Introduction to Object Oriented Programming

Roy Schwartz, The Hebrew University (67125)

Lecture 3:

Inheritance and Polymorphism

Last Week

- Scope
- Instance vs. static
- Minimal API
- Information Hiding

Lecture 3a: Overview

- Single Responsibility Principle **שאלה לדיון: איך נגדיר מחלקה?**
- Inheritance
- More on Inheritance
- Protected Modifier
- Overriding
- Polymorphism
- Polymorphism and OOP

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 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 האמנם?
- 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 text file and **counts** the number of words in it

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

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

האם התכנון נכון?
שומר על עקרונות הכימוס?

Single Responsibility Principle

Counter Example

- A class that has more than one role 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 our 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 word counter class. */
public class Counter {
        // Count number of words
        public void count(String[]){  … }
}
```

Lecture 3b: Overview

- Single Responsibility Principle
- Inheritance
- More on Inheritance
- Protected Modifier
- Overriding
- Polymorphism
- Polymorphism and OOP

Relations between Classes

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

Has-a Relation

קומפוזיציה

- 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 features of persons (they have a name, they have a mother, they can walk, talk, etc.)
- They add their own set of features (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 a type of*** *B*
  - *A* is denoted *B*'s **sub-class**, *B* is denoted *A*'s **super-class**
- Class *A* has all the features (i.e., data members, methods and constructors) of class *B*, and can also add its own features

Inheritance Example

```java
/** A person with a name and
    a mother */
public class Person {
    private String name;
    private Person mother;

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

```java
/** Student: A person with student
    ID that can take exams */
public class Student
        extends Person {
    private int id;

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

    …
}
```

```java
Student myStud = new Student( … );

// Running a method of the parent class
   (Person)
System.out.println(myStud.getName());

// Running a method of the sub-class
   (Student)
myStud.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?
  - By creating a new instance (**object**) of the Dog class

For example, if you have a class Dog that extends a class Animal,
then Dog is said to have an "is-a" relationship with Animal. This means every Dog is an Animal.

OTHH - using dog instanceof Animal will return true if dog is an instance of the Dog class
or any of its subclasses, given that Dog is a subclass of Animal.

Lecture 3c: Overview

- Single Responsibility Principle
- Inheritance
- More on Inheritance
- Protected Modifier
- Overriding
- Polymorphism
- Polymorphism and OOP

More on 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*
  - A *CS student* is a *student*, but she is also a *person*
  - No need to specify the **extends** keyword again

More on Inheritance (2)

- In java, each class can be the **super-class** of any number of classes (including none)
  - This does not apply to **primitives**
- 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 the *Object* class

The *Object* Class

- 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 { … }
```

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

Override Annotation: When overriding methods from the Object class (like equals, hashCode, or toString), some developers prefer to make the inheritance from Object explicit for clarity.

Object Class Properties

- The *Object* class provides a few valuable methods for each java class
  - toString() – returns a string representation of this object
  - equals(Object *other*) – does this object equal *other*?
  - …

private Data and Inheritance

- **private** elements (fields, methods and constructors) are not accessible to subclasses תזכורת: איך בכל זאת נוכל לגשת אליהם?
  - Trying to access them results in a compilation error
  - Much like any other class, **public** data is accessible to the subclass

Person Class Example

```
/**
   * A person with a name and a
   * mother
   */
public class Person {
      private String name;
      private Person mother;

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

```
/** 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);   // error (name is a private
                                        // field of the parent class)
            System.out.println(getName()); // a public method
      }
      …
}
```

Lecture 3d: Overview

- Single Responsibility Principle
- Inheritance
- More on Inheritance
- Protected Modifier
- Overriding
- Polymorphism
- Polymorphism and OOP

protected Modifier

- Data members, methods and constructors can get the **protected** modifier
  - Alternative to **public** or **private**
- Sub-classes (including transitive sub-classes) have access to **protected** elements they inherit

פתרון שני אפשר לבעיית הפרטיות

Person Class Revised

```
/**
   * A person with a name and a
   * mother
   */
public class Person {
      protected String name;
      protected Person mother;

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

```
/** 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 is works!
            System.out.println(getName()); // a public method
      }
      …
}
```

protected?

- Using the **protected** modifier should be done with care
- Although using it is often more convenient, the reasons for not using the **public** 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

כאשר לא נרצה לתת אפשרות לגשת לשדה פרטי או לשנותו,
אלא רק המחלקה היורשת תוכל לדעת עליו מידע

Lecture 3e: Overview

- Single Responsibility Principle
- Inheritance
- More on Inheritance
- Protected Modifier
- Overriding
- Polymorphism
- Polymorphism and OOP

Overriding

- Extending a class allows us to modify the behavior of inherited (**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 an object of the sub-class type results in calling the new implementation
  - Calling it from an object of type parent class will call the original one

Student Example Revised

```
/** A parent class*/
public class Parent{
        public void foo() {
                System.out.println("P");
        }
        …
}
```

```
/** A sub class*/
public class Child extends Parent {
        // Override foo()
        public void foo() {
                System.out.println("C");
        }
        …
}
```

```
Parent p = new Parent();
p.foo();
Child c = new Child();
c.foo();
```

Output:
P
C

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 person with a name and a
    mother */
public class Person {
    protected String name;
    protected Person mother;
    public Person(String name){
        this.name=name;
    }
    public String getName() {
        return this.name;
    }
    …
}
```

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

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

```
    /** 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());
		System.out.println(pers.getName());
	}
}
```

Output:
OOP stud OOP stud
Normal person

super and the Default Constructor

- In order to create a sub class object, one of parent object constructors must be called
- Either explicitly
  - Using **super**(…)
- Or Implicitly
  - Implicit calls are allowed only if the parent class has a default (parameter-less) constructor
  - Otherwise, not calling **super**() is a compilation error

Super and Constructors

```
public class A {
    public A() {
        system.out.println("A");
    }
    …
}

public class B {  אין בנאי דפולטיבי ריק
    public B(int b) {
        system.out.println("B");
    }
    …
}
```

```
public class C extends A {
    public C() {
        // No super() – implicit
        // call to A's default c'tor
        system.out.println("C");
    }
    …
}

C c = new C();
```

Output:

A

C

```
public class D extends B {
    public D() {
        // No super() and no default
        // constructor to B:
        // compilation error
        system.out.println("D");
    }
    …
}
```

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** זוכרים את ההבדל בין הרכבה לירושה?
  - Code-reuse is **not** a good reason to use inheritance
  - More on this to come

Lecture 3f: Overview

- Single Responsibility Principle
- Inheritance
- More on Inheritance
- Protected Modifier
- Overriding
- Polymorphism
- Polymorphism and OOP

Polymorphism

- One of the most important principles of object-oriented programming
- Stands in the basis of many other object-oriented principles
  - Reference vs. Content
  - Encapsulation
  - Inheritance
  - **Extensibility**
  - **Modularity**

Polymorphism

Definition

- From Greek – “many shapes”
  - A biological term in which an organism or species can have many different forms or stages
- In object oriented programming, polymorphism refers to the ability of an object to **take on many forms** (i.e., **types**)
- This ability is realized when **extending** a **class**

Polymorphism

Example

```
public class Animal {
      public String speak() { return “ha?”; }
      public void eat(int calories) {
                System.out.println(“Yammy”);
      }
}
```

```
public class Dog extends Animal {
        public String speak() {
                return “woof”;
        }
        public Person getOwner()  { … }
}
```

```
public class Cow extends Animal {
        public String speak() {
                return “moo”;
        }
        public void getMilk()  { … }

        public void eat(int calories)  {
                System.out.println(“YamYam”);
        }
}
```

Polymorphism

Example (2)

```
Cow myCow = new Cow();
Dog myDog = new Dog();
Animal myAnimal = myCow;
myAnimal.speak();
myCow.speak();
myCow.getMilk();
myAnimal.getMilk();
mydog.eat();
myCow.eat();
myAnimal.eat();
```

The Cow object takes the form of an animal (**Polymorphism**)

Animals can speak (myAnimal is a cow, so output

A cow is also an animal, so it can speak (“moo”)

Cows give milk

But animals can’t! Even though this object is **actually a cow** (**Compilation Error**)

All animals can *eat* (whether Animal.eat() was overridden or not)

Polymorphism is Useful

```
/** A function that get an animal argument of any type and make a sound. */
public void makeAnimalSpeak( ???? ) {
    ????.speak();
}

Cow myCow = new Cow();
Dog myDog = new Dog();
Animal myAnimal = new Cow();

makeAnimalSpeak(myCow);
makeAnimalSpeak(myDog);
makeAnimalSpeak(myAnimal);
```

It's the concrete object that counts!

moo

woof

moo

Polymorphism is Useful

```
/** Get an array of animals and makes each of them make a sound. */
public void makeAnimalsSpeak(        ????        ) {
                ????

}

Animal[] animals = new Animal[3];
animals[0] = myDog;
animals[1] = myCow;
animals[2] = myAnimal;

makeAnimalsSpeak(animals);
```

Reminder:

```
Dog myDog = new Dog();
Cow myCow = new Cow();
Animal myAnimal = myCow;
```

צריך לשקול מדוע לעשות זאת?

```
woof
moo
moo
```

Which Method Runs?

- When we call myObj.foo(), it's the concrete type of foo that is called
  - myAnimal.speak() // Moo for cows, woof for dogs
- On the other hand, we are only allowed to call methods defined in the **reference type**
  - Animal myAnimal = **new** Cow(…);
  - myAnimal.getMilk(); **// Error. Animals don't give milk.**

Shadowing

- If a sub-class defines a field with the same name as the parent class, it does not override it
  - Here, it's the **reference** type that matters
- Same goes for **static** data members and methods
  - Why?

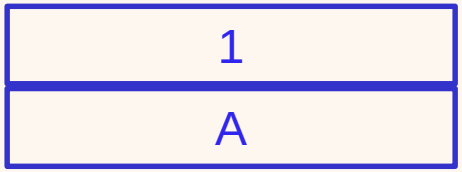
Shadowing Example

```
public class A {
        public int myInt = 1;
        public static void staticFoo() {
                System.out.println("A");
        }
}
```

```
public class B extends A {
        public int myInt = 2;
        public static void staticFoo() {
                System.out.println("B");
        }
}
```

```
A a = new B();
System.out.println(a.myInt);
a.staticFoo();
```

Use **A**.staticFoo() or **B**.staticFoo()



אם זה לא היה הפלט,
למה מלכתחילה ירושה?

Shadowing?

- Shadowing variables is generally considered bad practice
  - Confusing code
- This is hardly surprising, as variable shadowing requires defining **non-private** data members

Lecture 3g: Overview

- Single Responsibility Principle
- Inheritance
- More on Inheritance
- Protected Modifier
- Overriding
- Polymorphism
- Polymorphism and OOP

Polymorphism and Extensibility

- Polymorphism is so important because it allows us to build a program that is easy to extend
- Say now we build a new Animal class

```
class Goat extends Animal { … }
```

- Goat is an Animal, and thus it follows the Animal class API
- Consequently, makeAnimalsSpeak() will continue to work perfectly for objects of this class as well
  - Our program is **easy to extend**

Polymorphism and Flexibility

- Polymorphism allows us to modify the content type during runtime
- This allows us to modify the **behavior** of a variable during runtime

```
Animal myAnimal = new Cow();
myAnimal.speak();

myAnimal = new Dog();
myAnimal.speak();
```

moo

woof

Real Example

```
public class Number { … }

public class Double extends Number { … }

public class Integer extends Number { … }


/**
   * Sort array of numbers.
   */
public static void sort(Number[] numbers) {…}
```

How does it Work?

```
Double d_array[] = {5.7, -1.244, 8.0};

Integer i_array = {4, 2, 12};


sort(d_array);          // d_array is now: {8.0,5.7,-1.244};

sort(i_array);          // i_array is now: {12,4,2};
```

sort() doesn't care about the concrete type!
All that matters is that it **extends** Number

Polymorphism and Minimal API

- Recall: *when delivering a program, we want to share as few details as possible*
  - **Minimal API**
- We mentioned this principle when we discussed the **private** modifier
  - **Information Hiding**
- Polymorphism allows us to take this principle a step further
  - **Program to interface, not to implementation**

Program to interface, not to implementation

- When defining an API, we should attempt at using types higher at the class hierarchy אלא אם מחלקת אב לא מכירה מתודה של בן
- If our code only uses the API of the higher type, there is no reason to use more concrete classes
  - makeAnimalSpeak(**Animal** animal) and **not**

    makeCowSpeak(**Cow** cow)

Program to interface, not to implementation (2)

- **Generality**: the same code works for more objects (all Animals)
- **Extensibility**: adding a new type (**class** Goat **extends** Animal) is automatically supported by this code
- **Easier modification**: clients remain unaware of the specific type of objects they use
  - This Allows replacing implementation without affecting clients
  - This greatly reduces implementation dependencies
- **Easy to use code**: there are fewer higher level types than lower level ones

So far…

- What is a class?
  - Single Responsibility Principle
- Relations between objects
  - Composition (has-a)
  - Inheritance (is-a)
  - Class – object (instance-of)

So far…

- Inheritance
  - Protected methods, overriding, super
- Polymorphism
  - Objects can take multiple forms
  - Overriding, Shadowing
  - Polymorphism and other OOP principles

Next Week

- Abstract Classes
- Interfaces