简介

术语	缩写	说明
Java Development Kit	JDK	Java 开发工具包，提供了 Java 的开发环境和运行环境
Java Runtime Environment	JRE	Java 运行环境，为 Java 的运行提供了所需环境
Standard Edition	SE	标准版，用于桌面或简单服务器应用的 Java 平台
Enterprise Edition	EE	企业版，用于复杂服务器应用的 Java 平台

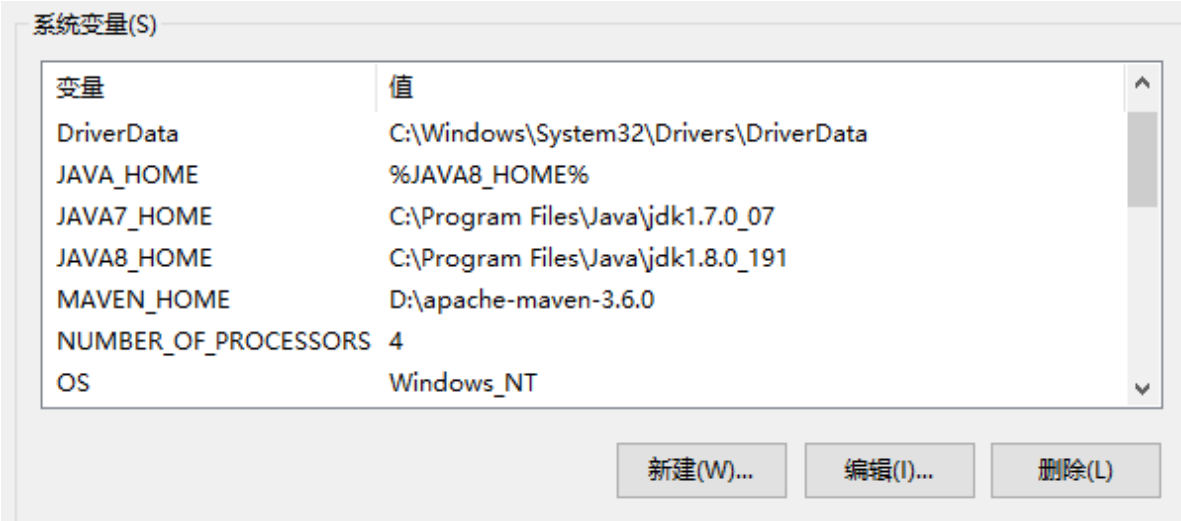
Java 特点

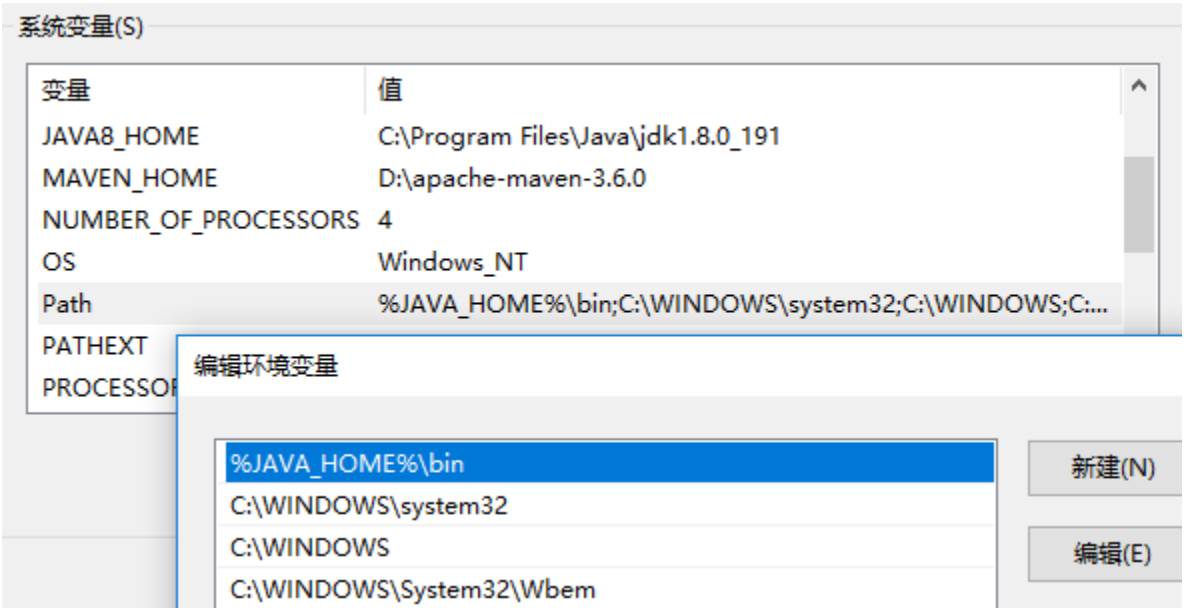
简单性、面向对象、分布式、健壮性、安全性、体系结构中立、可移植性、解释型、高性能、多线程、动态性

Java 跨平台原理

Java 程序是在虚拟机上运行的，不是直接在电脑上运行的，**JVM 也是一个软件**，用 C/C++ 开发的，不同的平台有不同的版本，**跨平台的是 Java 程序，不是 JVM。**Java 代码首先被编译成字节码文件，**字节码不能直接运行，必须通过 JVM 翻译成机器码才能运行。**不同平台下编译生成的字节码是一样的，但由 JVM 翻译成的机器码却不一样。即使将 Java 程序打包成可执行文件，仍然需要 JVM 的支持

环境配置与版本切换






```
# 检查是否安装成功
java -version
```

注释

```
// 最常用，其注释内容从//开始到本行结尾

/*
    长注释
*/

/**
 *  可以用来自动地生成文档，常用于类与方法注释
 */
```

数据类型

Java 语言是强类型语言，这就意味着必须为每一个变量声明一种类型。不同的数据类型，在内存中分配了不同的内存空间。在定义变量的时候，声明变量的数据类型，就会为变量 **合理的分配内存空间，避免空间浪费**

- 基本类型：**保存在栈内存中的简单数据段**，即这种值完全保存在内存中的一个位置
- 引用类型：**保存在堆内存中的对象**，即变量中保存的实际是对象的内存地址
    - 类、接口、数组、枚举、注解

基本数据类型（primitive type）

数据类型	关键字	存储需求	取值范围	默认值	包装类
字节型	byte	1	-128 ~ 127	0	Byte
短整型	short	2	-2^15 ~ 2^15-1	0	Short
整型	int	4	-2^31 ~ 2^31-1	0	Integer
长整型	long	8	-2^63 ~ 2^63-1	0	Long
单精度浮点型	float	4	-2^128 ~ 2^128	0.0	Float

数据类型	关键字	存储需求	取值范围	默认值	包装类
单精度浮点型	float	4	-2^128 ~ 2^128	0.0	Float
双精度浮点型	double	8	-2^1024 ~ 2^1024	0.0	Double
字符型	char	2	0 ~ 2^16-1	\u0000	Character
布尔型	boolean	1	true、false	false	Boolean

- long 类型的数值有一个后缀 `L` 或 `l`
- 二进制前缀：`0b` 或 `0B`
- 十六进制前缀：`0x` 或 `0X`
- 可以为数字字面量加下划线，只是为了让人更易读，Java编译器会去除这些下划线
  - `int i = 1_000_000`
- float 类型的数值有一个后缀 `F` 或 `f`，**浮点型没有后缀的话，默认为 double 类型**，也可以在浮点数值后面添加后缀 `D` 或 `d`
- char 类型的字面量值要用单引号括起来，用来存储 Unicode 编码的字符
  - 因为 Unicode 编码字符集中也包含了汉字，所以 **char 类型可以存储一个汉字**
  - Unicode 类编码占用 2 个字节，所以 char 类型的变量也是占用 2 个字节
- **多种数据类型做混合运算，先转换成容量最大的那种再做运算**
  - 八大基本数据类型，**除 boolean 类型之外都可以相互转换**
  - **byte、short、char 做混合运算的时候，会先转换成 int 再做运算**

```
short s1 = 1;
short s2 = 2;
// 会报错，需用int类型接受
short s = s1 + s2;
// 正确编译
int i = s1 + s2;

short s1 = 1;
// 会报错，由于s1+1运算时 会自动提升表达式的类型，所以结果是int型
s1 = s1 + 1
// 正确编译，因为+=是Java语言规定的运算符，Java编译器会对它进行特殊处理
s1 += 1;
```

枚举类型

有时变量的取值只在一个有限的集合内，枚举类型包括有限个命名的值。所有的枚举类型都是 Enum 类的子类。它们继承了这个类的许多方法

```
enum Color {
    red, black, white, green
}

public class Test {
    public static void main(String[] args) {
        // Color类型的变量只能存储这个类型声明中给定的某个枚举值，或者null值
        // null表示这个变量没有设置任何值
        Color color = Color.black;
        // 返回枚举常量名
        System.out.println(Color.black.toString());
        // 返回一个包含全部枚举值的数组
        Color[] colors = Color.values();
```

```
        // 将oneColor设置成Color.black
        Color oneColor = Enum.valueOf(Color.class,"black");
    }
}
```

参数传递

Java 中方法参数传递方式是 **按值传递**。基本类型，传递的是基本类型的 **字面量值的拷贝**，不改变其值。引用类型，传递的是该参数所引用的对象在堆中 **地址值的拷贝**，改变其值

- String 类虽然是引用数据类型，但它做参数传递时和基本数据类型是一样的

更多：Java 到底是值传递还是引用传递

运算符

在 Java 中，使用算术运算符 `+`、`-`、`*`、`/` 来表示加、减、乘、除运算，求余操作用 `%` 表示

```
// 整数被0除将会产生一个异常，而浮点数被0除将会得到无穷大或NaN结果
// ArithmeticException: / by zero
System.out.println(1/0);
// Infinity
System.out.println(1.0/0);
```

- `&&` 与 `&` 的区别
  - `&&`：具有短路效果。如果左边结果是 false，则右边不执行
  - `&`：无论左边是 false 还是 true，右边都会执行
- `||` 与 `|` 的区别
  - `||`：具有短路效果。如果左边结果是 true，则右边不执行
  - `|`：无论左边是 false 还是 true，右边都会执行
- `++n` 与 `n++` 的区别
  - 前缀形式会先完成加 1，然后再执行其他操作
  - 后缀形式会使用变量原来的值，执行完操作后，再进行加 1
  - 建议不要在表达式中使用 `++`，因为这样的代码很容易让人闲惑， 而且会带来烦人的 bug

```
int i = 1;
int x = i + ++i * i++;
// 执行++i，此时i=2，x=2
// 执行(++i)*i，此时i=2，x=4
// 执行i+(++i*i)，此时i=2，x=5
// 执行(i+++i*i)++，此时i=3，因为已经执行完操作，所以此时x仍为5
// x的值为5，i为3
```

- `^`：逻辑异或，两边只要是不一致就是 true
- `=`：赋值运算符
  - 基本类型，赋值运算符会直接改变变量的值，原来的值被覆盖掉
  - 引用类型，赋值运算符会改变引用中所保存的地址，原来的地址被覆盖掉。**但是原来的对象不会被改变**
- 移位运算符
  - `<<`：左移运算符，`num << 1`，相当于 num 乘以 2
  - `>>`：右移运算符，`num >> 1`，相当于 num 除以 2

- `>>>`：无符号右移，忽略符号位，空位都以 0 补齐

```
// 以下数据类型为int
 7：00000000000000000000000000000111
// -7的二进制为7的反码加上补码1，正数的原码、反码、补码相同
-7：11111111111111111111111111111001     // 末尾1表示-1，0表示-2

  7<<1：00000000000000000000000000001110      // 结果为14
  7>>1：00000000000000000000000000000011      // 结果为3
 -7<<1：11111111111111111111111111110010      // 结果为-14
 -7>>1：11111111111111111111111111111100      // 结果为-4
-7>>>1：01111111111111111111111111111100      // 结果为2147483644
```

- 三元运算符：`布尔表达式 ： 值1 ？ 值2`，表达式为 true，返回值1，表达式为 false，返回值2
- `==` 与 `equals()`
  - `==`：**基本数据类型比较的是值，引用数据类型比较的是内存地址**
  - `equals()`：如果被重写，则比较两个对象内容是否相等，否则，比较的是内存地址
  - String、Integer 等中的 equals 方法是被重写过的，因为 Object 的 equals 方法是比较的对象的内存地址，而 String 的 equals 方法比较的是对象的值

控制流程

- switch 语句将从与选项值相匹配的 case 标签处开始执行直到遇到 break 语句，或者执行到 switch 语句的结束处为止。如果没有相匹配的 case 标签，而有 default 子句，就执行这个子句
  - 条件括号中可以填写 byte、short、char、int 类型，即能够自动转换为 int 类型的都可以
  - 在 JDK7 之后可以填写 String 类型
  - **case 后面只能是常量**，不能是变量，**case 的值不能重复**
- while 循环本质上和 for 循环是一样的，可以相互替换，作用相同，建议使用 for 循环，因为变量会及早的从内存中消失，可以提高内存的使用效率
  - while 循环会首先检测循环条件，因此，循环体中的代码可能不会被执行，如果需要至少执行一次，可以使用 `do / while` 循环
- break、continue、return 区别
  - break：跳出一层循环，可以控制结束嵌套循环
  - continue：结束一趟循环，也可以像 break 那样加上循环的名字
  - return：它的作用不是结束循环的，而是结束方法

数组

数组是一种简单的数据结构，线性的结构，是相同数据类型的元素按一定顺序排列的集合。通过一个整型下标可以访问数组中的每一个值

- 数组一旦创建其长度是不可变的

初始化数组

数组初始化就是为数组开辟 **连续的内存空间**，并为每个数组元素赋予值。知道数组的首元素的内存地址，要查找的元素只要知道下标就可以快速的计算出偏移量，通过首元素内存地址加上偏移量可以快速计算出要查找元素的内存地址，通过内存地址快速定位该元素，所以数组查找元素的效率较高

- 动态初始化数组，会先在堆内存中分配这个数组，并且数组中每一个元素都采用 **默认值**

```java
// 静态初始化
int[] arr = {1, 2, 3, 4, 5};
// 动态初始化
int[] arr = new int[5];
```

二维数组

```java
// {{0,0,0},{0,0,0}}
int[][] arr = new int[2][3];

// 遍历
for (int i = 0; i < arr.length; i++) {
    for (int j = 0; j < arr[i].length; j++) {
        System.out.print(arr[i][j]);
    }
}
```

常用方法

```java
// 返回一个List
List<Integer> list = Arrays.asList(1, 2, 3);

// 排序，使用了优化的快速排序算法
Arrays.sort(arr);

// 二分法查找，参数为数组和需要查找的元素
int result = Arrays.binarySearch(arr, 3);

// 数组转字符串
String s = Arrays.toString(arr);

// 填充数组，参数为数组和需要填充的元素
Arrays.fill(arr, 5);

// 复制数组，参数为数组和新数组的长度
int[] newArr = Arrays.copyOf(arr, 2);

// 复制数组，参数为数组和需要复制的范围
int[] newArr = Arrays.copyOfRange(arr, 2, 3);
```

对象与类

面向过程	面向对象
把问题分解成一个一个步骤，每个步骤用函数实现，依次调用即可	将问题分解成一个一个步骤，**把每个步骤抽象成对象**，通过不同对象之间的调用，组合解决问题
注重 **过程**，把问题分解成多个不同的步骤，然后把各个步骤变成方法，使用的时候依次调用	注重 **对象之间的交互**，将复杂的事情简单化，把问题分解成各个对象，然后各个对象之间进行交互，每个对象内部都进行了封装
性能比面向对象高，类调用时需要实例化，开销比较大，比较消耗资源	易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统更加灵活、更加易于维护

- 类：**具有相同属性和方法的一组对象的集合**，是构造对象的模板或蓝图
- 包：在类名前面使用关键字 package 加入包名来避免命名冲突问题，通常使用倒写的域名命名
  - package 语句只能出现在 Java 文件的第一行，且只能有一个
  - 如果没有 package，默认表示无包名
  - Java.lang 包下所有类不需要手动导入，系统自动导入，Object 类，String 类都在这个包里面
- 方法：为了完成某个任务，对象所能执行的操作，可以提高代码的复用性
  - 不要在两个方法里面互相调用，程序会报出 StackOverflowError 错误
  - 方法自己调用自己叫做递归，必须要有结束条件，否则将会造成 StackOverflowError 错误
- 对象：**系统中描述客观事物的一个实体**

创建对象的五种方法

除了 clone 方法和反序列化外都调用了构造方法

- new 关键字：最常见也是最简单的创建对象的方式了。通过这种方式，可以调用任意的构造函数（无参的和有参的）
- Class 类的 newInstance 方法：newInstance 方法会调用无参的构造函数创建对象

```
// 第一种写法，需填入类的全限定名
Test test = (Test) Class.forName("com.test.Test").newInstance();

// 第二种方法
Test test = Test.class.newInstance();
```

- Constructor 类的 newInstance 方法：与 Class 类的 newInstance 方法类似，`java.lang.reflect.Constructor` 类里也有一个 newInstance 方法可以创建对象。可以通过这个 newInstance 方法调用有参数的和私有的构造函数

```
Constructor<Test> constructor = Test.class.getConstructor();
Test test = constructor.newInstance();
```

- 使用 clone 方法：调用一个对象的 clone 方法，JVM 就会创建一个新的对象，将之前对象的内容全部拷贝进去。用 clone 方法创建对象并不会调用任何构造函数，要使用 clone 方法，需要先实现 Cloneable 接口并重写 clone 方法

```java
public class Test implements Cloneable {
    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    public static void main(String[] args) throws
CloneNotSupportedException {
        Test test = new Test();
        Test new = (Test) test.clone();
    }
}
```

- 使用反序列化：当序列化和反序列化一个对象，JVM 会给我们创建一个单独的对象。在反序列化时，JVM 创建对象并不会调用任何构造函数。为了反序列化一个对象，需要先让类实现 Serializable 接口

```java
public class Test implements Serializable {

public static void main(String[] args) throws IOException,
ClassNotFoundException {
    // 序列化
    Test test = new Test();
    ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("test"));
    oos.writeObject(test);
    oos.flush();
    // 反序列化
    ObjectInputStream stream = new ObjectInputStream(new
FileInputStream("test"));
    Test newTest = (Test) stream.readObject();
    }
}
```

两种 newInstance 方法的区别

Class 类的 newInstance	构造器类的 newInstance
Class 类位于 Java 的 lang 包中	构造器类是 Java 反射机制的一部分
只能触发 **无参的构造方法** 创建对象	能触发 **无参的或有参的构造方法** 来创建对象
需要其构造方法是共有的或者对调用方法可见的	可以在特定环境下调用 **私有构造方法** 来创建对象
抛出类构造函数的异常	包装了一个 InvocationTargetException 异常

- Class 类本质上调用了反射包构造器类中无参数的 newInstance 方法，捕获了 InvocationTargetException，将构造器本身的异常抛出

更多：Java 中创建对象的 5 种方式

权限修饰符

可以限定其他类对该类、属性和方法的使用权限，**protected 和 private 不能用于修饰类**

修饰符	同一个类	同一个包	子类	其他包
public	○	○	○	○
protected	○	○	○	X
default	○	○	X	X
private	○	X	X	X

成员变量与局部变量

成员变量	局部变量
写在方法体的外面	写在方法体的里面
声明时可以不进行初始化值，**会自动以类型的默认值而赋值**	声明时必须进行初始化
可以被本类或其他类的方法进行调用	只能在声明局部变量的方法内进行调用
可以被访问控制修饰符及 static 所修饰	**不能被访问控制修饰符及 static 所修饰**
存在于堆内存，随着对象的创建而存在	存在于栈内存，随着方法的调用而自动消失

构造方法

为对象数据进行初始化

- 方法名与类名相同，**不能被继承，因此不能被重写，但可以被重载**
- 没有返回值类型，不能使用 return 返回值，不能被 void 修饰
- 如果一个类没有提供任何构造方法，系统 **默认提供一个无参数构造方法。**如果一个类已经手动的提供了构造方法，那么系统不会再提供任何构造方法。如果需要使用无参构造方法，就必须显式的写出
- 在创建对象的时候会 **默认执行无参构造方法，构造方法不能手动调用**

重载与重写

方法重载（Overload）	方法重写（Override）
发生在同一个类中	必须继承一个类或者实现一个接口
方法名相同，参数列表不同，与方法返回值类型、修饰符无关	必须具有 **相同的方法名、返回值类型和参数列表**
	重写的方法 **不能比被重写的方法拥有更低的访问权限，抛出更宽泛的异常**
	私有方法、构造方法不能被重写
	静态的方法不存在重写
	重写指的是成员方法，和成员变量无关

为什么不能根据返回类型来区分重载

因为调用时不能指定类型信息，编译器不知道你要调用哪个函数

```
// 当执行method(1,2)时，无法确定调用哪个方法
String method(int a, int b);
double method(int a, int b);
```

能否重写一个 private 或 static 的方法

- 不能重写私有方法，因为 private 修饰的变量和方法只能在当前类中使用，其他的类访问是不到
- 静态方法不能被重写，因为方法重写是基于运行时动态绑定的，而静态方法是编译时就静态绑定的。静态方法跟类的任何实例都不相关

static

- 修饰变量：程序运行时静态变量存放在方法区里面
- 修饰方法：**不用创建对象就能直接访问该方法**
  - 静态方法不能直接访问非静态的数据，静态方法不能使用 this 和 super
- 静态语句块：静态语句块在类加载阶段执行，**只执行一次**，并且是 **自上而下** 的顺序执行，**在构造方法之前执行**
- 被修饰的变量、方法、代码块都是属于 **类级别** 的，跟对象无关。可以通过 `类名.静态方法名` 的方式访问

静态变量与实例变量

- 语法定义上的区别：静态变量前要加 static 关键字，而实例变量前则不加
- 程序运行时的区别
  - 实例变量属于某个对象的属性，必须创建了实例对象，其中的实例变量才会被分配空间，才能使用这个实例变量
  - 静态变量不属于某个实例对象，而是属于类，所以也称为类变量，只要程序加载了类的字节码，不用创建任何实例对象，静态变量就会被分配空间，静态变量就可以被使用了

为什么不能用静态方法调用非静态方法

非静态方法是要与对象关联在一起的，必须创建一个对象后，才可以在该对象上进行方法调用，而静态方法调用时不需要创建对象，可以直接调用

当一个静态方法被调用时，可能还没有创建任何实例对象，如果从一个静态方法中发出对非静态方法的调用，那个非静态方法是关联到哪个对象上的呢，这个逻辑无法成立

This 与 Super

- This：代表着对当前对象的引用，在堆内存中的每个 Java 对象上都有一个 this 指向自己
  - 调用当前对象的非静态方法和数据
  - 可以区分成员变量和局部变量，使用 this 调用成员变量
  - 如果使用 this 调用构造方法的话，this **必须出现在构造方法的第一行**
- Super：代表的是当前子类对象中的父类型特征
  - 调用父类非静态方法和数据
  - 子类调用父类中的构造方法时，需要使用 super。一个构造方法第一行如果没有 `this()`，也没有显式的去调用 `super()`，**系统会默认调用** `super()`，如果已经有 this 了，那么就不会调用 super 了，**super 只能放在构造方法的第一行。**只是调用了父类中的构造方法，**并不会创建父类的对象**

Super 与 This 的区别

	调用成员变量	调用构造方法	调用成员方法
this	调用本类成员变量	调用本类构造方法	调用本类成员方法
super	调用父类成员变量	调用父类构造方法	调用父类成员方法

final

修饰	说明
类	无法被继承
方法	无法被重写
局部变量	一旦赋值，不可再改变
成员变量	必须初始化值
引用类型	该引用不可再重新指向其他的 Java 对象。但是 **地址值不能被改变，该引用指向的对象的属性值是可以修改的**

```
public final String str = "string";

public static void main(String[] args) {
    Test t = new Test();
    t.str = "another"; // 报错
    System.out.println(t.str + "append");// 可以运行，输出stringappend
}
```

封装

把客观事物封装成抽象的类，**隐藏对象的属性和实现细节，仅对外提供公共访问方式。**不用关心具体实现，提高安全性，提高了代码的复用性，减少耦合

- 封装的使用
  - 将成员变量用 private 修饰，private 仅仅是封装的一种体现形式，**封装不是私有**
  - 提供对应的 get 和 set 方法

继承

子类自动共享父类数据结构和方法的机制，是类之间的一种关系。在定义和实现一个类的时候，可以在一个已经存在的类的基础之上来进行，把这个已经存在的类所定义的内容作为自己的内容，并加入若干新的内容。**Java 只支持单继承和多层继承**，子类可以继承父类中的 **非 private 修饰的成员方法和成员变量，构造方法不能被继承**

- 如果一个类没有显示的继承其他类，那么这个类会 **默认继承 Object 类**，Object 是 SUN 公司提供的 Java 中的根类
- 提高了代码的复用性、维护性
- 让类与类之间产生了关系，是多态的前提

- 增强了类之间的耦合，软件开发的一个原则是 **高内聚，低耦合**
  - 内聚：一个模块内各个元素彼此结合的紧密程度
  - 耦合：一个软件里面不同模块之间相互连接的数量

多态

一种行为，多种状态。同一个接口，不同的实例执行不同的操作，在面向对象语言中，接口的多种不同的实现方式即为多态。**提高了程序的扩展性，降低了代码之间的耦合**

- 多态的实现
  - 有继承
  - 有方法重写
  - 有父类引用指向子类对象
- 动态绑定：在运行根据具体对象的类型进行绑定
- 静态绑定：在程序执行前已经被绑定

类型转换

子类向父类型进行转换，是 **自动类型转换**，也叫向上转型。父类向子类型转换，是 **强制类型转换**，也叫向下转型。如果有多个子类需要强转的话，需要先使用 instanceof 判断一下对象属于哪个子类

多态的使用

```
class Father {
    public int num = 111;
    public int ff = 666;
    public void same() { System.out.println("father same"); }
    public void differentFather(){ System.out.println("father different");
}
}

class Son extends Father {
    public int num = 222;
    public int ff = 999;
    public void same() { System.out.println("son same"); }
    public void differentSon(){ System.out.println("son different"); }
}

public class Test {
    public static void main(String[] args) {
        Father f = new Son();
        f.same();// 运行的是子类的方法，输出son same
        f.differentFather();// 输出father different
        ((Son) f).differentSon();// 调用子类特有方法，需强转，输出son different
        System.out.println(f.num);// 输出父类成员变量111
        System.out.println(f.ff);// 输出父类成员变量666
        System.out.println(((Son) f).ss);// 调用子类特有成员变量，需强转，输出999
    }
}
```

抽象类与接口

抽象类	接口
类的抽象，是一种模版设计	行为的抽象，是一种行为规范
有构造方法，可供非抽象子类创建对象	没有构造方法
不一定包含抽象方法，有抽象方法的一定是抽象类	**只能出现抽象方法，且必须被 `public abstract` 修饰，可省略不写**
可以有变量和常量	**只能出现常量，且必须被 `public static final` 修饰，可省略不写**
只支持单继承，可以继承抽象类和非抽象类，**非抽象类必须重写父类的所有抽象方法**	**接口间支持多继承，且只能继承接口**
可以实现接口	不可以实现别的接口

- 抽象类与接口都不能被实例化，都不能被 final 修饰
- 构造方法、被 static、final、private 修饰的方法不能声明为抽象方法

接口的优点

- 接口其实是一个特殊的抽象类，接口不是被类继承了，而是要被类实现
- 可以使项目分层，面向接口开发，提高开发效率
- 降低了代码之间的耦合度，提高了代码的可插拔性
- 开发中尽量使用接口，少用抽象类，一个类可以实现多个接口，却只能继承一个父类

默认方法

在 JDK1.8 之后可以为接口方法提供一个默认实现，**必须用 default 修饰符来标记方法**，这样一来就可以只关心需要的方法，而不用去实现不需要的方法

```
public interface PeopleDo {
    void say();

    default void say(String thing) {
        System.out.println(thing);
    }
}
```

- 默认方法冲突：如果先在一个接口中将一个方法定义为默认方法，然后又在超类或另一个接口中定义了 同样的方法，就会产生冲突
    - 超类优先：如果超类提供了一个具体方法，同名而且有相同参数类型的默认方法会被忽略
    - 接口冲突：如果一个超接口提供了一个默认方法，另一个接口提供了一个同名而且参数类型（不论是否是默认参数）相同的方法，必须覆盖这个方法来解决冲突

```
// 超类优先
public class EveryoneDo {
    public void say(String thing) {
        System.out.println(thing);
    }
}
//
public class Test extends EveryoneDo implements PeopleDo {
    @Override
```

```
    public void say() {}

    // 可以不重写，重写的是超类中的方法
    @Override
    public void say(String thing) {
        super.say(thing);
    }
}



// 接口冲突
public interface EveryoneDo {
    void say(String thing);
}
// 必须重写方法
public class Test implements PeopleDo,EveryoneDo {
    @Override
    public void say() {}

    @Override
    public void say(String thing) {}
}
```

Object 类

所有类的父类，所有类都 **隐式地** 继承 Object，因此省略了 extends Object 关键字

- `boolean equals()`：判断两个对象是否相同，Object 中的 equals 方法比较的是两个引用的内存地址。工作中，不应该比较内存地址，应该比较地址里面的内容，所以需要对 equals 方法进行重写

```
// 1.自反性
x.equals(x)

// 2.对称性
x.equals(y) == y.equals(x)

// 3.传递性
x.equals(y);y.equals(z);x.equals(z)

// 4.一致性，多次调用equals()方法结果不变
// 5.非空性，对任何不是null的对象equals(null)，结果都为false
x.equals(null)
```

- `Class<?> getClass()`：返回此 Object 的运行时的类型。不可重写，一般和 `getName()` 联合使用

```
Test test = new Test();
System.out.println(test.getClass().getName());
// 输出包名.类名：com.test.Test
```

- `String toString()`：返回 Java 对象的字符串表示形式，工作中一般对 toString 方法进行重写。如果直接打印一个引用数据类型的对象，系统会默认调用其 toString 方法。Object 中的

toString 方法返回的是：`类名@Java对象` **的内存地址经过哈希算法得出的 int 类型值再转换成十六进制**，一般将这个看做 Java 对象在堆中的内存地址

```
Test test = new Test();
System.out.println(test.toString());
// 输出包名.类名@内存地址：com.test.Test@1b6d3586
```

- `void finalize()`：不需要程序员去调用，由 **系统自动调用。**一个对象如果没有引用指向它，就会成为垃圾数据，等待垃圾回收器的回收，垃圾回收器在回收这个对象之前会自动调用该对象的 finalize 方法。程序员只能 **建议** 垃圾回收器回收垃圾 `System.gc()`

	说明
final	修饰类，不能被继承；修饰方法，不能被重写；修饰变量，只能赋值一次
finally	try 语句中的一个语句体，不能单独使用，语句体中的语句一定会执行
finalize	Object 中的一个方法，当没有引用指向某个对象时，对象的垃圾回收器在回收之前调用此方法

- wait、notify、notifyAll

wait	notify	notifyAll
调用该方法后当前线程进入睡眠状态	唤醒在该对象上等待的某个线程	唤醒在该对象上等待的所有线程

int hashcode()

返回散列值，实际上是返回一个 int 整数，默认是由对象的地址转换而来的。这个哈希码的作用是确定该对象在哈希表中的索引位置。同一个对象，如果该对象没有被修改，那么重复调用 `hashCode()` 返回的散列值都是相同的

为什么要有 hashCode

对底层是散列表的对象有提升性能的功能，散列表存储的是键值对，特点是：能根据 Key 快速的检索出对应的 Value。这其中就利用到了哈希码，从而可以快速找到所需要的对象，`hashCode()` 的作用就是 **获取哈希码**

如把对象加入 HashSet 时，HashSet 会先计算对象的 hashcode 值来判断对象加入的位置，同时也会与其他已经加入的对象的 hashcode 值作比较，如果没有相符的 hashcode，HashSet 会假设对象没有重复出现。但是如果发现有相同 hashcode 值的对象，这时会调用 equals 方法来检查 hashcode 相等的对象是否真的相同。如果两者相同，HashSet 就不会让其加入操作成功。如果不同的话，就会重新散列到其他位置。这样就 **大大减少了 equals 的次数，提高了执行速度**

hashCode() 与 equals()

- 两个对象相等，则 hashcode 一定也是相同的
- 两个对象相等，对两个对象分别调用 equals 方法都返回 true
- 两个对象有相同的 hashcode，也不一定是相等的
- equals 方法被重写，则 hashCode 方法也必须被重写
- `hashCode()` 的默认行为是对堆上的对象产生独特值。如果没有重写 `hashCode()`，则该 class 的两个对象无论如何都不会相等，即使这两个对象指向相同的数据

Object Clone()

`clone()` 是 Object 的 protected 方法，一个类不显式去重写 `clone()`，其它类就不能直接去调用该类实例的 clone 方法。clone 方法用于对象的克隆，一般想要克隆出的对象是 **独立** 的，即与原有的对象是分开的

- 浅拷贝：拷贝对象和原始对象的引用类型引用同一个对象
  - 基本数据类型进行值传递，引用数据类型进行引用传递
- 深拷贝：拷贝对象和原始对象的引用类型引用不同对象
  - 基本数据类型进行值传递，引用数据类型，创建一个新的对象，并复制其内容
- 深拷贝指的是该对象的成员变量（如果是可变引用）都应该克隆一份，浅拷贝指的是成员变量没有被克隆一份

clone 用法

- 克隆的对象要 **实现 Cloneable 接口**
  - `clone()` 不是 Cloneable 接口的方法，而是 Object 的一个 protected 方法。Cloneable 接口只是规定，如果一个类没有实现 Cloneable 接口又调用了 clone 方法，会抛出 `CloneNotSupportedException`
- **重写 clone 方法**，最好使用 public 修饰

```
// 浅拷贝：只拷贝了Person对象，而name没有拷贝
public class Person implements Cloneable {
    // 可变的成员变量
    private String name;

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    public static void main(String[] args) throws
CloneNotSupportedException {
        Person person = new Person();
        person.name = "习近平";

        Person personClone = (Person) person.clone();
        // 修改person的name
        person.name = "特朗普";
        // 输出结果"特朗普"
        System.out.println(person.name);
        // clone对象输出结果"习近平"
        System.out.println(personClone.name);
    }
}
```

```
// 深拷贝：不仅拷贝了Person对象，也拷贝了date成员变量
public class Person implements Cloneable {
    // 可变的成员变量
    public Date date;

    @Override
    public Object clone() throws CloneNotSupportedException {
        // 拷贝Person对象
```

```
        Person person = (Person) super.clone();
        // 将可变的成员变量也拷贝
        person.date = (Date) date.clone();
        // 返回拷贝的对象
        return person;
    }
}
```

clone 方法的保护机制

Object 中 `clone()` 是被 protected 修饰的，这样就可以保证只有在本类里面才能进行克隆对象。一般来说，protected 修饰的类或属性，对于自己、本包和其子类可见。所以 Person 类一个包的 Test 类，Person 类的子类 Someone 类应该是可以调用 clone 方法创建 person 对象的拷贝，然而事实上会报错

```
public class Person{}   // Person没有实现Cloneable接口，没有重写clone方法
public class Test{
    public static void main(String[] args){
        Person person = new Person();
        person.clone();// 'clone()' has protected access in
'java.lang.Object'
    }
}
public class Someone extends Person{
    public static void main(String[] args){
        Person person = new Person();
        person.clone();// 'clone()' has protected access in
'java.lang.Object'
    }
}
```

事实上，对于 protected 的成员或方法，**要分子类和父类是否在同一个包中**。与父类不在同一个包中的子类，只能访问自身从父类继承而来的受保护成员，而不能访问父类实例本身的受保护成员。上面的代码的问题在于 Person 与 Object **不是在同一个包下** 的，而 Person 直接访问了 Object 的 clone 方法，显然是不行的

更多：Java:由 Object.clone()而引出的 protected 权限问题、【JDK源码】java.lang.Object

为什么要用 clone()

在实际编程过程中，常常要遇到这种情况：有一个对象 A，在某一时刻 A 中已经包含了一些有效值，此时可能会需要一个和 A 完全相同新对象 B，并且此后对 B 任何改动都不会影响到 A 中的值，即 A 与 B 是两个独立的对象，但 B 的初始值是由 A 对象确定的。在 Java 语言中，用简单的赋值语句是不能满足这种需求的。要满足这种需求虽然有很多途径，但实现 clone 方法是其中最简单，也是最高效的手段

- 使用 clone 方法来拷贝一个对象即复杂又有风险，它会抛出异常，并且还需要类型转换。最好不要去使用 `clone()`，可以使用拷贝构造函数或者拷贝工厂来拷贝一个对象

clone 与 copy 的区别

```
// 假设有一个Person对象
Person one = new Person("one",18);
// copy了一下引用，one和two都指向内存中同一个object，这样one或two的一个操作都可能影响到对方
Person two = one;
// 使用clone，就可以得到一个one的拷贝，这样one和two之间互不影响
Person two = (Person)one.clone();
```

new 一个对象的过程和 clone 一个对象的过程区别

- new 操作符的本意是分配内存。程序执行到 new 操作符时，**首先去看 new 操作符后面的类型**，因为知道了类型，才能知道要分配多大的内存空间。分配完内存之后，再调用构造函数，填充对象的各个域，即对象的初始化，构造方法返回后，一个对象创建完毕，可以把他的引用（地址）发布到外部，在外部就可以使用这个引用操纵这个对象
- clone 在第一步是和 new 相似的，都是分配内存，调用 clone 方法时，分配的内存和原对象相同，然后再使用原对象中对应的各个域，填充新对象的域，填充完成之后，clone 方法返回，一个新的相同的对象被创建，同样可以把这个新对象的引用发布到外部

更多：详解 Java 中的 clone 方法、Object 对象你真理解了吗?

包装类

在 Java 中，一切皆对象，但八大基本类型却不是对象。所以八大基本数据类型都有相应的包装类，包装类都在 java.lang 包里

- 可以在对象中定义更多的功能方法，方便对基本类型进行操作
- 集合不允许存放基本类型数据，只能存放引用类型数据，如包装类
- 基本类型和包装类之间可以相互转换，即自动装箱与自动拆箱。使得在编程时能够更注重于业务的开发，而不是每转换一次就需要写一堆转换代码
- 有时候需要传递一个 Object 变量，所以不能传递基本数据类型，就可以使用包装类

包装类型	基本类型
传递内存地址	传递值
存放在堆中，通过对象的引用来调用	**栈内存中的简单数据段**
初始值为 null	**初始值视具体的类型而定**
需要用 new 关键字实例化	无需通过 new 关键字实例化

类型转换

```
// int转Integer
Integer i1 = Integer.valueOf(10);
// int转String
String s1 = 10 + "";

// Integer转String
String s2 = i1.toString();
// Integer转int
int i2 = i1.intValue();

// String转Integer
Integer i4 = Integer.valueOf("10");
// String转int
int i3 = Integer.parseInt("10");
```

- 装箱：将基本类型用对应的引用类型包装起来，**通过包装类的 `valueOf()` 实现**
- 拆箱：将包装类型转换为基本数据类型，**通过包装类的 `xxxValue()` 实现**

int 与 Integer 的比较

```java
// false，生成两个对象，内存地址不同
Integer a = new Integer(10);
Integer b = new Integer(10);
System.out.println(a == b);

// true，Integer变量和int变量比较时，会自动拆箱，变为两个int变量的比较
int c = 10;
System.out.println(a == c);

// false，非new生成的Integer变量指向的是常量池中的对象，而使用new生成的变量指向堆中新建的对象
// 两者在内存中的地址不同
Integer d = 10;
System.out.println(a == d);

// true，两个非new生成的Integer对象比较时，如果两个变量的值在区间-128到127之间，Java会进行缓存
// 除Integer外，Byte、Short、Long、Character、Boolean都支持缓存池
Integer e = 10;
System.out.println(d == e);
// false
Integer i = 128;
Integer j = 128;
System.out.println(i == j);
```

更多：java面试题之 int和 Integer的区别

反射

Java 反射机制是在 **运行时** 动态的获取信息和调用对象的方法，对于任意一个类，都能够获取这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性

程序中一般的对象的类型都是在编译期就确定下来的，而 Java 反射机制可以动态地创建对象并调用其属性，这样的对象的类型在编译期是未知的。所以可以通过反射机制直接创建对象，即使这个对象的类型在编译期是未知的

反射的作用

- 可以实现简单的反编译，获取类中的属性和方法等基本信息
- 获取类的属性、方法等，甚至可以调用 private 方法
- 在使用 IDE 时，当输入一个对象或类并想调用它的属性或方法时，输入点号，编译器就会自动列出它的属性或方法，这里就会用到反射
- 最重要的用途就是开发各种通用框架。很多框架都是配置化的，比如通过 XML 文件配置 Bean，为了保证框架的通用性，可能需要根据配置文件加载不同的对象或类，调用不同的方法，这个时候就必须用到反射，运行时动态的加载需要加载的对象

反射的优点

提高程序的灵活性和扩展性，降低耦合性。允许程序创建和控制任何类的对象，无需提前硬编码目标类

反射的缺点

- 影响性能：使用反射时，代码量更多，并且反射包括了一些动态类型，所以 JVM 无法对这些代码进行优化。因此，反射操作的效率要比那些非反射操作低得多，应该避免在经常被执行的代码或对性能要求很高的程序中使用反射

- 安全限制：使用反射技术要求程序必须在一个没有安全限制的环境中运行
- 破坏封装性：由于反射允许代码执行一些在正常情况下不被允许的操作，如访问私有的属性和方法，降低可移植性。反射代码破坏了抽象性，因此当平台发生改变的时候，代码的行为就有可能也随着变化

Class 对象

在类加载器将 `.class` 文件读取到内存中的时候，JVM 会创建这个 `.class` 文件的对象，并且只创建一个存放到 JVM 的方法区内存中，即在 **运行期间，一个类，只有一个 Class 对象产生。**在 `java.lang` 包下有个 Class 类，这个类就是 `.class` 文件的对象类型，任何类在被使用时，都会创建这个类的 Class 对象。反射相关的类一般都在 `java.lang.relfect` 包里

Class 和 `java.lang.reflect` 一起对反射提供了支持，`java.lang.reflect` 类库主要包含了以下三个类

- **Field** ： 可以使用 get 和 set 方法读取和修改 Field 对象关联的字段
- **Method** ： 可以使用 invoke 方法调用与 Method 对象关联的方法
- **Constructor** ： 可以用 Constructor 创建新的对象

获取 Class 对象的三种方式

```
// 1.调用某个对象的getClass方法，不常用，已经有对象了，需要反射的意义不大
Person person = new Person();
Class one = person.getClass();

// 2.使用Class类的forName静态方法，常用，字符串可以传入也可写在配置文件中等多种方法
Class two = Class.forName("com.test.Person");

// 3.直接获取某一个对象的class，不常用，需要导入类的包，依赖太强，不导包就抛编译错误
Class three = Person.class;

// 因为Person这个类在JVM中只有一个，所以内存地址应该都是相同的，指向堆中唯一的Class对象
System.out.println(one == two);     // true
System.out.println(two == three);   // true

// 也可以使用isInstance方法来判断是否为某个类的实例
System.out.println(one.isInstance(person));     // true
System.out.println(two.isInstance(person));     // true
System.out.println(three.isInstance(person));   // true
```

使用反射创建对象

```
// 1.通过Class对象的newInstance()方法
Test test = (Test) Class.forName("com.test.Test").newInstance();
// 或
Test test = Test.class.newInstance();

// 2.通过Constructor对象的newInstance()方法
Constructor<Test> constructor = Test.class.getConstructor();
Test test = constructor.newInstance();
```

获取方法

```
public class Person {
    public String name;
```

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

    public void show() {System.out.println(this.name);}

    public void say(String name){System.out.println(name);}
}

// 获取类或接口的所有方法，但不包括继承的方法
Method[] methods = person.getDeclaredMethods();
// 输出：访问修饰符 返回类型 包名.类名.方法名
// public java.lang.String com.test.Test.Show()
for (Method method : methods) {
    System.out.println(method);
}

// 获取类或接口的所有公共方法，包括继承的方法
Method[] methods = person.getMethods();

// 返回一个特定的方法，第一个参数为方法名称，后面的参数为方法的参数对应Class的对象
Method method = person.getDeclaredMethod("show");
Method method = person.getDeclaredMethod("say", String.class);

// 返回一个特定的方法，第一个参数为方法名称，后面的参数为方法的参数对应Class的对象
// 只能获取共有的方法，否则回会报NoSuchMethodException
Method method = person.getMethod("say", String.class);

// 获取类中所有的构造方法
Constructor[] constructors = person.getDeclaredConstructors();
for (Constructor constructor : constructors) {
    System.out.println(constructor);
}

// 获取类中所有的公共的构造方法
Constructor[] constructors = person.getConstructors();

// 返回一个特定的构造方法，参数为方法的参数对应Class的对象
Constructor constructor = person.getDeclaredConstructor(String.class);

// 返回一个特定的公共的构造方法，参数为方法的参数对应Class的对象
Constructor constructor = person.getConstructor(String.class);

// 使用反射调用类中的方法
Method method = person.getDeclaredMethod("say", String.class);
Object obj = person.newInstance();
method.invoke(obj, "习近平");
```

获取成员变量

```
// 获取类或接口的所有的成员变量，但不包括继承的成员变量
Field[] fields = person.getDeclaredFields();
// 输出：访问修饰符 数据类型 包名.类名.变量名
// public java.lang.String first.Person.name
for (Field field : fields) {
    System.out.println(field);
}
```

```
8
9  // 获取类或接口的所有的公共成员变量，包括继承的成员变量
10 Field[] fields = person.getFields();
11
12 // 返回一个特定的成员变量，参数为成员变量名
13 Field field = person.getDeclaredField("name");
14
15 // 返回一个特定的共有的成员变量，参数为成员变量名
16 Field field = person.getField("name");
17
18 // 使用反射获取类中指定的属性并赋值
19 Field field = person.getDeclaredField("name");
20 Object obj = person.newInstance();
21 // 从外部打破封装性
22 field.setAccessible(true);
23 field.set(obj, "习近平");
24 System.out.println(field.get(obj));
```

- 可以使用 `void setAccessible(true)` 方法，之后调用私有方法或成员变量，

获取父类或父接口

```
1 // 获取父类
2 Class sup = person.getSuperclass();
3
4 // 获取父接口
5 Class[] inter = person.getInterfaces();
```

更多：深入解析Java反射（）、深入解析Java反射（2）

Lambda 表达式

Lambda 表达式是 Java 8 添加的一个新特性，可以认为，Lambda 是一个匿名函数（相似于匿名内部类），作用是返回一个实现了接口的对象

- 虽然 Lambda 表达式对某些接口进行简单的实现，但是并不是所有的接口都可以使用 Lambda 表达式来实现，**要求接口中定义的必须要实现的抽象方法只能有一个**

```
1 // 该注解修饰函数式接口，即意味着接口中的抽象方法只能有一个，否则编译器会报错
2 // default方法不会造成任何影响，可以出现default方法
3 @FunctionalInterface
4 public interface Test {
5     void test();
6 }
```

- Lambda 表达式是一个匿名函数，主要关注方法的参数列表和方法体
  - `()`：描述参数列表
  - `{}`：描述方法体
  - `->`：Lambda 运算符，读作 `goes to`

```
1 // 无返回值的无参接口
2 @FunctionalInterface
3 public interface LambdaNoneReturnNoneParameter {
4     void test();
```

```
}

// 无返回值的单参接口
@FunctionalInterface
public interface LambdaNoneReturnSingleParameter {
    void test(int i);
}

// 无返回值的多参接口
@FunctionalInterface
public interface LambdaNoneReturnMultipleParameter {
    void test(int a, int b);
}

// 有返回值的无参接口
@FunctionalInterface
public interface LambdaSingleReturnNoneParameter {
    int test();
}

// 有返回值的单参接口
@FunctionalInterface
public interface LambdaSingleReturnSingleParameter {
    int test(int i);
}

// 有返回值的多参接口
@FunctionalInterface
public interface LambdaSingleReturnMultipleParameter {
    int test(int a, int b);
}
```

```
// 无参无返回值
LambdaNoneReturnNoneParameter l1 = () -> {
    System.out.println("无参无返回值");
};
l1.test();

// 单参无返回值
LambdaNoneReturnSingleParameter l2 = (int i) -> {
    System.out.println("\n单参无返回值传入参数：" + i);
};
l2.test(999);

// 多参无返回值
LambdaNoneReturnMutipleParameter l3 = (int a, int b) -> {
    System.out.println("\n多参无返回值传入参数：" + a + "，" + b);
};
l3.test(333, 555);

// 无参有返回值
LambdaSingleReturnNoneParameter l4 = () -> {
    System.out.println("\n无参有返回值");
    return 100;
};
int l4Result = l4.test();
System.out.println("无参有返回值结果：" + l4Result);
```

```
26
27  // 单参有返回值
28  LambdaSingleReturnSingleParameter l5 = (int i) -> {
29      System.out.println("\n单参有返回值传入参数：" + i);
30      return a;
31  };
32  int l5Result = l5.test(200);
33  System.out.println("单参有返回值结果：" + l5Result);
34
35  // 多参有返回值
36  LambdaSingleReturnMutipleParameter l6 = (int a, int b) -> {
37      System.out.println("\n多参有返回值传入参数：" + a + "，" + b);
38      return a + b;
39  };
40  int l6Result = l6.test(100, 200);
41  System.out.println("多参有返回值结果：" + l6Result);
```

Lambda 表达式语法精简

- 参数类型的省略：由于在接口中已经定义了参数，所以在 Lambda 表达式中参数的类型可以省略
  - 如果省略参数的类型，则所有的参数的类型都要省略

```
1  LambdaNoneReturnSingleParameter l2 = (i) -> {
2      System.out.println("\n单参无返回值传入参数：" + i);
3  };
```

- 参数小括号的省略：如果参数列表中，**参数的个数有且只有一个**，那么小括号可以省略，且仍然可以省略参数的类型

```
1  LambdaNoneReturnSingleParameter l2 = i -> {
2      System.out.println("\n单参无返回值传入参数：" + i);
3  };
```

- 方法体大括号的省略：如果方法体只有一条语句，那么此时大括号可以省略

```
1  LambdaNoneReturnSingleParameter l2 = i ->
2      System.out.println("\n单参无返回值传入参数：" + i);
```

- return 的省略：如果方法体 **只有一条语句，且是返回语句**，可以省略 return，**且必须要省略大括号**

```
1  LambdaSingleReturnSingleParameter l4 = () -> a;
```

方法引用

如果新建了许多接口的实现对象，其方法都是相同的，但是如果方法需要修改，那么修改的复杂度就随着对象数量的上升而上升，此时可以将一个 Lambda 表达式的实现指向一个已经写好的方法

- **返回值的类型和参数列表要与接口中定义的一致**

```
1  public class Test {
2      public static void main(String[] args) {
3          // 每次使用都实现相同的方法，则非常冗余
```

```
        LambdaSingleReturnSingleParameter l1 = i -> i * 2;
        LambdaSingleReturnSingleParameter l2 = i -> i * 2;
        // 一般的方法调用：将Lambda表达式的实现指向change方法
        LambdaSingleReturnSingleParameter l3 = i -> change(i);
        // 方法引用：引用方法隶属者的change方法
        // 方法隶属者：静态方法隶属者为类，非静态方法的隶属者是对象
        LambdaSingleReturnSingleParameter l4 = Test::change;
        LambdaSingleReturnSingleParameter l5 = new Test()::change;
    }

    private static int change(int i) {return i * 2;}

    private int change2(int i) {return i * 2;}
}
```

构造方法的引用

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

    public Person() {
        System.out.println("无参构造方法");
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
        System.out.println("有参构造方法");
    }
}

public class SimpleTest {
    public static void main(String[] args) {
        PersonCreator creator = () -> new Person();
        // 构造方法的引用
        PersonCreator creator = Person::new;
        // 调用getPerson方法得到person对象
        Person person = creator.getPerson();
        // 引用有参构造方法
        PersonCreator2 creator2 = Person::new;
        Person person = creator.getPerson("习近平", 18);
    }
}

interface PersonCreator{
    Person getPerson();
}

interface PersonCreator2{
    Person getPerson(String name, int age);
}
```

更多：Java-Lambda表达式和'方法引用'的对比和详解

代码块

使用 `{}` 括起来的代码被称为代码块

- 在方法中出现
  - 普通代码块：限定变量生命周期，及早释放，提高内存利用率
  - 同步代码块：被 synchronized 修饰，语句块会自动被加上内置锁，从而实现同步
- 在类中方法外出现
  - 构造代码块：每次调用构造方法都会执行，并且 **在构造方法前执行**
  - 静态代码块：被 static 修饰，**在类被加载的时候执行**，**且只执行一次**，常用来加载驱动

```
public class Block {
    {
        System.out.println("构造代码块");
    }

    static {
        System.out.println("静态代码块");
    }

    public void method() {
        {
            System.out.println("普通代码块");
        }

        synchronized (this) {
            System.out.println("同步代码块");
        }
    }
}
```

执行顺序

```
public class Father {
    {System.out.println("父类构造代码块");}

    static {System.out.println("父类静态代码块");}

    public void method() {
        {System.out.println("父类普通代码块");}

        synchronized (this) {System.out.println("父类同步代码块");}
    }
}

public class Son extends Father {
    {System.out.println("子类构造代码块");}

    static {System.out.println("子类静态代码块");}

    @Override
    public void method() {
        {System.out.println("子类普通代码块");}

        synchronized (this) {System.out.println("子类同步代码块");}
    }
}
```

```
25
26 public class Test {
27     public static void main(String[] args) {
28         Son son = new Son();
29         son.method();
30     }
31 }
32 // 父类静态代码块 -> 子类静态代码块 -> 父类构造代码块 -> 子类构造代码块
33 // -> 子类普通代码块 -> 子类同步代码块
```

内部类

内部类是定义在一个类的内部的类，内部类 **不能定义静态变量和方法。**接口内也可以使用内部类，但没必要

- 更好的实现隐藏：内部类可以被 private 与 protected 修饰
- **内部类拥有外部类的所有元素的访问权限**：可以直接调用所有外部类的成员，包括 private 修饰的变量和方法、非静态变量和方法等
- 可以间接的实现多重继承：**一个类中可以写多个内部类，分别继承不同的类**
- 可以避免实现的接口或继承的类中有同名的方法
- 当想要定义一个回调函数且不想编写大量代码时，使用匿名内部类比较便捷

成员内部类

定义在一个类的内部，可以被权限修饰符和 static 修饰

- 外部类想访问成员内部类的成员，必须 **先创建一个成员内部类的对象**
- 当成员内部类中拥有和外部类同名的成员变量或者方法时，**默认会调用成员内部类的成员**
- 如果要使用外部类的同名成员，可以通过 `外部类名.this.成员变量/成员方法` 调用，如果调用外部类的静态变量及方法，可以省略 this

```
1  public class OuterClass {
2      public int i1 = 1;
3      private static int i2 = 2;
4      public void outerMethod(){}
5      private static void outerStaticMethod(){}
6
7      class InnerClass {
8          private int i = 0;
9          public void innerMethod() {
10             System.out.println(i1);
11             System.out.println(i2);
12             outerMethod();
13             outerStaticMethod();
14         }
15     }
16
17     public static void main(String[] args) {
18         OuterClass outerClass = new OuterClass();
19         OuterClass.InnerClass oci = outerClass.new InnerClass();
20         innerClass.innerMethod();
21     }
22 }
```

局部内部类

定义在一个方法或者代码块里面的类，可以直接访问 **方法内或者代码块内的成员。不能被权限修饰符或 static 修饰**

- 如果要访问外部类的成员变量，则需先创建外部类对象

```
public class OuterClass {
    public int i1 = 1;
    private static int i2 = 2;

    public void outerMethod() {
    }

    private static void outerStaticMethod() {
    }

    public static void main(String[] args) {
        int i = 1
        class InnerClass {
            private int i = 1;

            public void method() {
                OuterClass outerClass = new OuterClass();
                System.out.println(outerClass.i1);
                System.out.println(OuterClass.i2);
                System.out.println(i);
            }
        }
        int x = new InnerClass().i;
        System.out.println(x);
    }
}
```

匿名内部类

没有名称的类，利用父类的构造函数和自身类体构造成一个类，其他地方不能引用，**没有构造器，不能实例化，只能使用一次**

```
public class One {
    public void show() {}
}

public class Two {
    public void show() {}
}

public class OuterClass {
    public static void main(String[] args) {
        OuterClass outerClass = new OuterClass();
        One one = new One() {
            private String name = "one";
            @Override
            public void show() {
                System.out.println("hello " + name);
            }
        };
        one.show();
```

```
        outerClass.two.show();
    }

    Two two = new Two() {
        private String name = "two";
        @Override
        public void show() {
            System.out.println("hello " + name);
        }
    };
}
```

静态内部类

被 static 修饰的成员内部类，不需要依赖于外部类的，可以看做静态变量，不能直接使用外部类的非静态变量或方法

```
public class OuterClass {
    private static int num = 9;

    public static void main(String[] args) {
        OuterClass.InnerClass.innerStaticMethod();
        InnerClass innerClass = new OuterClass.InnerClass();
        innerClass.innerMethod();
    }

    static class InnerClass {
        public static void innerStaticMethod(){ System.out.println(num); }
        public void innerMethod(){ System.out.println(num); }
    }
}
```

异常

- Java 号称安全性，异常考虑到可能出现的错误，使用异常处理后把错误处理和真正的工作分开来
- 代码更易组织，更清晰，复杂的工作任务更容易实现
- 更安全，不至于由于一些小的疏忽而使程序意外崩溃了
- 在 Java 中，所有的异常都有一个共同的祖先 java.lang 包中的 **Throwable 类**
    - Throwable 有两个重要的子类：**Exception、Error**

Error 与 Exception

- Error：描述了 Java 运行时系统的内部错误和资源耗尽错误。应用程序不应该抛出这种类型的对象。如果出现了这样的内部错误， 除了通告给用户，并尽力使程序安全地终止之外，再也无能为力了。这种情况很少出现
- Exception：程序本身可以处理的异常。又分为两个分支：一个分支派生于 RuntimeException，另一个分支包含其他异常。划分的依据是由程序错误导致的异常属于 RuntimeException；而程序本身没有问题，但由于像 I/O 错误这类问题导致的异常属于其他异常

可查异常与不可查异常

- 可查异常（checked exceptions）：编译器要求 **必须处置的异常**，即当程序中可能出现这类异常，要么用 try-catch 语句捕获它，要么用 throws 子句声明抛出它，否则编译不会通过。在正确的程序在运行中，很容易出现的、情理可容的异常状况。可查异常虽然是异常状况，但在一定程度上它的发生是可以预计的，而且一旦发生这种异常状况，就必须采取某种方式进行处理
    - 除了 RuntimeException 及其子类以外，其他的都属于可查异常
- 不可查的异常（unchecked exceptions）：编译器 **不要求强制处置的异常**，一般是由程序逻辑错误引起的，在程序中可以选择捕获处理，也可以不处理
    - 包括 RuntimeException 与其子类 和 Error

运行时异常与非运行时异常

- 运行时异常：RuntimeException 类及其子类都是不可查异常，Java 编译器不会检查它，当程序中可能出现这类异常，即使没有用 try-catch 语句捕获它，也没有用 throws 子句声明抛出它，也可以编译通过。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生

异常	说明
ArithmeticException	算数异常
ArrayIndexOutOfBoundsException	数组下标越界异常
ClassCastException	类型转换异常
ClassNotFoundException	找不到类异常
IllegalArgumentException	非法参数异常
IllegalStateException	非法状态异常
IndexOutOfBoundsException	下标越界异常
NullPointerException	空指针异常
NumberFormatException	数字格式异常

- 非运行时异常 （编译异常）：除 RuntimeException 类及其子类以外的异常。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过

Throwable 类常用方法

- `String getMessage()`：返回异常发生时的详细信息
- `String toString()`：返回异常发生时的简要描述
- `String getLocalizedMessage()`：返回异常对象的本地化信息。使用 Throwable 的子类覆盖这个方法，可以声称本地化信息
    - 如果子类没有覆盖该方法，则该方法返回的信息与 `getMessage()` 返回的结果相同
- `void printStackTrace()`：在控制台上打印 Throwable 对象封装的异常信息

```java
try {
    System.out.println(1 / 0);
} catch (Exception e) {
    // / by zero
    System.out.println(e.getMessage());
    // java.lang.ArithmeticException: / by zero
    System.out.println(e.toString());
    // / by zero
    System.out.println(e.getLocalizedMessage());
    // java.lang.ArithmeticException: / by zero
    // at com.test.Test.main(Test.java:10)
    e.printStackTrace();
}
```

异常处理

在 Java 应用程序中，异常处理机制为：**抛出异常，捕获异常**

- 抛出异常：任何 Java 代码都可以抛出异常，当一个方法出现错误引发异常时，方法创建异常对象并交付运行时系统，异常对象中包含了异常类型和异常出现时的程序状态等异常信息，运行时系统负责寻找处置异常的代码并执行
- 捕获异常：一个方法所能捕捉的异常，一定是 Java 代码在某处所抛出的异常，即异常总是 **先被抛出，后被捕捉**。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。潜在的异常处理器是异常发生时依次存留在调用栈中的方法的集合
    - 当异常处理器所能处理的异常类型与方法抛出的异常类型相符时，即为合适的异常处理器。运行时系统从发生异常的方法开始，依次回查调用栈中的方法，直至找到含有合适异常处理器的方法并执行。当运行时系统遍历调用栈而未找到合适的异常处理器，则运行时系统终止，同时，意味着 Java 程序的终止

try-catch 与 finally

- try 块：用于捕获异常。其后可接 0 个或多个 catch 块，如果没有 catch 块，则必须跟一个 finally 块
- catch 块：用于处理 try 捕获到的异常
- finally 块：无论是否捕获或处理异常，finally 块里的语句都会被执行，除非以下四种特殊情况
    - 在 finally 语句块中发生了异常
    - 在前面的代码中用了 `System.exit()` 退出程序
    - 程序所在的线程死亡
    - 关闭 CPU

try 块里面的内容称为监控区域。Java 方法在运行过程中出现异常，则 **创建异常对象**。将异常抛出监控区域之外，由运行时系统寻找匹配的 catch 子句以捕获异常。若有匹配的 catch 子句，则运行其异常处理代码，try-catch 语句结束。如果抛出的异常对象属于 catch 子句的异常类或子类，则认为生成的异常对象与 catch 块捕获的异常类型相匹配

```java
try {
    System.out.println();
} catch (NullPointerException e1) {
    e1.printStackTrace();
} catch (ClassCastException e2) {
    e2.printStackTrace();
}
// Java7之后，同一个catch子句中可以捕获多个异常类型
```

```java
try {
    System.out.println();
} catch (NullPointerException | ClassCastException e) {
    e.printStackTrace();
}

// 没有catch块，则必须写上finally块
try {
    System.out.println();
} finally {
    System.out.println("nothing");
}

try {
    System.out.println();
} catch (NullPointerException | ClassCastException e) {
    e.printStackTrace();
} finally {
    System.out.println("nothing");
}
```

finally 和 return 的执行顺序

- try 和 catch 中有 return
  - finally 中的代码总会执行，而且 finally 语句在 return 语句执行之后，在 return 返回之前执行

```java
// 输出try、finally、5（1+3+1）
// 如果出现异常，视出现位置而定，当前出错位置输出try、catch、finally、41（1+30+10）
public class Test {
    public static void main(String[] args) {
        Test test = new Test();
        // 传入1
        System.out.println(test.testReturn(1));
    }

    public int testReturn(int i) {
        try {
            System.out.println("try");
            // System.out.println(1 / 0);
            i += 3;
            return i + 1;
        } catch (Exception e) {
            System.out.println("catch");
            i += 30;
            return i + 10;
        } finally {
            System.out.println("finally");
            i += 300;
        }
    }
}
```

- finally 中有 return
  - 会直接返回，不会再去返回 try 或者 catch 中的返回值

```java
// 没有出现异常，则输出try、finally、404
// 如果出现异常，视出现位置而定，当前出错位置输出try、catch、finally、431（1+30+300+100）
public int testReturn(int i) {
    try {
        System.out.println("try");
        // System.out.println(1 / 0);
        i += 3;
        return i + 1;
    } catch (Exception e) {
        System.out.println("catch");
        i += 30;
        return i + 10;
    } finally {
        System.out.println("finally");
        i += 300;
        return i + 100;
    }
}
```

- finally 中对于返回变量做的改变是否会影响最终的返回结果
    - 如果 try 和 catch 的 return 是一个变量时且函数是从其中一个返回时，后面 finally 中语句即使有对返回的变量进行赋值的操作时，也不会影响返回的值

更多：finally 和 return 的执行顺序

throw 与 throws

- throw：用在方法体中，用来抛出一个 Throwable 类型的异常。 程序会在 throw 语句后立即终止，它后面的语句不执行，然后在包含它的所有 try 块中，从里向外寻找含有与其匹配的 catch 子句的 try 块
    - 如果抛出了检查异常，还应该在 **方法头部声明方法可能抛出的异常类型**，该方法的调用者也必须检查处理抛出的异常。如果所有方法都层层向上抛获取的异常，最终 JVM 会进行处理，打印异常消息和堆栈信息

```java
public class Test {
    public static void main(String[] args) {
        boolean flag = true;
        System.out.println("我会执行");
        if (flag) {
            throw new RuntimeException("test");
        }
        System.out.println("我不会被执行");
    }
}
```

- throws：如果一个方法可能会出现异常，但没有能力处理这种异常，可以在方法声明处用 throws 子句来声明抛出异常。throws 语句用在方法定义时声明该方法要抛出的异常类型，多个异常可使用逗号分割。当方法抛出异常列表的异常时，方法将不对这些类型及其子类类型的异常作处理，而抛向调用该方法的方法或对象，由他去处理。还可以层层向上抛获取的异常，最终必须要有能够处理该异常的调用者

```java
public class Test {
    public static void main(String[] args) throws NullPointerException {
        Test test = null;
        System.out.println(test.toString());
    }
}
```

throw	throws
用在方法体内，跟的是 **异常对象名**	用在方法声明后面，跟的是 **异常类名**
只能抛出一个异常对象名	可以跟多个异常类名，用逗号隔开
表示抛出异常，由方法体内的 **语句处理**	表示抛出异常，由该方法的 **调用者来处理**

自定义异常类

```java
// 继承Exception类
public class MyException extends Exception{
    // 创建构造方法，调用父类构造方法
    public MyException(String str){
        super(str);
    }
}
```

异常链

将异常发生的原因一个传一个串起来，即把底层的异常信息传给上层，这样逐层抛出。当程序捕获到了一个底层异常，在处理部分选择了继续抛出一个更高级别的新异常给此方法的调用者。 这样异常的原因就会逐层传递。这样，位于高层的异常递归调用 getCause 方法，就可以遍历各层的异常原因

- 异常链的实际应用很少，发生异常时候逐层上抛不是个好注意，上层拿到这些异常又能怎么如何处理，而且异常逐层上抛会消耗大量资源，因为要保存一个完整的异常链信息

更多：Java异常、深入理解Java中异常体系

泛型

把明确类型的工作推迟到创建对象或调用方法的时候，可以把运行时的问题提前到编译时期。**提供了编译期的类型安全**，确保把正确类型的对象放入集合中，避免了在运行时出现 ClassCastException 异常

- 可以明确集合中的数据类型，提高安全性、可读性，可以使用增强 for 循环遍历
- 代码更加简洁，不需要强制转换
- 程序更加健壮，只要编译时期没有警告，那么运行时期就不会出现 ClassCastException 异常

通配符

其实这些字母并没有实际用途，只有字面意思，只是一个约定的规范，`Test<T>` 与 `Test<A>` 在执行效果上并没有什么区别

- T：代表一个具体的 Java 类型
- E：代表 Element 的意思，或者 Exception 异常的意思
- K：代表 Key 的意思，通常与 V 一起配合使用
- V：代表 Value 的意思，通常与 K 一起配合使用

- S：代表 Subtype 的意思
- ?：**无限定的通配符**
- `? extends E`：**设定通配符上限**，接收 E 类型或者 E 的子类型
- `? super E`：**设定通配符下限**，接收 E 类型或者 E 的父类型

泛型的使用

- 泛型类：用于类的定义中，用户使用该类的时候，才把类型明确下来。通过泛型可以完成对一组类的操作对外开放相同的接口，如 List、Set、Map 等容器类。泛型类的子类，如果有明确的类型参数，需将用泛型的地方全部替换成传入的实参类型，否则需将泛型的声明也一起加到类中。**类上声明的泛型只对非静态成员有效**，如果静态方法要使用泛型的话，**必须将静态方法也定义成泛型方法**

```
public class Test<T> {
    private T t;

    public T getT() {
        return t;
    }

    public void setT(T t) {
        this.t = t;
    }

    public static void main(String[] args) {
        Test<Integer> test = new Test<>();
        test.setT(1);
        System.out.println(test.getT());
    }
}
```

- 泛型接口：与泛型类相似

```
public interface Test<T> {
    T doAnything();
}

// 当实现泛型接口的类，未传入泛型实参时，在声明类的时候，需将泛型的声明也一起加到类中
// 如果不声明泛型，编译器会报错
public class One<T> implements Test<T> {
    T doAnything();
}

// 当实现泛型接口的类，传入泛型实参时，需将用泛型的地方全部替换成传入的实参类型
public class One<String> implements Test<String> {
    String doAnything();
}
```

- 泛型方法：在调用方法的时候指明泛型的具体类型，类型参数写在返回值前面，声明的类型参数，还可以当作返回值的类型

```
public class Test {
    // 只有声明了<T>的方法才是泛型方法，泛型类中的使用了泛型的成员方法并不是泛型方法
    public <T> T method(T t){
        return t;
    }
```

```
6
7       public static <T> void staticMethod(T t) {
8           System.out.println("static方法");
9       }
10
11      public static void main(String[] args) {
12          Test test = new Test();
13          System.out.println(test.method("fuck"));
14          System.out.println(test.method(12));
15          staticMethod("fuck");
16          staticMethod(12);
17      }
18  }
```

泛型擦除

泛型是通过类型擦除来实现的，泛型信息只存在于代码编译阶段，在进入 JVM 之前，与泛型相关的信息会被擦除

无论何时定义一个泛型类型，都自动提供了一个相应的原始类型（raw type）。 原始类型的名字就是删去类型参数后的泛型类型名。擦除（erased）类型变量，并替换为限定类型（无限定的变量用 Object）

局限性

泛型擦除，是泛型能够与之前的 Java 版本代码兼容共存的原因。但也因为类型擦除，它会抹掉很多继承相关的特性，这是它带来的局限性

```
1   List<Integer> list = new ArrayList<>();
2   list.add(1);
3   // 因为泛型的限制，以下代码会报错，基于对类型擦除的了解，利用反射，就可以绕过这个限制
4   // list.add("String");list.add(27.12);会报错
5   Method method = list.getClass().getDeclaredMethod("add", Object.class);
6   method.invoke(list, "String");
7   method.invoke(list, 27.12);
8
9   for (Object o : list) {
10      System.out.println(o);  // 成功打印
11  }
```

可以把 `List<String>` 传递给一个接受 `List<Object>` 参数的方法吗

会导致编译错误，因为 `List<Object>` 可以存储任何类型的对象包括 String、Integer 等，而 `List<String>` 却只能用来存储 String

`Class<T>` 和 `Class<?>` 的区别

`Class<T>` 在实例化的时候，T 要替换成具体类，`Class<?>` 是个通配泛型，? 可以代表任何类型，主要用于声明时的限制情况

PECS 原则

- 如果要从集合中读取类型 T 的数据，并且不能写入，可以使用 `? extends 通配符` (Producer Extends)

- 如果要从集合中写入类型 T 的数据，并且不需要读取，可以使用 `? super 通配符`（Consumer Super）
- 如果既要存又要取，那么就不要使用任何通配符

泛型数组

Java 不能创建具体类型的泛型数组

```
// 这两行代码是无法在编译器中编译通过的。原因是类型擦除带来的影响
List<Integer>[] list = new ArrayList<Integer>[];
List<Boolean> list = new ArrayList<Boolean>[];
```

`List<Integer>` 和 `List<Boolean>` 在 JVM 中等同于 `List<Object>`，所有的类型信息都被擦除，程序也无法分辨一个数组中的元素类型具体是 `List<Integer>` 类型还是 `List<Boolean>` 类型

```
List<?>[] list = new ArrayList<?>[10];
list[1] = new ArrayList<String>();
List<?> v = list[1];
```

借助于无限定通配符却可以，？代表未知类型，所以它涉及的操作都基本上与类型无关，因此 JVM 不需要针对它对类型作判断，因此它能编译通过

更多：泛型就这么简单、java 泛型详解、Java 泛型，你了解类型擦除吗、JAVA泛型通配符T，E，K，V区别