Introduction to Programming

with Java, for Beginners

Casting
Polymorphism
Inheritance with Abstract Classes

Inheritance Recap

```java
public class Vehicle{
    private int registrationNumber;
    public int getRegNumber(){
        return registrationNumber;
    }
    public void setRegNumber(int num){
        registrationNumber = num;
    }
}
```

```java
public class Car extends Vehicle {
    //new instance variable, not inheritance
    private int numberOfDoors;
    public Car(int doors){
        numberOfDoors = doors;
    }
    public int getDoors() {
        return numberOfDoors;
    }
}
```

```
ESE112 1
```

Interactions I

```java
> Car c = new Car(2);
> c.getDoors()
2
> c.getRegNumber() //Inherited Method
0
> c.setRegNumber(45) //Inherited Method
> c.getRegNumber()
45
```

```
ESE112 2
```

Interactions II

```java
> Vehicle v = new Car(4); //valid
> v.setRegNumber(3)
> v.getRegNumber()
3
> v.getDoors()
Error
```

```java
> (Car)v to tell the computer to treat "v" as if it were actually of type Car. So, you could do:
```

```java
> ((Car)v).getDoors()
4
```

```
ESE112 3
```
**Cast Exception Error**
- Suppose you did not check if v is an instance of class Car
  - i.e. Vehicle v = new TwoWheeler();
  - ((Car)v).getDoors() //is perfectly legal statement
- You will get a runtime or semantic error
  - Known as Class Cast Exception
    - Saying TwoWheeler type cannot be cast to Car type

**Polymorphism**
- Polymorphism means many (poly) shapes (morph)
- In Java, polymorphism refers to the fact that you can have multiple methods with the same name in the same class
- There are two kinds of polymorphism:
  - Overloading: Two or more methods with different signatures
  - Overriding: A method in a subclass to "override" a method in the superclass that has the same signature
- We've already seen Overloading scenario with Constructors
  - E.g. public RealVector(double x, double y) {...}
  - public RealVector(double mag, Angle theta) {...}

**Method Overloading**
Method overloading occurs when
- A class has two or more methods with the same name but different signatures
  - i.e. the number, order, or types of their parameters differ

  ```java
  // the foo method is overloaded
  public void foo() {...}
  public void foo(int x) {...}
  public void foo(int x, double y) {...}
  ```
- When the foo(...) method is called, Java picks the one that "matches". E.g.
  ```java
  foo(10, 350.5);
  ```

**Overriding**
- Overriding occurs if
  - There are two or more methods with the same name and the same signature in an inheritance chain
  - For example, the Object class has a toString() method
    - It can be overridden in a subclass simply by creating a method with the same signature
      ```java
      public String toString() {...}
      ```
  - Java picks the "lowest" method in the inheritance chain possible
Some Methods cannot be overwritten

class Animal {
    final boolean canMove(int direction) { ... }
}

class Rabbit extends Animal {
    // inherits but cannot override canMove(int)
}

- Just like variables can be final, methods can also be final
- Methods that are final cannot be overridden in the subclass

Overriding Variables

- You can, but you shouldn't
- Possible for child class to declare variable with same name as variable inherited from parent class
  - one in child class is called shadow variable
  - confuses everyone!
- Child class already can gain access to inherited variable with same name
  - there's no good reason to declare new variable with the same name

Example

class Animal {
    String name;
    public Animal(){
        name = "Animal";
    }
}

> Dog d = new Dog();
> d.name
Dog

Some Variables Cannot be Shadowed

- class BorderLayout {
    public static final String NORTH = "North";
    ...
}

- If you were to create a subclass of BorderLayout, you would not be able to redefine NORTH
Abstract Features

- There are times when super class may not provide complete implementation for a method

- Example
  - Animal class has makeNoise method() which is not complete has noise is dependent on the Animal
    - We say that makeNoise is an abstract feature
  - If class Dog extends Animal, then it can complete this method by providing "woof" sound

Abstract Classes

- Abstract class typically used as partial description inherited by all its descendants

- Description insufficient to be useful by itself
  - cannot instantiated if defined properly

- Descendent classes supply additional information so that instantiation is meaningful
  - abstract class is generic concept in class hierarchy
  - class becomes abstract by including the `abstract` modifier in class header

Abstract Classes (contd..)

- An abstract class is a class
  - public abstract class Animal { .. }

- It cannot be instantiated
  - Illegal: Animal a = new Animal()

- It may have “abstract methods” i.e. methods with keyword abstract
  - Abstract methods are body-less i.e. no code within them

- It can also have regular/concrete methods
  - Methods with code in them

Setting up Inheritance with an Abstract Class

```java
public abstract class Animal{
    private double hunger;
    private boolean isAwake;

    public void eat(){
        hunger = 0;
    }
    public abstract String makeNoise();
}

/* Dog class */
public class Dog extends Animal {
    // The Dog class must have a concrete makeNoise method.
    // Otherwise, it won’t compile.
    public String makeNoise(){
        return “woof!”;
    }
}
```
Representing Abstract & Concrete Classes

Abstract
- Animal
  - eat()
  - sleep()
  - name()

Concrete
- Cat
  - makeNoise()
  - eat()
- Lion
  - makeNoise()
- Dog
  - makeNoise()
- Wolf
  - makeNoise()
- SeeingEyeDog
  - getTrackingCode()
  - getNoise()