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)