CSCI 1130 Java

①②③④⑤⑥⑦⑧⑨⑩⑪⑫⑬⑭⑮⑯⑰⑱⑲⑳㉑㉒㉓㉔㉕㉖㉗㉘㉙㉚㉛㉜㉝㉞㉟㊱㊲㊳㊴㊵㊶㊷㊸㊹㊺㊻㊼㊽㊾㊿

Implicit coercion

> （隐式）强制类型转换

Encapsulation

> 封装

formal parameters

> 形式参数（形参）

actual parameters

> 实参

non-volatile

> 不易丢失性

threshold

> 临界值，临界点

invocation

> 调用

arithmetic

> 算数

declare

> 声明

clause

> 从句，子句

propagate

> 传播

remedy

> 补救

liability

> 责任

hierarchical

> 分层的

backslash

> 反斜杠

qualify

> 限制，限定

cumbersome

> 麻烦

utils（utilities 的缩写）

> 工具集，实用程序

inherited

> 继承的

Ethernet

> 以太网（局域网）

Fraction

> 分式

numerator

> 分子

denominator

> 分母

immutable

> 不可变的

Lectures

Lecture 1

`.java`：$Java\ Program$

`.class`：$Java\ Byte\ Code$

> `.java` $\xrightarrow[javac]{Compile}$ `.class` $\xrightarrow[Java\ Virtual\ Machine]{Translate}$ `native code` (`Intel x86` / `Smartphone` / `Mac` / `ARM`)
>
> $Java\ Byte\ Code\ is\ still\ hard\ for\ human,\ so\ we\ learn\ the\ Java\ Language$ (`.java` $file$).

Java Virtual Machine (JVM) locates the class method main() from the class `Welcome` and starts execution from there.

> `javac` 指令由编译器执行，把 java program 编译成 Java Byte Code；`java` 指令由 JVM 执行，定位到 main() 方法并开始运行

示例：一个完整的 .java 文件

```
class Welcome {

  /* The Welcome Program
     -------------------
     Illustrates a simple program displaying
     a message.
  */

  public static void main (String [ ] args) {
    System.out.println("Welcome to Java!");
  }
}
```

> `class`：Java program source files (`.java`) contain definition of classes（类）.
>
> `{}`：Curly braces pair（花括号对），不能省略
>
> `public static void main (String [ ] args) {}`：
> $a\ \mathbf{method}\ (办法)\ of\ the\ class\ \mathbf{Welcome},\ named\ \mathbf{main()}.$
>
> 注意，办法后面必须紧跟一对小括号 `()`.
>
> `/* */`：$comments$，注释
>
> `main(String [] args)`：所有都不能省略，S 要大写.

OOP

> Object-oriented Programming，面向对象编程

Objects have two properties: fields (instance variables) and methods

> 字段（实例变量）和方法

A class (e.g. Customer) is a kind of mold or template to create objects

An object is a instance of a class

The object belongs to that class

> 例如，如果有一个名为 Customer 的类，we can create instances of this class, 例如 Michael, Bill. 这里的 Michael 和 Bill 就是对象.

OOP 步骤

1.define classes

2.While the program is running, we may create objects from these classes

3.We may store information in classes and objects.

4.We send messages to a class or an object to instruct it to perform a task.

Creating Objects

An object is created by sending a `new` message to a class

> `new` 关键字：创建 `class` 的一个实例

An object consists of fields to store data and methods to manipulate the data.

A message may instruct an object to change the fields (state) of the object itself through a method.

A method may return a value to the message sender

An object may return a value to the message sender through the method

Of course, we cannot send arbitrary messages to an arbitrary classes or objects.

> 有定义，有权限

总结:

面向对象编程（OOP）将实体（entities）视为对象 (object)，由类(class) 进行建模

对象和类可以在字段 (fields) 中存储数据，并在方法 (method) 中执行操作

Lecture 2

Quotes and Brackets

'	**Single Quote**	**char 'A'**
"	Double Quote	"String"
( )	Brackets/ Parentheses	((1+3) * a)

'	**Single Quote**	**char 'A'**
[ ]	Square Brackets	array[index]
{ }	Curly Brackets/ Braces	{ block/ method }
;	Semi-colon	end of statement
.	Dot/ Full-stop	object.member
,	Comma	parameter separator

Java Naming Convention

> Capitalization，大小写规则

keyword	**class, if, double, int, for, while**
packagename	javax.swing, javaapplication12
ClassName	String, JOptionPane, Double
FileName	HelloWorld.java, JOptionPane.class
methodName()	showMessageDialog(), parseInt()
fieldName	System.out, DrinkDispenser.cokeStock
variableName	numberOfStudents, bodyWeight
CONSTANT	JFrame.HEIGHT, Math.PI

Java Program Structure

```
import <packagename>.<someclassname>;
import <packagename>.*;

class <ClassName>
{
    <type> <fieldName1>;
    <type> <fieldName2>;

    <type> <methodName1>(...)
    {
    }

    <type> <methodName2>(...)
    {
    }
}
```

Example

```
import java.util.Vector;
import javax.swing.*;

class CurrencyConverter
{
    double rateHKtoEuro = 0.0885;
    double rateHKtoUS = 0.128;

    double convertEuro2US(double amountEuro)
    {
        // details skipped
    }

    double convertUS2HK(double amountUS)
    {
        // details skipped
    }
}
```

Data Item Declaration

data 具有 identifier (name) 和 type

Java Identifiers

> 变量命名规则

- Valid components

> Space is disallowed
>
> underscore (下划线) 可以

- Must start with a letter or underscore

> 不能数字开头

- Case sensitive

关键字 `new`：创建类的实例.

`ClassName` 和 `FileName` 首字母大写（recommend）

`fieldName` 小写（类中的一个域）

> $System.out$

```
import <packagename>.*;
```

> `.*`: all classes

`method` 中声明的变量是局部变量 (local variable).

`class` 中、`method` 外声明的变量是 `field name`，可以从类内或类外访问（全局？）.

Declaration of a Field/Local Variable

4 种声明语法

```
<type name> identifier;

<type name> identifier1, identifier2, … ;

<type name> identifier = <initial value>;

<type name> identifier1 = <initial value1>,        identifier2 = <initial
value2>, …;
```

> 这里 Type 可以是 primitive type 或者 class

Example

```
String myName = "Michael Fung";
double HangSengIndex = 17894.03, rainfall = 3.4;
```

Primitive Types

> 程序自带，直接使用

byte	**for storing a 8-bit signed integer**	**[-128 → 127]**
short	for storing a 16-bit signed integer	[-32768 → 32767]
int	for storing a 32-bit signed integer	[-2147483648 → 2147483647]
long	for storing a 64-bit signed integer	[-9223372036854775808 → 9223372036854775807]

byte	**for storing a 8-bit signed integer**	**[-128 → 127]**
float	for storing a 32-bit real number	[$\pm 10^{38}$ with floating point]
double	for storing a 64-bit real number	[$\pm 10^{308}$ with floating point]
char	for storing a single character	['A', 'B', ..., 'a', 'b', ..., '0', '1', ..., '!', '#']
boolean	for storing a true/false (logic) value	[true, false]

> char 单引号，string 双引号
>
> Java 中，变量类型没有 `unsigned`
>
> Java 的 `boolean` 类型和 `C` 的不同，`C` 中 0 为 False，任意非 0 整数为 True.

Type Casting

> 显式类型转换，强制类型转换

```
( <type_name> ) some_value
float f3 = (float) 3.14159;
```

```
        double d1 = 3.14159;            // ok
        double d2 = 3e8;                // ok
        double d3 = -0.27e-5;           // ok
        float  f1 = 3.14159;            // not ok

        float  f2 = 3.14159F;           // ok
        float  f3 = -0.27e-5f;          // ok
        float  f3 = (float) 3.14159;    // ok
        float  f4 = (float) d1;         // ok
```

> 注意，3.14159 并不是超出了 float 的限制才需要转换，而是 java 会把小数默认为 double 类型，double 类型的数不能赋给 float，所以需要转换（就 3.14159 而言，精度没有丢失）
>
> 直接在小数末尾加 f 或 F，也可以改变这种 default，然后无需 type casting 就可以赋给 float
>
> e-5: 10 的 -5 次方

`final`：常数，不允许改变

```java
class UStoHKConverter {
    static final double rate = 7.80;
    public static void main (String [ ] args) {
        double USdollar = 123.45;
        double HKdollar = USdollar * rate;
        System.out.println ("US" + USdollar + " -> HK" + HKdollar);
    }
}
```

> 注意，`System.out.println();` 不支持占位符（支持转义字符）.
>
> `System.out.printf();` 支持占位符.

Java 和 Python 的 concatenation 特性不同

> Java 把 string 和 number 串联，不需要先把 number 强制转换为 string，直接加号就可以.

1. `float f1 = 3.14159;`:
    - 这里的 `3.14159` 被视为一个 `double` 类型的常量.
    - `double` 是默认小数类型，所以编译器会报错，提示你不能将 `double` 直接赋值给 `float`，除非进行强制转换.
2. `float f2 = 3.14159F;`:
    - 后缀 `F` 或 `f` 明确表示该常量是 `float` 类型，因此可以直接赋值给 `float` 变量.

推荐总是使用 `F` 后缀来避免潜在的类型错误.

```java
class DataType {        /* testing of primitive data types */
    public static void main (String [ ] args) {
        byte int8 = -128;
        System.out.println(int8);
        int8 = (byte) 137;  /* type casting with an overflow */
        System.out.println(int8);
        boolean ok = (3 > 7);   /* boolean expression */
        System.out.println(ok);
        System.out.println("Hello \"World\" !!!");
        double GPA = 3;
        System.out.println("GPA = " + GPA);
        System.out.println("(int)(5.23)      = " + (int)(5.23));
        System.out.println("Math.floor(5.23) = " + Math.floor(5.23));
        System.out.println("Math.ceil(5.23)  = " + Math.ceil(5.23));
        System.out.println("Math.round(5.23) = " + Math.round(5.23));
        System.out.println("(int)(-5.23)      = " + (int)(-5.23));
```

```
        System.out.println("Math.floor(-5.23) = " + Math.floor(-5.23));
        System.out.println("Math.ceil(-5.23)  = " + Math.ceil(-5.23));
        System.out.println("Math.round(-5.23) = " + Math.round(-5.23));
    }
}
```

输出

```
-128
-119
false
Hello "World" !!!
GPA = 3.0
(int)(5.23)       = 5
Math.floor(5.23) = 5.0
Math.ceil(5.23)  = 6.0
Math.round(5.23) = 5
(int)(-5.23)      = -5
Math.floor(-5.23) = -6.0
Math.ceil(-5.23)  = -5.0
Math.round(-5.23) = -5
```

Java 中的 float 由 binary 表示，有一些特性

> 0.7 * 0.7 不等于 0.49（差一点点）
>
> 0.5 * 0.5 = 0.25（精确相等）

advanced example

```
Benz   peter    = new Benz();
Car    michael  = (Car) peter;
  // type-casting also works on class type objects
  // peter keeps a Benz object
  // we can "down-convert" it and consider it a Car
```

Integer-to-integer type casts

- byte → short → int → long　　Always ok
- long → int → short → byte　　May not...
- Overflow may occur, causing errors.

The Math Class

The class Math is provided with Java, offering various methods for simple mathematical operations.

To use such methods, we have to send message to the Math class:

```
double answer1;
answer1 = Math.sqrt(49);
```

methods in the math class

PI is a class constant field pre-stored with many digits of π.

Notice that sin, cos, tan, etc. uses radian instead of degree.

Don’t miss the pair of parenthesis in sending messages to the Math class!

General form of a such a message:
class_name.method_name(input);
Math.function(input);

Operators

Real and Integer Division

Short-hand Operators

Javadoc

小数点算1位

Lecture 3

Structured programming in Java

Branching: if / if-else

Two categories: if-statement and switch-statement

> 这里不讨论 switch 语句

if-else 和 C 基本一致

condition is a boolean expression.

An **else**-part attaches to the nearest available **if**,

Repetition: for loop

deterministic loop

> 确定性循环，知道循环次数

loop counter

The start portion is executed one-and-only-once.

If there are more than one statements in the loop-body, a pair of curly braces is needed.

The start-check-update complex usually concerns one single loop counter variable.

Either or both of the elements start, check, update and even the body could be absent.

- Nevertheless, the punctuations `;` must be there.

> 可以放空，但是分号一定要写

Nested for loops

Lecture 4

Method and Object

method 类似 C 的函数

Class with Fields and Methods

```
class MyVector {
    protected              double x, y, z;
    protected static final int dimension = 3;

    public void setX(double valueX) {
        x = valueX;
```

```
        // likewise for setY() and setZ()
    }

    public double length() {
        double answer;
        answer = Math.sqrt(x * x + y * y + z * z);
        return(answer);
    }
}
```

Method Declaration 语法

```
modifiers type method_name ( parameters )
{
    local variable declarations and statements
    return expression;  //depend on method type
}
```

modifier: public（所有class可访问）, private, static（不能乱删，静态上下文不能引用非静态方法）

type: void / a primitive type / a class type

> method 的 type 指它传回给 message sender 的数据的类型（void 不传）
>
> private 是其他类无法访问，本class内的其他method仍然可以访问（不然就没意义了，哪里都无法访问）

child 子类

method的传参要匹配，多个参数时每个都要声明变量类型

Example

```
class MoneyExchangeBox {
    static double toHKdollar(double USdollar) {
        return USdollar * 7.80;
    }

    static void box() {
        System.out.println("----------------------");
        System.out.println("|                     |");
        System.out.println("| US$" + 1 + " = HK$" + toHKdollar(1) + " |");
        System.out.println("|                     |");
        System.out.println("----------------------");
    }

    public static void main(String[] args) {
        box();
        System.out.println("US$" + 5 + " = HK$" + toHKdollar(5));
    }
}
```

To call/invoke a method

> write out method name with parentheses and `;`
>
> supply exact number of parameter(s)
>
> matched type
>
> correct order

C 的函数引用要按顺序（不然就要在开头声明），java可以随便放位置

类方法（静态方法）的调用：

1. 直接调用 `classMethod();`

在同一个类中，是可以直接调用 `classMethod();` 的，无需加类名前缀. 例如，在 `Example` 类的 `main` 方法或其他静态方法中，可以直接这样调用.

```
public class Example {
    public static void classMethod() {
        System.out.println("This is a class method");
    }

    public static void main(String[] args) {
        classMethod();  // 在同一个类中可以直接调用
    }
}
```

2. `Example.` 的作用：

`Example.` 是类名前缀，它明确指出我们要调用的是 `Example` 类中的方法. 这在以下情况下是必要或有用的：

- 在其他类中调用这个静态方法
- 当有命名冲突时，用于明确指定要使用的类
- 提高代码可读性，明确方法来源

例如：

```
public class AnotherClass {
    public static void main(String[] args) {
        Example.classMethod();  // 在其他类中必须使用类名来调用静态方法
    }
}
```

3. 静态导入：

如果你经常使用某个类的静态方法，可以使用静态导入来避免每次都写类名：

```java
import static com.example.Example.classMethod;

public class AnotherClass {
    public static void main(String[] args) {
        classMethod();  // 现在可以直接调用，无需类名
    }
}
```

4. 注意事项：

- 虽然可以通过对象实例调用静态方法（如 `obj.classMethod();`），但这不是推荐的做法，因为它可能导致混淆.
- 始终通过类名调用静态方法可以提高代码的清晰度和可维护性.

总结：

- 在同一个类中，可以直接调用静态方法，无需类名.
- 在其他类中，需要使用类名来调用静态方法.
- 类名前缀（如 `Example.`）用于明确指定方法所属的类，增加代码的清晰度.

Compound Statement (Code Block)

Public and Private Fields

Modifiers **public** and **private** do not only apply to methods, but also apply to fields.

```java
class AngryWife {
      private double accountBalance;    // a private variable
      public final char sex = 'F';  // a public constant
      public void withdraw()        // a method
      {
        // only methods of AngryWife can touch private fields
       // but not methods of PoorHusband
        accountBalance -= 100.00;
        ...
      }
      ...
{
```

- Modifier `public` indicates that a field or a method or a constructor can be referenced outside its own class, i.e. by other classes.
- Modifier `private` indicates that a field or a method or a constructor can ONLY be referenced within its own class.
- Modifier `protected` will be covered later.
    - Somehow in between public and private.

Send a new *Message*

Classes can accept a special message, namely new.

This special message is sent as follows:

```
new classname(parameters);
```

On receiving this message, a new object of the indicated class will be created.

Constructor

> 构造函数

We need a constructor to construct, create, initialize a new object for us.

- e.g. Filling in the fields

Constructor is one special kind of method of a class, bearing same name as the class.

Usage:

```
<classname> <variable_name>;
<variable_name> = new <classname>(...);

<classname> <variable_name> = new <classname>(...); // 可以合起来写. 类似 C 的 int i = 7;（声明和初始化同时）
```

> new 为新对象分配内存. 仅就这一功能而言，类似c的malloc()
>
> 调用对象的构造函数（用于初始化）
>
> 返回对新创建对象的引用（即内存地址）
> 作用：允许多个变量引用同一个对象，垃圾回收机制的基础

Constructors Example

```
class Boy {
    public int age;

    public Boy() {
        age = 0;
    } // Constructor 1
```

```
    public Boy(int initialAge) {
        age = initialAge;
    } // Constructor 2
}

class Family {
    public static void main(String[] args) {
        Boy michael = new Boy();
        Boy anson = new Boy(3); // michael 和 anson 存储对象(object)的引用
(reference)（地址）
    }
}
```

构造函数（Constructor）和方法（Method）有什么区别?

您提出了一个很好的问题. 实际上，您可能混淆了构造函数（constructor）和方法（method）的概念. 让我来澄清一下:

1. 构造函数 vs 方法:
    - 构造函数是一种特殊的方法，用于初始化对象.
    - 构造函数的名称必须与类名相同.
    - 构造函数没有返回类型，甚至不用写void.
2. new关键字和构造函数:
    - 当使用new关键字创建对象时，它会调用相应的构造函数，而不是普通方法.
    - 类不需要定义一个与类同名的普通方法.
3. 默认构造函数:
    - 如果你没有在类中定义任何构造函数，Java会自动提供一个无参数的默认构造函数.
    - 一旦你定义了任何构造函数，Java就不会再提供默认构造函数.
4. 示例:

```
public class MyClass {
    // 构造函数，不是方法
    public MyClass() {
        System.out.println("对象被创建了");
    }

    // 这是一个普通方法，不是构造函数
    public void myMethod() {
        System.out.println("这是一个方法");
    }
}

// 使用
MyClass obj = new MyClass();  // 调用构造函数
obj.myMethod();  // 调用普通方法
```

5. 多个构造函数：
    - 一个类可以有多个构造函数，这称为构造函数重载.

```
public class Person {
    private String name;
    private int age;

    // 无参构造函数
    public Person() {
        this.name = "Unknown";
        this.age = 0;
    }

    // 带参数的构造函数
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

总结：使用new关键字时，类不需要定义一个同名的方法，但需要有一个构造函数（如果没有显式定义，则使用默认构造函数）. 构造函数看起来像方法，但它有特殊的作用和语法规则.

注意：Constructors normally perform some initialization for the newly created object. They **must NOT** have modifiers such as **static, final**, ... Normally, the only permissible modifier is **public**.

Modifiers 固定为 public，且没有返回类型（不用写 void）

创建对象的四步骤

1.According to the known resource requirement (such as memory size) of the new object, the **new** message will *implicitly* cause the Java system to *allocate* resources such as *memory* block for storing the object.

2.The new object is then prepared with *a new set of instance fields and methods*, according to the class design.

3.After memory allocation, the corresponding *constructor* method is invoked (called) to *initialize* the new object.

4.An *object reference* is *returned* finally, to be stored in a field/ variable of the matching object type.

Object Reference

> 就是地址

static vs instance

> (class vs object)

	static / class	instance / object
Methods	static, a single copy	no modifier static, a copy for each object
Fields	static, a single copy	no modifier static, a copy for each object

We **MUST** create new object(s) before using instance methods and fields.

类方法和实例方法

Class and Instance Methods

- A method defined for a class is called a $class\ method$.

```
static void main()
```

- A method defined for an object is called an $instance\ method$ (no modifier static).

```
void deposit()
```

class method（类方法）和instance method（实例方法）是Java中两种不同类型的方法，它们有几个关键区别：

1. 声明方式：
   - class method: 使用 `static` 关键字声明
   - instance method: 不使用 `static` 关键字
2. 调用方式：
   - class method: 可以通过类名直接调用，也可以通过对象调用
   - instance method: 只能通过对象实例调用
3. 访问权限：
   - class method: 只能直接访问类的静态成员（静态变量和其他静态方法，不能访问实例字段）
   - instance method: 可以访问类的所有成员（静态和非静态）

4. this关键字：
    - class method: 不能使用 `this` 关键字
    - instance method: 可以使用 `this` 关键字引用当前对象
5. 内存分配：
    - class method: 属于类，只有一份
    - instance method: 属于对象，每个对象都有一份
6. 重写（Override）：
    - class method: 不能被重写
    - instance method: 可以被子类重写

示例：

```
public class Example {
    private static int staticVar = 0;
    private int instanceVar = 0;

    // 类方法（静态方法）
    public static void classMethod() {
        System.out.println("This is a class method");
        System.out.println(staticVar);  // 可以访问静态变量
        // System.out.println(instanceVar);  // 错误！不能直接访问实例变量
        // this.instanceMethod();  // 错误！不能使用this
    }

    // 实例方法
    public void instanceMethod() {
        System.out.println("This is an instance method");
        System.out.println(staticVar);  // 可以访问静态变量
        System.out.println(instanceVar);  // 可以访问实例变量
        this.anotherInstanceMethod();  // 可以使用this
    }

    public void anotherInstanceMethod() {
        System.out.println("Another instance method");
    }

    public static void main(String[] args) {
        // 调用类方法
        Example.classMethod();  // 通过类名调用

        Example obj = new Example();
        obj.classMethod();  // 通过对象调用（不推荐）

        // 调用实例方法
        // Example.instanceMethod();  // 错误！不能通过类名调用实例方法
        obj.instanceMethod();  // 正确，通过对象调用
    }
}
```

使用场景：

- 类方法通常用于不需要访问对象状态的操作，如工具函数或工厂方法.

- 实例方法用于需要访问或修改对象状态的操作.

Why Class Methods?

Intuitive and global in nature

- Do not require object creation before using such class methods.
- e.g. Math.sin() is a global and commonly used method.

Object factory (a programming pattern)

- Sometimes new and constructor is not suitable and we want a specialized class method to create object for us. (Advanced)
- e.g. createAccount()

The start-up method main() for Java Applications

- JVM requires a class method static void main() for start-up

Class Method Calling Syntax

```
ClassName.staticMethodName(parameters)  // ClassName. 是类名前缀
```

- Calling class (static) method of a class.

```
staticMethodName(parameters) // 同一个class中可以省略类名前缀
```

- Calling class (static) method of same class (omit class name).

```
someObject.staticMethodName(parameters) // 虽然可以通过对象实例调用静态方法，但不推荐，因为它可能导致混淆.
```

- Calling class (static) method of the class of some object (automatic class name inference/determination).

Instance Method Calling Syntax

```
someObject.methodName(parameters)
```

- Calling instance method of some object.

```
methodName(parameters) // 在一个实例方法内部，可以直接调用同一类的其他实例方法，无需使用对象引用

public class Example {
    public void methodA() {
        System.out.println("Method A");
        methodB();  // 直接调用另一个实例方法
    }

    public void methodB() {
        System.out.println("Method B");
    }
}
```

- Calling instance method of this object (itself).

```
this.methodName(parameters) // 在实例方法内部，可以使用this关键字来引用当前对象：

public class Counter {
    private int count = 0;

    public void increment() {
        this.count++;  // 使用this引用当前对象的count变量
    }

    public int getCount() {
        return this.count;  // 返回当前对象的count值
    }
}
```

- Calling instance method of this object (itself).

```
super.methodName(parameters) // 调用该对象的父类实例方法
```

- Calling super-class instance method of this object.

调用对象的父类实例方法是Java继承机制中的一个重要概念. 这通常通过`super`关键字来实现. 让我们详细探讨一下:

1. 基本语法:

在子类方法中调用父类方法的基本语法是:

```
super.parentMethodName(parameters);
```

2. 实际例子:

```
// 父类
class Animal {
    public void makeSound() {
        System.out.println("The animal makes a sound");
    }
}
```

```
// 子类
class Dog extends Animal {
    @Override
    public void makeSound() {
        // 调用父类的makeSound方法
        super.makeSound();
        // 然后添加Dog特有的行为
        System.out.println("The dog barks");
    }

    public void doSomething() {
        // 在其他方法中也可以调用父类方法
        super.makeSound();
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.makeSound();
    }
}
```

输出：

```
The animal makes a sound
The dog barks
```

3. 构造函数中调用父类构造函数：

在子类构造函数中，可以使用`super()`调用父类构造函数：

```
class Animal {
    protected String name;

    public Animal(String name) {
        this.name = name;
    }
}

class Dog extends Animal {
    public Dog(String name) {
        super(name);  // 调用父类的构造函数
    }
}
```

4. 注意事项：

- `super`关键字只能在非静态方法中使用.
- 如果子类和父类有同名方法，不使用`super`会调用子类的方法.
- 在构造函数中，`super()`必须是第一条语句.

5. 访问父类的成员变量：

虽然不常用，但也可以使用`super`访问父类的成员变量：

```
class Parent {
    protected int x = 10;
}

class Child extends Parent {
    private int x = 20;

    public void printX() {
        System.out.println("Child's x: " + this.x);
        System.out.println("Parent's x: " + super.x);
    }
}
```

6. 方法重写与super：

当子类重写父类方法时，使用`super`可以调用被重写的父类方法：

```
class Vehicle {
    public void start() {
        System.out.println("Vehicle is starting");
    }
}

class Car extends Vehicle {
    @Override
    public void start() {
        super.start();  // 调用父类的start方法
        System.out.println("Car is starting");
    }
}
```

7. 多层继承：

在多层继承中，`super`总是指向直接父类：

```
class Grandparent {
    public void method() {
        System.out.println("Grandparent's method");
    }
}

class Parent extends Grandparent {
    @Override
    public void method() {
        super.method();  // 调用Grandparent的method
        System.out.println("Parent's method");
    }
}

class Child extends Parent {
    @Override
    public void method() {
        super.method();  // 调用Parent的method
```

```
        System.out.println("Child's method");
    }
}
```

使用`super`调用父类方法是实现代码重用和扩展父类功能的重要机制. 它允许子类在保留父类行为的同时添加或修改自己的行为.

Example

```
public class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }

    // 实例方法
    public void sayHello() {
        System.out.println("Hello, my name is " + name);
    }

    // 带参数的实例方法
    public void greet(String otherName) {
        System.out.println("Hello " + otherName + ", nice to meet you!");
    }
}

public class Main {
    public static void main(String[] args) {
        // 创建Person对象
        Person person = new Person("Alice");

        // 调用无参数的实例方法
        person.sayHello();

        // 调用带参数的实例方法
        person.greet("Bob");
    }
}
```

注意事项:

- 必须先创建对象实例才能调用实例方法.
- 实例方法可以访问对象的实例变量和其他实例方法.
- 实例方法也可以访问类的静态成员（静态变量和静态方法）.

方法链

有时候，方法会返回对象本身，这样可以实现方法链：

```
public class StringBuilder {
    private String str = "";

    public StringBuilder append(String s) {
        this.str += s;
        return this;  // 返回对象本身
    }

    public String toString() {
        return str;
    }
}

// 使用方法链
StringBuilder sb = new StringBuilder();
String result = sb.append("Hello").append(" ").append("World").toString();
```

类字段和实例字段

Class Fields（类字段）和 Instance Fields（实例字段）是 Java 中两种不同类型的字段（或称为成员变量），它们有几个关键区别：

1. 声明方式：
    - Class Fields: 使用 `static` 关键字声明
    - Instance Fields: 不使用 `static` 关键字
2. 内存分配：
    - Class Fields: 只分配一次内存，被该类的所有实例共享
    - Instance Fields: 每个对象实例都有自己的一份拷贝
3. 访问方式：
    - Class Fields: 可以通过类名直接访问，也可以通过对象实例访问
    - Instance Fields: 只能通过对象实例访问
4. 生命周期：

- Class Fields: 随类的加载而创建，随类的卸载而销毁
- Instance Fields: 随对象的创建而创建，随对象的垃圾回收而销毁

5. 用途：
   - Class Fields: 用于存储所有实例共享的数据或常量
   - Instance Fields: 用于存储每个对象特有的状态

示例代码：

```
public class Example {
    // 类字段（静态字段）
    public static int classField = 0;

    // 实例字段
    public int instanceField = 0;

    public void incrementFields() {
        classField++;
        instanceField++;
    }

    public static void main(String[] args) {
        Example obj1 = new Example();
        Example obj2 = new Example();

        obj1.incrementFields();
        obj2.incrementFields();

        System.out.println("obj1 - Class Field: " + Example.classField);
        System.out.println("obj1 - Instance Field: " + obj1.instanceField);
        System.out.println("obj2 - Class Field: " + Example.classField);
        System.out.println("obj2 - Instance Field: " + obj2.instanceField);
    }
}
```

输出：

```
obj1 - Class Field: 2
obj1 - Instance Field: 1
obj2 - Class Field: 2
obj2 - Instance Field: 1
```

解释：

- `classField` 被两个对象共享，所以最终值为 2
- 每个对象有自己的 `instanceField`，所以各自的值为 1

6. 初始化：
   - Class Fields: 可以在静态初始化块中初始化
   - Instance Fields: 可以在构造函数或实例初始化块中初始化

```
public class Example {
    public static int classField;
```

```
    public int instanceField;

    // 静态初始化块
    static {
        classField = 10;
    }

    // 实例初始化块
    {
        instanceField = 20;
    }

    public Example() {
        // 构造函数中也可以初始化实例字段
        instanceField = 30;
    }
}
```

7. 多线程考虑：
    - Class Fields: 在多线程环境中需要特别注意，因为它们被所有实例共享
    - Instance Fields: 相对安全，因为它们属于特定的对象实例

总结：

- 类字段用于表示所有实例共享的状态或常量
- 实例字段用于表示每个对象特有的状态
- 选择使用哪种类型的字段取决于你的设计需求和数据的性质

Class Field Referencing Syntax

> 访问类字段

className.staticFieldName

- Referring to class (static) field of a class.

staticFieldName

- Referring to class (static) field of same class (omit class name).

someObject.staticFieldName

- Referring to class (static) field of the class of some object (automatic class name inference/determination).

访问类字段（也称为静态字段）的语法与访问实例字段有一些不同. 类字段是属于类本身的，而不是类的实例. 以下是访问类字段的主要方式：

1. 在类的内部访问：

在类的任何方法（包括静态方法）中，可以直接使用字段名来访问类字段：

```
public class Counter {
    public static int count = 0; // 类字段

    public static void increment() {
        count++; // 直接访问类字段
    }

    public void printCount() {
        System.out.println(count); // 非静态方法也可以直接访问
    }
}
```

2. 使用类名访问：

无论是在类内部还是外部，都可以使用类名来访问类字段. 这是最推荐的方式，因为它明确指出了字段属于哪个类：

```
public class Counter {
    public static int count = 0;

    public static void printCount() {
        System.out.println(Counter.count); // 使用类名访问
    }
}

// 在其他类中
public class Main {
    public static void main(String[] args) {
        System.out.println(Counter.count); // 通过类名访问
        Counter.count++; // 通过类名修改
    }
}
```

3. 通过实例访问：

虽然可以通过类的实例来访问类字段，但这不是推荐的做法，因为它可能导致混淆：

```
Counter c = new Counter();
System.out.println(c.count); // 可以，但不推荐
```

4. 在静态方法中：

静态方法只能直接访问类字段，不能直接访问实例字段：

```
public class Example {
    public static int staticField = 10;
    public int instanceField = 20;

    public static void staticMethod() {
        System.out.println(staticField); // 正确
        // System.out.println(instanceField); // 错误！不能直接访问实例字段
    }
}
```

5. 通过静态导入：

使用静态导入可以在不使用类名的情况下访问类字段：

```
import static com.example.Counter.count;

public class Main {
    public static void main(String[] args) {
        System.out.println(count); // 直接使用，无需类名
    }
}
```

6. 在子类中访问父类的类字段：

子类可以直接访问父类的非私有类字段：

```
public class Parent {
    protected static int sharedValue = 100;
}

public class Child extends Parent {
    public static void printValue() {
        System.out.println(sharedValue); // 直接访问父类的类字段
    }
}
```

7. 使用反射访问（高级用法）：

可以使用反射来访问类字段，包括私有字段：

```
import java.lang.reflect.Field;

public class Main {
    public static void main(String[] args) throws Exception {
        Field countField = Counter.class.getDeclaredField("count");
        countField.setAccessible(true);
        System.out.println(countField.get(null)); // 对于静态字段，传入 null
    }
}
```

注意事项：

- 类字段在内存中只有一个副本，被所有实例共享.
- 在多线程环境中访问或修改共享的类字段时，需要考虑同步问题.
- `final` 类字段必须在声明时或静态初始化块中初始化.
- 过度使用类字段可能导致紧耦合，影响程序的模块化和可测试性.

Why Class Fields?

- Intuitive

    - e.g. minimumBalance is kept once for all accounts in a bank rather than storing individually in each object.
- Saves memory space
    - Only keep a single copy in the class.
- Avoid updating multiple copies of data values
    - Consistency concern.
    - Efficiency concern.

Instance Field Referencing Syntax

> 访问实例字段

someObject.fieldName

- Referring to instance field of some object.

fieldName

- Referring to instance field of this object (itself).

this.fieldName

- Referring to instance field of this object (itself).

访问实例字段的语法非常直接，主要有以下几种方式：

1. 在类的内部访问：

在类的方法内部，可以直接使用字段名来访问实例字段：

```
public class Person {
    private String name; // 实例字段

    public void printName() {
        System.out.println(name); // 直接访问实例字段
    }

    public void setName(String newName) {
        name = newName; // 直接设置实例字段
    }
}
```

2. 使用 this 关键字：

虽然不总是必要，但使用 `this` 可以明确指出我们正在访问的是实例字段，特别是当有局部变量与实例字段同名时：

```
public class Person {
    private String name; // 实例字段

    public void setName(String name) {
        this.name = name; // 使用 this 来区分实例字段和参数
    }
}
```

3. 从类的外部访问：

从类的外部访问实例字段时，需要通过对象引用：

```
public class Main {
    public static void main(String[] args) {
        Person person = new Person();
        person.name = "Alice"; // 直接访问 (假设 name 是 public)
    }
}
```

4. 通过 getter 和 setter 方法访问：

通常，为了封装和数据保护，我们会将字段声明为 private，并通过公共的 getter 和 setter 方法来访问：

```
public class Person {
    private String name; // private 实例字段

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}

// 在其他类中使用
public class Main {
    public static void main(String[] args) {
        Person person = new Person();
        person.setName("Bob");
        System.out.println(person.getName());
    }
}
```

5. 在子类中访问父类的实例字段：

如果父类的实例字段是 protected 或 public，子类可以直接访问：

```java
public class Animal {
    protected String species;
}

public class Dog extends Animal {
    public void printSpecies() {
        System.out.println(species); // 直接访问父类的 protected 字段
    }
}
```

6. 使用反射访问（高级用法）：

在某些特殊情况下，可以使用反射来访问私有字段，但这通常不推荐：

```java
import java.lang.reflect.Field;

public class Main {
    public static void main(String[] args) throws Exception {
        Person person = new Person();
        Field nameField = Person.class.getDeclaredField("name");
        nameField.setAccessible(true);
        nameField.set(person, "Charlie");
        System.out.println(nameField.get(person));
    }
}
```

注意事项：

- 直接访问实例字段（特别是public字段）可能破坏封装性，通常建议使用getter和setter方法.
- 在多线程环境中访问共享的实例字段时，需要考虑同步问题.
- 访问 `final` 实例字段时，只能在构造方法或初始化块中设置它们的值.

Variables VS Constants

A data item that can change over time is called a variable.

A data item that cannot change over time is called a constant.

常量：在Java中使用 `final` 关键字声明

```java
public class Example {
    // 变量
    int count = 0;
    String name;

    // 常量
```

```
    final double PI = 3.14159;
    static final int MAX_USERS = 100;

    public void demonstrateVariablesAndConstants() {
        // 使用变量
        count = 10;
        name = "John";
        count++; // 可以改变

        // 使用常量
        System.out.println(PI);
        System.out.println(MAX_USERS);

        // 尝试修改常量会导致编译错误
        // PI = 3.14; // 错误!
        // MAX_USERS = 200; // 错误!
    }
}
```

Lecture 5

Object reference, FileO

An object can be taken care by more than one object references.

Primitive type entities	Class type entities
Don't need creation.	Need creation of object.
Store the primitive value.	Store an object reference.
Copy the value.	Copy the object reference, not the object itself

Copy object reference 类似 C 的指针传地址

Using objects in Methods

A method can access to its own local variables and fields.

A method can get data from message sender via parameters.

A method can return data to message sender.

Method parameters can be of Class Type

Method type can be of Class Type

In either/both cases, it involves object reference copying

Parameter Passing of Object Reference

If one of the parameters of a method is an object reference, object reference copying will occur.

```
// CUHK.java

class Employee {
    public double salary;

    // 构造函数，初始化工资
    public Employee() {
        this.salary = 5000; // 默认工资为5000
    }
}

class CUHK {
    private static void fire(Employee victim) {
        victim.salary = 0;
    }

    public static void main(String[] args) {
        Employee michael = new Employee(); // 创建一个员工实例
        CUHK.fire(michael);
        System.out.println(michael.salary);
    }
}
```

method `fire` 的参数是 object reference，不会创建新的实例，只是 copy object reference

> 类似 C 的形参和实参. 此处 victim 类似形参，michael 类似实参.

Return Object Reference

If the return value of a method is an object reference, object reference copying also occurs.

```
// HKSAR.java

class Employee {
    String name;

    public Employee(String name) {
```

```
        this.name = name;
    }
}

class CUHK {
    static Employee employ() {
        Employee newbie = new Employee("Anson");
        return newbie;
    }
}

class HKSAR {
    public static void main(String[] args) {
        Employee instructor = CUHK.employ();
        System.out.println(instructor.name);
    }
}
```

> 同样是 object reference copying, newbie copy 给了 instructor

Lecture 6

> Exception, Scoping

6.1 Exception

> 例外，异常

When running a program, there may be unexpected conditions or errors.

E.g.

- Keyboard cable damaged by mice bite.
- I/O Error (disk damage, disk full, etc.)

Java has defined some commonly used exception names:

- IOExpection, ArithmeticException, EOFException, etc.

In any **method**, we may:

- Raise exceptions (throw)

- Detect exceptions (*try*) and Handle exceptions (*catch*)
- Ignore exceptions (*declare throws* clause and propagate)
    - Ignore means the method would suicide.
    - The suicided method would become a killer (exception propagation).

6.1.1 IOException Objects

- Used when there are something gone wrong during general Input/Output operations.
- We may create (new) an IOException object from the class `java.io.IOException`.

```
import java.io.*;

public class Trigonometry {

    // 模拟从文件中读取并计算 tan(angle) 的方法
    public double tan(double angle) throws IOException { // 这个 method 有异常抛出，声明时必须 throws，否则编译错误
        double answer = 0.0;

        // 人为抛出 IOException
        // 假设我们尝试读取一个不存在的文件
        File file = new File("nonexistent_file.txt");
        if (!file.exists()) {
            throw new IOException("File not found: " + file.getName());
        }

        // 如果文件存在，通常这里会是读取文件并计算 tan(angle) 的逻辑
        // 但我们为了演示异常，直接抛出异常

        return answer;
    }

    public static void main(String[] args) {
        Trigonometry trig = new Trigonometry();

        try {
            // 调用 tan 方法，传入 30 度
            double value = trig.tan(30);
            System.out.println("tan(30) = " + value);
        } catch (IOException e) {
            // 处理抛出的 IOException
            System.err.println("An error occurred: " + e.getMessage());
        }
    }
}
```

若 `nonexistent_file.txt` 文件不存在，控制台输出为：

```
An error occurred: File not found: nonexistent_file.txt
```

`Java` 中，`IOException` 类提供了多个构造函数，允许开发者在不同场景下创建 `IOException` 对象.

这里的 `new IOException();` 是调用了 `IOException` 类的一个无参构造函数.

`IOException` 的常见构造函数声明

1. **无参构造函数**:

   ```
   public IOException()
   ```

   - 该构造函数创建一个没有详细错误信息的 `IOException` 对象.

2. **带错误消息的构造函数**:

   ```
   public IOException(String message)
   ```

   - 该构造函数接收一个 `String` 类型的参数，用来描述异常的详细信息. 你可以通过此构造函数向异常对象中传递自定义的错误信息. 例如:

     ```
     throw new IOException("File not found");
     ```

3. **带错误消息和原因的构造函数**（Java 1.6 及以上）：

   ```
   public IOException(String message, Throwable cause)
   ```

   - 该构造函数允许你传递一个详细的错误信息和一个导致当前异常的原因（另一个异常对象）. 这对于捕获并重新抛出异常时非常有用.

4. **带原因的构造函数**（Java 1.6 及以上）：

   ```
   public IOException(Throwable cause)
   ```

   - 该构造函数允许你传递另一个异常作为原因，而不需要提供额外的错误消息.

Examples:

- 无参构造函数:

  ```
  IOException ioException = new IOException();
  ```

- 带错误消息的构造函数:

```
IOException ioException = new IOException("An I/O error occurred");
```

- 带错误消息和原因的构造函数：

```
IOException ioException = new IOException("An I/O error occurred",
anotherException);
```

- 带原因的构造函数：

```
IOException ioException = new IOException(anotherException);
```

无参构造函数 `public IOException()` 源码

在 `Java` 中，`IOException` 是一个标准库类，位于 `java.io` 包中. 为了查看其源码，尤其是无参构造函数的定义，可以参考标准 Java Development Kit (JDK) 中的源代码.

`IOException` 是 `java.lang.Exception` 的子类，而 `Exception` 又是 `java.lang.Throwable` 的子类. `IOException` 的无参构造函数是继承自 `Throwable` 类的.

在 JDK 的 `IOException` 类中，无参构造函数简单地调用了父类（即 `Exception`）的无参构造函数. 以下是 `IOException` 类的部分源代码：

```
package java.io;

import java.lang.Exception;

/**
 * Signals that an I/O exception of some sort has occurred. This
 * class is the general class of exceptions produced by failed or
 * interrupted I/O operations.
 *
 * @see java.io.InputStream
 * @see java.io.OutputStream
 * @see java.nio.file.Files
 * @see java.nio.file.Paths
 *
 * @author  unascribed
 * @since   1.0
 */
public class IOException extends Exception {
    /**
     * Constructs an {@code IOException} with {@code null}
     * as its error detail message.
     */
    public IOException() {
        super();
```

```
    }

    /**
     * Constructs an {@code IOException} with the specified detail message.
     *
     * @param   message   The detail message (which is saved for later retrieval
     *                    by the {@link #getMessage()} method)
     */
    public IOException(String message) {
        super(message);
    }

    // Other constructors and methods...
}
```

- `public IOException()`：这个是 `IOException` 的无参构造函数，它调用了父类 `Exception` 的无参构造函数（`super();`）.
  - `super();` 是调用父类构造函数的语法. 这意味着 `IOException` 对象在创建时不会携带任何详细错误信息.
- **继承链**：
  - `IOException` 继承自 `Exception`，`Exception` 又继承自 `Throwable`.
  - `Throwable` 类的无参构造函数定义了它的行为，即创建一个没有详细信息的异常对象.

`Throwable` 类的无参构造函数

为了更好地理解无参构造函数的完整定义，这里是 `Throwable` 类中的无参构造函数：

```
public class Throwable implements Serializable {
    /**
     * Constructs a new throwable with {@code null} as its
     * detail message. The cause is not initialized, and may
     * subsequently be initialized by a call to {@link #initCause}.
     */
    public Throwable() {
        fillInStackTrace();
    }

    // Other code...
}
```

`fillInStackTrace()` 是什么?

> `fillInStackTrace()` 是 `Java` 中的一个方法，定义在 `java.lang.Throwable` 类中. 它的主要作用是记录当前的堆栈信息，通常用于异常对象的创建和调试. 当异常被创建时，`fillInStackTrace()` 会捕获程序执行的当前调用栈（即方法调用链）并将其存储在异常对象中. 这样，开发者可以通过 `getStackTrace()` 方法在异常处理时检索和查看调用栈信息.

`fillInStackTrace()` 方法定义

以下是 `Throwable` 类中 `fillInStackTrace()` 的定义：

```
public synchronized Throwable fillInStackTrace() {
    // Native method to fill in the stack trace.
    return this;
}
```

主要功能

> `fillInStackTrace()` 方法使用本地代码（native code）来捕获异常发生时的调用栈. 这个调用栈包含了异常发生时程序中所有函数调用的顺序，帮助开发者了解异常是在哪个位置被抛出的，以及它是如何通过调用链传播的.

作用

> - **记录调用栈**：当异常被创建时，`fillInStackTrace()` 会记录当前的调用栈信息. 这对于调试非常有帮助，开发者可以通过查看异常的堆栈跟踪来确定错误的来源.
> - **返回异常对象**：`fillInStackTrace()` 返回调用它的 `Throwable` 对象本身，允许链式调用.

如何使用

通常情况下，`fillInStackTrace()` 是由 Java 异常机制自动调用的，开发者不需要显式调用它. 每当一个异常对象被创建时，`Throwable` 类的构造函数会调用 `fillInStackTrace()` 来记录堆栈信息. 例如：

```
try {
    throw new IOException("An I/O error occurred");
} catch (IOException e) {
    e.printStackTrace();  // 打印栈跟踪信息，显示异常发生的位置
}
```

在上面的例子中，`throw new IOException()` 会自动调用 `fillInStackTrace()` 来记录 `IOException` 被抛出的调用堆栈. `e.printStackTrace()` 会打印这个堆栈信息，帮助你确定异常发生的具体位置.

返回值

> `fillInStackTrace()` 返回当前的异常对象 (i.e., `this`)，这意味着你可以把它和其他方法链式调用.

`native method` 是什么意思?

> `fillInStackTrace()` 是一个 **native method**，也就是说它是用平台相关的代码（通常是 C 或 C++）实现的，而不是用 Java 编写的. 原因是堆栈跟踪信息涉及到底层系统调用，Java 需要与操作系统和虚拟机交互，来获取执行线程的调用栈. 在标准 Java API 中，某些涉及到低级系统操作的功能（如堆栈跟踪、内存管理等）通常由本地代码实现，以确保效率和与底层平台的兼容性.

相关方法

- `getStackTrace()`：返回一个 `StackTraceElement[]` 数组，表示异常发生时的堆栈跟踪信息. 每个 `StackTraceElement` 包含类名、方法名、文件名和异常抛出的位置（行号）.
- `printStackTrace()`：将异常的堆栈跟踪信息输出到标准错误流，通常用于调试.

```
e.printStackTrace();  // 打印堆栈跟踪信息
```

什么时候用到 `fillInStackTrace()`？

尽管通常不需要直接调用 `fillInStackTrace()`，但在某些特殊情况下，它可以用于重新初始化异常对象的堆栈跟踪信息. 例如，如果你在捕获一个异常后，想要重新抛出它并更新堆栈跟踪信息，可以调用 `fillInStackTrace()` 来重置堆栈跟踪.

```
try {
    throw new IOException("Initial exception");
} catch (IOException e) {
    e.fillInStackTrace();  // 重置堆栈信息
    throw e;               // 重新抛出异常
}
```

```
import java.io.*;

public class Trigonometry {

    public double tan(double angle) throws IOException {
        double answer = 0.0;

        File file = new File("nonexistent_file.txt");
        if (!file.exists()) {
            throw new IOException("File not found: " + file.getName());
        }
```

```
        return answer;
    }

    public static void main(String[] args) {
        Trigonometry trig = new Trigonometry();

        try {
            double value = trig.tan(30);
            System.out.println("tan(30) = " + value);
        } catch (IOException e) {
            System.err.println("An error occurred: " + e.getMessage());
        }
    }
}
```

6.1.2 Detecting Exceptions

From the signature of `tan()`, we know that sending this message is risky:

```
public double tan(double angle) throws IOException { ... }
```

因此在 `main` 函数里我们采用 `try-catch` 块:

> `try` is responsible for detection
>
> `catch` is responsible for identification and handling

```
    public static void main(String[] args) {
        Trigonometry trig = new Trigonometry();

        try {
            double value = trig.tan(30);
            System.out.println("tan(30) = " + value); // This statement will not
be executed
        } catch (IOException e) {
            System.err.println("An error occurred: " + e.getMessage());
        }
    }
```

6.1.3 Try Block

In case of receiving any exceptions from any statements in the try block, **the rest (subsequent) statements in the try block will be skipped.**

> `try` 块只执行抛出异常的语句以及前面的语句，抛出异常后直接跳到 `catch` 块执行.

6.1.4 Catch Block

> In fact, `try` and `catch` are twins. `Try` is responsible for detection. `Catch` is responsible for identification and handling.

`catch` 块的语法：

```
catch (ExceptionType an_object_reference) {
    statements to remedy the condition;
}
```

> 这里的 `an_object_reference` 一般用 `e` ，最简便

`catch` 可以并列使用：

```
import java.io.*;
import java.util.Scanner;

public class Trigonometry {

    // 自定义异常类，用于非法角度
    static class InvalidAngleException extends Exception {
        public InvalidAngleException(String message) {
            super(message);
        }
    }

    // 模拟从文件中读取并计算 tan(angle) 的方法
    public double tan(double angle) throws IOException, InvalidAngleException {
        double answer = 0.0;

        // 模拟数学错误，比如除以 0 的情况
        if (angle == 90) {
            throw new ArithmeticException("Math error: tan(90) is undefined.");
        }

        if (angle == -90) {
            throw new ArithmeticException("Math error: tan(-90) is undefined.");
        }

        // 检查非法角度，比如负数
        if (angle < 0) {
            throw new InvalidAngleException("Invalid angle: Angle cannot be
negative.");
        }

        // 模拟文件不存在的情况
        File file = new File("nonexistent_file.txt");
```

```
        if (!file.exists()) {
            throw new IOException("File not found: " + file.getName());
        }

        // 假设正常计算 tan(angle)
        answer = Math.tan(Math.toRadians(angle));

        return answer;
    }

    public static void main(String[] args) {
        Trigonometry trig = new Trigonometry();
        Scanner scanner = new Scanner(System.in);  // 创建 Scanner 对象用于读取用户输
入

        try {
            // 让用户输入角度
            System.out.print("Please enter the angle in degrees: ");
            double angle = scanner.nextDouble();  // 读取用户输入的角度

            // 调用 tan 方法计算正切值
            double value = trig.tan(angle);
            System.out.println("tan(" + angle + ") = " + value);
        } catch (IOException e) {
            // 捕获并处理文件相关的异常
            System.err.println("File error: " + e.getMessage());
        } catch (ArithmeticException e) {
            // 捕获并处理数学错误
            System.err.println("Math error: " + e.getMessage());
        } catch (InvalidAngleException e) {
            // 捕获并处理自定义的非法角度异常
            System.err.println("Invalid angle error: " + e.getMessage());
        } catch (Exception e) {
            // 捕获所有其他未预料到的异常
            System.err.println("An unexpected error occurred: " +
e.getMessage());
        }

        System.out.println("Program continues...");
        scanner.close();  // 关闭 Scanner，防止资源泄露
    }
}
```

`try-catch` 设计的好处：

① 异常抛出后如果继续执行 `try` 块可能会出错，因此跳到 `catch`，起码保证程序正常终止.

② `try` 块中异常抛出前的语句都有正常执行，尽量减小异常导致的损失.

③ `catch` 块除了保证程序正常终止，也能对异常进行针对性的识别和处理，便于开发者调试代码.

> 不用背，自己乱写的

6.1.5 关于 `throw` 和 `try-catch`

① 当 `method` 内部抛出一个未被捕获的异常（如 `Exception`）时，**方法的执行会被立即中断**，异常会向上抛给调用该方法的代码.

② 若调用方法的代码没有处理该异常，则异常会继续向上抛，直到找到一个捕获该异常的地方.

③ 如果异常一路传播到 `main` 方法，且仍然没有被捕获，程序将异常终止，并输出堆栈跟踪信息.

Example 1：抛出异常后中断方法的执行

```
public class ExceptionExample {

    // 方法将抛出异常
    public static void methodThatThrowsException() throws Exception {
        System.out.println("Start of method");
        throw new Exception("Something went wrong!");  // 抛出异常，方法中断
        // 这一行永远不会被执行
        // System.out.println("This will never be printed.");
    }

    public static void main(String[] args) {
        try {
            methodThatThrowsException();  // 调用抛出异常的方法
            System.out.println("This will not be printed if an exception is
thrown.");
        } catch (Exception e) {
            // 捕获并处理异常
            System.out.println("Caught exception: " + e.getMessage());
        }

        System.out.println("Program continues after handling the exception.");
    }
}
```

运行结果：

```
Start of method
Caught exception: Something went wrong!
Program continues after handling the exception.
```

- **方法中断**：在 `methodThatThrowsException()` 方法中，一旦 `throw new Exception("Something went wrong!")` 被执行，方法立即中断，后面的 `System.out.println("This will never be printed.");` 永远不会执行.

- **异常传播**：异常被抛到 `main` 方法中的 `try` 块，并由 `catch` 块捕获. 捕获后，程序继续执行 `catch` 块之后的代码，即 `System.out.println("Program continues after handling the exception.");` . 或者传到 `main` 方法里仍然没有捕获，由 `main` 直接抛出整个程序，那么程序会异常终止并输出堆栈跟踪信息.

Example 2：异常传播

如果没有捕获异常，异常会一直向上抛，直到到达程序的最顶层，即 `main` 方法. 如果在 `main` 方法中也没有捕获，程序将异常终止.

```
public class ExceptionExample {

    public static void methodThatThrowsException() throws Exception {
        System.out.println("Start of method");
        throw new Exception("Something went wrong!");  // 抛出异常
    }

    public static void anotherMethod() throws Exception {
        methodThatThrowsException();  // 调用抛出异常的方法
        System.out.println("This will not be printed.");
    }

    public static void main(String[] args) throws Exception { // 注意：对于 vscode，如果内部代码可能抛出不被捕捉的 Exception，main 函数必须 throws Exception（继续向上抛）,才能终止程序并输出堆栈跟踪信息，否则会直接乱码（乱码和抛出异常或输出堆栈是不一样的）;对于 javac 编译器，不 throws Exception 它也无法正常编译，会返回错误报告，告诉你第几行没有抛出 Exception
        anotherMethod();  // 调用 anotherMethod，不捕获异常
        System.out.println("This will not be printed either.");
    }
}
```

运行结果：

```
Start of method
Exception in thread "main" java.lang.Exception: Something went wrong!
    at ExceptionExample.methodThatThrowsException(ExceptionExample.java:6)
    at ExceptionExample.anotherMethod(ExceptionExample.java:10)
    at ExceptionExample.main(ExceptionExample.java:14)
```

- **方法中断**：`methodThatThrowsException()` 抛出异常后，`anotherMethod()` 中的代码立即中断，后续的 `System.out.println("This will not be printed.");` 永远不会执行.
- **异常传播**：异常传播到 `main` 方法，因为 `main` 方法也没有捕获此异常，程序终止，并输出完整的堆栈跟踪信息.

6.1.6 受检异常 vs 非受检异常

> Appendix

回顾之前的并列 `catch` 块代码：

```
import java.io.*;
import java.util.Scanner;

public class Trigonometry {

    // 自定义异常类，用于非法角度
    static class InvalidAngleException extends Exception {
        public InvalidAngleException(String message) {
            super(message);
        }
    }

    // 模拟从文件中读取并计算 tan(angle) 的方法
    public double tan(double angle) throws IOException, InvalidAngleException {
        double answer = 0.0;

        // 模拟数学错误，比如除以 0 的情况
        if (angle == 90) {
            throw new ArithmeticException("Math error: tan(90) is undefined.");
        }

        if (angle == -90) {
            throw new ArithmeticException("Math error: tan(-90) is undefined.");
        }

        // 检查非法角度，比如负数
        if (angle < 0) {
            throw new InvalidAngleException("Invalid angle: Angle cannot be
negative.");
        }

        // 模拟文件不存在的情况
        File file = new File("nonexistent_file.txt");
        if (!file.exists()) {
            throw new IOException("File not found: " + file.getName());
        }

        // 假设正常计算 tan(angle)
        answer = Math.tan(Math.toRadians(angle));

        return answer;
    }

    public static void main(String[] args) {
        Trigonometry trig = new Trigonometry();
        Scanner scanner = new Scanner(System.in);  // 创建 Scanner 对象用于读取用户输
入

        try {
            // 让用户输入角度
```

```
            System.out.print("Please enter the angle in degrees: ");
            double angle = scanner.nextDouble();  // 读取用户输入的角度

            // 调用 tan 方法计算正切值
            double value = trig.tan(angle);
            System.out.println("tan(" + angle + ") = " + value);
        } catch (IOException e) {
            // 捕获并处理文件相关的异常
            System.err.println("File error: " + e.getMessage());
        } catch (ArithmeticException e) {
            // 捕获并处理数学错误
            System.err.println("Math error: " + e.getMessage());
        } catch (InvalidAngleException e) {
            // 捕获并处理自定义的非法角度异常
            System.err.println("Invalid angle error: " + e.getMessage());
        } catch (Exception e) {
            // 捕获所有其他未预料到的异常
            System.err.println("An unexpected error occurred: " +
e.getMessage());
        }

        System.out.println("Program continues...");
        scanner.close();  // 关闭 Scanner，防止资源泄露
    }
}
```

注意到 `tan` 方法的声明：

```
public double tan(double angle) throws IOException, InvalidAngleException {...}
```

抛出了 `IOException` 和 `InvalidAngleException` 两个可能出现的异常，但是对于同样可能出现的 `ArithmeticException` 没有抛出，为什么？这样不是会导致编译问题吗？

非也，这里要引入**受检异常**和**非受检异常**的概念.

在方法签名中，`throws` 关键字用于声明**受检异常**（**Checked Exceptions**），而 `ArithmeticException` 属于**非受检异常**（**Unchecked Exceptions**）.这两类异常的处理方式不同.

受检异常（Checked Exceptions）

> Any Exception subclass that are not subclasses of $RuntimeException$ are **checked exceptions** (aka checked by the compiler)
>
> 以上述代码为例：
>
> - `IOException` 和自定义的 `InvalidAngleException` 都是受检异常.
> - 受检异常在编译时会被检查，必须要么被捕获（用 `try-catch`），要么在方法签名中使用 `throws` 声明. (Checked exceptions, such as IOException, need to be $declared$ in a method or constructor's throws clause)

> - 在 `tan(double angle)` 方法中声明 `throws IOException, InvalidAngleException`，表示这个方法可能抛出这两种受检异常，调用者必须处理它们.

非受检异常（Unchecked Exceptions）

> - `ArithmeticException` 是一个非受检异常，它继承自 `RuntimeException` .
> - 非受检异常在运行时被检查，**编译器不会强制要求你捕获或声明它们**. 也就是说，如果方法中可能抛出非受检异常，你可以选择不声明 `throws`，因为这不会影响代码的编译. (Runtime exceptions, e.g. ArthmetricException, can be *optionally declared* in a method or constructor's throws clause)
> - 常见的非受检异常包括 `NullPointerException`、`ArithmeticException`、`ArrayIndexOutOfBoundsException` 等.

注意是可选择不声明，不是不允许声明，如果有声明自然是更好的.

为什么不需要 `throws ArithmeticException`?

因为 `ArithmeticException` 是非受检异常，它表示在运行时发生的错误，比如除以零等情况. 这类错误通常是**编程错误**，编译器不会强制要求你声明或处理它们.

6.2 Package

There are normally thousands of files accessible from the computer.

They are organized into directories.

- Layer-by-layer.
- Hierarchical.
- Tree-like structure.

打开电脑目录快捷键：win + E

Backslash under MS '\' as path separator

A `Java` program / application may consist of multiple classes.

As long as the classes are put into the same directory, they can be grouped and inter-used.

Can we put classes into different directories and let them inter-operate?

> i.e. use classes in other packages?

By putting related classes into a directory, they are said to be in a *package*.

To use a class in a package, we can fully qualify the reference:

```
java.util.Scanner s;
s = new java.util.Scanner(...);
double i = java.lang.Math.PI;
```

但是，完全限定 reference 很麻烦.

6.2.1 Import 声明

Another choice is to use the *import* declaration:

① Import all classes in a package (*):

```
import java.lang.*;
```

② Import a single class in a package:

```
import java.util.Scanner;
```

现在，代码会简化很多：

```
import java.lang.*;
import java.util.Scanner;

Scanner s;
s = new Scanner(...);
double i = Math.PI;
```

> 代码少的时候简化程度不明显，如果是大项目，`import` 带来的简化相当可观.

6.2.2 创建 Package

除了 `import` Java 自带的 Package，我们也可以自己编写需要的包.

例如，我们想创建以下目录结构：

```
src/
├── com/
|   └── example/
|       ├── Main.java
|       └── utils/
|           └── Helper.java
```

并且实现在 `Main.java` 中调用 `Helper.java` 的类方法，我们需要完成两个文件：

① `Helper.java`

```java
// 文件路径: src/com/example/utils/Helper.java
package com.example.utils;

public class Helper {
    public static void printMessage(String message) {
        System.out.println("Helper says: " + message);
    }
}
```

② `Main.java`

```java
// 文件路径: src/com/example/Main.java
package com.example;

// 导入com.example.utils包中的Helper类
import com.example.utils.Helper;

public class Main {
    public static void main(String[] args) {
        // 调用Helper类中的静态方法
        Helper.printMessage("Hello from the Main class!");
    }
}
```

运行代码:

① 将代码放入正确的目录结构下.

> 注意这一步需要我们在电脑上手动构建匹配的目录结构（文件夹结构），然后把 `.java` 文件放到正确位置，否则无法编译（找不到文件）.

② 在 `src` 目录运行以下命令编译和执行代码:

```
# 编译所有的 Java 文件
javac com/example/Main.java com/example/utils/Helper.java

# 运行主类
java com.example.Main
```

输出:

```
Helper says: Hello from the Main class!
```

注意: Only the Java Byte Code (`.class`) files of the package are recognized by other packages.

If we have only the Java Source Files of the package, what should we do?

> Compile the package into classes / JAR in advance

（不过貌似 `javac` 会帮你自动编译你需要用的 `.class` 文件，即使你忘记编译了）

6.2.3 关于文件结构

注意到前面的 `Main.java` 也用到了 `package` 声明：

```java
// 文件路径: src/com/example/Main.java
package com.example;

// 导入com.example.utils包中的Helper类
import com.example.utils.Helper;

public class Main {
    public static void main(String[] args) {
        // 调用Helper类中的静态方法
        Helper.printMessage("Hello from the Main class!");
    }
}
```

它会让编译器认识到这样的文件结构：

```
src/
├── com/
|   └── example/
|       ├── Main.java
|       └── utils/
|           └── Helper.java
```

`Main.java` 能不能把 `package` 去掉？像这样：

```java
// 导入com.example.utils包中的Helper类
import com.example.utils.Helper;

public class Main {
    public static void main(String[] args) {
        // 调用Helper类中的静态方法
        Helper.printMessage("Hello from the Main class!");
    }
}
```

是可以的，但是结构必须这么放：

```
src/
├── com/
|   └── example/
|       └── utils/
|           └── Helper.java
└── Main.java
```

还有一个 tricky 的地方. `Main.java` 这么写:

```java
package src;

// 导入com.example.utils包中的Helper类
import com.example.utils.Helper;

public class Main {
    public static void main(String[] args) {
        // 调用Helper类中的静态方法
        Helper.printMessage("Hello from the Main class!");
    }
}
```

是可以的，但是结构必须这么放:

```
src/
├── com/
|   └── example/
|       └── utils/
|           └── Helper.java
└── src/
    └── Main.java
```

注意 `vscode` 可能对 `src` 命名有一些冲突从而导致编译错误，对于这种 tricky 的情况最好不去踩雷，非要用的话就用 cmd 手动编译.

总之，package 和 import 都是从根目录下的第一层子目录开始声明，根目录本身不要写进去（会自动识别）. package 的作用在于提高 import 的适用范围，如果不 package 的话，这个 `.java` 文件最多只能 import 同层文件及其子文件， 不能往上级索引.

当使用 package 之后，import 的范围拓展到根目录下的所有文件，根目录是哪个由 package 的声明决定（package 的第一层是根目录下的第一层子目录）.

注意：如果类 A 要 import 类 B，A 和 B 的 package 必须有相同的目录结构.

which means, the following is *wrong*:

目录结构:

```
src/
└── src1/
    ├── com/
    |   └── example/
    |       └── utils/
    |           └── Helper.java
    └── src2/
        └── Main.java
```

`Main.java`:

```java
package src1.src2;

// 导入 src1.com.example.utils 包中的 Helper 类
import src1.com.example.utils.Helper;

public class Main {
    public static void main(String[] args) {
        // 调用Helper类中的静态方法
        Helper.printMessage("Hello from the Main class!");
    }
}
```

`Helper.java`:

```java
// 文件路径: ./com/example/utils/Helper.java
package com.example.utils;

public class Helper {
    public static void printMessage(String message) {
        System.out.println("Helper says: " + message);
    }
}
```

错误报告:

```
src1\src2\Main.java:4: 错误: 无法访问Helper
import src1.com.example.utils.Helper;
                             ^
  错误的源文件: .\src1\com\example\utils\Helper.java
    文件不包含类src1.com.example.utils.Helper
    请删除该文件或确保该文件位于正确的源路径子目录中.
src1\src2\Main.java:9: 错误: 找不到符号
        Helper.printMessage("Hello from the Main class!");
        ^
  符号:   变量 Helper
  位置: 类 Main
2 个错误
```

错误在于 `Helper.java` 的 package 声明和 `Main.java` 不一致. 在编译 `Main.java` 时，编译器能顺着 import 声明往下寻找，找到 `.\src1\com\example\utils\Helper.java` 文件. 但是该文件中的 package 结构和 `Main.java` 想要 import 的结构不一致，因此报错:

```
  错误的源文件: .\src1\com\example\utils\Helper.java
```

如果我们提前编译好 `Helper.java`，在编译 `Main.java` 时，编译器也会报错：

```
src1\src2\Main.java:4: 错误: 无法访问Helper
import src1.com.example.utils.Helper;
                            ^
  错误的类文件: .\src1\com\example\utils\Helper.class
    类文件包含错误的类: com.example.utils.Helper
    请删除该文件或确保该文件位于正确的类路径子目录中.
src1\src2\Main.java:9: 错误: 找不到符号
        Helper.printMessage("Hello from the Main class!");
        ^
  符号:   变量 Helper
  位置: 类 Main
2 个错误
```

简单来说，就是编译器虽然检索到了正确的 `.class` 文件位置，但是它认为 `Helper.class` 文件中的类是错的（注意，类只是类文件的一部分）

会导致编译器看到这样的结构：

```
src/
├── src1/
|   ├── com/
|   |   └── example/
|   |       └── utils/
|   |           ├── Helper.java
|   |           └── Helper.class
|   └── src2/
|       └── Main.java
└── com/
    └── example/
        └── utils/
            └── Helper.class 中的 Helper 类（这只是便于可视化理解  ，类实际上不存在于文件夹中，它是某个类文件中的一部分）
```

换句话说，仅类文件的位置正确不够，它的 package 方式也要和 import 它的其他文件一致，不然可能导致本来属于它（`Helper.class`）的一部分的类（例如 Helper 类）跑到了它外面，这是编译器无法忍受的.

改正：把 `Helper.java` 改成：

```java
// 文件路径: ./src1/com/example/utils/Helper.java
package src1.com.example.utils; // 这里前面加上了 src1.，保持目录结构一致(根目录一致).

public class Helper {
    public static void printMessage(String message) {
        System.out.println("Helper says: " + message);
    }
}
```

6.2.4 Advanced Package Sharing

对于不同项目可能重复使用的包，每个项目都复制进去是很麻烦的，因此 java 提供了一些**高级包共享**的方法：

① 将自定义包放入系统目录

> 可以把编写的 Java 包（如 `.class` 文件或 `.jar` 文件）放入某个**系统目录**，例如 `C:\Program Files\Java\`. 这样做的目的是把这些包放在一个公共位置，以便所有项目都能访问它们.

② 设置环境变量 `CLASSPATH`

为了让 Java 能够找到你放在系统目录中的包，需要设置环境变量 `CLASSPATH`. `CLASSPATH` 是 JVM 用来查找类和包的路径.

在 Windows 系统中，通常可以通过修改 `C:\Autoexec.bat` 文件来设置 `CLASSPATH`，或者直接在系统的环境变量中设置.

③ 不要忽略默认的 JDK 类路径

在设置 `CLASSPATH` 时，需要确保**默认的 JDK 类路径**（即 Java 标准库路径）和当前目录仍然包含在内. 否则，Java 会找不到标准库中的类（如 `java.lang.*`）或你当前正在开发的项目中的类.

④ 共享包而不需要多次复制

通过这种方法，你可以在不同的项目之间共享同一个包，而不需要为每个项目单独复制这些包. 共享包位于一个公共目录中，所有项目通过设置 `CLASSPATH` 或项目配置来引用该目录.

More details about ② **设置环境变量 `CLASSPATH`**

设置 `CLASSPATH` 是为了告诉 Java 虚拟机（JVM）和开发工具去哪里查找类文件（`.class` 文件）或 JAR 文件. 正确设置 `CLASSPATH` 可以确保你的 Java 程序在运行时能够找到所需的类和库.

Windows 系统中设置 `CLASSPATH`

有两种常见方法设置 `CLASSPATH`：通过命令行临时设置或通过系统环境变量永久设置.

方法 1：临时设置 `CLASSPATH`（命令行方式）

你可以在**命令行窗口**中临时设置 `CLASSPATH`. 这种设置只在当前命令行会话中有效，关闭命令行后就会失效.

```
set CLASSPATH=C:\path\to\your\classes;C:\path\to\your\jarfile.jar
```

- `C:\path\to\your\classes` 是包含 `.class` 文件的目录.
- `C:\path\to\your\jarfile.jar` 是 JAR 文件的路径.
- 如果有多个路径，用分号（`;`）分隔.

例如：

```
set CLASSPATH=C:\Program Files\Java\myClasses;C:\libs\mylibrary.jar
```

方法 2：永久设置 `CLASSPATH`（通过系统环境变量）

1. **右键点击**“此电脑”或“我的电脑”，选择**属性**.
2. 点击左侧的**高级系统设置**.
3. 在**系统属性**窗口中，点击**环境变量**.
4. 在**环境变量**窗口中，找到**系统变量**区域，点击**新建**，添加一个新的环境变量：
    - 名称：`CLASSPATH`
    - 值：输入你的类文件或 JAR 文件的路径. 例如：

      ```
      C:\Program Files\Java\myClasses;C:\libs\mylibrary.jar
      ```

5. 如果已经有 `CLASSPATH` 变量，点击**编辑**，然后在已有值后追加路径，使用分号分隔.
6. 点击**确定**保存更改.

Linux/macOS 系统中设置 `CLASSPATH`

方法 1：临时设置 `CLASSPATH`（命令行方式）

你可以在终端中临时设置 `CLASSPATH`，这只在当前终端会话中有效.

```
export CLASSPATH=/path/to/your/classes:/path/to/your/jarfile.jar
```

- `/path/to/your/classes` 是包含 `.class` 文件的目录.
- `/path/to/your/jarfile.jar` 是 JAR 文件的路径.
- 多个路径用冒号（`:`）分隔.

例如：

```
export CLASSPATH=/usr/local/java/myClasses:/usr/local/java/mylibrary.jar
```

方法 2：永久设置 `CLASSPATH`

你可以将 `CLASSPATH` 添加到 shell 的启动文件中，例如 `.bashrc`（或 `.bash_profile`）文件中，这样每次启动终端时都会自动加载.

1. 打开终端，编辑 `.bashrc` 或 `.bash_profile` 文件：

   ```
   nano ~/.bashrc
   ```

   或者：

   ```
   nano ~/.bash_profile
   ```

2. 在文件中添加一行：

   ```
   export CLASSPATH=/path/to/your/classes:/path/to/your/jarfile.jar
   ```

3. 保存并退出编辑器.
4. 使更改生效：

   ```
   source ~/.bashrc
   ```

   或者：

   ```
   source ~/.bash_profile
   ```

注意事项

- **包含当前目录**：为了确保 Java 能够找到当前项目中的类，建议将当前目录 `.` 包含在 `CLASSPATH` 中. 例如：

  ```
  set CLASSPATH=.;C:\path\to\your\classes;C:\path\to\your\jarfile.jar
  ```

  或在 Linux/macOS 中：

```
export CLASSPATH=.:/path/to/your/classes:/path/to/your/jarfile.jar
```

- **JDK 默认类路径**：如果你修改了 `CLASSPATH`，要确保仍然能够访问 JDK 自带的类库（如 `java.lang.*`），通常这会自动包含在 JVM 中，但如果你覆盖了 `CLASSPATH`，务必要包含默认的 JDK 类库.
- **命令行编译和运行**：
  - **编译**时，使用 `-cp` 参数指定 `CLASSPATH`：

    ```
    javac -cp .;C:\path\to\your\classes Main.java
    ```

  - **运行**时，同样使用 `-cp` 参数：

    ```
    java -cp .;C:\path\to\your\classes Main
    ```

总结

- **临时设置**：通过命令行/终端，使用 `set CLASSPATH`（Windows）或 `export CLASSPATH`（Linux/macOS）.
- **永久设置**：通过系统的环境变量（Windows）或编辑 `.bashrc`/`.bash_profile`（Linux/macOS）.
- **注意**：不要忘记把当前目录 `.` 包含在 `CLASSPATH` 中，以便找到当前项目中的类.

6.2.5 Java Variable Scope Rules

> 作用域

回顾一下之前学过的 `Data Item / Storage`：

Data items declared in a $class\ \{\}$: we call them $fields$.

- With modifier $static$: we call them $class\ fields$ (single copy)
- Without modifier static: we call them $instance\ fields$ (one copy for each object)

- With modifier $final$: we call them $constant\ [fields]$ (initialized once)
- Without modifier final: we call them $variable\ [fields]$ (can be modified)

- With modifier $public$: we call them $public\ [fields]$ (accessible outside class)
- With modifier $private$: we call them $private\ [fields]$ (accessible within class)
- With modifier $protected$: we call them $protected\ [fields]$ (inherited privacy)

- If they $store\ data\ directly$: the type should be $primitive\ types$ (e.g. int, double, char)
- If they $store\ references$: the type should be $SomeClass$ (object ref) or $array[]$ (array ref)

Data items declared in a $method()\ \{\}$: we call them $local\ variables$.

- With modifier $final$: we call them $local\ constants$.
- Usually no other modifier for local variables.

- If they $store\ data\ directly$: the type should be $primitive\ types$. (e.g. int, double, char)
- If they $store\ references$: the type should be $SomeClass$ (object ref) or $array[]$ (array ref)

The $scope$ of an identifier is defined to be the part of program in which the identifier is known or accessible.

Scope rules depend on the notion of $blocks$.

A block is a compound statement, which can contain $declarations\ of\ identifiers$ of its own.

The basic scoping rule:

- identifiers are accessible within the block in which they are declared.

注意：类成员变量（全局变量）和方法的局部变量可以同名，但是同一个方法内不能在外部块和内部块中声明同名的变量.

Within a method:

If the variable a is declared in the outer block, we cannot declare another variable with the same name a in the inner block.

> i.e. No nesting.

也就是说，这样是可以的：

```
public class Example {
    // 类的成员变量（全局变量）
    int a = 10;
```

```
    public void someMethod() {
        // 方法的局部变量，名字和成员变量相同
        int a = 20;

        // 打印局部变量 a
        System.out.println("局部变量 a: " + a);  // 输出 20

        // 访问类的成员变量 a，使用 this 关键字
        System.out.println("成员变量 a: " + this.a);  // 输出 10
    }

    public static void main(String[] args) {
        Example obj = new Example();
        obj.someMethod();
    }
}
```

输出：

```
局部变量 a: 20
成员变量 a: 10
```

这样是不行的：

```
public class Example {
    public void anotherMethod() {
        int a = 5;  // 在外部块中声明变量 a

        {
            int a = 10;  // 错误：不能在内部块中声明与外部块相同名称的变量
            System.out.println(a);
        }

        System.out.println(a);
    }

    public static void main(String[] args) {
        Example obj = new Example();
        obj.anotherMethod();
    }
}
```

编译时报错：

```
Example.java:6: 错误: 已在方法 anotherMethod()中定义了变量 a
            int a = 10;  // 错误：不能在内部块中声明与外部块相同名称的变量
                ^
1 个错误
```

正确做法：同一个方法内，内部块和外部块使用不同的变量名.

Lecture 7

More on Loop, Switch, char

7.1 for loop 特性

语法

```
for(start; check; update)
    body_statement;
```

① One or more or even all parts may be missing.

```
for( ; ; )
    ; // 这是合法的
```

> The missing parts are called **empty** statements.

② **check** is normally a **boolean** expression (ending condition).

③ If **check** is empty, it is considered always true.

④ continue 直接进入 update 环节

7.2 while loop 特性

语法

```
while(condition)
    statement;
```

① condition is a **boolean** expression.

② 进入循环前先 check condition，每次循环结束也要 check condition

7.3 do ... while

语法

```
do{
    statement;
} while (condition);
```

① condition is a **boolean** expression.

② 进入循环前先 do，然后 check do, check do

> 无论如何都至少执行一次

③ continue 的话，直接进入 check 环节

7.4 switch

示例

```
public class switch_test{
    public static void main(String [] args){
        char grade = 'C';
        double GPA;
        switch (grade){
            case 'D': GPA = 1.0;
                break; // break is optional
            case 'C': GPA = 2.0;
                break;
            case 'B': GPA = 3.0;
                break;
            case 'A': GPA = 4.0;
                break;
            default: GPA = 0.0; // optional

        }
        System.out.println("GPA = " + GPA);
    }
}
```

输出：

```
GPA = 2.0
```

注意：

① case labels **MUST NOT** be ranges.

> 如果要用范围判断，只能 `if-else`.

② 都不匹配就跳到 `default`. 没有 `default` 就直接结束.

③ fall-through 贯穿特性：从匹配到的 case 开始，往下执行全部代码（包括其他 case 和 default），直到遇到 break 或者方法结束.

> 如果想要避免贯穿行为，就要手动在每个 case 末尾添加 break

④ default 可以写在任何位置. 它执行有两种情况：其他代码贯穿到它这，或者所有 case 都没有匹配，它作为开头开始往下贯穿.

> 注意，如果下面的 case 能匹配到，它不会主动开始贯穿.

```
public class switch_test{
    public static void main(String [] args){
        int day = 3;
        switch (day) {
            case 3:  System.out.println("Wednesday");
            default: System.out.println("Invalid day");
            case 4:  System.out.println("Thursday");
        }
    }
}
```

输出：

```
Wednesday
Invalid day
Thursday
```

```
public class switch_test{
    public static void main(String [] args){
        int day = 4;
        switch (day) {
            case 3:  System.out.println("Wednesday");
            default: System.out.println("Invalid day");
            case 4:  System.out.println("Thursday");
        }
    }
}
```

输出：

```
Thursday
```

```
public class switch_test{
    public static void main(String [] args){
        int day = 5;
        switch (day) {
            case 3:  System.out.println("Wednesday");
            default: System.out.println("Invalid day");
            case 4:  System.out.println("Thursday");
        }
    }
}
```

输出：

```
Invalid day
Thursday
```

利用贯穿特性优化代码：

```
int star_count = 4;
switch (star_count) {
    case 1: System.out.println(“*”); break;
    case 2: System.out.println(“**”); break;
    case 3: System.out.println(“***”); break;
    case 4: System.out.println(“****”); break;
    case 5: System.out.println(“*****”); break;
}
```

利用贯穿特性使代码更简洁（但是可读性稍稍变弱）：

```
int star_count = 4;
switch (star_count) {
    case 5: System.out.print(“*”);
    case 4: System.out.print(“*”);
    case 3: System.out.print(“*”);
    case 2: System.out.print(“*”);
    case 1: System.out.println(“*”);
}
```

7.5 char

> java 内置的 primitive type
>
> 单引号

Example:

- A, B, C, D, E, ..., X, Y, Z
- a, b, c, d, e, ..., x, y, z
- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- !, @, #, $, %, ^, ...

Special (non-printable) ones:

- backspace `\b`, new line `\n`, tab `\t`
- most of these are actions.

Each character bears an unique ID

- Previously known as ASCII.
- It is extended and now called **Unicode** in Java.

隐式转换

```
public class char_test{
    public static void main(String [] args){
        char c = 'A';
        int n;
        n = c; // auto-conversion
        System.out.println(n);
    }
}
```

输出：

```
65
```

显式转换

```
public class char_test{
    public static void main(String [] args){
        int n = 65;
        char c;
        c = (char) n; // explicit-conversion
        System.out.println(c);
    }
}
```

输出：

```
A
```

```
public class char_test{
    public static void main(String [] args){
        char c = 'A';
        c++;
        System.out.println(c);
        int difference = 'Z' - 'D';
        System.out.println("'Z' - 'D' --> " + difference);
        difference = 'H' - 'h';
        System.out.println("'H' - 'h' --> " + difference);
        difference = ';' - 'x';
        System.out.println("';' - 'x' --> " + difference);
    }
}
```

输出：

```
B
'Z' - 'D' --> 22
'H' - 'h' --> -32
';' - 'x' --> -61
```

Hexadecimal (Base 16) Unicode

> 十六进制

'\u0041' means Unicode 65 -> 'A'

```
public class char_test{
    public static void main(String [] args){
        System.out.printf("abc\u0041def");
    }
}
```

输出：

```
abcAdef
```

> println 不支持占位符，但是支持转义字符（escape character）.

Lecture 8

Array and Table

和 C 类似，不允许不同类型数据存在同一个数组.

8.1 Array

声明和创建

> Declaration and Array Creation

示例

```
int[] i = new int[1000];
```

等价于

```
int[] i; // i is nothing yet
i = new int[1000];
```

or

```
int i[] = new int[1000];
```

在 Java 中，数组本质是一个类. 数组变量存的是数组对象的引用.

语法：

```
type[] array_name = new type[length];
type[] array_name = {value1, value2, ...};
e.g.
double[] GPA = new double[50];
String[] countryCode = new String[175];
String address[] = new String[30];
char[] vowels = {‘a’, ‘e’, ‘i’, ‘o’, ‘u’};
```

> `type` may be primitive type, class type or even an other array type.
>
> 注意：声明时，[] 内是长度，而不是下标的最大值（下标最大比长度小 1）

Properties

① A bounded (fixed length) and indexed collection of elements of the same type.

② Array length is fixed at the same time of creation.

③ Element access is done using an index [].

> a[-8] -> ArrayIndexOutOfBoundsException

④ To get the length (size) of an array:

`a.length`

> 本质上是对象的一个实例字段

不要写成 `a.length()` 和字符串混淆

示例：`main` 函数括号中的参数 -- 字符串数组

```
class TestArgs {
    public static void main(String[] args) {
        System.out.println("There are " + args.length + " arguments:");
        int i;
        for (i = 0; i < args.length; i++)
            System.out.println( args[i] );
        }
}
```

可以对不同输入作出不同响应：

```
E:\材料\大学\大二 term 1\CSCI 1130 Java\Lecture\Lecture 8 Array and Table>java
TestArgs
There are 0 arguments:

E:\材料\大学\大二 term 1\CSCI 1130 Java\Lecture\Lecture 8 Array and Table>java
TestArgs Apple Banana
There are 2 arguments:
Apple
Banana
```

> 运行开始时输入的参数叫做 command-line arguments
>
> 命令行输入的参数存入 JVM 创建的数组中，JVM 再传给 main()

普通传参，修改形参不影响实参；数组传参，传的是引用（地址），形参复制后，修改形参也会影响到实参（因为是直接改内存中的内容）

8.2 Table

> 这里指的是二维数组

语法

```
dataType[][] arrayName = new dataType[rows][columns];
```

Lecture 9

String Comparison

String 类

The String class defines a convenient data type for storing and manipulating a sequence of characters. i.e. text

> 注意：Strings 是对象，不是初始数据类型.

创建 String 对象

语法

```
String myName;
myName new String();
```

省略 `new` 关键字

String 对象创建时，可以省略 new 关键字.

```
String myName;
myName = "wxx";
```

> 只适用于 class String
>
> 双引号创建一个新的 String 实例，内容为常量.
>
> 但是变量只是存储 String 对象的引用，引用是可以修改的. 可以把其他 String 的引用赋给 `myName` 变量，与 `"wxx"` 字符串本身为常量不冲突.

String Properties

① String objects can be concatenated (joined together) using a special operator `'+'`

② Content of a **String** object could be empty.

```
String myName;
myname = "";
String yourName;
```

> 这里 `myName` 和 `yourName` 是不同的. `myName` 有所指，指向了一个空对象 `String object ""`；`yourName` 无所指，存了 null object reference

String Methods

String 类是 private and inaccessible by us 的.

To manipulate a String object, we must use the provided methods.

① To get the length of a String object

```
String myName = "wxx"
System.out.println(myName.length());
```

> 由于 `myName` 封装，没有实例字段（对象本身的属性）允许读取，获取长度只能将它作为 message 传给 `.length()` 方法.

注意与 `array` 区分：

```
int[] even_numbers = {2, 4, 6, 8, 10};
System.out.println(even_numbers.length);
```

> 数组对象的实例字段是可以访问的（当然，`length` 不能修改）.

② The instance method `char charAt(int)` returns you a `char` at the i^{th} position of the String object.

示例：竖直打印字符串

```
String myName = "Michael"
for(int i=0; i<myName.length(); i++)
    System.out.println(myName.charAt(i));
```

③ Compare 2 String objects 的内容

> 不能直接比较变量，这样比的是引用，如果两个变量引用不同对象，但是对象内容相同，比较引用的结果仍然是 “不同”.

示例（这通常是不对的）：

```
Scanner keyboard = new Scanner(System.in);
String myName = “Michael”;
String yourName = keyboard.nextLine();
if (myName == yourName)
    System.out.println(“Uk...Oh...”);
else
    System.out.println(“What’s up?!”);
```

无论键盘输入什么，输出永远是 `What's up?!`，因为引用了不同对象.

正确示例：使用 `.equals()` 方法

```
Scanner keyboard = new Scanner(System.in);
String myName = “Michael”;
String yourName = keyboard.nextLine();
if (myName.equals(yourName))
    System.out.println(“Uk...Oh...”);
else
    System.out.println(“What’s up?!”);
```

Lecture 10

Inheritance, Polymorphism

继承与多态

子类

> Sub-classing

Sometimes, we want to create a new class which supplements an existing class while keeping the original class

> 例如 add new fields, add new methods, modify original components

In Java, we do it using the `extends` mechanism.

```
class Original {
    // class field
    // instance field
    // constructor method
    // class method
    // instance method
}

class Subclass extends Original {
    /*
     * ALL the non-private original
     * fields and methods are
     * inherited in this new Subclass
     * UNLESS we modify them.
     */
     // We may optionally:
     // add class field
     // add instance field
     // add constructor method
     // add class method
     // add instance method
```

```
}
```

继承

> Inheritance

`extends` 关键字实现了继承. Several subclasses can inherit from the same `superclass` （父类）.

A subclass can `override` （覆写） inherited components (fields and methods)

A subclass can add new components as well.

`public/protected static/instance fields/methods` 会被继承.

`private static/instance fields/methods` 不会被继承.

> 虽然不会被继承，但可以在子类中重新声明.

Privacy

> 注意这个 privacy 和 能否继承不是一个概念.

Member(field/method) modifiers revisited:

- `public`：a member is accessible anywhere by any class
- `private`：a member is ONLY accessible by that class
- `protected`：a member is ONLY accessible by that class AND the subclasses of that class.

> protected member may be access from outside the package in which it is declared ONLY by code that is responsible for the implementation of the object (这里指的就是子类). 注意，这里说的访问不是真的突破了权限，而是把信息继承到自己身上，然后访问自己的.
>
> If a super class and its subclass are in different packages, the subclass inherits protected members of the superclass and may access such inherited protected of its own. But the subclass cannot access non-public members in superclass instances.

- No modifier (default): a member is ONLY accessible by classes in the same package

Super Construction

`super();` 调用父类构造函数

> 注意，子类只要继承就一定会先调用父类的构造函数（保证父类部分正确初始化），如果第一行不写 `super(parameters);` ，就默认调用无参构造函数，但如果父类没有显式定义无参构造函数就会报错. 所以建议子类构造函数都要先显式调用父类构造函数.

抽象类

> 只能被继承，无法直接实例化对象.

语法

```
// 抽象类定义
abstract class Account {
    protected String accountNumber;

    // 具体方法
    public void deposit(double amount) {
        System.out.println("Deposit: " + amount);
    }

    // 抽象方法（需要子类实现）
    public abstract void withdraw(double amount);
}

// 子类实现
class SavingsAccount extends Account {

    @Override
    public void withdraw(double amount) {
        System.out.println("Withdraw from SavingsAccount: " + amount);
    }
}
```

> If a class has an abstract method, the class must be declared as abstract too.

realize an abstract method

> 子类继承，override and provide actual implementations.

`@Override` 标签是可选的，但最好写上. 它能告诉编译器"我正在重写父类或接口中的方法"，如果编译器发现方法签名不匹配（例如拼写错误，参数类型错误），编译器会报错（不写 `@Override` 则会新创建一个方法）. 此外，`@Override` 标签也能提高可读性.

值得注意的是，当类没有显式继承某个父类，也没有用 `implements` 实现接口时，它默认隐式继承了 `Object` 类（后面会讲），因此它仍然可以重写一些方法：

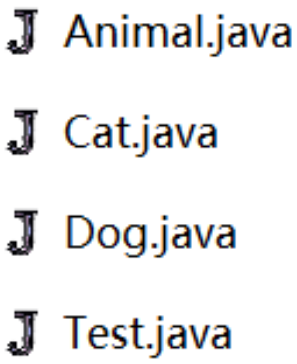
```java
class Person {
    String name;

    Person(String name) {
        this.name = name;
    }

    @Override
    public String toString() { // 重写 toString 方法
        return "Person{name='" + name + "'}";
    }
}
```

Example

文件结构：



`Animal.java`

```java
public abstract class Animal{
    public abstract void eat();
}
```

`Cat.java`

```
public class Cat extends Animal{
    public void eat(){
        System.out.println("Eat fish");
    }
    public void catchMouse(){
        System.out.println("Meow");
    }
}
```

`Dog.java`

```
public class Dog extends Animal{
    public void eat(){
        System.out.println("Eat bone");
    }
    public void guardHome(){
        System.out.println("Wolf");
    }
}
```

`Test.java`

```
public class Test {
    public static void main(String[] args){
        show(new Cat()); // call show by Cat object
        show(new Dog()); // call show by Dog object
    }
    public static void show(Animal a){
        a.eat();
        // judge type
        if (a instanceof Cat) { // do Cat's work
            Cat c = (Cat)a;
            c.catchMouse();
        } else if (a instanceof Dog){ // do Dog's work
            Dog d = (Dog)a;
            d.guardHome();
        }
    }
}
```

> 注意：虽然 `main` 中 `show()` 传入的参数是子类 `Cat` 或 `Dog` 的对象，但在 `show(Animal a)` 中，方法接收的参数类型是父类 `Animal`. 由于在编译时，Java 静态类型检查的依据是方法声明中的参数类型，而不是实际传入的对象类型，因此需要进行强制类型转换.

java 的多态性允许父类引用指向子类对象：

```
Animal a = new Cat(); // 父类引用指向子类对象
```

在这种情况下，父类引用只能访问父类中定义的方法（如 `eat()`）。如果需要调用子类中特有的方法（如 `catchMouse()`），就必须先将引用强制转换为子类类型：

```
Cat c = (Cat) a; // 将父类引用强制转换为子类引用
c.catchMouse(); // 现在可以调用子类特有的方法
```

强制类型转换告诉编译器："我确定这个 `Animal` 类型的引用实际上是一个 `Cat`（或 `Dog`）对象，你可以放心地将它视为 `Cat`（或 `Dog`）."

运行时类型检查

当你执行强制类型转换时，Java 会在运行时检查对象的实际类型。如果对象的实际类型与强制转换的目标类型不匹配，会抛出 `ClassCastException`.

例如：

```
Animal a = new Dog();

Cat c = (Cat) a; // 运行时抛出 ClassCastException，因为 a 实际是 Dog 类型
```

为了避免这种情况，通常会在强制类型转换之前使用 `instanceof` 检查对象的类型：

```
if (a instanceof Cat) {
    Cat c = (Cat) a; // 安全转换
    c.catchMouse();
}
```

静态类型

变量在编译时的类型，即变量声明时的类型.

静态类型在编译阶段确定，并且在整个程序中不会改变.

Java 是一种静态类型语言，所有变量的类型必须在编译时确定.

```
public abstract class Animal {
    public abstract void eat();
}

public class Cat extends Animal {
    @Override
```

```
    public void eat() {
        System.out.println("Cat eats fish");
    }

    public void catchMouse() {
        System.out.println("Cat catches mouse");
    }
}

public class Test {
    public static void main(String[] args) {
        Animal a = new Cat(); // 静态类型是 Animal，动态类型是 Cat
        a.eat(); // 调用 Cat 的 eat() 方法，合法，因为 Animal 定义了 eat() 方法

        // a.catchMouse(); // 编译错误，静态类型 Animal 中没有定义 catchMouse()

        // 强制转换
        if (a instanceof Cat) {
            Cat c = (Cat) a; // 强制将静态类型转为 Cat
            c.catchMouse(); // 合法，静态类型变为 Cat，可以调用 catchMouse()
        }
    }
}
```

多态

> Polymorphism，指同一个接口或方法在不同的对象上具有不同表现形式的能力. 多态可以通过方法重载（编译时多态）和重写（运行时多态）实现.

父类：general

子类：special

> 子类的对象也被认为是父类的对象.

编译时多态

> 静态多态.
>
> 方法重载（Method overloading）是实现编译时多态的方式.
>
> 同一个类中可以定义多个方法，这些方法具有**相同名称**但参数列表不同（参数类型、参数数量）.
>
> 方法的具体调用在编译阶段就确定.
>
> 注意：与返回值类型无关. 仅修改返回值类型无法实现方法重载.

示例：方法重载

```
class Calculator {
    // 重载方法：两个参数相加
    public int add(int a, int b) {
        return a + b;
    }

    // 重载方法：三个参数相加
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // 重载方法：小数相加
    public double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(10, 20));        // 调用第一个 add 方法
        System.out.println(calc.add(10, 20, 30));  // 调用第二个 add 方法
        System.out.println(calc.add(1.5, 2.5));    // 调用第三个 add 方法
    }
}
```

运行时多态

> 方法重写（Method Overriding），是实现运行时多态的方式.
>
> 父类的引用可以指向子类的对象，通过引用调用的方法在运行时才能确定.
>
> 运行时多态的核心机制是动态方法分派（Dynamic Method Dispatch）.

示例：方法重写

```
class Animal {
    // 父类方法
    public void speak() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    // 子类重写父类方法
    @Override
    public void speak() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    // 子类重写父类方法
    @Override
    public void speak() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a;

        // 父类引用指向子类对象
        a = new Dog();
        a.speak(); // 输出：Dog barks

        a = new Cat();
        a.speak(); // 输出：Cat meows
    }
}
```

> 动态方法分派机制通过 `@Override` 实现.
>
> 子类必须与父类有继承关系.
>
> 子类方法必须具有与父类方法相同的方法名，参数列表，返回值类型.
>
> 子类的访问修饰符不能比父类方法更严格（例如，父类方法是public，子类方法不能是protected）
>
> 子类方法可以添加新的功能，但不能改变父类方法的签名和行为.

Java 中，父类引用可以指向子类对象. 当父类引用指向子类对象时，只能访问父类中定义的字段和方法，无法直接访问子类中新定义的字段或方法.

> 可以这么理解：老一辈的人穿越到后代的身体里，但他仍然只会做自己原本会做的事. 他并不了解科技的新发展.

> 注意，子类引用不能指向父类对象，因为父类对象可能没有子类中定义的字段或方法，因此子类引用无法安全地操作父类对象.

解决方案：类型转换（Type Casting）

```
Octopus anOctopus = myWatch;    // 父类引用指向子类对象
((OctopusWatch)anOctopus).showTime(); // 强制转换后调用子类方法
```

强制转换后，编译器知道 `anOctopus` 是一个 `OctopusWatch` 对象（子类对象），因此可以调用子类特有的方法.

> Backward type casting. 如果父类引用需要使用子类特有的方法或字段，可以通过向下转型将父类引用转换为子类引用. 但需要确保父类引用指向的是子类对象，否则会抛出运行时异常（例如，如果父类引用指向父类对象，而不是子类对象，向下转型会导致 `ClassCastException`）.

Example: `toString()` method

Class `Object` is the super-ancester of all classes. It defines this instance method:

```
public String toString() // return a string representing the object
```

> 所有类都是 `Object` 类的子类（直接或间接）. 一个类如果没有显式声明继承哪个类，默认继承自 `Object` .

`obj.toString()` 是多态的实例方法. 所有类都继承了这个方法，但是可以通过 override 实现不同功能.

`println()` 打印对象就是在打印对象的 `toString()` 返回.

> 加号串联能把数字和字符串串联，其实是编译器先把数字对象转为它 `toString()` 的返回值.

Interface

> 接口. 在 Java 中，`interface` 是一种特殊的抽象类型，用于定义类可以实现的行为规范. 接口是一个完全抽象的类型，仅包含方法的**声明**（方法签名）和常量（static final 字段）.
>
> 接口中的方法默认是 `public` 和 `abstract` 的.
>
> 接口不能包含普通的实例字段，也不能直接实例化.

定义接口

使用关键字 `interface` 定义接口.

```
public interface Animal {
    void eat();    // 抽象方法
    void sleep();  // 抽象方法
}
```

类实现接口

使用关键字 `implements` 实现一个接口.

```
public class Dog implements Animal {
    @Override
    public void eat() {
        System.out.println("Dog eats food");
    }

    @Override
    public void sleep() {
        System.out.println("Dog sleeps");
    }
}

// 使用接口
public class Main {
    public static void main(String[] args) {
        Animal dog = new Dog(); // 接口引用指向实现类对象
        dog.eat();  // 输出：Dog eats food
        dog.sleep(); // 输出：Dog sleeps
    }
}
```

> 注意，类必须提供接口中所有抽象方法的具体实现，否则该类需要声明为 `abstract`.

接口特性

接口中的成员

抽象方法

- 接口中的方法默认是 public abstract 的.
- 例如，`void eat();` 等价于 `public abstract void eat();`
- 在实现类中，必须实现所有抽象方法.

静态常量

- 接口的字段默认是 `public static final` 的，即所有字段都是常量.

例如：

```
public interface Constants {
    int MAX_VALUE = 100; // 等价于 public static final int MAX_VALUE = 100;
}
```

默认方法（Default Methods, Java 8 引入）

- 接口可以包含默认实现的方法，使用关键字 `default` .
- 实现类可以选择重写这些方法.

例如：

```
public interface Animal {
    void eat();

    default void sleep() {
        System.out.println("Animal sleeps");
    }
}

public class Dog implements Animal {
    @Override
    public void eat() {
        System.out.println("Dog eats food");
    }

    // 可以选择重写默认方法
    @Override
    public void sleep() {
        System.out.println("Dog sleeps soundly");
    }
}
```

静态方法（Static Methods, Java 8 引入）

- 接口可以定义静态方法，静态方法不能被实现类重写.

例如：

```
public interface Animal {
    static void info() {
        System.out.println("This is an Animal interface");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal.info(); // 调用静态方法
    }
}
```

私有方法（Private Methods, Java 9 引入）

- 接口可以定义私有方法，用于辅助默认方法和静态方法的实现.
- 私有方法不能被实现类访问.

例如：

```
public interface Animal {
    default void action() {
        logAction(); // 调用私有方法
    }

    private void logAction() {
        System.out.println("Logging action...");
    }
}
```

接口的继承

接口可以继承其他接口，并可以多重继承：

```
public interface Animal {
    void eat();
}

public interface Pet extends Animal {
    void play();
}

public class Dog implements Pet {
    @Override
    public void eat() {
        System.out.println("Dog eats food");
    }

    @Override
    public void play() {
        System.out.println("Dog plays fetch");
    }
}
```

接口与抽象类的区别

> 待补充.

接口的多继承与实现

接口支持多继承，一个接口可以继承多个接口：

```java
public interface Flyable {
    void fly();
}

public interface Swimable {
    void swim();
}

public interface Bird extends Flyable, Swimable {
}
```

多实现：

一个类可以实现多个接口

```java
public class Duck implements Flyable, Swimable {
    @Override
    public void fly() {
        System.out.println("Duck flies");
    }

    @Override
    public void swim() {
        System.out.println("Duck swims");
    }
}
```

> 注意，Java 不支持类似 `C++` 的类的多继承，接口很好地弥补了这一点.

接口的优点

接口解耦：接口定义行为规范，具体实现由不同的类完成，减少类之间的耦合.

多实现：Java 不支持类的多继承，但支持类实现多个接口，从而实现多继承的效果.

代码的灵活性和扩展性：通过接口可以实现多态，增强代码的灵活性.

实际应用

```java
public interface Payment {
    void pay(double amount);
}

public class CreditCardPayment implements Payment {
    @Override
    public void pay(double amount) {
        System.out.println("Paid " + amount + " using Credit Card");
```

```
    }
}

public class PayPalPayment implements Payment {
    @Override
    public void pay(double amount) {
        System.out.println("Paid " + amount + " using PayPal");
    }
}

public class Main {
    public static void main(String[] args) {
        Payment payment = new CreditCardPayment();
        payment.pay(100.0);

        payment = new PayPalPayment();
        payment.pay(200.0);
    }
}
```

输出：

```
Paid 100.0 using Credit Card
Paid 200.0 using PayPal
```

The `clone()` Method

```
class Octopus {
    double value;
    String name;
    public Octopus(double value, String givenName) {
        this.value = value; // this usage
```

```
        name = givenName;
    }
    public Object clone() {
        Octopus newCard = new Octopus(0, name);
        newCard.value = this.value; // this usage
        return newCard;
    }
    public static void main(String [] args) {
        Octopus michael = new Octopus(100, "Michael Fung");
        Octopus copy;
        copy = (Octopus) michael.clone();
    }
}
```

`equals()`

```
public boolean equals(Octopus target)
{
    return this.value == target.value;
}
System.out.println("They have equal value: " +
    michael.equals(copy));
```

Lecture 11

Recursion

> 递归，divide and conquer：分而治之

Example: Summation

```
import java.util.*;

class Summation {
    public static int sum ( int n ) {
        if (n <= 1)
            return n;
```

```
        else
            return n + sum(n-1) ;
    }
    public static void main( String[] args) {
        System.out.print("Enter a number? ");
        int number = new Scanner(System.in).nextInt();
        System.out.printf("Sum(%d) = %d\n", number, sum(number));
    }
}
```

输出：

```
Enter a number? 100
Sum(100) = 5050
```

Example: Factorial

```
import java.util.*;

class Factorial {
    public static long factorial ( int n ) {
        if (n < 0) throw new ArithmeticException("-ve!");
        else if (n <= 1) return 1;
        else return n * factorial(n-1);
    }
    public static void main( String[] args) {
        System.out.print("Enter n? ");
        int n = new Scanner(System.in).nextInt();
        System.out.printf("%d! = %d\n", n, factorial(n));
    }
}
```

输出：

```
Enter n? 5
5! = 120
```

GUI

> Graphical User Interface
>
> 不考

A window is an object.

A button is an object.

We create windows and buttons from classes.

`java.awt`

> Abstract Windowing Toolkit

```
import java.awt.*;
import java.awt.event.*;

class SimpleGUI {
    public static void main(String[] args) {
        Frame myWindow = new Frame();
        myWindow.setTitle("Simple GUI");
        myWindow.setSize(200, 100);

        // 添加窗口关闭事件监听器
        myWindow.addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                System.exit(0); // 关闭程序
            }
        });

        myWindow.setVisible(true);
    }
}
```

`javax.swing`

`JOptionPane`

`JOptionPane` 是 **Java Swing** 提供的一个类，用于创建标准的对话框（Dialog）. 它可以用来显示信息、提示用户输入、确认操作等. `JOptionPane` 提供了一种简单的方式，可以快速创建各种类型的对话框，而不需要手动创建窗口和组件.

Tutorials

Tutorial 1

Tutorial 2

Programming Style

General naming conventions

1. Names representing classes must be nouns and written in mixed case starting with upper case. (Pascal case)

```
public class Hello
public class HelloWorld
```

2. Variable names must be in mixed case starting with lower case. (Camel case)

```
int iterationCounter
```

3. Names representing constants (**final** members) must be all uppercase using underscore to separate words.

```
MAX_ITERATIONS
```

4. Names representing methods must be verbs and written in mixed case starting with lower case. (Camel case)

```
getLength()
```

5. Abbreviations and acronyms should not be uppercase when used as name.

```
exportHtmlSource()
```

Comment

JavaDoc /** ----- */

Block comments /* ----- */

Single-line comments //

Tutorial 3

Primitive Data Type Boolean and Related Operators & MPF Exercise

Type Boolean

语法

```
boolean variableName = false;
variableOne = variableTwo <=0;
//正确返回true,错误返回false
```

Relational operators

`==` `!=` `>` `<` `>=` `<=` `=`

Floating point comparison

不能直接比较，时对时错.

正确方法:

```
Math.abs(f1-f2) < threshold
//threshold is very small, such as 0.0000001
```

Logical operators

Operator	Name	Example	Result
!	Logical NOT	! a	true if a is false false if a is ture
&&	Logical AND	a && b	true if both a and b are true
||	Logical OR	a || b	true if a or b, or both are true
^	XOR (异或)	a ^ b	true if the operand values are unequal

Short-Circuit Logical Operation

短路布尔求值

> a && b，若 a is false, 不用 check b
>
> a || b, 若 a is true, 不用 check b
>
> 优点：性能提升，减少错误

De Morgan's Law

德摩根律

!(a && b) = !a || !b

!(a || b) = !a && !b

Complex Boolean Expression

> 略

Precedence

()	Parentheses
< > ==	Relational / Comparisons
!	Negation
&	Bitwise AND
^	Bitwise XOR

()	Parentheses
|	Bitwise OR
&&	Logical AND
||	Logical OR

More About Expression

+=, ++, -- 在 java 中仍然可以使用.

Prime Number Checker

Get an integer input from user through console (use Scanner), then check that whether this input number is a prime number or not.

```
import java.util.Scanner;

class Prime {
    public static void main(String[] args) {
        boolean isPrime = true;
        Scanner scan = new Scanner(System.in);

        System.out.print("Enter a number: ");
        int num = scan.nextInt();
        int temp = 0;

        // 处理特殊情况
        if (num <= 1) {
            isPrime = false; // 1 或者更小的数字不是质数
        } else {
            int factorInTrial = (int) Math.sqrt(num);
            while (factorInTrial >= 2) {
                // divisibility test with different factors
                int remainder = num % factorInTrial;
                if (remainder == 0) {
                    System.out.println("Found factor of " + num + ": " +
factorInTrial);
                    isPrime = false; // 找到因子，num 不是质数
                    break;
                }
                factorInTrial--;
            }
        }

        System.out.println("Input number " + num + " is prime? " + isPrime);
        scan.close(); // 关闭扫描器
    }
}
```

Tutorial 4

Console Input & Branching & Assignment 2

Object creation

```
public class Point {
    public int x = 0;  // fields
    public int y = 0;

    // constructor, a special method for initializing a new object
    public Point(int a, int b) {
        x = a;  // copies the argument/ local variable a TO object field x
        y = b;  // copies the argument/ local variable b TO object field y
    }
}

// How to create an object of this class?
Point originOne = new Point(23, 94);  // a<-23, b<-94
```

> 第一个 Point 是 Class，第二个 Point 是 Method (Constructor)

java.util.Scanner Class

data type	method name	description
double	nextDouble()	Scans the next token of the input as a double.
float	nextFloat()	Scans the next token of the input as a float.
int	nextInt()	Scans the next token of the input as an int.
String	nextLine()	Advances this scanner past the current line and returns the input that was skipped.
long	nextLong()	Scans the next token of the input as a long.
short	nextShort()	Scans the next token of the input as a short.

Math function APIs

structure	description
static double pow(double a, double b)	Returns the value of the first argument raised to the power of the second argument, a is the base and b is the exponent.
static double sqrt(double a)	Returns the correctly rounded positive square root of a double value.
static double log(double a)	Returns the natural logarithm of a double value.
static double log10(double a)	Returns the base 10 logarithm of a double value.
static double log1p(double a)	Returns the natural logarithm of the sum of the argument and 1. 用于计算 $ln(1+x)$

Tutorial 5

Loop with if-else

Nested-Loop

Methods

Loop with if-else

sum of all odd/even numbers between 0 and 100 seperately.

```
public class SumOddEven {

    public static void main(String[] args) {
        sum_odd_even();
    }

    private static void sum_odd_even() {
        int sum_odd = 0;
```

```
        int sum_even = 0;

        for (int i = 1; i <= 100; i++) {
            if (i % 2 == 0) {
                sum_even += i; // 累加偶数
            } else {
                sum_odd += i; // 累加奇数
            }
        }

        System.out.println("Sum of all odd numbers: " + sum_odd);
        System.out.println("Sum of all even numbers: " + sum_even);
    }
}
```

Nested-Loop

1. Use Nested Loop to produce the following results:

....1

...2

..3

.4

5

2. Use Nested Loop to produce the following results:

....1

...22

..333

.4444

55555

```
public class NestedLoop {

    public static void main(String[] args) {
        nested_loop1();
        nested_loop2();
    }

    private static void nested_loop1() {
        for (int line = 1; line <= 5; line++) {
            for (int j = 1; j <= -line + 5; j++) {
                System.out.print("."); // 输出点
            }
            System.out.println(line); // 输出数字并换行
        }
    }
```

```
    private static void nested_loop2() {
        for (int line = 1; line <= 5; line++) {
            for(int j = 1; j <= (-1 * line + 5); j++) {
                System.out.print(".");
            }
            for(int k = 1; k <= line; k++) {
                System.out.print(line);
            }
            System.out.println();
        }
    }

}
```

注意到 We are using two similar inner loops to handle printing each lines，nested_loop2 中我们使用了相似的循环，但是写了两遍

Can we avoid repetitive works by using Method?

> 使用统一 Method

```
import java.util.Scanner;

public class NestedLoop {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.print("How many lines do you want: ");
        int totalLine = scan.nextInt();
        nested_loop2(totalLine);
    }

    private static void nested_loop2(int totalLine) {
        for (int line = 1; line <= totalLine; line++) {
            print_item(".", (-1 * line + totalLine));
            print_item(line+"", line);
            System.out.println();
        }
    }

    private static void print_item(String item, int num) {
        for (int i = 1; i<= num; i++)
            System.out.print(item);
    }

}
```

这里拓展了功能，可以自定义函数. print_item 办法实现了打印 num 次 item，nested_loop2 就不用写两个相似的循环了

注意传参，对参数进行操作，不会修改原地址的数值（不是传地址）

```
    private static void method3() {
        int number = 1;
        System.out.println("before the call, number = " + number);
        update(number);
        System.out.println("after the call, number = " + number);
    }

    static int update(int number) {
        number++;
        System.out.println("Inside the method, number = " + number);
        return number;
    }
```

输出如下：

```
before the call, number = 1
Inside the method, number = 2
after the call, number = 1
```

Tutorial 6

`JOptionPane`

Random Number

> Pseudo-random Number：伪随机数，可以复现

① `Math.random()`

> 注意：Math 类的方法无需显式导入. 直接使用即可.

语法

```
double randomValue = Math.random();
```

> 生成 $[0, 1)$ 的伪随机数.
>
> 左闭右开，概率均匀分布.

拓展

1. $[0, n)$

```
double random = Math.random() * n;
```

2. $[min, max)$

```
double random = Math.random() * (max - min) + min;
```

3. $[0, n)$ 整数

```
int randomInt = (int) (Math.random() * n); // 强制转换
```

4. $[min, max)$ 整数

```
int randomInt = (int) (Math.random() * (max - min) + min);
```

② class `Random`

> `Random` 类需要导入语句 `import java.util.Random;`
>
> 同样是伪随机.

`Random` 类的使用

1. 导入包

```
import java.util.Random;
```

2. 创建实例

```
Random random = new Random();  // 创建一个随机数生成器
```

> 如果不传入种子，默认构造函数使用当前时间戳作为种子.

也可以通过重载的构造函数传入种子（`long` 类型）.

```
Random random = new Random(12345L); // 使用种子 12345L
```

3. 调用实例方法

`Random` 类常用方法如下

方法	返回值类型	功能
`nextInt()`	`int`	返回一个随机的 `int` 值. 范围：所有可能 `int` 值，即 $[-2^{31}, 2^{31})$.
`nextInt(int bound)`	`int`	返回一个介于 `[0, bound)` 的随机整数（不包括 `bound`）.
`nextLong()`	`long`	返回一个随机的 `long` 值.
`nextFloat()`	`float`	返回一个介于 `[0.0, 1.0)` 的随机 `float` 值.
`nextDouble()`	`double`	返回一个介于 `[0.0, 1.0)` 的随机 `double` 值.
`nextBoolean()`	`boolean`	返回一个随机的布尔值（`true` 或 `false`）.
`nextBytes(byte[] bytes)`	`void`	填充一个字节数组，数组中的每个字节值都是随机生成的.
`setSeed(long seed)`	`void`	设置种子，重置随机数生成器的状态. 设置相同的种子将生成相同的随机数序列.
`nextGaussian()`	`double`	返回均值为 0.0，标准差为 1.0 的标准正态分布随机采样 `double` 值.

> `random.nextDouble()` 相当于 `Math.random()`.
>
> 上述方法中，`nextGaussian()` 是正态分布采样，其他为均匀分布采样.

Example

应用示例

```
import java.util.Random;

public class RandomExample {
    public static void main(String[] args) {
        // 创建一个 Random 对象
        Random random = new Random();

        // 生成一个随机整数
```

```
        int randomInt = random.nextInt();
        System.out.println("随机整数: " + randomInt);

        // 生成一个 [0, 100) 的随机整数
        int randomIntInRange = random.nextInt(100);
        System.out.println("随机整数 [0, 100): " + randomIntInRange);

        // 生成一个随机布尔值
        boolean randomBoolean = random.nextBoolean();
        System.out.println("随机布尔值: " + randomBoolean);

        // 生成一个随机浮点数 [0.0, 1.0)
        float randomFloat = random.nextFloat();
        System.out.println("随机浮点数 [0.0, 1.0): " + randomFloat);

        // 生成一个随机双精度数 [0.0, 1.0)
        double randomDouble = random.nextDouble();
        System.out.println("随机双精度数 [0.0, 1.0): " + randomDouble);

        // 生成一个随机长整数
        long randomLong = random.nextLong();
        System.out.println("随机长整数: " + randomLong);

        // 使用种子生成随机数（相同种子会生成相同的随机数序列）
        Random seededRandom = new Random(12345);
        System.out.println("相同种子生成的随机整数: " + seededRandom.nextInt());
    }
}
```

产生不重复整数

```
// 生成1-5随机排列（不重复）

import java.util.Random;

public class randomNumber{
    public static void main(String [] args){
        Random random = new Random();
        int count = 0;
        byte[] used = {0, 0, 0, 0, 0};
        while(count != 5){
            int tmp = random.nextInt(5) + 1;
            if(count == 0){
                System.out.print(tmp);
                used[tmp-1] = 1;
                count++;
            }
            else if(used[tmp-1] == 0){
                System.out.print(" " + tmp);
                used[tmp-1] = 1;
                count++;
            }
        }
```

```
    }
}
```

种子

> Random Seed

A **random seed** is a number used for initializing a $pseudo-random\ number\ generator$. The seed governs the behavior of the PRNG*. PRNG objects created with the $same$ random seed will produce $identical$ pseudo-random number sequences.

> 相同种子生成相同随机数.

```
import java.util.Random;
...
int seed = 11;
Random rngObj1 = new Random(seed);
Random rngObj2 = new Random(seed);
Random rngObj3 = new Random( );

System.out.printf("%12d\n", rngObj1.nextInt() );
System.out.printf("%.2f\n", rngObj1.nextFloat() );

System.out.printf("%12d\n", rngObj2.nextInt() );
System.out.printf("%.2f\n", rngObj2.nextFloat() );

System.out.printf("%12d\n", rngObj3.nextInt() );
System.out.printf("%.2f\n", rngObj3.nextFloat() );
```

输出:

```
-1158177819
0.71
-1158177819
0.71
1856327341
0.34
```

`Math.random()` 与 `Random` 类比较

特性	`Math.random()`	`Random` 类
范围	仅支持 [0, 1)	支持任意范围的随机数
类型支持	只能生成 `double` 类型	支持 `int`、`long`、`float` 等
种子控制	无法设置种子	可通过构造方法设置种子
灵活性	简单易用	更灵活，适合复杂需求

Tutorial 7

> File I/O & Assignment 4 Data Processor

`PrintStream` 类

① 输出到控制台

`System.out` 是一个默认的 `PrintStream` 实例：

```
public class Main {
    public static void main(String[] args) {
        // 打印字符串
        System.out.print("Hello, ");
        System.out.println("World!");

        // 打印整数和浮点数
        System.out.println(123);
        System.out.println(45.67);

        // 格式化输出
        System.out.printf("Name: %s, Age: %d, Salary: %.2f\n", "Alice", 30,
12345.678);
    }
}
```

控制台输出：

```
Hello, World!
123
45.67
Name: Alice, Age: 30, Salary: 12345.68
```

② 输出到文件

```java
import java.io.FileNotFoundException;
import java.io.PrintStream;

public class Main {
    public static void main(String[] args) {
        try {
            PrintStream ps = new PrintStream("output.txt");

            // 写入文件
            ps.println("Hello, File!");
            ps.println(123);
            ps.printf("Name: %s, Age: %d, Salary: %.2f\n", "Alice", 30,
12345.678);

            // 关闭流（隐式调用了 flush()）
            ps.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

文件内容：

```
Hello, File!
123
Name: Alice, Age: 30, Salary: 12345.68
```

③ 自动刷新功能

启用自动刷新时，每当打印换行符（`\n`）或显式调用 `flush()` 时，数据会自动刷新到目标设备.

`System.out` 默认自动刷新.

自定义的 `PrintStream` 实例默认不启用自动刷新.

但是想让数据因为忘记刷新缓冲区而没写入对象是很难的，因为 Java 有多重刷新保障. 当你没有设置

```java
PrintStream ps = new PrintStream(System.out, true);
```

时，关闭流 `ps.close();` 会隐式调用 `flush()`，即使忘记写关闭流，程序正常结束 `JVM` 关闭时，Java 也会尝试清空缓冲区并将数据写入对象.

> 本质是 `JVM` 垃圾回收机制. `PrintStream` 对象不再被引用时，垃圾回收器回收对象，在回收 `PrintStream` 对象时，其 `finalize()` 方法被调用，而 `finalize()` 方法会尝试关闭流，从而触发缓冲区刷新.

`Scanner` 类

① 从控制台读取数据

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in); // 从控制台读取输入
        System.out.print("请输入您的姓名：");
        String name = scanner.nextLine(); // 读取整行输入
        System.out.print("请输入您的年龄：");
        int age = scanner.nextInt(); // 读取整数
        System.out.print("请输入您的工资：");
        double salary = scanner.nextDouble(); // 读取双精度浮点数

        // 输出结果
        System.out.println("姓名：" + name);
        System.out.println("年龄：" + age);
        System.out.println("工资：" + salary);

        scanner.close(); // 关闭 Scanner
    }
}
```

控制台输出：

```
请输入您的姓名：Alice
请输入您的年龄：25
请输入您的工资：5000.50
姓名：Alice
年龄：25
工资：5000.5
```

② 从字符串中读取数据

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        String input = "Alice 25 5000.50";
        Scanner scanner = new Scanner(input); // 从字符串读取数据

        String name = scanner.next(); // 读取第一个单词
        int age = scanner.nextInt(); // 读取整数
        double salary = scanner.nextDouble(); // 读取双精度浮点数

        System.out.println("姓名: " + name);
        System.out.println("年龄: " + age);
        System.out.println("工资: " + salary);

        scanner.close();
    }
}
```

控制台输出:

```
姓名: Alice
年龄: 25
工资: 5000.5
```

③ 从文件中读取数据

> 注意 Scanner 构造函数传入的是文件对象，而不是文件名这个字符串.

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        try {
            File file = new File("data.txt");
            Scanner scanner = new Scanner(file); // 从文件读取数据

            while (scanner.hasNextLine()) { // 检查是否有下一行
                String line = scanner.nextLine(); // 读取一行
                System.out.println(line);
            }

            scanner.close();
        } catch (FileNotFoundException e) {
            System.out.println("文件未找到! ");
            e.printStackTrace();
        }
    }
```

```
}
```

`data.txt` 内容：

```
Alice 25 5000.50
Bob 30 6000.75
```

控制台输出：

```
Alice 25 5000.50
Bob 30 6000.75
```

④ 自定义分隔符

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        String input = "Alice,25,5000.50";
        Scanner scanner = new Scanner(input);
        scanner.useDelimiter(","); // 使用逗号作为分隔符

        String name = scanner.next(); // 读取第一个部分
        int age = scanner.nextInt(); // 读取第二个部分
        double salary = scanner.nextDouble(); // 读取第三个部分

        System.out.println("姓名：" + name);
        System.out.println("年龄：" + age);
        System.out.println("工资：" + salary);

        scanner.close();
    }
}
```

控制台输出：

```
姓名：Alice
年龄：25
工资：5000.5
```

⑤ 从 `URL` 中读取数据

```
import java.io.IOException;
import java.net.URL;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        try {
            // 指定目标 URL
            URL url = new URL("http://example.com"); // 替换为你想访问的 URL

            // 打开 URL 的输入流，并用 Scanner 包装
            Scanner scanner = new Scanner(url.openStream());

            // 逐行读取内容并输出到控制台
            while (scanner.hasNextLine()) {
                String line = scanner.nextLine();
                System.out.println(line);
            }

            // 关闭 Scanner
            scanner.close();
        } catch (IOException e) {
            System.out.println("发生错误：" + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

> 读的一般是 html 源代码

Exception Handling

> `try-catch` 块. Lecture 中有.

Overriding `toString()`

> 关于 `Object` 类可以查找 Lecture 对应内容.

```
public class Book {
    private String title;
    private String author;
    private int year;
    private int pages;
    public Book(String title, String author, int year, int pages) {
```

```
        this.title = title;
        this.author = author;
        this.year = year;
        this.pages = pages;
    }

    @Override
    public String toString() {
        return String.format(" \"%s\" by %s (%d) - %d pages", title, author,
year, pages);
    }


    public static void main(String[] args) {
        Book book1 = new Book("The Great Gatsby", "F. Scott Fitzgerald", 1925,
180);
    System.out.println(book1);
    }
}
```

控制台输出：

```
"The Great Gatsby" by F. Scott Fitzgerald (1925) - 180 pages
```

Tutorial 8

`String` 类

> Lecture 9 中也有介绍.

① 对象创建

```
String myName;
myName = new String();
```

② 省略 `new`

```
String myName;
myName = "*** Michael Fung! Hi :)";
```

③ 性质

String 在 java 中不是八大基本数据类型之一，而是类/对象.

可以加号串联.

内容可以为空. 空内容不等于空引用.

```
String myName;
myName = "";     // 空内容
String yourName; // 空引用
```

④ String Methods

String 的实例字段都是封装的，要查看 String 对象的各种属性都要使用对应的方法.

`.length()`

```
String myName = "Michael Fung";
System.out.println(myName.length()); // message
```

`.charAt(i)`

```
String myName = "Michael";
for (int i = 0; i < myName.length(); i++)
    System.out.println(myName.charAt(i));
```

输出:

```
M
i
c
h
a
e
l
```

`.equals(otherString)`

> 比较两个 String 对象的内容（不能直接 `==`，这比较的是引用）

⑤ String API functions

`.split()`

> 默认导入.

定义

```
public String[] split(String regex, int limit)
```

> `regex`：用作分隔符的正则表达式.
>
> `limit`：决定分割的次数和结果中空字符串的处理方式.
>
> limit 大于 0，最多分割 limit -1 次；limit 小于 0，尽可能多地分割，包含所有可能的空字符串.
> limit = 0, 尽可能多地分割，末尾的空字符串会被丢弃.
>
> 返回值：一个数组，元素是分割后的子字符串.

示例

limit > 0

```
String str = "a,b,c,d,e";
String[] result = str.split(",", 3);
// 输出：["a", "b", "c,d,e"]
```

limit < 0

```
String str = "a,,b,c,,";
String[] result = str.split(",", -1);
// 输出：["a", "", "b", "c", "", ""]
```

limit = 0

```
String str = "a,,b,c,,";
String[] result = str.split(",", 0);
// 输出：["a", "", "b", "c"]
```

用 `Scanner` 解析

```
String s = "Hello, This is JavaTpoint.";

// Create Scanner Object and pass string in it
Scanner scan = new Scanner(s);

// Check if the scanner has a token
System.out.println("Boolean Result: " + scan.hasNext());

// Print the string
System.out.println("String: " + scan.nextLine());

scan.close();
```

输出:

```
Boolean Result: true
String: Hello, This is JavaTpoint.
```

`.trim()`

> 用于删除字符串开头和结尾的空白字符.

```
public class Main {
    public static void main(String[] args) {
        String str = "   Hello,    Java!   ";
        System.out.println("After trim: '" + str.trim() + "'");
    }
}
```

```
After trim: 'Hello,    Java!'
```

> 中间的不会删除.

`String` 转 `int`

使用 `Integer` 类的静态方法 `.parseInt(s)`

```
String s = “150”;
int number = Integer.parseInt(s);
```

`int` 转 `String`

多种方法

```
String a = "" + 123;        // 隐式调用 toString
String a = Integer.toString(123); // 显式调用
String a = String.valueOf(123);
```

`String` 转 `Double`

```
String s = “3.14159”;
double number = Double.parseDouble(s);
```

`Double` 转 `String`

```
double d = 11.01;
String s = String.valueOf(d);
```

其他方法

1. `String substring(int beginIndex, int endIndex)`

提取字符串的子字符串（`beginIndex` 到 `endIndex`，左闭右开）

- **参数**：
  - `beginIndex`：起始索引（包含）.
  - `endIndex`：结束索引（不包含）.
- **返回值**：返回指定范围内的子字符串.
- **示例**：

```
String str = "Hello, Java!";
System.out.println(str.substring(7, 11)); // 输出：Java
```

---

2. `String substring(int beginIndex)`

用于提取从指定索引开始到字符串末尾的子字符串.

- **参数**：
  - `beginIndex`：起始索引（包含）.
- **返回值**：从 `beginIndex` 到末尾的子字符串.
- **示例**：

```
String str = "Hello, Java!";
System.out.println(str.substring(7)); // 输出：Java!
```

---

3. `char charAt(int index)`

获取字符串中指定索引处的字符.

- **参数**:
  - `index`：字符的索引（从 0 开始）.
- **返回值**：返回指定索引处的字符.
- **示例**:

```
String str = "Hello";
System.out.println(str.charAt(1)); // 输出：e
```

---

4. `boolean endsWith(String suffix)`

检查字符串是否以指定的后缀结尾.

- **参数**:
  - `suffix`：要检查的后缀字符串.
- **返回值**：如果字符串以 `suffix` 结尾，则返回 `true`，否则返回 `false`.
- **示例**:

```
String str = "Hello, Java!";
System.out.println(str.endsWith("Java!")); // 输出：true
System.out.println(str.endsWith("Hello")); // 输出：false
```

---

5. `boolean startsWith(String prefix)`

检查字符串是否以指定的前缀开始.

- **参数**:
  - `prefix`：要检查的前缀字符串.
- **返回值**：如果字符串以 `prefix` 开头，则返回 `true`，否则返回 `false`.
- **示例**:

```
String str = "Hello, Java!";
System.out.println(str.startsWith("Hello")); // 输出：true
System.out.println(str.startsWith("Java"));  // 输出：false
```

---

6. `int indexOf(String str)`

返回指定子字符串在字符串中首次出现的索引位置。如果子字符串不存在，则返回 `-1`.

- **参数**:
  - `str`：要查找的子字符串.

- **返回值**：子字符串的起始索引，如果没有找到，则返回 `-1`.
- **示例**：

```
String str = "Hello, Java!";
System.out.println(str.indexOf("Java")); // 输出：7
System.out.println(str.indexOf("World")); // 输出：-1
```

---

7. `int length()`

返回字符串的长度（字符串中字符的个数）.

- **参数**：无.
- **返回值**：字符串的长度.
- **示例**：

```
String str = "Hello, Java!";
System.out.println(str.length()); // 输出：12
```

StringBuilder

> 默认导入.

在 Java 中，`StringBuilder` 是一个用于创建和操作可变字符串的类. 与 `String` 类不同，`StringBuilder` 的字符串内容是 **可变的**（即可以直接修改内容，而无需创建新的实例）. 它非常适合需要频繁修改字符串内容的场景，例如字符串拼接、插入、删除等操作.

> `String` 对象的加号串联，实际上创建了一个新对象，而不是修改它们中的任何一个.

特点

1. **可变性：**
   - `String` 类中的字符串是不可变的，每次修改字符串都会创建一个新的字符串对象.
   - `StringBuilder` 的字符串是可变的，可以直接修改，不会创建新的对象.
2. **性能优越：**
   - 由于可变性，`StringBuilder` 在频繁进行字符串拼接或修改时性能更高（减少了对象的创建和销毁）.
3. **线程不安全：**
   - `StringBuilder` 是线程不安全的，但性能比线程安全的 `StringBuffer` 更高.
   - 如果需要线程安全的可变字符串类，可以使用 `StringBuffer`.

创建对象

1. 无参构造

创建一个空的 `StringBuilder` 对象，初始容量为 16.

```
StringBuilder sb = new StringBuilder();
```

> 容量的存在是为了优化性能和减少内存浪费. 字符数超过当前容量会自动扩容，规则是：新容量 = 旧容量 * 2 + 2

2. 指定初始容量

创建一个初始容量为 `capacity` 的 `StringBuilder` 对象.

```
StringBuilder sb = new StringBuilder(50);
```

3. 通过字符串初始化

创建一个包含指定字符串的 `StringBuilder` 对象.

```
StringBuilder sb = new StringBuilder("Hello");
```

常用方法

1. `append(String str)`

将指定的字符串追加到当前对象的末尾.

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(", Java!");
System.out.println(sb); // 输出：Hello, Java!
```

2. `insert(int offset, String str)`

在指定位置插入字符串.

```
StringBuilder sb = new StringBuilder("Hello!");
sb.insert(5, ", Java");
System.out.println(sb); // 输出：Hello, Java!
```

3. `replace(int start, int end, String str)`

替换从 `start` 到 `end` 索引之间的内容为新字符串.

```
StringBuilder sb = new StringBuilder("Hello, World!");
sb.replace(7, 12, "Java");
System.out.println(sb); // 输出：Hello, Java!
```

4. `delete(int start, int end)`

删除从 `start` 到 `end` 索引之间的内容（不包含 `end`）.

```
StringBuilder sb = new StringBuilder("Hello, Java!");
sb.delete(5, 7);
System.out.println(sb); // 输出：HelloJava!
```

5. `reverse()`

反转字符串内容.

```
StringBuilder sb = new StringBuilder("Hello");
sb.reverse();
System.out.println(sb); // 输出：olleH
```

6. `charAt(int index)`

返回指定索引处的字符.

```
StringBuilder sb = new StringBuilder("Hello");
System.out.println(sb.charAt(1)); // 输出：e
```

7. `length()`

返回当前字符串的长度.

```
StringBuilder sb = new StringBuilder("Hello");
System.out.println(sb.length()); // 输出：5
```

8. `setLength(int newLength)`

设置字符串的长度.

```
StringBuilder sb = new StringBuilder("Hello, Java!");
sb.setLength(5);
System.out.println(sb); // 输出：Hello
```

9. `capacity()`

返回当前 `StringBuilder` 的容量（不等于字符串长度）.

```
StringBuilder sb = new StringBuilder("Hello");
System.out.println(sb.capacity()); // 输出：16 + 字符串长度（21）
```

示例代码

```
public class Main {
    public static void main(String[] args) {
        // 创建 StringBuilder 对象
        StringBuilder sb = new StringBuilder("Hello");

        // 追加字符串
        sb.append(", Java!");
        System.out.println("After append: " + sb); // 输出：Hello, Java!

        // 插入字符串
        sb.insert(5, " World");
        System.out.println("After insert: " + sb); // 输出：Hello World, Java!

        // 替换字符串
        sb.replace(6, 11, "Beautiful");
        System.out.println("After replace: " + sb); // 输出：Hello Beautiful,
Java!

        // 删除字符串
        sb.delete(6, 15);
        System.out.println("After delete: " + sb); // 输出：Hello , Java!

        // 反转字符串
        sb.reverse();
        System.out.println("After reverse: " + sb); // 输出：!avaJ , olleH
    }
}
```

`StringBuilder` 与 `String` 的区别

特性	`StringBuilder`	`String`
可变性	可变，内容修改不会创建新对象.	不可变，每次修改都会创建新对象.
性能	修改字符串效率高，适合频繁拼接或修改操作.	修改字符串效率低.
线程安全性	非线程安全.	不适用于线程安全场景.
适用场景	频繁修改字符串的场景.	字符串内容不会改变的场景.

Tutorial 9

2D Array

```
int [ ][ ] m = new int[3][4];
```

```
int [ ] [ ] a = { { 1, 0, 2, 4 },
{ 2, 0, 4, 8 },
{ 4, 0, 9, 6 }};
```

> 只有声明的时候可以这样操作，赋值的时候不行.

Irregular 2D Array

```
int [ ] [ ] m = new int [3] [ ]; // m.length = 3 here
// m is an array of row array references
m[0] = new int[5]; // row 0 has 5 elements
m[1] = new int[2]; // row 1 has 2 elements
m[2] = new int[4]; // row 2 has 4 elements
m[0][0] = 9; // int elements as usual
m[2][3] = -7;
m[0] = null; // deleting the reference to first row
m[1] = m[2]; // copying a reference
```

增强型 `for` 循环

> 简化数组迭代运算

```
double [ ] rainfall = new double [12];
double sum = 0;
for (double item : rainfall) // item: rainfall[0], rainfall[1], ...
sum += item;
```

`Arrays` 类

`Arrays` is a utility class providing class methods.

```
import java.util.Arrays;
...
{ // rainfall[0], rainfall[1], rainfall[2],
    double[ ] monthJanFebMar = Arrays.copyOfRange(rainfall, 0, 3);
    Arrays.sort(rainfall); // 升序排序
    int index = Arrays.binarySearch(rainfall, 56); // find a value
    Arrays.fill(rainfall, 3, 11, 3.14159); // fill rainfall[3]...rainfall[10]
with PI
}
```

Tutorial 10

> Inheritance and Problem Solving

Tutorial 11

> 考前复习

Self-Learn Exercises

Instructions

1. Create a new NetBeans Java with Ant Application, e.g., name it as *WorkingProject*. **DO NOT TICK** Create Main Class option.
2. Each Java source file is **default without package**. Download and save given source file under *NetBeansProjects/WorkingProject/src/* folder. **Keep the given Java source filename** such as *Lab00Ex00.java*.

3. Change Main Class setting to *Lab00Ex00* under NetBeans Project Properties. Otherwise, right-click on *Lab00Ex00* to **Run file (Shift-F6)** every time
4. Finish the exercise according to instructions given in the associated PDF file. Edit, Build/Run, Test and Debug under NetBeans.
5. Login Gradescope, and enter the corresponding submission item. Browse and upload *NetBeansProjects/WorkingProject/src/Lab00Ex00.java* to obtain feedback on test results.

Assignments

Java 常用包

Math API

Java 的 Math 类提供了一系列用于数学运算的方法. 这个类位于 `java.lang` 包中（默认导入），所以不需要额外显式导入. 以下是 Math 类的主要用法和一些常用方法：

1. 基本运算：

```
// 绝对值
int absValue = Math.abs(-5);  // 结果: 5

// 最大值和最小值
int max = Math.max(10, 20);  // 结果: 20
int min = Math.min(10, 20);  // 结果: 10

// 幂运算
double power = Math.pow(2, 3);  // 结果: 8.0

// 平方根
double sqrt = Math.sqrt(16);  // 结果: 4.0

// 立方根
double cbrt = Math.cbrt(27);  // 结果: 3.0
```

2. 舍入方法：

```
// 向上舍入
double ceil = Math.ceil(3.1);  // 结果: 4.0

// 向下舍入
double floor = Math.floor(3.9);  // 结果: 3.0

// 四舍五入
long round = Math.round(3.5);  // 结果: 4
```

3. 三角函数：

```
// 正弦
double sin = Math.sin(Math.PI / 2);  // 结果: 1.0

// 余弦
double cos = Math.cos(Math.PI);  // 结果: -1.0

// 正切
double tan = Math.tan(Math.PI / 4);  // 结果: 1.0

// 反正弦
double asin = Math.asin(1);  // 结果: π/2

// 反余弦
double acos = Math.acos(0);  // 结果: π/2

// 反正切
double atan = Math.atan(1);  // 结果: π/4
```

4. 对数函数：

```
// 自然对数
double ln = Math.log(Math.E);  // 结果: 1.0

// 以10为底的对数
double log10 = Math.log10(100);  // 结果: 2.0
```

5. 随机数：

```
// 生成 0.0 到 1.0 之间的随机数
double random = Math.random();
```

6. 常量：

```
// 圆周率
double pi = Math.PI;  // 3.141592653589793

// 自然对数的底
double e = Math.E;  // 2.718281828459045
```

7. 角度和弧度转换：

```
// 角度转弧度
double radians = Math.toRadians(90);  // π/2

// 弧度转角度
double degrees = Math.toDegrees(Math.PI / 2);  // 90.0
```

8. 双曲函数：

```
// 双曲正弦
double sinh = Math.sinh(1);

// 双曲余弦
double cosh = Math.cosh(1);

// 双曲正切
double tanh = Math.tanh(1);
```

9. 符号相关：

```
// 符号函数
int sign = Math.signum(-5);  // 结果: -1.0

// 复制符号
double copySign = Math.copySign(3.14, -1);  // 结果: -3.14
```

10. 指数相关：

```
// e的x次幂
double exp = Math.exp(2);  // e^2

// e的x次幂减1
double expm1 = Math.expm1(1);  // e^1 - 1
```

使用示例：

```
public class MathExample {
    public static void main(String[] args) {
        // 计算圆的面积
        double radius = 5;
        double area = Math.PI * Math.pow(radius, 2);
        System.out.println("Circle area: " + area);

        // 计算三角形的斜边
        double a = 3, b = 4;
        double hypotenuse = Math.sqrt(Math.pow(a, 2) + Math.pow(b, 2));
        System.out.println("Hypotenuse: " + hypotenuse);

        // 生成1到10之间的随机整数
        int randomInt = (int)(Math.random() * 10) + 1;
        System.out.println("Random integer: " + randomInt);
    }
}
```

注意事项：

- Math 类的所有方法都是静态的，可以直接通过类名调用.
- 大多数 Math 方法使用 double 类型的参数和返回值.
- 三角函数方法使用弧度而不是角度，需要时可以使用 `toRadians()` 和 `toDegrees()` 进行转换.
- `random()` 方法生成的是伪随机数，对于需要高质量随机数的场景，考虑使用 `java.util.Random` 或 `java.security.SecureRandom`.

历年考题

易错点

8 primitive types

byte, short, int, long, float, double, char, boolean

先赋值，后打印

`println(a=1);`