Computer Science II
Handout 8

Abstract classes – rules summary

- Abstract classes cannot be instantiated (no `new` keyword)
  - Constructors are allowed, and will be invoked by subclasses
- Abstract methods must be overridden in any non-abstract subclasses
  - An abstract subclass may skip implementing all abstract methods
- Abstract methods are all non-static

A new access modifier appears!

- Within a class, `private` members may be accessed only within the same class
- Within a class, `public` members may be accessed by any class
- Within a class, `protected` members may be accessed only within the same class *or* by subclasses

Modifier	Class	Subclass	World
`public`	Y	Y	Y
`protected`	Y	Y	N
`private`	Y	N	N

https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Interfaces – even-more-abstract classes?

- A Java `interface` is a class-like definition

- An `interface` contains only static variables and abstract methods
  - Within an interface, all methods are abstract by default – no need for the `abstract` keyword

- Since all methods are abstract, nothing is implemented directly

Interfaces – even-more-abstract classes?

- Instead, other classes implement the abstract methods given by an interface
  - We say that a class *implements* an interface in the same way a subclass *extends* a superclass
- Interfaces are *contracts* that specify exactly which methods must be implemented
  - How they are implemented is a decision for the implementing class
- Interfaces specify the behaviour of a class
  - Implementing classes then define the individual actions

Interfaces – even-more-abstract classes?

- Writing an interface looks similar to writing a class

```
public interface MyInterface {
```

- Implementing an interface also should look familiar, just with new keywords

```
public class MyClass implements MyInterface {
```

Interfaces – abstract but not classes

- While similar to abstract classes, interfaces have several important differences:

	Interfaces…	**Abstract classes…**
Contain methods that are:	*Always* abstract	Either abstract or non-abstract
Contain variables that are:	*Always* static *and* final	Either static/instance, final/non-final
Contain members that are:	*Always* public	Either public, private, or protected
Are implemented/extended by:	Multiples classes	Only one class
Contain implementation:	Never	Sometimes
Contain constructors:	Never	Sometimes

Interfaces – example

```
public interface Car {
        public static final NUM_WHEELS = 4;
        public abstract double getSpeed();
}
```

```
public class Delorean implements Car {
        private int speed;

        public Delorean() {
                speed = 88;
        }

        public double getSpeed() {
                return (double)speed;
        }
}
```

Interfaces – example #2

- Create an interface **Geometry** that:
  - Stores a constant representing pi
  - Requires a getArea method that requires no parameters and returns a double
  - Requires a getVolume method that requires no parameters and returns a double

- Then create a class **Cylinder** that:
  - Stores attributes for radius and height
  - Has a constructor that sets both instance variables from parameters
  - Implements the Geometry interface appropriately
  - Has a toString method that prints the radius, height, area, and volume of the cylinder

```
public interface Geometry {
        public static final double PI = 3.1415926535;


}
```

```
public class Cylinder implements Geometry {
        private double radius;
        private double height;


        public Cylinder(double r, double h) {



        }

        public double getArea() {


        }

        public double getVolume() {
            return PI*radius*radius*height;
        }

        public String toString() {
            return "Dimensions: r=" + radius
                 + " h=" + height
                 + ", Area: " + getArea()
                 + ", Volume: " + getVolume();
        }

}
```

```
public class CylinderDemo {
	private double height;

	public static void main(String[] args) {
		Cylinder c = new Cylinder(5, 2);
		System.out.println(c);
	}
}
```

```
> Dimensions: r=5.0 h=2.0, Area: 219.911485745, Volume: 157.079632675
```

Interfaces vs abstract classes

- Choosing when to use an **interface** or an **abstract class** is something you must do depending on your program and the relationships between your classes
  - Consider whether using an interface makes more/less sense than using inheritance
  - Consider how many “extensions” will be needed
  - Consider how often the code may need to be updated (changes in an interface require changes in *every* class that implements)

Packages

- Packages are a way for Java to group together related classes
  - Typically, classes all belonging to one application/program

- This keeps things tidy and avoids naming conflicts
  - Classes in different packages do not require unique names

- Packages are imported using the `import` keyword

```
import packageName.ClassName;   // Imports one class
import packageName.*;           // Imports all classes
```

Packages

- Packages can contain other packages, and essentially form a “file system” that Java can use to find specific classes

```
import packageName.subPackageName.className;
import packageName.subPackageName.*;
```

- By default, Java always looks within the “current directory” for other classes: using `import` tells Java to look for named classes in other locations
  - We have already seen this in use:

```
import java.util.Scanner;
```

A new access modifier appears!

- Omitting an access specifier completely is the “default”, and is most convenient when creating packages
  - Its access level lies half-way between `protected` and `private`

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	**Y**	N
No modifier	Y	Y	**N**	N
`private`	Y	N	N	N

https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Java Standard Library (API)

- The Java Standard Library contains a *lot* of packages and classes!

  https://docs.oracle.com/javase/7/docs/api/

- Usually, these need to be imported

```
import java.util.Scanner;
```

- Sometimes, they are there *by default*

```
//import java.lang.*;
```

Java Standard Library (API)

- The java.lang package is always included, and contains fundamental classes for the Java language like:

  `String`

  `Math`

  `Double`

  `Integer`

  `Object`

- Every class inherits *implicitly* from this last class: **Object**
  - This is the root class of the class hierarchy

Java Standard Library (API) – Object

- The **Object** class contains:
  - A default, no-args constructor (so every class inherits one for free!)
  - A toString method that returns a String “representing” the Object (so every class inherits this for free!)
  - Several other generic utility methods

Polymorphism

- Inheritance allows for *polymorphism* in Java

- Literally, this term means “many forms”
  - In Java, it means that a single instance of a class may be treated as something *other than its declared type*

```
public class Shape { // ...

public class Circle extends Shape { // ...

public class Rectangle extends Shape { // ...
```

Polymorphism

- In other words, we can declare the *type* to match the superclass, and store references to subclass objects

```
public static void main(string[] args) {
        Shape s1, s2, s3;

        s1 = new Shape();
        s2 = new Circle(5);
        s3 = new Rectangle(3, 10);

        System.out.println(s1.getArea()); // Prints 0.0 (‘area’ is not initialized)
        System.out.println(s2.getArea()); // Prints 78.5398...
        System.out.println(s3.getArea()); // Prints 30.0
}
```

Polymorphism

- This makes it easier to define more generic methods

```
public void printArea(Shape s) {
        System.out.println("The shape's area is = " + s.getArea());
}
```

```
public static void main(string[] args) {
        Shape s1, s2;

        s1 = new Shape();
        s2 = new Circle(5);
        Rectangle r = new Rectangle(3, 10);

        printArea(s1); // Prints 0.0
        printArea(s2); // Prints 78.5398...
        printArea(r); // Prints 30.0
}
```