Recursion

Recursion is the process of defining something in terms of itself.

- A method that calls itself is said to be recursive.
- Recursion is an alternative form of program control.
- It is repetition without any loop.
• Try Fractal.
The **factorial** of a non-negative integer n, denoted by \( n! \), is the product of all positive integers less than and equal to n.

- Note that \( 0! = 1 \).
- For example,

\[
4! = 4 \times 3 \times 2 \times 1 \\
= 4 \times 3! \\
= 24.
\]

- Can you find the pattern?
  - \( n! = n \times (n - 1)! \)
  - In general, \( f(n) = n \times f(n - 1) \).
Write a program which determines $n!$.

```java
public static int factorial(int n) {
    if (n < 2) {
        return 1; // base case
    } else {
        return n * factorial(n - 1);
    }
}
```

• Note that there must be a **base case** in recursion.
• Time complexity: $O(n)$
• Can you implement the same method by using a loop?
Equivalence: Loop Version

```
...  
int s = 1;
for (int i = 2; i <= n; i++) {
    s *= i;
}
...
```

- Time complexity: $O(n)$
- One intriguing question is, Can we always turn a recursive method into a loop version of that?
- Yes, theoretically.\(^2\)

\(^2\)The Church-Turing Thesis proves it if the memory serves.
Remarks

- Recursion bears substantial overhead.
- So the recursive algorithm may execute a bit more slowly than the iterative equivalent.
- Additionally, a deeply recursive method depletes the call stack, which is limited, and causes stack overflow soon.
Memory Layout

Memory

2^{32} - 1

Stack

Heap

BSS (uninitialized)

Data (initialized)

Text (Code)

0
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 $F_0 = 0, F_1 = 1$. 

Example: Fibonacci Numbers

Zheng-Liang Lu

Java Programming
public static int fib(int n) {
    if (n < 2)
        return n;
    else
        return fib(n - 1) + fib(n - 2);
}

- Straightforward and clear!!!
- Yet, this algorithm isn’t efficient since it requires more time and memory.
- Time complexity: $O(2^n)$ (Why?!)
public static double fibIter(int n) {
    if (n < 2)
        return n;
    int x = 0, y = 1;
    for (int i = 2; i <= n; i++) {
        int z = x + y;
        x = y;
        y = z;
    }
    return y; // why not z?
}

• So it can be done in $O(n)$ time.
• It implies that the previous one is not optimal.
• Could you find a linear recursion for Fibonacci numbers?
• In fact, this problem can be done in $O(\log n)$ time!
Divide & Conquer

• We often use the divide-and-conquer strategy\(^3\) to decompose the original problem into subproblems, which are more manageable.
  • For example, selection sort.
• This benefits the program development as follows: easier to write, reuse, debug, modify, maintain, and also better to facilitate teamwork.

\(^3\)Aka the stepwise refinement.
Computational Thinking (CT) \(^5\)

- CT is essential to the development of computer applications.
- CT concepts are the mental processes and tangible outcomes associated with solving problems in computing.
  - mental process: abstraction, algorithm design, decomposition, pattern recognition
  - tangible outcome: automation, data representation, pattern generalization
- Involving mathematical thinking, engineering thinking, scientific thinking. \(^4\)

\(^4\)Also read http://rsta.royalsocietypublishing.org/content/366/1881/3717.full.

\(^5\)See https://edu.google.com/resources/programs/exploring-computational-thinking.
COMPUTATIONAL THINKING

DECOMPOSITION
Breaking big problems into smaller, easier to manage problems

PATTERN RECOGNITION
Analyze & look for a repeating sequence

ABSTRACTION
Remove parts of a problem that are unnecessary and make one solution work for multiple problems

ALGORITHM DESIGN
Step-by-Step instructions on how to do something

photo credit

Zheng-Liang Lu 
Java Programming
Abstraction

• The abstraction process is to decide what details we need to highlight and what details we can ignore.

• **Abstraction is everywhere.**
  - An algorithm is an abstraction of a step-by-step procedure for taking input and producing output.
  - A programming language is an abstraction of a set of strings, each of which is interpreted to some computation.
  - And more.

• The abstraction process also introduces **layers.**

• Well-defined **interfaces** between layers enable us to build large and complex systems.
Example: Computer Systems

**Software**

- Application Programs
- Libraries
- Operating System
  - Drivers
  - Memory Manager
  - Scheduler

**Hardware**

- Execution Hardware
  - System Interconnect (bus)
  - Memory Translation

- Controllers
  - I/O devices and Networking

- Controllers
  - Main Memory
Example: APIs

Optional arguments for input

Optional return value

Method Header

Method Body

Black box
Abstraction (Concluded)

- 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 actions.
class Lecture7 {

    // Object-Oriented Programming

} // Key words:
class, new, this, static, null, extends, super, abstract, final, interface, implements, protected
An **object** keeps its own states in **fields** (attributes) and exposes its behaviors through associated **methods**.

For example, describe about your cellphone.

- Attributes: battery status, 4G signal strength, phonebook, album, music, clips, and so on.
- Functions? You can name it.

To create these objects, we collect all attributes associated with functions and put them in a new **class**.
• Classes are the building blocks in every Java program.
• A class is the blueprint to create class instances, aka runtime objects.
• We define a new class as follows:
  • give a class name with the first letter capitalized, by convention;
  • declare data and function members in the class body.
• For example,

```java
public class Point {
    // data members
    double x, y;
}
```
Example: Points

```java
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;

        // create another instance of Point
        Point p2 = new Point();
        p2.x = 3;
        p2.y = 4;

        System.out.printf("(%.2f, %.2f)\n", p1.x, p1.y);
        System.out.printf("(%.2f, %.2f)\n", p2.x, p2.y);
    }
}
```

- Note that a class also acts as a (derived) type.
Data Members

• Each field may have an access modifier, say public and private.
  • public: accessible by all classes
  • private: accessible only within its own class
• In OO paradigm, we hide internal states and expose methods which perform actions on these fields.
  • So all fields should be declared private.
  • This is so-called encapsulation.
• However, this private modifier does not quarantine any information security.\textsuperscript{6}
  • What private is good for maintainability and modularity.\textsuperscript{7}

\textsuperscript{6}Thanks to a lively discussion on January 23, 2017.
\textsuperscript{7}Read http://stackoverflow.com/questions/9201603/are-private-members-really-more-secure-in-java.
As said, the fields are hidden.

So we provide **getters** and **setters** if necessary:
- **getter**: return the state of the object
- **setter**: set a value to the state of the object

For example, `getX()` and `getY()` are getters; `setX()` and `setY()` are setters in the class **Point**.
public class Point {
    // data members: fields or attributes
    private double x;
    private double y;

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

    public void setX(double new_x) { x = new_x; }
    public 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 both fields and methods:
  - + for public
  - - for private
Constructors

- A constructor follows the `new` operator, acting 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.
  - Recall method overloading.
- Note that constructors belong to the class but not objects.
  - In other words, constructors cannot be invoked by any object.
- If you don’t define any explicit constructor, Java assumes a default constructor for you.
  - Moreover, adding any explicit constructor disables the default constructor.
Parameterized Constructors

- You can initialize an object when the object is ready.
- For example,

```java
public class Point {
    ...
    // default constructor
    public Point() {
        // do something in common
    }

    // parameterized constructor
    public Point(double new_x, double new_y) {
        x = new_x;
        y = new_y;
    }
    ...
}
```
Self Reference

• You can refer to any (instance) member of the current object within methods and constructors by using this.
• The most common reason for using the this keyword is because a field is shadowed by method parameters.
  • Recall the variable scope.
• You can also use this to call another constructor in the same class, say this().
Example: Point (Revisited)

```java
public class Point {
    ...
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    ...
}
```

- However, the `this` operator cannot be used in static methods.
Instance Members

• Since this lecture, all members are declared w/o static, so-called instance members.
• These members are available only after the object is created.
• Semantically, each object has its own states associated with the methods applying on.
  • For example, getX() could be invoked when a specific Point object is assigned.
An object reference

ptr into heap

ptr to class data
instance data
instance data
instance data
instance data

the heap

class data

the method area
Static Members

- Static members belong to the class\(^9\).
- Those are ready **once the class is loaded**.
  - For example, `main()`.
- They can be invoked directly by the class name **without** any instance.
  - For example, `Math.random()` and `Math.PI`.
- Particularly useful for utility methods that perform work which is independent of instances.
  - For example, factory methods in design patterns.\(^{10}\)

---

\(^9\)Aka class members.

\(^{10}\)“Design pattern is a general reusable solution to a commonly occurring problem within a given context in software design.” by Wikipedia.
• A static method can access other static members. (Trivial.)
• However, static methods cannot access instance members directly. (Why?)
• For example,

```java
public double getDistanceFrom(Point that) {
    return Math.sqrt(Math.pow(this.x - that.x, 2)
        + Math.pow(this.y - that.y, 2));
}

public static double measure(Point first, Point second) {
    // You cannot use this.x and this.y here!
    return Math.sqrt(Math.pow(first.x - second.x, 2)
        + Math.pow(first.y - second.y, 2));
}
```