OPERATORS AND ASSIGNMENTS:

1.

What will be the output of the program?

```
class PassA
{
   public static void main(String [] args)
   {
      PassA p = new PassA();
      p.start();
   }

   void start()
   {
      long [] a1 = {3,4,5};
      long [] a2 = fix(a1);
      System.out.print(a1[0] + a1[1] + a1[2] + " ");
      System.out.println(a2[0] + a2[1] + a2[2]);
   }

   long [] fix(long [] a3)
   {
      a3[1] = 7;
      return a3;
   }
}
```

a) 12 15
b) 15 15
c) 3 4 5 3 7 5
d) 3 7 5 3 7 5

Answer: Option b

Explanation:

Output: 15 15

The reference variables a1 and a3 refer to the same long array object. When the [1] element is updated in the fix() method, it is updating the array referred to by a1. The reference variable a2 refers to the same array object.

So Output: 3+7+5+" "3+7+5

Output: 15 15 Because Numeric values will be added

2.

What will be the output of the program?

```
class Test
{
   public static void main(String [] args)
   {
      Test p = new Test();
      p.start();
   }

   void start()
   {
      boolean b1 = false;
      boolean b2 = fix(b1);
      System.out.println(b1 + " " + b2);
   }

   boolean fix(boolean b1)
   {
      b1 = true;
      return b1;
   }
}
```

a) true true
b) false true
c) true false
d) false false

Answer: Option b

Explanation:

The boolean b1 in the fix() method is a different boolean than the b1 in the start() method. The b1 in the start() method is not updated by the fix() method.

3.

What will be the output of the program?

```
class PassS
{
   public static void main(String [] args)
   {
      PassS p = new PassS();
      p.start();
   }

   void start()
   {
      String s1 = "slip";
      String s2 = fix(s1);
      System.out.println(s1 + " " + s2);
   }

   String fix(String s1)
   {
      s1 = s1 + "stream";
      System.out.print(s1 + " ");
      return "stream";
   }
}
```

a) slip stream
b) slipstream stream
c) stream slip stream

d) slipstream slip stream

Answer: Option d

Explanation:

When the fix() method is first entered, start()'s s1 and fix()'s s1 reference variables both refer to the same String object (with a value of "slip"). Fix()'s s1 is reassigned to a new object that is created when the concatenation occurs (this second String object has a value of "slipstream"). When the program returns to start(), another String object is created, referred to by s2 and with a value of "stream".

4.

What will be the output of the program?

```
class BitShift
{
   public static void main(String [] args)
   {
      int x = 0x80000000;
      System.out.print(x + " and  ");
      x = x >>> 31;
      System.out.println(x);
   }
}
```

a) -2147483648 and 1
b) 0x80000000 and 0x00000001
c) -2147483648 and -1
d) 1 and -2147483648

Answer: Option a

Explanation:

Option A is correct. The >>> operator moves all bits to the right, zero filling the left bits. The bit transformation looks like this:

Before: 1000 0000 0000 0000 0000 0000 0000 0000

After: 0000 0000 0000 0000 0000 0000 0000 0001

Option C is incorrect because the >>> operator zero fills the left bits, which in this case changes the sign of x, as shown.

Option B is incorrect because the output method print() always displays integers in base 10.

Option D is incorrect because this is the reverse order of the two output numbers.

5.

What will be the output of the program?

```
class Equals
{
   public static void main(String [] args)
   {
      int x = 100;
      double y = 100.1;
      boolean b = (x = y); /* Line 7 */
      System.out.println(b);
   }
}
```

a) true
b) false
c) Compilation fails
d) An exception is thrown at runtime

Answer: Option c

Explanation:

The code will not compile because in line 7, the line will work only if we use (x==y) in the line.
The == operator compares values to produce a boolean, whereas the = operator assigns a value to variables.

Option A, B, and D are incorrect because the code does not get as far as compiling. If we corrected this code, the output would be false

6.

What will be the output of the program?

```
class Test
{
   public static void main(String [] args)
   {
      int x=20;
      String sup = (x < 15) ? "small" : (x < 22)? "tiny" : "huge";
```

```
        System.out.println(sup);
    }
}
```

a) small
b) tiny
c) huge
d) Compilation fails

Answer: Option b

Explanation:

This is an example of a nested ternary operator. The second evaluation (x < 22) is true, so the "tiny" value is assigned to sup.

7.

What will be the output of the program?

```
class Test
{
    public static void main(String [] args)
    {
        int x= 0;
        int y= 0;
        for (int z = 0; z < 5; z++)
        {
            if (( ++x > 2 ) && (++y > 2))
            {
                x++;
            }
        }
        System.out.println(x + " " + y);
    }
}
```

a) 5 2
b) 5 3
c) 6 3

d) 6 4

Answer: Option c

Explanation:

In the first two iterations x is incremented once and y is not because of the short circuit && operator. In the third and forth iterations x and y are each incremented, and in the fifth iteration x is doubly incremented and y is incremented.

8.

What will be the output of the program?

```
class Test
{
   public static void main(String [] args)
   {
      int x= 0;
      int y= 0;
      for (int z = 0; z < 5; z++)
      {
         if (( ++x > 2 ) || (++y > 2))
         {
            x++;
         }
      }
   System.out.println(x + " " + y);
   }
}
```

a) 5 3
b) 8 2
c) 8 3
d) 8 5

Answer: Option b

Explanation:

The first two iterations of the for loop both x and y are incremented. On the third iteration x is incremented, and for the first time becomes greater than 2. The short circuit or

operator || keeps y from ever being incremented again and x is incremented twice on each of the last three iterations.

9.

What will be the output of the program?

```
class Bitwise
{
   public static void main(String [] args)
   {
      int x = 11 & 9;
      int y = x ^ 3;
      System.out.println( y | 12 );
   }
}
```

a) 0
b) 7
c) 8
d) 14

Answer: Option d

Explanation:

The & operator produces a 1 bit when both bits are 1. The result of the & operation is 9.
The ^ operator produces a 1 bit when exactly one bit is 1; the result of this operation is 10.
The | operator produces a 1 bit when at least one bit is 1; the result of this operation is 14.

10.

What will be the output of the program?

```
class SSBool
{
   public static void main(String [] args)
   {
      boolean b1 = true;
      boolean b2 = false;
      boolean b3 = true;
```

```
    if ( b1 & b2 | b2 & b3 | b2 ) /* Line 8 */
      System.out.print("ok ");
    if ( b1 & b2 | b2 & b3 | b2 | b1 ) /*Line 10*/
      System.out.println("dokey");
  }
}
```

a) ok
b) dokey
c) ok dokey
d) No output is produced
e) Compilation error

Answer: Option b

Explanation:

The & operator has a higher precedence than the | operator so that on line 8 b1 and b2 are evaluated together as are b2 & b3. The final b1 in line 10 is what causes that if test to be true. Hence it prints "dokey".

11.

What will be the output of the program?

```
class SC2
{
  public static void main(String [] args)
  {
    SC2 s = new SC2();
    s.start();
  }

  void start()
  {
    int a = 3;
    int b = 4;
    System.out.print(" " + 7 + 2 + " ");
    System.out.print(a + b);
    System.out.print(" " + a + b + " ");
```

```
    System.out.print(foo() + a + b + " ");
    System.out.println(a + b + foo());
  }

  String foo()
  {
    return "foo";
  }
}
```

a) 9 7 7 foo 7 7foo
b) 72 34 34 foo34 34foo
c) 9 7 7 foo34 34foo
d) 72 7 34 foo34 7foo

Answer: Option d

Explanation:

Because all of these expressions use the + operator, there is no precedence to worry about and all of the expressions will be evaluated from left to right. If either operand being evaluated is a String, the + operator will concatenate the two operands; if both operands are numeric, the + operator will add the two operands.

12.

What will be the output of the program?

```
class Test
{
  static int s;
  public static void main(String [] args)
  {
    Test p = new Test();
    p.start();
    System.out.println(s);
  }

  void start()
```

```
  {
    int x = 7;
    twice(x);
    System.out.print(x + " ");
  }

  void twice(int x)
  {
    x = x*2;
    s = x;
  }
}
```

a) 7 7
b) 7 14
c) 14 0
d) 14 14

Answer: Option b

Explanation:

The int x in the twice() method is not the same int x as in the start() method. Start()'s x is not affected by the twice() method. The instance variable s is updated by twice()'s x, which is 14.

13.

What will be the output of the program?

```
class Two
{
  byte x;
}

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

```
    PassO p = new PassO();
    p.start();
  }

  void start()
  {
    Two t = new Two();
    System.out.print(t.x + " ");
    Two t2 = fix(t);
    System.out.println(t.x + " " + t2.x);
  }

  Two fix(Two tt)
  {
    tt.x = 42;
    return tt;
  }
}
```

a) null null 42
b) 0 0 42
c) 0 42 42
d) 0 0 0

Answer: Option c

Explanation:

In the fix() method, the reference variable tt refers to the same object (class Two) as the t reference variable. Updating tt.x in the fix() method updates t.x (they are one in the same object). Remember also that the instance variable x in the Two class is initialized to 0.

14.

What will be the output of the program?

```
class BoolArray
{
  boolean [] b = new boolean[3];
```

```
    int count = 0;


    void set(boolean [] x, int i)
    {
        x[i] = true;
        ++count;
    }


    public static void main(String [] args)
    {
        BoolArray ba = new BoolArray();
        ba.set(ba.b, 0);
        ba.set(ba.b, 2);
        ba.test();
    }


    void test()
    {
        if ( b[0] && b[1] | b[2] )
            count++;
        if ( b[1] && b[(++count - 2)] )
            count += 7;
        System.out.println("count = " + count);
    }
}
```

a) count = 0
b) count = 2
c) count = 3
d) count = 4

Answer: Option c

Explanation: The reference variables b and x both refer to the same boolean array. count is incremented for each call to the set() method, and once again when the first if test is true. Because of the && short circuit operator, count is not incremented during the second if test.

15.

What will be the output of the program?

```
public class Test
{
   public static void leftshift(int i, int j)
   {
      i <<= j;
   }
   public static void main(String args[])
   {
      int i = 4, j = 2;
      leftshift(i, j);
      System.out.println(i);
   }
}
```

a) 2
b) 4
c) 8
d) 16

Answer: Option b

Explanation:

Java only ever passes arguments to a method by value (i.e. a copy of the variable) and never by reference. Therefore the value of the variable i remains unchanged in the main method.

If you are clever you will spot that 16 is 4 multiplied by 2 twice, (4 * 2 * 2) = 16. If you had 16 left shifted by three bits then 16 * 2 * 2 * 2 = 128. If you had 128 right shifted by 2 bits then 128 / 2 / 2 = 32. Keeping these points in mind, you don't have to go converting to binary to do the left and right bit shifts.

Objects and Collections:

1.

Suppose that you would like to create an instance of a new Map that has an iteration order that is the same as the iteration order of an existing instance of a Map. Which concrete implementation of the Map interface should be used for the new instance?

a) TreeMap
b) HashMap
c) LinkedHashMap
d) The answer depends on the implementation of the existing instance.

Answer: Option C

Explanation:

The iteration order of a Collection is the order in which an iterator moves through the elements of the Collection. The iteration order of a LinkedHashMap is determined by the order in which elements are inserted.

When a new LinkedHashMap is created by passing a reference to an existing Collection to the constructor of a LinkedHashMap the Collection.addAll method will ultimately be invoked.

The addAll method uses an iterator to the existing Collection to iterate through the elements of the existing Collection and add each to the instance of the new LinkedHashMap.

Since the iteration order of the LinkedHashMap is determined by the order of insertion, the iteration order of the new LinkedHashMap must be the same as the interation order of the old Collection.

2.

Which class does not override the equals() and hashCode() methods, inheriting them directly from class Object?

a) java.lang.String
b) java.lang.Double
c) java.lang.StringBuffer
d) java.lang.Character

Answer: Option C

Explanation:

java.lang.StringBuffer is the only class in the list that uses the default methods provided by class Object.

3.

Which collection class allows you to grow or shrink its size and provides indexed access to its elements, but whose methods are not synchronized?

a) java.util.HashSet
b) java.util.LinkedHashSet
c) java.util.List
d) java.util.ArrayList

Answer: Option D

Explanation:

All of the collection classes allow you to grow or shrink the size of your collection. ArrayList provides an index to its elements. The newer collection classes tend not to have synchronized methods. Vector is an older implementation of ArrayList functionality and has synchronized methods; it is slower than ArrayList.

4.

You need to store elements in a collection that guarantees that no duplicates are stored and all elements can be accessed in natural order. Which interface provides that capability?

a) java.util.Map
b) java.util.Set
c) java.util.List
d) java.util.Collection

Answer: Option B

Explanation:

Option B is correct. A set is a collection that contains no duplicate elements. The iterator returns the elements in no particular order (unless this set is an instance of some class that provides a guarantee). A map cannot contain duplicate keys but it may contain duplicate values. List and Collection allow duplicate elements.

Option A is wrong. A map is an object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value. The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The order of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the TreeMap class, make specific guarantees as to their order (ascending key order); others, like the HashMap class, do not (does not guarantee that the order will remain constant over time).

Option C is wrong. A list is an ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list. Unlike sets, lists typically allow duplicate elements.

Option D is wrong. A collection is also known as a sequence. The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list. Unlike sets, lists typically allow duplicate elements.

5.

Which interface does java.util.Hashtable implement?

a) Java.util.Map
b) Java.util.List
c) Java.util.HashTable
d) Java.util.Collection

Answer: Option A

Explanation:

Hash table based implementation of the Map interface.

6.

Which interface provides the capability to store objects using a key-value pair?

a) Java.util.Map
b) Java.util.Set
c) Java.util.List
d) Java.util.Collection

Answer: Option A

Explanation:

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

7.

Which collection class allows you to associate its elements with key values, and allows you to retrieve objects in FIFO (first-in, first-out) sequence?

a) java.util.ArrayList
b) java.util.LinkedHashMap
c) java.util.HashMap
d) java.util.TreeMap

Answer: Option B

Explanation:

LinkedHashMap is the collection class used for caching purposes. FIFO is another way to indicate caching behavior. To retrieve LinkedHashMap elements in cached order, use the values() method and iterate over the resultant collection.

8.

Which collection class allows you to access its elements by associating a key with an element's value, and provides synchronization?

a) java.util.SortedMap
b) java.util.TreeMap
c) java.util.TreeSet
d) java.util.Hashtable

Answer: Option D

Explanation:

Hashtable is the only class listed that provides synchronized methods. If you need synchronization great; otherwise, use HashMap, it's faster.

9.

Which is valid declaration of a float?

a) float f = 1F;
b) float f = 1.0;
c) float f = "1";
d) float f = 1.0d;

Answer: Option A

Explanation:

Option A is valid declaration of float.

Option B is incorrect because any literal number with a decimal point u declare the computer will implicitly cast to double unless you include "F or f"

Option C is incorrect because it is a String.

Option D is incorrect because "d" tells the computer it is a double so therefore you are trying to put a double value into a float variable i.e there might be a loss of precision.

10.

```
/* Missing Statement ? */
public class foo
{
   public static void main(String[]args)throws Exception
   {
      java.io.PrintWriter out = new java.io.PrintWriter();
      new java.io.OutputStreamWriter(System.out,true);
      out.println("Hello");
   }
```

}

What line of code should replace the missing statement to make this program compile?

a) No statement required.
b) import java.io.*;
c) include java.io.*;
d) import java.io.PrintWriter;

Answer: Option A

Explanation:

The usual method for using/importing the java packages/classes is by using an import statement at the top of your code. However it is possible to explicitly import the specific class that you want to use as you use it which is shown in the code above. The disadvantage of this however is that every time you create a new object you will have to use the class path in the case "java.io" then the class name in the long run leading to a lot more typing.

11.

What is the numerical range of char?

a) 0 to 32767
b) 0 to 65535
c) -256 to 255
d) -32768 to 32767

Answer: Option b

Explanation:

The char type is integral but unsigned. The range of a variable of type char is from 0 to 2^{16}-1 or 0 to 65535. Java characters are Unicode, which is a 16-bit encoding capable of representing a wide range of international characters. If the most significant nine bits of a char are 0, then the encoding is the same as seven-bit ASCII.

12.

Which of the following are Java reserved words?

1. run
2. import
3. default
4. implement

a) 1 and 2
b) 2 and 3
c) 3 and 4
d) 2 and 4

Answer: Option b

Explanation:

(2) - This is a Java keyword

(3) - This is a Java keyword

(1) - Is incorrect because although it is a method of Thread/Runnable it is not a keyword

(4) - This is not a Java keyword the keyword is implements

Inner Classes:

1.

Which is true about an anonymous inner class?

a) It can extend exactly one class and implement exactly one interface.
b) It can extend exactly one class and can implement multiple interfaces.
c) It can extend exactly one class or implement exactly one interface.
d) It can implement multiple interfaces regardless of whether it also extends a class.

Answer: Option C

Explanation:

Option C is correct because the syntax of an anonymous inner class allows for only one named type after the new, and that type must be either a single interface (in which case the anonymous class implements that one interface) or a single class (in which case the anonymous class extends that one class).

Option A, B, D, and E are all incorrect because they don't follow the syntax rules described in the response for answer Option C.

2.

```
class Boo
{
   Boo(String s) { }
   Boo() { }
}
class Bar extends Boo
{
   Bar() { }
   Bar(String s) {super(s);}
   void zoo()
```

```
  {
  // insert code here
  }
}
```

which one create an anonymous inner class from within class Bar?

a) Boo f = new Boo(24) { };
b) Boo f = new Bar() { };
c) Bar f = new Boo(String s) { };
d) Boo f = new Boo.Bar(String s) { };

Answer: Option B

Explanation:

Option B is correct because anonymous inner classes are no different from any other class when it comes to polymorphism. That means you are always allowed to declare a reference variable of the superclass type and have that reference variable refer to an instance of a subclass type, which in this case is an anonymous subclass of Bar. Since Bar is a subclass of *Boo*, it all works.

Option A is incorrect because it passes an int to the *Boo* constructor, and there is no matching constructor in the *Boo* class.

Option C is incorrect because it violates the rules of polymorphismâ€”you cannot refer to a superclass type using a reference variable declared as the subclass type. The superclass is not guaranteed to have everything the subclass has.

Option D uses incorrect syntax.

3.

Which is true about a method-local inner class?

a) It must be marked final.
b) It can be marked abstract.
c) It can be marked public.
d) It can be marked static.

Answer: Option B

Explanation:

Option B is correct because a method-local inner class can be abstract, although it means a subclass of the inner class must be created if the abstract class is to be used (so an abstract method-local inner class is probably not useful).

Option A is incorrect because a method-local inner class does not have to be declared *final* (although it is legal to do so).

C and D are incorrect because a method-local inner class cannot be made *public* (remember-you cannot mark any local variables as *public*), or *static*.

4.

Which statement is true about a static nested class?

a) You must have a reference to an instance of the enclosing class in order to instantiate it.
b) It does not have access to nonstatic members of the enclosing class.
c) It's variables and methods must be static.
d) It must extend the enclosing class.

Answer: Option B

Explanation:

Option B is correct because a static nested class is not tied to an instance of the enclosing class, and thus can't access the nonstatic members of the class (just as a static method can't access nonstatic members of a class).

Option A is incorrect because static nested classes do not need (and can't use) a reference to an instance of the enclosing class.

Option C is incorrect because static nested classes can declare and define nonstatic members.

Option D is wrong because it just is. There's no rule that says an inner or nested class has to extend anything.

5.

Which constructs an anonymous inner class instance?

a) Runnable r = new Runnable() { };
b) Runnable r = new Runnable(public void run() { });
c) Runnable r = new Runnable { public void run(){}};
d) System.out.println(new Runnable() {public void run() { }});

Answer: Option D

Explanation:

D is correct. It defines an anonymous inner class instance, which also means it creates an instance of that new anonymous class at the same time. The anonymous class is an implementer of the Runnable interface, so it must override the *run()* method of *Runnable*.

A is incorrect because it doesn't override the *run()* method, so it violates the rules of interface implementation.

B and C use incorrect syntax.

6.

```
class Foo

{

   class Bar{ }
```

```
}
class Test
{
   public static void main (String [] args)
   {
      Foo f = new Foo();
      /* Line 10: Missing statement ? */
   }
}
```

which statement, inserted at line 10, creates an instance of Bar?

a) Foo.Bar b = new Foo.Bar();
b) Foo.Bar b = f.new Bar();
c) Bar b = new f.Bar();
d) Bar b = f.new Bar();

Answer: Option B

Explanation:

Option B is correct because the syntax is correct-using both names (the enclosing class and the inner class) in the reference declaration, then using a reference to the enclosing class to invoke new on the inner class.

Option A, C and D all use incorrect syntax. A is incorrect because it doesn't use a reference to the enclosing class, and also because it includes both names in the new.

C is incorrect because it doesn't use the enclosing class name in the reference variable declaration, and because the new syntax is wrong.

D is incorrect because it doesn't use the enclosing class name in the reference variable declaration.

7.

```
public class MyOuter
{
   public static class MyInner
   {
      public static void foo() { }
   }
}
```

which statement, if placed in a class other than MyOuter or *MyInner*, instantiates an instance of the nested class?

a) MyOuter.MyInner m = new MyOuter.MyInner();
b) MyOuter.MyInner mi = new MyInner();
c) MyOuter m = new MyOuter();
d) MyOuter.MyInner mi = m.new MyOuter.MyInner();
e) MyInner mi = new MyOuter.MyInner();

Answer: Option A

Explanation:

MyInner is a static nested class, so it must be instantiated using the fully-scoped name of *MyOuter.MyInner*.

Option B is incorrect because it doesn't use the enclosing name in the new.

Option C is incorrect because it uses incorrect syntax. When you instantiate a nested class by invoking new on an instance of the enclosing class, you do not use the enclosing name. The difference between Option A and C is that Option C is calling new on an instance of the enclosing class rather than just new by itself.

Option D is incorrect because it doesn't use the enclosing class name in the variable declaration.

Threads:

1.

What is the name of the method used to start a thread execution?

a) init();
b) start();
c) run();
d) resume();

Answer: Option B

Explanation:

Option B is Correct. The start() method causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.

Option A is wrong. There is no init() method in the Thread class.

Option C is wrong. The run() method of a thread is like the main() method to an application. Starting the thread causes the object's run method to be called in that separately executing thread.

Option D is wrong. The resume() method is deprecated. It resumes a suspended thread.

2.

Which two are valid constructors for Thread?

1. Thread(Runnable r, String name)
2. Thread()
3. Thread(int priority)
4. Thread(Runnable r, ThreadGroup g)
5. Thread(Runnable r, int priority)

a) 1 and 3
b) 2 and 4
c) 1 and 2
d) 2 and 5

Answer: Option C

Explanation:

(1) and (2) are both valid constructors for Thread.

(3), (4), and (5) are not legal Thread constructors, although (4) is close. If you reverse the arguments in (4), you'd have a valid constructor.

3.

Which three are methods of the Object class?

1. notify();
2. notifyAll();
3. isInterrupted();
4. synchronized();
5. interrupt();
6. wait(long msecs);
7. sleep(long msecs);
8. yield();

a) 1, 2, 4
b) 2, 4, 5
c) 1, 2, 6
d) 2, 3, 4

Answer: Option C

Explanation:

(1), (2), and (6) are correct. They are all related to the list of threads waiting on the specified object.

(3), (5), (7), and (8) are incorrect answers. The methods isInterrupted() and interrupt() are instance methods of Thread.

The methods sleep() and yield() are static methods of Thread.

D is incorrect because synchronized is a keyword and the synchronized() construct is part of the Java language.

4.

```
class X implements Runnable
{
   public static void main(String args[])
   {
      /* Missing code? */
   }
   public void run() {}
}
```

Which of the following line of code is suitable to start a thread ?

a) Thread t = new Thread(X);
b) Thread t = new Thread(X); t.start();
c) X run = new X(); Thread t = new Thread(run); t.start();
d) Thread t = new Thread(); x.run();

Answer: Option C

Explanation:

Option C is suitable to start a thread.

5.

Which cannot directly cause a thread to stop executing?

a) Calling the SetPriority() method on a Thread object.
b) Calling the wait() method on an object.
c) Calling notify() method on an object.
d) Calling read() method on an InputStream object.

Answer: Option C

Explanation:

Option C is correct. notify() - wakes up a single thread that is waiting on this object's monitor.

6.

Which two of the following methods are defined in class Thread?

1. start()
2. wait()
3. notify()
4. run()
5. terminate()

a) 1 and 4
b) 2 and 3
c) 3 and 4
d) 2 and 4

Answer: Option A

Explanation:

(1) and (4). Only start() and run() are defined by the Thread class.

(2) and (3) are incorrect because they are methods of the Object class. (5) is incorrect because there's no such method in any thread-related class.

7.

Which three guarantee that a thread will leave the running state?

1. yield()
2. wait()
3. notify()
4. notifyAll()
5. sleep(1000)
6. aLiveThread.join()
7. Thread.killThread()

a) 1, 2 and 4
b) 2, 5 and 6
c) 3, 4 and 7
d) 4, 5 and 7

Answer: Option B

Explanation:

(2) is correct because wait() always causes the current thread to go into the object's wait pool.

(5) is correct because sleep() will always pause the currently running thread for at least the duration specified in the sleep argument (unless an interrupted exception is thrown).

(6) is correct because, assuming that the thread you're calling join() on is alive, the thread calling join() will immediately block until the thread you're calling join() on is no longer alive.

(1) is wrong, but tempting. The yield() method is not guaranteed to cause a thread to leave the running state, although if there are runnable threads of the same priority as the currently running thread, then the current thread will probably leave the running state.

(3) and (4) are incorrect because they don't cause the thread invoking them to leave the running state.

(7) is wrong because there's no such method.

8.

Which of the following will directly stop the execution of a Thread?

a) wait()
b) notify()
c) notifyall()
d) exits synchronized code

Answer: Option A

Explanation:

Option A is correct. wait() causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

Option B is wrong. notify() - wakes up a single thread that is waiting on this object's monitor.

Option C is wrong. notifyAll() - wakes up all threads that are waiting on this object's monitor.

Option D is wrong. Typically, releasing a lock means the thread holding the lock (in other words, the thread currently in the synchronized method) exits the synchronized method. At that point, the lock is free until some other thread enters a synchronized method on that object. Does entering/exiting synchronized code mean that the thread execution stops? Not necessarily because the thread can still run code that is not synchronized. I think the word directly in the question gives us a clue. Exiting synchronized code does not directly stop the execution of a thread.

9.

Which method must be defined by a class implementing the java.lang.Runnable interface?

a) void run()
b) public void run()
c) public void start()
d) void run(int priority)

Answer: Option B

a) Explanation:

Option B is correct because in an interface all methods are abstract by default therefore they must be overridden by the implementing class. The Runnable interface only contains 1 method, the void run() method therefore it must be implemented.

Option A and D are incorrect because they are narrowing the access privileges i.e. package(default) access is narrower than public access.

Option C is not method in the Runnable interface therefore it is incorrect.

10.

Which will contain the body of the thread?

b) run();
c) start();
d) stop();
e) main();

Answer: Option A

Explanation:

Option A is Correct. The run() method to a thread is like the main() method to an application. Starting the thread causes the object's run method to be called in that separately executing thread.

Option B is wrong. The start() method causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.

Option C is wrong. The stop() method is deprecated. It forces the thread to stop executing.

Option D is wrong. Is the main entry point for an application.

11.

Which method registers a thread in a thread scheduler?

a) run();
b) construct();
c) start();
d) register();

Answer: Option C

Explanation:

Option C is correct. The start() method causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.

Option A is wrong. The run() method of a thread is like the main() method to an application. Starting the thread causes the object's run method to be called in that separately executing thread.

Option B is wrong. There is no construct() method in the Thread class.

Option D is wrong. There is no register() method in the Thread class.

12.

Assume the following method is properly synchronized and called from a thread A on an object B:

wait(2000);

After calling this method, when will the thread A become a candidate to get another turn at the CPU?

a) After thread A is notified, or after two seconds.
b) After the lock on B is released, or after two seconds.
c) Two seconds after thread A is notified.
d) Two seconds after lock B is released.

Answer: Option A

Explanation:

Option A. Either of the two events (notification or wait time expiration) will make the thread become a candidate for running again.

Option B is incorrect because a waiting thread will not return to runnable when the lock is released, unless a notification occurs.

Option C is incorrect because the thread will become a candidate immediately after notification, not two seconds afterwards.

Option D is also incorrect because a thread will not come out of a waiting pool just because a lock has been released.

13.

Which of the following will not directly cause a thread to stop?

a) notify()
b) wait()
c) InputStream access
d) sleep()

Answer: Option A

Explanation:

Option A is correct. notify() - wakes up a single thread that is waiting on this object's monitor.

Option B is wrong. wait() causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

Option C is wrong. Methods of the InputStream class block until input data is available, the end of the stream is detected, or an exception is thrown. Blocking means that a thread may stop until certain conditions are met.

Option D is wrong. sleep() - Causes the currently executing thread to sleep (temporarily cease execution) for a specified number of milliseconds. The thread does not lose ownership of any monitors.

14.

Which class or interface defines the wait(), notify(),and notifyAll() methods?

a) Object
b) Thread
c) Runnable
d) Class

Answer: Option A

Explanation:

The Object class defines these thread-specific methods.

Option B, C, and D are incorrect because they do not define these methods. And yes, the Java API does define a class called Class, though you do not need to know it for the exam.

15.

```
public class MyRunnable implements Runnable
{
   public void run()
   {
      // some code here
   }
}
```

which of these will create and start this thread?

a) new Runnable(MyRunnable).start();
b) new Thread(MyRunnable).run();
c) new Thread(new MyRunnable()).start();
d) new MyRunnable().start();

Answer: Option C

Explanation:

Because the class implements Runnable, an instance of it has to be passed to the Thread constructor, and then the instance of the Thread has to be started.

A is incorrect. There is no constructor like this for Runnable because Runnable is an interface, and it is illegal to pass a class or interface name to any constructor.

B is incorrect for the same reason; you can't pass a class or interface name to any constructor.

D is incorrect because MyRunnable doesn't have a start() method, and the only start() method that can start a thread of execution is the start() in the Thread class.

GARBAGE COLLECTION:

1

When is the B object, created in line 3, eligible for garbage collection?

a) after line 5
b) after line 6
c) after line 7
d) There is no way to be absolutely certain.

Answer: Option D

Explanation:

No answer description is available

2.

```
class HappyGarbage01
{
   public static void main(String args[])
   {
      HappyGarbage01 h = new HappyGarbage01();
      h.methodA(); /* Line 6 */
   }
   Object methodA()
   {
      Object obj1 = new Object();
      Object [] obj2 = new Object[1];
      obj2[0] = obj1;
      obj1 = null;
      return obj2[0];
   }
}
```

Where will be the most chance of the garbage collector being invoked?

a) After line 9
b) After line 10
c) After line 11

d) Garbage collector never invoked in methodA()

Answer: Option D

Explanation:

Option D is correct. Garbage collection takes place after the method has returned its reference to the object. The method returns to line 6, there is no reference to store the return value. so garbage collection takes place after line 6.

Option A is wrong. Because the reference to obj1 is stored in obj2[0]. The Object obj1 still exists on the heap and can be accessed by an active thread through the reference stored in obj2[0].

Option B is wrong. Because it is only one of the references to the object obj1, the other reference is maintained in obj2[0].

Option C is wrong. The garbage collector will not be called here because a reference to the object is being maintained and returned in obj2[0].

3.

```
class Bar { }
class Test
{
   Bar doBar()
   {
      Bar b = new Bar(); /* Line 6 */
      return b; /* Line 7 */
   }
   public static void main (String args[])
   {
      Test t = new Test();  /* Line 11 */
      Bar newBar = t.doBar();  /* Line 12 */
      System.out.println("newBar");
      newBar = new Bar(); /* Line 14 */
      System.out.println("finishing"); /* Line 15 */
   }
}
```

At what point is the Bar object, created on line 6, eligible for garbage collection?

a) after line 12

b) after line 14

c) after line 7, when doBar() completes
d) after line 15, when main() completes

Answer: Option B

Explanation:

Option B is correct. All references to the Bar object created on line 6 are destroyed when a new reference to a new Bar object is assigned to the variable newBar on line 14. Therefore the Bar object, created on line 6, is eligible for garbage collection after line 14.

Option A is wrong. This actually protects the object from garbage collection.

Option C is wrong. Because the reference in the doBar() method is returned on line 7 and is stored in newBar on line 12. This preserver the object created on line 6.

Option D is wrong. Not applicable because the object is eligible for garbage collection after line 14.

4.

```
class Test
{
   private Demo d;
   void start()
   {
      d = new Demo();
      this.takeDemo(d); /* Line 7 */
   } /* Line 8 */
   void takeDemo(Demo demo)
   {
      demo = null;
      demo = new Demo();
   }
}
```

When is the Demo object eligible for garbage collection?

a) After line 7
b) After line 8
c) After the start() method completes
d) When the instance running this code is made eligible for garbage collection.

Answer: Option D

Explanation:
Option D is correct. By a process of elimination.

Option A is wrong. The variable d is a member of the Test class and is never directly set to null.

Option B is wrong. A copy of the variable d is set to null and not the actual variable d.

Option C is wrong. The variable d exists outside the start() method (it is a class member). So, when the start() method finishes the variable d still holds a reference.

5.

```
public class X
{
   public static void main(String [] args)
   {
      X x = new X();
      X x2 = m1(x); /* Line 6 */
      X x4 = new X();
      x2 = x4; /* Line 8 */
      doComplexStuff();
   }
   static X m1(X mx)
   {
      mx = new X();
      return mx;
   }
}
```

After line 8 runs. how many objects are eligible for garbage collection?

a) 0
b) 1
c) 2
d) 3

Answer: Option B
Explanation:
By the time line 8 has run, the only object without a reference is the one generated as a result of line 6. Remember that "Java is pass by value," so the reference variable x is not affected by the m1() method.

Ref: http://www.javaworld.com/javaworld/javaqa/2000-05/03-qa-0526-pass.html

Ref: http://www.javaworld.com/javaworld/javaqa/2000-05/03-qa-0526-pass.html

6.

```
public Object m()
{
    Object o = new Float(3.14F);
    Object [] oa = new Object[l];
    oa[0] = o; /* Line 5 */
    o = null;  /* Line 6 */
    oa[0] = null; /* Line 7 */
    return o; /* Line 8 */
}
```

When is the Float object, created in line 3, eligible for garbage collection?

a) just after line 5
b) just after line 6
c) just after line 7
d) just after line 8

Answer: Option C

Explanation:

Option A is wrong. This simply copies the object reference into the array.

Option B is wrong. The reference o is set to null, but, oa[0] still maintains the reference to the Float object.

Option C is correct. The thread of execution will then not have access to the object.

7.

```
class X2
{
   public X2 x;
   public static void main(String [] args)
   {
      X2 x2 = new X2();  /* Line 6 */
      X2 x3 = new X2();  /* Line 7 */
      x2.x = x3;
      x3.x = x2;
      x2 = new X2();
      x3 = x2; /* Line 11 */
      doComplexStuff();
   }
}
```

after line 11 runs, how many objects are eligible for garbage collection?

a) 0
b) 1
c) 2
d) 3

Answer: Option C

Explanation:

This is an example of the islands of isolated objects. By the time line 11 has run, the objects instantiated in lines 6 and 7 are referring to each other, but no live thread can reach either of them.

8.

What allows the programmer to destroy an object x?

a) x.delete()
b) x.finalize()
c) Runtime.getRuntime().gc()
d) Only the garbage collection system can destroy an object.

Answer: Option D

Explanation:

Option D is correct. When an object is no longer referenced, it may be reclaimed by the garbage collector. If an object declares a finalizer, the finalizer is executed before the object is reclaimed to give the object a last chance to clean up resources that would not otherwise be released. When a class is no longer needed, it may be unloaded.

Option A is wrong. I found 4 delete() methods in all of the Java class structure. They are:

1. delete() - Method in class java.io.File : Deletes the file or directory denoted by this abstract pathname.
2. delete(int, int) - Method in class java.lang.StringBuffer : Removes the characters in a substring of this StringBuffer.
3. delete(int, int) - Method in interface javax.accessibility.AccessibleEditableText : Deletes the text between two indices
4. delete(int, int) - Method in class : javax.swing.text.JTextComponent.AccessibleJTextComponent; Deletes the text between two indices

None of these destroy the object to which they belong.

Option B is wrong. I found 19 finalize() methods. The most interesting, from this questions point of view, was the finalize() method in class java.lang.Object which is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. This method does not destroy the object to which it belongs.

Option C is wrong. But it is interesting. The Runtime class has many methods, two of which are:

1. getRuntime() - Returns the runtime object associated with the current Java application.
2. gc() - Runs the garbage collector. Calling this method suggests that the Java virtual machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for quick reuse. When control returns from the method call, the virtual machine has made its best effort to recycle all discarded objects. Interesting as this is, it doesn't destroy the object.