3.1 It is not. First, instance variables should always be private.

You would declare a public accessor and/or mutator method for an instance variable of class C when objects of other classes need to be able to read and/or write that instance variable. If that is not the case, then there is no reason – and in fact, you should not – declare public accessor/mutator methods.

Here’s an example of when you would:

class Coordinate {  
 private double mX, mY;  
  
 // ...  
 public double getX() {  
 return mX;  
 }  
  
 public void setX(double pX) {  
 mX = pX;  
 }  
  
 public double getY() {  
 return mY;  
 }  
  
 public void setY(double pY) {  
 mY = pY;  
 }  
}  
  
class Line {  
 private Coordinate mEnd1, mEnd2;  
  
 // ...  
 public double length() {  
 double deltaX = mEnd1.getX() - mEnd2.getX();  
 double deltaY = mEnd1.getY() - mEnd2.getY();  
 return Math.*sqrt*(deltaX \* deltaX + deltaY \* deltaY);  
 }  
}

You would declare a protected accessor and/or mutator method for an instance variable of class C when objects of subclasses of class C need to be able to call the protected accessor/mutator methods to read and write the private instance variables of C. (For those who advocate declaring such variables protected so they are directly accessible in subclasses, then protected accessor/mutator methods would not be required). Here is an example:

class TwoCoordinate {  
 private double mX, mY;  
  
 // ...  
 protected double getX() {  
 return mX;  
 }  
  
 protected void setX(double pX) {  
 mX = pX;  
 }  
  
 protected double getY() {  
 return mY;  
 }  
  
 protected void setY(double pY) {  
 mY = pY;  
 }  
}  
  
class ThreeDCoordinate extends TwoCoordinate {  
 protected double mZ;  
 public ThreeDCoordinate(double pX, double pY, double pZ) {  
 // mX and mY are not accessible in this class because they are  
 // private in the superclass. However, we may read and write  
 // then by calling the protected getX(), setX(), getY(), and  
 // setY() methods  
 // Note: if mX and mY were declared as protected in the superclass  
 // then we could directly access them here.  
 setX(pX);  
 setY(pY);  
 setZ(pZ);  
 }  
}

You would declare a private accessor and/or mutator method for an instance variable of a class C when the only accesses to that instance variable will be by the instance methods of class C. Here is an example:

class C {  
 private int mX;  
 // ...  
 // Assume the only place in the code where we are ever going to access mX is in the instance methods of this class. Then we can provide private accessor/mutator methods to permit other instance methods to access mX.  
  
 private int getX() {  
 return mX;  
 }  
  
 private void setX(int pX) {  
 mX = pX;  
 }  
   
 public int someInstanceMethod() {  
 return getX() - 3 + getY();  
 }  
}

Note: a lot of programmers would forgo the accessor/mutator methods in this case and would simply write this code this way:

class C {  
 private int mX;  
  
 // ...  
 public int someInstanceMethod() {  
 return mX - 3 + mY;  
 }  
}

And there’s nothing wrong with that it’s just personal opinions.

3.2 a. Legal. An object variable declared using the superclass type can refer to an object of the subclass type.

b. Illegal. An object variable declared using the subclass type cannot refer to an object of the superclass type.

c. Illegal for the same reason as b.

d. Legal for the same reason as a.

3.3 This is true. A subclass object differs from a superclass object in some way, and one way they usually differ is that subclass objects contain instance variables that are not found in superclass objects.

3.4 This is false. A subclass object will contain all of the instance variables that are declared in the superclass (i.e., they are all inherited, regardless of the superclass accessibility modifiers).

3.5

1-3. x1, x2, and x3. All instance variables of class are accessible in all instance methods of the class.

4-6. x1 and x2. public and protected instance variables of a superclass are directly accessible in subclasses.

7-9. See 4-6.

10-12. None of them. A superclass object has no access to instance variables of a subclass object.

13-15. See 10-12.

16-18. c1Method1() and c2Method1(). public and protected instance methods of a superclass are directly callable from subclasses.

19-21. See 16-18.

22-24. None of them. A superclass object has no access to instance methods of a subclass object.

25. Three: x1, x2, x3.

26. Six: x1, x2, x3 (inherited) and y1, y2, y3.

27. Nine: x1, x2, x3, y1, y2, y3 (inherited) and z1, z2, z3.

3.6 A method that has the same name as some other method but a different signature. For example:

class MyMath {  
 public int max(int a, int b) {  
 if (a >= b) return a;  
 else return b;  
 }  
  
 public String max(String a, String b) {  
 // String.compareTo() returns 0 if a == b and 1 if a > b  
 if (a.compareTo(b) >= 0) return a;  
 else return b;  
 }  
}

Note: also that overloaded methods may be in the same class or the overloaded methods may be in a superclass and subclass.

3.7 A method in a subclass that has the same name and signature as a method in a superclass. The subclass method will do something different than the superclass method.

class C {  
 // When we foo on a C object we add a and b.  
 public int foo(int a, int b) {  
 return a + b;  
 }  
}  
  
class D extends C {  
 int mD;  
  
 // When we foo on a D object we add a and b and divide by mD  
 @Override  
 public int foo(int a, int b) {  
 return super.foo(a, b) / mD;  
 }  
  
 private int getD() {  
 return mD;  
 }  
}

3.8 Accidental overloading occurs when we intend to override a method, but we incorrectly specify the signature:

class C {  
 // The signature of this method is <foo: int, int>.  
 public int foo(int a, int b) {  
 return a + b;  
 }  
}  
  
class D extends C {  
 int mD;  
  
 // foo is accidentally overloaded. The signature of this method is <foo: double, double>  
 public int foo(double a, double b) {  
 return super.foo(a, b) / mD;  
 }  
  
 private int getD() {  
 return mD;  
 }  
}

By adding the @Override annotation:

class C {  
 // The signature of this method is <foo: int, int>.  
 public int foo(int a, int b) {  
 return a + b;  
 }  
}  
  
class D extends C {  
 int mD;  
  
 // foo() is accidentally overloaded. The signature of this method is <foo: double, double>  
 // The @Override annotation will cause the Java compiler to detect the accidental overloading and will emit a syntax error for this code  
 @Override  
 public int foo(double a, double b) {  
 return super.foo(a, b) / mD;  
 }  
  
 private int getD() {  
 return mD;  
 }  
}

3.9 Using the super reserved word. See the code example in 3.7

3.10 True. Overloaded methods are two methods with the same name but different signatures. If we think of the method name as including the data types of the parameters, then the overload methods are truly different methods, e.g.,

public class MyMath {  
 public int max(int a, int b) { // Think of this method as being named maxIntInt()  
 if (a >= b) return a;  
 else return b;  
 }  
  
 public String max(String a, String b) { // Think of this method as being named maxStringString()  
 if (a.compareTo(b) >= 0) return a;  
 else return b;  
 }  
}

think of the name of the first max method as being named maxIntInt() and the name of the second max method as being maxStringString(). If Java didn’t permit overloading, then we would have to resort to this naming trick.

3.11 False, From Inheritance: Section 11:

An overridden method is one that appears in a subclass and does something different than the inherited method from the superclass. Overriding always occurs in the context of an inheritance relationship: a subclass overrides a method inherited from a superclass. It is not possible for a method in class C to override another method, also in class C, because it is not legal to have two declared methods in a class with the same signatures.

3.12 True, Overriding can occur in a superclass/subclass relationship

public class C {  
 // When we foo on a C object we add a and b  
 public int foo(int a, int b) {  
 return a + b;  
 }  
}  
  
public class D extends C {  
 // When we foo on a D object we add a, b, and c  
 public int foo(int a, int b, int c) {  
 return a + b + c;  
 }  
}

3.13 True. In fact, this is what overriding is all about. Subclass objects (which are a more specific type of superclass object) frequently need to do something different when a method is called on the subclass object as opposed to the same method being called on a superclass object.

3.15 False. Subclasses override methods that would normally be inherited from superclasses.

3.16 Using the **super** reserved word:

public class TwoDCoordinate {  
 private double mX, mY;  
  
 public TwoDCoordinate(double pX, double pY) {  
 setX(pX);  
 setY(pY);  
 }  
  
 protected double getX() {  
 return mX;  
 }  
  
 protected void setX(double pX) {  
 mX = pX;  
 }  
  
 protected double getY() {  
 return mY;  
 }  
  
 protected void setY(double pY) {  
 mY = pY;  
 }  
}  
  
public class ThreeDCoordinate extends TwoDCoordinate {  
 private double mZ;  
  
 public ThreeDCoordinate(double pX, double pY, double pZ) {  
 *// Call the superclass ctor to initialize mX and mY* super(pX, pY);  
 setZ(pZ);  
 }  
  
 public double getZ() {  
 return mZ;  
 }  
  
 public void setZ(double pZ) {  
 mZ = pZ;  
 }  
}

3.17 An abstract class must be declared with the abstract reserved word. The primary difference is that an abstract class cannot be instantiated. Also, an abstract class may declare methods that are not implemented. The fundamental purpose of an abstract class is to serve as a container for instance variables and instance methods that will be inherited by all subclasses of the abstract class.

4.1 makeSound()

4.2 Bee.java

4.3 Amphibian.java

4.4 Frog.java

4.5 Main.java

5 package h02\_50 Main.java and View.java

6.1 A nested class is a class that is declared within a class. It may be static or non-static. A non-static nested class is an inner class (if it’s not local or anonymous).

public class Outer {  
 public class Inner {  
   
 }  
}

6.2 A local class is declared in a block (a block, or compound statement, is a pair of braces {} containing stuff.

public class Outer {  
 private void someMethod() {  
 private class Local {  
  
 }  
 }  
}

6.3 An anonymous class doesn’t have a name. An anonymous class and an object of the class are created at the same time. An anonymous class isn’t actually a class declaration but rather an expression. Therefore, it may appear whatever an expression is legal. Anonymous classes are used in situations where the desired class is very small (usually an anonymous class is one that implements an interface) and only one object of that class will ever be created.

button.addActionListener(  
 new ActionListener() {  
 @Override  
 public void actionPerformed(ActionEvent e) {  
 *// ...* }  
 });