Abstract Classes
by David Matuszek
Edited by Diana Palsetia

Abstract methods
- You can declare an object without defining it:
  ```java
  Person p;
  ```
- Similarly, you can declare a method without defining it:
  ```java
  public abstract void draw(int size);
  ```
  Notice that the body of the method is missing
- A method that has been declared but not defined is an abstract method

Abstract classes I
- Any class containing an abstract method is an abstract class
- You must declare the class with the keyword `abstract`:
  ```java
  abstract class MyClass {...}
  ```
- An abstract class is incomplete
  - It has “missing” method bodies
- You cannot instantiate (create a new instance of) an abstract class

Abstract classes II
- You can extend (subclass) an abstract class
  - If the subclass defines all the inherited abstract methods, it is “complete” and can be instantiated
  - If the subclass does not define all the inherited abstract methods, it too must be abstract
- You can declare a class to be abstract even if it does not contain any abstract methods
  - This prevents the class from being instantiated
Why have abstract classes?

- Suppose you wanted to create a class `Shape`, with subclasses `Oval`, `Rectangle`, `Triangle`, `Hexagon`, etc.
- You don’t want to allow creation of a “Shape”
  - Only particular shapes make sense, not generic ones
  - If `Shape` is abstract, you can’t create a new `Shape`
  - You can create a new `Oval`, a new `Rectangle`, etc.
- Abstract classes are good for defining a general category containing specific, “concrete” classes

An example abstract class

- `public abstract class Animal {
  abstract String makeNoise();
}
- This class cannot be instantiated
- Any non-abstract subclass of `Animal` must provide the `makeNoise` method

Why have abstract methods?

- Suppose you have a class `Animal`, but it isn’t abstract
  - `Animal` should not have a `makeNoises()` method
  - Each subclass of `Animal` should have a `makeNoises()` method

- Now suppose you have a variable `Animal someAnimal`; where `someAnimal` refers some subclass object (such as a `Dog`)
  - e.g. `Animal someAnimal = new Dog();`

  - It is a syntax error to say `someAnimal.makeNoises()`
    - We need to convert it type of `someAnimal` to `Dog` in order to make it work

  - If `someAnimal` was an input parameter to a method, compiler can’t tell in advance what kind of value will be in the variable (i.e. object it is referring to)

Solution to Problem

- `abstract class Animal {
  abstract void makeNoises();
}
- class Dog extends Animal {
  void makeNoises() {
  …
  }
}

- `Animal someAnimal = new Dog();`
  - This is legal, because a `Dog` is a `Animal`
  - However, `Animal someAnimal = new Animal();` is no longer legal
- `someAnimal.makeNoises();`
  - This is legal, because every actual instance must have a `makeNoises()` method
Example Elaborated

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

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

/* Dog class */
public class Dog extends Animal {
    // The Dog class must have a concrete makeNoise method.
    // Otherwise, it won’t compile.
    public void makeNoise(){
        System.out.println("woof!");
    }
}
```

Example with Shapes

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

class Star extends Shape {
    void draw() { ... }
}

class Crescent extends Shape {
    void draw() { ... }
}

Shape someShape = new Star();
// This is legal, because a Star is a Shape
Shape someShape = new Shape();
// is no longer legal
someShape.draw();
// This is legal, because every actual instance must have a draw() method
```