Introduction to Object Oriented Programming

(Hebrew University, CS 67125 / Spring 2014)

Lecture 4

Abstract Classes

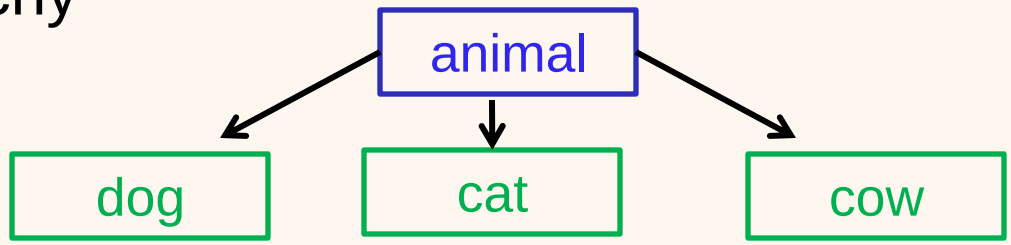
Interfaces

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 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(), …

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:
  - This makes our design complicated
  - This is conceptually wrong, since the speaking method in each class does the same thing (although differently)
    - It should have **the same API**!
  - Users have to know a different method for each class

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:
  - This makes our design complicated
  - This is conceptually wrong, since the speaking method in each class does the same thing (although differently)
    - It should have **the same API**!
  - Users have to know a different method for each class

❑ A bigger problem will be introduced next week (stay tuned!)

Option II

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

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

- Let all extending classes override this method

Option II – Problems

- Better, but still
- What if some class forgets to override speak()?
  - Better to **force** classes to override speak()

Option II – Problems

- 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!

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 is 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();
}
```

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 {
}
```

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

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 exactly 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 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?

More on Abstract Classes (2)

- What happens when we try to invoke **super**.speak() when speak() is abstract?

- **Compilation error!**

Abstract Classes and Modifiers

- 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 since to create an instance of a general animal

Abstract Class – what is it Good for?

- Cases where the top level(s) of our inheritance tree are not concrete classes
  - It makes no since 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

So far…

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

Interfaces

- An *interface* is a reference type, similar to a class, that can *only* contain
  - Constants (**final static** members)
  - Abstract methods

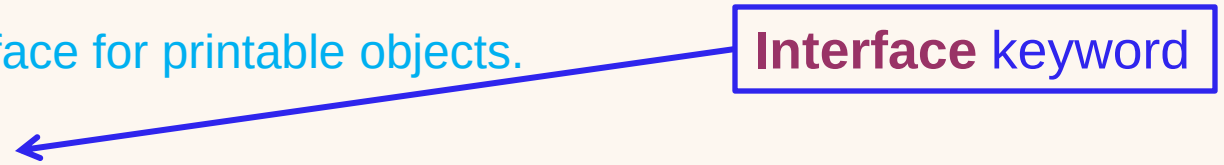
- Interfaces cannot be instantiated
  - They can only be *implemented* by classes or *extended* by other interfaces

Interface Example

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

Interface Example

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



Interface Example

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

Interface keyword

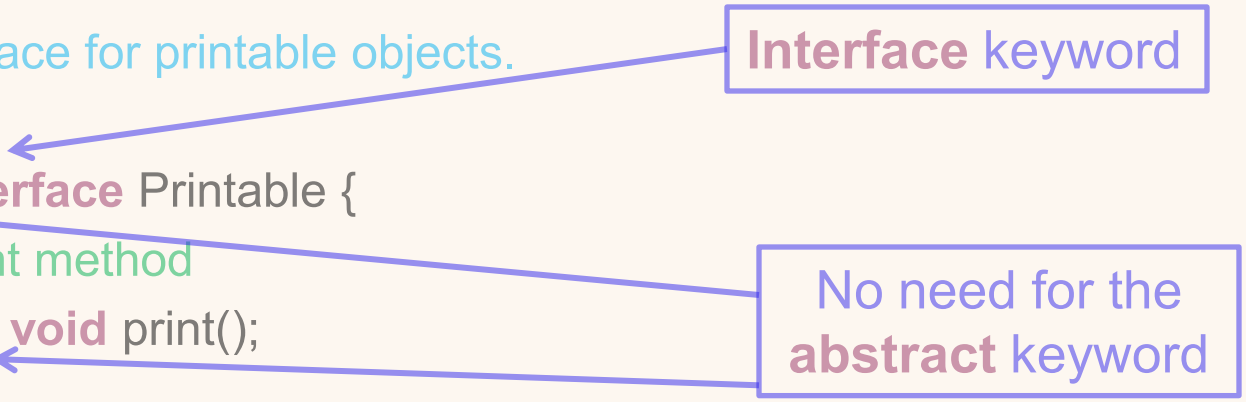
No need for the **abstract** keyword

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 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

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

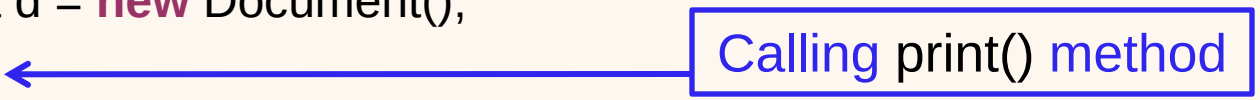
print() method implementation

Interface Example

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

Interface Example

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



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 some **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
  - The only **public** methods these classes define are the ones defined by the interface

Interfaces and Modifiers

- Interfaces cannot be declared **private** or **protected** methods
  - Only **public**

- Interfaces cannot declare members
  - Only **final static** members

Interfaces and Multiple Inheritance

- Interfaces are not part of the class hierarchy
  - Although they work in combination with classes

- In Java, a class can inherit from 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** and the types of **all the interfaces** that they implement

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

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

      public int subFoo() { … }
}
```

Sub-Interfaces

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

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

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

      public int subFoo() { … }
}
```

Sub-Interfaces

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

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

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

      public int subFoo() { … }
}
```

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)

Interfaces and Contracts Example

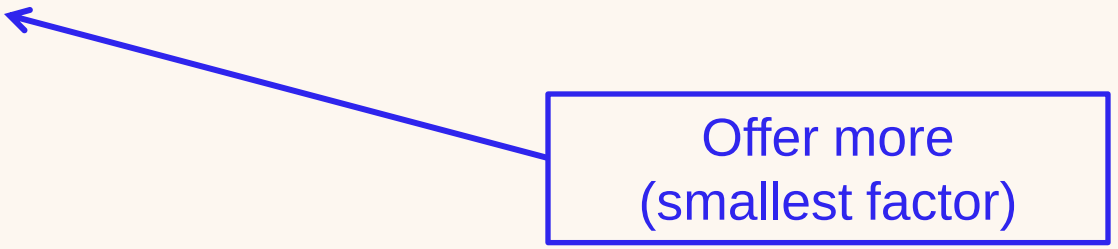
```
public interface FactorFinder {
        /**
         * @return a > 1 factor of the given positive integer n. Return n iff n is prime
         */
        public int factorOf (int n);
}

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;
        }
}
```

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;
        }
}
```



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);
        }
}
```

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 Abstract Classes

- On the face of it, interface 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, …

Interfaces and Abstract Classes (3)

- In cases of uncertainty, favor interfaces
  - You can only **extend** one class, but **implement** as many classes as you want

Common Java Interfaces

- The java collection framework (see in 2 weeks) 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

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, …

So far…

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