Java语言基础、代码设计原则及最佳实践

1990年末，Sun公司成立“Green计划”团队，由詹姆斯·高斯林（James Gosling）博士带领，开始为智能家电编写软件，鉴于当时C++等计算机语言在嵌入式平台开发上的局限性，团队决定开发一种新的语言，取名Oak。1992年，Green团队完成了新平台的部分功能，同年11月，Green计划被转化成“FirstPerson有限公司”， FirstPerson团队在华纳公司发布电视机顶盒发布会被有限电视界认为给用户太多权利而争标失败，与3DO公司另一笔机顶盒交易也没有成功。Green项目夭折，FirstPerson团队一半成员被调。

1994年夏天，互联网和浏览器出现，Gosling 意识到这是个机会，将Oka进行小规模改造，1994年秋，团队中 Naughton 与 Jonathan 完成了第一个Java语言网页浏览器，因Oak被注册，改名为Java。

关于Java名字由来：Java原本是指印尼著名的爪哇岛，爪哇岛盛产咖啡，据说当研究出Java语言的牛人们在为它命名时由于闻到香浓的咖啡味，遂决定采用此名称。而在Java语言里，还有几个与咖啡相关的名称，JavaBeans（咖啡豆）、NetBeans（网络豆）。

1995年初，Sun公司发布Java语言，向互联网所有用户公开，包括源代码。

Java 1.0 1996年01月23日

Java 1.1 1997年02月18日

Java 1.2 1998年12月04日

Java 1.3 2000年05月18日

Java 1.4 2004年02月06日

Java 1.5 2004年09月30日

Java 6 2006年12月11日

Java 7 2011年07月29日

Java 8 2014年03月18日

Java 9 2017年09月22日

Java 10 2018年03月21日

Java 11 2018年09月25日

Java 12 2019年03月19日

1. Java语言基础

Java是一种基于Java虚拟机（JVM）、跨平台、面向对象的程序设计语言。

1.1 Java程序开发流程

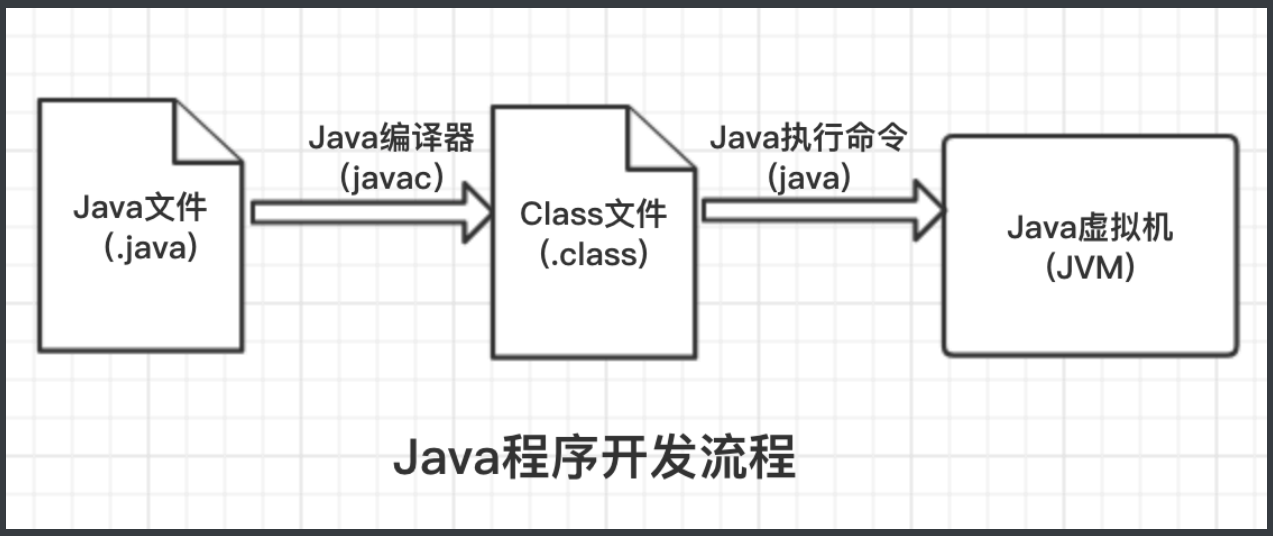

图1 Java程序开发流程

下面通过实际操作来演示开发流程：

1. 如何编写一个Java程序

要运行一个Java程序，必须至少编写一个Java程序文件(.java文件)，这个文件必须含有一个固定格式的main方法，格式为:

```
public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello, World!");

    }

}
```

2. 如何在Java程序中引用已有的本地软件包

在编译阶段和运行阶段都可以使用-classpath来指定本地软件包位置(也可以是文件夹)作为类寻找路径，以便找到代码中引用的类文件。

3. 如何在Java程序中引用外部软件包

引用外部的软件包与引入本地类似，只不过需要事先下载到本地，在实际中不必这么繁琐，使用强大的项目管理工具Maven，可实现自动解析并引入远程资源库（一个由Maven管理的软件包仓库）中的软件包。

1.2 Java虚拟机

Java虚拟机（以下简称为JVM）是一个虚拟的计算机设备，它屏蔽了不同操作系统的差异，使相同的代码（字节码）在多种不同类型的操作系统上都能运行，即通常所说的“一次编译，到处运行”，这只需要在不同的操作系统上安装相应的JVM即可实现。

1.2.1 支持多种编程语言

Java语言的程序编译成字节码后就可以运行在JVM上，但JVM本身却不并只是可以运行Java语言编写的程序，一切能编译成标准Java字节码的程序语言的编写的代码都可以在JVM上运行，例如Scala、Kotlin等，因为JVM是面向字节码的运行平台，它与Java语言并没有强制绑定关系。

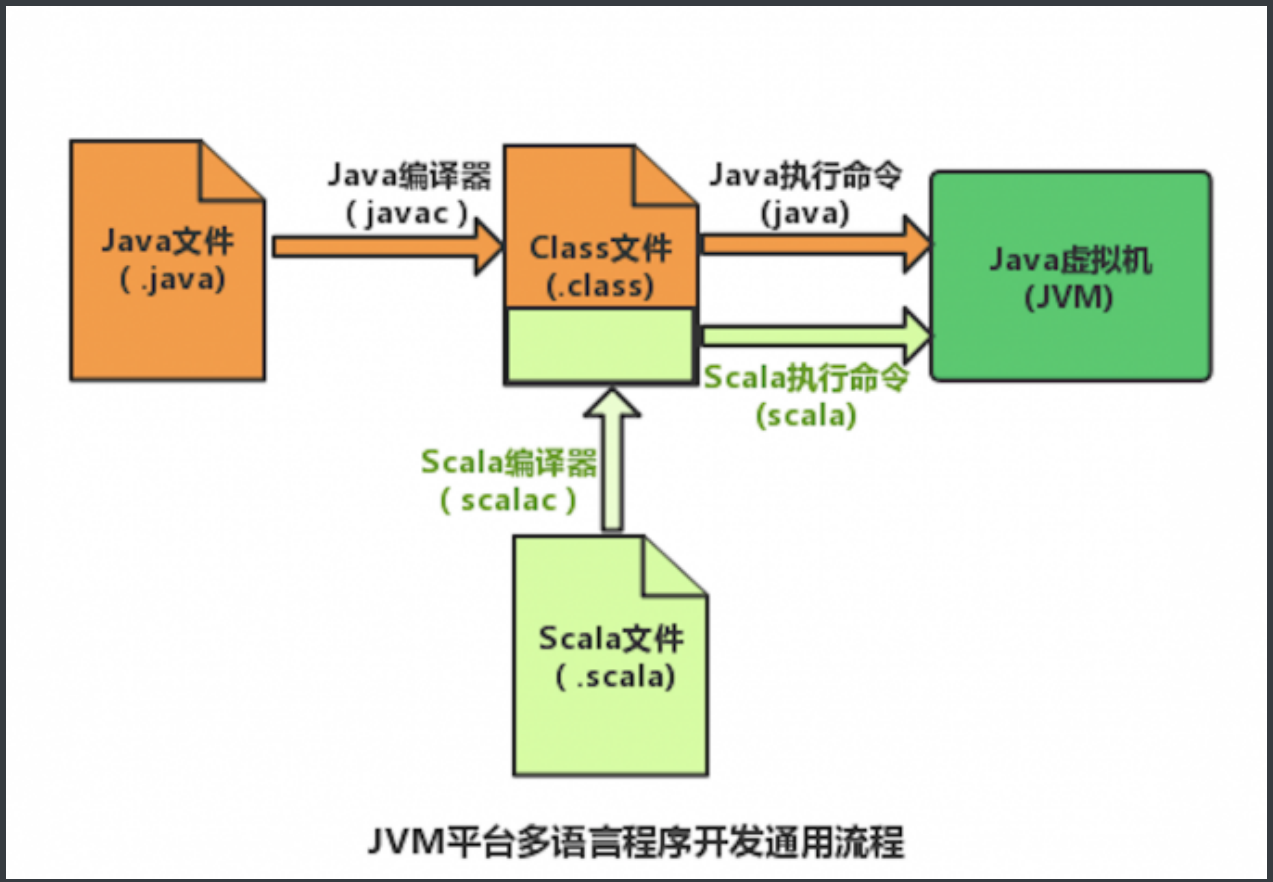

图2 Java平台多编程语言支持示意图

上图2展示Java和Scala语言是如何通过一样的开发流程运行在JVM上的。所有能在JVM平台运行的语言都可以称为JVM平台语言。JVM平台语言的代码之间可以相互调用。

1.2.2 运行时架构

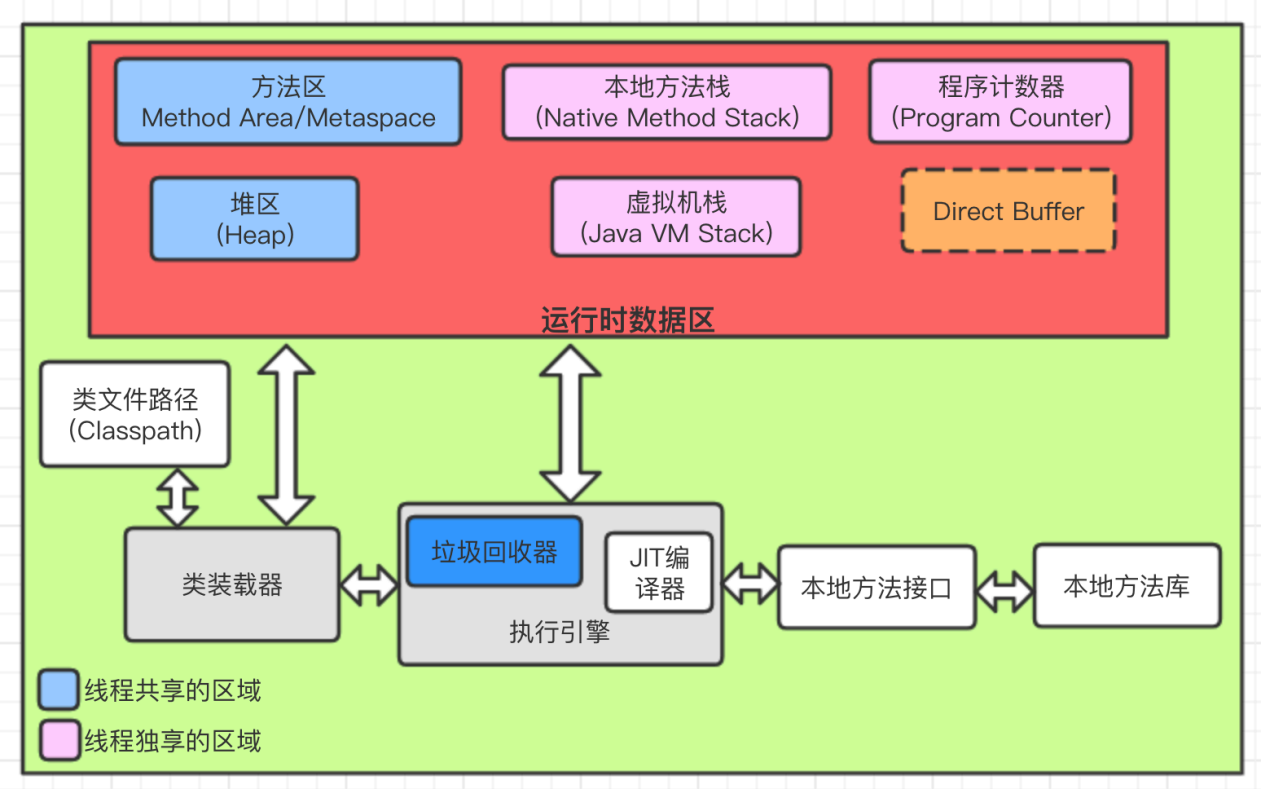

图3 JVM运行时架构

1.2.3 内存模型

Java内存模型（JMM）规定了所有的对象都存储在主内存中，每个线程还有自己的工作内存（又叫本地内存），线程的工作内存中保存了该线程中用到的主内存中对象变量的副本，包括对象的成员变量等，线程对这些变量的所有操作都必须在工作内存中进行，而不能直接读写主内存。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量的传递均需要自己的工作内存和主内存之间进行数据同步，为了保证共享的主内存在被多线程访问时的正确性（可见性、有序性、原子性），内存模型定义了共享内存读写操作的行为规范，即什么时候做数据同步以及如何做数据同步。

需要注意的是，主内存和工作内存都是抽象概念，与JVM内存结构中的堆、栈、方法区等并没有直接对应关系，因为它们的划分标准并不是同一个层次的。例如工作内存对应的具体存储空间（介质、地址）是不确定的，涉及到CPU缓存、寄存器以及其它的硬件和编译器优化等方面。

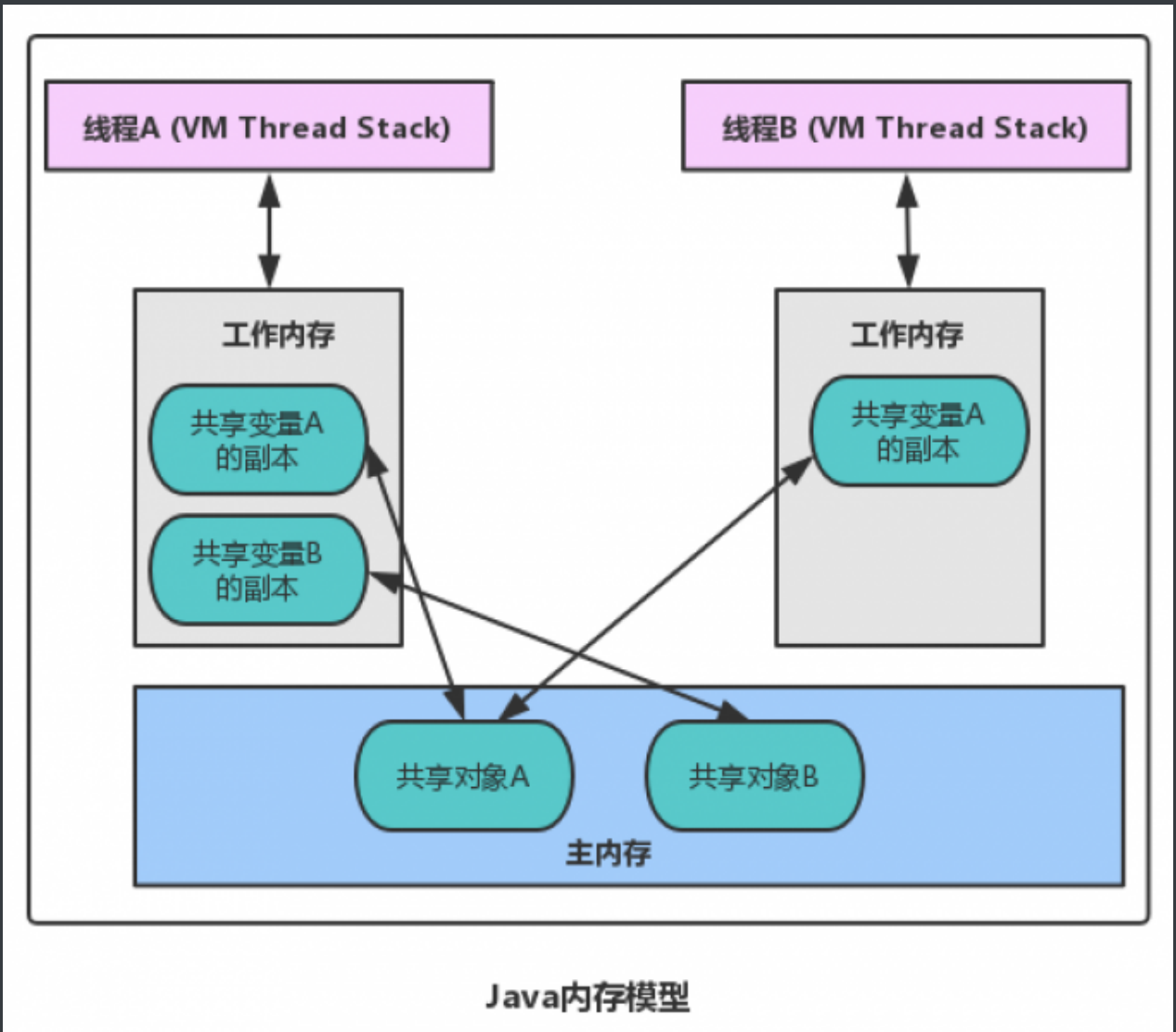

图4 JVM内存模型

1.3 Java平台体系结构

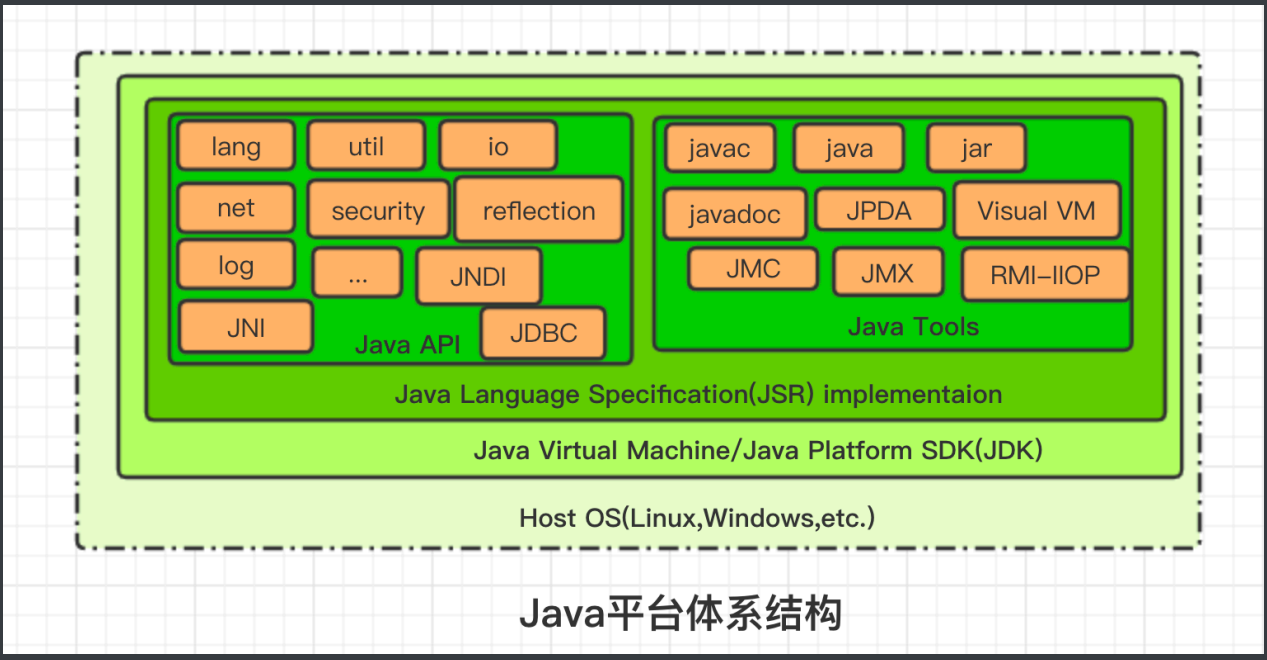

图5 Java平台体系结构

1.4 Java规范及实现

Java语言是一个由标准化发展而来，并且朝着标准化发展而去的语言，这种标准化通过JSR来管理，JSR由JCP负责实施执行。
下面解释下这两个概念：

JCP：Java Community Process，是一个开放性的 Java技术规范管理组织 ，成立于1998年，官网，由社会各界Java团体或者个人组成的社区，其职责是规划和领导Java的发展，不断更新Java技术规范，其成员列表在这里。其中，由24个成员组成的执行委员会则是该组织的最高决策机构，负责规划Java的技术演进方向。

JSR：Java Specification Requests，Java规范请求草案，是由JCP成员向JCP执行委员会提交的Java发展议案，经过一系列流程后，最终通过的JSR会体现在未来的Java平台中。

以下列举一些JSR规范：

编号	内容	备注
JSR 14	Add Generic Types To The Java Programming Language	泛型支持
JSR 133	JavaTM Memory Model and Thread Specification	内存模型及线程
JSR 365	Contexts and Dependency Injection (CDI) 2.0	依赖注入管理
JSR 369	Java Servlet 4.0	
JSR 370	Java API for RESTful Web Services (JAX-RS) 2.1	
JSR 367	The Java API for JSON Binding (JSON-B) 1.0	

JSR 374	Java API for JSON Processing (JSON-P)1.1	
JSR 380	Bean Validation 2.0	
JSR 250	Common Annotations 1.3	通过注解规范
JSR 338	Java Persistence 2.2	持久化规范
JSR 356	Java API for WebSocket 1.1	
JSR 919	JavaMail 1.6	
JSR 107	JCACHE - Java Temporary Caching API	内存缓存规范
JSR 352	Batch Applications for the Java Platform	批量处理任务规范
JSR 305	Annotations for Software Defect Detection	软件缺陷检测规范

表1 一些JSR规范列表

JSR定义的只是规范，具体实现可以由JVM平台本身或者第三方来完成。

每一个Java语言主要版本的发布，都会有相应的规范文档随之公布，其中主要包含语言规范和虚拟机规范。例如最新的JDK 12的规范文档为：

The Java Virtual Machine Specification, Java SE 12 Edition

The Java Language Specification, Java SE 12 Edition

这是学习Java语言最好的资料。

1.5 Java编程指南

1.5.1 包

Java中的包可以代表两种含义，一种是package包，一种是jar(Java Archives)包。

package

package包是Java中组织代码结构的基本单位，它是带有可见性语义的细粒度的类的集合，默认情况下一个类只能被它所在package包内的类访问到。所以一般将一些功能上相关或者结构上相似的类放进同一package中。

同时package也给类引入了命名空间，每个类都属于唯一一个确定的package。同一个package内不允许有两个类同名，不同package里允许有同名的类。

jar

jar(Java Archives)包由一个或者多个package经过归档压缩形成的包，通常以.jar作为文件后缀，以便发布、依赖管理，值得一提的是，它们是可跨平台使用的。

1.5.2 接口与类

作为一门经典的面向对象编程（Object Oriented Programming, OOP）语言，Java严格恪守OOP的信条 - 封装、继承、多态。 接口（interface）和类（class）作为OOP的载体在实际编程中扮演着重要角色。可以说，在Java程序中一切皆是类(接口)。封装、继承、多态这些特性都是通过类（接口）机制来实现。

什么是接口?

接口是对某一类对象特征与行为的抽象，它定义了某一类对象一个或多个方法。接口和它的方法都是抽象的，没有实际的代码逻辑，因而也不能用它创建对象。

```
public interface IHello {
    void sayHello();
}
```

什么是类?

类是对某一类对象属性与行为的抽象与实现，它定义并实现了方法。它可以是单独存在，也可以在其中实现某些已知的接口的方法。类可以用来创建对象，这也是类的最大用途。

```
public class TextHelloImpl implements IHello{

    @Override
    public void sayHello() {
        System.out.println("say hello by text.");
    }

}
```

还有抽象类?

抽象类（abstract class）是含有抽象方法的类，一般作为接口与其实现类之间的桥梁的存在。抽象类无法用来创建对象。

```
public abstract class AbstractHello implements IHello{

    private String helloText = "你好";

}
```

1.5.2.1 类访问权限作用域

Java中类（属性、方法）访问权限作用域修饰符有四种，分别是public、protected、private和defaut（默认的访问权限是package）。除了默认的作用域外，分别使用**public、protected、private**关键字表示。不同的修饰符拥有的访问权限作用域如下表所示。

修饰符	本类	本包中的类	子类	其它任何类
public	√	√	√	√
protected	√	√	√	×
default	√	√	×	×
private	√	×	×	×

√：代表有权限访问

×：代表无访问权限

表2 不同修饰符的访问权限列表

1.5.2.2 类构造器

类构造器是在创建类对象的时调用的一个方法，用来完成对象初始化工作，Java中类构造器是一个特殊的Java方法，它的方法名必须与类名完全相同，它由访问权限、类名、构造参数（可选）组成。不含有构造参数的构造器称为无参构造器。下面代码是一个例子：

```
public class Cat {

    private String name;
    private String color;
    private int age;

    public Cat() {
    }

    public Cat(String name, String color, int age) {
        this.name = name;
        this.color = color;
        this.age = age;
    }

}
```

一个类可以不定义构造器，这时编译器将会为它自动生成一个无参构造器，这个构造器的方法体为空。

1.5.2.3 类的静态方法

Java中支持类的静态方法，即不需要创建类对象，就能调用的方法。下面代码示例定义了包括静态方法的类：

```
public class Utils {

    public static String joinStringWithUnderline(String str1,String str2) {
        return str1+"_"+str2;
    }

}
```

然后，其它类可以这样调用，

```
public class CallStaticMethodDemo {

    public static void main(String[] args) {
        String joinedString = Utils.joinStringWithUnderline("abc","123");
        System.out.println(joinedString);//打印的内容为abc_123
    }

}
```

1.5.2.4 类的继承

Java中类的继承只允许单一继承，即每个类有且只有一个父类，要么显式指明父类，否则将自动隐式继承一个父类——Object类，Object类是Java中所有类的父类，可以称它为根类，它定义了所有类对象都应该具备的最基本的一些方法，例如toString(), hashCode()。 Java中类的继承是可传递的，即一个继承的父类的属性和方法可以传递给继承它的子类。

1.5.2.5 类的多态

Java中类的多态本质上是方法代码的运行时绑定，它是通过面向接口编程实现的。下面用一个实际例子来演示多态性。

```
public class AudioHelloImpl implements IHello {

    @Override
    public void sayHello() {
        System.out.println("say hello by audio.");
    }

}
```

```
public class PolymorphismDemo {

    public static void main(String[] args) {

        IHello textHello = new TextHelloImpl();
        IHello audioHello = new AudioHelloImpl();
        hello(textHello);//打印say hello by text.
        hello(audioHello);//打印say hello by audio.

    }

    public static void hello(IHello hello) {
        hello.sayHello();
    }

}
```

以上代码的执行结果是打印以下内容：

```
say hello by text.
say hello by audio.
```

1.5.3 数据类型

Java是一种强类型语言，它规定的数据类型可分为两大类，一类是基本数据类型，一类是引用数据类型，引用类型又可分为类对象类型、接口类型、数组类型等几种。 Java中有8种基本数据类型（primitive types）,每种基本类型都有对应的对象类型。关于基本类型与对象类型的主要区别在于：

- 基本类型属于只能用于数值运算，它不属于对象。而对象类型不仅能用于数值运算，而且可以通过自身方法调用完成一些功能，例如，如果有Integer类型变量i，可以调用i.longValue()方法将其转换成long型变量。

- 基本类型存储空间在栈内分配，而对象类型在堆内分配。例如定义int i=0；，则i的值直接存储在栈空间中。定义Integer i（i此时是对象）= new Integer(5)；这样，i对象数据存储在堆中，i的引用存储在栈中，通过栈中的引用来操作对象。
- 在进行方法调用传参时，基本类型的参数会进行值传递，而对象类型的是进行地址传递。

基本类型（对象类型）	字节	默认值	取值范围	示例
byte(Byte)	1	0	-2^7-2^7-1	byte b=10;
char(Character)	2	' \u0000'	0-2^16-1	char c='c';
short(Short)	2	0	-2^15-2^15-1	short s=10;
int(Integer)	4	0	-2^31-2^31-1	int i=10;
long(Long)	8	0	-2^63-2^63-1	long n=10L;
float(Float)	4	0.0f	-2^31-2^31-1	float f=10.0F
double(Double)	8	0.0d	-2^63-2^63-1	double d=10.0;
boolean(Boolean)	4/1	false	true/false	boolean flag=true;

表3 Java基本数据类型
说明：

- 在声明定义基本类型字面量变量时，如果不明确指定，整数型的默认是int，带小数的默认是double。例如：int i = 3;double d = 3.14;
- boolean类型单独使用是占4个字节，在数组中是1个字节。
- 如果需要在计算中不允许任何的误差，应该使用BigDecimal类，而不是float或者double。
- 从Java 5开始，基本类型与其及对应对象类型之间可以实现自动包装/解包（Wrapper/UnWrapper)，即实现在编程中通用。
- 从Java 7开始，可以加前缀0b来代表二进制数，例如：byte b = 0b1000;表示b的数值是8。数字字面量可以加上下划线提高易读性，例如：int i = 200_0000;代表i的数值是200万。
- 多线程应用场景下，为保证long和double类型变量读写都是原子操作，应使用volatile关键字修饰变量。

1.5.4 运算符及运算符的优先级

Java语言提供了一套丰富的运算符用于各种类型的数值计算。按类型可分为以下几类：

- 算术运算符

```
+，-，*，  /，%，++，--
```

- 关系运算符

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

- 位运算符

```
&, |, ^, ~, <<, >>, >>>
```

- 逻辑运算符

```
&&, ||, !
```

- 赋值运算符

```
=, +=, -=, *=, /=等
```

- instanceof运算符

这个运算符比较特殊，它用于判断一个对象是否为一个特定类型（接口、类），它的格式为：

```
( Object reference variable ) instanceof  (Class/Interface type)
```

如果instanceof运算符左侧变量代表的对象是右侧类（包括其所有子类）或接口的一个对象，那么结果为真。

下面是一个例子：

```
String name = "James";
boolean result = name instanceof String; // 由于 name 是 String 类型，所以返回真
```

另外一个例子：

```
class Drink {
}
class Cola extends Drink {
}

public class InstanceofDemo {
    public static void main(String[] args) {
      Drink drink = new Drink();
```

```
        Drink drinkCola = new Cola();
        Cola cola = new Cola();
        //    Cola colaDrink = new Drink();编译错误
        System.out.println(drink instanceof Drink);//true
        System.out.println(drink instanceof Cola);//false
        System.out.println(drinkCola instanceof Drink);//true
        System.out.println(drinkCola instanceof Cola);//true
        System.out.println(cola instanceof Drink);//true
        System.out.println(cola instanceof Cola);//true
    }
}
```

- 其他运算符

```
?:, ., (), []
```

实际编程中经常会遇到多个运算符一同使用的情形，例如int value = 3*5-10/2+3，那么这时各个运算符优先级如何确定呢，Java中运算符的优先级排列如表4所示，其中排在上面的优先级高于下面的。

类别	操作符	结合顺序
其它	() [] . (点操作符)	从左到右
一元	+ - ! ~	从右到左
算术	* / %	从左到右
算术	+ -	从左到右
移位	>> >>> <<	从左到右
关系	>> = << =	从左到右
相等	== !=	从左到右
按位与	&	从左到右
按位异或	^	从左到右
按位或	|	从左到右
逻辑与	&&	从左到右
逻辑或	||	从左到右
条件	? :	从右到左
赋值	= + = - = * = / =%= >> = << =& = ^ = | =	从右到左

表4 Java运算符优先级

注：结合是指运算符与相邻运算数（运算符）结合的顺序，通常都是从左到右。从右向左的运算符最典型的就是负号，例如3+-4，表示3加-4，即符号首先和运算符右侧的内容结合。

其实在实际的开发中，不需要去记忆运算符的优先级别，也不要刻意使用运算符的优先级别，对于不清楚优先级的地方可直接使用小括号显示指明优先级，示例代码：

```
    int value = 3*5-10/2+3;
    int value = (3*5)-(10/2)+3; //这样更直观
```

这样做便于代码编写，也便于代码的阅读和维护。

1.5.5 程序流控制

Java中的程序控制语法与C/C++等语言基本一致，主要有包括 **if-else，for，continue，break，return，while，do-while**等关键字。下面逐一介绍：

1.5.5.1 if-else

```
        public static void main(String[] args) {

            int a = 3;
            int b = 4;

            if (a > b) {
                System.out.println("a 大于 b");
            }
            else {
                System.out.println("a 不大于 b");
            }

            if (a > b) {
                System.out.println("a 大于 b");
            }
            else if (a < b ){
                System.out.println("a 小于 b");
            }
            else {
                System.out.println("a 等于 b");
            }

        }
```

以上代码的执行结果是打印以下内容：

```
    a 不大于 b
    a 小于 b
```

1.5.5.2 for、continue、break、return

for用作循环，它有两种形式，一种形式是：

```
    for (int i = 0; i < 3; i++) {
      System.out.println(i);
    }
```

以上代码的执行结果是打印以下内容：

```
0
1
2
```

另一种形式是：

```
List<String> flowerNameList = Arrays.asList("rose","chrysanthemum");
for (String flowerName : flowerNameList) {
  System.out.println(flowerName);
}
```

以上代码的执行结果是打印以下内容：

```
rose
chrysanthemum
```

continue、break、return这几个关键字用于精确流程控制，可用于for, while, do-while循环代码块中，continue表示当满足某种条件时，结束本次循环，继续执行下次循环。break表示当满足某种条件时，结束本次循环，并跳出循环代码块，不再执行本循环块中剩余的循环。return则表示结束本次循环并终止本方法中剩余代码的执行，返回方法的调用处位置。以下分别使用代码演示：

```
        for (int i = 0; i < 3; i++) {
            if (i == 1) {
              continue;
            }
            System.out.println(i);
        }
        System.out.println("continue test finish");
```

以上代码的执行结果是打印以下内容：

```
0
2
continue test finish
```

```
            for (int i = 0; i < 3; i++) {
                if (i == 1) {
                  break;
                }
                System.out.println(i);
            }
            System.out.println("break test finish");
```

以上代码的执行结果是打印以下内容：

```
    0
    break test finish
```

```
            for (int i = 0; i < 3; i++) {
                if (i == 1) {
                  return;
                }
                System.out.println(i);
            }
            System.out.println("return test finish");
```

以上代码的执行结果是打印以下内容：

```
    0
```

1.5.5.3 while、do-while

whille、do-while循环与for循环类似，只不过它只需要一个循环条件，例如

```
            int i = 0;
            while (i < 3) {
                System.out.println(i);
                i++;
            }
```

以上代码的执行结果是打印以下内容：

```
0
1
2
```

```
        int i = 0;
        do {
            System.out.println(i);
            i++;
        } while (i < 3);
```

以上代码的执行结果是打印以下内容：

```
0
1
2
```

1.5.6 异常

Java中的异常（Exception）用于标记程序中可能发生的一些错误，异常可分为两大类，分别叫运行时异常（RuntimeException）和编译时异常(Checked Exception)。

它们的不同之处在于，运行时异常不一定要在代码中处理，而编译时异常则是必须要的。

1.5.6.1 运行时异常

下面是一个运行时异常的例子：

```
public class ExceptionDemo {

    public static void main(String[] args) {

        System.out.println(toLowerCaseWarpper("Abc"));
        System.out.println(toLowerCaseWarpper(null));

    }

    /**
     * 将参数字符串中的大写字母全部转为小写
```

```
         * @param str 待转换字符串
         * @return 转换后的结果
         */
        public static String toLowerCaseWarpper(String str) {
            if (str == null) {
                throw new NullPointerException("字符串参数为空。");
            }
            return str.toLowerCase();
        }

    }
```

上面代码的执行结果是打印以下内容：

```
    abc
    Exception in thread "main" java.lang.NullPointerException: 字符串参数为空。
      at
    com.mfw.cc.course1.demo.ExceptionDemo.toLowerCaseWarpper(ExceptionDemo.java
    :23)
      at com.mfw.cc.course1.demo.ExceptionDemo.main(ExceptionDemo.java:11)
```

当第一次调用toLowerCaseWarpper方法时一切正常，第二次调用时，由于传入的参数主空值（null），从而引发了NullPointerException异常的抛出（throw）。NullPointerException是一个运行时异常，在调用可能抛出它的方法时，如果不想处理或者无法处理这种异常，可以不用catch处理。

1.5.6.2 编译时异常

与运行时异常不同的是，调用一个会抛出编译时异常的方法的时，如果没有对异常进行处理，那么编译器就会报错。例如，以下代码：

```
        public static byte[] getUTF8Bytes(String str) {
           //编译时下面这行会报错：Unhandled exception type
    UnsupportedEncodingException
            byte[] bytes =  str.getBytes("UTF-8");
            return bytes;
        }
```

由于getBytes()方法可能会抛出异常UnsupportedEncodingException，UnsupportedEncodingException是一个编译时异常，而代码中并没有对其处理，因而编译会失败。

解决方式有两种，一种是在调用它的方法签名后声明throws异常列表，表示程序知道可能会发生异常，但不处理，只作传递给上层调用者。示例代码如下：

```
    public static byte[] getUTF8Bytes(String str) throws
UnsupportedEncodingException {
        byte[] bytes =  str.getBytes("UTF-8");
        return bytes;
    }
```

另一种处理方式是，把调用的地方放进try-catch异常捕获代码块进行处理。示例代码如下：

```
    public static byte[] getUTF8Bytes(String str) {
        byte[] bytes = null;
        try {
            bytes = str.getBytes("UTF-8");
        } catch (UnsupportedEncodingException e) {
          //捕获到异常后进行相应处理，在这里是打印出异常堆栈
            e.printStackTrace();
        }
        return bytes;
    }
```

1.5.7 泛型

泛型（Generic type 或者 generics）是从JDK 1.5开始支持的一个功能特性，泛型是对对象类型的参数化。泛型可用于类、接口和方法的定义中，分别被称为泛型类、泛型接口、泛型方法。

先来看一个泛型类的例子，

```
    public static void main(String[] args) {

        List list = new ArrayList();
        list.add("string1");
        list.add("string2");
        list.add(1609);

        for(int i = 0; i < list.size(); i++) {
            String str = (String) list.get(i);
            System.out.println(str);
        }

    }
```

运行以上代码会引发异常，异常信息如下：

```
string1
string2
Exception in thread "main" java.lang.ClassCastException: class
java.lang.Integer cannot be cast to class java.lang.String
```

显然，List中出现了非字符串类型的元素，导致后面转型成String时出错。

像这种低级的错误能不能在编译期就发现呢？答案是肯定的，使用泛型可以做到。

对以上代码使用泛型进行改造，代码如下：

```
    public static void main(String[] args) {

        List<String> list = new ArrayList<String>();
        list.add("string1");
        list.add("string2");
        //下面这句代码如果不注释掉，将会提示编译错误
        //list.add(1609);

        for(int i = 0; i < list.size(); i++) {
            String str = list.get(i);
            System.out.println(str);
        }

    }
```

运行以上代码，现在错误没了，因为编译时已经发现了错误，而这一切利益于接口List和其实现类ArrayList对泛型的支持，运行结果如下：

```
string1
string2
```

接着演示一个泛型方法的例子，代码如下：

```
        public static void main(String[] args) {

            String str = genericMethod("blah");
            System.out.println("泛型方法返回结果: "+str);
            Integer i = genericMethod(80);
            System.out.println("泛型方法返回结果: "+i);

        }

        public static <T>  T genericMethod(T anyObject) {
            System.out.println("传入的参数类型
    为: "+anyObject.getClass().getSimpleName());
            return anyObject;
        }
```

以上代码的执行结果是打印以下内容:

```
    传入的参数类型为: String
    泛型方法返回结果: blah
    传入的参数类型为: Integer
    泛型方法返回结果: 80
```

通过这个例子可以看出，通过将某些方法泛型化，容易编写出类型无关的通用模板，可以使方法专注于实现与类型无关的逻辑。

由以上介绍可知，使用泛型的好处是能在编译期发现类型不匹配错误，能通过实现通用化模板代码大幅减少重复编码。

1.5.8 多线程

以上的代码示例中我们都是通过一个main函数来运行的，当运行时，JVM会自动启动一个名称main的线程去运行代码。这个线程被称为主线程，当然，在编程中也可以启用一个新的线程的执行某些代码，Java中多线程编程相当容易实现，可通过以下两种方式实现。

方式一，继承Thread类，Thread类是JDK中表示线程的类。

```
    class CounterThread extends Thread {

        private int counter;

        public CounterThread(String threadName, int counter) {
```

```
            super();
            this.setName(threadName);
            this.counter = counter;
        }

        public void run() {
            for (int i = 0; i < counter; i++) {
                    System.out.println("["+getName()+"] count to " + i + "/" +
    counter);
            }
            System.out.println("["+getName()+"] count to "+counter+"
    finished");
        }

    }

    public class ThreadDemo {
        public static void main(String[] args) {

            new CounterThread("counter3Thread",3).start();
            new CounterThread("counter5Thread",5).start();

        }
    }
```

运行以上代码，可能会出现以下结果：

```
    [counter3Thread] count to 0/3
    [counter5Thread] count to 0/5
    [counter3Thread] count to 1/3
    [counter3Thread] count to 2/3
    [counter3Thread] count to 3 finished
    [counter5Thread] count to 1/5
    [counter5Thread] count to 2/5
    [counter5Thread] count to 3/5
    [counter5Thread] count to 4/5
    [counter5Thread] count to 5 finished
```

上面代码通过创建两个CounterThread对象，每个CounterThread对象都是一个线程，通过在其上调用start()方法来启动这个线程分别执行计数打印。需要说明的是，最终输出结果的顺序是不确定的，因为两个线程的执行时机是无法预知，但可以确定的是每个线程的输出结果是有序的。

方式二，实现Runnable接口类，并用它直接创建一个Thread类。

```
public class CounterRunnable implements Runnable {

  private int counter;

  public CounterRunnable(int counter) {
    super();
    this.counter = counter;
  }

  @Override
  public void run() {
    for(int i = 0; i < counter; i++) {
      System.out.println(i+"/"+counter);
    }
        System.out.println("count to "+counter+" finished");
  }

}

public class ThreadDemo {
   public static void main(String[] args) {
        CounterRunnable counter3Runnable = new CounterRunnable(3);
        CounterRunnable counter5Runnable = new CounterRunnable(5);
        new Thread(counter3Runnable).start();
        new Thread(counter5Runnable).start();
    }
}
```

运行以上代码，可能会出现以下结果（输出结果的顺序是不确定的，原因同方式一中的说明）：

```
0/3
0/5
1/3
1/5
2/3
2/5
3/5
count to 3 finished
4/5
count to 5 finished
```

与方式一不同的是，这里使用的继承Runnable接口的实现多线程，接受一个Runnable的实现，并在线程启动后（通过start()方法）执行其中的代码。这种方式是推荐的方式，它可以实现线程控制与执行逻辑的解耦。

多线程编程是大型应用程序开发中常用的一项技术手段，在实际应用中，JDK标准类型已经提供了很多创建和使用线程的工具类，它们在java.util.concurrent包中，使用它们能很方便地进行多线程编程。同时，需要注意的是，多线程编程是一项很复杂的工作，在使用多线程编程时必须对锁、线程同步、内存屏障、指令重排序等机制有一定了解，否则很容易引发很多意想不到的错误。

2. 代码设计原则

2.1 极简主义原则

代码极简主义的含义是在能达到预期结果的前提下使代码保持最简洁明了。

这样做的成本是：会多花费一些时间构思、对比各种实现方案。 这样做的收益是：能保持代码量最小，便于维护，避免复杂性过高带来的潜在代码漏洞以及维护困难。

2.2 不重复发明轮子原则

在做程序开发之前，应先多了解团队内部、开源社区等是否已有代码资源可以用于项目开发中，如果已有，则应首先考虑使用它们。

这样做的成本是：需要预先沟通、学习来了解一些已有类库API。 这样做的收益是：可以避免重复性劳动、提高开发效率。

2.3 开闭原则

开闭原则是指软件实体如类、方法等应当对扩展开放，对修改关闭，其含义是一个软件实体应该通过扩展已有代码来实现需求变化，而不是通过修改已有的代码来实现。扩展的方式可以是继承、聚合等。

开闭原则是OOP思想最基础的设计原则。

这样做的成本是：编码之前需要对待解决问题有比较全面、深入的分析，以便抽象出代码层次结构，明确开与闭的部分。 这样做的收益是：可以提高代码复用性，避免重复性开发，减少由于直接修改原代码引发错误的风险。可以提高代码可维护性，使后续的代码维护人员无须通读已有代码即可实现代码扩展。

2.4 依赖倒转原则

实际编程中，都会存在高层模块调用低层模块代码的情况，通常的做法是高层代码通过直接创建低层代码对象来完成，

但这样做造成了两者之间的强绑定关系。依赖倒转原则要求低层代码尽量都要有抽象类或者接口（或者两者都有），高层代码则只依赖低层代码的抽象接口，例如List list = new ArrayList();

这样做的成本是：要为每个暴露给外部使用的功能主体提供抽象接口。 这样做的收益是：高层对低层实现的变化是无感知的，实现了功能解耦。

3. 最佳实践

3.1 合理地使用包组织代码结构

Java工程代码应该使用按照设计层次划分成不同的包（package），这样不仅能体现出工程结构层次清晰，还能充分利用包作用域对代码进行安全防护。

3.2 恰当地使用类、方法作用域

类、方法的作用域是用来作访问控制的，这意味着当程序员编写一个Java类时，可以选择哪些其它类拥有该类、方法的使用权，这种访问控制在大型多人协作的工程中尤为重要。

一个类或者方法是内部使用还是对外提供服务，对类作者的职责要求是完全不一样。其中一个最直接的问题是，当一个对外提供服务的类在后续维护、更新时都要考虑上游使用者的兼容性与升级问题。

3.3 识别并使用代码设计模式

设计模式是一套被反复广泛使用过代码设计经验和技巧的总结，它的价值在于使用这些设计模式编写的代码是可重用性强、高效率、易于理解、便于扩展的。因此，当进行代码设计时如果能使用合适的设计模式，将会提升代码品质，但这种能力需要长期学习和积累，所以最好从开始阶段就做起。

3.4 融入开源世界

Java生态有大量优秀的开源软件包，其中很多都是经历了多年不断地技术演进，已趋于成熟，使用它们进行项目开发可以减少重复的编码，提高生产效率。由于这些软件都是开源的，开发者很容易对其实现原理和源码进行研究，并可以尝试对其进行优化和扩展，这也是开源的价值观所在，有想法或者疑问还可以与开源社区互动，这是Java世界的一个魅力之处。

所以，学习、使用、分享，去融入Java开源世界。

3.5 使用相对通用的软件包版本

不管是JDK还是各种开源软件包，版本兼容性问题是一个长期困扰开发者的难题，在同一项目引入多个包，可能引发相互之间版本不兼容，最终导致项目无法按期进行。所以在使用开源软件包时，一定要确认相互之间的兼容性，选择互相支持的版本组合，还有就是不要随意升级某一个软件包版本。

3.6 代码评审常态化

代码评审是保证代码质量的重要手段，应将其作为软件开发的一个重要环节，全面落实。实际执行中，最好制定出代码评审规范，使之标准化，流程化。