Exercise: Common Greatest Divisor (GCD)

Let $a$ and $b$ be two positive integers. Write a program to calculate $\text{GCD}(a, b)$.

- Recall the Euclidean algorithm.$^1$
- For example,

\[
\text{GCD}(54, 32) = \text{GCD}(32, 22) \\
= \text{GCD}(22, 10) \\
= \text{GCD}(10, 2) \\
= 0.
\]

$^1$See https://en.wikipedia.org/wiki/Euclidean_algorithm.
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 \( F_0 = 0, F_1 = 1 \).
• Straightforward and clear!!!
• Yet, this algorithm isn’t efficient since it requires more time and memory.
• Time complexity: $O(2^n)$. (Why?!)
fib(4)

17: return fib(4)
0: call fib(4)

10: return fib(3)

1: call fib(3)

16: return fib(2)

11: call fib(2)

return fib(3) + fib(2)

7: return fib(2)

2: call fib(2)

8: call fib(1)

13: return fib(1)

12: call fib(1)

14: return fib(0)

return fib(1) + fib(0)

4: return fib(1)

3: call fib(1)

5: call fib(0)

6: return fib(0)

return 1

return 0

return 1
• So it can be done in $O(n)$ time.
• The previous one (by recursion) is not optimal in time.
• 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\(^2\) 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.

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

- 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.

---

3 Also read http://rsta.royalsocietypublishing.org/content/366/1881/3717.full.
4 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
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

}
Object

- 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**.
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.\(^5\)
  • What private is good for maintainability and modularity.\(^6\)

\(^5\)Thanks to a lively discussion on January 23, 2017.
\(^6\)Read http://stackoverflow.com/questions/9201603/are-private-members-really-more-secure-in-java.
Function Members

- 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**.
Example: Point (Encapsulated)

```java
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

- **Modifiers can be placed before both fields and methods:**
  - `+` for **public**
  - `−` for **private**

<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>
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 \texttt{static}, so-called \texttt{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, \texttt{getX()} could be invoked when a specific Point object is assigned.
an object reference

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\(^8\).  
• 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.\(^9\)

---

\(^8\)Aka class members.  
\(^9\)“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 to 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));
}
```