1. Immutable Objects and Classes（不可变对象和不可变类）

不可变对象（Immutable object）是指一旦创建，其内容就无法被修改的对象。
不可变类（Immutable class）是指如果一个类的所有对象都是不可变的，那么这个类被称为不可变类。
不可变类的例子如下。

```
public class Circle {
    private double radius;
    public Circle() { }
    public Circle(double radius) {
        this.radius = radius;
    }
    public double getRadius() {
        return radius;
    }
}
```

为什么说这里Circle类是不可变类呢，因为radius字段被声明为private，这意味着外部代码无法直接访问或修改它，而类中没有提供setRadius方法来允许外部代码修改radius的值。一旦对象被创建，radius的值就无法被改变，所以radius的值只能在对象创建时通过构造函数设置，一旦设置就无法修改。
但仅仅将所有字段设为private，并且没有提供修改字段的方法（如set方法），并不足以使一个类成为不可变类，还有其他因素需要考虑。例如，如果类中包含可变对象的引用，那么即使字段是private，对象的状态仍然可能被外部代码修改。

1.1 不可变类的三个条件

因此一个不可变类必须满足以下三个条件：
1.所有字段必须是私有的（private）：这是为了隐藏内部状态，防止外部代码直接访问或修改字段。
2.不提供任何修改字段的方法（如set方法）：这样可以确保字段的值在对象创建后不会被改变。
3.不提供任何返回可变字段引用的访问器方法（如get方法）：如果类中包含可变对象的引用（如数组、集合或其他可变类的实例），那么返回这些引用可能会导致外部代码修改对象的状态。例如，如果类中有一个ArrayList字段，提供一个get方法返回这个ArrayList的引用，外部代码就可以通过这个引用修改列表的内容，从而破坏不可变性。

1.2 变量的作用域（Scope of Variables）

1.2.1 局部变量

对于局部变量来说，其作用域从其声明开始，一直持续到包含该变量的代码块（block）结束。代码块是指由花括号 {} 包围的一段代码，例如方法体、循环体或条件语句体。
当然局部变量在使用前必须显式初始化，否则编译器会报错，因为局部变量没有默认值。

1.2.2 类变量

类变量（包括实例变量和静态变量）可以在类的任何位置声明，通常放在类的顶部，与方法分开。
实例变量和静态变量的作用域是整个类。这意味着它们可以在类的任何方法中被访问。再次复习一遍，实例变量属于对象实例，而静态变量属于类本身。

1.3 this关键字

这里再次复习一遍this关键字。this关键字是一个引用，它指向当前对象本身。也就是说，它指向调用该方法或访问该字段的实例。
我们上次说了当局部变量（如方法参数或局部变量）与类的实例字段同名时，局部变量会“隐藏”实例字段。在这种情况下，可以使用this关键字来明确引用类的实例字段。
如下所示。

```
public class Foo {
    private int i = 5; // 实例变量，每个Foo对象都有独立的i值
    private static double k = 0; // 静态变量，所有Foo对象共享同一个k值

    void setI(int i) { // 实例方法，用于设置实例变量i的值
        this.i = i; // 使用this关键字来明确引用当前对象的实例变量i
    }

    static void setK(double k) { // 静态方法，用于设置静态变量k的值
        Foo.k = k; // 直接使用类名Foo来引用静态变量k
    }
}
```

这里i是实例变量，因此可以使用this关键字，而k是静态变量，属于类本身，所以对象都共享同一个静态变量，因此不能使用this关键字，而是使用类名。

其实this关键字还可以用于 调用同一个类的另一个构造函数，作为其第一条语句。这称为构造函数重载（Constructor Overloading）。
如下所示。

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

    // 无参构造函数
    public Person() {
        this("Unknown", 0); // 调用另一个构造函数
    }

    // 带两个参数的构造函数
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void printInfo() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
```

下面的代码同时体现了这两点。

```
public class Circle {
    private double radius;

    // 带参数的构造函数
    public Circle(double radius) {
        this.radius = radius; // 使用this关键字来引用当前对象的实例变量radius
    }

    // 无参构造函数
    public Circle() {
        this(1.0); // 使用this关键字调用带参数的构造函数
    }

    // 计算圆的面积的方法
    public double getArea() {
        return this.radius * this.radius * Math.PI; // 使用this关键字来明确引用当前对象的实例变量radius
    }
}
```

1.4 类抽象和封装（Class Abstraction and Encapsulation）

类抽象是指将类的实现细节与类的使用分离。这通常通过提供类的公共接口（API）来实现，而隐藏类的内部实现细节。
通过抽象，类的创建者可以定义类的行为和功能，即提供类的描述和公共方法，这些方法定义了类可以执行的操作，而不必让使用者了解类的具体实现方式。这使得使用者可以专注于如何使用类，通过这些方法与类交互，而不需要知道类内部的具体实现。
而不让用户知道类内部是如何实现的，这也就是封装（encapsulated），将数据字段设置为private，只开放一些公共接口（API）。
封装是指将对象的状态（属性）和行为（方法）捆绑在一起，并隐藏对象的内部实现细节。这通常是通过将类的字段设置为私有（private）并提供公共方法（如getter和setter）来访问和修改这些字段来实现的。
如下图所示。

类的实现对于客户端来说是像一个黑盒子（black box），客户端不需要了解类的内部实现细节，只需要通过类提供的接口与类交互。
类契约（Class Contract）包括公共方法（public methods）和公共常量（public constants）的签名（signatures）。这些是类对外提供的接口，定义了客户如何使用这个类。客户使用类通过类契约（contract of the class)。客户只需要了解如何通过公共接口与类交互，而不需要了解类的内部实现。

1.5 实践

我们有多种方式可以帮助我们理解我们应该创建什么样的类。
1.从现实生活角度，如果一系列物体有一些共性，那么这系列物体便可以被我们创建为一个类，这些共性就是其的属性，然后我们给这些属性添加上一些行为，最后还可以根据这些物体在现实中的情景以及我们的需要再添加一些额外的行为。
2.从数据结构的角度，如果我们需要一种特殊的数据结构，这种数据结构可以帮我们存储一系列东西，那我们也可以创建对应的类，并根据我们对这种数据结构的需要定义其中的属性和行为。

如果你还有所疑问你也可以看看下面的这些例子，从例子中找到感觉，下面的例子会先给出对应的UML图和说明，然后后面会附带上代码。

1.5.1 Loan类

```
public class Loan {
     private double annualInterestRate;
     private int numberOfYears;
     private double loanAmount;
     private java.util.Date loanDate;
     public Loan() {
        this(2.5, 1, 1000);
     }
     public Loan(double annualInterestRate, int numberOfYears, double
loanAmount) {
        this.annualInterestRate = annualInterestRate;
        this.numberOfYears = numberOfYears;
        this.loanAmount = loanAmount;
        loanDate = new java.util.Date();
     }
     public double getMonthlyPayment() {
        double monthlyInterestRate = annualInterestRate / 1200;
        double monthlyPayment = loanAmount * monthlyInterestRate / (1 -
(Math.pow(1 / (1 + monthlyInterestRate), numberOfYears * 12)));
```

```
        return monthlyPayment;
    }
    public double getTotalPayment() {
        double totalPayment = getMonthlyPayment() * numberOfYears * 12;
        return totalPayment;
    }
}
```

1.5.2 BMI类

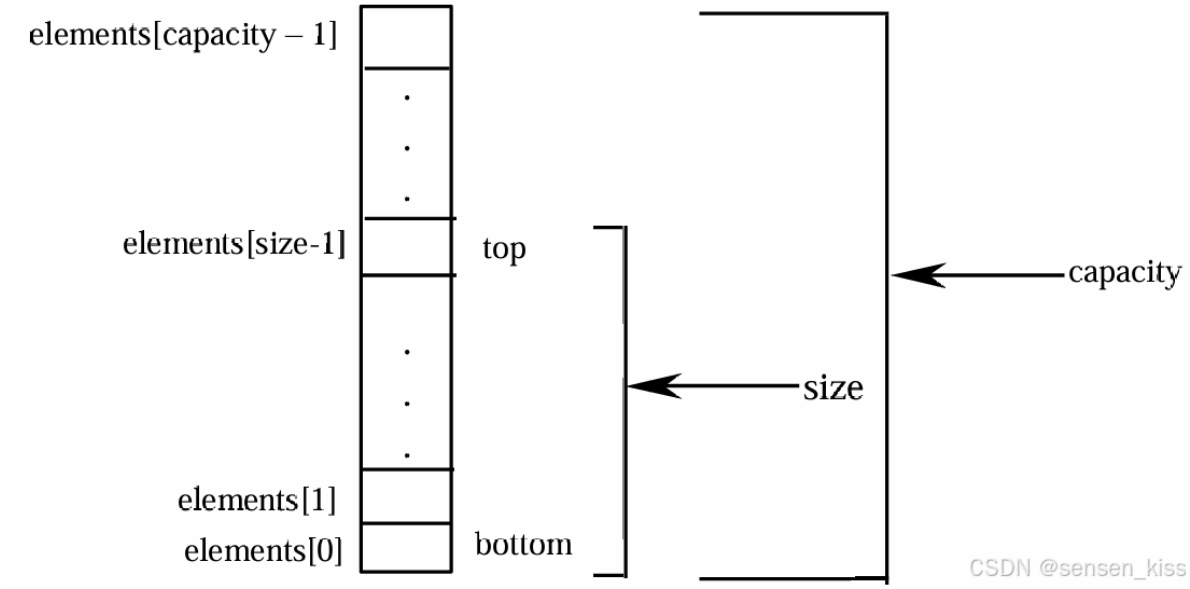
```
public class BMI {
    private String name;
    private int age;
    private double weight; // in pounds
    private double height; // in inches
    public static final double KILOGRAMS_PER_POUND = 0.45359237;
    public static final double METERS_PER_INCH = 0.0254;
    public BMI(String name, int age, double weight, double height) {
        this.name = name; this.age = age; this.weight = weight; this.height =
height;}
    public double getBMI() {
        double bmi = weight * KILOGRAMS_PER_POUND /  ((height * METERS_PER_INCH)
* (height * METERS_PER_INCH));
        return Math.round(bmi * 100) / 100.0;
    }
    public String getStatus() {
        double bmi = getBMI();
        if (bmi < 16) return "seriously underweight";
        else if (bmi < 18) return "underweight";
        else if (bmi < 24) return "normal weight";
        else if (bmi < 29) return "over weight";
        else if (bmi < 35) return "seriously over weight";
        else return "gravely over weight";
    }
    public String getName(){
        return name;
    }
    public int getAge(){
        return age;
    }
    public double getWeight(){
        return weight;
    }
    public double getHeight() {
        return height;
    }
}
```

1.5.3 课程类

```
public class Course {
    private String courseName;
    private String[] students = new String[100];
    private int numberOfStudents;
    public Course(String courseName) {
        this.courseName = courseName;
    }
    public void addStudent(String student) {
        if(numberOfStudents >= students.length){
            String[] temp = new String[students.length * 2];
            System.arraycopy(students, 0, temp, 0, students.length);
            students = temp;
        }
        students[numberOfStudents++] = student;
    }
    public String[] getStudents() {
        return students;
    }
    public int getNumberOfStudents() {
        return numberOfStudents;
    }
    public String getCourseName() {
        return courseName;
    }
}
```

1.5.4 整数栈

栈这里便不再过多介绍，在这学期的INT202种我们也详细复习了栈，其的特点就是FILO（先入后出），跟将子弹上到弹匣里一样。
我们这里需要注意容量（capacity）和当前存储数据的大小（size）不一样就好，如下图所示。



```
public class StackOfIntegers {
    private int[] elements;
    private int size;
    public static final int DEFAULT_CAPACITY = 16;
    public StackOfIntegers() {
        this(DEFAULT_CAPACITY);
    }
    public StackOfIntegers(int capacity) {
        elements = new int[capacity];
    }
    public void push(int value) {
        if (size >= elements.length) {
            int[] temp = new int[elements.length * 2];
            System.arraycopy(elements, 0, temp, 0, elements.length);
            elements = temp;
        }
        elements[size++] = value;
    }
    public int pop() {
        return elements[--size];
    }
    public int peek() {
        return elements[size - 1];
    }
    public int getSize() {
        return size;
    }
}
```

这里代码最精妙的地方在于pop方法，pop方法需要实现两个功能，返回最后一项的值，然后移除最后一项。我们并没有先返回最后一项，然后再将原来的地方抹除，而是通过size控制现在栈的大小，而这里是先将size减小1，然后刚好变成了原来数组里最后一个元素的index值。
还需要注意这里必须是--size而不是size--，可以参考下面的代码。

```
int size = 5;
int value = size--; // value 被赋值为 5，然后 size 变为 4
System.out.println("Value: " + value); // 输出 5
System.out.println("Size: " + size);  // 输出 4
```

```
int size = 5
int value = --size; // size 先变为 4，然后 value 被赋值为 4
System.out.println("Value: " + value); // 输出 4
System.out.println("Size: " + size);  // 输出 4
```

2. String类

前面我们说过String其实也是一个类，因此我们创建一个新的字符串对象可以使用new关键字，格式如下。

```
String newString = new String(stringLiteral);
```

例子如下。

```
String message = new String("Welcome to Java");
```

Java提供了一种更简便的方式来创建字符串，即直接使用字符串字面量。
即上面的例子可以简便结果如下。

```
String message = "Welcome to Java";
```

这种简化的初始化方式在内部使用了字符串池（String Pool）机制，即所谓的“字符串内部化”（String Interning）。

2.1 字符串内部化（String Interning）

字符串池是Java虚拟机（JVM）中的一个内存区域，用于存储字符串字面量。
当你使用简化的初始化方式创建字符串时，JVM会首先检查字符串池中是否已经存在相同的字符串。如果存在，JVM会返回池中已有的字符串对象的引用；如果不存在，JVM会将新的字符串对象添加到池中，并返回其引用。
因此其的优点如下：
1.字符串内部化可以节省内存空间，因为相同的字符串字面量只会在池中存储一次。
2.提高字符串比较的性能，因为可以直接比较字符串对象的引用，而不需要逐个字符进行比较。

与之不同的是使用 new 关键字创建字符串，这会在堆内存中创建一个新的字符串对象，即使它的值与字符串池中的对象相同。因此两种方法创建的对象的引用值不同。

2.2 String类是不可变类

String类也是前文说的不可变类。

```
String s = "Java";
s = "HTML";
System.out.println(s); // 输出的结果是HTML
```

这里虽然输出的结果是HTML，但其实是代码并没有修改原来的字符串对象 "Java"，而是让变量 s 引用了一个新的字符串对象，该对象表示字符串"HTML"。其实是和前面的渐变方式创建字符串一样，s = "HTML"时创建了一个新的字符串并将其引用赋值给了s，而原来的字符串对象"Java"仍然存在于字符串池中，但由于没有变量引用它，因此它现在是一个未被引用的对象。

2.3 相关方法

2.3.1 比较字符串

1.equals(Object obj): boolean 检查调用此方法的字符串是否与参数字符串内容相同。它是值比较，而不是引用比较。
如下所示。

```
String s1 = new String("Welcome");
String s2 = "Welcome";
if (s1.equals(s2)){ // true
// s1和s2有一样的值
}
if (s1 == s2) { // false
// s1和s2的引用值不同
}
```

这里使用两种方法创建创建了字符串，前者是在字符串池中创建，后者是在堆中创建，因此引用的对象不同，但引用的对象的内容是相同的。
下面的代码结果不同。

```
String s1 = "Welcome";
String s2 = "Welcome";
if (s1.equals(s2)){ // true
// s1 and s2 have the same contents
}
if (s1 == s2) { // true
// s1 and s2 have the same reference
}
```

因为字符串内部化，所以现在 s1 和 s2 指向同一个对象。

2.compareTo(String str): int 将调用此方法的字符串与参数字符串按字典顺序进行比较，并返回一个整数。如果调用字符串在参数字符串之前，返回负数；如果相等，返回0；如果调用字符串在参数字符串之后，返回正数。
字典顺序是比较字符串时常用的方法，它基于字符的Unicode值进行比较。示例如下。

```
String s1 = new String("Welcome");
String s2 = new String("Welcome");
if (s1.compareTo(s2) > 0) {
    // s1 is greater than s2
} else if (s1.compareTo(s2) == 0) {
    // s1 and s2 have the same contents
} else{
    // s1 is less than s2
}
```

这里代码会运行中间的部分。

更多的比较字符串的方法如下：
1.equalsIgnoreCase(String str): boolean 比较调用此方法的字符串对象是否与参数字符串对象str的内容相同，忽略大小写。返回true如果内容相同（不区分大小写），否则返回false。
2.compareToIgnoreCase(String str): int 与compareTo相同，但是比较时忽略大小写。
3.regionMatches(int toffset, String str, int offset, int len): boolean 检查调用此方法的字符串对象从toffset开始的特定区域是否与参数字符串对象str从offset开始的特定区域匹配。返回true如果区域匹配，否则返回false。
4.regionMatches(Boolean ignoreCase，int toffset, String str, int offset, Int len): boolean 与上一个方法相同，但是可以指定比较是否忽略大小写。返回true如果区域匹配（根据ignoreCase参数决定是否忽略大小写），否则返回false。
5.startsWith(String prefix): boolean 检查调用此方法的字符串对象是否以指定的前缀字符串prefix开

始。返回true如果字符串以指定前缀开始，否则返回false。
6.endsWith(String suffix): boolean 检查调用此方法的字符串对象是否以指定的后缀字符串suffix结束。返回true如果字符串以指定后缀结束，否则返回false。

2.3.2 获取字符串长度

length(): int 返回字符串的长度，即字符的数量。

2.3.3 检索字符串中的单个字符

charAt(int index): char 这个方法返回指定索引处的字符。索引是从 0 开始的，所以第一个字符的索引是 0，第二个字符的索引是 1，依此类推。
注意这里charAt返回的是 char 而不是 string 类型。
下图展示了上面两个方法的使用。

2.3.4 字符串连接

concat(String str): String将指定字符串连接到调用此方法的字符串的结尾。

2.3.5 子字符串

1.substring(int beginIndex): String 从beginIndex开始截取到字符串结尾的子字符串。
2.substring(int beginIndex, int endIndex): String 从beginIndex开始截取到endIndex之前的子字符串。也就是包含 beginIndex，不包含 endIndex。
例子如下。

```
String s1 = "Welcome to Java";
String s2 = s1.substring(0, 11) + "HTML";
```

如下图所示。

因此 s2 将变成 "Welcome to HTML"。

2.3.6 在字符串中查找字符或子字符串

1.indexOf(char ch): int返回字符 ch 在字符串中第一次出现的索引。如果未找到该字符，则返回 -1。
2.indexOf(char ch, int fromIndex): int 返回字符 ch 在字符串中从 fromIndex 之后第一次出现的索引。如果未找到该字符，则返回 -1。
3.indexOf(String str)： int 返回指定子字符串在此字符串中第一次出现的索引。如果未找到该字符，则返回 -1。
4.indexOf(String str,int formIndex): int 返回子字符串 str 在字符串中从 fromIndex 之后第一次出现的索引。如果未找到该子字符串，则返回 -1。
5.lastIndexOf(char ch): int 返回字符 ch 在字符串中最后一次出现的索引。如果未找到该字符，则返回 -1。
6.lastIndexOf(char ch, int fromIndex): int 返回字符 ch 在字符串中从 fromIndex 之前最后一次出现的索引。如果未找到该字符，则返回 -1。
7.lastIndexOf((String str)： int 返回子字符串 s 在字符串中最后一次出现的索引。如果未找到该子字符串，则返回 -1。
8.lastIndexOf(String str,int fromIndex): int 返回子字符串 s 在字符串中从 fromIndex 之前最后一次出现的索引。如果未找到该子字符串，则返回 -1。
下图展示了这些例子。

2.3.7 通过模式匹配、替换和分割

1.toLowerCase(): String 返回一个新的字符串，该字符串是调用此方法的字符串转换为小写形式的结果。
2.toUpperCase(): String 返回一个新的字符串，该字符串是调用此方法的字符串转换为大写形式的结果。
3.trim(): String 返回一个新的字符串，该字符串是调用此方法的字符串去除两端空白字符后的结果。
4.replace(char oldChar, char newChar): String 返回一个新的字符串，该字符串是调用此方法的字符串中所有匹配的字符 oldChar 被替换为 newChar 后的结果。
5.replaceFirst(String oldString, String newString): String 返回一个新的字符串，该字符串是调用此方法的字符串中第一个匹配的子字符串 oldString 被替换为 newString 后的结果。
6.replaceAll(String oldString, String newString): String 返回一个新的字符串，该字符串是调用此方法的字符串中所有匹配的子字符串 oldString 被替换为 newString 后的结果。
7.split(Stirng delimiter): String[] 返回一个字符串数组，该数组包含调用此方法的字符串按照指定分隔符 delimiter 分割后的子字符串。
例子如下：
1."Hello".toLowerCase() 返回 "hello"。
2."hello".toUpperCase() 返回 "HELLO"。
3." Hello World ".trim() 返回 "Hello World"。
4."Hello World".replace('l', 'r') 返回 "Hemmo World"。
5."Hello World".replaceFirst("World", "Java") 返回 "Hello Java"。
6."Hello World".replaceAll("l", "AB") 返回 "HeABABo World"。
7."one,two,three".split(",") 返回 ["one", "two", "three"]。

2.3.7.1 使用正则表达式作为参数

在Java中，String 类的 replaceAll、replaceFirst、split 和 matches 方法可以接受正则表达式作为参数来指定搜索模式。
示例如下。

```
String s = "a+b$#c".replaceAll("[$+#]", "NNN");
System.out.println(s);
```

[Error: You can't use 'macro parameter character #' in math mode、+ 和 # 三个字符。每当在字符串中找到这些字符中的任意一个时，它都会被替换为 "NNN"。
因此最后的结果是"aNNNbNNNNNc"。

相关的方法如下。
1.matches(String regex): Boolean 返回一个布尔值，指示调用它的字符串是否匹配给定的正则表达式模式。
2.replaceAll(String regex, String replacement): String 返回一个新字符串，该字符串是调用它的字符串中所有匹配的子字符串被替换为指定的替换字符串后的结果。
3.replaceFirst(String regex, String replacement): String 返回一个新字符串，该字符串是调用它的字符串中第一个匹配的子字符串被替换为指定的替换字符串后的结果。
4.split(String regex): String[] 返回一个字符串数组，该数组包含调用它的字符串按照给定的正则表达式模式分割后的子字符串数组。

下图是正则表达式的参考表。

x：匹配指定的字符 x。例如，如果 x 是字母 a，那么它将匹配字符串中的 a。
.：匹配除换行符以外的任何单个字符。

(ab|cd)：匹配 ab 或 cd。
[abc]：匹配 a、b 或 c 中的任意一个字符。
[^abc]：匹配除了 a、b 或 c 之外的任何字符。
[a-z]：匹配从 a 到 z 之间的任意单个字符。
[^a-z]：匹配不在 a 到 z 范围内的任意单个字符。
[a-e[m-p]]：匹配从 a 到 e 或从 m 到 p 之间的任意单个字符。
[a-e&&[c-p]]：匹配 a-e 和 c-p 两个范围的交集中的任意单个字符。
\d：匹配一个数字，等同于 [0-9]。
\D：匹配一个非数字字符。
\w：匹配一个单词字符（字母、数字或下划线）。
\W：匹配一个非单词字符。
\s：匹配任何空白字符（包括空格、制表符、换页符等）。
\S：匹配任何非空白字符。
p*：匹配模式 p 出现零次或多次。
p+：匹配模式 p 出现一次或多次。
p?：匹配模式 p 出现零次或一次。
p{n}：匹配模式 p 恰好出现 n 次。
p{n,}：匹配模式 p 至少出现 n 次。
p{n,m}：匹配模式 p 出现 n 到 m 次（包含 n 和 m）。

示例如下：
1."Java is fun".matches("Java.*")
正则表达式 "Java.*" 表示以 "Java" 开头，后面跟着任意数量（包括零个）的任意字符。因此结果是 true，因为字符串 "Java is fun" 符合这个模式。
2.

```
String[] tokens = "Java,C;C#.C+".split("[,;.]");
for (int i = 0; i < tokens.length; i++)
    System.out.println(tokens[i]);
```

正则表达式[,;.]定义了一个字符集，包含逗号,、分号;和句点.。这些字符被视为分隔符。split方法按照这些分隔符将原始字符串分割成多个子字符串。
因此分割后的子字符串数组tokens打印出来的结果如下。
Java
C
C#
C++

3.

```
String s = "Java Java Java".replaceAll("v\\w", "wi");
// "Jawi Jawi Jawi"
String s2 = "Java Java Java".replaceFirst("v\\w", "wi");
// "Jawi Java Java"
String[] s3 = "Java1HTML2Perl".split("\\d");
// ["Java", "HTML", "Perl"]
```

正则表达式 "v\\w" 会匹配 v 后面紧跟的单词字符，但s1是全部替换，s2是只替换第一个。
正则表达式 \\d 表示匹配任何数字字符。因此该字符串被数字1和2分割就会得到["Java", "HTML", "Perl"]。

4.社保号码
正则表达式 [\d]{3}-[\d]{2}-[\d]{4} 用于匹配社会安全号码格式。其中 \d 表示一个数字，{3} 表示恰好重复3次，- 是字面意义上的短横线字符。
这个模式匹配 "xxx-xx-xxxx" 格式的字符串，其中 x 是一个数字。

```
String ssn = "123-45-6789";
boolean matches = ssn.matches("\\d{3}-\\d{2}-\\d{4}");
System.out.println(matches); // 输出 true
```

5.匹配偶数
正则表达式 [\d*[02468] 用于匹配以0、2、4、6或8结尾的数字。
\d 表示一个数字，* 表示前面的元素可以出现任意次数（包括零次），[02468] 表示匹配这些数字中的任意一个。这个模式匹配任何以0、2、4、6或8结尾的数字。

```
String evenNumber = "12345678";
boolean matches = evenNumber.matches("\\d*[02468]\\d*");
System.out.println(matches); // 输出 true
```

6.匹配电话号码
正则表达式 \\([1-9][\\\d]{2}\\) [\\d]{3}-[\\d]{4} 用于匹配电话号码格式。
\\(和 \\) 分别匹配左右括号，因为括号在正则表达式中具有特殊含义，所以需要使用反斜杠进行转义。
[1-9] 匹配1到9之间的任何一个数字（确保电话号码的第一位不是0）。
\\d{2} 匹配恰好两个数字，[\\d]{3} 匹配恰好三个数字，- 是字面意义上的短横线字符。
这个模式匹配 "(xxx) xxx-xxxx" 格式的字符串，其中 x 是一个数字。

```
String phoneNumber = "(123) 456-7890";
boolean matches = phoneNumber.matches("\\([1-9]\\d{2}\\) \\d{3}-\\d{4}");
System.out.println(matches); // 输出 true
```

2.3.8 将字符和数字值转换为字符串：

1.valueOf() 系列方法：将不同类型的数据（如基本数据类型、对象等）转换为字符串。
valueOf() 方法主要用于将基本数据类型（如 int、double、boolean 等）转换为对应的包装类对象（如 Integer、Double、Boolean 等）。这些包装类对象通常有一个 toString() 方法，可以进一步将对象转换为字符串。

```
int number = 123;
Integer integerObject = Integer.valueOf(number);
String numberString = integerObject.toString();
System.out.println(numberString); // 输出 "123"
```

2.toString() 方法：许多类都重写了此方法，以提供其实例的字符串表示。
toString() 方法是 Object 类中的一个方法，所有Java类都继承自 Object 类，因此所有类都有 toString() 方法。默认情况下，toString() 方法返回一个包含类名和对象哈希码的字符串。然而，许多类都重写了此方法，以提供其实例的字符串表示。

```
String str = "Hello";
System.out.println(str.toString()); // 输出 "Hello"

Integer num = 123;
System.out.println(num.toString()); // 输出 "123"
```

2.3.9 命令行参数

在Java中，可以通过 public static void main(String[] args) 方法接收命令行参数。args 数组包含传递给Java应用程序的参数，每个参数都是一个字符串。
在Java中，public static void main(String[] args) 方法是每个Java应用程序的入口点，即程序开始执行的地方。当你运行一个Java应用程序时，你可以从命令行向该程序传递参数，这些参数存储在args数组中。
示例如下。

```
public class MyProgram {
    public static void main(String[] args) {
        if (args.length > 0) {
            for (int i = 0; i < args.length; i++) {
                System.out.println("Argument " + (i + 1) + ": " + args[i]);
            }
        } else {
            System.out.println("No arguments were provided.");
        }
    }
}
```

在命令行中输入。

```
javac MyProgram.java
java MyProgram Hello World
```

这里第一行使用 javac 命令（Java编译器）将 .java 源文件编译成 .class 字节码文件，因为Java虚拟机（JVM）只能执行编译后的字节码，而不是人类可读的源代码。
如果没有执行第一行直接运行.java文件，JVM不知道如何执行.java文件，从而会受到一个错误信息。
正确执行后，会得到结果如下。

```
Argument 1: Hello
Argument 2: World
```

再给出一个例子。

```
public class Calculator {
    public static void main(String[] args) {
        // 检查参数数量是否正确
        if (args.length != 3) {
            System.out.println("Usage: java Calculator operand1 operator
operand2");
            System.exit(0);
        }

        int result = 0;
```

```
        try {
            // 将操作数转换为整数
            int operand1 = Integer.parseInt(args[0]);
            int operand2 = Integer.parseInt(args[2]);

            // 根据操作符进行计算
            switch (args[1].charAt(0)) {
                case '+':
                    result = operand1 + operand2;
                    break;
                case '-':
                    result = operand1 - operand2;
                    break;
                case '*':
                    result = operand1 * operand2;
                    break;
                case '/':
                    // 检查除数是否为零
                    if (operand2 == 0) {
                        throw new ArithmeticException("Cannot divide by zero");
                    }
                    result = operand1 / operand2;
                    break;
                default:
                    System.out.println("Invalid operator: " + args[1]);
                    System.exit(1);
            }
        } catch (NumberFormatException e) {
            System.out.println("Invalid number: " + e.getMessage());
            System.exit(1);
        } catch (ArithmeticException e) {
            System.out.println("Error: " + e.getMessage());
            System.exit(1);
        }

        // 打印结果
        System.out.println("Result: " + result);
    }
}
```

之后在命令行中输入。

```
javac Calculator.java
java Calculator 1 + 2
```

得出结果。

```
3
```

2.4 StringBuilder 和 StringBuffer 类

StringBuilder 和 StringBuffer 类，它们是 String 类的替代品，提供了更灵活的字符串操作能力。
StringBuilder 和 StringBuffer 类提供了与 String 类似的方法，用于创建和操作字符串，这些类可以用于任何需要字符串的场景。
StringBuffer 是同步的（synchronized），意味着它是线程安全的。同步机制确保了同一时刻只有一个线程可以执行 StringBuffer 的方法，从而避免了多线程同时修改同一个 StringBuffer 对象时可能出现的竞争条件。
StringBuilder 是非同步的（non-synchronized），这意味着它不是线程安全的。在多线程环境中，如果多个线程同时调用 StringBuilder 的方法，可能会导致数据不一致或其他问题。
StringBuilder 和 StringBuffer 提供了一系列方法，如 append()、insert() 和 delete() 等，可以在不创建新字符串对象的情况下修改字符串内容。
与 String 类不同，String 是不可变的（immutable），一旦创建，其内容不能被修改。任何对 String 的修改都会生成一个新的 String 对象。

2.4.1 构造函数

java.lang.StringBuilder	
+StringBuilder()	Constructs an empty string builder with capacity 16.
+StringBuilder(capacity: int)	Constructs a string builder with the specified capacity.
+StringBuilder(s: String)	Constructs a string builder with the specified string.

1.StringBuilder() 这个构造函数创建了一个空的StringBuilder对象，其初始容量为16个字符。这意味着StringBuilder对象一开始有足够的空间来存储16个字符，如果添加的字符超过这个容量，它会自动扩展。
2.StringBuilder(int capacity) 这个构造函数允许你指定创建的StringBuilder对象的初始容量。你可以传递一个整数值capacity来设置初始容量，这将决定StringBuilder对象一开始可以存储多少个字符。
3.StringBuilder(String s) 这个构造函数创建了一个StringBuilder对象，并用指定的字符串s来初始化它。传递给构造函数的字符串s将被用作StringBuilder对象的初始内容，初始容量与传入的字符串 s 的长度是一致的。

2.4.2 相关方法

java.lang.StringBuilder	
+append(data: char[]): StringBuilder	Appends a char array into this string builder.
+append(data: char[], offset: int, len: int): StringBuilder	Appends a subarray in data into this string builder.
+append(v: *aPrimitiveType*): StringBuilder	Appends a primitive type value as a string to this builder.
+append(s: String): StringBuilder	Appends a string to this string builder.
+delete(startIndex: int, endIndex: int): StringBuilder	Deletes characters from startIndex to endIndex.
+deleteCharAt(index: int): StringBuilder	Deletes a character at the specified index.
+insert(index: int, data: char[], offset: int, len: int): StringBuilder	Inserts a subarray of the data in the array to the builder at the specified index.
+insert(offset: int, data: char[]): StringBuilder	Inserts data into this builder at the position offset.
+insert(offset: int, b: *aPrimitiveType*): StringBuilder	Inserts a value converted to a string into this builder.
+insert(offset: int, s: String): StringBuilder	Inserts a string into this builder at the position offset.
+replace(startIndex: int, endIndex: int, s: String): StringBuilder	Replaces the characters in this builder from startIndex to endIndex with the specified string.
+reverse(): StringBuilder	Reverses the characters in the builder.
+setCharAt(index: int, ch: char): void	Sets a new character at the specified index in this builder.

1.append(char[] data): StringBuilder 将一个字符数组追加到这个字符串构建器中。
2.append(char[] data, int offset , int len): StringBuilder 将字符数组data中从offset开始的len个字符追加到这个字符串构建器中。
3.append(PrimitiveType v): StringBuilder 将一个基本类型值（如int、double等）追加为字符串形式到这个字符串构建器中。
4.append(String s):StringBuilder 将一个字符串追加到这个字符串构建器中。
5.delete(int startIndex, int endIndex): StringBuilder 删除从startIndex到endIndex（不包含endIndex）的字符。
6.deleteCharAt(int index): StringBuilder 删除指定索引处的字符。
7.insert(int index, char[] data, int offset, int len): StringBuilder 在指定索引处插入字符数组data中从offset开始的len个字符。
8.insert(int offset, char[] data): StringBuilder 在指定位置offset插入字符数组。
9.insert(int offset, PrimitiveType v): StringBuilder 在指定位置插入一个基本类型值（如int、double等）转换成的字符串。
10.insert(int offset, String s): StringBuilder 在指定位置插入一个字符串。
11.replace(int startIndex, int endIndex, String s): StringBuilder 用指定的字符串s替换从startIndex到endIndex（不包含endIndex）的字符。
12.reverse(): StringBuilder 反转字符串构建器中的字符顺序。
13.setCharAt(int index,char ch): void 设置指定索引处的新字符。

示例如下。

```
StringBuilder stringBuilder = new StringBuilder();
// 创建了一个新的空StringBuilder对象
stringBuilder.append("Java");
// 追加字符串"Java"
stringBuilder.insert(2,"HTML and ");
// 在索引2的位置插入"HTML and "
stringBuilder.delete(3, 4);
// 删除从索引3到4的字符（不包含4）
stringBuilder.deleteCharAt(5);
// 删除索引值为5的字符
stringBuilder.reverse();
// 反转StringBuilder对象中的字符串
stringBuilder.replace(4, 8, "HTML");
// 替换StringBuilder对象中从索引4到索引8的字符串为"HTML"（不包含8）
stringBuilder.setCharAt(0, 'w');
// 设置0处的字符为"w"
```

还有一些其他方法。
1.toString(): String 返回一个字符串对象，该对象是此字符串构建器当前内容的表示。
2.capacity(): int 返回此字符串构建器的容量，即它能够存储的最大字符数。
3.charAt(index: int): char 返回指定索引处的字符。
4.length(): int 返回此构建器中的字符数。
5.setLength(int newLength): void 设置此构建器的新长度。如果新长度小于当前长度，此方法截断开构建器；如果新长度大于当前长度，此方法用 null 字符填充构建器。
6.substring(int startInde): String 返回从 startIndex 开始的子字符串。
7.substring(int startIndex, int endIndex): String 返回从 startIndex 到 endIndex-1 的子字符串。
8.trimToSize(): void 减少字符串构建器使用的存储大小，使其与当前长度相匹配。

2.5 Character类

Character类和String类比较像，但是String类代表一个不可变的字符序列，Character类代表一个单独的字符，其是可变的。

2.5.1 相关方法

1.Character(char value) 使用指定的字符值构造一个字符对象。
2.charValue(): char 返回此对象中的字符值。
3.compareTo(Character anotherCharacter): int 将此字符与另一个字符对象进行比较。如果此字符在字母顺序中位于参数字符之前，则返回一个负整数；如果位于之后，则返回一个正整数；如果两个字符相等，则返回0。
4.equals(Character anotherCharacter): boolean 如果此字符与另一个字符相等，则返回true。
5.isDigit(char ch): boolean 如果指定的字符是数字，则返回true。
6.isLetter(char ch): boolean 如果指定的字符是字母，则返回true。
7.isLetterOrDigit(char ch): boolean 如果指定的字符是字母或数字，则返回true。
8.isLowerCase(char ch): boolean 如果字符是小写字母，则返回true。
9.isUpperCase(char ch): boolean 如果字符是大写字母，则返回true。
10.toLowerCase(char ch): char 返回指定字符的小写形式。
11.toUpperCase(char ch): char 返回指定字符的大写形式。

示例如下。

```
public class CharacterExample {
    public static void main(String[] args) {
        Character charObject = new Character('b');

        // 使用 equals 方法比较 Character 对象
        boolean isEqualB = charObject.equals(new Character('b')); // 返回 true
        boolean isEqualD = charObject.equals(new Character('d')); // 返回 false

        // 使用 compareTo 方法比较 Character 对象
        int compareToA = charObject.compareTo(new Character('a')); // 返回 1
        int compareToB = charObject.compareTo(new Character('b')); // 返回 0
        int compareToC = charObject.compareTo(new Character('c')); // 返回 -1
        int compareToD = charObject.compareTo(new Character('d')); // 返回 -2

        // 打印结果
        System.out.println("charObject.equals(new Character('b')): " +
isEqualB);
        System.out.println("charObject.equals(new Character('d')): " +
isEqualD);
        System.out.println("charObject.compareTo(new Character('a')): " +
compareToA);
        System.out.println("charObject.compareTo(new Character('b')): " +
compareToB);
        System.out.println("charObject.compareTo(new Character('c')): " +
compareToC);
        System.out.println("charObject.compareTo(new Character('d')): " +
compareToD);
    }
}
```

3. 类的设计

我们在上学期的CPT203中详细学习了面向对象编程的三个核心概念：内聚性（Coherence）、职责分离（Separating responsibilities）和重用性（Reuse）。这些原则有助于设计清晰、灵活且可维护的类。
CPT203相关知识
1.内聚性（Coherence）：
内聚性指的是一个类应该描述一个单一的实体（entity），并且该实体的所有属性和行为都应该与该实体的概念紧密相关。一个类应该“做一件事并做好”，这意味着类中的属性和方法都与类的主要目的相关联。内聚性好的类通常更容易理解和维护，因为它们具有明确和一致的职责。
2.职责分离（Separating responsibilities）：
职责分离是指将具有多个职责的单一实体分解为多个类，每个类负责一部分职责。
当一个类承担了过多的职责时，这可能导致类变得复杂和难以管理。通过将不同的职责分配给不同的类，可以简化每个类的设计，提高代码的可读性和可维护性。
3.重用性（Reuse）：
重用性是指设计类时考虑到类的可重用性，即在不同的上下文中可以多次使用同一个类。通过创建可重用的类，可以避免代码重复，减少开发时间和维护成本。重用性也有助于提高代码的一致性，因为相同的功能在不同的地方使用相同的实现。

除了这三个核心思想以外，还有一些标准编程风格和命名的约定，些约定有助于编写清晰、一致且易于理解的代码。
1.遵循标准的Java编程风格和命名约定：
编写代码时应该遵循Java社区广泛接受的编程风格和命名规则，例如变量、方法和类名应该清晰、简洁且具有描述性。
即为类、数据字段和方法选择有意义的名称。
类名、字段名和方法名应该能够清楚地表达其作用和用途，以便于其他开发者理解代码的功能和行为。
2.将数据声明放在构造函数之前，构造函数放在方法之前：
类的字段声明应该在构造函数之前，这样可以让字段的初始化在对象创建时立即进行。
构造函数应该在其他方法之前，因为构造函数用于初始化对象，而其他方法通常依赖于对象的初始化状态。
3.尽可能提供一个公共的无参构造函数，并重写equals方法和toString方法（返回一个字符串）：
提供一个无参的公共构造函数可以让类的实例更容易被创建，即使没有特定的初始化参数。
重写equals方法可以确保类的实例可以根据其内容进行比较，而不仅仅是引用比较。
重写toString方法可以提供类的实例的字符串表示，这对于调试和日志记录非常有用。

3.1 继承（Inheritance）

前面我们说的是类设计中基础的一些概念，以及我们在上一章中介绍了我们该怎么想着去创造一个类。
现在我们回到最初需要类的动机上，在现实世界中，许多实体（如圆形、矩形和三角形）具有共同的特征和行为（例如，颜色、是否填充、创建日期等），而这些共同的特征和行为可以在代码中表示为共享的数据字段和方法。
下面我们介绍类的一个进阶知识——继承（Inheritance）。
继承是一种机制，允许一个子类（sub-class）扩展另一个父类（super-class）。
子类继承父类的属性和方法，从而可以重用已有的代码，而不需要重新编写。继承可以帮助我们设计和实现类，使得代码结构更加清晰，易于理解和扩展。父类可以提供一个通用的框架，而子类可以专注于特定的实现细节。

我们一般用关键字 abstract 声明抽象类或抽象方法，当一个类被声明为abstract时，它不能被实例化，即你不能创建这个类的对象。
抽象类通常用作其他类的基类，提供通用的属性和方法，这些属性和方法可以在子类中被具体实现。
抽象类可以包含抽象方法和具体方法。
如果一个类继承自抽象类，它必须实现抽象类中的所有抽象方法，除非该子类也被声明为抽象。
抽象方法是不包含方法体（即方法的实现部分）的方法，而抽象方法必须在抽象类中声明。
抽象方法用于定义方法的规范或接口，具体的实现由继承该抽象类的子类提供。

因此我们根据前面的例子，我们可以创建一个图形的父类，它定义了所有图形共有的属性和方法。然后我们再创建对应的具体的图形类，这些类继承父类。
UML图如下。

代码如下。

```
public abstract class GeometricObject {
    private String color = "white";
    private boolean filled;
    private java.util.Date dateCreated;

    protected GeometricObject() {
        dateCreated = new java.util.Date();
    }

    protected GeometricObject(String color, boolean filled) {
        this();
        this.color = color;
        this.filled = filled;
    }

    public String getColor() { return color; }
    public void setColor(String color) { this.color = color; }
    public boolean isFilled() { return filled; }
    public void setFilled(boolean filled) { this.filled = filled; }
    public java.util.Date getDateCreated() { return dateCreated; }

    public String toString() {
        return "color: " + color + ", filled: " + filled +
                ", created on " + dateCreated;
    }

    /** Abstract method getArea */
    public abstract double getArea();

    /** Abstract method getPerimeter */
    public abstract double getPerimeter();
}
```

创建完GeometricObject类后，再创建Circle类和Rectangle类继承GeometricObject类。

```
public class Circle extends GeometricObject {
    private double radius;

    public Circle() { }
    public Circle(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }
```

```
    public void setRadius(double radius) {
        this.radius = radius;
    }

    public String toString() {
        return "Circle with radius is " + radius + ", " + super.toString();
    }

    public double getArea() {
        return radius * radius * Math.PI;
    }

    public double getPerimeter() {
        return 2 * radius * Math.PI;
    }

    public double getDiameter() {
        return 2 * radius;
    }
}
```

```
public class Rectangle extends GeometricObject {
    private double width;
    private double height;
    public Rectangle() {
       // super();
    }
    public Rectangle(double width, double height) {
       this();
       this.width = width;
       this.height = height;
    }
    public Rectangle(double width, double height, String color, boolean filled)
{
       super(color, filled);
       this.width = width;
       this.height = height;
    }
    public double getWidth() { return width; }
    public void setWidth(double width) { this.width = width; }
    public double getHeight() { return height; }
    public void setHeight(double height) { this.height = height; }
    public double getArea() {
       return width * height;
    }
    public double getPerimeter() {
       return 2 * (width + height);
   }
}
```

我们可以再测试一下。

```
public static void main(String[] args) {
        // Declare and initialize two geometric objects
```

```
        GeometricObject geoObject1 = new Circle(5);
        GeometricObject geoObject2 = new Rectangle(5, 3);
        // Display circle
        displayGeometricObject(geoObject1);
        // Display rectangle
        displayGeometricObject(geoObject2);
        System.out.println("The two objects have the same area? " +
equalArea(geoObject1, geoObject2));
    }

    /** A method for displaying a geometric object */
    public static void displayGeometricObject(GeometricObject object) {
        System.out.println(object); // object.toString()
        System.out.println("The area is " + object.getArea());
        System.out.println("The perimeter is " + object.getPerimeter());
    }

    /** A method for comparing the areas of two geometric objects */
    public static boolean equalArea(GeometricObject object1, GeometricObject
object2) {
        return object1.getArea() == object2.getArea();
    }
```

这里测试的时候我们geoObject1和geoObject2两个对象声明的时候用的类型是GeometricObject，这里使用的是父类的引用类型来引用子类的实例，这便是多态性（polymorphism），后面我们会再详细介绍多态性。

3.1.1 声明子类

前面的代码中我们先声明了一个父类，再声明了两个子类。我们声明子类的时候，它自动继承其父类的所有属性（字段）和方法。这意味着子类可以使用父类中定义的所有公共（public）和受保护（protected）属性和方法，而无需重新编写这些代码。
而子类可以添加自己的属性，这些属性在父类中不存在，这些新属性可以满足子类特有的需求，而不影响父类。（例如Circle类里面的radius，Rectangle类里的width）。
子类还可以定义自己的方法，这些方法在父类中不存在，这些新方法可以提供新的功能，或者对现有功能进行扩展。（例如Circle类里面的getRadius()，Rectangle类里面的setWidth())
子类可以重写（Override）父类的方法，以提供特定于子类的行为。
重写方法时，子类的方法签名必须与父类中的方法签名相同。通过重写，子类可以改变父类方法的行为，使其更适合子类的需求。（例如Circle类和Rectangle类里的getArea()和getPerimeter())

我们前面的测试代码中，我们声明了两个GeometricObject类的对象geoObject1和geoObject2，我们可以调用getArea()方法，这里其实使用的是子类Circle和Rectangle中的方法，这同样因为多态性，但我们不能直接调用子类的方法，例如我们无法使用geoObject2调用getWidth()，因为在编译时，geoObject2的静态类型是GeometricObject，而不是Rectangle。为了调用getWidth()方法，需要将geoObject2向下转型为Rectangle类型才可以调用。

3.1.2 构造函数

超类的构造函数不会被继承。子类需要显式或隐式地调用超类的构造函数。
1.显式调用：
使用super关键字和超类构造函数的参数来显式调用超类的构造函数。
2.隐式调用：
如果没有显式使用super关键字，那么超类的无参构造函数（即不接受任何参数的构造函数）会在子类

构造函数的第一行自动被调用，除非使用this关键字调用了另一个构造函数。在这种情况下，链条中最后一个构造函数将调用超类的构造函数。（这里的链条指的是一个构造函数使用this关键字调用了另一个构造函数，这种调用形成了一个构造函数链，其中每个构造函数可以调用同一个类中的其他构造函数。）

示例如下。

```
public A(args) {
    super(); // 显式调用超类的无参构造函数
    // some statements
}
```

```
public A(args) {
    // 隐式调用超类的无参构造函数
    // some statements
}
```

这两种写法等效。

3.2 super关键字

super关键字我们前面就使用过，它具体的功能是什么呢?
super关键字用于引用当前对象的父类。在Java中，每个类都有一个直接的父类（除了 Object 类，它是所有类的根类）。

它具体的作用主要是以下两点。
1.构造函数链式调用：
super关键字用于在子类的构造函数中调用父类的构造函数，这称为构造函数链式调用（Constructor Chaining）。这确保了父类的初始化逻辑在子类对象创建时被执行。
2.调用超类方法：
当子类重写了父类的方法时，可以使用super关键来调用被重写（Override）的父类方法。这允许访问在子类中被隐藏的父类方法。

3.2.1 构造函数链式调用（Constructor Chaining）

前面说了一种链条关系，但这里更多是强调构造一个类的实例需要沿着继承链调用超类的构造函数。它们其实都想表达的是当多个构造函数在一起形成了一个构造函数链时，这个链可以确保对象在完全构造之前，所有的初始化逻辑都按顺序执行。
例子如下。

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }
    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
```

```
    }
    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```

这里先是创建了一个Person父类，然后Employee子类继承Person类下面有一个无参数构造类，它会调用另一个构造函数，最后是Faculty子类它继承Employee类。
因此我们使用new Faculty()会调用Faculty类的无参构造参数，然后因为继承关系，它隐式调用super()，因此调用了Employee类的无参构造函数，然后这个函数又显式调用了另一个带有String参数的构造函数，然后又是因为继承关系，所以它又隐式调用super()，因此调用了Person类的无参构造函数。这个链就结束了，因此链的最后一个最先执行，也就是先执行Person类的无参构造函数。
最后结果如下。

```
(1) Person's no-arg constructor is invoked
(2) Invoke Employee’s overloaded constructor
(3) Employee's no-arg constructor is invoked
(4) Faculty's no-arg constructor is invoked
```

3.2.2 调用超类方法

前面的Circle类就是一个例子。

```
public String toString() {
        return "Circle with radius " + radius + ", " + super.toString();
    }
```

这里是重写父类里的toString()方法，因此现在使用super关键字调用被隐藏的父类方法。

我们这里顺便再详细说一下方法重写（Method overriding）的概念。方法重写是指在子类中修改父类中已定义的方法的实现。这是多态性的一种表现，允许子类根据需要改变从父类继承来的方法的行为。
在子类中，方法重写通过使用与父类中被重写的方法相同的签名（方法名和参数列表）来实现。
可以使用 @Override 注解来显式声明一个方法是重写父类中的方法。这个注解是可选的，但使用它可以提高代码的可读性和减少出错的机会。
例如Circle类的这个toString()方法就是重写了父类GeometricObject里的toString()方法。

3.3 Obejct类

前面提到在Java中，java.lang.Object 是所有类的根类（超类）。
因此这意味着每个Java类都直接或间接地继承自Object类。当你定义一个类时，如果没有显式指定它继承自哪个类，那么它默认继承自java.lang.Object类。
Object类提供了一些通用的方法，这些方法对所有对象都是可用的，例如 equals(), hashCode(), toString() 等。
下面的代码是等价的。

```
public class GeometricObject {
    // some statements
}
```

```
public class GeometricObject extends Object {
    // some statements
}
```

3.3.1 toString()方法

toString()方法用于返回对象的字符串表示形式。
Object 类的默认实现返回一个字符串，该字符串由对象的类名、@符号和一个数字组成，这个数字是对象的哈希码（hash code）的无符号十六进制表示。
示例如下。

```
class Loan {
    // Loan 类的具体实现
}

public class Main {
    public static void main(String[] args) {
        Loan loan = new Loan();
        System.out.println(loan.toString());
    }
}
```

输出的结果可能类似于Loan@10f87f48其中Loan是对象的类名，@10f87f48是对象的哈希码。这种默认格式不是非常用户友好或信息丰富，因为它主要包含技术细节（类名和哈希码）。因此我们应该重写一个toString()方法从而返回一个更有信息量、更友好的字符串表示形式。

3.3.2 方法重载（Overloading）和方法重写（Overriding）的对比

我们前面说了方法重写（Overriding），比如重写toString()方法。方法重写是指在子类中覆盖父类中具有相同签名（即方法名和参数列表相同）的方法。它需要方法签名相同，包括返回类型和参数列表。

```
public abstract class GeometricObject {
    public abstract double getArea();
}

class Circle extends GeometricObject {
    @Override
    public double getArea() {
        return Math.PI * radius * radius;
    }
}
```

方法重载（Overloading）是指在同一个类中可以创建多个同名的方法，但这些方法必须具有不同的参数列表（即参数的数量或类型不同）。
这是编译时多态性，即在同一个类中，根据传入参数的不同，可以调用同名但参数不同的方法。
方法重载不要求方法具有相同的返回类型，参数列表的不同足以区分重载的方法。
示例如下。

```java
public class Overloading {
    public static int max(int num1, int num2) {
        if (num1 > num2) return num1;
        return num2;
    }

    public static double max(double num1, double num2) {
        if (num1 > num2) return num1;
        return num2;
    }

    public static void main(String[] args) {
        System.out.println(max(1, 2)); // 调用第一个max方法，参数类型为int
        System.out.println(max(1, 2.3)); // 调用第二个max方法，参数类型为double
    }
}
```

3.3.2.1 方法匹配（Method Matching）和方法绑定（Method Binding）

这两个的实现靠的是Java的两个不同机制，方法匹配（Method Matching）和方法绑定（Method Binding）。
1.方法匹配（Method Matching）：当一个类中存在多个同名但参数列表不同的方法时，即当遇到重载方法时，编译器在编译时根据方法的参数类型、参数数量和参数顺序来确定调用哪个方法。
例如，如果一个类中有两个名为doSomething的方法，一个接受int参数，另一个接受double参数，那么编译器会根据调用时提供的参数类型来决定使用哪个方法。
2.方法绑定（Method Binding）：当子类重写了父类中的方法时，即当遇到重写方法时，Java虚拟机（JVM）在运行时动态绑定到最具体的重写方法实现。
这意味着即使引用变量的类型是父类类型，实际执行的也是子类中重写的方法。
例如，如果Child类重写了Parent类的doSomething方法，那么即使有一个Parent类型的引用指向Child对象，调用doSomething方法时也会执行Child类中的实现。

3.3.3 多态性（Polymorphism）、动态绑定（Dynamic Binding）以及泛型编程（Generic Programming）

1.多态性（Polymorphism）：
多态性是指子类型的对象可以在需要父类型的地方使用。在Java中，这意味着一个对象可以被视为它的任何父类的实例。
2.动态绑定（Dynamic Binding）：
动态绑定是指Java虚拟机（JVM）在运行时动态决定使用哪个方法实现。当一个方法被调用时，JVM会根据传入参数的实际类型（而不是引用类型）来决定调用哪个方法。
所以方法匹配（Method Matching）和方法绑定（Method Binding）都是Java中动态绑定（Dynamic Binding）的一部分，但它们发生在不同的阶段。
3.泛型编程（Generic Programming）：
我们在之前的数据结构课中详细学过这一机制。
泛型编程是Java提供的一种机制，允许类和方法在不知道具体类型的情况下被定义和使用。泛型参数在实例化时指定，这允许编写出更通用、更灵活的代码。
泛型类和接口可以使用类型参数（如T、E等），这些参数在实例化类或接口时确定。

下面的代码体现了前两个概念。

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

```
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }

    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

class GraduateStudent extends Student {
    // ...
}

class Student extends Person {
    public String toString() {
        return "Student";
    }
}

class Person /* extends Object*/ {
    public String toString() {
        return "Person";
    }
}
```

输出的结果如下。

```
Student
Student
Person
java.lang.Object@12345678
```

最后一行的哈希码可能不一样。
这里的多态性体现在m方法接受一个Object类型的参数，这意味着它可以接收任何类型的对象，包括GraduateStudent、Student、Person和Object。
动态绑定是指在运行时根据对象的实际类型来调用相应的方法。在这个例子中，toString()方法在Object类中定义，并被Person、Student和GraduateStudent类重写。当m方法被调用时，JVM根据传入对象的实际类型来决定调用哪个toString()方法。

下面的代码体现了泛型编程。

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

    public Box(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}
```

在这个泛型类Box中，T是一个类型参数，你可以在实例化Box类时指定具体类型，如Box<String>或Box<Integer>。

3.3.3.1 动态绑定（Dynamic Binding）

假设有一个对象 o 是类 C1 的实例（o = new C1()），其中 C1 是 C2 的子类，C2 是 C3 的子类，依此类推，直到 Cn-1 是 Cn 的子类。
Cn 是最一般的类（例如 Object），而 C1 是最具体的类（即 o 的具体类型）。
当对象 o 调用方法 m 时，Java虚拟机（JVM）会按照继承层次结构从 C1 开始向上搜索方法 m 的实现。
搜索顺序是从最具体的类（C1）到最一般的类（Cn，例如 Object）。
一旦找到第一个匹配的方法实现，搜索停止，并调用该方法。

比如前面说的这个例子。

```
public class PolymorphismDemo {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }

    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

class GraduateStudent extends Student {
    // ...
}

class Student extends Person {
    public String toString() {
        return "Student";
    }
}

class Person extends Object {
    public String toString() {
        return "Person";
    }
}
```

当 m(new GraduateStudent()) 被调用时，GraduateStudent 对象的实际类型是 GraduateStudent，但由于没有自己的 toString() 方法，它将使用 Student 类的 toString() 方法，输出 "Student"。
当 m(new Student()) 被调用时，Student 对象的实际类型是 Student，输出 "Student"。
当 m(new Person()) 被调用时，Person 对象的实际类型是 Person，输出 "Person"。
当 m(new Object()) 被调用时，Object 对象的实际类型是 Object，输出 "java.lang.Object@12345678"（Object 类的默认 toString() 方法返回值）。

3.3.4 类型转换（Casting）

类型转换是将一个类的对象转换为另一个类的对象。在继承体系中，子类的对象可以被视为父类的对象，因为子类继承了父类的所有属性和方法。

3.3.4.1 隐式转换（Implicit Casting）

在特定情况下，Java编译器会自动将一种类型的值转换为另一种类型，而不需要开发者显式地编写转换代码。这种转换通常发生在基本数据类型（primitive types）之间，以及对象和它们的包装类（wrapper classes）之间。
在这里是当一个子类的对象被赋值给父类类型的变量时，会自动进行类型转换。这种转换是隐式的，不需要显式地进行类型转换。
因此向上转型（Upcasting）都是安全的，因此都是隐式的。
例如前面例子中的一下代码。

```
m(new Student());
```

等价于下面代码

```
Object o = new Student(); // 隐式转换
m(o);
```

Object o = new Student(); 将 Student 类型的对象赋值给 Object 类型的变量 o，这时发生了隐式转换，因为 Student 是 Object 的子类，所以 Student 类的对象可以被视为 Object 类型。

下面的代码是常见的隐式转换。

```
int i = 1;
double a = i; // 隐式转换：int 到 double
```

3.3.4.2 显式转换（Explicit Casting）

显式类型转换是开发者明确指示编译器进行特定类型转换的操作。这通常在需要向下转型（Downcasting）时使用，即从父类类型转换为子类类型。
但是需要注意的是向下转型（Downcasting）并不总是安全的。向下转型需要显式进行，并且只有在确信对象是目标类型时才应该进行。
例如前面的代码左右颠倒就无法运行，如下所示。

```
Student b = o;
```

这里会有编译错误。因为如果 o 的类型不是 Student 或 Student 的子类，那么尝试将 o 赋值给 Student 类型的变量 b 将导致编译失败。
Java 是一种静态类型语言（变量类型声明后，其类型在整个程序的生命周期内都是固定的，相反的非静态类型语言比如Python里一个变量的类型就是可以改变的），编译器在编译时需要知道所有变量和对象的确切类型。如果 o 可能不是 Student 类型，直接赋值给 Student 类型的变量可能导致类型不匹配的问题。

当你需要将一个对象从父类类型转换为子类类型时，你必须使用显式类型转换来告诉编译器对象的实际类型。这是因为编译器在编译时只知道变量的静态类型，而不知道对象的实际运行时类型。
显式类型转换使用圆括号 () 语法，其中包含目标类型和要转换的对象。
因此上面的代码正确的命令如下。

```
Student b = (Student)o;
```

这行代码显式地告诉编译器 o 实际上是 Student 类型的对象。即使 o 被声明为 Object 类型，编译器现在也知道 o 是 Student 类型，因此可以安全地将 o 赋值给 Student 类型的变量 b。
这种转换是必要的，因为如果 o 不是 Student 类型，直接赋值将导致运行时错误（ClassCastException）。
这与基础数据类型的转换相似，如下。

```
int i = (int)1.23;
```

这行代码将 double 类型的值 1.23 显式转换为 int 类型。这种转换是必要的，因为 int 类型不能直接接受 double 类型的值。
这种转换确保了类型安全，防止了可能的数据精度丢失或范围问题。

由于向下转型可能导致类型不匹配和运行时错误。在进行向下转型之前，通常使用 instanceof 操作符来检查对象的实际类型，以确保转型的安全性。
instanceof是一个二元操作符，用于检查对象是否是某个类或其子类的实例。如果检查结果为true，说明对象是该类的实例；否则为false。
示例如下。

```
Object myObject = new Student();
...
if (myObject instanceof Student) {
    System.out.println("The student GPA is " + ((Student)myObject).getGPA());
}
```

这里先创建了一个Student对象，并将其赋值给Object类型的变量myObject。然后使用instanceof操作符检查myObject是否是Student类的实例。如果是，就将myObject强制转换为Student类型，并调用getGPA()方法获取学生的GPA值，然后打印出来。
下面给出了一个前面圆形和长方形类的例子。

```
public class CastingDemo{
    public static void main(String[] args){
        Object object1 = new Circle(1);
        Object object2 = new Rectangle(1, 1);
        displayObject(object1);
        displayObject(object2);
    }

    public static void displayObject(Object object) {
        if (object instanceof Circle) {
            System.out.println("The circle radius is " +
((Circle)object).getRadius());
            System.out.println("The circle diameter is " +
((Circle)object).getDiameter());
        } else if (object instanceof Rectangle) {
            System.out.println("The rectangle width is " +
((Rectangle)object).getWidth());
        }
    }
}
```

在 instanceof 操作符的帮助下判断对象的类型，从而进行类型转换以访问特定类的方法，这是一种安全的方法，可以避免类型转换错误。

3.3.3 equals()方法

equals()方法用于比较两个对象的内容是否相等。在Java中，Object类提供了equals()方法的默认实现，该实现仅比较两个对象的引用是否相同（即是否是同一个对象）。
默认实现如下。

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

这里只是简单地比较参数对象是否和this对象是一个对象。
因此下面的代码展示了Circle类方法重写的equals()方法，这个方法比较了两个Circle对象的半径是否相等。

```
public boolean equals(Object o) {
    if (o instanceof Circle) {
        return radius == ((Circle)o).radius;
    } else {
        return false;
    }
}
```

如果o是Circle类的实例，那么将其强制转换为Circle类型，并比较当前对象的半径（radius）和传入对象的半径是否相等。如果半径相等，返回true；否则，返回false。如果传入的对象不是Circle类的实例，直接返回false。

3.5 泛型编程（Generic programming）

泛型编程（Generic programming）是一种编程范式，它允许编写出可以对不同类型的数据进行操作的代码，而不需要指定具体的数据类型。
我们现在对前面的知识进行一个总结，多态（Polymorphism）允许使用一个接口来处理不同类型的对象。在Java等语言中，多态通常通过继承和方法重写来实现。
多态包含两部分：
1.方法的多态性：如果一个方法的参数类型被声明为一个超类（例如Object），那么你可以向这个方法传递任何该超类的子类对象（例如Student或String）。
2.动态绑定：在运行时，会根据实际传递的对象类型来决定调用哪个具体的方法实现。这意味着同一个方法调用可以有不同的行为，这取决于调用它的对象的实际类型。
而泛型编程是多态的一种扩展，它允许方法或类在定义时不指定具体的数据类型，而是在调用时指定。这样可以编写出更加通用和可重用的代码。
定义泛型方法或类时，可以使用类型参数（例如）来表示方法或类可以处理的任何类型。泛型编程还提供了类型安全泛型提供了编译时类型检查，这意味着如果代码中存在类型不匹配的问题，编译器会在编译时就报错，而不是在运行时。
在数据结构中，泛型编程和多态的结合非常有用，因为它们允许你创建可以处理任何类型元素的通用数据结构，如列表、集合、映射等。
其的优势有以下三点：
1.通用数据结构：例如，你可以创建一个泛型列表List<T>，它可以存储任何类型的元素，如List<Student>或List<String>。
2.类型安全：泛型列表在编译时检查元素类型，确保你只能添加和检索正确类型的元素，从而避免类型

转换错误。
3.代码重用：泛型数据结构可以在不同的上下文中重复使用，而不需要为每种数据类型编写专门的代码。

3.6 ArrayList类

我们前面说array时，我们说array的大小是固定的，那有没有没有固定大小的数组呢？有的，下面介绍的ArrayList类正是一个可变的数组，可以储存任意数量的对象，它同时也是前面说的泛型编程的一个体现，java.util.ArrayList<T>，其中<T>是一个类型参数，可以在创建ArrayList实例时指定具体的类型。例如，你可以创建一个ArrayList来存储String类型的对象，代码如下。

```
ArrayList<String> stringList = new ArrayList<String>();
```

下图介绍了ArrayList类中的各个方法。

1.ArrayList()
创建一个空的列表。
2.add(Object o): void
在列表的末尾添加一个新的元素o。
3.add(int index, Object o): void
在列表的指定索引位置index添加一个新的元素o。
4.clear(): void
移除列表中的所有元素。
5.contains(Object o): boolean
如果列表包含元素o，则返回true，否则返回false。
6.get(int index): Object
返回列表中指定索引位置index的元素。
7.indexOf(Object o): int
返回列表中第一个匹配元素o的索引。
8.isEmpty(): boolean
如果列表不包含任何元素，则返回true，否则返回false。
9.lastIndexOf(Object o): int
返回列表中最后一个匹配元素o的索引。
10.remove(Object o): boolean
移除列表中的第一个匹配元素o，如果成功移除，则返回true，否则返回false。
11.size(): int
返回列表中元素的数量。
12.remove(int index): Object
移除列表中指定索引位置index的元素，并返回被移除的元素。
13.set(int index, Object o): Object
将列表中指定索引位置index的元素设置为o，并返回被替换的元素。

下面展示两段代码。

```
public class TestArrayList {
    public static void main(String[] args) {  // Warnings
        java.util.ArrayList citylist = new java.util.ArrayList();
        citylist.add("London"); citylist.add("New York"); citylist.add("Paris");
        citylist.add("Toronto"); citylist.add("Hong Kong");
        System.out.println("List size? " + citylist.size());
        System.out.println("Is Toronto in the list? " +
```

```
                citylist.contains("Toronto"));
        System.out.println("The location of New York in the list? " +
                citylist.indexOf("New York"));
        System.out.println("Is the list empty? " + citylist.isEmpty()); // false
        citylist.add(2, "Beijing");
        citylist.remove("Toronto");
        for (int i = 0; i < citylist.size(); i++)
            System.out.print(citylist.get(i) + " ");
        System.out.println();
        // Create a list to store two circles
        java.util.ArrayList list = new java.util.ArrayList();
        list.add(new Circle(2));
        list.add(new Circle(3));
        System.out.println(((Circle)list.get(0)).getArea());
    }
}
```

```
public class TestArrayList {
    public static void main(String[] args) {
        // Generics: eliminates warnings
        java.util.ArrayList<String> citylist = new java.util.ArrayList<String>
();
        citylist.add("London"); citylist.add("New York"); citylist.add("Paris");
        citylist.add("Toronto"); citylist.add("Hong Kong");
        System.out.println("List size? " + citylist.size());
        System.out.println("Is Toronto in the list? " +
                citylist.contains("Toronto"));
        System.out.println("The location of New York in the list? " +
                citylist.indexOf("New York"));
        System.out.println("Is the list empty? " + citylist.isEmpty()); // false
        citylist.add(2, "Beijing");
        citylist.remove("Toronto");
        for (int i = 0; i < citylist.size(); i++)
            System.out.print(citylist.get(i) + " ");
        System.out.println();
        // Create a list to store two circles
        java.util.ArrayList<Circle> list = new java.util.ArrayList<Circle>();
        list.add(new Circle(2));
        list.add(new Circle(3));
        System.out.println(list.get(0).getArea()); // no casting needed
    }
}
```

上面的代码会有风险警告，因为ArrayList可以储存任何类型的对象。这可能导致在运行时因为添加不兼容的类型出现ClassCastException。
下面的代码使用泛型确保了列表中只能存储指定类型的元素（String和Circle）尝试添加其他类型的元素会导致编译错误，从而在编译时就避免了潜在的问题。

3.6.1 MyStack类

下面是一个ArrayList的应用。

实际代码如下。

```
public class MyStack {
    private java.util.ArrayList list = new java.util.ArrayList();

    public void push(Object o) {
        list.add(o);
    }

    public Object pop() {
        Object o = list.get(getSize() - 1);
        list.remove(getSize() - 1);
        return o;
    }

    public Object peek() {
        return list.get(getSize() - 1);
    }

    public int search(Object o) {
        return list.lastIndexOf(o);
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }

    public int getSize() {
        return list.size();
    }

    public String toString() {
        return "stack: " + list.toString();
    }
}
```

```
public class TestMyStack {
    public static void main(String[] args) {
        MyStack s = new MyStack();
        s.push(1);
        s.push(2);
        System.out.println(s.pop()); // 2
        System.out.println(s.pop()); // 1

        MyStack s2 = new MyStack();
        s2.push("New York");
        s2.push("Washington");
        System.out.println(s2.pop()); // Washington
        System.out.println(s2.pop()); // New York
    }
}
```

3.7 访问修饰符

我们再复习一遍访问修饰符。
1.private：
这是最严格的访问级别。
private成员只能被定义它们的类本身访问。
不能在同一个包中的其他类或子类中访问，除非通过公共或受保护的方法。
2.默认（default）：
当没有指定访问修饰符时，成员具有包级访问权限（也称为默认访问级别）。
默认情况下，成员可以被同一个包中的任何其他类访问，但不能被子类访问，除非子类也在同一个包中。
3.protected：
protected成员可以被定义它们的类本身、同一个包中的任何其他类以及不同包中的子类访问。
这意味着即使子类不在同一个包中，它们也可以访问protected成员。
4.public：
这是最宽松的访问级别。
public成员可以被任何其他类访问，无论它们是否在同一个包中，也无论它们是否是子类。
因此可见性顺序从最严格到最宽松的顺序是：private < 默认 < protected < public。
区别如下图所示。

下图展示了一个例子。

在UML中，各种符号和标记用于表示类的不同特性和成员的可见性。
+：表示公共（public）访问权限。
-：表示私有（private）访问权限。
~：表示默认（default）或包级（package）。
#：表示受保护（protected）访问权限。
下划线（underlined）：表示静态（static)成员。

现在回到类的问题上来。
关于方法重写（Override）和访问修饰符有一个规则：子类在覆盖父类中的方法时，不能降低该方法的访问级别。
但是子类可以将提高该方法的访问级别。
如果父类中的一个方法是受保护的（protected），子类在覆盖这个方法时，可以将其访问级别改为public。这是因为public的访问级别比protected更宽松，允许更多的代码访问这个方法。
注意这里private无法被子类提高访问等级，因为子类不能直接访问这个方法，更不能重写（Override）它或改变其访问修饰符。这条规则的核心是，子类在覆盖父类中的方法时，不能将访问级别设置得比父类中的方法更严格。换句话说，子类不能"削弱"方法的访问权限。
例如，如果一个方法在父类中被定义为public，那么在子类中覆盖这个方法时，它也必须是public。子类不能将其改为protected或private。
这个规则的目的是为了保持或增加方法的可见性，以确保子类不会意外地限制对这些方法的访问。这有助于维护代码的兼容性和可预测性。
通过不允许降低访问级别，可以确保任何依赖于父类方法访问级别的代码，在切换到子类时仍然能够正常工作。

3.7.1 方法重写（Override）的规则

1.可访问性：
只有可访问的实例方法才能被子类覆盖。这意味着子类只能覆盖父类中具有default或public或protected访问级别的方法。
2.私有方法：
private方法不能被子类覆盖，因为它在定义它的类外部是不可见的。子类无法访问父类中的private方法，因此也就无法覆盖它。

3.完全不相关的私有方法：
如果子类中的一个方法在父类中被定义为private，那么这两个方法实际上是完全不相关的。子类中的方法并不是对父类中方法的覆盖，而是一个新的方法。

3.7.2 静态（static）方法的规则

1.继承：
静态方法可以被子类继承。这意味着子类可以使用父类中的静态方法。
2.不能覆盖：
静态方法不能被子类覆盖。这是因为静态方法属于类本身，而不是类的实例。因此，子类不能提供一个与父类中静态方法具有相同签名的新实现。
3.隐藏（Hide）：
如果子类中重新定义了一个与父类中静态方法具有相同名称和参数的静态方法，那么父类中的静态方法会被隐藏。这意味着在子类中调用该方法时，将执行子类中的定义，而不是父类中的静态方法。

3.7.3 final修饰符

当一个变量被声明为final，意味着这个变量是一个常量，它的值在初始化后不能被改变。
例如：final static double PI = 3.14159;声明了一个名为PI的常量，它是一个静态的final变量，表示圆周率π的值，并且这个值在程序运行期间是不可更改的。

当一个方法被声明为final，意味着这个方法不能被子类覆盖（override）。
子类不能提供覆盖该方法的新实现，因此final方法确保了方法的行为在所有子类中都是一致的。

当一个类被声明为final，意味着这个类不能被其他类继承（extend）。
没有其他类可以成为final类的子类，因此final类限制了类的扩展，确保了类的设计不会被其他开发者修改。

4. 练习

4.1 基础练习

1.以下代码中的Student类是否是不可变的，如果不是，原因是什么?

```
public class Student {
    private int id;
    private BirthDate birthDate;
    public Student(int ssn, int year, int month, int day) {
        id = ssn;
        birthDate = new BirthDate(year, month, day);
    }
    public int getId() {
        return id;

    }
    public BirthDate getBirthDate() {
        return birthDate;
    }
}

public class BirthDate {
    private int year;
    private int month;
```

```
    private int day;
    public BirthDate(int newYear, int newMonth, int newDay) {
        year = newYear;
        month = newMonth;
        day = newDay;
    }
    public void setYear(int newYear) {
        year = newYear;
    }
    public int getYear() {
        return year;
    }
}

public class Test {
    public static void main(String[] args) {
        Student student = new Student(123, 1998, 1, 1);
    }
}
```

不是可变的，因为我们可以这么修改里面的对象：student.getBirthDate().setYear(2050);
你可以在main方法中尝试一下。

```
public class Test {
    public static void main(String[] args) {
        Student student = new Student(111223333, 1998, 1, 1);
        student.getBirthDate().setYear(2050);
        // Now the student birth year is changed:
        System.out.println(student.getBirthDate().getYear()); // 输出的结果是2050
    }
}
```

2.对于下面的代码，运行结果是什么?

```
String s1 = "Hello World";
String s2 = new String("Hello World");
String s3 = "Hello World";
System.out.println((s1 == s2) + " " + (s1.equals(s2)));
```

false true
s1 和 s3 都引用字符串池中的同一个字符串对象，而使用new关键字的s2是引用的是堆中的一个新创建的字符串对象。
前面说过字符串是引用类型，==比较的就是其引用之，因此这里s1和s2的引用对象不一样，结果是false。后者equals比较的是引用的对象的内容是否相同。由于s1和s2的内容一样，所以s1.equals(s2)的结果是true。

3.类似的问题如下，运行结果是什么?

```
String s1 = "Hello World";
String s2 = new String("Hello World");
String s3 = "Hello World";
System.out.println((s1 == s3) + " " + (s1.equals(s3)));
```

true true
因为s1和s3的引用对象一致，所以第一个结果是true，而内容一致，所以第二个结果也是true。
刚刚两题里s1，s2，s3的关系，如下图所示。

4.下面说法中错误的是哪个?

1. Abstract classes can have constructors, but cannot be instantiated
2. All methods in an abstract class must be abstract
3. Abstract methods do not have a body and must be implemented by subclasses
4. Abstract methods must be declared inside an abstract class
5. A subclass that does not implement/override all abstract methods from its abstract superclass must be declared abstract

错误的是2，抽象类中的方法不必须都是抽象的，它们可以有具体实现。

1. 抽象类可以有构造函数，但是不能被实例化。
2. 抽象方法没有方法体，必须由子类实现。
3. 抽象方法必须在抽象类中声明。
4. 如果子类没有实现/覆盖其抽象超类中的所有抽象方法，则该子类也必须声明为抽象。

5.下面使用了Overloading还是Overriding?

```
public class Test {
    public static void main(String[] args) {
        B a = new A();
        a.p(10.0);
        a.p(10);
    }
}

class B {
    public void p(double i) {
        System.out.println(i * 2);
    }
}

class A extends B {
    public void p(int i) {
        System.out.println(i);
    }
}
```

Overloading，因为A类里的p方法和B类中的p方法签名不同（参数类型不同）。

6.下面使用了Overloading还是Overriding?

```
public class Test {
    public static void main(String[] args) {
        B a = new A();
        a.p(10.0);
        a.p(10);
    }
}

class B {
```

```
    public void p(double i) {
        System.out.println(i * 2);
    }
}

class A extends B {
    public void p(double i) {
        System.out.println(i);
    }
}
```

Overriding，因为A类中p方法的签名与B类中的p方法签名相同（参数类型相同）。
当然你可以自己输出一下这里的结果检验这两道题的结果，你可能会发现一个比较困惑的地方，那就是上一题的结果都是20，而不是一个是10，一个是20。

你可以尝试修改一下，会发现。

这里我将两个方法的类型进行了对调，这里提示报错，方法接受int类型，但是我给的是double类型，我再修改一下数据类型。

这里虽然显示了一个override，但过段时间会消失，我们能清楚地看到这里是2usages，也就是这两个方法都是在B类里就执行了，这不难理解，因为这两个数据类型都是int，这两个对象被声明为B类，所以调用对应的方法，就是B类的这个接受int类型的方法。但是我们回到之前的问题，一个是10.0，一个是10，为什么都被double类型的方法接受了呢?
我们把前面的几个问题综合在一起，可以发现。

第10行报错不难理解，因为b是B类型的，B类型的q方法需要int参数，而提供的是double参数。而对于第11行，c虽然实际上是一个A类对象，但是被声明为B类型，编译器在编译时会根据变量的静态类型（即B类型）来解析方法调用。由于B类中没有定义接受double类型参数的q方法，编译器无法找到匹配的方法，因此会报错。
这里其实也是因为一个子类的对象被赋值给父类类型的变量时，会发生隐式转换。通过隐式转换，编译器将a当作B类去处理，因此编译器会按照B类型去处理。
详情请看下面的知识点补充。（也可以多看看前面的多边形的例子，由于这里声明时候使用的类型是B，所以编译器在B类中发现了可以执行的方法，就直接执行了这个方法。）

4.1.1 编译时解析/静态绑定（Static Binding）和运行时解析/动态绑定（Dynamic Binding）

1.编译时解析（Static Binding）
在编译时，编译器会根据变量的静态类型（即声明时的类型）和方法的签名来确定调用哪个方法。
例如，如果你有一个接口 I 和一个实现了 I 的类 C，那么在编译时，如果有一个 I 类型的变量调用方法，编译器会根据变量的静态类型（即 I）来解析方法调用，而不是变量的实际类型（可能是 C）。

它与方法匹配有所不同，方法匹配发生在编译时，它基于变量或参数的静态类型来决定调用哪个方法。这是多态性的一种表现，允许在同一个类中定义多个同名但参数列表不同的方法（方法重载）。编译器会根据调用方法时提供的参数类型来选择正确的方法版本。
例如，如果有一个类中定义了两个名为 doSomething 的方法，一个接受 int 参数，另一个接受 double 参数，那么编译器会根据调用时提供的参数类型来决定使用哪个方法。
2.运行时解析（Dynamic Binding）
在运行时，如果子类重写了父类中的方法，JVM会根据对象的实际类型来调用相应的方法。

因此我们回到前面的问题，对于变量a来说，现在其声明的类型是B，那么在调用方法时，编译器根据其现在声明的类型B安排了对应的方法，其接受double类型，虽然10是int类型，其可以自动转换为double类，所以依然这个方法会接受。而如果方法只接受int类型，虽然其实际类型是A，但是编译器依然按照声明的类型B寻找对应方法，发现接受的是int类，因此会报编译错误。如果变量a的声明类型是A，那么就会是符合方法匹配（Method Matching），编译器会根据调用方法时提供的参数类型来选择正确的方法版本。

7.下列关于equals()方法说法错误的是哪一个?

1. Method signature is public boolean equals(Object obj)
2. Returns true if obj is not null
3. Checks if this and obj refer to the same object
4. Uses instanceof to check if obj is of the same type
5. Compares each field relevant to equality after casting obj

错误的是2，因为equals()方法的目的是判断两个对象是否相等，即它们的内容是否相同，不是null完全不足以说明。

1. 这是equals()方法的正确签名。
2. 这是Object类中equals()方法的默认行为，但通常在重写时会进行更深入的内容比较。
3. 在使用instanceof检查之后，通常会将对象转换为适当的类型，并比较内容。
4. 这是equals()方法重写时的常见做法，即在确认对象类型相同后，比较每个与等价性相关的字段。

4.2 进阶练习

4.2.1 MyPoint类

类中包含：
表示坐标的实例变量（数据字段）x 和 y，以及它们的获取器（getter）方法。
一个空构造函数，用于创建一个坐标为 (0.0, 0.0) 的点。
一个构造函数，用于根据指定的坐标创建一个点。
一个名为 distance 的实例方法，用于返回此点与另一个 MyPoint 类型点之间的距离。
一个同样名为 distance 的静态方法，用于返回两个 MyPoint 对象之间的距离。
并且编写一个测试程序，创建三个点：(0.0, 0.0)，(10.25, 20.8) 和 (13.25, 24.8)，并使用两种 distance 实现来显示它们之间的距离。
注意这里实例方法和静态方法之间的区别。
示例代码如下。

```
public class MyPoint {
    // 实例变量
    double x;
    double y;

    // 空构造函数，创建一个坐标为 (0.0, 0.0) 的点
    public MyPoint() {
        this.x = 0.0;
        this.y = 0.0;
    }

    // 根据指定的坐标创建一个点
    public MyPoint(double x, double y) {
        this.x = x;
        this.y = y;
    }
```

```
    // 获取器方法
    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    // 实例方法，返回此点与另一个 MyPoint 类型点之间的距离
    public double distance(MyPoint other) {
        return Math.sqrt(Math.pow(this.x - other.x, 2) + Math.pow(this.y -
other.y, 2));
    }

    // 静态方法，返回两个 MyPoint 对象之间的距离
    public static double distance(MyPoint p1, MyPoint p2) {
        return Math.sqrt(Math.pow(p1.x - p2.x, 2) + Math.pow(p1.y - p2.y, 2));
    }

    // 测试程序
    public static void main(String[] args) {
        // 创建三个点
        MyPoint p1 = new MyPoint(0.0, 0.0);
        MyPoint p2 = new MyPoint(10.25, 20.8);
        MyPoint p3 = new MyPoint(13.25, 24.8);

        // 使用实例方法计算距离
        double distance1 = p1.distance(p2);
        System.out.println("Distance between p1 and p2 (instance method): " +
distance1);

        // 使用静态方法计算距离
        double distance2 = MyPoint.distance(p1, p2);
        System.out.println("Distance between p1 and p2 (static method): " +
distance2);
    }
}
```

4.2.2 大数类

编写一个函数，输出前10个具有50位十进制数字且能被2或3整除的数。
示例代码如下。

```
import java.math.BigInteger;

public class BigDivisibleNumbers {
    public static void main(String[] args) {
        printNumbers();
    }

    public static void printNumbers() {
        // 定义50位数的最小值（10^49）
        BigInteger current = BigInteger.TEN.pow(49);
```

```
        // 定义常量大数2和3，用于取模运算
        BigInteger TWO = BigInteger.valueOf(2);
        BigInteger THREE = BigInteger.valueOf(3);

        int count = 0;
        while (count < 10) {
            // 检查是否能被2或3整除
            if (current.mod(TWO).equals(BigInteger.ZERO) ||
                    current.mod(THREE).equals(BigInteger.ZERO)) {
                // 转换为字符串并补零（确保输出为50位）
                String numStr = current.toString();
                System.out.println(numStr);
                count++;
            }
            current = current.add(BigInteger.ONE); // 递增
        }
    }
}
```

4.2.3 Person类

要求如下：
Person类及其两个子类 Student 和 Employee。
Employee 类的两个子类 Faculty 和 Staff。
Person 有 name（姓名）、address（地址）、phone（电话号码）和 email（电子邮件）。
Student 有 classStatus（年级状态，如新生、大二、大三或大四），定义为常量。
Employee 有 office（办公室）、salary（薪水）和 dateHired（雇用日期）。
Faculty 有 officeHours（办公时间）和 rank（职称）。
Staff 有 title（职称）。
每个类添加一个只接受 name 参数的构造函数。
重写每个类的 toString 方法，以显示类名和人名。
确定 Person 类中 name 的访问修饰符。
创建一个数组，包含 Person、Student、Employee、Faculty 和 Staff 对象。
使用多态性调用它们的 toString() 方法。
示例代码如下。

```
import java.time.LocalDate;

// Person class
class Person {
    protected String name;
    protected String address;
    protected String phone;
    protected String email;

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

    @Override
    public String toString() {
        return "Person: " + name;
    }
```

```
}

// Student class
class Student extends Person {
    public static final String[] STATUS = {"freshman", "sophomore", "junior",
"senior"};
    private int classStatus;

    public Student(String name, int status) {
        super(name);
        this.classStatus = status;
    }

    @Override
    public String toString() {
        return "Student: " + name + ", Status: " + STATUS[classStatus];
    }
}

// Employee class
class Employee extends Person {
    protected String office;
    protected double salary;
    protected LocalDate dateHired;

    public Employee(String name, String office, double salary, LocalDate
dateHired) {
        super(name);
        this.office = office;
        this.salary = salary;
        this.dateHired = dateHired;
    }

    @Override
    public String toString() {
        return "Employee: " + name + ", Office: " + office;
    }
}

// Faculty class
class Faculty extends Employee {
    private String officeHours;
    private String rank;

    public Faculty(String name, String office, double salary, LocalDate
dateHired, String officeHours, String rank) {
        super(name, office, salary, dateHired);
        this.officeHours = officeHours;
        this.rank = rank;
    }

    @Override
    public String toString() {
        return "Faculty: " + name + ", Rank: " + rank;
    }
```

```
}

// Staff class
class Staff extends Employee {
    private String title;

    public Staff(String name, String office, double salary, LocalDate dateHired,
String title) {
        super(name, office, salary, dateHired);
        this.title = title;
    }

    @Override
    public String toString() {
        return "Staff: " + name + ", Title: " + title;
    }
}

// 测试程序
public class TestPerson {
    public static void main(String[] args) {
        Person[] people = new Person[5];
        people[0] = new Person("John Doe");
        people[1] = new Student("Jane Smith", 1);
        people[2] = new Employee("Alice Johnson", "123 Main St", 50000,
LocalDate.of(2020, 6, 1));
        people[3] = new Faculty("Bob Brown", "456 Elm St", 60000,
LocalDate.of(2018, 8, 15), "8am-5pm", "Associate Professor");
        people[4] = new Staff("Charlie Davis", "789 Maple St", 45000,
LocalDate.of(2019, 3, 20), "Manager");

        for (Person p : people) {
            System.out.println(p);
        }
    }
}
```

4.2.4 继承版MyStack类

将这次学习的MyStack类改为继承自ArrayList<Object>类。此外，还需要编写一个测试程序，该程序提示用户输入五个字符串，并将它们以相反的顺序显示出来。
示例代码如下。

```
import java.util.ArrayList;
public class MyStack extends ArrayList<Object> {
    // 检查栈是否为空
    public boolean isEmpty() {
        return super.isEmpty();
    }

    // 获取栈的大小
    public int getSize() {
        return super.size();
    }
```

```
    // 返回栈顶元素
    public Object peek() {
        if (isEmpty()) return null;
        return super.get(getSize() - 1);
    }

    // 移除并返回栈顶元素
    public Object pop() {
        if (isEmpty()) return null;
        return super.remove(getSize() - 1);
    }

    // 向栈中添加元素
    public void push(Object o) {
        super.add(o);
    }

    // 查找元素在栈中的位置
    public int search(Object o) {
        return super.lastIndexOf(o);
    }

    // 返回栈的字符串表示
    public String toString() {
        return "stack: " + super.toString();
    }
}
```

```
import java.util.Scanner;

public class TestMyStack {
    public static void main(String[] args) {
        MyStack stack = new MyStack();
        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter five strings:");
        for (int i = 0; i < 5; i++) {
            stack.push(scanner.nextLine());
        }

        scanner.close();

        System.out.println("Strings in reverse order:");
        while (!stack.isEmpty()) {
            System.out.println(stack.pop());
        }
    }
}
```