Introduction to Object Oriented Programming

Roy Schwartz, The Hebrew University (67125)

Lecture 4:

Abstract Classes and Interfaces

Last Week

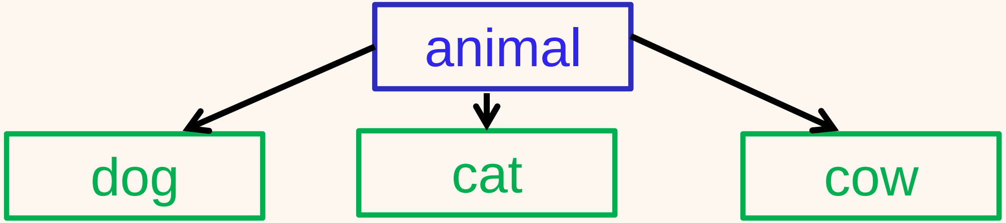
- Single Responsibility Principle
- Inheritance
- Overriding
- Polymorphism

Lecture 4a: Overview

- Motivation for using Abstract Classes
- Abstract Classes
- Interfaces
- More on Interfaces
- A few Examples

A Case Study

- Say we want to build a family of animals
  - Dogs, cats, cows, …
- Each animal has (amongst others) a heart, and can breath
- A reasonable way to do it is by defining a class hierarchy



Animal Class

```
/**
 * An animal class.
 * Animals breath and have a heart
 */
public class Animal {
        private Heart heart;

        public void breath() { … }
}
```

- Dog, Cat and Cow can **extend** Animal

Animals

- We would also like every animal to be able to speak
  - It makes sense to put the “speaking” code in the Animal class
- However, every animal makes its own sound
  - Dogs bark, cats meow, cows moo, …
  - How would the Animal.speak() method look?

Option I

- Don’t use a general Animal.speak() method
  - Let each specific animal class define its own speaking method
  - Dog.bark(), Cat.meow(), …

Problems with Option I

- This makes our design complicated
- This is conceptually wrong, since the speaking method in each class does the same thing (although differently)
  - Users have to know a different method for each class
  - It should have **the same API**!
- **Most importantly**: the benefits of using **polymorphism** are not exploited with this option
  - There is no way for objects of extending Animal classes to make a sound when accessed via an Animal reference

Option II

- Implement an empty Animal.speak() method that does nothing

```
public void speak() { }
```

- Let all extending classes override this method

```
public class Dog extends Animal {
    public void speak {
        System.out.println(“haw”);
    }
}
```

Problems with Option II

- Better, but still
- What if some class forgets to override speak()?
  - Better to **force** classes to override speak()
- How does a general Animal object sound like?
  - Animal animal = **new** Animal(…);
  - animal.speak(); // nothing happens!

Lecture 4b: Overview

- Motivation for using Abstract Classes
- Abstract Classes
- Interfaces
- More on Interfaces
- A few Examples

Solution – Abstract Classes

- Abstract classes are classes from which we cannot create an instance
  - Defined using the **abstract** keyword
  - This way, no Animal object can be created

```
public abstract class Animal { … }

Animal animal = new Animal(…);   // Compilation error
```

Abstract Classes

- Abstract classes allow us to define **abstract** methods
  - Methods with no implementation
  - Every (non-abstract) sub-class of an **abstract** class must implement all **abstract** methods
  - Otherwise, code won’t compile

```
public abstract class Animal {
    // An abstract speak method.
    // To be implemented by Animal sub-classes.
     public abstract void speak();
}
```

Notice the syntax:
No ‘{‘,’}’, no code, just ‘;’

Sub-Classes

```
public class Dog extends Animal {
    // Implementing the abstract speak() method.
     public void speak() {
          System.out.println(“haw”);
    }
}

public class Cat extends Animal {
}
```

No **abstract** keyword in implementation

No speak() implementation: *compilation error*

More on Abstract Classes

- A sub-class of an **abstract** class can also be **abstract**
  - In this case, it behaves the same as any other **abstract** class (i.e., we cannot create an instance of this class)
  - It doesn’t have to implement any of the **abstract** methods (although it can)
- An **abstract** class can define regular data members and methods, just like any other class
- **static** methods **cannot** be declared **abstract**

More on Abstract Classes (2)

- What happens when we try to invoke **super**.speak() when speak() is **abstract**?
  - **Compilation error!**
- Abstract methods cannot be declared **private**
  - Only **public** or **protected**
  - Why?

Abstract Class – what is it Good for?

- Cases where the top level(s) of our inheritance tree are not concrete classes
  - It makes no sense to create an instance of a general animal
- When we want to force an API on a group of inheriting classes
  - But the parent class cannot provide a reasonable implementation for this API

Lecture 4c: Overview

- Motivation for using Abstract Classes
- Abstract Classes
- Interfaces
- More on Interfaces
- A few Examples

Interfaces

- An *interface* is a reference type, similar to a class, that can *only* contain
  - Constants (**final static** data members)
  - Abstract methods
- Interfaces cannot be instantiated
  - They can only be *implemented* by classes or *extended* by other interfaces

Interface Example

```
/* An interface for printable objects.  */
public interface Printable {
      // A print method
       public void print();
}

public class Document implements Printable {
      // Implementing the Printable.print() method
       public void print() {
               …
      }
      …
}
```

Interface keyword

No need for the abstract keyword

implements keyword

print() method implementation

Interface Example

```
    ….
    public static void main(String args[]) {
            Document d = new Document();
            d.print();
            Printable p = new Printable();
    }
}
```

Calling print() method

Compilation error

Why Use Interfaces?

- Interfaces represent *contracts* that classes accept
  - Unlike classes, they do not represent something in the world, but a **requirement** that is shared among various classes of various types
- Examples:
  - *Printable*: for classes that can be printed
  - *Comparable*: for classes that can be compared to other classes
  - *Clonable*: for classes that can be cloned
- Interfaces speak about *what*, not about *how*

Interfaces as APIs

- As you recall, we are always trying to build classes with *minimal API*
- Interfaces can be used to define the API used by a set of classes
  - A group of classes that all implement the same interface
  - In this case, the only **public** methods these classes define are the ones defined by the interface

Interfaces and Modifiers

- Interfaces cannot declare **private** or **protected** methods
  - Only **public**
- Interfaces cannot declare data members
  - Only **final static** data members

Extending Interfaces

- Interface can have sub-interfaces
  - Using the **extends** keyword
- This is useful in cases where we want to define several types of *contracts* or *behaviors* that share a few methods
- Classes that implement a sub-interface must implement both the methods of the sub-interface and the methods of the super-interface

Sub-Interfaces

```
public interface MyInterface {
        public void superFoo();
}

public interface MySubInterface extends MyInterface {
       public int subFoo();
}

public class MyClass implements MySubInterface {
        public void superFoo() { … }
        public int subFoo() { … }
}
```

Lecture 4d: Overview

- Motivation for using Abstract Classes
- Abstract Classes
- Interfaces
- More on Interfaces
- A few Examples

Interfaces and Contracts

- The API contract of an interface specifies what every implementing class must provide
- Any implementing class can extend the contract
  - In the sense of specifying more and offering more
  - But may **not offer less**
- An implementing class may require fewer pre-conditions
  - I.e., handle inputs that the interface considers illegal
  - But **never more pre-conditions** (i.e. consider more input illegal)

Analogy: Real World Contracts

- Say you go to a shop and want to buy a product for some amount of money
- The seller, if she wishes, can give you more than just this product
  - A present, a more advanced alternative, etc.
  - But she **cannot** give you less (part of the product, an inferior alternative)
- Moreover, the seller can accept fewer pre-conditions
  - I.e., offer a discount
  - She cannot, however, ask for more money than is stated on price stamp

Interfaces and Contracts Example

```
public interface FactorFinder {
        /** @return a > 1 factor of the given positive integer n. Return n iff n is prime */
        public int factorOf (int n);
}

public class SmallestFactorFinder implements FactorFinder {
        /** @return the smallest prime factor of the integer n */
        public int factorOf (int n) {
             for (int i = 2 ;  ;  ++i)
                if (n%i ==  0)
                    return i;
        }
}
```

Offer more
(smallest factor)

Interfaces and Contracts Example (2)

```
public interface ArrayManipulator {
        /** Perform some manipulation on array. @param array – a non empty array */
        public int manipulate(int[] array);
}

public class ArrayPrinter implements ArrayManipulator {
        /** Print array. Do nothing if array is empty.  */
        public int  manipulate(int[] array) {
             for (int i: array)
                System.out.println(i);
        }
}
```

Fewer pre-conditions (array may be empty)

Interfaces and Multiple Inheritance

- Interfaces are not part of the class hierarchy
  - Although they work in combination with classes
- In Java, a class can **extend** only one class but, it can **implement** any number of interfaces

public class MyClass **implements** *MyInterface1, MyInterface2, …*

- Therefore, objects can have **multiple types**
  - The type of **their own class,** the types of **all the classes** they extend (directly and indirectly) and the types of **all the interfaces** that they implement (also, directly and indirectly)

Polymorphism and Interfaces

- The benefits of using polymorphism apply to interfaces as well
- In other words, an object of class C can be accessed via a reference of any of its **super classes**, or any of its **interfaces**

Polymorphism Example

```
public class MyClass extends MyParentClass implements Printable, Clonable { … }
```

All the following are legal:

```
MyClass myObj = new MyClass();
MyParentClass  myParentObj = myObj;
Object obj = myObj;
Printable myPrintableObj = myObj;
Clonable myClonableObj = myObj;
```

Access via Parent / Interface Reference

- One important thing to remember is that access by a parent / interface reference only gives us access to the API of the parent class / interface
  - Other methods / data members defined by the class itself are **not accessible**
- For example, *myPrintableObj1* (of type *Printable*) can only call the *print*() method, but not other methods of MyClass, such as *clone*() (defined by the *Clonable* interface)

Interfaces and Abstract Classes

- On the face of it, interfaces and abstract classes are similar
  - Both allow the creation of class hierarchies
  - Both force requirements on classes that use them
  - Both cannot be instantiated
- It is not always clear which one we should use

Interfaces and Abstract Classes (2)

- If the *is-a* relation holds between two types, then you should use inheritance (**extends**)
  - A dog **is an** animal, a car **is a** vehicle
- If the common property is more of a contract, or a specific behavior defined by one class and used by another, use interface (**implements**)
  - Printability, clonability, comparability, …
- In cases of uncertainty, favor **interfaces**

Lecture 4e: Overview

- Motivation for using Abstract Classes
- Abstract Classes
- Interfaces
- More on Interfaces
- A few Examples

Common Java Interfaces

- The java collection framework (soon to come) holds many useful ***data structures*** tools
- ***interface*** *java.util.Collection*
  - A general purpose data structure
  - add(), remove(), size(), …
- ***interface*** *java.util.List* ***extends*** *Collection*
  - A collection that allows access by index
  - get(index), set(index, value), …

Collection Interfaces

- These classes do have a “class-like” character
  - A list is a concrete thing, not exactly a contract
- Nevertheless, these interfaces represent the “what” and not the “how”
  - There are many ways to implement a list: linked list, array list, …
  - All these implementations share the same API
  - As a result, the type that represents this API (List) is declared interface and not abstract class

Common Java Abstract Classes

- ***abstract class*** *java.lang.Number*
  - A general number class
  - intValue(), floatValue(), …
  - Subclasses: Integer, Double, …

Real Example

```
public class Double extends Number implements Comparable,Clonable

public class Integer extends Number implements Comparable,Clonable

public class String implements Comparable

/** Apply bubble sort algorithm on arrayToSort. */
public void bubbleSort(Comparable[ ] arrayToSort) { … }

/** Create an array of n version of toClone. */
public Clonable[ ] cloneNTimes(Clonable toClone, int n) { … }
```

How does it Work?

```
Comparable[ ] array1 = new Double[] { … };

Comparable[ ] array2 = new String[] { … };

Comparable[ ] array3 = new Integer[] { … };

// Sort array1, array2 and array3
bubbleSort(array1);
bubbleSort(array2);
bubbleSort(array3);
```

So far…

- Abstract Classes
  - Define a family of classes
  - Cannot be instantiated

- Interfaces
  - Defines a contract accepted by implementing classes
  - A class can implement as many interfaces as it wishes

Next Week

- Introduction to Design Patterns
- The Façade Design Pattern