Abstract Classes and Interfaces
Java is “safer” than Python

- Python is very dynamic – classes and methods can be added, modified, and deleted as the program runs
  - If you have a call to a function that doesn’t exist, Python will give you a runtime error when you try to call it
- In Java, everything has to be defined before the program begins to execute
  - If you have a call to a function that doesn’t exist, the compiler marks it as a syntax error
  - Syntax errors are far better than runtime errors
  - Among other things, they won’t make it into distributed code
  - To achieve this, Java requires some additional kinds of classes
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`:
  ```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
Example abstract class

```java
public abstract class Pet {
    public abstract String makeSound();
    public void eat(Food food) {
        food.calories = 0;
    }
}
```

- This class cannot be instantiated
- Every subclass of Pet must provide an implementation of the `makeSound` method
- Eclipse examples: Pet.java, Food.java, Dog.java
Why have abstract classes?

• We know all **Pets** eat
  – The effect on the food is always the same!
  – Instead of implementing the **eat** method for each subclass of **Pet**, we can implement it once and inherit it

• On the other hand, different **Pets** make different sounds
  – We want to make sure that each subclass implements the **makeSound** method

• We don’t want to instantiate a **Pet** object because we don’t know what it is (it is abstract!)

• But we can have a variable of type **Pet** point to a subclass of **Pet** (e.g., **Dog**)
  ```java
  Pet p = new Dog();
  ```
Interfaces

• An interface declares (describes) methods but does not supply bodies for them interface
  
  public interface Shape {
    public double area();
    public double perimeter();
    public void draw();
  }

• All the methods are implicitly public and abstract
• You can add these qualifiers if you like, but why bother?
• You cannot instantiate an interface
• An interface is like a very abstract class – none of its methods are defined
• An interface may also contain constants (final variables)
Designing interfaces

• Most of the time, you will use built-in Java interfaces
• Sometimes you will want to design your own
• You would write an interface if you want classes of various types to all have a certain set of capabilities
• For example, you want to have different Shape classes with similar functionality
Implementing an interface

- You extend a class, but you implement an interface
- A class can only extend (subclass) one other class, but it can implement as many interfaces as you like
- Example:
  ```java
  public class Square implements Shape { ... }
  ```
- When you say a class implements an interface, you are promising to define all the methods that were declared in the interface
- Eclipse examples: Shape.java, Circle.java, Square.java, ShapeDemo.java
Partially implementing an interface

- It is possible to define some but not all of the methods defined in an interface:
  ```java
  abstract class Triangle implements Shape {
      public void perimeter() {...};
  }
  ```
- Since this class does not supply all the methods it has promised, it is an abstract class
- You must label it as such with the keyword abstract
- Eclipse example: Triangle.java
- You can even extend an interface (to add methods):
  ```java
  public interface FunkyShape extends Shape { ... }
  ```
Why have interfaces?

• **Reason 1:** A class can only extend one other class, but it can implement multiple interfaces
  – This lets the class fill multiple “roles”
  – Some examples in Java’s graphics packages

• **Reason 2:** You can write methods that work for more than one kind of class
  – Same as for abstract classes
Adapter classes

• What if an interface has a lot of methods but you only care about a few and don’t want to implement the rest?
• An adapter class implements an interface and provides empty method bodies
  – You can override only the methods you care about

```java
public class SomeShape implements Shape {
  public double area() { }
  public double perimeter() { }
  public void draw() { }
}
```

• This isn’t elegant, but it does work
• Java provides a number of adapter classes
Vocabulary

• **abstract method** – a method which is declared but not defined (it has no method body)
• **abstract class** – a class which either (1) contains abstract methods, or (2) has been declared abstract
• **instantiate** – to create an instance (object) of a class
• **interface** – similar to a class, but contains only abstract methods (and possibly constants)
• **adapter class** – a class that implements an interface but has only empty method bodies