Reminder

args 返回的都是 String[]类型的，我们要进行转换
args is array of String

返回类型前以及**类或接口声明前的修饰符顺序**并不影响代码的功能

向 method 传入参数的时候，低精度的 primitive 会被自动转成高精度，但是高精度不能转成低精度；可以传入 child reference type 作为替代

泛型不能不适用 primitive type，参考 ArrayList

使用带有 throws 的 method，必须用 catch

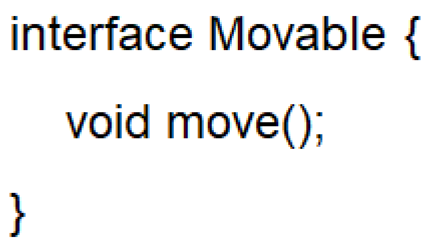
静态方法不能调用非静态方法

静态变量只能在类中声明，不能在方法中声明

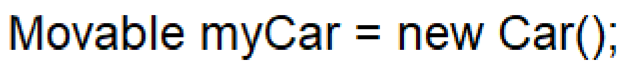
接口中的变量必须在定义时赋值，因为接口中的变量默认是 static final，也就是说它们是常量，必须在声明时初始化。

在看 inherit method 的时候，注意他们的 Modifier

interface 虽然不能实例化，但是可以用来声明



```
interface Movable {
    void move();
}
```



```
Movable myCar = new Car();
```

这里 Car 是 class name

但是在 for 循环中不要紧，因为都是 for block 运行完后再运行 i
++i 先加 1，再返回新的值。
i++ 先使用 i 的当前值，之后再加 1

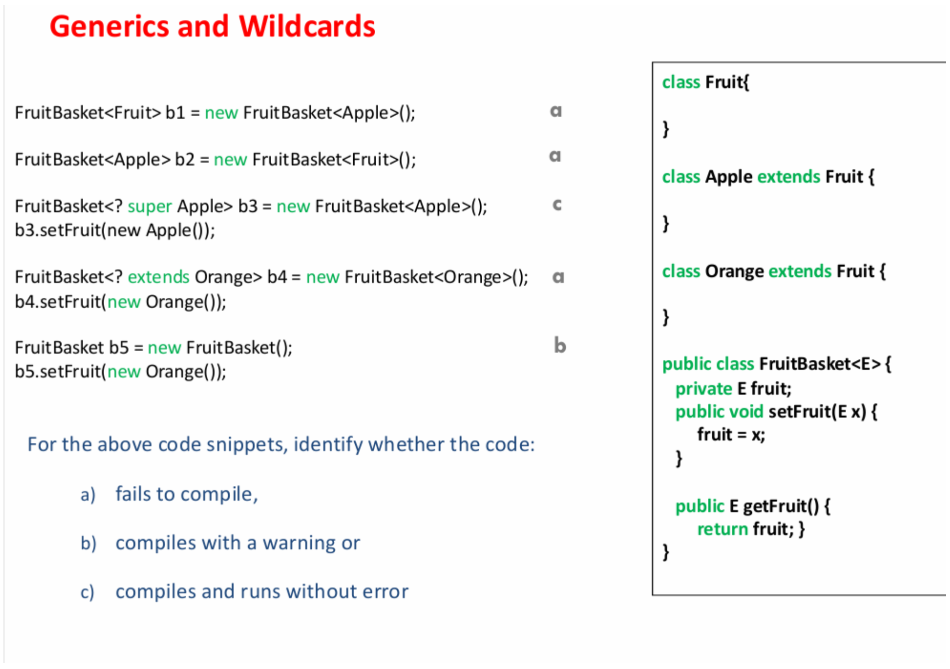
primitive type are all lower case

method “throws” 要放在参数后面

String: length()
array: length
arraylist: size()
map: size()

import java.util.*;
import java.io.*;
import org.junit.*

Can we assign inherited types with generics? NO



Generics and Wildcards

```
FruitBasket<Fruit> b1 = new FruitBasket<Apple>();
```
a

```
FruitBasket<Apple> b2 = new FruitBasket<Fruit>();
```
a

```
FruitBasket<? super Apple> b3 = new FruitBasket<Apple>();
b3.setFruit(new Apple());
```
c

```
FruitBasket<? extends Orange> b4 = new FruitBasket<Orange>();
b4.setFruit(new Orange());
```
a

```
FruitBasket b5 = new FruitBasket();
b5.setFruit(new Orange());
```
b

For the above code snippets, identify whether the code:

a) fails to compile,

b) compiles with a warning or

c) compiles and runs without error

```
class Fruit{

}

class Apple extends Fruit {

}

class Orange extends Fruit {

}

public class FruitBasket<E> {
  private E fruit;
  public void setFruit(E x) {
     fruit = x;
  }

  public E getFruit() {
     return fruit; }
}
```

上界 ? extends T 可读，不可写
下界 ? super T 可写，不可读

overload 只要方法名相同，signature 不同就行

interface extends interface

String x [] = {“1”, “2”}; 是合法的

我们可以在 override 的时候返回父类型的子类型

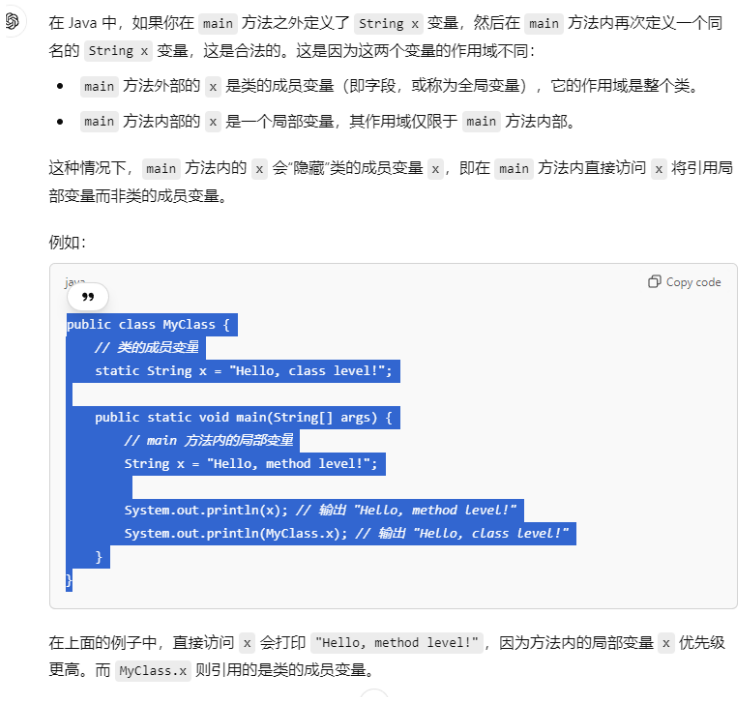
java

```java
class Animal {
    public Animal getAnimal() {
        return new Animal();
    }
}

class Dog extends Animal {
    // 返回类型是 Animal 的子类 Dog，这是合法的
    @Override
    public Dog getAnimal() {
        return new Dog();
    }
}
```

General problem

```java
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> originalList = new ArrayList<>();
        originalList.add("Hello");
        originalList.add("World");

        // 创建一个新的 ArrayList，它是 originalList 的副本
        ArrayList<String> copiedList = new ArrayList<>(originalList);

        System.out.println("Original List: " + originalList);  // 输出 [Hello, World]
        System.out.println("Copied List: " + copiedList);      // 输出 [Hello, World]
    }
}
```

上面是浅拷贝



在 Java 中，如果你在 `main` 方法之外定义了 `String x` 变量，然后在 `main` 方法内再次定义一个同名的 `String x` 变量，这是合法的。这是因为这两个变量的作用域不同：

- `main` 方法外部的 `x` 是类的成员变量（即字段，或称为全局变量），它的作用域是整个类。
- `main` 方法内部的 `x` 是一个局部变量，其作用域仅限于 `main` 方法内部。

这种情况下，`main` 方法内的 `x` 会"隐藏"类的成员变量 `x`，即在 `main` 方法内直接访问 `x` 将引用局部变量而非类的成员变量。

例如：

```java
public class MyClass {
    // 类的成员变量
    static String x = "Hello, class level!";

    public static void main(String[] args) {
        // main 方法内的局部变量
        String x = "Hello, method level!";

        System.out.println(x); // 输出 "Hello, method level!"
        System.out.println(MyClass.x); // 输出 "Hello, class level!"
    }
}
```

在上面的例子中，直接访问 `x` 会打印 `"Hello, method level!"`，因为方法内的局部变量 `x` 优先级更高。而 `MyClass.x` 则引用的是类的成员变量。

java

```java
public void exampleMethod() {
    int x = 10; // 外部定义的 x

    for (int i = 0; i < 5; i++) {
        int x = i * 2; // 错误! 不能在同一方法作用域中定义重复变量名
        System.out.println(x);
    }
}
```

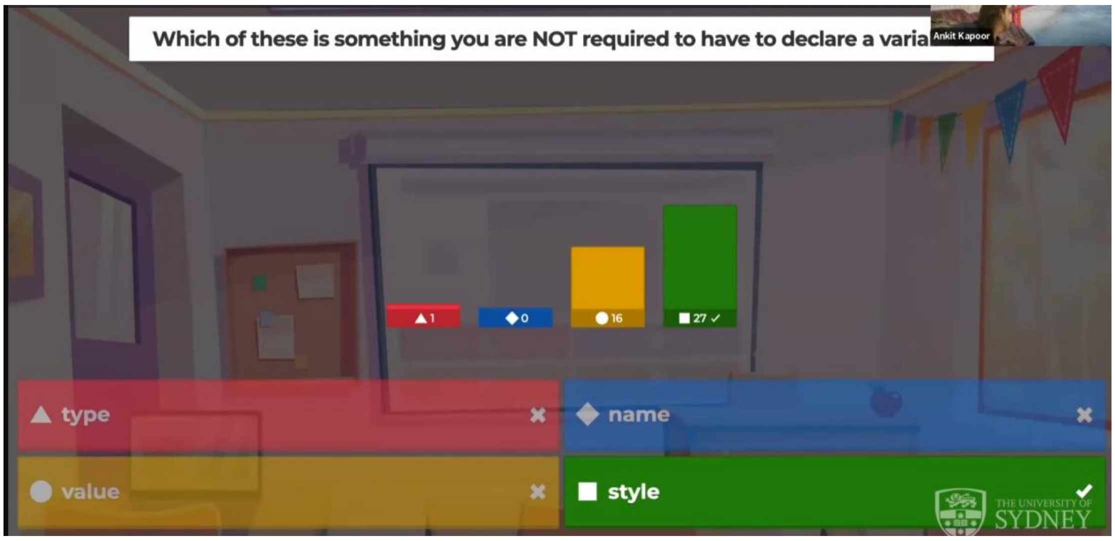
在上面的代码中，`for` 循环内的 `int x = i * 2;` 会导致编译错误，因为同一方法中已经存在一个名为 `x` 的变量，即使它在 `for` 循环之外。即便作用域不同，Java 仍不允许在同一个方法的不同代码块中重复定义同名变量。

不过，如果 `for` 循环外的 `x` 没有提前定义，代码就可以正常运行，例如：

java

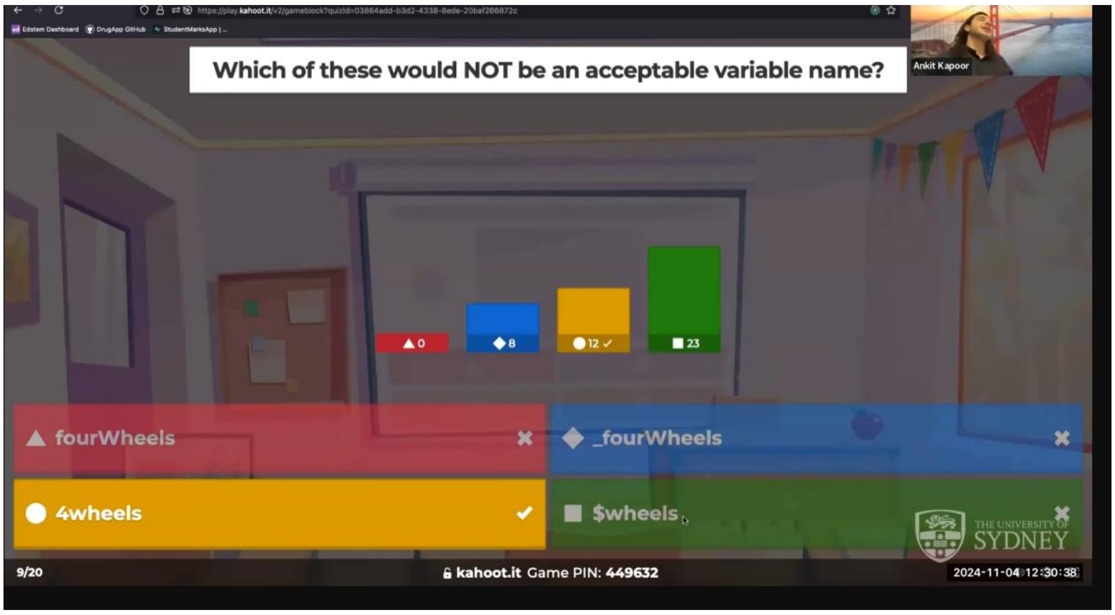
```java
public void exampleMethod() {
    for (int i = 0; i < 5; i++) {
        int x = i * 2; // 合法，作用域仅限于循环内部
        System.out.println(x);
    }

    int x = 10; // 合法，作用域在循环外
    System.out.println(x);
}
```

这种情况下，`for` 循环中的 `x` 和循环外的 `x` 是两个不同的变量，因为它们的作用域互不重叠。

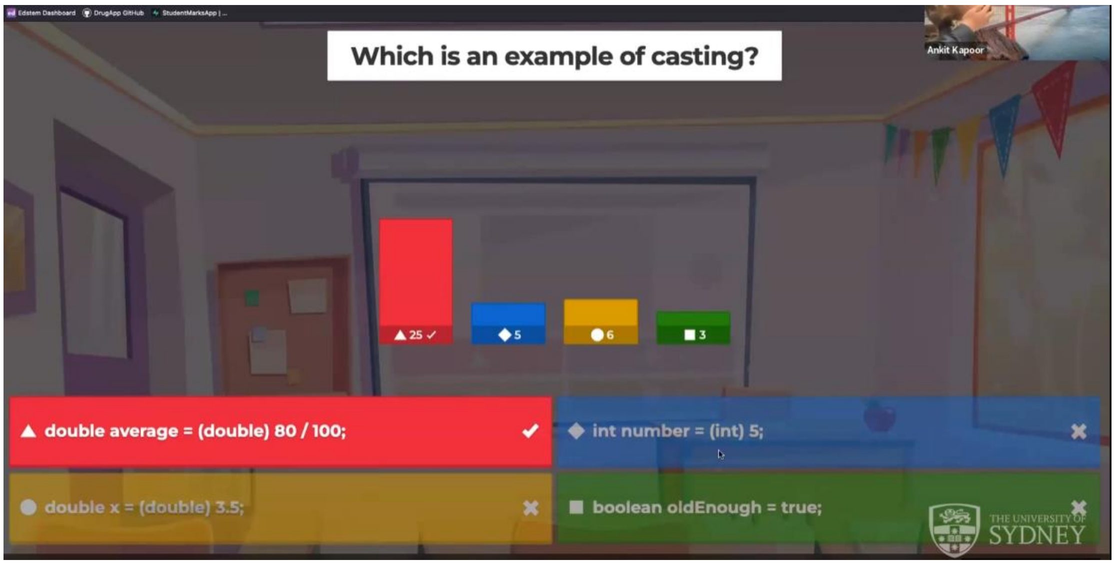

Which of these is something you are NOT required to have to declare a varia

▲ 1 ◆ 0 ● 16 ■ 27 ✓

▲ type ✖
◆ name ✖
● value ✖
■ style ✓

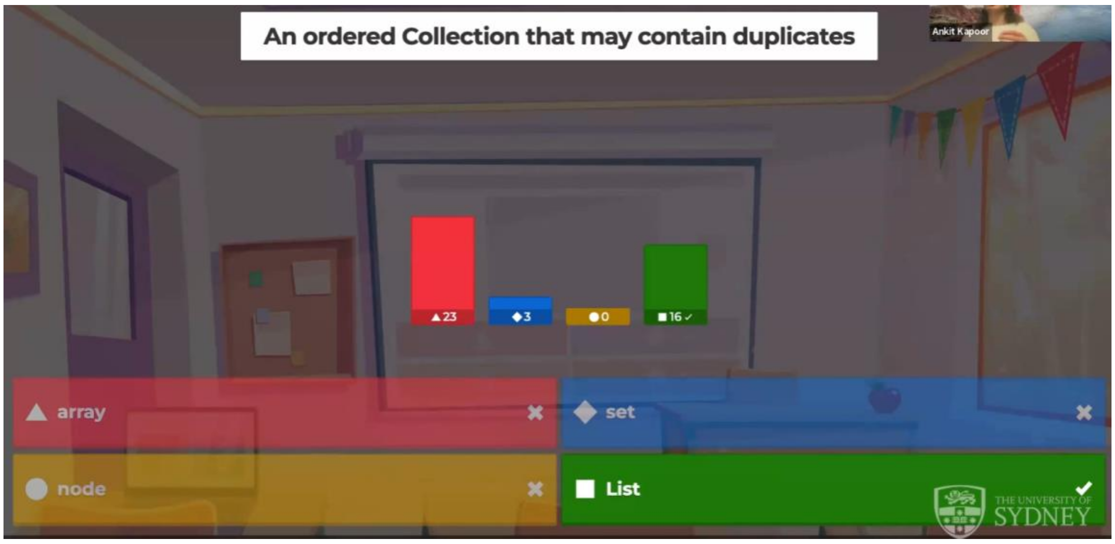

Which of these would NOT be an acceptable variable name?

▲ fourWheels ✖
◆ _fourWheels ✖
● 4wheels ✓
■ $wheels ✖


Which is an example of casting?

▲ double average = (double) 80 / 100; ✓
◆ int number = (int) 5; ✖
● double x = (double) 3.5; ✖
■ boolean oldEnough = true; ✖

80/100 return 0 since both 80 and 100 are int.

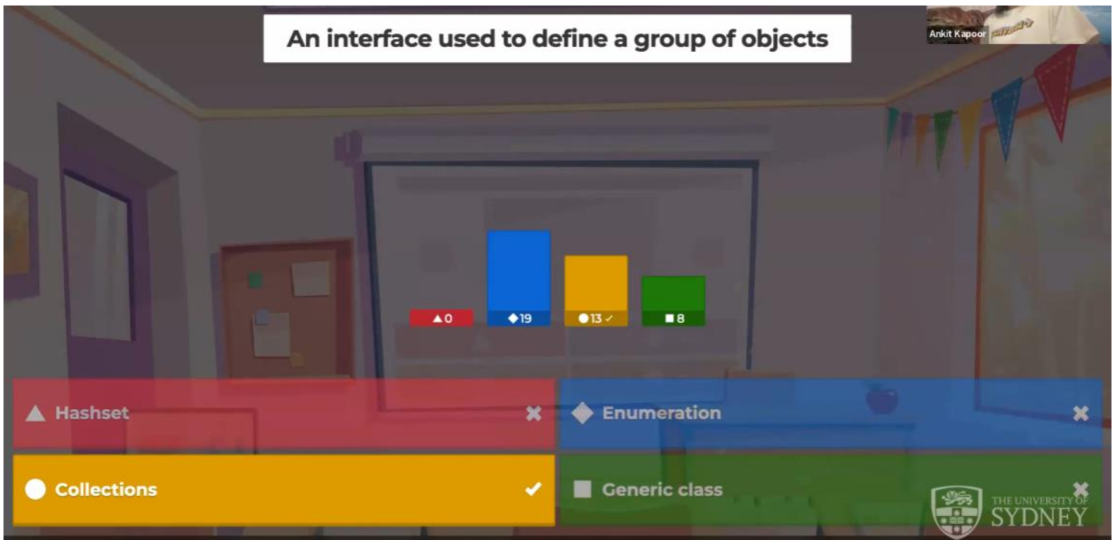

An ordered Collection that may contain duplicates

▲ array ✖
◆ set ✖
● node ✖
■ List ✓

这里 Collection 指的是一个框架，array is not a Collection

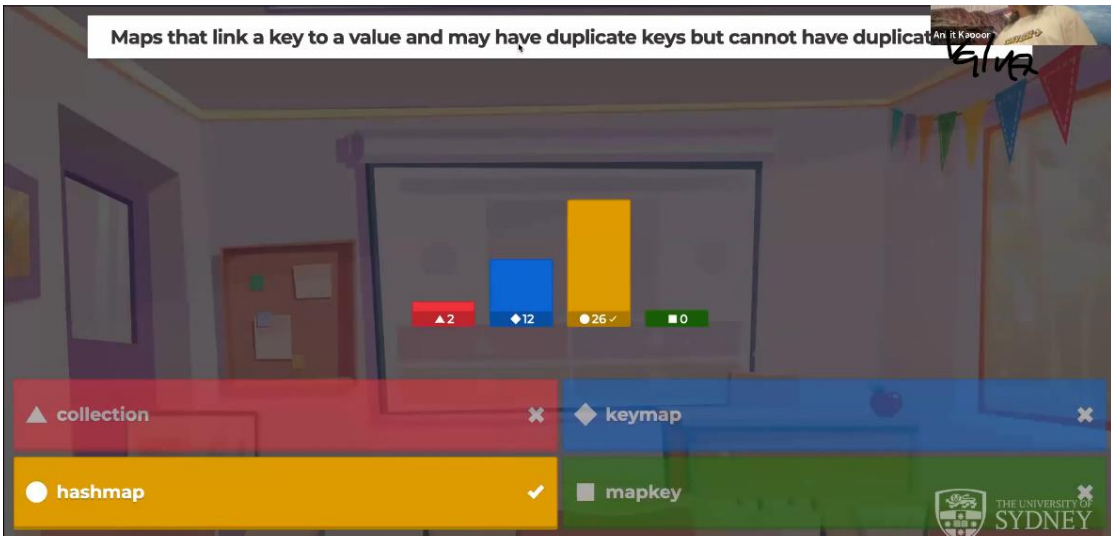
Collection 的主要子接口

- **List**：有序的集合，允许重复元素。
    - 实现类：
        - ArrayList
        - LinkedList
        - Vector（包括其子类 Stack）
- **Set**：不允许重复元素的集合。
    - 实现类：
        - HashSet
        - LinkedHashSet
        - TreeSet（有序集合，按自然顺序或自定义顺序排序）
- **Queue**：按特定顺序处理元素的集合，通常用于实现队列数据结构。
    - 实现类：
        - LinkedList（实现了 Queue 接口）
        - PriorityQueue（优先级队列）
        - ArrayDeque（双端队列）


An interface used to define a group of objects

▲ Hashset ✖
◆ Enumeration ✖
● Collections ✔
■ Generic class ✖

Collection 是 interface



Maps that link a key to a value and may have duplicate keys but cannot have duplicat

▲2 ◆12 ●26 ✓ ■0

▲ collection ✖

◆ keymap ✖

● hashmap ✓

■ mapkey ✖

参考 hash collide

在 Java 中，private 修饰的属性**不会被子类继承**，也就是说，子类无法直接访问父类中的 private 属性。

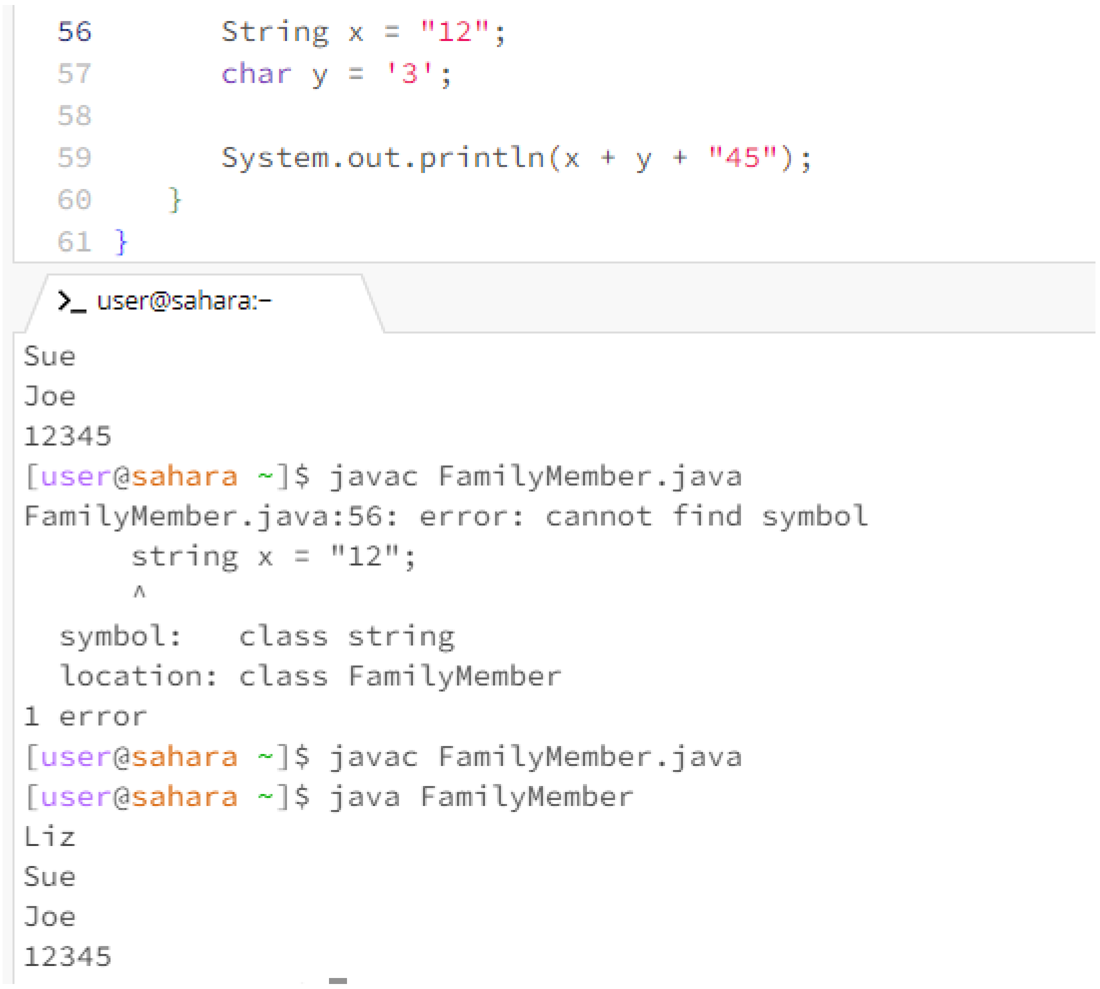
带 hash 的数据结构是无序的

父类构造方法的隐式调用：
- 当创建子类对象时，Java 会在子类的构造方法中自动添加对父类构造方法的调用，即 super()。
- 如果父类有无参构造方法，Java 会隐式调用这个无参构造方法。

父类构造方法带参数的情况：
- 如果父类**没有无参构造方法**，但有带参构造方法，则 Java 编译器无法自动调用，因为找不到无参构造。
- 在这种情况下，**子类构造方法必须显式调用带参的父类构造方法**，并提供正确的参数，否则会导致编译错误。

如果一个类**没有定义任何构造方法**，编译器会自动为这个类生成一个**无参构造方法**。这个默认的无参构造方法可以让类被实例化而不需要传递参数

Char 和 string 合并不需要先把 char 转换成 string



```
        String x = "12";
        char y = '3';

        System.out.println(x + y + "45");
    }
}
```

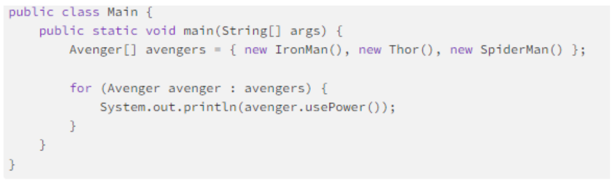
>_ user@sahara:~

```
Sue
Joe
12345
[user@sahara ~]$ javac FamilyMember.java
FamilyMember.java:56: error: cannot find symbol
        string x = "12";
        ^
  symbol:   class string
  location: class FamilyMember
1 error
[user@sahara ~]$ javac FamilyMember.java
[user@sahara ~]$ java FamilyMember
Liz
Sue
Joe
12345
```

public static void main(String [] args)

Math.PI for pi

In the main method, create an array of Avenger objects containing instances of IronMan, Thor and SpiderMan. Use a loop to call usePower() on each Avenger.我需要先创建一个 main class



```
public class Main {
    public static void main(String[] args) {
        Avenger[] avengers = { new IronMan(), new Thor(), new SpiderMan() };

        for (Avenger avenger : avengers) {
            System.out.println(avenger.usePower());
        }
    }
}
```

Console Input：通常称为 **Standard Input (stdin)**。它指的是从控制台获取输入，用户在程序运行过程中直接从控制台或终端输入数据。例如，Java 中的 Scanner 类可以用于从 System.in 获取控制台输入。
Command Input：通常称为 **Command-Line Arguments** 或 **Command-Line Input**。它指的是在启动程序时通过命令行传入的参数，这些参数作为程序启动时的输入。例如，Java 中的 args 参数数组就是从命令行获取的输入。

Question 2

✓ SUBMITTED 1 point

Which of the below are **methods** of the class `System`?

- [ ] err
- [x] console
- [x] getProperty
- [x] lineSeparator

Submitted

Question

SAVED 1 poir

Select all **valid** overloading methods.

- [x]
```java
public void change(){...}
public void change(int mode){...}
```

- [ ]
```java
public String getState(){...}
public int getState(){...}
```

- [x]
```java
public int calc(int a, float b, int c){...}
public int calc(int a, int b, float c){...}
```

- [x]
```java
public float doOp(){...}
private float doOp(int flags){...}
```

Question

✓ SUBMITTED 1 point

Suppose you have a generic and non-generic class:

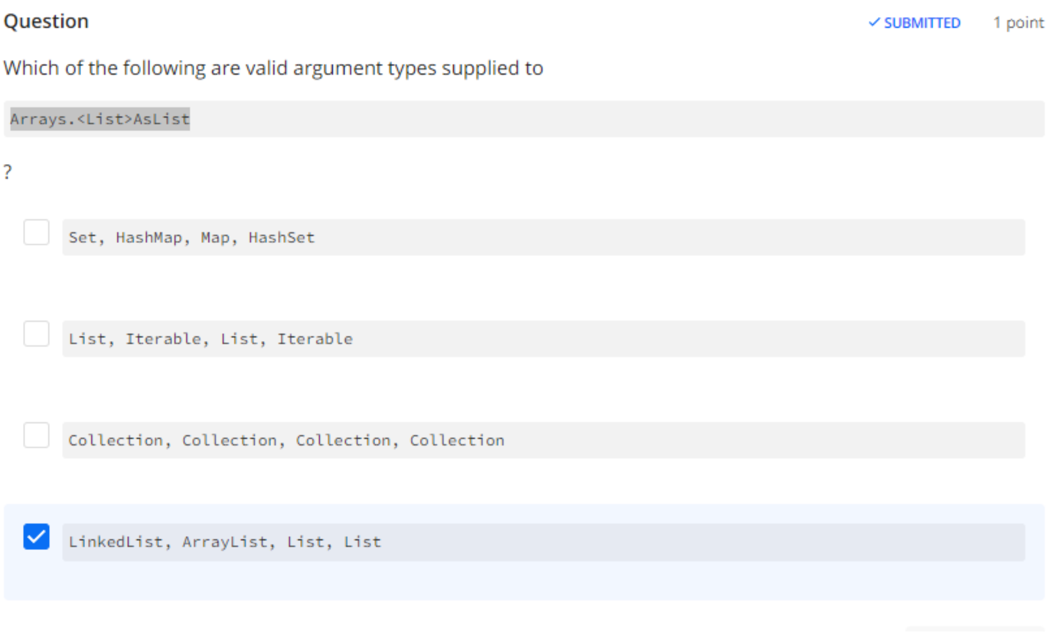
```java
public class One;
public class Two<T>;
```

Which lines of code are valid?

- [ ] JAVA
```java
One o = new One<String>();
Two t = new Two<String>();
```

- [x] JAVA ✓
```java
public static boolean comp(One o, Two<Integer> t){
    return o.equals(t);
}
```

- [ ] JAVA
```java
One o = new One<String>();
One a = o;
Two<String> t = new Two<String>();
Two<Object> w = t;
```



Question ✓ SUBMITTED 1 point

Which of the following are valid argument types supplied to

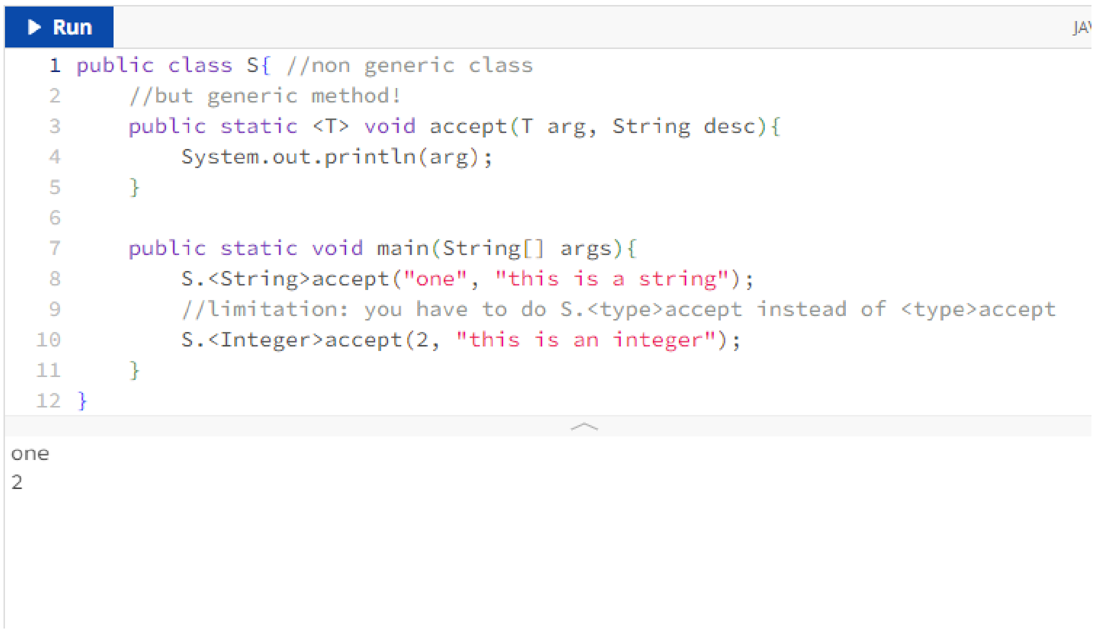
```
Arrays.<List>AsList
```

?

- [ ] `Set, HashMap, Map, HashSet`
- [ ] `List, Iterable, List, Iterable`
- [ ] `Collection, Collection, Collection, Collection`
- [✓ `LinkedList, ArrayList, List, List`]

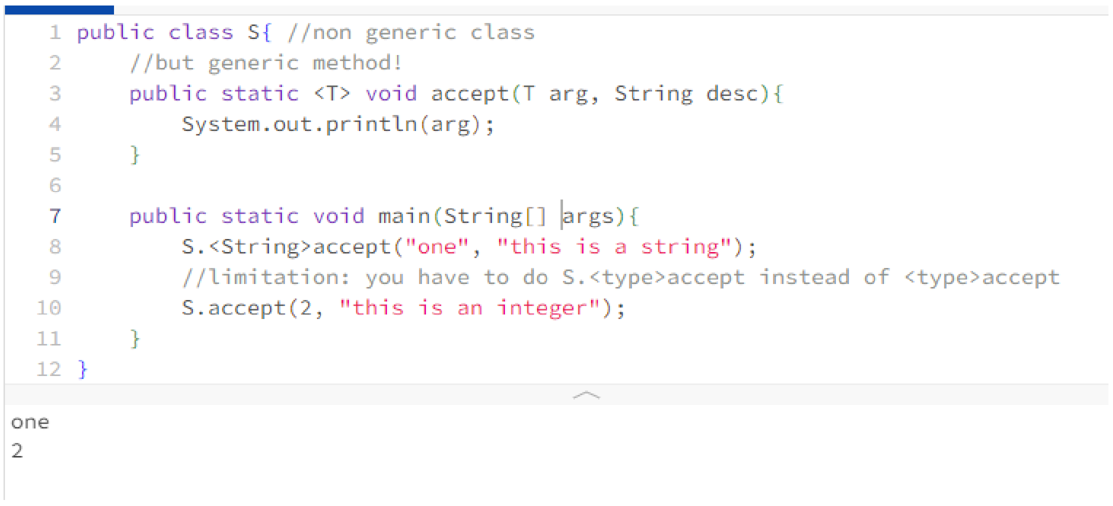
```
public static <T> List<T> asList(T... a)
```

`Arrays.asList()` 是 Java 中 `Arrays` 类的一个静态方法，它将一个数组转换为 `List`。但是，`Arrays.asList()` 是一个泛型方法，因此需要指定数组元素的类型。



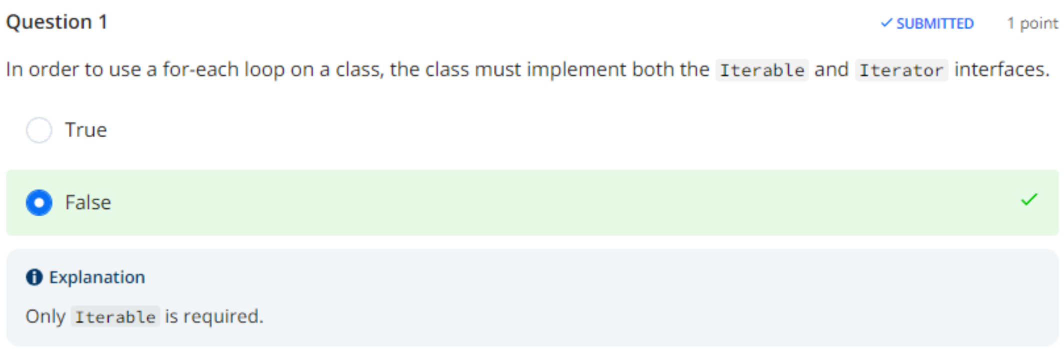
```
public class S{ //non generic class
    //but generic method!
    public static <T> void accept(T arg, String desc){
        System.out.println(arg);
    }

    public static void main(String[] args){
        S.<String>accept("one", "this is a string");
        //limitation: you have to do S.<type>accept instead of <type>accept
        S.<Integer>accept(2, "this is an integer");
    }
}
```

```
one
2
```



```
public class S{ //non generic class
    //but generic method!
    public static <T> void accept(T arg, String desc){
        System.out.println(arg);
    }

    public static void main(String[] args){
        S.<String>accept("one", "this is a string");
        //limitation: you have to do S.<type>accept instead of <type>accept
        S.accept(2, "this is an integer");
    }
}
```

```
one
2
```



Question 1

✓ SUBMITTED 1 point

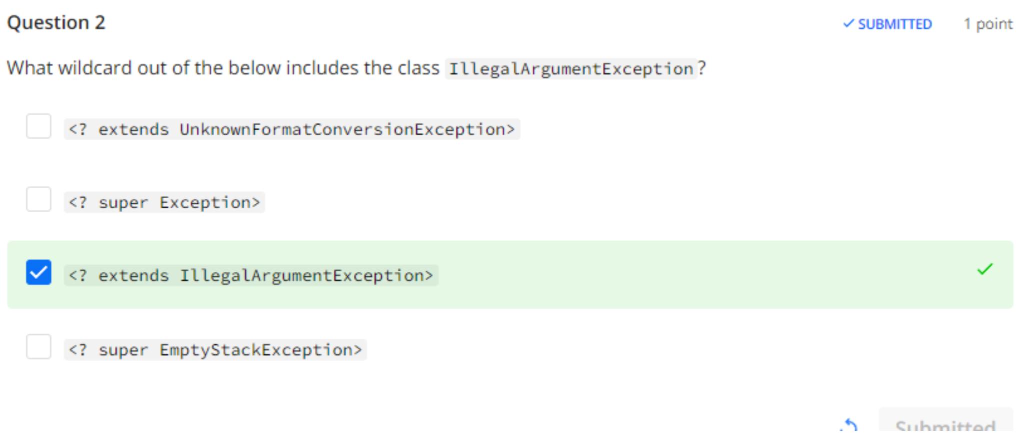
In order to use a for-each loop on a class, the class must implement both the `Iterable` and `Iterator` interfaces.

True [✓ False] ✓

Explanation

Only `Iterable` is required.

我们 extends iterable，但是 iterable 包含 iterator abstract method



Question 2

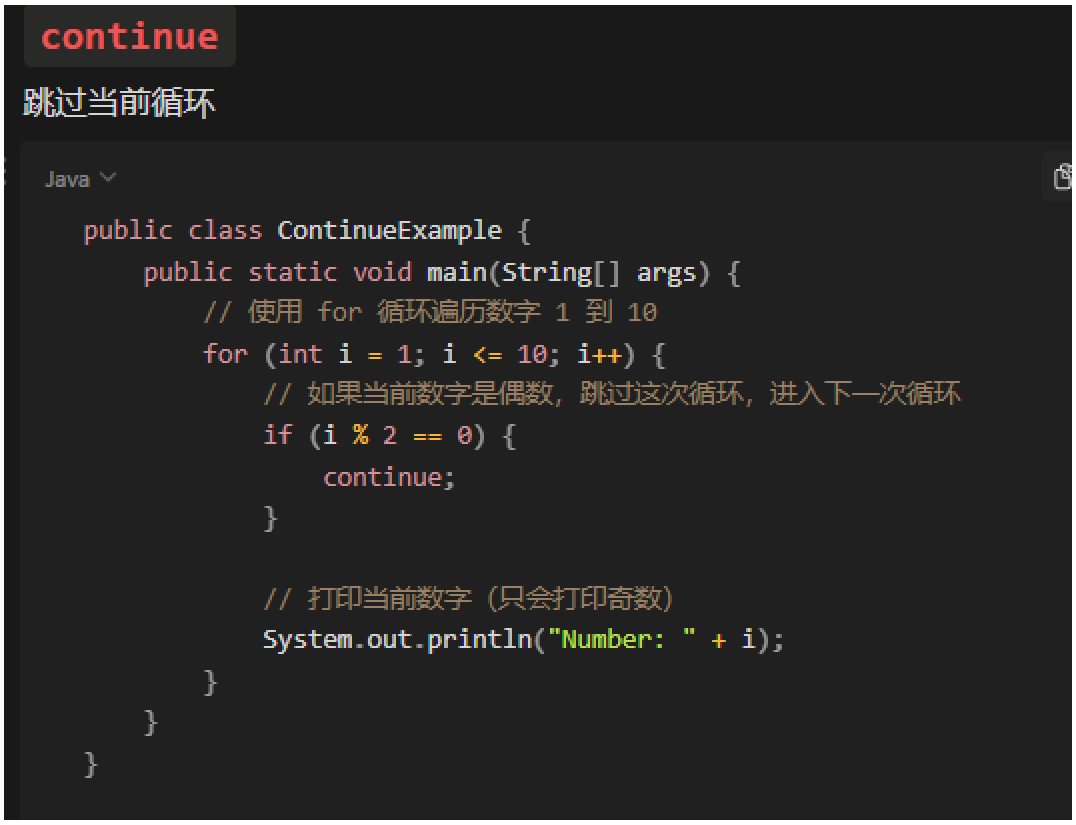
✓ SUBMITTED 1 point

What wildcard out of the below includes the class `IllegalArgumentException`?

- ☐ `<? extends UnknownFormatConversionException>`
- ☐ `<? super Exception>`
- ☑ `<? extends IllegalArgumentException>` ✓
- ☐ `<? super EmptyStackException>`

<? extends Exception>

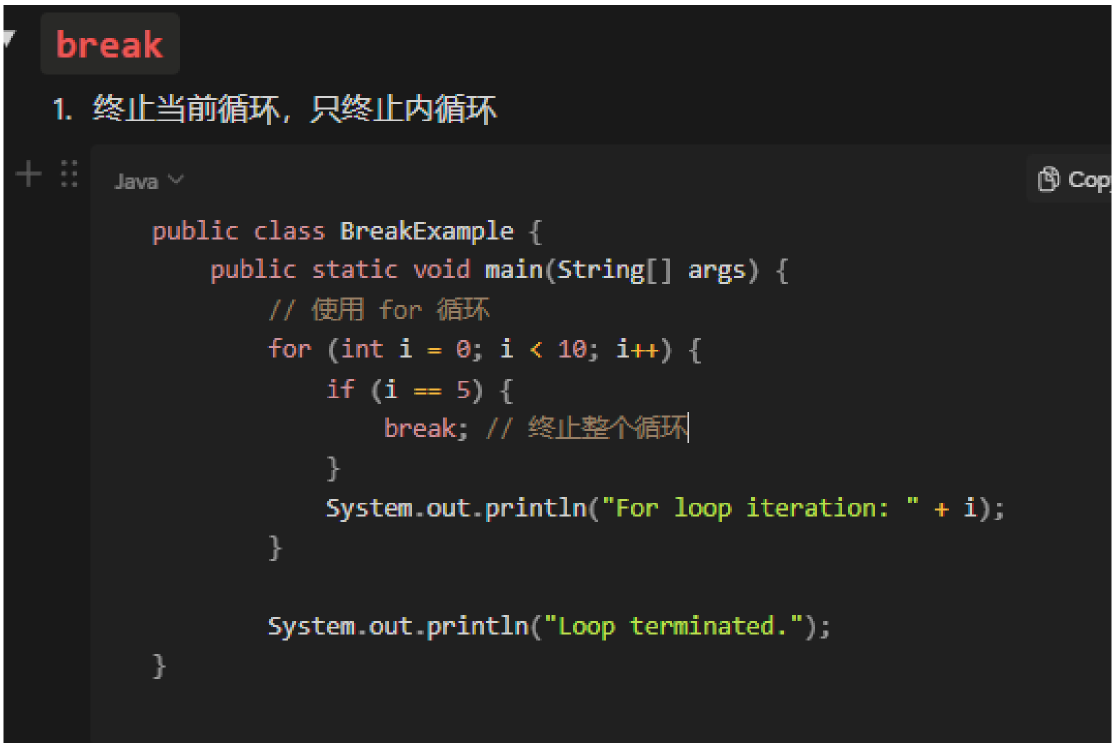
- This wildcard means any class that extends Exception, which also includes IllegalArgumentException, since IllegalArgumentException is a subclass of RuntimeException, and RuntimeException extends Exception.
- Therefore, ? extends Exception would also match IllegalArgumentException.



continue

跳过当前循环

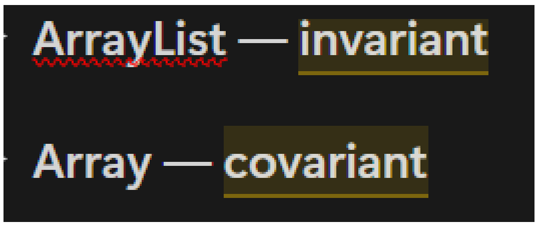
```java
public class ContinueExample {
    public static void main(String[] args) {
        // 使用 for 循环遍历数字 1 到 10
        for (int i = 1; i <= 10; i++) {
            // 如果当前数字是偶数，跳过这次循环，进入下一次循环
            if (i % 2 == 0) {
                continue;
            }

            // 打印当前数字（只会打印奇数）
            System.out.println("Number: " + i);
        }
    }
}
```



break

1. 终止当前循环，只终止内循环

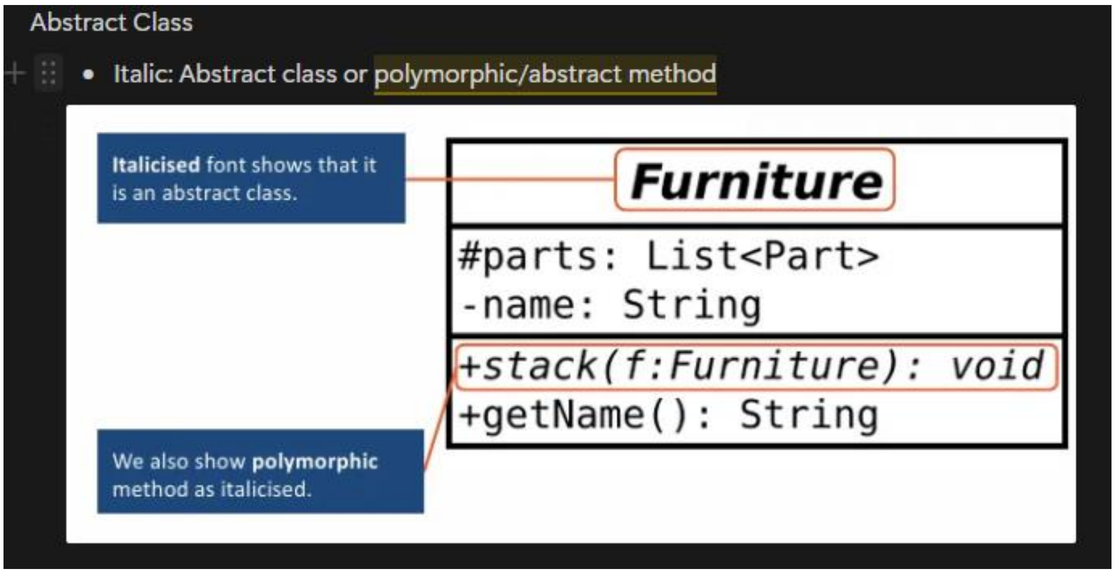
```java
public class BreakExample {
    public static void main(String[] args) {
        // 使用 for 循环
        for (int i = 0; i < 10; i++) {
            if (i == 5) {
                break; // 终止整个循环
            }
            System.out.println("For loop iteration: " + i);
        }

        System.out.println("Loop terminated.");
}
```



ArrayList — invariant

Array — covariant

UML



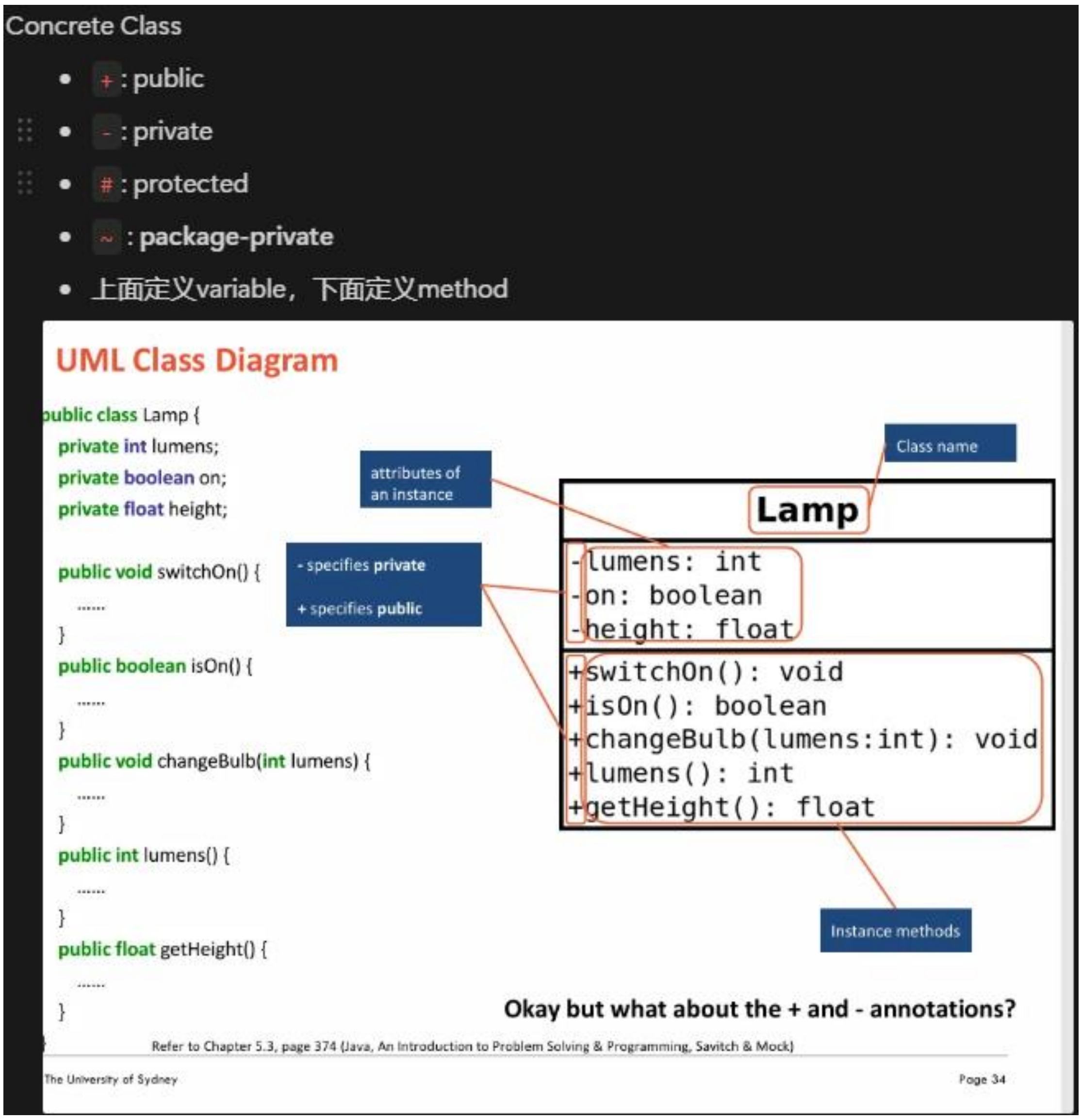
Abstract Class

- Italic: Abstract class or polymorphic/abstract method

Italicised font shows that it is an abstract class.

We also show **polymorphic** method as italicised.

```
Furniture
#parts: List<Part>
-name: String
+stack(f:Furniture): void
+getName(): String
```



Concrete Class

- + : public
- - : private
- # : protected
- ~ : package-private
- 上面定义variable，下面定义method

UML Class Diagram

```
public class Lamp {
  private int lumens;
  private boolean on;
  private float height;

  public void switchOn() {
    ......
  }
  public boolean isOn() {
    ......
  }
  public void changeBulb(int lumens) {
    ......
  }
  public int lumens() {
    ......
  }
  public float getHeight() {
    ......
  }
}
```

```
Lamp
-lumens: int
-on: boolean
-height: float
+switchOn(): void
+isOn(): boolean
+changeBulb(lumens:int): void
+lumens(): int
+getHeight(): float
```

Okay but what about the + and - annotations?

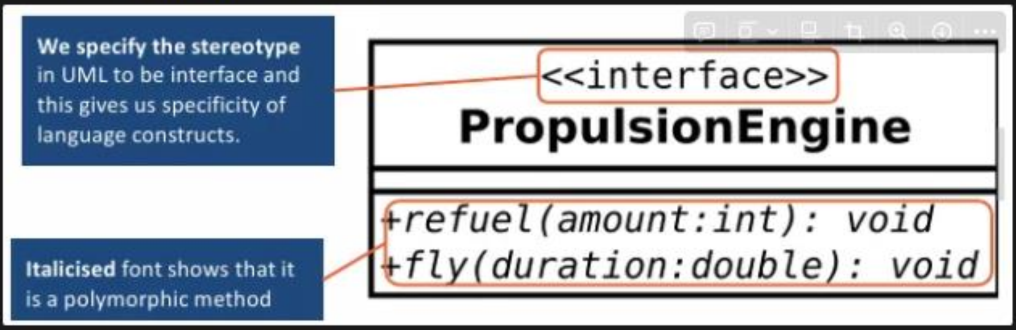
Refer to Chapter 5.3, page 374 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

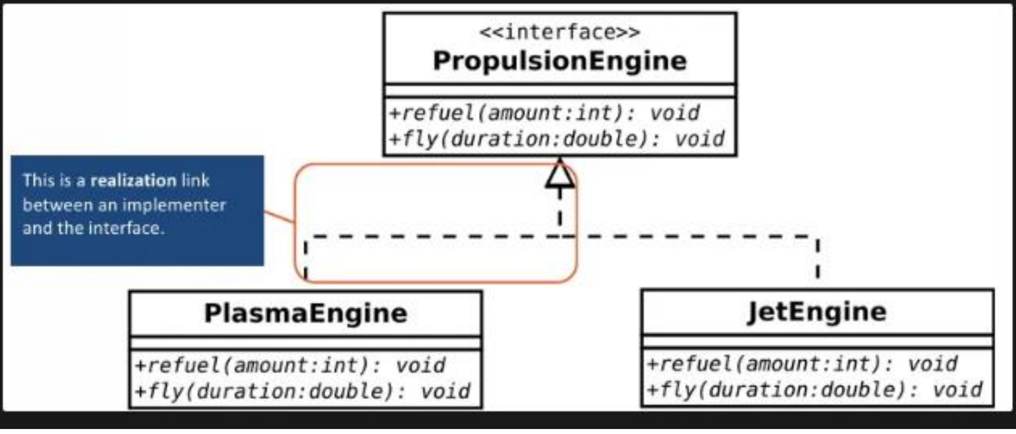
The University of Sydney

Page 34

Interface

- have interface label at the top


We specify the stereotype in UML to be interface and this gives us specificity of language constructs.
<<interface>>
PropulsionEngine
+refuel(amount:int): void
+fly(duration:double): void
Italicised font shows that it is a polymorphic method


<<interface>>
PropulsionEngine
+refuel(amount:int): void
+fly(duration:double): void
This is a **realization** link between an implementer and the interface.
PlasmaEngine
+refuel(amount:int): void
+fly(duration:double): void
JetEngine
+refuel(amount:int): void
+fly(duration:double): void

Data Type

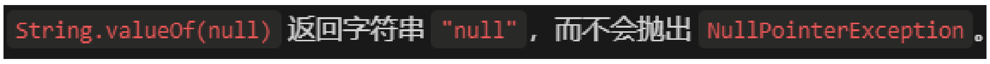
String

- String 类的 contains() 方法用于检查一个字符串是否包含另一个子字符串。该方法返回一个布尔值，表示目标字符串是否包含指定的子字符串。

```
boolean containsJava = sentence.contains("Java");
```

- String.valueOf（）用于转换 primitive type 为 sting

```
System.out.println("Title: " + String.valueOf(getTitle()));
```

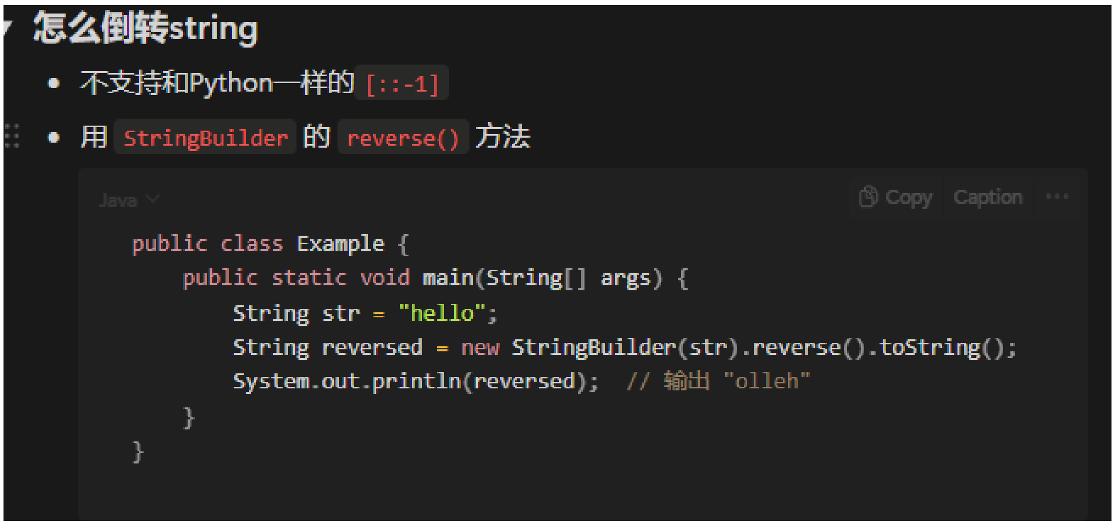

String.valueOf(null) 返回字符串 "null"，而不会抛出 NullPointerException。

- String 可以直接和 primitive type 进行 concatenate

```
String mix = "Value: " + 10 + 5;  // 结果为 "Value: 105"，而不是 "Value: 15"
```

String mix = "Value: " + (10 + 5);   // 结果为 "Value: 15"

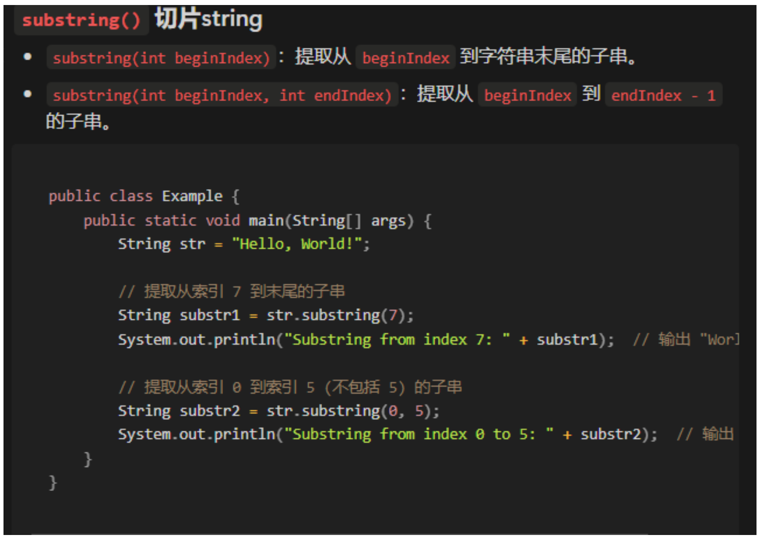
- String 不支持 for loop，但是可以使用 StringBuilder 里面的 reverse()方法


怎么倒转string

- 不支持和Python一样的 [::-1]
- 用 StringBuilder 的 reverse() 方法

```java
public class Example {
    public static void main(String[] args) {
        String str = "hello";
        String reversed = new StringBuilder(str).reverse().toString();
        System.out.println(reversed);  // 输出 "olleh"
    }
}
```

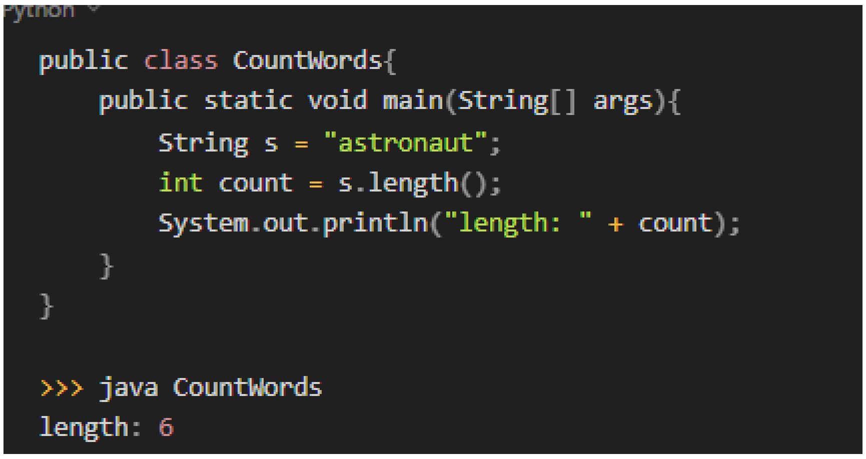
- String 切片只能用 substring()


substring() 切片string

- substring(int beginIndex)：提取从 beginIndex 到字符串末尾的子串。
- substring(int beginIndex, int endIndex)：提取从 beginIndex 到 endIndex - 1 的子串。

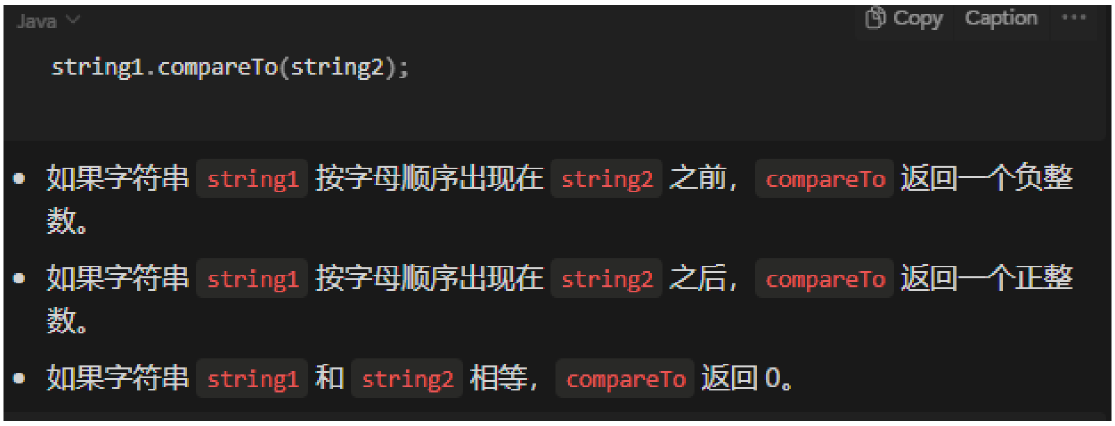
```java
public class Example {
    public static void main(String[] args) {
        String str = "Hello, World!";

        // 提取从索引 7 到末尾的子串
        String substr1 = str.substring(7);
        System.out.println("Substring from index 7: " + substr1);  // 输出 "Wor

        // 提取从索引 0 到索引 5 (不包括 5) 的子串
        String substr2 = str.substring(0, 5);
        System.out.println("Substring from index 0 to 5: " + substr2);  // 输出
    }
}
```

- 获取长度需要用 length()



```
public class CountWords{
    public static void main(String[] args){
        String s = "astronaut";
        int count = s.length();
        System.out.println("length: " + count);
    }
}

>>> java CountWords
length: 6
```

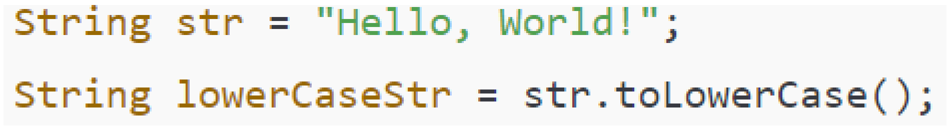
- compareTo()



```
string1.compareTo(string2);
```

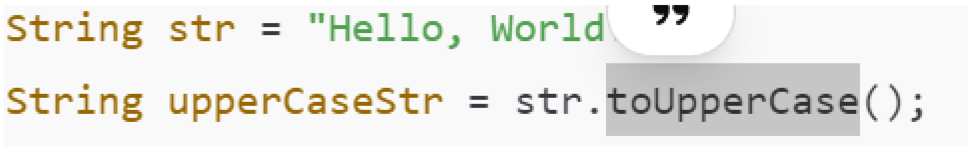
- 如果字符串 `string1` 按字母顺序出现在 `string2` 之前，`compareTo` 返回一个负整数。
- 如果字符串 `string1` 按字母顺序出现在 `string2` 之后，`compareTo` 返回一个正整数。
- 如果字符串 `string1` 和 `string2` 相等，`compareTo` 返回 0。

- toLowerCase()



```
String str = "Hello, World!";
String lowerCaseStr = str.toLowerCase();
```

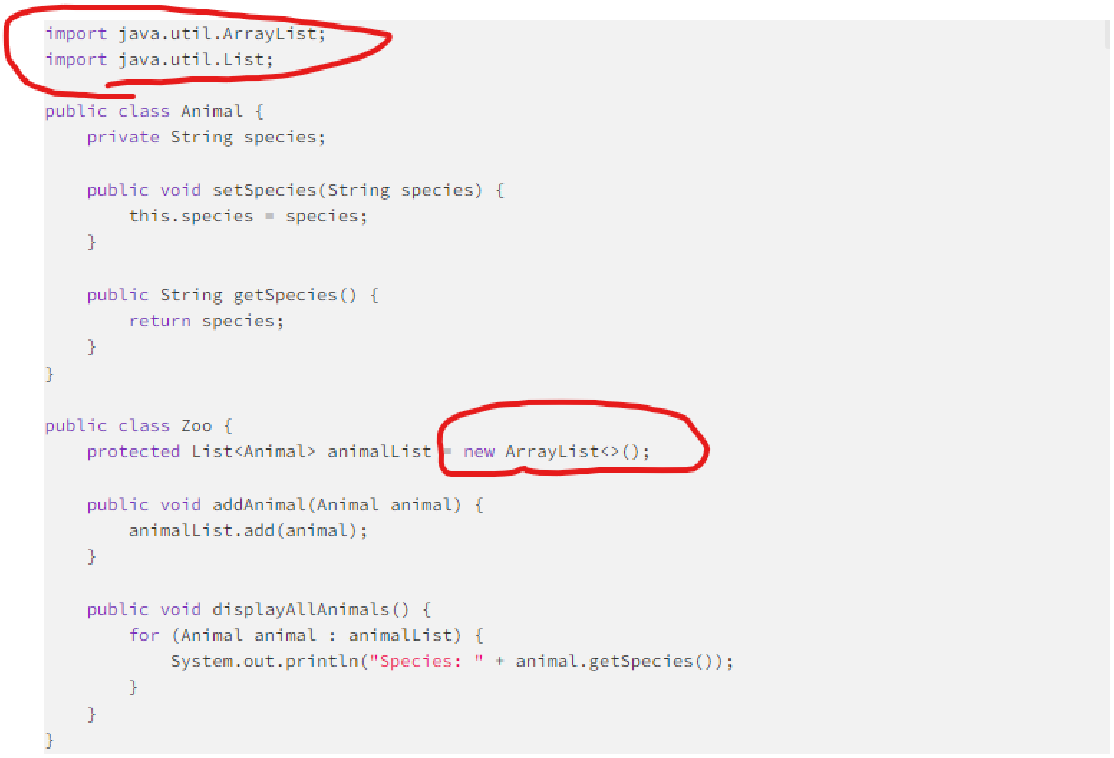
- toUpperCase()



```
String str = "Hello, World
String upperCaseStr = str.toUpperCase();
```

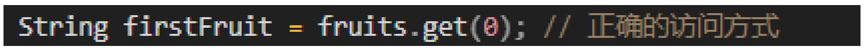
ArrayList（Dynamic Array）

- 需要 import java.util.Arrays;
- 我需要在使用 ArrayList 的时候先 import，其次我在声明 arrayList 的时候要直接使用 new，否则不会分配实际的存储空间，会造成 nullpointer exception



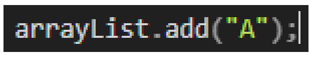
```java
import java.util.ArrayList;
import java.util.List;

public class Animal {
    private String species;

    public void setSpecies(String species) {
        this.species = species;
    }

    public String getSpecies() {
        return species;
    }
}

public class Zoo {
    protected List<Animal> animalList = new ArrayList<>();

    public void addAnimal(Animal animal) {
        animalList.add(animal);
    }

    public void displayAllAnimals() {
        for (Animal animal : animalList) {
            System.out.println("Species: " + animal.getSpecies());
        }
    }
}
```

- 只能使用 get()，不能和 Dynamically allocated Array 一样用 Array[idx]



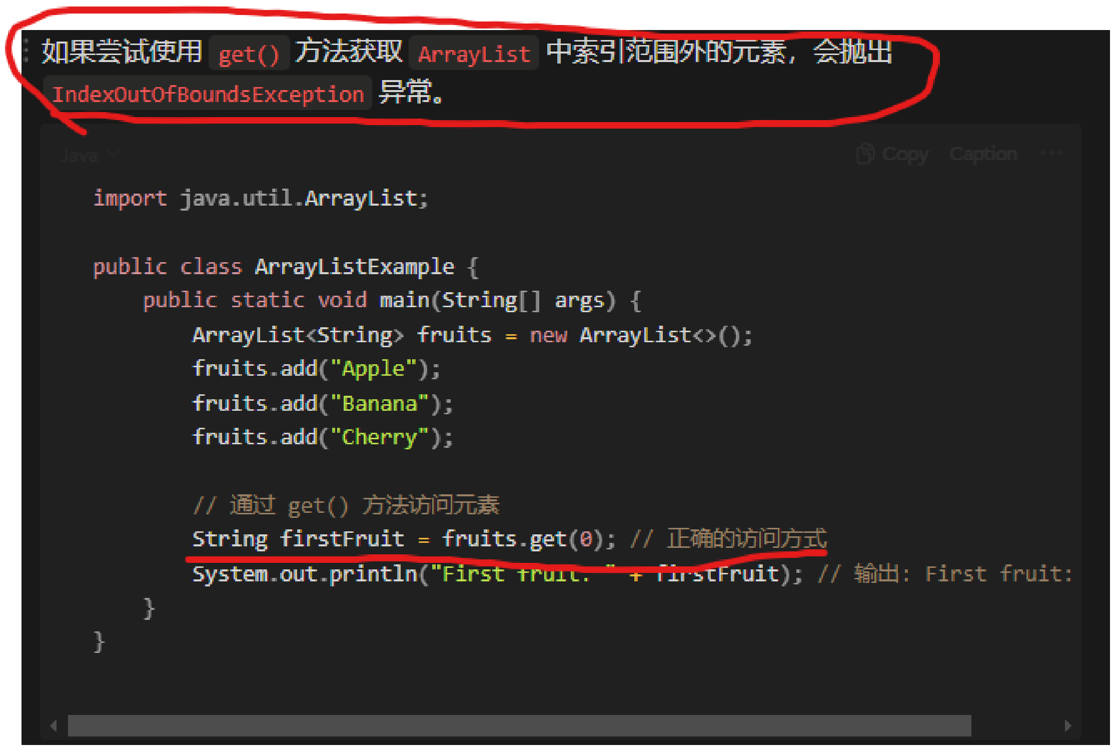
```java
String firstFruit = fruits.get(0); // 正确的访问方式
```

- add()



```java
arrayList.add("A");
```

- get()


如果尝试使用 `get()` 方法获取 `ArrayList` 中索引范围外的元素，会抛出 `IndexOutOfBoundsException` 异常。

```java
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");

        // 通过 get() 方法访问元素
        String firstFruit = fruits.get(0); // 正确的访问方式
        System.out.println("First fruit: " + firstFruit); // 输出: First fruit:
    }
}
```

- set()

```java
// 替换第二个元素 (索引为 1)
int idx = 1; // 要替换的元素索引
String newFruit = "Blueberry"; // 新的元素
fruits.set(idx, newFruit); // 替换元素
```

- remove()：可以用 idx 或者 object

```java
// 弹出最后一个元素，即使用 remove() 实现 pop()
String poppedElement = list.remove(list.size() - 1);
```

```java
// 假设 current 是我们要移除的 Line 对象的引用
Line current = line1;

// 从 lines 中移除 current 指向的对象
lines.remove(current);
```

- size(): 和 Dynamically allocated Array 不同，这里没有 length attribute

```java
size = list.size();
```

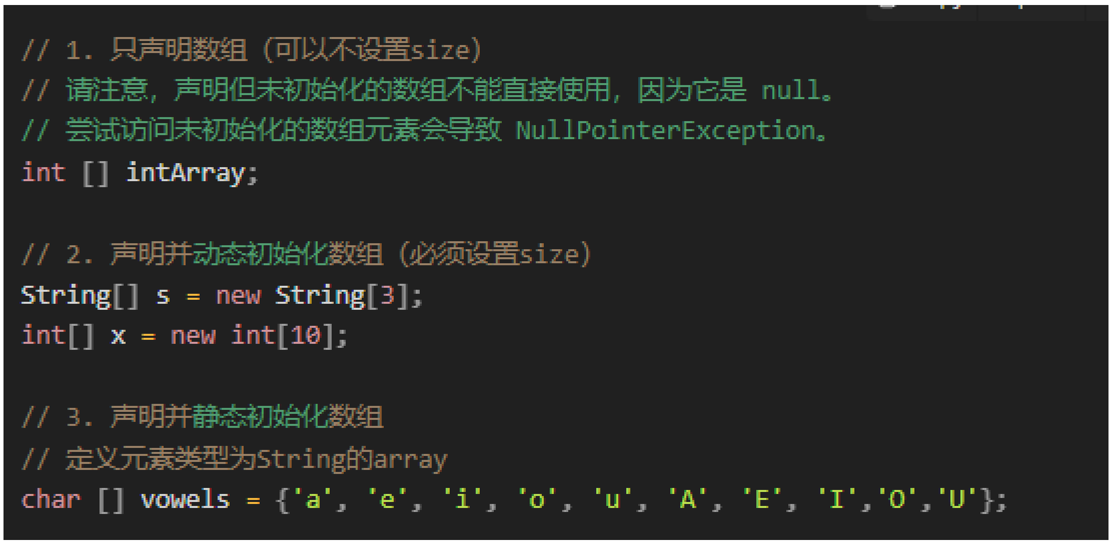
- 打印和普通的 Dynamically allocated Array 不同

```java
// 使用 System.out.println() 打印 ArrayList
System.out.println("ArrayList: " + list);
```

- 用 isEmpty()判断是否是空 arraylist

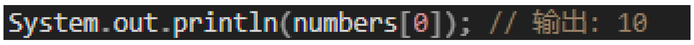
Array(Dynamically allocated Array)

- import java.util.Arrays;
- Initial



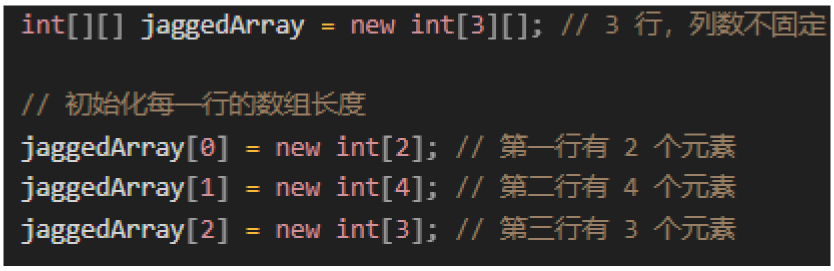
```
// 1. 只声明数组（可以不设置size）
// 请注意，声明但未初始化的数组不能直接使用，因为它是 null。
// 尝试访问未初始化的数组元素会导致 NullPointerException。
int [] intArray;

// 2. 声明并动态初始化数组（必须设置size）
String[] s = new String[3];
int[] x = new int[10];

// 3. 声明并静态初始化数组
// 定义元素类型为String的array
char [] vowels = {'a', 'e', 'i', 'o', 'u', 'A', 'E', 'I','O','U'};
```

- 支持 Array[idx]



```
System.out.println(numbers[0]); // 输出: 10
```

- Jagged array



```
int[][] jaggedArray = new int[3][]; // 3 行，列数不固定

// 初始化每一行的数组长度
jaggedArray[0] = new int[2]; // 第一行有 2 个元素
jaggedArray[1] = new int[4]; // 第二行有 4 个元素
jaggedArray[2] = new int[3]; // 第三行有 3 个元素
```

- Default value

1. 基本数据类型数组的默认值

- **整数类型**（`byte`, `short`, `int`, `long`）：默认值为 `0`。
- **浮点类型**（`float`, `double`）：默认值为 `0.0`。
- **字符类型**（`char`）：默认值为 `'\u0000'`（即 ASCII 值为 0 的字符）。
- **布尔类型**（`boolean`）：默认值为 `false`。

2. 引用类型数组的默认值

- **对象类型数组**：默认值为 `null`，即数组中的每个元素会被初始化为 `null`（对于非基本数据类型的数组，引用会指向 `null`）。

- 打印 array：需要用到 Arrays.toString(A)，光用 toString 是没用的

```
// 打印array的某个元素
String[] x = {"123"};
System.out.println(x[0]);
123

// 打印整个array
System.out.println(Arrays.toString(s)); //[null, null, null]


public class Main {
    public static void main(String[] args) {
        String[] array = {"Apple", "Banana", "Cherry"};
        System.out.println(array); // 输出类似: [Ljava.lang.String;@7f31245a
    }
}


// 打印数组对象的哈希码
System.out.println(x.toString());
```

- Array Slice

```
// 指定切片的起始和结束位置（结束位置不包括）
int start = 2; // 从索引 2 开始
int end = 7;   // 到索引 7 结束（不包括 7）

// 使用 Arrays.copyOfRange 方法进行切片
int[] slicedArray = Arrays.copyOfRange(originalArray, start, end);
```

- Length

```
int[] intArray = [1, 2, 3];
int len = intArray.length;// 3

// 获取数组的最后一个元素
int intArray = intArray[intArray.length - 1]; // 3
```

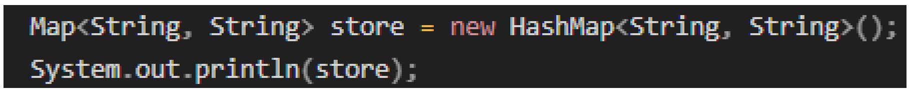
- null 和 Empty array 的区别

```
public static void main(String[] args) {
    int[] a={};
    if (a == null)
        System.out.println("a is null");

    int[] b=null;
    if (b == null)
        System.out.println("b is null");
}

>>> java xxx
b is null
```

- Array 不能通过+和另外一个 array 合并

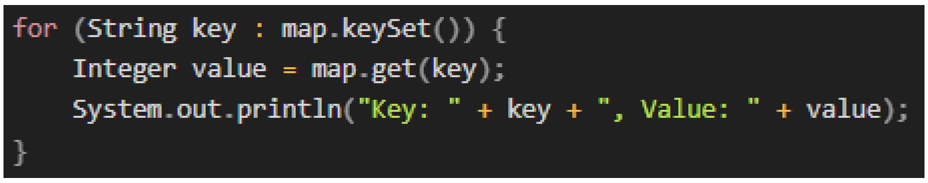
Map

- import java.util.*;
- 打印和 arraylist 类似



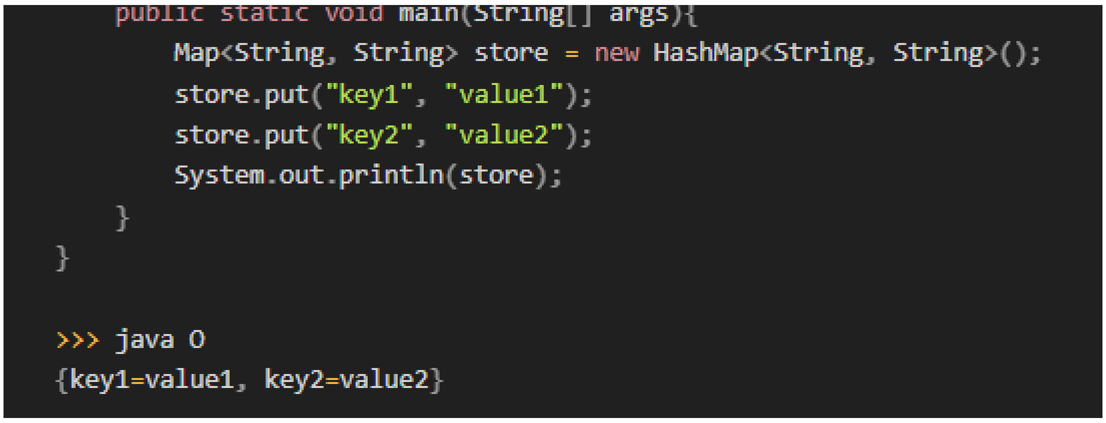
```
Map<String, String> store = new HashMap<String, String>();
System.out.println(store);
```

- keySet()遍历 map



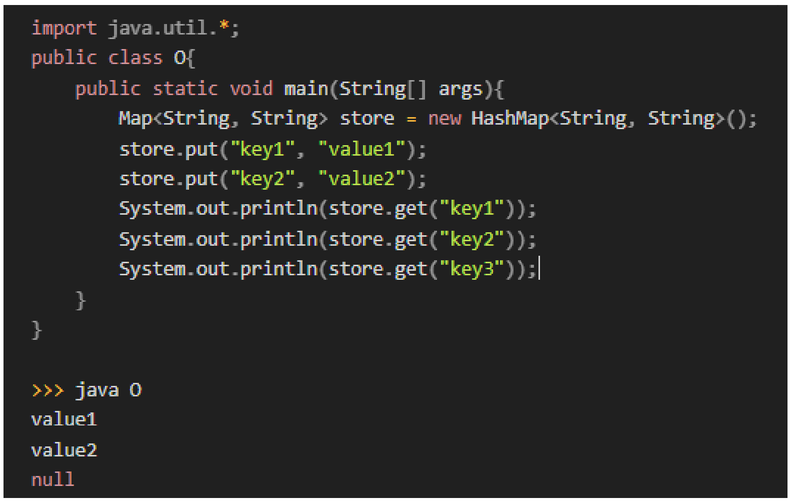
```
for (String key : map.keySet()) {
    Integer value = map.get(key);
    System.out.println("Key: " + key + ", Value: " + value);
}
```

- put()：会覆盖已存在的



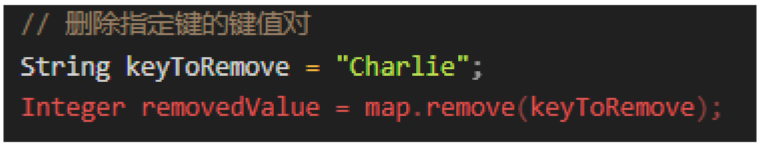
```
        public static void main(String[] args){
            Map<String, String> store = new HashMap<String, String>();
            store.put("key1", "value1");
            store.put("key2", "value2");
            System.out.println(store);
        }
    }

    >>> java O
    {key1=value1, key2=value2}
```

- get()：获取不存在的 key 会返回 null

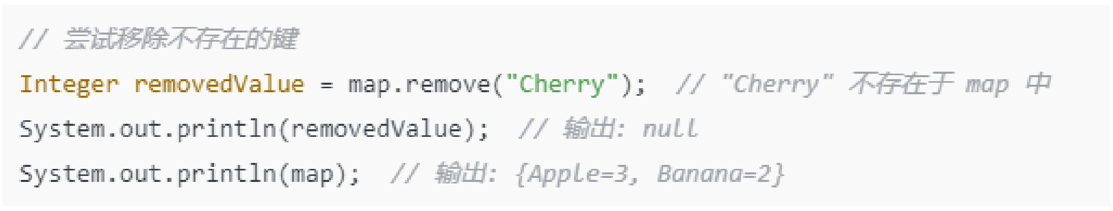


```
import java.util.*;
public class O{
    public static void main(String[] args){
        Map<String, String> store = new HashMap<String, String>();
        store.put("key1", "value1");
        store.put("key2", "value2");
        System.out.println(store.get("key1"));
        System.out.println(store.get("key2"));
        System.out.println(store.get("key3"));
    }
}

>>> java O
value1
value2
null
```

- remove()移除不存在的 key 会返回 null



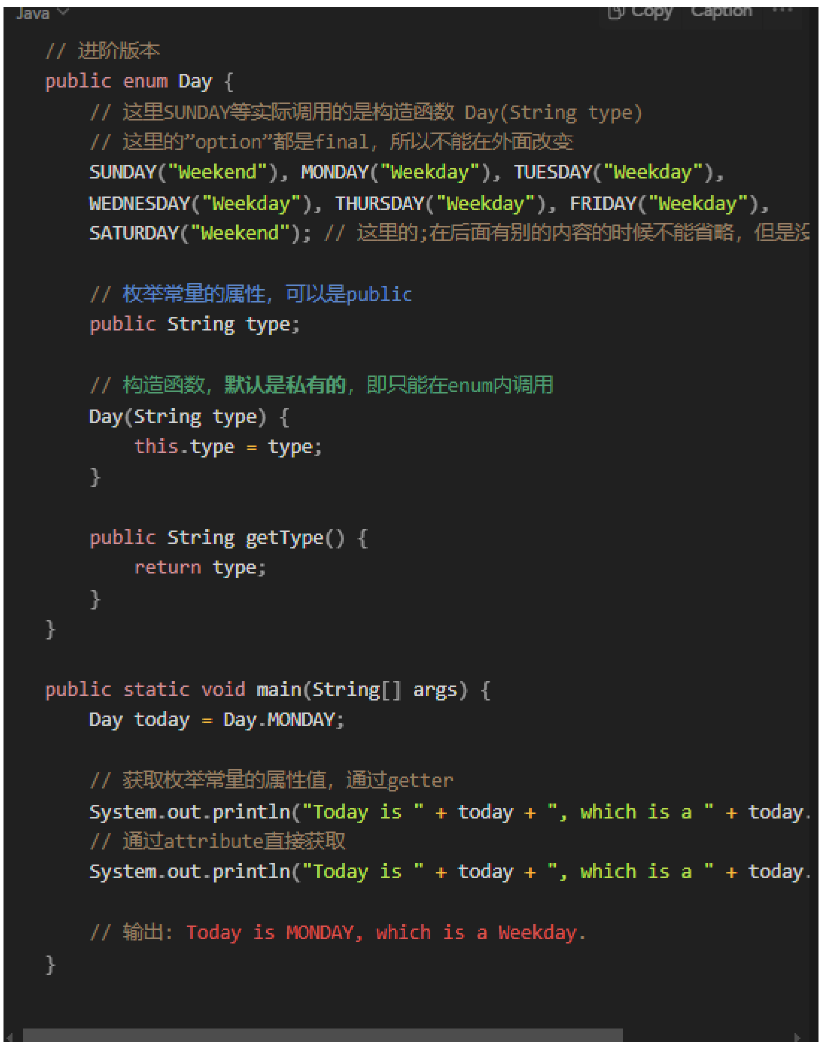
```java
// 删除指定键的键值对
String keyToRemove = "Charlie";
Integer removedValue = map.remove(keyToRemove);
```



```java
// 尝试移除不存在的键
Integer removedValue = map.remove("Cherry");  // "Cherry" 不存在于 map 中
System.out.println(removedValue);  // 输出: null
System.out.println(map);  // 输出: {Apple=3, Banana=2}
```

Enums

- 供的枚举中，构造函数 是用来初始化每个枚举常量的属性的。这个构造函数是私有的，因为枚举的构造函数默认是私有的，外部无法直接创建枚举实例，只能通过定义枚举常量来创建。

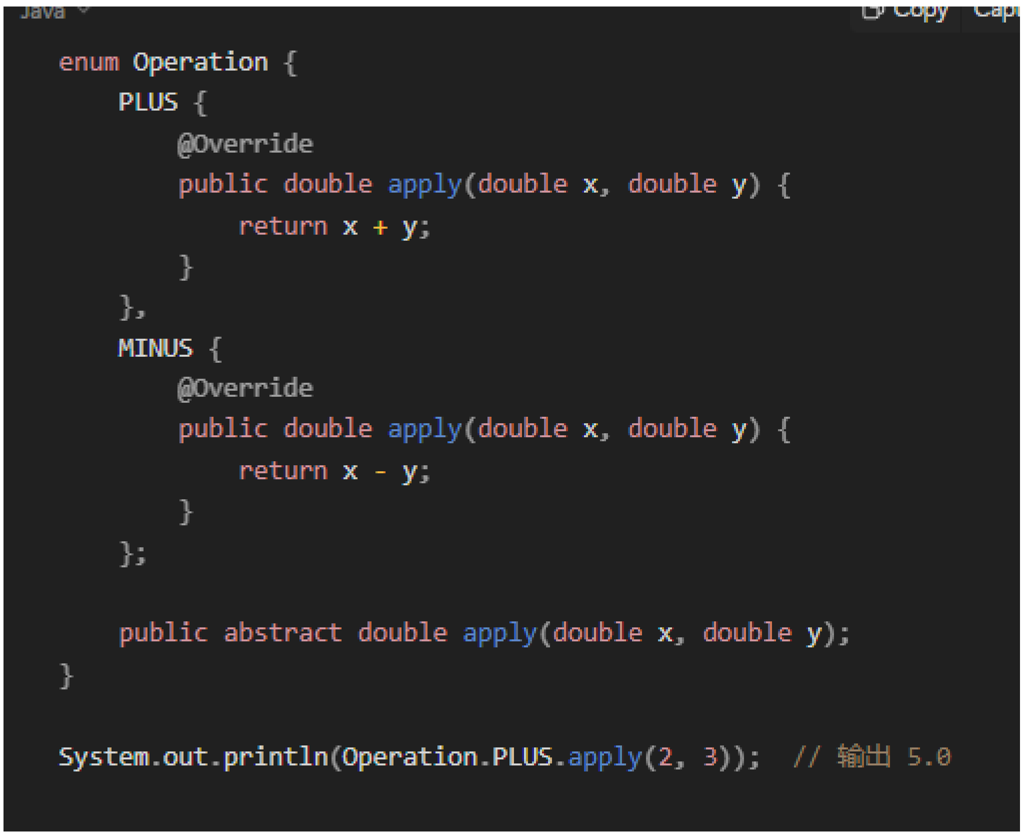


```java
// 进阶版本
public enum Day {
    // 这里SUNDAY等实际调用的是构造函数 Day(String type)
    // 这里的"option"都是final，所以不能在外面改变
    SUNDAY("Weekend"), MONDAY("Weekday"), TUESDAY("Weekday"),
    WEDNESDAY("Weekday"), THURSDAY("Weekday"), FRIDAY("Weekday"),
    SATURDAY("Weekend"); // 这里的;在后面有别的内容的时候不能省略，但是

    // 枚举常量的属性，可以是public
    public String type;

    // 构造函数，默认是私有的，即只能在enum内调用
    Day(String type) {
        this.type = type;
    }

    public String getType() {
        return type;
    }
}

public static void main(String[] args) {
    Day today = Day.MONDAY;

    // 获取枚举常量的属性值，通过getter
    System.out.println("Today is " + today + ", which is a " + today.
    // 通过attribute直接获取
    System.out.println("Today is " + today + ", which is a " + today.

    // 输出: Today is MONDAY, which is a Weekday.
}
```

- public String color; 是正确的，即 attributes 不一定是 private 的
- Enum.RED 返回的是枚举，但是 print 会调用 Enum.toString()方法

```java
enum Color {
    RED, GREEN, BLUE;
}

public class Main {
    public static void main(String[] args) {
        Color color = Color.RED;  // 这是一个 Color 类型的枚举常量
        System.out.println(color);  // 输出 "RED" (默认调用 color.toString())
    }
}
```

- Enum of abstract method

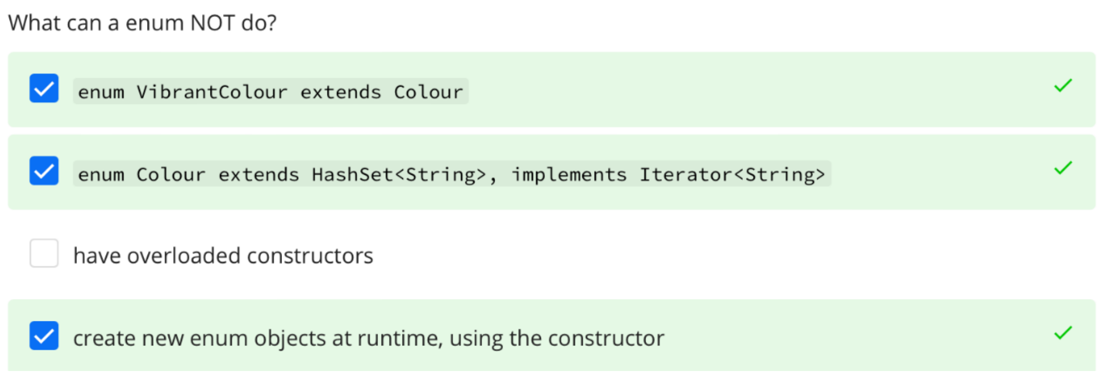


```java
enum Operation {
    PLUS {
        @Override
        public double apply(double x, double y) {
            return x + y;
        }
    },
    MINUS {
        @Override
        public double apply(double x, double y) {
            return x - y;
        }
    };

    public abstract double apply(double x, double y);
}

System.out.println(Operation.PLUS.apply(2, 3));  // 输出 5.0
```

- with abstract method and constructor

```
public enum Operation {
    ADD("+") {
        @Override
        public double apply(double x, double y) { return x + y; }
    },
    SUBTRACT("-") {
        @Override
        public double apply(double x, double y) { return x - y; }
    };

    private final String symbol;

    Operation(String symbol) { this.symbol = symbol; }
    public String getSymbol() { return symbol; }

    public abstract double apply(double x, double y);
}

public class Test {
    public static void main(String[] args) {
        System.out.println(Operation.ADD.apply(10, 5));      // 输出: 15.0
        System.out.println(Operation.SUBTRACT.apply(10, 5)); // 输出: 5.0
    }
}
```

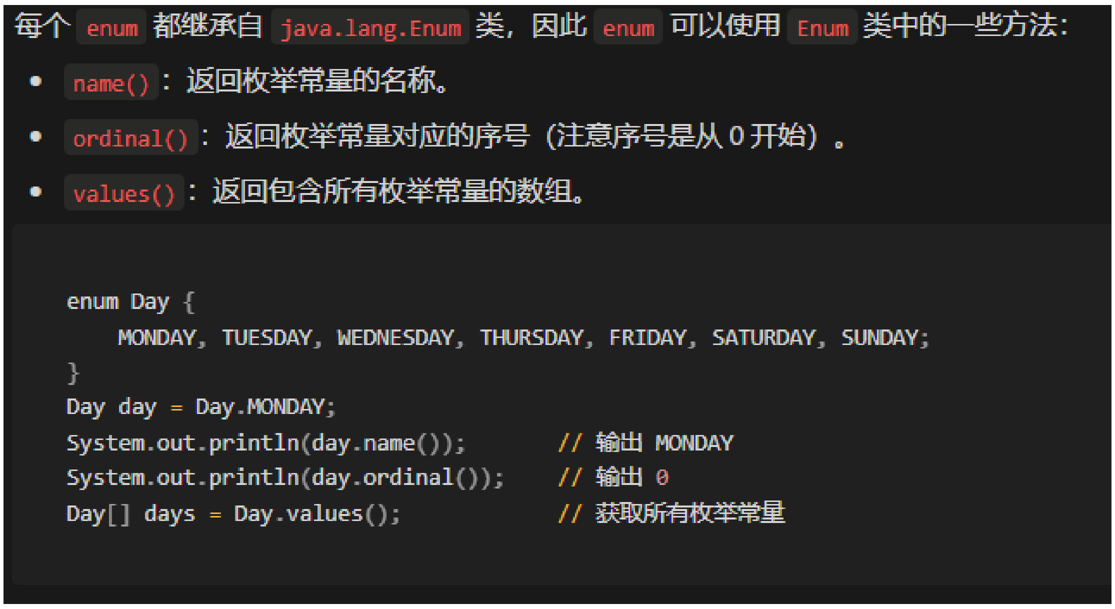
- enum 不能通过 extends 关键字继承其他类，因为 enum 本质上已经隐式地继承了 java.lang.Enum 类。由于 Java 不支持多继承，而 enum 已经继承了 Enum 类，因此它不能再继承其他类。



What can a enum NOT do?

- [✓ `enum VibrantColour extends Colour`] ✓
- [✓ `enum Colour extends HashSet<String>, implements Iterator<String>`] ✓
- [ ] have overloaded constructors
- [✓ create new enum objects at runtime, using the constructor] ✓

- 对应方法要记住


每个 `enum` 都继承自 `java.lang.Enum` 类，因此 `enum` 可以使用 `Enum` 类中的一些方法：

- `name()`：返回枚举常量的名称。
- `ordinal()`：返回枚举常量对应的序号（注意序号是从 0 开始）。
- `values()`：返回包含所有枚举常量的数组。

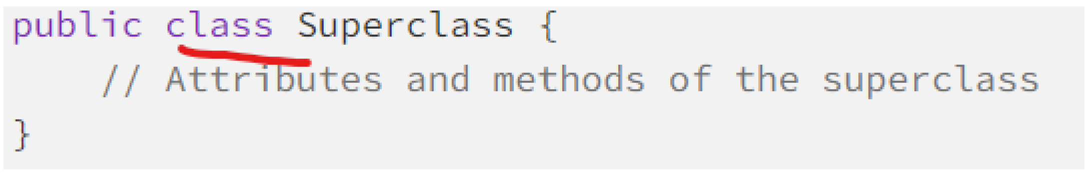
```
enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
}
Day day = Day.MONDAY;
System.out.println(day.name());       // 输出 MONDAY
System.out.println(day.ordinal());    // 输出 0
Day[] days = Day.values();            // 获取所有枚举常量
```

name（）不是对应的 attribute 啥的

- Day.valueOf(“MONDAY”) 返回的是 Day 类型的 MONDAY

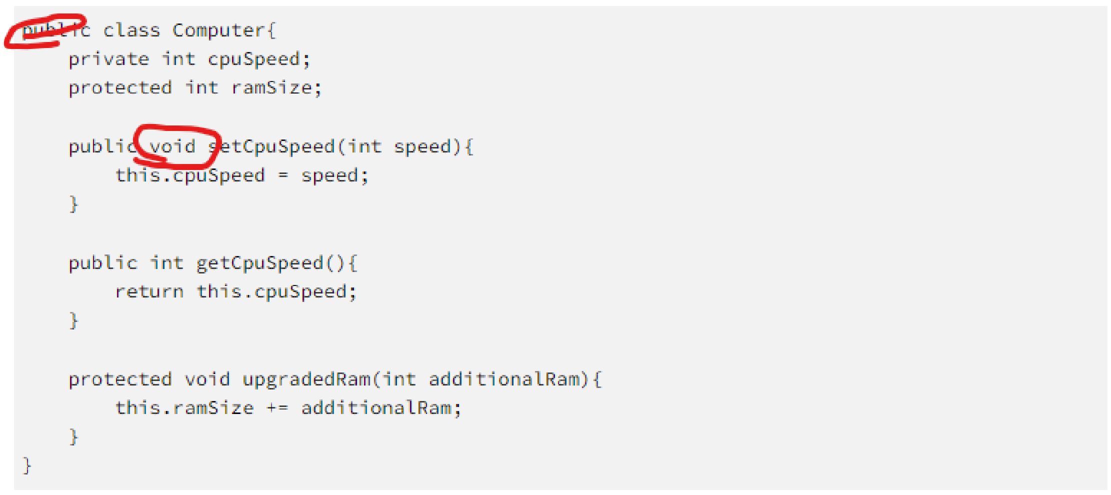
Class, methods, and attributes

- 再写的时候，class 需要小写



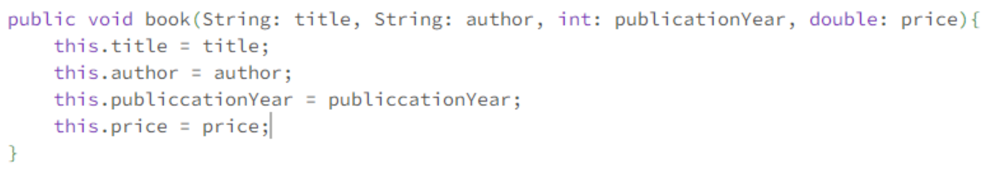
```
public class Superclass {
    // Attributes and methods of the superclass
}
```

- Class 分为 public 和普通 class，每个 public class 要和文件名一样，下面就是忘记带 public 了，并且 setter 和 getter 也要带 void

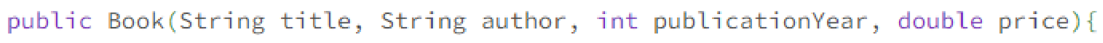


```
public class Computer{
    private int cpuSpeed;
    protected int ramSize;

    public void setCpuSpeed(int speed){
        this.cpuSpeed = speed;
    }

    public int getCpuSpeed(){
        return this.cpuSpeed;
    }

    protected void upgradedRam(int additionalRam){
        this.ramSize += additionalRam;
    }
}
```

- Class name 需要首字母大写
- 不需要带上“:”

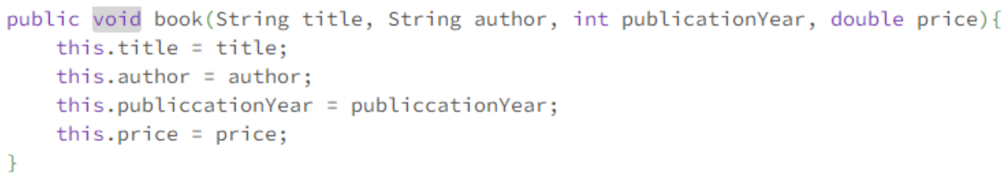


```
public void book(String: title, String: author, int: publicationYear, double: price){
    this.title = title;
    this.author = author;
    this.publiccationYear = publiccationYear;
    this.price = price;
}
```



```
public Book(String title, String author, int publicationYear, double price){
```

- Constructor 不能要 void，名字要和 className 一致



```
public void book(String title, String author, int publicationYear, double price){
    this.title = title;
    this.author = author;
    this.publiccationYear = publiccationYear;
    this.price = price;
}
```

```java
public class Book {
    private String title;
    private String author;
    private int publiccationYear;
    private double price;

    public Book(String title, String author, int publicationYear, double price){
        this.title = title;
        this.author = author;
        this.publiccationYear = publiccationYear;
        this.price = price;
    }
```

- an attribute or method is declared protected, it is accessible within the same package, as well as in subclasses (child classes) located in different packages.
- 包私有 modifier，我们通过不在 attributes 或 methods 前面添加任何 modifier 来获得，也叫 default modifier，Members with default access are accessible to other classes within the same package but are not accessible from classes outside the package, even if those classes are subclasses.

```java
class Company {
    String companyName; // package-private attribute

    Company(String name) {
        this.companyName = name;
    }
}

class Department {
    public void printCompanyName() {
        Company company = new Company("Tech Solutions");
        System.out.println("Company Name: " + company.companyName);
    }
}
```

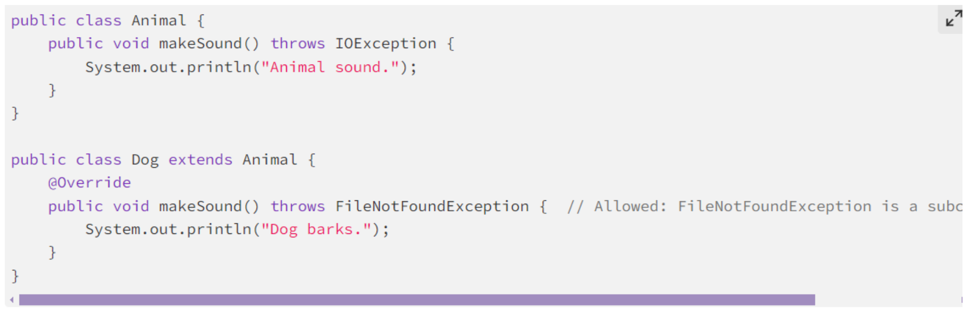
访问级别	访问范围	定义方式
包私有	同包内的类可以访问，不同包无法访问	不加任何访问修饰符
public	所有包中的类均可访问	在成员或类前加上 `public` 关键字

不要忘记即使是 default modifier，我们也要声明 return type

```
void printLibraryInfo(){
    System.out.println(this.bookCount);
}
```

Class inheritance and Polymorphism

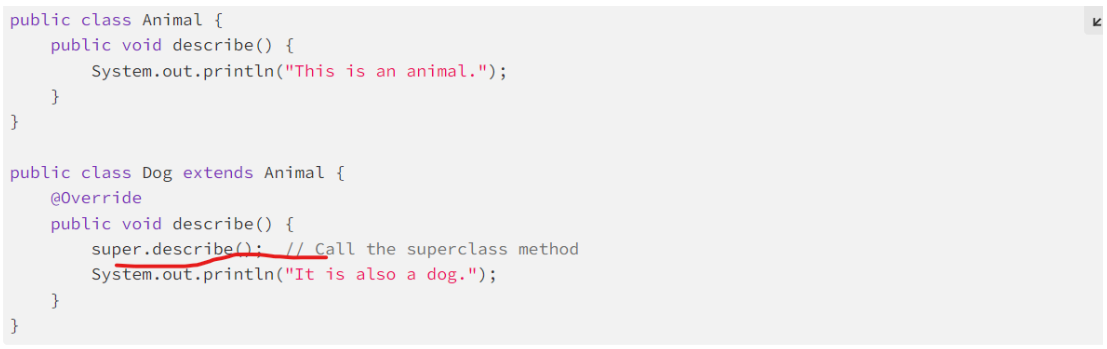
- 只继承 public 和 protected
- extends
- 子类使用父类的 constructor 会导致错误，因为父不一定包含子类所需的 attribute
- Override method 必须保持
  - 方法签名的一致性。**方法签名**包括方法的名称、参数类型、参数数量和参数顺序。
  - 子类方法的访问修饰符必须至少与父类方法的访问修饰符相同或更宽松. public > protected > default > private
  - **Compatible Exception Handling : If the superclass method declares checked exceptions, the subclass method can only throw the same exceptions or subclasses of those exceptions.**



```
public class Animal {
    public void makeSound() throws IOException {
        System.out.println("Animal sound.");
    }
}

public class Dog extends Animal {
    @Override
    public void makeSound() throws FileNotFoundException {  // Allowed: FileNotFoundException is a subc
        System.out.println("Dog barks.");
    }
}
```

In this example, `makeSound()` in `Dog` can throw `FileNotFoundException` because it is a subclass of `IOException`. However, if `Dog`'s `makeSound()` tried to throw an unrelated checked exception, it would result in a compilation error.

- super(), When you use super, it must be the first statement in the subclass's constructor



```
public class Animal {
    public void describe() {
        System.out.println("This is an animal.");
    }
}

public class Dog extends Animal {
    @Override
    public void describe() {
        super.describe();  // Call the superclass method
        System.out.out.println("It is also a dog.");
    }
}
```

- When you create an object of a subclass, Java automatically calls the constructor of the superclass first to ensure that all the inherited attributes are properly initialised before any additional initialisation takes place in the subclass.
- If you omit super（）, Java will implicitly insert a call to the no-argument constructor of the superclass. If the superclass doesn't have a no-argument constructor, you'll

encounter a compilation error unless you explicitly call another constructor in the superclass.

- Overloading constructor

```java
// No-argument constructor
public Rectangle() {
    this.width = 0;
    this.height = 0;
}

// Constructor with one parameter
public Rectangle(int size) {
    this.width = size;
    this.height = size;
}

// Constructor with two parameters
public Rectangle(int width, int height) {
    this.width = width;
    this.height = height;
}
```

this 必须在第一行

```java
class Car {
    String model;
    int year;
    double price;

    // 默认构造函数
    public Car() {
        this("Unknown", 0);  // 调用带两个参数的构造函数
    }

    // 带两个参数的构造函数
    public Car(String model, int year) {
        this(model, year, 0.0);  // 调用带三个参数的构造函数
    }

    // 带三个参数的构造函数
    public Car(String model, int year, double price) {
        this.model = model;
        this.year = year;
        this.price = price;
}
```

- Polymorphism 在遍历 animals 集合时，你是不能直接调用 Dog 类特有的方法的

Suppose you want to create a list of different animals and call `makeSound()` on each one. You can do this by creating an `ArrayList` of `Animal` objects and adding instances of `Dog`, `Cat`, and `Animal` to it. When you loop through the list and call `makeSound()`, Java will dynamically invoke the correct version of the method for each object.

Here's how it looks in code:

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<Animal> animals = new ArrayList<>();

        animals.add(new Dog());
        animals.add(new Cat());
        animals.add(new Animal());

        for (Animal animal : animals) {
            animal.makeSound();
        }
    }
}
```

The output will be:

```
Dog barks.
Cat meows.
Animal makes a sound.
```

- instanceof 不仅可以识别父类，也可以识别接口。

```
public void checkAnimalType(Animal animal) {
    if (animal instanceof Dog) {
        Dog dog = (Dog) animal;  // Downcasting to Dog
        System.out.println("This animal is a dog.");
        dog.bark();
    } else if (animal instanceof Cat) {
        Cat cat = (Cat) animal;  // Downcasting to Cat
        System.out.println("This animal is a cat.");
        cat.meow();
    } else {
        System.out.println("This is some other type of animal.");
        animal.makeSound();
    }
}
```

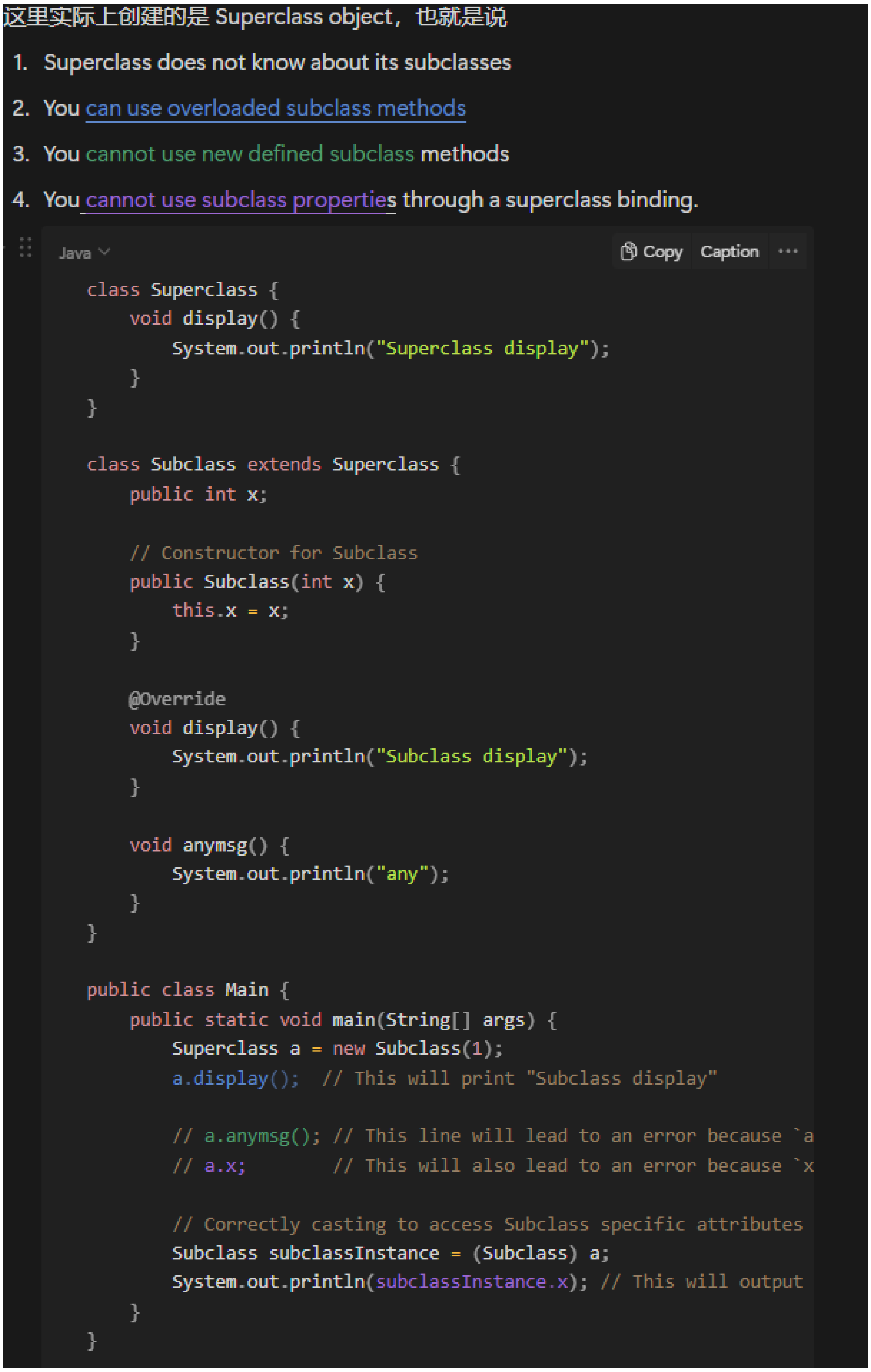
- Runtime polymorphism - override

```
public class Animal {
    public void makeSound() {
        System.out.println("Animal makes a sound.");
    }
}

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

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

- Compile time polymorphism - overload

```
public class MathOperations {
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }

    public int add(int a, int b, int c) {
        return a + b + c;
    }
}
```

- Superclass a = new Subclass();



这里实际上创建的是 Superclass object，也就是说

1. Superclass does not know about its subclasses
2. You can use overloaded subclass methods
3. You cannot use new defined subclass methods
4. You cannot use subclass properties through a superclass binding.

```java
class Superclass {
    void display() {
        System.out.println("Superclass display");
    }
}

class Subclass extends Superclass {
    public int x;

    // Constructor for Subclass
    public Subclass(int x) {
        this.x = x;
    }

    @Override
    void display() {
        System.out.println("Subclass display");
    }

    void anymsg() {
        System.out.println("any");
    }
}

public class Main {
    public static void main(String[] args) {
        Superclass a = new Subclass(1);
        a.display();  // This will print "Subclass display"

        // a.anymsg(); // This line will lead to an error because `a
        // a.x;        // This will also lead to an error because `x

        // Correctly casting to access Subclass specific attributes
        Subclass subclassInstance = (Subclass) a;
        System.out.println(subclassInstance.x); // This will output
    }
}
```

File IO

- `import java.io.*;`
- File I/O 可以在 method throws IO exception 或者 method 内 try+catch

```java
// 方法声明 throws IOException，以便在遇到 I/O 错误时抛出异常
public void readFile(String filePath) throws IOException {
    FileReader fileReader = new FileReader(filePath);
    BufferedReader bufferedReader = new BufferedReader(fileReader);

    String line;
    while ((line = bufferedReader.readLine()) != null) {
        System.out.println(line);
    }

    bufferedReader.close(); // 记得关闭资源
}
```

```java
// 不抛出异常，而是在方法内处理
public void readFile(String filePath) {
    try {
        FileReader fileReader = new FileReader(filePath);
        BufferedReader bufferedReader = new BufferedReader(fileReader);

        String line;
        while ((line = bufferedReader.readLine()) != null) {
            System.out.println(line);
        }

        bufferedReader.close(); // 记得关闭资源
    } catch (IOException e) {
        // 在方法内部处理 IOException
        System.out.println("An I/O error occurred: " + e.getMessage());
    }
}
```

Console-based: stdin

- if nextInt() is called and the user enters a non-integer value, Java will raise an InputMismatchException
- try with resources will automatically close file，可以看到这里通过 ； 分割 （） 声明了 2 个资源

```java
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

public class TryWithResourcesExample {
    public static void main(String[] args) {
        try (FileReader fileReader = new FileReader("example.txt");
             BufferedReader bufferedReader = new BufferedReader(fileReader)) {

            String line;
            while ((line = bufferedReader.readLine()) != null) {
                System.out.println(line);
            }

        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
        }
    }
}
```

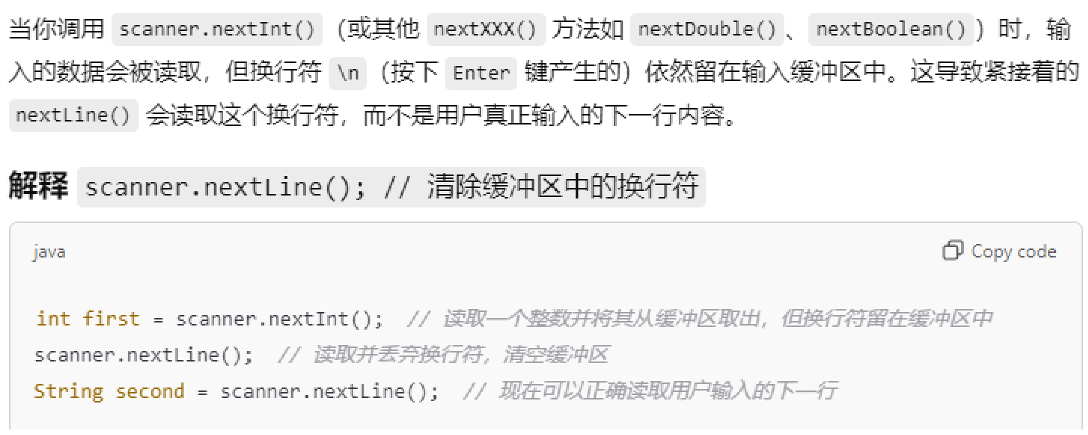
Scanner

```java
public class ConsoleInputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter your name: ");
        String name = scanner.nextLine(); // Reads a full line of text

        System.out.print("Enter your age: ");
        int age = scanner.nextInt(); // Reads an integer

        System.out.println("Hello, " + name + "! You are " + age + " years old.");

        scanner.close();
    }
}
```

- import java.util.Scanner;
- System.in, which enables the program to read from the console (stdin).

```java
Scanner scanner = new Scanner(System.in);
```

- `next()` reads the next token, which is typically a single word.
- nextLine()读取当前行，包括 Enter 产生的换行符（所以上面的代码没问题）
- nextXxxx()


当你调用 `scanner.nextInt()`（或其他 `nextXXX()` 方法如 `nextDouble()`、`nextBoolean()`）时，输入的数据会被读取，但换行符 `\n`（按下 `Enter` 键产生的）依然留在输入缓冲区中。这导致紧接着的 `nextLine()` 会读取这个换行符，而不是用户真正输入的下一行内容。

解释 `scanner.nextLine(); // 清除缓冲区中的换行符`

java Copy code

```java
int first = scanner.nextInt();  // 读取一个整数并将其从缓冲区取出，但换行符留在缓冲区中
scanner.nextLine();  // 读取并丢弃换行符，清空缓冲区
String second = scanner.nextLine();  // 现在可以正确读取用户输入的下一行
```

PrintWriter

```java
import java.io.PrintWriter;

public class ConsoleOutputExample {
    public static void main(String[] args) {
        PrintWriter writer = new PrintWriter(System.out, true); // Enable auto-flushing

        writer.println("Welcome to Java Text I/O!");
        writer.printf("This program demonstrates %s output.\n", "formatted");

        int score = 95;
        writer.printf("Your score is: %d out of 100\n", score);

        writer.close();
    }
}
```

- `import java.io.PrintWriter;`
- System.out, which enables the program to write to the console (stdin).
- By setting the second parameter to true, we enable auto-flushing, so the output is printed to the console as soon as it’s written.
- %s for strings and %d for integers，%.2f 既可以接收 float 又可以接受 double

---

From file: FileReader + FileWriter

- For binary files, such as images or serialised objects, Java provides FileInputStream and FileOutputStream, which handle raw bytes rather than characters.
- FileNotFoundException VS IOException
    - IOException: can found file, but read lead to exception
    - FileNotFoundException: cannot find file

FileReader

- read()读取一个字符

1. 返回一个字符的整数值（`0` 到 `65535` 范围内的 Unicode 编码）

如果 `read()` 方法成功读取到一个字符，它会返回该字符的 Unicode 编码（一个整数值）。这个整数值可以转换成字符，通常通过 `(char)` 强制转换来处理。

例如：

```java
int i = fileReader.read();
char ch = (char) i; // 将读取到的整数转换为字符
```

2. 返回 `-1` 表示已到达文件末尾

当 `read()` 方法到达文件的末尾时，它会返回 `-1`，表示没有更多的字符可以读取。

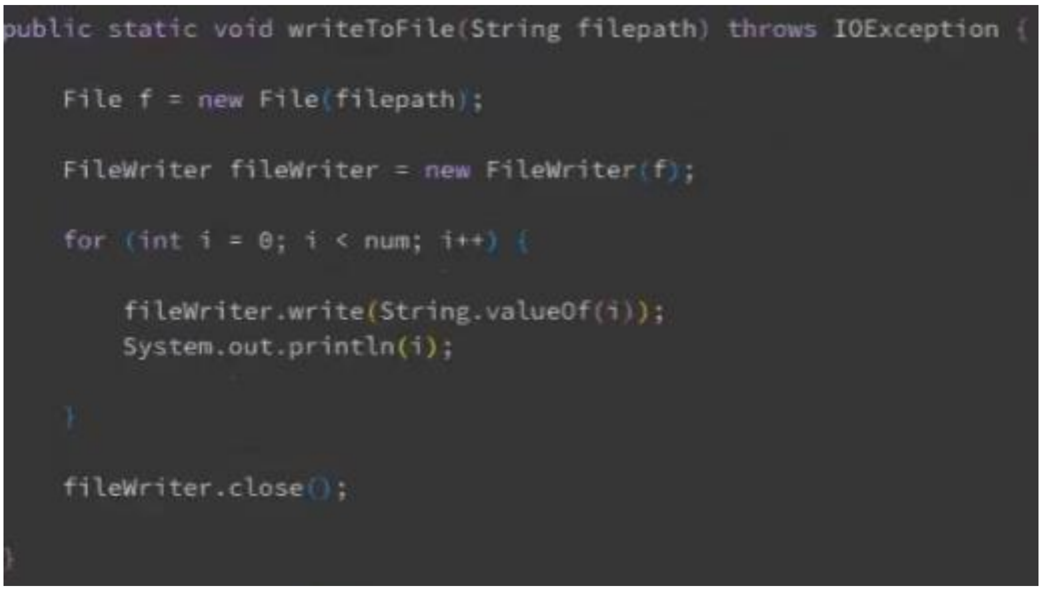
例如：

```java
int i = fileReader.read();
if (i == -1) {
    System.out.println("End of file reached.");
}
```

Scanner

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

public class ReadFileScannerExample {
    public static void main(String[] args) {
        try {
            File file = new File("example.txt");  // 创建 File 对象
            Scanner scanner = new Scanner(file);  // 创建 Scanner 对象

            while (scanner.hasNextLine()) {
                String line = scanner.nextLine();  // 逐行读取内容
                System.out.println(line);
            }

            scanner.close();  // 关闭 Scanner
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

FileWriter



```java
public static void writeToFile(String filepath) throws IOException {

    File f = new File(filepath);

    FileWriter fileWriter = new FileWriter(f);

    for (int i = 0; i < num; i++) {

        fileWriter.write(String.valueOf(i));
        System.out.println(i);

    }

    fileWriter.close();

}
```

- FileWriter need to close after use; 注意是 FileWriter 需要 close 不是 file 需要 close; FileWriter (f,true)

  ```java
  try (FileWriter fileWriter = new FileWriter("output.txt", true)) { // true 表示追加模式
  ```

- write() method was used, ;表示追加模式

Abstract class

- extends

  ```
  public class Truck extends Vehicle
  ```

- 不实现 abstract class 的 ABSTRACT method 会报错 compiling error
- **抽象类中的字段**：不需要实现，子类会继承这些字段。子类可以访问这些字段，并且可以对它们进行修改，或者通过构造方法或方法进行初始化。
- 可以有 constructor
- 需要 abstract 标明抽象 method；不需要 default

```
public abstract class Animal {
    private String name;
    private int age;

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

    public void eat() {
        System.out.println(name + " is eating.");
    }

    public abstract void makeSound();
}
```

```
public abstract <T> void addItem(T item); // 泛型方法，使用类型参数 T
```

```
public abstract void processList(List<? extends Number> list); // 使用通配符 ? extends
```

Interface

- implements

  ```
  public class Car implements Drivable
  ```

- 不需要 abstract 标明抽象 method；需要 default

```java
public interface Drivable {
    void accelerate(int increment);
    void brake(int decrement);

    default void park() {
        System.out.println("Parking the vehicle.");
    }
}
```

- Interfaces cannot have constructors, as they do not hold any state.
- 所有接口（interface）中定义的变量默认都是 **static** 和 **final** 的。这是 Java 接口的一个特性。
- polymorphism

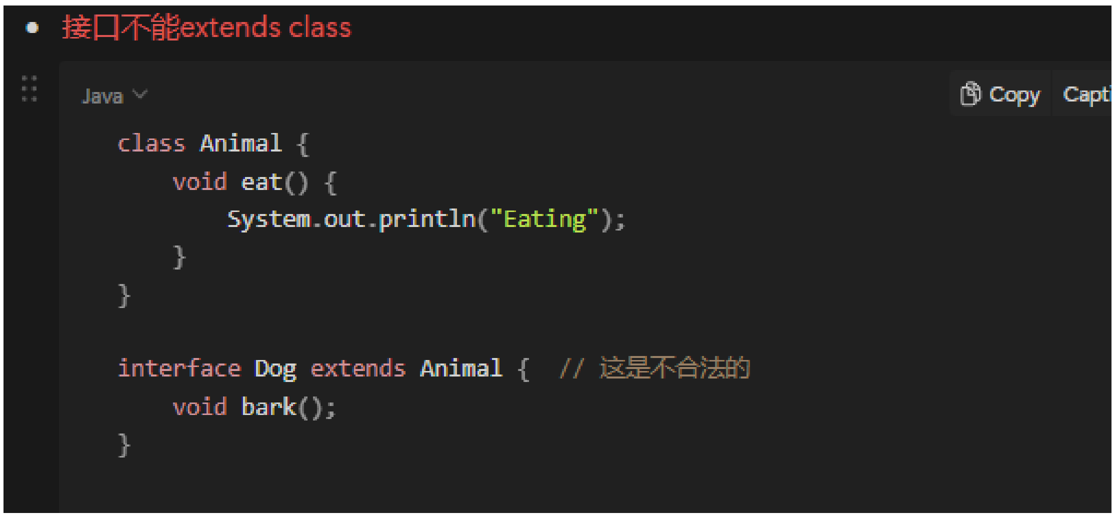
```java
public class VehicleService {
    public void testDrive(Drivable drivable) {
        drivable.accelerate(10);
        drivable.brake(5);
    }
}
```

In this example, `testDrive()` takes a `Drivable` object as its parameter, allowing it to interact with any object that implements `Drivable`. You could pass a `Car` or `FlyingCar` to `testDrive()`:

```java
VehicleService service = new VehicleService();
Car myCar = new Car(120, 50);
FlyingCar myFlyingCar = new FlyingCar();

service.testDrive(myCar);       // Works because Car is Drivable
service.testDrive(myFlyingCar); // Works because FlyingCar is Drivable
```

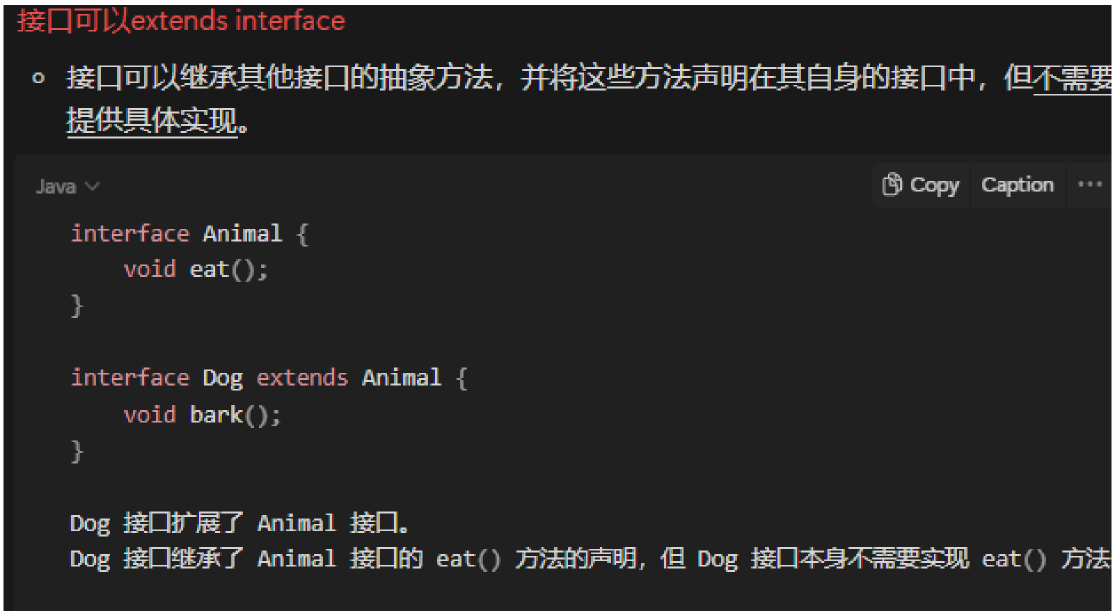
接口适用于无关类共享行为，例如不同类都可以实现 Flyable。
抽象类适用于相关类共享代码，例如各种动物的 Animal 类。

```java
class ClassName extends parrentClass implements InterfaceName1, InterfaceName1 {}
```



- 接口不能extends class

```java
class Animal {
    void eat() {
        System.out.println("Eating");
    }
}

interface Dog extends Animal {  // 这是不合法的
    void bark();
}
```

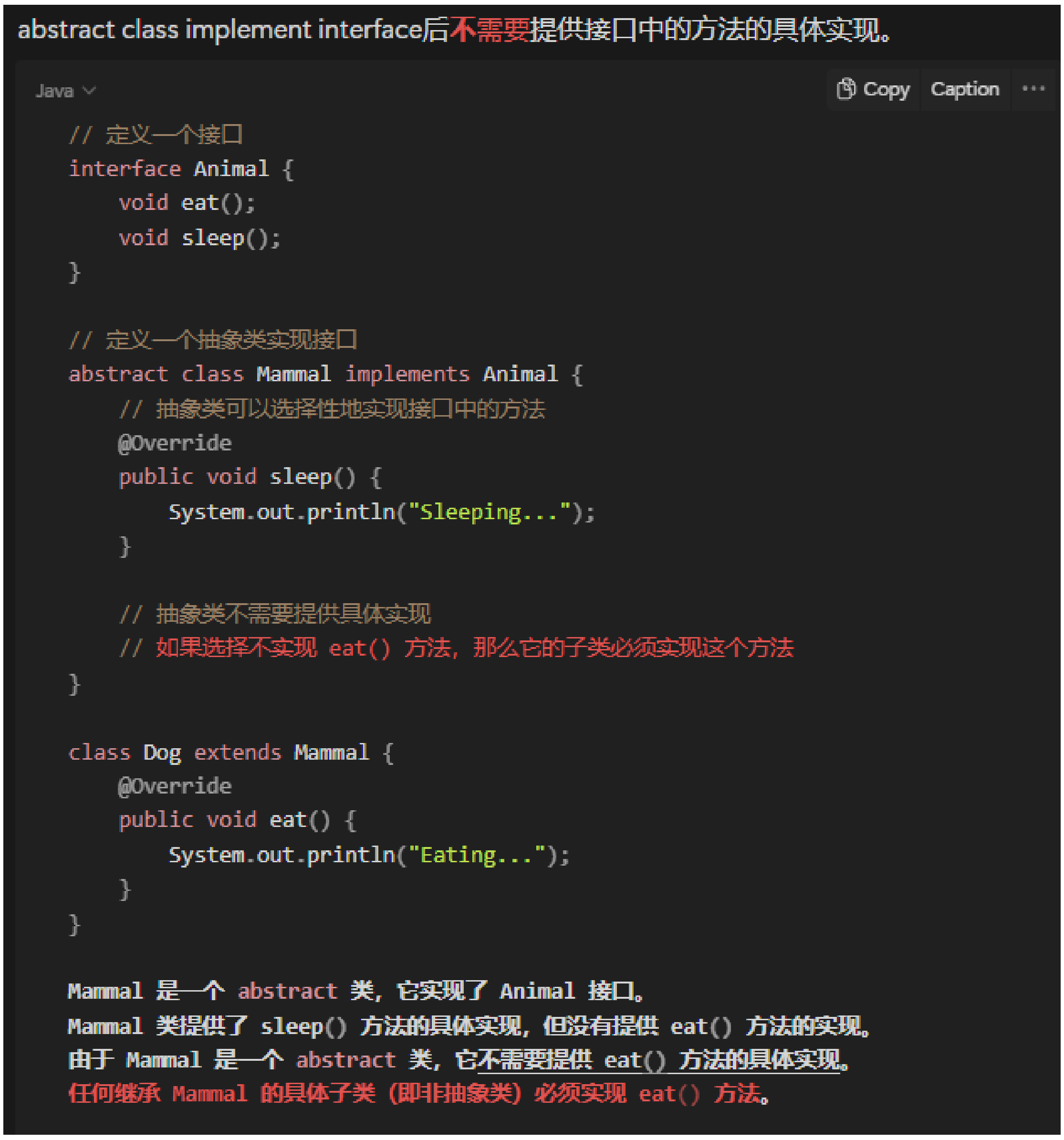


接口可以extends interface

- 接口可以继承其他接口的抽象方法，并将这些方法声明在其自身的接口中，但不需要提供具体实现。

```java
interface Animal {
    void eat();
}

interface Dog extends Animal {
    void bark();
}
```

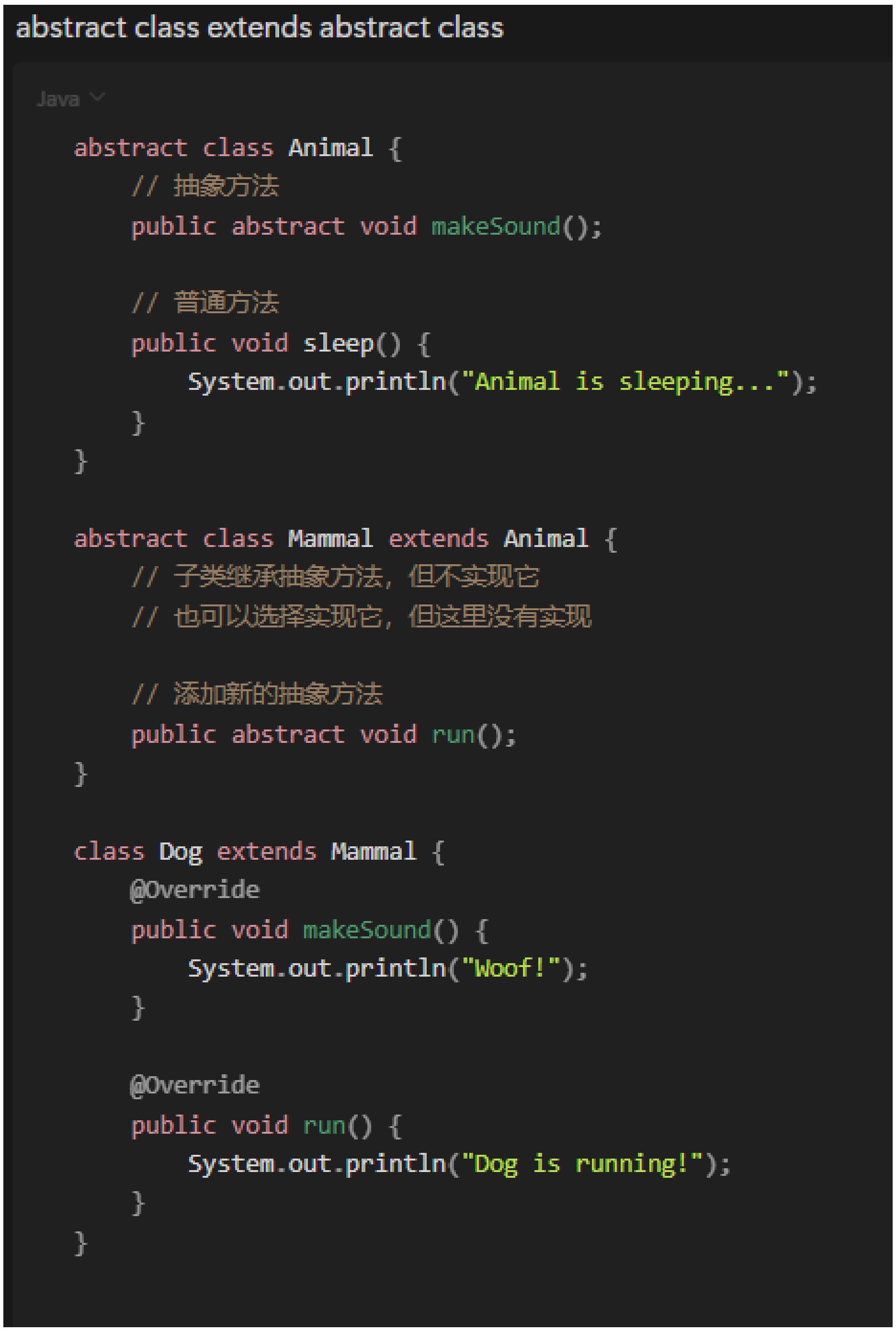
Dog 接口扩展了 Animal 接口。
Dog 接口继承了 Animal 接口的 eat() 方法的声明，但 Dog 接口本身不需要实现 eat() 方法



abstract class implement interface后不需要提供接口中的方法的具体实现。

```java
// 定义一个接口
interface Animal {
    void eat();
    void sleep();
}

// 定义一个抽象类实现接口
abstract class Mammal implements Animal {
    // 抽象类可以选择性地实现接口中的方法
    @Override
    public void sleep() {
        System.out.println("Sleeping...");
    }

    // 抽象类不需要提供具体实现
    // 如果选择不实现 eat() 方法，那么它的子类必须实现这个方法
}

class Dog extends Mammal {
    @Override
    public void eat() {
        System.out.println("Eating...");
    }
}
```

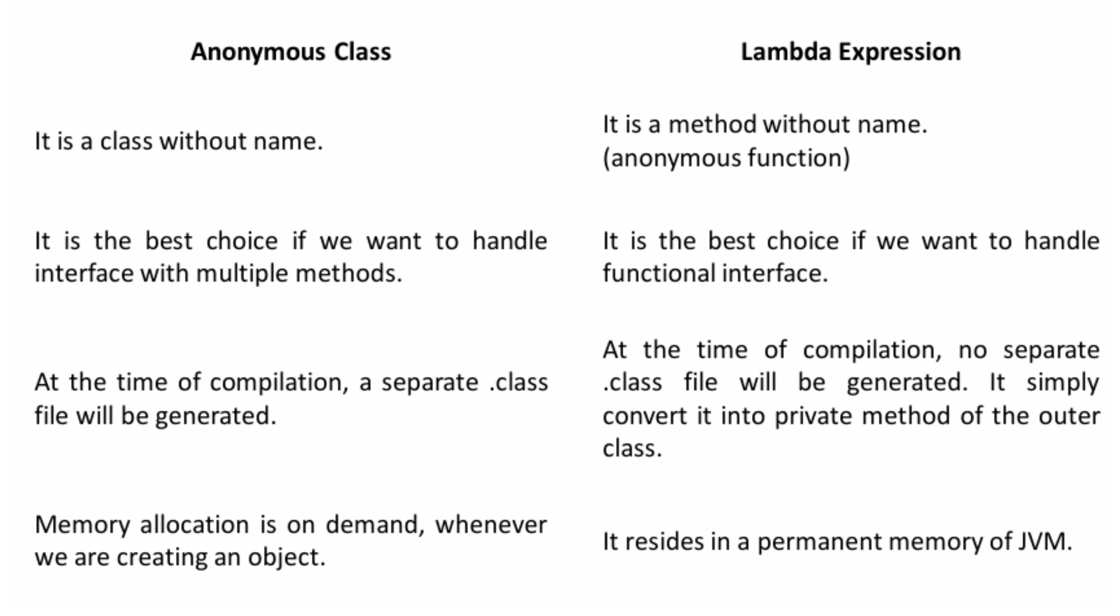
Mammal 是一个 abstract 类，它实现了 Animal 接口。
Mammal 类提供了 sleep() 方法的具体实现，但没有提供 eat() 方法的实现。
由于 Mammal 是一个 abstract 类，它不需要提供 eat() 方法的具体实现。
任何继承 Mammal 的具体子类（即非抽象类）必须实现 eat() 方法。


abstract class extends abstract class

Java

```java
abstract class Animal {
    // 抽象方法
    public abstract void makeSound();

    // 普通方法
    public void sleep() {
        System.out.println("Animal is sleeping...");
    }
}

abstract class Mammal extends Animal {
    // 子类继承抽象方法，但不实现它
    // 也可以选择实现它，但这里没有实现

    // 添加新的抽象方法
    public abstract void run();
}

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

    @Override
    public void run() {
        System.out.println("Dog is running!");
    }
}
```

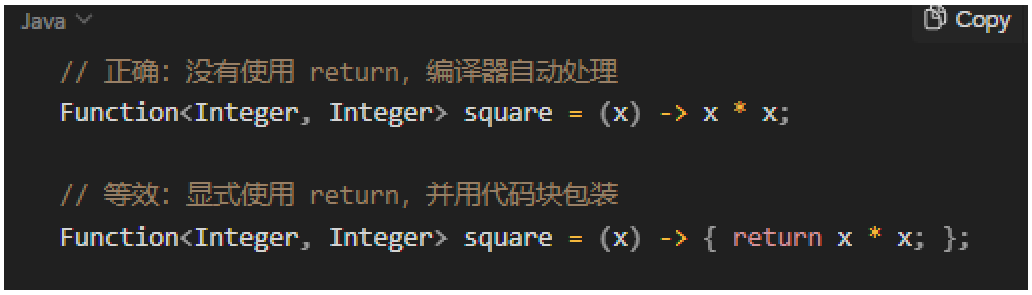
Anonymous & Lambda



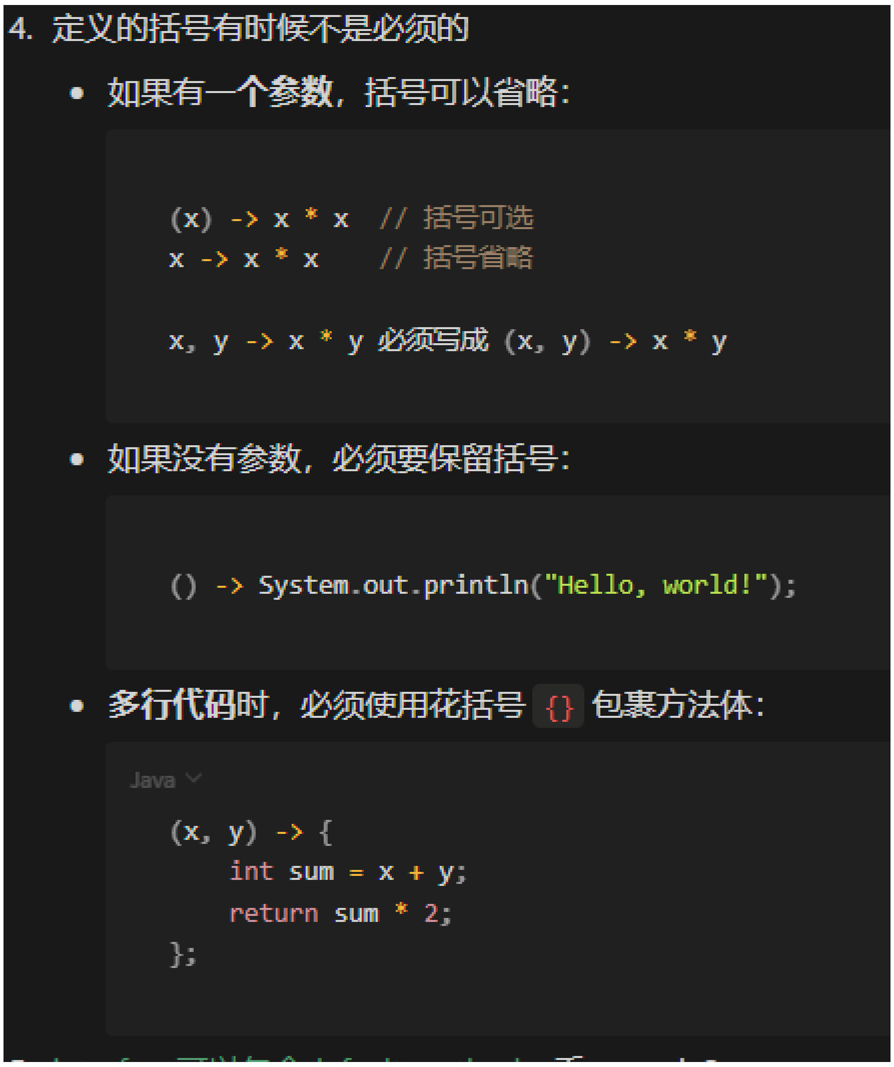
Anonymous Class	Lambda Expression
It is a class without name.	It is a method without name. (anonymous function)
It is the best choice if we want to handle interface with multiple methods.	It is the best choice if we want to handle functional interface.
At the time of compilation, a separate .class file will be generated.	At the time of compilation, no separate .class file will be generated. It simply convert it into private method of the outer class.
Memory allocation is on demand, whenever we are creating an object.	It resides in a permanent memory of JVM.

Lambda

1. 只能由 require **an interface** that declares only **one abstract** method.
2. 结尾需要;



```java
// 正确：没有使用 return，编译器自动处理
Function<Integer, Integer> square = (x) -> x * x;

// 等效：显式使用 return，并用代码块包装
Function<Integer, Integer> square = (x) -> { return x * x; };
```



4. 定义的括号有时候不是必须的

- 如果有一个参数，括号可以省略：

```
(x) -> x * x  // 括号可选
x -> x * x    // 括号省略

x, y -> x * y 必须写成 (x, y) -> x * y
```

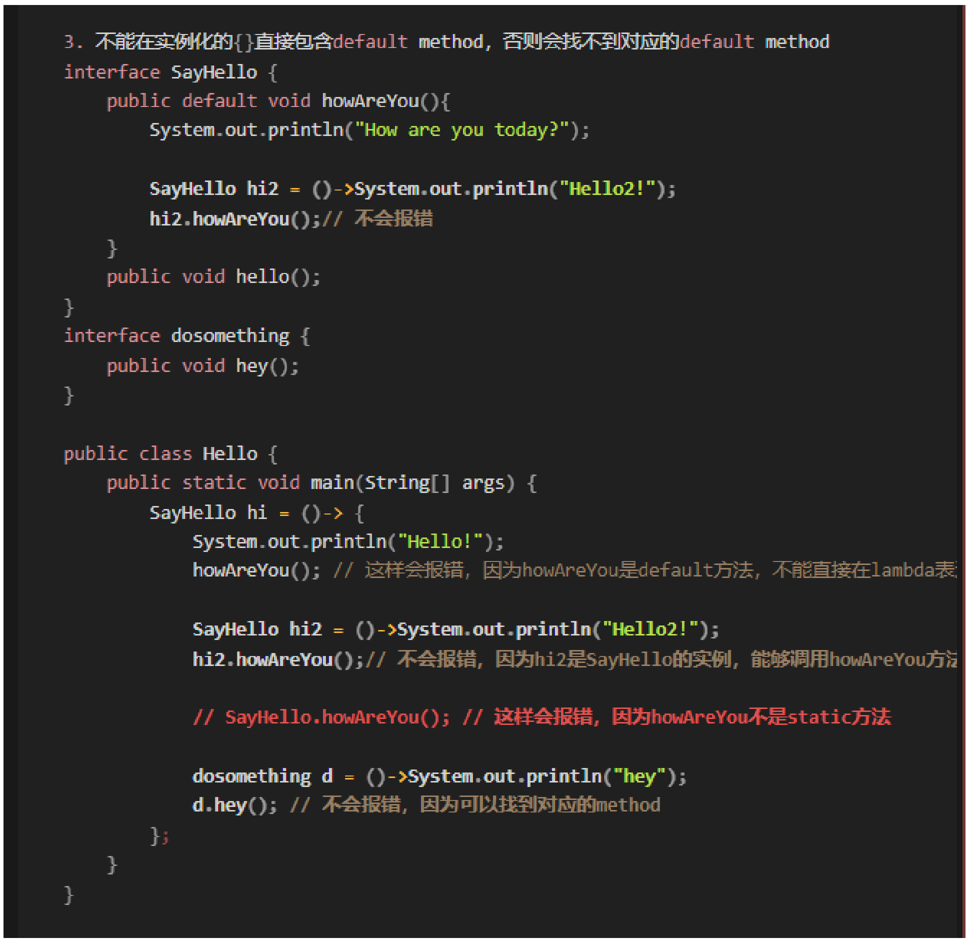
- 如果没有参数，必须要保留括号：

```
() -> System.out.println("Hello, world!");
```

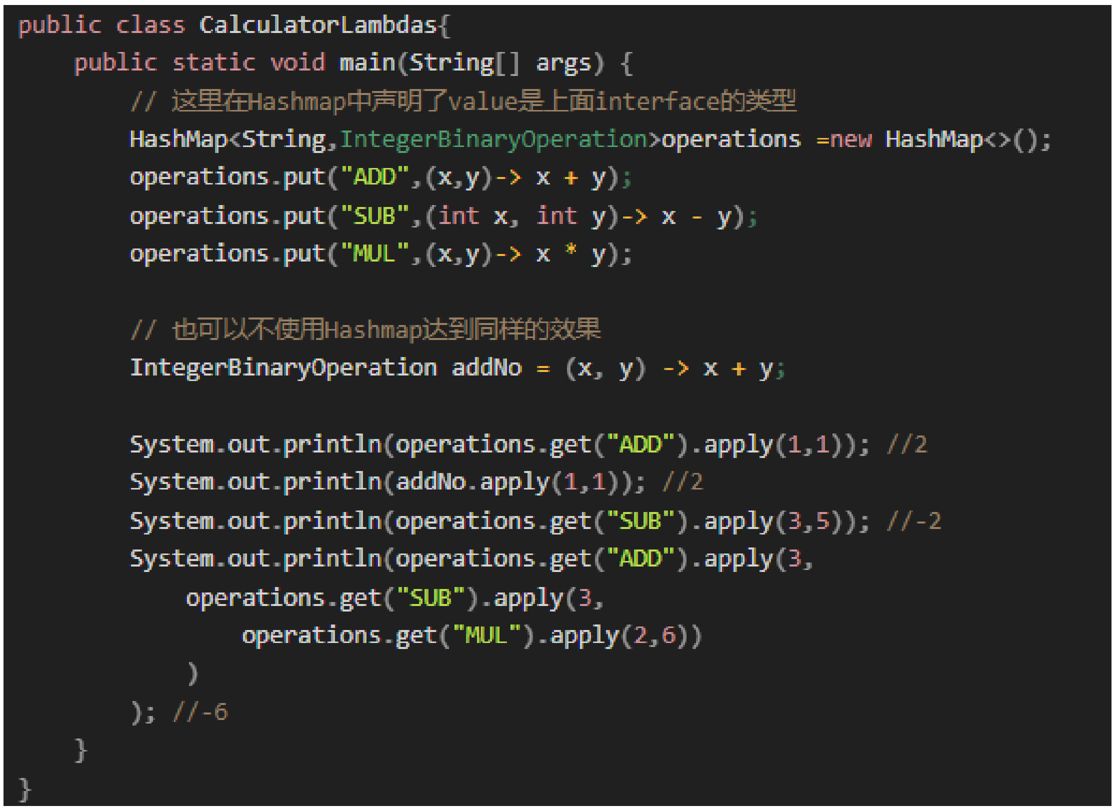
- 多行代码时，必须使用花括号 `{}` 包裹方法体：

```Java
(x, y) -> {
    int sum = x + y;
    return sum * 2;
};
```

3. interface 可以包含 default method

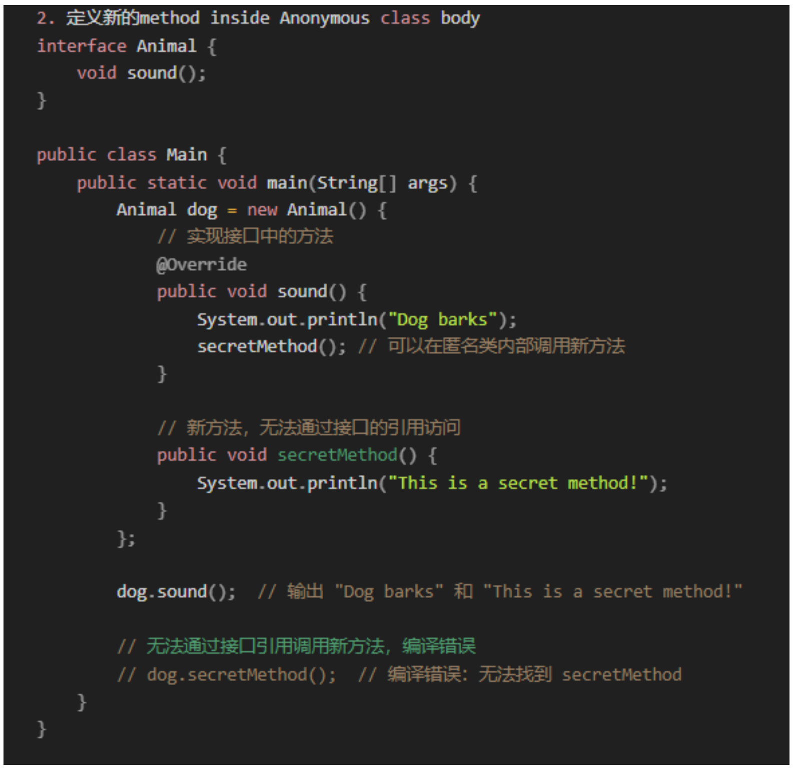


```java
3. 不能在实例化的{}直接包含default method，否则会找不到对应的default method
interface SayHello {
    public default void howAreYou(){
        System.out.println("How are you today?");

        SayHello hi2 = ()->System.out.println("Hello2!");
        hi2.howAreYou();// 不会报错
    }
    public void hello();
}
interface dosomething {
    public void hey();
}

public class Hello {
    public static void main(String[] args) {
        SayHello hi = ()-> {
            System.out.println("Hello!");
            howAreYou(); // 这样会报错，因为howAreYou是default方法，不能直接在lambda表

            SayHello hi2 = ()->System.out.println("Hello2!");
            hi2.howAreYou();// 不会报错，因为hi2是SayHello的实例，能够调用howAreYou方

            // SayHello.howAreYou(); // 这样会报错，因为howAreYou不是static方法

            dosomething d = ()->System.out.println("hey");
            d.hey(); // 不会报错，因为可以找到对应的method
        };
    }
}
```



```java
public class CalculatorLambdas{
    public static void main(String[] args) {
        // 这里在Hashmap中声明了value是上面interface的类型
        HashMap<String,IntegerBinaryOperation>operations =new HashMap<>();
        operations.put("ADD",(x,y)-> x + y);
        operations.put("SUB",(int x, int y)-> x - y);
        operations.put("MUL",(x,y)-> x * y);

        // 也可以不使用Hashmap达到同样的效果
        IntegerBinaryOperation addNo = (x, y) -> x + y;

        System.out.println(operations.get("ADD").apply(1,1)); //2
        System.out.println(addNo.apply(1,1)); //2
        System.out.println(operations.get("SUB").apply(3,5)); //-2
        System.out.println(operations.get("ADD").apply(3,
            operations.get("SUB").apply(3,
                operations.get("MUL").apply(2,6))
            )
        ); //-6
    }
}
```

Anonymous

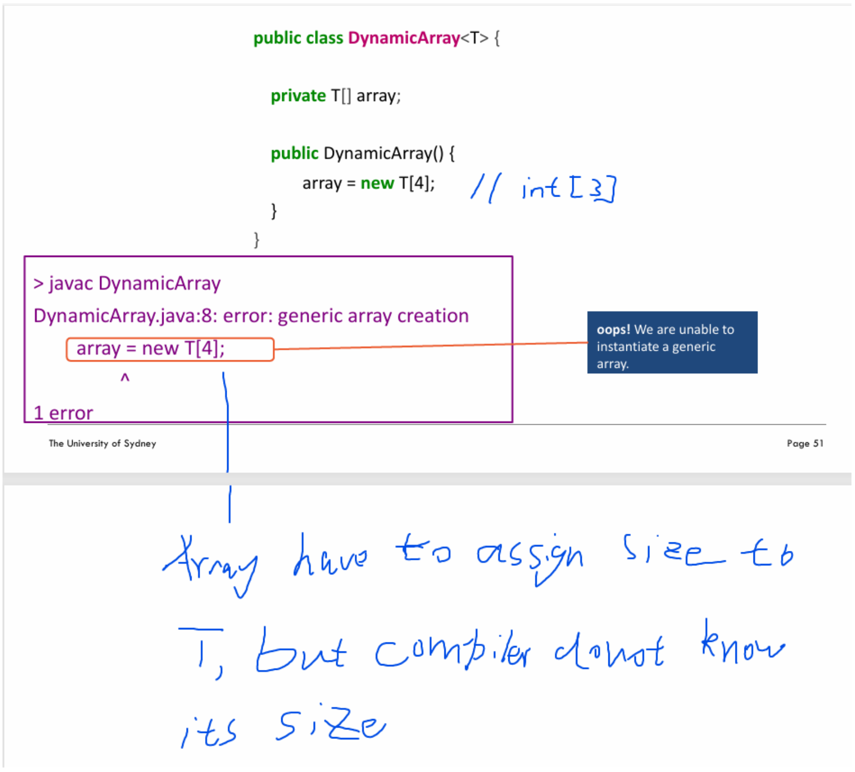


```
2. 定义新的method inside Anonymous class body
interface Animal {
    void sound();
}

public class Main {
    public static void main(String[] args) {
        Animal dog = new Animal() {
            // 实现接口中的方法
            @Override
            public void sound() {
                System.out.println("Dog barks");
                secretMethod(); // 可以在匿名类内部调用新方法
            }

            // 新方法，无法通过接口的引用访问
            public void secretMethod() {
                System.out.println("This is a secret method!");
            }
        };

        dog.sound();  // 输出 "Dog barks" 和 "This is a secret method!"

        // 无法通过接口引用调用新方法，编译错误
        // dog.secretMethod();  // 编译错误: 无法找到 secretMethod
    }
}
```

- 可以用 class, abstract class, interface 实现
- 注意最后要在}后面加上 ;
- 最好父类都是无参构造函数

Generic type

- 不管什么时候使用都要用<>标明，比如自定义了 Node<k,v> 那么就要用 for(Node<k,v> node; nodes){}
- 不能直接 T 数组，会有 compile Exception



```
public class DynamicArray<T> {

    private T[] array;

    public DynamicArray() {
        array = new T[4];   // int[3]
    }
}
```

```
> javac DynamicArray
DynamicArray.java:8: error: generic array creation
    array = new T[4];
         ^
1 error
```

Array have to assign size to T, but compiler donot know its size

```
array = (T[]) new Object[4]; // 使用 Object 数组并强制转换
```

```
public <T> void exampleMethod() {
    ArrayList<T> list = new ArrayList<>();  // 使用钻石操作符，类型由T推断
    // 你可以将list添加不同类型的元素，前提是它们符合T类型的约束
}
```

- implement generic interface

public class ListContainer<T> implements Container<K> {可以嘛

不可以。`public class ListContainer<T> implements Container<K>` 这样的写法是无效的，原因在于：

1. **泛型类和接口的类型参数必须匹配：**
   在 `ListContainer<T>` 中，`T` 是类型参数，代表 `ListContainer` 类的类型参数。而 `Container<K>` 是接口的类型参数，代表 `Container` 接口的类型参数。如果你使用 `Container<K>`，你需要确保接口和类中都使用相同的类型参数。

- multiple generic for 泛型类

```java
public class Pair<K, V> {
    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }
}
```

In this `Pair` class, `K` and `V` are type parameters representing the types of the key and value, respectively. This class can be instantiated with any types:

```java
Pair<String, Integer> student = new Pair<>("Alice", 90);
System.out.println("Student: " + student.getKey() + ", Score: " + student.getValue());
```

- 泛型方法，注意要显示声明，下面期望传入的 array 里面包含同种元素 T，所以不能传入{1, 2, "aaa", 3}

```java
public static <T> void printArray(T[] array) {
    for (T element : array) {
        System.out.print(element + " ");
    }
    System.out.println();
}
```

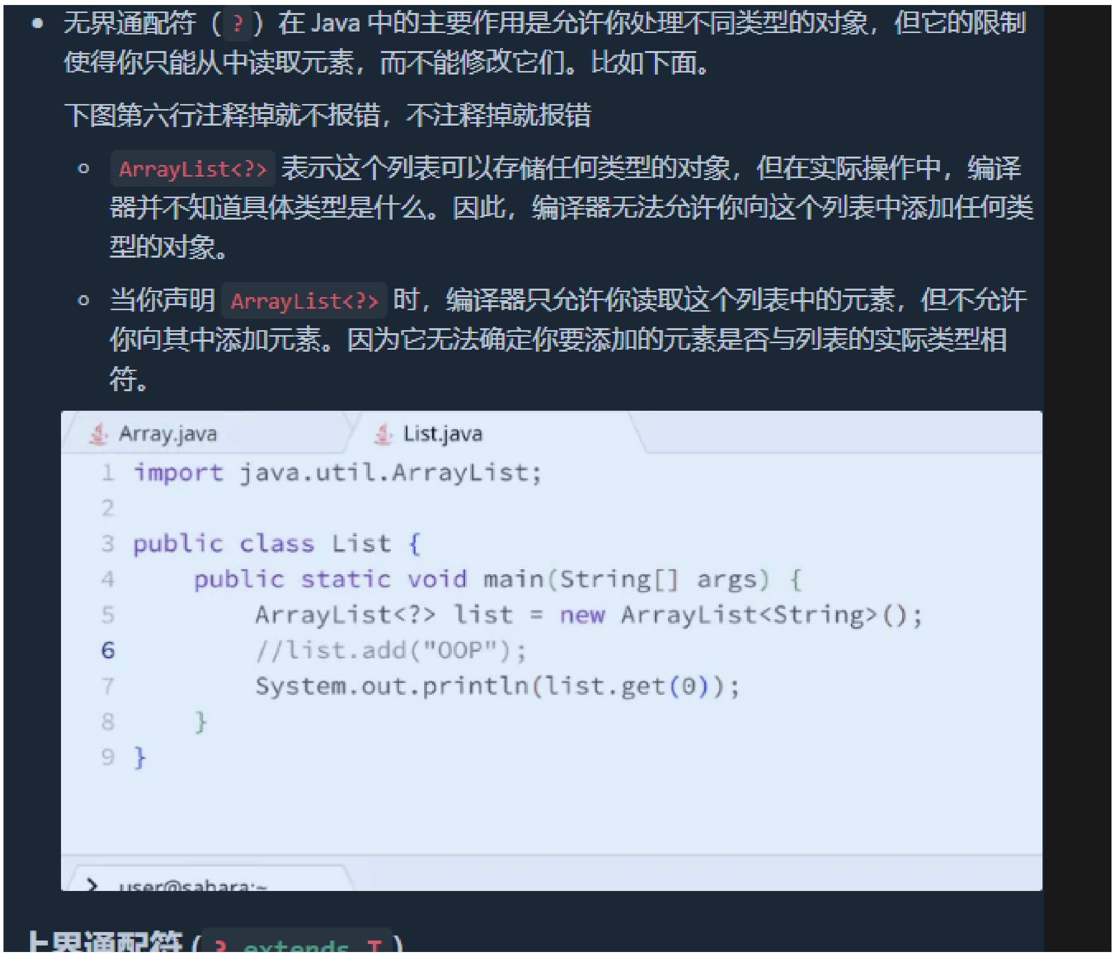
- bounded generic

```java
public <T extends Comparable<T>> T getMax(T a, T b) {
    return a.compareTo(b) > 0 ? a : b;
}
```

```java
public class Box<T extends Number> {
    private T value;

    // 构造方法
    public Box(T value) {
        this.value = value;
    }
```

Wildcard

- 无界通配符？一般用来读



- 无界通配符（`?`）在 Java 中的主要作用是允许你处理不同类型的对象，但它的限制使得你只能从中读取元素，而不能修改它们。比如下面。

  下图第六行注释掉就不报错，不注释掉就报错

  - `ArrayList<?>` 表示这个列表可以存储任何类型的对象，但在实际操作中，编译器并不知道具体类型是什么。因此，编译器无法允许你向这个列表中添加任何类型的对象。
  - 当你声明 `ArrayList<?>` 时，编译器只允许你读取这个列表中的元素，但不允许你向其中添加元素。因为它无法确定你要添加的元素是否与列表的实际类型相符。

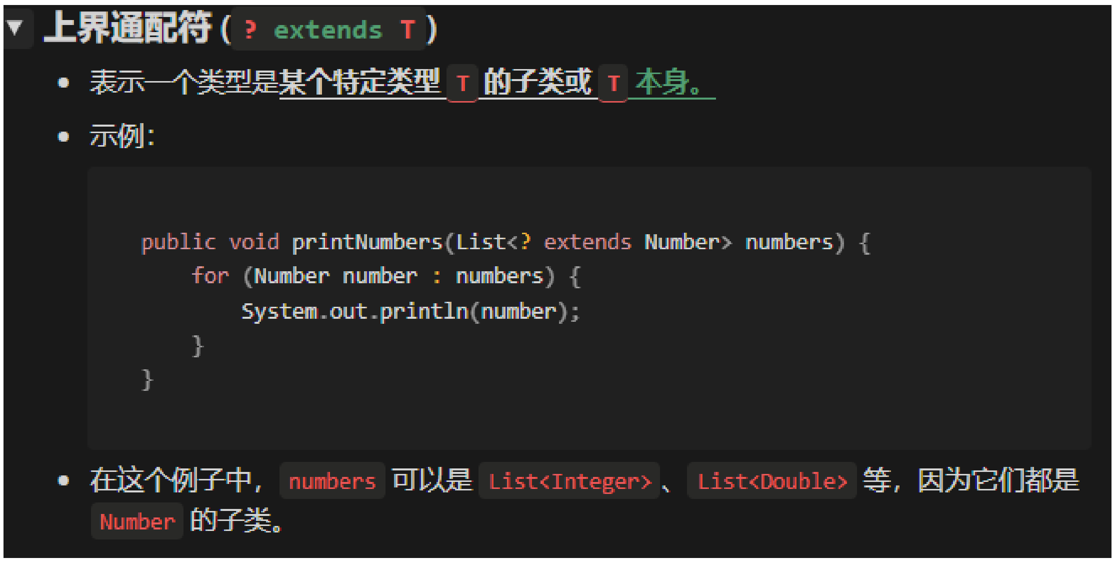
Array.java   List.java

```
1 import java.util.ArrayList;
2
3 public class List {
4     public static void main(String[] args) {
5         ArrayList<?> list = new ArrayList<String>();
6         //list.add("OOP");
7         System.out.println(list.get(0));
8     }
9 }
```

> user@sahara:~

上界通配符 (? extends T)

- upperbound -- extends



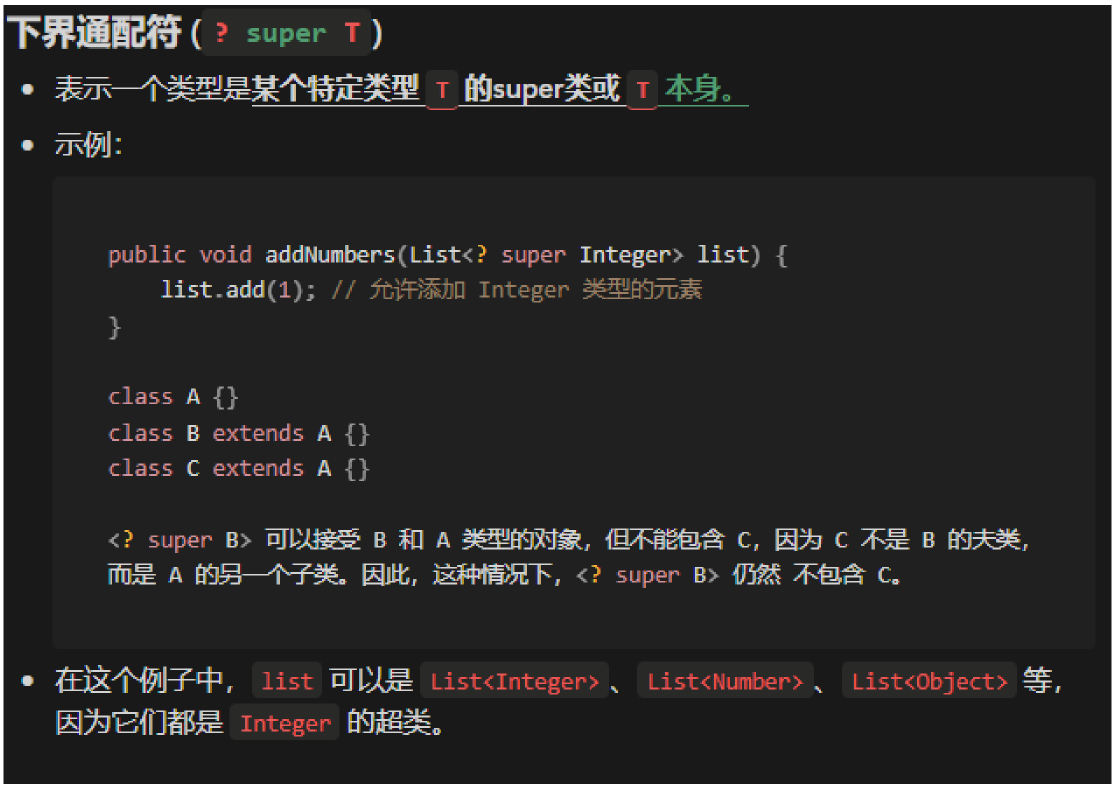
上界通配符 (`? extends T`)

- 表示一个类型是某个特定类型 `T` 的子类或 `T` 本身。
- 示例：

```
public void printNumbers(List<? extends Number> numbers) {
    for (Number number : numbers) {
        System.out.println(number);
    }
}
```

- 在这个例子中，`numbers` 可以是 `List<Integer>`、`List<Double>` 等，因为它们都是 `Number` 的子类。

- lowerbound -- super



下界通配符（`? super T`）

- 表示一个类型是某个特定类型 `T` 的super类或 `T` 本身。
- 示例：

```
public void addNumbers(List<? super Integer> list) {
    list.add(1); // 允许添加 Integer 类型的元素
}

class A {}
class B extends A {}
class C extends A {}

<? super B> 可以接受 B 和 A 类型的对象，但不能包含 C，因为 C 不是 B 的夫类，
而是 A 的另一个子类。因此，这种情况下，<? super B> 仍然 不包含 C。
```

- 在这个例子中，`list` 可以是 `List<Integer>`、`List<Number>`、`List<Object>` 等，因为它们都是 `Integer` 的超类。

```
class Inventory<T> {
    private List<T> items = new ArrayList<>();

    public void addItem(T item) {
        items.add(item);
    }

    public List<? extends T> getItems() {
        return items;
    }
}
```

In this `Inventory` class, `addItem` allows adding items of type `T`, while `getItems` returns a list of `T` or any subtype of `T`. This flexibility makes the `Inventory` system adaptable, allowing it to handle different types of items while maintaining type safety.

Here's how you might use `Inventory` with specific item types:

泛型方法：方法的类型参数是在方法签名中声明的。例如：

```java
public <T> void printArray(T[] array) {
    for (T element : array) {
        System.out.print(element + " ");
    }
}
```

泛型类：类的类型参数是在类声明时声明的，这些类型参数在整个类中都可以使用。类的泛型参数会在方法内部直接使用，而不需要在方法中重新声明。

例如：

```java
public class Box<T> {
    private T item;

    public void setItem(T item) {
        this.item = item;
    }

    public T getItem() {
        return item;
    }
}
```

在这个例子中，`T` 是类 `Box` 的泛型类型参数，所以 `setItem` 和 `getItem` 方法中都直接使用了 `T`。

Java Collection

注意，没有 array

- **List**: An **ordered collection** that allows **duplicate elements**, such as a list of items in a shopping cart.
- **Set**: An **unordered collection** that does **not allow duplicates**, such as a set of unique usernames.
- **Queue**: A collection that holds elements to be processed in a specific order, like tasks in a processing queue.
- **Map**: A collection of key-value pairs, like a dictionary, where each key is associated with a specific value.

Junit

- `import org.junit.jupiter.api.*;`

```java
@Test
void testAddition() {
    int result = 2 + 3;
    Assertions.assertEquals(5, result, "Expected 5 as the result of 2 + 3");
}
```

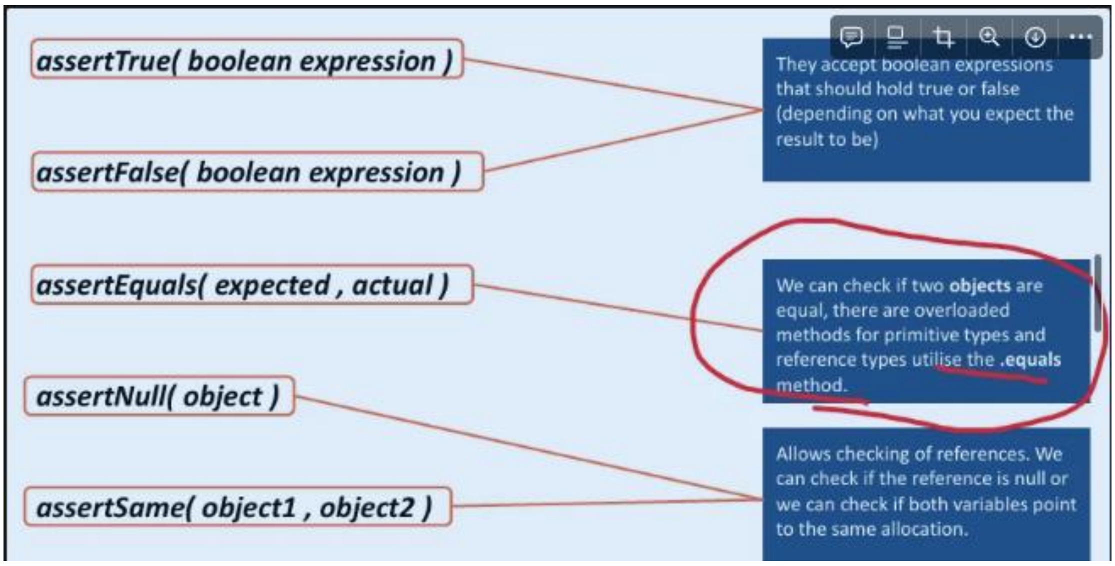
```java
@Test
void testSubtraction() {
    int result = 10 - 5;
    Assertions.assertNotEquals(10, result, "Result should not be 10");
}
```

```java
@Test
void testIsLegalAge() {
    int age = 20;
    Assertions.assertTrue(age >= 18, "Age should be at least 18 to be considered legal");
}
```

```java
@Test
void testIsNonZero() {
    int number = -1;
    Assertions.assertFalse(number == 0, "Number should not be zero");
}
```

```java
@Test
void testItemNotFound() {
    String item = null; // Simulating a search result where the item wasn't found
    Assertions.assertNull(item, "Item should be null if not found");
}
```

```java
@Test
void testUserProfileFound() {
    String profile = "User Profile Data"; // Simulating a successful profile retrieval
    Assertions.assertNotNull(profile, "Profile should not be null when found");
}
```



assertTrue(boolean expression)

assertFalse(boolean expression)

They accept boolean expressions that should hold true or false (depending on what you expect the result to be)

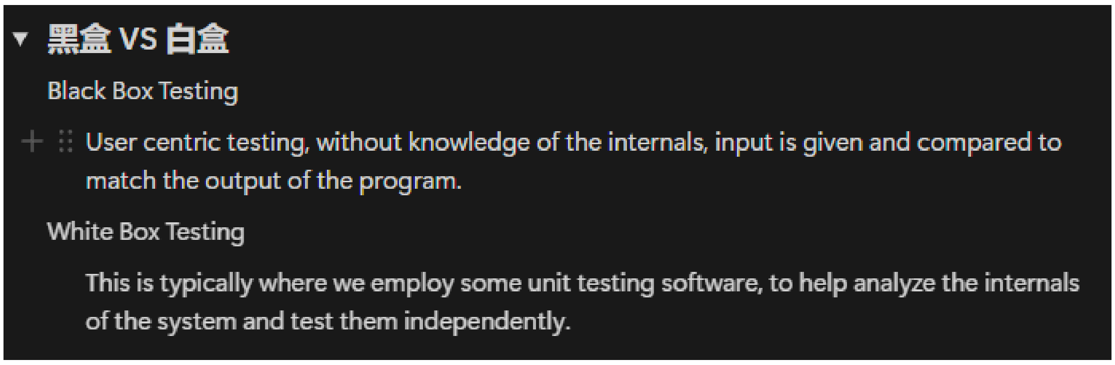
assertEquals(expected , actual)

We can check if two **objects** are equal, there are overloaded methods for primitive types and reference types utilise the **.equals** method.

assertNull(object)

assertSame(object1 , object2)

Allows checking of references. We can check if the reference is null or we can check if both variables point to the same allocation.



黑盒 VS 白盒

Black Box Testing

User centric testing, without knowledge of the internals, input is given and compared to match the output of the program.

White Box Testing

This is typically where we employ some unit testing software, to help analyze the internals of the system and test them independently.

throws VS throw

If a custom exception is a checked exception (extends Exception but not RuntimeException), it can be declared with the throws keyword instead of being caught.

If it's an unchecked exception (extends RuntimeException), it does not require handling or declaring.

Run time Exception unchecked

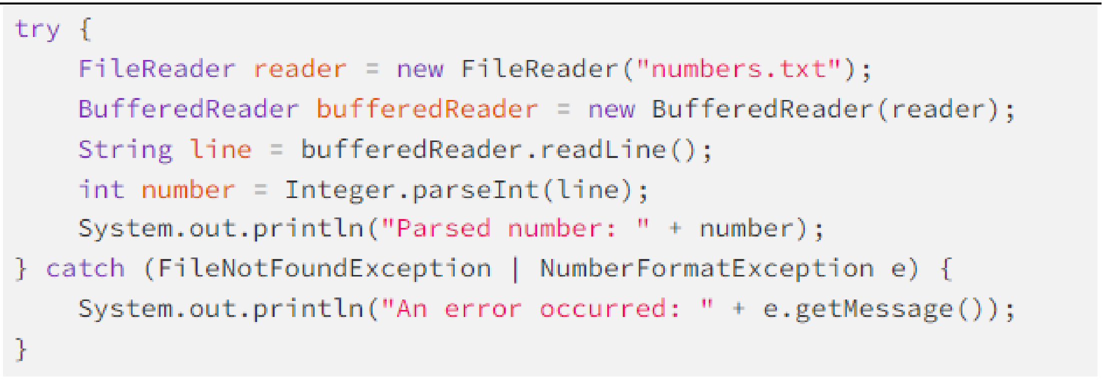
- we cannot determine them on compile time because they are Grammarly correct Such as null pointer, zero division, fille not found
- 自定义的话需要 extends RuntimeException

Compile time Exception checked

- syntax error, import package error,
- 自定义的话需要 extends Exception

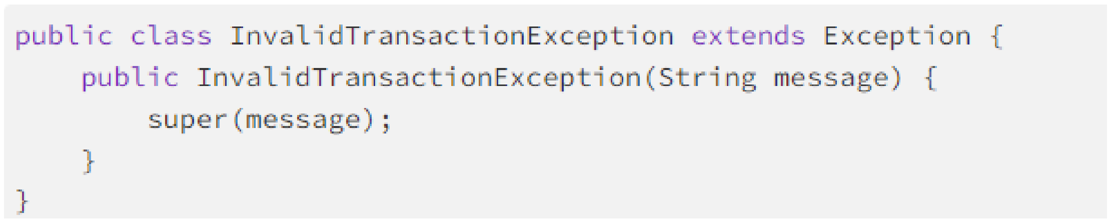
Error, cannot be handle

Stackoverflow



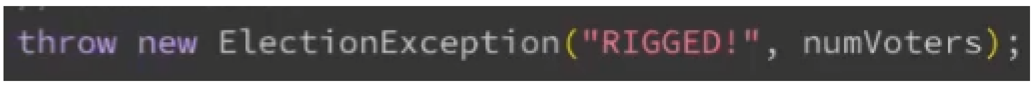
```
try {
    FileReader reader = new FileReader("numbers.txt");
    BufferedReader bufferedReader = new BufferedReader(reader);
    String line = bufferedReader.readLine();
    int number = Integer.parseInt(line);
    System.out.println("Parsed number: " + number);
} catch (FileNotFoundException | NumberFormatException e) {
    System.out.println("An error occurred: " + e.getMessage());
}
```

- customized exception



```
public class InvalidTransactionException extends Exception {
    public InvalidTransactionException(String message) {
        super(message);
    }
}
```

- Exception 要 new

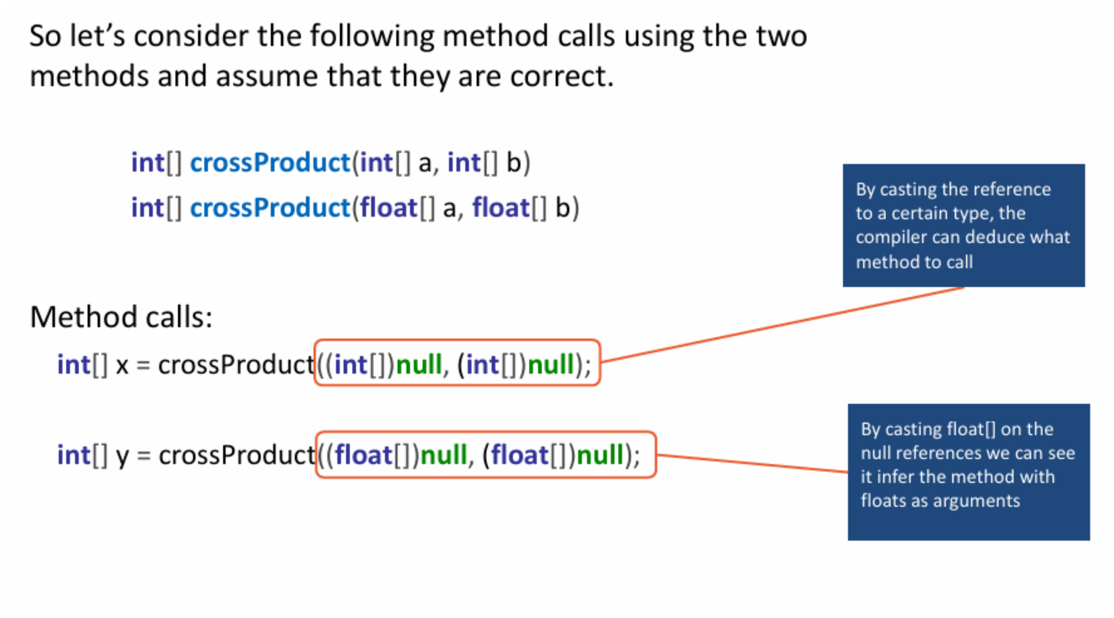


```
throw new ElectionException("RIGGED!", numVoters);
```

看清楚题目的要求，如果是 method 需要 throws error，则应该写成下面这样

```java
class InvalidGPAException extends Exception {
    public InvalidGPAException(String message) {
        super(message);
    }
}

public class Student {
    public double calculateGPA(int totalCredits, int totalGradePoints) throws InvalidGPAException {
        if (totalCredits == 0) {
            throw new InvalidGPAException("Total credits cannot be zero when calculating GPA.");
        }
        return (double) totalGradePoints / totalCredits;
    }

    public static void main(String[] args) {
        Student student = new Student();
        try {
            double gpa = student.calculateGPA(0, 120);
            System.out.println("GPA: " + gpa);
        } catch (InvalidGPAException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Overload



So let's consider the following method calls using the two methods and assume that they are correct.

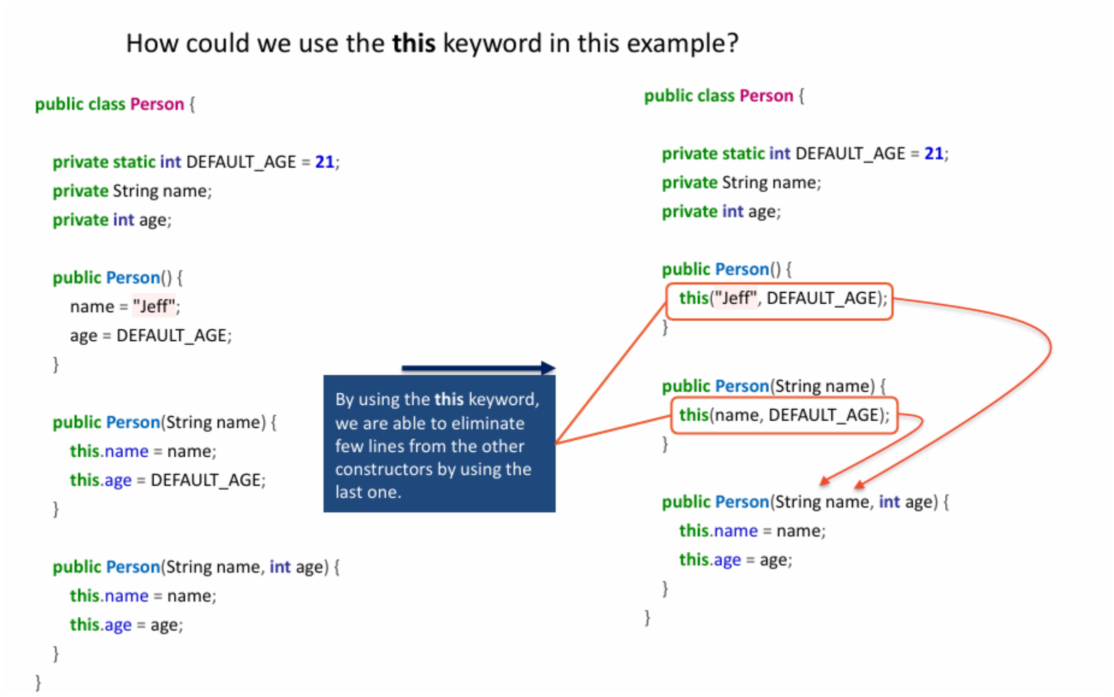
```java
int[] crossProduct(int[] a, int[] b)
int[] crossProduct(float[] a, float[] b)
```

Method calls:

```java
int[] x = crossProduct((int[])null, (int[])null);

int[] y = crossProduct((float[])null, (float[])null);
```

By casting the reference to a certain type, the compiler can deduce what method to call

By casting float[] on the null references we can see it infer the method with floats as arguments



How could we use the **this** keyword in this example?

```java
public class Person {

  private static int DEFAULT_AGE = 21;
  private String name;
  private int age;

  public Person() {
    name = "Jeff";
    age = DEFAULT_AGE;
  }

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

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

By using the **this** keyword, we are able to eliminate few lines from the other constructors by using the last one.

```java
public class Person {

  private static int DEFAULT_AGE = 21;
  private String name;
  private int age;

  public Person() {
    this("Jeff", DEFAULT_AGE);
  }

  public Person(String name) {
    this(name, DEFAULT_AGE);
  }

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