1. Which are true? (Choose all that apply.)

A. “X extends Y” is correct if and only if X is a class and Y is an interface

B. “X extends Y” is correct if and only if X is an interface and Y is a class

C. “X extends Y” is correct if X and Y are either both classes or both interfaces

D. “X extends Y” is correct for all combinations of X and Y being classes and/or interfaces

1. ☑ **C** is correct.

☒ **A** is incorrect because classes implement interfaces, they don’t extend them. **B** is incorrect because interfaces only “inherit from” other interfaces. **D** is incorrect based on the preceding rules. (OCA Objective 7.5)

2. Given:

```
class Rocket {
  private void blastOff() { System.out.print("bang "); }
}
public class Shuttle extends Rocket {
  public static void main(String[] args) {
    new Shuttle().go();
  }
  void go() {
    blastOff();
    // Rocket.blastOff();  // line A
  }
  private void blastOff() { System.out.print("sh-bang "); }
}
```

Which are true? (Choose all that apply.)

A. As the code stands, the output is `bang`

B. As the code stands, the output is `sh-bang`

C. As the code stands, compilation fails

D. If line A is uncommented, the output is `bang bang`

E. If line A is uncommented, the output is `sh-bang bang`

F. If line A is uncommented, compilation fails.

2. ☑ **B** and **F** are correct. Since `Rocket.blastOff()` is `private`, it can't be overridden, and it is invisible to class `Shuttle`.
☒ **A, C, D,** and **E** are incorrect based on the above. (OCA Objective 6.4)

3. Given that the `for` loop's syntax is correct, and given:

```
import static java.lang.System.*;
class _ {
  static public void main(String[] __A_V_) {
    String $ = "";
    for(int x=0; ++x < __A_V_.length; )   // for loop
      $ += __A_V_[x];
    out.println($);
  }
}
```

And the command line:

```
java _ - A .
```

What is the result?

A. `-A`

B. `A.`

C. `-A.`

3. ☑ **B** is correct. This question is using valid (but inappropriate and weird) identifiers, static imports, `main()`, and pre-incrementing logic. (Note: You might get a compiler warning when compiling this code.)
☒ **A, C, D, E, F,** and **G** are incorrect based on the above. (OCA Objective 1.2)

4. Given:

```
1. enum Animals {
2.   DOG("woof"), CAT("meow"), FISH("burble");
3.   String sound;
4.   Animals(String s) { sound = s; }
5. }
6. class TestEnum {
7.   static Animals a;
8.   public static void main(String[] args) {
9.     System.out.println(a.DOG.sound + " " + a.FISH.sound);
10.  }
11. }
```

What is the result?

A. `woof burble`

B. Multiple compilation errors

C. Compilation fails due to an error on line 2

D. Compilation fails due to an error on line 3

E. Compilation fails due to an error on line 4

F. Compilation fails due to an error on line 9

4. ☑ **A** is correct; `enums` can have constructors and variables.
☒ **B, C, D, E,** and **F** are incorrect; these lines all use correct syntax. (OCA Objective 1.2)

5. Given two files:

```
1. package pkgA;
2. public class Foo {
3. int a = 5;
4. protected int b = 6;
5. public int c = 7;
6. }
```

```
3. package pkgB;
4. import pkgA.*;
5. public class Baz {
6.   public static void main(String[] args) {
7.     Foo f = new Foo();
8.     System.out.print(" " + f.a);
9.     System.out.print(" " + f.b);
10.    System.out.println(" " + f.c);
11.  }
12. }
```

What is the result? (Choose all that apply.)

A. `5 6 7`

B. `5` followed by an exception

C. Compilation fails with an error on line 7

D. Compilation fails with an error on line 8

E. Compilation fails with an error on line 9

F. Compilation fails with an error on line 10

5. ☑ **D** and **E** are correct. Variable `a` has default access, so it cannot be accessed from outside the package. Variable `b` has protected access in `pkgA`.
☒ **A, B, C,** and **F** are incorrect based on the above information. (OCA Objectives 1.4 and 6.5)

6. Given:

```
1. public class Electronic implements Device
     { public void doIt() { } }
2.
3. abstract class Phone1 extends Electronic { }
4.
5. abstract class Phone2 extends Electronic
     { public void doIt(int x) { } }
6.
7. class Phone3 extends Electronic implements Device
     { public void doStuff() { } }
8.
9. interface Device { public void doIt(); }
```

What is the result? (Choose all that apply.)

A. Compilation succeeds

B. Compilation fails with an error on line 1

C. Compilation fails with an error on line 3

D. Compilation fails with an error on line 5

E. Compilation fails with an error on line 7

F. Compilation fails with an error on line 9

6. ☑ **A** is correct; all of these are legal declarations.
☒ **B, C, D, E,** and **F** are incorrect based on the above information. (OCA Objective 7.5)

7. Given:

```
4. class Announce {
5.   public static void main(String[] args) {
6.     for(int __x = 0; __x < 3; __x++) ;
7.     int #lb = 7;
8.     long [] x [5];
9.     Boolean []ba[];
10.  }
11. }
```

What is the result? (Choose all that apply.)

A. Compilation succeeds

B. Compilation fails with an error on line 6

C. Compilation fails with an error on line 7

D. Compilation fails with an error on line 8

E. Compilation fails with an error on line 9

7. ☑ **C** and **D** are correct. Variable names cannot begin with a #, and an array declaration can't include a size without an instantiation. The rest of the code is valid.
☒ **A, B,** and **E** are incorrect based on the above. (OCA Objective 2.1)

8. Given:

```
3. public class TestDays {
4.   public enum Days { MON, TUE, WED };
5.   public static void main(String[] args) {
6.     for(Days d : Days.values() )
7.       ;
8.     Days [] d2 = Days.values();
9.     System.out.println(d2[2]);
10.  }
11. }
```

What is the result? (Choose all that apply.)

A. `TUE`

B. `WED`

C. The output is unpredictable

D. Compilation fails due to an error on line 4

E. Compilation fails due to an error on line 6

F. Compilation fails due to an error on line 8

G. Compilation fails due to an error on line 9

8. ☑ **B** is correct. Every `enum` comes with a `static values()` method that returns an array of the `enum`'s values in the order in which they are declared in the `enum`.
☒ **A, C, D, E, F,** and **G** are incorrect based on the above information. (OCP Objective 1.2)

9. Given:

```
4. public class Frodo extends Hobbit {
5.   public static void main(String[] args) {
6.     int myGold = 7;
7.     System.out.println(countGold(myGold, 6));
8.   }
9. }
10. class Hobbit {
11.   int countGold(int x, int y) { return x + y; }
12. }
```

What is the result?

A. 13

B. Compilation fails due to multiple errors

C. Compilation fails due to an error on line 6

D. Compilation fails due to an error on line 7

E. Compilation fails due to an error on line 11

9. ☑ **D** is correct. The `countGold()` method cannot be invoked from a static context.
☒ **A, B, C,** and **E** are incorrect based on the above information. (OCA Objective 6.2)

10. Given:

```
interface Gadget {
  void doStuff();
}
abstract class Electronic {
  void getPower() { System.out.print("plug in "); }
}
public class Tablet extends Electronic implements Gadget {
  void doStuff() { System.out.print("show book "); }
  public static void main(String[] args) {
    new Tablet().getPower();
    new Tablet().doStuff();
  }
}
```

Which are true? (Choose all that apply.)

A. The class `Tablet` will NOT compile

B. The interface `Gadget` will NOT compile

C. The output will be `plug in show book`

D. The `abstract` class `Electronic` will NOT compile

E. The class `Tablet` CANNOT both extend and implement

10. ☑ **A** is correct. By default, an interface's methods are `public` so the `Tablet.doStuff` method must be public, too. The rest of the code is valid.
☒ **B, C, D,** and **E** are incorrect based on the above. (OCA Objective 7.5)

11. Given that the Integer class is in the java.lang package and given:

```
1. // insert code here
2. class StatTest {
3.   public static void main(String[] args) {
4.     System.out.println(Integer.MAX_VALUE);
5.   }
6. }
```

Which, inserted independently at line 1, compiles? (Choose all that apply.)

A. `import static java.lang;`

B. `import static java.lang.Integer;`

C. `import static java.lang.Integer.*;`

D. `static import java.lang.Integer.*;`

E. `import static java.lang.Integer.MAX_VALUE;`

F. None of the above statements are valid import syntax

11. ☑ **C** and **E** are correct syntax for static imports. Line 4 isn't making use of `static import` so the code will also compile with none of the imports.
☒ **A, B, D,** and **F** are incorrect based on the above. (OCA Objective 1.4)

12. Given:

```
interface MyInterface {
  // insert code here
}
```

Which lines of code—inserted independently at `insert code here`—will compile? (Choose all that apply.)

A. `public static m1() {;}`

B. `default void m2() {;}`

C. `abstract int m3();`

D. `final short m4() {return 5;}`

E. `default long m5();`

F. `static void m6() {;}`

12. ☑ **B, C,** and **F** are correct. As of Java 8, interfaces can have `default` and `static` methods ☒ **A, D,** and **E** are incorrect. **A** has no return type; **D** cannot have a method body; and **E** needs a method body. (OCA Objective 7.5)

13. Which are true? (Choose all that apply.)

A. Java is a dynamically typed programming language

B. Java provides fine-grained control of memory through the use of pointers

C. Java provides programmers the ability to create objects that are well encapsulated

D. Java provides programmers the ability to send Java objects from one machine to another

E. Java is an implementation of the ECMA standard

F. Java's encapsulation capabilities provide its primary security mechanism

13. ☑ **C** and **D** are correct.

☒ **A** is incorrect because Java is a statically typed language. **B** is incorrect because it does not provide pointers. **E** is incorrect because JavaScript is an implementation of the ECMA standard, not Java. **F** is incorrect because the use of bytecode and the JVM provide Java's primary security mechanisms.

1. Given:

```
public abstract interface Frobnicate { public void twiddle(String s); }
```

Which is a correct class? (Choose all that apply.)

A.
```
public abstract class Frob implements Frobnicate {
  public abstract void twiddle(String s) { }
}
```
B.
```
public abstract class Frob implements Frobnicate { }
```
C.
```
public class Frob extends Frobnicate {
  public void twiddle(Integer i) { }
}
```
D.
```
public class Frob implements Frobnicate {
  public void twiddle(Integer i) { }
}
```
E.
```
public class Frob implements Frobnicate {
  public void twiddle(String i) { }
  public void twiddle(Integer s) { }
}
```

1. ☑ B and E are correct. B is correct because an `abstract` class need not implement any or all of an interface's methods. E is correct because the class implements the interface method and additionally overloads the `twiddle()` method.
☒ A, C, and D are incorrect. A is incorrect because `abstract` methods have no body. C is incorrect because classes implement interfaces; they don't extend them. D is incorrect because overloading a method is not implementing it. (OCA Objectives 7.1 and 7.5)

2. Given:

```
class Top {
  public Top(String s) { System.out.print("B"); }
}
public class Bottom2 extends Top {
  public Bottom2(String s) { System.out.print("D"); }
  public static void main(String [] args) {
    new Bottom2("C");
    System.out.println(" ");
  }
}
```

What is the result?

A. `BD`

B. `DB`

C. `BDC`

D. `DBC`

E. Compilation fails

2. ☑ E is correct. The implied `super()` call in `Bottom2`'s constructor cannot be satisfied because there is no no-arg constructor in `Top`. A default, no-arg constructor is generated by the compiler only if the class has no constructor defined explicitly.
☒ A, B, C, and D are incorrect based on the above. (OCA Objective 6.3)

3. Given:

```
class Clidder {
  private final void flipper() { System.out.println("Clidder"); }
}
public class Clidlet extends Clidder {
  public final void flipper() { System.out.println("Clidlet");  }
  public static void main(String [] args) {
    new Clidlet().flipper();
  }
}
```

What is the result?

A. `Clidlet`

B. `Clidder`

C.
```
Clidder
Clidlet
```

D.
```
Clidlet
Clidder
```

E. Compilation fails

Special Note: The next question crudely simulates a style of question known as "drag-and-drop." Up through the SCJP 6 exam, drag-and-drop questions were included on the exam. As of spring 2014, Oracle DOES NOT include any drag-and-drop questions on its Java exams, but just in case Oracle's policy changes, we left a few in the book.

3. ☑ A is correct. Although a `final` method cannot be overridden, in this case, the method is private and, therefore, hidden. The effect is that a new, accessible, method flipper is created. Therefore, no polymorphism occurs in this example, the method invoked is simply that of the child class, and no error occurs.
☒ B, C, D, and E are incorrect based on the preceding. (OCA Objective 7.2)

Special Note: This next question crudely simulates a style of question known as "drag-and-drop." Up through the SCJP 6 exam, drag-and-drop questions were included on the exam. As of spring 2014, Oracle DOES NOT include any drag-and-drop questions on its Java exams, but just in case Oracle's policy changes, we left a few in the book.

4. Using the fragments below, complete the following code so it compiles. Note that you may not have to fill in all of the slots.

Code:

```
class AgedP {
  __________ __________ __________ __________ __________
  public AgedP(int x) {
    __________ __________ __________ __________ __________
  }
}
public class Kinder extends AgedP {
  __________ __________ __________ _________ ________ __________
  public Kinder(int x) {
    __________ __________ __________ __________ __________ ();
  }
}
```

Fragments: Use the following fragments zero or more times:

AgedP	super	this	
(	)	{	}
;			

4. Here is the answer:

```
class AgedP {
  AgedP() {}
  public AgedP(int x) {
  }
}
public class Kinder extends AgedP {
  public Kinder(int x) {
    super();
  }
}
```

As there is no droppable tile for the variable `x` and the parentheses (in the `Kinder` constructor) are already in place and empty, there is no way to construct a call to the superclass constructor that takes an argument. Therefore, the only remaining possibility is to

created by the compiler because another constructor is already present. (OCA Objectives 6.3 and 7.4) Note: As you can see, many questions test for OCA Objective 7.1, we're going to stop mentioning objective 7.1.

5. Given:

```
class Bird {
  { System.out.print("b1 "); }
  public Bird() { System.out.print("b2 "); }
}
class Raptor extends Bird {
  static { System.out.print("r1 "); }
  public Raptor() { System.out.print("r2 "); }
  { System.out.print("r3 "); }
  static { System.out.print("r4 "); }
}
class Hawk extends Raptor {
  public static void main(String[] args) {
    System.out.print("pre ");
    new Hawk();
    System.out.println("hawk ");
  }
}
```

What is the result?

A. `pre b1 b2 r3 r2 hawk`

B. `pre b2 b1 r2 r3 hawk`

C. `pre b2 b1 r2 r3 hawk r1 r4`

D. `r1 r4 pre b1 b2 r3 r2 hawk`

E. `r1 r4 pre b2 b1 r2 r3 hawk`

F. `pre r1 r4 b1 b2 r3 r2 hawk`

G. `pre r1 r4 b2 b1 r2 r3 hawk`

H. The order of output cannot be predicted

I. Compilation fails

Note: You'll probably never see this many choices on the real exam!

5. ☑ D is correct. Static `init` blocks are executed at class loading time; instance `init` blocks run right after the call to `super()` in a constructor. When multiple `init` blocks of a single type occur in a class, they run in order, from the top down.
☒ A, B, C, E, F, G, H, and I are incorrect based on the above. Note: You'll probably never see this many choices on the real exam! (OCA Objective 6.3)

6. Given the following:

```
1. class X { void do1() { } }
2. class Y extends X { void do2() { } }
3.
4. class Chrome {
5.   public static void main(String [] args) {
6.     X x1 = new X();
 7.     X x2 = new Y();
 8.     Y y1 = new Y();
 9.     // insert code here
10. } }
```

Which of the following, inserted at line 9, will compile? (Choose all that apply.)

A. `x2.do2();`

B. `(Y)x2.do2();`

C. `((Y)x2).do2();`

D. None of the above statements will compile

6. ☑ C is correct. Before you can invoke `Y`'s `do2` method, you have to cast `x2` to be of type `Y`.
☒ A, B, and D are incorrect based on the preceding. B looks like a proper cast, but without the second set of parentheses, the compiler thinks it's an incomplete statement. (OCA Objective 7.3)

7. Given:

```
public class Locomotive {
  Locomotive() { main("hi"); }

  public static void main(String[] args) {
    System.out.print("2 ");
  }
  public static void main(String args) {
    System.out.print("3 " + args);
  }
}
```

What is the result? (Choose all that apply.)

A. `2` **will be included in the output**

B. `3` **will be included in the output**

C. `hi` **will be included in the output**

D. Compilation fails

E. An exception is thrown at runtime

7. ☑ A is correct. It's legal to overload `main()`. Since no instances of `Locomotive` are created, the constructor does not run and the overloaded version of `main()` does not run. ☒ B, C, D, and E are incorrect based on the preceding. (OCA Objectives 1.3 and 6.3)

8. Given:

```
3. class Dog {
4.   public void bark() { System.out.print("woof "); }
5. }
6. class Hound extends Dog {
7.   public void sniff() { System.out.print("sniff "); }
8.   public void bark() { System.out.print("howl "); }
9. }
10. public class DogShow {
11.   public static void main(String[] args) { new DogShow().go(); }
12.   void go() {
13.     new Hound().bark();
14.     ((Dog) new Hound()).bark();
15.     ((Dog) new Hound()).sniff();
16.   }
17. }
```

What is the result? (Choose all that apply.)

A. `howl howl sniff`

B. `howl woof sniff`

C. `howl howl` **followed by an exception**

D. `howl woof` **followed by an exception**

E. Compilation fails with an error at line 14

F. Compilation fails with an error at line 15

8. ☑ F is correct. Class `Dog` doesn't have a `sniff` method.
☒ A, B, C, D, and E are incorrect based on the above information. (OCA Objectives 7.2 and 7.3)

9. Given:

```
 3. public class Redwood extends Tree {
 4.   public static void main(String[] args) {
 5.     new Redwood().go();
 6.   }
 7.   void go() {
 8.     go2(new Tree(), new Redwood());
 9.     go2((Redwood) new Tree(), new Redwood());
10.   }
11.   void go2(Tree t1, Redwood r1) {
12.      Redwood r2 = (Redwood)t1;
13.      Tree t2 = (Tree)r1;
14.   }
15. }
16. class Tree { }
```

What is the result? (Choose all that apply.)

A. An exception is thrown at runtime

B. The code compiles and runs with no output

C. Compilation fails with an error at line 8

D. Compilation fails with an error at line 9

E. Compilation fails with an error at line 12

F. Compilation fails with an error at line 13

9. ☑ A is correct. A `ClassCastException` will be thrown when the code attempts to downcast a `Tree` to a `Redwood`.
☒ B, C, D, E, and F are incorrect based on the above information. (OCA Objective 7.3)

10. Given:

```
3. public class Tenor extends Singer {
4.   public static String sing() { return "fa"; }
5.   public static void main(String[] args) {
6.     Tenor t = new Tenor();
7.     Singer s = new Tenor();
8.     System.out.println(t.sing() + " " + s.sing());
9.   }
10. }
11. class Singer { public static String sing() { return "la"; } }
```

What is the result?

A. `fa fa`

B. `fa la`

C. `la la`

D. Compilation fails

E. An exception is thrown at runtime

10. ☑ **B is correct. The code is correct, but polymorphism doesn't apply to `static` methods.**
☒ **A, C, D, and E are incorrect based on the above information. (OCA Objectives 6.2 and 7.2)**

11. Given:

```
3. class Alpha {
4.   static String s = " ";
5.   protected Alpha() { s += "alpha "; }
6. }
7. class SubAlpha extends Alpha {
8.   private SubAlpha() { s += "sub "; }
9. }
10. public class SubSubAlpha extends Alpha {
11.   private SubSubAlpha() { s += "subsub "; }
12.   public static void main(String[] args) {
13.     new SubSubAlpha();
14.     System.out.println(s);
15.   }
16. }
```

What is the result?

A. `subsub`

B. `sub subsub`

C. `alpha subsub`

D. `alpha sub subsub`

E. Compilation fails

F. An exception is thrown at runtime

11. ☑ C is correct. Watch out, because `SubSubAlpha` extends `Alpha`! Because the code doesn't attempt to make a `SubAlpha`, the private constructor in `SubAlpha` is okay.
☒ A, B, D, E, and F are incorrect based on the above information. (OCA Objectives 6.3 and 7.2)

12. Given:

```
3. class Alpha {
4.   static String s = " ";
5.   protected Alpha() { s += "alpha "; }
6. }
7. class SubAlpha extends Alpha {
8.   private SubAlpha() { s += "sub "; }
9. }
10. public class SubSubAlpha extends Alpha {
11.   private SubSubAlpha() { s += "subsub "; }
12.   public static void main(String[] args) {
13.     new SubSubAlpha();
14.     System.out.println(s);
15.   }
16. }
```

What is the result?

A. h hn x

B. hn x h

C. b h hn x

D. b hn x h

E. bn x h hn x

F. b bn x h hn x

G. bn x b h hn x

H. Compilation fails

12. ☑ C is correct. Remember that constructors call their superclass constructors, which execute first, and that constructors can be overloaded.
☒ A, B, D, E, F, G, and H are incorrect based on the above information. (OCA Objectives 6.3 and 7.4)

13. Given:

```
3. class Mammal {
4.   String name = "furry ";
5.   String makeNoise() { return "generic noise"; }
6. }
7. class Zebra extends Mammal {
8.   String name = "stripes ";
9.   String makeNoise() { return "bray"; }
10. }
11. public class ZooKeeper {
12.   public static void main(String[] args) { new ZooKeeper().go(); }
13.   void go() {
14.     Mammal m = new Zebra();
15.     System.out.println(m.name + m.makeNoise());
16.   }
17. }
```

What is the result?

A. `furry bray`

B. `stripes bray`

C. `furry generic noise`

D. `stripes generic noise`

E. Compilation fails

F. An exception is thrown at runtime

13. ☑ A is correct. Polymorphism is only for instance methods, not instance variables. ☒ B, C, D, E, and F are incorrect based on the above information. (OCA Objective 6.3)

14. Given:

```
1. interface FrogBoilable {
2.   static int getCtoF(int cTemp) {
3.     return (cTemp * 9 / 5) + 32;
4.   }
5.   default String hop() { return "hopping "; }
6. }
7. public class DontBoilFrogs implements FrogBoilable {
8.   public static void main(String[] args) {
9.     new DontBoilFrogs().go();
10.  }
11.  void go() {
12.    System.out.print(hop());
13.    System.out.println(getCtoF(100));
14.    System.out.println(FrogBoilable.getCtoF(100));
15.    DontBoilFrogs dbf = new DontBoilFrogs();
16.    System.out.println(dbf.getCtoF(100));
17.  }
18. }
```

What is the result? (Choose all that apply.)

A. `hopping 212`

B. Compilation fails due to an error on line 2

C. Compilation fails due to an error on line 5

D. Compilation fails due to an error on line 12

E. Compilation fails

F. An exception is thrown at runtime

14. ☑ E and G are correct. Neither of these lines of code uses the correct syntax to invoke an interface's static method.
☒ A, B, C, D, and F are incorrect based on the above information. (OCP Objectives 6.2 and 7.5)

15. Given:

```
interface I1 {
  default int doStuff() { return 1; }
}
interface I2 {
  default int doStuff() { return 2; }
}
public class MultiInt implements I1, I2 {
  public static void main(String[] args) {
    new MultiInt().go();
  }
  void go() {
    System.out.println(doStuff());
  }
  int doStuff() {
    return 3;
  }
}
```

What is the result?

A. 1

B. 2

C. 3

D. The output is unpredictable

E. Compilation fails

F. An exception is thrown at runtime

15. ☑ E is correct. This is kind of a trick question; the implementing method must be marked `public`. If it was, all the other code is legal, and the output would be 3. If you understood all the multiple inheritance rules and just missed the access modifier, give yourself half credit.

☒ A, B, C, D, and F are incorrect based on the above information. (OCP Objective 7.5)

16. Given:

```
interface MyInterface {
  default int doStuff() {
    return 42;
  }
}
public class IfaceTest implements MyInterface {
  public static void main(String[] args) {
    new IfaceTest().go();
  }
  void go() {
    // INSERT CODE HERE
  }
  public int doStuff() {
    return 43;
  }
}
```

Which line(s) of code, inserted independently at // INSERT CODE HERE, will allow the code to compile? (Choose all that apply.)

A. `System.out.println("class: " + doStuff());`

B. `System.out.println("iface: " + super.doStuff());`

C. `System.out.println("iface: " + MyInterface.super.doStuff());`

D. `System.out.println("iface: " + MyInterface.doStuff());`

E. `System.out.println("iface: " + super.MyInterface.doStuff());`

F. None of the lines, A–E will allow the code to compile

16. ☑ A and C are correct. A uses correct syntax to invoke the class's method, and C uses the correct syntax to invoke the interface's overloaded `default` method.
☒ B, D, E, and F are incorrect. (OCP Objective 7.5)

1. Given:

```
class CardBoard {
  Short story = 200;
  CardBoard go(CardBoard cb) {
    cb = null;
    return cb;
  }
  public static void main(String[] args) {
    CardBoard c1 = new CardBoard();
    CardBoard c2 = new CardBoard();
    CardBoard c3 = c1.go(c2);
    c1 = null;
    // do Stuff
} }
```

When `// do Stuff` is reached, how many objects are eligible for garbage collection?

A. 0

B. 1

C. 2

D. Compilation fails

E. It is not possible to know

F. An exception is thrown at runtime

1. ☑ **C** is correct. Only one `CardBoard object (c1)` is eligible, but it has an associated `Short` wrapper object that is also eligible.
☒ **A, B, D, E,** and **F** are incorrect based on the above. (OCA Objective 2.4)

2. Given:

```
public class Fishing {
  byte b1 = 4;
  int i1 = 123456;
  long L1 = (long)i1;       // line A
  short s2 = (short)i1;     // line B
  byte b2 = (byte)i1;       // line C
  int i2 = (int)123.456;    // line D
  byte b3 = b1 + 7;         // line E
}
```

Which lines WILL NOT compile? (Choose all that apply.)

A. Line A

B. Line B

C. Line C

D. Line D

E. Line E

2. ☑ **E** is correct; compilation of line E fails. When a mathematical operation is performed on any primitives smaller than `int`s, the result is automatically cast to an integer.
☒ **A, B, C,** and **D** are all legal primitive casts. (OCA Objective 2.1)

3. Given:

```
public class Literally {
  public static void main(String[] args) {
    int i1 = 1_000;       // line A
    int i2 = 10_00;       // line B
    int i3 = _10_000;     // line C
    int i4 = 0b101010;    // line D
    int i5 = 0B10_1010;   // line E
    int i6 = 0x2_a;       // line F
  }
}
```

Which lines WILL NOT compile? (Choose all that apply.)

A. Line A

B. Line B

C. Line C

D. Line D

E. Line E

F. Line F

3. ☑ **C** is correct; line **C** will NOT compile. As of Java 7, underscores can be included in numeric literals, but not at the beginning or the end.
☒ **A, B, D, E,** and **F** are incorrect. **A** and **B** are legal numeric literals. **D** and **E** are examples of valid binary literals, which were new to Java 7, and **F** is a valid hexadecimal literal that uses an underscore. (OCA Objective 2.1)

4. Given:

```
class Mixer {
  Mixer() { }
  Mixer(Mixer m) { m1 = m; }
  Mixer m1;
  public static void main(String[] args) {
    Mixer m2 = new Mixer();
    Mixer m3 = new Mixer(m2);  m3.go();
    Mixer m4 = m3.m1;          m4.go();
    Mixer m5 = m2.m1;          m5.go();
  }
  void go() { System.out.print("hi "); }
}
```

What is the result?

A. `hi`

B. `hi hi`

C. `hi hi hi`

D. Compilation fails

E. `hi`, followed by an exception

F. `hi hi`, followed by an exception

4. ☑ **F** is correct. The `m2` object's `m1` instance variable is never initialized, so when `m5` tries to use it, a `NullPointerException` is thrown.
☒ **A, B, C, D,** and **E** are incorrect based on the above. (OCA Objectives 2.1 and 2.3)

5. Given:

```
class Fizz {
  int x = 5;
  public static void main(String[] args) {
    final Fizz f1 = new Fizz();
    Fizz f2 = new Fizz();
    Fizz f3 = FizzSwitch(f1,f2);
    System.out.println((f1 == f3) + " " + (f1.x == f3.x));
  }
  static Fizz FizzSwitch(Fizz x, Fizz y) {
    final Fizz z = x;
    z.x = 6;
    return z;
} }
```

What is the result?

A. `true true`

B. `false true`

C. `true false`

D. `false false`

E. Compilation fails

F. An exception is thrown at runtime

5. ☑ **A** is correct. The references `f1`, `z`, and `f3` all refer to the same instance of `Fizz`. The `final` modifier assures that a reference variable cannot be referred to a different object, but `final` doesn't keep the object's state from changing.
☒ **B, C, D, E,** and **F** are incorrect based on the above. (OCA Objective 2.2)

6. Given:

```
public class Mirror {
  int size = 7;
  public static void main(String[] args) {
    Mirror m1 = new Mirror();
    Mirror m2 = m1;
    int i1 = 10;
    int i2 = i1;
    go(m2, i2);
    System.out.println(m1.size + " " + i1);
  }
  static void go(Mirror m, int i) {
    m.size = 8;
    i = 12;
  }
}
```

What is the result?

A. `7 10`

B. `8 10`

C. `7 12`

D. `8 12`

E. Compilation fails

F. An exception is thrown at runtime

6. ☑ **B** is correct. In the `go()` method, `m` refers to the single `Mirror` instance, but the `int i` is a new `int` variable, a detached copy of `i2`.
☒ **A, C, D, E,** and **F** are incorrect based on the above. (OCA Objectives 2.2 and 2.3)

7. Given:

```
public class Wind {
  int id;
  Wind(int i) { id = i; }
  public static void main(String[] args) {
    new Wind(3).go();
    // commented line
  }
  void go() {
    Wind w1 = new Wind(1);
    Wind w2 = new Wind(2);
    System.out.println(w1.id + " " + w2.id);
  }
}
```

When execution reaches the commented line, which are true? (Choose all that apply.)

A. The output contains `1`

B. The output contains `2`

C. The output contains `3`

D. Zero `Wind` objects are eligible for garbage collection

E. One `Wind` object is eligible for garbage collection

F. Two `Wind` objects are eligible for garbage collection

G. Three `Wind` objects are eligible for garbage collection

7. ☑ **A**, **B,** and **G** are correct. The constructor sets the value of `id` for `w1` and `w2`. When the commented line is reached, none of the three `Wind` objects can be accessed, so they are eligible to be garbage collected.
☒ **C, D, E,** and **F** are incorrect based on the above. (OCA Objectives 1.1, 2.3, and 2.4)

8. Given:

```
3. public class Ouch {
4.   static int ouch = 7;
5.   public static void main(String[] args) {
6.     new Ouch().go(ouch);
7.     System.out.print(" " + ouch);
8.   }
9.   void go(int ouch) {
10.    ouch++;
11.    for(int ouch = 3; ouch < 6; ouch++)
12.      ;
13.    System.out.print(" " + ouch);
14.  }
15. }
```

What is the result?

A. 5 7

B. 5 8

C. 8 7

D. 8 8

E. Compilation fails

F. An exception is thrown at runtime

8. ☑ **E** is correct. The parameter declared on line 9 is valid (although ugly), but the variable name `ouch` cannot be declared again on line 11 in the same scope as the declaration on line 9.
☒ **A, B, C, D,** and **F** are incorrect based on the above. (OCA Objectives 1.1 and 2.1)

9. Given:

```
public class Happy {
  int id;
  Happy(int i) { id = i; }
  public static void main(String[] args) {
    Happy h1 = new Happy(1);
    Happy h2 = h1.go(h1);
    System.out.println(h2.id);
  }
  Happy go(Happy h) {
    Happy h3 = h;
    h3.id = 2;
    h1.id = 3;
    return h1;
  }
}
```

What is the result?

A. `1`

B. `2`

C. `3`

D. Compilation fails

E. An exception is thrown at runtime

9. ☑ **D** is correct. Inside the `go()` method, `h1` is out of scope.
☒ **A, B, C,** and **E** are incorrect based on the above. (OCA Objectives 1.1 and 6.1)

10. Given:

```
public class Network {
  Network(int x, Network n) {
    id = x;
    p = this;
    if(n != null) p = n;
  }
  int id;
  Network p;
  public static void main(String[] args) {
    Network n1 = new Network(1, null);
    n1.go(n1);
  }
  void go(Network n1) {
    Network n2 = new Network(2, n1);
    Network n3 = new Network(3, n2);
    System.out.println(n3.p.p.id);
  }
}
```

What is the result?

A. `1`

B. `2`

C. `3`

D. `null`

E. Compilation fails

10. ☑ **A** is correct. Three `Network` objects are created. The `n2` object has a reference to the `n1` object, and the `n3` object has a reference to the `n2` object. The S.O.P. can be read as, "Use the `n3` object's `Network` reference (the first `p`), to find that object's reference (`n2`), and use that object's reference (the second `p`) to find that object's (`n1`'s) `id`, and print that `id`."
☒ **B, C, D,** and **E** are incorrect based on the above. (OCA Objectives, 2.2, 2.3, and 6.4)

11. Given:

```
3. class Beta { }
4. class Alpha {
5.   static Beta b1;
6.   Beta b2;
7. }
8. public class Tester {
9.   public static void main(String[] args) {
10.    Beta b1 = new Beta();     Beta b2 = new Beta();
11.    Alpha a1 = new Alpha();   Alpha a2 = new Alpha();
12.    a1.b1 = b1;
13.    a1.b2 = b1;
14.    a2.b2 = b2;
15.    a1 = null;  b1 = null;  b2 = null;
16.    // do stuff
17.  }
18. }
```

When line 16 is reached, how many objects will be eligible for garbage collection?

A. 0

B. 1

C. 2

D. 3

E. 4

F. 5

11. ☑ **B** is correct. It should be clear that there is still a reference to the object referred to by `a2`, and that there is still a reference to the object referred to by `a2.b2`. What might be less clear is that you can still access the other `Beta` object through the static variable `a2.b1`—because it's static.
☒ **A, C, D, E,** and **F** are incorrect based on the above. (OCA Objective 2.4)

12. Given:

```
public class Telescope {
  static int magnify = 2;
  public static void main(String[] args) {
    go();
  }
  static void go() {
    int magnify = 3;
    zoomIn();
  }
  static void zoomIn() {
    magnify *= 5;
    zoomMore(magnify);
    System.out.println(magnify);
  }
  static void zoomMore(int magnify) {
    magnify *= 7;
  }
}
```

What is the result?

A. `2`

B. `10`

C. `15`

D. `30`

E. `70`

F. `105`

G. Compilation fails

12. ☑ **B** is correct. In the `Telescope` class, there are three different variables named `magnify`. The `go()` method's version and the `zoomMore()` method's version are not used in the `zoomIn()` method. The `zoomIn()` method multiplies the class variable `* 5`. The result (`10`) is sent to `zoomMore()`, but what happens in `zoomMore()` stays in `zoomMore()`. The S.O.P. prints the value of `zoomIn()`'s `magnify`.
☒ **A, C, D, E, F,** and **G** are incorrect based on the above. (OCA Objectives 1.1 and 6.6)

13. Given:

```
3. public class Dark {
4.   int x = 3;
5.   public static void main(String[] args) {
6.     new Dark().go1();
7.   }
8.   void go1() {
9.     int x;
10.    go2(++x);
11.  }
12.  void go2(int y) {
13.    int x = ++y;
14.    System.out.println(x);
15.  }
16. }
```

What is the result?

A. 2

B. 3

C. 4

D. 5

E. Compilation fails

F. An exception is thrown at runtime

13. ☑ **E** is correct. In `go1()` the local variable `x` is not initialized.
☒ **A, B, C, D,** and **F** are incorrect based on the above. (OCA Objectives 2.1 and 2.3)

1. Given:

```
class Hexy {
  public static void main(String[] args) {
    int i = 42;
    String s = (i<40)?"life":(i>50)?"universe":"everything";
    System.out.println(s);
  }
}
```

What is the result?

A. `null`

B. `life`

C. `universe`

D. `everything`

E. Compilation fails

F. An exception is thrown at runtime

1. ☑ **D** is correct. This is a ternary nested in a ternary. Both ternary expressions are false.
☒ **A, B, C, E,** and **F** are incorrect based on the above. (OCA Objective 3.1 and 3.3)

2. Given:

```
public class Dog {
  String name;
  Dog(String s) { name = s; }
  public static void main(String[] args) {
    Dog d1 = new Dog("Boi");
    Dog d2 = new Dog("Tyri");
    System.out.print((d1 == d2) + " ");
    Dog d3 = new Dog("Boi");
    d2 = d1;
    System.out.print((d1 == d2) + " ");
    System.out.print((d1 == d3) + " ");
  }
}
```

What is the result?

A. `true true true`

B. `true true false`

C. `false true false`

D. `false true true`

E. `false false false`

F. An exception will be thrown at runtime

2. ☑ **C** is correct. The == operator tests for reference variable equality, not object equality. ☒ **A, B, D, E,** and **F** are incorrect based on the above. (OCA Objectives 3.1 and 3.2)

3. Given:

```
class Fork {
  public static void main(String[] args) {
    if(args.length == 1 | args[1].equals("test")) {
      System.out.println("test case");
    } else {
      System.out.println("production " + args[0]);
    }
  }
}
```

And the command-line invocation:

```
java Fork live2
```

What is the result?

A. `test case`

B. `production live2`

C. `test case live2`

D. Compilation fails

E. An exception is thrown at runtime

3. ☑ **E** is correct. Because the short-circuit (`||`) is not used, both operands are evaluated. Since `args[1]` is past the `args` array bounds, an `ArrayIndexOutOfBoundsException` is thrown.
☒ **A, B, C,** and **D** are incorrect based on the above. (OCA Objectives 3.1 and 3.2)

4. Given:

```
class Feline {
  public static void main(String[] args) {
    long x = 42L;
    long y = 44L;
    System.out.print(" " + 7 + 2 + " ");
    System.out.print(foo() + x + 5 + " ");
    System.out.println(x + y + foo());
  }
  static String foo() { return "foo"; }
}
```

What is the result?

A. `9 foo47 86foo`

B. `9 foo47 4244foo`

C. `9 foo425 86foo`

D. `9 foo425 4244foo`

E. `72 foo47 86foo`

F. `72 foo47 4244foo`

G. `72 foo425 86foo`

H. `72 foo425 4244foo`

I. Compilation fails

4. ☑ **G** is correct. Concatenation runs from left to right, and if either operand is a `String`, the operands are concatenated. If both operands are numbers, they are added together.
☒ **A, B, C, D, E, F, H,** and **I** are incorrect based on the above. (OCA Objective 3.1)

5. **Note**: Here's another old-style drag-and-drop question…just in case.

Place the fragments into the code to produce the output `33`. Note that you must use each fragment exactly once.

```
CODE:
class Incr {
  public static void main(String[] args) {
    Integer x = 7;
    int y = 2;

    x    ___  ___;
    ___  ___  ___;
    ___  ___  ___;
    ___  ___  ___;

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

FRAGMENTS:

```
y    y    y    y

y    x    x

-=   *=   *=   *=
```

5. Answer:

```
class Incr {
  public static void main(String[] args) {
    Integer x = 7;
    int y = 2;

    x *= x;
    y *= y;
    y *= y;
    x -= y;

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

Yeah, we know it's kind of puzzle-y, but you might encounter something like it on the real exam if Oracle reinstates this type of question. (OCA Objective 3.1)

6. Given:

```
public class Cowboys {
  public static void main(String[] args) {
    int x = 12;
    int a = 5;
    int b = 7;
    System.out.println(x/a + " " + x/b);
  }
}
```

What is the result? (Choose all that apply.)

A. `2 1`

B. `2 2`

C. `3 1`

D. `3 2`

E. An exception is thrown at runtime

6. ☑ **A** is correct. When dividing `int`s, remainders are always rounded down.
☒ **B, C, D,** and **E** are incorrect based on the above. (OCA Objective 3.1)

7. Given:

```
3. public class McGee {
4.   public static void main(String[] args) {
5.     Days d1 = Days.TH;
6.     Days d2 = Days.M;
7.     for(Days d: Days.values()) {
8.       if(d.equals(Days.F)) break;
9.       d2 = d;
10.    }
11.    System.out.println((d1 == d2)?"same old" : "newly new");
12.  }
13.  enum Days {M, T, W, TH, F, SA, SU};
14. }
```

What is the result?

A. `same old`

B. `newly new`

C. Compilation fails due to multiple errors

D. Compilation fails due only to an error on line 7

E. Compilation fails due only to an error on line 8

F. Compilation fails due only to an error on line 11

G. Compilation fails due only to an error on line 13

7. ☑ **A** is correct. All this syntax is correct. The for-each iterates through the `enum` using the `values()` method to return an array. An `enum` can be compared using either `equals()` or `==`. An `enum` can be used in a ternary operator's boolean test.
☒ **B**, **C**, **D**, **E**, **F**, and **G** are incorrect based on the above. (OCA Objectives 3.1, 3.2, and 3.3)

8. Given:

```
4. public class SpecialOps {
5.   public static void main(String[] args) {
6.     String s = "";
7.     boolean b1 = true;
8.     boolean b2 = false;
9.     if((b2 = false) | (21%5) > 2) s += "x";
10.    if(b1 || (b2 = true))         s += "y";
11.    if(b2 == true)                s += "z";
12.    System.out.println(s);
13.  }
14. }
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. `x` will be included in the output

C. `y` will be included in the output

D. `z` will be included in the output

E. An exception is thrown at runtime

8. ☑ **C** is correct. Line 9 uses the modulus operator, which returns the remainder of the division, which in this case is `1`. Also, line 9 sets `b2` to `false`, and it doesn't test `b2`'s value. Line 10 would set `b2` to `true`; however, the short-circuit operator keeps the expression `b2 = true` from being executed.
☒ **A, B, D,** and **E** are incorrect based on the above. (OCA Objectives 3.1, and 3.2)

9. Given:

```
3. public class Spock {
4.   public static void main(String[] args) {
5.     int mask = 0;
6.     int count = 0;
7.     if( ((5<7) || (++count < 10)) | mask++ < 10 )    mask = mask + 1;
8.     if(  (6 > 8) ^ false)                            mask = mask + 10;
9.     if( !(mask > 1) && ++count > 1)                  mask = mask + 100;
10.    System.out.println(mask + " " + count);
11.  }
12. }
```

Which two are true about the value of mask and the value of count at line 10? (Choose two.)

A. `mask` is 0

B. `mask` is 1

C. `mask` is 2

D. `mask` is 10

E. `mask` is greater than 10

F. `count` is 0

G. `count` is greater than 0

9. ☑ **C** and **F** are correct. At line 7 the `||` keeps `count` from being incremented, but the `|` allows `mask` to be incremented. At line 8 the `^` returns true only if exactly one operand is true. At line 9 `mask` is `2` and the `&&` keeps `count` from being incremented.
☒ **A, B, D, E,** and **G** are incorrect based on the above. (OCA Objective 3.1)

10. Given:

```
3. interface Vessel { }
4. interface Toy { }
5. class Boat implements Vessel { }
6. class Speedboat extends Boat implements Toy { }
7. public class Tree {
8.   public static void main(String[] args) {
9.     String s = "0";
10.    Boat b = new Boat();
11.    Boat b2 = new Speedboat();
12.    Speedboat s2 = new Speedboat();
13.    if((b instanceof Vessel) && (b2 instanceof Toy))  s += "1";
14.    if((s2 instanceof Vessel) && (s2 instanceof Toy)) s += "2";
15.    System.out.println(s);
16.  }
17. }
```

What is the result?

A. `0`

B. `01`

C. `02`

D. `012`

E. Compilation fails

F. An exception is thrown at runtime

10. ☑ **D** is correct. First, remember that `instanceof` can look up through multiple levels of an inheritance tree. Also remember that `instanceof` is commonly used before attempting a downcast; so in this case, after line 15, it would be possible to say `Speedboat s3 = (Speedboat)b2;`.
☒ **A, B, C, E,** and **F** are incorrect based on the above. (OCA Objective 3.1)

11. Given:

```
10. boolean b1 = false;
11. boolean b2;
12. int x = 2, y = 5;
13. b1 = 2-12/4 > 5+-7 && b1 != y++>5 == 7%4 > ++x | b1 == true;
14. b2 = (2-12/4 > 5+-7) && (b1 != y++>5) == (7%4 > ++x) | (b1 == true);
15. System.out.println(b1 + " " + b2);
```

What is the result? (This is a tricky one. If you want a hint, go take another look at the operator precedence rant in the chapter.)

A. `true true`

B. `false true`

C. `true false`

D. `false false`

E. Compilation fails

F. An exception is thrown at runtime

11. ☑ **A** is correct. We're pretty sure you won't encounter anything as horrible as this on the real exam. But if you got this one correct, pat yourself on the back! The way to tackle a problem like this is to evaluate the expression in stages. In this case you might solve it like so:

1. Given that `toLowerCase()` is an aptly named `String` method that returns a `String`, and given the code:

```
public class Flipper {
  public static void main(String[] args) {
    String o = "-";
    switch("RED".toLowerCase()) {
      case "yellow":
        o += "y";
      case "red":
        o += "r";
      case "green":
        o += "g";
    }
    System.out.println(o);
  }
}
```

What is the result?

A. `-`

B. `-r`

C. `-rg`

D. Compilation fails

E. An exception is thrown at runtime

1. ☑ **C** is correct. As of Java 7 it's legal to switch on a String, and remember that switches use "entry point" logic.

☒ **A, B, D,** and **E** are incorrect based on the above. (OCA Objective 3.4)

2. Given:

```
class Plane {
  static String s = "-";
  public static void main(String[] args) {
    new Plane().s1();
    System.out.println(s);
  }
  void s1() {
    try { s2(); }
    catch (Exception e) { s += "c"; }
  }
  void s2() throws Exception  {
    s3();  s += "2";
    s3();  s += "2b";
  }
  void s3() throws Exception {
    throw new Exception();
  }
}
```

What is the result?

A. -

B. `-c`

C. `-c2`

D. `-2c`

E. `-c22b`

F. `-2c2b`

G. `-2c2bc`

H. Compilation fails

2. ☑ **B** is correct. Once `s3()` throws the exception to `s2()`, `s2()` throws it to `s1()`, and no more of `s2()`'s code will be executed.

☒ **A, C, D, E, F, G,** and **H** are incorrect based on the above. (OCA Objectives 8.2 and 8.4)

3. Given:

```
try { int x = Integer.parseInt(“two”); }
```

Which could be used to create an appropriate `catch` block? (Choose all that apply.)

A. `ClassCastException`

B. `IllegalStateException`

C. `NumberFormatException`

D. `IllegalArgumentException`

E. `ExceptionInInitializerError`

F. `ArrayIndexOutOfBoundsException`

3. ☑ **C** and **D** are correct. `Integer.parseInt` can throw a `NumberFormatException`, and `IllegalArgumentException` is its superclass (that is, a broader exception).

☒ **A, B, E,** and **F** are not in `NumberFormatException`’s class hierarchy. (OCA Objective 8.5)

4. Given:

```
public class Flip2 {
  public static void main(String[] args) {
    String o = "-";
    String[] sa = new String[4];
    for(int i = 0; i < args.length; i++)
      sa[i] = args[i];
    for(String n: sa) {
      switch(n.toLowerCase()) {
        case "yellow": o += "y";
        case "red":    o += "r";
        case "green":  o += "g";
      }
    }
    System.out.print(o);
  }
}
```

And given the command-line invocation:

```
Java Flip2 RED Green YeLLow
```

Which are true? (Choose all that apply.)

A. The string `rgy` will appear somewhere in the output

B. The string `rgg` will appear somewhere in the output

C. The string `gyr` will appear somewhere in the output

D. Compilation fails

E. An exception is thrown at runtime

4. ☑ **E** is correct. As of Java 7 the syntax is legal. The `sa[]` array receives only three arguments from the command line, so on the last iteration through `sa[]`, a `NullPointerException` is thrown.

☒ **A, B, C,** and **D** are incorrect based on the above. (OCA Objectives 1.3, 5.2, and 8.5)

5. Given:

```
1. class Loopy {
2.   public static void main(String[] args) {
3.     int[] x = {7,6,5,4,3,2,1};
4.     // insert code here
5.       System.out.print(y + " ");
6.     }
7.   }
8. }
```

Which, inserted independently at line 4, compiles? (Choose all that apply.)

A. `for(int y : x) {`

B. `for(x : int y) {`

C. `int y = 0; for(y : x) {`

D. `for(int y=0, z=0; z<x.length; z++) { y = x[z];`

E. `for(int y=0, int z=0; z<x.length; z++) { y = x[z];`

F. `int y = 0; for(int z=0; z<x.length; z++) { y = x[z];`

5. ☑ **A**, **D,** and **F** are correct. **A** is an example of the enhanced `for` loop. **D** and **F** are examples of the basic `for` loop.

☒ **B, C,** and **E** are incorrect. **B** is incorrect because its operands are swapped. **C** is incorrect because the enhanced `for` must declare its first operand. **E** is incorrect syntax to declare two variables in a `for` statement. (OCA Objective 5.2)

6. Given:

```
class Emu {
  static String s = "-";
  public static void main(String[] args) {
    try {
      throw new Exception();
    } catch (Exception e) {
        try {
          try { throw new Exception();
          } catch (Exception ex) { s += "ic "; }
          throw new Exception(); }
        catch (Exception x) { s += "mc "; }
        finally { s += "mf "; }
    } finally { s += "of "; }
    System.out.println(s);
} }
```

What is the result?

A. `-ic of`

B. `-mf of`

C. `-mc mf`

D. `-ic mf of`

E. `-ic mc mf of`

F. `-ic mc of mf`

G. Compilation fails

6. ☑ **E** is correct. There is no problem nesting `try/catch` blocks. As is normal, when an exception is thrown, the code in the `catch` block runs, and then the code in the `finally` block runs.

☒ **A, B, C, D,** and **F** are incorrect based on the above. (OCA Objectives 8.2 and 8.4)

7. Given:

```
3. class SubException extends Exception { }
4. class SubSubException extends SubException { }
5.
6. public class CC { void doStuff() throws SubException { } }
7.
8. class CC2 extends CC { void doStuff() throws SubSubException { } }
9.
10. class CC3 extends CC { void doStuff() throws Exception { } }
11.
12. class CC4 extends CC { void doStuff(int x) throws Exception { } }
13.
14. class CC5 extends CC {  void doStuff()  { } }
```

What is the result? (Choose all that apply.)

A. Compilation succeeds

B. Compilation fails due to an error on line 8

C. Compilation fails due to an error on line 10

D. Compilation fails due to an error on line 12

E. Compilation fails due to an error on line 14

7. ☑ **C** is correct. An overriding method cannot throw a broader exception than the method it's overriding. Class CC4's method is an overload, not an override.

☒ **A, B, D,** and **E** are incorrect based on the above. (OCA Objectives 8.2 and 8.4)

8. Given:

```
 3. public class Ebb {
 4.   static int x = 7;
 5.   public static void main(String[] args) {
 6.     String s = "";
 7.     for(int y = 0; y < 3; y++) {
 8.       x++;
 9.       switch(x) {
10.         case 8: s += "8 ";
11.         case 9: s += "9 ";
12.         case 10: { s+= "10 "; break; }
13.         default: s += "d ";
14.         case 13: s+= "13 ";
15.       }
16.     }
17.     System.out.println(s);
18.   }
19.   static { x++; }
20. }
```

What is the result?

A. `9 10 d`

B. `8 9 10 d`

C. `9 10 10 d`

D. `9 10 10 d 13`

E. `8 9 10 10 d 13`

F. `8 9 10 9 10 10 d 13`

G. Compilation fails

8. ☑ **D** is correct. Did you catch the static initializer block? Remember that switches work on "fall-through" logic and that fall-through logic also applies to the default case, which is used when no other case matches.

☒ **A, B, C, E, F,** and **G** are incorrect based on the above. (OCA Objective 3.4)

9. Given:

```
3. class Infinity { }
4. public class Beyond extends Infinity {
5.   static Integer i;
6.   public static void main(String[] args) {
7.     int sw = (int)(Math.random() * 3);
8.     switch(sw) {
9.       case 0: {  for(int x = 10; x > 5; x++)
10.                   if(x > 10000000) x = 10;
11.                 break; }
12.      case 1: {  int y = 7 * i;  break;  }
13.      case 2: {  Infinity inf = new Beyond();
14.                 Beyond b = (Beyond)inf;  }
15.    }
16.  }
17. }
```

And given that line 7 will assign the value 0, 1, or 2 to `sw`, which are true? (Choos apply.)

A. Compilation fails

B. A `ClassCastException` might be thrown

C. A `StackOverflowError` might be thrown

D. A `NullPointerException` might be thrown

E. An `IllegalStateException` might be thrown

F. The program might hang without ever completing

G. The program will always complete without exception

9. ☑ **D** and **F** are correct. Because `i` was not initialized, case 1 will throw a `NullPointerException`. Case 0 will initiate an endless loop, not a stack overflow. Case 2's downcast will *not* cause an exception.

☒ **A, B, C, E,** and **G** are incorrect based on the above. (OCA Objectives 3.4 and 8.5)

10. Given:

```
3. public class Circles {
4.   public static void main(String[] args) {
5.     int[] ia = {1,3,5,7,9};
6.     for(int x : ia) {
7.       for(int j = 0; j < 3; j++) {
8.         if(x > 4 && x < 8) continue;
9.         System.out.print(" " + x);
10.        if(j == 1) break;
11.        continue;
12.      }
13.      continue;
14.    }
15.  }
16. }
```

What is the result?

A. `1 3 9`

B. `5 5 7 7`

C. `1 3 3 9 9`

D. `1 1 3 3 9 9`

E. `1 1 1 3 3 3 9 9 9`

F. Compilation fails

10. ☑ **D** is correct. The basic rule for unlabeled `continue` statements is that the current iteration stops early and execution jumps to the next iteration. The last two `continue` statements are redundant!

☒ **A, B, C, E,** and **F** are incorrect based on the above. (OCA Objectives 5.2 and 5.5)

11. Given:

```
3. public class OverAndOver {
4.   static String s = "";
5.   public static void main(String[] args) {
6.     try {
7.       s += "1";
8.       throw new Exception();
9.     } catch (Exception e) { s += "2";
10.    } finally { s += "3"; doStuff(); s += "4";
11.    }
12.    System.out.println(s);
13.  }
14.  static void doStuff() { int x = 0; int y = 7/x; }
15. }
```

What is the result?

A. `12`

B. `13`

C. `123`

D. `1234`

E. Compilation fails

F. `123` followed by an exception

G. `1234` followed by an exception

H. An exception is thrown with no other output

11. ☑ **H** is correct. It's true that the value of `String` `s` is `123` at the time that the divide-by-zero exception is thrown, but `finally()` is *not* guaranteed to complete, and in this case `finally()` never completes, so the `System.out.println` (S.O.P) never executes.

☒ **A, B, C, D, E, F,** and **G** are incorrect based on the above. (OCA Objectives 8.2 and 8.5)

12. Given:

```
3. public class Wind {
4.   public static void main(String[] args) {
5.     foreach:
6.     for(int j=0; j<5; j++) {
7.       for(int k=0; k< 3; k++) {
8.         System.out.print(" " + j);
9.         if(j==3 && k==1) break foreach;
10.        if(j==0 || j==2) break;
11.      }
12.    }
13.  }
14. }
```

What is the result?

A. `0 1 2 3`

B. `1 1 1 3 3`

C. `0 1 1 1 2 3 3`

D. `1 1 1 3 3 4 4 4`

E. `0 1 1 1 2 3 3 4 4 4`

F. Compilation fails

12. ☑ **C** is correct. A `break` breaks out of the current innermost loop and carries on. A labeled `break` breaks out of and terminates the labeled loops.

☒ **A, B, D, E,** and **F** are incorrect based on the above. (OCA Objectives 5.2 and 5.5)

13. Given:

```
3. public class Gotcha {
4.   public static void main(String[] args) {
5.     // insert code here
6.
7.   }
8.   void go() {
9.     go();
10.  }
11. }
```

And given the following three code fragments:

```
I.   new Gotcha().go();

II.  try { new Gotcha().go(); }
     catch (Error e) { System.out.println("ouch"); }

III. try { new Gotcha().go(); }
     catch (Exception e) { System.out.println("ouch"); }
```

When fragments I–III are added, independently, at line 5, which are true? (Choose all that apply.)

A. Some will not compile

B. They will all compile

C. All will complete normally

D. None will complete normally

E. Only one will complete normally

F. Two of them will complete normally

13. ☑ **B** and **E** are correct. First off, `go()` is a badly designed recursive method, guaranteed to cause a `StackOverflowError`. Since `Exception` is not a superclass of `Error`, catching an `Exception` will not help handle an `Error`, so fragment III will not complete normally. Only fragment II will catch the `Error`.

☒ **A, C, D,** and **F** are incorrect based on the above. (OCA Objectives 8.1, 8.2, and 8.4)

14. Given the code snippet:

```
String s = "bob";
String[] sa = {"a", "bob"};
final String s2 = "bob";
StringBuilder sb = new StringBuilder("bob");

// switch(sa[1]) {          // line 1
// switch("b" + "ob") {     // line 2
// switch(sb.toString()) {  // line 3

// case "ann":  ;           // line 4
// case s:      ;           // line 5
// case s2:     ;           // line 6
}
```

And given that the numbered lines will all be tested by uncommenting one `switch` statement and one `case` statement together, which line(s) will FAIL to compile? (Choose all that apply.)

A. line 1

B. line 2

C. line 3

D. line 4

E. line 5

F. line 6

G. All six lines of code will compile

14. ☑ **E** is correct. A `switch`'s `cases` must be compile-time constants or `enum` values.

☒ **A, B, C, D, F,** and **G** are incorrect based on the above. (OCA Objective 3.4)

15. Given that `IOException` is in the `java.io` package and given:

```
1. public class Frisbee {
2.   // insert code here
3.     int x = 0;
4.     System.out.println(7/x);
5.   }
6. }
```

And given the following four code fragments:

```
I.   public static void main(String[] args) {
II.  public static void main(String[] args) throws Exception {
III. public static void main(String[] args) throws IOException {
IV.  public static void main(String[] args) throws RuntimeException {
```

If the four fragments are inserted independently at line 2, which are true? (Choose all that apply.)

A. All four will compile and execute without exception

B. All four will compile and execute and throw an exception

C. Some, but not all, will compile and execute without exception

D. Some, but not all, will compile and execute and throw an exception

E. When considering fragments II, III, and IV, of those that will compile, adding a `try/catch` block around line 4 will cause compilation to fail

15. ☑ **D** is correct. This is kind of sneaky, but remember that we're trying to toughen you up for the real exam. If you're going to throw an `IOException`, you have to import the java.io package or declare the exception with a fully qualified name.

☒ **A, B, C,** and **E** are incorrect. **A, B,** and **C** are incorrect based on the above. **E** is incorrect because it's okay both to handle and declare an exception. (OCA Objectives 8.2 and 8.5)

16. Given:

```
2. class MyException extends Exception { }
3. class Tire {
4.   void doStuff() {  }
5. }
6. public class Retread extends Tire {
7.   public static void main(String[] args) {
8.     new Retread().doStuff();
9.   }
10.  // insert code here
11.    System.out.println(7/0);
12.  }
13. }
```

And given the following four code fragments:

```
I.   void doStuff() {
II.  void doStuff() throws MyException {
III. void doStuff() throws RuntimeException {
IV.  void doStuff() throws ArithmeticException {
```

When fragments I–IV are added, independently, at line 10, which are true? (Choose all that apply.)

A. None will compile

B. They will all compile

C. Some, but not all, will compile

D. All those that compile will throw an exception at runtime

E. None of those that compile will throw an exception at runtime

F. Only some of those that compile will throw an exception at runtime

16. ☑ **C** and **D** are correct. An overriding method cannot throw checked exceptions that are broader than those thrown by the overridden method. However, an overriding method *can* throw `RuntimeExceptions` not thrown by the overridden method.

☒ **A, B, E,** and **F** are incorrect based on the above. (OCA Objective 8.1)

1. Given:

```
public class Mutant {
  public static void main(String[] args) {
    StringBuilder sb = new StringBuilder("abc");
    String s = "abc";
    sb.reverse().append("d");
    s.toUpperCase().concat("d");
    System.out.println("." + sb + ". ." + s + ".");
  }
}
```

Which two substrings will be included in the result? (Choose two.)

A. `.abc.`

B. `.ABCd.`

C. `.ABCD.`

D. `.cbad.`

E. `.dcba.`

1. ☑ **A** and **D** are correct. The `String` operations are working on a new (lost) `String` not `String s`. The `StringBuilder` operations work from left to right.

☒ **B**, **C**, and **E** are incorrect based on the above. (OCA Objectives 9.2 and 9.1)

2. Given:

```
public class Hilltop {
  public static void main(String[] args) {
    String[] horses = new String[5];
    horses[4] = null;
    for(int i = 0; i < horses.length; i++) {
      if(i < args.length)
        horses[i] = args[i];
      System.out.print(horses[i].toUpperCase() + " ");
    }
  }
}
```

And, if the code compiles, the command line:

```
java Hilltop eyra vafi draumur kara
```

What is the result?

A. `EYRA VAFI DRAUMUR KARA`

B. `EYRA VAFI DRAUMUR KARA null`

C. An exception is thrown with no other output

D. `EYRA VAFI DRAUMUR KARA`, and then a `NullPointerException`

E. `EYRA VAFI DRAUMUR KARA`, and then an `ArrayIndexOutOfBoundsException`

F. Compilation fails

2. ☑ **D** is correct. The `horses` array's first four elements contain `Strings`, but the fifth is null, so the `toUpperCase()` invocation for the fifth element throws a `NullPointerException`.

☒ **A, B, C, E,** and **F** are incorrect based on the above. (OCA Objectives 4.1 and 1.3)

3. Given:

```
public class Actors {
  public static void main(String[] args) {
    char[] ca = {0x4e, \u004e, 78};
    System.out.println((ca[0] == ca[1]) + " " + (ca[0] == ca[2]));
  }
}
```

What is the result?

A. `true true`

B. `true false`

C. `false true`

D. `false false`

E. Compilation fails

3. ☑ **E** is correct. The Unicode declaration must be enclosed in single quotes: `'\u004e'`. If this were done, the answer would be **A**, but that equality isn't on the OCA exam.

☒ **A, B, C,** and **D** are incorrect based on the above. (OCA Objectives 2.1 and 4.1)

4. Given:

```
1. class Dims {
2.   public static void main(String[] args) {
3.     int[][] a = {{1,2}, {3,4}};
4.     int[] b = (int[]) a[1];
5.     Object o1 = a;
6.     int[][] a2 = (int[][]) o1;
7.     int[] b2 = (int[]) o1;
8.     System.out.println(b[1]);
9. } }
```

What is the result? (Choose all that apply.)

A. 2

B. 4

C. An exception is thrown at runtime

D. Compilation fails due to an error on line 4

E. Compilation fails due to an error on line 5

F. Compilation fails due to an error on line 6

G. Compilation fails due to an error on line 7

4. ☑ **C** is correct. A `ClassCastException` is thrown at line 7 because `o1` refers to an `int[][]`, not an `int[]`. If line 7 were removed, the output would be 4.

☒ **A, B, D, E, F,** and **G** are incorrect based on the above. (OCA Objective 4.2)

5. Given:

```
import java.util.*;
public class Sequence {
  public static void main(String[] args) {
    ArrayList<String> myList = new ArrayList<String>();
    myList.add("apple");
    myList.add("carrot");
    myList.add("banana");
    myList.add(1, "plum");
    System.out.print(myList);
  }
}
```

What is the result?

A. `[apple, banana, carrot, plum]`

B. `[apple, plum, carrot, banana]`

C. `[apple, plum, banana, carrot]`

D. `[plum, banana, carrot, apple]`

E. `[plum, apple, carrot, banana]`

F. `[banana, plum, carrot, apple]`

G. Compilation fails

5. ☑ **B** is correct. `ArrayList` elements are automatically inserted in the order of entry; they are not automatically sorted. `ArrayLists` use zero-based indexes, and the last `add()` inserts a new element and shifts the remaining elements back.

☒ **A, C, D, E, F,** and **G** are incorrect based on the above. (OCA Objective 9.4)

6. Given:

```
3. class Dozens {
4.   int[] dz = {1,2,3,4,5,6,7,8,9,10,11,12};
5. }
6. public class Eggs {
7.   public static void main(String[] args) {
8.     Dozens [] da = new Dozens[3];
9.     da[0] = new Dozens();
10.    Dozens d = new Dozens();
11.    da[1] = d;
12.    d = null;
13.    da[1] = null;
14.    // do stuff
15.  }
16. }
```

Which two are true about the objects created within `main()`, and which are eligible for garbage collection when line 14 is reached?

A. Three objects were created

B. Four objects were created

C. Five objects were created

D. Zero objects are eligible for GC

E. One object is eligible for GC

F. Two objects are eligible for GC

G. Three objects are eligible for GC

6. ☑ **C** and **F** are correct. `da` refers to an object of type "`Dozens` array," and each `Dozens` object that is created comes with its own "`int` array" object. When line 14 is reached, only the second `Dozens` object (and its "`int` array" object) are not reachable.

☒ **A, B, D, E,** and **G** are incorrect based on the above. (OCA Objectives 4.1 and 2.4)

7. Given:

```
public class Tailor {
  public static void main(String[] args) {
    byte[][] ba = {{1,2,3,4}, {1,2,3}};
    System.out.println(ba[1].length + " " + ba.length);
  }
}
```

What is the result?

A. `2 4`

B. `2 7`

C. `3 2`

D. `3 7`

E. `4 2`

F. `4 7`

G. Compilation fails

7. ☑ **C** is correct. A two-dimensional array is an "array of arrays." The length of `ba` is 2 because it contains 2 one-dimensional arrays. Array indexes are zero-based, so `ba[1]` refers to `ba`'s second array.

☒ **A, B, D, E, F,** and **G** are incorrect based on the above. (OCA Objective 4.2)

8. Given:

```
3. public class Theory {
4.   public static void main(String[] args) {
5.     String s1 = "abc";
6.     String s2 = s1;
7.     s1 += "d";
8.     System.out.println(s1 + " " + s2 + " " + (s1==s2));
9.
10.    StringBuilder sb1 = new StringBuilder("abc");
11.    StringBuilder sb2 = sb1;
12.    sb1.append("d");
13.    System.out.println(sb1 + " " + sb2 + " " + (sb1==sb2));
14.  }
15. }
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. The first line of output is `abc abc true`

C. The first line of output is `abc abc false`

D. The first line of output is `abcd abc false`

E. The second line of output is `abcd abc false`

F. The second line of output is `abcd abcd true`

G. The second line of output is `abcd abcd false`

8. ☑ **D** and **F** are correct. Although `String` objects are immutable, references to `Strings` are mutable. The code `s1 += "d";` creates a new `String` object. `StringBuilder` objects are mutable, so the `append()` is changing the single `StringBuilder` object to which both `StringBuilder` references refer.

☒ **A, B, C, E,** and **G** are incorrect based on the above. (OCA Objectives 9.2 and 9.1)

9. Given:

```
public class Mounds {
 public static void main(String[] args) {
   StringBuilder sb = new StringBuilder();
   String s = new String();
   for(int i = 0; i < 1000; i++) {
     s = " " + i;
     sb.append(s);
   }
   // done with loop
 }
}
```

If the garbage collector does NOT run while this code is executing, approximately how many objects will exist in memory when the loop is done?

A. Less than 10

B. About 1000

C. About 2000

D. About 3000

E. About 4000

9. ☑ **B** is correct. `StringBuilders` are mutable, so all of the `append()` invocations are acting on the same `StringBuilder` object over and over. `Strings`, however, are immutable, so every `String` concatenation operation results in a new `String` object. Also, the string `" "` is created once and reused in every loop iteration.

☒ **A, C, D,** and **E** are incorrect based on the above. (OCA Objectives 9.2 and 9.1)

10. Given:

```
3. class Box {
4.   int size;
5.   Box(int s) { size = s; }
6. }
7. public class Laser {
8.   public static void main(String[] args) {
9.     Box b1 = new Box(5);
10.    Box[] ba = go(b1, new Box(6));
11.    ba[0] = b1;
12.    for(Box b : ba) System.out.print(b.size + " ");
13.  }
14.  static Box[] go(Box b1, Box b2) {
15.     b1.size = 4;
16.     Box[] ma = {b2, b1};
17.     return ma;
18.  }
19. }
```

What is the result?

A. `4 4`

B. `5 4`

C. `6 4`

D. `4 5`

E. `5 5`

F. Compilation fails

10. ☑ **A** is correct. Although `main()`'s `b1` is a different reference variable than `go()`'s `b1`, they refer to the same `Box` object.

☒ **B, C, D, E,** and **F** are incorrect based on the above. (OCA Objectives 4.1, 6.1, and 6.6)

11. Given:

```
public class Hedges {
  public static void main(String[] args) {
    String s = "JAVA";
    s = s + "rocks";
    s = s.substring(4,8);
    s.toUpperCase();
    System.out.println(s);
  }
}
```

What is the result?

A. `JAVA`

B. `JAVAROCKS`

C. `rocks`

D. `rock`

E. `ROCKS`

F. `ROCK`

G. Compilation fails

11. ☑ **D** is correct. The `substring()` invocation uses a zero-based index and the second argument is exclusive, so the character at index 8 is NOT included. The `toUpperCase()` invocation makes a new `String` object that is instantly lost. The `toUpperCase()` invocation does NOT affect the `String` referred to by `s`.

☒ **A, B, C, E, F,** and **G** are incorrect based on the above. (OCA Objective 9.2)

12. Given:

```
1. import java.util.*;
2. class Fortress {
3.   private String name;
4.   private ArrayList<Integer> list;
5.   Fortress() { list = new ArrayList<Integer>(); }
6.
7.   String getName() { return name; }
8.   void addToList(int x) { list.add(x); }
9.   ArrayList getList() { return list; }
10. }
```

Which lines of code (if any) break encapsulation? (Choose all that apply.)

A. Line 3

B. Line 4

C. Line 5

D. Line 7

E. Line 8

F. Line 9

G. The class is already well encapsulated

12. ☑ **F** is correct. When encapsulating a mutable object like an `ArrayList`, your getter must return a reference to a copy of the object, not just the reference to the original object.

☒ **A, B, C, D, E,** and **G** are incorrect based on the above. (OCA Objective 6.5)

13. Given:

```
import java.util.function.Predicate;
public class Sheep {
  public static void main(String[] args) {
    Sheep s = new Sheep();
    s.go(() -> adder(5, 1) < 7);    // line A
    s.go(x -> adder(6, 2) < 9);     // line B
    s.go(x, y -> adder(3, 2) < 4);  // line C
  }
  void go(Predicate<Sheep> e) {
    Sheep s2 = new Sheep();
    if(e.test(s2))
      System.out.print("true ");
    else
      System.out.print("false ");
  }
  static int adder(int x, int y) {
    return x + y;
  }
}
```

What is the result?

A. `true true false`

B. Compilation fails due only to an error at line A

C. Compilation fails due only to an error at line B

D. Compilation fails due only to an error at line C

E. Compilation fails due only to errors at lines A and B

F. Compilation fails due only to errors at lines A and C

G. Compilation fails due only to errors at lines A, B, and C

H. Compilation fails for reasons not listed

13. ☑ **F** is correct. Predicate lambdas take exactly one parameter; the rest of the code is correct.

☒ **A, B, C, D, E, G,** and **H** are incorrect based on the above. (OCA Objective 9.5)

14. Given:

```
import java.time.*;
import java.time.format.*;
public class Shiny {
  public static void main(String[] args) {
    DateTimeFormatter f1 =
              DateTimeFormatter.ofPattern("MMM dd, yyyy");
    LocalDate d = LocalDate.of(2018, Month.JANUARY, 15);
    LocalDate d2 = d.plusDays(1);
    System.out.print(f1.format(d) + " ");
    System.out.println(d2.format(f1));
  }
}
```

What is the result?

A. `2018-01-15 2018-01-15`

B. `2018-01-15 2018-01-16`

C. `Jan 15, 2018 Jan 15, 2018`

D. `Jan 15, 2018 Jan 16, 2018`

E. Compilation fails

F. An exception is thrown at runtime

14. ☑ **D** is correct. Invoking the `plusDays()` method creates a new object, and both `LocalDate` and `DateTimeFormatter` have `format()` methods.

☒ **A, B, C, E,** and **F** are incorrect based on the above. (OCA Objective 9.3)

15. Given:

```
import java.util.*;
public class Jackets {
  public static void main(String[] args) {
    List<Integer> myList = new ArrayList<>();  // line 5
    myList.add(new Integer(5));
    myList.add(42);                            // line 7
    myList.add("113");                         // line 8
    myList.add(new Integer("7"));              // line 9
    System.out.println(myList);
  }
}
```

What is the result?

A. `[5, 42, 113, 7]`

B. Compilation fails due only to an error on line 5

C. Compilation fails due only to an error on line 8

D. Compilation fails due only to errors on lines 5 and 8

E. Compilation fails due only to errors on lines 7 and 8

F. Compilation fails due only to errors on lines 5, 7, and 8

G. Compilation fails due only to errors on lines 5, 7, 8, and 9

15. ☑ **C** is correct. The only error in this code is attempting to add a `String` to an `ArrayList` of `Integer` wrapper objects. Line 7 uses autoboxing, and lines 6 and 9 demonstrate using a wrapper class's two constructors.

☒ **A, B, D, E, F,** and **G** are incorrect based on the above. (OCA Objectives 2.5 and 9.4)

16. Given that `adder()` returns an `int`, which are valid `Predicate` lambdas? (Choose all that apply.)

A.
```
x, y -> 7 < 5
```

B.
```
x -> { return adder(2, 1) > 5; }
```

C.
```
x -> return adder(2, 1) > 5;
```

D.
```
x -> { int y = 5;
       int z = 7;
       adder(y, z) > 8; }
```

E.
```
x -> { int y = 5;
       int z = 7;
       return adder(y, z) > 8; }
```

F.
```
(MyClass x) -> 7 > 13
```

G.
```
(MyClass x) -> 5 + 4
```

16. ☑ **B, E** and **F** use correct syntax.

☒ **A, C, D**, and **G** are incorrect. **A** passes two parameters. **C**, a `return`, must be in a code block, and code blocks must be in curly braces. **D**, a block, must have a `return` statement. **G,** the result, is not a `boolean`. (OCA Objective 9.5)

17. Given:

```
import java.util.*;
public class Baking {
  public static void main(String[] args) {
    ArrayList<String> steps = new ArrayList<String>();
    steps.add("knead");
    steps.add("oil pan");
    steps.add("turn on oven");
    steps.add("roll");
    steps.add("turn on oven");
    steps.add("bake");
    System.out.println(steps);
  }
}
```

What is the result?

A. `[knead, oil pan, roll, turn on oven, bake]`

B. `[knead, oil pan, turn on oven, roll, bake]`

C. `[knead, oil pan, turn on oven, roll, turn on oven, bake]`

D. The output is unpredictable

E. Compilation fails

F. An exception is thrown at runtime

17. ☑ **C** is correct. `ArrayLists` can have duplicate entries.

☒ **A, B, D, E,** and **F** are incorrect based on the above. (OCA Objective 9.4)

18. Given:

```
import java.time.*;
public class Bachelor {
  public static void main(String[] args) {
    LocalDate d = LocalDate.of(2018, 8, 15);
    d = d.plusDays(1);
    LocalDate d2 = d.plusDays(1);
    LocalDate d3 = d2;
    d2 = d2.plusDays(1);
    System.out.println(d + " " + d2 + " " + d3);  // line X
  }
}
```

Which are true? (Choose all that apply.)

A. The output is: `2018-08-16 2018-08-17 2018-08-18`

B. The output is: `2018-08-16 2018-08-18 2018-08-17`

C. The output is: `2018-08-16 2018-08-17 2018-08-17`

D. At line X, zero `LocalDate` objects are eligible for garbage collection

E. At line X, one `LocalDate` object is eligible for garbage collection

F. At line X, two `LocalDate` objects are eligible for garbage collection

G. Compilation fails

18. ☑ **B** and **E** are correct. A total of four `LocalDate` objects are created, but the one created using the `of()` method is abandoned on the next line of code when its reference variable is assigned to the new `LocalDate` object created via the first `plusDays()` invocation. The reference variables are swapped a bit, which accounts for the dates not printing in chronological order.

☒ **A, C, D, F,** and **G** are incorrect based on the above. (OCA Objectives 2.4 and 9.3)

19. Given that `e` refers to an object that implements `Predicate`, which could be valid code snippets or statements? (Choose all that apply.)

A. `if(e.test(m))`

B. `switch (e.test(m))`

C. `while(e.test(m))`

D. `e.test(m) ? "yes" : "no";`

E. `do {} while(e.test(m));`

F. `System.out.print(e.test(m));`

G. `boolean b = e.test(m));`

19. ☑ **A, C, D, E, F,** and **G** are correct; they all require a `boolean`.

☒ **B** is incorrect. A `switch` doesn't take a `boolean`. (OCA Objective 9.5)