**Advanced Java**

**Week 3 – Chapter 13 – Abstract Classes and Interfaces**

An *abstract* class cannot be used to create objects. However, it can contain abstract methods, which are implemented in the (concrete) subclasses. We use abstract classes to establish a contract of sorts; any concrete child of our abstract class must agree to implement all of the methods defined. The abstract class can be partially defined (that is, contain methods that the child classes inherit and run as is) but can also include *abstract methods*, which provide the method signature and nothing more.

(1) A non-abstract class can still contain an abstract method

(a) true (b) false

(2) To create an instance of an abstract class, use the new operator as always.

(a) true (b) false

(3) A class that contains abstract methods must be abstract.

(a) true (b) false

(4) You may create an abstract child class from a concrete parent class.

(a) true (b) false

(5) You can create an abstract child class from an abstract parent class.

(a) true (b) false

(6) A subclass can over-ride a concrete parent method and make it abstract.

(a) true (b) false

(7) You can use an abstract class as though it were a data type.

(a) true (b) false

(8) Abstract methods can be static or non-static.

(a) true (b) false

(9) The constructor of an abstract class is public.

(a) true (b) false

(10) The terms *abstract class* and *interface* are interchangeable.

(a) true (b) false

**What are some differences between an *abstract class* and an *interface*?**

An abstract class is object-oriented. It offers the basic data that a child class should have, and the methods that it should implement. An abstract class defines what the object is and what it can do. In other words, an abstract class defines a contract that implementing child classes must fulfill. We use abstract classes to define common behavior among several closely-related subclasses.

An interface is functionally-oriented. It defines the functionality that an inheriting object should have, regardless of what that inheriting object is or does. Therefore, an interface definition consists of definitions of public members, with no implementation. We use interfaces to provide common functionality to several unrelated subclasses.

An abstract class is inherited using “extends.”

A child class can extend only one parent, abstract or otherwise.

An interface is inherited using “implements.”

A child class can implement many interfaces.

An abstract class can contain constructors.

An interface does not contain constructors.

An abstract class can make use of access modifiers for data members and methods.

An interface cannot have access modifiers; by default, all methods are public.

An abstract class can contain data members that can be final or non-final.

An interface cannot contain data members, only constants.

An abstract class can extend another abstract class and can implement many interfaces.

An interface can only extend another interface.

**What are the obligations on a class that implements an interface?**

That class must provide an implementation (which can be empty) for every method defined in the interface. For example, MouseListener provides five method definitions:

MouseClicked(MouseEvent e)

MouseEntered(MouseEvent e)

MouseExited(MouseEvent e)

MousePressed(MouseEvent e)

MouseReleased(MouseEvent e)

If we don’t want to make any use of, say, MouseReleased, we simply write in our code

public void MouseReleased(MouseEvent e){}

and Java is perfectly happy; we’ve provided a minimal (empty) implementation for that method.

We’ll create a class hierarchy for geometric shapes, with the abstract class Shape at the top:

|  |
| --- |
| *Shape* |
| # double x, y // upper left corner of bounding box  # Color: c  # boolean: fill  + boolean SHAPE\_DEFAULT\_FILL = false  + Color SHAPE\_DEFAULT\_COLOR = Color.gray  + boolean SHAPE\_SET\_FILL = true // use with setFill  + boolean SHAPE\_SET\_OUTLINE = false // use with setFill |
| # Shape() // sets Color and fill to default values  + Color getColor()  + void setColor(Color c)  + boolean getFill()  + void setFill(boolean f)  + double getX();  + double getY();  + void setX(double x);  + void setY(double y)  + double *getArea()*  *+* double *getPerimeter()*  + void *drawShape()* |

|  |
| --- |
| Ellipse |
| #double radius1, radius2 |
| + Ellipse(double x, double y,  double r1, double r2) |

|  |
| --- |
| Rectangle |
| #double width, height |
| + Rectangle(double x, double y,  double width, double height) |

|  |
| --- |
| Square |
|  |
| + Square(double x, double y,  double width) |

|  |
| --- |
| Circle |
|  |
| + Circle(double x, double y,  double r) |

For now the drawing methods will simply write text to the console.

**HOMEWORK ASSIGNMENT**

For next week, implement the entire hierarchy of classes as defined, with the drawShape method doing nothing more than writing text to the screen. Make it fairly detailed: for instance, it should specify the upper-left coordinates and the various parameters that make up the shape and specify the color and such. Then design and add at least one additional shape.