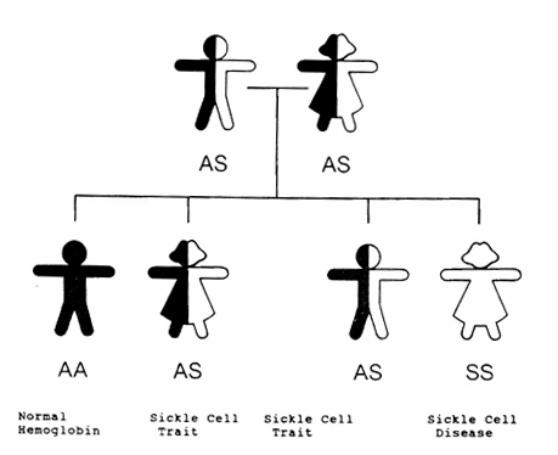
Introduction to Object Oriented Programming

(Hebrew University, CS 67125 / Spring 2014)

Lecture 3

Inheritance and Overloading



What is a Class?

- Think of a class as something that you can describe in 2-3 words at most
  - Dog, bicycle, printer, calculator, button, file reader
  - This description is usually a good candidate for the **class name**

- A class should have additional functionality compared to other existing classes
  - A dog's *name* should not be a class, but can be represented as a **String**
  - On the other hand, a dog's *tail* might deserve a class of its own

What is a Class (2)?

- A class should be a general concept, from which we can create specific instances
  - Though *a class of static methods* is a counter-example
- A concrete and specific item should be defined as an object of a more general class
  - Pluto is a Dog object, MyPrinter is a Printer object

Single Responsibility Principle

- A class should have a single responsibility
  - That responsibility should be **entirely encapsulated** by the class
- All its services should be narrowly aligned with that responsibility

Single Responsibility Principle
Counter Example

- Consider a class that both **reads** a file a text file and **count** the number of words in it

```
public class ReaderAndCounter {
    // Read a text file
    public void read() { … }

    // Count the number of words
    public void count() { … }
}
```

Single Responsibility Principle
Counter Example

- A class that has more than one responsibility is **more likely to change**
  - We might want/need to change either *read*() or *count*()

- Changes are bug-prone, require re-testing of our program, and are generally expensive
  - The larger the responsibility of the class, the harder it is to change it

Doing more than one Thing

- Our program usually does more than one thing
  - Some class needs to handle it

- This class should be a **manager** class which uses other classes to perform the actual tasks
  - Assuming API is minimal, changes will only affect the classes of the specific tasks, not the manager

Manager Class

```
/**
 * A manager class.
 * Reads a file and counts its words.
 */
public class Manager {
    // Manage reading and counting
    public void manage() {
        Reader reader  = new Reader(…);
        String[] lines = reader.read();
        ….
        Counter counter = new Counter(…);
        counter.count(lines);
    }
}
```

```
/**
 * A class that reads a text file.
 */
public class Reader {
    // Read a text file
    public String[] read() { … }
}
/**
 * A class that counts the
 * number of words.
 */
public class Counter {
    // Count number of words
    public void count(String[])
        { … }
}
```

Relations between Classes

- Various relations exist between different classes
- Object-oriented languages allow us to define these relations in our code

Has-a Relation

- The most basic relation between classes is the *has-a* relation (also called **composition**)
- This relation is formed where one object “belongs” to another object
  - A person **has a** name, bicycles **have** wheels, etc.
- Composition is implemented in java in a very straight-forward manner
  - Using **data members**

Composition Example

```
public class Person {
    // A person has a name and a mother (it composes them)
    private String name;
    private Person mother;

    …
}
```

Is-a Relation

- Another important relation between classes is the *is-a* relation

- Consider a class that is a more specific version of an existing class
  - A *student* **is a** *person*
  - Students share all the properties of persons (they have a name, they have a mother, they can walk, talk, etc.)
  - They add their own set of properties (they have their student id, they can take exams, etc.)

Inheritance

- OO languages define a way to represent the *is-a* relation – inheritance
- Class *A* inherits (or **extends**, in java) class *B*, if *A is B*
  - *A* is denoted *B*'s **sub-class**, *B* is denoted *A*'s **super-class**
- Class *A* has all the properties (i.e., members, methods and constructors) of class *B*, and can also add its own data

Inheritance Example

```
/**
 * A student is a person that has a student ID, and can take exams
 */
public class Student extends Person {
    /** A student has (composes) a student id */
    private int id;

    /** Take an exam */
    public void takeExam() { … }

    …
}
```

Instance-of Relation

- The *is-a* relation should not be confused with the *instance-of* relation
  - Pluto is also a dog
  - Not a **type** of dog, but a **concrete** dog

- How is *instance-of* represented in java?

More about Inheritance

- Inheritance is recursive
  - class *A* can extend class *B*, which extends class *C*, …
- Inheritance is transitive
  - If *A* **extends** *B* and *B* **extends** *C* ➔ *A* (implicitly) extends *C*
  - No need to specify the **extends** keyword again

More about Inheritance (2)

- In java, each class can be the super-class of any number of classes (including none)

- In contrast, each class can extend at most one class
  - If no super-class is mentioned (via the **extends** keyword), the class is the sub-class of java.lang.Object

java.lang.Object

- Every class in java extends java.lang.Object
  - Either directly:

```
public class Person extends Object { … }
```

  - Or indirectly (implicitly extending Object):

```
public class Person { … }
```

  - Or transitively

```
public class Student extends Person { … }
```

- java.lang.Object is the only java class that doesn't extend any other class

java.lang.Object Properties

- java.lang.Object provides a few valuable methods for every java class
  - toString() – return a string representation of this object
  - equals(Object other) – does this object equals other?
  - …

Private Data and Inheritance

- **private** data (members, methods and constructors) are not accessible in a subclass
  - Trying to access them results in a compilation error
  - Much like any other class, **public** data is accessible to the subclass

Person Class Example

```
public class Person {
    // A person has a name and a mother (it composes them)
    private String name;
    private Person mother;

    public String getName() {
        return this.name;
    }
    …
}
```

Inheritance Example

```
/**
 * A student is a person that has a student ID, and can take exams
 */
public class Student extends Person {
    /** A student has (composes) a student id */
    private int id;

    /** Take an exam */
    public void takeExam() {
        System.out.println(name);          // Compilation error –
                                           // name is private
        System.out.println(getName());     // Using a public method is ok
    }
    …
}
```

Protected Modifier

- Members, methods and constructors can get the **protected** modifier
  - Alternative to **public** or **private**

- **protected** properties are accessible to this class and all sub-classes (including transitive sub-classes)

Person Class Revised

```
public class Person {
    // A person has a name and a mother (it composes them)
    protected String name;
    protected Person mother;

    public String getName() {
        return this.name;
    }
    …
}
```

Student Example Revised

```
/**
 * A student is a person that has a student ID, and can take exams
 */
public class Student extends Person {
    /** A student has (composes) a student id */
    private int id;

    /** Take an exam */
    public void takeExam() {
        System.out.println(name);          // Now it works –
                                           // name is protected
        System.out.println(getName());     // Using a public method is ok
    }
    …
}
```

Protected?

- Using the **protected** modifier should be done with care

- Although using it is often more convenient, the reasons for not using the **private** modifier apply here as well
  - **protected** data is part of the class's API
  - It is harder to understand how to extend a class
  - Harder to modify a class that uses **protected** data

Protected (2)?

- Generally, we should prefer the **private** modifier whenever possible
  - Alternatives to the **protected** modifier (such as getters and setters) are usually available
- Nevertheless, sometimes it is necessary to use **protected** data
  - Knowing when comes with expertise
  - See TA for some examples

Overriding

- Extending a class allows to modify the behavior of an existing (**public** or **protected**) methods
  - This is called **overriding**
- The procedure is simple – re-implement the method using the same signature
  - Same *name*, *return value* and *parameters*
- Calling the method from the sub-class results in calling the new implementation
  - Calling it from the parent class will call the original one

super

- Sometimes we want to use the parent implementation, and add some more operations of our own

- Using the **super** keyword gives us access to our parent-class
  - **super**.method() calls the super-class implementation
  - In a constructor, **super**(…) calls the super constructor

Student Example Revised

```
/**
 * A student is a person that has a student ID, and can take exams
 */
public class Student extends Person {
    /** A student has (composes) a student id */
    private int id;

    /**
     * A constructor that receives the student's name and id.
     */
    public Student (String name, int id) {
            // Call parent constructor with name
            super(name);
            this.id = id;
    }
```

Student Example Revised

```
    /** Modify the behavior of getName() to return the name twice */
    public String getName() {
            return super.getName()+ " " + super.getName();
    }
    …
} // end of Student class
```

Using the Student Class

```
/**
 * A tester for the student class
 */
public class StudentTest {
    public static void main (String[] args) {
        Student stud = new Student("OOP stud", 12345);
        Person pers = new Person("normal person");

        System.out.println(stud.getName());// print "OOP stud OOP stud"
        System.out.println(pers.getName());// print "normal person"
    }
}
```

Inheritance, what is it Good for?

- Inheritance represents the *is-a* relation
  - Class *A* should not extend class *B* if *A* is **not** a *B*
- Inheritance also serves as a code-reuse mechanism
  - Class *A* can use class *B*'s methods without re-implementing them
- Nevertheless, other code-reuse alternatives exist (**composition**)
  - Code-reuse is **not** a good reason to use inheritance
  - More on this to come

So far…

- What is a class?
  - Single Responsibility Principle
- Relations between objects
  - Composition (has-a)
  - Inheritance (is-a)
- Inheritance
  - Protected methods, overriding, super