Java 应用技术学习笔记

RandomStar

目录

1：Introduction

1.1 Java 语言的特性

- 简洁
  - 没有 C/C++ 中的指针，没有多重继承和运算符重载的语法特性
- 面向对象
  - Java 设计的核心是如何复用代码，并且支持继承封装多态
- Interpreted
  - 代码会编译成字节码 (bytecode) 在 Java 虚拟机上运行
  - Java 是跨平台的，可以在所有有 JVM 的计算机上运行，JMV 可以设置不同的安全等级
- 鲁棒性
  - 强类型机制，异常处理，垃圾内存自动搜集等机制

1.2 JDK 的版本

- Java Standard Edition (J2SE)
  - 可以用于开发客户端应用和 app
- Java Enterprise Edition (J2EE)
  - 可以用于开发服务端应用，比如 Java Servlets
- Java Micro Edition (J2ME)
  - 开发手机应用
- 本课程中的内容主要是 **J2SE**

1.3 Java 程序的组成

- 一个简单的 Java 程序

```
public class HelloWorld{
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

- Java 代码编译的过程
  - Java 代码文件的后缀名是.java，被 Java 编译器编译之后编译成.class 文件，之后在 JVM 中运行
- 每个 Java 程序至少需要一个 class，每个 class 都有唯一的 class name
  - block 是 java 程序的组件，比如一个类就是一个 class block
- Java 程序中的错误
  - 语法错误 Syntax Error：会在编译期间被检查出来，因此有语法错误的时候无法通过编译
  - 运行时错误 Run time Error：
  - 程序逻辑上本身的错误

□ 作业里一道莫名其妙的题目

- 对于下面的这样一个 Java 程序，它**可以通过编译**但是运行的时候会因为**没有入口 main 函数**而出现错误
  - 这是因为 Java 的编译器会把类编译成 class 文件，下面这个类是一个合法的类，没有语法错误，因此可以通过编译
  - 但是在 JVM 中执行的时候会因为没有入口 main 函数而出现错误

```
public class main {
    public void main() {
        System.out.println("Hello World!");
```

```
    }
}
```

2. Java 的基本语法

2.1 Java 的变量命名

- 变量：包含大小写字母，数字，下划线和 $ 符号
  - Java 中的变量不能用数字开头，也不能是保留字
- 变量的定义和声明
  - Java 中**不区分变量的声明和定义**，这是和 C/C++ 的最大不同
    - C/C++ 中的编译器**不会为基本类型赋予初始默认值**
  - Java 对于方法的局部变量，Java 以**编译时错误**来保证**变量在使用前都能得到恰当的初始化**
    - 但是 Java 对于方法内的单个变量不会赋予初始值，对于数组会赋予默认的值
- 对于下面这样一段代码
  - 输出的结果是 0，因为**变量进行了初始化但是没有赋予具体值**
  - Java 中 int 类型的默认值就是 0

```java
public class JavaPractice {
    static int array[] = new int[5];
    public static void main(String a[]) {
        System.out.println(array[0]);
    }
}
```

- **常量**定义 `final datatype CONSTANTNAME = VALUE;`
  - final 的主要用法有如下三种
    - 修饰变量，表示变量的值**不能改变**，但是可以进行任何合法形式的初始化，相当于 C/C++ 中的 const 类型

        - 修饰类对象，表示这个变量不能再赋值成其他的对象，比如一个对象被 new 了之后，就不能再把它 new 成一个新的对象
    - 修饰方法 method，表示一个 Java 函数不可更改，**不能被重载**
    - 修饰类，表示这个类**不能被继承**，类中的所有方法也就变成了 final 类型的
- Java 中的变量类型
  - 数值型变量：Java 中整数类型的范围和**运行的机器无关**，这一点和 C/C++ 不同
    - byte 8bit 有符号类型
    - short 16bit 有符号类型
    - int 32bit 有符号类型
    - long 64bit 有符号类型
    - float 和 double 分别是 32bit 和 64bit 的 IEEE754 标准
  - **字符类型 char**
    - 编码方式是 Unicode
    - Java 中的一个 char 类型变量占 16bits，也就是 2 个字节
- 数值类型的读入
  - 首先需要定义 Scanner 读入器 `Scanner input = new Scanner(System.in)`
  - 之后可以用 input 的 `nextInt` 方法读取下一个整数，其他数据类型同理
- Java 中的数值运算：支持加减乘除取余等各种操作
  - 除法中如整数除法的结果是整数，浮点数除法的结果浮点数
  - 浮点数中 double 比 float 更加精确，double 精确到 16 位，float 精确到 8 位小数
  - 几个特殊的浮点数值
    - `Double.POSITIVE_INFINITY` 正无穷大
    - `Double.NEGATIVE_INFINITY` 负无穷大

    * `Double.NaN` 不是数字
    * 可以用 Double.isNaN 来判断是否为数字

- 类型转换：byte，short，int，float，double，long，char 之间可以进行类型转换
  - 其中整形向浮点型的转换可能会造成精度的损失
  - 浮点数向整形转换的时候会丢弃小数部分
  - Java 不支持 C++ 中的自动强制类型转换，有需要的类型转换必须显式地声明

2.2 选择语句

- 布尔类型变量 (boolean) 值只有 true 和 false，Java 中的大小关系符和 C/C++ 一致

2.2.1 if 语句和 switch 语句

- 和 C/C++ 基本一致，没啥好学的
- if 语句有单个 if，if-else，多层嵌套 if 等写法，**else 和最近的 if 匹配**
- switch 语句也跟 C/C++ 基本一致，有 break 有 default
  - 布尔类型不能用在 switch 的选择里，下面的代码是错误的

```
boolean x;
switch(x) {
        //
}
```

2.2.2 逻辑运算符

- Java 中有如下逻辑运算符
  - ！逻辑否，&& 逻辑且，|| 逻辑或，**^ 逻辑异或**
  - && 和 || 的运算按照短路的方式来求值，如果第一个操作数已经可以确定表达式的值，后面的就不需要进行运算了
  - & 和 | 也可以进行逻辑运算，区别是这两个不用短路的方式来求值

优先级	运算符分类	结合顺序	运算符
由高到低	分隔符	左结合	. [] () ; ,
	一元运算符	右结合	! ++ -- - ~
	算术运算符 移位运算符	左结合	* / % + - << >> >>>
	关系运算符	左结合	< > <= >= instanceof(Java 特有) = = !=
	逻辑运算符	左结合	! && || ~ & | ^
	三目运算符	右结合	布尔表达式?表达式1:表达式2
	赋值运算符	右结合	= *= /= %= += -= <<= >>= >>>= &= *= |=

图 1: image-20200708231822126

2.3 数学函数

- 常见的两个常数：PI 和 E 表示圆周率和自然对数的底数
- 常见的数学类方法
  - 三角函数
  - 幂
  - 高斯函数和舍入方法
  - 最大，最小，绝对值和随机

2.4 字符和字符串类型

- Java 中的字符类型有两种表示方式，ASCII 编码和 Unicode 编码模式
  - Unicode 编码模式的形式是前缀\u+ 四位十六进制数，可以表示从 0000 到 FFFF 一共 65536 个字符
  - 常见的 ASCII 码
    - ′0′ - ′9′ 在 ASCII 码中式 48 到 57
    - ′A′ - ′Z′ 式 65-90，′a′-′z′ 是 97-122
- 字符串内置方法

`isDigit(ch)`	Returns true if the specified character is a digit.
`isLetter(ch)`	Returns true if the specified character is a letter.
`isLetterOfDigit(ch)`	Returns true if the specified character is a letter or digit.
`isLowerCase(ch)`	Returns true if the specified character is a lowercase letter.
`isUpperCase(ch)`	Returns true if the specified character is an uppercase letter.
`toLowerCase(ch)`	Returns the lowercase of the specified character.
`toUpperCase(ch)`	Returns the uppercase of the specified character.

图 2:

- 字符串类型 String

- 一些简单的内置方法

`length()`	Returns the number of characters in this string.
`charAt(index)`	Returns the character at the specified index from this string.
`concat(s1)`	Returns a new string that concatenates this string with string s1.
`toUpperCase()`	Returns a new string with all letters in uppercase.
`toLowerCase()`	Returns a new string with all letters in lowercase.
`trim()`	Returns a new string with whitespace characters trimmed on both sides.

图 3:

`equals(s1)`	Returns true if this string is equal to string `s1`.
`equalsIgnoreCase(s1)`	Returns true if this string is equal to string `s1`; it is case insensitive.
`compareTo(s1)`	Returns an integer greater than `0`, equal to `0`, or less than `0` to indicate this string is greater than, equal to, or less than `s1`.
`compareToIgnoreCase(s1)`	Same as `compareTo` except that the comparison is case insensitive.
`startsWith(prefix)`	Returns true if this string starts with the specified prefix.
`endsWith(suffix)`	Returns true if this string ends with the specified suffix.

图 4:

- 读取字符串的方式：
  - 使用 next() 从有效字符开始扫描，遇到第一个分隔符或者结束符的时候结束，将结果作为字符串返回
  - nextLine 扫描当前行所有的字符串作为结果返回
- 获取字串，使用 `substring` 方法，必须要有的参数是起始位置 beginIndex，结束位置 endIndex 可以缺省，默认值是字符串末尾
- 访问单个字符和子串的位置
- 字符串类型是不可变的，不能对 String 中的内容做出改变，同时如果在函数中对 String 进行赋值操作也不能改变主函数里的 String，比如下面这样一段代码，最后的输出还是 A,B

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

`indexOf(ch)`	Returns the index of the first occurrence of `ch` in the string. Returns `-1` if not matched.
`indexOf(ch, fromIndex)`	Returns the index of the first occurrence of `ch` after `fromIndex` in the string. Returns `-1` if not matched.
`indexOf(s)`	Returns the index of the first occurrence of string `s` in this string. Returns `-1` if not matched.
`indexOf(s, fromIndex)`	Returns the index of the first occurrence of string `s` in this string after `fromIndex`. Returns `-1` if not matched.
`lastIndexOf(ch)`	Returns the index of the last occurrence of `ch` in the string. Returns `-1` if not matched.
`lastIndexOf(ch, fromIndex)`	Returns the index of the last occurrence of `ch` before `fromIndex` in this string. Returns `-1` if not matched.
`lastIndexOf(s)`	Returns the index of the last occurrence of string `s`. Returns `-1` if not matched.
`lastIndexOf(s, fromIndex)`	Returns the index of the last occurrence of string `s` before `fromIndex`. Returns `-1` if not matched.

图 5:

```
        String a = new String("A");
        String b = new String("B");
        swap(a, b);
        System.out.println(a + "." + b);
    }
    static void swap(String x, String y) {
        y = x;
    }
}
```

  - String 的 concat 和 substring 等方法都不是在原来的字符串上操作的，而是生成了一个新的字符串

- 格式化输出
  - `System.out.printf(format, items);` 具体的用法和 C 语言的 printf 类似
  - 占位符的具体格式包括
    - `%[index$][标识]*[最小宽度][. 精度] 转换符`
    - index 表示从第几个位置开始计算来进行格式化，起始为 1

* 最小宽度是格式化之后最小的长度，当输出结果小于最小宽度的时候用标识符填补空格，没有标识符的时候用空格填充

  · 字符串可用标识：- 表示左对齐，默认的是右对齐

  · 整数和浮点数可用标识：

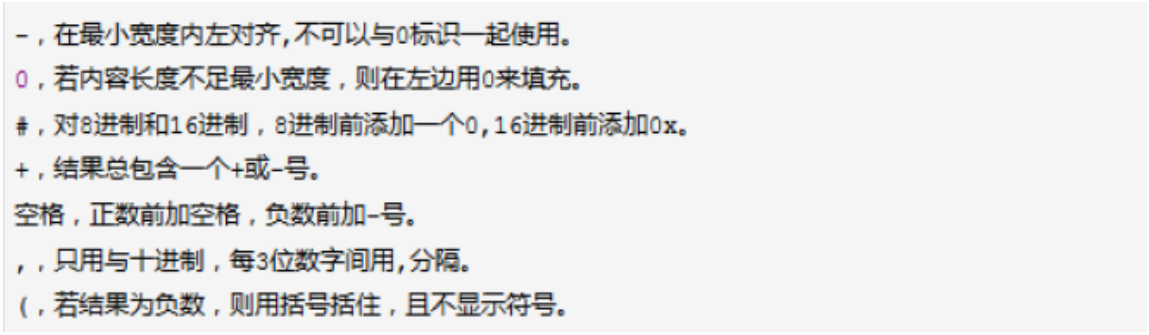

图 6:

  · 对日期进行格式化

* 精度用于设置浮点数保留几位小数

* 转换符用于指定转化的格式，有 f，d 等等

2.5 循环

- 和 C/C++ 基本一致，有 for 循环，while 循环和 do-while 循环
  - 要注意区别 while 循环和 do-while 循环
  - `for(;;){}` 和 `while(true){}` 等价
  - Java 中也有 `break` 和 `continue` 来结束或者跳出循环

2.6 方法

- 将程序中的一部分过程抽象成一个方法，起到代码复用的作用，一个方法的定义包含如下如内容
  - modifier 修饰符，包括方法的 public/private/protected 和是否是 static，final 等属性

  - return value type 返回值类型
  - method name 方法名
  - formal parameters 形式参数
    * 方法名 + 形式参数称为方法的签名
    * 与之相对应的是 actual parameter or argument 实际参数，也就是调用方法的时候使用的参数
    * 实参的内容不会因为方法中的操作被改变
  - body 方法的具体内容
- 方法的重载：相同的方法名，不同的参数表构成重载关系
  - 只有返回值类型不相同的时候不构成重载，这种写法会引起编译错误，不能使用
  - 重载可以有返回值类型的区别，但是一定会有**参数表的区别**
- 局部变量：定义在方法内部的变量叫做局部变量
  - 局部变量的可调用范围是从定义开始到不再包含这个变量的方法内部区域为止 (比如 for 循环中定义的变量只能在 for 循环中调用)

2.7 数组

2.7.1 一维数组

- Java 中数组的定义方式 `datatype[ ] array`，中括号也可以放在后面，然后需要用 `new datatype[size]` 来声明数组的大小
  - 可以用 length 来获得数组的长度，但要注意这个和 String 类型的 `length()` **方法**不一样，数组里的 length 是 state
  - 数组的长度定义之后就**不能改变**
  - 数组的长度确定之后，里面所有元素的值会被设定为默认值
    * 对于数值类型的数组，默认值是 0
    * 对于 char 类型，默认值是`\u0000`

    - 对于 boolean 类型，默认值是 false

  - 数组可以在定义的时候直接赋予若干值，此时数组的长度会被自动设定为值的个数，但是下面这种方式是错误的

```
double[] list;
list = {1, 2, 3, 4};
```

  - Java 数组和 C++ 数组的**区别**

    - Java 的数组定义在**堆**上，C++ 中直接声明大小的数组分配在栈上，动态分配的数组在堆上

    - Java 的 [ ] 运算符被会检查数组边界，防止下标溢出，并且没有指针运算

    - Java 中命令行参数是 `String[] args`，其中 `args[0]` 是第一个参数，**程序名没有存储**在 args 中需要在启动 Java 程序的时候就输入

  - 数组的遍历

    - 用下标 i 去遍历一个数组

    - 用 `elementType value: arrayRefVar` 的形式来遍历

  - 数组的拷贝：`arraycopy(sourceArray, src_pos, targetArray, tar_pos, length);` 自带的处理方法

    - 在拷贝的过程中，如果被拷贝的数组存在多余的元素，则赋以默认值，如果小于原始数组长度，则只拷贝前面的元素

  - 数组作为方法的参数

    - Anonymous Array 匿名数组：在方法调用中的参数里直接写一个数组，没有用变量去引用，如 `new int[]{1,2,3}` 直接作为参数

    - 将数组作为参数的时候，数组作为一个引用传入，方法中改变数组的值将会影响到原来的数组，但是匿名数组没有变量名

2.7.2 多维数组

- 二维数组的定义：`dataType[][] refVar;`

  - 如何 new：`new dataType[rowSize][columnSize]`
  - Java 的二维数组每一列大小可以不同，并不需要完全相同！
    * 比如 `int[][] a = {{1, 2}, {3, 4, 5}};`
- 如何创建一个二维数组？以下四种写法中，只有第 2 行的是对的
  - 总结起来就是中括号可以放在前面也可以放在后面，但是**声明的时候不能直接指定数组的大小**
  - 数组的大小需要在 new 的时候指定，并且二维数组**一定要有行的数目**

```
int a[3][2] = {{1, 2}, {2, 3}, {3, 4}};
int a[][] = new int[3][];
int[][] a = new int[][3];
int[][] a = new int[][];
```

3. Java OOP

3.1 类和对象

3.1.1 类的定义和初始化

- 一个对象拥有状态 (state) 和行为 (behavior) 两个属性，状态定义了对象的内容，表现定义了对象可以进行哪些操作，类就是用于定义同一类对象的
  - Java 中用变量定义状态，用方法定义行为
  - Java 中的类也有**构造函数**，在对象构造的时候调用，但是对象的定义也可以不需要构造函数，当类定义里**没有显式声明的构造函数**时 Java 编译器会自动调用一个 default constructor
- Java 中方法和成员变量的引用方式：`objectRefVar.methodName(arguments)`
  - Reference Data Fields 有**默认值**
    * String 类型是 null，数值类型是 0，布尔类型是 false，char 类型是`\u0000`，但是 Java 对于**方法中的局部变量**不会赋予默认值
    * 比如 `int x` 后直接对其进行 print 会发生编译错误，因为变量没有初始化

  - 和 C++ 的区别
    * 编译器会为这些数据成员进行默认初始化，实际上是把刚分配的对象内存都置 0
    * 在对象里定义一个引用，并且没有初始化的时候，默认为 null
    * Java 中**在默认的初始化动作之后才进行指定初始化**，比如下面这段代码中变量 i 先变成 0 在被赋值为 999
      · 也就是说一定会有用默认值去初始化类成员变量的这个过程
      · 并且 C++ 不支持直接在类定义里给成员变量赋值，但是 Java 可以

```
public class Value {
    int i = 999;
}
```

  - Java 中数据成员的初始化过程是
    * 先进行**默认的初始化**
    * 在进行类定义里的初始化
    * 构造函数初始化

- 对象和基本变量类型的区别
  - 基本变量类型在拷贝的时候只是拷贝一个值
  - 对象在拷贝中改变的是变量对于对象的**引用**
    * 比如 c1=c2 的赋值导致 c1 也指向了 c2 指向的对象，而 c1 指向的对象不再被引用，会触发 JVM 的**垃圾回收**机制
  - 对于不需要使用的对象可以给它赋值为 null，这样就会触发 JVM 的垃圾回收机制释放内存空间

□3.1.2 JVM--垃圾回收机制

- **Java 的垃圾回收机制**
  - 对于不再被引用的对象和被赋值为 null 的对象就会触发 Java 的垃圾回收机制

  - 一般而言如果类自己管理内存，程序员就应该警惕**内存泄漏**的问题
    - 内存泄漏是指没有指针/引用指向一段存内存，导致这部分内存无法被调用和修改
    - 内存泄漏的另一个常见的来源是缓存，把对象引用到了缓存中就容易内存泄漏
    - 缓存可以用**软引用**来实现
      - Java 提供了强引用，软引用，弱引用和虚引用 4 种引用方式
      - 平时我们用的是**强引用**，强引用**只要存在就不会被当作垃圾回收**，真不行了就抛出 OOM 异常，`object = null;` 就会导致原本是强引用被垃圾回收
      - 用软引用关联的对象在即将发生内存异常之前会被列入垃圾回收的范围而被回收，如果内存还是不够才会抛出内存异常 (OOM, out of memory) 的问题
      - 软引用主要应用于内存敏感的高速缓存，在安卓系统中经常用到

3.1.3 其他特性

- instance 和 staic——和 C++ 相同
  - instance 是变量的实例，instance variable 属于特定的实例，instance method 由类的一个实例调用
  - static 类型的变量和方法被一个类的所有成员**共享**
- 值传递和引用传递
  - Java 本质上是一种值传递
  - 对象中的一个数组实际上是一个 reference variable 的数组
- 不可变类 Immutable class
  - Java 自带了很多不可变的类，比如 String 和一些基本的**包装类**
    - 不可变类的优点：更加容易设计，实现和使用，更加安全
  - 编写不可变类的原则
    - 不提供可以修改对象状态的方法 (mutator)

    * 保证类不会被扩展，具体的做法是声明为 `final` 类型
        · 不可变类使所有的 state 都是 final 和 private 的
    * 确保对于任何可变组件的互斥访问
    * 必要时可以进行拷贝保护
  - 不可变对象，只有一种状态，也就是在创建的时候的状态
    * 本质上是**线程安全**的
    * 可以提供**静态工厂**，把频繁被请求的实例缓存起来
    * **缺点**是对于每一个不同的值都需要一个单独的对象
  - 为了确保不可变性，类不允许自己被子类化，除了使类成为 final 之外，还可以让类的所有构造函数都变为私有的或者**包级私有的 (protected)**，并添加公共的静态工厂来代替公有的 constructor
    * 用到了**常量池技术**

```
public class Complex {
    private final double re;
    private final double im;
    private Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }
    public static Complex valueOf(double re, double im) {
        return new Complex(re, im);
    }
}
```

- this 关键字：和 C++ 相同
  - 是**对象本身的引用**的名字，用于访问 hidden data fields
  - 一个常见的用法是在类的 constructor 中去调用其他的 constructor
- 对象构建的 TIPS

- 基于 Builder 构造
  - 遇到多个 constructor 参数时可以考虑用 Builder，可以解决类定义中的参数过多导致需要重载很多次的问题 (导致代码的可读性变差，代码质量也下降)
    - 另一种方案是设置无参数的构造方法，通过 setter 方法来设置参数，但是这样不能保证对象的一致性
    - 这种方法也使得类不能设置为不可变类
- 静态工厂方法代替构造器：代码的可读性更高，不必再每次调用的时候创建一个新的对象，可以返回原类型的任何子类型对象
- 单例对象构建：通常用于那些本质上唯一的对象，如文件系统、窗口管理器等
  - 有特权的客户端可以通过反射机制调用私有构造器，要抵御这种攻击可以修改构造器，要求创建第二个实例的时候抛出异常
  - 枚举类方法：编写包含单个元素的枚举

□3.1.4 包 (Package) 和包管理

- package 可以方便管理和组织 java 文件的目录结构，防止不同文件之间的命名冲突
  - 作为 Java 代码源文件的第一条语句，如果缺省则指定为无名包
  - 编译器在编译源文件阶段不检查目录结构
- 类的导入
  - 可以在每个类前面添加完整的包名
  - 也可以使用 import 语句导入整个包，比如 `import java.time.*`
  - 也可以只导入包中的特定类
- package 静态导入
  - 不仅可以导入类，也可以导入静态的方法和静态域，比如 `import static java.lang.System.*;` 之后就可以使用 `out.println`
- Java 的 import 和 C++ 的 include 的区别

- C++ 中要用 #include 把外部声明加载进来，C++ 编译器只能查看正在编译的文件和 #include 的文件，Java 编译器可以查看其他文件，只是告诉编译器要去哪里查看
- Java 中显式地给出包名的时候不需要 import，而 C++ 要引用别的文件一定要 #include
- package 和 import 更类似于 namespace 和 using

3.2 包装类

- 类和类之间的关系
  - Association 关联
  - Aggregation 聚合
  - Composition 组合
    - 组合的关系比聚合更加紧密
- Wrapper class 包装类
  - 实现了对 Java 中的各类基本数据类型的包装
  - 对象是 immutable 的，创建之后就不能改变
  - 包装类的构造函数有多种
    - 比如整型的包装类 Integer 可以用一个 int 类型来构造，也可以用 String 类型来构造
  - 数值包装类
    - 每个**数值型**的包装类中有常数 `MAX_VALUE` 和 `MIN_VALUE` 分别代表这个数据类型中的**可能的最大值和最小值**
    - 所有的**数值型包装类**实现了向其他数值类型转换的方法，比如 doubleValue，intValue 等等
      - 静态方法 valueOf：参数是一个字符串，产生对应的数值类型的值
      - JDK1.5 以上的版本允许包装类和原本的内置类型进行自动转换
  - BigInteger 类和 BigDecimal 类

    - 用于处理大数值的包装类

- **String 类**的语法特性
    - `immutable` 不可修改类，需要通过 `charAt( )` 方法访问字符串中的单个元素
        - 比如 `String s = "Hello"; s = "Java";` 此时只是新构造出了一个 “Java” 的 String 对象并让 s 指向这个对象，原来的 “Hello” 对象并没有消失
    - 有连接，比较，获取子串，查找元素和子串等多种操作
- 自动装箱：AutoBoxing
    - 基本类型在运算的时候，JVM 会将其装箱成安全的包装类来使用

3.3 Constant Pool 常量池

- Java Constant Pool **常量池**技术
    - 可以方便快捷地创建某些对象，当需要的时候就从池里取出来，常量池实际上就是一个**内存空间**存在于方法区中
    - JVM 将源代码编译成 class 文件之后，会用一部分字节分类存储常量，集中在 class 中的一个区域存放，包含了关于类方法接口中的常量，也包括字符串常量
        - 比如 `String s = "java"` 则在编译期可以识别为和 java 是同一个字符串的，都会自动优化成常量，也就是说如果有多个字符串的值为 java 则他们都会引用自同一 String 对象

```
String s1 = "Hello";
String s2 = "Hello";
String s3 = "Hel" + "lo";
String s4 = "Hel" + new String("lo");
String s5 = new String("Hello");
String s6 = s5.intern();
String s7 = "H", s8 = "ello";
String s9 = s7 + s8;
s1 == s2 true;
s1 == s3 true; // 编译期进行了一定的优化
```

```
s1 == s4 false;
s1 == s5 false;
s1 == s9 false; // s9 是两个变量拼起来的
s1 == s6 true;
```

    * 对于上面这一段代码，s1，s2 和 s3 是相同的 (因为常量池技术) 而 s1 和 s4 是不同的，s1 和 s9 也是不同的，因为 s4 和 s9 不是常量，在编译期没有确定是不是常量
        · 而 s1 和 s6 是相等的，和 s5 是不同的，因为 s5 位于堆中，**intern 方法会试图将 Hello 这个值加入常量池中**，而此时常量池中已经有了 Hello 所以直接返回了其地址

- Java 中定义的**包装类大部分实现了常量池技术**，只有**浮点数类型的包装类没有实现**
    * 其中 Byte,Short,Integer,Long,Character 只对-128-127 的对象使用常量池
        · Double 类型是没有缓存的，所以就会有如下结果

```
Double x = 1.1;
Double y = 1.1;
x == y // -> false!
```

    * Integer 利用**缓存**机制实现了常量池，缓存了 256 个对象，主要是常用的证书

3.4 String 的语法特性

- String 的 intern 方法
    - 需要 1.7 以上的 JDK
    - intern 方法设计的初衷就是要重用 String 对象，节约内存消耗
    - Java 的运行时 Data Area
        * 堆 heap 中存放创建的实例对象
        * 方法区中存储了已经被 JVM 加载的类的信息和静态变量，编译器编译的代码
        * JDK1.7 之后常量池被放入到堆空间中，导致 intern 的功能发生了变化
    - 这里有一部分比较重要的，等开学之后再看

- String 的其他语法
  - String 的匹配，代替和分割 split，支持正则表达式匹配
  - 包装类大多都有 `toString()` 的**静态方法**用来将其转换成 String 类型
  - String 的 format 方法:根据指定的格式生成 String,比如 `String s = String.format(“%7.2f%6d-4s”,45.556, 14, “AB”);`
  - 关于 Java 中 string 的一些细节
    - Java 中的 String **不是以空字符'\0'** 结尾的
    - Java 中的 String 不可改变，是 **final 类型**
    - 在 String 池中维护
    - 比较是否相同的时候要用 equals 方法，不要用 ==
    - 使用 indexOf 等方法去查询元素的位置
    - 使用 subString 方法去获取子串，因为 == 比较的是引用的对象，equals 比较的是 String 所代表的值
    - + 运算完成了对 String 的重载
    - 使用 trim 方法删除首尾空格
    - split 方法支持正则表达式
    - 不要**存储敏感信息**在 String 中

□StringBuilder 和 StringBuffer

- 是 String 的以一种替代品，String 可以使用的地方一定也能用这两个，但是更加灵活
  - Builder 和 Buffer 拥有 toString，capacity，length，setLength 和 charAt 等方法
- 三者的比较
  - 在执行速度上，`StringBuilder>StringBuffer>String`

        * 比如对于 String s = "abcd"，如果我们执行 s = s + 1 实际上执行之后的 **s 跟原本的 s 不是同一个对象**而是生成了一个新的对象 (因为 String 是不可变的)，**原来的对象被垃圾回收了**，导致 String 的执行效率非常低

    - 线程安全

        * Builder 是**线程非安全**的，Buffer 是线程安全的，当有多个线程区使用某个字符串时，StringBuilder 不安全

▫ 正则表达式

- 其实我只记得 * 和 + 这几个符号

3.5 枚举类 Enum

- JDK1.5 才有的新类型，采用 enum 关键字定义，所有的枚举类型都继承自 Enum 类型
    - 通常常量用 `public final static` 来定义，在枚举类中可以用如下方式定义

```
public enum Light{
    public final static RED = 1, GREEN = 2, YELLOW = 3;
}
```

- Java 的枚举**本质上是 int 值**
    - 通过公有的 final 静态域为每个枚举常量导出实例的类
    - 由于没有 constructor，枚举类型是真正的 final，是实例受控的，是单例的泛型化
    - 比如上面这个枚举类的定义，如果定义了一个 Light 类型的变量，那么任何非空对象一定属于 Light 的三个值之一
- 枚举类的特性
    - 枚举类是 final，不能被继承
    - 含有 `values()` 的静态方法，可以**按照声明顺序返回其值数组**
    - `ordinal()` 方法，返回**枚举值在枚举类中的顺序**，根据声明时候的顺序决定 (从 0 开始)
    - 可以用 valueOf 来得到枚举实例，用 toString 将枚举转化为可以打印的字符串

Regular Expression	*Matches*	*Example*
x	a specified character x	Java matches Java
.	any single character	Java matches J..a
(ab|cd)	ab or cd	ten matches t(en|im)
[abc]	a, b, or c	Java matches Ja[uvwx]a
[^abc]	any character except a, b, or c	Java matches Ja[^ars]a
[a-z]	a through z	Java matches [A-M]av[a-d]
[^a-z]	any character except a through z	Java matches Jav[^b-d]
[a-e[m-p]]	a through e or m through p	Java matches [A-G[I-M]]av[a-d]
[a-e&&[c-p]]	intersection of a-e with c-p	Java matches [A-P&&[I-M]]av[a-d]
\d	a digit, same as [0-9]	Java2 matches "Java[\\d]"
\D	a non-digit	$Java matches "[\\D][\\D]ava"
\w	a word character	Java1 matches "[\\w]ava[\\w]"
\W	a non-word character	$Java matches "[\\W][\\w]ava"
\s	a whitespace character	"Java 2" matches "Java\\s2"
\S	a non-whitespace char	Java matches "[\\S]ava"
*p**	zero or more occurrences of pattern *p*	aaaabb matches "a*bb" ababab matches "(ab)*"
p+	one or more occurrences of pattern *p*	a matches "a+b*" able matches "(ab)+.*"
p?	zero or one occurrence of pattern *p*	Java matches "J?Java" Java matches "J?ava"
p{n}	exactly n occurrences of pattern *p*	Java matches "Ja{1}.*" Java does not match ".{2}"
p{n,}	at least n occurrences of pattern *p*	aaaa matches "a{1,}" a does not match "a{2,}"
p{n,m}	between n and m occurrences (inclusive)	aaaa matches "a{1,9}" abb does not match "a{2,9}bb"

图 7:

- 比如 `Light L =  Light.valueOf("RED");`
- 枚举类型也支持 switch 语句直接对实例进行选择
- enum 类型可以可以关联不同的数据，也可以**添加任意的方法和域来增强枚举类型**，比如添加构造函数来丰富枚举类的结构
- **特定于常量**的方法实例
  * 缺点是难以共享代码，可以借助策略枚举，将处理委托给另一个枚举类型

```
public enum Operation{
    PLUS{
        double apply(double x, double y){
            return x+y;
        }
    }
    MINUS{
        double apply(double x, double y){
            return x-y;
        }
    }
    abstract double apply(double x, double y);
}
```

- 这一部分听说期末考会考编程题，需要练练

□□□□□□□□□□□□

3.6 继承和多态

3.6.1 继承

- Java 中继承的关键词是 `extends`
- super 方法：
  - 子类没有继承基类的构造函数，但是子类中可以用关键字 `super` 去调用基类的构造函数

  - 如果不显式地声明 super，则会**自动调用基类的无参数的构造函数**
  - super 关键字可以调用基类的构造方法
- override：覆写父类的方法
  - 子类会继承父类所有的方法，但是可以在子类中对父类的方法进行重载，此时调用子类的该方法就会调用新定义的，覆写了原本的方法
  - **private 类型的方法不能被覆写**
  - static 类型的方法也不能被覆写，如果在子类中被重定义了，那么父类的该静态方法就会被 hidden
  - JDK1.5 开始增加了 `@override` 注解来声明一个覆写——不要忘了写 override
    - 每个类都会有一个 equals 的方法，要注意区分重载和覆写
  - 方法的覆写发生在继承的子类中，而**重载在继承的子类和基类中都会发生**
- Object 类是 java.lang.Object 中定义的基本类型，如果一个类没有声明继承自何处，那么就是继承自 Object 类
  - Object 中有 toString 方法，默认情况下会显示一个实例的类名和 @ 符号后面跟一个代表这个实例的数字
  - ~~万物起源 Object~~

3.6.2 多态 Polymorphism

- 多态意味着基类可以被引用作为一个子类来使用
- Java 的动态绑定 (dynamic binding) 特性
  - 假如一系列类的继承关系如下 (其中 Cn 在 Java 中式 Object 类)

图 8:

  - JVM 会从 C1 开始寻找某个方法 P 直到找到一个具体的实现为止，然后搜索停止，调用第一个找到的方法
  - method match 方法匹配，在**编译期**按照参数类型和个数来进行匹配，而动态绑定是在运行时寻找对应的方法

- Generic Programming 元编程
  - 当一个方法的参数是 superclass 的时候，可以用他的任意一个子类作为参数，但是具体的调用会动态地决定

- Casting objects
  - 可以在有继承结构中的类型之间互相切换，比如对一个参数要求为 Object 的方法，可以用 new Student() 作为其参数，此时会发生从 Student 到 Object 的隐式转换
  - 当从基类转换到子类的时候必须有显示的声明，但不一定总是能成功
    - `instanceof` 操作符可以测试一个对象是不是某个类的实例
  - Java 和 C++ 的区别
    - Java 转换失败的时候会抛出异常，而 C++ 转换失败会产生一个 nullptr，这样就会在一个操作中完成测试和类型转换
    - Java 中的类型转换需要和 instanceof 结合使用，先用 instanceof 判断，然后来进行转换

- equals 方法
  - 默认的 equals 方法在 Object 中是这样定义的

```
public boolean equals(Object obj) {
    return this == obj;
}
```

  - 可以在自定义类中 override 这个 equals 方法
  - == 比 equals 的要求更高，会检测两个引用变量是不是引用的同一个对象，而 equals 只要求内容相同

3.7 Java 的内置模板类

- ArrayList 任意长度的任意类型数组
  - 可以用 `ArrayList<typename>` 来定义一个 typename 类型的任意长数组
  - 相比于普通数组，拥有更多功能比如 add，remove 和 clear，和普通数组之间可以互相转换
  - ArrayList 可以用 `java.util.Collections.max` 和 `min` 方法来获取其中的最大值
  - 此外 java.util.Collections 中还有 sort 和 shuffle 等方法
- MyStack 类
  - 一个内置的 Stack 结构，支持栈的 pop，push，peek 等多种操作

3.8 关键词 protected

- protected 关键词表示一个类或者数据或者方法可以**被同一个包或者子类任意访问**
- Java 中状态和方法可访问的范围从小到大是 private，default，protected，public
  - private 只能在同个类中访问
  - default 可以在 private 的基础上同个包中访问
  - protected 在此基础上可以被子类访问
  - public 可以被随便访问
- 继承的时候子类在覆写方法的时候不能把 Accessibility 弱化，只能往上调

3.9 嵌套类

3.9.1 复合优先于继承

- 继承打破了代码的封装性，子类依赖于基类中特定功能的实现，如果基类随着版本变化会破坏子类
- 复合 composition 就是不继承而是在新的类中增加一个 private state 并引用现有类的一个实例

3.9.2 嵌套类 nested class

- 可以在**类的内部再定义一个类**，分为**静态嵌套类**和**非静态嵌套类**
  - 其中非静态嵌套类最重要，也被称为内部类 inner，分为
    - 在一个类中直接定义的内部类
    - 在方法中定义的局部类 `local class`
    - 匿名内部类 `anonymous class`
  - 内部类的作用：内部类之间的 **private 方法可以共享**因此经常被作为辅助类
  - public 内部类可以在外部类之外调用，但是对于非静态类，必须要使用一个外部类的对象来创建
  - 内部类如果在外部类可以直接 new，但是在外部类的外面使用需要**先 new 出一个外部类，在用外部类 new 出一个内部类**
  - 静态内部类不需要通过外部类的对象来创建，外部类的变量和非静态方法都不能调用
  - 有内部类的时候在编译成 class 文件的时候会产生多个 class 文件，**每有一个类定义就会产生一个 class 文件**
- Local class 局部类
  - 定义在一个方法内部，只能在方法的内部实例化
  - 方法内部类的对象不能使用该内部类所在方法的非 final 局部变量
  - 方法的局部变量位于栈上，只存在于该方法的生命期内
    - 但是该方法结束之后，在方法你创建的内部类对象可能仍然存在于堆中
  - 只有 final 和 abstract 可以用来修饰方法内部类
  - 静态方法内的方法内部类只能放为外部类的静态成员

3.10 抽象类和接口 Abstract class and Interfaces

- Interface 接口：用来定义各种类的表现

3.10.1 Abstract class 抽象类

- 抽象类的语法特性
  - 抽象类**不能实例化出对象**，抽象方法**不能在非抽象类中使用**
  - 抽象类不能使用 new 操作符，但是依然可以定义构造方法并在子类中调用
    - 抽象类的非抽象子类可以创建对象
    - 可以作为一种数据类型
  - Java 中的抽象类表示一种继承关系，一个类**只能继承一个抽象类**
  - 抽象方法一定属于抽象类，抽象类不一定需要有抽象方法
  - 子类也可以是抽象类，不管是继承了一个抽象类还是一个具体的类
- 抽象方法
  - 只有方法体，没有方法名
  - 继承了抽象方法的子类必须 override 这个方法，否则这个子类也必须声明为抽象类，最终必须有子类 override 这个方法，否则这些类定义都不能实例化出对象

3.10.2 Interface 接口

- 接口只包含常数和抽象方法，用于指定对象的通用行为，定义的方法如下
  - 接口是抽象方法的集合

```
public interface InterfaceName {
        constant declarations;
        abstract method signatures;
}
```

- 接口也不能实例化出一个对象，但是接口名**可以作为变量类型**来使用
  - 抽象类中所有的数据都是 `public final static` 类型，所有的方法都是 `public abstract` 类型
  - 接口中的方法不是在接口中实现的，只能由实现接口的类来具体实现接口中的方法

  - JDK1.8 以后接口里可以写静态方法和方法体
  - 接口可以继承其他的接口
- 实现接口的关键字：用 `implements` 关键字来让一个类实现接口中的方法
  - 如果实现接口的类是抽象类，就可以先不用实现
  - 一个类可以有多个接口，但是只能继承自一个类
- marker interface 标记接口
  - **没有任何方法和属性**的接口，仅仅表明它的类属于某个特定的类型，这个接口只起到了标记作为
  - 主要用于
    * 建立一个公共的父接口
    * 向一个类添加数据类型
- 接口和抽象类的区别
  - 抽象类的方法可以有方法体
  - 抽象类中的成员变量可以实各种类型的，接口中只能是 `public static final` 类型
  - 接口中不能含有静态代码块和静态方法
  - 一个类只能继承一个抽象类，但是可以实现多个接口

4. Java 高级语法特性

4.1 异常处理

- Java 中的异常类型
  - 注意异常和错误的区别，系统错误是 JVM 抛出的，异常是由程序引起的，可以被检测出来
- 异常类型

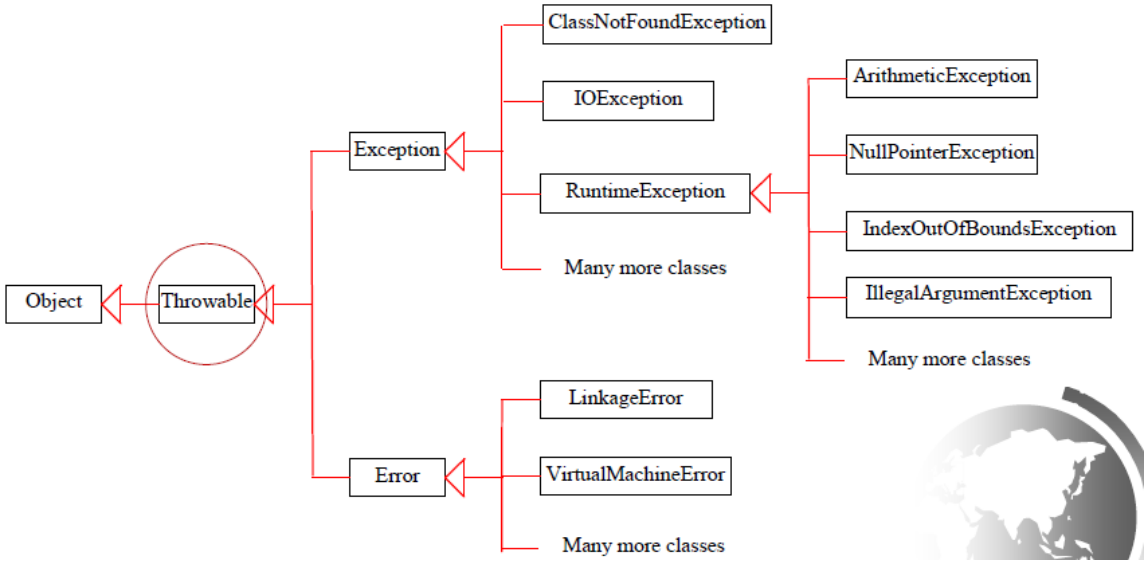

图 9:

  - Unchecked Exceptions 不可恢复的逻辑错误，包括 RuntimeException 和 Error
    - 比如 NullPointerException 和 IndexOutOfBoundsException
    - Java 不需要对 unchecked exceptions 进行检查
  - checked exceptions 需要进行 catch

- Declaring Exceptions 方法需要声明检查的异常的种类，关键字为 throws
- Throwing Exceptions
  - 当发现了错误类型的时候可以创建一个合适的错误类型将其抛出
  - 语法是 `throw new TheException()`
  - try-catch 语句

```
try{
        statements;//Statementsthatmaythrowexceptions
}
catch(Exception1exVar1){
```

```
        handlerforexception1;
}
catch(Exception2exVar2){
        handlerforexception2;
}
catch(ExceptionNexVar3){
        handlerforexceptionN;
}
```

  - tyr-catch 语句块中可以加入 finally 子句，**不管有没有找到错误都会被执行**
    - 抛出异常之后如果没有 finally 就不会再往下执行，有 finally 的时候抛出异常之后还会执行 finally 中的语句

- Java 的异常检查使用规范
  - 不要忽略 check exception：捕获就必须要处理
  - 不要捕获 unchecked exception
  - 不要一次捕获所有的异常
  - 使用 finally 语句块释放资源，但是 final 块不能抛出异常
  - 抛出自定义异常异常时带上原始异常信息
  - 打印异常的时候带上异常堆栈
  - 不要同时使用异常机制和返回值
  - try 不要太庞大，不然代码的可读性会降低
  - 守护线程中需要 catch runtime exception

4.2 Assertion 断言

- 包含一个在程序运行过程中一定是真的布尔表达式，用 `assert assertion` 进行定义
  - 当断言被创建的时候，Java 会计算这个表达式，如果结果是 false 就会抛出 AssertionsError 这个异常

- AssertionError 有多种构造函数，用于匹配 message 的 data type

```
public class AssertionDemo {
public static void main(String[] args) {
        int i; int sum = 0;
        for (i = 0; i < 10; i++) {
                sum += i;
        }
        assert i == 10;
        assert sum > 10 && sum < 5 * 10 : "sum is " + sum;
        }
}
```

- 不要在 public 的方法中使用 assertion 而应该使用 exception handling

4.3 文本读写

- File 类：一个由文件名和路径组成的包装类，提供了一系列文件信息和修改操作
  - 不包含读写文件内容的方法
  - 文件读写需要 Scanner 和 PrintWriter
- PrintWriter 用于写入文件
  - 提供了一系列重载的 print 和 println 方法
  - 构造函数需要文件名作为变量，可以指定编码方式，比如 UTF-8
- Scanner 用于读文件
  - 初始化之后和标准输入一样使用 nextXXX 来读写
  - Scanner 的构造方式
    - 错误方式 `Scanner in = new Scanner("file.txt");` 构造了一个带字符串参数的 Scanner
    - 正确方式 `Scanner in = new Scanner(new File("file.txt"), "UTF-8");`
      - 需要用 File 对象进行构造

- URL 类：从 web 中读取信息
  - 构造方式 `URL url = new URL("www.xxxxxxx");`
  - 完成构造之后可以用 openStream 打开一个输入流，用 Scanner 进行读取
    - `Scanner input = new Scanner(url.openStream);`

4.4 Generics 泛型

- 是一种对变量类型进行参数化的功能，类似于 C++ 中的模板
  - 优点是能在编译期就发现一些错误而不是运行时
  - 一个泛型类或者方法可以声明类名和函数名作为变量

4.4.1 泛型方法

- 泛型方法的类型参数声明要卸载返回值类型前面，用尖括号括起来，比如
  - `public static <E> void printArray(E[] input);`
  - 类型参数可以有多个参数，有逗号隔开，类型参数可以被用来声明返回值类型
  - **有界的**类型参数
    - 用 extends 关键字 + 类型的上界来控制类型的范围
    - 比如 `<T extends Comparable<T>>` 表示 T 必须是可以比较大小的类型

4.4.2 泛型类

- 类型参数的声明添加在**类名的后面**
- 参数化类型没有实际类型参数的继承关系
  - 比如 `List<Integer> list = new List<Object>` 会编译错误
  - 泛型的继承关系需要通过通配符
    - 使用？代替具体类型参数，比如 List<?> 可以代表所有具体类型 List 的父类
    - 比如 `List<? extends Number>` 表示参数的泛型上限为 Number 类型

    * `List<? super Number>` 表示参数的泛型下限为 Number 类型

- 不允许用泛型类型来创建泛型数组
- 泛型类的所有实例都有相同的运行时类，所以泛型类的静态变量和方法是被它的所有实例共享的。所以在静态方法、数据域或初始化语句中，为了类而引用泛型参数是非法的