Object Oriented Programming with Java

CHAP 1: JAVA PROGRAMMING BASICS

Object Oriented Design: By Aphrodice Rwagaju

Why OOP?

- OOP features were incorporated into programming languages to help developers produce code that is:
  - Easier to test and so of higher quality
  - Easier to maintain
  - Easier to re-use in future projects

- OOP seeks to achieve these objectives by making code more modularized (i.e. data is stored self-sufficient classes)

Concepts of OOP

- **Inheritance**: sub class extends a super class
- **Encapsulation**: hiding instance variables by making them private and accessing them using public setters and public getters.
- **Abstraction**: providing unimplemented methods to be implemented in implementers classes
- **Polymorphism**: Properties of object which allow it to take multiple forms

Defining the Classes

- When designing your program think about what data should be grouped together, e.g
  - A person's name, address, DOB and other details
  - A coordinate's X and Y values

- Classes are often a model of real world things, i.e. a *User* class models a person…

```
class User {
   String name;
   int age;
}
```

Methods and Variables

- We can think of methods as what a class ***does*** and variables as what a class ***knows***
- Methods that are not called by other classes should be ***private*** or ***protected***
- Variables not accessed by other classes should be ***private*** or ***protected***
- Methods should not be too long (50 lines?) as long methods are harder to test and maintain

Encapsulation

- Encapsulation is about concealing the functionality of a class from other classes
- A class should provide an interface of public methods which other classes can call to manipulate that classes data
- This means that the functionality of a class can be changed without breaking the entire program
- **Encapsulation is about hiding data using private instance variables and access them using public setters and public getters in other classes**

Encapsulation: Getters and Setters

- It's common practice to prevent direct access to instance variables from other classes, and instead provide ***getter*** and ***setter*** methods

```
class Shape {
   protected int color;

   public void setColor(int color) {
      this.color = color;
   }
   public int getColor() {
      return color;
   }
}
```

Why Encapsulation?

- We can modify a class without changing how other classes interact with it
- We can make a field read-only by only providing a getter
- We can protect our class from invalid values by checking values in the setter, e.g.

```
public void setColor(int color) {
   if (color >= 0)
      this.color = color;
}
```

Inheritance

- Inheritance enables us to share functionality between different classes, and thus avoid duplication of code

- We should look for generalizations that can be made about our classes, e.g. if we have a ***Student*** class and ***Instructor*** class, and they both have names, DOBs etc, then we could extract that data to a more general superclass called ***Person***

Advantages of inheritance

- Code reusability
- Extensibility
- Overriding
- Data hiding

Naming Conventions

- Class names should start with uppercase and use Camel case
- Methods and variables should start with lowercase and use camel case
- You should use meaningful names but not too long…

```
int n;                                  // Short but meaningless
int numberOfUsersInThisProgram;         // Bit too long
int numUsers;                                       // Good!
```

Packages

- Packages in Java are like namespaces in other languages. There are used to:
  - Group related classes together – e.g. all the classes in a particular project
  - Differentiate between classes from different sources – e.g. if you create a class which has the same name as another class in a library you are using

Package Naming

- You can call your packages whatever you like, but if your code is going to be made publicly available, you'll want to choose package names that are unique

- Sun recommends that you combine your company's TLD, domain name, and package name, e.g.
  - org.clintonhealthaccess.ilms
  - rw.ac.rca.smis

Abstraction

- Inheritance is a very useful feature of Java because it allows us to extract common functionality from classes into super-classes

- For example, a Shape class may contain the functionality common to different shape classes, and we can write...

```
Shape s1 = new Triangle();
Shape s2 = new Circle();
Shape s3 = new Rectangle();
```

Abstraction

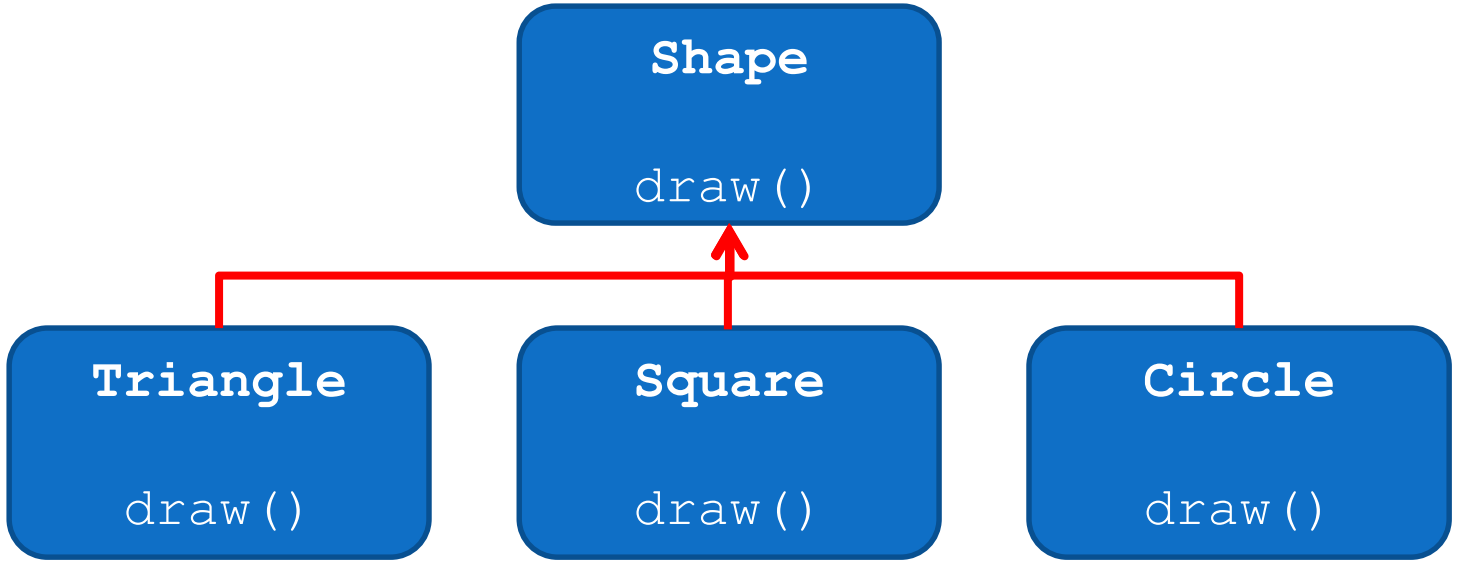
- Different subclass objects can even be stored in the same static array or collection, e.g.

```
Shape[] myShapes = new Shape[2];
myShapes[0] = new Triangle();
myShapes[1] = new Circle();
```

or

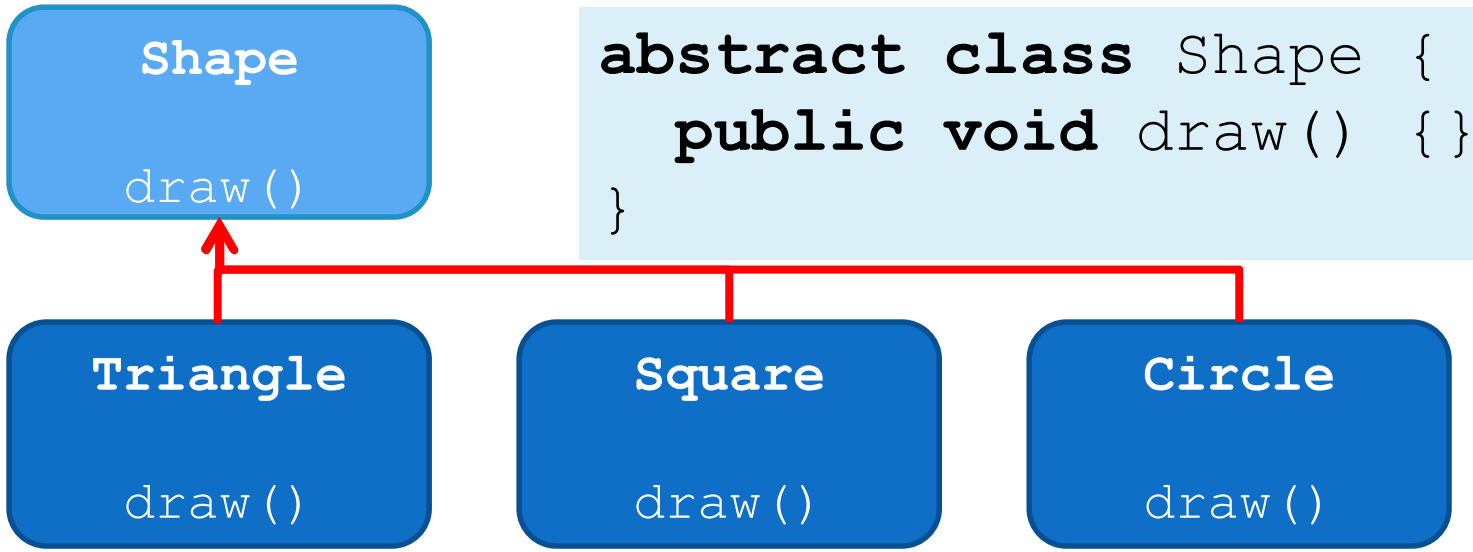
```
ArrayList<Shape> myList = new ArrayList<Shape>();
myList.add(new Triangle());
myList.add(new Circle());
```

Abstraction: Example



- How would one implement the draw method in the shape class?
- Should the shape class ever be instantiated?

Abstraction: Example




```
abstract class Shape {
  public void draw() {}
}
```

- Declaring the class as `abstract` prevents it from instantiated, i.e.

```
Shape s = new Shape();
```

Compiler error

Abstraction: Abstract methods

- Methods can also be declared `abstract`
- This means that they don't have an implementation
- Subclasses MUST provide an implementation or be abstract themselves
- A class with any abstract methods must be abstract itself, e.g.

```
abstract class Shape {
  public abstract void draw();
}
```

Abstraction: Abstract methods

```
class Shape {
  public abstract void draw();
}
```

Wrong: A class with abstract methods must be abstract

```
abstract class Shape {
  public abstract void draw();

  public String toString() {
    return "I am Abstract Class Shape";
  }
}
```

OK: An abstract class can have a mixture of abstract and normal methods

Abstraction: Abstract methods

```
abstract class Shape {
  public abstract void draw();
}
```

```
class Square extends Shape {
  public void draw() {
    ...
  }
}
```

```
abstract class Polygon extends Shape {
}
```

OK: A subclass can implement all abstract methods

OK: A subclass doesn't have to implement a method if it is abstract as well

Polymorphism

- Polymorphism refers to many forms, or it is a process that performs a single action in different ways
- It occurs when we have many classes related to each other by inheritance
- Polymorphism is of two different types, i.e., **compile-time** polymorphism and **runtime** polymorphism
  - Compile-time polymorphism is achieved by method **overloading**
  - Runtime polymorphism is achieved by method **overriding**

Polymorphism: Overloading vs Overriding

- Next session …

References

- https://stackify.com/oops-concepts-in-java/
- http://java.sun.com/docs/books/tutorial/java/IandI/abstract.html
- https://www.mygreatlearning.com/blog/oops-concepts-in-java/

EoF