Introduction to Object Oriented Programming

Roy Schwartz, The Hebrew University (67125)

Lecture 2:

More OOP Principles

Last Week

- Introduction to the course
- What is object oriented programming?
- Objects
- Classes
- Types
- Constants

Lecture 2a: Overview

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

Local Variables

- Variables defined inside a method
- Can be defined anywhere inside a method
  - At the top of the method
  - inside an **if**/**while**/**for** block, etc.
- Much like data members
  - Each local variable has a **type**
    (Either a **primitive** or a **reference**)
  - Local variables can be declared **constants**

ניתן לשנות שדות של אובייקט קבוע, אך לא את האובייקט עצמו

```
void foo() {
        int a;

        …

        if ( … ) {
            char b = 'a';
        }
        …
        final String c = "hello";
}
```

Java Scope

- A scope is any piece of code that lies between brackets (‘{‘, ‘}’)
  - Class content: **class** MyClass { … }
  - Methods: **public static void** main(String args[]) { … }
  - Loops, conditions: **if** (…) { … }, **while** (…) { … }
- The scope of a variable determines its visibility and its accessibility
  - Local variables are **not** accessible from outside their scope
  - Local variables **are** accessible from an internal scope

Scope Example

- Local variables are **not** accessible from outside their scope

```
if ( … ) {
    int internalNum = 5;

    …
}

System.out.println( internalNum );
```

Error!

Scope (2)

- Local variables **are** accessible from an internal scope

```
int externalNum = 5;
if ( … ) {
    System.out.println( externalNum );
}
```

Namespace Pollution

Define variables in the most internal scope

שיקולי הנדסת תוכנה -
היכן נכון להגדיר משתנים?

- When writing code, you should try your best to declare your variables in the **most internal scope**
  - Declaring a variable in an external scope where it is never used makes the code **harder to understand**
  - Consequently, **harder to maintain** and **update**

Namespace Pollution

Variables Declared in an External Scope for no Reason

דוגמה לתכנון: זווית סיבוב של רחפן בעת פנייה

```
String myStr = null;
while (…) {
    if (…) {
        for (…) {
            myStr = …;
            System.out.println(myStr)
            …
        }
    }
}
```

*Hard to **understand**, **maintain** and **update***

Namespace Pollution – Solution

Define variables in the most internal scope

```
while (…) {
    if (…) {
        for (…) {
            String myStr = …;
            System.out.println(myStr)
            …
        }
    }
}
```

Namespace Pollution Disclaimer

- There are performance issues with running code in each loop iteration
  - Running the same code over and over is a waste of time and resources
- In the context of declaring variables, this is a problem only if
  - The **same** object is created at each iteration
  - This object is **complex** (i.e., containing many data members)

```
for (…) {
        // some code
}
```

Lecture 2b: Overview

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

The `static` Modifier

- The **static** modifier associates a variable or method with the **class** rather than an **object**
  - Can be applied to both variables and methods

מתי נכון להצהיר על משתנה או פונקציה סטטיים בתוך מחלקה?
למה זה נכון?

Static Members

- A variable that is declared **static** is associated with the class itself and not with an instance of it
- Static variables are also called ***class*** *variables*
  - *Non-static* variables are called ***instance*** variables
- We use **static** data members to store information that is not associated with a given **object**, but is relevant to the **class**

באיזה מצב נרצה לעשות דבר כזה?

Example

Number of Objects Counter

```
class Dog {
    // Count the number of dogs. This counter is not specific to some
    // Dog instance, but to the Dog class
    static int nDogs = 0;
    String name;
    int nSiblings;

    Dog(…) {
        // Dog.nDogs is increased each time a new Dog is created
        Dog.nDogs += 1;
        ...
    }
    ...
}
```

Example (2)

```
class Dog {
    static int nDogs = 0;

    Dog(){
        Dog.nDogs += 1;
    }

    public static void main(String[] args) {
        Dog dog1 = new Dog();
        System.out.println("nDogs (dog1): " + dog1.nDogs);
        Dog dog2 = new Dog();
        System.out.println("nDogs (dog1): " + dog1.nDogs);
        System.out.println("nDogs (dog2): " + dog2.nDogs);
    } // main
} //class
```

This is the right way to go

כאשר נרצה שהמחלקה תהיה מעודכנת כמה עצמים נוצרו ממנה

For a static member:
nDogs (dog1): 1
nDogs (dog1): 2
nDogs (dog2): 2

This will issue a warning

מה הסיכון בכך? כיצד נמנע ממנו?

Example (3)

```
class Dog {
    int nDogs = 0;

    Dog(){
        nDogs += 1;
    }

    public static void main(String[] args) {
        Dog dog1 = new Dog();
        System.out.println("nDogs (dog1): " + dog1.nDogs);
        Dog dog2 = new Dog();
        System.out.println("nDogs (dog1): " + dog1.nDogs);
        System.out.println("nDogs (dog2): " + dog2.nDogs);
    } // main
} //class
```

For a non-static member:

nDogs (dog1): 1
nDogs (dog1): 1
nDogs (dog2): 1

כאן המחלקה "לא יודעת" כמה עצמים נוצרו ממנה,
והאחריות לעדכון תהיה על כל אחד מהמופעים שלה

Static Methods

- Methods that are declared as **static** do not act upon any particular object
  - They cannot access any instance (non-static) member
- **static** methods can be used to access **static** members
- They can also encapsulate a given task / algorithm that is independent of a given object

מתי נכון לעשות כזו מתודה?
האם לא כדאי לשמור מחלקה בנפרד לכאלו מתודות?

Static Method Example

```
class Dog {
    // Count the number of dogs
    static int nDogs = 0;
    String name;
    int nSiblings;

    Dog(…) {
        // Dog.nDogs is increased
        // each time a new Dog is
        // created
        Dog.nDogs = Dog.nDogs + 1;
        ...
    }

    // Get number of Dogs
    static int getDogsCounter() {
        return Dog.nDogs;
    }
    …
}
```

Why should a Method be Declared Instance (non-Static)?

- A method is associated with a specific object if it requires access to some of its members
  - And/or if it requires access to other **instance** methods
- If these conditions do not hold, than the method is not related to any specific object
  - Only to the class
  - This is a good indication that it should be declared **static**

What’s Wrong with this Code?

```
class Dog {
        public static void main(String[] args) {
                int nDogs = 0;
                Dog dog1 = new Dog();
                nDogs += 1;
                System.out.println("noOfInstances (dog1): "  + nDogs);
                Dog dog2 = new Dog();
                nDogs += 1;
                System.out.println("noOfInstances (dog2): "  + nDogs);
        } // main
} //class
```

- Logical Structure
- Code Repetition
- Permissions

אין עדכון סטטי על כל יצירת מופע

A Class of Static Methods

- We can write a class that is a collection of **static** methods
- Such a class isn't meant to define new type of objects
  - It is just used as a library for utilities that are related in some way

Example

A Math Class

```
/*
 * A library of mathematical methods.
 */
class Math {
    // Computes the sine of a given angle.
    static double sin(double x) { ... }

    // Computes the natural logarithm of a given number.
    static double log(double x) { ... }
    ...
}
```

Lecture 2c: Overview

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

Sharing our Code

- In the real world, reusing our code is highly desired
  - “Code” could refer to a stand-alone software, a software module, or even a single class
  - Our code can be used by we *ourselves*, *Colleagues*, *General public*, *Customers* , …
- In order to make our code attractive, **usability** and **user-friendliness** is one of the key features of any code we write

API

- *Application programming interface* (*API*) is the programming gateway to our code
  - Which methods should be used and how
- Each piece of code we deliver should contain information about how to use it
  - What are the classes, members and methods
  - What are the relations between the classes (see next week)
  - How to use the code

נסו לחשוב על הגדרת תאימות של קלט ופלט
קריאה וכתיבה של קבצים
קריאה מתוך, או כתיבה לתוך בסיס נתונים

Minimal API

- Software programs tend to be complex units
  - Even simple programs can reach thousands of lines of code

חשבו על שתי דרכים
לצמצום הבעיה הנ"ל

- When delivering a program, we want to share as few details as possible
  - A **minimal API**

למה?

- Most implementation details should not be revealed

אם ככה, איך אעבוד עם קולגות לצוות?

Why not Share?

- The more information we provide about our code, the harder it is for users to learn how to use it
  - Fewer details are easier to grasp

ועדיין, איזו בעיה מציבה כתיבת הערות מרובה?
מדוע כדאי שהקוד "יתעד" את עצמו?

- More importantly, providing details about our code makes it **harder** for us **to modify** it later

למה זה נכון?


5:45


CASIO
SUN 6-30
10:58 50
MUDMAN


9:42
slide to unlock

Example

Time Class

```
/* A time class. Represents time of day. Allows comparison between times. */
class Time {
    // time of day
    int hour, minute, second;

    // A constructor that sets the current time of day
    Time() { ... }

    // Is other time before this time? This method uses the convert() method
    boolean isBefore(Time other) { … }

    // A helper method: converts time to num of seconds from start of day
    int convert() { … }
}
```

Time Class API

- The Time class is used for comparing between times
  - Why should users of this class know about the internal time representation?

```
int hour, minute, second;
```

  - Why should they know about the internal convert method?

```
int convert() { … }
```

- This information is not required for using the class
  - Being exposed to it actually makes it **harder** to use it

Time Class API (2)

- Say we deliver the code, and people start using it
  - *Success!*
- Sometimes later, we want to upgrade our system
  - Change from 24h to AM/PM
  - Stop using the hour/minute/seconds format אילו שינויים נדרשים בכל אחד מהמקרים?
  - Stop using the convert() method
- The basic functionality of our code remains the same
  - The internal, technical details are changed

Time Class API (2)

- People that know our code, have to “forget” about the old API, and learn new API
  - Hard, frustrating, bug prone
- Pieces of code that use our internal representation **have to be modified**!
  - Even though our code still does the same thing, just differently
  - Changing code is expensive, time-consuming and bug-prone

Example

Time Class Minimal API

```
/* A time class. Represents time of day. Allows comparison between times. */
class Time {
    // time of day
    Time();

    // Is other time before this time? This method uses the convert() method
    boolean isBefore(Time other);
}
```

נקי, פשוט

Lecture 2d: Overview

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

Information Hiding

- One of the key components in object-oriented programming
- Provides a formal way to supply users only with the minimal API required for working with our code

Modifiers

- Java (and other OO languages) allows to define each data member and method as either **public** or **private**
- **public** data members/methods are visible to everyone
  - Objects from every class can access them
- **private** data members/methods are only visible to objects in the containing class
  - Objects from other classes cannot use them
  - Trying to do so results in an **error**

Private and Public

- Data members, methods (instance or **static**) and constructors can all be declared **public** or **private**
  - What happens when there is no modifier?
  - Classes can be declared **public** (but **not private**)

See later in the course

Example

Time Class Improved

```
/*
 * A time class. Represents time of day. Allows comparison between times.
 */
public class Time {
    // time of day
    private int hour, minute, second;

    // A constructor that sets the current time of day
    public Time() { ... }

    // Is other time before this time? This method uses the convert() method
    public boolean isBefore(Time other) { … }

    // A helper method: converts time into num of seconds from start of day
    private int convert() { … }
}
```

Example

Time Class Improved

```
    // A helper method: converts time into num of seconds from start of day
    private int convert() {
        hour = …;                    // Methods inside the Time class have access
                                     // to private members.
    }
}
```

Users only See the Public API

```
/* A time class. Represents time of day. Allows comparison between times. */
public class Time {
    // A constructor that sets the current time of day
    public Time();

    // Is other time before this time? This method uses the convert() method
    public boolean isBefore(Time other);
}
```

private methods and members
(as well as method code!) are invisible

Example

Using the Time Class

```
/* A tester for the time class */
public class TimeTester {
    public static void main(String args[]) {
        Time t1 = new Time();                    // ok (Constructor is public)
        Time t2 = new Time();                    // ok (Constructor is public)

        System.out.println(t1. isBefore(t2)); // ok (isBefore() is public)

        System.out.println(t1.hour);             // Compilation error.
        t2.second = 2;                           // Compilation error.
        int converted = t2.convert();            // Compilation error.
    }
}
```

What should be Declared Private?

- A general rule-of-thumb is: all your data members should be declared **private**
  - Very few exceptions to this rule: mostly **static final** data members such as Math.*PI*
- At design time, decide what is the general (minimal) API your code provides
  - Make all other methods **private**

Lecture 2e: Overview

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

Getters and Setters

- Say we have a Person class with a name data member
  - We want to allow other classes to know the name of each Person
  - We might also like other classes to be able to modify name
  - But name is a data member, so it should be declared **private**
- Solution: use **public** getter and setter methods
  - getName() and setName()
  - Initial value is set during construction

Person Class

```java
/* A person class. */
public class Person {
	// A person's name
	private String name;

	// A constructor that gets the
	// person's name
	public Person(String personName) {
		name = personName;
	}

	// Name getter
	public String getName() {
		return name;
	}

	// Name setter
	public void setName(String newName) {
		name = newName;
	}
} // end Person class
```

Using Person Class

```
/* A tester for the Person class */
public class PersonTester {
	public static void main(String args[]) {
		Person p1 = new Person("John");

		System.out.println(p1.getName());  // Alternative to p1.name

		p1.setName("Ben");                 // Alternative to p1.name = …
	}
}
```

Why use Getters and Setters?

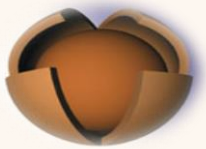
- We might want to prevent objects from other classes to modify the value
  - Provide only a getter method (**no setter**)
- We can add other stuff to the getter and/or the setter
  - Sanity checks, Conversion, etc.
- Most importantly: we can modify our implementation at a later stage, **without changing the API**
  - Person.name can be modified to be an array of chars, while API (*getName*()) stays the same

More on Information Hiding

- Don’t reveal your implementation details by using very indicative names
  - This applies to whether a getter method retrieves a saved value or calculates it
  - Use *getDifference*() and not *calculateDifference*()
- And also to which data structure you are using
  - *getDogs*() and not *getDogs**LinkedList***()

Private is not Secret!

- A common misconception is that **private** means secret
  - Sensitive information (e.g., passwords) should not be stored in **private** data members
- Some java mechanisms can be used to access private data (see later in the course)
- The **private** modifier is used for **better design**
- If you want to protected your data, **encrypt** it
  - More to come next year



Encapsulation

- The grouping of related ideas into **one unit**, which can then be referred to by a **single name**
  - Saves/organizes computer memory
  - Saves human memory – represents a **conceptual chunk** that can be considered and manipulated as a **single idea**

Encapsulation and Information Hiding

- Encapsulation states that we should put data members along with the methods that operate on these data members
- It also states that the internal implementation of each class should be hidden
  - **Information hiding**
- More to come later in the course

So far…

- Scope
  - Local variables
  - Scope – a block of code inside { … }
  - Access is **permitted** inside inner blocks, **denied** from outside

- Namespace Pollution
  - Define variables in the most **internal** scope

So far…

- Static Members
  - One copy per class
- Static Methods
  - Only access static members
- A class of static methods
  - General purpose utilities

So far…

- Information hiding
  - Easier to use code
  - Easier for us to modify it
- **private** and **public**
  - Members should generally be declared **private**
  - Use getters and setters
- Encapsulation

Next Week

- Single Responsibility Principle
- Inheritance
- Polymorphism