Inheritance

Lecture 31

Borrowing from slides by Alan Hu, Kurt Eiselt, Paul Carter, and Tamara Munzner
News

- Homework 3 is online
  - Due December 4

- Final exam: Dec 11, 8:30 am, OSBO A
  - Length: 2:30
Reading Assignments

- Reading for this week: interfaces
  - Edition 2: Ch 11.1-11.3

- Reading for this/next week: inheritance
  - Edition 3: Ch 10
  - Edition 2: Ch 13
Recap: Inheritance

- You can declare a new class as an extension of an existing one.
- The new class automatically inherits all the instance fields and methods of the old class.
- The new class can add/change fields and methods.

```java
public class ChildClass extends ParentClass {
    ...
    put any additional fields and methods here
    ...
}
```
Recap: Inheritance Terminology

- The child class **extends** the parent class.
- The parent class is called the **superclass**.
- The child class is called the **subclass**.
- The child class (subclass) inherits stuff from its parent (superclass), so it has more stuff (fields and methods).
- Mnemonic: your parents are your **superiors**.
Recap: Inheritance Example

- Suppose you are writing a simulation/game.
- There will be various things on the playing field, so you’ll have a Thing class, with instance fields like its position and direction.
- Some of the things will move, so you might write a MovingThing class, which is similar, but also has attributes like velocity.
- You might make an AcceleratingThing class, to make it easy to model, e.g., a spaceship.
Recap: Accessing the Superclass

- Even though the subclass has all the instance fields and methods of the superclass, Java still thinks of it as a separate class. So, the subclass can’t directly access anything private!

- What do you do?
  - Use the public accessor/mutator methods.
  - Have the superclass make things public.
  - Have the superclass make things protected.
Recap: Superclass Constructor

- Recall the special use of \texttt{this} in constructors?
  - \texttt{this} as \underline{first line of constructor} calls a different constructor for the same object
  - E.g., UBCStudent class

- Similar trick to call a superclass constructor using the \texttt{super} keyword:
  - \texttt{super} call must be first line of constructor
Recap: Inheritance Hierarchy in UML

```
Thing

MovingThing

AcceleratingThing
```

“is-a” relation
(This is where the terms subclass and superclass really come from.)
Objectives

- More practice with the idea of inheritance
- Review scope and learn how to override methods.
- Learn how to access the superclass:
  - Fields and methods
  - Constructors
  - Overridden methods
Overriding

- You can declare a new class as an extension of an existing one.
- The new class automatically inherits all the instance fields and methods of the old class.
- The new class can add/change fields and methods.

```java
public class ChildClass extends ParentClass {
    // anything new here is added to the ChildClass

    // anything with same signature as in ParentClass overrides the ParentClass
}
```
Recap: Variable Scope

- *Scope* tells you which declarations you can see from which points in the program.
  - The scope of a variable is the places in the program where the variable can be accessed.
  - Or you can think of scope as the places where you can see some variable.
Recap: Variable Scope

- Packages contain classes
  - Within a package, everyone can see all classes and non-private stuff within classes
- Classes contain methods and fields
  - All methods can see other methods and fields in same class
Recap: Variable Scope

- Methods contain statements, executed one-by-one
  - You can see declarations that happened already
  - Parameters are like declarations that happen at start of method.
  - Curly braces define a compound statement, which limits scope

Example:

```java
public static void main(String[] args) {
    int a = 0;
    System.out.println(args[0]);
    int b = 2*a;
    int a = 3;  // Not OK!
    b = c+a;  // Not OK! -- can’t see c
    int c;
}
```
Recap: Variable Scope

- Compound statement limits scope
  - You can see out, but not in.

Example:
```java
public static void main(String[] args) {
    int a = 0;
    int b = 0;
    {
      // start of a block
      int a = 2; // shadows other a
      int c = a+b;
      System.out.println(c); // prints 2
    }
    System.out.println(a); // prints 0
    System.out.println(c); // Not OK!
}
```
Shadowing and Overriding

These are basically the same concept:

- If you have two declarations of the same signature (name and parameters), the closer declaration wins.
Shadowing and Overriding

These are basically the same concept:

- If you have two declarations of the same signature (name and parameters), the closer declaration wins.

Overriding

```java
public class Class1{
    public void a(double b) {...}
}
public class Class2 extends Class1 {
    public void a(double b) {...} // override!
}
```
Shadowing and Overriding

- These are basically the same concept:
  - If you have two declarations of the same signature (name and parameters), the closer declaration wins.

- Overriding

  ```java
  public class Class1 {
      public void a(double b) {...}
  }

  public class Class2 extends Class1 {
      public void a(int b) {...} // no override
  }
  ```
Shadowing and Overriding

These are basically the same concept:

- If you have two declarations of the same signature (name and parameters), the closer declaration wins.

Overriding

```java
public class Class1 {
    public void a(double b) {...}
}

public class Class2 extends Class1 {
    public void a(double x) {...} // override?
}
```
Calling Overridden Methods from the Subclass

- Assume method `a` in `Class2` needs to call method `a` in `Class1`
  - But that method is hidden...

- This can be done using `super` keyword
  - Works like `this` keyword, but on superclass:

```java
public class Class2 extends Class1 {
    public void a(double x) {
        ...
        // do some stuff
        super.a(10.0); // any parameter...
        ...
        // do some more stuff
    }
}
```
Interfaces vs. Superclasses

We learned these as completely separate concepts:

- An **interface** is a contract, specifying some methods that must be implemented by any class that claims to implement the interface.

- A **superclass** is a class from which other classes can **inherit** methods and instance fields, so we can reuse the superclass’s implementation.
Interfaces vs. Superclasses

But they have similarities...

- Both allow creating different, new classes that share some of the same methods, e.g.
  - `Double` and `WordCount` both implement `Comparable`, so they both have `int compareTo()`.
  - `Thing` and `MovingThing` both inherit from `Thing`, so they have e.g., `setXPos()`.
Interfaces vs. Superclasses

But they have similarities…

- Both allow declaring references that can point to different kinds of objects, e.g.,
  - `Comparable x;`
  - `x = new Double(3.14);`
  - `x = new WordCount(“the”);`
  - `Thing y;`
  - `y = new Thing(10,20,0.0,1.0);`
  - `y = new MovingThing(1,1,0,1,1,1,1);`
Interfaces vs. Superclasses

But they have similarities...

- Therefore, both allow polymorphism, e.g.,
  - `Comparable x;`
  - ...
  - `if (x.compareTo(...)) ...`
  - `Thing y;`
  - ...
  - `y.display(...);`
Interfaces vs. Superclasses

But they have similarities…

- They even have similar UML:
Interfaces vs. Superclasses

They have differences, too…

- Java allows implementing multiple interfaces

```
<<Interface>>
Comparable
...
```

```
<<Interface>>
Pettable
...
```

```
Cat
...
```

OK!
Interfaces vs. Superclasses

But they have differences, too…

- Java does NOT allow multiple inheritance…

Not OK! Not allowed in Java.
Interfaces vs. Superclasses

Other important differences:

- An interface provides **no** implementations.
- **Everything** in a (super)class is implemented.
Interfaces vs. Superclasses

Other important differences:

- To implement interface, a class must implement everything.
- Subclasses automatically inherit superclass implementation. Can optionally override.
Interfaces vs. Superclasses

Other important differences:

- You **cannot create objects** of an interface type (only references).
- You **can create objects** of the (super)class type. (You can create references, too.)