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.\(^1\)
- Once one defines the return type (except void), the method should guarantee to return a value or an object of that type.

\(^1\)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?
}
...```

Zheng-Liang Lu Java Programming 194 / 243
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.
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 ?
    }
}

Zheng-Liang Lu Java Programming 199 / 243
The Math class provides basic mathematical functions and 2 global constants `Math.PI` and `Math.E`.

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!

---

2 The constant $\pi$ is a mathematical constant, the ratio of a circle’s circumference to its diameter, commonly approximated as 3.141593.

3 The constant $e$ is the base of the natural logarithm. It is approximately equal to 2.71828.
• 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

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.
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,

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

- 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?
<table>
<thead>
<tr>
<th></th>
<th>Space required for factorial(4) n: 4</th>
<th></th>
<th>Space required for factorial(2) n: 2</th>
<th></th>
<th>Space required for factorial(1) n: 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Space required for factorial(3) n: 3</td>
<td>2</td>
<td>Space required for factorial(3) n: 3</td>
<td>3</td>
<td>Space required for factorial(2) n: 2</td>
</tr>
<tr>
<td></td>
<td>Space required for factorial(4) n: 4</td>
<td></td>
<td>Space required for factorial(3) n: 3</td>
<td></td>
<td>Space required for factorial(1) n: 1</td>
</tr>
<tr>
<td>4</td>
<td>Space required for factorial(1) n: 1</td>
<td></td>
<td>Space required for factorial(3) n: 3</td>
<td></td>
<td>Space required for factorial(2) n: 2</td>
</tr>
<tr>
<td></td>
<td>Space required for factorial(3) n: 3</td>
<td></td>
<td>Space required for factorial(3) n: 3</td>
<td></td>
<td>Space required for factorial(2) n: 2</td>
</tr>
<tr>
<td>6</td>
<td>Space required for factorial(3) n: 3</td>
<td></td>
<td>Space required for factorial(3) n: 3</td>
<td></td>
<td>Space required for factorial(3) n: 3</td>
</tr>
<tr>
<td>7</td>
<td>Space required for factorial(4) n: 4</td>
<td></td>
<td>Space required for factorial(4) n: 4</td>
<td></td>
<td>Space required for factorial(4) n: 4</td>
</tr>
<tr>
<td>8</td>
<td>Space required for factorial(4) n: 4</td>
<td></td>
<td>Space required for factorial(4) n: 4</td>
<td></td>
<td>Space required for factorial(4) n: 4</td>
</tr>
<tr>
<td>9</td>
<td>Space required for factorial(4) n: 4</td>
<td></td>
<td>Space required for factorial(4) n: 4</td>
<td></td>
<td>Space required for factorial(4) n: 4</td>
</tr>
</tbody>
</table>
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.\(^5\)

\(^5\)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

Stack

Heap

BSS (uninitialized)

Data (initialized)

Text (Code)

$2^{32} - 1$
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;
    
    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.\textsuperscript{6}

\textsuperscript{6}See http://introcs.cs.princeton.edu/java/23recursion/.
Divide and Conquer

• For program development, we use the divide-and-conquer strategy 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.

7 Aka stepwise refinement.
Computational Thinking

- Computational thinking is taking an approach to solving problems, designing systems and understanding human behavior that draws on concepts fundamental to computing.  
  - solve problems: mathematical thinking  
  - design systems: engineering thinking  
  - understand human behavior: scientific thinking

---

8 Read http://rsta.royalsocietypublishing.org/content/366/1881/3717.full.
9 Design and evaluate a large and complex system that operates within the constraints of the real world.
**Abstraction**
Problem Formulation

"how does a mudslide work?"

**Analysis**
Solution Execution and Evaluation

visualize the consequence of thinking

**Automation**
Solution Expression

build simple model of gravity

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
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
Observation in Real World

- Look around.
- We can easily find many examples for real-world objects.
  - For example, a person with a bottle of water.
- Real-world objects all have states and behaviors.
  - What states can the object need?
  - What 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.”
Objects

- An object keeps its states in fields (or attributes) and exposes its behaviors through methods.
- Plus, we hide internal states and expose methods which perform actions on the aforesaid states.
- This is so-call encapsulation, which is one of OO features.\(^{10}\)
- Before we create the objects, we need to define a new class as their prototype (or concept).

\(^{10}\)The rest of features in OO are inheritance and polymorphism, which we will see later.
Classes

- We often find many objects all of the same kind.
  - For example, student A and student B are two instances of “student”.
  - Every student needs a name and a student ID.
  - Every student should do homework and pass the final exams.

- A class is the blueprint to create class instances which are runtime objects.
  - In the other word, an object is an instance of some associated class.

- In Java, classes are the building blocks in every program.
- Once the class is defined, we can use this class to create objects.
Example: Points in 2D Coordinate

```java
class Point {
    // data members: so-called fields or attributes
    double x, y;
}

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.\(^{11}\)
  - What private is good for maintainability and modularity.\(^{12}\)

---

\(^{11}\)Thanks to a lively discussion on January 23, 2017.
\(^{12}\)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:
  • 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; }
}
```
Exercise: Phonebook

```java
public class Contact {
    private String name;
    private String phoneNumber;

    public String getName() { return name; }
    public String getPhoneNumber() { return phoneNumber; }

    public void setName(String new_name) { name = new_name; }
    public void setPhoneNumber(String new_phnNum) {
        phoneNumber = new_phnNum;
    }
}
```
public class PhonebookDemo {

    public static void main(String[] args) {
        Contact c1 = new Contact();
        c1.setName("Arthur");
        c1.setPhoneNumber("09xxnnnnnnn");

        Contact c2 = new Contact();
        c1.setName("Emma");
        c1.setPhoneNumber("09xxnnnnnnn");

        Contact[] phonebook = {c1, c2};

        for (Contact c: phonebook) {
            System.out.printf("%s: %s
", c.getName(),
                           c.getPhoneNumber());
        }
    }
}
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
Constructors

- A constructor follows 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.
  - Recall method overloading.
- 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 you.
- Moreover, adding any explicit constructor disables the default constructor.
Parameterized Constructors

- You can provide specific information to objects by using parameterized constructors.
- 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.
- You can also use this to call another constructor in the same class by invoking 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

• You may notice that, until now, all members are declared w/o static.
• These members are called instance members.
• These instance members are available only after the object is created.
• This implies that each object has its own states and does some actions.
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

- The static members belong to the class\(^\text{14}\), and are shared between the instance objects.
- These members are ready once the class is loaded.
  - For example, the main method.
- They can be invoked directly by the class name without using any instance.
  - For example, `Math.random()` and `Math.PI`.
- They are particularly useful for utility methods that perform work that is independent of instances.
  - For example, factory methods in design patterns.\(^\text{15}\)

---

\(^\text{14}\) As known as class members.

\(^\text{15}\) “Design pattern is a general reusable solution to a commonly occurring problem within a given context in software design.” by Wikipedia.
Memory used by JVM

<table>
<thead>
<tr>
<th>Area</th>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>Objects</td>
</tr>
<tr>
<td>Stack</td>
<td>Methods</td>
</tr>
<tr>
<td></td>
<td>Local Variables</td>
</tr>
<tr>
<td></td>
<td>Reference Variables</td>
</tr>
<tr>
<td>Code</td>
<td>Byte Code</td>
</tr>
<tr>
<td>Static</td>
<td>Static Methods and Data</td>
</tr>
</tbody>
</table>

- **-Xss**: The Java Thread Stack Size, the default is OS and JVM dependent, and it can range 256k-to-1MB. The default should be tuned down to a range that doesn’t cause StackOverflow. I often use 128k-192k. Since the default -Xss is high, tuning it down can help save on memory used and given back to the Guest OS.

- **-Xmx**: The JVM max Heap Size

**Perm Size** is an area additional to the -Xmx (Max Heap) value and is not GC’ed because it contains class-level information.

“other mem” is additional mem required for NIO buffers, JIT code cache, classloaders, Socket Buffers (receive/send), JNI, GC internal info.

\[
\text{JVM Memory} = \text{JVM Max Heap (-Xmx value)} + \text{JVM Perm Size (-XX:MaxPermSize)} + \text{NumberOfConcurrentThreads} \times (-Xss value) + \text{“other mem”}
\]
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));
}
```

...
Example: Count of Points

```java
public class Point {
    
    private static int numOfPoints = 0;

    public Point() {
        numOfPoints++;
    }

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

    ... 
}
```