SELF TEST ANSWERS

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) { }
}
```

Answer:

☑ **B** is correct, an `abstract` class need not implement any or all of an interface's methods. **E** is correct, the class implements the interface method and additionally overloads the `twiddle()` method.

☒ **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.
(Objective 5.4)

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

Answer:

☑ **E** is correct. The implied `super()` call in Bottom2's constructor cannot be satisfied because there isn't a 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. (Objective 1.6)

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

Answer:

☑ **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. (Objective 5.3)

4. Using the **fragments** below, complete the following **code** so it compiles. Note, you may not have to fill 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	
(	)	{	}
;			

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 create a call to the no-argument superclass constructor. This is done as: `super();`. The line cannot be left blank, as the parentheses are already in place. Further, since the superclass constructor called is the no-argument version, this constructor must be created. It will not be created by the compiler because there is another constructor already present.
(Objective 5.4)

5 Which statement(s) are true? (Choose all that apply.)

A. Cohesion is the OO principle most closely associated with hiding implementation details

B. Cohesion is the OO principle most closely associated with making sure that classes know about other classes only through their APIs

C. Cohesion is the OO principle most closely associated with making sure that a class is designed with a single, well-focused purpose

D. Cohesion is the OO principle most closely associated with allowing a single object to be seen as having many types

Answer:

☑ Answer **C** is correct.

☒ **A** refers to encapsulation, **B** refers to coupling, and **D** refers to polymorphism.
(Objective 5.1)

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.  }
11. }
```

Which, 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

Answer:

☑ **C** is correct. Before you can invoke `Y`'s `do2` method you have to cast `x2` to be of type `Y`. Statement **B** looks like a proper cast but without the second set of parentheses, the compiler thinks it's an incomplete statement.

☒ **A, B** and **D** are incorrect based on the preceding.
(Objective 5.2)

7. Given:

1. ClassA has a ClassD
2. Methods in ClassA use public methods in ClassB
3. Methods in ClassC use public methods in ClassA
4. Methods in ClassA use public variables in ClassB

Which is most likely true? (Choose the most likely.)

A. ClassD has low cohesion

B. ClassA has weak encapsulation

C. ClassB has weak encapsulation

D. ClassB has strong encapsulation

E. ClassC is tightly coupled to ClassA

Answer:

☑ **C** is correct. Generally speaking, public variables are a sign of weak encapsulation.

☒ **A, B, D,** and **E** are incorrect, because based on the information given, none of these statements can be supported.
(Objective 5.1)

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

Answer:

☑ **F** is correct. Class `Dog` doesn't have a `sniff` method.

☒ **A, B, C, D,** and **E** are incorrect based on the above information. (Objective 5.2)

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

Answer:

☑ **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. (Objective 5.2)

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

Answer:

☑ **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. (Objective 5.2)

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

Answer:

☑ **C** is correct. Watch out, `SubSubAlpha` extends `Alpha`! Since 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. (Objective 5.3)

12. Given:

```
3. class Building {
4.   Building() {  System.out.print("b ");  }
5.   Building(String name) {
6.     this();   System.out.print("bn " + name);
7.   }
8. }
9. public class House extends Building {
```

```
10.   House() {  System.out.print("h ");  }
11.   House(String name) {
12.     this();   System.out.print("hn " + name);
13.   }
14.   public static void main(String[] args) { new House("x "); }
15. }
```

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

Answer:

☑ **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. (Objectives 1.6, 5.4)

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

Answer:

☑ **A** is correct. Polymorphism is only for instance methods.

☒ **B, C, D, E,** and **F** are incorrect based on the above information. (Objectives 1.5, 5.4)

14. You're designing a new online board game in which Floozels are a type of Jammers, Jammers can have Quizels, Quizels are a type of Klakker, and Floozels can have several Floozets. Which of the following fragments represent this design? (Choose all that apply.)

A.
```
import java.util.*;
interface Klakker { }
class Jammer { Set<Quizel> q; }
class Quizel implements Klakker { }
public class Floozel extends Jammer { List<Floozet> f; }
interface Floozet { }
```

B.
```
import java.util.*;
class Klakker { Set<Quizel> q; }
class Quizel extends Klakker { }
class Jammer { List<Floozel> f; }
class Floozet extends Floozel { }
public class Floozel { Set<Klakker> k; }
```

C.
```
import java.util.*;
class Floozet { }
class Quizel implements Klakker { }
class Jammer { List<Quizel> q; }
interface Klakker { }
class Floozel extends Jammer { List<Floozet> f; }
```

D.
```
import java.util.*;
interface Jammer extends Quizel { }
interface Klakker { }
interface Quizel extends Klakker { }
interface Floozel extends Jammer, Floozet { }
interface Floozet { }
```

Answer:

☑ **A** and **C** are correct. The phrase "type of" indicates an "is-a" relationship (extends or implements), and the phrase "have" is of course a "has-a" relationship (usually instance variables).

☒ **B** and **D** are incorrect based on the above information.
(Objective 5.5)

15. Given:

```
 3. class A { }
 4. class B extends A { }
 5. public class ComingThru {
 6.   static String s = "-";
 7.   public static void main(String[] args) {
 8.     A[] aa = new A[2];
 9.     B[] ba = new B[2];
10.     sifter(aa);
11.     sifter(ba);
12.     sifter(7);
13.     System.out.println(s);
14.   }
15.   static void sifter(A[]... a2)    { s += "1"; }
16.   static void sifter(B[]... b1)    { s += "2"; }
17.   static void sifter(B[] b1)       { s += "3"; }
18.   static void sifter(Object o)     { s += "4"; }
19. }
```

What is the result?

A. `-124`

B. `-134`

C. `-424`

D. `-434`

E. `-444`

F. Compilation fails

Answer:

☑ **D** is correct. In general, overloaded var-args methods are chosen last. Remember that arrays are objects. Finally, an int can be boxed to an Integer and then "widened" to an Object.

☒ **A, B, C, E,** and **F** are incorrect based on the above information.
(Objective 1.5)

This page intentionally left blank