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 ?
    }
}

A Math Toolbox: Math Class

- The Math class provides basic mathematical functions and 2 global constants Math.PI\(^1\) and Math.E\(^2\).
- 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!

---

\(^1\)The constant π is a mathematical constant, the ratio of a circle’s circumference to its diameter, commonly approximated as 3.141593.

\(^2\)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.
• 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?
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.\(^4\)

\(^4\)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?!)
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.\(^5\)

Divide and Conquer

- For program development, we use the divide-and-conquer strategy\(^6\) 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.

---

\(^6\)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.\(^7\)
  - solve problems: mathematical thinking
  - design systems: engineering thinking\(^8\)
  - understand human behavior: scientific thinking

---

\(^7\)Read http://rsta.royalsocietypublishing.org/content/366/1881/3717.full.

\(^8\)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

Computational Thinking 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

**Hardware**

- Execution Hardware
- System Interconnect (bus)
- Memory Translation
- Controllers
  - I/O devices and Networking
- Controllers
  - Main Memory
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.\(^9\)
- Before we create the objects, we need to define a new class as their prototype (or concept).

\(^9\)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
public class Point {
    // data members: so-called fields or attributes
    double x, y;
}
```

```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 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.\(^{10}\)
  • What private is good for maintainability and modularity.\(^{11}\)

\(^{10}\)Thanks to a lively discussion on January 23, 2017.
\(^{11}\)Read [http://stackoverflow.com/questions/9201603/are-private-members-really-more-secure-in-java](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**.
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; }
}
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\n", 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/](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>get x(): double</td>
</tr>
<tr>
<td>get y(): double</td>
</tr>
<tr>
<td>set x(double): void</td>
</tr>
<tr>
<td>set y(double): void</td>
</tr>
</tbody>
</table>
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 without 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\textsuperscript{13}, and are shared between the instance objects.
- These members are ready \textit{once the class is loaded}.
  - For example, the main method.
- They can be invoked directly by the class name \textit{without} using any instance.
  - For example, \texttt{Math.random()} and \texttt{Math.PI}.
- They are particularly useful for utility methods that perform work that is independent of instances.
  - For example, factory methods in design patterns\textsuperscript{14}.

\textsuperscript{13}As known as class members.

\textsuperscript{14}“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

**Area** | **Contains**
---|---
Heap | Objects
Stack | Methods
Stack | Local Variables
Stack | Reference Variables
Code | Byte Code
Static | Static Methods and Data

**-Xms** : The JVM Max Heap Size

**-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.

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

JVM Memory = JVM Max Heap (-Xmx value) + JVM Perm Size (-XX:MaxPermSize) + NumberOfConcurrentThreads * (-Xss value) + "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));
}
...```

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;
    }
    ...
}
• In some situations, you may create the only instance of the class.

```java
public class Singleton {

    // Do now allow to invoke the constructor by other classes.
    private Singleton() {}

    // Will be ready as soon as the class is loaded.
    private static Singleton INSTANCE = new Singleton();

    // Only way to obtain this singleton by the outside world.
    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```
Garbage Collection (GC)\textsuperscript{16}

- Java handles deallocation\textsuperscript{15} automatically.
  - Timing: preset period or when memory stress occurs.
- GC is the process of looking at the heap, identifying if the objects are in use, and deleting those unreferenced objects.
- An object is unreferenced if the object is no longer referenced by any part of your program. (How?)
  - Simply assign \texttt{null} to the reference to make the object unreferenced.
- Note that you may invoke \texttt{System.gc()} to execute the deallocation procedure.
  - However, frequent invocation of GC is time-consuming.

\textsuperscript{15}Release the memory occupied by the unused objects.
\textsuperscript{16}http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html
The method `finalize()` conducts a specific task that will be executed right before the object is reclaimed by GC.

- For example, closing files and terminating network connections.

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 Garbage {
    private static int numOfObjKilled = 0;

    public void finalize() {
        numOfObjKilled++;
    }

    public static void main(String[] args) {
        double n = 1e7;
        for (int i = 1; i <= n; i++)
            new Garbage(); // lots of unreferenced objects
        System.out.println(numOfObjKilled);
    }
}
```

- You may try different number for instance creation.
- The number of the objects reclaimed by GC is uncertain.
HAS-A Relationship

- **Association** is a weak relationship where all objects have their own lifetime and there is no ownership.
  - For example, teacher ↔ student; doctor ↔ patient.
- If A uses B, then it is an **aggregation**, stating that B exists independently from A.
  - For example, knight ↔ sword; company ↔ employee.
- If A owns B, then it is a **composition**, meaning that B has no meaning or purpose in the system without A.
  - For example, house ↔ room.
Example: Lines

- +2: two **Point** objects used in one **Line** object.
public class Line {
    private Point head, tail;

    public Line(Point p1, Point p2) {
        head = p1;
        tail = p2;
    }

    public double getLength() {
        return head.getDistanceFrom(tail);
    }

    public static double measureLength(Line line) {
        return line.getLength();
    }
}

Zheng-Liang Lu
Java Programming
More Examples

- Circle, Triangle, and Polygon.
- Book with Authors.
- Lecturer and Students in the classroom.
- Zoo with many creatures, say Dog, Cat, and Bird.
- Channels played on TV.
- More.
More About Objects

• **Inheritance**: passing down states and behaviors from the parents to their children.

• **Interfaces**: requiring objects for the demanding methods which are exposed to the outside world.

• **Polymorphism**

• **Packages**: grouping related types, and providing access controls and name space management.

• **Immutability**

• **Enumeration types**

• **Inner classes**
First IS-A Relationship: Inheritance

- The relationships among Java classes form class hierarchy.
- We can define new classes by inheriting commonly used states and behaviors from predefined classes.
- A class is a subclass of some class, which is so-called the superclass, by using the extends keyword.
  - For example, B extends A.
- In semantics, B is a special case of A, or we could say B specializes A.
  - For example, human and dog are two specific types of animals.
- When both B and C are subclasses of A, we say that A generalizes B and C. (Déjà vu.)
- Note that Java allows single inheritance only.
Example

```java
class Animal {
    String name;
    int weight;

    Animal(String s, int w) { name = s; weight = w; }

    void eat() { weight++; }
    void exercise() { weight--; }
}

class Human extends Animal {
    Human(String s, int w) { super(s, w); } // why?
    void writeCode() {}
}

class Dog extends Animal {
    Dog(String s, int w) { super(s, w); }

    void watchDoor() {}
}
```
Recall that the keyword `this` is used to refer to the object itself.

You can use the keyword `super` to refer to (non-private) members of the superclass.

Note that `super()` can be used to invoke the constructor of its superclass, just similar to `this()`.
See Fig. 3-1 in p. 113 of Evans and Flanagan.
Constructor Chaining

- As the constructor is invoked, the constructor of its superclass is invoked accordingly.

- You might think that there will be a whole chain of constructors called, all the way back to the constructor of the class `Object`, the topmost class in Java.

- So every class is an immediate or a distant subclass of `Object`.

- For example, `finalize()` and `toString()` are inherited from `Object`.
  - `toString()`: return a string which can be any information stored in the object.
Example

```
class A {
    A() { System.out.println("A is creating..."); }
}

class B extends A {
    B() { System.out.println("B is creating..."); }

    @Override // annotation
    public String toString() { return "say something"; }
}

public class ConstructorChainingDemo {
    public static void main(String[] args) {
        B b = new B();
        System.out.println(b);
    }
}
```

- The `println()` method (and similar methods) can take an object as input, and invoke `toString()` method implicitly.
Method Overriding

• A subclass is allowed to change the behavior inherited from its superclass, if necessary.
• If one defines an instance method with its signature and return type, all identical to the one defined in its superclass, then we call it method overriding.\(^{18}\)
  • Recall that method overloading occurs only in the same class.
• Note that you could invoke the overridden method by using super.

\(^{18}\)Notice that the static methods do not follow this rule.
Binding

- Association of the method definition to the method call is known as binding.

- The binding which can be resolved at the compilation time is known as static binding or early binding.
  - They are the static, private or final methods.\(^\text{19}\)

- If the compiler is not able to resolve the binding, such binding is known as dynamic binding or late binding.
  - For example, method overriding.

\(^{19}\text{We will see the final keyword soon.}\)
• When there are multiple implementations of the method in the inheritance hierarchy, the one in the “most derived” class (the furthest down the hierarchy) always overrides the others, even if we refer to the object through a reference variable of the superclass type.\(^{20}\)
  
  • As you can see in Cat Simon.

• This is so-called **subtype polymorphism**.

\(^{20}\)An overridden method in Java acts like a virtual function in C++.
Polymorphism

- The word **polymorphism** literally means “many forms.”
- Java allows 4 types of polymorphism:
  - coercion (casting)
  - ad hoc polymorphism (overloading)
  - subtype polymorphism
  - parametric polymorphism (**generics**)
- Modeling polymorphism in a programming language lets you create a **uniform** interface to different kinds of operands, arguments, and objects.

---

Example: Uniform Interface

class Student {
    void doMyJob() { /* Do not know the detail yet. */}
}

class HighSchoolStudent extends Student {
    void doHomework() {
    }

    @Override
    void doMyJob() { doHomework(); }
}

class CollegeStudent extends Student {
    void writeFinalReports() {
    }

    @Override
    void doMyJob() { writeFinalReports(); }
}
public class PolymorphismDemo {

    public static void main(String[] args) {
        HighSchoolStudent h = new HighSchoolStudent();
        goStudy(h);
        CollegeStudent c = new CollegeStudent();
        goStudy(c);
    }

    // uniform interface, multiple implementations;
    // enhance the flexibility by reducing dependency,
    // for future extension (scalability)
    public static void goStudy(Student s) {
        s.doMyJob();
    }

    /* no need to write these methods
    public static void goStudy(HighSchoolStudent s) {
        s.doHomework();
    }

    public static void goStudy(CollegeStudent s) {
        s.writeFinalReports();
    } */
}
Subtype Polymorphism

- For convenience, let $U$ be a subtype of $T$.
- **Liskov Substitution Principle** states that $T$-type objects may be replaced with $U$-type objects without altering any of the desirable properties of $T$ (correctness, task performed, etc.).\(^22,23\)


\(^{23}\) Also see [https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)).
Casting

- **Upcasting** (widening conversion) is to cast the $U$ object to the $T$ variable.

  ```java
  T t = new U();
  ```

- **Downcasting** (narrow conversion) is to cast the $T$ variable to a $U$ variable.

  ```java
  U u = (U) t; // t is T variable reference to a U object.
  ```

- Upcasting is always allowed, but downcasting is allowed only when a $U$ object is passed to the $U$-type variable.
  - Java type system makes sure that the referenced object provides services adequate for $T$ type.
• However, type-checking in compilation time is unsound.
• The operator `instanceof` checks if the referenced object is of the type in question.
Example

```java
class T {}
class U extends T {}

public class InstanceofDemo {
    public static void main(String[] args) {
        T t1 = new T();

        System.out.println(t1 instanceof U); // output false
        System.out.println(t1 instanceof T); // output true

        T t2 = new U(); // upcasting

        System.out.println(t2 instanceof U); // output true
        System.out.println(t2 instanceof T); // output true

        U u = (U) t2; // downcasting; this is ok.

        u = (U) new T(); // pass the compilation; fail during execution!
    }
}
```
Abstraction, Method Overriding, and Polymorphism

- JVM invokes the appropriate method for the current object by looking up from the bottom of the class hierarchy to the top.
  - These methods are also called virtual methods.
- This preserves the behaviors of the subtype objects and the super-type variables play the role of placeholder.
- We often manipulate objects in an abstract level; we don’t need to know the details when we use them.
  - For example, computers, cellphones, driving.