class Lecture6 {

    "Methods"

}
Methods

- Methods can be used to define reusable code, and organize and simplify code.
- The idea of function originates from math, that is,
  \[ y = f(x), \]
  where \( x \) is the input parameter\(^1\) and \( y \) is the function value.
- In computer science, each input parameter should be declared with a specific type, and a function should be assigned with a return type.

\(^1\)Recall the multivariate functions. The input can be a vector, say the position vector \((x, y, z)\).
\(^2\)Aka procedures and functions.
Example: max

Define a method

```
public static int max(int num1, int num2) {
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}
```

Invoke a method

```
int z = max(x, y);
```
• The *modifier* could be *static* and *public* (for now).
• The *returnType* could be primitive types and reference types.
  • If the method does not return any value, then the return type is *void*.
• The *listOfParameters* is the input of the method, separated by commas if there are multiple items.
  • Note that a method could have no input.\(^3\)
• The method name and the parameter list together are called the *method signature*.\(^4\)

---
\(^3\)For example, \texttt{Math.random()}.  
**\(^4\)Method overloading** depends this. We will see it soon.
More Observations

- There are alternatives to the method `max()`:

```java
public static int max(int x, int y) {
    if (x > y) {
        return x;
    } else {
        return y;
    }
}
```

```java
public static int max(int x, int y) {
    return x > y ? x : y;
}
```
“All roads lead to Rome.”
– Anonymous

“但如你根本並無招式，敵人如何來破你的招式？”
– 風清揚，笑傲江湖。第十回。傳劍
The return Statement

- The `return` statement is the end point of the method.
- A `callee` is a method invoked by a `caller`.
- The callee returns to the caller if the callee
  - completes all the statements (w/o a `return` statement, say `main()`);
  - reaches a `return` statement;
  - throws an `exception` (introduced later).
- As you can see, the `return` statement is not necessarily at the bottom of the method.\(^5\)
- Once one defines the return type (except `void`), the method `should` guarantee to return a value or an object of that type.

\(^5\) Thanks to a lively discussion on November 22, 2015.
Bad Examples

```java
...  
public static int fun1() {
    while (true);
    return 0; // unreachable code
}

public static int fun2(int x) {
    if (x > 0) {
        return x;
    }
    // what if x < 0?
}
...  
```
Method Invocation

Note that the input parameters are sort of variables declared within the method as placeholders.

When calling the method, one needs to provide arguments, which must match the parameters in order, number, and compatible type, as defined in the method signature.
• In Java, method invocation uses **pass-by-value**.

• When the callee is invoked, the **program control** is transferred from the caller to the callee.

• For each invocation of methods, OS creates a **frame** which stores necessary information, and the frame is pushed in the call stack.

• The callee transfers the program control back to the caller once the callee finishes its job.
(a) The main method is invoked.
(b) The max method is invoked.

(c) The max method is being executed.
(d) The max method is finished and the return value is sent to k.
(e) The main method is finished.
Variable Scope

- A variable scope refers to the region where a variable can be referenced.
- A pair of balanced curly braces defines the variable scope.
- In general, variables can be declared in class level, method level, or loop level.
- We cannot duplicate the variables whose names are identical in the same level.
```java
public class ScopeDemo {

    public static int x = 1; // class level, also called a field

    public static void main(String[] args) {
        System.out.println(x); // output 1
        int x = 2; // method level, also called local variable
        x++; System.out.println(x); // output 3
        addOne();
        System.out.println(x); // output ?
    }

    public static void addOne() {
        x = x + 1;
        System.out.println(x); // output ?
    }
}
```
A Math Toolbox: Math Class

- The Math class provides basic mathematical functions and 2 global constants Math.PI\(^6\) and Math.E\(^7\).
- All methods are public and static.
  - For example, max, min, round, ceil, floor, abs, pow, exp, sqrt, cbrt, log, log10, sin, cos, asin,acos, and random.
- Full document for Math class can be found here.
- You are expected to read the document!

---

\(^6\)The constant \(\pi\) is a mathematical constant, the ratio of a circle’s circumference to its diameter, commonly approximated as 3.141593.

\(^7\)The constant \(e\) is the base of the natural logarithm. It is approximately equal to 2.71828.
Method Overloading

- Methods with the same name can coexist and be identified by the method signatures.

```java
...
public static int max(int x, int y) { ... }
// different numbers of inputs
public static int max(int x, int y, int z) { ... }
// different types
public static double max(double x, double y) { ... }
...
```
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.

Recursion is a common pattern in nature.
• 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 > 0)
        return n * factorial(n - 1);
    else
        return 1; // base case
}
...```

- 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

```java
... 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.\(^9\)

\(^9\)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.
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?!)
... public static double fibIter(int n) {
    if (n < 2)
        return n == 1 ? 1 : 0;

    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 recursive one is not optimal.
• Could you find a linear recursion for Fibonacci numbers?
• You may try more examples.\(^{10}\)

\(^{10}\)See http://introcs.cs.princeton.edu/java/23recursion/.
Divide and Conquer

• For program development, we use the divide-and-conquer strategy\textsuperscript{11} 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

\textsuperscript{11}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.

---

12 You should read this:

http://rsta.royalsocietypublishing.org/content/366/1881/3717.full
Computational Thinking Is Everywhere!

• 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
Example: Methods as Control Abstraction

[Diagram showing method structure with optional arguments for input and optional return value, labeled 'Method Header' and 'Method Body'.]
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 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.
Classes

- 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.
public class Point {
    // data members: so-called 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 curly braces {}, contains data members (fields) and function members (methods).
Data Members

- As mentioned earlier, these fields are the states of the object.
- Each field may have an access modifier, say public and private.
  - **public**: accessible by all classes
  - **private**: accessible only within its own class
- We can decide if these fields are accessible!
- In practice, all fields should be declared private to fulfill the concept of encapsulation.
- However, this private modifier does not quarantine any security.\(^\text{13}\)
  - What private is good for maintainability and modularity.\(^\text{14}\)

---

\(^{13}\) Thanks to a lively discussion on January 23, 2017.

Function Members

• As said, the fields are hidden.
• So we provide getters and setters if necessary:
  • getters: return some 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; }
}
```