Java Abstract

Lab setup

- Create a new Java project called "JavaAbstract"

Abstract classes

Abstract classes are declared using the keyword `abstract`. An abstract class can have abstract and non-abstract methods. Like interfaces, abstract classes cannot be instantiated.

Abstract classes can include constructors, static methods and fields, and final methods. When extending an abstract class, you must provide an implementation for all abstract methods.

1. We can write an abstract class called `Shape` and use it as the base class to define other classes of Shape.

```java
abstract class Shape {

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

}
```

Figure 1 Shape abstract class

```java
public class Rectangle extends Shape {
    private double length;
    private double width;

    public Rectangle(double length, double width){
        this.length = length;
        this.width = width;
    }

    @Override
    public double area() {
        return length*width;
    }

    @Override
    public double perimeter() {
        return 2*length + 2*width;
    }
}
```

Figure 2 Rectangle class derived from Shape.

2. We define a `Rectangle` from `Shape`. This class is shown in Figure 2. We can add new fields such as length and width to facilitate the implementation of `area` and `perimeter`. Again, we must provide the implementation for all methods declared abstract in `Shape`.

Graded Submission Task 1:

1. Define the class `Circle which extends Shape.`
2. Instantiate an object for each Shape in an array and print out the area and perimeter of each Shape

** STOP **

Ensure you have saved the source files in your project. Make a copy of your Eclipse project at this point (you can do this from your system's file explorer). Name the file labelling it as Task 1. You will submit this copy at the end of this lab.

Abstract classes vs Interfaces

Abstract classes are very similar to interfaces. The two main differences between them are that abstract classes can declare fields that are not static and final, and define public, protected, and private concrete methods (concrete meaning they have a body). Fields of interfaces are automatically public, static, and final while methods declared and defined (as final/default methods) are public.

Classes also can only `extend` one class, abstract or not, but they can `implement` any number of interfaces (facilitating multiple inheritance).

Graded Submission Task 2

You have been provided with the beginnings of a game implemented in Java called DungeonCrawler. For now, there exists the abstract class `Entity` referring to any entities such as player characters or enemies, and a class extending that called `Dummy` which may be used for testing. There is also an interface called `Consumable`. Items which have a limited number of uses are intended to implement this interface.

1. Write the abstract class for an `Item` in a video game and extend two subclasses `Weapon` and `ConsumableItem`.
2. Implement `HealingItem` and `DamagingItem`, which extend `ConsumableItem`.
3. Both of these classes have a field for the amount of healing/damage they cause. They perform the same function as `ConsumableItem` i.e. they decrement `usesLeft` by one whenever used. They should also heal/damage the target entity by the amount.

4. Test what you have implemented in a driver. Specifically,

    a. Create an ArrayList of Items and add to it at least one of each `Weapon`, `HealingItem`, and `DamagingItem` objects.
    b. Print the ID and description for each item.
    c. Create a `Dummy` with some amount of health and "use" the `HealingItem` and `DamagingItem` on it to test its health value properly updates.
    d. "Attack" the `Dummy` with the `Weapon` and check its health.

`Item` should have `protected` fields `itemID`, `description`, and `weight`. To avoid rewriting code, you can implement a constructor in `Item` that is only accessible to it and its subclasses to initialize the fields `itemID`, `description` and `weight`.

`Weapon` is an item that has an attack value and a method `attack(Entity target)`.

`ConsumableItem` is an item with a limited number of uses. It is an abstract class. It has a field `usesLeft`. We intend to derive two more classes from `ConsumableItem`. These two classes are `HealingItem` and `DamagingItem`.

Normally, because `ConsumableItem` implements `Consumable`, it must provide a definition for `use(Entity target)`, however, because `ConsumableItem` is an abstract class, we can postpone providing the definition until creating its children by declaring `use(Entity target)` to be `abstract`.

Alternatively, because all `ConsumableItem` objects should decrement `usesLeft` by one, we could provide that as the definition for `use(Entity target)` in `ConsumableItem` and further override the method in `HealingItem` and `DamagingItem` while using `super` to call the `ConsumableItem` version.

The latter choice leads to repetitive code, but also forces the developer to provide the definition of `use(Entity target)` for each class that extends `ConsumableItem`.

** STOP **

Ensure you have saved the source files in your project. Make a copy of your Eclipse project at this point (you can do this from your system's file explorer). Name the file labelling it as Task 2. You will submit this copy at the end of this lab.

Lab wrap-up

In this lab we discussed abstract classes. Abstract classes are classes which cannot be instantiated. They are declared using the keyword abstract. Abstract classes can include both abstract and non-abstract (concrete) methods. Abstract classes can also define constructors, static methods and fields, and final methods. When a class extends an abstract class, that class must provide implementation for all abstract methods, or the extending class must also be abstract. The defined constructors cannot be called except by extending classes using super because abstract classes cannot be instantiated.

We used the example of the abstract class Shape. Different types of shapes calculate their area and perimeter differently. They also need different information to perform those calculations. We defined the class Rectangle which is derived from Shape. A Rectangle needs length and width information to calculate area and perimeter so defined fields for those. We then override area and perimeter to do the proper calculations. If you recall from geometry: all squares are rectangles

We also defined the Circle class which derived shape. Circle needs its radius to and PI to calculate the area and perimeter (circumference). The radius is included as a field. We probably don't want all Circles to have their own definition of PI so we could define it as a static final field in Circle.

We also discussed the differences between abstract classes and interfaces. In implementation, the main differences are that interfaces cannot define fields with are not public, static, and final and their methods are always public and never concrete unless defined as final or default. Abstract classes should be used when we need to define an outline and behaviors for a subset of classes. Interfaces should be used to define subsets of classes which have the same behaviors.

Classes cannot extend more than one class, but they can implement more than one interface (multiple inheritance). This means a Rectangle could not be both a Shape and an Animal, but it could be either a Shape or a Rectangle and Runnable, Moveable, Flippable, and Cloneable.

Submission notes for graded tasks:

- Submit a zipped folder containing the Eclipse projects for each graded task.
- For each project, ensure you have included the src folder containing the .java file, the .classpath file, and the .project file.
- Ensure that each project is named appropriately to correspond with each task.
- Zip the projects together and name the zipped folder “Lab15_firstName_lastName”.
- Submit the zipped file to the respective Canvas link.

Rubrics	
Task 1 (allotted points) - Compilation (2) - Circle class defined (2) - Circle extends Shape (2) - Circle defines area and perimeter (2) - Demonstrate outputs of derived Shapes in driver (2)	Task 2 (allotted points) - Compilation (2) - Item base class (2) - Weapon and ConsumableItem defined (2) - HealingItem and DamagingItem implemented (2) - Driver demonstrates functionality (2)