Example: Fibonacci Numbers

Write a program which determines \( F_n \), the \((n + 1)\)-th Fibonacci number.

- The first 10 Fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, and 34.
- The sequence of Fibonacci numbers can be defined by the recurrence relation

\[
F_n = F_{n-1} + F_{n-2},
\]

where \( n \geq 2 \) and \( F_0 = 0, F_1 = 1 \).
• This recursive implementation is straightforward.
• Yet, this algorithm isn’t efficient since it requires more time and memory.
• Time complexity: $O(2^n)$ (Why?!)
static double fibIter(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;

    int x = 0, y = 1;
    for (int i = 2; i <= n; i++) {
        int z = x + y;
        x = y;
        y = z;
    }
    return y;
}

• So it can be done in $O(n)$ time.
• It implies that the recursive one is not optimal.
• Could you find a linear recursion for Fibonacci numbers?
• You may try more examples.\footnote{See http://introcs.cs.princeton.edu/java/23recursion/}
Divide and Conquer

- For program development, we use the divide-and-conquer strategy\(^2\) to decompose the original problem into subproblems, which are more manageable.
  - For example, selection sort.
- Pros: easier to write, reuse, debug, modify, maintain, and also better facilitating teamwork

\(^2\)Aka stepwise refinement.
Computational Thinking

• To think about computing, we need to be attuned to three fields: science, technology, and society.
• Computational thinking shares with
  • mathematical thinking: the way to solve problems
  • engineering thinking: the way to design and evaluating a large, complex system
  • scientific thinking: the way to understand computability, intelligence, the mind and human behavior.

---

\(^3\)You should read this:
http://rsta.royalsocietypublishing.org/content/366/1881/3717.full
The essence of computational thinking is abstraction.

- An **algorithm** is an abstraction of a step-by-step procedure for taking input and producing some desired output.
- A **programming language** is an abstraction of a set of strings each of which when interpreted effects some computation.
- And more.

The abstraction process, which is to decide what details we need to highlight and what details we can ignore, underlies computational thinking.

The abstraction process also introduces layers.

- Well-defined **interfaces** between layers enable us to build large, complex systems.
Example: Abstraction of Computer System

Software

Application Programs

Libraries

Operating System

Drivers

Memory Manager

Scheduler

Execution Hardware

System Interconnect (bus)

Memory Translation

Controllers

I/O devices and Networking

Controllers

Main Memory

Hardware
Example: Methods as Control Abstraction

Optional arguments for input

Optional return value

Method Header

Method Body

Black box
• Control abstraction is the abstraction of actions while data abstraction is that of data structures.

• One can view the notion of an object as a way to combine abstractions of data and code.
class Lecture7 {

    // Objects and Classes

}

// Key words:

class, new, this, static, null, extends, super, abstract, final, interface, implements, protected
Observations for Real Objects

• Look around.
• We can easily find many examples for real-world objects.
  • For example, a person and his/her bottle of water.
• Real-world objects all have states and behaviors.
  • What possible states can the object be in?
  • What possible behaviors can the object perform on the states?
• Identifying these states and behaviors for real-world objects is a great way to begin thinking in object-oriented programming.
• From now, OO is a shorthand for “object-oriented.”
Software Objects

- An object keeps its states in **fields** and exposes its behaviors through **methods**.
- Plus, internal states are hidden and the interactions to the object are only performed through an object’s methods.
- This is so-call **encapsulation**, which is one of OO features.
- Note that the other OO features are **inheritance** and **polymorphism**, which we will see later.
We often find many individual objects all of the same kind.
  - For example, each bicycle was built from the same blueprint so that each contains the same components.

In OO terms, we say that your bicycle is an instance of the class of objects known as Bicycle.

A class is the blueprint to create class instances which are runtime objects.

Classes are the building blocks of Java applications.
class Point {
    // data members: fields or attributes
    double x, y;
}

public class PointDemo {
    public static void main(String[] args) {
        // now create a new instance of Point
        Point p1 = new Point();
        p1.x = 1;
        p1.y = 2;
        System.out.printf("(%d, %d)\n", p1.x, p1.y);

        // create another instance of Point
        Point p2 = new Point();
        p2.x = 3;
        p2.y = 4;
        System.out.printf("(%d, %d)\n", p2.x, p2.y);
    }
}
Class Definition

• First, give a class name with the first letter capitalized, by convention.

• The class body, surrounded by balanced braces {}, contains data members (fields) and function members (methods) for objects.
Data Members

• The fields are the states of the object.
• The field may have an access modifier, say `public` and `private`.
  - `public`: accessible by all classes
  - `private`: accessible only within its own class
• You can decide if these fields are accessible!
• In practice, all fields should be declared `private`.
• However, this `private` modifier does not quarantine any security.\(^4\)
  - What private is good for **maintainability** and **modularity**.\(^5\)

---

\(^4\) Thanks to a lively discussion on January 23, 2017.
Function Members

- As said, the fields are hidden.
- So we may need **getters** and **setters** if necessary:
  - getters: return the state of the object
  - setter: set the state of the object
- For example, \texttt{getX()} and \texttt{getY()} are getters while \texttt{setX()} and \texttt{setY()} are setters in the class \texttt{Point}. 
Example: Point (Encapsulated)

```java
class Point {
    // data members: fields or attributes
    private double x;
    private double y;

    // function members: methods
    double getX() { return x; }
    double getY() { return y; }

    void setX(double new_x) { x = new_x; }
    void setY(double new_y) { y = new_y; }
}
```
Unified Modeling Language

• Unified Modeling Language (UML) is a tool for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems.

• Free software:
  • http://staruml.io/ (available for all platforms)

---

Example: Class Diagram for Point

<table>
<thead>
<tr>
<th>Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>-x: double</td>
</tr>
<tr>
<td>-y: double</td>
</tr>
<tr>
<td>+getX(): double</td>
</tr>
<tr>
<td>+getY(): double</td>
</tr>
<tr>
<td>+setX(double): void</td>
</tr>
<tr>
<td>+setY(double): void</td>
</tr>
</tbody>
</table>

- Modifiers can be placed before the fields and the methods:
  - + for public
  - - for private
Constructors

• A constructor is called by the new operator.
• A constructor acts like other methods.
• However, its names should be identical to the name of the class and it has no return type.
• A class may have several constructors if needed.
  • Constructors can be overloaded.
• Note that the constructors are used only during the objection creation.
  • Constructors cannot be invoked by any object.
• If you don’t define any explicit constructor, Java assumes a default constructor for your class.
• Moreover, adding any explicit constructor disables the default constructor.
Parameterized Constructors

- You can provide specific information to the parameterized constructor during the object creation.
- For example,

```java
class Point {

    // default constructor
    Point() {
        // do something in common
    }

    // parameterized constructor
    Point(double new_x, double new_y) {
        x = new_x;
        y = new_y;
    }

    // do something
}
```