+`, `0`, "," , and `(`
- Conversions: `b`, `c`, `d`, `f`, and `s`
- If your conversion character doesn't match your argument type, an exception will be thrown.

```
System.out.format("%d", 12.3);

Exception in thread "main" java.util.IllegalFormatConversionException:
d != java.lang.Double
```

- Not all combinations of the format flags are correct and an **IllegalFormatFlagsException** is thrown in such a case.
  For eg: using left-justify together with 0 padding is illegal ie "-0" flags used together.

IMP:
Which will compile and run without exception? (Choose all that apply.)
A. `System.out.format("%b", 123);`
B. `System.out.format("%c", "x");`
C. `System.out.printf("%d", 123);`
D. `System.out.printf("%f", 123);`
E. `System.out.printf("%d", 123.45);`
F. `System.out.printf("%f", 123.45);`
G. `System.out.format("%s", new Long("123"));`
Answer:
✹ **A , C, F,** and **G** are correct. The `%b` (boolean) conversion character returns `true` for any non-`null` or non-boolean argument.
✹ **B** is incorrect, the `%c` (character) conversion character expects a character, not a String. **D** is incorrect, the `%f` (floating-point) conversion character won't automatically promote an integer type. **E** is incorrect, the `%d` (integral) conversion character won't take a floating point number. (Note: The `format()` and `printf()` methods behave identically.) (Objective 3.5)

Generics & Collections

Overriding hashCode() and equals() (Objective 6.2)

- **`equals()`, `hashCode()`, and `toString()` are `public`.**

TABLE 7-1 Methods of Class Object Covered on the Exam

Method	Description
`boolean equals (Object obj)`	Decides whether two objects are meaningfully equivalent.
`void finalize()`	Called by garbage collector when the garbage collector sees that the object cannot be referenced.
`int hashCode()`	Returns a hashcode `int` value for an object, so that the object can be used in Collection classes that use hashing, including Hashtable, HashMap, and HashSet.
`final void notify()`	Wakes up a thread that is waiting for this object's lock.
`final void notifyAll()`	Wakes up *all* threads that are waiting for this object's lock.
`final void wait()`	Causes the current thread to wait until another thread calls `notify()` or `notifyAll()` on this subject.
`String toString()`	Returns a "text representation" of the object.

- Override `toString()` so that `System.out.println()` or other methods can see something useful, like your object's state.
- **Use == to determine if two reference variables refer to the same object.**
- **Use `equals()` to determine if two objects are meaningfully equivalent.**
- If you don't override `equals()`, your objects won't be useful hashing keys.
- If you don't override `equals()`, different objects can't be considered equal.

There's a potential limitation lurking here: if you don't override a class's **`equals()`** method, you won't be able to use those objects as a key in a hashtable and you probably won't get accurate Sets, such that there are no conceptual duplicates.

The `equals()` method in class Object uses only the == operator for comparisons, **so unless you override `equals()`, two objects are considered equal only if the two references refer to the same object**.

```
public class EqualsTest {
       public static void main (String [] args) {
              Moof one = new Moof(8);
              Moof two = new Moof(8);
              if (one.equals(two)) {
                     System.out.println("one and two are equal");
              }
       }
}
class Moof {
       private int moofValue;
       Moof(int val) {
              moofValue = val;
       }
       public int getMoofValue() {
              return moofValue;
       }
       public boolean equals(Object o) {
              if ((o instanceof Moof) && (((Moof)o).getMoofValue()
                                          == this.moofValue)) {
                     return true;
              } else {
                     return false;
              }
       }

       public int hashCode() { return (moofValue * 17); }
}
```

Remember that the `equals()`**,** `hashCode()`**, and** `toString()` ***methods are all*** `public`***. The following would not be a valid override of the*** `equals()` ***method, although it might appear to be if you don't look closely enough during the exam:***

```
class Foo { boolean equals(Object o) { } } // compiler error!
```

- **Strings and wrappers override `equals()` and make good hashing keys.**
- **When overriding `equals()`, use the `instanceof` operator to be sure you're evaluating an appropriate class.**
- When overriding `equals()`, compare the objects' significant attributes.
- Highlights of the `equals()` contract:
  - Reflexive: `x.equals(x)` is `true`.
  - Symmetric: If `x.equals(y)` is `true`, then `y.equals(x)` must be `true`.
  - Transitive: If `x.equals(y)` is `true`, and `y.equals(z)` is `true`, then `z.equals(x)` is `true`.
  - Consistent (or **idempotent**): Multiple calls to `x.equals(y)` will return the same result.
  - **Null: If x is not `null`, then `x.equals(null)` is false.**
- **If `x.equals(y)` is true, then `x.hashCode() == y.hashCode()` is true.**
- **If you override `equals()`, override `hashCode()`.**
- HashMap, HashSet, Hashtable, LinkedHashMap, & LinkedHashSet use hashing.
- **An appropriate `hashCode()` override sticks to the `hashCode()` contract.**
- **An efficient `hashCode()` override distributes keys evenly across its buckets.**
- An overridden `equals()` must be at least as precise as its `hashCode()` mate.
- **To reiterate: if two objects are equal, their hashcodes must be equal so that in a container which uses hashing, the two equal object instances will always result in being given the same hashcode and hence put in the same bucket where they can be compared for equality. If the hashcodes are different then the equal will never be true as it will not be able to reach the first object instance in the container (which is in a different bucket) and hence the comparison will fail.**
- **It's legal for a `hashCode()` method to return the same value for all instances (although in practice it's very inefficient).**
  In real-life hashing, it's not uncommon to have more than one entry in a bucket. Hashing retrieval is a two-step process.
  1. ***Find the right bucket (using*** `hashCode()`***)***
  2. ***Search the bucket for the right element (using*** `equals()` ***).***

- Highlights of the `hashCode()` contract:
  - Consistent: multiple calls to `x.hashCode()` return the same integer.
  - If `x.equals(y)` is `true`, `x.hashCode() == y.hashCode()` is `true`.
  - If `x.equals(y)` is `false`, then `x.hashCode() == y.hashCode()` can be either `true` or `false`, but `false` will tend to create better efficiency.
  - **`transient` variables aren't appropriate for `equals()` and `hashCode()`.**

```
class SaveMe implements Serializable{
       transient int x;
       int y;
       SaveMe(int xVal, int yVal) {
              x = xVal;
              y = yVal;
       }
       public int hashCode() {
              return (x ^ y); // Legal, but not correct to
                              // use a transient variable
       }
       public boolean equals(Object o) {
```

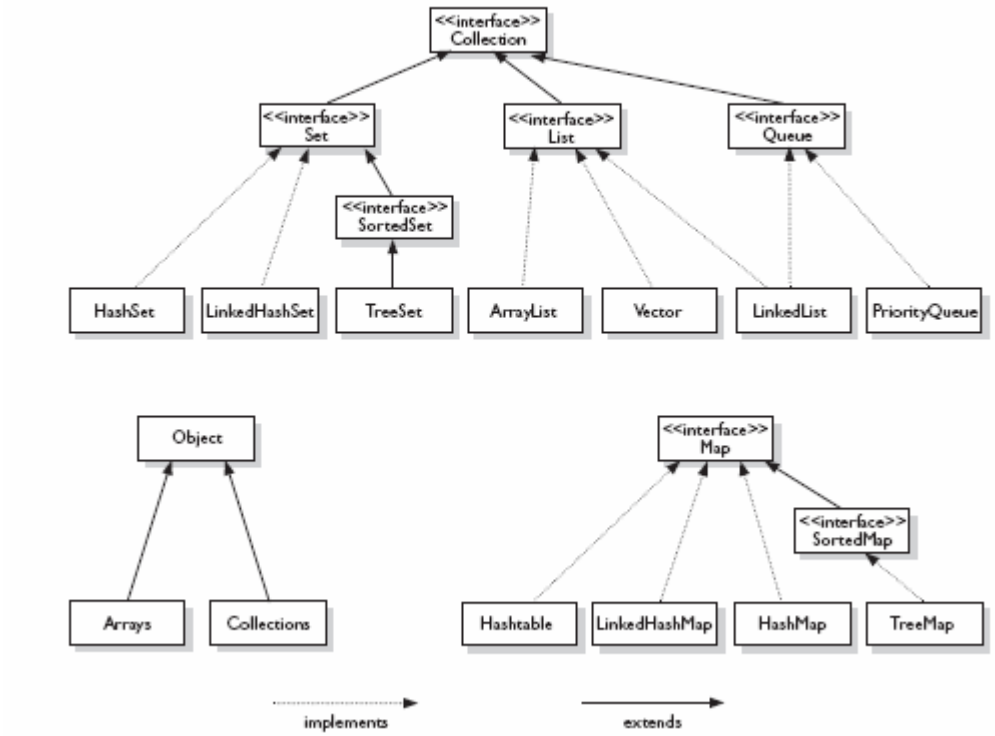
```
                    if (o instance of SaveMe) {
                        SaveMe test = (SaveMe)o;
                        // Legal, not correct to use transient
                        if (test.y == y && test.x == x) {
                            return true;
                        } else {
                            return false;
                        }
                    }
                }
            }
```

The problem occurs when this object is serialized and then deserialized then the transient variable will be reset to the default value and the hashcode will be different for the deserialized object instance. Again, post deserialization, the equals() method will return false too if it depends on the comparison of values of transient instance members.

Collections (Objective 6.1)

- Common collection activities include adding objects, removing objects, verifying object inclusion, retrieving objects, and iterating.
- Three meanings for "collection":
  - **collection** Represents the data structure in which objects are stored
  - **Collection** java.util interface from which Set and List extend
  - **Collections** A class that holds static collection utility methods
- Four basic flavors of collections include Lists, Sets, Maps, Queues:
  - **Lists of things** Ordered, duplicates allowed, with an index.
  - **Sets of things** May or may not be ordered and/or sorted; duplicates not allowed.
  - **Maps of things with keys** May or may not be ordered and/or sorted; duplicate keys are not allowed.
  - **Queues of things to process** Ordered by FIFO or by priority.

FIGURE 7-2 The interface and class hierarchy for collections



- Four basic sub-flavors of collections **Sorted, Unsorted, Ordered, Unordered**.
- **Ordered** Iterating through a collection in a specific, non-random order.

- **Sorted** Iterating through a collection in a sorted order.
- Sorting can be alphabetic, numeric, or programmer-defined (by overriding **compareTo**() in **Comparable** interface or implementing **Comparator** interface and overriding **compare**() method).

Key Attributes of Common Collection Classes (Objective 6.1)

- **ArrayList**: Fast iteration and fast random access.
- **Vector**: It's like a slower ArrayList, but it has synchronized methods.
- **LinkedList**: Good for adding elements to the ends, i.e., stacks and queues.
- **HashSet**: Fast access, assures no duplicates, provides no ordering.
- **LinkedHashSet**: No duplicates; iterates by insertion order.
- **TreeSet**: No duplicates; iterates in sorted order.
- **HashMap**: Fastest updates (key/value pairs); allows one `null` key, many `null` values.
- **Hashtable**: Like a slower HashMap (as with Vector, due to its synchronized methods). No `null` values or `null` keys allowed.
- **LinkedHashMap**: Faster iterations; iterates by insertion order or last accessed; allows one `null` key, many `null` values.
- **TreeMap**: A sorted map.
- **PriorityQueue**: A to-do list ordered by the elements' priority.

Using Collection Classes

- **Collections hold only Objects, but primitives can be autoboxed.**
- Iterate with the enhanced `for`, or with an Iterator via `hasNext()` & `next()`.
- `hasNext()` determines if more elements exist; the Iterator does NOT move.
- `next()` returns the next element AND moves the Iterator forward.
- **To work correctly, a Map's keys must override `equals()` and `hashCode()`.**
- Queues use `offer()` to add an element, `poll()` to remove the head of the queue, and `peek()` to look at the head of a queue.

Sorting and Searching Arrays and Lists

- Sorting can be in natural order, or via a Comparable or many Comparators.
- **Implement Comparable using `compareTo()`; provides only one sort order.**
- **Create many Comparators to sort a class many ways; implement `compare()`.**
- **To be sorted and searched, a List's elements must be *comparable*.**
- To be searched, an array or List must first be sorted, as binarySearch requires the list/array to be sorted first.

Utility Classes: Collections and Arrays

- Both of these java.util classes provide
  - A `sort()` method. Sort using a Comparator or sort using natural order.
  - A `binarySearch()` method. Search a pre-sorted array or List.
- `Arrays.asList()` creates a List from an array and links them together.
- `Collections.reverse()` reverses the order of elements in a List.
- `Collections.reverseOrder()` returns a Comparator that sorts in reverse.
- Lists and Sets have a `toArray()` method to create arrays.

```
import java.util.*;
import static java.lang.System.*;
import java.io.*;

class Animal
{
        String name;

        Animal(String name) {this.name = name;}
```

```
        String getName() {return name;}
}

class Dog extends Animal { Dog(String name) {super(name);} }
class Cat extends Animal { Cat(String name) {super(name);} }
class Horse extends Animal { Horse(String name) {super(name);} }


public class CollectionTest
{
        enum Pets
        {
                CAT(1), DOG(2), HORSE(3);
                Pets(int index) { this.index = index;}
                private int index;
                public int getIndex() {return this.index;}
        }

        static class PQsort implements Comparator<Integer>
        { // inverse sort
                public int compare(Integer one, Integer two) {
                        return two - one; // unboxing
                }
        }

        public static void main(String[] args)
        {
                arrayListTest();
                arrayTest();
                setTest();
                mapTest();
                queueTest();
        }

        static void queueTest()
        {
                int[] ia = {1,5,3,7,6,9,8 }; // unordered data

                PriorityQueue<Integer> pq1 = new PriorityQueue<Integer>(); // use natural order
                for(int x : ia) // load queue
                        pq1.offer(x);

                for(int x : ia) // review queue
                        out.print(pq1.poll() + " ");
                out.println();

                PQsort pqs = new PQsort(); // get a Comparator
                PriorityQueue<Integer> pq2 = new PriorityQueue<Integer>(10,pqs); // use
Comparator

                for(int x : ia) // load queue
                        pq2.offer(x);
                out.println("size " + pq2.size());
```

```
		out.println("peek " + pq2.peek());
		out.println("size " + pq2.size()); // size does not reduce after peek()
		out.println("poll " + pq2.poll());
		out.println("size " + pq2.size()); // size reduces after poll

		for(int x : ia) // review queue
			out.print(pq2.poll() + " ");
	}

	static void mapTest()
	{
		Map<Pets, Animal> hmap = new HashMap<Pets, Animal>(); // unordered map
		// ordered map by insertion order
		Map<Pets, Animal> lmap = new LinkedHashMap<Pets, Animal>();
		Map<Pets, Animal> tmap = new TreeMap<Pets, Animal>(); // sorted map

		Dog d = new Dog("dog1");
		Cat c = new Cat("cat1");
		Horse h = new Horse("horse1");

		hmap.put(Pets.DOG, d);
		hmap.put(Pets.CAT, c);
		hmap.put(Pets.HORSE, h);

		lmap.put(Pets.DOG, d);
		lmap.put(Pets.CAT, c);
		lmap.put(Pets.HORSE, h);

		tmap.put(Pets.DOG, d);
		tmap.put(Pets.CAT, c);
		tmap.put(Pets.HORSE, h);

		out.print("HashMap: (fastest, unordered)"); printMap(hmap);
		out.print("LinkedHashMap: (insertion order)");printMap(lmap);
		out.print("TreeMap: (sorted by key)");printMap(tmap);

	}

	static void printMap(Map<Pets, Animal> map)
	{
		Iterator<Map.Entry<Pets, Animal>> it = map.entrySet().iterator();
		while(it.hasNext()) {
			Map.Entry<Pets, Animal> pairs = it.next();
			out.print("[" + pairs.getKey().getIndex() + " " +
							pairs.getValue().getName() + "]" );
		}
		out.println();
	}

	static void setTest()
	{
		// sets dont allow duplicates
		Set<Integer> hset = new HashSet<Integer>(); // faster, unordered sets
```

```
		Set<Integer> lset = new LinkedHashSet<Integer>(); // ordered by insertion order
		Set<Integer> tset = new TreeSet<Integer>(); // sorted set

		for (int i = 5; i > 0; i--) {
			hset.add(i);
			lset.add(i);
			tset.add(i);
		}

		hset.add(1); //-- adding duplicate will not result in a
		// duplicate element appended to a Set. It will simply be
		// ignored. So size() will return same before and after the
		// above add(1).

		out.println("HashSet (unordered): " + hset);
		out.println("LinkedHashSet (ordered by insertion order): " + lset);
		out.println("TreeSet: (sorted) " + tset);


	}

	static void arrayTest()
	{
		int[] arr = {34,211,32,5,23};
		Arrays.sort(arr);
		for (int i : arr) {
			out.print(i + " ");
		}
		DVDInfo[] dvdArray = new DVDInfo[7];

		populateList(dvdArray);
		Arrays.sort(dvdArray); // uses comparable's compareTo()
		for (DVDInfo d : dvdArray) {
			out.println (d);
		}
		ActorSort as = new ActorSort();
		// uses actor sort comparator's compare()
		Arrays.sort(dvdArray, as);
		// Get a list out of array.
		List<DVDInfo> aList = Arrays.asList(dvdArray);
		out.println(aList);
		DVDInfo di = new DVDInfo("test", "test", "test");
		//aList.add(di); -- fails as we cannot append to a list
		// which is mapped to an array of fixed size.
		aList.set(1,di); // use set instead! works!
		out.println(aList);
		// passing dvdArray ensures that returned array is of
		// type DVDInfo[].
		// if we dont pass any argument to toArray()  then
		// Object[] is returned.
		DVDInfo[] dArr = new DVDInfo[dvdArray.length];
		dArr = aList.toArray(dArr);
		for (DVDInfo d : dArr) {
```

```
                out.println (d);
        }
}

static void arrayListTest()
{
        List<String> strList = new ArrayList<String>();
        List<Integer> intList = new ArrayList<Integer>();

        String s = "there!";
        strList.add(s);
        strList.add("hi");

        // Sorting list of strings
        out.println("unsorted: " + strList);
        // this works coz String implements Comparable i/f
        // providing implementation of the compareTo()
        // which is used to sort alphabetically.
        Collections.sort(strList);
        out.println("sorted: " + strList);

        out.println(strList.size());
        for(String a: strList)
                out.println(a);
        strList.remove("hi");
        out.println(strList.size());

        intList.add(20); // uses autoboxing
        for(int i : intList)
                out.println(i);

        // demo use of Comparable which allows only sorting by title.
        ArrayList<DVDInfo> dvdList = new ArrayList<DVDInfo>();
        populateList(dvdList); // adds the file data to the ArrayList
        // default is to use compareTo() for sorting
        Collections.sort(dvdList);
        out.println(dvdList);

        // demo use of Comparator which allows sorting
        // by title, genre and leadActor
        GenreSort gs = new GenreSort();
        // passing comparator uses compare() method in the comparator
        Collections.sort(dvdList, gs);
        out.println(dvdList);

        // demo iterating a list using iterator.
        // alternatively we can iterate a list using
        // enhanced for loop.
        Iterator<DVDInfo> iter = dvdList.iterator();
        while (iter.hasNext()) {
                out.println(iter.next().getTitle());
        }
```

```
	}

	static void populateList(List<DVDInfo> list)
	{
		try {
			File f = new File("./dvdinfo.txt");
			FileReader fr = new FileReader(f);
			BufferedReader br = new BufferedReader(fr);
			String s = null;

			while((s = br.readLine()) != null) {
				String[] sarr = s.split("/");

				DVDInfo dvdInfo = new DVDInfo(sarr[0], sarr[1], sarr[2]);
				list.add(dvdInfo);
			}
			fr.close();
			br.close();
		} catch (FileNotFoundException ex) {
			ex.printStackTrace();
		} catch (IOException ioe) {
			ioe.printStackTrace();
		}
	}

	static void populateList(DVDInfo[] arr)
	{
		try {
			File f = new File("./dvdinfo.txt");
			FileReader fr = new FileReader(f);
			BufferedReader br = new BufferedReader(fr);
			String s = null;

			int i = 0;
			while((s = br.readLine()) != null) {
				String[] sarr = s.split("/");

				DVDInfo dvdInfo = new DVDInfo(sarr[0], sarr[1], sarr[2]);
				arr[i] = dvdInfo;
				i++;
			}
			fr.close();
			br.close();
		} catch (FileNotFoundException ex) {
			ex.printStackTrace();
		} catch (IOException ioe) {
			ioe.printStackTrace();
		}
	}
}

class DVDInfo implements Comparable<DVDInfo> // or just Comparable
{
```

```
	String title;
	String genre;
	String leadActor;

	DVDInfo(String t, String g, String a) {
		title = t; genre = g; leadActor = a;
	}

	public String toString() {
		return title + " " + genre + " " + leadActor + "\n";
	}

	// getters and setter go here
	public String getTitle() { return this.title; }
	public String getGenre() { return this.genre; }
	public String getLeadActor() { return this.leadActor; }

	// Can implement only one of sorting by title/genre/leadActor.
	public int compareTo(DVDInfo d) // or compareTo(Object o)
	{
		return title.compareTo(d.getTitle());
	}
}

// Implements sorting by genre
class GenreSort implements Comparator<DVDInfo>
{
	public int compare(DVDInfo one, DVDInfo two)
	{
		return one.getGenre().compareTo(two.getGenre());
	}
}

// Implements sorting by leadActor
class ActorSort implements Comparator<DVDInfo>
{
	public int compare(DVDInfo one, DVDInfo two)
	{
		return one.getLeadActor().compareTo(two.getLeadActor());
	}
}
```

TABLE 7-4 Key Methods in Arrays and Collections

Key Methods in java.util.Arrays	Descriptions
`static List asList(T[])`	Convert an array to a List, (and bind them).
`static int binarySearch(Object[], key)` `static int binarySearch(primitive[], key)`	Search a sorted array for a given value, return an index or insertion point.
`static int binarySearch(T[], key, Comparator)`	Search a Comparator-sorted array for a value.
`static boolean equals(Object[], Object[])` `static boolean equals(primitive[], primitive[])`	Compare two arrays to determine if their contents are equal.
`public static void sort(Object[ ] )` `public static void sort(primitive[ ] )`	Sort the elements of an array by natural order.
`public static void sort(T[], Comparator)`	Sort the elements of an array using a Comparator.
`public static String toString(Object[])` `public static String toString(primitive[])`	Create a String containing the contents of an array.
Key Methods in java.util.Collections	**Descriptions**
`static int binarySearch(List, key)` `static int binarySearch(List, key, Comparator)`	Search a "sorted" List for a given value, return an index or insertion point.
`static void reverse(List)`	Reverse the order of elements in a List.
`static Comparator reverseOrder()` `static Comparator reverseOrder(Comparator)`	Return a Comparator that sorts the reverse of the collection's current sort sequence.
`static void sort(List)` `static void sort(List, Comparator)`	Sort a List either by natural order or by a Comparator.

IMP: Given a properly prepared String array containing five elements, which range of results could a proper invocation of `Arrays.binarySearch()` produce?

A. 0 through 4
B. 0 through 5
C. -1 through 4
D. -1 through 5
E. -5 through 4
F. -5 through 5
G. -6 through 4
H. -6 through 5

Answer:

- **G** is correct. **If a match is found, `binarySearch()` will return the index of the element that was matched. If no match is found, `binarySearch()` will return a negative number that, if inverted and then decremented, gives you the insertion point (array index) at which the value searched on should be inserted into the array to maintain a proper sort.**
- **A, B, C, D, E, F,** and **H** are incorrect based on the above. (Objective 6.5)

TABLE 7-5 Key Methods in List, Set, and Map

Key Interface Methods	List	Set	Map	Descriptions
`boolean add(element)` `boolean add(index, element)`	X X	X		Add an element. For Lists, optionally add the element at an index point.

`boolean contains(object)`
`boolean containsKey(object key)`
`boolean containsValue(object value)`
	X	X	
			X
			X
				Search a collection for an object (or, optionally for Maps a key), return the result as a `boolean`.

`object get(index)` `object get(key)`	X		 X	Get an object from a collection, via an index or a key.
`int indexOf(object)`	X			Get the location of an object in a List.
`Iterator iterator()`	X	X		Get an Iterator for a List or a Set.
`Set keySet()`			X	Return a Set containing a Map's keys.
`put(key, value)`			X	Add a key/value pair to a Map.

`remove(index)`
`remove(object)`
`remove(key)`
	X
	X
		
		X
			
			X
				Remove an element via an index, or via the element's value, or via a key.

`int size()`	X	X	X	Return the number of elements in a collection.
`Object[] toArray()` `T[] toArray(T[])`	X	X		Return an array containing the elements of the collection.

Generics

- Generics let you enforce compile-time type safety on Collections (or other classes and methods declared using generic type parameters).
- **An ArrayList<Animal> can accept references of type Dog, Cat, or any other subtype of Animal** (subclass, or if Animal is an interface, implementation).
- When using generic collections, **a cast is not needed to get (declared type) elements out of the collection**. With non-generic collections, a cast is required:

```
List<String> gList = new ArrayList<String>();
List list = new ArrayList();
// more code
String s = gList.get(0); // no cast needed
String s = (String)list.get(0); // cast required
```

- You can pass a generic collection into a method that takes a non-generic collection, but the results may be disastrous. The compiler can't stop the method from inserting the wrong type into the previously type safe collection.

```
import java.util.*;
import static java.lang.System.*;

public class GenericsTest
{
    public static void main(String[] args)
    {
        List<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(3);
        insert(list);
        out.println(list);
    }

    static void insert(List list)
```

```
    {
        list.add(new Integer(42));
        list.add("Test"); // Yes it works! Unfortunately!
    }
}
```

O/P: [1, 3, 42, Test]
Compiler does issue a warning though:

Note: E:\Test\SCJP5\GenericsTest.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

- **If the compiler can recognize that non-type-safe code is potentially endangering something you originally declared as type-safe, you will get a compiler warning.** For instance, if you pass a List<String> into a method declared as
  `void foo(List aList) { aList.add(anInteger); }`
  the compiler will issue a warning because the `add()` method is potentially an "unsafe operation."
- Remember that "compiles without error" is not the same as "compiles without warnings." On the exam, a compilation *warning* is not considered a compilation *error* or *failure*.
- **Generic type information does not exist at runtime—it is for compile-time safety only.** Mixing generics with legacy code can create compiled code that may throw an exception at runtime.
- **Polymorphic assignments applies only to the base type, not the generic type parameter.**
  You can say `List<Animal> aList = new ArrayList<Animal>(); // yes`
  You can't say `List<Animal> aList = new ArrayList<Dog>(); // no`
  Compiler error! Incompatible types.
- The **polymorphic assignment rule applies everywhere an assignment can be made.** The following are NOT allowed:
  ```
  void foo(List<Animal> aList) { } // cannot take a List<Dog>
  List<Animal> bar() { } // cannot return a List<Dog>
  ```
- Wildcard syntax allows a generic method, accept subtypes (or supertypes) of the declared type of the method argument:
  ```
  void addD(List<Dog> d) {} // can take only <Dog>
  void addD(List<? extends Dog>) {} // take a <Dog> or <Beagle>
  ```
- **The wildcard keyword `extends` is used to mean either "extends" or "implements." So in `<? extends Dog>`, `Dog` can be a class or an interface.**
- **When using a wildcard, `List<? extends Dog>`, the collection can be accessed but not modified.**
- **When using a wildcard, `List<?>`, any generic type can be assigned to the reference, but for access only, no modifications.**
- `List<Object>` refers only to a `List<Object>`, while `List<?>` or `List<? extends Object>` can hold any type of object, but for access only.
- Declaration conventions for generics use **T for type and E for element**:
  ```
  public interface List<E> // API declaration for List
  boolean add(E o) // List.add() declaration
  ```
- The generics type identifier can be used in class, method, and variable declarations:
  ```
  class Foo<t> { } // a class
  T anInstance; // an instance variable
  Foo(T aRef) {} // a constructor argument
  void bar(T aRef) {} // a method argument
  T baz() {} // a return type
  ```
  The compiler will substitute the actual type.

- You can use more than one parameterized type in a declaration:
  `public class UseTwo<T, X> { }`
- You can declare a generic method using a type not defined in the class:
  `public <T> void makeList(T t) { }`
  is NOT using T as the return type. This method has a `void` return type, but to use T within the method's argument you must declare the <T>, which happens before the return type.

```
import java.util.*;

class Animal {}
class Dog extends Animal {}
class BullDog extends Dog {}

public class Generics
{
   public static void main(String[] args)
   {
      List<Animal> list1 = new ArrayList<Animal>();
      // Cannot assign a Dog type list to an Animal type list!
      // List<Animal> list2 = new ArrayList<Dog>(); Compiler Error!
      addAnimal1(list1);

      List<Dog> dogList = new ArrayList<Dog>();

      // Cannot pass a Dog type list to an Animal type list!
      //addAnimal1(dogList); // Compiler Error!

      // Same is true for a list of Object type.
      // It can only accept a List of Object type and not
      // a list of Dog type.
      //addAnimal4(dogList); // Compiler Error!

      // Its Okay to pass any type to the List<?>
      addAnimal5(dogList);

      // Can pass a Dog type list to a method which takes
      // list of type Animal or its subtypes (ie uses ?  extends wildcard)
      // but then we cannot modify the passed in list in the method.
      addAnimal2(dogList); // OK

      // Can pass a Dog type list to a method which takes
      // list of type Dog or its super types (ie uses ? super wildcard)
      // and we can modify the list too.
      // But we cannot pass a list of type which is a subtype of Dog!
      addAnimal3(dogList);
      List<BullDog> bullDogList = new ArrayList<BullDog>();
      //addAnimal3(bullDogList); // Compiler Error!
   }

   public static void addAnimal1(List<Animal> animals)
   {
      animals.add(new Dog()); // OK to add a Dog to list of Animal.
   }

   public static void addAnimal2(List<? extends Animal> animals)
   {
      //animals.add(new Dog()); // Compiler error! cannot change the list now!
   }

   public static void addAnimal3(List<? super Dog> animals)
   {
      animals.add(new Dog()); //OK! Now we can modify the list.
   }
```

```
   public static void addAnimal4(List<Object> animals)
   {
      animals.add(new Dog()); // OK! to add Dog which IS-A Object.
   }

   public static void addAnimal5(List<?> animals)
   {
      // Wildcard without super again disallows additions!
      // Behaves like : List<? extends Object>
      //animals.add(new Dog()); // Compiler Error! cannot change list now!
   }

   // Generic Method!
   // We need to declare the type before the method prototype.
   public <T> void method1(List<T> list)
   {

   }
}
```

Figure out which will compile:

```
1) List<?> list = new ArrayList<Dog>();
2) List<? extends Animal> aList = new ArrayList<Dog>();
3) List<?> foo = new ArrayList<? extends Animal>();
4) List<? extends Dog> cList = new ArrayList<Integer>();
5) List<? super Dog> bList = new ArrayList<Animal>();
6) List<? super Animal> dList = new ArrayList<Dog>();
```

The correct answers (the statements that compile) are 1, 2, and 5.

Analysis:
3: You cannot use wildcard in the object creation part.
4: Integer is not a Dog.
6: Dog is not a super type for Animal, it's a subtype. So the assignment is wrong.

What is wrong with the statement below:

```
private Map<String, int> accountTotals = new HashMap<String, int>();
```

You cannot use a primitive type as a type parameter.

Use of equals() and hashCode() when using a HashXXX collection type.

```
import java.util.*;

public class Group extends HashSet<Person>
{
       public static void main(String[] args)
       {
              Group g = new Group();
              g.add(new Person("Hans"));
              g.add(new Person("Lotte"));
              g.add(new Person("Jane"));
              g.add(new Person("Hans"));
              g.add(new Person("Jane"));
              System.out.println("Total: " + g.size());
       }

       /* If we have add(Object o) here then
        * its neither an override nor an overload
        * and compiler complains. Had it been
```

```
         * just extending HashSet then the method
         * will be add(Object o) and it will be
         * an override.
         */
        public boolean add(Person o)
        {
                System.out.println("Adding: " + o);
                return super.add(o);
        }
}

class Person
{
        private final String name;
        public Person(String name) { this.name = name; }
        public String toString() { return name; }

        /* We need to override equals and hashCode together
         * such that if equals() returns true then
         * hashCode() should return the same value for the
         * two object instances.
         */
        public boolean equals(Object o)
        {
                if (o instanceof Person) {
                        Person p = (Person)o;
                        return this.toString().equals(p.toString());
                }
                return false;
        }

        /* If we dont override hashCode then the HashSet will
         * store duplicate persons too coz all person objects
         * will be placed in new buckets.
         */
        public int hashCode()
        {
                return this.toString().length();
        }
}
```

Searching Lists:

```
import java.util.*;

class ListSearch
{
        public static void main(String[] args) {
                String[] sa = {"foo", "bar", "baz" };
                List<String> list = Arrays.asList(sa);
                Collections.sort(list);
                System.out.println(list);
                System.out.println("foo is at index: " +
                                Collections.binarySearch(list,"foo"));
        }
}
```

O/P:

```
[bar, baz, foo]
foo is at index: 2
```

```
import java.util.*;
```

```
public class LongWordFinder {
	public static void main(String[] args)
	{
		String[] array = { "123", "12345678", "1", "12", "1234567890"};
		List<String> list = Arrays.asList(array);
		Collection<String> resultList = getLongWords(list);
		System.out.println(resultList);

		List<Integer> input = new ArrayList<Integer>();
		input.add(5);
		input.add(20);
		List<Integer> output = null;
		output = process(input);
		System.out.println(output);
	}

	/* Note: We define E IS-A CharSequence and then use E for
	 * method argument and return type collections.
	 */
	public static <E extends CharSequence> Collection<E>
			getLongWords(List<E> coll)
	{
		Collection<E> longWords = new ArrayList<E>();
		for (E word : coll)
			if (word.length() > 6)
				longWords.add(word);
		return longWords;
	}

	public static <E extends Integer> List<E> process(List<E> nums)
	{
		List<E> list = new ArrayList<E>();
		list.addAll(nums);
		return list;
	}
}
```

Inner Classes

Inner Classes

- A "regular" inner class is declared *inside* the curly braces of another class, but *outside* any method or other code block.
- An inner class is a full-fledged member of the enclosing (outer) class, so it can be marked with an access modifier as well as the **`abstract`** **or** **`final`** modifiers. (Never both `abstract` and `final` together— remember that `abstract` *must* be subclassed, whereas `final` *cannot* be subclassed).
- An inner class instance shares a special relationship with an instance of the enclosing class. This relationship gives the **inner class access to *all* of the outer class's members, including those marked `private`.**
- To instantiate an inner class, you must have a reference to an instance of the outer class.
- From code within the enclosing class, you can instantiate the inner class using only the name of the inner class, as follows:

```
MyInner mi = new MyInner();
```

- From code outside the enclosing class's instance methods, you can instantiate the inner class only by using both the inner and outer class names, and a reference to the outer class as follows:

```
MyOuter mo = new MyOuter();
MyOuter.MyInner inner = mo.new MyInner();
```

- From code within the inner class, the keyword `this` holds a reference to the inner class instance. To reference the *outer* `this` (in other words, the instance of the outer class that this inner instance is tied to) precede the keyword `this` with the outer class name as follows: **`MyOuter.this`**;

Example: A simple LinkedList using Generics and InnerClass.

```
public class LinkList<E>
{
	// Head of the list.
	private Node<E> head;
	private static int x = 20;

	LinkList()
	{
		head = null;
	}

	// Inner class Node.
	private final class Node<E>
	{
		private Node<E> next;
		private E data;

		public Node(E data)
		{
			this.data = data;
			this.next = null;
		}

		public E getData() { return data;}
		public Node<E> getNextNode() { return this.next;}
		public void setNextNode(Node<E> node)
		{
			this.next = node;
		}
	}

	public static void main(String[] args)
	{
		LinkList<String> strList = new LinkList<String>();
		strList.add("Watsh");
		strList.add("Rajneesh");
		strList.add("Rocks!");
		strList.print();

		LinkList<Integer> intList = new LinkList<Integer>();
		intList.add(12);
		intList.add(122);
		intList.add(11);
		intList.print();

		methodInnerClass();

		// Anonymous inner class.
		Runnable r = new Runnable() {
			public void run() {
				System.out.println("From Anonymous inner class");
			}
		};
		Thread t = new Thread(r);
		t.start();

		// Argument local anonymous inner class
		threadLauncher(new Runnable() {
			public void run() {
				System.out.println("From Argument Local
                                        Anonymous inner class");
			}
		});
```

```
		Outer.StaticInnerClass staticInner = new Outer.StaticInnerClass();
		staticInner.go();

		Outer.NonStaticInnerClass nonStaticInner =
					new Outer().new NonStaticInnerClass();
		nonStaticInner.go();
	}

	// Append to list.
	public Node<E> add(E data)
	{
		if (head == null) {
			head = new Node<E>(data);
			return head;
		}

		Node<E> p = null;
		for (p = head; p.getNextNode() != null ;p = p.getNextNode());
		p.setNextNode(new Node(data));
		return p.getNextNode();
	}

	// Print the list.
	public void print()
	{
		Node<E> p = null;
		for (p = head; p != null;p = p.getNextNode()) {
			System.out.println(p.getData());
		}
	}

	// demo method local inner class
	public static void methodInnerClass()
	{
		final int DISTANCE = 20;
		// can be marked final or abstract
		final class MethodInner
		{
			private int xyz;
			// can access only final variable in a method
			// but can access all variables (even private)
			// in the outer class.
			public MethodInner(int i) { xyz = i + DISTANCE + x; }

			public int getXyz() { return xyz;}
		}
		// Must instantiate the class within the method body.
		MethodInner mi = new MethodInner(10);
		System.out.println(mi.getXyz());
	}

	// for demo of argument local anonymous inner class
	public static void threadLauncher(Runnable r)
	{
		Thread t = new Thread(r);
		t.start();
	}
}

class Outer
{
	static class StaticInnerClass
	{
		public void go()
		{
			System.out.println("From Static Inner Class");
		}
	}

	class NonStaticInnerClass
	{
```

```
            public void go()
            {
                  System.out.println("From Non Static Inner Class");
            }
      }
}
```

```
O/P: (ordered by insertion order)
Watsh
Rajneesh
Rocks!
12
122
11
50 - from methodInnerClass()
From Anonymous inner class
From Argument Local Anonymous inner class
From Static Inner Class
From Non Static Inner Class
```

Method-Local Inner Classes

- A method-local inner class is defined within a method of the enclosing class.
- For the inner class to be used, you must instantiate it, and **that instantiation must happen within the same method, but *after* the class definition code**.
- A method-local inner class **cannot use variables declared within the method (including parameters) unless those variables are marked `final`.**
- The only modifiers you can apply to a method-local inner class **are `abstract` and `final`.** (Never both at the same time, though.)

Anonymous Inner Classes

- Anonymous inner classes have no name, and their type must be either a subclass of the named type or an implementer of the named interface.
- An anonymous inner class is always created as part of a statement; **don't forget to close the statement after the class definition with a curly brace.** This is a rare case in Java, a curly brace followed by a semicolon.
- Because of polymorphism, the only methods you can call on an anonymous inner class reference are those defined in the reference variable class (or interface), even though the anonymous class is really a subclass or implementer of the reference variable type.
- **An anonymous inner class can extend one subclass *or* implement one interface. Unlike non-anonymous classes (inner or otherwise), an anonymous inner class cannot do both.** In other words, it cannot both extend a class *and* implement an interface, nor can it implement more than one interface.
- An argument-local inner class is declared, defined, and automatically instantiated as part of a method invocation. The key to remember is that the class is being defined within a method argument, so the syntax will end the class definition with a curly brace, followed by a closing parenthesis to end the method call, followed by a semicolon to end the statement: `});`

Static Nested Classes

- Static nested classes are inner classes marked with the `static` modifier.
- A static nested class is *not* an inner class, it's a top-level nested class.
- Because the nested class is static, it does not share any special relationship with an instance of the outer class. In fact, you don't need an instance of the outer class to instantiate a static nested class.
- Instantiating a static nested class requires using both the outer and nested class names as follows:

```
BigOuter.Nested n = new BigOuter.Nested();
```

- A static nested class cannot access non-static members of the outer class, since it does not have an implicit reference to any outer instance (in other words, the nested class instance does not get an *outer* `this` reference).

Threads

Defining, Instantiating, and Starting Threads (Objective 4.1)

- Threads can be created by extending Thread and overriding the `public void run()` method.
- Thread objects can also be created by calling the Thread constructor that takes a Runnable argument. The Runnable object is said to be the *target* of the thread.
- **You can call `start()` on a Thread object only once.** If `start()` is called more than once on a Thread object, it will throw a **RuntimeException**. (**IllegalThreadStateException**)

```
2. class MyThread extends Thread {
3.     public static void main(String [] args) {
4.             MyThread t = new MyThread();
5.             t.start();
6.             System.out.print("one. ");
7.             t.start(); // IllegalThreadStateException
8.             System.out.print("two. ");
9.     }
10.    public void run() {
11.            System.out.print("Thread ");
12.    }
   }
```

- It is legal to create many Thread objects using the same Runnable object as the target.
- When a Thread object is created, it does not become a *thread of execution* until its `start()` method is invoked. When a Thread object exists but hasn't been started, it is in the *new* state and is not considered *alive*.

The Thread class itself implements Runnable. (After all, it has a `run()` method that we were overriding.) This means that you could pass a Thread to another Thread's constructor:

```
Thread t = new Thread(new MyThread());
```

This is a bit silly, but it's legal. In this case, you really just need a Runnnable, and creating a whole other Thread is overkill.

```
3. class MyThread extends Thread {
4.     public static void main(String [] args) {
5.             MyThread t = new MyThread();
6.             Thread x = new Thread(t); // Legal and OK!
7.             x.start();
8.     }
9.     public void run() {
10.            for(int i=0;i<3;++i) {
11.                    System.out.print(i + "..");
12.            }
       }
   }
```

Calling a `run()` method directly just means you're invoking a method from whatever thread is currently executing, and the `run()` method goes onto the current call stack rather than at the beginning of a new call stack. So calling run() method does not start a new thread of execution rather it will be executed on the stack of the current thread of execution.

```
Runnable r = new Runnable();
r.run(); // Legal, but does not start a separate thread
```

Transitioning Between Thread States (Objective 4.2)

- Once a new thread is started, it will always enter the runnable state.
- The thread scheduler can move a thread back and forth between the runnable state and the running state.

- For a typical single-processor machine, only one thread can be running at a time, although many threads may be in the runnable state.
- There is no guarantee that the order in which threads were started determines the order in which they'll run.
- There's no guarantee that threads will take turns in any fair way. It's up to the thread scheduler, as determined by the particular virtual machine implementation. If you want a guarantee that your threads will take turns regardless of the underlying JVM, you can use the `sleep()` method. This prevents one thread from hogging the running process while another thread starves. (In most cases, though, `yield()` works well enough to encourage your threads to play together nicely.)
- A running thread may enter a blocked/waiting state by a `wait()`, `sleep()`, or `join()` call.
- A running thread may enter a blocked/waiting state because it can't acquire the lock for a synchronized block of code.
- When the sleep or wait is over, or an object's lock becomes available, the thread can only reenter the runnable state. It will *go* directly from waiting to running (well, for all practical purposes anyway).
- A dead thread cannot be started again.

Sleep, Yield, and Join (Objective 4.2)

- Sleeping is used to delay execution for a period of time, and no locks are released when a thread goes to sleep.
- A sleeping thread is guaranteed to sleep for at least the time specified in the argument to the `sleep()` method (unless it's interrupted), but there is no guarantee as to when the newly awakened thread will actually return to running.
- The **`sleep()`** method is a `static` method that sleeps the currently executing thread's state. **One thread *cannot* tell another thread to sleep.**
- The **`setPriority()`** method is used on Thread objects to give threads a priority of between **1 (low) and 10 (high)**, although priorities are not guaranteed, and not all JVMs recognize 10 distinct priority levels—some levels may be treated as effectively equal.

```
Thread.MIN_PRIORITY (1)
Thread.NORM_PRIORITY (5) - default thread priority
Thread.MAX_PRIORITY (10)
```

- If not explicitly set, a thread's priority will have the **same priority as the priority of the thread that created it.**
- The **`yield()`** method *may* cause a running thread to back out if there are runnable threads of the same priority. There is no guarantee that this will happen, and there is no guarantee that when the thread backs out there will be a *different* thread selected to run. **A thread might yield and then immediately reenter the running state.**
- The closest thing to a guarantee is that at any given time, when a thread is running it will usually not have a lower priority than any thread in the runnable state. If a low-priority thread is running when a high-priority thread enters runnable, the JVM will usually preempt the running low-priority thread and put the high-priority thread in.
- When one thread calls the **`join()`** method of another thread, **the currently running thread will wait until the thread it joins with has completed.** Think of the `join()` method as saying, "Hey thread, I want to join on to the end of you. Let me know when you're done, so I can enter the runnable state."

Concurrent Access Problems and Synchronized Threads (Obj. 4.3)

- **`synchronized`** methods prevent more than one thread from accessing an object's critical method code simultaneously.
- You can use the `synchronized` keyword as a method modifier, or to start a synchronized block of code.

- To synchronize a block of code (in other words, a scope smaller than the whole method), you must specify an argument that is the object whose lock you want to synchronize on.
- While only one thread can be accessing synchronized code of a particular instance, multiple threads can still access the same object's *un*synchronized code.
- **When a thread goes to sleep, its locks will be unavailable to other threads.**
- **`static`** methods can be **`synchronized`**, using the lock from the java.lang.Class instance representing that class. So a class level lock on a static synchronized method will only allow one thread to run at any point in time.

Communicating with Objects by Waiting and Notifying (Obj. 4.4)

- The **`wait()`** method lets a thread say, "there's nothing for me to do now, so put me in your **waiting pool** and notify me when something happens that I care about." Basically, a `wait()` call means "wait me in your pool," or "add me to your waiting list."
- The **`notify()`** method is used to **send a signal to one and only one of the threads that are waiting in that same object's waiting pool.**
- The `notify()` method can NOT specify which waiting thread to notify.
- The method **`notifyAll()`** works in the same way as `notify()`, **only it sends the signal to *all* of the threads waiting on the object.**
- **All three methods—`wait()`, `notify()`, and `notifyAll()`—must be called from within a `synchronized` context!** A thread invokes `wait()` or `notify()` on a particular object, and the thread must currently hold the lock on that object.

TABLE 9-2 Key Thread Methods

Class Object	Class Thread	Interface Runnable
wait ()	start()	run()
notify()	*yield()*	
notifyAll()	*sleep()*	
	join()	

Deadlocked Threads (Objective 4.4)

- Deadlocking is when thread execution grinds to a halt because the code is waiting for locks to be removed from objects.
- Deadlocking can occur when a locked object attempts to access another locked object that is trying to access the first locked object. In other words, **both threads are waiting for each other's locks to be released; therefore, the locks will *never* be released!**
- Deadlocking is bad. Don't do it.

Given:

```
public class Letters extends Thread
{
	private String name;
	public Letters(String name) { this.name = name; }

	public void write()
	{
		System.out.print(name);
		System.out.print(name);
	}

	public static void main(String[] args)
	{
		new Letters("X").start();
		new Letters("Y").start();
	}
}
```

We want to guarantee that the output can be either XXYY or YYXX, but never XYXY or any other combination. Which of the following method definitions could be added to the Letters

class to make this guarantee? (Choose all that apply.)

A. `public void run() { write(); }`
B. `public synchronized void run() { write(); }`
C. `public static synchronized void run() { write(); }`
D. `public void run() { synchronized(this) { write(); } }`
E. `public void run() { synchronized(Letters.class) { write(); } }`
F. `public void run() { synchronized(System.out) { write(); } }`
G. `public void run() { synchronized(System.out.class) { write(); } }`

Answer:

- **E** and **F** are correct. **E** and **F** both cause both threads to lock on the same object, which will prevent the threads from running simultaneously, and guarantee `XXYY` or `YYXX`. It's a bit nusual to lock on an object like `System.out`, but it's perfectly legal, and both threads are locking on the same object.
- **A** can't guarantee anything since it has no synchronization. **B** and **D** both synchronize on. an instance of the Letters class—but since there are two different instances in the `main()` method, the two threads do not block each other and may run simultaneously, resulting in output like `XYXY`. **C** won't compile because it tries to override `run()` with a `static` method, and also calls a non-`static` method from a `static` method. **G** won't compile because `System.out.class` is nonsense. A class literal must start with a class name. `System.out` is a field not a class, so `System.out.class` is not a valid class literal. (Objective 4.3)

Note: Letters is a class so to lock on the class such that the two difference instances of Letters thread use the same lock (class level locking), we use **Letters.class.**
System.out is a **static member of System** class so we cannot use System.out.class rather should use just **System.out** to lock on the static object instance.

Given the following

```
1. public class WaitTest {
2.     public static void main(String [] args) {
3.            System.out.print("1 ");
4.            synchronized(args){
5.                   System.out.print("2 ");
6.                   try {
7.                          args.wait();
8.                   }
9.                   catch(InterruptedException e){}
10.           }
11.           System.out.print("3 ");
12.    }
13.}
```

What is the result of trying to compile and run this program?

O/P: 1 2

1 and 2 will be printed, but there will be no return from the `wait` call because no other thread will notify the main thread, so 3 will never be printed. It's frozen at line 7. notify() or notifyAll() must be called on the object args before the main thread will come out of wait().

Given the scenario: This class is intended to allow users to write a series of messages, so that each message is identified with a timestamp and the name of the thread that wrote the message:

```
public class Logger {
       private StringBuilder contents = new StringBuilder();
       public void log(String message) {
              contents.append(System.currentTimeMillis());
              contents.append(": ");
              contents.append(Thread.currentThread().getName());
              contents.append(message);
              contents.append("\n");
       }
       public String getContents() { return contents.toString(); }
}
```

How can we ensure that instances of this class can be safely used by multiple threads?

A. This class is already thread-safe.
B. Replacing `StringBuilder` with `StringBuffer` will make this class thread-safe.
C. Synchronize the `log()` method only.
D. Synchronize the `getContents()` method only.
E. Synchronize both `log()` and `getContents()`.
F. This class cannot be made thread-safe.

Answer:
✹ **E** is correct. Synchronizing the `public` methods is sufficient to make this safe, so **F** is false. This class is not thread-safe unless some sort of synchronization protects the changing data.
✹ **B** is not correct because although a `StringBuffer` is synchonized internally, we call `append()` multiple times, and nothing would prevent two simultaneous `log()` calls from mixing up their messages. **C** and **D** are not correct **because if one method remains unsynchronized, it can run while the other is executing, which could result in reading the contents while one of the messages is incomplete**, or worse. (You don't want to call `toString()` on the `StringBuilder` as it's resizing its internal character array.) (Objective 4.3)

Given:
```
public static synchronized void main(String[] args) throws
                            InterruptedException {
       Thread t = new Thread();
       t.start();
       System.out.print("X");
       t.wait(10000); -- first use synchronized(t) to get lock of t
       System.out.print("Y");
}
```
What is the result of this code?
A. It prints `X` and exits.
B. It prints `X` and never exits.
C. It prints `XY` and exits almost immeditately.
D. It prints `XY` with a 10-second delay between `X` and `Y`.
E. It prints `XY` with a 10000-second delay between `X` and `Y`.
F. The code does not compile.
G. An exception is thrown at runtime.
Answer:
✹ **G** is correct. The code does not acquire a lock on `t` before calling `t.wait()`, so it throws an `IllegalThreadStateException`. The method is `synchronized`, but it's not synchronized on `t` so the exception will be thrown. If the wait were placed inside a `synchronized(t)` block, then the answer would have been **D**.
✹ **A, B, C, D, E,** and **F** are incorrect based the logic described above. (Objective 4.2)

Given:
```
public class TwoThreads {
       static Thread laurel, hardy;
       public static void main(String[] args) {
              laurel = new Thread() {
                     public void run() {
                            System.out.println("A");
                            try {
                                   hardy.sleep(1000);
                            } catch (Exception e) {
                                   System.out.println("B");
                            }
                            System.out.println("C");
                     }
              };
              hardy = new Thread() {
                     public void run() {
                            System.out.println("D");
                            try {
```

```
                        laurel.wait();
                    } catch (Exception e) {
                        System.out.println("E");
                    }
                    System.out.println("F");
                }
            };
            laurel.start();
            hardy.start();
        }
}
```

Which letters will eventually appear somewhere in the output? (Choose all that apply.)

A. `A`
B. `B`
C. `C`
D. `D`
E. `E`
F. `F`
G. The answer cannot be reliably determined.
H. The code does not compile.

Answer: A, C, D, E and F will be printed.
hardy.sleep() will resolve to Thread.sleep() as sleep is static method.
laurel.wait() does not synchronize on laurel object – ie. Does not take a lock on laurel before calling wait() on the object so it will cause an IllegalThreadStateException and this exception will be caught in the catch all handler which is catching Exception class itself!

Development

Using javac and java (Objective 7.2)

- Use `-d` to change the destination of a class file when it's first generated by the `javac` command.
- The `-d` option can build package-dependent destination classes on-the-fly if the *root* package directory already exists.
- Use the `-D` option in conjunction with the `java` command when you want to set a system property.
- System properties consist of name=value pairs that must be appended directly behind the `-D`, for example, `java -Dmyproperty=myvalue`.
- Command-line arguments are always treated as Strings.
- The `java` command-line argument 1 is put into array element 0, argument 2 is put into element 1, and so on.

Searching with java and javac (Objective 7.5)

- Both `java` and `javac` use the same algorithms to search for classes.
- Searching begins in the locations that contain the classes that come standard with J2SE.
- Users can define secondary search locations using classpaths.
- Default classpaths can be defined by using OS environment variables.
- A classpath can be declared at the command line, and it overrides the default classpath.
- A single classpath can define many different search locations.
- In Unix classpaths, forward slashes (`/`) are used to separate the directories that make up a path. In Windows, backslashes (`\`) are used.
- In Unix, colons (:) are used to separate the paths within a classpath. In Windows, semicolons (;) are used.
- In a classpath, to specify the current directory as a search location, use a dot (.)
- In a classpath, once a class is found, searching stops, so the order of locations to search is important.

Packages and Searching (Objective 7.5)

- When a class is put into a package, its fully qualified name must be used.
- An `import` statement provides an alias to a class's fully qualified name.
- In order for a class to be located, its fully qualified name must have a tight relationship with the directory structure in which it resides.
- A classpath can contain both relative and absolute paths.
- An absolute path starts with a `/` or a `\`.
- Only the final directory in a given path will be searched.

JAR files (Objective 7.5)

- An entire directory tree structure can be archived in a single JAR file.
- JAR files can be searched by `java` and `javac`.
- When you include a JAR file in a classpath, you must include not only the directory in which the JAR file is located, but the name of the JAR file too.
- For testing purposes, you can put JAR files into `.../jre/lib/ext`, which is somewhere inside the Java directory tree on your machine.

Static Imports (Objective 7.1)

- You must start a static import statement like this: `import static`
- You can use static imports to create shortcuts for `static` members (static variables, constants, and methods) of any class.