› Answers and Explanations

Bullets mark each step in the process of arriving at the correct solution.

1. The answer is D.
   - When you instantiate an object, you tell the compiler what type it is (left side of the equal sign) and then you call the constructor for a specific type (right side of the equal sign). Here's where you can put `is-a` to good use. You can construct an object as long as it `is-a` version of the type you have declared. You can say, "Make me a sandwich, and I'll say thanks for lunch," and you will always be right, but you can't say, "Make me lunch and I'll say thanks for the sandwich." What if I made soup?
   - Option A is correct, since Lunch *is-a* Object.
   - Options B and E are correct since Lunch *is-a* Lunch, and Sandwich *is-a* Sandwich. (Don't confuse the variable name sandwich with the type Lunch. It's a bad programming choice, since a reader expects that a variable named sandwich is an object of the class Sandwich, but it's legal.)
   - Option C is correct since Sandwich *is-a* Lunch.
   - Option D is incorrect. Object is not a Lunch. It's not reversible. The one on the right has to have an *is-a* relationship to the one on the left in that order. Option D will not compile.

2. The answer is E.
   - Superclass refers to grandparents as well as parents, all the way up the inheritance line, and subclass refers to grandchildren as well as children, all the way down the inheritance line, so all three relationships are true.

3. The answer is A.
   - A `Freshman` object will have direct access to its own private variables.
   - The instance variables for `UnderClassman` and `Student` are private and can only be accessed within that class.
   - Please note that it is considered a violation of encapsulation to declare instance variables public, and points will be taken off on the AP Computer Science A exam if you do so!

4. The answer is B.
   - Since the super call must be the first thing in the constructor, the Freshman constructor will hang on to the parameter `middleSchoolCode` and use the super call to pass the remaining parameters to the constructor for Underclassman. It will then set its own instance variable to the `middleSchoolCode` parameter. (It's worth noting that the Underclassman constructor will set its own two variables and pass the remaining variables to its super constructor.)

5. The answer is D.
   - Options A and B are incorrect. They show an incorrect usage of "extends". The keyword `extends` belongs on the class declaration, not the method declaration.
   - Option C is incorrect. It puts data types next to the arguments (actual parameters) in the `super` method call. Types only appear next to the formal parameters in the method declaration.
   - Option D is correct. It has fixed both of the above errors. The method with the same name in the superclass is called using the `super.methodName` notation, and it is passed the two parameters it is expecting. Then the `initialPlanner` method is called.
   - Option E is incorrect. It attempts to assign the parameters directly to their corresponding variables in the Student class. The variables in the parent class should be private, so the child doesn't know if they exist, and can't access them even if they do. Also, the child class shouldn't assume that assigning variables is all the parent class method does. Surely `attendClass` involves more than that!

6. The answer is C.
   - Option A is valid. It instantiates an array of `Club` objects. The right side of the instantiation specifies that there will be two entries, but it does not instantiate those entries. This is a valid statement, but note that the entries in this array are null.
   - Option B is valid. It uses an initialization list to instantiate the array of `Club` objects. `Club` has overloaded constructors. There is a constructor that takes an `ArrayList<String>` as an argument, so the first entry is correct, and there is a no-argument constructor, so the second entry is correct.
   - Option C is not valid. It uses an initialization list to instantiate an array of `SchoolClub` objects, but the list includes not only a `SchoolClub` object, but also a `Club` object. We could make a `Club[]` and include `SchoolClub` objects, but not the other way around. Remember the *is-a* designation. A `Club` is not a `SchoolClub`.
   - Option D is valid. It instantiates an array of `SchoolClub` objects by giving an initialization list of correctly instantiated `SchoolClub` objects.

7. The answer is A.
   - The first statement in the constructor of an extended class is a super call to the constructor of the parent class. This may be an implicit call to the no-argument constructor (not explicitly written by you) or it may be an explicit call to any of the superclass's constructors.
   - Option B is incorrect. It calls the super constructor but passes a parameter of the wrong type.
   - Option C is incorrect. `this.SchoolClub` has no meaning.
   - Option D is incorrect. This option is syntactically correct. The no-argument constructor will be called implicitly, but the members `ArrayList` instance variable will not be set.
   - Option E is incorrect. The super call must be the first statement in the constructor.

8. The answer is C.
   - First of all, remember that when an object is printed, Java looks for that object's `toString` method to find out how to print the object. (If there is no `toString` method, the memory address of the object is printed, which is not what you want!)
   - Variable p is of type Present. That's what the compiler knows. But when we instantiate variable p, we make is a KidsPresent. That's what the runtime environment knows. So when the runtime environment looks for a toString method, it looks in the KidsPresent class. The KidsPresent toString prints both the contents and the appropriate age.

9. The answer is B.
   - Since p is declared as a Present, and the Present class has a `setContents` method, the compiler is fine with this sequence.
   - When `setContents` is called, the run-time environment will look first in the KidsPresent class and it will not find a `setContents` method. But it doesn't give up! Next it looks in the parent class, which is Present and there it is. So the run-time environment happily uses the Present class's `setContents` method and successfully sets the Present class's contents variable to "blocks".
   - When the KidsPresent `toString` is called, the first thing it does is call `super.toString()`, which accesses the value of the contents variable. Since that variable now equals "blocks", the code segment works as you might hope it would and prints "contains blocks for a child age 4".

10. On the AP Computer Science A exam, you are often asked to extend a class, as you were in this question. Check your answer against my version below. A few things to notice:

- The constructor takes x, y, and z but then immediately passes x and y to the Point class constructor through the super(x, y) call.
- x and y, their setters and getters are in the Point class; they don't need to be duplicated here. Only variable z, setZ and getZ need to be added.

```
public class Point3D extends Point
{
    private int z;

    // Part a
    public Point3D(int myX, int myY, int myZ)
    {
        super(myX, myY);
        z = myZ;
    }

    // Part b
    public void setZ(int myZ)
    {
        z = myZ;
    }

    public int getZ()
    {
        return z;
    }
}
```