ITI 1121. Introduction to Computing II *

Marcel Turcotte
School of Electrical Engineering and Computer Science

Version of January 13, 2013

Abstract

- Object-oriented programming
  - Encapsulation

*These lecture notes are meant to be looked at on a computer screen. Do not print them unless it is necessary.

Object-Oriented Programming — OO

- OO and programs design
- encapsulation
- information hiding
- the interface of class
- an object has a state
- composing with objects
- classes vs objects
- anatomy of class (class implementation)
- variables: instance variables, class variables, local variables and parameters (formal and effective/actual)
- methods: class methods and instance methods
- constructors -1-
- *ad hoc* polymorphism (method overloading)
- inheritance (software re-use)
- constructors -2-
- polymorphism: inheritance, abstract classes and interfaces

Object-Oriented Programming and software design

What is object-oriented programming?

The design of a software system is an **abstract activity**, in the sense that you cannot see or touch the product that is being built.

The design of software systems in terms of classes and objects, **makes this process more concrete**, you can actually draw diagrams that correspond to classes and objects, or imagine the classes and objects as physical entities.

This conceptualization allows us to think about the interactions amongst parts of the software more easily (naturally).

Abstraction

An **abstraction** allows us to ignore the details of a concept and to focus on its important characteristics.

The term "**black-box**" is often used as an analogy for an abstraction.

Procedural Abstraction

The programming methodology used throughout most of CS I is called "structured programming".

Its main mean of abstraction is a **procedure** (called **routine** or **function** in some languages, or **method** in Java).

If the method is well designed, and depends only on its formal parameters, than a method can be used as a black box — you shouldn't be concerned about its implementation.

For example, one can use a sorting method without knowing about the particular algorithm that it uses — in most circumstances.

$\Rightarrow$ This way to structure a program works best when the data and the number of types are small.

Data abstraction

Records and **structures** were amongst the first forms of data abstraction.

They allow to model the data and to handle the data as a unit.

Consider modeling a coordinate, point, in a two dimensional plane.

Data abstraction

A **point** could be declared as follows:

```
class Point {
    int x;
    int y;
}
```

which would be used as follows:

```
static Point add( Point a, Point b ) {
    Point result = new Point();
    result.x = a.x + b.x;
    result.y = a.y + b.y;
    return result;
}
```

Calling the method add:

```
p3 = add( p1, p2 );
```

Data abstraction

Imagine determining if two points are equal.

Without data abstraction, every time the information about a particular point to be passed to a method, you would need to list all the variables that characterize a point.

```
boolean equal( int x1, int y1, int x2, int y2 ) {
    // ...
}
```

Data abstraction

Whenever a change to the design has to be made (adding or removing variables, 2D to 3D) you would have to update the program in several places.

The type system is not used as effectively as it could, as shows the following example, where the formal parameters are equal(int x1, int y1, int x2, int y2), but the effective parameters are: equal(x1, x2, y1, y2).

Not to mention that it can be quite tedious to have to list all the necessary variables.

```
int equal( int x1, int y1, int y3, int x2, int y2, int y3 ) {
    // ...
}
vs
int equal( Point p1, Point p2 ) {
    // ...
}
```

Data abstraction

However!

Consider writing classes modelling shapes, idioms such as this one are likely to occur at many places inside the programs:

```
if ( isCircle( shape ) ) {
   drawCircle( shape );
} else if ( isSquare( shape ) ) {
   drawSquare( shape );
else {
   // etc
}
```

Data abstraction

```
if ( isCircle( shape ) ) {
   drawCircle( shape );
} else if ( isSquare( shape ) ) {
   drawSquare( shape );
else {
   // etc
}
```

1. Image adding a new kind of shapes; this would involve going through all the programs that are known to use this record type and finding all the places where you would need to add new cases;
2. This process is rather complex since the statements acting on the data are separated from the data;
3. A change to the record will have implications that are scattered throughout the programs.

Procedural Abstraction

+

Data Abstraction

=

Object-Oriented Programming

The central concept of object-oriented programming is the object.

An **object** consists of:

- data abstractions (variables)
- procedural abstractions (methods)

A software system consists of a collection of objects interacting together to solve a common task.

(Java is an object-oriented programming language)

A step back

We should take a step back and say that one of the first activities of software development consists of describing the objects, classes, and their interactions.

This process is often called **object-oriented analysis**.

It is important to note that this analysis can be done without knowing the details about the implementation.

Activities of Software Development

1. Requirements analysis;
   (defining the problem)

2. Design;
   (how to break the system into subsystems and what are the interactions between these subsystems)

3. Programming;

4. Quality assurance;

5. Project management.

$\Rightarrow$ OO helps with all those activities, and certainly facilitates the implementing the design.

Scalability

OO is at its best for the design of large software systems, which in general involves several people to work on the same project, and sometimes over the course of several months even years.

Coupling

We say that two classes are coupled if the implementation of at least one of them depends on the implementation of the other. That is one class has access to the variables of the other.

Maintainability of a system is inversely proportional to the coupling — the number of dependencies.

A well designed software system should be such the "interface" of the classes are clearly and precisely designed so that the implementation of these interfaces can be replaced with no effect on the rest of the system.

Examples

- **E-commerce:** customers, items, inventory, transactions, journal . . . ;
- **Chess game:** pieces, board, users;
- **Factory:** production lines, robots, items, parts, . . . ;

We see that for some objects , there is only one "instance" — this is the case for the board in the chess game application.

For others, there will be many objects of the same "class" all sharing the same properties and behaviours.

However, each of these objects is **unique** and has a **state** (the current values of its properties) that may or may not be the same as other objects.

Object

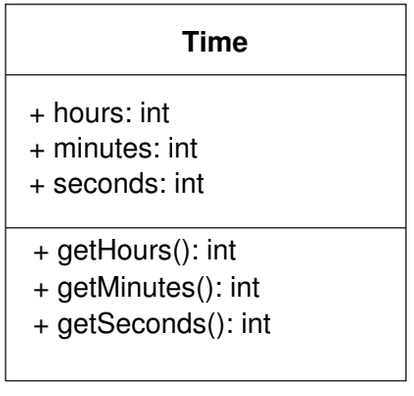
An object has:

- properties, which describe its current **state**;
- behaviours: what it can do, its responses to queries.

Unified Modeling Language (UML)

A standard graphical language for modelling object-oriented software; developed in the mid-1990s.

A class diagram is represented by a box divided in three parts: the name of the class, the attributes and the methods.



Further notation will be introduced along with concepts.

⇒ Simon Benett, Steve McRobb and Ray Farmer (1999) *Object-Oriented Systems Analysis and Design using UML*. McGraw-Hill.

How to?

Fine, an object contains data together with the methods that are transforming the data, but how to specify the content of an object.

The **class** defines the characteristics of a collection of objects.

There can be several "instances" (examples) of a class but each object is the instance of a single class.

The properties (the state) of an object are specified with help of instance variables (also called attributes).

The behaviour of an object is specified by its instance methods.

Class vs Object

At the beginning, it is often not clear what is the class and what is the object.

Objects are entities that only exist at run-time.

Objects are "examples" (instances) of a class.

In a sense, the class is like a blue-print that characterizes a collection of objects.

Class vs Object

In the case of a chess game application there can be a class which describes the properties and behaviours that are common to all the Pieces.

During the execution of the program, there will be many "instances" created: black king, white queen, etc.

Class vs Object

Are instance and object two different concepts?

No. Instance and object refer to the same concept, the word instance is used in sentences of the of the form "the instance of the class . . . ", when talking about the role of an object.

The word object is used to designate a particular instance without necessarily referring to its role, "insert the object into the data structure".

You cannot design an object, you are designing a class that specifies the characteristics of a collection of objects. The class then serves to create the instances.

Naming classes

Use **singular nouns** whose **first letter is capitalized**.

Conventions are very important to make the programs more readable.

The following declaration,

```
Counter counter;
```

clearly indicates a reference variable, **counter**, to be used to designate an instance of the class **Counter**.

A practical example is **System.out.println(“hello”)**.

Instance variables

The class lists all the variables that each object must have in order to model the given concept.

When a variable is declared to be an instance variable, it means that all the instances of the class will reserve space for the variable.

These can be primitive or reference variables.

Reference variables can used be to implement **association** relationships between objects: an event has a starting time and an ending time, where Event and Time might be two classes.

Instance method

An instance method, is a method that has access to the instance variables.

Counter

Perhaps the simplest object to describe is the **Counter**. Imagine the kind of device like those used at base-ball games to record the scores, strikes, etc.

The device that I imagine has a window that displays the current value and a single button, which every time it is pressed increases the value of the counter by 1.

A counter has to have a **state**, which is the current value of the counter.

There has to be a way to read the content of the counter.

There has to be a way to increase the value of the counter (and maybe a way to reset the value - another button).

Why not?

A single value has to be recorded!

Furthermore, the value can be represented with one of the primitive data types of Java such as `int` or `long`.

Here is a counter

```
int counter1;
```

Why not?

I can initialize it and increase its value whenever I want or need to

```
counter1 = 0;
counter1++; // stands for counter1 = counter1 + 1
```

I can have as many counters as want, here is a new counter

```
int counter2;
```

I can even have a whole array of them

```
int[] counters;
counters = new int[5];
```

$\Rightarrow$ What is the problem then?

Why not?

The problem is that it is not modelling the concept of a counter appropriately. Nothing prevents us from increasing the value by as much as we want,

```
counter1 = counter1 + 5;
// ...
counter1 = counter1 - 1;
// ...
```

Why not?

On a more technical note, in Java, such counter cannot be shared.

What do you mean shared?

Why not?

```
public static void process() {
  // Declares a counter
  int counter = 0;
  // Uses it for a while
  // ...
  // Gives it to a second method
  subProcess( counter );
  // Unless subProcess returns the new value *and* process assigns
  // this new value to counter, counter is unchanged.
}
```

But also, that **subProcess** might have been written by someone else, which means that we don't have the control on the ways that counter is used — we may even not have access to the source code, **subProcess** might decrease the value of the counter, etc.

Process may forget to update the counter!

OO to the rescue!

```
public class Counter {
    private int value = 0;

    public int getValue() {
      return value;
    }

    public void incr() {
        value++;
    }

    public void reset() {
        value = 0;
    }
}
```

```
public class Test {
    public static void main( String[] args ) {
        Counter counter = new Counter();
        System.out.println( "counter.getValue()->"
                                    +counter.getValue() );
        for ( int i=0; i<5; i++ ) {
            counter.incr();
            System.out.println( "counter.getValue()->"
                                    +counter.getValue() );
        }
    }
}
```

```
public class Test {
    public static void main( String[] args ) {
        Counter counter = new Counter();
        System.out.println( "counter.getValue()->"
                                    +counter.getValue() );
        for ( int i=0; i<5; i++ ) {
            counter.incr();
            System.out.println( "counter.getValue()->"
                                    +counter.getValue() );
        }
        counter.value = -9;
    }
}

Test.java:13: value has private access in Counter
        counter.value = -9;
                ^
```