1. Java简介与安装

1.1 Java简介

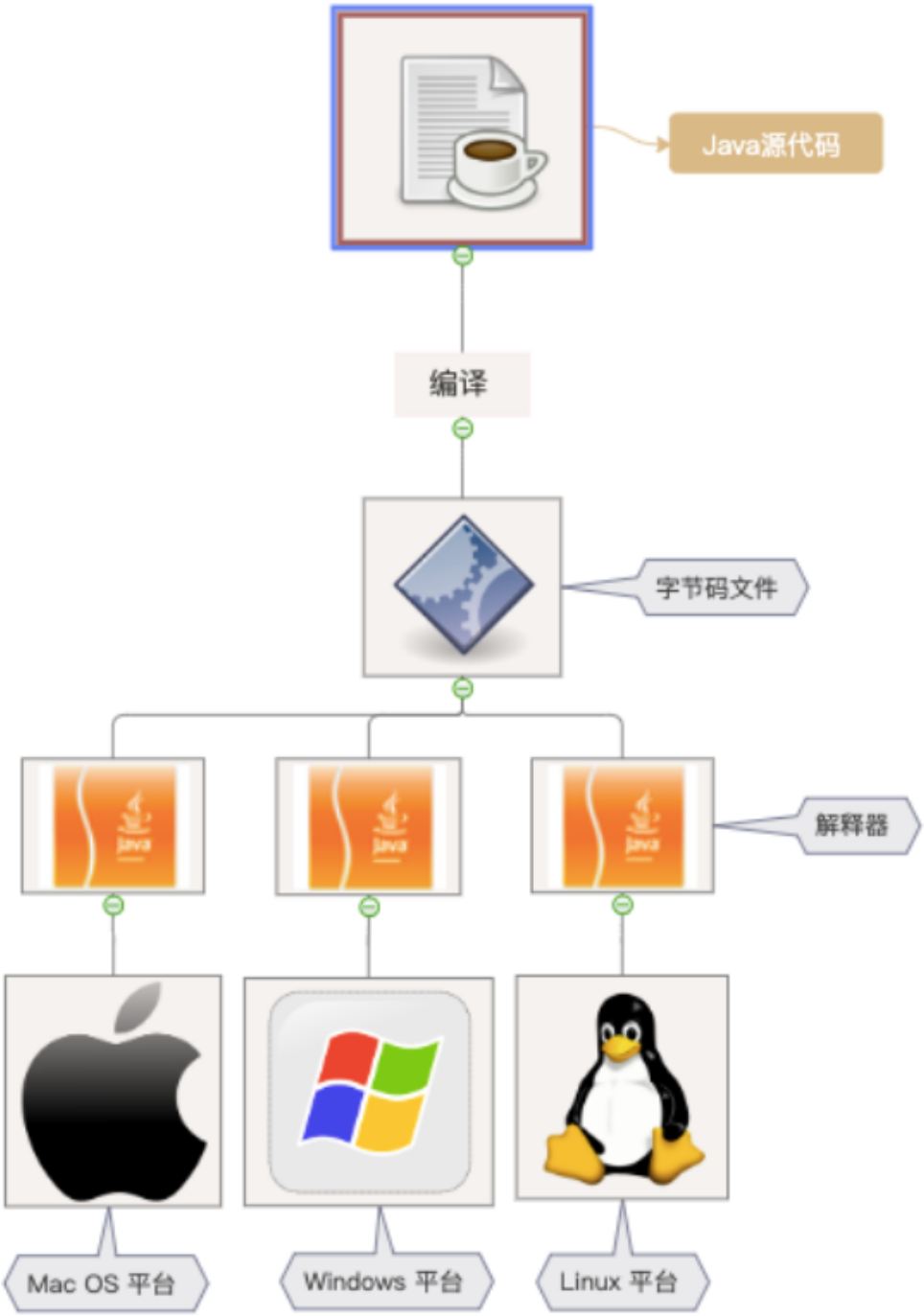
1995 年 5 月 23 日，sun公司宣布Java 语言诞生。Java 语言之父`詹姆斯·高斯林`

目前 Java 提供以下三个版本：

- **Java Platform, Enterprise Edition**（Java EE：Java 平台企业版）
- **Java Platform, Standard Edition**（Java SE：Java 平台标准版）
- **Java Platform, Micro Edition**（Java ME：Java 平台微型版）

语言特性：

- 面向对象
  - 面向对象是一种编程思想。面向对象的主要思想是：围绕着我们所操纵的 “事物”（即对象）来设计软件
- 跨平台
  - `一次编译，到处执行`
  - Java 的思想是，将代码编译为`中间语言`，中间语言是**字节码**，解释器是 Java 虚拟机（`JVM`）。字节码文件可以通用，JVM 是特定于平台的。



- 异常处理
  - 以抛出异常的方式来进行异常处理，在 Java 中可以使用 `catch` 关键字来捕获在 `try` 语句块中所发生的异常。
- 垃圾回收机制
- 自动垃圾回收机制，让使用 Java 编写的代码更加健壮，**降低了内存泄漏和溢出的风险。**
- ……

java的版本:

- JDK 1.0（1996 年 1 月 23 日）
- ……
- Java SE 7（2011 年 7 月 28 日）
- Java SE 8（2014 年 3 月 18 日）
- Java SE 9（2017 年 9 月 21 日）

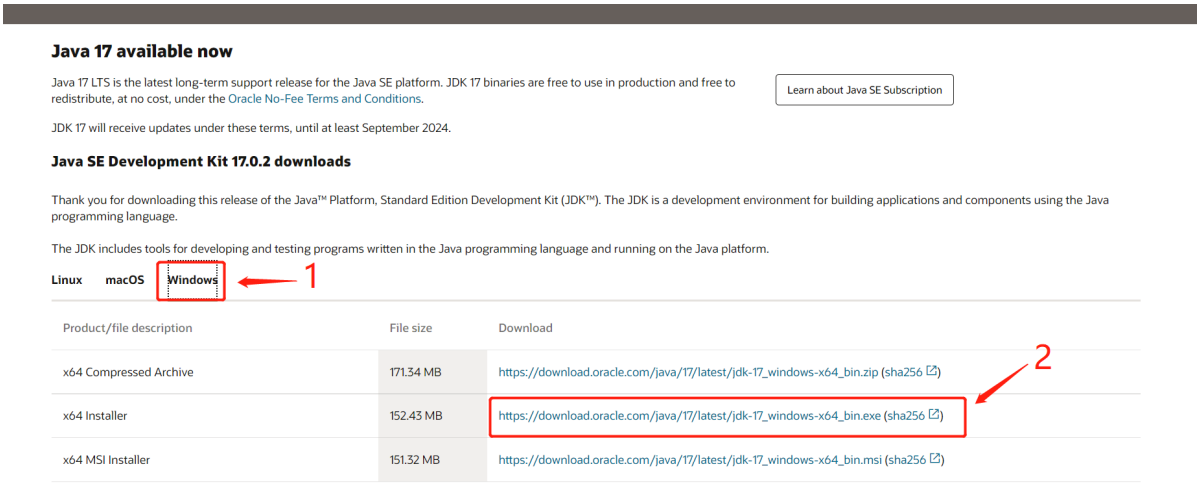
- Java SE 10（2018 年 3 月 20 日）
- ……
- Java SE 17（2021 年 10月 19日）

1.2 Java安装

在安装 Java 之前，我们需要理解两个名词，JDK 和 JRE：

- **JRE**：Java Runtime Environment （Java 运行时环境）
- **JDK**：Java Development Kit （Java 开发工具包）

下载link : [jdk17 download]



安装包下载完成后，打开安装包，开始安装。

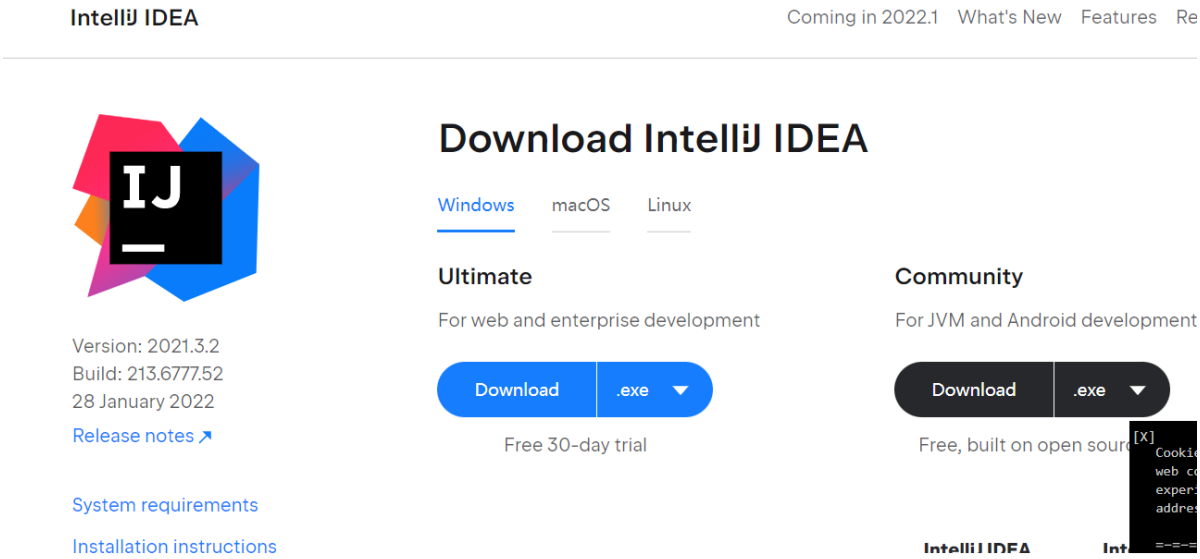
安装完成后，**无需到系统变量里配置Path**

1.3 Java集成开发工具安装

IDE 即 `Integrated Development Environment` 的缩写，中文意为**集成开发环境**，是用于提供程序开发环境的应用程序，一般包括代码编辑器、编译器、调试器和图形用户界面等工具。集成了代码编写功能、分析功能、编译功能、调试功能等一体化的开发软件服务套件。

1.3.1 IntelliJ IDEA

下载link : [IDEA download]



1.4 demo

以上安装了java环境和Java的开发工具，在idea里新建一个项目，完成我们一个简单demo

- 控制台输出 **hello world**!

代码如下：

```
public class Demo1 {
    public static void main(String[] args) {
        System.out.println("hello world!");
    }
}
```

结果如下：

```
hello world!
```

2. Java基础

上面的demo示例代码中有以下几点：

1. 大小写敏感
2. 类
3. 主方法
4. 源代码文件

2.1 大小写敏感

java语言是大小写敏感的，就是指所有的命名都区分大小写 (文件名、类名、方法名等等)。

2.2 类

```
public class Demo1 {
    //something
}
```

其中，`public` 是一个关键字，它属于访问控制符，它表示这个类是公开的。关于什么是访问控制符，后面还会讲到。

紧接着的 `class` 也是一个关键字用于声明类， `Demo1` 是类名，类名的首字母要大写。

如果类名由多个单词组成，那么每个单词的首字母都要大写(**驼峰命名规则**)。

2.3 主方法

示例代码的 `class` 内部（指的是类名 `Demo1` 后面用大括号 `{}` 包含的内容），这个方法被称为主方法。每个类只能拥有一个主方法。

需要特别注意的是：**所有的 Java 程序都从主方法开始执行。**

```
public static void main(String[] args) {
        //do something
    }
```

另外补充一点，类的内部不仅可以包含一个主方法，也可以包含多个**方法**。在学习方法的概念之前，我们将主要在主方法中编写示例代码

2.4 源代码文件

既然源代码需要提供给计算机执行，我们就要将源代码以文件的形式保存在计算机的磁盘上。

需要注意的是：源代码文件的命名必须与类名相同，且后缀名为`.java`。例如：`Demo1` 类对应的源代码文件名应该为 `Demo1.java`。

2.5 Java 标识符

在 Java 中，标识符通常用来给类、对象、变量、方法、接口、自定义数据类型命名。

命名规范：

- 以字母（**A-Z** 或者 **a-z**），美元符号（**$**）或下划线（_）开始；
- **首字母后**可以是字母、数字、下划线的任意组合；
- 正如我们前面所提到的，标识符是**大小写敏感**的；
- 需要特别注意的是，Java 中的**关键字**不能被用作标识符。

以下是合法的标识符命名实例：

- $name
- _World
- Hello

- world

以下不合法的标识符命名实例：

- ￥color
- 12name
- *abc
- final

2.6 Java 关键字和保留字

关键字（`Keyword`）是 Java 语言中的特殊标记。它已经被语言本身预先使用，因此我们不能使用关键字作为我们标识符的命名。

例如 Java 基本类型的 `int`、`boolean`，流程控制语句中的 `if`、`for`，访问修饰符 `public`，以及一些用于声明和定义 Java 类、包、接口的 `class`、`package`、`interface`。

而保留字（`Reserved word`）可能是未来的关键字，也就是说可能在未来的版本中，Java 语言作为特殊标记。

无论是关键字还是保留字，我们都要记住：不能使用它们作为我们的代码中的标识符。

2.7 空行和注释

注释：

看如下java 源码中的注释代码段：

```
    /**
     * Prints a String and then terminate the line.  This method behaves as
     * though it invokes <code>{@link #print(String)}</code> and then
     * <code>{@link #println()}</code>.
     *
     * @param x  The <code>String</code> to be printed.
     */
    public void println(String x) {
        synchronized (this) {
            print(x);
            newLine();
        }
    }
```

Java 语言提供了三种类别的注释：

1. 单行注释

单行注释用于注释一行文本，它以双斜线开始，后面跟上要注释的内容，其写法为：

```
// 被注释的内容
```

在 Java 代码中，它是这样的：

```
public class Demo1 {
    public static void main(String[] args) {
        // print hello world
        System.out.println("hello world");
    }
}
```

2. 多行注释

多行注释用于注释多行文本，它以 `/*` 开头，以 `*/` 结尾，其写法为：

```
/*
被注释的第一行内容
被注释的第二行内容
被注释的第三行内容
*/
```

3. 文档注释

Java 中还有一种特殊的多行注释 —— 文档注释，它以 `/**` 开头，以 `*/` 结尾，如果有多行，则每行都以 `*` 开头，其在代码中的写法为：

```
    /**
     * Prints a String and then terminate the line.  This method behaves as
     * though it invokes <code>{@link #print(String)}</code> and then
     * <code>{@link #println()}</code>.
     *
     * @param x  The <code>String</code> to be printed.
     */
    public void println(String x) {
        synchronized (this) {
            print(x);
            newLine();
        }
    }
```

这种特殊的多行注释需要写在类和方法的定义处。另外通常在程序开头加入作者，时间，版本，要实现的功能等内容注释，方便程序的维护以及程序员的交流。

空行：

空行就是空白行，与注释一样，同样不会被编译器解析。适当地使用空行，可以让代码的结构看起来更好看。

2.8 变量

在`Java`语言中，我们需要做两件事才能创建一个变量：

1. 给变量起一个名字
2. 定义变量的数据类型

创建变量的过程也叫**声明变量**，声明变量的语法如下：

```
DataType 变量名;
```

实例如下:

```
int a;
```

上述代码，声明了一个名字为`a`，类型为**整型**的变量。

2.8.1. 给变量赋值

上述的变量`a`为`int`类型（整型），因此只能存放整数。

一旦声明了一个变量，我们就可以使用赋值语句为其赋值，实例如下：

```
// 变量声明语句
int a;
// 赋值语句
a = 1;
```

为变量分配值的语句，就称为赋值语句。需要特别提醒的是，语句中`=`的意义不同于数学中的等号，在 Java 中，`=`是赋值符号。

对于变量的声明和赋值操作，我们也可以将这两条语句合并成一条语句，实例如下：

```
int a = 1;
```

在作用域范围内，变量的值能够随时访问或重新赋值，比如：

```
class PrintVariable {
    public static void main(String[] args) {
        int a = 20;
        System.out.println("修改前变量a=" + a);
        // 将变量a重新赋值为100
        a = 100;
        System.out.println("修改后变量a=" + a);
    }
}
```

2.8.2 变量命名规范

- 变量的名称可以是**任何合法的标识符**，以字母，美元符号`$`或下划线`_`开头。但是，按照约定俗成，变量应始终以字母开头，不推荐使用美元符号和下划线开头；
- **开头后续的字符可以是字母、数字、美元符号或下划线。**为变量选择名称时，推荐使用完整的英文单词，不推荐使用单词缩写，更不要使用中文拼音。这样做有利于代码的阅读和理解。另外请牢记，选择的名称不能是**关键字**或**保留字**；
- **变量命名区分大小写**；
- **变量命名应采用小驼峰命名法。**所谓小驼峰命名法，就是如果你选择的名称只包含一个单词，那么用全部小写字母拼写该单词；如果名称包含多个单词，请将第二个单词起的每个单词的第一个字母都大写；
- **如果变量存储了一个常量值，要将每个字母大写并用下划线字符分隔每个单词。**比如`static final int MAX_NUM = 100`。按照约定俗成，除了常量命名的情况，下划线字符永远不会在其他地方使用。

2.8.3 常量

所谓常量，就是恒常不变的量。我们可以将常量理解成一种特殊的变量。

与变量不同的是，一旦它被赋值后，在程序的运行过程中不允许被改变。常量使用`final`关键字修饰：

```
final DataType 常量名 = 常量值;
```

tip : 常量的命名规范与普通变量有所不同，要将每个字母大写并用下划线字符分隔每个单词。

如果我们尝试在代码中修改常量的值：

```
class ConstantTest {
    public static void main(String[] args) {
        // 声明并初始化常量 TOTAL_NUM
        final int TOTAL_NUM = 200;
        // 对 TOTAL_NUM 重新赋值
        TOTAL_NUM = 20;
    }
}
```

编译执行代码，编译器将会报错：

```
java: 无法为最终变量TOTAL_NUM分配值
```

其实IDE工具已经有如下错误提示：

```
    // 声明并初始化常量 TOTAL_NUM
    final int TOTAL_NUM = 200;
    // 对 TOTAL_NUM 重新赋值
    TOTAL_NUM = 20;
}
```

Cannot assign a value to final variable 'TOTAL_NUM'

Make 'TOTAL_NUM' not final Alt+Shift+Enter More actions... Alt+Enter

```
final int TOTAL_NUM = 200
```

Java 语言定义了以下**4 种变量**：

- 实例变量（见代码中`instanceVariable`）
- 类变量（见代码中`classVariable`）又被称为**静态字段、静态变量**
- 局部变量（见代码中`localVariable`）
- 参数（见代码中`parameter`和`args`）

```
public class KindsOfVariables {
    // 1.实例变量
    public int instanceVariable = 1;
    // 2.类变量
    public static int classVariable;

    public void demoMethod(int parameter) { // 3.参数
        // 4.局部变量
        int localVariable;
    }

    public static void main(String[] args) {
        // 入口方法
    }
}
```

2.8.4 实例变量

```
public class Student {
    // 实例变量 name
    public String name;
    // 实例变量 age
    public int age;

    // 构造方法
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // 打印学生基本信息的方法
    public void printInfo() {
        System.out.println("学生姓名为：" + name);
        System.out.println("学生年龄为：" + age);
```

```
    }

    public static void  main(String[] args) {
        // 实例化两个学生对象
        Student student1 = new Student("小张", 20);
        Student student2 = new Student("小李", 18);
        // 分别调用打印学生基本信息方法，打印两个对象下的两个实例变量
        student1.printInfo();
        System.out.println("--------分割线--------");
        student2.printInfo();
    }
}
```

运行结果如下：

```
学生姓名为：小张
学生年龄为：20
--------分割线--------
学生姓名为：小李
学生年龄为：18
```

2.8.5 类变量

类变量在类范围内使用`static`关键字修饰，因此类变量又被称为**静态字段、静态变量**。

`static`修饰符告诉编译器，无论类被实例化多少次，类变量始终只有一个变量副本。只要类被加载到内存中，它就会存在。

另外，类变量可以被声明为**常量**，通过使用`final`关键字以表示变量永远不会改变。例如：`public static final NAME = "咖啡"`，这里的`NAME`就是不会改变的常量。再次提醒，在常量的命名规范中，要将字母全部大写。

例子：

```
public class Course {

    // 类变量 courseType
    public static String courseType = "Java";
    // 常量 COURSE_NAME
    public static final String COURSE_NAME = "基础教程";

    public static void main(String[] args) {
        // 分别打印类变量和常量
        System.out.println(Course.courseType);
        System.out.println(Course.COURSE_NAME);
    }
}
```

运行结果：

```
Java
基础教程
```

类变量和类相关，因此**不需要使用`new`关键字**实例化对象后再调用，可以直接通过**类名 + `.`点运算符 + 类变量名**的方式调用。

上述代码中，`courseType`和`COURSE_NAME`都使用`static`关键字修饰，它们都可以直接通过`Course.变量名`的方式调用。

2.8.6 局部变量

局部变量是在方法范围内被声明和使用的。它们没有任何关键字修饰，可以根据**变量声明的位置**来认定局部变量（即方法的左花括号和右花括号之间），因此，局部变量只可以对声明它们的方法可见。方法返回后，它们将被销毁。

在`main`方法中的局部变量实例：

```
public static void main(String[] args) {
    // 局部变量 name
    String name = "小慕";
    // 局部变量 age
    int age = 20;
    System.out.println("姓名：" + name);
    System.out.println("年龄：" + age);
}
```

我们再来看一个自定义方法中的局部变量实例：

```
public class PrintNumber {

    public void printer() {
        int num = 10;
        for(int i = 1; i <= num; i ++) {
            System.out.println(i);
        }
    }

}
```

局部变量和方法相关，因此只能在方法内部局部定义和使用，在第二个实例中没有代码注释。

2.8.7 参数

参数是用于传递给方法的变量（例如入口方法`main`中的`args`），它们可以在方法中的任何位置被调用。在方法执行的期间位于内存中，方法返回后被销毁。

例如，上面实例变量的实例中，`Student`类的构造方法就接收两个参数，如下：

```
// Student 类构造方法
public Student(String name, int age) {  // name 和 age 就是传递给Student构造方法的参数
    this.name = name;
    this.age = age;
}
```

注意，方法体中的`this.name`和`this.age`指代的是**实例变量**，而`name`和`age`是参数，它们被用于赋值给实例变量。

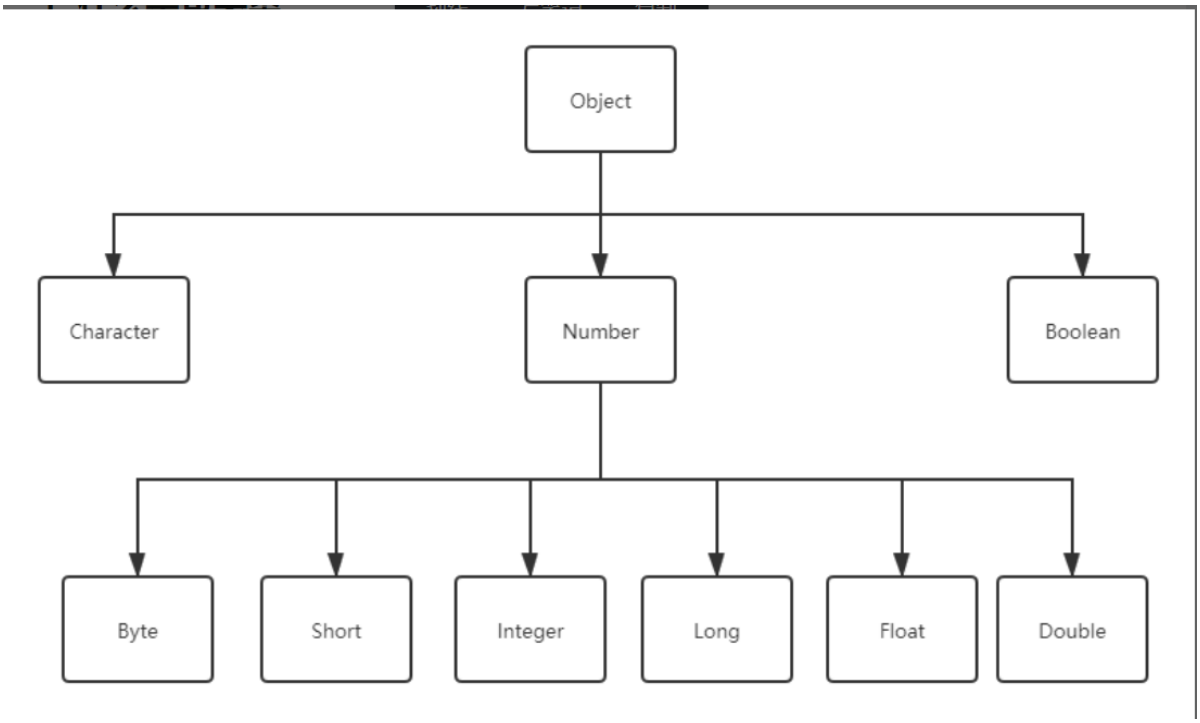
2.9 基本数据类型

在 Java 中，一共有两大数据类型：

- 基本数据类型（内置数据类型）
    - `byte，short，int，long，float，double，char 和 boolean`
- 引用数据类型

Java 有 8 种基本数据类型，Java 中的每个基本类型都被包装成了一个类，这些类被称为包装类。

包装类可以分为 3 类：`Number`、`Character`、`Boolean`，如下所示：



每个包装类中，官方都提供了一些方法供我们使用：

例如 **Integer类**

- `toString()`：返回一个表示该 Integer 值的 String 对象；
- `static Integer valueOf(String str)`：返回保存指定的 String 值的 Integer 对 象；
- `int parseInt(String str)`：返回包含在由 str 指定的字符串中的数字的等价整数值。

还有一些常用的常量：

1. `MAX_VALUE`：表示 int 型可取的最大值；
2. `MIN_VALUE`：表示 int 型可取的最小值；
3. `TYPE`：表示基本类型 Class 实例。

```
public class WrapperClassDemo1 {

    public static void main(String[] args) {
        int maxValue = Integer.MAX_VALUE;
        int minValue = Integer.MIN_VALUE;
        System.out.println("int 类型可取的最大值" + maxValue);
        System.out.println("int 类型可取的最小值" + minValue);
    }

}
```

运行结果如下：

```
int 类型可取的最大值2147483647
int 类型可取的最小值-2147483648
```

其它包装类的用法类似。

3.0 java运算符

`Java`提供了一组丰富的运算符来操作变量。 我们可以将所有`Java`运算符分为以下几类：

- 算术运算符
- 关系运算符
- 位运算符
- 逻辑运算符
- 赋值运算符
- 其他运算符

3.0.1 算术运算符

在例子中，初始化两个整型变量a、b：`int a = 2;` `int b = 4;`

运算符	描述	例子
+	加法运算符 （也用于字符串连接）	a + b 等于 6
-	减法运算符	a - b 等于 -2
*	乘法运算符	a * b 等于 8
/	除法运算符	b / a 等于 2
%	取余运算符	b % a 等于 0
++	自增运算符	a ++ 等于 3
--	自减运算符	b – 等于 3

`++i`表示先加1再引用`i`， `i++`表示先引用`i`再加1，`--`也是类似。

列子如下：

```java
public static void main(String[] args) {
        int a = 2;
        int b = 4;
        int i = a + b;
        int i1 = a - b;
        int i2 = a * b;
        int i3 = a / b;
        int i4 = a % b;
        int i5 = a++;
        int i6 = b--;
        System.out.println("i = " + i);
        System.out.println("i1 = " + i1);
        System.out.println("i2 = " + i2);
        System.out.println("i3 = " + i3);
        System.out.println("i4 = " + i4);
        System.out.println("i5 = " + i5);
        System.out.println("i6 = " + i6);
}
```

运行结果如下：

```
i = 6
i1 = -2
i2 = 8
i3 = 0
i4 = 2
i5 = 2
i6 = 4
```

3.0.2 关系运算符

关系运算符又称为**比较运算符**，比较的结果是一个布尔类型的值（`true`或`false`）。

在例子中，初始化两个整型变量a、b：`int a = 2;` `int b = 4;`

运算符	描述	例子
`==`	检查如果两个操作数的值是否相等，如果相等则条件为真。	(a == b) 为假
`!=`	检查如果两个操作数的值是否相等，如果值不相等则条件为真。	(a != b) 为真
`>`	检查左操作数的值是否大于右操作数的值，如果是那么条件为真。	(a > b) 为假
`<`	检查左操作数的值是否小于右操作数的值，如果是那么条件为真。	(a < b)为真
`>=`	检查左操作数的值是否大于或等于右操作数的值，如果是那么条件为真。	(a >= b)为假
`<=`	检查左操作数的值是否小于或等于右操作数的值，如果是那么条件为真。	(a <= b)为真

3.0.3 位运算符

Java 语言还提供了对**整数类型**执行按位和移位操作的运算符，称作**位运算符。**

它在实际的编码中并不常用，这部分内容了解即可。

假设`a = 60，b = 13`;它们的二进制格式表示将如下：

```
a = 0011 1100
b = 0000 1101
-----------------
a & b = 0000 1100
a | b = 0011 1101
a ^ b = 0011 0001
~a = 1100 0011
```

下表列出了位运算符的基本运算，假设整数变量`a 的值为 60`和`变量 b 的值为 13`：

运算符	描述	例子
&（按位与）	如果相对应位都是1，则结果为1，否则为0	（a&b），得到12，即0000 1100
|（按位或）	如果相对应位都是 0，则结果为 0，否则为 1	（a | b）得到61，即0011 1101
^（按位异或）	如果相对应位值相同，则结果为0，否则为1	（a ^ b）得到49，即0011 0001
~（按位取反）	按位取反运算符翻转操作数的每一位，即0变成1，1变成0。	（~a）得到-61，即1100 0011
<< （左位移）	按位左移运算符。左操作数按位左移右操作数指定的位数。	a << 2得到240，即1111 0000
>> （右位移）	按位右移运算符。左操作数按位右移右操作数指定的位数。	a >> 2得到15即 1111
>>> （零填充右移）	按位右移补零操作符。左操作数的值按右操作数指定的位数右移，移动得到的空位以零填充。	a>>>2得到15即0000 1111

3.0.4 逻辑运算符

逻辑运算符可以在表达式中生成组合条件，例如在执行特定语句块之前必须满足的两个或多个条件。使用逻辑运算符，可以描述这些组合条件。逻辑运算的返回结果只能为真或假。

Java 语言中的逻辑运算符，如下表所示：

在例子中，初始化两个整型变量a、b：`int a = 0;` `int b = 1;`

运算符	描述	例子
&&（逻辑与）	当且仅当两个操作数都为真，条件才为真。	(a && b)为假
||（逻辑或）	如果任何两个操作数任何一个为真，条件为真。	(a || b)为真
！（逻辑非）	用来反转操作数的逻辑状态。如果条件为真，则逻辑非运算符将得到假。	!(a && b)为假

3.0.5 短路运算

`&&`和`||`运算符存在**短路**行为。短路的意思是：只有在需要的时候才会判断第二个操作数的真假。例如：

```
class LogicOperators {
    public static void main(String[] args){
        int a = 1, b = 2;
        if((a == 2) && (b == 2)) {
            System.out.println("a和b都等于2");
        }
        if((a == 1) || (b == 1)) {
            System.out.println("a等于1或b等于1");
        }
    }
}
```

运行结果：

```
a等于1或b等于1
```

程序解析：有两个整型变量`a`和`b`，值分别为`1`和`2`。第一个`if`语句的条件为逻辑与运算，其第一个操作数`a == 2`为假，所以无论第二个操作数是真是假，都不去判断，条件直接被判定为假；第二个`if`语句的条件为逻辑或运算， 其第一个操作数`a == 1`为真，所以无论第二个操作数是真是假，都不会去判断，条件直接被判定为真。这就是所谓的短路。

3.0.6 赋值运算符

赋值运算符是为指定变量分配值的符号。下标列出了常用 Java 中常用的赋值运算符：

运算符	描述	例子
=	简单的赋值运算符。将值从右侧操作数分配给左侧操作数。	c = a + b将a + b的值赋给c
+=	加和赋值运算符。它将右操作数添加到左操作数，并将结果分配给左操作数。	c + = a等于c = c + a
-=	减和赋值运算符。它从左侧操作数中减去右侧操作数，并将结果分配给左侧操作数。	c -= a等效于c = c – a
*=	乘和赋值运算符。它将右操作数与左操作数相乘，并将结果分配给左操作数。	c *= a等效于c = c * a
/ =	除和赋值运算符。它将左操作数除以右操作数，并将结果分配给左操作数。	c /= a等于c = c / a

下例：

```
public class OperateDemo {
    public static void main(String[] args) {
        // 分组初始化三个变量 num1、num2、result，值分别为 20、10、0
        int num1 = 20, num2 = 10,  result = 0;
        System.out.println("初始值：");
        System.out.println("num1=" + num1);
        System.out.println("num2=" + num2);
        System.out.println("result=" + result);
        System.out.println("开始赋值运算：");
        result += num1;
        System.out.println("result += num1 结果为：" + result);
        result -= num2;
        System.out.println("result -= num2 结果为：" + result);
        result *= num1;
        System.out.println("result *= num1 结果为：" + result);
        result /= num2;
        System.out.println("result /= num2 结果为：" + result);
        result %= 15;
        System.out.println("result %= 15 结果为：" + result);
    }
}
```

运行结果：

```
初始值：
num1=20 num2=10 result=0
开始赋值运算：
result += num1 结果为：20
result -= num2 结果为：10
result *= num1 结果为：200
result /= num2 结果为：20
result %= 15 结果为：5
```

3.0.7 其他运算符

条件运算符： **(? :)**

条件运算符也称为**三元运算符**。该运算符由三个操作数组成，用于判断**布尔表达式**。它的目的是确定应将哪个值分配给变量。条件运算符的语法为：

```
变量 = (布尔表达式) ? 值1 : 值2
```

如果布尔表达式为真，就将`值1`分配变量，否则将`值2`分配给变量。

如下例子：

```
public class ConditionalOperators {
    public static void main(String[] args) {
        int age = 15;
        System.out.println(age >= 18 ? "你已经成年" :  "你还未成年");
    }
}
```

运行结果：

```
你还未成年
```

instanceof 运算符

了解 instanceof 运算符需要一些面向对象的前置知识

`instanceof`运算符将对象与指定类型进行比较，检查对象是否是一个特定类型（类类型或接口类型）。

instanceof 运算符的语法为：

```
( Object reference variable ) instanceof  (class/interface type)
```

如果`instanceof`左侧的变量所指向的对象，是`instanceof`右侧类或接口的一个对象，结果为真，否则结果为假。

```
public class InstanceOfOperators1 {
    public static void main(String[] args) {
        String name = "zhangsan";
        boolean b = name instanceof String;
          System.out.println("结果为: " + b);
    }
}
```

由于字符串变量`name`是`String`类型，所以执行代码，屏幕会打印：

```
结果为：true
```

注意，`instanceof`运算符不能用于操作基本数据类型，如果将字符串类型`name`变量改为一个`char`类型的变量，编译代码将会报错:

```
java: 意外的类型
  需要：引用
  找到：    char
```

3.0.8 运算符的优先级

当多种运算符在一同使用的时候，会有一个执行先后顺序的问题。

下表中的运算符按优先顺序排序。运算符越靠近表格顶部，其优先级越高。具有较高优先级的运算符将在具有相对较低优先级的运算符之前计算。同一行上的运算符具有相同的优先级。

类别	操作符	关联性
后缀	() [] . (点操作符)	左到右
一元	+ + - ！ ~	**从右到左**
乘性	* /%	左到右
加性	+ -	左到右
移位	>> >>> <<	左到右
关系	>> = << =	左到右
相等	== !=	左到右
按位与	&	左到右
按位异或	^	左到右
按位或	|	左到右
逻辑与	&&	左到右
逻辑或	||	左到右
条件	? :	**从右到左**
赋值	= + = - = * = / =%= >> = << =&= ^ = | =	**从右到左**
逗号	,	左到右

当**相同优先级**的运算符出现在同一表达式中时，如何控制它们计算的先后。看一个实例：

```
public class OperateorsPriority {
    public static void main(String[] args) {
        int a = 2;
        int b = 4;
        int c = 6;
        int result = a + b - c + a;
        System.out.println("result = " + result);
    }
}
```

在计算`result`的语句的右侧，`+`、`-`两个运算符优先级相同，如果我们不加以控制，将按照从左到右顺序计算，打印结果为`result = 2`；但是如果我们想先计算`a + b`和`c + a`的值，再计算两者之差，我们可以使用括号`()`将其顺序进行控制：`(a + b) - (c + a)`，再执行代码将打印我们想要的结果：`result = -2`。

3.1 Java 条件语句

3.1.1 if 语句

当我们需要根据给定的条件来决定是否执行一段代码时，`if`语句就用上了。`if`块仅在与其关联的布尔表达式为`true`时执行。

```
if (条件) {
    // 当条件成立时执行此处代码
}
```

例子：

```
public class IfStatement1 {
    public static void main(String args[]) {
        int age = 18;
        if(age >= 18) {
            System.out.println("你成年了！");
        }
    }
```

运行结果如下：

```
你成年了!
```

当**条件**`age >= 18`成立时，也就是布尔表达式计算结果为`true`(真)，`if`语句块会执行。显然上面实例中`age`变量的值为 18，条件是成立的，执行程序，将会打印“你成年了！”。

如果语句块内只有一条语句，我们也可以去掉花括号`{}`：

```
if(age >= 18)
    System.out.println("你成年了！");
```

即使`if`语句块只有一条语句，也并不推荐这种不易于阅读的写法。

3.1.2 if ... else ... 语句

`if` 语句可以结合 `else` 来使用，当布尔表达式计算结果为 `false` （假）时，`else` 语句块将会执行。

`if .. .else` 语句用于有条件地执行两个代码块的其中一个，具体执行哪一个代码块，取决于布尔条件的结果。

```
if (条件) {
    // 如果条件成立，执行此处代码
} else {
    // 如果条件不成立，执行此处代码
}
```

例子：

```
public class IfElseStatement1 {
    public static void main(String args[]) {
        int age = 15;
        if(age >= 18) {
            System.out.println("你成年了！");
        } else {
            System.out.println("你没成年！");
        }
    }
}
```

运行结果：

```
你没成年！
```

Java 支持使用**条件表达式**（又称三目运算符）`表达式1 ? 表达式2 : 表达式3` 来简化 `if else` 语句的代码。

```
public class IfElseStatement2 {
    public static void main(String args[]) {
        int age = 15;
        System.out.println(age >= 18 ? "在中国你已经成年" :  "在中国你还未成年");
    }
}
```

3.1.3 if ... else if ... else 语句

`if` 语句可以结合 `else if` 来实现更复杂的程序分支结构：

```
if (条件1) {
    // 如果条件1成立，执行此处代码
} else if (条件2) {
    // 如果条件1不成立，并且条件2成立，执行此处代码
} else {
    // 如果条件1、条件2都不成立，执行此处代码
}
```

例子：

`if` 语句可以搭配任意多数量的 `else if` 语句使用，但是只能有一个 `else`。

```
// 根据给定分数向屏幕打印评级
public class IfElseIfStatement {
      public static void main(String args[]) {
        int score = 70;
        if (score >= 90) {
            System.out.println("优秀");
        } else if(score >= 70) {
            System.out.println("良好");
        } else if (score >= 60) {
            System.out.println("及格");
        } else {
            System.out.println("不及格");
        }
    }
}
```

运行结果：

```
良好
```

3.1.4 嵌套 if ... else 语句

你也可以在另一个 `if` 或者 `else if` 语句中使用 `if` 或者 `else if` 语句:

```
if(条件1){
   // 如果条件1为真，执行这里的语句
   if(条件2){
      ////如果条件2为真，执行这里的语句
   }
}
```

例子：

```
public class IfElseStatement1 {
    public static void main(String[] args) {
        // 初始化整型变量age，值为25
        int age = 25;
        int sex = 1;  // 此处用sex变量表示性别，1：男  2：女
        if(age >= 20) {
```

```
            System.out.println("你成年了！");
            if(sex == 2) {
                System.out.println("并且到了法定的结婚年龄");
            }
            if(sex == 1 && age >= 22) {
                System.out.println("并且到了法定的结婚年龄");
            }
        } else {
            System.out.println("在中国你还未成年");
        }
    }
}
```

运行结果：

```
你成年了！
并且到了法定的结婚年龄
```

3.1.5 switch 语句

`switch` 条件语句可以理解为简写版的多个 `if .. else` 语句。`switch` 语句的语法如下：

```
switch (值) {
    case 值1:
       语句1.1
       ...
       语句n.1
       break;
    case 值2:
        语句2.1
        ...
        语句2.n
        break;
    default:
        语句n.1
        ...
        语句n.n
}
```

switch case 语句有如下规则：

- switch 语句中的变量类型可以是： `byte`、`short`、`int`、 `char` 或者 `String`；
- switch 语句可以拥有多个 case 语句。每个 case 后面跟一个要比较的值和冒号；
- case 语句中的值的数据类型必须与变量的数据类型相同，而且只能是常量或者字面常量；
- 当变量的值与 case 语句的值相等时，那么 case 语句之后的语句开始执行，直到 break 语句出现才会跳出 switch 语句；
- 当遇到 break 语句时，switch 语句终止。程序跳转到 switch 语句后面的语句执行。case 语句不一定要包含 break 语句。如果没有 break 语句出现，程序会继续执行下一条 case 语句，直到出现 break 语句；

- switch 语句可以包含一个 default 分支，该分支一般是 switch 语句的最后一个分支（可以在任何位置，但建议在最后一个）。**default 在没有 case 语句的值和变量值相等的时候执行。default 分支不需要 break 语句。**

例子：

```
public class SwitchStatement1 {
    public static void main(String args[]) {
        int i = 2;
        switch (i) {
          case 1:
              // i 的值不等于1，所以不执行此处代码
              System.out.println("i的值为1");
                break;
          case 2:
              // i 的值等于2，所以执行此处代码
              System.out.println("i的值为2");
              break;
          default:
              // case 2 分支已执行并break，所以此处代码不会执行
              System.out.println("i的值既不等于1，也不等于2");
        }
    }
}
```

从 JDK8 开始，`switch` 语句可以与 `String` 值一起使用：

```
public class SwitchStatement3 {
   public static void main(String args[]) {
      String day = "TUESDAY";
      switch (day) {
         case "Wednesday" :
            System.out.println("星期一");
            break;
         case "TUESDAY" :
            System.out.println("星期二");
            break;
         case "WEDNESDAY" :
            System.out.println("星期三");
            break;
         case "THURSDAY" :
            System.out.println("星期四");
            break;
         case "FRIDAY" :
            System.out.println("星期五");
            break;
         case "SATURDAY" :
            System.out.println("星期六");
            break;
         case "SUNDAY" :
            System.out.println("星期天");
      }
   }
}
```

运行结果：

```
星期二
```

在 Java 中，条件语句主要有 `if` 语句和 `switch` 语句两种。在实际的编码中，条件语句非常常用，要根据合适的场景选择使用，例如对于多个 `==` 判断的情况下，使用 `switch` 语句就更加清晰。而对于复杂的条件表达式，选择 `if` 语句就更适合。

3.2 Java 循环语句

3.2.1 while 循环

`while`循环是最简单的循环形式。

```
while (条件) {
    // 循环体，条件成立时执行
    ...
}
// 循环完成后执行
```

`while`循环在每次循环开始前，首先会判断条件是否成立，如果计算结果为`true`， 就会执行循环体内部语句。如果计算结果为`false`，会跳出循环，执行后续代码。

例子：

```
public class WhileLoop2 {
    public static void main(String[] args) {
        int sum = 0; // 累加和
        int i = 1;
        while (i <= 100) {
            sum = sum + i; // 使sum和i相加，并将值再次赋值给sum
            i ++; // i自增1
        }
        System.out.println("1到100的累加和为：" + sum);
    }
}
```

运行结果：

```
1到100的累加和为：5050
```

3.2.2 do while 循环

`do while`循环的功能与`while`循环类似，不同点在于：`while`循环是先判断条件，再执行循环体；而`do while`循环则是先执行循环体，再判断条件，如果条件成立继续执行循环，条件不成立则终止循环。

```
do {
    // 循环体
} while (条件);
```

无论条件成立与否，`do while`循环都至少执行一次。而`while`循环可能一次都不会执行。

例子：

```
public class DoWhileLoop {
    public static void main(String[] args) {
        int sum = 0; // 累加和
        int i = 1;
        do {
            sum = sum + i; // 使sum和i相加，并将值再次赋值给sum
            i ++; // i自增1
        } while (i <= 100);
        System.out.println("1到100的累加和为：" + sum);
    }
}
```

运行结果：

```
1到100的累加和为：5050
```

3.2.3 for 循环

`for`循环是一种特殊的`while`循环，也是被使用最广泛的循环。它使用**计数器**来实现循环。在关键字`for`后面的括号中，会有三个语句，第一个语句是**变量声明语句**，允许声明一个或多个**整型变量**；第二个语句是条件，条件的检测方式与`while`循环相同（每次循环开始前判断条件成立与否）；第三个语句是迭代语句，虽然可以是任何语句，但该语句通常用于递增或递减变量。

```
for (变量声明; 条件; 迭代语句) {
    // 循环体
}
```

例子：

```
public class ForLoop1 {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i ++) {
            System.out.println("Java is grate !");
        }
    }
}
```

3.2.4 break 和 continue 关键字

break

`break`关键字可以出现在一个循环中，用以**跳出当前循环。**

```
public class BreakKeywords1 {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i ++) {
            if(i == 3) { // 条件语句
                System.out.println("当i等于3时，终止循环！");
                break;
            }
            System.out.println(i);
        }
    }
}
```

运行结果：

```
1
2
当i等于3时，终止循环！
```

上面是一段循环打印数字的代码，当循环到达第3次迭代时，执行条件分支中的语句，将终止循环。注意：`break`语句通常配合`if`语句使用。

对于**多层循环**的情况，`break`语句只能终止它**自己所在的那一层循环**，并**不会影响**到外层循环。

```
public class BreakKeywords2 {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i ++) {
            // 打印i的值
            System.out.println(i);
            for (int j = 1; j<=5; j ++) {
                if(j == 3) { // 条件语句
                    System.out.println("当j等于3时，终止循环！");
                    break;
                }
                // 打印j的值
                System.out.println(j);
            }
        }
    }
}
```

运行结果：

```
1
1
2
当j等于3时，终止循环！
2
1
```

```
2
当j等于3时，终止循环！
3
1
2
当j等于3时，终止循环！
4
1
2
当j等于3时，终止循环！
5
1
2
当j等于3时，终止循环！
```

上面的代码有两个`for`循环嵌套，`break`语句位于内层循环，所以当表达式`j == 3`成立时，会跳出内层循环，进而继续执行外层循环。

continue

`continue`关键字可以跳过**当次循环**，继续执行下一次循环。

```
public class ContinueKeywords {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i ++) {
            if(i == 3) {
                System.out.println("当i等于3时，跳过~");
                continue;
            }
            System.out.println(i);
        }
    }
}
```

运行结果：

```
1
2
当i等于3时，跳过~
4
5
```

在多层嵌套的循环中，`continue`语句只能跳过它自己所在位置的循环。

3.3 Java 字符串

字符串(`string`)是由零个或多个字符组成的有限序列。简单来说，字符串就是一串字符。

字符串类型`String`是引用类型.

3.3.1 创建字符串

```
public class StringTest1 {
    public static void main(String[] args) {
        // 创建一个空字符串
        String str1 = "";
        // 创建一个只包含一个字符的字符串
        String str2 = "q";
        // 创建包含多个字符的字符串
        String str3 = "java is great!";
        // 依次打印它们
        System.out.println(str1);
        System.out.println(str2);
        System.out.println(str3);
    }
}
```

运行结果：

```

q
java is great!
```

字符串**可以声明为空**，因此第一行将打印一个空行。

3.3.2 创建多行字符串

自 **Java 13** 以后，我们可以使用三个引号来表示一个多行字符串，被称为“文本块”。文本块常用来表示多行的或大段的文字。

```
public class StringTest3 {
    public static void main(String[] args) {
        String str = """
            Java 很棒!
            PHP 很棒！！
            GO 更棒！！！""";
        System.out.println(str);
    }
}
```

运行结果：

```
Java 很棒!
PHP 很棒！！
GO 更棒！！！
```

这里需要注意的是，文本块中起始的**三引号后面要另起一行**.

3.2.3 字符串的连接

我们可以使用加法运算符`+`将字符串和任意其他的数据类型进行连接操作。

可以将两个或多个字符串连接在一起，例如：

```
public class StringJoinTest1 {
    public static void main(String[] args) {
        // 定义两个字符串
        String str1 = "Hello";
        String str2 = "Java";
        // 将字符串str1连接一个空格，再连接str2，最后再连接一个感叹号
        String str3 = str1 + " " + str2 + "! ";
        System.out.println(str3);
    }
}
```

运行结果：

```
Hello Java!
```

3.2.4 字符串的不可变性

字符串有一个重要特性：不可变性。也就是说，字符串一经创建便无法修改。

```
public class StringTest4 {
    public static void main(String[] args) {
        // 创建一个字符串 str
        String str = "hello java!";
        // 对 str 重新赋值
        str = "Java is great!";
        System.out.println(str);
    }
}
```

运行结果：

```
Java is great!
```

上述例子中，变量`str`没有修改，而是变量`str`的“**指向**”被修改了。

当对`str`重新赋值时，在内存中先创建了字符串"Java is great"，然后改变了变量`str`的指向，原来的"hello java!"并没有发生改变。

3.4 Java 数组

数组是**相同类型的数据**按照**顺序**组成的一种**引用数据类型。**

它在实际的编码中并不常用，这部分内容了解即可。实际项目开发中常用的为**集合**(List, Map等)，这一部分放在下一节讲。

3.4.1 数组声明和初始化

- **中括号跟在数据类型后**：`DataType[] arrayName;`
- **中括号跟在数组名称后**：`DataType arrayName[];`

在实际的编码中，我们更常用的是第一种语法。

```
// 声明一个int类型的数组，名称为 firstIntArray:
int[] firstIntArray;  // 推荐写法
int firstIntArray[];

// 声明一个整型的数组：
int[] intArray;
// 创建数组，长度为10
intArray = new int[10];
```

声明时同时创建：

```
DataType[] arrayName = new DataType[数组长度];

// 声明时同时创建
int[] intArray = new int[10];
```

数组的下标从0开始，最后一个元素的下标为数组长度减1。**引用元素时，不能超过其下标最大值**

例如上面的intArray，不能用 **intArray[10]** 来获取元素，会抛出异常。

数组的操作：

使用 `数组名[下标]`

```
public class ArrayDemo {
    public static void main(String[] args) {
        // 初始化整型数组arr：
        int[] arr = {1, 2, 3};
        // 引用arr的第一个元素：
        int element1 = arr[0];
        System.out.println("数组arr中，第一个元素为：" + element1);

        // 修改下标为1的元素为4
        arr[1] = 4;
        System.out.println("数组arr中，第二个元素为：" + arr[1]);

        // 打印数组arr的长度：
        System.out.println("数组arr的长度为：" + arr.length);
    }
}
```

运行结果：

```
数组arr中，第一个元素为：1
数组arr中，第二个元素为：4
数组arr的长度为：3
```

3.4.2 数组的迭代

我们可以使用循环引用依次打印数组中的每个元素。

例子：

```
public class ArrayDemo5 {
    public static void main(String[] args) {
        // 初始化一个整型数组
        int[] arr = {1,2,3,4,5};
        // 使用for循环遍历数组arr
        for (int i = 0; i < arr.length; i++) {
            // 打印位置i的元素
            System.out.println("索引位置" + i + "的元素为：" + arr[i]);
        }
    }
}
```

运行结果：

```
索引位置0的元素为：1
索引位置1的元素为：2
索引位置2的元素为：3
索引位置3的元素为：4
索引位置4的元素为：5
```

增强 for 循环

也是`for each`循环

```
for(变量声明: 数组或列表) {
    // 循环体
}
```

例子：

```
public class ForEachLoop {
    public static void main(String[] args) {
        // 初始化字符串数组 words
        String[] words = {"Welcome ", "to ", "China"};
        for(String word: words) {
            System.out.print(word);
        }
    }
}
```

运行结果：

```
Welcome to China
```

`for each`循环的代码看起来更加清晰。但**增强`for`循环**无法指定遍历顺序，也无法获取数组的索引。

3.4.3 多维数组

在Java中，多维数组也是非常常用的数据结构，下面以二维数组为例进行介绍

二维数组有 3 种声明方式：

- **中括号跟在数据类型后**：`DataType[][] arrayName;`**（最推荐写法）**
- **中括号跟在数组名后**：`DataType arrayName[][];`
- **中括号一个在前，一个在后**：`DataType[] arranName[];`

创建

```
javaDataType[][] arrayName = new int[行数][列数];
```

```
// 声明并创建一个2行3列的数组
int[][] intArray = new int[2][3];

// 创建一个3行2列的二维数组并赋值
int[][] intArray = {{1,2}, {3,4}, {5,6}};
// 为第1行第1列的元素赋值：
intArray[0][0] = 10;
```

二维数组的迭代

一维数组使用单层`for`循环就可以遍历，二维数组的遍历需要使用双层嵌套`for`循环。

例子：

```
public class ArrayDemo7 {
    public static void main(String[] args) {
        // 初始化一个二维数组
        int[][] intArray = {{1,2,7}, {3,4}, {5,6}};
        // 遍历intArray
        for(int i = 0; i < intArray.length; i++) {
            for(int j = 0; j <intArray[i].length; j++) {
                // 打印索引位置[i][j]的元素：
                System.out.println((i+1) + "行" + (j+1) + "列的元素为：" +
intArray[i][j]);
            }
            // 打印一个空行
            System.out.println();
        }
    }
```

```
}
```

数组是**引用数据类型**。它在创建时，会在内存中开辟一个**连续的空间**；数组是**同一数据类型**的集合

4. Java 面向对象

4.1 Java 方法

在前面我们已经了解过方法的概念，Java 程序的入口`main()`就是一个方法。`System.out.println();`语句中`println()`也是一个方法。

在 Java 中，定义一个方法的语法为：

```
访问修饰符 返回类型 方法名(参数列表) {
    若干语句;
    return 方法返回值;
}
```

- **访问修饰符有 4 种情况**：`public`、`private`、`protected`，也可以省略（`default`）。由于涉及到后面的面向对象相关知识，本节统一使用`public`修饰方法；
- **返回类型**：可以是任何的数据类型或`void`，如果方法没有返回值，返回类型设置为`void`；
- **方法名**：方法名的命名规范和变量相同；
- **参数列表**：参数是变量的一种类型，参数变量的作用域在方法内部；
- **方法体**：方法内部的一些语句。当方法返回值为`void`时，可以省略`return`语句。

4.2 方法传值

4.2.1 基本类型参数

基**本类型参数的传递**，是调用方值的复制。双方各自的后续修改，互不影响。简单来讲，方法内部对参数变量的任何操作，都不会影响到方法外部的变量。

```
class Car {
    public void speedUp(int speed) {
        System.out.println("小汽车加速前，速度为：" + speed);
        speed ++;
        System.out.println("小汽车加速后，速度为：" + speed);
    }

    public static void main(String[] args) {
        // 定义小汽车初始速度变量
        int speed = 10;
        // 实例化Car类，创建一个car对象
        Car car = new Car();
        // 调用car对象下的speed方法
        car.speedUp(speed);
        // 打印调用方法后速度参数的值
        System.out.println("调用speedUp方法后，调用方的speed参数为：" + speed);
    }
}
```

运行结果：

```
小汽车加速前，速度为：10
小汽车加速后，速度为：11
调用speedUp方法后，调用方的speed参数为：10
```

4.2.2 引用类型参数

引用类型参数的传递，调用方的变量，和接收方的参数变量，地址指向的是同一个对象。双方任意一方对这个对象的修改，都会影响对方。

例子：

```
class NBATeam {
    // 替换第一个球员方法
    public void replaceFirstPlayer(String[] players, String playerName) {
        // 替换第一个球员
        System.out.println("将第一个球员替换：");
        players[0] = playerName;
    }

    public static void main(String[] args) {
        String[] players = {"詹姆斯", "科比", "杜兰特", "乔丹"};

        System.out.println("球队中现有球员：");
        for (String player : players) {
            System.out.println(player);
        }

        // 创建team对象并调用其替换球员方法
        NBATeam team = new NBATeam();
        team.replaceFirstPlayer(players, "皮蓬");

        System.out.println("替换后球员：");
        for (String player : players) {
            System.out.println(player);
        }
    }
}
```

运行结果：

```
球队中现有球员：
詹姆斯  科比    杜兰特  乔丹
将第一个球员替换：
替换后球员：
皮蓬    科比    杜兰特  乔丹
```

4.2.3 可变参数

```
参数类型... 参数名
```

例子：

```
class VariableParameter {
    public void sum(int... n) {
        int sum = 0;
        // 可以对可变参数进行迭代
        for (int i: n) {
            sum = sum + i;
        }
        System.out.println("sum=" + sum);
    }

    public static void main(String[] args) {
        // 创建对象
        VariableParameter variableParameter = new VariableParameter();
        // 调用方法，传递一个参数
        variableParameter.sum(1);
        // 调用方法，传递两个参数
        variableParameter.sum(2, 3);
        // 调用方法，传递三个参数
        variableParameter.sum(5, 6, 7);
    }
}
```

运行结果：

```
sum=1
sum=5
sum=18
```

上述实例中，在主方法中给`sum`方法传参时，可选择一个或多个参数传递。

4.3 方法重载

方法重载是指在一个类中定义多个**同名**的方法，但要求每个方法具有**不同的参数的类型**或**参数的个数**。调用重载方法时，Java 编译器能通过检查调用的方法的**参数类型**和个数选择一个恰当的方法。

4.3.1 自定义方法的重载

例如，在`Student`类中，有多个`study`方法：

```
public class Student {
    public void study() {
        System.out.println("同学真好学!");
    }

    public void study(String name) {
        System.out.println(name + "同学真好学!");
    }
```

```
    public void study(String name, int age) {
        System.out.println(name + "同学真好学!" + "他今年" + age + "岁了");
    }

    public static void main(String[] args) {
        // 实例化学生对象
        Student student = new Student();
        // 调用无参数方法
        student.study();
        // 调用单参数方法
        student.study("Colorful");
        // 调用双参数方法
        student.study("小雨", 20);
    }
}
```

运行结果：

```
同学真好学!
Colorful同学真好学!
小雨同学真好学!他今年20岁了
```

代码中的三个`study`都是重载方法。通常来说，方法重载的返回值类型都是相同的。

如果我们在`Student`类中再增加一个方法：

```
public String study() {
    return "学习Java语言";
}
```

注意，上述的方法不是重载方法，因为我们已经在`Student`类中定义了无参方法`study`。

判断一个方法是否是重载方法的原则：**方法名相同**，**参数类型**或**参数个数不同**。

方法重载提高了程序的兼容性和易用性，为方法提供了多种可能性。

4.4 Java类和对象

类是一个抽象的概念，可以将它理解成“模板”，可用于产生对象；而对象是一个具体的事物。

4.4.1 定义类

定义类的语法，如下定义一个学生类：

学生可以有姓名、性别、年龄等特征 除了这些，还可以有学习课程、发表文章等行为。有了这些特征和行为，我们就可以抽象出一个学生类：

```
public class Student {
    // 定义属性（特征）
    String name;  // 姓名

    String sex;  // 男 | 女

    String age;  // 年龄
```

```
    // 定义方法（行为）
    public void studyCourse() {
        System.out.println("学习课程");
    }

    public void postArticle() {
        System.out.println("发表文章");
    }

}
```

4.4.2 对象的实例化

在 Java 中，使用 `new` 关键字实例化对象，其语法为：

```
类名 对象名称 = new 类名();
```

按上面新建的类举例子：

```
Student student = new Student();
```

4.4.3 调用属性和方法

对象实例化后，可以调用其属性和方法，其语法为：

```
// 调用属性
对象名.属性名;
// 调用方法
对象名.方法名();
```

4.4.4 构造方法

在类中，可定义一个构造方法，也称为构造函数或构造器。它用于一些初始化操作，构造方法的语法为：

```
public 构造方法名(参数) {
    // 代码块
}
```

需要注意的是，与普通的自定义方法不同，**构造方法没有返回类型**，并且方法名要**与类名同名。**

另外，如果我们没有定义构造方法，系统会有一个默认的构造方法。换句话说：任何一个类都会有一个构造方法。

构造方法只能在对象的实例化时使用，也就是说，**构造方法只能配合 `new` 关键字使用。**

根据参数的有无，构造方法可分为两种：

- **无参构造方法**

  在 Student` 类中，我们可以定义一个无参构造方法：

```
// 定义构造方法，无返回值并且命名与类名相同
public Student() {
    // 执行输出语句
    System.out.println("无参构造方法执行了...");
}
```

使用 `new` 关键字初始化一个对象：

```
Student student = new Student();
```

运行结果：

```
无参构造方法执行了...
```

- **有参构造方法**

  当创建实例对象时，我们经常需要同时初始化这个实例的成员属性，例如：

```
Student student = new Student();
student.name = "小雨";
student.age = "18";
```

虽然这样能够给我们的成员属性进行赋值，但每次需要编写 2 行这样的代码。能不能在创建对象时，就对这些属性进行赋值呢？有参构造方法解决了这样的问题，我们可以在 `Student` 类内部定义一个有参构造方法：

```
// 有参构造方法
public Student(String studentName, String studentAge) {
    // 将参数变量赋值给实例变量
    name = studentName;
    age = studentAge;
}
```

构造函数中，我们将参数变量的值赋值给实例变量。使用 `new` 关键字调用构造方法：

```
Student student1 = new Student("小明", "18");
System.out.println("名字为：" + student1.name);
System.out.println("年龄为：" + student1.age);
```

运行结果：

```
名字为：小明
年龄为：18
```

需要特别注意的是，我们在上面两个有参构造方法中，成员属性的命名和参数变量的命名是不同的，如果参数列表中参数变量的命名和实例属性相同。将无法完成对实例属性的赋值，也就是说，下面的写法是错误的：

```
public Student(String name) {
    name = name;
}
```

运行结果：

```
名字为：null
```

4.4.5 this 关键字

为了解决上面无法使用与成员属性同名称的参数对成员属性进行赋值的问题，我们可以使用 `this` 关键字：

```
public Student(String name) {
    this.name = name;
}
```

在方法内部，`this` 关键字是当前对象的默认引用，简单说，**谁调用了它谁就是 `this`**。因此，通过 `this.field` 就可以访问当前实例的字段，解决实例变量和参数变量名称冲突的问题。

运行结果：

```
名字为：小帅
```

`this` 关键字可以解决实例变量和参数变量冲突的问题；`this` 关键字也可以调用同一类中其他的成员方法。

4.5 封装

提到面向对象的三大特征：**封装、继承、多态。**

类的基本作用就是封装代码。封装将类的一些特征和行为**隐藏**在类内部，不允许类外部直接访问。

封装可以被认为是一个**保护屏障**，防止该类的代码和数据被外部类定义的代码随机访问。

我们可以通过类提供的方法来实现对隐藏信息的操作和访问。**隐藏**了对象的信息，**留出**了访问的接口。

封装有两个特点：

1. **只能通过规定的方法访问数据；**
2. **隐藏类的实例细节，方便修改和实现。**

4.5.1 为什么需要封装

封装具有以下优点：

- 封装有利于提高类的内聚性，适当的封装可以让代码更容易理解与维护；
- 一些关键属性只允许类内部可以访问和修改，增强类的安全性；
- 隐藏实现细节，为调用方提供易于理解的接口；

4.5.2 实现封装

在 Java 语言中，如何实现封装呢？需要 3 个步骤。

1. 修改属性的可见性为`private`；
2. 创建公开的 getter 和 setter 方法，分别用于属性的读写；
3. 在 getter 和 setter 方法中，对属性的合法性进行判断。

我们来看一个 NBA 球员类`NBAPlayer`：

```
class NBAPlayer {
    // 姓名
    String name;
    // 年龄
    int age;
}
```

在**类内部**（即类名后面`{}`之间的区域）定义了成员属性`name`和`age`，我们知道，在类外部调用处可以对其属性进行修改：

```
NBAPlayer player = new NBAPlayer();
player.age = -1;
```

如下是实例代码：

```
public class NBAPlayer {
    // 姓名
    String name;
    // 年龄
    int age;

    public static void main(String[] args) {
        NBAPlayer player = new NBAPlayer();
        player.age = -1;
        System.out.println("球员年龄为：" + player.age);
    }
}
```

运行结果：

```
球员年龄为：-1
```

我们通过**对象名.属性名**的方式对`age`赋值为`-1`，显然，球员的年龄为`-1`是反常理的。

下面我们对`NBAPlayer`类进行封装。

1. 我们可以使用私有化访问控制符修饰类内部的属性，让其只在类的内部可以访问：

   ```
   // 用private修饰成员属性，限定只能在当前类内部可以访问
   private String name;
   private int age;
   ```

`private`关键字限定了其修饰的成员只能在类内部访问，这样之后就无法在类外部使用`player.age =-1`这样的赋值方式进行赋值了。

2. 创建**公开的**（public） `getter` 和 `setter`方法：

```
// 通常以get+属性名的方式命名 getter，返回对应的私有属性
public String getName() {
    return name;
}

// 通常以set+属性名的方式命名 setter，给对应属性进行赋值
public void setName(String name) {
    this.name = name;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}
```

`getter`就是取属性值，`setter`就是给属性赋值，这样在类的外部就可以通过调用其方法对属性进行操作了。

3. 对属性进行逻辑判断，以`age`属性的`setter`方法为例：

```
public void setAge(int age) {
    // 判断参数age的合法性
    if(age < 0) {
        this.age = 0;
    } else {
        this.age = age;
    }
}
```

在`setAge`方法中，我们将参数`age`小于 0 的情况进行了处理，如果小于 0，直接将`age`赋值为0。除了给默认值的方式，我们也可以抛出异常，提示调用方传参不合法。

在类外部对属性进行读写：

```
NBAPlayer player = new NBAPlayer();
// 对属性赋值：
player.setName("詹姆斯");
player.setAge(35);
// 获取属性：
System.out.println("姓名：" + player.getName());
System.out.println("年龄：" + player.getAge());
```

4.6 继承

继承是面向对象软件技术当中的一个概念。

如果一个类别 B “继承自” 另一个类别 A，就把这个 B 称为 “A 的子类”，而把 A 称为 “B 的父类别” 也可以称 “A 是 B 的超类”。

继承可以使得子类具有父类别的各种属性和方法，而不需要再次编写相同的代码。

特点：

- Java 中的继承为单一继承，也就是说，一个子类只能拥有一个父类，一个父类可以拥有多个子类。
- 所有的 Java 类都继承自 `Java.lang.Object`，所以 `Object` 是所有类的祖先类，除了 Object 外，所有类都必须有一个父类。我们在定义类的时候没有显示指定其父类，它默认就继承自 `Object` 类。
- 子类一旦继承父类，就会继承父类所有开放的特征，不能选择性地继承父类特征。
- 继承体现的是类与类之间的关系，这种关系是 `is a` 的关系，也就是说满足 `A is a B` 的关系就可以形成继承关系。

4.6.1 实现继承

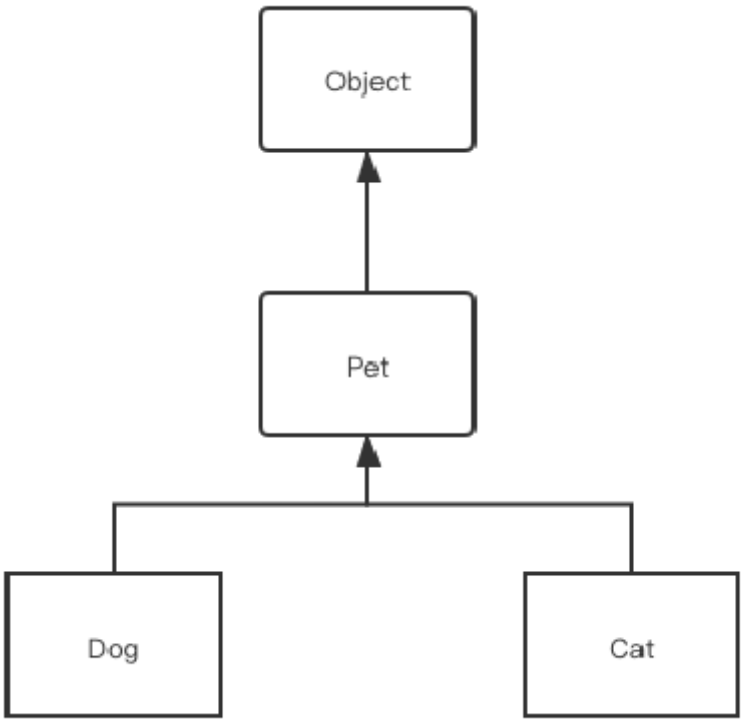
定义父类 `SuperClass`：

```
// 父类
class SuperClass {
    ...
}
```

在 Java 语言中，我们通过 `extends` 关键字声明一个类继承自另一个类：

```
// 子类
class SubClass extends SuperClass {
    ...
}
```

例如，宠物猫和宠物狗都是宠物，都有昵称、年龄等属性，都有吃东西、叫喊等行为。我们可以定义一个父类：宠物类。并且宠物猫和宠物狗类都继承宠物类，继承树形图如下:



代码实现：

```
public class Pet {
    private String name;  // 昵称
    private int age;      // 年龄

    // getter 和 setter
    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    // 吃东西
    public void eat() {
        System.out.println(this.getName() + "在吃东西");
    }

    // 叫喊
    public void shout() {
        System.out.println("宠物会叫");
    }
}
```

父类宠物类中 `name` 和 `age` 都是私有属性，而对应的 `getter`、`setter` 方法，`eat` 和 `shout` 方法都是公有方法。

宠物狗类：

```
public class Dog extends Pet {
    // 特有属性体重
    private float weight;

    // getter和setter
    public float getWeight() {
        return weight;
    }

    public void setWeight(float weight) {
        this.weight = weight;
    }

    // 特有的方法 run
    public void run() {
        System.out.println("胖成了" + this.getWeight() + "斤的狗子在奔跑");
    }
}
```

宠物狗类有一个自己特有的属性 `weight`，还有一个特有的方法 `run`。

宠物猫类：

```
public class Cat extends Pet {
    public void sleep() {
        System.out.println(this.getName() + "睡大觉zzz");
    }
}
```

宠物猫类有一个特有的方法 `sleep`，在方法中可以调用其父类 `Pet` 的 `getName` 方法。

调用类的方法：

```
// 实例化一个宠物狗
Dog dog = new Dog();
dog.setName("欢欢");
dog.setWeight(30f);
// 调用继承自父类的公有方法
dog.eat();
// 调用其特有方法
dog.run();

// 实例化一个宠物猫
Cat cat = new Cat();
cat.setName("豆豆");
// 调用继承自父类的公有方法
cat.eat();
```

```
// 调用其特有方法
cat.sleep();
```

运行结果：

```
欢欢在吃东西
胖成了30.0斤的狗子欢欢在奔跑
豆豆在吃东西
豆豆睡大觉zzz
```

4.6.2 方法重写

如果一个类从它的父类继承了一个方法，如果这个方法没有被标记为 `final` 或 `static`，就可以对这个方法进行重写。重写的好处是：能够定义特定于子类类型的行为，这意味着子类能够基于要求来实现父类的方法。

例子：

在上述父类 `Pet` 中有一个 `shout` 方法，我们知道小猫和小狗的叫声是不同的，此时可以使用**方法重写**，在 `Dog` 类和 `Cat` 类中重写 `shout` 方法。

Dog类：

```
class Dog extends Pet{
    // 重写 shout 方法
    public void shout() {
        System.out.println(this.getName() + "汪汪汪地叫~");
    }
}
```

Cat 类：

```
class Cat extends Pet{
    @Override  // 使用注解
    public void shout() {
        System.out.println(this.getName() + "喵喵喵地叫~");
    }
}
```

Tips：在要重写的方法上面，可以选择使用 `@Override` 注解，让编译器帮助检查是否进行了正确的重写。如果重写有误，编译器会提示错误。虽然这个注解不是必须的，但建议日常编码中，在所有要重写的方法上都加上 `@Override` 注解，这样可以避免我们由于马虎造成的错误。

可以使用对象实例调用其重写的方法：

```
// 实例化一个宠物狗
Dog dog = new Dog();
dog.setName("欢欢");
// 调用重写方法
dog.shout();

// 实例化一个宠物猫
Cat cat = new Cat();
cat.setName("豆豆");
// 调用重写方法
cat.shout();
```

运行结果：

```
欢欢汪汪汪地叫~
豆豆喵喵喵地叫~
```

方法重写和方法重载的区别

- Java 中的方法重写（`Overriding`）是说子类重新定义了父类的方法。方法重写必须有相同的**方法名，参数列表和返回类型**。覆盖者访问修饰符的限定**大于等于**父类方法。
- 而方法重载（`Overloading`）发生在**同一个类**里面两个或者是多个方法的方法名相同但是参数不同的情况。

4.6.3 访问修饰符

为了实现对类的封装和继承，Java 提供了访问控制机制。通过访问控制机制，类的设计者可以掩盖变量和方法来达到维护类自身状态的目的，而且还可以将另外一些需要暴露的变量和方法提供给别的类进行访问和修改。

Java 一共提供了 4 种访问修饰符：

1. **private**：私有的，只允许在本类中访问；
2. **protected**：受保护的，允许在同一个类、同一个包以及不同包的子类中访问；
3. **默认的**：允许在同一个类，同一个包中访问；
4. **public**：公共的，可以再任何地方访问。

访问控制修饰符	同一个类	同一个包	不同包的子类	不同包的非子类
private（私有的）	✓	×	×	×
default（默认的）	✓	✓	×	×
protected（受保护的）	✓	✓	✓	×
public（公共的）	✓	✓	✓	✓

4.6.4 super 关键字

`super` 是用在子类中的，目的是访问**直接父类**的变量或方法。注意：

- super 关键字只能调用父类的 `public` 以及 `protected` 成员；
- super 关键字可以用在子类构造方法中调用父类构造方法；
- super 关键字不能用于静态（`static`）方法中。

4.6.5 调用父类构造方法

父类的构造方法既不能被继承，也不能被重写。

可以使用 `super` 关键字，在子类构造方法中要调用父类的构造方法，语法为：

```
super(参数列表)
```

例子：

子类 `Dog` 的构造方法中调用父类构造方法

```
public Dog(String name) {
    super(name);
    System.out.println("小狗实例被创建了");
}
```

调用父类属性, 方法

子类中可以引用父类的成员变量，语法为：

```
super.成员变量名
super.方法名(参数列表)
```

super 与 this 的对比

`this` 关键字指向**当前类对象的引用**，它的使用场景为：

- 访问当前类的成员属性和成员方法；
- 访问当前类的构造方法；
- 不能在静态方法中使用。

`super` 关键字指向**父类对象的引用**，它的使用场景为：

- 访问父类的成员属性和成员方法；
- 访问父类的构造方法；
- 不能在静态方法中使用。

4.6.6 final 关键字

`final` 关键字可以作用于类、方法或变量，分别具有不同的含义。在使用时，必须将其放在变量类型或者方法返回之前，建议将其放在访问修饰符和 `static` 关键字之后，例如：

```
// 定义一个常量
public static final int MAX_NUM = 50;
```

final 作用于类

当 `final` 关键字用于类上面时，这个类不会被其他类继承:

```
final class FinalClass {
    public String name;
}

// final类不能被继承，编译会报错
public class SubClass extends FinalClass {
}
```

final 作用于方法

当父类中方法不希望被重写时，可以将该方法标记为 `final`：

```
class SuperClass {
    public final void finalMethod() {
            System.out.println("我是final方法");
    }
}

class SubClass extneds SuperClass {
    // 被父类标记为final的方法不允许被继承，编译会报错
    @Override
    public void finalMethod() {
    }
}
```

final 作用于变量

对于实例变量，可以使用 final 修饰，其修饰的变量在初始化后就不能修改：

```
class Cat {
    public final String name = "小花";
}
```

实例化 Cat 类，重新对 `name` 字段赋值：

```
Cat cat = new Cat();
cat.name = "小白";
```

4.7 Java 多态

多态是面向对象最重要的特性。

多态的实现条件、**什么是向上转型**以及**什么是向下转型**，

会使用`instanceof`运算符来检查对象引用是否是类型的实例。

多态顾名思义就是**多种形态**，是指对象能够有多种形态。在面向对象中最常用的多态性发生在当**父类引用指向子类对象**时。在面向对象编程中，所谓多态意指相同的消息给予不同的对象会引发不同的动作。换句话说：多态意味着允许不同类的对象对同一消息做出不同的响应。

4.7.1 实现多态

在 Java 中实现多态有 3 个必要条件：

1. **满足继承关系**
2. **要有重写**
3. **父类引用指向子类对象**

例子：

例如，有三个类`Pet`、`Dog`、`Cat`：
父类Pet：

```
class Pet {
    // 定义方法 eat
    public void eat() {
        System.out.println("宠物吃东西");
    }
}
```

子类Dog, Cat 继承Pet：

```
class Dog extends Pet { // 继承父类
    // 重写父类方法 eat
    public void eat() {
        System.out.println("狗狗吃狗粮");
    }
}
子类Cat继承Pet
class Cat extends Pet { // 继承父类
    // 重写父类方法 eat
    public void eat() {
        System.out.println("猫猫吃猫粮");
    }
}
```

在代码中，我们看到`Dog`和`Cat`类继承自`Pet`类，并且都重写了其`eat`方法。

现在已经满足了实现多态的前两个条件，那么如何**让父类引用指向子类对象**呢？我们在`main`方法中编写代码：

```
public void main(String[] args) {
    // 分别实例化三个对象，并且保持其类型为父类Pet
    Pet pet = new Pet();
    Pet dog = new Dog();
    Pet cat = new Cat();
    // 调用对象下方法
    pet.eat();
    dog.eat();
    cat.eat();
}
```

运行结果：

```
宠物吃东西
狗狗吃狗粮
猫猫吃猫粮
```

在代码中，`Pet dog = new Dog();`、`Pet cat = new Cat();`这两个语句，把`Dog`和`Cat`对象转换为`Pet`对象，这种把一个子类对象转型为父类对象的做法称为**向上转型**。父类引用指向了子类的实例。也就实现了多态。

4.7.2 向上转型

向上转型又称为自动转型、隐式转型。向上转型就是父类引用指向子类实例，也就是子类的对象可以赋值给父类对象。例如：

```
Pet dog = new Dog();
```

这个是因为`Dog`类继承自`Pet`类，它拥有父类`Pet`的全部功能，所以如果`Pet`类型的变量指向了其子类`Dog`的实例，是不会出现问题的。

向上转型实际上是把一个子类型安全地变成了更加**抽象**的父类型，由于所有类的根类都是`Object`，我们也把子类类型转换为`Object`类型：

```
Cat cat = new Cat();
Object o = cat;
```

4.7.3 向下转型

向上转型是父类引用指向子类实例，那么如何让**子类引用指向父类实例**呢？使用**向下转型**就可以实现。向下转型也被称为强制类型转换。例如：

```
// 为Cat类增加run方法
class Cat extends Pet { // 继承父类
    // 重写父类方法 eat
    public void eat() {
```

```
        System.out.println("猫猫吃猫粮");
    }

    public void run() {
        System.out.println("猫猫跑步");
    }

    public static void main(String[] args) {
        // 实例化子类
        Pet cat = new Cat();
        // 强制类型转换，只有转换为Cat对象后，才能调用其下面的run方法
        Cat catObj = (Cat)cat;
        catObj.run();
    }
}
```

运行结果：

```
猫猫跑步
```

我们为`Cat`类新增了一个`run`方法，此时我们无法通过`Pet`类型的`cat`实例调用到其下面特有的`run`方法，需要向下转型，通过`(Cat)cat`将`Pet`类型的对象强制转换为`Cat`类型，这个时候就可以调用`run`方法了。

使用向下转型的时候，要注意：**不能将父类对象转换为子类类型，也不能将兄弟类对象相互转换。**以下两种都是错误的做法：

```
// 实例化父类
Pet pet = new Pet();
// 将父类转换为子类
Cat cat = (Cat) pet;

// 实例化Dog类
Dog dog = new Dog();
// 兄弟类转换
Cat catObj = (Cat) dog;
```

不能将父类转换为子类，因为子类功能比父类多，多的功能无法凭空变出来。兄弟类之间不能转换，这就更容易理解了，兄弟类之间同样功能不尽相同，不同的功能也无法凭空变出来。

4.7.4 instanceof 运算符

`instanceof`运算符用来检查对象引用是否是类型的实例，或者这个类型的子类，并返回布尔值。如果是返回`true`，如果不是返回`false`。通常可以在运行时使用`instanceof`运算符指出某个对象是否满足一个特定类型的实例特征。其使用语法为：

```
<对象引用> instanceof 特定类型
```

例如，在向下转型之前，可以使用`instanceof`运算符判断，这样可以提高向下转型的安全性：

```
Pet pet = new Cat();
if (pet instanceof Cat) {
        // 将父类转换为子类
        Cat cat = (Cat) pet;
}
```

4.8 Java抽象类

4.8.1 概念和特点

在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。

值得注意的是，一个抽象类不能直接实例化，但类的其他功能依然存在；既然不能被实例化，那么它必须被继承才能被使用。

4.8.2 为什么需要抽象类

当某个父类只知道其子类**应该包含什么方法**，但不知道子类**如何实现这些方法**的时候，抽象类就派上用场了。使用抽象类还有一个好处，类的使用者在创建对象时，就知道他必须要使用某个具体子类，而不会误用抽象的父类，因此对于使用者来说，**有一个提示作用。**

例如，父类 `Pet` 被子类 `Cat` 和 `Dog` 继承，并且都重写了父类的 `eat` 方法：

```
class Pet {
    // 定义方法 eat
    public void eat() {
        System.out.println("宠物都会吃东西");
    }
}

class Dog extends Pet { // 继承父类
    // 重写父类方法 eat
    @Override
    public void eat() {
        System.out.println("狗狗吃狗粮");
    }
}

class Cat extends Pet { // 继承父类
    // 重写父类方法 eat
    @Override
    public void eat() {
        System.out.println("猫猫吃猫粮");
    }
}
```

小猫类、小狗类这些子类都重写了宠物类中的 `eat` 方法，我们知道每种宠物都有吃的行为，宠物表示了一个抽象的概念。那么宠物类的实例化和方法调用就没有了实际意义：

```
Pet pet = new Pet();
pet.eat();
```

我们知道了**抽象类不能被实例化**，此时可以将父类设定为抽象类，使用 `abstract` 关键字来声明抽象类，`abstract` 关键字必须放在 `class` 关键字前面：

```
// 声明抽象类
abstract class Pet {
    ...
}
```

如果你尝试实例化抽象类 `Pet`，编译器将会报错：

```
Pet.java:4: 错误: Pet是抽象的; 无法实例化
        new Pet();
        ^
1 个错误
```

使用抽象类，我们既可以通过父类和子类的继承关系，来限制子类的设计随意性，也可以避免父类的无意义实例化。

4.8.3 抽象方法

抽象类中可以包含**抽象方法**，它是没有具体实现的方法。换句话说，与普通方法不同的是，抽象方法没有用 `{}` 包含的方法体。

抽象类可以定义一个完整的编程接口，从而为子类提供实现该编程接口所需的所有方法的方法声明。抽象类可以只声明方法，而不关心这些方法的具体实现，而子类必须去实现这些方法。

上面我们将 `Pet` 父类改为了抽象类，其中包含了 `eat` 方法的具体实现，而实际上这个方法是不需要具体实现的，每种宠物都有各自的具体实现。此时，就可以将 `eat` 方法改为抽象方法：

```
abstract class Pet {
    abstract void eat();
}
```

我们可以看到抽象方法使用 `abstract` 关键字声明，它没有方法体，而直接使用 `;` 结尾。

子类必须实现父类中的抽象方法，假如 `Dog` 类继承了抽象类 `Pet`，但没有实现其抽象方法 `eat`：

```
class Dog extends Pet {
}
```

编译执行代码，将会报错：

```
Dog.java:1: 错误: Dog不是抽象的, 并且未覆盖Pet中的抽象方法eat()
public class Dog extends Pet {
       ^
1 个错误
```

上述报错中可知，如果我们不想在 `Dog` 中重写抽象方法 `eat()`，那么可以将 `Dog` 也改为抽象类：

```
abstract class Dog extends Pet {
}
```

抽象方法不能是 `final`、`static` 和 `native` 的；并且抽象方法不能是私有的，即不能用 `private` 修饰。因为抽象方法一定要被子类重写，私有方法、最终方法以及静态方法都不可以被重写，因此抽象方法不能使用 `private`、`final` 以及 `static` 关键字修饰。而 `native` 是本地方法，它与抽象方法不同的是，它把具体的实现移交给了本地系统的函数库，没有把实现交给子类，因此和 `abstract` 方法本身就是冲突的。

4.9 Java接口

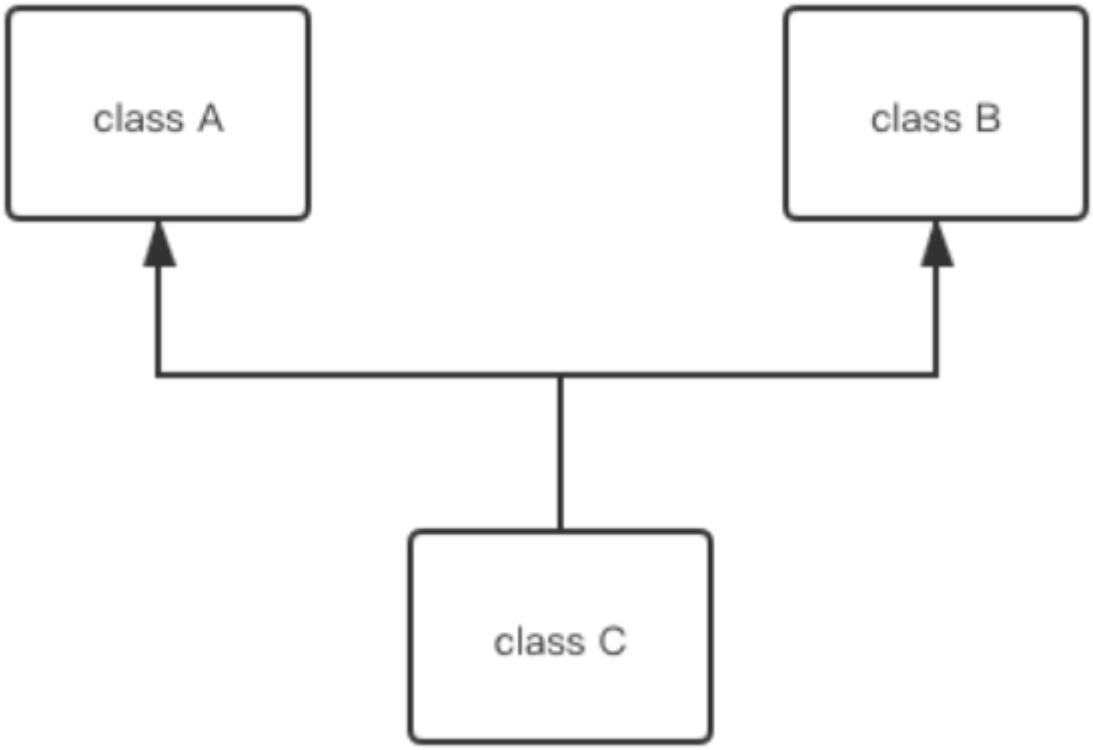
4.9.1 概念

> Java 接口是一系列方法的声明，是一些方法特征的集合，一个接口只有方法的特征没有方法的实现。

在 Java 中，被关键字 `interface` 修饰的 class 就是一个接口。接口定义了一个行为协议，可以由类层次结构中任何位置的任何类实现。接口中定义了一组抽象方法，都没有具体实现，实现该接口的类必须实现该接口中定义的所有抽象方法。

4.9.2 为什么需要接口

我们知道 Java 仅支持单继承，也就是说一个类只允许有一个直接父类，这样保证了数据的安全。Java 不支持下图所示的多继承：



接口就是为了解决 Java 单继承这个弊端而产生的，虽然一个类只能有一个直接父类，但是它可以实现多个接口，没有继承关系的类也可以实现相同的接口。继承和接口的双重设计既保持了类的数据安全也变相实现了多继承。

4.9.3 接口的定义和实现

1) 定义接口

使用 `interface` 关键字声明一个接口：

```
public interface Person {
    ...
}
```

接口声明需要两个元素：`interface` 关键字和接口名称，`public` 修饰符表示该接口可以在任何包的任何类中使用，如果为显示指定访问修饰符，则该接口只能被在同包中的类使用。

2) 接口主体

接口主体中，可以定义常量和方法声明：

```
public interface Person {
    final String NAME = "我是Person接口中的常量";
    void walk();
    void run();
}
```

上面的 `Person` 就是一个接口，这个接口定义了一个常量 `NAME` 和两个抽象方法 `walk()`、`run()`。

接口比抽象类更加“抽象”，它下面不能拥有具体实现的方法，必须全部都是抽象方法，所有的方法默认都是 `public abstract` 的，所以在接口主体中的方法，这两个修饰符无需显示指定。

接口除了**方法声明**外，还可以包含**常量声明**。在接口中定义的所有的常量默认都是 `public`，`static`，和 `final` 的。

接口中的成员声明不允许使用 `private` 和 `protected` 修饰符。

3) 实现接口

接口定义了一些行为协议，而实现接口的类要遵循这些协议。`implements` 关键字用于实现接口，一个类可以实现一个或多个接口，当要实现多个接口时，`implements` 关键字后面是该类要实现的以逗号分割的接口名列表。其语法为：

```
public class MyClass implements MyInterface1, MyInterface2 {
   ...
}
```

下面是实现了 `Person` 接口的 `Student` 类的示例代码：

```
public class Student implements Person {
    @Override
    public void walk() {
        // 打印接口中的常量
        System.out.println(Person.NAME);
        System.out.println("学生可以走路");
    }

    @Override
    public void run() {
        System.out.println("学生可以跑步");
    }
}
```

上述代码中，`Student` 类实现了 `Person` 接口。值得注意的是，可以使用**接口名。常量名**的方式调用接口中所声明的常量：

```
String name = Person.NAME;
```

4.9.4 接口继承

接口也是存在继承关系的。接口继承使用 `extends` 关键字。例如，声明两个接口 `MyInterface1` 和 `MyInterface2`，`MyInterface2` 继承自 `MyInterface1`：

```
// MyInterface1.java
public interface MyInterface1 {
    void abstractMethod1();
}

// MyInterface2.java
public interface MyInterface2 extends MyInterface1 {
    void abstractMethod2();
}
```

当一个类实现 `MyInterface2` 接口，将会实现该接口所继承的所有抽象方法：

```
// MyClass.java
public class MyClass implements MyInterface2 {
    @Override
    public void abstractMethod2() {
                ...
    }

    @Override
    public void abstractMethod1() {
                ...
    }
}
```

值得注意的是，一个接口可以继承多个父接口，接口名放在 `extends` 后面，以逗号分割，例如：

```
// MyInterface1.java
public interface MyInterface1 {
    void abstractMethod1();
}

// MyInterface2.java
public interface MyInterface2 {
    void abstractMethod2();
}

// MyInterface3.java
public interface MyInterface3 extends MyInterface1, MyInterface2 {
    void abstractMethod3();
}
```

补充一点，当一个实现类存在 `extends` 关键字，那么 `implements` 关键字应该放在其后：

```
public class MyClass extends SuperClass implements MyInterface {
   ...
}
```

4.9.5 默认方法和静态方法

从 **JDK 1.8** 开始，接口中可以定义默认方法和静态方法。与抽象方法不同，实现类可以不实现默认方法和类方法。

1) 默认方法

- **声明**

我们可以使用 `default` 关键字，在接口主题中实现**带方法体**的方法，例如：

```
public interface Person {
    void run();

    default void eat() {
        System.out.println("我是默认的吃方法");
    }
}
```

- **调用和重写**

在实现类中，可以不实现默认方法：

```
public class Student implements Person {
    @Override
    public void run() {
        System.out.println("学生可以跑步");
    }
}
```

我们也可以在实现类中重写默认方法，重写不需要 `default` 关键字：

```
public class Student implements Person {
    @Override
    public void run() {
        System.out.println("学生可以跑步");
    }

    // 重写默认方法
    @Override
    public void eat() {
        // 使用 接口名.super.方法名() 的方式调用接口中默认方法
        Person.super.eat();
        System.out.println("学生吃东西");
    }
}
```

如果想要在实现类中调用接口的默认方法，可以使用**接口名.super. 方法名 ()** 的方式调用。这里的 **接口名.super** 就是接口的引用。

- **使用场景**

当一个方法不需要所有实现类都进行实现，可以在接口中声明该方法为默认方法；使用默认方法还有一个好处，当接口新增方法时，将方法设定为默认方法，只在需要实现该方法的类中重写它，而不需要在所有实现类中实现。

2) 静态方法

- **声明**

使用 `static` 关键字在接口中声明静态方法，例如：

```
public interface Person {
    void walk();
    // 声明静态方法
    static void sayHello() {
        System.out.println("Hello Test!");
    }
}
```

- **调用**

类中的静态方法只能被子类继承而不能被重写，同样在实现类中，**静态方法不能被重写。**如果想要调用接口中的静态方法，只需使用 **接口名。类方法名** 的方式即可调用：

```
public class Student implements Person {
    @Override
    public void walk() {
        // 调用接口中的类方法
        Person.sayHello();
        System.out.println("学生会走路");
    }
}
```

---

1. 接口的方法默认是 public ，所有方法在接口中不能有实现（Java 8 开始接口方法可以有默认实现），而抽象类可以有非抽象的方法；
2. 接口中除了 static 、final 变量，不能有其他变量，而抽象类可以；
3. 一个类可以实现多个接口，但只能实现一个抽象类。接口自己本身可以通过 extends 关键字扩展多个接口；
4. 接口方法默认修饰符是 public ，抽象方法可以有 public 、protected 和 default 这些修饰符（抽象方法就是为了被重写所以不能使用 private 关键字修饰！）；
5. 从设计层面来说，抽象是对类的抽象，是一种模板设计，而接口是对行为的抽象，是一种行为的规范。

4.9.6 多个接口中的重名成员解决方法

1) 多个接口存在重名默认方法

例如有两个接口 `MyInteface1.java` 和 `MyInterface2.java`，存在相同签名的默认方法：

```
public interface MyInterface1 {
    default void defaultMethod() {
        System.out.println("我是MyInterface1接口中的默认方法");
    }
}

public interface MyInterface2 {
    default void defaultMethod() {
        System.out.println("我是MyInterface2接口中的默认方法");
    }
}
```

当实现类实现两个接口时，同名的默认方法将会发生冲突，解决办法是**在实现类中重写这个默认方法**：

```
public class MyClass implements MyInterface1, MyInterface2 {
    public void defaultMethod() {
        System.out.println("我是重写的默认方法");
    }
}
```

还有一种情况：实现类所继承的父类中也存在与默认方法的同名方法，此时存在三个同名方法：

```
// 声明父类，并在父类中也定义同名方法
public class SuperClass {
    public void defaultMethod() {
        System.out.println("我是SuperClass中的defaultMethod()方法");
    }
}

// 实现类继承父类，并实现两个接口
public class MyClass extends SuperClass implements MyInterface1, MyInterface2 {
}
```

实例化 `MyClass` 类，调用其 `defaultMethod()` 方法：

```
MyClass myClass = new MyClass();
myClass.defaultMethod();
```

此时编译执行，不会报错：

```
我是SuperClass中的defaultMethod()方法
```

实际上，在没有重写的情况下，它执行了实现类的父类 `SuperClass` 的 `defaultMethod()` 方法。

2) 多个接口中存在重名常量

例如有两个接口，存在重名的常量：

```
public interface MyInterface1 {
    final int NUM = 100;
}

public interface MyInterface2 {
    final int NUM = 200;
}
```

此时在实现类中，我们可以使用**接口名。常量名**的方式分别调用：

```
public MyClass implements MyInterface1, MyInterface2 {
      System.out.println(MyInterface1.NUM);
      System.out.println(MyInterface2.NUM);
}
```

当实现类将入一个继承关系时：

```
class SuperClass {
    static int NUM = 300;
}

public MyClass extends SuperClass implements MyInterface1, MyInterface2 {
    System.out.println(NUM);
}
```

当父类中的属性或常量与接口中的常量同名时，子类无法分辨同名的 `NUM` 是哪一个。编译程序将会报错：

```
MyClass.java:4: 错误: 对NUM的引用不明确
        System.out.println(NUM);
                           ^
  SuperClass 中的变量 NUM 和 MyInterface1 中的变量 NUM 都匹配
1 个错误
```

此时只有在子类中声明 `NUM`，才可以通过编译：

```
public MyClass extends SuperClass implements MyInterface1, MyInterface2 {
    int NUM = 3;
    System.out.println(NUM);
}
```

4.9.7 小结

接口和抽象类的区别：

Java 的接口是为了解决其单继承的弊端而产生的，可以使用 `interface` 关键字来声明一个接口，接口内部不能有具体的方法实现。可以使用 `implements` 关键字来实现接口，一个接口可以继承多个父接口，接口名放在 `extends` 后面，以逗号分割。从 Java 8 开始，接口中可以定义默认方法和静态方法。

4.10 Java内部类

1. 概念

在 Java 语言中，可以将一个类定义在另一个类里面或者一个方法里面，我们把这样的类称为内部类。

与之对应的，包含内部类的类被称为外部类。请阅读下面的代码：

```
// 外部类 Car
public class Car {
    // 内部类 Engine
    class Engine {
        private String innerName = "发动机内部类";
    }
}
```

代码中，`Engine` 就是内部类，而 `Car` 就是外部类。

2. 分类

Java 中的内部类可以分为 4 种：**成员内部类、静态内部类、方法内部类和匿名内部类。**

2.1 成员内部类

- **定义**

成员内部类也称为普通内部类，它是最常见的内部类。可以将其看作外部类的一个成员。在成员内部类中无法声明静态成员。

如下代码中声明了一个成员内部类：

```
// 外部类 Car
public class Car {
    // 内部类 Engine
    private class Engine {
        private void run() {
            System.out.println("发动机启动了！");
        }
    }
}
```

我们在外部类 `Car` 的内部定义了一个成员内部类 `Engine`，在 `Engine` 下面有一个 `run()` 方法，其功能是打印输出一行字符串：“发动机启动了！”。

另外，需要注意的是，与普通的 Java 类不同，含有内部类的类被编译器编译后，会生成两个独立的字节码文件：

```
Car$Engine.class
Car.class
```

内部类 `Engine` 会另外生成一个字节码文件，其文件名为：**外部类类名 $ 内部类类名.class。**

- **实例化**

内部类在外部使用时，无法直接实例化，需要借助外部类才能完成实例化操作。关于成员内部类的实例化，有 3 种方法：

1. 我们可以通过 `new 外部类().new 内部类()` 的方式获取内部类的实例对象：

实例演示

```
// 外部类 Car
public class Car {

    // 内部类 Engine
    private class Engine {
        private void run() {
            System.out.println("发动机启动了！");
        }
    }

    public static void main(String[] args) {
        // 1.实例化外部类后紧接着实例化内部类
        Engine engine = new Car().new Engine();
        // 2.调用内部类的方法
        engine.run();
    }
}
```

运行结果：

```
发动机启动了！
```

1. 我们可通过先实例化外部类、再实例化内部类的方法获取内部类的对象实例：

```
public static void main(String[] args) {
    // 1.实例化外部类
    Car car = new Car();
    // 2.通过外部类实例对象再实例化内部类
    Engine engine = car.new Engine();
    // 3.调用内部类的方法
    engine.run();
}
```

编译执行，成功调用了内部类的 run () 方法：

```
$javac Car.java
java Car
发动机启动了!
```

1. 我们也可以在外部类中定义一个获取内部类的方法 `getEngine()`，然后通过外部类的实例对象调用这个方法来获取内部类的实例：

实例演示

```
// 外部类 Car
public class Car {

    // 获取内部类实例的方法
    public Engine getEngine() {
        return new Engine();
    }

    // 内部类 Engine
    private class Engine {
        private void run() {
            System.out.println("发动机启动了！");
        }
    }

    public static void main(String[] args) {
        // 1.实例化外部类
        Car car = new Car();
        // 2.调用实例方法getEngine(),获取内部类实例
        Engine engine = car.getEngine();
        // 3.调用内部类的方法
        engine.run();
    }
}
```

运行结果：

```
发动机启动了!
```

这种设计在是非常常见的，同样可以成功调用执行 `run()` 方法。

- **成员的访问**

成员内部类可以直接访问外部类的成员，例如，可以在内部类的中访问外部类的成员属性：

实例演示

```
// 外部类 Car
public class Car {

    String name;

    public Engine getEngine() {
        return new Engine();
    }
```

```
    // 内部类 Engine
    private class Engine {
        // 发动机的起动方法
        private void run() {
            System.out.println(name + "的发动机启动了！");
        }
    }

    public static void main(String[] args) {
        // 实例化外部类
        Car car = new Car();
        // 为实例属性赋值
        car.name = "大奔";
        // 获取内部类实例
        Engine engine = car.getEngine();
        // 调用内部类的方法
        engine.run();
    }
}
```

观察 `Engine` 的 `run()` 方法，调用了外部类的成员属性 `name`，我们在主方法实例化 `Car` 后，已经为属性 `name` 赋值。

运行结果：

```
大奔的发动机启动了！
```

相同的，除了成员属性，成员方法也可以自由访问。这里不再赘述。

还存在一个同名成员的问题：如果内部类中也存在一个同名成员，那么优先访问内部类的成员。可理解为就近原则。

这种情况下如果依然希望访问外部类的属性，可以使用 `外部类名.this.成员` 的方式，例如：

实例演示

```
// 外部类 Car
public class Car {

    String name;

    public Engine getEngine() {
        return new Engine();
    }
    // 汽车的跑动方法
    public void run(String name) {
        System.out.println(name + "跑起来了！");
    }

    // 内部类 Engine
    private class Engine {
        private String name = "引擎";
        // 发动机的起动方法
```

```
        private void run() {
            System.out.println("Engine中的成员属性name=" + name);
            System.out.println(Car.this.name + "的发动机启动了！");
            Car.this.run(Car.this.name);
        }
    }

    public static void main(String[] args) {
        // 实例化外部类
        Car car = new Car();
        // 为实例属性赋值
        car.name = "大奔";
        // 获取内部类实例
        Engine engine = car.getEngine();
        // 调用内部类的方法
        engine.run();
    }
}
```

运行结果：

```
Engine中的成员属性name=引擎
大奔的发动机启动了！
大奔跑起来了！
```

请观察内部类 `run()` 方法中的语句：第一行语句调用了内部类自己的属性 `name`，而第二行调用了外部类 `Car` 的属性 `name`，第三行调用了外部类的方法 `run()`，并将外部类的属性 `name` 作为方法的参数。

2.2 静态内部类

- **定义**

静态内部类也称为嵌套类，是使用 `static` 关键字修饰的内部类。如下代码中定义了一个静态内部类：

```
public class Car1 {
    // 静态内部类
    static class Engine {
        public void run() {
            System.out.println("我是静态内部类的run()方法");
            System.out.println("发动机启动了");
        }
    }
}
```

- **实例化**

静态内部类的实例化，可以不依赖外部类的对象直接创建。我们在主方法中可以这样写：

```
// 直接创建静态内部类对象
Engine engine = new Engine();
// 调用对象下run()方法
engine.run();
```

运行结果：

```
我是静态内部类的run()方法
发动机启动
```

- **成员的访问**

在静态内部类中，只能直接访问外部类的静态成员。例如：

实例演示

```
public class Car1 {

    String brand = "宝马";

    static String name = "外部类的静态属性name";

    // 静态内部类
    static class Engine {
        public void run() {
            System.out.println(name);
        }
    }

    public static void main(String[] args) {
        Engine engine = new Engine();
        engine.run();
    }
}
```

在 `run()` 方法中，打印的 `name` 属性就是外部类中所定义的静态属性 `name`。编译执行，将会输出：

```
外部类的静态属性name
```

对于内外部类存在同名属性的问题，同样遵循就近原则。这种情况下依然希望调用外部类的静态成员，可以使用 `外部类名.静态成员` 的方式来进行调用。这里不再一一举例。

如果想要访问外部类的非静态属性，可以通过对象的方式调用，例如在 `run()` 方法中调用 `Car1` 的实例属性 `brand`：

```
public void run() {
    // 实例化对象
    Car1 car1 = new Car1();
    System.out.println(car1.brand);
}
```

2.3 方法内部类

- **定义**

方法内部类，是定义在方法中的内部类，也称局部内部类。

如下是方法内部类的代码：

实例演示

```
public class Car2 {

    // 外部类的run()方法
    public void run() {
        class Engine {
            public void run() {
                System.out.println("方法内部类的run()方法");
                System.out.println("发动机启动了");
            }
        }
        // 在Car2.run()方法的内部实例化其方法内部类Engine
        Engine engine = new Engine();
        // 调用Engine的run()方法
        engine.run();
    }

    public static void main(String[] args) {
        Car2 car2 = new Car2();
        car2.run();
    }
}
```

运行结果：

```
方法内部类的run()方法
发动机启动了
```

如果我们想调用方法内部类的 `run()` 方法，必须在方法内对 `Engine` 类进行实例化，再去调用其 `run()` 方法，然后通过外部类调用自身方法的方式让内部类方法执行。

- **特点**

与局部变量相同，局部内部类也有以下特点：

- 方法内定义的局部内部类只能在方法内部使用；
- 方法内不能定义静态成员；
- 不能使用访问修饰符。

也就是说，`Car2.getEngine()` 方法中的 `Engine` 内部类只能在其方法内部使用；并且不能出现 `static` 关键字；也不能出现任何的访问修饰符，例如把方法内部类 `Engine` 声明为 `public` 是不合法的。

2.4 匿名内部类

- **定义**

匿名内部类就是没有名字的内部类。使用匿名内部类，通常令其实现一个抽象类或接口。请阅读如下代码：

实例演示

```
// 定义一个交通工具抽象父类，里面只有一个run()方法
public abstract class Transport {
    public void run() {
        System.out.println("交通工具run()方法");
    }

    public static void main(String[] args) {
        // 此处为匿名内部类，将对象的定义和实例化放到了一起
        Transport car = new Transport() {
            // 实现抽象父类的run()方法
            @Override
            public void run() {
                System.out.println("汽车跑");
            }
        };
        // 调用其方法
        car.run();

        Transport airPlain = new Transport() {
            // 实现抽象父类的run()方法
            @Override
            public void run() {
                System.out.println("飞机飞");
            }
        };
        airPlain.run();

    }
}
```

运行结果：

```
汽车跑
飞机飞
```

上述代码中的抽象父类中有一个方法 `run()`，其子类必须实现，我们使用匿名内部类的方式将子类的定义和对象的实例化放到了一起，通过观察我们可以看出，代码中定义了两个匿名内部类，并且分别进行了对象的实例化，分别为 `car` 和 `airPlain`，然后成功调用了其实现的成员方法 `run()`。

- **特点**

- 含有匿名内部类的类被编译之后，匿名内部类会单独生成一个字节码文件，文件名的命名方式为：`外部类名称$数字.class`。例如，我们将上面含有两个匿名内部类的 `Transport.java` 编译，目录下将会生成三个字节码文件：

```
Transport$1.class
Transport$2.class
Transport.class
```

- 匿名内部类没有类型名称和实例对象名称；
- 匿名内部类可以继承父类也可以实现接口，但二者不可兼得；
- 匿名内部类无法使用访问修饰符、`static`、`abstract` 关键字修饰；
- 匿名内部类无法编写构造方法，因为它没有类名；
- 匿名内部类中不能出现静态成员。

- **使用场景**

由于匿名内部类没有名称，类的定义和实例化都放到了一起，这样可以简化代码的编写，但同时也让代码变得不易阅读。当我们在代码中只用到类的一个实例、方法只调用一次，可以使用匿名内部类。

3. 作用

3.1 封装性

内部类的成员通过外部类才能访问，对成员信息有更好的隐藏，因此内部类实现了更好的封装。

3.2 实现多继承

我们知道 Java 不支持多继承，而接口可以实现多继承的效果，但实现接口就必须实现里面所有的方法，有时候我们的需求只是实现其中某个方法，内部类就可以解决这些问题。

下面示例中的 `SubClass`，通过两个成员内部类分别继承 `SuperClass1` 和 `SuperClass2`，并重写了方法，实现了多继承：

```
// SuperClass1.java
public class SuperClass1 {
    public void method1() {
        System.out.println("The SuperClass1.method1");
    }
}

// SuperClass2.java
public class SuperClass2 {
    public void method2() {
        System.out.println("The SuperClass2.method2");
    }
}

// SubClass.java
public class SubClass {
```

```
    // 定义内部类1
    class InnerClass1 extends SuperClass1 {
        // 重写父类1方法
        @Override
        public void method1() {
            super.method1();
        }
    }

    // 定义内部类2
    class InnerClass2 extends SuperClass2 {
        // 重写父类2方法
        @Override
        public void method2() {
            super.method2();
        }
    }

    public static void main(String[] args) {
        // 实例化内部类1
        InnerClass1 innerClass1 = new SubClass().new InnerClass1();
        // 实例化内部类2
        InnerClass2 innerClass2 = new SubClass().new InnerClass2();
        // 分别调用内部类1、内部类2的方法
        innerClass1.method1();
        innerClass2.method2();
    }
}
```

编译执行 `SubClass.java`，屏幕将会打印：

```
$ javac SubClass.java
$ java SubClass
The SuperClass1.method1
The SuperClass1.method2
```

3.3 解决继承或实现接口时的方法同名问题

请阅读如下代码：

```
// One.java
public class One {
    public void test() {
    }
}
// Two.java
public interface Two {
    void test();
}
// Demo.java
public class Demo1 extends One implements Two {
    public void test() {
```

```
    }
}
```

此时，我们无法确定 `Demo1` 类中的 `test()` 方法是父类 `One` 中的 `test` 还是接口 `Two` 中的 `test`。这时我们可以使用内部类解决这个问题：

```
public class Demo2 extends One {

    // 重写父类方法
    @Override
    public void test() {
        System.out.println("在外部类实现了父类的test()方法");
    }

    // 定义内部类
    class InnerClass implements Two {
        // 重写接口方法
        @Override
        public void test() {
            System.out.println("在内部类实现了接口的test()方法");
        }
    }

    public static void main(String[] args) {
        // 实例化子类Demo2
        Demo2 demo2 = new Demo2();
        // 调用子类方法
        demo2.test();
        // 实例化子类Demo2的内部类
        InnerClass innerClass = demo2.new InnerClass();
        // 调用内部类方法
        innerClass.test();
    }
}
```

运行结果：

```
在外部类实现了父类的test()方法
在内部类实现了接口的test()方法
```

4.11 String & StringBuilder

1）String

1. String 对象的创建
2. 获取字符串长度
3. 字符串查找
4. 字符串截取
5. 字符串切割
6. 字符串大小写转换

7. 字符串比较

需要特别注意的是，在比较字符串内容是否相同时，必须使用`equals()`方法而不能使用`==`运算符。示例：

```
public class StringMethod6 {
    public static void main(String[] args) {
        // 用两种方法创建三个内容相同的字符串
        String str1 = "hello";
        String str2 = "hello";
        String str3 = new String("hello");
        System.out.println("使用equals()方法比较str1和str2的结果为：" +
str1.equals(str2));
        System.out.println("使用==运算符比较str1和str2的结果为：" + (str1 == str2));
         System.out.println("使用equals()方法比较str1和str3的结果为：" +
str1.equals(str3));
        System.out.println("使用==运算符比较str1和str3的结果为：" + (str1 == str3));
    }
}
```

运行结果：

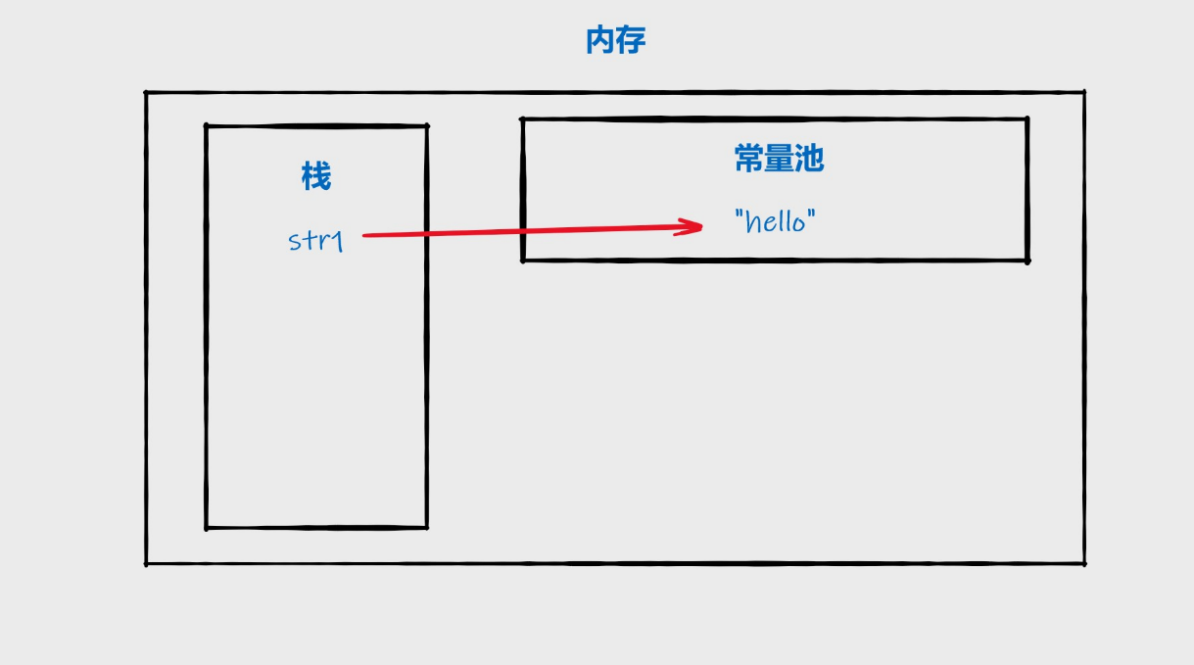
```
使用equals()方法比较str1和str2的结果为：true
使用==运算符比较str1和str2的结果为：true
使用equals()方法比较str1和str3的结果为：true
使用==运算符比较str1和str3的结果为：false
```

代码中三个字符串`str1`，`str2`和`str3`的内容都是`hello`，因此使用`equals()`方法对它们进行比较，其结果总是为`true`。

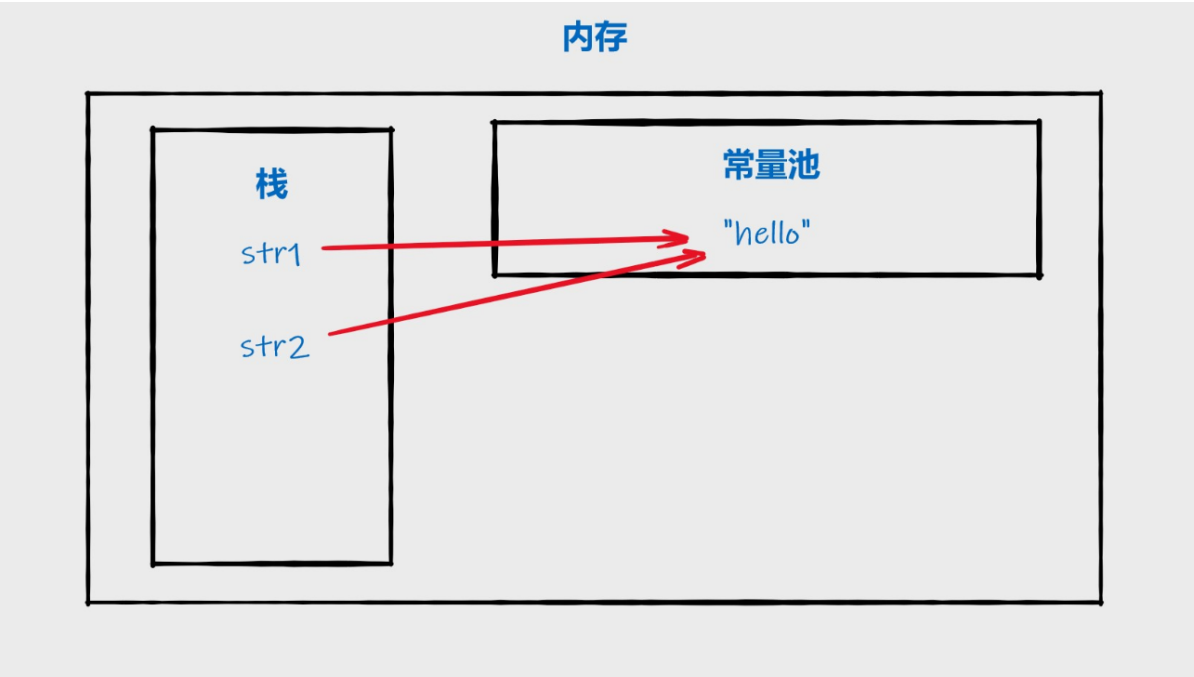
注意观察执行结果，其中使用`==`运算符比较`str1`和`str2`的结果为`true`，但使用`==`运算符比较的`str1`和`str3`的结果为`false`。这是因为`==`运算符比较的是两个变量的**地址**而不是内容。

要探究其原因，就要理解上述创建字符串的代码在计算机内存中是如何执行的。下面我们通过图解的形式来描述这三个变量是如何在内存中创建的。

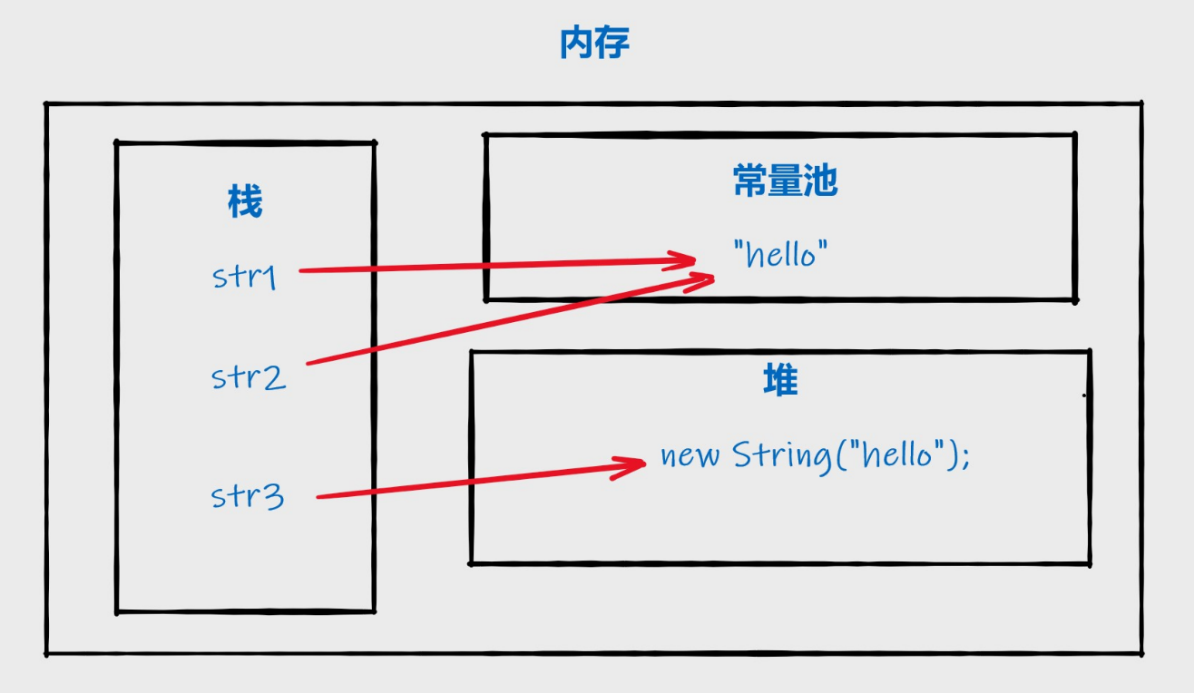
1）当执行`String str1 = "hello;"`语句时，会在内存的**栈空间**中创建一个`str1`，在**常量池**中创建一个"hello"，并将`str1`指向`hello`。



2）当执行`String str2 = "hello";`语句时，栈空间中会创建一个`str2`，由于其内容与`str1`相同，会指向常量池中的同一个对象。所以`str1`与`str2`指向的地址是相同的，这就是`==`运算符比较`str1`和`str2`的结果为`true`的原因。



3）当执行`String str3 = new String("hello");`语句时，使用了`new`关键字创建字符串对象，由于对象的实例化操作是在内存的**堆空间**进行的，此时会在栈空间创建一个`str3`，在堆空间实例化一个内容为`hello`的字符串对象，并将`str3`地址指向堆空间中的`hello`，这就是`==`运算符比较`str1`和`str3`的结果为`false`的原因。



2）StringBuilder

`StringBuilder` 与 `String` 不同，它具有**可变性**。相较 `String` 类不会产生大量无用数据，性能上会大大提高。

因此对于需要频繁操作字符串的场景，建议使用 `Stringbuilder` 类来代替 `String` 类。

`StringBuffer` 是 `StringBuilder` 的前身，在早期的 `Java` 版本中应用非常广泛，它是 `StringBuilder` 的**线程安全版本**但实现线程安全的代价是**执行效率的下降**。

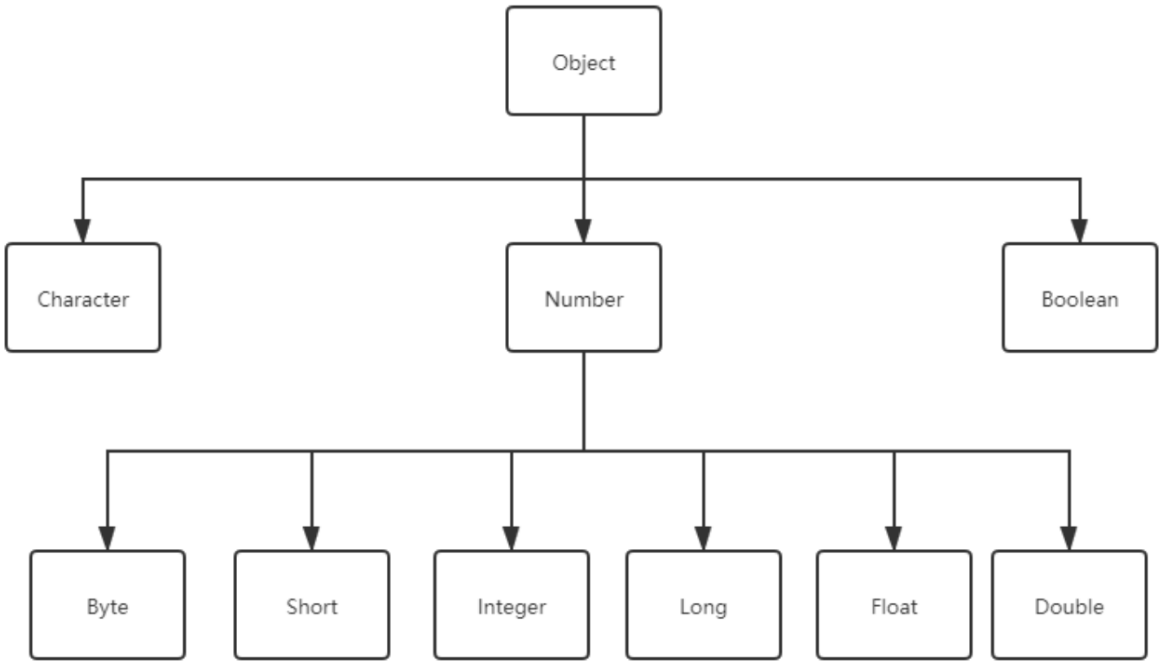
对比 `StringBuilder` 和 `StringBuffer` 的接口文档，它们的接口基本上完全一致。为了提升我们代码的执行效率，在如今的实际开发中 `StringBuffer` 并不常用。

4.12 包装类

1. 什么是包装类

Java 有 8 种基本数据类型，Java 中的每个基本类型都被包装成了一个类，这些类被称为包装类。

包装类可以分为 3 类：`Number`、`Character`、`Boolean`，包装类的架构图如下所示：



2. 为什么需要包装类

Java 数据类型被分为了基本数据类型和引用数据类型。

对于简单的运算，开发者可以直接使用基本数据类型。但对于需要对象化交互的场景（例如将基本数据类型存入集合中），就需要将基本数据类型封装成 Java 对象，这是因为基本数据类型不具备对象的一些特征，没有对象的属性和方法，也不能使用面向对象的编程思想来组织代码。出于这个原因，包装类就产生了。

包装类就是一个类，因此它有属性、方法，可以对象化交互。

3. 基本数据类型与包装类

下表列出了基本数据类型对应的包装类。这些包装类都位于 `java.lang` 包下，因此使用包装类时，我们不需要手动引入。

基本数据类型	对应的包装类
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

除了 `int` 对应的包装类名称为 `Integer` 以及 `char` 对应的包装类名称 `Character`，其他 `6` 种数据类型对应的包装类，命名都为其基本数据类型的首字母的大写。

4. 包装类常用方法

4.1 Number 类

1）构造方法

```
// 以 int 型变量作为参数创建 Integer 对象
Integer num = new Integer(3);
// 以 String 型变量作为参数创建 Integer 对象
Integer num = new Integer("8");
```

2） 常用方法

- `byte byteValue()`：以 byte 类型返回该 Integer 的值；
- `int compareTo(Integer anotherInteger)`：在数值上比较两个 Integer 对象。如果这两个值相等，则返回 0；如果调用对象的数值小于 anotherInteger 的数值，则返回负值；如果调用对象的数值大于 anotherInteger 的数值，则返回正值；
- `boolean equals(Object obj)`：比较此对象与指定对象是否相等；
- `int intValue()`：以 int 类型返回此 Integer 对象；
- `int shortValue()`：以 short 类型返回此 Integer 对象；
- `toString()`：返回一个表示该 Integer 值的 String 对象；
- `static Integer valueOf(String str)`：返回保存指定的 String 值的 Integer 对 象；
- `int parseInt(String str)`：返回包含在由 str 指定的字符串中的数字的等价整数值。

4.2 Boolean 类

Boolean 类将基本类型为 boolean 的值包装在一个对象中。一个 Boolean 类型的对象只包含一个类型为 boolean 的字段。此外，此类还为 boolean 和 String 的相互转换提供了许多方法，并提供了处理 boolean 时非常有用的其他一些常量和方法。

1）构造方法

- `Boolean(boolean value)`：创建一个表示 value 参数的 boolean 对象（很少使用）；
- `Boolean(String s)`：以 String 变量作为参数，创建 boolean 对象。此时，如果传入的字符串不为 null，且忽略大小写后的内容等于 “true”，则生成 Boolean 对象值为 true，反之为 false。（很少使用）。

2） 常用方法

- `boolean booleanValue()`：将 Boolean 对象的值以对应的 boolean 值返回；
- `boolean equals(Object obj)`：判断调用该方法的对象与 obj 是否相等，当且仅当参数不是 null，而且与调用该方法的对象一样都表示同一个 boolean 值的 Boolean 对象时， 才返回 true；
- `boolean parseBoolean(Sting)`：将字符串参数解析为 boolean 值；
- `String toString()`：返回表示该 boolean 值的 String 对象；
- `boolean valueOf(String s)`：返回一个用指定的字符串表示值的 boolean 值。

4.13 枚举类

1. 什么是枚举类

枚举是一个被命名的整型常数的集合。枚举在生活中非常常见，列举如下：

- 表示星期几：`SUNDAY`、`MONDAY`、`TUESTDAY`、`WEDNESDAY`、`THURSDAY`、`FRIDAY`、`SATURDAY`就是一个枚举；
- 性别：`MALE`（男）、`FEMALE`（女）也是一个枚举；
- 订单的状态：`PAIDED`（已付款）、`UNPAIDED`（未付款）、`FINISHED`（已完成），`CANCELED`（已取消）。

知道了什么是枚举，我们就很容易理解什么是枚举类了，简单来说，**枚举类就是一个可以表示枚举的类**，当一个类的对象只有**有限个、确定个**的时候，我们就可以定义一个枚举类来存放这些对象。

2. Enum 类

`java.lang.Enum`类 是 Java 语言枚举类型的公共基类，我们使用`enum`关键字定义的枚举类，是隐式继承自`Enum`类的，下面我们来看一下`Enum`类的常用方法：

- `values()`：返回枚举类型的对象数组。改方法可以很方便的遍历所有的枚举值；
- `valueOf()`：可以把一个字符串转换为对应的枚举类对象。要求字符串必须是枚举类对象的“名字”，如果不是，会抛出`IllegalArguementException`；
- `toString()`：返回当前枚举类对象常量的名称。

这 3 个方法使用起来比较简单，因此我们写在一个实例中，代码如下：

```
/**
 * @author colorful@TaleLin
 */
public class EnumDemo3 {

    public static void main(String[] args) {
        Sex male = Sex.MALE;
        System.out.println("调用 toString() 方法：");
        System.out.println(male.toString());

        System.out.println("调用 values() 方法：");
        Sex[] values = Sex.values();
        for (Sex value : values) {
            System.out.println(value);
        }

        System.out.println("调用 valueOf() 方法：");
        Sex male1 = Sex.valueOf("MALE");
        System.out.println(male1);
    }

}

/**
 * 使用 enum 关键字定义枚举类，默认继承自 Enum 类
```

```
 */
enum Sex {
    // 1.提供当前枚举类的多个对象，多个对象之间使用逗号分割，最后一个对象使用分号结尾
    MALE("男"),
    FEMALE("女"),
    UNKNOWN("保密");

    /**
     * 2.声明枚举类的属性
     */
    private final String sexName;

    /**
     * 3.编写构造方法，为属性赋值
     */
    Sex(String sexName) {
        this.sexName = sexName;
    }

    // 提供 getter 和 setter

    public String getSexName() {
        return sexName;
    }
}
```

运行结果：

```
调用 toString() 方法：
MALE
调用 values() 方法：
MALE
FEMALE
UNKNOWN
调用 valueOf() 方法：
MALE
```

值得注意的是，当调用`valuOf()`方法时，我们传递的对象的“名字”，在枚举类中并不存在，此时会抛出运行时异常：`IllegalArgumentException`，实例如下：

```
/**
 * @author colorful@TaleLin
 */
public class EnumDemo3 {

    public static void main(String[] args) {
        System.out.println("调用 valueOf() 方法：");
        Sex male1 = Sex.valueOf("MALE1");
        System.out.println(male1);
    }

}

/**
```

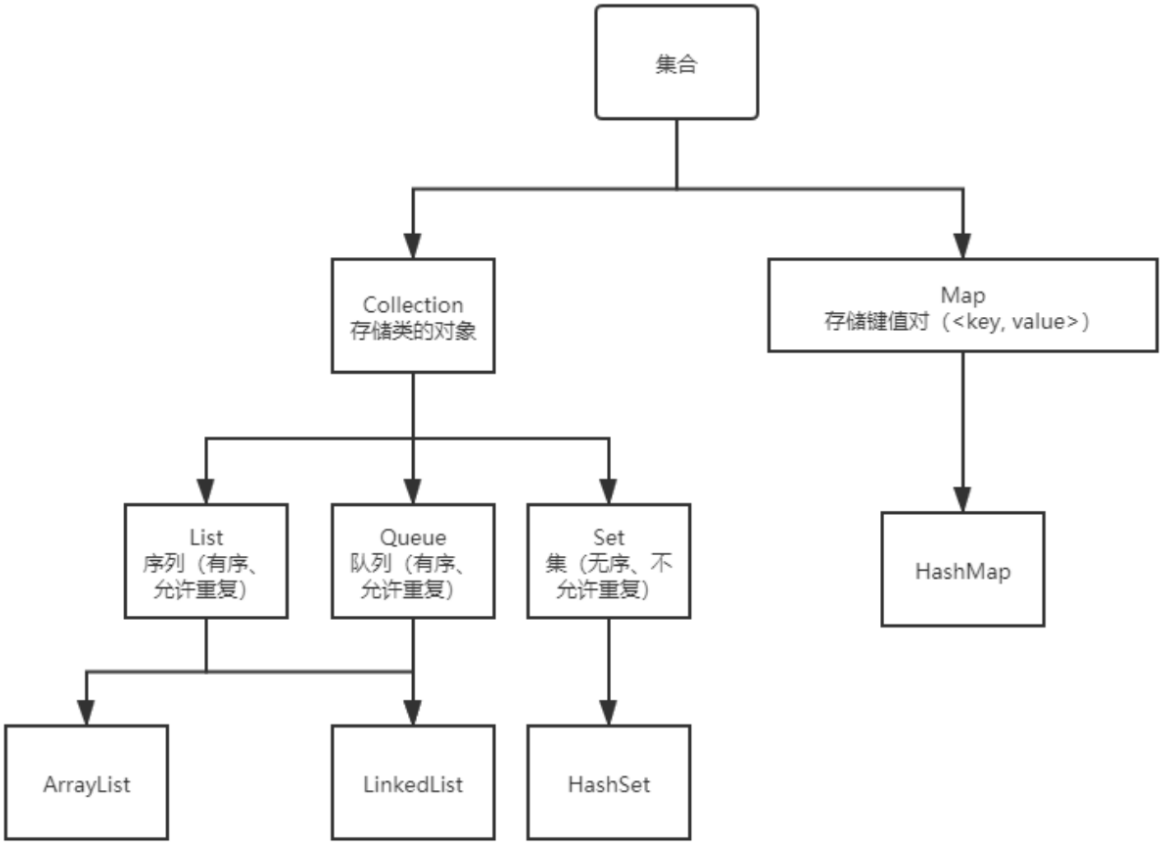
```
 * 使用 enum 关键字定义枚举类，默认继承自 Enum 类
 */
enum Sex {
    // 1.提供当前枚举类的多个对象，多个对象之间使用逗号分割，最后一个对象使用分号结尾
    MALE("男"),
    FEMALE("女"),
    UNKNOWN("保密");

    /**
     * 2.声明枚举类的属性
     */
    private final String sexName;

    /**
     * 3.编写构造方法，为属性赋值
     */
    Sex(String sexName) {
        this.sexName = sexName;
    }

    // 提供 getter 和 setter

    public String getSexName() {
        return sexName;
    }
}
```

运行结果：

```
调用 valueOf() 方法：
Exception in thread "main" java.lang.IllegalArgumentException: No enum constant
Sex.MALE1
    at java.base/java.lang.Enum.valueOf(Enum.java:273)
    at Sex.valueOf(EnumDemo3.java:17)
    at EnumDemo3.main(EnumDemo3.java:8)
```

4.14 集合

Java 中集合主要分为`java.util.Collection`和`java.util.Map`两大接口。



最下方的`ArrayList`、`LinkedList`、`HashSet`以及`HashMap`都是常用实现类。

1 Collection

`java.util.Collection`接口的实现可用于存储 Java 对象。例如，慕课网的所有学生可以视为一个`Collection`。

`Collection`又可以分为三个子接口，分别是：

1. `List`：序列，必须按照顺序保存元素，因此它是有序的，允许重复；
2. `Queue`：队列，按照排队规则来确定对象产生的顺序，有序，允许重复；
3. `Set`：集，不能重复。

2 Map

`java.util.Map`接口的实现可用于表示“键”（key）和“值”（value）对象之间的映射。一个映射表示一组“键”对象，其中每一个“键”对象都映射到一个“值”对象。因此可以通过键来查找值。例如，慕课网的每一个学生都有他自己的账户积分，这个关联关系可以用`Map`来表示。

3 常用的实现类

1. **ArrayList 实现类**
2. **LinkedList 实现类**
3. **HashSet 实现类**
4. **HashMap 实现类**

4.15 异常处理

1. Java 异常类架构

在 Java 中，通过 `Throwable` 及其子类来描述各种不同类型的异常。

1.1 Throwable 类

`Throwable` 位于 `java.lang` 包下，它是 Java 语言中所有错误（`Error`）和异常（`Exception`）的父类。

`Throwable` 包含了其线程创建时线程执行堆栈的快照，它提供了 `printStackTrace()` 等接口用于获取堆栈跟踪数据等信息。

主要方法：

- `fillInStackTrace`： 用当前的调用栈层次填充 `Throwable` 对象栈层次，添加到栈层次任何先前信息中；
- `getMessage`：返回关于发生的异常的详细信息。这个消息在 `Throwable` 类的构造函数中初始化了；
- `getCause`：返回一个 `Throwable` 对象代表异常原因；
- `getStackTrace`：返回一个包含堆栈层次的数组。下标为 0 的元素代表栈顶，最后一个元素代表方法调用堆栈的栈底；
- `printStackTrace`：打印 `toString()` 结果和栈层次到 `System.err`，即错误输出流。

1.2 Error 类

`Error` 是 `Throwable` 的一个直接子类，它可以指示合理的应用程序不应该尝试捕获的**严重问题**。这些错误在应用程序的控制和处理能力之外，编译器不会检查 `Error`，对于设计合理的应用程序来说，即使发生了错误，本质上也无法通过异常处理来解决其所引起的异常状况。

常见 `Error`：

- `AssertionError`：断言错误；
- `VirtualMachineError`：虚拟机错误；
- `UnsupportedClassVersionError`：Java 类版本错误；
- `OutOfMemoryError` ：内存溢出错误。

1.3 Exception 类

`Exception` 是 `Throwable` 的一个直接子类。它指示合理的应用程序可能希望捕获的条件。

`Exception` 又包括 `Unchecked Exception`（非检查异常）和 `Checked Exception`（检查异常）两大类别。

1）Unchecked Exception （非检查异常）

`Unchecked Exception` 是编译器不要求强制处理的异常，包含 `RuntimeException` 以及它的相关子类。我们编写代码时即使不去处理此类异常，程序还是会编译通过。

常见非检查异常：

- `NullPointerException`：空指针异常；
- `ArithmeticException`：算数异常；
- `ArrayIndexOutOfBoundsException`：数组下标越界异常；
- `ClassCastException`：类型转换异常。

2）Checked Exception（检查异常）

`Checked Exception` 是编译器要求必须处理的异常，除了 `RuntimeException` 以及它的子类，都是 `Checked Exception` 异常。我们在程序编写时就必须处理此类异常，否则程序无法编译通过。

常见检查异常：

- `IOException`：IO 异常
- `SQLException`：SQL 异常

2. 如何处理异常

在 Java 语言中，异常处理机制可以分为两部分：

1. **抛出异常**：当一个方法发生错误时，会创建一个异常对象，并交给运行时系统处理；
2. **捕获异常**：在方法抛出异常之后，运行时系统将转为寻找合适的异常处理器。

Java 通过 5 个关键字来实现异常处理，分别是：`throw`、`throws`、`try`、`catch`、`finally`。

异常总是先抛出，后捕获的。

2.1 throw

我们可以使用 `throw` 关键字来抛出异常，`throw` 关键字后面跟异常对象

```
public class ExceptionDemo2 {
    // 打印 a / b 的结果
    public static void divide(int a, int b) {
        if (b == 0) {
            // 抛出异常
            throw new ArithmeticException("除数不能为零");
        }
        System.out.println(a / b);
    }

    public static void main(String[] args) {
        // 调用 divide() 方法
        divide(2, 0);
    }
}
```

运行结果：

```
Exception in thread "main" java.lang.ArithmeticException: 除数不能为零
    at ExceptionDemo2.divide(ExceptionDemo2.java:5)
    at ExceptionDemo2.main(ExceptionDemo2.java:12)
```

2.2 throws

可以通过 `throws` 关键字声明方法要抛出何种类型的异常。如果一个方法可能会出现异常，但是没有能力处理这种异常，可以在方法声明处使用 `throws` 关键字来声明要抛出的异常。例如，汽车在运行时可能会出现故障，汽车本身没办法处理这个故障，那就让开车的人来处理。

`throws` 用在方法定义时声明该方法要抛出的异常类型，如下是伪代码：

```
public void demoMethod() throws Exception1, Exception2, ... ExceptionN {
    // 可能产生异常的代码
}
```

`throws` 后面跟的异常类型列表可以有一个也可以有多个，多个则以 `,` 分割。当方法产生异常列表中的异常时，将把异常抛向方法的调用方，由调用方处理。

throws 有如下使用规则：

1. 如果方法中全部是非检查异常（即 `Error`、`RuntimeException` 以及的子类），那么可以不使用 `throws` 关键字来声明要抛出的异常，编译器能够通过编译，但在运行时会被系统抛出；
2. 如果方法中可能出现检查异常，就必须使用 `throws` 声明将其抛出或使用 `try catch` 捕获异常，否则将导致编译错误；
3. 当一个方法抛出了异常，那么该方法的调用者必须处理或者重新抛出该异常；
4. 当子类重写父类抛出异常的方法时，声明的异常必须是父类所声明异常的同类或子类。

2.3 try & catch & finally

```
try {
    // 可能会发生异常的代码块
} catch (Exception e1) {
    // 捕获并处理try抛出的异常类型Exception
} catch (Exception2 e2) {
    // 捕获并处理try抛出的异常类型Exception2
} finally {
    // 无论是否发生异常，都将执行的代码块
}
```

1. `try` **语句块**：用于监听异常，当发生异常时，异常就会被抛出；
2. `catch` **语句块**：`catch` 语句包含要捕获的异常类型的声明，当 `try` 语句块发生异常时，`catch` 语句块就会被检查。当 `catch` 块尝试捕获异常时，是按照 `catch` 块的声明顺序从上往下寻找的，一旦匹配，就不会再向下执行。因此，如果同一个 `try` 块下的多个 `catch` 异常类型有父子关系，应该将子类异常放在前面，父类异常放在后面；
3. `finally` **语句块**：无论是否发生异常，都会执行 `finally` 语句块。`finally` 常用于这样的场景：由于 `finally` 语句块总是会被执行，所以那些在 `try` 代码块中打开的，并且必须回收的物理资源（如数据库连接、网络连接和文件），一般会放在 `finally` 语句块中释放资源。

`try` 语句块后可以接零个或多个 `catch` 语句块，如果没有 `catch` 块，则必须跟一个 `finally` 语句块。简单来说，`try` 不允许单独使用，必须和 `catch` 或 `finally` 组合使用，`catch` 和 `finally` 也不能单独使用。

Java 7 以后，`catch` 多种异常时，也可以像下面这样简化代码：

```
try {
    // 可能会发生异常的代码块
} catch (Exception | Exception2 e) {
    // 捕获并处理try抛出的异常类型
} finally {
    // 无论是否发生异常，都将执行的代码块
}
```