

## Module2: Java和OOP是什么

### 1 Java是什么

#### C++和Java的区别

- 程序员本质上避免了许多在C和C++等语言中常见的内存和安全问题。这主要是因为其创造者在语言中内置了一些非常强大的机制，如自动内存管理。

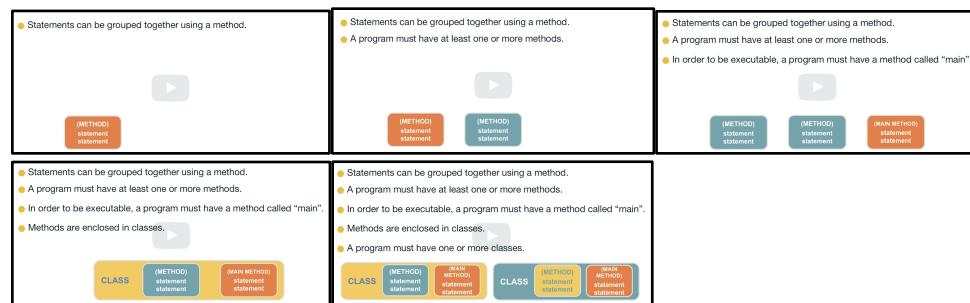
#### C++和C#的区别

- C#是微软开发的，C#深度集成了微软的.NET框架，提供了丰富的类库和工具支持，极大地方便了开发者的工作。

### 2 Comment syntax - //

### 3 一个最简单的Java program

```
1 // A simple program that prints text on the terminal
2 public class HelloWorld {
3     public static void main(String[] args) {
4         System.out.println("Hello,World!");
5     }
6 }
```



- 先有statement ( 比如System.out.println("Hello,World!"); )
- 再有method ( 函数，比如public static void main ( ) {}。method以小写开头main )
- 再有class (类，public class HelloWorld。类以大写开头Main )

- 一个java program必须要有至少一个class和一个main函数
- 并不是每一个class必须要有一个main函数  
*(只有一个class的java程序，main函数就在这个class内  
如果有多个class的java程序，main通常单独作为一个class存在)*

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

1. BankAccount 类: 表示银行账户

```
java
public class BankAccount {
    private String accountNumber;
    private double balance;

    public BankAccount(String accountNumber, double initialBalance) {
        this.accountNumber = accountNumber;
        this.balance = initialBalance;
    }

    public String getAccountNumber() {
        return accountNumber;
    }

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        }
    }
}
```

2. Customer 类: 表示客户

```
java
public class Customer {
    private String name;
    private BankAccount bankAccount;

    public Customer(String name, BankAccount bankAccount) {
        this.name = name;
        this.bankAccount = bankAccount;
    }

    public String getName() {
        return name;
    }

    public BankAccount getBankAccount() {
        return bankAccount;
    }
}
```

3. Bank 类: 管理账户的银行

```
java
import java.util.HashMap;
import java.util.Map;

public class Bank {
    private Map<String, BankAccount> accounts = new HashMap<>();

    public void addAccount(BankAccount account) {
        accounts.put(account.getAccountNumber(), account);
    }

    public BankAccount getAccount(String accountNumber) {
        return accounts.get(accountNumber);
    }
}
```

4. Main 类: 启动应用程序

```
java
public class Main {
    public static void main(String[] args) {
        // 创建银行
        Bank bank = new Bank();

        // 创建银行账户和客户
        BankAccount account1 = new BankAccount("123456", 1000.0);
        Customer customer1 = new Customer("Alice", account1);

        BankAccount account2 = new BankAccount("789012", 500.0);
        Customer customer2 = new Customer("Bob", account2);

        // 添加账户到银行
        bank.addAccount(account1);
        bank.addAccount(account2);

        // 操作账户
        customer1.getBankAccount().deposit(500);
        customer2.getBankAccount().withdraw(200);

        // 输出结果
        System.out.println(customer1.getName() + "的账户余额: " + customer1.getBalance());
        System.out.println(customer2.getName() + "的账户余额: " + customer2.getBalance());
    }
}
```

只有一个class的java program

有多个class的java program

在一个Java项目中，你可以有多个包含`public static void main(String[] args)`方法的类。每个类都可以有自己的`main`方法。这是完全允许的，也是很常见的做法，尤其是在大型项目或开发过程中，需要在同一个项目中测试多个独立的类时。

#### 具体说明：

- 每个类独立：每个包含`main`方法的类都可以独立运行。你可以通过指定类名来运行具体的`main`方法。例如，如果你有三个类`ClassA`、`ClassB`和`ClassC`，每个类都有一个`main`方法，你可以选择运行任意一个类的`main`方法。

```
java
// ClassA.java
public class ClassA {
    public static void main(String[] args) {
        System.out.println("This is Class A");
    }
}

// ClassB.java
public class ClassB {
    public static void main(String[] args) {
        System.out.println("This is Class B");
    }
}

// ClassC.java
public class ClassC {
    public static void main(String[] args) {
        System.out.println("This is Class C");
    }
}
```

- 运行方式：在命令行或IDE中运行时，你需要指定要运行的类。例如：

- 在命令行中运行`ClassA`的`main`方法：  

```
java ClassA
```
- 在IDE中，通常可以右键点击包含`main`方法的类文件，然后选择“Run”来运行该类的`main`方法。

(一个java program中可以有多个main，每个class都可以有一个main函数)

## 4 编译器vs解释器

compiler编译器 - **translate一次**。生成一个字节码文件，之后都run字节码文件。

interpreter解释器 - **一行一行翻译，每次run都翻译。**

### Advantages

#### Compilers

- ◆ Generally Faster
- ◆ Only needs to compile once
- ◆ Ex: C, C++

#### Interpreters

- ◆ Require translation each time
- ◆ Ex: Python, PHP

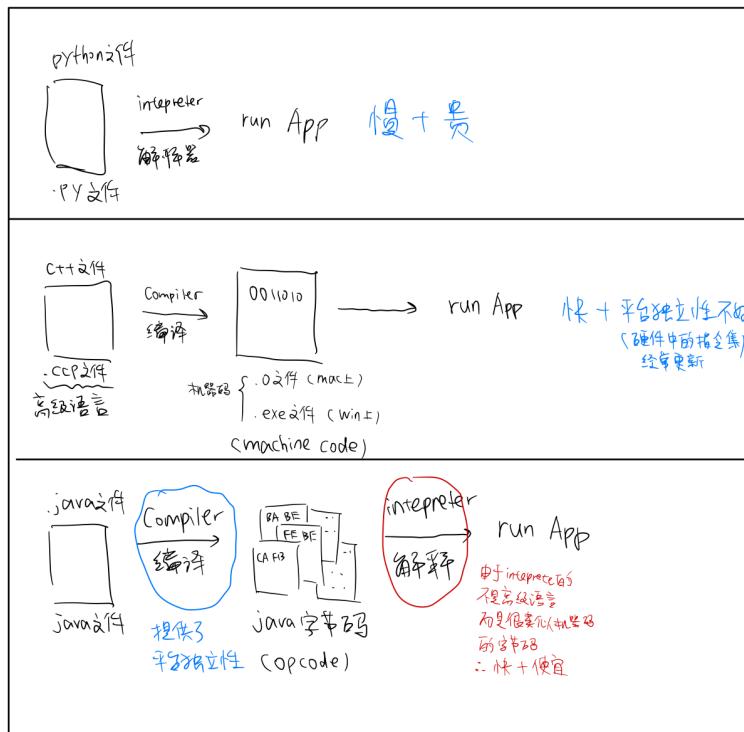
处理字节码的是指令集。

而大公司每年都会更新换代硬件更新指令集

而针对每一个电脑（硬件配置），都会有一个特别的compiler编译器

导致今天编写的C++语句，昨天能被旧compiler编译，明天就需要新的compiler了

## 5 Java的运行逻辑



红色部分也经常被称为 JVM - Java虚拟机

- java字节码文件也叫bicode文件
- 注意 - run的两步：
  - javac HelloWorld.java
  - java HelloWorld
- javac的c是compiler的意思

## 6 Java program - class (类)

### 6.1 所有命名的variable都叫identifier ( class, method, variable, element)

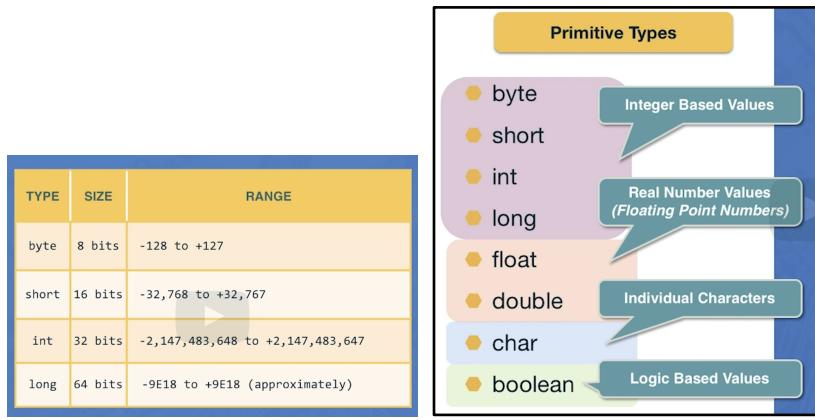
- They can contain letters, digits, \_, and \$
- A digit cannot be a starting character
- Identifiers cannot be reserved words

所有的命名首字母必须小写

### 6.2 Reserved words:

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

### 6.3 Type



- 每个variable必须有自己的type。
- Java中一共有8种原生types
- String不是一个Type，而是一种内建的class，可以看[这里](#)
- 特殊例子：在initialize一些int无法包含的数字时，如99,999.999,999
  - ❖ 我们无法用int，因为超出了2,147,483,647的范围
  - ❖ 我们因此需要定义long reallyBigNum = 99,999,999,999;
  - ❖ 然而，系统会默认99,999,999,999是个int，所以会出现compiler error
  - ❖ 我们需要手动在数字后面加个“L”让compiler知道我们故意定义这个int为long
  - ❖ 正确表达ong reallyBigNum = 99,999,999,999L;
  - ❖ 同理如以下两个例子：



6.4 java是static语言（静态语言），意思是如果你要用一个variable，你就要先declear这个variable。这意味着不可能像Python那样一行一行的run。

6.5 Java program - class (类) 例子：

*表示 variable 的 size*

*Java identifier*

*Class header*

*Main method*

*Variable declaration statement*

*Variable Assignment Statement (that variable initialization)*

*Print line*

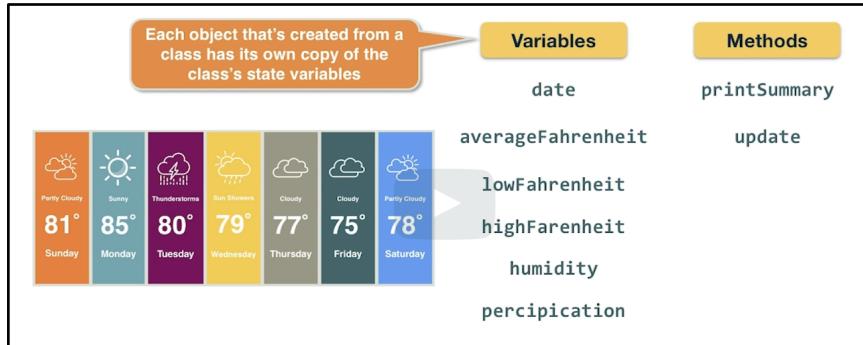
*also perform some simple numeric calculations.*

```

1 public class FahrenheitToCelsius {
2     public static void main(String[] args) {
3         int saturdayFahrenheit; // 单位°F
4         int sundayFahrenheit;
5         saturdayFahrenheit = 78;
6         sundayFahrenheit = 81;
7         double saturdayCelsius = (5.0/9) * (saturdayFahrenheit - 32);
8         double sundayCelsius = (5.0/9) * (sundayFahrenheit - 32);
9         System.out.println("Weekend Averages");
10        System.out.println("Saturday: " + saturdayCelsius);
11        System.out.println("Sunday: " + sundayCelsius);
12    }
13 }
```

12 boolean headToPark = saturdayFahrenheit > 69 ? true : false; (boolean语句 )

- 所有语句以";"结尾
- 变量：先declear，再initialize



variables是变量

methods是 改变/显示 变量的语句

```

code - bash - 79x34
lawn-143-215-285-97:code GeorgiaTech$ ls
FahrenheitToCelsius.java
lawn-143-215-285-97:code GeorgiaTech$ javac FahrenheitToCelsius.java
lawn-143-215-285-97:code GeorgiaTech$ ls
FahrenheitToCelsius.class
lawn-143-215-285-97:code GeorgiaTech$ java FahrenheitToCelsius
Weekend Averages
Saturday: 25.555555555555557
Sunday: 27.222222222222225
lawn-143-215-285-97:code GeorgiaTech$ 
```

```

public class FahrenheitToCelsius {
    public static void main(String[] args) {
        int saturdayFahrenheit;
        int sundayFahrenheit;
        saturdayFahrenheit = 78;
        sundayFahrenheit = 81;
        double saturdayCelsius = (5.0/9) * (saturdayFahrenheit - 32);
        double sundayCelsius = (5.0/9) * (sundayFahrenheit - 32);
        System.out.println("Weekend Averages");
        System.out.println("Saturday: " + saturdayCelsius);
        System.out.println("Sunday: " + sundayCelsius);
    }
}
```

## 7 状态变量 ( state variable )

状态变量是class中的variable

✗State variable within objects created from the same class will have the same values.

原因如下：

状态变量（State variables）是对象的实例变量，每个对象都有自己的一组实例变量。即使这些对象是由同一个类创建的，它们的状态变量也可以有不同的值。每个对象的状态变量独立于其他对象，这正是面向对象编程的重要特性之一。通过这种方式，不同的对象可以有不同的状态，尽管它们共享相同的类定义。

举个例子：

```
java 复制代码

public class Car {
    private int speed;
    private int fuelLevel;

    public Car(int speed, int fuelLevel) {
        this.speed = speed;
        this.fuelLevel = fuelLevel;
    }

    public int getSpeed() {
        return speed;
    }

    public int getFuelLevel() {
        return fuelLevel;
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car(50, 10);
        Car car2 = new Car(80, 5);

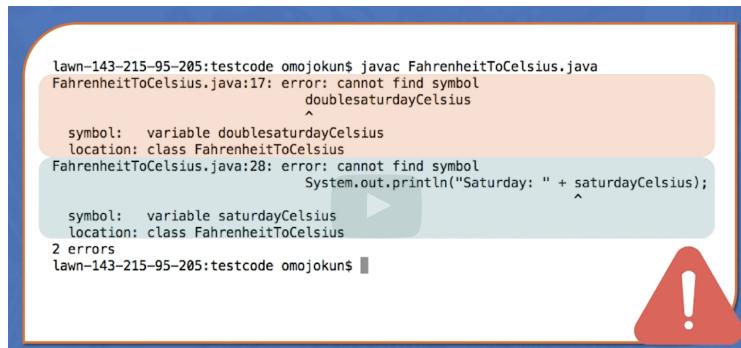
        System.out.println("Car1 - Speed: " + car1.getSpeed() + ", Fuel Level: " +
                           car1.getFuelLevel());
        System.out.println("Car2 - Speed: " + car2.getSpeed() + ", Fuel Level: " +
                           car2.getFuelLevel());
    }
}
```

在上面的例子中，`car1` 和 `car2` 是由同一个 `Car` 类创建的对象，但它们的状态变量（`speed` 和 `fuelLevel`）具有不同的值。因此，状态变量在不同对象中可以有不同的值，即使这些对象来自同一个类。

## Module 3: Terms, Predefined Classes, Input/Output

### 1 报错

#### 1.1 Compile Error



```
lawn-143-215-95-205:testcode omojokun$ javac FahrenheitToCelsius.java
FahrenheitToCelsius.java:17: error: cannot find symbol
    doublesaturdayCelsius
           ^
symbol:   variable doublesaturdayCelsius
location: class FahrenheitToCelsius
FahrenheitToCelsius.java:20: error: cannot find symbol
    System.out.println("Saturday: " + saturdayCelsius);
                           ^
symbol:   variable saturdayCelsius
location: class FahrenheitToCelsius
2 errors
lawn-143-215-95-205:testcode omojokun$
```

因为double saturdayCelsius中间少了个空格，导致无法complier

#### 1.2 Runtime Error (Arithmetic Exception)

syntax都是正确的，但是逻辑无法实现

例子：

- 除零错误：因为除以零在运行时会引发 ArithmeticException

The screenshot shows a Java code editor with the following code:

```
public class DivideByZeroTest {
    public static void main(String[] args) {
        int x = 0;
        System.out.println("Done!");
    }
}
```

Next to it is a terminal window showing the output of the command `javac DivideByZeroTest.java` followed by `java DivideByZeroTest`. The output includes the error message:

```
Exception in thread "main" java.lang.ArithmaticException: / by zero
at DivideByZeroTest.main(DivideByZeroTest.java:3)
```

Below this, there is a code example with a red box around the line `major.substring(3,100);`:

```
String major = "computer science";
major.substring(3,100);
```

A callout diagram illustrates the execution flow:

- A purple box labeled `major` has an orange arrow pointing to a purple box labeled `substring(3,100)`.
- The `substring(3,100)` box is connected to a larger box containing the string `"Computer Science"` followed by a sequence of digits.
- A red exclamation mark icon is positioned above the `substring(3,100)` box.

### 1.3 Logical Error

syntax，数学运算都能实现，但是结果是错的

### 2 Comment lines

- Line comments: //
- Block (or multi-line) comments: /\* \*/
- Javadoc comments: javadoc是自动生成的，把source code里所有comment自动生成在一个html中

### 3 变量的scope

The screenshot shows two snippets of Java code. The left snippet is from `FahrenheitToCelsius.java`:

```
1 public class FahrenheitToCelsius {
2     public static void main(String[] args) {
3         int saturdayFahrenheit;
4         int sundayFahrenheit;
5         saturdayFahrenheit = 78;
6         sundayFahrenheit = 81;
7         double saturdayCelsius = (5.0/9) * (saturdayFahrenheit - 32);
8         double sundayCelsius = (5.0/9) * (sundayFahrenheit - 32);
9         System.out.println("Weekend Averages");
10        System.out.println("Saturday: " + saturdayCelsius);
11        System.out.println("Sunday: " + sundayCelsius);
12    }
13
14    public static void anotherMethod() {
15        double averageFahrenheit = (saturdayFahrenheit + sundayFahrenheit) / 2.0 ;
16    }
17}
```

A callout box points to the line `int saturdayFahrenheit;` with the text "One method cannot see the variables that are declared inside another." and "会报compile error". Another callout at the bottom points to the line `In plain words, one method cannot see the variables that are declared inside another`.

The right snippet is also from `FahrenheitToCelsius.java`:

```
1 public class FahrenheitToCelsius {
2     public static void main(String[] args) {
3         int saturdayFahrenheit;
4         int sundayFahrenheit;
5         saturdayFahrenheit = 78;
6         sundayFahrenheit = 81;
7         double saturdayCelsius = (5.0/9) * (saturdayFahrenheit - 32);
8         double sundayCelsius = (5.0/9) * (sundayFahrenheit - 32);
9         System.out.println("Weekend Averages");
10        System.out.println("Saturday: " + saturdayCelsius);
11        System.out.println("Sunday: " + sundayCelsius);
12    }
13
14    public static void anotherMethod() {
15        int saturdayFahrenheit;
16        saturdayFahrenheit = 100;
17        sundayFahrenheit = 100;
18        double averageFahrenheit = (saturdayFahrenheit + sundayFahrenheit) / 2.0 ;
19    }
20}
```

A callout box points to the line `int saturdayFahrenheit;` with the text "两个methods可以定义完全一样的variable，并且 initialize成不同的数字". Another callout at the bottom points to the line `In another method, and I could give them completely different values.`

在一个method中declare的variable，在另一个method中无法被调用，会出现compile error

### 4 Java中的符号

#### 4.1 \用来表达char中的符号：如，在char中就是 \'

\t: print一格tab

\n: print一格新行

4.2 在Java中，如果计算double someNum = 9/2; 会得到4，小数点后会被截断。

如果想要得到4.5，有两个方法：

- double someNum = 9.0/2;
- double someNum = 9/2D;

To make the above operation of nine divided by two actually return 4.5, we have some options:

- Write out the nine (numerator) and/or two (denominator) as a double literal. So either of these work:

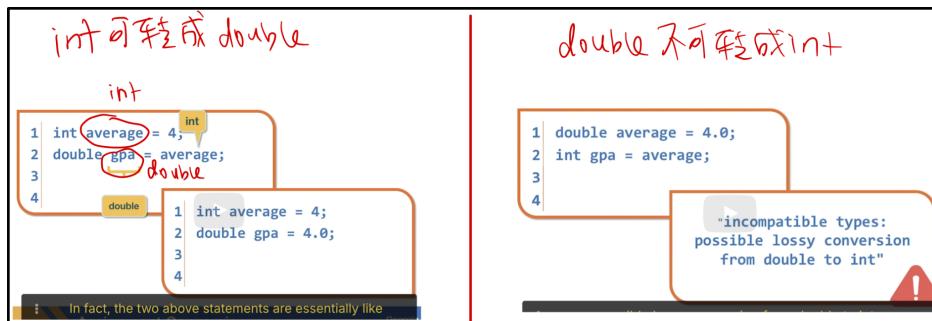
```
9.0 / 2 = 4.5  
9 / 2.0 = 4.5  
9.0 / 2.0 = 4.5
```

The third one above has both as double literals, but only one double is needed to cancel out integer division.

- Override the default typing of the 9 and/or 2 by appending an F or D so that at least one value is treated as a float or double, respectively. To illustrate:

```
9D / 2 = 4.5  
9 / 2D = 4.5  
9D / 2D = 4.5
```

4.3 int可转成double，double不可转成int



4.4 Casting

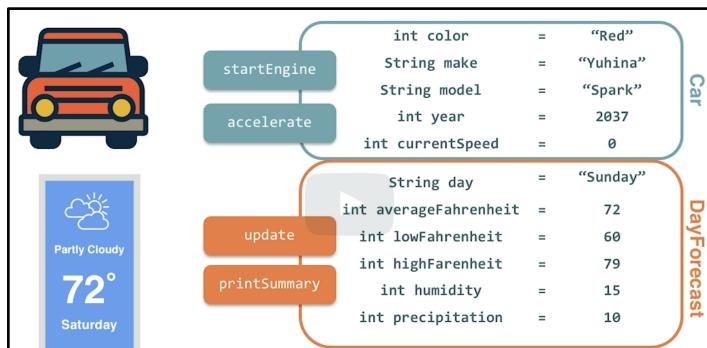
如果想让一个variable的输出是一个特定的type，可以在之前加( )，这叫casting

例子：

- (double) (5/9) → (5/9)=0.556, 所以5/9会得到0 → (double) 5/9 = 0.0
- (double) 5/9 → 5.0/9 → java识别出这是一个double类型的运算 → 结果是0.556

## 5 两个class - Car和DayForecast

### 5.1 Class中的内容



一个class有三个东西：reference variable , constructor , method ( 注意顺序 )

Class的variable以大写开头

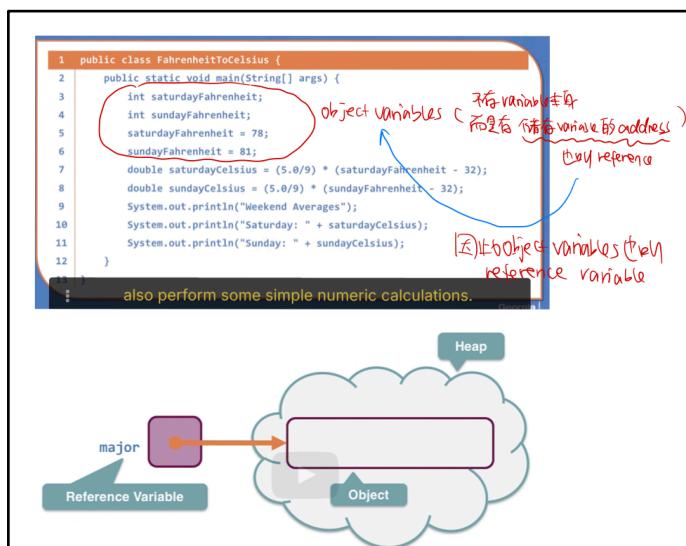
Class : Car

- state variables :
  - color
  - make
  - model
  - year
  - currentSpeed
- Method:
  - startEngine
  - accelerate

Class : DayForecast

- state variables:
  - day
  - averageFahrenheit
  - lowFahrenheit
  - highFahrenheit
  - humidity
  - precipitation
- Method:
  - update
  - printSummary

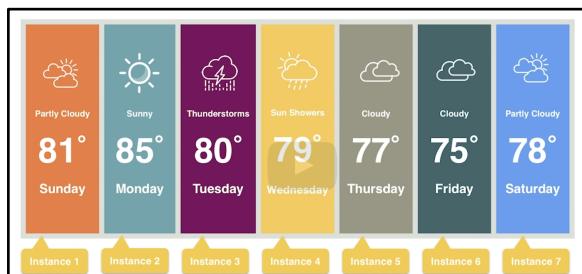
### 5.2 Reference variable



`object variable`(也就是`state variables`)存的是储存在Heap里的variable的address(俗称reference)

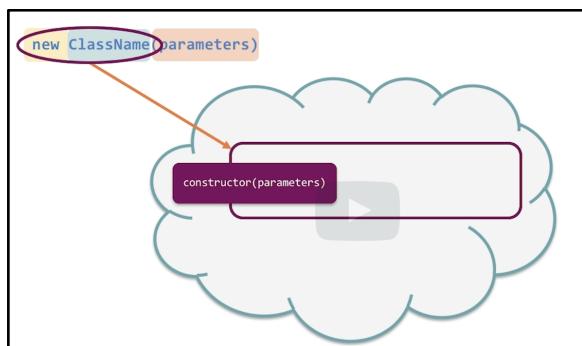
所以`object variable`也叫`reference variable`

### 5.3 instantiation : 创建object的过程



三种instantiation的方法：

这里拿一个内建 class - String 来举例子：

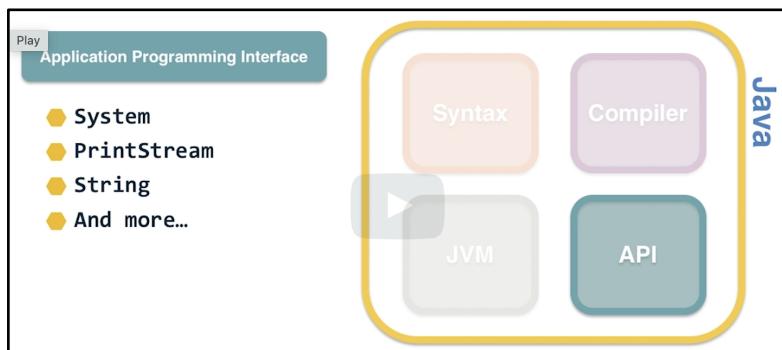


已经有一个叫String的class了

- 方法1: `String major = new String("哈哈") ;`
- 方法2:
  - `String major;`
  - `major = new String("哈哈");`
- 方法3: `String major = "哈哈";`  
(注意，只有内建class可以用方法3，自己customize的class只能方法12)

这里的“哈哈”，是instance的名字，是identifier的一种。

## 6 Java的调用方法



Java由4部分组成

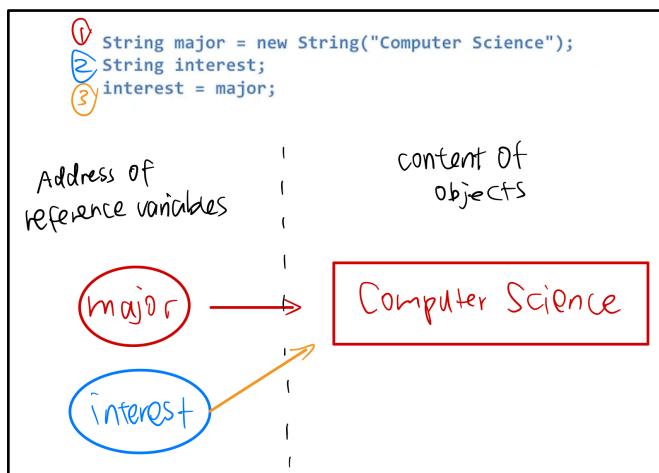
System.out.println("Hello world!");

这里的System和println都是standard library，java内置的library是API的一种

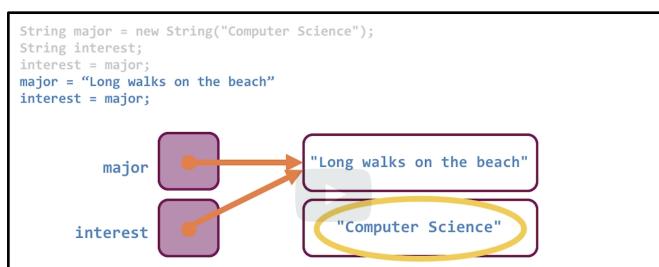
### 6.1 . 叫dot operator

### 6.2 identifier.methodName(); – Java中的方法调用

## 7 Reference variable的储存



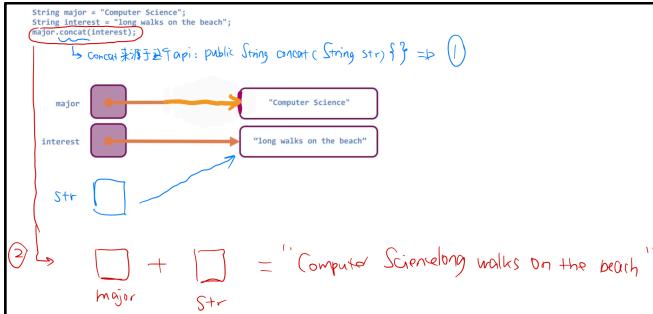
如果两个variables指向同一个object，他们叫aliases，改变任意一个variable，两个将会同时改变



Java的automatic memory management - garbage collection

当“computer science”被创造出来但是没有reference variable的address指向后，java会自动清理掉

## 8 API调用的底层逻辑 - 不改变变量本身



注意：major.concat(interest) 并没有改变major值！

例子：

```

public class StringTwist {
    public static void main(String[] args) {
        String funnyStr = "Computer!Science,!long!walks!on!the!beach";
        funnyStr.replace("!", " ");
        funnyStr.replace("the beach", ""); "Computer!Science, long walks on the beach"
        funnyStr.replace(" ", " "); "Computer!Science, !long!walks!on!the!beach"
        funnyStr.replace("wal", "roc"); "Computer!Science, !long!rocks!on!the!beach"
        funnyStr.toUpperCase(); "COMPUTER!SCIENCE, !LONG!WALKS!ON!THE!BEACH"
        System.out.println(funnyStr);
    }
}

```

COMPUTER SCIENCE ROCKS ON

## 9 Java中位置1用0表示

length is 16

"Computer Science"

012345678.....15

```

String major = "computer science";
major.substring(3, 6);

Return Value "put"
major → substring(3,6) → "Computer Science" 012345678.....15

```

```

String major = "computer science";
major.substring(3, 100);

java.lang.StringIndexOutOfBoundsException: String index out of range: 100 !

```

major → substring(3,100) → "Computer Science" 012345678.....15

## 10 Input and output潜在问题例子

```

public class FahrenheitToCelsius {
    public static void main(String[] args) {
        double fahrenheit = 98.6;
    }
}

```

```
        double celsius = (fahrenheit - 32) * 5 / 9;
        System.out.println(fahrenheit + " degrees Fahrenheit is equal to " + cel-
sius + " degrees Celsius.");
    }
}
```

运行上面的代码时，输出结果可能会显示一个非常长的小数序列。例如：

vbnet

复制代码

```
98.6 degrees Fahrenheit is equal to 37.0000000000001 degrees Celsius.
```

可以在println行中变通：

方法1：

```
String.format("%.1f degrees Fahrenheit is equal to %.1f degrees Celsius.", fahr-
enheit, celsius)
```

方法2：

```
System.out.printf("%.1f degrees Fahrenheit is equal to %.1f degrees Celsius", fa-
renheit, celsius)
```

把上面这一行放入println的括号中即可

String.format()会识别括号中的特殊符号，%.1f 叫占位符，代表把所有type的数字转化为小数点后1位的float，然后以string呈现整句话。

%d - 整数占位符； %.1f - float占位符； %s - string占位符

%,d - 显示带逗号千分位的整数占位符 %,.1f - 显示带逗号千分位的float (eg: 19,222,922.2)

%10s - 10char大小的空间内string右对齐 %-10s - 10char大小的空间内string左对齐

只有要规定小数点后位数的需要加点

运行这两个示例代码时，输出结果如下：

vbnet

复制代码

```
98.6 degrees Fahrenheit is equal to 37.0 degrees Celsius.
```

## 11 静态方法(类中的method) vs 实例方法(类中的method)

- **静态方法**：不依赖于类的实例，可以通过类名直接调用。例如，在你的代码中，`main` 方法是静态的，因此JVM可以直接运行它，而不需要创建 `FahrenheitToCelsius` 类的实例。
  - 一般来说起到帮助或者工具般作用的method都是static的
- **实例方法**：通常，实例方法需要通过类的实例来调用。如果 `main` 方法不是静态的（即没有 `static` 关键字），你将无法直接运行它，而必须先创建该类的对象，然后通过该对象来调用 `main` 方法。

假设我们有一个非静态的方法`convert`，它需要通过类的实例来调用：

java

复制代码

```
public class TemperatureConverter {  
    public double convert(double fahrenheit) {  
        return (fahrenheit - 32) * 5 / 9;  
    }  
  
    public static void main(String[] args) {  
        TemperatureConverter converter = new TemperatureConverter(); // 实例化类  
        double celsius = converter.convert(98.6); // 调用实例方法  
        System.out.println("98.6 degrees Fahrenheit is equal to " + celsius + " degrees Celsius");  
    }  
}
```

## 12 静态方法(类中的method) vs 静态变量(类中的变量)

### 不是一回事

**静态方法**：静态方法是属于类本身的方法，而不是属于类的实例的方法。你可以在**不创建类的实例的情况下直接调用静态方法**。静态方法通常用于不依赖于对象状态的操作。例如，`main` 方法就是一个静态方法，因为它是Java程序的入口点，不需要依赖任何对象状态。

**静态变量**：静态变量（或类变量）是属于整个类的，而不是属于某个对象的实例变量。这意味着**所有类的实例共享同一个静态变量**。如果某个实例修改了静态变量的值，那么该变量的值在所有实例中都会被修改。

### 静态方法 (static method)

静态方法的例子

```
java
public class MathUtil {
    public static int add(int a, int b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        int sum = MathUtil.add(5, 3); // 直接通过类名调用静态方法
        System.out.println("Sum: " + sum);
    }
}
```

在这个例子中，`add` 方法是静态的，所以我们可以直接通过 `MathUtil.add(5, 3)` 调用它，而无需创建 `MathUtil` 类的实例。

(如果line 2没有static，在main中就要先instantiateMathUtil这个类的object)

(同理main函数也是在Main类中运行的，所以是的，main函数可直接且必须在一个class中run)

### 静态变量 (static variable)

静态变量的例子

```
java
public class Counter {
    public static int count = 0;

    public Counter() {
        count++;
    }
}

public class Main {
    public static void main(String[] args) {
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        System.out.println("Count: " + Counter.count); // 输出 2，两个实例共享同一个静态变量
    }
}
```

在这个例子中，`count` 是一个静态变量，所有 `Counter` 类的实例共享这个变量。当创建 `Counter` 的新实例时，`count` 的值会增加，因为所有实例共享同一个 `count` 变量。

修改后的代码和输出结果：  
如果将 `public static int count = 0;` 修改为 `public int count = 0;`，代码将变成如下：

```
java
public class Counter {
    public int count = 0;

    public Counter() {
        count++;
    }
}

public class Main {
    public static void main(String[] args) {
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        System.out.println("c1 Count: " + c1.count); // 输出 1
        System.out.println("c2 Count: " + c2.count); // 输出 1
    }
}
```

( 静态变量 )

( 非静态变量 )

静态变量 - 意思是一个class中的variable是被这个类下的不同objects共享的

(每次生成一个Counter类的object，count++，创建了两个objects，于是Counter.count() = 2)

(但是class中的constructor必须是static的，没有的话variable数字就不共享)

- 若没有static的话，这个变量称为实例变量

## 13 静态方法无法直接访问非静态变量

- 但是在在一个类中，静态method可以通过在这个method中创建这个类的实例，并引用该实例来访问这些变量。

错误示例

```
java
public class MyClass {
    private int instanceVariable = 10; // 非static成员变量

    public static void myStaticMethod() {
        // 试图直接访问非static成员变量，会导致编译错误
        System.out.println(instanceVariable); // 编译错误：无法从静态上下文中引用非静态变量
    }

    public static void main(String[] args) {
        myStaticMethod(); // 调用static方法
    }
}
```

正确示例

```
java
public class MyClass {
    private int instanceVariable = 10; // 非static成员变量

    public static void myStaticMethod() {
        ① // 创建类的实例
        MyClass myObject = new MyClass();
        ② // 通过实例访问非static成员变量
        System.out.println(myObject.instanceVariable);
    }

    public static void main(String[] args) {
        myStaticMethod(); // 调用static方法
    }
}
```

( 无法直接在静态method中访问非静态变量 )      ( 可通过在静态method中创建类的实例来访问非静态变量 )

## 13 private和public的区别

### 13.1 类中的变量

1. 使用 `public` 修饰符

```
java
public class Counter {
    public int count = 0;

    public void increment() {
        count++;
    }
}

public class Main {
    public static void main(String[] args) {
        Counter c1 = new Counter();
        c1.increment();
        System.out.println("c1 Count: " + c1.count); // 输出 1

        // 可以直接修改count变量，因为它是public的
        c1.count = 10;
        System.out.println("c1 Count after manual change: " + c1.count); // 输出 10
    }
}
```

在这个例子中，`count` 变量是 `public` 的，因此可以直接在 `Main` 类中访问和修改 `count` 变量。这种访问权限虽然方便，但可能会导致类的内部状态在外部被随意更改，导致潜在的错误。

2. 使用 `private` 修饰符

```
java
public class Counter {
    private int count = 0;

    public void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}

public class Main {
    public static void main(String[] args) {
        Counter c1 = new Counter();
        c1.increment();
        System.out.println("c1 Count: " + c1.getCount()); // 输出 1

        // c1.count = 10; // 错误！无法直接访问，因为count是private的

        // 必须通过类提供的方法来访问count
        // 如果想要修改count的值，必须提供一个修改的方法
    }
}
```

public:

private:

变量可以被Counter.count()调出来看

也可以从外部c1.count = 10更改

变量不能被Counter.count()调出来，必须使用

method才能调出来看，如果没有method (like getCount() )是查看count variable的，那就完全看不到了是多少 )

- Java的类的method的modifier写的顺序：
  - public static final void xxx(); – 先public再static, final

### 13.2 类中的method

一个类中public的method可以被其他class的method调用

- 甚至在同一个src文件夹中的其他java文件都可以调用此public method

如果是private的method，只能被自己类中的method调用

### 13.3 类

public类每一个java文件只能有一个，所有的public method和public variable存在的前提是它们在一个public的类中。

- 类：public类在一个java文件中只能有一个
- method：public method必须只能存在于public的类中
- variable：public variable必须只能存在于public的类中
- static只能运用于method和variable，类的header中没有static这一说

### 13.4 Constructor 构造函数

构造函数有3种modifiers：public，private，protected



```

1  public class Insect {
2     private double weight;
3     private int age;
4     private int size;
5
6     public Insect(double initweight, int initage, int initsize) {
7         weight = initweight;
8         age = initage;
9         size = initsize;
10    }
11
12    public static void main(String[] args) {
13        Insect buzz1 = new Insect(initweight:10,initage:20,initsize:30);
14        Insect buzz2 = new Insect(initweight:120,initage:40,initsize:10);
15        Insect buzz3 = new Insect(initweight:110,initage:30,initsize:15);
16    }
}

```

**public**：

可以正常创造object，想创建多少个object创建多少个

其他java文件的类也可以使用Insect buzz4 = new Insect( )来创建object

**protected:**

可以正常创造object，想创建多少个object创建多少个

只有本文件内的类，和这个Insect类的子类可以使用Insect buzz4 = new Insect( )来创建object

**默认值：**

如果在构造函数中不写任何modifier，默认只有本文件内的类可以使用Insect buzz4 = new Insect( )来创建object。

```
1 package classTest;
2
3 public class Buzz {
4     private int size;
5     private String name;
6
7     Buzz(double weight, int storage, int instances) {
8         weight = instances;
9         storage = instances;
10        instances = instances;
11    }
12
13    public static void main(String[] args) {
14        Buzz buzz1 = new Buzz(12.5, 12.5, 12.5);
15        Buzz buzz2 = new Buzz(12.5, 12.5, 12.5);
16        Buzz buzz3 = new Buzz(12.5, 12.5, 12.5);
17    }
18}
```

**private:**

private分2种情况：

i) 无法创造任何object，类中通常是static的method，这些static的method被其他的类中的method当作调用。

```
public class MathUtils {
    // 这个构造函数是私有的，阻止外部类创建实例
    private MathUtils() { }

    // 公共静态方法：计算一个数的平方
    public static int square(int number) {
        return number * number;
    }

    // 在另一个java文件中（一个java文件只能有一个public类）：
    public class Main {
        public static void main(String[] args) {
            int square = MathUtils.square(5); // 计算 5 的平方
            System.out.println("Square of 5: " + square); // 输出: Square of 5: 25
        }
    }
}
```

ii) 只能创建唯一一个object。类中通常有1 2 3

```

public class MyClass {
    // 1 私有的静态变量，存储唯一的实例。当有第一个object被创建，就永远被储存在这里。
    private static MyClass instance; ③

    // 2 私有构造函数，防止外部直接实例化
    private MyClass() {
        System.out.println("MyClass instance created!");
    }

    // 3 公共的静态方法，用于获取唯一的实例。这是一个method，return的值的类型是一个类。
    public static MyClass getInstance() {
        if (instance == null) {
            instance = new MyClass(); // 仅在首次调用时创建实例
        }
        return instance;
    }
}

public class Main {
    public static void main(String[] args) {
        // 尝试获取 MyClass 的实例
        MyClass myClass1 = MyClass.getInstance();
        MyClass myClass2 = MyClass.getInstance(); // 第二次调用

        if (myClass1 == myClass2) {
            System.out.println("Both instances are the same.");
        } else {
            System.out.println("Instances are different.");
        }
    }
}

```

① 每次调用 getInstance() 都会返回同一个对象  
② 私有构造函数  
③ 私有静态变量  
④ 之后再通过 getInstance() 调用 object

结果：Both instances are the same.

### 13.5 public private protected 总结

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
none/default	Y	Y	N	N
private	Y	N	N	N

## 14 变量大写含义 - 常数

```
java 复制代码

public class ParkSettings {
    public static final int MIN_PARK_TEMP = 70;
}
```

类中变量大写通常意味着：

- 是常数
- 不可更改
- 一般final和大写变量一同使用
- 使用下划线 \_ 来分隔单词

## 15 类可以没有constructor

可以

constructor也叫构造函数

### 1. 默认构造函数

如果你没有为类定义任何构造函数，Java会自动生成一个无参的默认构造函数。这个默认构造函数不执行任何特殊操作，只是简单地调用父类的无参构造函数（如果有的话）并初始化对象的实例。

例如：

```
java 复制代码

public class MyClass {
    // 没有显式定义构造函数
}

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass(); // 使用默认构造函数
    }
}
```

在这个例子中，`MyClass` 没有显式定义构造函数，但你仍然可以创建 `MyClass` 的实例，这是因为Java自动生成了一个默认构造函数。

### 2. 显式定义的构造函数

如果你显式定义了一个构造函数（无论是否有参数），Java就不会再为你生成默认构造函数。这意味着，如果你定义了一个带参数的构造函数，但没有定义无参构造函数，那么你将无法使用无参构造函数创建对象。

例如：

```
java 复制代码

public class MyClass {
    public MyClass(int value) {
        // 带参数的构造函数
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass(10); // 使用带参数的构造函数
        // MyClass obj2 = new MyClass(); // 错误！因为没有无参
    }
}
```

在这个例子中，`MyClass` 定义了一个带参数的构造函数，但没有定义无参构造函数，因此你不能使用无参构造函数来创建 `MyClass` 的实例。

( 如果不写构造函数，系统会默认生成一个无参数构造函数 ) ( 也可以写构造函数，自己定义想要的参数 )

```
java
```

```
public class MyClass {
    public MyClass() {
        // 无参构造函数
    }

    public MyClass(int value) {
        // 带参数的构造函数
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass obj1 = new MyClass(); // 使用无参构造函数
        MyClass obj2 = new MyClass(10); // 使用带参数的构造函数
    }
}
```

( 既有无参数构造函数, 又有自定义参数构造函数 )

## 16 Java API -Scanner

使用Scanner的第一步是创建一个对象, 如下面的main方法所示:

```
java
```

```
public class FahrenheitToCelsius {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in); // 读取键盘输入
    }
}
```

这个单一的Scanner对象最终将处理我们程序中的所有输入, 在本课中我们将逐步扩展它的功能。我们将这个对象命名为`input`, 以表示我们将用它来做什么 (即读取输入), 但正如你所学到的, 变量名是由程序员决定的。

当继续学习这门语言时, 不要对未来版本中因各种原因添加的新类感到惊讶。Java是一门非常活跃的语言。

```
java
```

```
import java.util.Scanner;

public class ScannerExample {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        int number = input.nextInt(); // 读取第一个令牌并转换为int
        String word1 = input.next(); // 读取第二个令牌并作为字符串返回
        String word2 = input.next(); // 读取第三个令牌并作为字符串返回

        System.out.println("Number: " + number);
        System.out.println("First Word: " + word1);
        System.out.println("Second Word: " + word2);
    }
}
```

输出结果

如果我们输入:

```
arduino
```

```
78 long
walks
```

程序输出将会是:

```
mathematica
```

```
Number: 78
First Word: long
Second Word: walks
```

通过这个示例, 你可以看到如何使用Scanner对象的方法来读取和处理控制台输入的令牌。

### 关于`nextInt()`方法

是的, `nextInt()`方法已经在Java的`Scanner`类库中定义好了。`Scanner`类是Java标准库的一部分, 提供了方便的输入处理方法。`nextInt()`是其中一个方法, 用于从输入流中读取下一个整数。

### `Scanner`类和`nextInt()`方法

`Scanner`类是在`java.util`包中定义的。它包含许多方法来处理不同类型的输入, 如整数、浮点数、字符串等。`nextInt()`方法专门用于读取并返回下一个整数。以下是`Scanner`类中`nextInt()`方法的定义 (简化版):

```
java
```

```
public class Scanner {
    // 构造方法
    public Scanner(InputStream source) {
        // 初始化代码
    }

    // nextInt() 方法
    public int nextInt() {
        // 读取并返回下一个整数的代码
    }

    // 其他方法
}
```

这个输入在Scanner模版的眼中看来是这样的：78 long\nwalks\n

16.1 在一个java文件中写好class1，class2，我可以直接在class2中instantiate一个class1的object，然后在class2中用这个object吗？

- 答案：可以。
- 不同class可以互相调用，但是不可出现循环调用
- 例子：

```
java
// 定义Class1
class Class1 {
    // Class1的私有属性
    private String message;

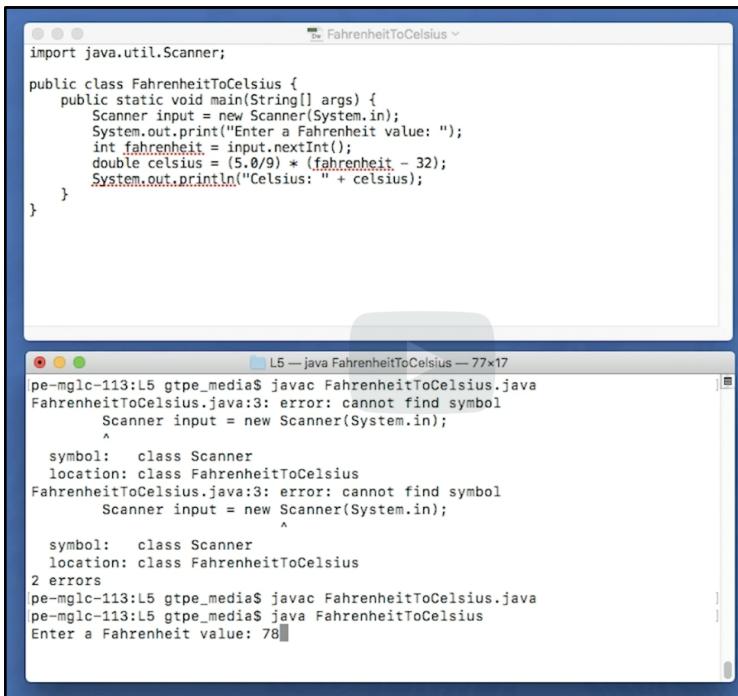
    // Class1的构造方法
    public Class1(String message) {
        this.message = message;
    }

    // Class1的公共方法，用于显示消息
    public void displayMessage() {
        System.out.println("Message from Class1: " + message);
    }
}

// 定义Class2
public class Class2 {
    public static void main(String[] args) {
        // 在Class2中实例化Class1的对象
        Class1 class1Object = new Class1("Hello from Class1!");

        // 调用Class1对象的方法
        class1Object.displayMessage();
    }
}
```

16.2 写好后测试

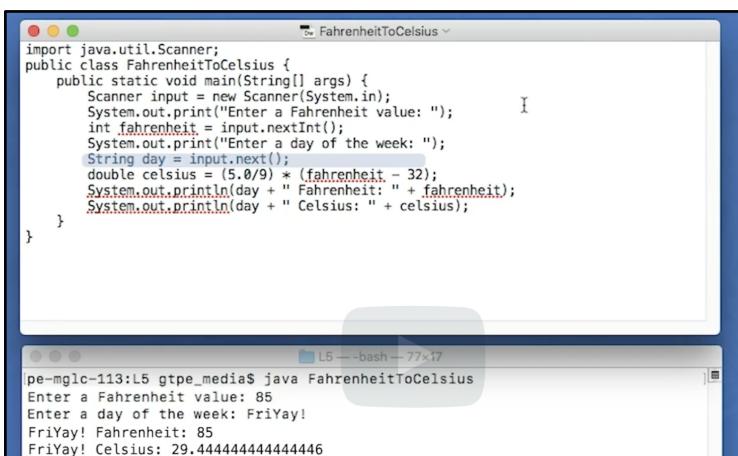


```
import java.util.Scanner;
public class FahrenheitToCelsius {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter a Fahrenheit value: ");
        int fahrenheit = input.nextInt();
        double celsius = (5.0/9) * (fahrenheit - 32);
        System.out.println("Celsius: " + celsius);
    }
}
```

```
pe-mglc-113:L5 gtpe_media$ javac FahrenheitToCelsius.java
FahrenheitToCelsius.java:3: error: cannot find symbol
  Scanner input = new Scanner(System.in);
                     ^
symbol:   class Scanner
location: class FahrenheitToCelsius
FahrenheitToCelsius.java:3: error: cannot find symbol
  Scanner input = new Scanner(System.in);
                     ^
symbol:   class Scanner
location: class FahrenheitToCelsius
2 errors
pe-mglc-113:L5 gtpe_media$ javac FahrenheitToCelsius.java
pe-mglc-113:L5 gtpe_media$ java FahrenheitToCelsius
Enter a Fahrenheit value: 78
```

用之前要先 `import java.util.Scanner;`

在没有input之前，java程序的下一步指令一直是：wait



```
import java.util.Scanner;
public class FahrenheitToCelsius {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter a Fahrenheit value: ");
        int fahrenheit = input.nextInt();
        System.out.print("Enter a day of the week: ");
        String day = input.next();
        double celsius = (5.0/9) * (fahrenheit - 32);
        System.out.println(day + " Fahrenheit: " + fahrenheit);
        System.out.println(day + " Celsius: " + celsius);
    }
}
```

```
pe-mglc-113:L5 gtpe_media$ java FahrenheitToCelsius
Enter a Fahrenheit value: 85
Enter a day of the week: FriYay!
FriYay! Fahrenheit: 85.0
FriYay! Celsius: 29.44444444444446
```

此时Scanner input上的内容是：`85\nFriYay!\n`

## 17 Java的类class

class会根据功能分成多个包，比如

- System就是一个类
- System在java.lang这个包中

除了java.lang中的类外，其他包中的类的调用都需要import

- java.util1.util2.util3.util4.Scanner ; - 其中的点可以划分包的不同层级
- import long.walks.\*; - 代表walks这个包中的所有类都会被引入
- 可能会有两个不同的包共享同一个类名，比如long.walks和short.walks
  - 此时code中写 walks str就会冲突。必须写明short.walks str;

## 18 抽象类 vs 接口类

### 18.1 抽象(类) abstract

#### 抽象类

```
// 抽象类
abstract class Animal {
    // 抽象方法
    abstract void makeSound();
    // 普通方法
    void sleep() {
        System.out.println("Sleeping...");
    }
}
```

#### 普通类

```
// 普通类, 继承自抽象类
class Dog extends Animal {
    // 实现抽象方法
    void makeSound() {
        System.out.println("Bark");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.makeSound(); // 输出: Bark
        dog.sleep(); // 输出: Sleeping...
    }
}
```

普通类使用extends语句使用抽象类中的methods

## 18.2 接口(类) Interface

```
// 动物接口
interface Animal {
    void makeSound();
}

// 狗的实现类
class Dog implements Animal {
    public void makeSound() {
        System.out.println("Bark");
    }
}

// 猫的实现类
class Cat implements Animal {
    public void makeSound() {
        System.out.println("Meow");
    }
}
```

普通类使用`implements`语句使用接口中的methods

## 18.3 比较

抽象类：

- 可以包含constructor，但不能被实例化 (不能写成`String str = new String("哈哈")`这种)，只能被其子类实例化。 ( 注意：普通父类可以实例化 )
- 可以包含method : method里可以有内容，可以被继承或重写。
- 可以包含variable。
- 一个类只能继承一个抽象类 ( java不支持多继承)

接口类：

接口本质也是一个抽象类，是一种更纯粹的抽象类

- 没有constructor

- 接口中的method必须是default是public和non-static的，大部分都不写具体method，而是交给implements这个接口的类去自定义，且method不能既是default又是static的
- 接口中的variable都默认是public, static, final的
- 接口中的静态方法 (static method) 不能在类中被重写
- 一个类可以implements多个接口
- 接口不仅可以被implements，也是可以被extends继承的！



接口中也可以全都是常量，比如用于化学或者物理专业的库：

以下是一个更简单的示例，它存储了一些在化学相关程序中可能有用的常量：

```
java
public interface ChemistryConstants {
    public static final double AVOGADROS_NUMBER = 6.02214199e23; // 阿伏伽德罗常数
    public static final double FARADAY_CONSTANT = 96485.33; // 法拉第常数
    public static final double COULOMB_CONSTANT = 8.987551e9; // 库仑常数
    public static final double PLANCK_CONSTANT = 6.62607004e-34; // 普朗克常数
}
```

但这不是最佳方法，因为这些名字占用了class中可能用到的名字，也违背了interface作为说明书，而不是“包含数字的表格”的用意。更好的办法如下：

```
java
public final class ChemistryConstants {
    private ChemistryConstants() {
        // 私有构造函数防止实例化
    }

    public static final double AVOGADROS_NUMBER = 6.02214199e23;
    public static final double FARADAY_CONSTANT = 96485.33;
    public static final double COULOMB_CONSTANT = 8.987551e9;
    public static final double PLANCK_CONSTANT = 6.62607004e-34;
}
```

通过private class (){} 来限制实例化。同时把这个class设为public。

这样用户可以通过ChemistryCondtants.AVOGADROS\_MUMBER这样来调用，且不存在违背接口本意的一系列问题。

## 19 工厂法创建类 (接口-类-工厂类-main函数 )

- 拿动物举例。
- a 所有动物都会的methods放接口里
- b 每个动物各自特有的methods放各自动物类里
- c 工厂类中只有一个method来创建各个动物object
- d Main函数中用工厂类创建各个动物的object
- e **使用接口作为返回类型**

```

// 定义一个动物接口
public interface Animal {
    void makeSound();
}

// 实现具体的动物类
public class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Bark");
    }
}

public class Cat implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow");
    }
}

// 创建一个工厂类，用于创建动物对象
public class AnimalFactory {
    // 静态工厂方法，根据输入类型返回不同的动物对象
    public static Animal createAnimal(String type) {
        if (type.equalsIgnoreCase("Dog")) {
            return new Dog();
        } else if (type.equalsIgnoreCase("Cat")) {
            return new Cat();
        } else {
            return null;
        }
    }
}

// 测试静态工厂方法
public class Main {
    public static void main(String[] args) {
        Animal dog = AnimalFactory.createAnimal("Dog");
        if (dog != null) {
            dog.makeSound(); // 输出: Bark
        }

        Animal cat = AnimalFactory.createAnimal("Cat");
        if (cat != null) {
            cat.makeSound(); // 输出: Meow
        }
    }
}

```

method 的 result type: Interface

method 不是 static 的  
而是 instantiate  $\Rightarrow$  return 的结果  
type 和

是静态方法  
不带 instantiate

注意，createAnimal这个method本身return的结果是类！

- 比如return new Dog(); 返回的结果的type是一个类！
- 这里只是细化了以下，把类细化成了Interface
- (这些都不在代码上写出来，但是背后的逻辑是这样的)

工厂方法的好处是什么？

- 封装性。AnimalFactory 类封装了对象创建的逻辑，客户端代码通过调用工厂方法 createAnimal 来获得具体的 Animal 对象，而无需了解具体的实现细节使用Class xxx = new Class()来创建。
- 静态性。无需创建工厂类的object即可使用工厂类的method。

- 灵活性。可以随便加新的动物，比如bird

## 20 决策语句 ( 条件语句 )

决策语句 ( decision-making statements ) , 也常被称为条件语句 ( conditional statements ) :

- if
- if-else
- switch

```
if ( ) {
} else {
}
```

三种决策语句

if语句的符号

Which of the following "if" statements will compile? There may be one or more correct answers.

if (cars < 20)
 System.out.println("Parking Available");

if (cars < 20) System.out.println("Parking Available");

if cars < 20:
 System.out.println("Parking Available");

if (cars < 20) {System.out.println("Parking Available");}

if (cars < 20) {
 System.out.println("Parking Available");
}

✓

Relational Operators	
Name	Symbol
less than	<
greater than	>
less than or equal to	<=
greater than or equal to	>=

Equality Operators	
Name	Symbol
equal to	==
not equal to	!=

- 没有 {} 也可以, 注意()后 也没有冒号: - ( 注意 , "==" , "==" )
- {}后没有; 语句后才有 ;

## 21 比较语句

Unicode Value			Character		
48	0	73	I	98	b
49	1	74	J	99	c
50	2	75	K	100	d
51	3	76	L	101	e
52	4	77	M	102	f
53	5	78	N	103	g
54	6	79	O	104	h
55	7	80	P	105	i
56	8	81	Q	106	j
57	9	82	R	107	k
58	:	83	S	108	l
59	:	84	T	109	m
60	<	85	U	110	n
61	=	86	V	111	o
62	>	87	W	112	p
63	?	88	X	113	q
64	@	89	Y	114	r
65	A	90	Z	115	s
66	B	91	[	116	t
67	C	92	\`	117	u
68	D	93	]	118	v
69	E	94	^	119	w
70	F	95	-	120	x
71	G	96	`	121	y
72	H	97	a	122	z

( 字母和数字间也可以比较, 比如 ('Z' < 'a') is true and ('0' < 'A') is also true. 小>大>标>字

- 关系运算符不能用于字符串 , 因此在以下声明中 , 类似于 (x > y) 的表达式是非法的 :

```
java
String x = "park";
String y = "home";
```

- 但是可以使用 String 类的 compareTo 方法来比较两个string的首字母的unicode :

调用字符串对象的字符将与一个字符串参数的字符进行比较。以下是使用上述 x 和 y 的一个示例:

```
java
int result = x.compareTo(y);
```

- result是一个正数8 , 因为park的p unicode是112 , home的h unicode是104。112-104=8

- (x.compareTo(y) > 0) 会return : True

## 22 运算符 Operator (运扩观螺)

### 22.1 运算符号>扩号>关>逻

- **关系运算符 relational operator**  
 $>$   $<$   $==$   $>=$   $<=$   $!=$
- **逻辑运算符 logical operator**
  - **$\&\&$  and**
  - **$\| \mid$  or**
  - **$!$  not**

You are given the following variables.

```
boolean x = true;
boolean y = true;
boolean z = false;
```

Which of the following logical expressions are "true"? There may be one or more correct answers.

$(x \&\& !z)$

$(y \mid z)$

$((z \mid \mid !x) \&\& (y \mid z))$

$(x \mid \mid !x \&\& z)$

$(x \&\& !x \mid \mid z)$

✓

- 关系运算符 **优先级高于** 逻辑运算符
- 逻辑运算符中 : 和 **优先级高于** 或 (特别注意这一条 )
- 括号 **优先级高于** 所有运算符

离散数学内容 :

- if (a) == True, !a就是false
- if (a) == False, !a就是true

a	b	$a \&\& b$	$a \mid \mid b$
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

(a和b的 **和** 与 **或** 比较)

括号的作用 :

使用括号显式地将两个除法操作数组分组，如下所示，虽然不是必须的，但可以帮助提高代码的可读性：

```
java  
if (((saturdayFahrenheit + sundayFahrenheit) / 2) >= MIN_TEMP) {  
    System.out.println("Yay! Nice weekend average.");  
}
```

你还可以通过将平均值计算放在单独的语句中，并将结果存储在一个变量中来简化条件：

```
java  
double weekendAverage = (saturdayFahrenheit + sundayFahrenheit) / 2;  
  
if (weekendAverage >= MIN_TEMP) {  
    System.out.println("Yay! Nice weekend average.");  
}
```

这种方法特别有用，因为如果我们在程序的后续部分需要使用计算出的平均值，由于它已经存储在变量中，可以直接使用。

\*第二种方法比第一种方法好

## 22.2 逻辑运算符短路求值 (Short-circuit Evaluation)

- a && b
- 如果a是false的，java会直接在&&停住，print False，而不再检测b

如何利用短路求值机制？

```
java  
raining || !isValidTemp()
```

- 把复杂的逻辑运算符放在第二个，这样永远先detect简单的一个，如果得到结果就不检测右边的第二个。

24 else-if

## 24.1 嵌套Nesting

- 将一个 if 或 if-else 语句放在另一个 if 或 if-else 语句

## 24.2 else匹配哪个if (Dangling Else悬挂问题)

- else会匹配离的最近的上面的if，而忽略远端上面的if

A screenshot of a Java code editor window titled "DanglingElse.java". The code contains a simple if-else statement:

```
public class DanglingElse {
    public static void main(String[] args) {
        int num = 9;
        if (num > 0)
            if (num < 10)
                System.out.println("aaa");
            else
                System.out.println("bbb");
    }
}
```

The first "if" block and its corresponding "else" block are highlighted with yellow circles, indicating they share the same brace.

### 24.3 三元条件运算符 (Ternary Conditional Operator)



- 第一个操作数是一个条件，因此会产生一个布尔值。
- 如果条件为真，则运算符返回第二个操作数 (expression1) 的计算结果。
- 否则，运算符返回第三个操作数 (expression2) 的结果。

例子：

考虑以下 if-else 语句：

```
java
if (rainInput.startsWith("y")) {
    raining = true;
} else {
    raining = false;
}
```

以下是使用三元运算符的等效单行代码：

```
java
raining = rainInput.startsWith("y") ? true : false;
```

以下是使用三元运算符的解决方案：

```
java
System.out.println(fahrenheit + " " + ((fahrenheit == 1) || (fahrenheit == -1)
    ? "degree"
    : "degrees"));
```

(如果是1度or-1度，则单位显示"degree"，else显示"degrees")

### 24.4 if-else if-else (第一种多路分支 Multi-way Branching )

```
if ( ) { }
else if { }
else { }
```

## 24.5 Switch (第二种多路分支)

The screenshot shows a Java code editor with two examples of switch statements. The left example uses if-else if-else logic:

```
java
switch (expression) {
    case value1:
        statement(s)
        break;
    case value2:
        statement(s)
        break;
    default:
        statement(s)
}
```

The right example uses a switch statement:

```
// 使用 if-else if-else
if (day == 1) {
    System.out.println("Monday");
} else if (day == 2) {
    System.out.println("Tuesday");
} else if (day == 3) {
    System.out.println("Wednesday");
} else {
    System.out.println("Invalid day");
}

// 使用 switch
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    default:
        System.out.println("Invalid day");
}
```

A note at the bottom right says: "在这个例子中, `switch` 语句更容易阅读和理解。"

优点:

- 不用一直写else if , else if , 只用case就好
- 更简洁 , 一看就是一个开关的功能
- switch限制(day)中变量类型更好管理: char byte short int String enum Character Byte Short Integer

特点 :

- condition中只用写变量variable , 不用写变量需要等于多少
- 最后一个default没有break

## 24.6 4种if-else的写法

1. if ( ) { } else {method; }
2. if ( ) method;
3. ( ) || ( ) ? method 1 : method 2
4. switch (variable) {  
    case variable = a:  
        method1;  
        break;  
  
    case variable = b:

```

        method2;
        break;
    default:
        method3;
}

```

## 25 迭代Iteration

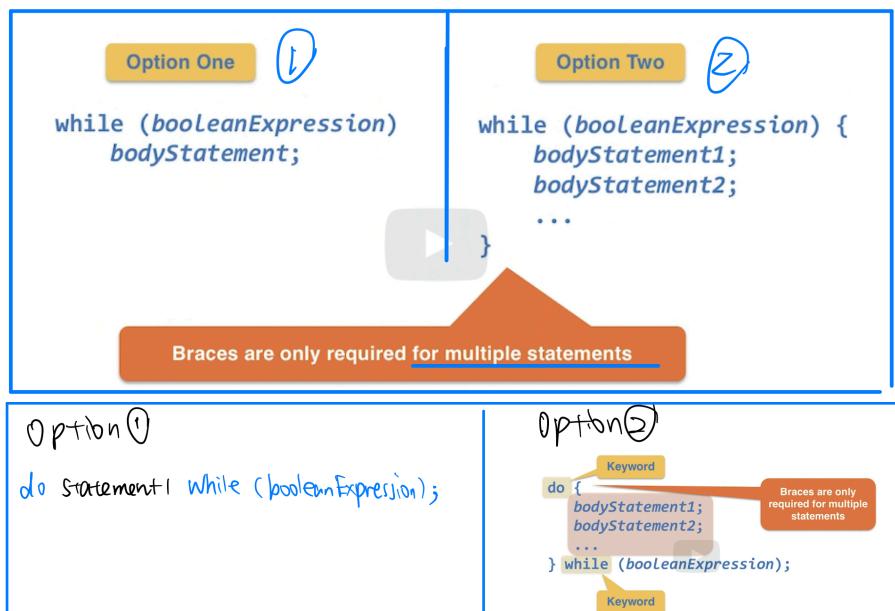
迭代是：

- 优雅地重复执行语句块，同时尽量减少代码的重复
- 它涉及循环执行一段代码，直到某个条件为假。
- 要靠一个counter variable来停止迭代**

三种iteration：

- while 语句
- do-while 语句
- for 语句

### 25.1 while & do-while (4种表达方式)



while和do-while区别：

- do-while** 总是会至少执行一次循环体，而 **while** 则在条件一开始就不满足的情况下可能一次也不会执行。

do-while的while (expression);后还可以跟 statement;

```
int x = 1;
do {
    System.out.print(x + "-");
    x+=3;
}
while (x < 10);
System.out.println("stop!");
```

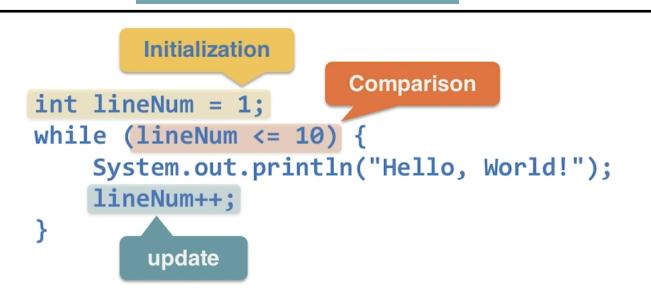
What is the output of the fragment?

stop!  
1-4-7-10-

1-4-7-stop!

while迭代的code setting:

- initialization of a counter variable
- a max value
- an update of the variable in the loop body



例子1：

Java code in the editor:

```
public class HelloWorldLoop {
    public static void main(String[] args) {
        int lineNumber = 1;
        while (lineNumber <= 10) {
            System.out.println("Hello, World! " + lineNumber);
            lineNumber++;
        }
    }
}
```

Terminal output:

```
pe-mglc-113:code gtpe_media$ javac HelloWorldLoop.java
pe-mglc-113:code gtpe_media$ java HelloWorldLoop
Hello, World! 1
Hello, World! 2
Hello, World! 3
Hello, World! 4
Hello, World! 5
Hello, World! 6
Hello, World! 7
Hello, World! 8
Hello, World! 9
Hello, World! 10
pe-mglc-113:code gtpe_media$
```

(while语句 )

Java code in the editor:

```
public class HelloWorldLoop {
    public static void main(String[] args) {
        int lineNumber = 1;
        do {
            System.out.println("Hello, World! " + lineNumber);
            lineNumber++;
        } while (lineNumber <= 10);
    }
}
```

(do-while语句 )

例子2:

```

import java.util.Scanner;
public class FahrenheitToCelsius {
    public static void main(String[] args) {
        final int MAX_TEMP = 140;
        final int MIN_PARK_TEMP = 70;
        final int SWIMMING_POOL = 80;
        Scanner input = new Scanner(System.in);
        System.out.print("Enter a Fahrenheit value: ");
        int fahrenheit = input.nextInt();
        while (fahrenheit >= MAX_TEMP) {
            System.out.println("Error: The Fahrenheit value must be lower than "
                + MAX_TEMP);
            System.out.print("Please enter another Fahrenheit value: ");
            fahrenheit = input.nextInt();
        }
        System.out.print("Enter a day of the week: ");
        String day = input.next();
        double celsius = (5.0 / 9) * (fahrenheit - 32);
        System.out.printf("%s %s", day, " Celsius");
        System.out.printf("%s %.1f %s", " is ", celsius, " Celsius");
        if ((fahrenheit >= 95)) {
            System.out.println("Go to the swimming pool!");
        } else if ((fahrenheit >= 70) && (fahrenheit < 85)) {
            System.out.println("Go to the park!");
        } else if ((fahrenheit >= 50) && (fahrenheit < 70)) {
            System.out.println("Stay at home and eat the pie!");
        } else if ((fahrenheit >= 32) && (fahrenheit < 50)) {
            System.out.println("Make a snowman!");
        }
    }
}

```

- 如何停掉一个无限循环的while语句 : ctrl + C

## 25.2 for语句

for迭代的code setting :

**Keyword** `i = 1 i <= 10 i++`

```

for (initStatement; condition; updateStatement) {
    bodyStatement1;
    bodyStatement2; e.g. System.out.println ("Hello, World!");
    ...
}

```

**Initialization** `int lineNumber = 1;`

**Comparison** `while (lineNum <= 10) {`

**update** `lineNum++;`

**while迭代的code setting:**

- initialization of a **counter variable**
- a **max value**
- an **update of the variable** in the loop body

( for )      ( while )

例子 :

```

public class HelloWorldLoop {
    public static void main(String[] args) {
        int lineNumber;
        for (lineNumber = 1; lineNumber <= 10; lineNumber++) {
            System.out.println("Hello, World! " + lineNumber);
        }
        System.out.println("lineNumber's value after the loop is " + lineNumber);
    }
}

```

```

[pe-mgc-113:code gtpe_media$ javac HelloWorldLoop.java
[pe-mgc-113:code gtpe_media$ java HelloWorldLoop
Hello, World! 1
Hello, World! 2
Hello, World! 3
Hello, World! 4
Hello, World! 5
Hello, World! 6
Hello, World! 7
Hello, World! 8
Hello, World! 9
Hello, World! 10
lineNumber's value after the loop is 11
pe-mgc-113:code gtpe_media$ ]

```

记住在for parenthesis(插入语) 之前要先给lineNum定义type : ini lineNumber ;

也可把type写在for parenthesis(插入语) 中 : for (int lineNumber = 1; lineNumber <=10; lineNumber++)

Nested Loop

```

public class Powers {
    public static void main(String[] args) {
        final int LIMIT = 40;
        for (int base = 2; base <= 5; base++) {
            System.out.println("Powers of " + base + " under " + LIMIT);
            for (int pow = 1; pow <= 40; pow *= base) {
                System.out.println(pow);
            }
            System.out.println();
        }
    }
}

```

pe-mglc-113:code gtpe\_media\$ java Powers

Powers of 2 under 40  
1  
2  
4  
8  
16  
32  
Powers of 3 under 40  
1  
3  
9  
27  
Powers of 4 under 40  
1  
4  
16  
Powers of 5 under 40  
1  
5  
25

pe-mglc-113:code gtpe\_media\$

(一个迭代语句中有另一个迭代语句 - nested loop )

### 25.3 while vs for

使用场景：

- **for语句**通常用于：
  - 循环次数已知的情况下（例如遍历数组、集合）。
  - 需要明确的初始化和迭代控制的情况下。
- **while语句**通常用于：
  - 循环次数不确定，循环直到条件不再满足为止（例如等待某个条件的发生）。
  - 可能完全不需要执行循环体的情况下（条件一开始就是false）。

### 26 break和continue in 迭代

```

import java.util.Scanner;
public class FahrenheitToCelsiusBreak {
    public static void main(String[] args) {
        final int MIN_TEMP = -200;
        final int MAX_TEMP = 200;
        Scanner input = new Scanner(System.in);
        double totalFahrenheit = 0;
        int validFahrenheits = 0;
        int fahrenheit;
        for (int i = 1; i <= 5; i++) {
            System.out.print("Enter a Fahrenheit value: ");
            fahrenheit = input.nextInt();
            if ((fahrenheit >= MIN TEMP) && (fahrenheit <= MAX TEMP)) {
                totalFahrenheit = totalFahrenheit + fahrenheit;
                validFahrenheits++;
            } else {
                System.out.println("Invalid value.");
                break;
            }
        }
        if (validFahrenheits > 0) {
            System.out.println("Average Fahrenheit Temperature: "
                    + totalFahrenheit/validFahrenheits);
        }
    }
}

```

pe-mglc-113:code gtpe\_media\$ javac FahrenheitToCelsiusBreak.java

pe-mglc-113:code gtpe\_media\$ java FahrenheitToCelsiusBreak

Enter a Fahrenheit value: 88  
Enter a Fahrenheit value: 95  
Enter a Fahrenheit value: 100  
Enter a Fahrenheit value: 1331  
① Invalid value.  
Average Fahrenheit Temperature: 94.33333333333333

pe-mglc-113:code gtpe\_media\$

① 不在 MIN ~ max 范围内  
② 选else跳出  
③ else跳出  
④ 从跳出语句中退出  
⑤ break语句帮助从跳出语句中退出来  
(while, for)

### break语句

(注意：if-else不是迭代语句，所以break不是从if-else中break出来)

```

import java.util.Scanner;
public class FahrenheitToCelsiusContinue {
    public static void main(String[] args) {
        final int MIN_TEMP = -200;
        final int MAX_TEMP = 200;
        Scanner input = new Scanner(System.in);
        double totalFahrenheit = 0;
        int validFahrenheits = 0;
        int fahrenheit;
        for (int i = 1; i <= 5; i++) {
            System.out.print("Enter a Fahrenheit value: ");
            fahrenheit = input.nextInt();
            if ((fahrenheit >= MIN_TEMP) && (fahrenheit <= MAX_TEMP)) {
                totalFahrenheit = totalFahrenheit + fahrenheit;
                validFahrenheits++;
            } else {
                System.out.println("Invalid value.");
            }
        }
        if (validFahrenheits > 0) {
            System.out.println("Average Fahrenheit Temperature: " +
                totalFahrenheit / validFahrenheits);
        }
    }
}

```

The terminal window shows the execution of the program. It prompts for five Fahrenheit values: 133, 8, 91, 98, and 89. The invalid value 133 is skipped due to the continue statement. The output shows the average of the remaining valid values: 84.5.

### continue语句

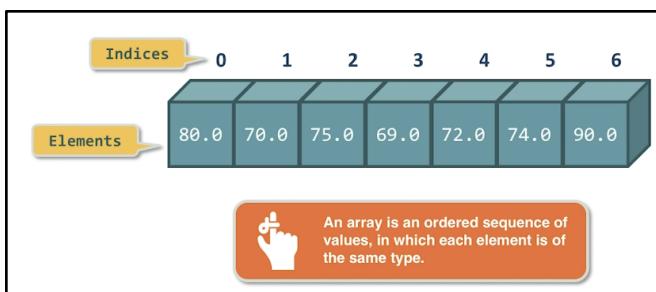
(在迭代语句中继续，即便是输入的值不符合，也不会中断进程，而是忽略该invalid值，继续等待下一个valid值)

- 记住，除了迭代语句可以用break，switch-use-break也是要用到break的。

## Module 4: Arrays, Methods (数组，类的方法)

### 1 数组 arrays

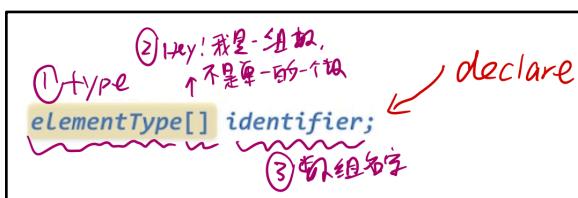
#### 1.1 数组的创建逻辑



Array: an ordered sequence of values

- 每个value有一个index
- 每个value叫element
- 所有value的type都一样

数组declare过程：



以下两种array的declare都可以

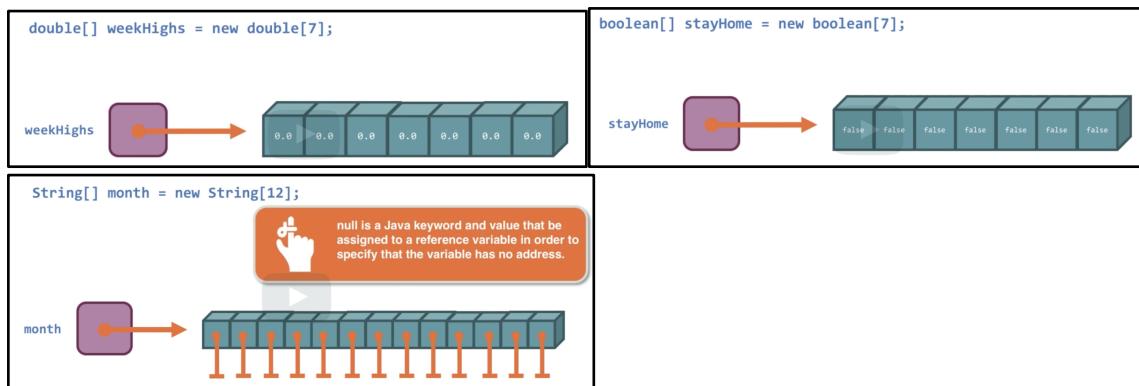
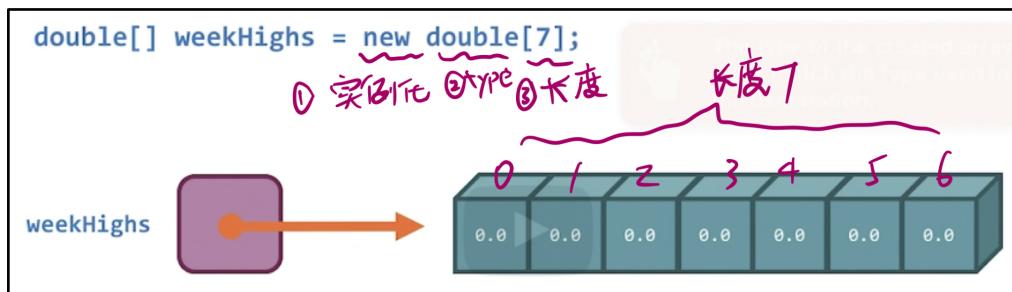
- Type[ ] arrayName;
- Type arrayName[ ];

例子：

step1:



step2:



数组declare好，还没储存数据时，是default状态：

- 如果数组type是numeric的，default状态每个element=0（若是int，则是0，若是double，则是0.00）
- 如果数组type是non-numeric的，default状态每个element=false
- 如果数组type是String的，那这种数组叫**对象数组 (object array)**, default状态每个element=NULL, 意思是相比与前两个type，String的element都还没有给String array的elements安排address

## 1.2 创建数组的2种方式

### 方式1:

- `double[] weekHighs = {80, 70, 75, 69, 72, 74, 90};`

注意：

不可以分两步！

`double[] weekHighs;`  
`weekHighs= {80, 70, 75, 69, 72, 74, 90};`

这是错的

### 方式2:

- `double[] weekHighs = new double[7];`  
`weekHighs[0] = 80;`  
`weekHighs[1] = 70;`  
`weekHighs[2] = 75;`  
`weekHighs[3] = 69;`  
`weekHighs[4] = 72;`  
`weekHighs[5] = 74;`  
`weekHighs[6] = 90;`

如果还没有已知的数据存进数组，可以用迭代的方式将数据存入数组：

- `Scanner input = new Scanner(System.in);`  
`double[] weekHighs = new double[7];`  
`int i = 0;`  
`for (i = 0; i < 7; i++) {`  
`weekHighs[i] = input.nextDouble();`  
}

2 数组 + 迭代 的组合

```
public class Averager {
    public static void main(String[] args) {
        double[] weekHighs = {80, 70, 75, 69, 72, 74, 90};

        double averageHighs = (weekHighs[0] + weekHighs[1] + weekHighs[2]
            + weekHighs[3] + weekHighs[4] + weekHighs[5]
            + weekHighs[6]) / 7;
        System.out.println("Average is: " + averageHighs);
    }
}
```

iteration	1	2	3	4	5	6	7
index at start:	0	1	2	3	4	5	6
weekHighs[index]:	80.0	70.0	75.0	69.0	72.0	74.0	90.0
highSum after body:	80.0	150.0	225.0	294.0	366.0	440.0	530.0

## 2.1 数组+迭代 组合优势

- 当数组中有非常多数的时候，迭代可以有效优化简化代码
- 迭代中无硬数字，如7改成weekHighs.length，如果数组数据有更新，迭代代码不用动

## 2.2 数组+迭代 for-each语句

若有数组 int numbers [ ] = {10, 20, 30, 40, 50, 60, 70};

在对其进行迭代操作时：

```
int i = 0;
for (i = 0; i < numbers.length; i++) { }
```

有时候如果不小心写成i <= numbers.length就容易造成“Index out of bounds errors”，说白了就是如果i=7了，但是numbers没有numbers[7]，就会报错。

- Java为我们准备了一个for-each语法，从而不用担心这个问题：

```
public class ForEachExample {
    public static void main(String[] args) {
        // 创建一个包含7个元素的数组
        int[] numbers = {10, 20, 30, 40, 50, 60, 70}; public class Averager {
            public static void main(String[] args) {
                double[] weekHighs = {80, 70, 75, 69, 72, 74, 90}; // 初始化
                double highSum = 0;
                for (double dayHigh : weekHighs) {
                    highSum = highSum + dayHigh;
                }
                double averageHighs = highSum / weekHighs.length;
                System.out.println(averageHighs);
            }
        }
    }
}
```

for-each语句用例：

( 算平均数 )

( 查询 )

for (int number: numbers) { }；代替 for (i = 0; i < numbers.length; i++) { }；

注意：我们要自己给数组numbers中的每个element取个名，这里就取名为number

## 2.3 数组+迭代 遇到空值问题

那么，如果我们有一个或多个 `null` 值元素的数组，并使用相同的 `for` 语句来搜索它，会发生什么呢？下面是一个这样的程序。请注意，由于 `concepts[1]` 没有显式赋值，因此它默认是 `null`。

```
java                                     复制代码
public class SparseArraySearch {
    public static void main(String args[]) {
        String[] concepts = new String[5];
        concepts[0] = "abstraction";
        concepts[2] = "polymorphism";
        concepts[3] = "inheritance";
        concepts[4] = "encapsulation";
        String result = "not found";
        for (String concept : concepts) {
            if (concept.equals("polymorphism")) {
                result = "found";
                break;
            }
        }
        System.out.println(result);
    }
}
```

( 遇到的问题 )

```
public class SparseArraySearch {
    public static void main(String args[]) {
        String[] concepts = new String[5];
        concepts[0] = "abstraction";
        concepts[2] = "polymorphism";
        concepts[3] = "inheritance";
        concepts[4] = "encapsulation";
        String result = "not found";
        for (String concept : concepts) {
            if ((concept != null) && (concept.equals("polymorphism"))) {
                result = "found";
                break;
            }
        }
        System.out.println(result);
    }
}
```

( 解决问题 )

编译它，不会报错。但是运行它，会报错NullPointerException

why：

- `String[] concepts = new String[5];` 创造的 string array 默认所有 element 是 Null。
- 由于 null 引用不指向实际对象，因此我们不能合法地使用它来调用方法。所以在迭代遍历到 `concepts[1]` 时，由于得不到 `concepts[1]` 的地址，而无法获取 `concepts[1]` 的值，因而抛出了 NullPointerException。此时，程序因错误而终止。

## 2.4 数组+迭代 更改数组长度

如果你想调整数组的大小，你需要创建一个新的数组，并将旧数组的元素复制到新数组中。例如：

```
java                                     复制代码
int[] anArray = new int[5]; // 原始数组，长度为5
int[] newArray = new int[10]; // 新数组，长度为10

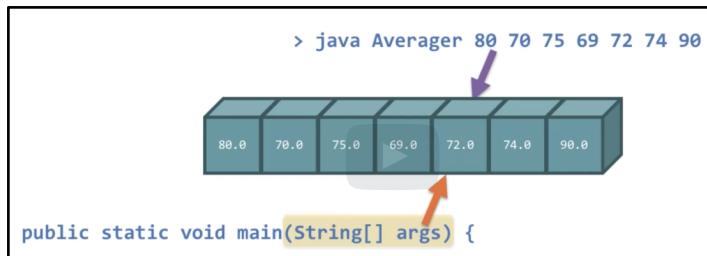
// 复制旧数组的元素到新数组
for (int i = 0; i < anArray.length; i++) {
    newArray[i] = anArray[i];
}

// 使用新数组
anArray = newArray;
```

通过这种方法，数组 `anArray` 的长度从5变成了10。

3 public static void main(String[] args) { };

Java 的 Main 类中的 main 函数中的 reference variables 就是一个叫 args 的 array：



- 这个叫args的array中包含的element需要我们在终端编译的时候填入
- 如 (terminal : java Average 10 20 30) ,
- 此后(String [ ] args)的内容, 即10 20 30, 作为reference variables传入main method 中 )

```

public class Averager {
    public static void main(String[] args) {
        double sum = 0;
        for (String num : args) {
            sum += Double.parseDouble(num); // 将字符串转换为double并累加到sum
        }
        double average = (args.length > 0) ? (sum / args.length) : 0;
        System.out.println("Average is: " + average);
    }
}

```

**示例**

假设在命令行中运行程序时输入以下参数:

bash 复制代码

```
java Averager 10 20 30
```

`args` 数组的内容将是 `{"10", "20", "30"}`。

- `sum` 最终将累加到 60 ( $10 + 20 + 30$ )。
- `average` 将是 60 除以 3, 即 20。
- 最终输出为: `Average is: 20.0`。

如果没有输入参数, 程序会输出 `Average is: 0`。

- 也可以手动把值输入进这个叫args的array , 如上图 , 得到平均值为20

## 4 二元数组 - 矩阵

- 4.1 创建矩阵的两种方式

方法1 :

```

java
double[][] array2d = {
    {90.0, 85.0, 80.0},
    {88.0, 74.0, 92.0},
    {72.0, 68.0, 84.0}
};

这个二维数组可以被视为一个矩阵, 如下所示:
复制代码
90.0 85.0 80.0
88.0 74.0 92.0
72.0 68.0 84.0

```

方法2:

```
int[][] a2d = new int[4][2]
```

(a) How many rows and columns does array a2d have?  
answer with two integers in this format: rows,columns

4,2

✓

(b) How many elements are in the entire array?  
Enter only the integer answer.

8

✓

```

{ { 90.0 85.0 80.0 },
{ 88.0 74.0 92.0 },
{ 72.0 68.0 84.0 }

90.0 85.0 80.0 [0,0] [0,1] [0,2]
88.0 74.0 92.0 [1,0] [1,1] [1,2]
72.0 68.0 84.0 [2,0] [2,1] [2,2]

```

- 4.2 访问一个矩阵 : `array2d[][]`

```

java
public class MatrixExample {
    public static void main(String[] args) {
        double[][] array2d = {
            {90.0, 85.0, 80.0},
            {88.0, 74.0, 92.0},
            {72.0, 68.0, 84.0}
        };

        // 访问92.0并打印
        System.out.println("The value at array2d[1][2] is: " + array2d[1][2]);

        // 访问74.0并打印
        System.out.println("The value at array2d[1][1] is: " + array2d[1][1]);
    }
}

程序输出:
less
复制代码
The value at array2d[1][2] is: 92.0
The value at array2d[1][1] is: 74.0

```

## 5 矩阵遍历方式

```

java
double[][] array2d = {{80, 70, 75}, {69, 72, 74}};

该数组表示如下的矩阵:
复制代码
80 70 75
69 72 74

```

( 矩阵数据 )

## 5.1 行优先遍历 (row-major approach)

代码:

```
java 行数 ①
for (int row = 0; row < array2d.length; row++) {
    for (int col = 0; col < array2d[row].length; col++) { ②小框架 —— 遍历列
        System.out.println("array2d[" + row + "][" + col + "] = " + array2d[row][col]);
    }
} ③第 row 行的长度(即这一行的列数)
```

输出顺序:

```
css
array2d[0][0] = 80
array2d[0][1] = 70
array2d[0][2] = 75
array2d[1][0] = 69
array2d[1][1] = 72
array2d[1][2] = 74
```

80 → 70 → 75  
↓  
69 → 72 → 74

## 5.2 列优先遍历 (column-major approach)

代码:

```
java 从第一行开始来确认矩阵的列数
< ①大框架 —— 先看第一行有多少列, 确认矩阵的列数
for (int col = 0; col < array2d[0].length; col++) {
    for (int row = 0; row < array2d.length; row++) { ②小框架 —— 从列数为0的第1行, 第2行, 再到列数为1的第1行, 第2行, 再到列数为2的第1行, 第2行。
        System.out.println("array2d[" + row + "][" + col + "] = " + array2d[row][col]);
    }
} ③有多少行数
```

输出顺序:

```
css
array2d[0][0] = 80
array2d[1][0] = 69
array2d[0][1] = 70
array2d[1][1] = 72
array2d[0][2] = 75
array2d[1][2] = 74
```

80 ↓ 70 ↓ 75  
69 ↓ 72 ↓ 74

- 注意：矩阵每一行的列数是相同的，100%。

## 6 不规则数组

- 创建不规则数组2种方式

```
double[][] array2d = {{80, 70, 75},  
                      {69, 72, 74, 90}};
```

(方法1 )

① 创建一个行数为2，  
列数不定的不规则矩阵  
② 第一行分配{3个}  
③ 第二行分配{4个}

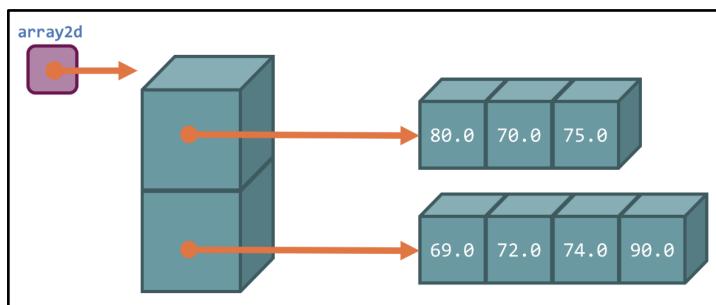
```
double[][] array2d = new double[2][];
// 为第一行分配一个包含3个元素的新数组。
array2d[0] = new double[3];
array2d[0][0] = 80; • }④
array2d[0][1] = 70; • }④
array2d[0][2] = 75; • }④
// 为第二行分配一个包含4个元素的新数组。
array2d[1] = new double[4];
array2d[1][0] = 69; • }⑤
array2d[1][1] = 72; • }⑤
array2d[1][2] = 74; • }⑤
array2d[1][3] = 90; • }⑤
```

( 方法2 )

- 这两种方法创建的数组都是这样的：

```
行 0: [80.0, 70.0, 75.0]
行 1: [69.0, 72.0, 74.0, 90.0]
```

- 背后的逻辑：



## 7 类中的Method (函数)

```
public class ArraySearch {
    public static void main(String args[]) {
        String[] concepts = {"abstraction", "polymorphism",
                             "inheritance", "encapsulation"};
        String result = "not found";
        for (String concept : concepts) {
            if (concept.equals("polymorphism")) {
                result = "found";
                break;
            }
        }
        System.out.println(result);
    }
}
```

原先的code

缺点：不停的重复大量code

```

L9 — bash — 54x24
pe-mglc-113:19 gtpe_media$ javac ArraySearch.java
pe-mglc-113:19 gtpe_media$ java ArraySearch
found
found
pe-mglc-113:19 gtpe_media$ 

public class ArraySearch {
    public static void main(String args[]) {
        String[] concepts = {"abstraction", "polymorphism",
                             "inheritance", "encapsulation"};
        String result = searchStringArray("polymorphism", concepts);
        System.out.println(result);

        result = searchStringArray("inheritance", concepts);
        System.out.println(result);
    }
}

public static String searchStringArray(String target,
                                      String[] array) {
    String result = "not found";
    for (String element : array) {
        if (element.equals(target)) {
            result = "found";
            break;
        }
    }
    return result; //一个method必须有return, 即使是void
}

```

通过method简化后的code

### 结构：

- public static String method名(参数 )
- 这里的参数是一个叫target的string，和一个叫array的数组（名字可随意起）
  - 注意，真正引入参数的时候可以有其他的名字，直接往里面填名字即可
- 只要method有参数，这些参数必须被method中的statement使用
- 一个method必须有return

### 在调用method之前：

- 必须先declare这个method中涉及的参数
  - 这里就是二中的String[ ] concepts和“polymorphism”这个关键词

**优点**：如果这个任务要重复100次，可以用很简单的code完成

## 8 Method的重载 (Method Overloading)

```
java

public class ArraySearch2 {
    public static boolean searchArray(String target, String[] array) {
        boolean result = false;
        for (String element : array) {
            if (element != null && (element.equals(target))) {
                result = true;
                break;
            }
        }
        return result;
    }

    public static boolean searchArray(int target, int[] array) {
        boolean result = false;
        for (int element : array) {
            if (element == target) {
                result = true;
                break;
            }
        }
        return result;
    }
}
```

```
java

public class SomeOtherProgram2 {
    public static void main(String args[]) {
        String[] lullabies = {"Wheels on the Bus", "Twinkle, Twinkle Little Star",
                             "Itsy Bitsy Spider", "Swing Low Sweet Chariot",
                             "Amazing Grace"};
        System.out.println(ArraySearch2.searchArray("Humpty Dumpty", lullabies));

        int[] weekHighs = {80, 70, 75, 69, 72, 74, 90};
        System.out.println(ArraySearch2.searchArray(90, weekHighs));
    }
}
```

ArraySearch2类中：

- 第一个引入参数是string的searchArray method作用于蓝色线条
- 第二个引入参数是int的searchArray method作用于红色线条

method overloading会自动匹配引入参数的数据类型所需要的函数

但是，如果引入参数的数据类型不匹配任何编写好的method，则会报错：

错误  
假设我们在 `SomeOtherProgram2.java` 中输入了以下方法调用：

```
java
ArraySearch2.searchArray(90.0, weekHighs)
```

这里引入参数的数据类型是double (90.0)，已写好的的method只能承接string和int

- 所以这条代码会报错

最常见的method overloading例子 - println

### 使用`println`的重载

事实证明，重载对我们来说并不是一个新概念。在这门课程中，当我们使用`println`方法时，我们已经应用了这个概念，它具有多个版本。比如，有一个版本接受`String`，另一个版本接受`int`，还有其他几个版本接受其他基本类型。因此，以下调用由于重载都成功编译并运行：

```
java 复制代码
System.out.println("90"); // 调用 println(String)
System.out.println(90); // 调用 println(int)
System.out.println(90L); // 调用 println(long)
System.out.println(90.0F); // 调用 println(float)
System.out.println(90.0); // 调用 println(double)
System.out.println('9'); // 调用 println(char)
System.out.println(true); // 调用 println(boolean)
```

这种不是method overloading：

```
java 复制代码
public static int searchArray(int target, int[] array)
public static boolean searchArray(int target, int[] array)
```

method overloading引入参数的类型需要不同

- 而这里的引入参数的类型是一样的，都是int target, int[ ] array
- method overloading的return type也可以不一样
  - 意味着如果这里把第二个的int target, int[ ] array改成string target, string[ ] array，而不改变返回值的类型int和boolean。那这样也算是method overloading

9 foo函数 (foo method)

以下是一个简单的`foo`函数示例：

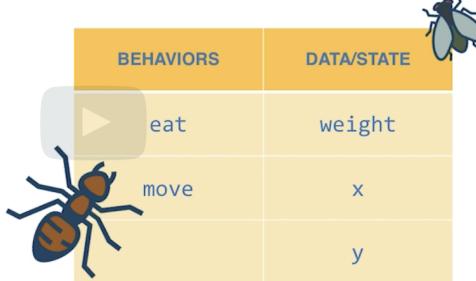
```
java 复制代码
public class Example {
    public static void main(String[] args) {
        foo(); // 调用 foo 函数
    }

    public static void foo() {
        System.out.println("This is the foo function.");
    }
}
```

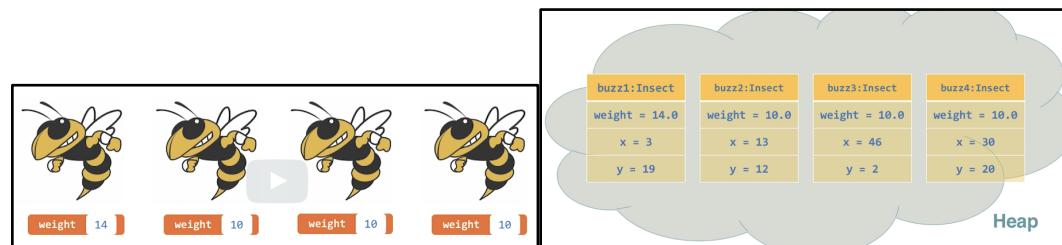
- foo()就是一个占位符，不return任何信息。你也可以定义foo，让foo返回一些文字。

## Module 5 6 7: write code, 继承

### 1 写一个类 (课中教授)

```
public class Insect {  
}  

```

#### 1.1 类中的实例变量 Instance Variables



- 第一个insect吃了一个事物，weight增长4
- 在这个例子中，weight, x, y都是insect object的instance variables
- instance variables保持private，无法被外界调用
- 一共有4个objects

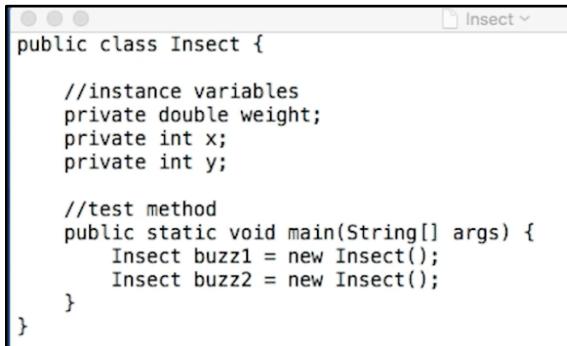
```
java  
  
public class Insect {  
    private double weight;  
    private int x;  
    private int y;  
}
```

#### 1.2 封装性 Encapsulation

通过将instance variable给private了，仅允许类内的方法访问实例变量。这样外界无法对比如weight进行insert.weight = 1;这种更改。

- 这种特性叫**封装性 (Encapsulation)**

### 1.3 类中的main method



```

public class Insect {
    //instance variables
    private double weight;
    private int x;
    private int y;

    //test method
    public static void main(String[] args) {
        Insect buzz1 = new Insect();
        Insect buzz2 = new Insect();
    }
}

```

这里将main method写在Insect类中，是可以的。JVM会自动寻找带有static的main函数开始执行。

- 我脑海里想的另一个方法：把public static void main函数放在一个public class Main中执行是错误的。因为一个java文件只能有一个public的类。所以通常在java中，main method都在这个像Insect这个类中存在，不单独建立。
- 在C++，和Python中，main可以单独在最下面。

### 1.4 类中的constructor 构造函数

每一个class都有一个default constructor.

Instance Variable Type	Default Type
numeric primitive types	0
boolean	false
class (object type)	null

只要创建好一个类的object，这个object的实例变量将被设置为默认值

比如上面的例子：

- weight的类型是double，weight被默认设置为0.0
- x的类型是int，x被默认设置为0
- y的类型是int，y被默认设置为0

Constructor的结构：

- `public class名(实例变量的type和实例变量initial值) {`

```
实例变量1 = 实例变量initial值 ;  
实例变量2 = 实例变量initial值 ;  
实例变量3 = 实例变量initial值 ; }
```

```
1 public class Insect {  
2     private double weight;  
3     private int age;  
4     private int size;  
5  
6     public Insect(double initweight, int initage, int initsize) {  
7         weight = initweight;  
8         age = initage;  
9         size = initsize;  
10    }  
11  
12    Run | Debug | Run main | Debug main  
13    public static void main(String[] args) {  
14        Insect buzz1 = new Insect(initweight:10,initage:20,initsize:30);  
15        Insect buzz2 = new Insect(initweight:120,initage:40,initsize:10);  
16        Insect buzz3 = new Insect(initweight:110,initage:30,initsize:15);  
17    }  
18 }
```

如果不写constructor，系统会自动default一个constructor：

```
1 public class Insect {  
2     private double weight;  
3     private int age;  
4     private int size;  
5  
6     Run | Debug | Run main | Debug main  
7     public static void main(String[] args) {  
8         Insect buzz1 = new Insect();  
9         Insect buzz2 = new Insect();  
10        Insect buzz3 = new Insect();  
11    }  
12 }
```

如果constructor是这种default的状态，那instantiate一个object的时候括号里不用填任何东西。

## 1.5 类中的method

```
1  public class Insect {  
2      private double weight;  
3      private int x;  
4      private int y;  
5  
6      Insect(double initweight, int initx, int inity) { } Constructor  
7          weight = initweight;  
8          x = initx;  
9          y = inity;  
10     }  
11  
12     private static final double DIST_WEIGHT_LOSS_FACTOR = 0.1;  
13  
14     // methods  
15     public void eat(double amount) { } method - eat  
16         System.out.println("Nibble Nibble");  
17         weight = weight + amount;  
18     }  
19  
20     public void move(int newX, int newY) { } method - move  
21         double distance = calculateDistance(x, y, newX, newY);  
22         if (distance > 0) {  
23             x = newX;  
24             y = newY;  
25             weight = weight * DIST_WEIGHT_LOSS_FACTOR * distance;  
26             System.out.printf("Moved %.2f units\n", distance);  
27         } else {  
28             System.out.println("Staying put");  
29         }  
30     }  
31  
32     private static double calculateDistance(double x1, double y1, double x2, double y2) { } 工具  
33         return Math.sqrt((y2 - y1) * (y2 - y1) + (x2 - x1) * (x2 - x1));  
34     }  
35  
36     Run | Debug | Run main | Debug main  
37     public static void main(String[] args) { } main方法  
38         Insect bug1 = new Insect(initweight:10, initx:100, inity:90);  
39         Insect bug2 = new Insect(initweight:9.5, -300, inity:400);  
40  
41         bug1.move(newX:1,newY:10);  
42         bug2.move(-300,newY:400);  
43     }  
44 }
```

(注意，依然在 Insect 类内)

```

1  public class Insect {
2      private double weight;
3      private int x;
4      private int y;      public static 官并未违反封装性
5
6      public static final double DIST_WEIGHT_LOSS_FACTOR = 0.0001;
7      private static int population = 0;    大写的变量代表constant,无法被改
8
9      public Insect(double initweight, int initx, int inity) {
10         weight = initweight;
11         x = initx;
12         y = inity;
13         population = population + 1;
14     }
15
16     // methods
17     public void eat(double amount) {
18         System.out.println("Nibble Nibble");
19         weight = weight + amount;
20     }
21
22     public void move(int newX, int newY) {
23         double distance = calculateDistance(x, y, newX, newY);
24         if (distance > 0) {
25             x = newX;
26             y = newY;
27             weight = weight * DIST_WEIGHT_LOSS_FACTOR * distance;
28             System.out.printf("Moved %.2f units\n", distance);
29         } else {
30             System.out.println("Staying put");
31         }
32     }

```

此处两个更新：

- 第一个是一个知识点，就是variable如果是大写的，自动是不可更改的constant，modifier写成public static**没有破坏封装性**
- 加了一个population的private variable，没创建一个object就加1用于监控有多少个object被create。同样**没有破坏封装性**

## 2 访问器和修改器 (accessors and mutators)

### 2.1 访问器getter

可以在类中加入以下访问器，方便其他class来访问我的类的private数据

```

//methods

public double getWeight() {
    return weight;
}

public int getX() {
    return x;
}

public int getY() {
    return y;
}

```

注意，访问器没有static，因为static的method无法访问非static的variable

- 即如果加上了static，就无法访问weight, x, y

- 访问器**没有破坏封装性**，since它只能return private变量的值，但是无法更改他们的值
- 访问器也叫get method ( 注意method永远都是get+variable的组合)
- 访问器没有parameters，因为他们不引入更新，只是get值而已
- 不需要所有private variable都搞getter，只搞一个private variable的getter也行

The image shows two terminal windows side-by-side. The left window contains the source code for the `Insect` class, which includes a constructor, several getter methods (`getWeight`, `getX`, `getY`, `getPopulation`), a static method `eat`, and a move method. The right window contains the source code for the `InsectClient` class, which creates an `Insect` object and prints its properties. A red box highlights the `getPopulation` method in the `Insect` class and the corresponding line in the `InsectClient` class. Another red box highlights the `getPopulation` line in the `InsectClient` class. Below the windows, the terminal output shows the printed values: `gless`, `13.0`, `31`, `0`, and `1`.

```

//constructor
public Insect(double initWeight, int initX, int initY) {
    weight = initWeight;
    x = initX;
    y = initY;
    population++;
}

//methods
public double getWeight() {
    return weight;
}

public int getX() {
    return x;
}

public int getY() {
    return y;
}

public static int getPopulation() {
    return population;
}

public void eat(double amount) {
    System.out.println("Nibble Nibble");
    weight = weight + amount;
}

public void move(int newX, int newY) {
}

```

```

public class InsectClient {
    public static void main(String args[]) {
        System.out.println(Insect.produceRandomFact());
        Insect bug1 = new Insect(13, 31, 0);
        System.out.println(bug1.getWeight());
        System.out.println(bug1.getX());
        System.out.println(bug1.getY());
        System.out.println(Insect.getPopulation());
    }
}

```

```

gless
13.0
31
0
1
pe-mglc-113:code gtpe_media$ 

```

- 注意，如果要get的variable是static的，getter也要加上static

## 2.2 修改器setter

The image shows a terminal window containing Java code for a `setX` method. The code checks if the new value is legal (greater than or equal to 0) before assigning it to the `x` variable. It also includes other methods like `getY`, `setY`, and two static boolean methods `isLegalX` and `isLegalY`.

```

public void setX(int newX) {
    if (isLegalX(newX)) {
        x = newX;
    }
}

public int getY() {
    return y;
}

public void setY(int newY) {
    if (isLegalY(newY)) {
        y = newY;
    }
}

public static boolean isLegalX(int newX) {
    return (newX >= 0 ? true : false);
}

public static boolean isLegalY(int newY) {
    return (newY >= 0 ? true : false);
}

```

- setter的type必须是void
- setter是有parameter的，因为要改变值，所以要写入值

```

Insect.java
public class Insect {
    //methods
    public double getWeight() {
        return weight;
    }

    public int getX() {
        return x;
    }

    public void setX(int newX) {
        if (isLegalX(newX)) {
            x = newX;
        }
    }

    public int getY() {
        return y;
    }

    public void setY(int newY) {
        if (isLegalY(newY)) {
            y = newY;
        }
    }

    public static boolean isLegalX(int newX) {
        return (newX >= 0 ? true : false);
    }

    public static boolean isLegalY(int newY) {
        return (newY >= 0 ? true : false);
    }
}

InsectClient.java
public class InsectClient {
    public static void main(String args[]) {
        Insect bug1 = new Insect(13, 31, 0);
        System.out.println(bug1.getX());
        bug1.setX(-314);
        System.out.println(bug1.getX());
        bug1.setX(133);
        System.out.println(bug1.getX());
    }
}

Terminal Output:
pe-mglc-113:code gtpe_media$ java InsectClient
31
31
133
pe-mglc-113:code gtpe_media$ 

```

注意：

- setter的话只set x和y的，其他的全部keep internally
- Setter methods should include validation before setting the variable if applicable.

### 3 this语句的用法

#### 3.1 this用于Constructor重构

```

Insect.java
public class Insect {
    //instance variables
    private double weight;
    private int x;
    private int y;

    //static constants/variables
    public static final int DEFAULT_X = 0;
    public static final int DEFAULT_Y = 0;
    public static final double DIST_WEIGHT_LOSS_FACTOR = .0001;
    private static int population = 0;
    private static final String[] FACTS = {
        "The two main groups of insects are winged and wingless",
        "There are more than 1 million insect species",
        "Insects are cold-blooded",
        "Spiders are not considered insects"
    };

    //constructors
    public Insect(double initWeight) {
        weight = initWeight;
        x = DEFAULT_X;
        y = DEFAULT_Y;
        population++;
    }

    public Insect(double initWeight, int initX, int initY) {
        weight = initWeight;
        x = initX;
        y = initY;
        population++;
    }
}

InsectClient.java
public class InsectClient {
    public static void main(String args[]) {
        Insect bug1 = new Insect(13);
        System.out.println(bug1.getWeight());
        System.out.println(bug1.getX());
        bug1.setX(-314);
        System.out.println(bug1.getX());
        bug1.setX(133);
        System.out.println(bug1.getX());

        Insect bug2 = new Insect(31);
        System.out.println(bug2.getWeight());
        System.out.println(bug2.getX());
        System.out.println(bug2.getY());
        System.out.println(Insect.getPopulation());
    }
}

Terminal Output:
pe-mglc-113:code gtpe_media$ java InsectClient
31
31
133
31.0
0
2
pe-mglc-113:code gtpe_media$ 

```

此处写了两个constructors

不同之处：引入的parameters不同，实现了重构

用this语句实现constructor重构，这种模式叫**Constructor Chaining**

```

public class Insect {
    //instance variables
    private double weight;
    private int x;
    private int y;

    //static constants/variables
    public static final int DEFAULT_X = 0;
    public static final int DEFAULT_Y = 0;
    public static final double DIST_WEIGHT_LOSS_FACTOR = .0001;
    private static int population = 0;
    private static final String[] FACTS = {
        "The two main groups of insects are winged and wingless",
        "There are more than 1 million insect species",
        "Insects are cold-blooded",
        "Spiders are not considered insects"
    };

    //constructors
    public Insect(double initWeight) {
        this(initWeight, DEFAULT_X, DEFAULT_Y);
    }

    public Insect(double initWeight, int initX, int initY) {
        weight = initWeight;
        x = initX;
        y = initY;
        population++;
    }

    //methods
    public double getWeight() {
    }
}

```

Handwritten annotations:

- Red bracket above the first constructor:  $\{ \text{weight} = \text{initWeight} \}$
- Red bracket above the second constructor:  $\{ \text{weight} = \text{initWeight}, \text{x} = \text{DEFAULT\_X}, \text{y} = \text{DEFAULT\_Y} \}$
- Blue bracket below the second constructor:  $\{ \text{weight} = \text{initWeight}; \text{x} = \text{initX}; \text{y} = \text{initY}; \text{population}++ \}$

( 输出和上面3中的一摸一样 )

注意：

- constructor chaining要从较不具体的constructor写到比较具体的constructor，如上所示
- 但是真正写code的时候，先把比较具体的constructor写出来，再到前几行去写较不具体的constructor

### 3.2 this用于constructor的reference

this: 用来refer我们initialize的这个object

```

public Insect(double initWeight, int initX, int initY) {
    weight = initWeight;
    x = initX;
    y = initY;
    population++;
}

public Insect(double weight, int x, int y) {
    this.weight = weight;
    this.x = x;
    this.y = y;
    population++;
}

```

( 两边效果等同 )

( 一般用右边效果最佳 )

this可以简化constructor，避免使用一些新的parameter variable name，使你头脑风暴 instead, 把新的reference variable name直接用成reference variable name作为parameter

- 用this.reference\_name作为constructor中这个object的reference

Since `toString()` is inherent to every object,

we should be able to just call it on bug2 without doing anything special to insect.

#### 4 System.out.println(object)的output

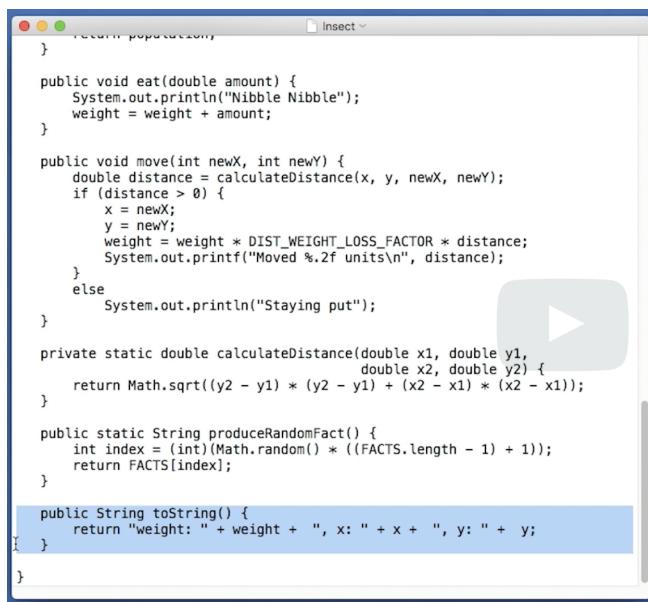
假如还是上面的例子：

- 在Main类的main函数中：

```
Insect bug1 = new Insect(20.0, 2, 1);
System.out.println(bug1);
```

- 这会print出来一段乱码Insect@4b1210ee
  - 这段乱码代表这个object的所有信息的string格式，为什么？
  - 因为System.out.println(bug1);会**自动调用toString() method**，使其变成：
    - System.out.println(bug1.toString());

因此，如果我们在Insect类中主动重构一个to.String() method：



```
return population;
}

public void eat(double amount) {
    System.out.println("Nibble Nibble");
    weight = weight + amount;
}

public void move(int newX, int newY) {
    double distance = calculateDistance(x, y, newX, newY);
    if (distance > 0) {
        x = newX;
        y = newY;
        weight = weight * DIST_WEIGHT_LOSS_FACTOR * distance;
        System.out.printf("Moved %.2f units\n", distance);
    }
    else
        System.out.println("Staying put");
}

private static double calculateDistance(double x1, double y1,
                                       double x2, double y2) {
    return Math.sqrt((y2 - y1) * (y2 - y1) + (x2 - x1) * (x2 - x1));
}

public static String produceRandomFact() {
    int index = (int)(Math.random() * ((FACTS.length - 1) + 1));
    return FACTS[index];
}

public String toString() {
    return "weight: " + weight + ", x: " + x + ", y: " + y;
}
```

那么我们在写System.out.println(bug1);后，可以返回一行优美的关于bug1这个object的信息

- 返回值：weight: 20.0, x: 2, y:1

#### 5 写一个类 (小项目-掷骰子游戏)

一般来说：

- 名词代表类

- 动词代表方法

在视频中，名词“骰子Die”即使类，动词“掷Roll”即是方法。

开始写了。分两步。

**第一步创建骰子的类die。**

**第二步创建一轮游戏的类crap。**

```

import java.util.Random;
public class Die {
    public static final int SIDES = 6;
    private int faceValue;
    private Random rand;
    public Die() {
        faceValue = 1;
        rand = new Random();
    }
    public int roll() {
        faceValue = rand.nextInt(SIDES) + 1;
        return faceValue;
        /* We could also use the Math.random method
         * low + (int)(Math.random() * ((high-low) + 1))
         */
    }
    public int getFaceValue() {
        return faceValue;
    }
    public String toString() {
        return "Die with face value: " + faceValue;
    }
    public static void main(String[] args) {
        Die die1 = new Die();
        System.out.println(die1.toString());
        System.out.println(die1.roll());
        System.out.println(die1.roll());
    }
}

```

(查看random包的api中关于nextInt这个method)

(编译后)

(整个die类的编写方法 )

注意：

- import的时候包首字母大写

```

1 import java.util.Random;
2
3 public class Die {
4
5     public static int SIDES = 6;
6     public int faceValue;
7     public Random rand;
8
9     public Die() {
10         this.faceValue = 1;
11         rand = new Random();
12     }
13
14     public int roll(){
15         this.faceValue = rand.nextInt(SIDES) + 1;
16         return this.faceValue;
17     }
18
19     public String toString(){
20         return "Die with face value: " + faceValue;
21     }
22
23     public static void main(String[] args) {
24         Die die1 = new Die();
25         System.out.println(die1);
26         System.out.println(die1.roll());
27         System.out.println(die1.roll());
28     }
29 }

```

(我写的，output和上面的output一样 )

注意：

toString( )的statement要return，因为java中一定要return一个符合定义类型的值

- 在这里toString的type是String，所以一定要return一个string，而不是print一个string

```
J Insect.java 3 J Craps.java X
InputAndOutput > src > J Craps.java > ...
1  public class Craps {
2
3     private int points;
4     private int first_round_points;
5     private int second_round_points;
6     Die die1;
7     Die die2;
8
9     public Craps(){
10    die1 = new Die();
11    die2 = new Die();
12 }
13
14     public int toss(){
15    die1.roll();
16    die2.roll();
17    points = die1.faceValue + die2.faceValue;
18    return points;
19 }
20
21     public void play(){
22    first_round_points = toss();
23
24    if ((first_round_points == 7) || (first_round_points == 11)) { ①
25        System.out.println("You roll: " + first_round_points);
26        System.out.println("You win!");
27    } else if ((first_round_points == 2) || (first_round_points == 3) || (first_round_points == 12)) {
28        System.out.println("You roll: " + first_round_points);
29        System.out.println("You lost!");
30    } else {
31        System.out.printf("You roll: " + first_round_points + "\nOhh, you enter the 2nd round.\nNow you need to roll % to win, but if you roll 7, you lost.\n", first_round_points);
32        round2();
33    }
34 }
35
36     public void round2(){
37    boolean game_continue = true;
38
39    while (game_continue) {
40
41        second_round_points = toss();
42
43        if (second_round_points == first_round_points){
44            System.out.println("You roll: " + second_round_points);
45            System.out.println("You win!");
46            game_continue = false;
47        } else if (second_round_points == 7) {
48            System.out.println("You roll: " + second_round_points);
49            System.out.println("You lost!");
50            game_continue = false;
51        } else {
52            System.out.println("You roll: " + second_round_points);
53            System.out.println("Game continues for one more round");
54        }
55    }
56 }
57
58 Run | Debug | Run main | Debug main
59 public static void main(String[] args) {
60     Craps craps1 = new Craps(); ② (要在main中创建
61     craps1.play();             不要在play中自动创建对象)
62 }
```

(我写的用于启动游戏的类 - Craps 碰！)

```
[Running] cd "/Users/bangjie/Downloads/GT OMSCS/Java edX/InputAndOutput/src/" && javac Craps.java && java Craps
You roll: 9
Ohh, you enter the 2nd round.
Now you need to roll 9 to win, but if you roll 7, you will lost.
You roll: 2
Game continues for one more round
You roll: 6
Game continues for one more round
You roll: 5
Game continues for one more round
You roll: 11
Game continues for one more round
You roll: 9
You win!

[Done] exited with code=0 in 0.493 seconds
[Running] cd "/Users/bangjie/Downloads/GT OMSCS/Java edX/InputAndOutput/src/" && javac Craps.java && java Craps
You roll: 7
You win!
```

( output)

笔记：

1 void配合System.out.println使用 ( 返回值既可能是string , 又可能是函数 )

String配合return使用 ( 返回值都是文字 )

2 注意 , 这里创建一个crap1的object , 这个crap1中创建了die1 die2 两个 objects

3 if ( a || b || c )中 , 每一个variable都要括号括起来

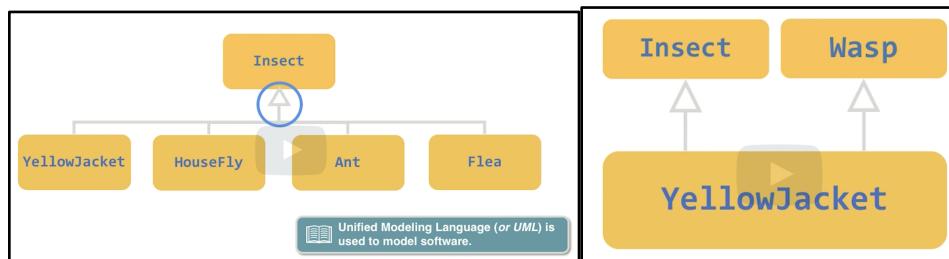
- ( a || b || c )
- ( (a) || (b) || (c) )

4 如果想要在print中用占位符 , 要用printf , 不能用println

5 感觉这么写一个类逻辑更好 :

- 1 先写constructor
- 2 再写主要的method
- 3 再写main
- 过程中有任何variable再创建variable并放在前面的行

6 继承



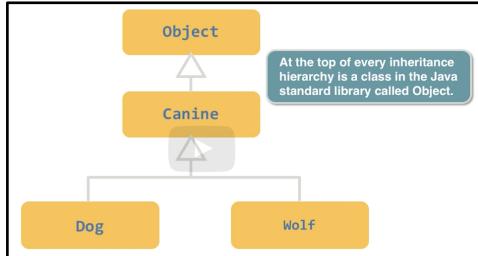
(不被允许, wasp中文是黄蜂)

- 这里的箭头是UML(Unified modeling language)中的一种表示继承的符合。UML是一种软件开发中表示关系的图表标识语言
- Insect是parent class
- 每个child class只能有一个parent class
  - 所以右图的yellowJacket有俩parent classes是不被允许的
- 但是继承可以有一个链 , 像这样是可以被允许的

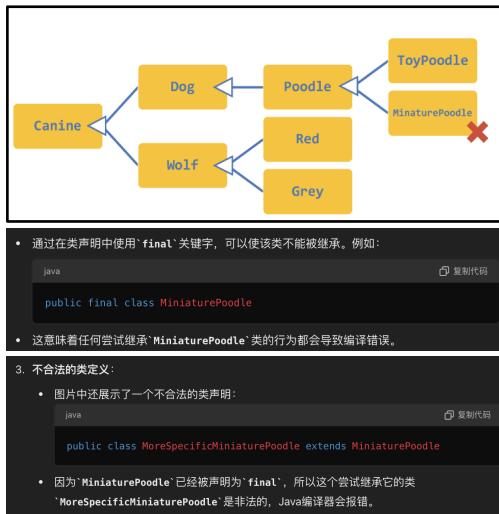


(此例中，YellowJacket也是一种Insect，因为它继承了Insect类)

- 每一个parent class上面还有一个class，叫Object (注意，O是大写的，和instantiate出来的object不是一回事)



- 这里的~~×~~在UML中表示的是：MinaturePoodle类不允许再有子类



## 关于final

- final在variable中表示无法更改，经常和大写变量一起使用，并且因为其反正也不可更改，modifier一般是public的。例如：public final MAX\_NUMBER = 100;
- final在类中表示这个类不能成为任何其他类的父类

继承的例子：

```
public class Canine {
    protected double size;

    public Canine(double size) {
        this.size = size;
    }

    public void bark() {
        System.out.println("Woof Woof");
    }
}
```

(Parent class - 不需要特定modifier)

```
public class Dog extends Canine {
    protected String name;

    public Dog(String name, double size) {
        super(size);
        this.name = name;
    }

    public void fetch() {
        System.out.println("Run");
        System.out.println("Cinch");
        System.out.println("Return");
    }
}
```

(Child class - 加extends表示继承Canine)

```
public class Wolf extends Canine {
    protected int rank;

    public Wolf(double size, int rank) {
        super(size);
        this.rank = rank;
    }

    public int getRank() {
        return rank;
    }

    public void bark() { //3 times the default Canine bark
        for (int i = 1; i <= 3; i++)
            super.bark();
    }

    public static void main(String[] args) {
        Wolf alpha = new Wolf(17.1, 1);
        alpha.bark();
    }
}
```

(Child class)

注意子类继承时，是用：

- **super (父类constructor变量)**
- **super.method( )**

这2个语句来的

i) 父类中constructor有的parameter，子类必须有

- 子类通过super语句继承，并且parameter中也要保持有这个parameter
- 这里的**super语句继承绝对不能动**，比如super(size)不能改为this.size = size;

ii) 父类中没有的method可以自己随便写

iii) 父类中有的method可以自己重构

iv) Wolf子类中bark method的语句中，**调用了父类的bark语句，这种调用方式使用super来实现**。output是：“Woof Woof Woof Woof Woof”

v) 如果父类中的**method改成了final**，此method就禁止重构了，子类必须继承父类的这个method，而不能更改。

```

public class Canine {
    protected double size;

    public Canine(double size) {
        this.size = size;
    }

    public final void bark() {
        System.out.println("Woof Woof");
    }
}

pe-mglc-113:code gtpe_media$ javac Wolf.java
Wolf.java:13: error: bark() in Wolf cannot override bark()
in Canine
    public void bark() { //3 times the default Canine
                                ^
overridden method is final
1 error
pe-mglc-113:code gtpe_media$ 

```

(将bark method改成final的Parent class ) ( Wolf中的重构bark开始报错)

## 7 继承中的抽象类

使用 `final` 表示已是最终定义

使用`abstract` 表示未完全定义的**方法和类**

- abstract method是一个有declaration但是没有definiation的方法
- abstract类中才能有abstract的方法，但同时也可有非abstract的方法
- 一个abstract的父类中的abstract方法，在子类中必须得到定义，否则报错

```

public abstract class Canine {
    protected double size;
    public Canine(double size) {
        this.size = size;
    }
    public void bark() {
        System.out.println("Woof Woof");
    }
    public abstract void groom();
}

public class Dog extends Canine {
    private String name;
    public Dog(String name, double size) {
        super(size);
        this.name = name;
    }

    public void fetch() {
        System.out.println("Wag");
        System.out.println("Climb");
        System.out.println("Return");
    }

    public String toString() {
        return "Name: " + name + "; Size: " + size;
    }

    public void groom() {
        System.out.println("Groom");
    }
}

```

- 一个abstract方法，如`public abstract void groom( );`中，可以在parameter里面加一些variable来限制引入的参数，这样子类也必须引入这些参数，如：
  - `public abstract void groom(int x, int y)`

### 7.1 抽象类和普通继承类的区别？

**普通继承类**：父类正常写，无需特殊字符，子类写`extends`来继承即可。

- 父类中定义好的method，子类可以选择重构，也可以选择不重构

- 普通继承类可以被instantiate成一个object。

**继承中的抽象类**：父类要在public后加一个abstract，子类也通过extends继承，但父类中的method可不定义，而这些未定义method在子类中必须重新写定义好(即必需重构这个method)。

- 如public abstract aMethod( )
- 注意，都不用写后面的{ }
- 然后子类必须在类中把abstract的aMethod定义好了，才行
- 抽象类不可以被instantiate成一个object。

## 8 所有object的类都是Object

所有object的类是Object，也就意味着所有object继承Object类的method

- 比如object继承了Object类的toString( ) method
- 比如.equals( )会根据括号中的内容和object输出的值比较，return true/false

The screenshot shows the Java API documentation for the `equals` method of the `Object` class. The left pane displays the Javadoc for `equals`, which defines it as a method that indicates whether some other object is "equal" to the one. It specifies that the method is reflexive, symmetric, transitive, and consistent with hashCode(). The right pane lists several other methods of the `Object` class:

- `public boolean equals(Object obj)`
- `public String toString()`
- `public final Class getClass()`
- `public int hashCode()`
- `protected Object clone() throws CloneNotSupportedException`
- `public final void notify()`
- `public final void notifyAll()`
- `public final void wait(long timeout) throws InterruptedException`
- `public final void wait(long timeout, int nanoseconds) throws InterruptedException`
- `public final void wait() throws InterruptedException`
- `protected void finalize() throws Throwable`

( Object类的api )

( Object类的api中典型常用的methods )

注意：equals method是有参数的，如果想要override，即重构它，主要就是重构里面的parameters

这意味着什么？

- 这意味着两个不同类的object也是可以互相比较的

```
java 复制代码

Dog dog1 = new Dog("Spot", 9.6);
YellowJacket buzz = new YellowJacket();
dog1.equals(buzz);
```

## 9 多态性 Polymorphism

### 9.1 多态性例子1

上面刚说到的两个不同类的object可以互相比较，这就说多态性的一个例子

### 9.2 多态性例子2

#### 什么是多态性？

##### 示例：动物类和多态性

首先，假设我们有一个 Animal 类， Dog 类和 Cat 类都继承自 Animal 类。

```
java 复制代码

class Animal {
    public void makeSound() {
        System.out.println("Some generic animal sound");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Woof Woof");
    }
}

class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow Meow");
    }
}
```

在上面的例子中， Dog 类和 Cat 类都继承了 Animal 类，并且分别重写了 makeSound() 方法。

##### 多态性的例子

现在，假设我们有一个方法来播放动物的声音：

```
java 复制代码

public class Main {
    public static void playAnimalSound(Animal animal) {
        animal.makeSound();
    }

    public static void main(String[] args) {
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        playAnimalSound(myDog); // 输出 "Woof Woof"
        playAnimalSound(myCat); // 输出 "Meow Meow"
    }
}
```

(一个父类有多个子类)

(static的method中的parameter的type填父类)

- public static void playAnimalSound(Animal animal)这个method的parameter的type实际上可以承接所有子类
  - playAnimalSound (myDog);
  - playAnimalSound (myDog);
  - playAnimalSound (任何时候instantiate的，父类为Animal的object);
  
- 同时，在instantiate object的时候，type是父类，而不是子类：
  - 是Animal myDog = new Dog();
  - 不是Dog myDog = new Dog();

在某个method中的parameter是一个父类时，其parameter可以是任何子类实例化的object，这被称为多态性。

### 9.3 多态性例子3

```
public class GroomEverything {
    public static void main(String[] args) {
        Canine[] groomer = {
            new Wolf(17.01, 3),
            new Poodle("richie", 9, "Lux Brand", "Rich Brand"),
            new Wolf(16, 5),
            new Poodle("pixy", 4, "Top Shelf", "Only the Best"),
        };
        for (Canine c : groomer) { //groom everything
            c.groom();
        }
    }
}
```

在创建一个array时，这个array里存的element的type要相同。多态性使得array的type是一个父类的时候，所有子类都可以被存进去。

- 这里的父类是Canine，子类 ( Wolf, Poodle, Wolf, Poodle) 的object都被存进grommer这个array
- 注意：array不是list！

### 9.4 多态性例子4

#### 9.5 多态性例子5

```
Canine pixy;
pixy = new Poodle(...);
pixy.bark();
```



Canine是Poodle的父类。在实例化的时候，直接这样，也可以正常run，并且区分pixy的type

- pixy的type是Poodle，而Poodle是Canine的子类

“接口”和“实现”分别指的是什么？

### 示例：接口与实现

让我们通过一个具体的例子来说明接口和实现的概念。

首先，我们定义一个名为 Animal 的接口，它规定了 makeSound 方法：

```
java 复制代码
interface Animal {
    void makeSound(); // 接口中方法没有实现
}
```

接下来，我们创建两个类 Dog 和 Cat，它们都实现了 Animal 接口，并提供了 makeSound 方法的具体实现：

```
java 复制代码
class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Woof Woof");
    }
}

class Cat implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow Meow");
    }
}
```

上面的所有 method 都没有定义的类 - Animal，称为接口

- 换言之，接口中所有 method 都是 abstract method

下面的把所有没有定义的 method 都定义好，这叫实现

## 10 接口类 Interface

接口的作用就是：

你不是想创造一个车（名词）的 object 嘛，

```
1  public class Car {  
2  
3 }
```

但是车有哪些动作（动词）呢？我通过 interface 全部给你：

```

interface Groomable {
    void groom();
}

interface Driveable {
    void drive();
}

interface Repairable {
    void repair();
}

```

你只需要去自己的Car类中具体实现这个接口中的抽象method即可：

现在，我们有一个 Car 类，它实现了这三个接口：

```

java                                         ⚡ 复制代码

class Car implements Groomable, Driveable, Repairable {
    @Override
    public void groom() {
        System.out.println("Car is being groomed.");
    }

    @Override
    public void drive() {
        System.out.println("Car is being driven.");
    }

    @Override
    public void repair() {
        System.out.println("Car is being repaired.");
    }
}

```

然后在Main类的main函数中调用时，还可以让一个对象同时是多种接口类的实例：

现在我们可以创建一个 Car 对象，并通过不同的接口引用它：

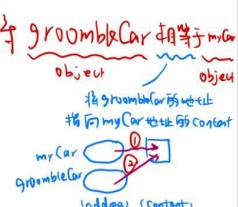
```

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car(); ① Step1: 创建对象 — myCar

        // 通过Groomable接口引用
        Groomable groomableCar = myCar; ② Step2: 将 myCar 这个指针的 object —> groomableCar，并将 groomableCar 等于 myCar
        groomableCar.groom(); // 输出 "Car is being groomed."

        // 通过Driveable接口引用
        Driveable driveableCar = myCar; ③ Step3: 同上
        driveableCar.drive(); // 输出 "Car is being driven." ④ Step4: 同上
        repairableCar.repair(); // 输出 "Car is being repaired."
    }
}

```



此时，myCar, groomableCar, driveableCar, repairableCar这四个name都是同一个object

- myCar.groom()
- groomableCar.groom()

- 以上两个语句实现了一样的功能
- 也实现了一个object同时是多种接口类和他自己类的实例

以上这个例子，是多态性的第四个表现

## 11 接口vs抽象类vs普通继承类

**接口类**更多是给你一个说明书去build一个class。

- 类似根据接口build车的大概功能（开，修，加油），且这些功能必须有

**抽象类**更多的是给你一个父class，以此父class去build一群子class

- 类似你已经build好Car类了，然后以此为父class，build丰田，本田，宝马等一群子classes。子类可以用父类中的method，也可以不用，也可以重构。

**普通继承类**和抽象类是一个概念

- 只不过普通继承类中不能有抽象method，而抽象类中可以有抽象method。
- 另一个是普通继承类的父类可以实例化，但是抽象类不能实例化。

注意：

- 一个没有定义接口中所有methods的抽象类是可以编译的
- interface的header没有class，直接是interface Groomble {}
- **interface接口类**中的method都是abstract的，但是都不需要再在method header写public和abstract。例如：
  - void groom(); 而不是public abstract void groom()
  - interface中的method基本默认都是public的。
- **abstract抽象类**中的method可以是abstract的也可以是非abstract的，但是如果是一个abstract的method，需要在method header中写abstract。例如：

```
public abstract class GroomClass {  
    public void someFunction() { //抽象类中的非抽象method
```

```
    ...
    ...
}

public abstract void groom( );           //抽象类中的抽象method

}
```

- Interface和abstract通常不需要execute，而是只是编译compile。然后被同文件夹中的其他类调用，最后写好后所有类，把类在Main类的main函数中instantiate好，execute Main.java即可。
- 一个abstract类可以再implements无限多的接口类：



The screenshot shows a Java code editor window titled "Canine — Edited". The code defines an abstract class named "Canine" that implements the "Groomable" interface. The class has a protected attribute "size" and two methods: a constructor that initializes "size" and a "bark" method that prints "Woof Woof" to the console.

```
public abstract class Canine implements Groomable {
    protected double size;

    public Canine(double size) {
        this.size = size;
    }

    public void bark() {
        System.out.println("Woof Woof");
    }
}
```

本课程中，你需要掌握：

- 描述接口目的
- 编写接口
- 实现接口

## 12 接口vs协议

### 12.1 接口例子

```
示例：接口（Java）
假设我们有一个接口 PaymentProcessor，它定义了一个处理支付的方法 processPayment。
java
```

```
interface PaymentProcessor {
    void processPayment(double amount);
}

class CreditCardProcessor implements PaymentProcessor {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing credit card payment of $" + amount);
    }
}

class PayPalProcessor implements PaymentProcessor {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing PayPal payment of $" + amount);
    }
}

public class Main {
    public static void main(String[] args) {
        PaymentProcessor processor = new CreditCardProcessor();
        processor.processPayment(100.00);

        processor = new PayPalProcessor();
        processor.processPayment(200.00);
    }
}
```

一般来说，n词是class，v词是method

接口主要就是一个说明书，其他类（名词）按照这个说明书来build method（动词）

协议不仅

- 1 实现了接口的方法，

还定义了通信过程中的：

- 2 数据格式是否正确
- 3 按照一定的交换顺序

## 12.2 协议例子

```
示例：协议（Java）
假设我们要定义一个“协议”，用于处理不同的文件读取器。这个协议不仅要求实现读取方法，还要求
遵循一些额外的规则，例如检查文件的格式是否正确。
java
abstract class FileHeader {
    // 这是一个数据读取的协议类
    public final String readFilePath() {
        if (validateFilePath(filePath)) {
            System.out.println("fileContent: " + content);
        } else {
            System.out.println("Invalid file format.");
        }
    }

    // 子类必须重写此方法
    protected abstract String read(String filePath);

    // 一读到空字符串，就返回空字符串
    protected boolean validateFilePath(String filePath) {
        return filePath != null && filePath.endsWith(getSupportedExtension());
    }

    // 子类必须重写支持的文件扩展名
    protected abstract String getSupportedExtension();
}

class TextFileHeader extends FileHeader {
    @Override
    protected String read(String filePath) {
        return "Text content from " + filePath;
    }

    @Override
    protected String getSupportedExtension() {
        return ".txt";
    }
}

class CSVFileHeader extends FileHeader {
    @Override
    protected String read(String filePath) {
        return "CSV content from " + filePath;
    }

    @Override
    protected String getSupportedExtension() {
        return ".csv";
    }
}

public class Main {
    public static void main(String[] args) {
        FileHeader textHeader = new TextFileHeader();
        textHeader.readfile("example.txt"); // 正常 "file content: Text content from example.txt"

        FileHeader csvHeader = new CSVFileHeader();
        csvHeader.readfile("example.csv"); // 通过 "file content: CSV content from example.csv"

        // 如果设置不支持CSV格式
        textHeader.readfile("example.csv"); // 报错 "Invalid file format."
    }
}
```

总结来说，协议就是一个更加detailed的接口或者更加detailed的抽象类

- 具体在它除了提供一个method的说明书，还规定了整个代码逻辑中涉及到的数据的数据格式和交换顺序
  - 但是数据格式和交换顺序部分的method同样也是可以像普通抽象类一样，是可以交给子类去重构的

说白了：

- 接口只管软件层面的代码
- 协议在管完软件层面的代码后，还validate了数据格式，在数据从应用层传送到网络层之前，按一定顺序排列了数据使其更快更高效的配合协议层

## 13 类型参数 Type Parameters

类型参数是在定义接口和方法时，引入一个或多个类型参数，而无需在定义时确定具体类型。通常使用大写字母表示，例如 T、E、K、V

比如public void (String name);

- 比起上面的，一般在接口中是这样的：public void (T);
  - 表示括号中只能接受一个单一type的variable

## 14 排序算法

一些经过时间考验的排序算法：

归并排序 ( Merge Sort )  
插入排序 ( Insertion Sort )  
冒泡排序 ( Bubble Sort )  
快速排序 ( Quicksort )  
选择排序 ( Selection Sort )

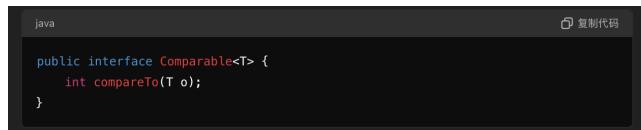
### 14.1 compareTo method

compareTo要实现的功能：比较两个object中的variable的大小：

此处目的不是为了了解一个method内部到底写了什么。

- 相反，是学习在不用去管method中到底写了什么的同时可以正常使用这个method

1 首先，java.lang的api中有这么一个接口：



T代表Type，o代表object。接口中泛型类型的标准写法

<T> 是java api中常用的一种标识符，叫泛型类型参数 ( Type Parameters )

- 表示如果一个类的object用了这个接口中的method，那么这个method引入的参数parameter必须和这个object的类相同。比如：

```
Car car1 = new Car();
car1.compareTo (Car car2);
◦ 上面语句的括号中的object也必须和car1一样
```

- 通常java的class文件中只是public interface Comparable {}，

- 但是如果是遵循接口写出来的java文件，写public interface Comparable<T> {} 也是有的。比如③这里的Comparable<Person>

2 之后，当我想在一个class中实现比较这个动作的时候，因为java.lang有这么一个接口Comparable，提供了一个说明书，我们可以按照这个叫“Comparable”的说明书来build这个class中有关compare的method。先把class写好：

```

1  class Person{
2      private String name;
3      private int age;
4
5      public Person(String name, int age) {
6          this.name = name;
7          this.age = age;
8      }
9
10     @Override
11     public String toString() {
12         return name + ": " + age;
13     }
14 }
```

( 这是一个Person类，现在我想比较两个人的信息 ( age ) )

```

1  class Person implements Comparable<Person> { ①
2      private String name;
3      private int age;
4
5      public Person(String name, int age) {
6          this.name = name;
7          this.age = age;
8      }
9
10     @Override
11     public int compareTo(Person other) { ②按照api正确输入type + other 的格式
12         // 按年龄升序排列
13         return Integer.compare(this.age, other.age); ③调用Integer.compare方法
14     }
15
16     @Override
17     public String toString() { ④Integer也是一个java api中的method, parameter是 (object, object)
18         return name + ": " + age;
19     }
20 }
```

① implements这个接口时，遵循接口中 Comparable<T> 的写法，在 < > 中写上这个类，作为 parameter 的 type。  
 ② Comparable <Person>, 因为类是 Person  
 ③ override 的语句可自定义，只要保证 return type 是 int.  
 ④ Integer 也是一个 java api 中的 method, parameter 是 (object, object)

- object1.compareTo(object2);
- 如果 object1 > object2, return 正数
  - 如果 object1 < object2, return 负数
  - 如果 object1 = object2, return 0

( 3 这是按照“说明书”实现好了的接口，在Person类中的样子 )

```

public class Main {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        // 创建三个 Person 对象
        Person person1 = new Person(name:"Alice", age:30);
        Person person2 = new Person(name:"Bob", age:25);

        // 比较 person1 和 person2
        int result1 = person1.compareTo(person2);
        System.out.println("Comparing " + person1 + " and " + person2 + ":" + result1);
        // 输出: 正数 (因为 30 > 25)
    }
}
```

( 4 在main中execute这个compareTo函数)

compareTo 例子2:

```

Java_Course > src > J Wolf.java > ...
1  public class Wolf extends Canine implements Groomable, Comparable {
2      protected int rank;
3
4      public Wolf(double size, int rank) {
5          super(size);
6          this.rank = rank;
7      }
8
9      public int getRank() {
10         return rank;
11     }
12
13     public void bark() {
14         // 3 times the standard Canine bark
15         for (int i = 1; i <= 3; i++) {
16             super.bark();
17         }
18     }
19
20     public void groom() {
21         System.out.println("lick");
22     }
23
24     public int compareTo(Object anotherWolf) {
25         return -(rank - ((Wolf)anotherWolf).rank);
26     }
}

```

rank并不是啥神乎其神的statement, 只是一个变量而已

① 强制把anotherWolf的type变为Wolf  
 这不是啥神乎其神的命令句, (这里只是为了以防万一, 接口要求compareTo函数输入的值的type必须为Wolf)  
 这就是个质量, 若结果是7  
 则return -7, 仅此而已

## 14.2 quickSort method

quickSort method要实现的功能是将一个array中的不同elements，比如elements是Class Person (name, age)，按照age排序elements。且quickSort函数中要套用compareTo语句。

1 一样，首先，java.lang的api中有这么一个接口：

```

java
public interface Comparable<T> {
    int compareTo(T o);
}

```

```

class Person {
    private String name;
    private int age;

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

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

```

```

class Person implements Comparable<Person> {
    private String name;
    private int age;

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

    @Override
    public int compareTo(Person other) {
        return Integer.compare(this.age, other.age); // 根据年龄排序
    }

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

```

遵循接口的类型参数，  
 此处强制的都是Person类的object  
 ，不是Person

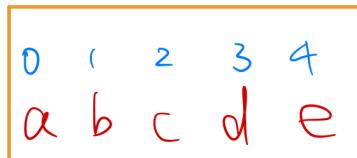
( 2 写好一个Person类 - (name+age) ) ( 3 重构compareTo method , 方便之后在quickSort算法中使用 )

4 在写main函数之前，在Main类中写quickSort函数

```

21 public class Main{
22
23     public static void quickSort(Object[] arr) {
24         // 这里进行排序逻辑
25         for (int i = 0; i < arr.length - 1; i++) {
26             for (int j = i + 1; j < arr.length; j++) { 选择位于"0"的 "a"
27                 if (((Person) arr[i]).compareTo((Person) arr[j]) > 0) { 如果 a 比 b 大
28                     Object temp = arr[i]; 创建一个新类，暂存 "a"
29                     arr[i] = arr[j]; 把位于 "1" 的 "b" 放到 "0" 的位置
30                     arr[j] = temp; 把 "a" 放到位置 "1"，完成从小到大排序
31                 }
32             }
33         }
34     }
35
36     public static void main(String[] args) {
37         void foo();
38     }
39
40 }
```

*选择位于"0"的 "a"*  
*如果 a 比 b 大*  
*创建一个新类，暂存 "a"*  
*把位于 "1" 的 "b" 放到 "0" 的位置*  
*把 "a" 放到位置 "1"，完成从小到大排序*



## 5 完成main函数：

```

38 public static void main(String[] args) {
39     Person[] people = {
40         new Person(name:"Alice", age:30),
41         new Person(name:"Bob", age:25),
42         new Person(name:"Charlie", age:35)
43     };
44
45     quickSort(people);
46
47     for (Person person : people) {
48         System.out.println(person);
49     }
50 }
51 }
```

问题 ⑤ 输出 调试控制台 终端 端口 SQL CONSOLE

[Running] cd "/Users/bangjie/Downloads/GT OMSCS/Java edX/Java\_Course/src/" && javac Main.java && java Main

Bob: 25  
Alice: 30  
Charlie: 35

[Done] exited with code=0 in 0.6 seconds

14.3 Array.sort()

```

1 import java.util.Arrays; //注意，Arrays是哪里 import 的
2
3 public class Wolf extends Canine implements Groomable, Comparable<Wolf> {
4     protected int rank;
5
6     public Wolf(double size, int rank) {
7         super(size);
8         this.rank = rank;
9     }
10
11    public int getRank() {
12        return rank;
13    }
14
15    public void bark() {
16        // 3 times the standard Canine bark
17        for (int i = 1; i <= 3; i++) {
18            super.bark();
19        }
20    }
21
22    public void groom() {
23        System.out.println("lick");
24    }
25
26    public int compareTo(Wolf anotherWolf) {
27        return -(rank - anotherWolf.rank);
28    }
29
30    public String toString() {
31        return ("Rank " + rank + ", Size " + size);
32    }
33
34 Run | Debug
35 public static void main(String[] args) {
36     Wolf[] pack = {
37         new Wolf(size:17.1, rank:2),
38         new Wolf(size:3, rank:10),
39         new Wolf(size:9.2, rank:7),
40         new Wolf(size:9.1, rank:8),
41         new Wolf(size:17.01, rank:3),
42         new Wolf(size:16.2, rank:1),
43         new Wolf(size:16, rank:4),
44         new Wolf(size:16, rank:5),
45         new Wolf(size:10, rank:6),
46         new Wolf(size:5, rank:9)
47     };
48     System.out.println("Unsorted Pack: " + Arrays.toString(pack));
49     Arrays.sort(pack); // ①将这 Tarray 从大到小排序
50     System.out.println("====="); // ②迭代着把这 Tarray 中的
51     System.out.println("Sorted Pack: " + Arrays.toString(pack)); // 每一组数据连成 String
52 }

```

Array.sort( )的运用在现阶段就像黑箱一样。14.3.1 ~ 14.3.3列举了三个Array.sort( )背后的算法：Selection Sort , Merge Sort, and Sort.

#### 14.3.1 Selection Sort

Index					minIndex	Comparisons
0	1	2	3	4		
3	9	6	1	2		◆◆◆◆ PASS 1
1	9	6	3	2		◆◆◆ PASS 2
1	2	6	3	9		◆◆ PASS 3
1	2	3	6	9		◆ PASS 4
1	2	3	6	9		DONE

```

public static void selectionSort (int[] list) {
    int minIndex;
    int nextSmallest;

    for (int unSortedStart = 0; unSortedStart < list.length-1; unSortedStart++) {
        minIndex = unSortedStart;
        for (int currentIndex = unSortedStart+1; currentIndex < list.length; currentIndex++) {
            if (list[currentIndex] < list[minIndex]) {
                minIndex = currentIndex;
            }
        }

        nextSmallest = list[minIndex];
        list[minIndex] = list[unSortedStart];
        list[unSortedStart] = nextSmallest;
    }
}

```

( 没啥特别的，第一次迭代中找到最小的，放在位置0，第二次迭代找到最小的，放在位置1，以此类推 )

算法需要进行  $n-1$  轮遍历，其中  $n$  是列表的大小。

在每一轮遍历中，都会从未排序部分找到最小的元素，并将其放在正确的位置。

### 14.3.2 Merge Sort

归并排序（Merge Sort）是一种经典的排序算法，它通过不断地将列表一分为二，直到列表无法再分割为止。接着，它将这些分割出来的部分按照适当的顺序合并成排序的子列表，直到形成一个完整的有序列表。接下来，我将结合图片中的内容，逐步解释归并排序的过程。

#### 第一步：将列表不断对半分割

归并排序的第一步是将列表分割成两半，然后再将每一半继续分割，直到每个子列表中只剩下一个元素为止。这个过程可以一直进行，直到无法再继续分割。

例如，假设我们有一个列表 [6, 11, 2, 7, 5, 3, 13, 10]，算法将列表不断对半分割：

1. 第一次分割：[6, 11, 2, 7] 和 [5, 3, 13, 10]
2. 第二次分割：[6, 11] 和 [2, 7]，以及 [5, 3] 和 [13, 10]
3. 第三次分割：[6]、[11]、[2]、[7]，以及 [5]、[3]、[13]、[10]

此时，每个子列表中只包含一个元素，因此分割过程结束。

#### 第二步：将单元素子列表合并成有序子列表

现在，我们开始合并这些单元素子列表，形成有序的子列表。合并时会比较两个子列表的第一个元素，并将较小的元素放入新的子列表中，然后继续合并剩下的元素。

1. 合并 [6] 和 [11]：得到 [6, 11]
2. 合并 [2] 和 [7]：得到 [2, 7]
3. 合并 [5] 和 [3]：得到 [3, 5]
4. 合并 [13] 和 [10]：得到 [10, 13]

此时，我们得到了四个有序的子列表 [6, 11]、[2, 7]、[3, 5] 和 [10, 13]。

#### 第三步：继续合并，形成更大的有序子列表

接下来，我们继续合并这些子列表，形成更大的有序子列表：

1. 合并 [6, 11] 和 [2, 7]：得到 [2, 6, 7, 11]
2. 合并 [3, 5] 和 [10, 13]：得到 [3, 5, 10, 13]

此时，我们得到了两个更大的有序子列表 [2, 6, 7, 11] 和 [3, 5, 10, 13]。

#### 第四步：最后的合并，形成完整的有序列表

最后，我们将这两个有序子列表进行合并，得到最终的有序列表：

1. 合并 [2, 6, 7, 11] 和 [3, 5, 10, 13]：得到 [2, 3, 5, 6, 7, 10, 11, 13]

此时，整个列表已经被排序完成。

MergeSort函数，以及其中引用了的merge函数见下面[递归的内容](#)

### 14.3.3 Insertion Sort

好的，让我们通过一个具体的例子来演示插入排序算法如何工作。我们将使用两个列表：一个逆序排列的列表 `[5, 4, 3, 2, 1]` 和一个已排序的列表 `[1, 2, 3, 4, 5]`。

#### 例子1：逆序列表 `[5, 4, 3, 2, 1]`

##### 1. 初始状态：

- 已排序子列表: `[5]`
- 未排序子列表: `[4, 3, 2, 1]`

##### 2. 第一轮迭代：

- 取出 `4`，将其插入到已排序子列表中。
- `4` 小于 `5`，因此 `5` 向右移动一位。
- 插入后：
  - 已排序子列表: `[4, 5]`
  - 未排序子列表: `[3, 2, 1]`

##### 3. 第二轮迭代：

- 取出 `3`，将其插入到已排序子列表中。
- `3` 小于 `5` 和 `4`，因此 `5` 和 `4` 向右移动一位。
- 插入后：
  - 已排序子列表: `[3, 4, 5]`
  - 未排序子列表: `[2, 1]`

##### 4. 第三轮迭代：

- 取出 `2`，将其插入到已排序子列表中。
- `2` 小于 `5`、`4` 和 `3`，因此这三个元素向右移动一位。
- 插入后：
  - 已排序子列表: `[2, 3, 4, 5]`
  - 未排序子列表: `[1]`

##### 5. 第四轮迭代：

- 取出 `1`，将其插入到已排序子列表中。
- `1` 小于 `5`、`4`、`3` 和 `2`，因此这四个元素向右移动一位。
- 插入后：
  - 已排序子列表: `[1, 2, 3, 4, 5]`
  - 未排序子列表: `[]`

最终结果是 `[1, 2, 3, 4, 5]`。

#### 例子2：已排序列表 `[1, 2, 3, 4, 5]`

##### 1. 初始状态：

- 已排序子列表: `[1]`
- 未排序子列表: `[2, 3, 4, 5]`

##### 2. 第一轮迭代：

- 取出 `2`，将其插入到已排序子列表中。
- `2` 大于 `1`，无需移动。
- 插入后：
  - 已排序子列表: `[1, 2]`
  - 未排序子列表: `[3, 4, 5]`

##### 3. 第二轮迭代：

- 取出 `3`，将其插入到已排序子列表中。
- `3` 大于 `2`，无需移动。
- 插入后：
  - 已排序子列表: `[1, 2, 3]`
  - 未排序子列表: `[4, 5]`

##### 4. 第三轮迭代：

- 取出 `4`，将其插入到已排序子列表中。
- `4` 大于 `3`，无需移动。
- 插入后：
  - 已排序子列表: `[1, 2, 3, 4]`
  - 未排序子列表: `[5]`

##### 5. 第四轮迭代：

- 取出 `5`，将其插入到已排序子列表中。
- `5` 大于 `4`，无需移动。
- 插入后：
  - 已排序子列表: `[1, 2, 3, 4, 5]`
  - 未排序子列表: `[]`

最终结果依然是 `[1, 2, 3, 4, 5]`。

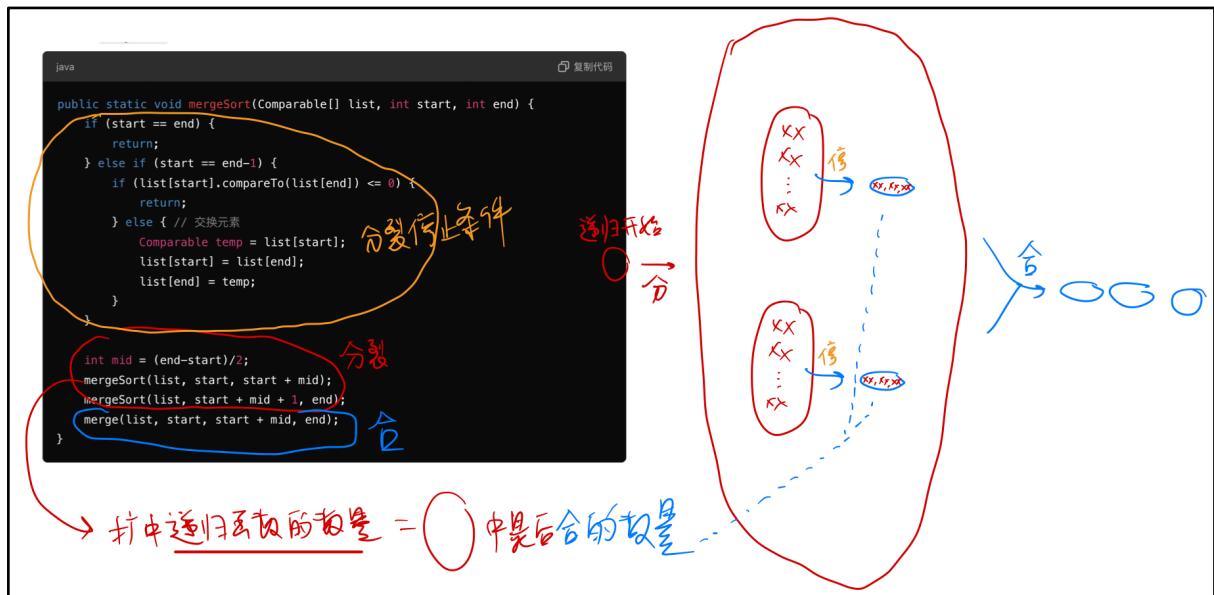
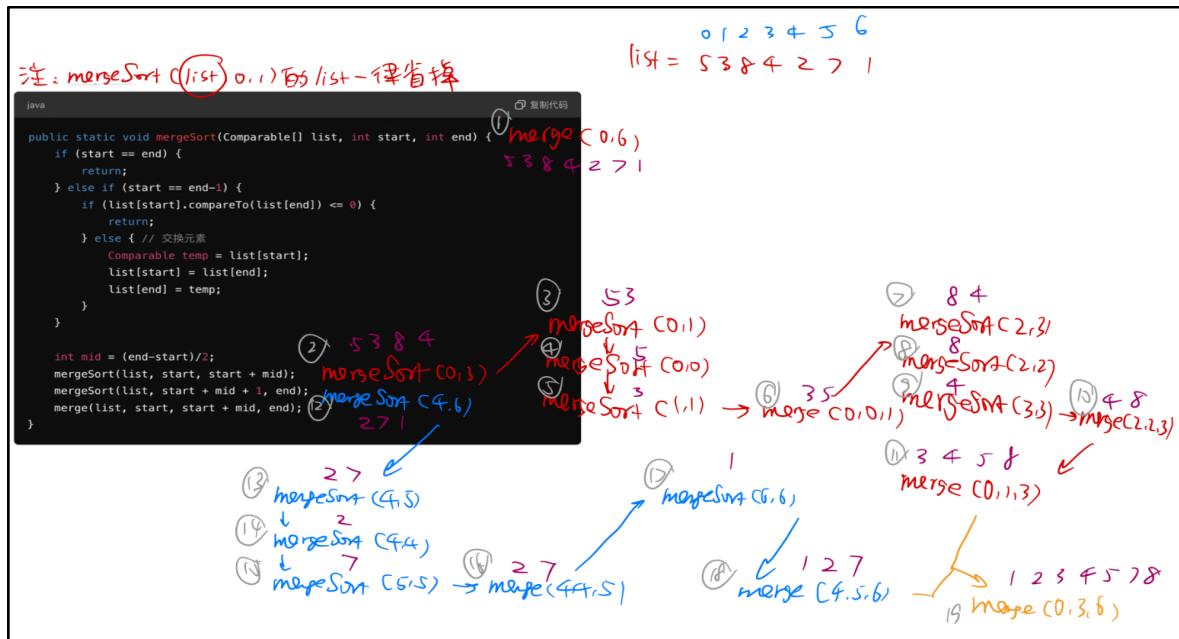
```
public static void insertionSort(int[] list) {  
    // 从第二个元素开始（因为第一个元素可以认为是已经排序的）  
    for (int unsortedStart = 1; unsortedStart < list.length; unsortedStart++) {  
        // 保存当前需要插入的元素  
        int nextInsert = list[unsortedStart];  
        // 从已排序部分的最后一个元素开始比较  
        int currentIndex = unsortedStart - 1;  
  
        // 在已排序部分找到合适的位置插入当前元素  
        while (currentIndex >= 0 && list[currentIndex] > nextInsert) {  
            // 如果当前元素比需要插入的元素大，则将其向右移动一位  
            list[currentIndex + 1] = list[currentIndex];  
            // 继续向前比较  
            currentIndex--;  
        }  
  
        // 找到插入位置，将元素插入  
        list[currentIndex + 1] = nextInsert;  
    }  
}
```

这个InsertSort函数有一个嵌套迭代，for : while :，所以复杂度是O( $n^2$ )

(自己模拟试试吧，我试了确实是的，但是好反人性)

插入排序在处理已排序或几乎已排序的列表时非常高效，但在处理逆序或乱序的列表时表现较差，**时间复杂度为O(n^2)**。

## 15 递归



递归代码人脑无法完全阻止，趁早放弃脑子想的方法

直接理解文字：

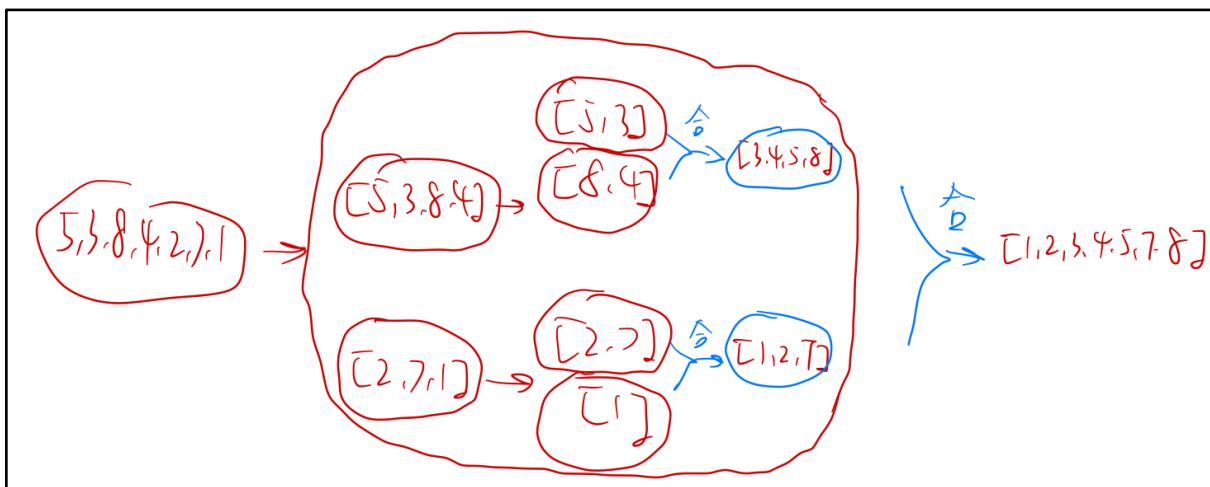
给定一个list = [5,3,8,4,2,7,1]

1 除非list中只有1个数或2个数，其他时候都不停分裂（分裂停止条件）

2 怎么分裂呢？按照中间位置把list劈成左半边和右半边，直到list中只有1个数或2个数，其他时候都不停分裂。（分裂）

3 当第n轮分裂的时候，如果分裂成的2分有一份return停止了，那这2份就merge起来，然后第n轮分裂好merge起来的，再和n-1轮分裂出来的另一个merge，merge到第1轮。递归完成。

- （分裂之后的statement，这些statement特别适合算法中树里面的那种递归运算，因为每一轮分裂在遇到停止条件后会又递归会第n-1，n-2，..., 1一轮去）



merge函数：

java

复制代码

```
private static void merge(Comparable[] list, int leftHalfStart, int rightHalfStart
    int leftHalfSize = rightHalfStart - leftHalfStart + 1;
    int rightHalfSize = end - rightHalfStart;

    Comparable[] leftHalf = new Comparable[leftHalfSize];
    Comparable[] rightHalf = new Comparable[rightHalfSize];

    for (int i=0; i<leftHalfSize; ++i)
        leftHalf[i] = list[leftHalfStart + i];
    for (int j=0; j<rightHalfSize; ++j)
        rightHalf[j] = list[rightHalfStart + 1+ j];
```

创建左边的array  
创建好右边的array

```
int i = 0, j = 0, k = leftHalfStart;
while (i < leftHalfSize && j < rightHalfSize) {
    if (leftHalf[i].compareTo(rightHalf[j]) <= 0) {
        list[k] = leftHalf[i];
        i++;
    } else {
        list[k] = rightHalf[j];
        j++;
    }
    k++;
}

while (i < leftHalfSize) {
    list[k] = leftHalf[i];
    i++;
    k++;
}

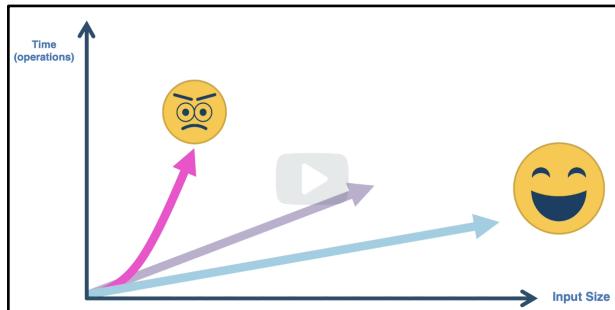
while (j < rightHalfSize) {
    list[k] = rightHalf[j];
    j++;
    k++;
}
```

[3,5,7,10] [4,6,8]

66 66 66

⇒ [3,4,5,6,7,8]  
⇒ [3,4,5,6,7,8,10]

复杂度被定义为算法在输入规模方面的效率（时间或空间需求）。我们特别关注输入增大时时间的表现（而不是空间）。



算法所需时间随数据量增长，不曲线增长的算法都可以说是好算法。

## 16.1 线性时间

- 当一个函数中有一个for迭代，或是一个while迭代时，算法通常是线性增加的。
- 当一个函数中有一个嵌套for迭代，通常是曲线增加的。（`for () { for () {} }`）

```
public static int linearSearch(Comparable target, Comparable[] list) {
    int index = 0;
    while (index < list.length) {
        if (list[index].compareTo(target) == 0)
            return index;
        else
            index++;
    }
    return -1;
}

public static void selectionSort (Comparable[] list) {
    int minIndex;
    Comparable nextSmallest;

    for (int unSortedStart = 0; unSortedStart < list.length-1; unSortedStart++) {
        minIndex = unSortedStart;
        for (int currentIndex = unSortedStart+1; currentIndex < list.length; currentIndex++) {
            if (list[currentIndex].compareTo(list[minIndex]) < 0) {
                minIndex = currentIndex;
            }
        }
        nextSmallest = list[minIndex];
        list[minIndex] = list[unSortedStart];
        list[unSortedStart] = nextSmallest;
    }
}
```

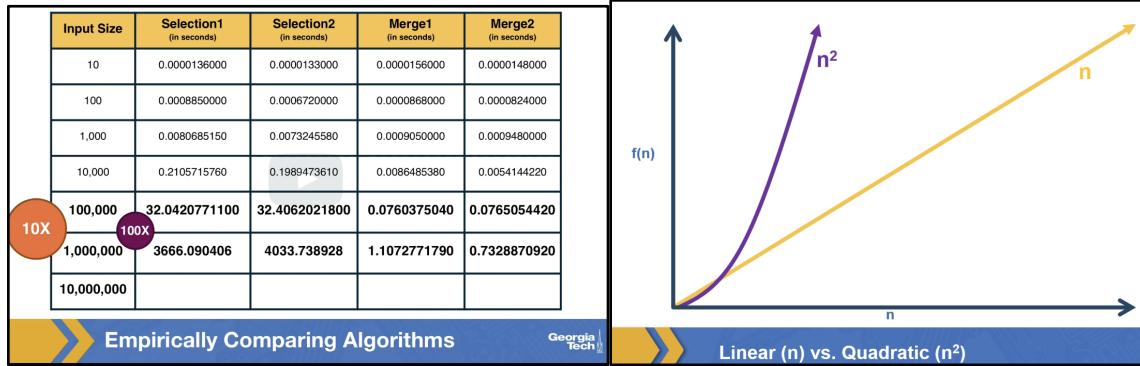
线性增长

曲线增长，涉及到一个array中2个数之间的比较

注意：if-else不是迭代，只是条件运算符

## 16.2 曲线增长时间

SelectionSort vs MergeSort算法：



随着数据量的增加：

- SelectionSort的速度是曲线增加 (quadratic) 的，不好
- MergeSort的速度增长慢得多，好

例子：排序一个小数组  
假设我们有一个小的数据组要排序，比如 [5, 3, 8, 1]。

1. 第一次遍历：
  - 我们从整个数组中找到最小的数。在 [5, 3, 8, 1] 中，1 是最小的，所以我们把 1 和第一个元素 5 交换位置。
  - 数组变为 [1, 3, 8, 5]。
2. 第二次遍历：
  - 现在我们忽略第一个元素（因为它已经在正确位置），在剩下的 [3, 8, 5] 中找最小的数。
  - 3 是最小的，它已经在正确位置，不需要交换。
  - 数组保持为 [1, 3, 8, 5]。
3. 第三次遍历：
  - 忽略前两个元素，在 [8, 5] 中找最小的数。
  - 5 是最小的，所以我们交换 5 和 8。
  - 数组变为 [1, 3, 5, 8]。
4. 最后一次遍历：
  - 最后剩下最后一个元素 8，已经在正确位置。

在这个例子中，我们一共进行了 6 次比较 (3 + 2 + 1)。注意到我们每次比较的次数减少 1，这与公式  $C(n) = (n-1) + (n-2) + (n-3) + \dots + 1 = \frac{1}{2} * n * (n - 1) = (n^2 - n) * \frac{1}{2}$  相符。

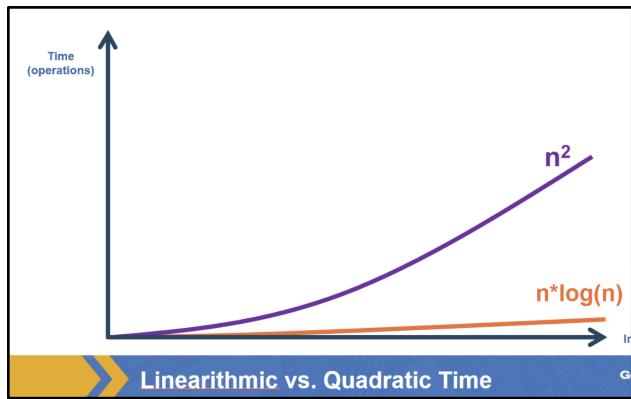
SelectionSort的复杂度： $C(n) = (n-1) + (n-2) + (n-3) + \dots + 1 = \frac{1}{2} * n * (n - 1) = (n^2 - n) * \frac{1}{2}$

当n足够大时，SelectionSort的算法复杂度是曲线增长的。(quadratic)

我们称之为： $O(n^2)$

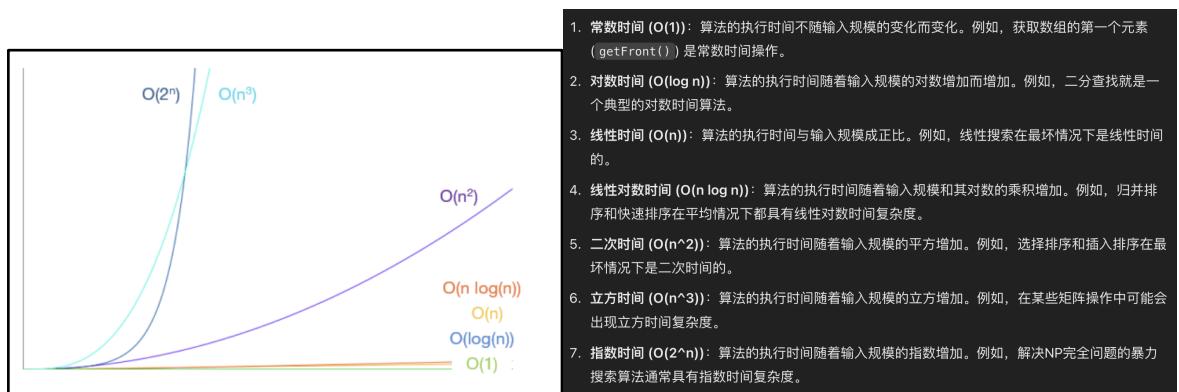
### 16.3 线性对数时间

线性 linear + 对数 logarithmic



这种增长速度不会像二次方增长那么快，比 $n^2$ ，和 $n$ 都要慢

## 16.4 所有时间对比



Growth Rate	Big-O notation
Constant	$O(1)$
Logarithmic	$O(\log(n))$
Linear	$O(n)$
Linearithmic	$O(n \log(n))$
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(a^n)$

## 17 线性对数算法的经典 - 二分查找法

### 二分查找发 : The Binary Search Algorithm

二分查找算法前提：

- 列表元素必须是排序

## 步骤：

- 找到列表的中间元素。
- 将中间元素的值与目标项的值进行比较。
- 重复以上步骤，直到在步骤3中找到目标项

如果单个元素不是目标项，则目标项不在列表中。结束。

我们通过一个已排序的数字列表来跟踪这个算法，并使用目标值13。

列表为：2 3 5 6 7 10 11 13

这里，我们有一个偶数长度的列表，所以中间有两个元素（6和7）。

列表为：2 3 5 6 7 10 11 13

在这种情况下，可以任选其中一个作为中间元素。这里，我们选择第二个中间元素。即7。

列表为：2 3 5 6 7 10 11 13

目标13大于7，所以我们忽略7以及它左边的所有内容：

列表为：2 3 5 6 7 10 11 13

现在，我们回到第一步，重新开始算法，但仅在右端子列表中执行：

子列表为：10 11 13

中间元素是11。

子列表为：10 11 13

目标13大于11，所以我们将忽略11及其左侧子列表：

子列表为：10 11 13

此时，我们剩下下一个单元元素列表13，它与目标匹配。

子列表为：13

通过这个过程，我们成功找到了目标值13。

## 二分查找法 vs 线性查找法：

线性查找在最佳情况下的运行时间是常数时间,  $O(1)$  ( 目标数正好是第一个 )

- (注意，是线性查找，不是线性排序)

二分查找在最佳情况下的运行时间也是常数时间,  $O(1)$  ( 目标数恰好在正中间 )

线性查找在最差情况下的运行时间是线性时间,  $O(n)$

二分查找在最差情况下的运行时间是对数时间，即一直从中间分，分到没有了。 $O(\log_2(n))$

## 二分查找法复杂度：

array中有多少数

复杂度 ( 需要运算多少次 )

n	$\log_2(n)$
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10
1,048,576	20

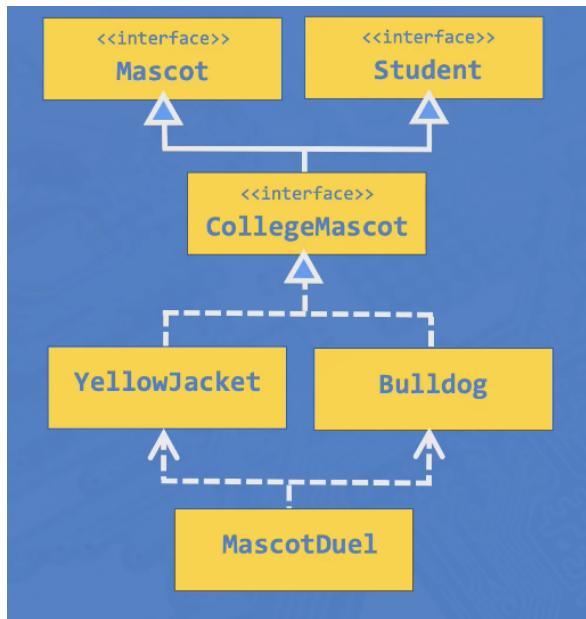
### 彩蛋：复杂度中的多态性

此处为了让学生体验复杂度，教授给了这个代码。有意思的是，这个numbers array的type不是int，而是Integer这个class。为什么？

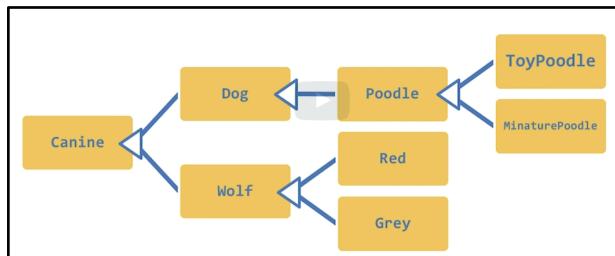
- 因为Integer这个Java api中的类实现了comparable这个接口，可以两个值进行比较。  
当然类型为int的也是可以比较的，只是这里让我们理解了用Integer的意义。

```
java
Integer[] numbers = new Integer[1000000];
Random rand = new Random();
for (int i = 0; i < numbers.length; i++) {
    numbers[i] = rand.nextInt(1000000);
}
```

你可以将其插入到主方法中，并尝试调用搜索方法，使用一些目标值并打印结果。请记住，这个方法是用于对象排序的，因此这段代码创建的是一个 Integer 数组，而不是 int 数组。另外，如果你查看Java API文档中的 Integer 类，你会看到它实现了 Comparable 接口。



1. `Mascot` 和 `Student` 是两个接口。
2. `CollegeMascot` 是另一个接口，它继承了 `Mascot` 和 `Student`。
3. `YellowJacket` 和 `Bulldog` 是实现了 `CollegeMascot` 接口的两个类
4. `MascotDuel` 类通过虚线箭头连接到 `YellowJacket` 和 `Bulldog`，即在 `MascotDuel` 中创建了 `YellowJacket` 和 `Bulldog` 对象的实例来实现 `MascotDuel` 类。
  - 虚线指向 class 不表示继承，表示在本 class 中使用了上面 class 的 object

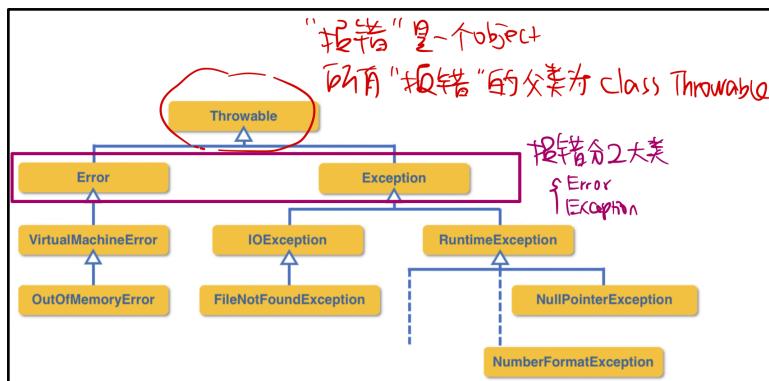


这种实线指向，且没有注明是interface的，就是普通父类，或者抽象类间的继承，可以看到，每个类最多只能继承一个父类。

## Module 8: 异常和文件输入/输出

### 1 6种java报错

异常是一个对象object。



( 注意 , 这是一个UML图 , 箭头的含义表示上面的class是这个class的父类 )

**Error**是电脑出错了 , 比如 : 当 JVM 没有足够的内存继续执行你的程序时。

- 无法compile无法run

**Exception**是你出错了 , 比如 : 你代码写错了 , int的type你输入了一个string这类的。

- 可以run , 问题出在runtime.

本课程focus在Exceptions上 , 也就是你自己的错误造成的程序运行停止。

7种Exceptions :

- NullPointerException
- ArrayIndexOutOfBoundsException
- StringIndexOutOfBoundsException
- ArithmeticException
- ClassCastException
- IllegalArgumentException
- FileNotFoundException

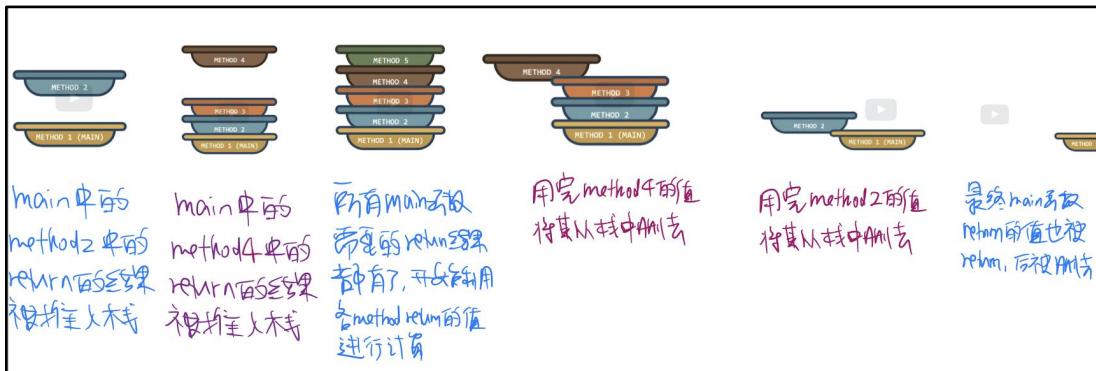
报错的整个过程 :

```

import java.util.Scanner;
public class FahrenheitToCelsiusExceptions {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter a Fahrenheit value: ");
        int fahrenheit = input.nextInt();
        double celsius = (5.0/9) * (fahrenheit - 32);
        System.out.printf("Fahrenheit: %d\n", fahrenheit);
        System.out.printf("Celsius: %.1f\n", celsius);
    }
}
  
```

Exception in thread "main" java.util.InputMismatchException  
at java.util.Scanner.throwFor(Scanner.java:854)  
at java.util.Scanner.next(Scanner.java:1485)  
at java.util.Scanner.nextInt(Scanner.java:2117)  
at java.util.Scanner.nextInt(Scanner.java:2076)  
at FahrenheitToCelsiusExceptions.main(FahrenheitToCelsiusExceptions.java:6)

Java程序运行过程中栈的变化：



当出现报错时，可以从下到上查看栈变化过程：



2 try-catch method

```
假设我们有以下代码:  
java  
try {  
    statement(s);  
} catch (ExceptionType identifier) {  
    statement(s);  
}
```

如果statement(s)真的报错了，它会匹配catch函数的错误type，然后执行相应的程序

这个过程可以：

- 发现错误
- 如果不主动抛出错误，java文件无法compile。
- 对可能出现的错误执行不同的解决方法

catch函数：

- 匹配catch如果statement throw出来的错误不和任何一个catch函数的错误匹配，系统会正常报出正常的错误。比如上面再上面的InputMismatchException。

- 如果有多个catch函数的错误都匹配，那这个try-catch函数只会执行第一个与之错误匹配的catch对应的程序。

catch的exception的顺序很重要：

```

import java.util.Scanner;
import java.util.InputMismatchException;

public class FahrenheitToCelsiusExceptions {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter a Fahrenheit value: ");
        try {
            int fahrenheit = input.nextInt();
            double celsius = (5.0 / 9) * (fahrenheit - 32);
            System.out.printf("Fahrenheit: %d\n", fahrenheit);
            System.out.printf("Celsius: %.1f\n", celsius);
            double x = 1331 / fahrenheit;
        } catch (InputMismatchException ime) {
            System.out.println("Sorry, that wasn't an int.");
            System.out.println("Please re-run the program again.");
        } catch (ArithmaticException ae) {
            System.out.println("You entered an invalid number:");
            System.out.println(ae.getMessage());
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}

```

*注意：exception是一个class，:: 扩号中是object*

*越该放在下面的catch中*

```

import java.util.Scanner;
import java.util.InputMismatchException;

public class FahrenheitToCelsiusExceptions {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter a Fahrenheit value: ");
        try {
            int fahrenheit = input.nextInt();
            double celsius = (5.0 / 9) * (fahrenheit - 32);
            System.out.printf("Fahrenheit: %d\n", fahrenheit);
            System.out.printf("Celsius: %.1f\n", celsius);
            double x = 1331 / fahrenheit;
        } catch (InputMismatchException | ArithmaticException e) {
            System.out.println("Sorry, that wasn't a valid value.");
            System.out.println("Please re-run the program again.");
            System.out.println("However, enter a non-zero integer.");
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}

```

*注意：这里创建了一个object，而非2个*

*可以用“or”即“|”把2种报错合在一个catch中*

**异常捕获的顺序**：catch 块的顺序非常重要。更具体的异常（如 InputMismatchException 和 ArithmaticException）应放在更通用的异常（如 Exception）之前，否则更通用的异常会先捕获所有异常，使得后面的 catch 块永远无法到达。

**Exception 捕获块的兜底作用**：Exception 是所有异常的父类，用于捕获那些没有明确处理的异常，确保程序在异常情况下仍然能进行适当的处理，而不会直接崩溃。

**注意**：exception是class，所以每一种报错出现的时候都是创造了一种这个exception的object

和statement相关的语句都要放入try中：

The image shows two side-by-side Java code editors. The left editor contains the following code:

```
import java.util.Scanner;
import java.util.InputMismatchException;
public class FahrenheitToCelsiusExceptions {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter a Fahrenheit value: ");
        try {
            int fahrenheit = input.nextInt();
        } catch(InputMismatchException e) {
            System.out.println("Sorry, that wasn't an int.");
            System.out.println("Please re-run the program again");
        }
        double celsius = (5.0/9) * (fahrenheit - 32);
        System.out.printf("Fahrenheit: %d\n", fahrenheit);
        System.out.printf("Celsius: %.1f\n", celsius);
    }
}
```

The right editor contains the same code, but with annotations:

```
import java.util.Scanner;
import java.util. InputMismatchException;
public class FahrenheitToCelsiusExceptions {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter a Fahrenheit value: ");
        try {
            int fahrenheit = input.nextInt();
            double celsius = (5.0/9) * (fahrenheit - 32);
            System.out.printf("Fahrenheit: %d\n", fahrenheit);
            System.out.printf("Celsius: %.1f\n", celsius);
        } catch(InputMismatchException e) {
            System.out.println("Sorry, that wasn't an int.");
            System.out.println("Please re-run the program again");
        }
    }
}
```

Annotations in red highlight the try-catch block and the assignment statement inside it. A green checkmark indicates the code on the right is correct.

在try-catch函数中，要测试的statement中，直接或间接用到这个statement的语句，也要包含在try中，不然会达不到发现错误的效果。为什么？

- 因为如果和这个statement间接相关的语句在try外，会无法compile。
- (记住，java文件要先javac来compile，然后java xxx来执行)

### 3 异常控制循环 Exception Controlled Loops

try-catch函数在真实coding中的用例 - 异常控制循环：

```

import java.util.Scanner;
import java.util.InputMismatchException;

public class FahrenheitToCelsiusExceptions {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        boolean success = false;
        int fahrenheit = 0; 利用try-catch函数  
实现一个loop的功能

        while (!success) { ① 当出现输入值type有问题时，重新读取
            try {
                System.out.print("Enter a Fahrenheit value: ");
                fahrenheit = input.nextInt();
                success = true; 退出while loop ④
            } catch (InputMismatchException e) {
                input.nextLine(); // 清理输入流中的换行符
                System.out.println("Sorry, that wasn't an int.");
                System.out.println("Please try again");
            }
        } ②只要出现InputMismatch，success都不会变为true，会继续在while loop中执行try
        double celsius = (5.0 / 9) * (fahrenheit - 32);
        System.out.printf("Fahrenheit: %d\n", fahrenheit);
        System.out.printf("Celsius: %.1f\n", celsius);
    }
}

```

input.nextLine()语句进入下一行很重要，不然下一次输入的时候，上一次输入的值还在里面。所以要换行。

- 比如我输入thirty报错了，如果没有这行代码，下一次我输入正确的30，系统会理解我的input是30，还是无法正常运行。

#### 4 try-catch好处

<p>使用 System.exit() 的版本：</p> <pre> java import java.util.Scanner;  public class FahrenheitToCelsiusNoExceptions {     public static void main(String[] args) {         Scanner input = new Scanner(System.in);         int fahrenheit = 0;          System.out.print("Enter a Fahrenheit value: "); <span style="color: red;">① 先检查输入的type是否正确</span>         if (input.hasNextInt()) {             fahrenheit = input.nextInt();         } else {             System.out.println("Sorry, that wasn't an int.");             System.out.println("Please re-run the program again");             System.exit(1); <span style="color: red;">~不正确直接return 1;</span>         }          double celsius = (5.0 / 9) * (fahrenheit - 32);         System.out.printf("Fahrenheit: %d\n", fahrenheit);         System.out.printf("Celsius: %.1f\n", celsius);     } } </pre>	<p>两边实现功能完全一样</p> <p>使用异常处理的版本：</p> <pre> import java.util.Scanner; import java.util.InputMismatchException;  public class FahrenheitToCelsiusExceptions {     public static void main(String[] args) {         Scanner input = new Scanner(System.in);         System.out.print("Enter a Fahrenheit value: ");          try {             int fahrenheit = input.nextInt();             double celsius = (5.0 / 9) * (fahrenheit - 32);             System.out.printf("Fahrenheit: %d\n", fahrenheit);             System.out.printf("Celsius: %.1f\n", celsius);         } catch (InputMismatchException e) {             System.out.println("Sorry, that wasn't an int.");             System.out.println("Please re-run the program again");         }     } } </pre> <p><span style="color: blue;">不预先检验输入值的type，且会返回相应的错误信息</span></p>
--	--

- 左边的System.exit(1)确实能实现一样的功能，但是在程序有10000行时，debug起来非常麻烦，不知道error出在哪，也不知道是哪种type的error，只会return 1；

- 右边的会给出报错信息，而且不会主动检查输入值，对程序运行时的栈友好

## 5 try-catch-finally method (catch错误)

```

import java.util.Scanner;
import java.util.InputMismatchException;

public class FahrenheitToCelsiusExceptions {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        boolean success = false;
        int fahrenheit = 0;

        while (!success) {
            try {
                System.out.print("Enter a Fahrenheit value: ");
                fahrenheit = input.nextInt();
                success = true;
            } catch (InputMismatchException e) {
                input.nextLine(); // 若输入错误，则换一行再输入
                System.out.println("Sorry, that wasn't an int.");
                System.out.println("Please try again");
            }
        }

        input.nextLine(); // 清理操作
        System.out.print("Enter a day of the week: ");
        String day = input.nextLine();
        double celsius = (5.0 / 9) * (fahrenheit - 32);
        System.out.printf("%s Fahrenheit: %d\n", day, fahrenheit);
        System.out.printf("%s Celsius: %.1f\n", day, celsius);
    }
}

```

```

import java.util.Scanner;
import java.util.InputMismatchException;

public class FahrenheitToCelsiusExceptions {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        boolean success = false;
        int fahrenheit = 0;

        while (!success) {
            try {
                System.out.print("Enter a Fahrenheit value: ");
                fahrenheit = input.nextInt();
                success = true;
            } catch (InputMismatchException e) {
                System.out.println("Sorry, that wasn't an int.");
                System.out.println("Please try again");
            }
        }

        input.nextLine(); // 清理操作
        System.out.print("Enter a day of the week: ");
        String day = input.nextLine();
        double celsius = (5.0 / 9) * (fahrenheit - 32);
        System.out.printf("%s Fahrenheit: %d\n", day, fahrenheit);
        System.out.printf("%s Celsius: %.1f\n", day, celsius);
    }
}

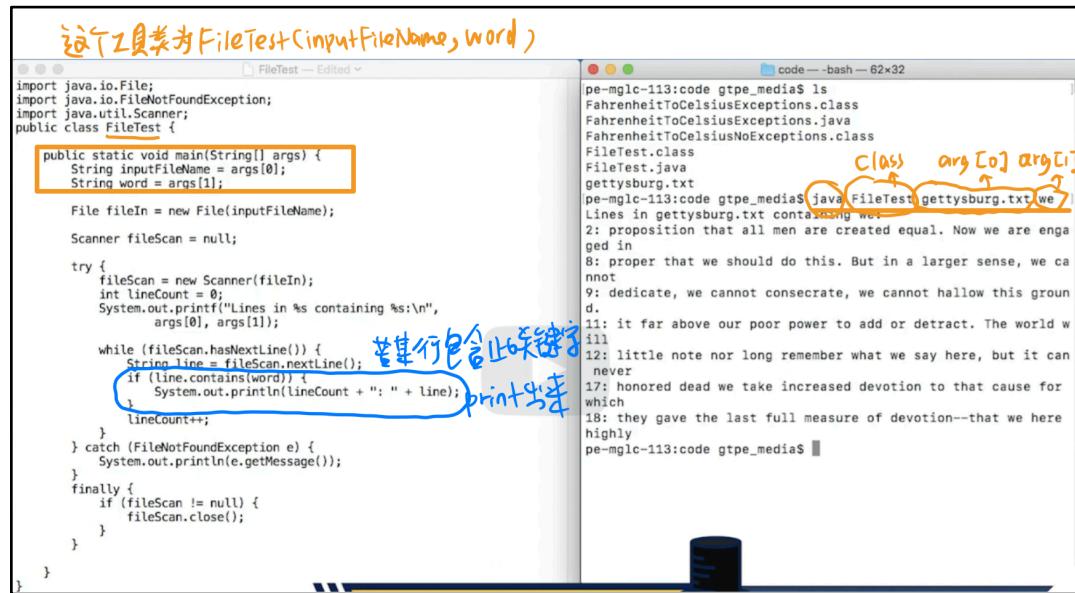
// 手写注释
// 若输入成功，换一行再输入并输入 "day"
// 若输入失败，清理操作后，等待下一个input
// 若成功也执行，等待下一个input
// 若失败也执行，等待下一个input
// 若成功或失败都执行
// 若失败，清理操作放在finally中
// ① finally保证try是否成功都换行

```

finally语句保证try的结果无论是成功，还是报错，都会运行finally语句中的命令。

- 此处是换一行等待下一个input

### 5.1 try-catch-finally 例子



这个工具类为 `FileTest(inputFileName, word)`

```

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
public class Filetest {
    public static void main(String[] args) {
        String inputFileName = args[0];
        String word = args[1];
        File fileIn = new File(inputFileName);
        Scanner fileScan = null;
        try {
            fileScan = new Scanner(fileIn);
            int lineCount = 0;
            System.out.printf("Lines in %s containing %s:\n", args[0], args[1]);
            while (fileScan.hasNextLine()) {
                String line = fileScan.nextLine();
                if (line.contains(word)) {
                    System.out.println(lineCount + ": " + line);
                }
                lineCount++;
            }
        } catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
        }
        finally {
            if (fileScan != null) {
                fileScan.close();
            }
        }
    }
}

```

class arg [0] arg [1]

Lines in gettysburg.txt containing we:  
2: proposition that all men are created equal. Now we are engaged in  
8: proper that we should do this. But in a larger sense, we cannot  
9: dedicate, we cannot consecrate, we cannot hallow this ground.  
11: it is far above our poor power to add or detract. The world will  
12: little note nor long remember what we say here, but it can never  
17: honored dead we take increased devotion to that cause for which  
18: they gave the last full measure of devotion--that we here highly

首先这是一个FileTest工具类，有2个参数：

- arg[0]代表要识别的文章
- arg[1]代表要寻找的关键字

要实现的功能：

- 在x行，如果有这个关键字，print出来，并且在所有输出的第一行print“本文档有某某关键字”

怎么调用这个函数：

- java class arg[0] arg[1] - 这样输入在terminal中就可以调用

用一个finally语句优雅的保证了：

- 如果成果扫描完整个文档，那么程序会自动关掉文档
- 如果报错了，程序也会自动关掉文档

这样当程序运行完成，不会留着文档在后台运行，一个字优雅。

注意：这个例子还出现了一个之前没谈论过的exception种类：[FileNotFoundException](#)

6 class头上的throws Exception ( declare错误 )

```

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.io.PrintWriter;

public class FileTest {
    public static void main(String[] args) {
        String inputFileFileName = args[0];
        String word = args[1];

        File fileIn = new File(inputFileName);
        File fileOut = new File(word + "In" + inputFileFileName);

        Scanner fileScan = null;
        PrintWriter filePrint = null;

        try {
            fileScan = new Scanner(fileIn);
            filePrint = new PrintWriter(fileOut);

            int lineCount = 0;
            filePrint.printf("Lines in %s containing %s:\n", args[0], args[1]);

            while (fileScan.hasNextLine()) {
                String line = fileScan.nextLine();
                if (line.contains(word)) {
                    filePrint.println(lineCount + ": " + line);
                }
                lineCount++;
            }
        } catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
        } finally {
            if (fileScan != null) {
                fileScan.close();
            }
            if (filePrint != null) {
                filePrint.close();
            }
        }
    }
}

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileTest {
    public static void main(String[] args) throws FileNotFoundException {
        String fileName = args[0];
        String word = args[1];

        File file = new File(args[0]);
        Scanner scan = null;

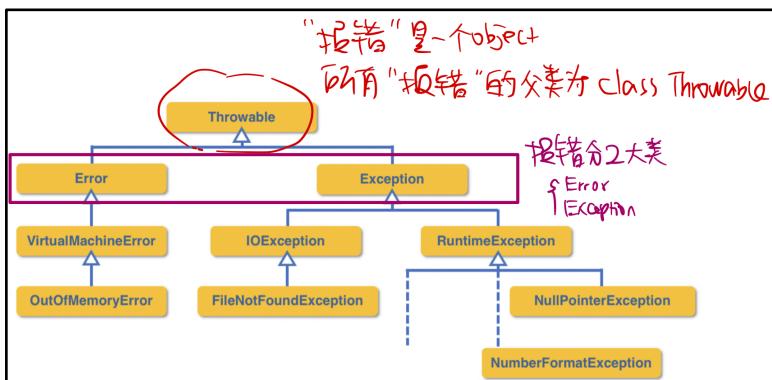
        scan = new Scanner(file);
        int lineCount = 0;
        System.out.printf("Lines in %s containing %s:\n", args[0], args[1]);

        while (scan.hasNextLine()) {
            String line = scan.nextLine();
            if (line.contains(word)) {
                System.out.println(lineCount + ": " + line);
            }
            lineCount++;
        }
        scan.close();
    }
}

```

如果没有try-catch, code中不处理报错, java文件就无法编译。另一种解决方法: 把throws Exception写在main方法header

## 7 Runtime Exception不用catch或declare



未检查异常 ( Unchecked Exceptions ) 是 `RuntimeException` 及其子类 , 表示编程错误 , 如空指针异常或算术异常。Java 编译器不强制要求我们 `catch` 或 `declare` 这些异常 , 因为强制处理会使代码繁琐且难以维护。

已检查异常 ( Checked Exceptions ) 是 `Exception` 类的子类 , 但不包括 `RuntimeException` 。 Java 强制要求你 `catch` 或 `declare` 这些异常 , 因为它们通常涉及程序无法控制的外部情况 , 如文件未找到。未处理这些异常会导致程序无法编译。

## 8 重构一个Exception

重构一个exception , 用 `new xxxException( )` 来 declare -

```

import java.util.Scanner;
import java.util.InputMismatchException;

public class FahrenheitToCelsiusExceptions {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter a Fahrenheit value: ");
        try {
            int fahrenheit = input.nextInt();
            double celsius = (5.0/9) * (fahrenheit - 32);
            System.out.printf("Fahrenheit: %d\n", fahrenheit);
            System.out.printf("Celsius: %.1f\n", celsius);
            if (fahrenheit == 0) {
                throw new DivideByZeroException();
            }
            double x = 1331/fahrenheit; (在-一个try-catch语句中
                                         嵌套一个throw exception)
        } catch(InputMismatchException ime) {
            System.out.println("Sorry, that wasn't an int.");
            System.out.println("Please re-run the program again");
        }
        catch(DivideByZeroException dze) {
            System.out.println("Oops. You entered an invalid number:");
            System.out.println(dze.getMessage());
        }
        catch(Exception e) {
            System.out.println(e.getMessage());
        }
    }
}

```

若出现 fahrenheit=0, 会抛出 "Divide by zero"

```

public class DivideByZeroException extends ArithmeticException {
    public DivideByZeroException() {
        super("Divide by zero.");
    }
}

```

### C Java中子类继承的 Super用法)

```

// 父类: Animal
class Animal {
    String name;

    // 父类的构造函数
    public Animal(String name) {
        this.name = name;
    }

    // 父类的方法
    public void makeSound() {
        System.out.println("Some generic animal sound");
    }
}

// 子类: Dog
class Dog extends Animal {

    // 子类的构造函数, 调用父类的构造函数
    public Dog(String name) {
        super(name); // 调用父类的构造函数
    }

    // 子类重写父类的方法
    @Override
    public void makeSound() { // Super. method 调用父类
        super.makeSound(); // 调用父类的方法
        System.out.println(name + " says: Woof Woof!");
    }
}

```

用 super (参数) 调用父类  
constructor  
method

(这个例子还展示了，可以在一个try-catch语句中嵌套一个自己重构的Exception)

## 9 使用try-catch函数，利用csv文件做数据分析

**② 先将未排序的data，排序后存入CSV**

```

public static void main(String[] args) {
    Wolf[] pack = {
        new Wolf(17, 1, 2),
        new Wolf(3, 10),
        new Wolf(9, 2, 7),
        new Wolf(9, 1, 8),
        new Wolf(17, 0, 3),
        new Wolf(16, 2, 1),
        new Wolf(16, 4),
        new Wolf(16, 5),
        new Wolf(10, 6),
        new Wolf(5, 9)
    };

    Arrays.sort(pack);

    File fileOut = new File("SortedWolves.csv");
    PrintWriter filePrint = null;
    try {
        filePrint = new PrintWriter(fileOut);
        for (Wolf wolf : pack) {
            filePrint.println(wolf.getRank() + "," + wolf.getSize());
        }
    } catch (FileNotFoundException e) {
        System.out.println(e.getMessage());
    }

    finally {
        if (filePrint != null) {
            filePrint.close();
        }
    }
}

```

**CSV文件：**

10,3.0
9,5.0
8,9.1
7,9.2
6,10.0
5,16.0
4,16.0
3,17.01
2,17.1
1,16.2

**①**

```

public static void main(String[] args) {
    Wolf[] pack = {
        new Wolf(3, 1, 3),
        new Wolf(3, 10),
        new Wolf(6, 2, 7),
        new Wolf(9, 1, 8),
        new Wolf(17, 0, 3),
        new Wolf(16, 2, 1),
        new Wolf(16, 4),
        new Wolf(16, 5),
        new Wolf(10, 6),
        new Wolf(5, 9)
    };

    System.out.println("Unsorted Pack: " + Arrays.toString(pack));
    Arrays.sort(pack);
    System.out.println("Sorted Pack: " + Arrays.toString(pack));
}

```

输出结果如下：

Unsorted Pack: [Rank 3, Size 3.0, Rank 10, Size 9.0, Rank 7, Size 8.0, Rank 8, Size 1.0, Rank 6, Size 16.0, Rank 5, Size 16.2]
Sorted Pack: [Rank 10, Size 3.0, Rank 9, Size 8.0, Rank 8, Size 6.0, Rank 7, Size 1.0, Rank 6, Size 16.0, Rank 5, Size 16.2]

**③a 方法一**

```

public static void main(String[] args) {
    File fileIn = new File("SortedWolves.csv");
    Scanner fileScan = null;
    String[] tokens = null;
    double[] allWeights = new double[10];
    int index = 0;
    try {
        fileScan = new Scanner(fileIn);
        String line = null;
        while (fileScan.hasNextLine()) { ① 从行0开始 ②
            line = fileScan.nextLine(); ③ 下一行
            tokens = line.split(","); ④ 找到 ","
            allWeights[index] = Double.parseDouble(tokens[1]); ⑤ 把","后的type为double的数加入 allWeights 这个array中
            index++; ⑥ 完另一个, 这里 ⑦
        }
    } catch (FileNotFoundException e) {
        System.out.println(e.getMessage());
    }

    finally {
        if (fileScan != null) {
            fileScan.close();
        }
    }
}

```

**但是这种方法有个问题：**

**③b 方法二**

```

public static void main(String[] args) {
    File fileIn = new File("SortedWolves.csv");
    Scanner fileScan = null;
    Scanner wolfScan = null;
    double[] allWeights = new double[10];
    int index = 0;
    try {
        fileScan = new Scanner(fileIn);
        String line = null;
        while (fileScan.hasNextLine()) { ① 从行0开始
            line = fileScan.nextLine(); ② 下一行
            wolfScan = new Scanner(line);
            wolfScan.useDelimiter(","); ③ 此行内用","分割内容
            allWeights[index] = wolfScan.nextDouble(); ④ 将下一个type为double的数加入 allWeights 中
            index++; ⑤
        }
    } catch (FileNotFoundException e) {
        System.out.println(e.getMessage());
    }

    finally {
        if (fileScan != null) {
            fileScan.close();
        }
    }
}

```

方法1和方法2最终都会得到一个allWeights的array，这个array是一个只有wolf体重值的列表，并且从小到大排序

- 注意这个方法中因为有的可能报错的地方，如：FileNotFoundException
- 注意最后都会在finally语句中关掉这个csv文件避免其在后台空转

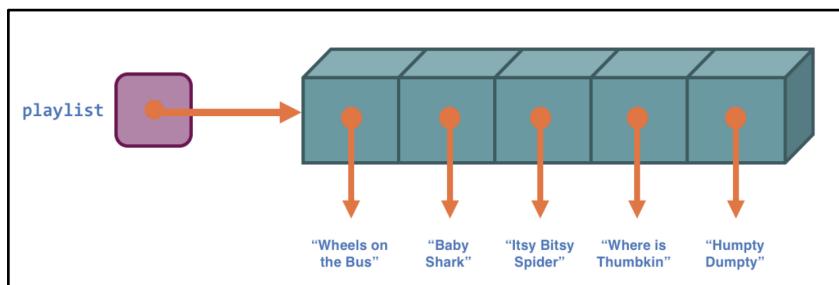
## Module 9: List接口

### 1 数组列表&链表都是list接口的实现

本章将探讨 List 接口，并展示了如何以不同的方式实现相同的概念。接着：

- 创建和操作 ArrayList ( 数组列表 ) 和 LinkedList ( 链表 )。
- 理解数组和链表的数据结构差异。

### 2 数组列表 ArrayList



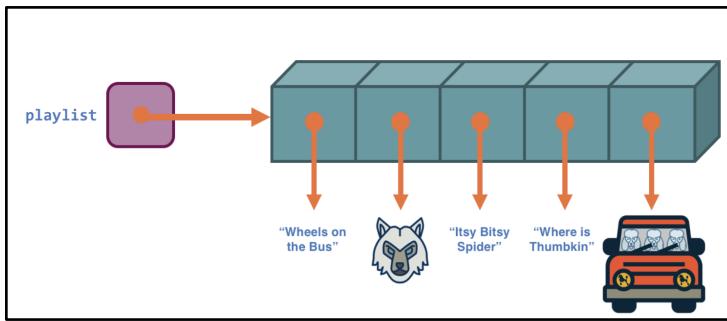
创建不限制变量type的ArrayList：

没有限制的 ArrayList 创建:	指定初始容量的 ArrayList 创建:
<pre>java ArrayList playlist = new ArrayList();</pre>	<pre>java ArrayList playlist = new ArrayList(5);</pre>

创建限制变量type的ArrayList：

<pre>java ArrayList&lt;elementType&gt; aList = new ArrayList&lt;&gt;();</pre>	<pre>java ArrayList&lt;String&gt; playlist = new ArrayList&lt;&gt;(5);</pre>
---	--

不限制变量的ArrayList是真的不限制变量type，而不是暂时不定义变量type，看下图：



### 3 数组列表 vs 数组 7大区别

#### 1. 大小可变性

- **Array**: 固定大小。一旦声明了数组的大小，就无法改变其大小。
- **ArrayList**: 动态大小。**ArrayList** 的大小是可变的，随着元素的添加或删除，**ArrayList** 的大小会自动调整。

例子:

```
java 复制代码
// Array
int[] array = new int[5]; // 创建一个大小为5的数组
array[0] = 1;
array[1] = 2;
// 不能添加更多的元素，如果超出大小会抛出 ArrayIndexOutOfBoundsException

// ArrayList
ArrayList<Integer> arrayList = new ArrayList<>();
arrayList.add(1); // 添加元素，大小自动增加
arrayList.add(); // 添加更多元素
arrayList.add();
// 可以继续添加元素，大小会自动调整
```

#### 2. 类型限制

- **Array**: 可以存储原始数据类型（如 `int`, `char`, `double` 等）和对象类型。
- **ArrayList**: 只能存储对象类型（如 `Integer`, `String`, `Double` 等）。如果需要存储原始数据类型，必须使用其对应的包装类（如 `Integer` 代替 `int`）。

例子:

```
java 复制代码
// Array storing primitive types
int[] intArray = new int[3];
intArray[0] = 1;

// ArrayList storing objects (primitive types need to be wrapped)
ArrayList<Integer> intArrayList = new ArrayList<>();
intArrayList.add(1); // 只能存储 Integer 对象
```

#### 3. 性能

- **Array**: 由于是固定大小的结构，内存分配较为连续，性能通常比 **ArrayList** 更高，特别是在对数组进行频繁的读取操作时。
- **ArrayList**: 由于是动态大小的结构，每次调整大小时，可能会涉及到重新分配和复制元素，性能比数组稍低，但在需要动态调整大小的场景下更为灵活。

例子:

```
java 复制代码
// Array has faster access times
int[] intArray = {1, 2, 3};
int firstElement = intArray[0]; // O(1) 时间复杂度

// ArrayList access is slightly slower due to additional overhead
ArrayList<Integer> intArrayList = new ArrayList<>();
intArrayList.add(1);
intArrayList.add(2);
int firstArrayListElement = intArrayList.get(0); // 仍然是 O(1)，但有额外开销
```

#### 4. 内存占用

- **Array**: 使用较少的内存，因为它直接存储元素。
- **ArrayList**: 由于 **ArrayList** 包装了数组，并且包含了额外的属性和方法，因此其内存占用比数组稍高。

#### 5. 内置方法

- **Array**: 没有内置的方法用于操作数据结构，所有操作（如排序、搜索）都需要通过手动编码或使用外部库来实现。
- **ArrayList**: 提供了一系列内置方法，例如 `add`, `remove`, `get`, `set`, `size`, `contains` 等，用于直接操作列表。

例子:

```
java 复制代码
// Array does not have built-in methods
int[] intArray = {1, 2, 3};
// Manual search
boolean found = false;
for (int i = 0; i < intArray.length; i++) {
    if (intArray[i] == 2) {
        found = true;
        break;
    }
}

// ArrayList has built-in methods
ArrayList<Integer> intArrayList = new ArrayList<>();
intArrayList.add(1);
intArrayList.add(2);
intArrayList.add(3);
boolean containsTwo = intArrayList.contains(2); // 直接调用 contains 方法
```

#### 6. 多维支持

- **Array**: 支持多维数组（例如二维、三维数组等）。
- **ArrayList**: 仅支持一维结构，但你可以创建 **ArrayList** 的 **ArrayList** 来模拟多维结构。

例子:

```
java 复制代码
// Multi-dimensional array
int[][] multiArray = new int[2][3];
multiArray[0][0] = 1;

// Simulating multi-dimensional ArrayList
ArrayList<ArrayList<Integer>> multiArrayList = new ArrayList<>();
ArrayList<Integer> innerList = new ArrayList<>();
innerList.add(1);
multiArrayList.add(innerList);
```

#### 8. 线程安全

- **Array**: 数组本身不是线程安全的。
- **ArrayList**: 也不是线程安全的，如果需要在多线程环境下使用，需要手动同步（可以使用 `Collections.synchronizedList` 来创建一个同步的 **ArrayList**）。

概括：

1. Array大小不能调，固定好了。ArrayList可编辑element，且每次add/delete都很耗性能
2. Array储存的都是原始type，ArrayList储存的都是object这个type，故ArrayList很占内存
3. Array没有接口去操作数组本身，ArrayList有一堆接口操作ArrayList，如add delete
4. Array支持储存矩阵，ArrayList不支持矩阵
5. Array和ArrayList的线程都不安全。

## 4 自动拆装箱 Autoboxing & Auto-unboxing

```
import java.util.ArrayList;

public class IntArrayListDemo {
    public static void main(String[] args) {
        ArrayList<Integer> scores = new ArrayList<>();
        scores.add(90); // 自动装箱
        scores.add(95); // 输入的是int, JVM会自动将其转换为Integer的object
        scores.add(110);
        scores.add(99);
        scores.add(105); // int转换为Integer的object
        System.out.println(scores.toString());
    }

    int sum = 0;
    for (Integer score : scores) { // 使用for-each循环
        sum += score; // 自动拆箱
    } // 在进行加减时, JVM又会把Integer的object转换回int进行计算, 叫Auto-unboxing
    System.out.println("Total points: " + sum);
}
}
```

Integer 是 Java 中的一个包装类，用于将原始数据类型 int 封装成对象，以便在需要对对象的上下文中使用。Integer 类提供了许多有用的方法，例如用于解析字符串为整数、比较整数值、获取二进制、八进制或十六进制字符串表示等。以下是 Integer 类的一些关键部分和常见方法的示例：

Integer 类的定义与重要部分（简化版）

```
java
public final class Integer extends Number implements Comparable<Integer> {
    public static final int MIN_VALUE = 0x80000000; // -231
    public static final int MAX_VALUE = 0x7fffffff; // 231-1
    public static final Class<Integer> TYPE = (Class<Integer>) Class.getPrimitiveTypeForName("int");
    private final int value; // 实际存储整数的字段

    // 构造函数
    public Integer(int value) {
        this.value = value;
    }

    public Integer(String s) throws NumberFormatException {
        this.value = parseInt(s, 10);
    }

    // 常用方法
    public int intValue() {
        return value;
    }

    public static Integer valueOf(int i) {
        return new Integer(i);
    }

    public static Integer valueOf(String s) throws NumberFormatException {
        return Integer.valueOf(parseInt(s, 10));
    }

    public static int parseInt(String s) throws NumberFormatException {
        return parseInt(s, 10);
    }

    public static int parseInt(String s, int radix) throws NumberFormatException {
        // 解析字符串为整数
    }

    public static Integer decode(String nn) throws NumberFormatException {
        // 从字符串中读取整数
    }
}
```

int sum = new Integer(3);

True or false: The previous statement will not compile.

True

False

✓

### 4.1 Integer vs int (为什么要类化java的基础类型primitive type?)

为什么在ArrayList中插入Integer，而不是int。Integer有什么优点呢？

- 可以看到右边，Integer这个类有非常多的method可以用于本身int的运算。因此在需要对数字进行繁琐操作运算的应用场景，Integer比int好。

## 5 泛型类

泛型类 ( generic class ) 的四种最常见例子：

```
public class Bin<T> { → 规定 Bin 的 constructor 中的参数  
    private T content;  
  
    public Bin(T content) {  
        this.content = content;  
    }  
  
    public T getContent() {  
        return content;  
    }  
  
    public void setContent(T content) {  
        this.content = content;  
    }  
  
    public static void main(String[] args) {  
        Bin<String> test = new Bin<>("I'm a basic String.");  
        // test.setContent(new Car("Yuhina", "Spark", 2037)); // 编译错误  
        System.out.println(test.getContent());  
    }  
}
```

① 只能是唯一的 - 种 type  
② 此处 declare 了 Bin 中参数类型为 String  
③ 将其内容强转为类型的 Car 的 Bin，将会报错

```
public class Bin2<T> { → 限定一种 type，但有多个参数  
    private T content1;  
    private T content2;  
  
    public Bin2(T content1, T content2) {  
        this.content1 = content1;  
        this.content2 = content2;  
    }  
  
    public T getContent1() {  
        return content1;  
    }  
  
    public void setContent1(T content1) {  
        this.content1 = content1;  
    }  
  
    public T getContent2() {  
        return content2;  
    }  
  
    public void setContent2(T content2) {  
        this.content2 = content2;  
    }  
  
    public static void main(String[] args) {  
        Bin2<String> test = new Bin2<>("I'm a basic String", "short and stout");  
        System.out.println(test.getContent1());  
        System.out.println(test.getContent2());  
    }  
}
```

String      String

( 泛型 class - 带规定 type 的 <T> 表示的 class )

( 限定一种 type，但是有多个参数 )

```
public class Bin2Diff<X, Y> { → 两种不同 type 的参数都可以接受  
    private X content1;  
    private Y content2;  
  
    public Bin2Diff(X content1, Y content2) {  
        this.content1 = content1;  
        this.content2 = content2;  
    }  
  
    public X getContent1() {  
        return content1;  
    }  
  
    public void setContent1(X content1) {  
        this.content1 = content1;  
    }  
  
    public Y getContent2() {  
        return content2;  
    }  
  
    public void setContent2(Y content2) {  
        this.content2 = content2;  
    }  
  
    public static void main(String[] args) {  
        Bin2Diff<String, Car> test = new Bin2Diff<>("My dream car", new Car("Yuhina"));  
        System.out.println(test.getContent1());  
        System.out.println(test.getContent2());  
    }  
}
```

String 类型      Car 类型

```
java  
public interface Comparable<T> {  
    int compareTo(T o);  
}  
  
public class BinCompare<T extends Comparable<T>> {  
    private T content1;  
    private T content2;  
  
    public BinCompare(T content1, T content2) {  
        this.content1 = content1;  
        this.content2 = content2;  
    }  
  
    public T greaterValue() {  
        return (content1.compareTo(content2) >= 0) ? content1 : content2;  
    }  
  
    public static void main(String[] args) {  
        BinCompare<String> test = new BinCompare<>("I'm a basic String", "short and stout");  
        System.out.println(test.greaterValue());  
    }  
}
```

继承这个抽象类，且比较的两个类型必须同属一个 type

( 多种 type，多个参数 )

( 一个 type，且此泛型类中要继承重构另一个泛型 class )

## 6 ArrayList 缺点 - why 链表

当要往ArrayList中添加新元素，而length又不够时，实际上时需要重新创建了一个新数组列表并复制旧数组元素进去。很消耗的资源。

- 逻辑上新添加的element的右边的所有element都要向右移，操作也很繁琐。

LinkedList不要求我们提供初始大小。

创建一个新的链表object：

```
GenericLinkedList<elementType> favBabySongs = new GenericLinkedList<>();
```

## 7 链表类中相关method

The diagram shows the Java code for the `Node` class and a linked list structure. The `Node` class is defined as:

```
private class Node<E> {  
    E data;  
    Node<E> next;  
    Node(E data, Node<E> next) {  
        this.data = data;  
        this.next = next;  
    }  
}
```

Annotations explain:

- ② 为谁“戴头”的node? Node?
- ① 这里的node类就绪

解释:

- Node 类: Node 类是链表的基础, 每个节点存储数据 (data) 以及指向下一个节点的引用 (next)。
- 泛型参数: `Node<E>` 使用了泛型参数 `E`, 这意味着节点可以存储任何类型的数据。
- next 变量: `next` 是一个指向另一个 `Node` 对象的引用, 这使得节点能够链接在一起形成链表。
- 构造函数: 构造函数接受数据和下一个节点的引用, 并将它们初始化为节点的实例变量。

在链表中, 每个节点不仅存储数据, 还需要存储一个指向下一个节点的引用, 以确保链表的连贯性。

例子帮助理解:

假设你有一个链表, 节点 A 中的数据是 10, 节点 B 中的数据是 20, 节点 C 中的数据是 30, 链表的结构大致如下:

```
rust  
A -> B -> C -> null
```

车库里没有留下这种痕迹，是人为制造的class罢了

这里, 节点 A 的 `next` 引用指向节点 B, 节点 B 的 `next` 引用指向节点 C, 节点 C 的 `next` 引用为 `null`, 表示链表的末尾。

具体到代码的解释:

```
java  
Node<Integer> nodeA = new Node<(10, null);  
Node<Integer> nodeB = new Node<(20, nodeA);  
Node<Integer> nodeC = new Node<(30, nodeB);
```

先创建末尾节点  
创建中间节点, 并指向尾节点  
创建第一个节点, 并指向中间节点

(3)

当你创建 `nodeA` 时, `next` 被初始化为 `null`, 因为这是链表的最后一个节点。

当你创建 `nodeB` 时, 你的 `next` 引用指向 `nodeA`, 因此 `nodeB` 后面跟着 `nodeA`。

类似地, `nodeC` 的 `next` 引用指向 `nodeB`, 这样就把节点链接起来, 形成了链表。

总结:

`next` 的类型是 `Node<E>` 本身, 这意味着它指向链表中的下一个节点。这个设计使得链表能够通过节点的自引用类型, 将所有节点链接在一起, 形成一个动态的数据结构。

( Node类 )

( 链表例子 )

### 2. 创建并链接节点

代码:

```
java  
new Node<String>(newData, null);
```

解释:

- 节点创建: 上面的代码创建了一个新的 `Node` 对象, 并将其数据类型指定为 `String`。
- 初始化: `newData` 是节点存储的数据, `null` 表示当前节点是链表中的最后一个节点, 因此没有后继节点。

这个示例展示了如何创建一个链表的第一个节点 (也称为“头节点”), 此时链表中只有一个节点, 且没有链接到其他节点。

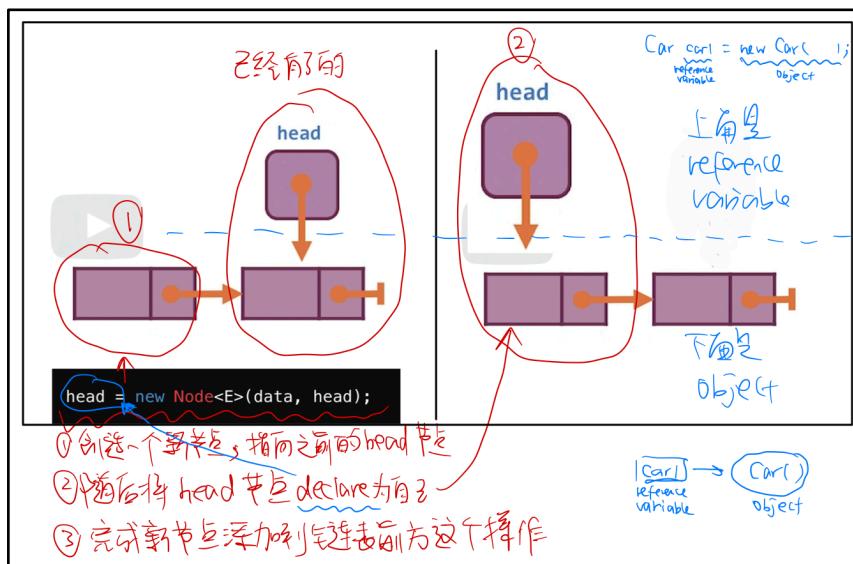
```
public class LinkedListExample {  
  
    // 定义节点类  
    private static class Node<E> {  
        E data; // 节点存储的数据  
        Node<E> next; // 指向下一个节点的引用  
  
        // 构造函数  
        Node(E data, Node<E> next) {  
            this.data = data;  
            this.next = next;  
        }  
    }  
  
    public static void main(String[] args) {  
        // 创建链表的节点  
        Node<Integer> node3 = new Node<(30, null); // 最后一个节点, next为null  
        Node<Integer> node2 = new Node<(20, node3); // 中间节点, next指向node3  
        Node<Integer> node1 = new Node<(10, node2); // 第一个节点, next指向node2  
  
        // head 指向第一个节点  
        Node<Integer> head = node1; // head 指向首节点  
  
        // 遍历链表并打印每个节点的数据  
        Node<Integer> current = head;  
        while (current != null) {  
            System.out.println(current.data);  
            current = current.next; // 移动到下一个节点  
        }  
    }  
}
```

( 创建一个末尾链表节点 ,

( 创建一个head指向首节点 )

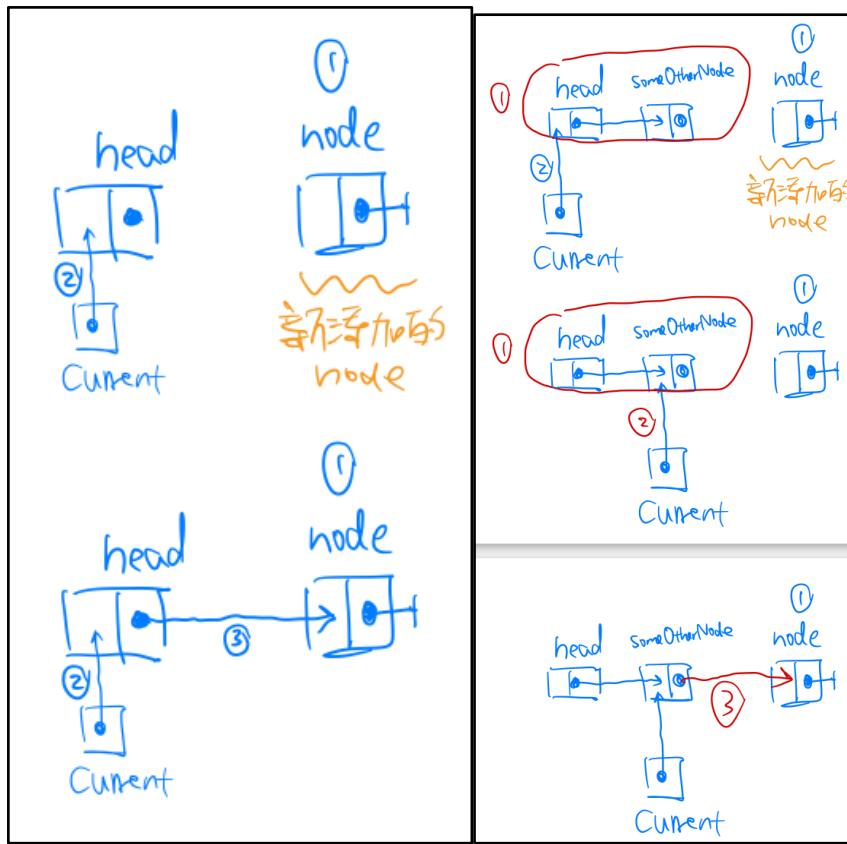
null表示没有指向任何节点, 即末尾节点 )

## 7.1 将新节点添加到链表前方



## 7.2 将新节点添加到链表末尾

```
public void addToRear(E newData) {  
    Node<E> node = new Node<E>(newData, null); ①  
    Node<E> current = head; ②  
    if (head == null) {  
        head = node; ③  
    }  
    else {  
        ①  
        while (current.next != null) {  
            current = current.next; ②  
        }  
        current.next = node; ③  
    }  
}
```



current是一个reference variable

Car car1 = new Car();  
reference variable object

## 8 链表类

```

public class GenericLinkedList<E> {
    // 内部类 Node, 表示链表中的一个节点
    private class Node<E> {
        E data;           // 节点中存储的数据
        Node<E> next;   // 指向下一个节点的引用

        // 构造函数, 用于创建一个节点
        Node(E data, Node<E> next) {
            this.data = data;
            this.next = next;
        }
    }

    // 构造函数, 用于创建一个空的链表
    public GenericLinkedList() {
        head = null; // 当链表为空时, head 为 null
    }

    // 检查链表是否为空
    public boolean isEmpty() {
        return (head == null); // 当 head 为 null 时, 链表为空
    }

    // 在链表头部添加一个新的节点
    public void addToFront(E newData) {
        head = new Node<E>(newData, head); // 创建一个新节点, 并将其设为链表的头节点
    }

    // 在链表尾部添加一个新的节点
    public void addToRear(E newData) {
        Node<E> node = new Node<E>(newData, null); // 创建一个新节点, next 为 null
        Node<E> current = head;

        if (isEmpty()) { // 如果链表为空, 将 head 指向新节点
            head = node;
        } else {
            // 否则遍历链表, 直到找到最后一个节点
            while (current.next != null) {
                current = current.next;
            }
            // 将最后一个节点的 next 指向新节点
            current.next = node;
        }
    }

    // 返回链表中所有元素的字符串表示
    public String toString() {
        Node<E> current = head; // 从头节点开始遍历
        String result = ""; // 用于存储结果的字符串

        // 遍历链表, 直到没有更多的节点
        while (current != null) {
            result += current.data.toString() + "\n"; // 将每个节点的数据添加
            current = current.next; // 移动到下一个节点
        }
        return result;
    }

    // 检查链表中是否包含某个特定元素
    public boolean contains(E target) {
        if (isEmpty()) { // 如果链表为空, 直接返回 false
            return false;
        }

        boolean found = false; // 用于标记是否找到目标元素
        Node<E> current = head; // 从头节点开始遍历

        // 遍历链表, 直到找到目标元素或到达链表末尾
        while ((current != null) && (!found)) {
            if (target.equals(current.data)) {
                found = true; // 找到目标元素
            } else {
                current = current.next; // 移动到下一个节点
            }
        }
        return found; // 返回是否找到目标元素
    }

    // 主方法, 测试链表的功能
    public static void main(String[] args) {
        GenericLinkedList<String> favBabySongs = new GenericLinkedList<String>();
        favBabySongs.addToFront("Humpty Dumpty"); // 在链表头部添加 "Humpty Dumpty"
        favBabySongs.addToRear("Swing Low Sweet Chariot"); // 在链表尾部添加 "Swing Low Sweet Chariot"
        favBabySongs.addToFront("Itsy Bitsy Spider"); // 在链表头部添加 "Itsy Bitsy Spider"
        favBabySongs.addToRear("Twinkle, Twinkle Little Star"); // 在链表尾部添加 "Twinkle, Twinkle Little Star"
        favBabySongs.addToFront("Wheels on the Bus"); // 在链表头部添加 "Wheels on the Bus"

        // 打印链表中所有元素
        System.out.println(favBabySongs.toString());
        // 检查链表中是否包含 "Humpty Dumpty" 和 "Baby Shark"
        System.out.println(favBabySongs.contains("Humpty Dumpty")); // 应输出 true
        System.out.println(favBabySongs.contains("Baby Shark")); // 应输出 false
    }
}

```

**手写注释:**

- 在链表类中创建一个Node类, 以便能在链表类中调用。
- Node类的构造方法: 定义了head指向第一个节点。
- 头指针head: 实例化了一个Node类的object, 叫head。
- 链表的构造函数: 将head这个reference variable指向一个空的object。
- isEmpty()方法: 若head都是空的, 则表示链表为空。
- addToFront()方法: 看7.1。
- addToRear()方法: 看7.2。
- toString()方法: 把链表中的data打印成这样:

```

public class GenericLinkedList<E> {
    public E removeFromRear() {
        E removedData;
        if (isEmpty()) {
            removedData = null; // 链表为空, return null
        } else if (head.next == null) {
            removedData = head.data; // 链表只有一个element
            head = null; // ① head → [data] ←
        } else {
            Node<E> current = head;
            while (current.next != null) {
                current = current.next;
            }
            removedData = current.next.data; // ② current → [data] ← → [data] ←
            current.next = null; // ③ current.next = null; // ④ current → [data] ← → [null] ←
        }
        return removedData;
    }

    public E removeFromFront() {
        if (isEmpty()) {
            return null;
        }
        E removedData = head.data; // ① head → [data] ← → ...
        head = head.next; // ② head → [data] ← → [null] ←
        return removedData; // return 这个值
    }
}

```

**手写注释:**

- 从链表中移除元素, 并返回element的值的函数。
- 若链表为空, return null。
- 若链表只有一个元素, 将head设为null, 并返回该元素的值。
- 遍历链表, 直到找到最后一个节点, 然后将该节点的next设为null, 并返回该节点的值。
- 从链表头部移除元素, 并返回该元素的值。
- 遍历链表, 直到找到第一个节点, 然后将head设为下一个节点, 并返回该节点的值。

## 9 数组列表 vs 链表 2大区别

### ArrayList的优点和缺点

优点：

1. 快速随机访问：ArrayList 允许通过索引进行快速的随机访问，时间复杂度为  $O(1)$ 。这使得它在频繁读取或查询元素时表现优异。
2. 顺序存储：ArrayList 的底层是数组，存储元素是连续的，这有利于缓存的局部性，可以带来性能提升。
3. 动态扩容：ArrayList 可以自动调整其容量，当容量不足时会自动扩展。

缺点：

1. 插入和删除操作较慢：在 ArrayList 中间插入或删除元素时，需要移动后续元素以保持顺序，时间复杂度为  $O(n)$ 。
2. 内存浪费：由于 ArrayList 的底层是数组，可能会在预留空间时浪费一些内存，特别是在频繁扩展容量时。

### LinkedList的优点和缺点

优点：

1. 插入和删除操作较快：在 LinkedList 中间插入或删除元素仅需要调整相邻节点的引用，时间复杂度为  $O(1)$ ，不需要移动其他元素。
2. 灵活的内存分配：LinkedList 不需要连续的内存空间，每个节点独立分配内存，可以高效利用内存。

缺点：

1. 访问速度慢：LinkedList 访问元素需要从头开始遍历，时间复杂度为  $O(n)$ ，不适合频繁访问的场景。
2. 内存开销大：LinkedList 中每个节点除了存储数据，还需要存储指向前后节点的引用，相比 ArrayList，需要更多的内存空间。

### 总结

- ArrayList 适合频繁访问和查询的场景，但在需要频繁插入或删除时表现较差。
- LinkedList 适合需要频繁插入或删除的场景，但在随机访问时效率较低。

但不需频繁的访问和查询

数组列表和链表的共同缺点：占内存

## 10 递归

没有停止条件的递归最终会因为stackOverflow报错而终止

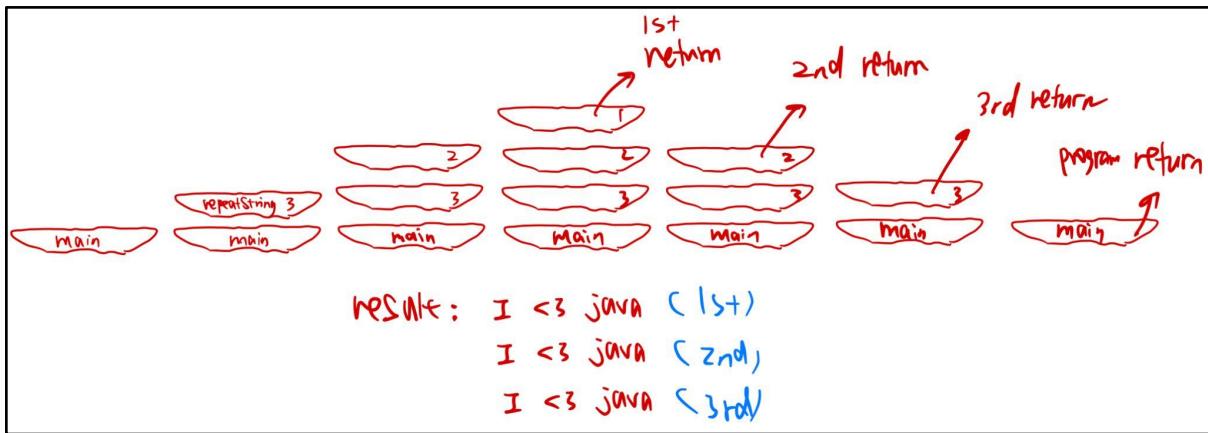
设置一个停止条件，然后每次运行都迫近一点点这个停止条件。最终停止。

图中代码 repeatString("i <3 Java", 1331) 含义：

- repeat1331次，“i <3 Java”
  - 每运行一次，1331-1。当数字变为0，return，结束。

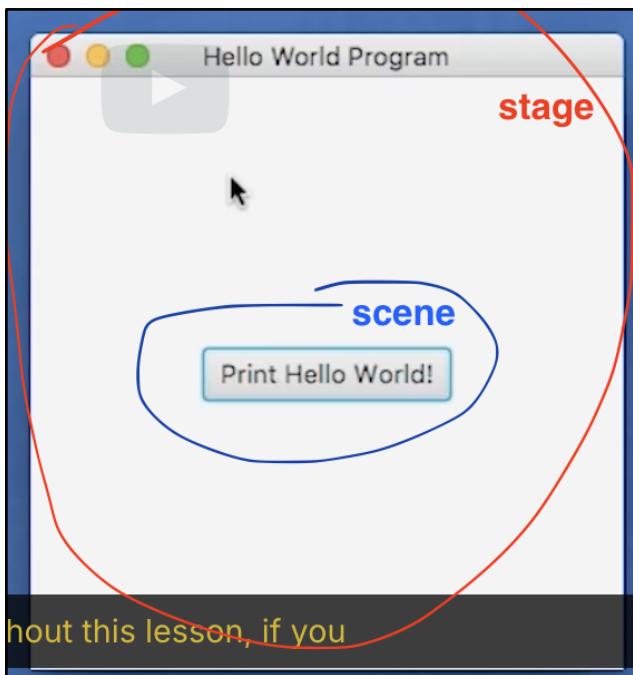
一个简化的，停止条件只有3的例子：

```
public class Repeater {  
  
    public static void main(String[] args) {  
        repeatString("I <3 Java", 3);  
    }  
  
    public static void repeatString(String toRepeat, int times) {  
        if (times <= 0) {  
            return;  
        }  
        System.out.println(toRepeat);  
        repeatString(toRepeat, times - 1);  
    }  
}
```



Module 10: JavaFX

1 javaFX - 一个GUI库



javaFX是java中的一个GUI库

javaFX的简单例子：

```

import javax.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    public void start(Stage mainStage) {
        mainStage.setTitle("Hello World Program");
        Button btn = new Button();
        btn.setText("Print Hello World!");
        btn.setOnAction(new CustomEventHandler());
        StackPane root = new StackPane();
        root.getChildren().add(btn);
        Scene scene = new Scene(root, 300, 300);
        mainStage.setScene(scene);
        mainStage.show();
    }

    private class CustomEventHandler implements EventHandler<ActionEvent> {
        public void handle(ActionEvent event) {
            System.out.println("Hello World!");
        }
    }
}

```

寻找start方法，并执行start

javafx库中application类的method，用于开始一个前端

创建一个stage（即window）这个window的名字

创建button按钮上显示的文字

让button添加到StackPane上

创建一个layout，将button添加到layout/将button添加到StackPane上

创建一个Scene，将此设置为stage的Scene

Scene大小为300x300

```

public class HelloWorld extends Application {
    public void start(Stage mainStage) {
        mainStage.setTitle("Hello World Program");
        Button btn = new Button();
        btn.setText("Print Hello World!");
        btn.setOnAction(new CustomEventHandler());
        StackPane root = new StackPane();
        root.getChildren().add(btn);
        Scene scene = new Scene(root, 300, 300);
        mainStage.setScene(scene);
        mainStage.show();
    }

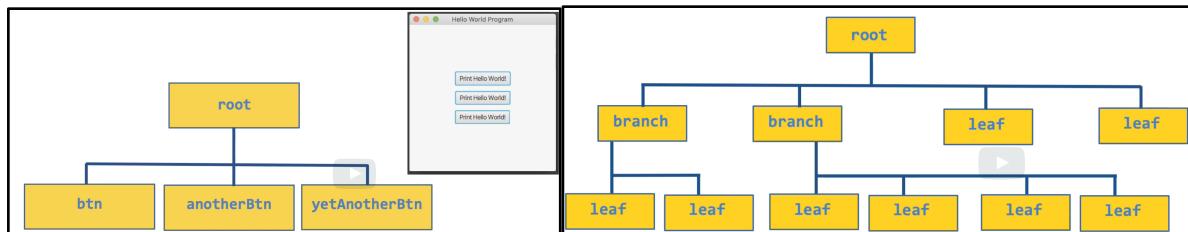
    private class CustomEventHandler implements EventHandler<ActionEvent> {
        public void handle(ActionEvent event) {
            System.out.println("Hello World!");
        }
    }
}

```



- **Layouts:** FlowPane, GridPane, BorderPane, HBox, StackPane, VBox, etc
- **Shapes:** Circle, Rectangle, etc
- **Images:** ImageView
- **Controls:** Button, CheckBox, TextField, Label

有很多种layout ( 横的 , 坚的 , 四宫格的 ) , shapes ( 圆的 , 方的 ) , images ( 图片 ) , controls(例如按钮 , check框 )



( 一个root可以有很多elements。比如这里 , 一个root有三个buttons )

## 2 功能接口 Functional Interface

功能接口 Functional interface - 一个interface中只有一个method , 那这个method叫做功能接口。例如EventHandler这个功能接口 :

```

1 interface EventHandler {
2     void handle(T event);
3 }

```

EventHandler这个接口中只有一个method

```

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World Program");

        Button btn = new Button();
        btn.setText("Print Hello World!");
        btn.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });

        StackPane root = new StackPane();
        root.getChildren().add(btn);

        Scene scene = new Scene(root, 300, 300);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}

private class CustomEventHandler implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hello World!");
    }
}

这是一个工具类，且这个类中只有一个method  
所以也叫“functional interface”工具接口  
备注：CustomEventHandler Class使用了EventHandler这个接口

```

```

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HelloWorldanon extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World Program (Anon)");
        Button btn = new Button();
        btn.setText("Print Hello World!");

        EventHandler<ActionEvent> handler = event -> {
            System.out.println("Hello World!");
        };

        StackPane root = new StackPane();
        root.getChildren().add(btn);

        Scene scene = new Scene(root, 300, 300);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}

public class HelloWorldanonFinal extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World Program (AnonFinal)");

        Button btn = new Button();
        btn.setText("Print Hello World!");

        btn.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });

        StackPane root = new StackPane();
        root.getChildren().add(btn);

        Scene scene = new Scene(root, 300, 300);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}

```

## 2.1 用lambda (->) 表达式来使用功能接口

```

1 @FunctionalInterface
2 interface Greeting {
3     void sayHello(String name);
4 }
5
6 public class GreetingWay implements Greeting {
7     void sayHello(String name) {
8         System.out.println("Hello" + name);
9     }
10}
11
12 public class Main {
13     Run | Debug | Run main | Debug main
14     public static void main(String[] args) {
15         GreetingWay greetingWay = new GreetingWay();
16         greetingWay.sayHello("Jay");
17     }
18 }

```

```

1 @FunctionalInterface
2 interface Greeting {
3     void sayHello(String name);
4 }
5
6 public class Main {
7     Run | Debug | Run main | Debug main
8     public static void main(String[] args) {
9         Greeting greeting = (name) -> System.out.println("Hello " + name);
10        greeting.sayHello(name:"Jay");
11 }

```

引入的参数，就算没有括号（ ）也是可以的

当功能接口作为一个参数时，都不用再instantiate了。直接将Greeting greeting = name-> System.out.println(...)，->的右边的东西放入括号即可。

比如上面的：

```

btn.setOnAction(handler EventHandler<> {
    public void handle(ActionEvent event) {
        System.out.println("Hello World");
    }
})

```

}

## 直接变成

```
btn.setOnAction( (event) -> System.out.println("Hello World") );
```

也可以把括号去掉 : `btn.setOnAction( event -> System.out.println("Hello World") );`

## 3 JavaFX的API

JavaFX属于Oracle , 可以到oracle的JavaFX API看到更多的layout的代码 , 然后调用他们实现自己的网页

**Using JavaFX UI Controls**

This tutorial covers built-in JavaFX UI controls available in the JavaFX API.

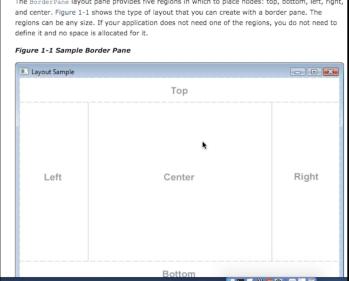
The document contains the following chapters:

- Label
- Button
- Radio Button
- Toggle Button
- Checkbox
- Choice Box
- Text Field
- Password Field
- Scroll Bar
- Scroll Pane
- List View
- Table View
- Tree View
- Tree Table View
- Combo Box

**BorderPane**

The `BorderPane` layout pane provides five regions in which to place nodes: top, bottom, left, right, and center. Figure 1-1 shows the type of layout that you can create with a border pane. The regions are of equal size. If your application does not need one of the regions, you do not need to define it and no space is allocated for it.

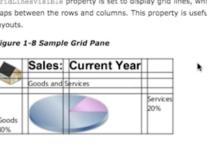
*Figure 1-1 Sample Border Pane*



**GridPane**

The `GridPane` layout pane enables you to create a flexible grid of rows and columns in which to lay out nodes. Nodes can be placed in any cell in the grid and can span cells as needed. A grid pane is useful for creating forms or any layout that is organized in rows and columns. Figure 1-8 shows a grid pane that contains an icon, title, subtitle, text and a pie chart. In this figure, the `gridLinesVisible` property is set to display grid lines, which show the rows and columns and the gaps between the rows and columns. This property is useful for visually debugging your `GridPane` layouts.

*Figure 1-8 Sample Grid Pane*



<https://www.oracle.com/java/technologies/javase/javafx-overview.html>

## 4 用JavaFX库实现温度转换器的前端

```

import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.Button;
import javafx.scene.control.ComboBox;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.control.Label;
import javafx.stage.Stage;

public class TemperatureConverterGUI extends Application {
    public void start(Stage stage) {
        Button convertButton = new Button(); instantiate -> "convert" 按钮
        convertButton.setText("Convert");

        ComboBox<String> pickScaleFrom = new ComboBox();
        ComboBox<String> pickScaleTo = new ComboBox();
        pickScaleFrom.getItems().add("Fahrenheit");
        pickScaleFrom.getItems().add("Celsius");
        // or use: pickScaleFrom.getItems().addAll("Fahrenheit", "Celsius");

        pickScaleTo.getItems().add("Fahrenheit");
        pickScaleTo.getItems().add("Celsius");
        // or use: pickScaleTo.getItems().addAll("Fahrenheit", "Celsius");

        pickScaleFrom.getSelectionModel().selectFirst(); 先选则了 "From" 才能选 "To"
        pickScaleTo.getSelectionModel().selectLast();

        Label from = new Label("From:");
        Label to = new Label("To:");

        TextField userInput = new TextField();
        Label inputValue = new Label("Input value: ");
        Label result = new Label(); instantiate 一个输入框 instantiate 一个 result 标签 ValueInput 中的值

        convertButton.setOnAction(event -> {
            String temperatureString = userInput.getCharacters().toString();
            try {
                double temperature = Double.parseDouble(temperatureString);
                String scaleFrom = pickScaleFrom.getValue();
                String scaleTo = pickScaleTo.getValue();
                double conversionResult = convert(scaleFrom, scaleTo, temperature);
                result.setText(String.format("%.2f", conversionResult));
            } catch (NumberFormatException e) {
                Alert a = new Alert(Alert.AlertType.ERROR);
                a.setTitle("Error");
                a.setHeaderText("Invalid Temperature");
                a.setContentText("That is not a valid temperature.");
                a.showAndWait();
            }
        });

        HBox input = new HBox();
        input.setAlignment(Pos.CENTER);
        input.getChildren().addAll(inputValue, userInput); 前面, 将 userInput 打包成 "input" 这个 HBOX
        input.getChildren().addAll(userInput);
        // or use: input.getChildren().addAll(inputValue, userInput);

        HBox scales = new HBox();
        scales.setAlignment(Pos.CENTER);
        scales.setSpacing(10);
        scales.getChildren().add(from);
        scales.getChildren().add(pickScaleFrom);
        scales.getChildren().add(to);
        scales.getChildren().add(pickScaleTo);
        // or use: scales.getChildren().addAll(from, pickScaleFrom, to, pickScaleTo);
    }

    VBox root = new VBox();
    root.setAlignment(Pos.CENTER);
    root.setSpacing(10);
    root.getChildren().add(input);
    root.getChildren().add(scales);
    root.getChildren().add(convertButton);
    root.getChildren().add(result);
    // or use: root.getChildren().addAll(input, scales, convertButton, result);

    Scene scene = new Scene(root, 400, 400);
    stage.setTitle("Temperature Converter");
    stage.setScene(scene);
    stage.show();
}

private double convert(String from, String to, double value) { 后端方法
    double converted = 0;
    if (from.equals(to)) {
        converted = value;
    } else if (from.charAt(0) == 'F') {
        converted = (value - 32) * (5.0 / 9);
    } else {
        converted = value * (9.0 / 5) + 32;
    }
    return converted;
}

```

```
javac  
--module-path /path/to/javafx-sdk-11.0.2/lib  
--add-modules javafx.controls HelloWorld.java
```

```
mvn archetype:generate  
-DgroupId=com.example  
-DartifactId=HelloWorldJavaFX  
-DarchetypeArtifactId=maven-archetype-quickstart  
-DinteractiveMode=false
```

\$ Java Programming Problems:

- 必须要在file中建一个src，然后把java文件放src中，vsc的代码才不会红色曲线报错
- 每个 .java 文件最多只能有一个 public class
  - 除了public类之外，你可以在同一个文件中定义其他类，但这些类不能是 public。这些类的名称不需要与文件名相同。

所以，如果你在 `aaa.java` 文件中定义了一个公共类 `aaa`，你可以在同一个文件中定义其他非公共类，如 `bbb`，而不需要将它们分成多个文件。例如：

```
java  
// 文件名: aaa.java  
public class aaa {  
    // 这是一个公共类，文件名必须与类名相同  
}  
  
// 你可以在同一个文件中定义其他类，但它们不能是public  
class bbb {  
    // 这是一个非公共类，文件名不需要与它匹配  
}  
  
class ccc {  
    // 你也可以定义其他非公共类  
}
```

- 看Java API文档的时候，那些method从页面的Method Summary开始看

OVERVIEW	MODULE	PACKAGE	CLASS	USE	TREE	DEPRECATED	INDEX	HELP
PREV CLASS	NEXT CLASS		FRAMES	NO FRAMES	ALL CLASSES	SUMMARY	NESTED	FIELD   CONSTR   METHOD
			DETAIL	FIELD	CONSTR	METHOD		
<b>Method Summary</b>								
<b>All Methods</b> <b>Instance Methods</b> <b>Concrete Methods</b>								
Modifier and Type	Method					Description		
void	close()					Closes this scanner.		
Pattern	delimiter()					Returns the Pattern this Scanner is currently using to match delimiters.		
Stream<MatchResult>	findAll(String patString)					Returns a stream of match results that match the provided pattern string.		
Stream<MatchResult>	findAll(Pattern pattern)					Returns a stream of match results from this scanner.		

- indentation在Java中不要求，在Python中要求