Introduction to Object Oriented Programming

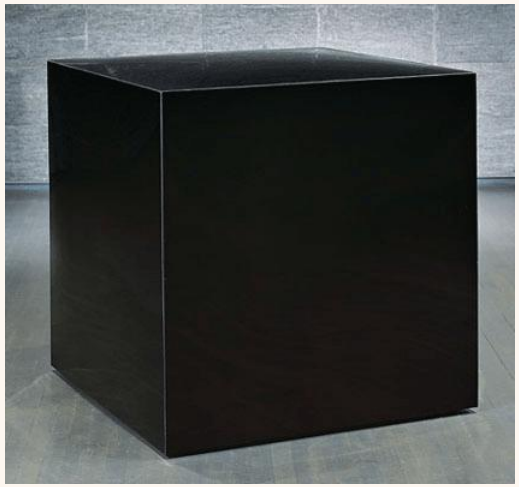
(Hebrew University, CS 67125 / Spring 2014)

Lecture 2

Scope

Encapsulation and Information Hiding

Instance vs. Static



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
  - Variables (member or local) are not accessible from outside their scope

Scope

- Variables (member or local) are not accessible from outside their scope

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

    …
}

System.out.println(internalNum);    // Compilation error – internalNum
                                    // is inaccessible
```

Scope (2)

- Variables (member or local) **are** accessible from an internal scope

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

Namespace Pollution

Define variables in the most internal scope

- When writing code, you should always 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**

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

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

Namespace Pollution

Define variables in the most internal scope

- There are minor performance issues with declaring a variable in each loop iteration
  - They are hardly ever meaningful
  - Mainly in cases where the variable is of a “heavy” type, and changes in every loop are minor

- heavy class – which contains a lot of fields and requires a lot of memory

The `Static` Modifier

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

- Methods that are declared as **static** do not act upon any particular object
  - They cannot access any non-static member
  - They just encapsulate a given task/algorithm

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 variables to store information that is not associated with a given object, but is relevant to the class

Example

Number of Objects Counter

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

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

    ...
}
```

Static Methods

- Methods that are declared as **static** do not act upon any particular object
  - They cannot access any 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

Static Method Example

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

    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;
    }
    …
}
```

When should a Method be non-static?

- A method is associated with a specific object if it has access to some of its members
  - And/or if it has 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**

Why should Static Methods be Encapsulated in any Class?

- Logical structure of our code
  - getDogsCounter() should be a part of the **Dog** class
  - **Understandable** code

- Permissions
  - **static** methods of class A can always call other **static** methods from class A
  - This is not always the case with **static** methods from other classes
  - See later today

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) { ... }

    ...
}
```

So far…

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

We Write Code for People to Use it

- Usability and user-friendliness is one of the key features of any code we write
- “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
  - Costumers
  - …

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

Minimal API

- Software programs tend to be complex units
  - Even simple programs can reach thousands of lines of code
- When delivering a program, we want to share as few details as possible
  - A **minimal API**
- Most implementation details should not be revealed

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

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

Example
A 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 before(Time other) { … }

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

Problems with Time

- 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

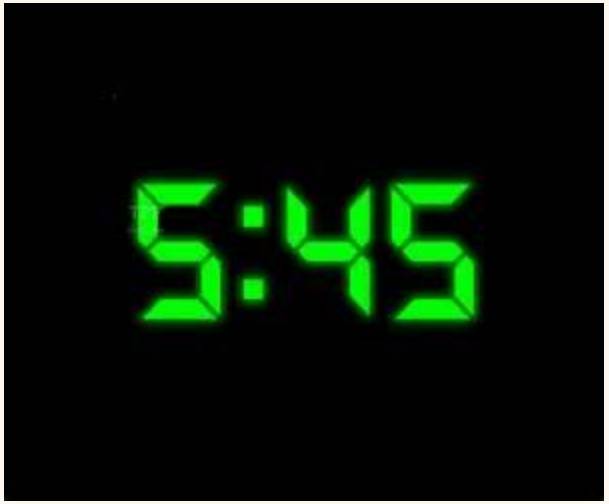
Problems with Time (2)

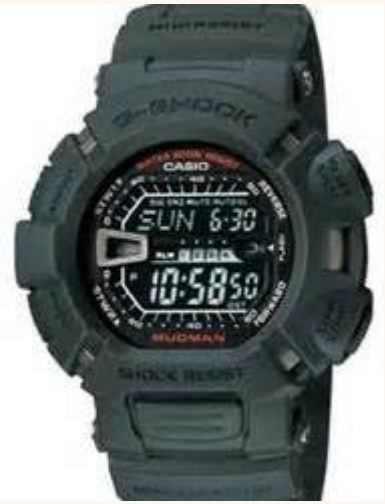
- Say we deliver the code, and people start using it
  - *Success!*
- Sometimes later, we want to upgrade our system
  - Change from 24h to AM/PM
  - Stop using the hour/minute/seconds format
  - Stop using the convert() method
  - ….
- The basic functionality of our code remains the same
  - The internal,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

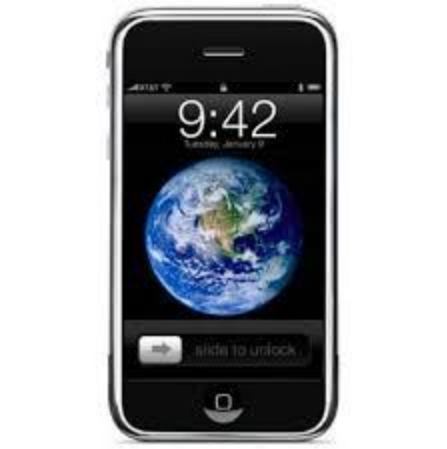
Problems with Time (2)

- 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


5:45


CASIO
SUN 6:30
10:58 50
MUDMAN
SHOCK RESIST


9:42
slide to unlock

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** members/methods are visible to everyone
  - Objects from every class can access them
- **private** members/methods are only visible to objects in the containing class
  - Objects from other classes cannot use them
  - Trying to do so results in a **compilation error**

Private and Public

- Members, methods (instance or **static**) and constructors can all be declared **public** or **private**

- Classes can be declared **public** but **not private**
  - What happens when there is no modifier? 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 before(Time other) { … }
```

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.
    }
}
```

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.
        Time t2 = new Time();                    // ok.

        System.out.println(t1.before(t2));       // ok.

        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** members such as Math.*PI*

- At design time, decide what is the general (minimal) API your code provides
  - Make all other methods **private**

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

```
/*
 * 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;
    }
```

Person Class

```
    // 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 = …

        System.out.println(p1.getName());
    }
}
```

Why use Getters and Setters?

- We might not want to allow 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

- 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** 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…

- 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