Java Programming

U Hou Lok

Department of Computer Science and Information Engineering,
National Taiwan University

Java 273
22 Aug.–2 Sep., 2016

```java
class Lecture6 {

"Methods"

}

// keywords:
return
```

Methods[2]

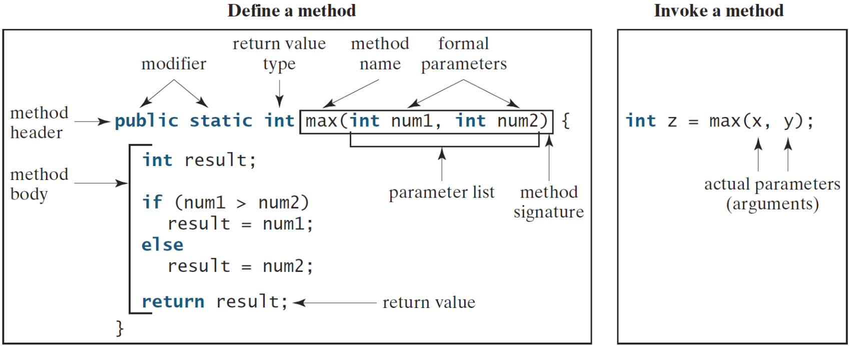
- Methods can be used to define reusable code, and organize and simplify code.
- The idea of function originates from math, that is,

$$y = f(x),$$

  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
modifier
return value type
method name
formal parameters
method header
method body
public static int max(int num1, int num2) {
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
parameter list
method signature
return value
Invoke a method
int z = max(x, y);
actual parameters (arguments)

```
...
modifier returnType methodName(listOfParameters) {
// method body
}
...
```

- So far, the *modifier* could be static and public.
- The *returnType* could be primitive types, reference types[3], and void.[4]
- The *listOfParameters* is the input of the method, separated by commas if there are multiple items.
- Note that a method can have no input.[5]
- The method name and the parameter list together constitute the method signature.[6]

[3] A method can return an object.

[4] Recall that a void method does not return a value.

[5] For example, **Math.random**().

[6] It is the key to the method overloading. 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

""

–

The return Statements

- 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).
- Note that the return statement is not necessarily at the bottom of the method.[7]
- Once one defines the return type (except void), the method should return a value or an object of that type.

[7] Thanks to a lively discussion on November 22, 2015.

- However, your method should make sure that the return statement is available for all conditions.
- For example,

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

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

Method Invocation

pass the value i

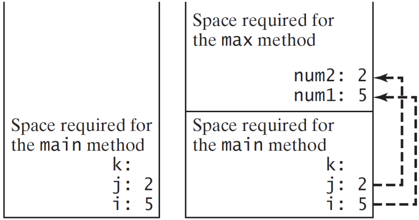
pass the value j

```java
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum of " + i +
    " and " + j + " is " + k);
}
```

```java
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

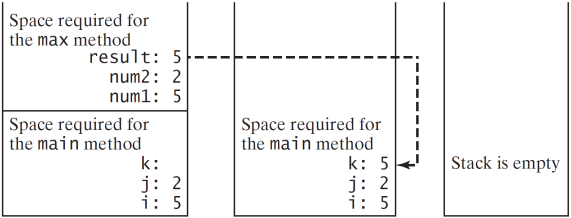
- Note that the input parameters are sort of variables declared within the method as placeholders.
- When calling the method, one needs to provide arguments, which must match the parameters in order, number, and compatible type, as defined in the method signature.

- In Java, method invocation uses pass-by-value.
- When the callee is invoked, the program control is transferred to the caller.
- For each invocation of methods, OS creates an 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

- The variable scope is the range where the variable can be referenced.
- Variables can be declared in class level, method level, and loop level.
- In general, a balanced curly brackets defines a particular scope.
- One can declare variables with the same name in different levels of scopes.
- Yet, one cannot declare the variables with the same name in the same scope.

Example

```java
public class ScopeDemo {

static int i = 1; // class level

public static void main(String[] args) {
System.out.printf("%d\n", i); //output 1
int i = 2; // method level; local
i++;
System.out.printf("%d\n", i); // output 3
p();
System.out.printf("%d\n", i); // output ?
}

static void p() {
i = i + 1;
System.out.printf("%d\n", i); // output ?
}
}
```

Exercise

Write a program which allows the user to enter the math grades one by one (-1 to exit), and outputs a histogram.

```
Enter (-1 to exit): 40
Enter (-1 to exit): 90
Enter (-1 to exit): 86
Enter (-1 to exit): 56
Enter (-1 to exit): 78
Enter (-1 to exit): 64
Enter (-1 to exit): -1
End of Input...
Total: 6
 90 ~ 100: *
 80 ~  89: *
 70 ~  79: *
 60 ~  69: *
  0 ~  59: **
```

```java
...
int[] hist = new int[5];
// hist[0]: the counter for 90 ~ 100
// hist[1]: the counter for 80 ~ 89
// ...
// hist[4]: the counter for 0 ~ 59

Scanner in = new Scanner(System.in);
int inputValue;
do {
do {
System.out.println("Enter x? (-1 to exit)");
inputValue = in.nextInt();
} while (inputValue != -1 && (inputValue < 0 || inputValue > 100));
if (inputValue >= 90) ++hist[0];
else if (inputValue >= 80) ++hist[1];
else if (inputValue >= 70) ++hist[2];
else if (inputValue >= 60) ++hist[3];
else if (inputValue >= 0) ++hist[4];
else System.out.println("End of input.");
} while (inputValue != -1);
in.close();

int total = 0;
for (int i = 0; i < hist.length; i++) {
```

```
    total += hist[i];
    }
    System.out.printf("Total: %d\n", total);

    int[] tick = {0, 60, 70, 80, 90, 101};
    for (int i = 0; i < 5; i++) {
    System.out.printf("%3d ~ %3d: ", tick[i], tick[i + 1] - 1);
    for (int j = 0; j < hist[i]; j++) {
    System.out.printf("*");
    }
    System.out.printf("\n");
    }
    ...
```

- Shall we rearrange the codes in order to reduce the complexity of the main method?

A Math Toolbox: **Math** Class

- The **Math** class provides basic mathematical functions and 2 global constants **Math.PI**[8] and **Math.E**[9].
- 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!

[8] π is a mathematical constant, the ratio of a circle's circumference to its diameter, commonly approximated as 3.141593.

[9] e is the base of the natural logarithm. It is approximately equal to 2.71828.

Method Overloading

- Overloading methods enables us to define the methods with the same name as long as their signatures are different.
- This can make programs clearer and more readable.
  - Name conflict is fine.
  - Make sure the signatures differ; the overloaded methods work on different input parameters.
- Note that you cannot overload methods merely based on different modifiers or return types.

Example

(Revisit) Printing stars

Rewrite a program which outputs a certain number of lines with some symbol determined by the user. Also, the program repeats itself until the input number is negative. Use * if the input for the symbol is left blank.

```java
...
static void print(int n) {
for (int i = 1; i <= n; i++) {
for (int j = 1; j <= i; j++) {
System.out.printf("*");
}
System.out.println();
}
}

static void print(int n, String m) {
for (int i = 1; i <= n; i++) {
for (int j = 1; j <= i; j++) {
System.out.printf("%s", m);
}
System.out.println();
}
}
...
```

```java
...
while (true) {
System.out.println("Enter n = ?");
n = input.nextInt();
if (n > 0) {
System.out.println("Enter a symbol?");
m = input.next();
if (m.equals("")) print(n);
else print(n, m);
} else {
System.out.println("Bye.");
break;
}
}
...
```

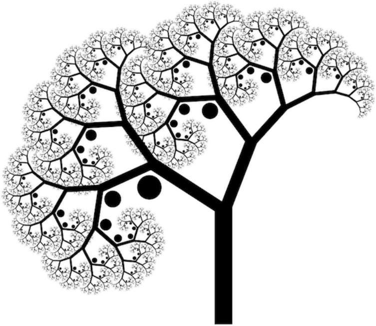
- The method **next**() returns a string from the keyboard.
- The method **equals**() is a **String** method used to compare if the two strings are identical.

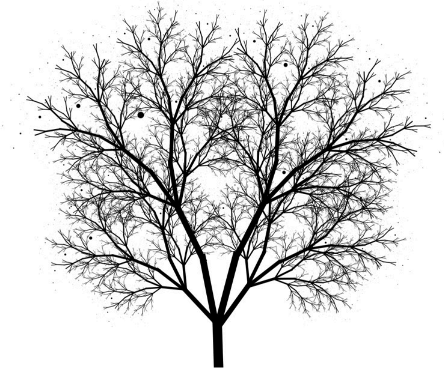
Recursion[10]

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.

[10] Recursion is a commom pattern in nature.





- Try Fractal.

Example

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,

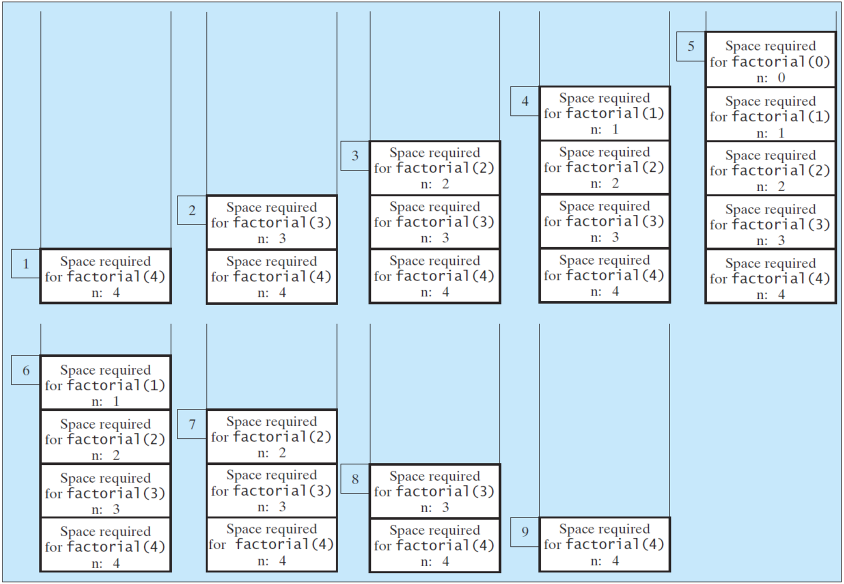
$$\begin{aligned} 4! &= 4 \times 3 \times 2 \times 1 \\ &= 4 \times 3! \\ &= 24. \end{aligned}$$

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

Write a program which determines $n!$.

```
...
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?


1
Space required for factorial(4) n: 4
2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
3
Space required for factorial(2) n: 2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
4
Space required for factorial(1) n: 1
Space required for factorial(2) n: 2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
5
Space required for factorial(0) n: 0
Space required for factorial(1) n: 1
Space required for factorial(2) n: 2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
6
Space required for factorial(1) n: 1
Space required for factorial(2) n: 2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
7
Space required for factorial(2) n: 2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
8
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
9
Space required for factorial(4) n: 4

Equivalence: Loop Version

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

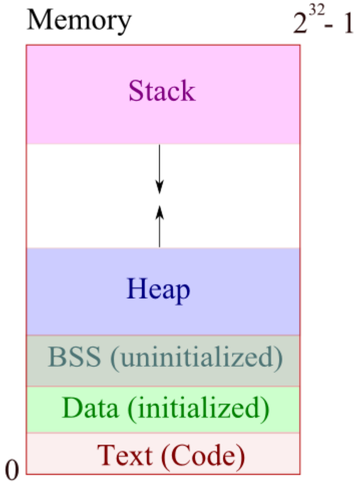
- Time complexity: $O(n)$
- One intriguing question is, Can we always turn a recursive method into a loop version of that?
- Yes, theoretically.[11]

[11] 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



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

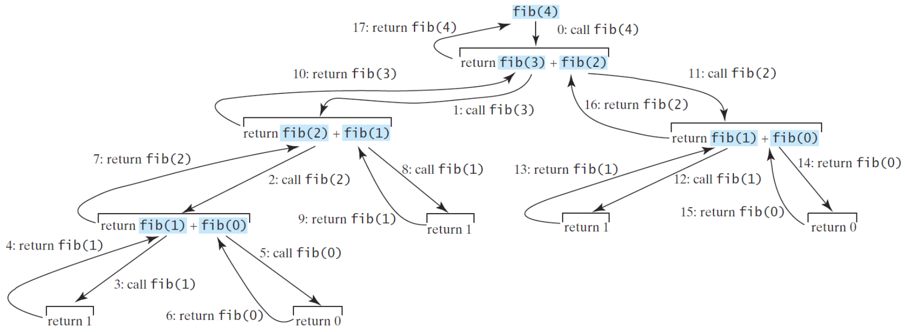
$$F_n = F_{n-1} + F_{n-2},$$

where $F_0 = 0, F_1 = 1$, and $n \geq 2$.

```java
    ...
    static int fib(int n) {
    if (n > 0) return fib(n - 1) + fib(n - 2);
    else if (n == 1) return 1;
    else return 0;
    }
    ...
```

- This recursive implementation is straightforward.
- Yet, this algorithm isn't efficient since it requires more time and memory.
- Time complexity: $O(2^n)$ (Why?!)
- You may check Master Theorem.[12]

[12] See https://math.dartmouth.edu/archive/m19w03/public_html/Section5-2.pdf.


fib(4)
0: call fib(4)
17: return fib(4)
return fib(3) + fib(2)
10: return fib(3)
1: call fib(3)
11: call fib(2)
16: return fib(2)
return fib(2) + fib(1)
return fib(1) + fib(0)
7: return fib(2)
2: call fib(2)
8: call fib(1)
9: return fib(1)
return 1
13: return fib(1)
12: call fib(1)
14: return fib(0)
15: return fib(0)
return 1
return 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

```java
...
static double fibIter(int n) {
int x = 0, y = 1;
if (n >= 2) {
for (int i = 2; i < n; ++i) {
int tmp = x + y;
x = y;
y = tmp;
}
return y;
} else return (n == 1) ? y : x;
}
...
```

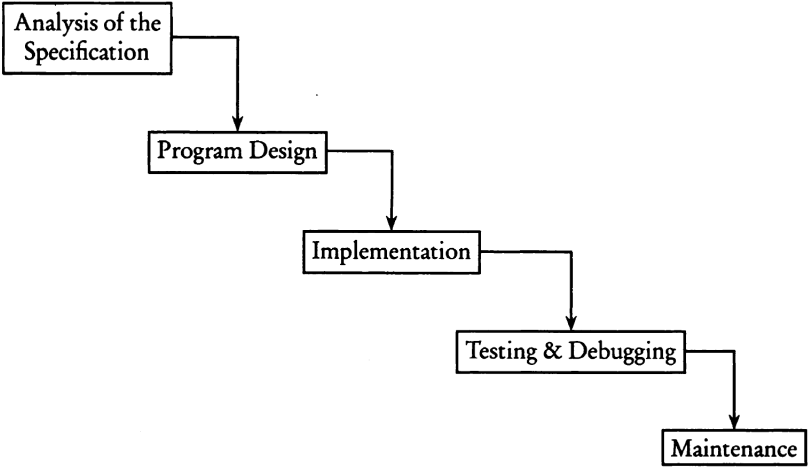
- So it can be done in $O(n)$ time.
- It implies that the recursive one is not optimal.
- Could you find a linear recursion for Fibonacci numbers?

Divide and Conquer

- For program development, we use the divide-and-conquer strategy[13] 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

[13] Aka stepwise refinement.

Program Development: Waterfall Model



Computational Thinking[14]

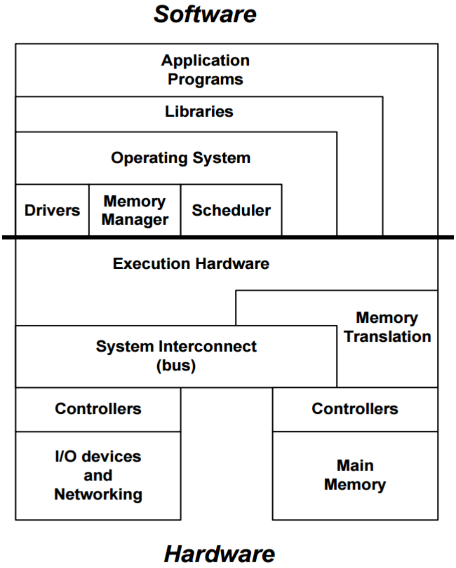
- To think about computing, we need to be attuned to three fields: science, technology, and society.
- Computational thinking shares with
  - mathematical thinking: the way to solve problems
  - engineering thinking: the way to design and evaluating a large, complex system
  - scientific thinking: the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

[14] You should read this: http://rsta.royalsocietypublishing.org/content/366/1881/3717.full

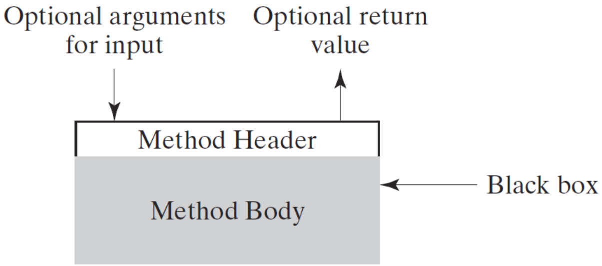
Computational Thinking Is Everywhere!

- 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



Example: Methods as Control Abstraction



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.

```java
class Lecture7 {

// Objects and Classes

}

// Reserved words:
class, new, this, extends, super, interface, implements, abstract,
    static, protected, null, default, final
// References: Ch.5 - 6 in Learning Java
```

Observations for Real Objects

- Look around.
- We can easily find many examples for real-world objects.
  - For example, students and an instructor.
- Real-world objects all have states and behaviors.
  - What possible states can the object be in?
  - What possible behaviors can the object perform?
- 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 stores 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 OO features are encapsulation, inheritance, and polymorphism.

Classes

- We often find many individual objects all of the same kind.
- For example, each bicycle was built from the same set of blueprints and therefore 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.

Example: Points in 2D Coordinate

```java
class Point {
    double x, y; // fields: data member
}
```

```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;
        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.
- We may need an entry point to run the program if the class is used to run the whole program.

Data Members

- The fields are the states of the object.
- May have an access modifier, say public and private.
  - public: accessible from all classes
  - private: accessible only within its own class
- Access modifiers realize encapsulation!
- In other words, you can decide if these fields are accessible!
- In OO practice, all the fields should be declared private.

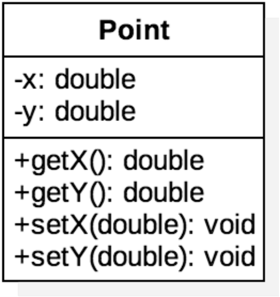
Function Members

- As said, the fields are hidden by encapsulation.
- So we may need accessors and mutators if necessary.
  - Accessors: return the state of the object
  - Mutators: set the state of the object
- For example, **getX**() and **getY**() are accessors, and **setPoint**(double, double) is a mutator in the class **Point**.

Example: Point (Encapsulated)

```
class Point {
    private double x;
    private double y;

    double getX() { return x; }
    double getY() { return y; }

    void setX(double a) { x = a; }
    void setY(double a) { y = a; }
    void setPoint(double a, double b) {
        x = a;
        y = b;
    }
}
```

Example: Class Diagram for Point



- Modifiers are placed before the fields and the methods:
  - $+$ for public
  - $-$ for private

Constructors

- A constructor is called by the new operator.
- Constructors are like special methods except that they use the name of the class and have no return type.
- Note that the constructors are not the part of the instances.
- Like other methods, the constructors can also be overloaded.
- If you don't define an 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 as the object is instantiated.
- For example,

```java
class Point {
    ...

    Point() {} // restore a default constructor;

    // parameterized constructor
    Point(double a, double b) {
        x = a;
        y = b;
    }
    ...
}
```

Example: Point (Revisited)

```
public class PointDemo {
    public static void main(String[] args) {
        Point p1 = new Point(1, 2);
        System.out.printf("(%f, %f)", p1.getX(), p1.getY());
        // output (1, 2)
    }
}
```

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.
- You can also use this to call another constructor in the same class by invoking this().

Example: Point (Revisited)

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

- Note that the this operator is referenced to instance members only, but not members declared static.

Instance Members and Static Members

- You may notice that all the members are declared w/o static.
- These members belong to some specific object.
- They are called instance members.
- This implies that these instance members are available only when the object is created.
- Those declared w/ static are static members, aka class members.

Static Members

- Static members mean that there is only one copy of the static members, no matter how many objects of the class are created.
- The static members belong to the class, and are shared between the instance objects.
- They are ready once the class is loaded.
- They can be invoked directly by the class name without creating an instance.
- For example, **Math.random**().

- A static method can access other static members.
- However, static methods cannot access to instance members directly. (Why?)
- For example,

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

    static double distanceBetween(Point p1, Point p2) {
        return Math.sqrt(Math.pow(p1.x - p2.x, 2) + Math.pow(p1.y - p2.
            y, 2));
    }
...
```

Example: Count of Points

```java
class Point {
    ...
    private static int cnt = 0;

    Point() {
        cnt++;
    }

    Point(int x, int y) {
        this(); // calling the constructor with no input argument;
            should be placed in the first line in the constructor
        this.x = x;
        this.y = y;
    }

    ...
}
```

Garbage Collection (GC)[15]

- Java handles deallocation automatically.
- Automatic garbage collection is the process of looking at the heap memory, identifying whether or not the objects are in use, and deleting the unreferenced objects.
- An object is said to be unreferenced if the object is no longer referenced by any part of your program.
  - Simply assign null to the reference-type variable to make the object referenced by this variable unreferenced.
- So the memory used by these objects can be reclaimed.

[15] http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

finalize() Method

- The method **finalize**() conducts a specific task that will be executed as soon as the object is about to be reclaimed by GC.
- The **finalize**() method can be only invoked prior to GC.
- In practice, it must not rely on the **finalize**() method for normal operations. (Why?)

Example

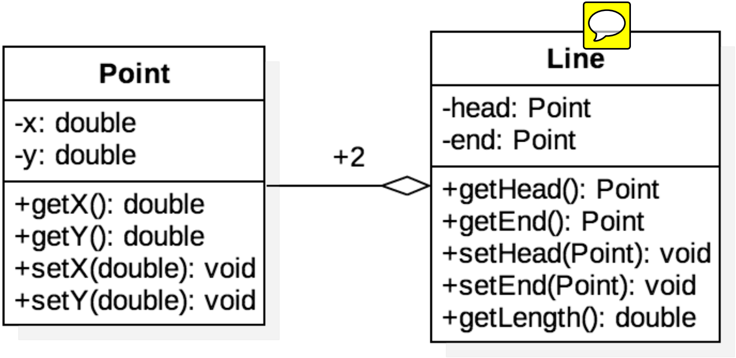
```java
public class finalizeDemo {
    static int numOfPointKilled = 0;

    public void finalize() {
        numOfPointKilled++;
    }

    public static void main(String[] args) {
        double n = 1e7;
        for (int i = 1; i <= n; i++)
            new finalizeDemo();
        System.out.println(numOfPointKilled);
    }
}
```

- You may try different number for instance creation.
- The number of the objects reclaimed by GC is not deterministic.

HAS-A Relationship

- Association is a relationship where all objects have their own lifecycle and there is no owner.
  - For example, teacher $\leftrightarrow$ student
- Aggregation is a specialized form of association where all objects have their own lifecycle, but there is ownership and child objects do not belong to another parent object.
  - For example, knight $\leftrightarrow$ sword
- Composition is a specialized form of aggregation and we can call this as a "death" relationship.
  - For example, house $\leftrightarrow$ room

Example: Lines on 2D Cartesian Coordinate



- +2: **Line** object contains two **Point** objects.

More Examples

- More geometric objects, say **Circle**, **Triangle**, and **Polygon**.
- **Complex** number $(a + bi)$ equipped with $+ - \times \div$ and so on.
- **Book** with **Author**s.
- **Lecturer** and **Student**s in the classroom.
- **Zoo** with many creatures, say **Dog**, **Cat**, and **Bird**.
- **Channel**s played on **TV**.

More Relationships Between Classes

- Inheritance: passing down states and behaviors from the parents to their children
- Interfaces: grouping the methods, which belongs to some classes, as an interface to the outside world
- Packages: grouping related types, providing access protection and name space management