Programming to an Interface Principle

Overview

- the word interface is overloaded
    - there is the concept of interface, but there is also the Java construct interface
    - you can program to an interface, without having to actually use a Java interface

- "Program to an interface" really means "Program to a supertype"
    - the declared type of the variables should be a supertype, usually an abstract class or interface
    - the objects assigned to those variables can be of any concrete implementation of the supertype
    - the class declaring them doesn't have to know about the actual object types

- do not declare variables to be instances of a particular concrete class
    - instead, commit only to an interface defined by an abstract class (interface or abstract)

- always program for the interface and not for the implementation
    - will lead to flexible code which can work with any new implementation of the interface

- programming to an interface is a common theme of the design patterns

Overview (cont'd)

- manipulating objects solely in terms of the interface is beneficial to clients

    - clients do not need to know the specific types of objects they use
        - as long as the objects adhere to the interface that clients expect

    - clients do not need to know the classes that implement these objects
        - they only know about the abstract class(es) defining the interface

- "programming to an interface" greatly reduces implementation dependencies between subsystems

- we can use interface types on variables, return types of methods or parameter types in a method

- the point is to exploit polymorphism by programming to a supertype so that the actual runtime object is not locked into the code

Polymorphism

- I know this is probably redundant, but just to make sure we are on the same page, imagine an abstract class Animal, with two concrete implementations, Dog and Cat

- programming to an implementation would be:

```
Dog d = new Dog();
d.bark();
```

- declaring the variable “d” as type Dog (a concrete implementation of Animal) forces us to code to a concrete implementation

Polymorphism (cont’d)

- programming to an interface/supertype would be:

```
Animal animal = new Dog();
animal.makeSound();
```

- we know it is a Dog, but we can now use the animal reference polymorphically

- even better, rather than hardcoding the instantiation of the subtype (like new Dog()) into the code, assign the concrete implementation object at runtime:

```
a = getAnimal();
a.makeSound();
```

- we do not know WHAT the actual animal subtype is
  - all we care about is that it knows how to respond to makeSound()

Abstract Classes vs. Interfaces

- with support of default methods in interfaces since the launch of Java 8, the gap between when to use an interface and when to use an abstract classes has been reduced

- variables in interfaces are public static final
  - abstract classes can have other access modifiers for variables (private, protected, etc.)

- methods in interfaces are public or public static
  - methods in abstract classes can be private and protected too

- utilize abstract classes to establish a relationship between interrelated objects
  - when you want to share code among several closely related classes then this common state or behavior can be put in the abstract class

Abstract Classes vs. Interfaces (cont'd)

- utilize interfaces to establish a relationship between unrelated classes
  - the interfaces Comparable and Cloneable are implemented by many unrelated classes

- utilize interfaces if you want to specify the behavior of a particular data type, but are not concerned about who implements its behavior

- utilize interfaces if you want to take advantage of multiple inheritance

- **one is not better than the other**

- you might create an interface and then have an abstract class implement that interface