Abstraction and Interface

Friday, February 10, 2023 6:01 PM

Abstract Class

An *abstract class* is a superclass that represents an abstract concept, and therefore should not be instantiated. For example, a maze program could have several different maze components—paths, walls, entrances, and exits. All of these share certain features (e.g., location, and a way of displaying). They can therefore all be declared as subclasses of the abstract class `MazeComponent`. The program will create path objects, wall objects, and so on, but no instances of `MazeComponent`.

An abstract class may contain *abstract methods*. An abstract method has no implementation code, just a header. The rationale for an abstract method is that there is no good default code for the method. Every subclass will need to override this method, so why bother with a meaningless implementation in the superclass? The method appears in the abstract class as a placeholder. The implementation for the method occurs in the subclasses. If a class contains any abstract methods, it *must* be declared an abstract class.

Notice 1: So, abstract class is a super class that cannot create object

Notice 2: Only abstract class can contain abstract method

Notice 3: Subclass must override all abstract method of its abstract super class

The `abstract` Keyword

An abstract class is declared with the keyword `abstract` in the header:

```
public abstract class AbstractClass
{    ...
```

The keyword `extends` is used as before to declare a subclass:

```
public class SubClass extends AbstractClass
{    ...
```

If a subclass of an abstract class does not provide implementation code for all the abstract methods of its superclass, it too becomes an abstract class and must be declared as such to avoid a compile-time error:

```
public abstract class SubClass extends AbstractClass
{    ...
```

Here is an example of an abstract class, with two concrete (nonabstract) subclasses.

Important example:

```
public abstract class Shape
{
    private String name;

    //constructor
    public Shape(String shapeName)
    { name = shapeName; }

    public String getName()
    { return name; }

    public abstract double area();
    public abstract double perimeter();

    public double semiPerimeter()
    { return perimeter() / 2; }
}

public class Circle extends Shape
{
    private double radius;

    //constructor
    public Circle(double circleRadius, String circleName)
    {
        super(circleName);
        radius = circleRadius;
    }

    public double perimeter()
    { return 2 * Math.PI * radius; }

    public double area()
    { return Math.PI * radius * radius; }
}
```

```
public class Square extends Shape
{
    private double side;

    //constructor
    public Square(double squareSide, String squareName)
    {
        super(squareName);
        side = squareSide;
    }

    public double perimeter()
    { return 4 * side; }

    public double area()
    { return side * side; }
}
```

Notice 4: an abstract class can have instance variable and instance method.

Notice 5: It is possible that abstract class does not have an abstract method

Notice 6: An abstract class may or may not have constructors.

INTERFACES

Interface

An *interface* is a collection of related methods, either abstract (headers only) or default (implementation provided in the interface). Default methods are new in Java 8, and will not be tested on the AP exam. Non-default (i.e., abstract) methods will be tested on the exam and are discussed below.

The classes that implement a given interface may represent objects that are vastly different. They all, however, have in common a capability or feature expressed in the methods of the interface. An interface called `FlyingObject`, for example, may have the methods `fly` and `isFlying`. Some classes that implement `FlyingObject` could be `Bird`, `Airplane`, `Missile`, `Butterfly`, and `Witch`. A class called `Turtle` would be unlikely to implement `FlyingObject` because turtles don't fly.

Notice 7: a non-abstract class that implement interface must override all non-default (abstract) methods

abstract == non-default

method == default method

Defining an Interface

An interface is declared with the `interface` keyword. For example,

```
public interface FlyingObject
{
    void fly();          //method that simulates flight of object
    boolean isFlying();  //true if object is in flight,
                         //false otherwise
}
```

The `implements` Keyword

Interfaces are implemented using the `implements` keyword. For example,

```
public class Bird implements FlyingObject
{
    ...
```

This declaration means that two of the methods in the `Bird` class must be `fly` and `isFlying`. Note that any subclass of `Bird` will automatically implement the interface `FlyingObject`, since `fly` and `isFlying` will be inherited by the subclass.

A class that extends a superclass can also *directly* implement an interface. For example,

```
public class Mosquito extends Insect implements FlyingObject
{
    ...
```

Notice 8: A class can only extend one super class, but it can implement multiple interfaces

NOTE

1. The `extends` clause must precede the `implements` clause.
2. A class can have just one superclass, but it can implement any number of interfaces:

```
public class SubClass extends SuperClass
        implements Interface1, Interface2, ...
```

Interface vs. Abstract Class

- Use an abstract class for an object that is application-specific but incomplete without its subclasses.
- Consider using an interface when its methods are suitable for your program but could be equally applicable in a variety of programs.
- An interface typically doesn't provide implementations for any of its methods, whereas an abstract class does. (In Java 8, implementation of default methods is allowed in interfaces.)
- An interface cannot contain instance variables, whereas an abstract class can.
- It is not possible to create an instance of an interface object or an abstract class object.

Notice 9: Non-default method in interface cannot be implemented, but default method can be implemented. Non-abstract methods in abstract class can have implementation, but abstract method cannot have implementation.

Let's do some practice question:

11. Consider these class declarations:

```
public class Person
{
    ...
}

public class Teacher extends Person
{
    ...
}
```

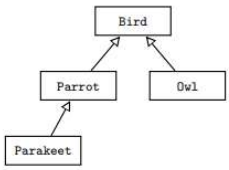
Which is a true statement?

I `Teacher` inherits the constructors of `Person`.
II `Teacher` can add new methods and private instance variables.
III `Teacher` can override existing private methods of `Person`.

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) II and III only

12. Which statement about abstract classes and interfaces is *false*?
(A) An interface cannot implement any non-default instance methods, whereas an abstract class can.
(B) A class can implement many interfaces but can have only one superclass.
(C) An unlimited number of unrelated classes can implement the same interface.
(D) It is not possible to construct either an abstract class object or an interface object.
(E) All of the methods in both an abstract class and an interface are public.

13. Consider the following hierarchy of classes:



A program is written to print data about various birds:

```
public class BirdStuff
{
    public static void printName(Bird b)
    { /* implementation not shown */ }

    public static void printBirdCall(Parrot p)
    { /* implementation not shown */ }

    //several more Bird methods

    public static void main(String[] args)
    {
        Bird bird1 = new Bird();
        Bird bird2 = new Parrot();
        Parrot parrot1 = new Parrot();
        Parrot parrot2 = new Parakeet();
        /* more code */
    }
}
```

Assuming that none of the given classes is abstract and all have default constructors, which of the following segments of `/* more code */` will *not* cause an error?

(A)
```
printName(parrot2);
printBirdCall((Parrot) bird2);
```

(B)
```
printName((Parrot) bird1);
printBirdCall(bird2);
```

(C)
```
printName(bird2);
printBirdCall(bird2);
```

(D)
```
printName((Parakeet) parrot1);
printBirdCall(parrot2);
```

(E)
```
printName((Owl) parrot2);
```

Refer to the classes below for Questions 14 and 15.

```
public class ClassA
{
    //default constructor not shown ...

    public void method1()
    { /* implementation of method1 */ }
}

public class ClassB extends ClassA
{
    //default constructor not shown ...

    public void method1()
    { /* different implementation from method1 in ClassA*/ }


    public void method2()
    { /* implementation of method2 */ }
}
```

14. The `method1` method in `ClassB` is an example of
    (A) method overloading.
    (B) method overriding.
    (C) polymorphism.
    (D) information hiding.
    (E) procedural abstraction.

15. Consider the following declarations in a client class.

```
ClassA ob1 = new ClassA();
ClassA ob2 = new ClassB();
```

Which of the following method calls will cause an error?

I `ob1.method2();`

II `ob2.method2();`

III `((ClassB) ob1).method2();`

(A) I only
(B) II only
(C) III only
(D) I and III only
(E) I, II, and III

19. Consider the `Computable` interface below for performing simple calculator operations:

```
public interface Computable
{
    /** Return this Object + y. */
    Object add(Object y);

    /** Return this Object - y. */
    Object subtract(Object y);

    /** Return this Object * y. */
    Object multiply(Object y);
}
```

Which of the following is the *least* suitable class for implementing `Computable`?

(A) `LargeInteger` `//integers with 100 digits or more`

(B) `Fraction` `//implemented with numerator and //denominator of type int`

(C) `IrrationalNumber` `//nonrepeating, nonterminating decimal`

(D) `Length` `//implemented with different units, such //as inches, centimeters, etc.`

(E) `BankAccount` `//implemented with balance`

26. Consider the `Orderable` interface and the partial implementation of the `Temperature` class defined below:

```
public interface Orderable
{
    /** Returns -1, 0, or 1 depending on whether the implicit
     *  object is less than, equal to, or greater than other.
     */
    int compareTo (Object other);
}

public class Temperature implements Orderable
{
    private String scale;
    private double degrees;

    //default constructor
    public Temperature ()
    { /* implementation not shown */ }

    //constructor
    public Temperature(String tempScale, double tempDegrees)
    { /* implementation not shown */ }

    public int compareTo(Object obj)
    { /* implementation not shown */ }

    public String toString()
    { /* implementation not shown */ }

    //Other methods are not shown.
}
```

Here is a program that finds the lowest of three temperatures:

```
public class TemperatureMain
{
    /** Find smaller of objects a and b. */
    public static Orderable min(Orderable a, Orderable b)
    {
        if (a.compareTo(b) < 0)
            return a;
        else
            return b;
    }

    /** Find smallest of objects a, b, and c. */
    public static Orderable minThree(Orderable a,
           Orderable b, Orderable c)
    {
        return min(min(a, b), c);
    }

    public static void main(String[] args)
    {
        /* code to test minThree method */
    }
}
```

Which are correct replacements for `/* code to test minThree method */`?

I
```
Temperature t1 = new Temperature("C", 85);
Temperature t2 = new Temperature("F", 45);
Temperature t3 = new Temperature("F", 120);
System.out.println("The lowest temperature is " +
        minThree(t1, t2, t3));
```

II
```
Orderable c1 = new Temperature("C", 85);
Orderable c2 = new Temperature("F", 45);
Orderable c3 = new Temperature("F", 120);
System.out.println("The lowest temperature is " +
        minThree(c1, c2, c3));
```

III
```
Orderable c1 = new Orderable("C", 85);
Orderable c2 = new Orderable("F", 45);
Orderable c3 = new Orderable("F", 120);
System.out.println("The lowest temperature is " +
        minThree(c1, c2, c3));
```

(A) II only
(B) I and II only
(C) II and III only
(D) I and III only
(E) I, II, and III