Computer Science 225
Advanced Programming
Siena College
Spring 2017

Topic Notes: Abstract Classes and Interfaces

Abstract classes

Recall our earlier example:

See Example: Overriding

Here, we saw that the implementation of the `getName` method in class `Student` was sometimes, but not always overridden by implementations in classes that extended `Student`.

Suppose instead that we wanted to require that all classes that extend the `Student` class would provide their own specialized version of `getName`. If we include an actual implementation in the `Student` class, there is no way to require this.

However, we can replace the version in `Student` with an *abstract method*. An abstract method's header includes the `abstract` keyword, and does not include an implementation:

```
    protected abstract void getName();
```

Once we have done this, class `Student` must also be declared to be *abstract class* by adding the `abstract` keyword to its class header.

```
abstract class Student
```

See Example: AbstractClass

Some notes about abstract classes:

- An abstract class in Java is one which is allowed to include abstract method headers.
- An abstract class can still include *concrete methods* (*i.e.*, those with full implementations).
- An abstract class can contain declarations of instance and class variables.
- Any *concrete class* (*i.e.*, non-abstract class) that extends an abstract class will be required to provide an implementation for each abstract method. It would still be allowed to override any non-abstract methods.
- Since an abstract class is not a complete class definition, a program cannot instantiate an abstract class.

- It is good practice to include an `@Override` annotation for concrete implementations of abstract methods.
- Abstract classes are very useful to "factor out" some common implementation from a set of related classes, while requiring some additional methods whose implementations are necessarily specific to the derived classes.

In a UML class diagram, an abstract class is indicated with the `<<abstract>>` annotation with the class name, and abstract methods are indicated by italics.

Interfaces

Recall that Java does not allow multiple inheritance. Every class (except `Object`) must extend exactly one other. If a class does not explicity extend some class, it implicitly extends `Object`.

We also have seen that there are times when single inheritance does not capture the object hierarchy as well as we might like. Recall the huge list of shapes from a recent lab. All squares are rhombuses and all squares are rectangles, but it is not the case that either all rhombuses are rectanges or all rectangles are rhombuses. For the class diagram there, you had to choose one or the other.

Java's *interface* mechanism can help in these situations. An interface in Java is essentially a "completely abstract class" – it contains a list of method headers but no implementations. However, the syntax for both the definition and usage of interfaces differs from that of classes:

```
public interface MyInterface {
    void aMethod(int i);
}
public class MyClass implements MyInterface {
    public void aMethod(int i) {
        //implementation
    }
}
```

Note the `interface` keyword in place of the `class` keyword in the definition of the interface, and the `implements` keyword in place of the `extends` keyword in the usage of an interface.

Essentially, for a class `A` that implements an interface `B`, `A` is promising that it will provide implementations of all of the methods listed in `B`. Not to do so would result in a compile error. This allows a programmer to store a reference to an object of type `A` in a variable of type `B`, but will mean that the object when used through the variable of type `B` is restricted only to those methods listed in the `B` interface.

Some notes about interfaces in Java:

- A class can implement zero or more interfaces.

- A class that implements multiple interfaces separates the intefaces that it implements by commas.
- A concrete class that implements one or more interfaces must implement all methods defined in the interface.
- An abstract class that implements one or more interfaces can omit some or all of the implementations, but it essentially inherits any omitted methods from the interface(s) as new abstract method.
- Interfaces can extend other interfaces (using the `extends` keyword)

Advantages of interfaces:

- They encourage smart application design
- They promote readability
- They promote maintainability
- They allow flexibility, offering many of the benefits of multiple inheritance without the added language complexity (See: C++)

Some rules to keep in mind about interfaces:

- An interface cannot extend a class, only another interface.
- Interface methods are implicitly abstract.
- Interface methods may not be declared as `final`, since they must be implemented.
- Interface methods are implicitly `public` (as opposed to default protection).
- Interfaces can only define variables that are `public`, `static`, and `final`.

The fact that Java allows a class to implement multiple interfaces introduces the potential for name collisions when more than one interface includes methods of the same name.

- If the methods have different signatures, the methods are overloaded.
- If they have the same signature and return type, the methods are collapsed into one. That is, a single matching implementation satisfies all such interfaces.
- If they have the same signature but different return types, it will produce a compilation error.