Abstract Classes and Interfaces

Abstract methods

- You can *declare* an object without *defining* it:

```
Person p;
```

- Similarly, you can declare a *method* without defining it:

```
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:

```
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 int eat();
      abstract void breathe();
  }
  ```
- This class cannot be instantiated
- Any non-abstract subclass of Animal must provide the `eat()` and `breathe()` methods

Why have abstract methods?

- Suppose you have a class Shape, but it *isn't* abstract
  - Shape should *not* have a draw() method
  - Each subclass of Shape *should* have a draw() method
- Now suppose you have a variable Shape figure; where figure contains some subclass object (such as a Star)
  - It is *a syntax error* to say figure.draw(), because the Java compiler can't tell in advance what kind of value will be in the figure variable
  - A class "knows" its superclass, but doesn't know its subclasses
  - An object knows its class, but a class doesn't know its objects
- **Solution:** Give Shape an *abstract* method draw()
  - Now the class Shape is abstract, so it can't be instantiated
  - The figure variable cannot contain a (generic) Shape, because it is impossible to create one
  - Any object (such as a Star object) that *is* a (kind of) Shape *will* have the draw() method
  - The Java compiler can depend on figure.draw() being a legal call and does not give a syntax error

A problem

- `class Shape { ... }`
-
  ```
  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
- `someShape.draw();`
  - This is a syntax error, because *some* `Shape` might not have a `draw()` method
  - Remember: ***A class knows its superclass, but not its subclasses***

A solution

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