

COMP 2021 OOP

COMP 2021 OOP

00. Interpreter vs. Compiler

Java Compiler

Java Interpreter

Distinction

01. Java Basic

1.1 命名

1.2 byte 数据类型简介

1.3 参数传递(Argument Passing)

1.4 Variable Scope

1.5 static

1.6 单例模式 (Singleton Pattern)

1.7 classpath

1.8 package

1.9 Constant

1.10 Escape Sequence

1.11 Character Operations

1.12 Formatting Output

1.13 Array

1.14 Multi-dimensional Arrays

1.15 BigInteger and BigDecimal

1.16 Unit Testing with JUnit

1.17 Shallow/deep Copy of Objects

1.18 Iterator(范型+instanceof使用)

1.19 Java Container

02. Class and Object

2.1 Class Definition

2.2 Primitive Type vs. Reference Type

2.3 Class Relationships

2.4 封装类 (Wrapper Classes)

Autoboxing and Unboxing

03. 封装 (Encapsulation)

04. 继承 (Inheritance)

4.1 继承

4.2 super 与 this 关键字

4.3 java.lang.Object class

public String toString()

public boolean equals(Object other)

HashSet 存储对象的无序数组

public int hashCode()

public Object clone()

05. 多态 (Polymorphism)

5.1 多态

instanceof (运行阶段动态判断)

5.1 Overloading & Overriding

5.2 Static Binding and Dynamic Binding

5.3 Abstract class
5.4 Interface
5.5 Implement
5.6 Interface and Polymorphism
5.7 extends 和 implements 的区别
5.8 Abstract class 和 interface 区别

06. Exception

常见异常类型：
try / catch / finally
Enum
自定义异常 throws 和 throw

07. Generics

Generics 范型
Type Inference类型自动判断
Multiple Type Parameters
Raw type
Bounded Type Parameters
Generics, Inheritance, and Subtypes
Wildcards(通配符)

08. Unified Modeling Language (UML)

General relationship
Instance-level
Navigability of Association
Abstract and Interface Classes
Class-level
UML Object Diagrams
Example Class Diagram
UML Example

08. Thread

Thread Methods
Processes and Threads
实现线程
Concurrency(并发)
线程的调度(Coordination of Threads)
wait(), notify(), and notifyAll()
The Producer-Consumer
Thread States
守护线程 setDaemon(true) // 不考

09. Swing // 不考

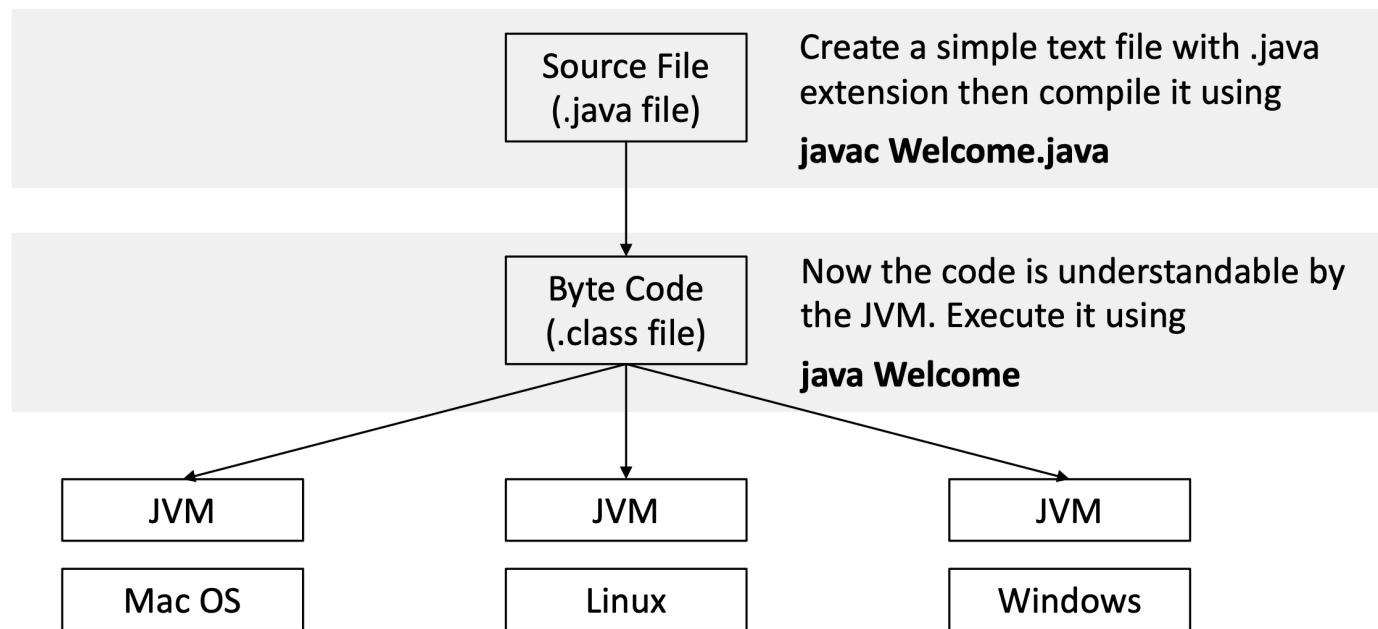
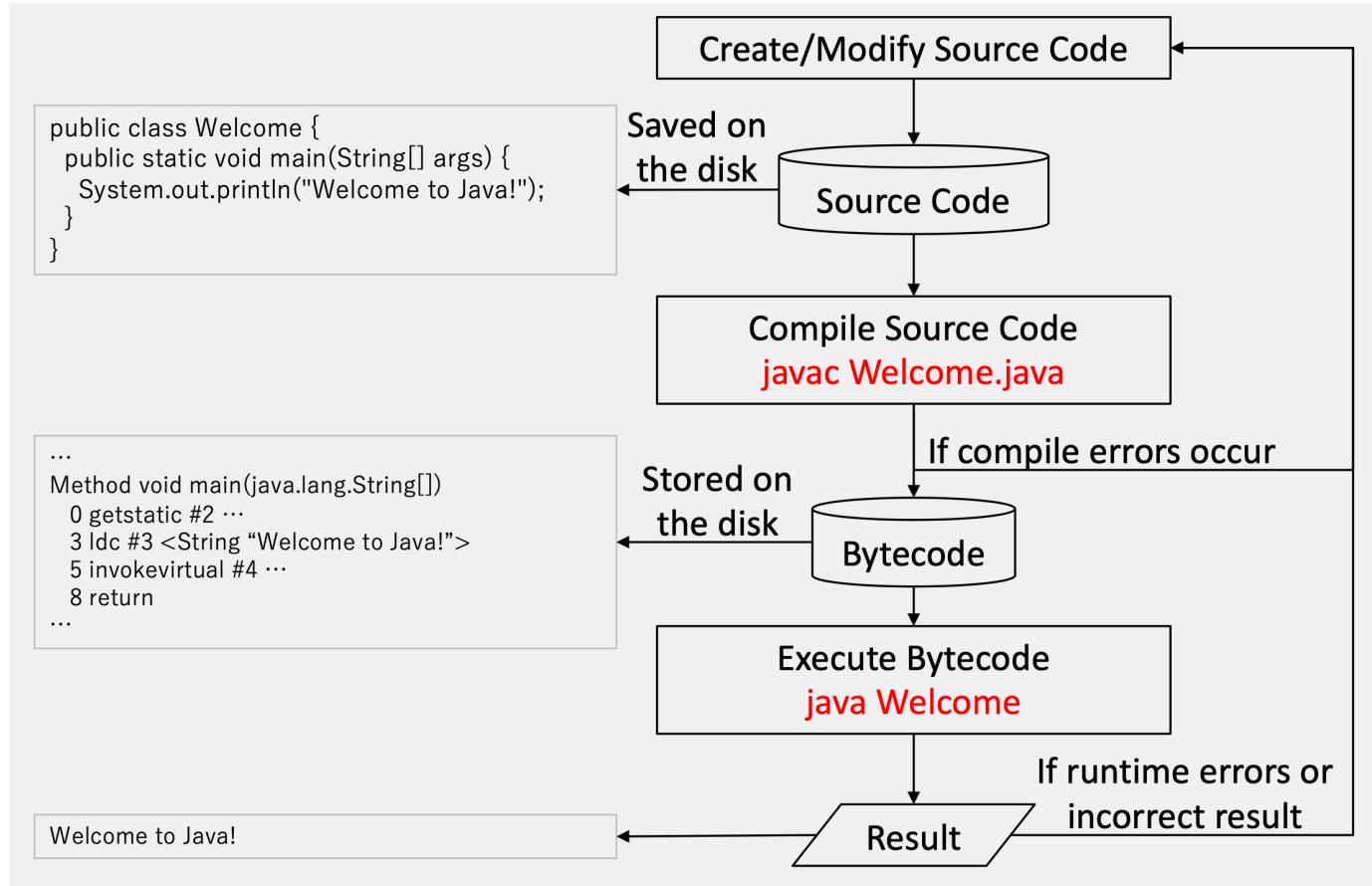
JFrame
JDialog
Swing's Components
Using Top-Level Containers
Adding Components to the Content Pane
Layout Managers and Panel

10. Nested Class, Local Class, Lambda Expression

Nested Classes
Local Class
Lambda Expression

00. Interpreter vs. Compiler

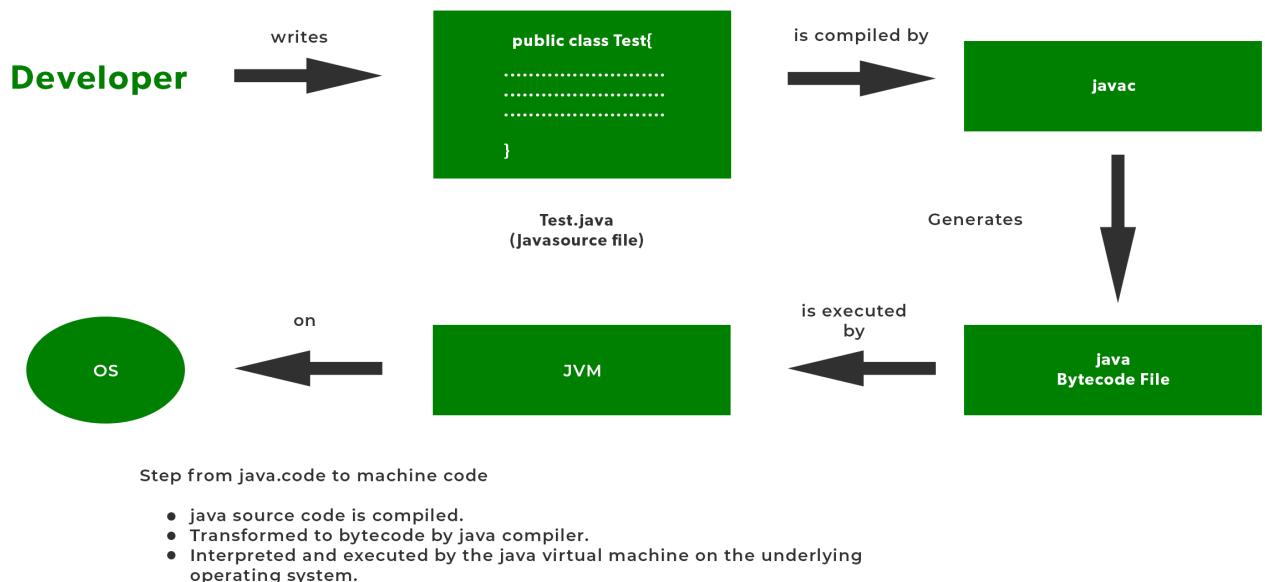
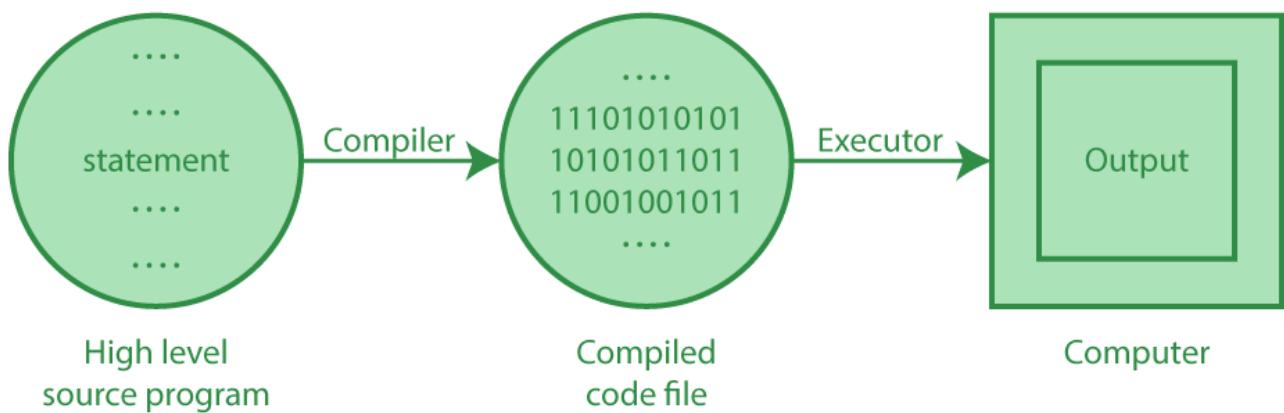
Compiler 和 Interpreter 是将程序从编程或脚本语言翻译成机器语言的两种不同方法。



Java Compiler

- 它一次性扫描完整的源代码，然后突出显示错误。
- 它需要更多的内存来生成字节码。
- 它通过检查类型错误、语法等来检查程序的正确性。
- 如果需要，它还会向我们的程序添加一些额外的代码。

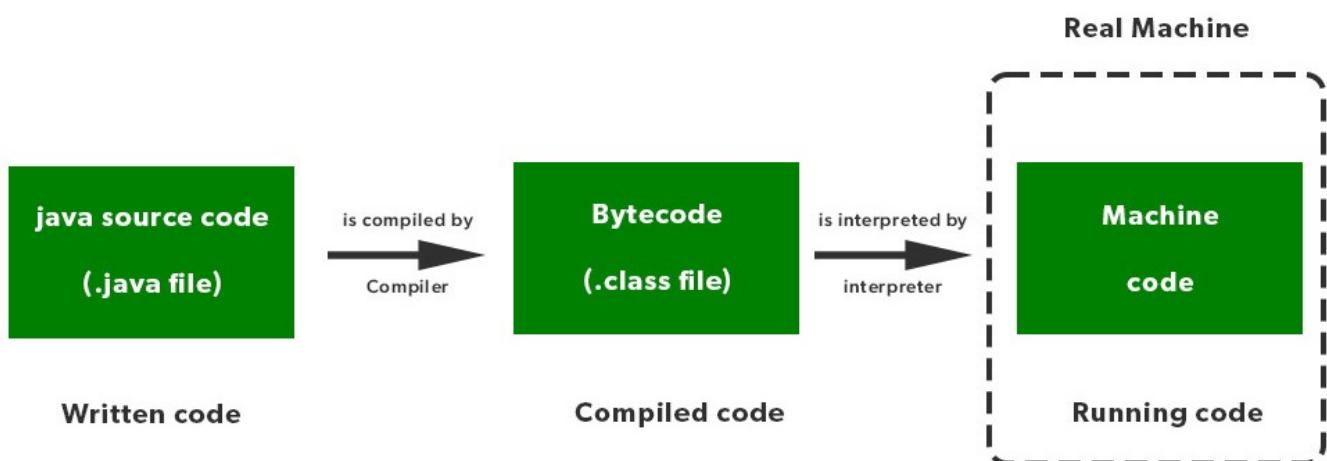
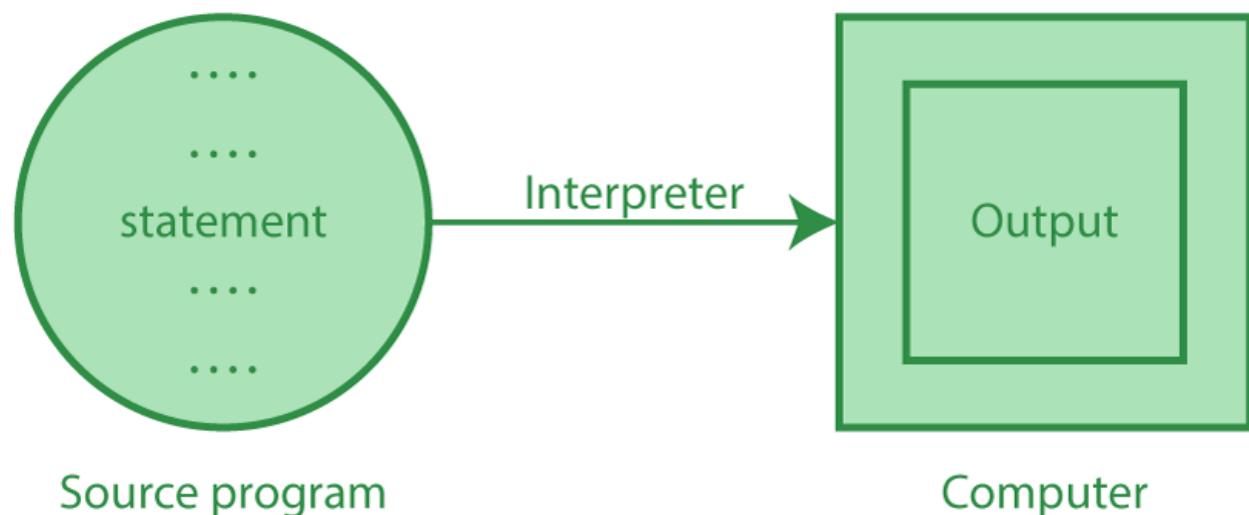
Java 中的编译器将整个源代码翻译成机器代码文件或任何中间代码，然后执行该文件。它与平台无关。字节码基本上是编译器在编译其源代码后生成的中间代码。Java 编译器可以在命令提示符下使用“javac.exe”命令进行操作。下面给出了一些编译器选项：



Java Interpreter

- 将字节码转换为机器的本机代码。
- 这个过程是逐行完成的。
- 如果错误出现在任何一行，则该过程将在那里停止。

同样，解释器是将高级程序语句转换为汇编级语言的计算机程序。它在程序运行时将代码转换为机器代码。下面给出了一些解释器选项：



Distinction

- **interpreter** 逐行扫描程序并将其转换为机器代码，而 **compiler** 首先扫描整个程序，然后将其转换为机器代码。
- **interpreter** 一次显示一个错误，而 **compiler** 同时显示所有错误和警告。
- 在 **interpreter** 中，错误发生在扫描每一行之后，而在 **compiler** 中，错误发生在扫描整个程序之后。
- 在 **interpreter** 中，调试速度更快，而在 **compiler** 中，调试速度很慢。
- **interpreter** 中的执行时间更多，而 **compiler** 中的执行时间更少。
- Java、Python等语言使用 **interpreter**，C、C++等语言使用 **compiler**

01. Java Basic

Access modifier	same class	same package	diff pakage	任意位置
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
Default	✓	✓	✗	✗
private	✓	✗	✗	✗

范围从大到小排序: public > protected > Default > private

1.1 命名

Class

- 应该以大写字母开头
- 应该是一个名词
- 使用合适的词汇, 而不是缩写词

```
public class Employee {  
}
```

Interface

- 应该以大写字母开头。
- 它应该是一个形容词, 如Runnable, Remote, ActionListener。
- 使用合适的词汇, 而不是缩写词。

```
interface Printable {  
}
```

Method

- 它应该以小写字母开头。
- 它应该是一个动词, 如main(), print(), println()。
- 如果名称包含多个单词, 则以小写字母和大写字母开头, 例如actionPerformed()

```
class Employee  
{  
// method  
void draw(){  
  
}  
}
```

Variable

变量可以由字母、数字、下划线(_)和美元符号(\$)组成

```
int _2 = 10;
int $a = 10;
int _a = 10;
int $1 = 20;
```

Package

Package 名称全部小写，以避免与类或接口(classes or interfaces)的名称冲突。

```
// Hero.java
package hk.edu.polyu.comp.comp2021.example;

class Hero{}
```

Constant

Constant 应该是大写字母，如RED, YELLOW。

如果名称包含多个单词，则应该用下划线(_)分隔，例如MAX_PRIORITY。

它可以包含数字，但不能作为第一个字母。

```
class Employee {
    //constant
    static final int MIN AGE = 18;
}
```

1.2 byte数据类型简介

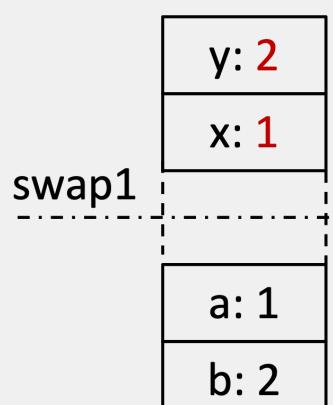
byte是四个整数类型 (byte、short、int、long) 中取值范围最小的整型数据类型，具体如下：

数据类型名称	内存大小	取值范围
byte	8b(1B)	-128~127(2^8)
short	16b(2B)	-32768~32767(2^{16})
int	32b(4B)	-2147483648~2147482647(2^{32})
long	64b(8B)	-9223372036854775808~9223372036854775807(2^{64})

1.3 参数传递(Argument Passing)

For **primitives**, the primitive values are copied

```
void swap1(int x, int y){  
    int z = x;  
    x = y;  
    y = z;  
}  
  
...  
  
int a = 1, b = 2;  
swap1(a, b);  
System.out.println(a+", "+b);
```



1,2

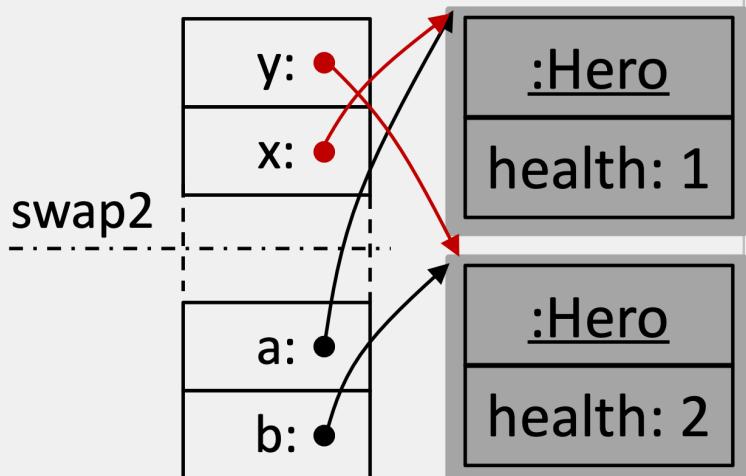
For **references**, the addresses are copied

```

void swap2(Hero x, Hero y){
    int z = x.health;
    x.health = y.health;
    y.health = z;
}

```

...



```

// suppose a and b have health 1 and 2
swap2(a, b);
System.out.println(a.health+", "+b.health);

```

2,1

6

1.4 Variable Scope

属性 (Field): 声明它的整个class

局部变量 (Local variable): 从声明语句到block的末尾

形参 (Formal parameter): 本质是一个名字, 不占用内存空间, 不是实际存在变量, 声明它的整个method

实参 (argument): 是在调用时传递给method的参数。实参可以是常量、变量、表达式、函数等, 无论实参是什么类型的量, 在进行函数调用时, 它们都必须具有确定的值, 以便把这些值传送给形参。

```

// Example
add(int a, int b){
    // 这里的a和b就是 形参 (Formal parameter)
}

add(1, 2) // 这里的1和2就是实参 (argument)

```

1.5 static

static关键字的用途：

- 方便在没有创建对象的情况下进行调用(方法/变量)。
- 显然，被static关键字修饰的方法或者变量不需要依赖于对象来进行访问，只要类被加载了，就可以通过类名去进行访问。
- static可以用来修饰类的成员方法、类的成员变量，另外也可以编写static代码块来优化程序性能
- static方法也成为静态方法，由于静态方法不依赖于任何对象就可以直接访问，因此对于静态方法来说，是没有this的，因为不依附于任何对象，既然都没有对象，就谈不上this了，并且由于此特性，在静态方法中不能访问类的非静态成员变量和非静态方法，因为非静态成员变量和非静态方法都必须依赖于具体的对象才能被调用。

```
public class Test extends Base{  
    static{  
        System.out.println("test static");  
    }  
    public Test(){  
        System.out.println("test constructor");  
    }  
    public static void main(String[] args) {  
        new Test();  
    }  
}  
  
class Base{  
    static{  
        System.out.println("base static");  
    }  
    public Base(){  
        System.out.println("base constructor");  
    }  
}
```

```
// output:  
base static  
test static  
base constructor  
test constructor
```

1.6 单例模式 (Singleton Pattern)

单例模式 (Singleton Pattern) 是 Java 中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

注意：

- 1、单例类只能有一个实例。
- 2、单例类必须自己创建自己的唯一实例。
- 3、单例类必须给所有其他对象提供这一实例。

意图：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

主要解决：一个全局使用的类频繁地创建与销毁。

何时使用：当您想控制实例数目，节省系统资源的时候。

如何解决：判断系统是否已经有这个单例，如果有则返回，如果没有则创建。

关键代码：构造函数是私有的。

```
public class SingleObject {  
  
    //创建 SingleObject 的一个对象  
    private static SingleObject instance = new SingleObject();  
  
    //让构造函数为 private, 这样该类就不会被实例化  
    private SingleObject(){}  
  
    //获取唯一可用的对象  
    public static SingleObject getInstance(){  
        return instance;  
    }  
  
    public void showMessage(){  
        System.out.println("Hello World!");  
    }  
}
```

```
// 从 singleton 类获取唯一的对象  
public class SingletonPatternDemo {  
    public static void main(String[] args) {  
  
        //不合法的构造函数  
        //编译时错误：构造函数 SingleObject() 是不可见的  
        //SingleObject object = new SingleObject();  
  
        //获取唯一可用的对象  
        SingleObject object = SingleObject.getInstance();  
  
        //显示消息  
        object.showMessage();  
    }  
}
```

```
}
```

```
// 执行程序，输出结果：  
Hello World!
```

1.7 classpath

默认的类装入器使用“classpath”中定义的路径列表来搜索类。

A classpath is just a group of paths

```
...\\jdk\\src;...\\mylib\\src; // On Windows  
.../jdk/src:.../mylib/src // On MacOS.
```

可以在环境变量中设置或传递给Java工具作为一个参数

```
set classpath =...\\jdk\\lib;...\\mylib1;  
java -classpath ...\\jdk\\lib;...\\mylib2 <other parameters>
```

1.8 package

为了便于管理大型软件系统中数目众多的类，解决类命名冲突的问题，Java引入了包 (package)

Fully Qualified Name (FQN): packagename.ClassName

The screenshot shows a Java development environment with a project named 'Tutorial'. The 'lab04_Brackets' class is selected in the project tree. The code editor displays the following Java code:

```
package Tutorial;  
  
/*  
 * From {@code comp2011.lec3.Brackets}.  
 */  
public class lab04_Brackets {  
    // Running time: O( n ).  
    2 usages  
    public static boolean isBalanced(char[] s) {  
        int count = 0;  
        for (int i = 0; i < s.length; i++) {  
            if (s[i] == '(') count++;  
            if (s[i] == ')') count--;  
            if (count < 0) return false;  
        }  
        return count == 0;  
    }  
}
```

1.9 Constant

```
public class MyMath{  
    public static final double PI = 3.14;  
}
```

final 关键字

浓缩成一句话：final修饰的东西不能变

final 可以用来修饰变量（包括类属性、对象属性、局部变量和形参）、方法（包括类方法和对象方法）和类。

- final修饰的类无法继承（使用 final 关键字声明类，就是把类定义定义为最终类，不能被继承）
- final修饰的方法无法覆盖，（如果final 用于修饰方法，该方法不能被子类重写）
- final修饰的变量只能赋一次值
- final修饰的引用一旦指向某个对象，则不能再重新指向其它对象，但该引用指向的对象内部的数据是可以修改的
- final修饰的实例变量必须手动初始化，不能采用系统默认值。
- final修饰的实例变量一般和static联合使用，称为常量。

1.10 Escape Sequence

转义字符串 (Escape Sequence)

Escape Sequence	Name
\b	Backspace
\t	Tab
\n	Linefeed
\r	Carriage Return
\\"	Backslash
\\"	Double quote

```
System.out.println("Double quote \" in String literal");  
System.out.println("Backslash \\ in String literal");
```

```
// Output  
Double quote " in String literal  
Backslash \ in String literal
```

1.11 Character Operations

字符操作 (Character Operations)

Casting between char and numeric types

```
int i = 'a'; // i = 97  
char c = (char) i; //c = 'a'
```

Comparing and testing characters

```
if (c >= 'A' && c <= 'Z')  
    System.out.println(c + " is an uppercase letter");  
else if (c >= 'a' && c <= 'z')  
    System.out.println(c + " is a lowercase letter");  
else if (c >= '0' && c <= '9')  
    System.out.println(c + " is a numeric character");
```

Methods in the Character class

Method	Method
Character.isDigit(c)	Character.isLowerCase(c)
Character.isLetter(c)	Character.isUpperCase(c)
Character.isLetterOrDigit(c)	Character.toLowerCase(c)
	Character.toUpperCase(c)

15

String

```
String s1 = "Java ", s2 = "!";
```

Method	Description	Result
s1.length()	Returns the number of characters in s1	5
s1.charAt(0)	Returns the character at the specified index from s1	'J'
s1.concat(s2)	Returns a new string that concatenates s1 with s2 Note: the same as s1 + s2	"Java !"
s1.toUpperCase()	Returns a new string with all letters in uppercase	"JAVA "
s1.toLowerCase()	Returns a new string with all letters in lowercase	"java "
s1.trim()	Returns a new string with whitespace characters trimmed on both sides	"Java"

```
String s1 = "Java", s2 = "va";
```

Method	Description	Result
s1.equals(s2)	Returns true if s1 and s2 have the same content Note s1 == s2 tests something different	false
s1.startsWith(s2)	Returns true if s1 starts with s2	false
s1.endsWith(s2)	Returns true if s1 ends with s2	true
s1.substring(1, 3)	Returns the substring of s1 starting from position 1 and ending at position 2 (= 3 - 1) Note s1.charAt(3) is not included in the substring	"av"
s1.indexOf('a', 0)	Returns the first index in s1 starting from 0 where 'a' appears	1
s1.indexOf(s2, 0)	Returns the first index in s1 starting from 0 where s2 appears	2
s1.lastIndexOf('v', 3)	Returns the last index in s1 before position 3 where 'a' appears	2
s1.lastIndexOf(s2, 3)	Returns the last index in s1 before position 3 where s2 appears	2

17

Conversion between Strings and Numbers

```
// String to numbers
int intValue = Integer.parseInt(intString);
double doubleValue = Double.parseDouble(doubleString);
```

```
// Numbers to String
String s1 = number + "";
String s2 = String.valueOf(number);

// + operator (conduct string concatenation if one operand is string)
int x = 1, y = 2;
String a = "a", b = "b";
// x+y = 3
// a+x = "a1"
// x+a = "1a"
// a+b = "ab"
```

StringBuilder

字符串是不可变的，也就是说，它们的内容一旦设置就永远不会改变

在Java的早期版本中，用于计算

```
String a = "a" + "b" + "c" + "d";
```

将创建7个字符串：

```
"a", "b", "c", "d", "ab", "abc", "abcd"
```

从Java 1.7开始，编译器将在上述表达式中使用StringBuilder进行String连接

```
StringBuilder builder = new StringBuilder("a");
builder.append("b").append("c").append("d");
String a = builder.toString(); // Convert to String
```

1.12 Formatting Output

格式化输出 (Formatting Output)

```
System.out.printf(format, items);
```

Specifier	Output	Example
%b	A Boolean value	true or false
%c	A character	'a'
%d	A decimal integer	30
%f	A floating point number	45.460000
%s	A string	"Java is cool"

```
String name = "Jack";
int count = 5;
float amount = 3.5f;
System.out.printf("Hi %s, count is %d, amount is %f!", name, count, amount);
// output: Hi Jack, count is 5, amount is 3.500000!
```

1.13 Array

Variable declaration

```
double[ ] myDoubles;
```

Array creation

```
myDoubles = new double[10]; // initialized to 0's
// or
double[ ] myEvenMoreDoubles = new double[20];
```

Equivalence (==), assignment and argument passing

```
myDoubles == myEvenMoreDoubles // compare reference
myDoubles = myEvenMoreDoubles; // assign reference
```

Length

```
int len = myDoubles.length; // !!! field, not method
```

Initializer

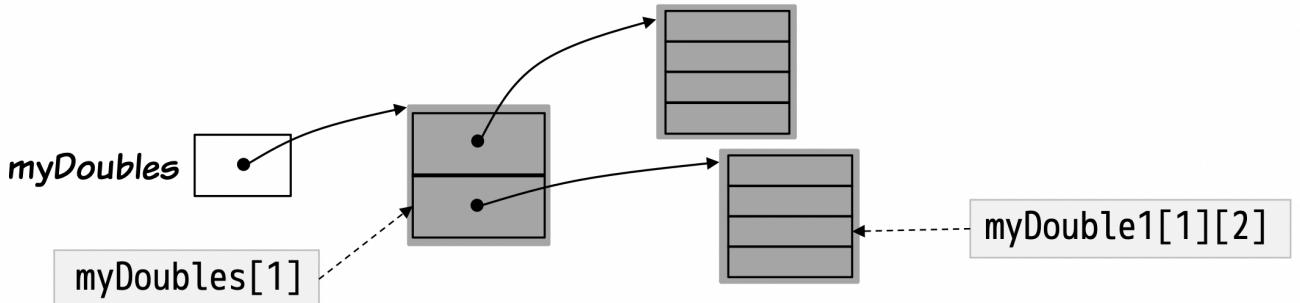
```
double[ ] myDoubles = {1.9, 2, 3.5, 9.1}; // all-in-one
double[] myDoubles;
myDoubles = {1.9, 2, 3.5, 9.1}; // error! not in assignment
```

1.14 Multi-dimensional Arrays

Array of arrays

```
double [][] myDoubles;  
myDoubles = new double[2][4];
```

- An array of 2 elements
- Each element is an array of 4 elements



Ragged Arrays

多维数组的每个元素本身就是一个数组，可能具有不同的长度

```
int[][] myIntegers = {  
    {1, 2, 3, 4, 5},  
    {2, 3, 4, 5},  
    {3, 4, 5},  
    {4, 5},  
    {5}  
};
```

```
int[][] myIntegers = new int[5][];  
myIntegers[0] = new int[5];  
myIntegers[1] = new int[4];  
myIntegers[2] = new int[3];  
myIntegers[3] = new int[2];  
myIntegers[4] = new int[1];
```

```
for(int i = 0; i < myIntegers.length; i++){  
    int[] ele = myIntegers[i];  
    for(int j = 0; j < ele.length; j++){  
        ...  
    }  
}
```

1.15 BigInteger and BigDecimal

To compute with very large integers or high precision floating- point values

- Package **java.math**
- Both are immutable

```
BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("264476865342114456798");
BigInteger c = a.multiply(b);
System.out.println(c);
```

1.16 Unit Testing with JUnit

测试代码

```
public class IMath {
    /** Returns an integer approximation to the
     * square root of x.
    */
    public static int isqrt(int x) {
        int guess = 1;
        while (guess * guess < x) {
            guess++;
        }
        return guess;
    }
}
```

```

import org.junit.Test;
import static org.junit.Assert.*;
/** A JUnit test class to test the class IMath. */
public class IMathTest {
    /** Test isqrt. */
    @Test
    public void testIsqrt0() {
        assertEquals("...", 0, IMath.isqrt(0));
    }
    @Test
    public void testIsqrt1() {
        assertEquals("...", 1, IMath.isqrt(1));
    }
    @Test
    public void testIsqrt2() {
        assertEquals("...", 1, IMath.isqrt(2));
    }
    ...
}

```

JUnit tests are usually organized into test classes which are only used for testing.

Annotating a method with the `@org.junit.Test` annotation turns that method into a test.

Use an assert method to check the expected result of the code execution versus the actual result.

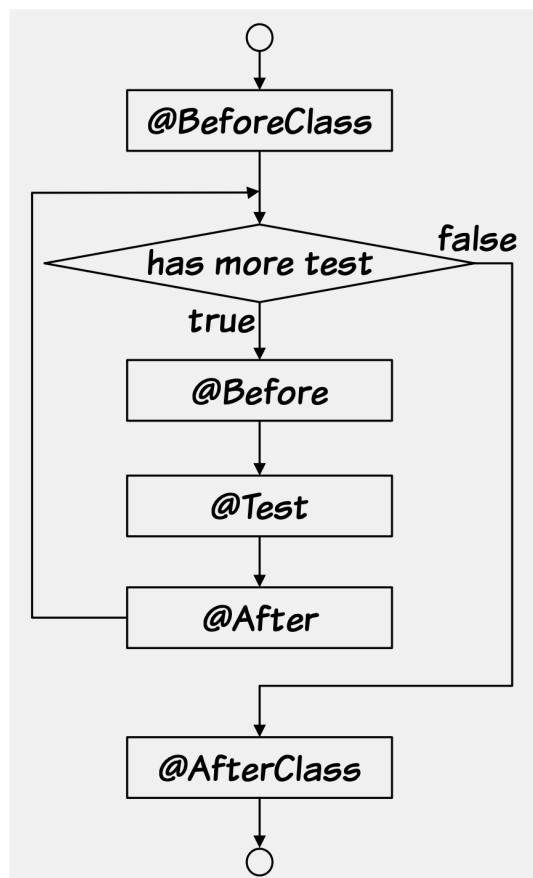
Provide meaningful messages in assert statements to help developers understand the test result.

Tests should be independent!

39

Lifecycle Methods

- ❖ A JUnit class may contain life-cycle methods
 - Always public void no-arg
 - `@BeforeClass`: The *static* method will be executed before execution of any test method in the class
 - `@AfterClass`: The *static* method will be executed after all test methods of this class are executed
 - `@Before`: The *instance* method will be called before each test method in the class
 - `@After`: The *instance* method will be called after each test method in the class



Example:

```
public class CoordinateTest {
    private Coordinate p;

    @Before
    public void setUp() {
        c = new Coordinate(10, 10);
    }

    @Test
    public void testSetX() {
        c.setX(20);
        assertEquals(20, c.getX());
        assertEquals(10, c.getY());
    }

    @Test
    public void testSetY() {
        c.setY(30);
        assertEquals(30, c.getY());
        assertEquals(10, c.getX());
    }
}
```

1.17 Shallow/deep Copy of Objects

以往我们想复制一个对象时，最自然的操作就是直接赋值给另一个变量

```
public class Person{

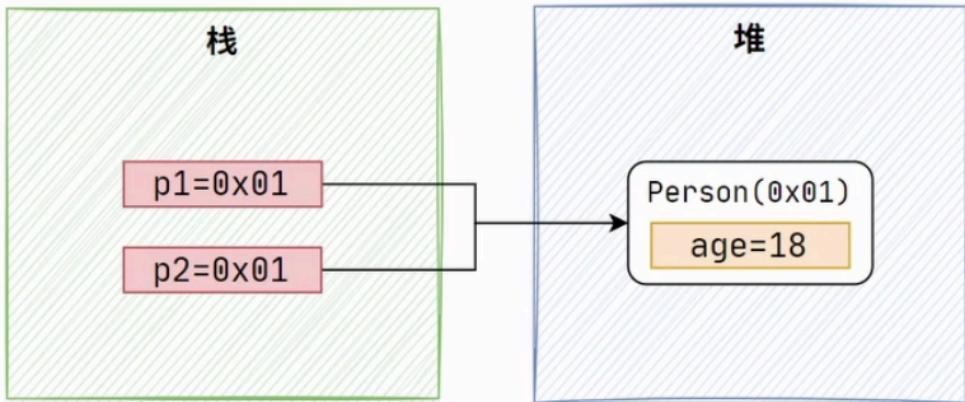
    public int age;
    public Person(int age) {
        this.age = age;
    }

    public static void main(String[] args) {
        Person p1 = new Person(18);
        Person p2 = p1;
        p2.age = 20;
        // 复制了对象地址, p1和p2指向了同一个对象, 任意一个变量操作属性都会影响到另一个变量

        System.out.println(p1 == p2); // true
    }
}
```

```
        System.out.println(p1.age); // 20
    }
}
```

引用拷贝



这种对同一个对象的操作，当然算不上真正的复制，所以引用拷贝并不算对象拷贝(深拷贝和浅拷贝)

- 浅拷贝(Shallow Copy of Objects)

浅拷贝(Shallow Copy of Objects)是创建一个新对象，这个对象有着原始对象属性值的一份精确拷贝。如果属性是基本类型，拷贝的就是基本类型的值，如果属性是引用类型，拷贝的就是内存地址，所以如果其中一个对象改变了这个地址，就会影响到另一个对象。

在Java中，Object提供了一个clone()方法

```
class Object{
    protected native Object clone() throws CloneNotSupportedException;
}
```

如果子类不重写该方法，并将其声明为public，那外部就调用不了对象的clone()

子类在重写时直接调用Object的clone()方法即可

```
public class Person implements Cloneable{

    public int age;

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

    @Override
    public Person clone(){
        try {
            return (Person) super.clone();
        } catch (CloneNotSupportedException e) {

```

```

        return null;
    }

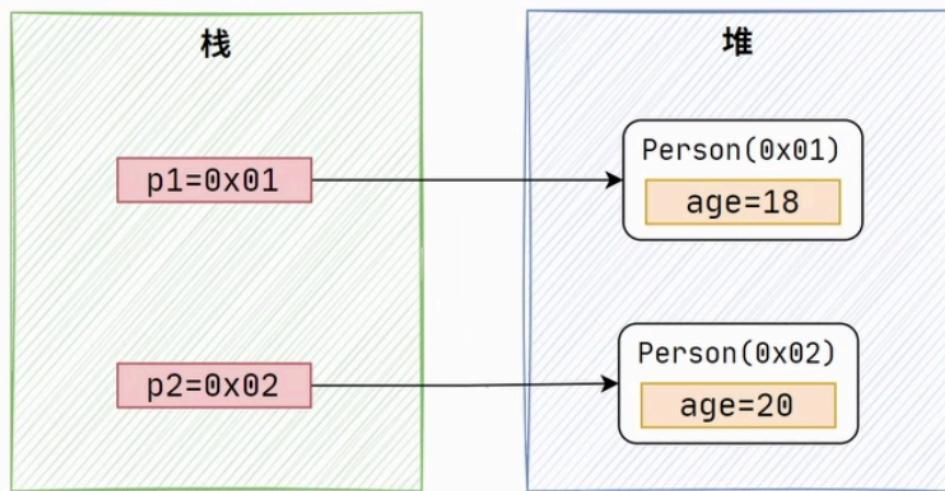
}

public static void main(String[] args) {
    Person p1 = new Person(18);
    Person p2 = p1.clone();
    p2.age = 20;

    System.out.println(p1 == p2); // false
    System.out.println(p1.age); // 18
}
}

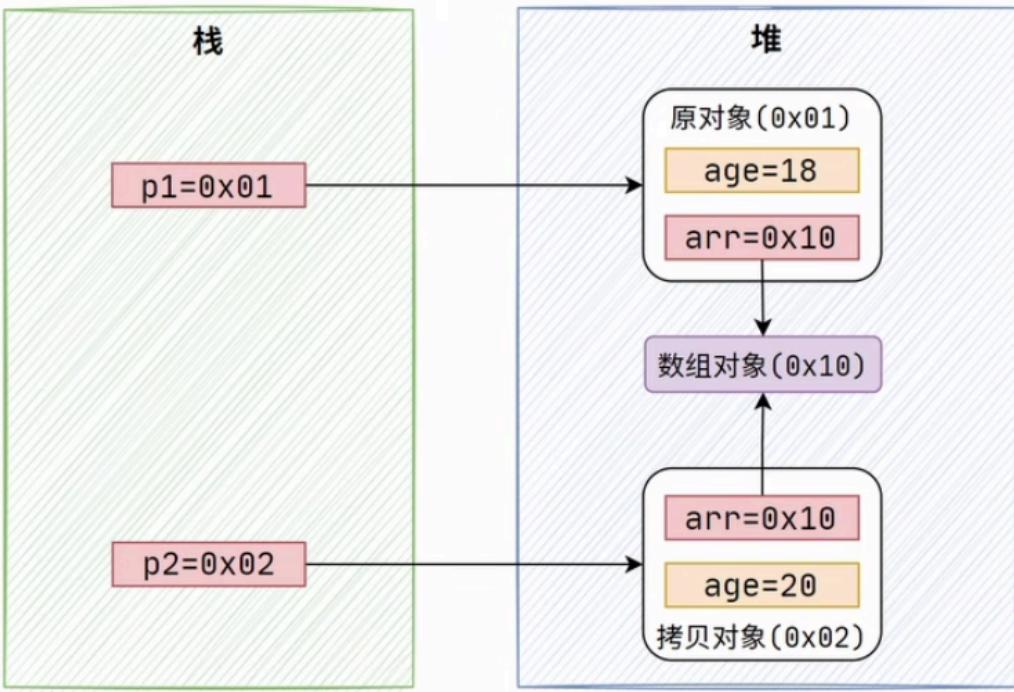
```

浅拷贝



但是如果拷贝的对象中有属性是引用类型，那这种浅拷贝的方式只会复制该属性的引用地址。即拷贝对象和原对象的属性都指向了同一个对象，如果对这个属性进行一些操作，则会影响到另一个对象的属性。

浅拷贝



若想将对象中的引用类型属性也进行拷贝那就得用深拷贝了

- 深拷贝(Deep Copy of Objects)

深拷贝(Deep Copy of Objects)是将一个对象从内存中完整的拷贝份出来，包括引用对象，从堆内存中开辟一个新的区域存放新对象，且修改新对象不会影响原对象。

我们将 clone() 方法稍微修改一下，clone出对象后我们再对对象的属性进行一次拷贝。这样就完成了属性的复制

```
public class Person implements Cloneable{  
  
    public int age;  
    public int[] arr = new int[]{1, 2, 3};  
  
    public Person(int age) {  
        this.age = age;  
    }  
  
    @Override  
    public Person clone(){  
        try {  
            Person p = (Person) super.clone();  
            p.arr = this.arr.clone(); // 调用引用类型的clone()方法  
            return p;  
        } catch (CloneNotSupportedException e) {  
            return null;  
        }  
    }  
}
```

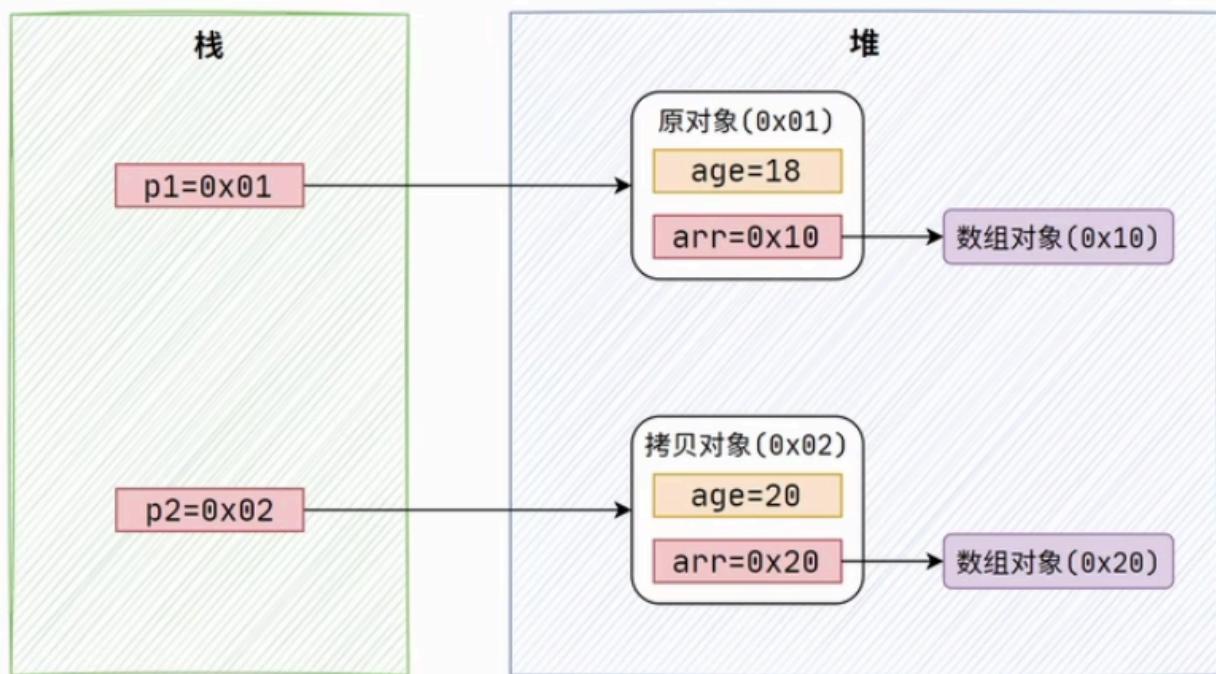
```

public static void main(String[] args) {
    Person p1 = new Person(18, new int[]{1, 2, 3});
    Person p2 = p1.clone();
    p2.age = 20;
    p2.arr = new int[]{3, 2, 1};
    System.out.println(p1 == p2); // false
    System.out.println(p1.age); // 18
    System.out.println(p2.age); // 20
    System.out.println(p1.arr[0]); // 1
    System.out.println(p2.arr[0]); // 3
}

```

}

深拷贝



1.18 Iterator(范型+instanceof使用)

Iterator 迭代器

```
public static void main(String[] args) {
    ArrayList<Integer> arr = new ArrayList<>();
    arr.add(1);
    arr.add(2);
    arr.add(3);

    for (int j : arr) {
        System.out.print(j);
    }
}
```

Output: 123

它在 Java 中有一个经典的 For-Each 循环。

但是为什么 Java 知道如何基于 ArrayList 进行迭代 `for (int j : arr)` 呢?

实际上, Java 已经将上述循环部分代码编译成:

```
Iterator i = arr.iterator();
while (i.hasNext()) {
    System.out.println(i.next());
}
```

在 `Iterator` 类的帮助下, Java 可以到达 ArrayList 中的每个元素, 以及知道序列中当前元素的下一个元素。

迭代器是一种 软设计模式(soft design pattern), 描述了扫描一系列元素的过程, 一次一个元素, 其中元素包括容器、来自网络的流或一系列计算等(container, streaming from network, or a serial of computation, etc.)。

在 Java 中, 迭代器被抽象为 `java.util.Iterator` 接口。

Iterator Interface 迭代器接口

接口中有三个方法 `java.util.Iterator`:

开始时, 有一个 参考点(reference points) 指向 序列中的第一个元素

- `hasNext()`: 如果当前引用有下一个元素, 返回 `true`, 否则返回 `false`。
- `next()`: 返回当前 指向的引用(pointing reference) 并将引用更新到下一个元素(update the reference to the **next** element.)
- `remove()`: 移除最近被方法调用返回的元素 `next()`。如果该元素已被删除或方法没有返回任何元素, 则返回异常。`next()`

我们可以使用通用循环来遍历迭代器序列。

```
public class Animal {
    public void move(){
```

```

        System.out.println("动物在移动");
    }
}

class Bird extends Animal {
    public void move(){
        System.out.println("鸟儿在飞翔");
    }
    public void eatCorn(){
        System.out.println("鸟儿在吃玉米");
    }
}

class Cat extends Animal {
    public void move(){
        System.out.println("猫咪走猫步");
    }
    public void catchMouse(){
        System.out.println("猫抓老鼠");
    }
}

```

```

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.lang.Object;

public class GenericTest {
    public static void main(String[] args) {
        // 1. 不使用范型来iterator
        List list = new ArrayList();
        // 准备对象
        Cat c = new Cat();
        Bird b = new Bird();
        // 将对象加入到集合中
        list.add(c);
        list.add(b);

        // 遍历集合，取出cat让它抓老鼠，取出Bird让它飞！
        Iterator i = list.iterator();
        while (i.hasNext()) { // 如果有元素，返回object
            // Animal a = i.next(); 没有这个语法，通过迭代器取出的就是Object
            Object o = i.next(); // 返回 当前指向的引用 并将引用更新到下一个元素
            if (o instanceof Animal) {
                Animal a = (Animal) o;
                a.move();
                // 猫咪走猫步
                // 鸟儿在飞翔
            }
        }
    }
}

```

```
// 2. 使用范型来iterator
List<Animal> mylist = new ArrayList<Animal>();
Cat c1 = new Cat();
Bird b1 = new Bird();
// 将对象加入到集合中
mylist.add(c1);
mylist.add(b1);

// 获取迭代器
// 这个表示迭代器迭代的是Animal类型。
// 使用泛型之后，每一次迭代返回的数据都是Animal类型。
Iterator<Animal> it = mylist.iterator();
while (it.hasNext()){
    Animal animal = it.next(); // 这里不需要进行强制类型转换了。直接调用。
    animal.move();
    // 猫咪走猫步
    // 鸟儿在飞翔
    // 猫咪走猫步
    // 鸟儿在飞翔
    if (animal instanceof Cat){
        Cat c2 = (Cat) animal;
        c2.catchMouse(); // 猫抓老鼠
    }
    if (animal instanceof Bird){
        Bird b2 = (Bird) animal;
        b2.eatCorn(); // 鸟儿在吃玉米
    }
}

for (Animal animal : mylist) {
    System.out.print(animal);
    // Draft.Cat@5ca881b5
    // Draft.Bird@24d46ca6
    if (animal instanceof Cat){
        Cat c2 = (Cat) animal;
        c2.catchMouse(); // 猫抓老鼠
    }
    if (animal instanceof Bird){
        Bird b2 = (Bird) animal;
        b2.eatCorn(); // 鸟儿在飞翔
    }
}
}
```

1.19 Java Container

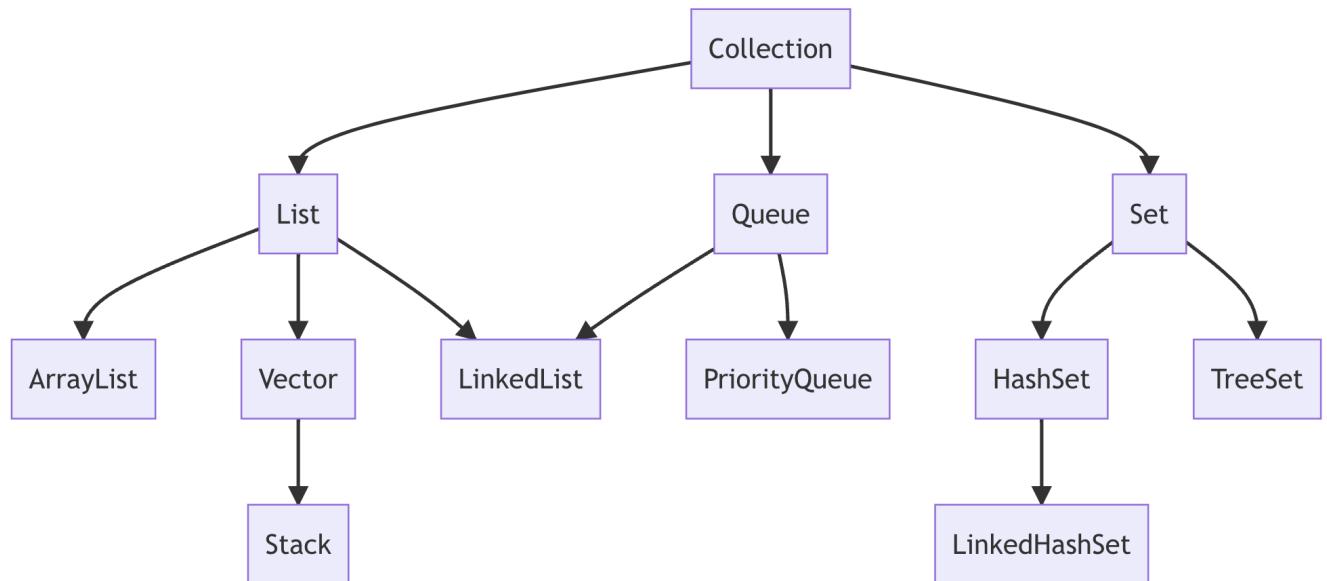
Java 容器是一种 可将若干个对象放在一起存储的数据结构，与数组类似，不同点在于容器

存储数据类型可以不同，可将所有类型数据放入同一个容器类型中存储，而无需创建不同类型的容器分别存放相对应的类型对象

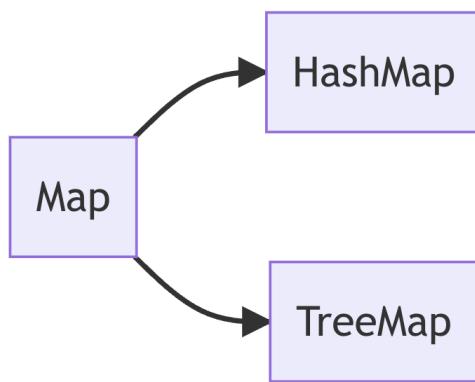
长度可变，意味着无需担心初始容器大小不够出现下标越界的异常，在容器初始化时设定一个初始值，当容器的元素数量达到容器设定的存储阈值时，容器将自动扩充容器大小

容器定义了一系列对存储数据操作的方法，使用时无需关注容器的内部实现，即可完成数据存取。

Java Collection 容器类包含 **List**、**ArrayList**、**Vector**、**HashTable**、**HashSet**



Java Map 容器类包含 **HashMap**、**TreeMap**



02. Class and Object

2.1 Class Definition

2.1 类的定义

一般来说，一个类被定义成两个部分，静态部分和动态部分。

- **静态部分**: 类的 **属性 (field)**。用来描述此类实例的特征或状态，如，人的姓名、年龄、性别、身高等。这个部分会存储在此类实例的内存中。
- **动态部分**: 类的 **方法 (method)**。定义了此类实例的行为，能够对此类实例进行的操作，如，人的动作、说话、走路等。这个部分不会存储在此类实例的内存中。
- Object Method Invocation 对象方法调用

```
public class Person { // 人类

    // 静态部分 - 属性 (field)
    String name; // 姓名
    int age; // 年龄
    boolean gender; // 性别 true-男 false-女

    // 动态部分 - 方法 (method)
    public void walk() { // 走路
        System.out.println("I'm walking.");
    }

    public void speak() { // 说话
        System.out.println("I'm speaking.");
    }

    public int howOld() { // 获取年龄
        System.out.println(age); // 获取实例的age属性值
        return age;
    }

    public void updateName(String newName) { // 更新姓名
        name = newName; // 更新实例的name属性值
    }
}
```

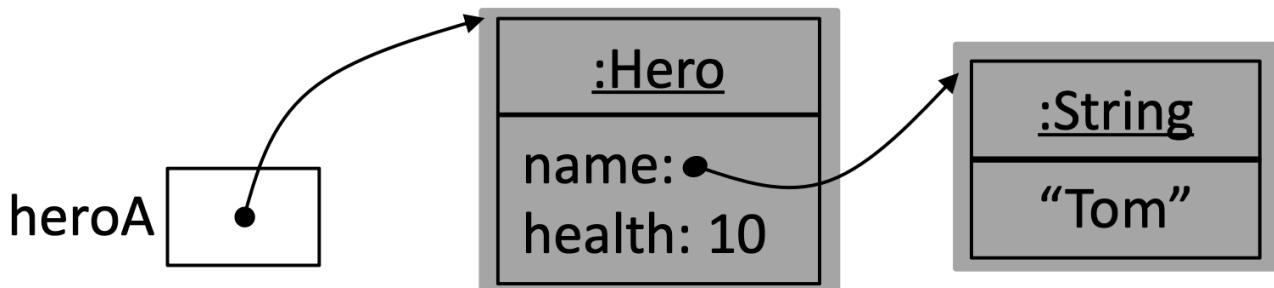
2.2 Primitive Type vs. Reference Type

- **基本数据类型(Primitive types)**: boolean, byte, short, char, int, long, float, and double (**如果值一样，那么他们的地址也一样**)

length 5

- 所有其他类型都是**引用数据类型(reference types)**: 包括可实例化类定义的类型以及数组类型: String,

Scanner, int[], String[], etc. **Reference variables store addresses** (like pointers in C)



2.3 Class Relationships

Dependence ("uses-a") 关系：

- 比如，人类用到了天气类，因为一个人的行为会被天气影响到，有依托的关系；

Aggregation ("has-a") 关系：

- 一个A类包括了另一个B类。A类有B类的一个属性，或A类中创建了B类的一个实例，或A类的方法中有B类的一个参数。
- 比如，人类中有一个工作类，因为一个人有工作这一属性。人类也可以有一个是人类，因为可以用来表示亲人、朋友这些属性。

Inheritance ("is-a") 关系：

- 能够表示相对一般的类和相对具体的类之间的关系，被称为 **继承 (Inheritance)** 的关系。继承类能够继承被继承类的属性和方法，同时也可以有自己特有的部分。
- 比如，**学生** 类是人类中的一种相对具体的类，因此**学生**类可以继承**人类**。常识告诉我们，**学生**同时拥有人的属性和方法。与此同时，**学生**还能够拥有一些特有的属性和方法，比如，年纪、学校等属性，进入学校、更新学校等行为。

2.4 封装类 (Wrapper Classes)

- 包装类没有无参数构造函数 (no-arg constructors)
- 所有包装类的实例都是不可变的 (immutable)，即一旦创建对象，它们的内部值就不能改变。
- java为了将基本数据类型，可以以对象的形式操作，提供了对应基本数据类型的包装类

八大基本数据类型 (primitive typed values) 和其 wrapper classes：

```
byte          Byte
short         Short
int           Integer
long          Long
float         Float
double        Double
char          Character
boolean       Boolean
```

Integer是基本数据类型int的封装类，需要使用new来创建对象，默认值是null

基本类型和Integer类型混合使用时，Java会自动通过拆箱和装箱实现类型转换

Integer作为一个对象类型，封装了一下方法和属性，我们可以利用这些方法操作数据

Autoboxing and Unboxing

- unboxing就是自动 wrapper classes 转化为 primitive typed values
- boxing就是自动将 primitive typed values 转化为 wrapper classes

在Java SE5之前，如果要生成一个数值为10的Integer对象，必须这样进行：

```
Integer i = new Integer(10);
```

而在从Java SE5开始就提供了自动装箱的特性，如果要生成一个数值为10的Integer对象，只要这样就可以了：

```
Integer i = 10; //自动装箱
// 系统自动为我们执行了 Integer i = Integer.valueOf(10);

int j = i; // 自动拆箱(Auto-unboxing) 自动根据数值的类型创建Integer对象将Integer对象自动拆箱为
int
// 系统自动为我们执行了 int j = i.intValue();
```

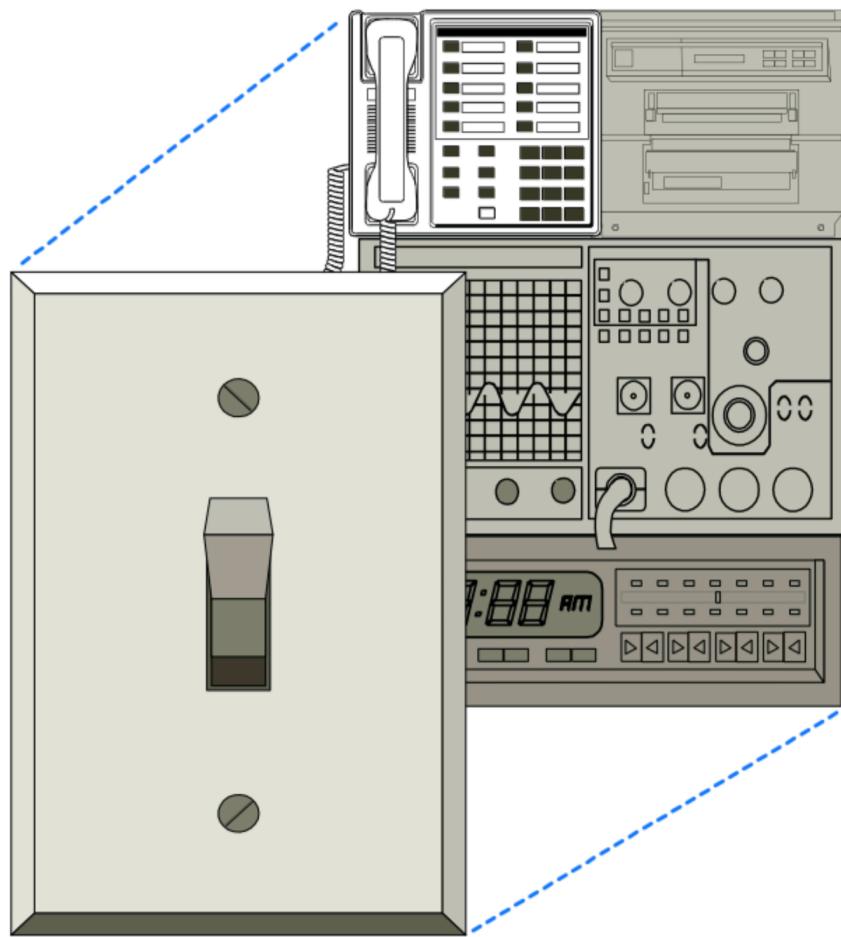
Example:

```
char a = 'a', d = 'd';
Character b = new Character('b');
a = b; // unboxing 将包装类转换为基本数据类型
b = d; // boxing
char c = new Character('c');
Character d = 'd';
```

03. 封装(Encapsulation)

3. 封装

- 封装 (Encapsulation) 是指把对象的内部属性 和 行为的实现细节 隐藏起来，使得对象的内部属性和行为只能通过公共接口来访问。如图：



目的：

- 某一类的程序员可以自由设计类的内部属性和行为的实现方法，无需关心使用此类的其他类程序员会受到其内部实现细节的影响；
- 此类的程序员只需关心此类的行为对外的开放接口，如此类行为的名称、参数类型、返回值类型、用法等，能够让其他程序员使用这些接口；
- 能够提高可靠性和适应性，因为内部行为实现细节的改变不会影响到其他部分以及其用户。程序员只改变内部实现细节，就能方便的维护和增加新的功能。

实现Java封装的步骤

1. 修改属性的可见性来限制对属性的访问（一般限制为private），例如：

```
public class Person {  
    private String name;  
    // 私有只读属性name。只有类内部可访问和修改，对用户隐藏。  
}
```

这段代码中，将 **name** 和 **age** 属性设置为私有的，只能本类才能访问，其他类都访问不了，如此就对信息进行了隐藏。

2. 对每个值属性提供对外的公共方法访问，也就是创建一对赋取值方法，用于对私有属性的访问，例如：

```
public String getName(){ // 访问器(accessor)
    return name;
}

// 访问器方法getName。留给用户获取name属性的唯一途径，确保只读，不会被用户随意修改。
public void setName(String name){ // 修改器(mutator) 方法
    this.name = name;
}
```

采用 **this** 关键字是为了解决实例变量（`private String name`）和局部变量（`setName(String name)`中的`name`变量）之间发生的同名的冲突。

3. 若用户需要访问某对象的姓名，则：

```
Person p1 = new Person("Alice");
// System.out.println(p1.name); // Not allowed
System.out.println(p1.getName());
```

04. 继承(Inheritance)

4.1 继承

- 子类(Sub Class/ Child Class/ Derived class)可以拥有父类(Super Class/ Parent Class/ Base class)非 **private** 的属性、方法 ==> 子类不能继承父类的**private**属性和方法
- 子类可以拥有自己的属性和方法，即子类可以对父类进行扩展
- 子类可以用自己的方式实现父类的方法
- 可重用性(Reusability):
- 父类(Super Class)的构造函数不会继承到子类(Sub Class)中
- 父类(Super Class)的field和method一般为**protected**

可以从中继承的类：同package / public的class 才可以继承

package x.y;

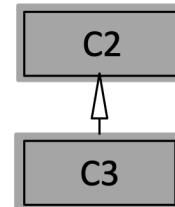
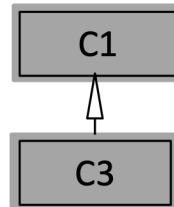
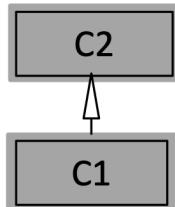
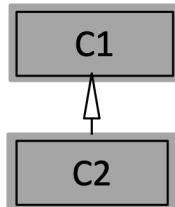
class C1 ...

package x.y;

public class C2 ...

package m.n;

class C3 ...



		p.A	p.B	p.C extends p.A	q.D extends p.A	q.E
p.A	Field (static or not)	public	Y	Y	Y	Y
	protected	Y	Y	Y	Y	N
	(default)	Y	Y	Y	N	N
	private	Y	N	N	N	N
p.B	Method (static or not)	public	Y	Y	Y	Y
	protected	Y	Y	Y	Y	N
	(default)	Y	Y	Y	N	N
	private	Y	N	N	N	N

- p, q: packages
- A,B,C,D,E:classes

4.2 super 与 this 关键字

super关键字：我们可以通过super关键字来实现对父类成员的访问，用来引用当前对象的父类。

this关键字：指向自己的引用。

```

class Animal {
    void eat() {
        System.out.println("animal : eat");
    }
}
  
```

```

    }
}

class Dog extends Animal {
    void eat() {
        System.out.println("dog : eat");
    }
    void eatTest() {
        this.eat();    // this 调用自己的方法
        super.eat();  // super 调用父类方法
    }
}

public class Test {
    public static void main(String[] args) {
        Animal a = new Animal();
        a.eat(); // animal : eat
        Dog d = new Dog();
        d.eatTest(); // dog : eat // animal : eat
    }
}

```

4.3 java.lang.Object class

Java中的每个class(直接或间接地)都扩展了 (directly or indirectly) extends the **java.lang.Object class**

- If no inheritance is specified when a class is defined, the super-class of the class is Object.

```

public class Date{
    private int year;
    private int month;
    private int day;
    ...
}

```

is equivalent to

```

public class Date extends Object{
    private int year;
    private int month;
    private int day;
    ...
}

```

- **public String toString()**
- **public boolean equals(Object other)**
- **public int hashCode()**
- **public Object clone()**

```
public class Date extends Object{  
    private int year;  
    private int month;  
    private int day;  
    private String name;  
  
    public Date(int year, int month, int day, String name) {  
        this.year = year;  
        this.month = month;  
        this.day = day;  
        this.name = name;  
    }  
}
```

public String toString()

起初我们默认实现(default implementation)返回一个String，都是由对象的类名@符号和该对象的十六进制值组成

```
Date rocky = new Date(2004, 1, 7, "Rocky");  
System.out.println(rocks); // 返回对象的字符串表示形式 Output: Date@15037e15  
// 也可以写成 System.out.println(rocks + "");
```

现在我们重写 (Override) **toString**

```
public String toString(){  
    StringBuilder builder = new StringBuilder();  
    builder.append("Name: ").append(name).append(" ")  
        .append("[Year: ").append(year).append("; ")  
        .append("Month: ").append(month).append("; ")  
        .append("Day: ").append(day).append("]");  
    return builder.toString(); // Name: Rocky [Year: 2004; Month: 1; Day: 7]  
}
```

```
System.out.println(rocks); // Name: Rocky [Year: 2004; Month: 1; Day: 7]  
// 不需要写成 System.out.println(rocks.toString());
```

public boolean equals(Object other)

起初我们默认实现(default implementation) compares the two references using ==

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

现在我们重写 (Override) **equals** ==> 让他可以比较 Name

```
public boolean equals(Object o) {  
    if (o instanceof Date) {  
        return getName().equals(((Date)o).getName());  
    }  
    else  
        return false;  
}
```

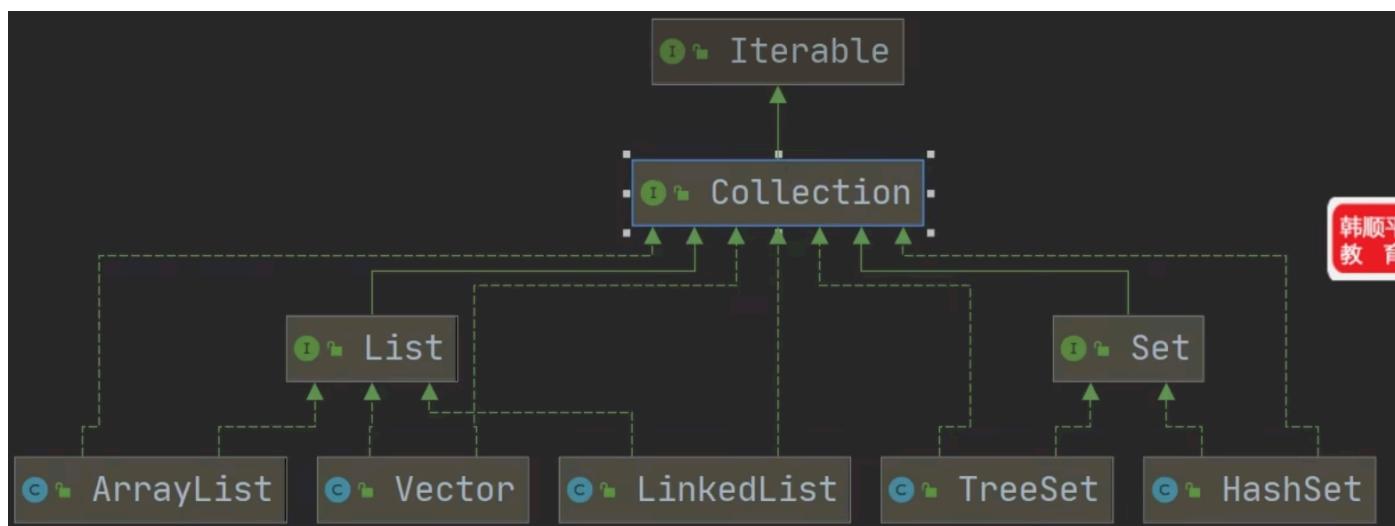
```
public static void main(String[] args) {  
    Date rocky = new Date(2004, 1, 7, "Rocky");  
    System.out.println(rocky); // Name: Rocky [Year: 2004; Month: 1; Day: 7]  
  
    Date gigi = new Date(2003, 4, 8, "Gigi");  
    System.out.println(rocky.equals(gigi)); // false  
}
```

HashSet 存储对象的无序数组

- List 是一个接口，继承于 Collection 的接口。代表着有序的队列。
- Set 是一个接口，继承于 Collection 的接口。代表着无序的队列。

List中元素可以重复，并且是有序的（这里的有序指的是按照放入的顺序进行存储。如按照顺序把1, 2, 3存入List，那么，从List中遍历出来的顺序也是1, 2, 3）。

Set中的元素不可以重复，并且是无序的（从set中遍历出来的数据和放入顺序没有关系）。



- HashSet实现了Set接口
- HashSet实际上是HashMap (基于 HashMap 来实现的)

```
public HashSet() {  
    map = new HashMap<>();  
}
```

- HashSet 不允许有重复元素的集合 (**Collision**:多个值可以映射到同一个 桶 bucket)
- HashSet 中的元素实际上是对象。
- HashSet 允许有 null 值，但是只能有一个null
- HashSet 是无序的，即不会记录插入的顺序。(HashSet 不是线程安全的， 如果多个线程尝试同时修改 HashSet，则最终结果是不确定的。您必须在多线程访问时显式同步对 HashSet 的并发访问。)
- 存储在数组中的元素称为桶(buckets)

Implementation of methods:

- contains(o)
- add(o)
- remove(o)

Example 1:

```
public static void main(String[] args) {  
    HashSet set = new HashSet();  
    // 在执行add方法后，会返回一个boolean值  
    // 如果添加成功，返回 true，否则返回false  
    // 可以通过 remove 指定删除哪个对象  
    System.out.println(set.add("A")); // true  
    System.out.println(set.add("B")); // true  
    System.out.println(set.add("C")); // true  
    System.out.println(set.add("A")); // false 重复元素，添加失败  
  
    System.out.println(set); // [A, B, C]  
    System.out.println(set.contains("A")); // true  
    set.remove("A");  
    System.out.println(set); // [B, C]  
    System.out.println(set.contains("A")); // false  
  
    set = new HashSet(); // 或者使用 set.clear(); 清空set  
    System.out.println(set); // []  
    set.add("lucy"); // 添加成功  
    set.add("lucy"); // 添加失败  
    set.add(new Dog("tom")); // 添加成功  
    set.add(new Dog("tom")); // 添加成功，因为是两个不同对象  
    System.out.println(set); // [Dog{name='tom'}, Dog{name='tom'}, lucy]  
    // 如果Dog类不改写toString方法，则会输出 [Draft.Dog@36baf30c, Draft.Dog@7a81197d,  
    lucy]  
}
```

```

class Dog {
    private String name;
    public Dog(String name){
        this.name = name;
    }

    @Override
    public String toString() {
        return "Dog{" +
            "name='" + name + '\'' +
            '}';
    }
}

```

Example 2:

```

public static void main(String[] args) {
    Date rocky = new Date(2004, 1, 7, "Rocky");
    System.out.println(rocks); // Name: Rocky [Year: 2004; Month: 1; Day: 7]

    Date gigi = new Date(2003, 4, 8, "Gigi");
    System.out.println(rocks.equals(gigi)); // false

    // 创建一个 HashSet 对象 arr, 用于保存整形元素:
    HashSet<Integer> arr = new HashSet<Integer>();

    // HashSet 类提供了很多有用的方法, 添加元素可以使用 add() 方法
    arr.add(rocks.year); // 2004
    arr.add(gigi.year); // 2003
    arr.add(3);
    arr.add(4);
    arr.add(3); // 重复的元素不会被添加
    arr.add(4); // 重复的元素不会被添加

    System.out.println("HashSet: " + arr); // HashSet是无序的 [2003, 3, 2004, 4]
    System.out.println("HashCode value: " + arr.hashCode()); // HashSet value:
4014
    System.out.println("HashSet size: " + arr.size()); // 4

    for (int i : arr){ // 逐个输出数组元素的值
        System.out.println(i); // 2003 3 2004 4
    }

    Integer num = 123;
    System.out.println(num.hashCode()); // 123

    arr.remove(3);
}

```

```
System.out.println(arr); // [2003, 2004, 4]

arr.clear();
System.out.println(arr); // []
}
```

public int hashCode()

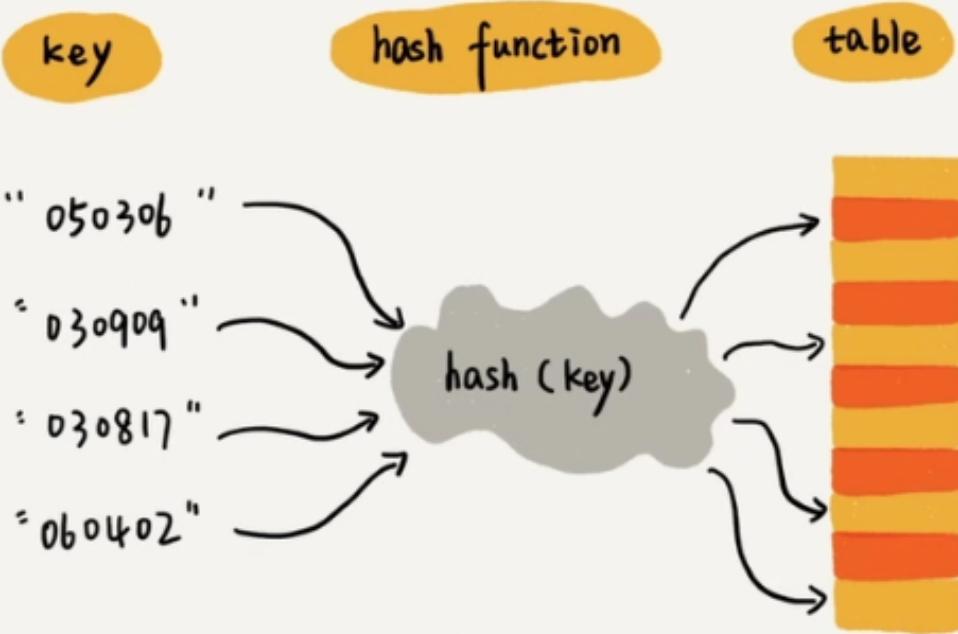
Java中 HashSet 的 hashCode() 方法用于获取此 HashSet 实例的 hashCode 值。它返回一个整数值，该值是此 HashSet 实例的 hashCode 值。如果 equals() 是 true，那么两个值当 hashCode 值一定会一样

首先让我们看看 int hashCode() 方法的默认实现(default implementation) 是怎么样的

```
class Myclass {  
}
```

- 如果没有重写 public int hashCode() 方法，那他通常会将 内存地址转换为int数值进行返回
- 对象内存地址经过哈希算法转换的一个数字。可以等同看做内存地址

```
public static void main(String[] args) {  
  
    Object o = new Object();  
    int hashCodeValue = o.hashCode();  
    System.out.println(hashCodeValue); // 1175962212  
  
    Myclass a = new Myclass();  
    System.out.println(a.hashCode()); // 2055281021  
  
    Integer num = 123;  
    System.out.println(num.hashCode()); //123  
}
```



我们可以用 hashCode() 获取到这个int数值就是哈希码(散列码) ==> 确定对象在哈希表中的索引位置

hashCode() 用来定位索引位置，提高效率的同时可能会发生哈希冲突(**Collision**: multiple values can be mapped to the same bucket)，当哈希冲突时，我们就要通过equals() 来判断冲突对象是否相等

```
import java.util.HashSet; // HashSet 类位于 java.util 包中，使用前需要引入它
```

Possible implementations of hashCode()

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public String toString() {
        return "Point{" +
            "x=" + x +
            ", y=" + y +
            '}';
    }
}
```

```

public int hashCode(){
    // return new Random().nextInt();
    return x + y;
}

public static void main(String[] args) {
    Point p = new Point(3,4);
    Point p2 = new Point(1,1);
    Point p3 = new Point(5,6);

    Set<Point> set = new HashSet<Point>(); // p.hashCode() == 7
    set.add(p); // put into storage[7]
    set.add(p2); // put into storage[2]
    set.add(p3); // put into storage[11]

    System.out.println(p.hashCode()); // 7
    System.out.println(p2.hashCode()); // 2
    System.out.println(p3.hashCode()); // 11

    System.out.println(set.contains(p)); // true

    for (Point i : set) { // 方法一：使用 for 逐个输出数组元素的值
        System.out.print(i + " ");
        // Point{x=1, y=1} Point{x=3, y=4} Point{x=5,
y=6}
    }

    Iterator i = set.iterator(); // 方法二：使用 hasNext() 逐个输出数组元素的值
    while(i.hasNext()) {
        System.out.print(i.next() + " ");
        // Point{x=1, y=1} Point{x=3, y=4} Point{x=5, y=6}
    }
}

```

hashCode() 和 HashSet() 和 equals() 通常结合使用

public Object clone()

05. 多态 (Polymorphism)

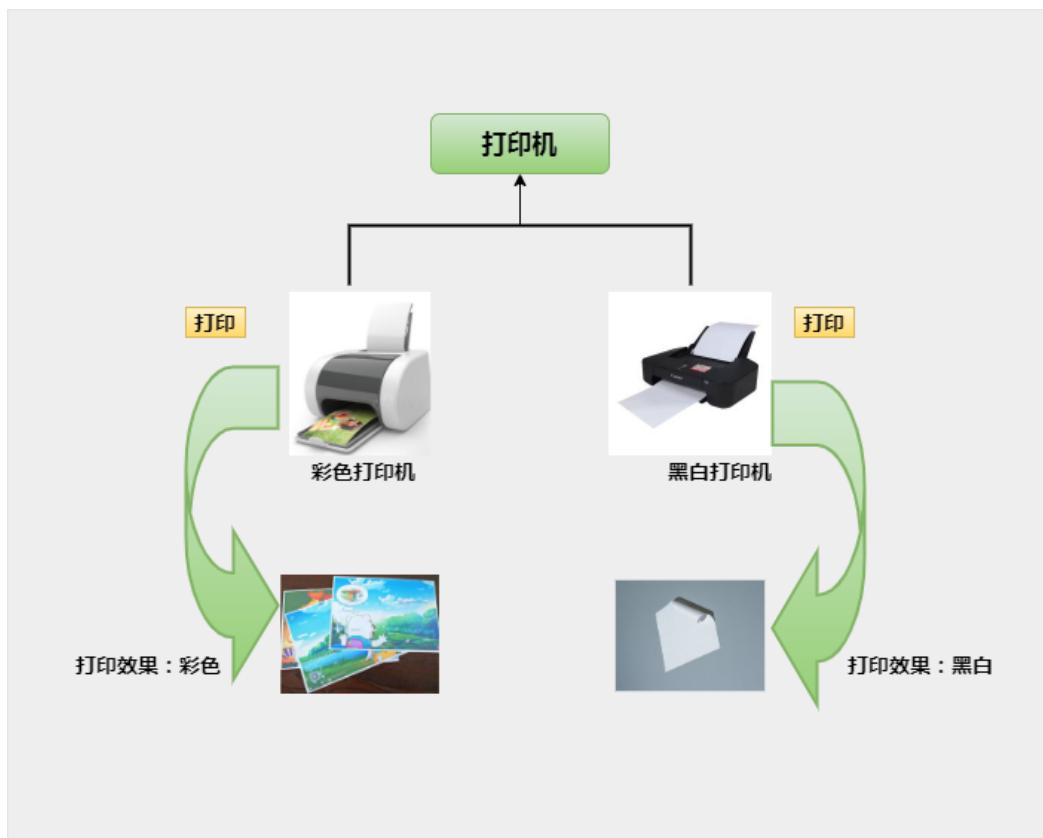
5.1 多态

封装是有了独立体

继承是对象与对象之间存在继承关系

多态是父类型的引用可以指向一个子类型的对象

- **多态 (Polymorphism)** 是一种在继承中作用于方法的机制。一个多态的方法，在被不同子类的对象调用时，能灵活地做出不同的对应行为。能够使一个方法有多种状态。
- **向上转型 (upcasting)**: 子类型 ==> 父类型，自动类型转换
- **向下转型 (downcasting)** : 父类型 ==> 子类型，强制类型转换，需要加强制类型转换符
- 无论是向上转型还是向下转型，两种类型之间必须要有继承关系。



```
public class Animal {  
    public void move(){  
        System.out.println("动物在移动");  
    }  
}
```

```
public class Cat extends Animal {  
    public void move(){ // Override  
        System.out.println("猫咪走猫步");  
    }  
    public void catchMouse(){  
        System.out.println("猫抓老鼠");  
    }  
}
```

```
public class Bird extends Animal {  
    public void move(){  
        System.out.println("鸟儿在飞翔");  
    }  
}
```

```
package Draft;  
  
public class Test {  
    public static void main(String[] args) {  
        //以前编写的程序  
        Animal a1 = new Animal();  
        a1.move(); // 动物在移动  
  
        Cat c1 = new Cat();  
        c1.move(); // 猫咪走猫步  
        c1.catchMouse(); // 猫抓老鼠  
  
        Bird b1 = new Bird();  
        b1.move(); // 鸟儿在飞翔
```

- java程序永远都分为编译阶段和运行阶段。
- 先分析编译阶段，再分析运行阶段，编译无法通过，根本是无法运行的。
- 编译阶段编译器检查a2这个引用的数据类型为Animal，由于Animal.class字节码当中有move()方法，所以编译通过了。这个过程我们称为静态绑定，编译阶段绑定。只有静态绑定成功之后才有后续的运行。
- 在程序运行阶段，JVM堆内存当中真实创建的对象是Cat对象，那么以下程序在运行阶段一定会调用Cat对象的move()方法，此时发生了程序的动态绑定，运行阶段绑定。
- 无论是Cat类有没有重写move方法，运行阶段一定调用的是Cat对象的move方法，因为底层真实对象就是Cat对象。
- 父类型引用指向子类型对象这种机制导致程序存在编译阶段绑定和运行阶段绑定两种不同的形态/状态，这种机制可以成为一种多态语法机制。

1. 使用多态语法机制(父类型引用指向子类型对象【多态】)

```
Animal a2 = new Cat(); // ==> 猫是一个动物 // 向上转型(upcasting)
a2.move(); // 猫咪走猫步
// 编译器看a2的时候只会当a2是一个 Animal a2, 实际上a2在运行时实际上就是Cat, 但是a2的类型是
Animal

// a2.catchMouse(); // error 因为编译阶段编译器检查到a2的类型是Animal类型
// 从Animal.class字节码文件当中查找catchMouse()没有找到该方法
```

- Animal和Cat之间存在继承关系, Animal是父类, Cat是子类
- Cat is a Animal ==> 合理的
- new cat() 创建的对象的类型是cat, a2这个引用的数据类型是Animal, 可见它们进行了类型转换
- 子类型转换成父类型, 称为向上转型(upcasting), 或者称为自动类型转换。
- Java中允许这种语法: 交类型引用指向子类型对象。

2. 那么怎么样才能运行 catchMouse() 呢?

- 我们可以将a2强制类型转换为Cat类型。
- a2的类型是Animal (父类), 转换成Cat类型 (子类), 被称为向下转型(downcastiag)强制类型转换

什么时候需要使用向下转型呢?

当调用的方法是子类型中特有的, 在父类型当中不存在, 必须进行向下转型。例: a2.catchMouse();

```
Cat c2 = (Cat)a2; // downcastiag
c2.catchMouse(); // 猫抓老鼠
```

3. 父类型引用指向子类型对象 【多态】

```
Animal a3 = new Bird();
// Cat c3 = (Cat)a3; // Compiles successfully but ClassCastException is thrown
at runtime
```

4. 怎么避免向下转型出现的ClassCastException呢?

- 使用`instanceof`运算符可以避免出现以上的异常, 运算结果只返回true /false:
- instanceof运算符也称为类型比较运算符, 因为它将实例(instance)与类型(type)进行比较
- 如果我们将 instanceof 运算符应用于任何具有 null 值的变量, 它将返回 false

instanceof (运行阶段动态判断)

- 第一： instanceof可以在运行阶段动态判断引用指向的对象的类型。
- 第二： instanceof的语法： (引用 instanceof 类型)
- 第三： instanceof运算符的运算结果只能是： true/false
- 第四： c是一个引用， c变量保存了内存地址指向了堆中的对象。

假设(c instanceof Cat) 为true表示： c引用指向的堆内存中的java对象是一个Cat。

假设(c instanceof Cat)为false表示： c引用指向的堆内存中的java对象不是一个Cat。

- 在进行强制类型转换(向下转型)之前，建议采用instanceof运算符进行判断，避免ClassCastException异常的发生。这是一种编程好习惯。

假设： (a instanceof Animal)

true表示： a这个引用指向的对象是一个Animal类型。

false表示： a这个引用指向的对象不是一个Animal类型。

```
// 当a3引用指向的对象确实是一个Cat的时候才能写 Cat c3 = (Cat)a3;
if (a3 instanceof Cat){ // false
    Cat c3 = (Cat)a3;
    c3.catchMouse();
} else if (a3 instanceof Bird) { // true
    Bird b2 = (Bird)a3;
    b2.eatCorn(); // 鸟儿在吃玉米
}
}
```

总结

```
public class Test2 {
    public static void main(String[] args) {
        //父类型引用指向子类型对象
        // 向上转型
        Animal a1 = new Cat();
        Animal a2 = new Bird();

        //向下转型【只有当访问子类对象当中特有的方法】
        if (a1 instanceof Cat) {
            Cat c1 = (Cat) a1;
            c1.catchMouse(); // 猫抓老鼠
        }
        if (a2 instanceof Bird) {
            Bird b1 = (Bird) a2;
            b1.eatCorn(); // 鸟儿在吃玉米
        }
    }
}
```

```
    }
}
}
```

5.1 Overloading & Overriding

- **方法重载 (Method Overloading):** 方法同名，参数(parameters/arguments)不同

```
void run(){System.out.println("Vehicle is running");}
void run(a){System.out.println( a + "Vehicle is running");}
```

- **方法重写 (Method Overriding):**

如果子类提供了由其父类之一声明的方法的特定实现，则称为方法覆盖。

1. 该方法必须与父类中的方法同名
2. 该方法必须具有与父类中相同的参数
3. 必须存在 IS-A 关系（继承）

```
class Vehicle{
    void run(){
        System.out.println("Vehicle is running");
    }
}

//Creating a child class
class Bike2 extends Vehicle{
    void run(){
        System.out.println("Bike is running safely");
    }

    public static void main(String args[]){
        Bike2 obj = new Bike2();
        obj.run(); // Bike is running safely
    }
}
```

5.2 Static Binding and Dynamic Binding

Connecting a method call to the method body is known as binding.

1. 静态绑定 (Static Binding / Early Binding) :在编译时（由编译器）确定对象的类型时，称为静态绑定。

如果类中有任何私有、最终或静态方法，则存在静态绑定

```
public class Hero {  
    public static String msg = ".H";  
    public static void doSomething(){  
        System.out.println("H" + msg);  
    }  
}
```

```
public class Monster extends Hero{  
    public static String msg = ".M";  
    public static void doSomething(){  
        System.out.print("M" + msg + "-");  
        System.out.println("M" + Hero.msg);  
    }  
}
```

```
public static void main(String[] args) {  
    Hero h = new Hero();  
    h.doSomething(); // H.H  
    System.out.println(h.msg); // .H  
    Monster m = new Monster();  
    m.doSomething(); // M.M-M.H  
    System.out.println(m.msg); // .M  
    h = m;  
    h.doSomething(); // H.H  
    System.out.println(h.msg); // .H  
}
```

2. 动态绑定 (Dynamic Binding / Late Binding): 当对象的类型在运行时确定时，称为动态绑定

```
class Animal{  
    void eat(){  
        System.out.println("animal is eating...");  
    }  
}  
  
class Dog extends Animal{  
    void eat(){  
        System.out.println("dog is eating...");  
    }  
}
```

```

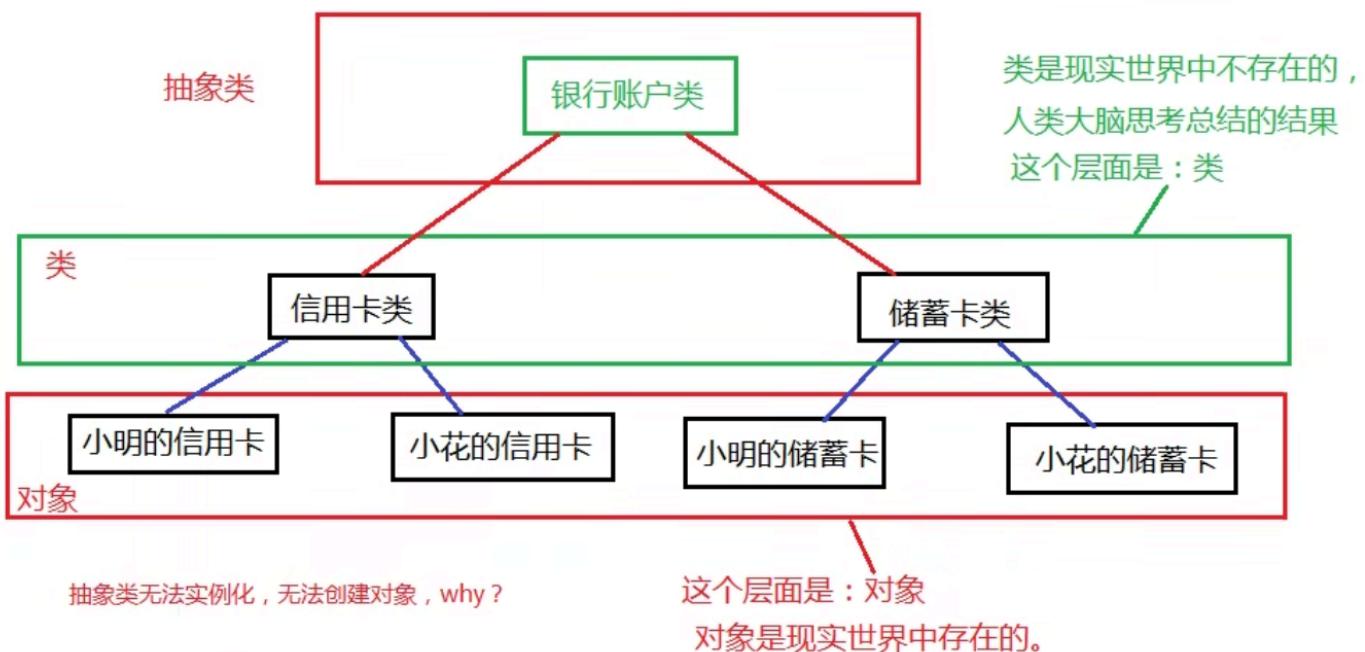
}

public static void main(String args[]){
    Animal a = new Dog();
    a.eat(); // dog is eating...
}
}

```

5.3 Abstract class

- 类到对象叫实例化，对象到类是抽象
- 抽象类是：类和类之间有共同特征，将这些具有共同特征的类再进一步抽象形成了抽象类。由于类本身是不存在的，所以抽象类无法创建对象。属于引用数据类型
- 抽象类和抽象类实际上可能还会有共同特征，还可以进一步再抽象。



为什么银行账户类不能创建对象？

因为创建的时候不知道创建信用卡还是储蓄卡，银行账户抽象类和具体的对象之间还有一层隔着

所以抽象类是无法实例化的，无法创建对象的，所以抽象类是用来被子类继承的。

- 抽象类虽然无法实例化，但是抽象类有构造方法，这个构造方法是供子类使用的

用 `abstract` 关键字声明的类在Java中称为抽象类，它可以有抽象和非抽象方法(`abstract` and `non-abstract` methods) 抽象方法表示没有实现的方法，没有方法体(method body)的方法。如：

```

abstract class Account{

    public abstract void doSome (); // 抽象方法 abstract methods

    public void doSome (){ // 非抽象方法 non-abstract method
        System.out.println("This is a non-abstract method");
    }

}

```

抽象方法特点是：

特点1：没有方法体，以分号结尾。

特点2：前面修饰符列表中有abstract关键字。

```

// 银行账户类
abstract class Account{
    public abstract void deposit();
}

```

一个非抽象的类，继承抽象类，必须将抽象类中的抽象方法进行重写

即：如果父类是抽象类，那么非抽象的子类需要实现父类中的所有抽象方法

```

// 子类继承抽象类，子类可以实例化对象
class Credit_A extends Account{
    public void deposit(){ // 必须重写 因为这是一个从抽象到具体的过程
        System.out.println("start deposit with A card");
    }
}

```

```

class Credit_B extends Account{
    public void deposit(){ // 必须重写 因为这是一个从抽象到具体的过程
        System.out.println("start deposit with B card");
    }
}

```

如果CreditB是抽象类的话，那么这个Animal中继承过来的抽象方法也可以不去重写/覆盖/实现

```

abstract class Credit extends Account{
    public abstract void deposit(); // 可以不去重写/覆盖/实现
}

```

```

public class AbstractTest {
    public static void main(String[] args) {
        // Account a = new Account(); // 错误: Account是抽象的, 无法实例化
        Account a = new Credit_A(); // 向上转型 这就是面对抽象编程
        a.deposit(); // 编译器认为这是Account 但是运行时就是具体的Credit
        // Output: start deposit with A card

        // 多态
        Account b = new Credit_B();
        b.deposit(); // Output: start deposit with B card
    }
}

```

5.4 Interface

- 接口也是一种"引用数据类型"。编译之后也是一个class字节码文件。
- 接口是完全抽象的。（抽象类是半抽象。）或者也可以说接口是特殊的抽象类。
- 接口支持多继承，一个接口可以继承多个接口。
- 继承接口可以是多继承，也就是如果父亲是接口（interface）那么可以有多个父亲。继承类只能是单继承，也就是如果父亲属于类（class），那么父亲只能有一个

```

interface A{

}

interface B extends A{

}

interface C extends A,B{

}

public interface D{ // 因为 interface 一定是 public 的所以 public 可以省略不写

}

```

- 接口中只包含两部分内容，一部分是：常量(值不可改变的变量)。另一部分是：抽象方法。
- public static final 可以省略不写，因为在接口中只有常量和抽象方法
- public abstract 可以省略不写，并且不能有方法体
- 接口所有元素都是public修饰的（公开的）

```

interface MyMath{
    // 常量
    // public static final double PI = 3.1415926;
    // public static final 可以省略不写，因为在接口中只有常量和抽象方法
    double PI = 3.1415926;
    int k = 100; // ==> public static final int k = 100;

    // 抽象方法，不能有方法体
    // public abstract int sum(int a, int b);
    // public abstract 可以省略不写
    int sum(int a, int b);
    int sub(int a, int b);
}

```

5.5 Implement

类和类之间叫做继承，类和接口之间叫做实现。仍然可以将"实现(implements)"看做"继承(extends)"。

继承使用extends关键字完成。实现使用implements关键字完成。

当一个非抽象的类实现接口的话，必须将接口中所有的抽象方法全部实现（覆盖、重写）

```

interface MyMath{ // 特殊的抽象类，完全抽象的，叫做接口(interface)
    double PI = 3.1415926;
    int sum(int a, int b);
    int sub(int a, int b);
}

```

```

// 我们可以用一个抽象类去实现(继承)他
abstract class Circle implements MyMath{

```

```

}
/*

```

同时如果我们想要用一个普通的 class 去实现(继承)他的话，需要重写接口的方法，因为接口是完全抽象的

```

class Triangle implements MyMath{ // error
    // 需要重写接口的方法
}
*/

```

```

class Triangle implements MyMath{
    public int sum(int a, int b){ // 实现(继承)接口的 class 重写的访问权限必须是 public
        return a + b;
    }
    public int sub(int a, int b){
        return a - b;
    }
}

```

```
}
```

5.6 Interface and Polymorphism

接口和他的多态使用

```
public static void main(String[] args) {
    // new MyMath(); ==> error: MyMath 是interface抽象的，无法实例化

    // interface 可以使用 Polymorphism
    MyMath m = new Triangle(); // 多态，父类型的引用指向子类型的对象
    int sum = m.sum(10,20); // Output: 30
    System.out.println(sum);

    int diff = m.sub(20,10); // Output: 10
    System.out.println(diff);
}
```

对于计算机来说，一个机箱上有多个接口，一个接口是按键盘的，一个接口是按鼠标的，一个接口是接电源的，一个接口是接显示器的.....所以java中一个类也可以实现多个接口

- 一个类可以实现多个接口，但是要重写所有接口中的方法
- 这种机制弥补了java中类和类只支持单继承的缺陷
- 实际上单继承是为了简单而出现的，现实世界中存在多继承，java中的接口弥补了单继承带来的缺陷。

```
// Task: 一个类可以实现多个接口，但是要重写所有接口中的方法
interface A{
    void a();
}

interface B extends A{
    void b();
}

interface C extends A,B{
    void c();
}

class Word implements A,B,C{ // 一个类可以实现多个接口，其实就类似于多继承
    public void a(){ // 实现/重写A接口的方法a()

    }
    public void b(){ // 实现/重写B接口的方法b()

    }
    public void c(){ // // 实现/重写C接口的方法c()

    }
}
```

```
}
```

5.7 extends 和 implements 的区别

继承(extends) 和 实现 (implements) 可以同时存在

继承(extends) 要在 实现 (implements) 前面

```
class Animal{  
  
}  
/*  
可飞翔的接口（是一对翅膀）  
能插拔的就是接口。（没有接口你怎么插拔）  
内存条插到主板上，他们之间有接口。内存条可以更换。  
接口通常提取的是行为动作  
*/  
interface Flyable{  
    void fly();  
}  
// 普通的 Cat 类，没有 interface  
class Cat extends Animal {  
  
}  
// 通过 Flyable接口 插到 mouse类，让鼠鼠获得飞行能力  
class mouse extends Animal implements Flyable{  
    public void fly(){  
        System.out.println("鼠鼠获得飞行能力！");  
    }  
}  
class pig extends Animal implements Flyable {  
    public void fly(){  
        System.out.println("冲天飞猪！");  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Flyable m = new mouse();  
        m.fly(); // 鼠鼠获得飞行能力！  
  
        Flyable p = new pig();  
        p.fly(); // 冲天飞猪！  
    }  
}
```

5.8 Abstract class 和 interface 区别

- Abstract class 是半抽象的，Abstract class中有构造方法。
- interface 是完全抽象的，interface 中没有构造方法。
- class 和 class 之间只能单继承，interface 和 interface 之间支持多继承。
- 一个 class 可以同时实现多个 interface，一个 Abstract class 只能继承一个类（单继承）
- interface 中只允许出现常量和抽象方法

06. Exception

常见异常类型：

Java中的异常分为两大类：

- Checked Exception(非Runtime Exception)
- Unchecked Exception(Runtime Exception)

算数异常类：ArithmeticeXception

空指针异常类型：NullPointerException

类型强制转换类型：ClassCastException

数组负下标异常：NegativeArrayException

数组下标越界异常：ArrayIndexOutOfBoundsException

违背安全原则异常：SecurityException

文件已结束异常：EOFException

文件未找到异常：FileNotFoundException

字符串转换为数字异常：NumberFormatException

操作数据库异常：SQLException

输入输出异常：IOException

方法未找到异常：NoSuchMethodException

下标越界异常：IndexOutOfBoundsException

系统异常：SystemException

创建一个大小为负数的数组错误异常：NegativeArraySizeException

数据格式异常: NumberFormatException

安全异常: SecurityException

不支持的操作异常: UnsupportedOperationException

网络操作在主线程异常: NetworkOnMainThreadException

请求状态异常: IllegalStateException (extends RuntimeException ,

父类: IllegalComponentStateException

在不合理或不正确时间内唤醒一方法时出现的异常信息。换句话说, 即 Java 环境或 Java 应用不满足请求操作)

网络请求异常: HttpHostConnectException

子线程Thread更新UI view 异常: ViewRootImpl\$CalledFromWrongThreadException

证书不匹配的主机名异常: SSLEngineException

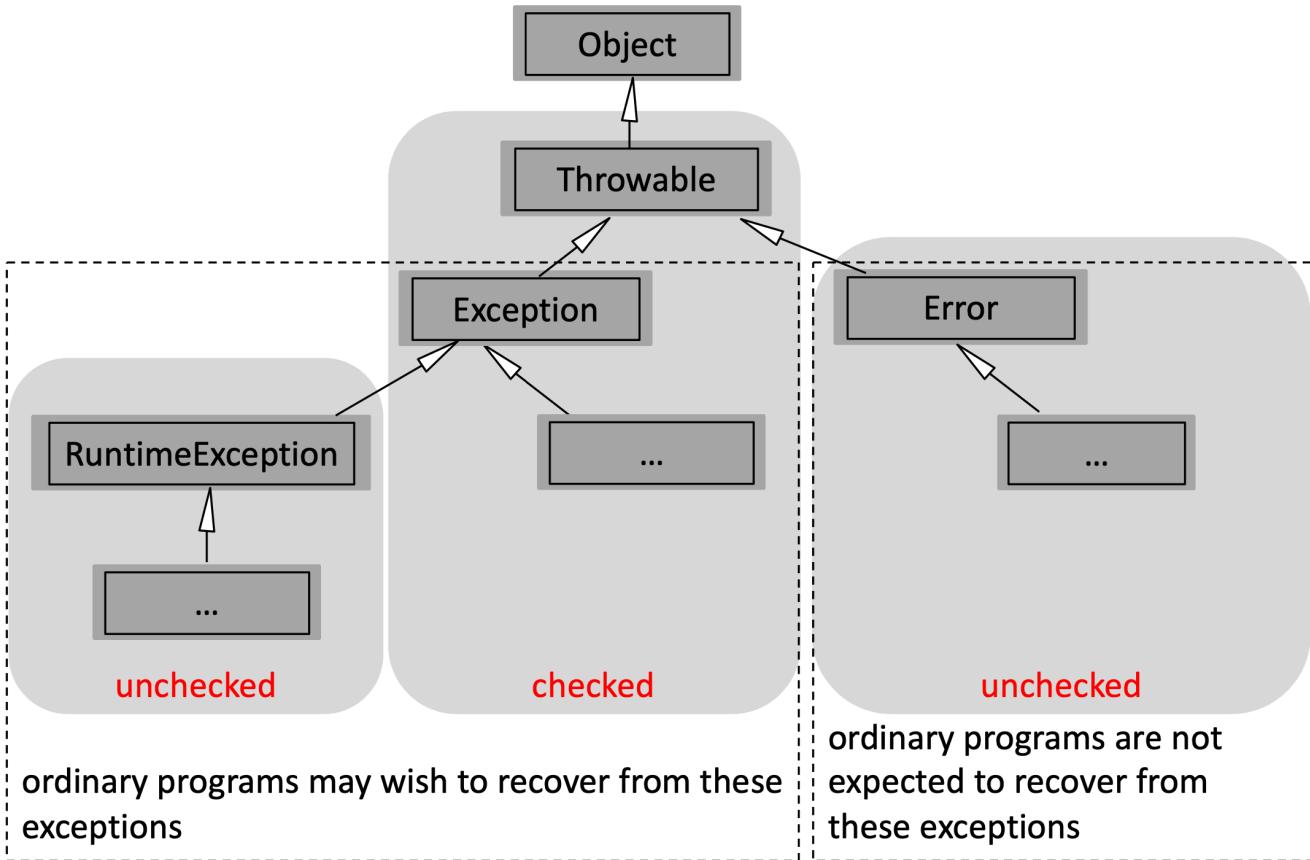
反射Method.invoke(obj, args...)方法抛出异常: InvocationTargetException

EventBus使用异常: EventBusException

非法参数异常: IllegalArgumentException

参数不能小于0异常: ZeroException

- 异常(Exception) 在java中以类和对象的形式存在
- 程序执行过程中发生了不正常的情况叫做: 异常
- java提供把该异常信息打印输出到控制台, 供程序员参考
- 程序是看到异常信息之后, 可以对程序进行修改, 让程序更加的健壮(Robust)



`Object ==> Throwable (可抛出的) ==> Error (不可处理, 直接退出JVM) 和 Exception (可处理的)`

`Exception ==>`

编译时异常 (ExceptionSubClass) (要在编写程序阶段必须预先对这些异常进行处理, 如果不处理编译器报错)

运行时异常 (RuntimeException) (在编写程序阶段程序员可以预先处理, 也可以不管)

```

public static void main(String[] args) {
    /*
     * main 方法中调用doSome() 方法
     * 因为dosome() 方法声明位置上有: throws ClassNotFoundException
     * 我们在调用doSome() 方法的时候必须对这种异常进行预先的处理。
     *如果不处理, 编译器就报错。
     *编译器报错信息: Unhandled exception: java. Lang. ClassNotFoundException
     */
    doSome(); // 编译时异常, 报错
    /*
     * 代码报错的原因是什么?
     * 因为dosome() 方法声明位置上使用了: throws ClassNotFoundException
     * 而ClassNotFoundException 是编译时异常。必须编写代码时处理, 没有处理, 编译器报错。
     */
}
/*
 * doSome 方法在方法声明的位蛋上使用了: throws ClassNotFoundException
 * 这个代码表示doSome() 方法在执行过程中, 有可能会出现ClassNotFoundException异常。

```

叫做类没找到异常。这个异常直接父类是：Exception，所NcLassNotFoundException 属于编译时异常。

```
@throws ClassNotFoundException
*/
public static void doSome() throws ClassNotFoundException {
    System.out.println("doSome");
}
```

Java语言中对异常的处理包括两种方式：

- 【异常上抛】：在方法声明的位置上，使用throws关键字，抛给上一级。谁调用我，我就抛给谁。
- 【异常捕捉】：使用try..catch语句进行异常的捕捉。这件事发生了，谁也不知道，因为我给抓住了。

```
// 【异常上抛】使用throws关键字
public static void main(String[] args) throws ClassNotFoundException {
    doSome(); // 运行成功
}
public static void doSome() throws ClassNotFoundException {
    System.out.println("doSome");
}
```

```
// 【异常捕捉】使用try..catch
public static void main(String[] args) throws ClassNotFoundException {
    try {
        doSome(); // 运行成功
    } catch (ClassNotFoundException e) {
        throw new RuntimeException(e);
    }
}
public static void doSome() throws ClassNotFoundException {
    System.out.println("doSome");
}
```

try / catch / finally

- try 中包含了可能产生异常的代码
- 当 try 中的代码出现异常时，出现异常下面的代码不会执行，马上会跳转到相应的 catch 语句块中，如果没有异常不会跳转到 catch 中
- finally 表示不管是出现异常，还是没有出现异常，finally 里的代码都执行，finally 和 catch 可以分开使用，但 finally 必须和 try 一块使用。finally 在任何情况下都会执行，通常在 finally 里关闭资源

```
// catch 后面的小括号中的类型可以是具体的异常类型，也可以是该异常类型的父类型
// catch 可以写多个。建议 catch 的时候，精确的一个一个处理。这样有利于程序的调试
// catch 写多个的时候，从上到下，必须遵守从小到大。
```

```
try {
    //创建新入流
```

```

fis = new FileInputStream("/Users/chenziyang/Desktop/Sample.txt");
String s = null; // 这里一定会出现空指针异常!
s.toString();
// 读文件
fis.read();
} catch(FileNotFoundException e){
    System.out.println("文件不存在! ");
} catch(IOException e){
    System.out.println("读文件报错了! ");
} catch (NullPointerException e) {
    e.printStackTrace();
} finally { // 流使用完需要关闭，因为流是占用资源的。
    if (fis != null) { // 避免空指针异常
        try {
            fis.close(); // close()方法有异常，采用捕捉的方式
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
}

```

Example:

计算两个int类型数据的商，计算成功返回true，计算失败返回false

```

public static boolean divide(int a, int b){
    try {
        int c = a / b;
        // 代码没有发生异常，表示执行成功，继续运行 (eg. int c = 10 / 5)
        // 代码发生异常，表示执行失败，直接跳到 catch语句 (eg. int c = 10 / 0)
        return true;
    } catch (Exception e){
        // 程序执行到此处表示以上程序出现了异常!
        // 表示执行失败!
        return false;
    }
}

```

```

public static void main(String[] args) {
    // System.out.println(10 / 0); error: java.lang.ArithmetricException: / by zero

    boolean result = divide(10, 2);
    System.out.println(result); // true

    boolean result2 = divide(10, 0);
    System.out.println(result2); // false
}

```

思考：以上的这个方法设计没毛病，挺好，返回true或false表示两种情况，但是在以后的开发中，有可能遇到一个方法的执行结果可能包括三种情况，四种情况，五种情况不等，但是每一个都是可以数清楚的，一枚一枚都是可以列举出来的。这个布尔类型就无法满足需求了。此对需要使用java语言中的枚举类型

Enum

- 一枚一枚可以列举出来的，才建议使用枚举类型
- 枚举编译之后也是生成class文件
- 枚举也是一种引用数据类型
- 枚举中的每一个值可以看做是常量

```

enum Result{
    SUCCESS, FAIL;
}

```

```

public static Result divide(int a, int b){
    try {
        int c = a / b;
        return Result.SUCCESS;
    } catch (Exception e){
        return Result.FAIL;
    }
}

```

```

public static void main(String[] args) {
    Result r = divide(10, 2);
    System.out.println(r == Result.SUCCESS ? "计算成功" : "计算失败"); // 计算成功
}

```

Example:

```

public enum Color{

```

```

RED(10), GREEN(20), BLUE(30);
int number; // fields

Color(int num){ // constructors
    this.number = num; // 让 RED(10) 里面的10赋值到number里
}
int getNumber(){
    return number
}
int compareNumber(Color clr){ // methods
    // return this.number - clr.number;
    return this.number.compareTo(clr.number) // 小于返回负数，等于返回零，大于返回正数
}
}

```

```

public static void main(String[] args) {
    System.out.println(Color.RED.getNumber()); // 10
    System.out.println(Color.GREEN.getNumber()); // 20
    int result = Color.RED.compareNumber(2); // result = -8

    // Enum 可以用switch语句
    switch (Season.RED) {
        case RED:
            System.out.println("红色");
            break;
        case GREEN:
            System.out.println("绿色");
            break;
        case BLUE:
            System.out.println("蓝色");
            break;
    } // Output: 红色
}

```

自定义异常 throws 和 throw

我们可以自己定义异常类 Define Your Own Exceptions

JDK内置的异常肯定是不够的用的。在实际的开发中，有很多业务，这些业务出现异常之后，JDK内部是没有的

自定义异常：

- 第一步：编写一个类继承Exception 或者RuntimeException
- 第二步：提供两个构造方法，一个无参数的，一个带有string参数的

```
// 第一种 自定义 编译时异常(Exception) 异常
```

```

class MyException1 extends Exception { // 编译时异常 checked
    public MyException1(){ // 提供两个构造方法，一个无参数的，一个带有string参数的

    }
    public MyException1(String s){
        super(s);
    }
}

// 第二种 自定义 运行时异常(RuntimeException) 异常
class MyException2 extends RuntimeException { // 运行时异常 unchecked
    public MyException2(){ // 提供两个构造方法，一个无参数的，一个带有string参数的

    }
    public MyException2(String s){
        super(s);
    }
}

// 第三种 自定义 编译时异常(Exception) 异常
class MyException3 extends Throwable{ // Checked

}

// 第四种 自定义 运行时异常(Error) 异常
class MyException4 extends Error{ // Unchecked

}

// 第五种 自定义 编译时异常(Exception) 异常
class MyException5 extends MyException1{ // Checked

}

```

```

import java.util.Scanner;
public class MyException {

    public void exception() throws MyException2 {
        System.out.println("Enter 0 to trigger exception: ");
        int value;
        Scanner sc = new Scanner(System.in);
        value = sc.nextInt();
        if (value == 0){
            throw new MyException2("二号异常");
            // Exception in thread "main" Draft.MyException2: 二号异常
        }
        System.out.println("没有异常");
    }
}

```

```

public static void main(String[] args) {
    MyException1 e1 = new MyException1("一号异常"); // 创建异常对象
    // 只 new 了异常对象，并没有手动抛出

    e1.printStackTrace(); // 打印异常堆栈信息 ==> Draft.MyException1: 一号异常

    String msg = e1.getMessage(); // 获取异常简单描述信息
    System.out.println(msg); // Output: 一号异常

    MyException e2 = new MyException();
    e2.exception(); // Exception in thread "main" Draft.MyException2: 二号异常
}
}

```

Example: 假设Stack里面只能压栈5次

```

public class MyStack {

    public void push(Object o) throw MyStackoperationException {
        if (index >= elements.length - 1){
            throw new MyStackoperationException("压栈失败");
        }
        //程序能够执行到此处说明 stack 还有位置
        data[++top] = o;
        System.out.print("压栈成功");
    }

    public void pop() throw MyStackoperationException {
        if (index < 0) {
            throw new MyStackoperationException("弹栈失败");
        }
        //程序能够执行到此处说明 stack 没有空
        data[top--];
        System.out.println("弹栈成功");
    }
}

```

```

public static void main(String[] arg) {
    MyStack stack = new MyStack();

    try {
        stack.push(new Object()); // 压栈成功
        stack.push(new Object()); // 压栈成功
        stack.push(new Object()); // 压栈成功

```

```

        stack.push(new Object()); // 压栈成功
        stack.push(new Object()); // 压栈成功
        // 此时 stack 满了
        stack.push(new Object()); // 不会运行，直接跳到 catch
    } catch (MyStackoperationException e) {
        System.out.println(e.getMessage()); // Output: 压栈失败
    }

    try {
        stack.pop(); // 弹栈成功
        // 此时 stack 已经空了，弹栈失败
        stack.pop(); // 不会运行，直接跳到 catch
    } catch (MyStackoperationException e) {
        System.out.println(e.getMessage()); // Output: 弹栈失败
    }
}

```

总结异常中的关键字：

- 异常捕捉：try catch finally
- throws 在方法声明位置上使用，表示上报异常信息给调用者
- throw 手动抛出异常！

07. Generics

Generics 范型

我们使用 <> (The Diamond)来表示 范型 (Generic Types)

- 泛型类(generic class) 是一个类 参数化(parameterized) 的w.r.t.一个或多个类型。
- 原始类型(raw type) 是没有任何 type arguments 的 generic class or interface 的名称

常用简写：

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

```
public class Stack<T>{    // Stack<T>'s 原始类型(raw type) is Stack
                           // T must be a 引用类型(reference type) !
{
                           // No Stack<int> or Stack<char>!!!
```

以前我们Copy and modify:

```
public class CarStack{
    public CarStack(){ ... }
    public void push(Car c){ ... }
    public Car top(){ ... }
    public void pop(){ ... }
}
```

```
public class FoodStack{
    public FoodStack(){ ... }
    public void push(Food s){...}
    public Food top(){ ... }
    public void pop(){ ... }
}
```

现在我们用Generics:

```
public class Stack{
    public Stack(){ ... }
    public void push(Object s){ ... }
    public Object top(){ ... }
    public void pop(){ ... }
}
```

```
Stack s = new Stack();
s.push(car);
Food f = (Food)(s.top());
```

以前我们的class

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

现在我们的class

```

/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}

```

Type Inference类型自动判断

ArrayList<>这里的类型会自动判断，自动类型判断又称为钻石表达式(The Diamond)！

```

public static void main(String[] args){
    List<Animals> myList = new ArrayList<>; // 类型自动推断!
}

```

```

public class BoxDemo {

    public static <U> void addBox(U u, java.util.List<Box<U>> boxes) {
        Box<U> box = new Box<>();
        box.set(u);
        boxes.add(box);
    }

    public static <U> void outputBoxes(java.util.List<Box<U>> boxes) {
        int counter = 0;
        for (Box<U> box: boxes) {
            U boxContents = box.get();
            System.out.println("Box #" + counter + " contains [" + boxContents.toString() +
" ]");
            counter++;
        }
    }

    public static void main(String[] args) {
        java.util.ArrayList<Box<Integer>> listOfIntegerBoxes = new java.util.ArrayList<>();
        BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);
        BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);
        BoxDemo.addBox(Integer.valueOf(30), listOfIntegerBoxes);
        BoxDemo.outputBoxes(listOfIntegerBoxes);
    }
}

```

```
}
```

```
// Output:  
Box #0 contains [10]  
Box #1 contains [20]  
Box #2 contains [30]
```

Multiple Type Parameters

泛型类可以有多个类型参数。例如，泛型OrderedPair类，它实现了泛型Pair接口：

```
public interface Pair<K, V> {  
    public K getKey();  
    public V getValue();  
}  
  
public class OrderedPair<K, V> implements Pair<K, V> {  
  
    private K key;  
    private V value;  
  
    public OrderedPair(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
  
    public K getKey() { return key; }  
    public V getValue() { return value; }  
}
```

```
// Java 编译器可以从OrderedPair<String, Integer>中推断出K和V类型  
// 所以可以使用The Diamond表示法缩短这些语句：  
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);  
Pair<String, String> p2 = new OrderedPair<String, String>("hello", "world");
```

代码 **new OrderedPair<String, Integer>** 将 K 实例化为 String 并将 V 实例化为 Integer。

因此，OrderedPair的构造函数的参数类型分别为String和Integer。由于 autoboxing，将String和int传递给类是有效的。

您还可以将类型参数（即K或V）替换为参数化类型（即List）

例如，使用OrderedPair<K, V>示例：

```
OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new Box<Integer>(...));
```

Raw type

原始类型是没有任何类型参数的泛型类或接口的名称。例如，给定泛型Box类：

```
public class Box<T> {
    public void set(T t) {

    }
}
```

要创建Box的参数化类型 (parameterized type)，您需要为形式类型参数T (formal type parameter T) 提供一个实际的类型参数 (actual type argument)：

```
Box<Integer> intBox = new Box<>();
```

如果省略了实际的类型参数 (actual type argument is omitted)，则创建一个原始类型 (raw type) Box：

```
Box rawBox = new Box();
```

所以，Box 是泛型类型 (generic type) Box<T> 的原始类型 (raw type)。

但是，非泛型类 (non-generic class) 或接口类型 (interface type) 不是原始类型

Bounded Type Parameters

有时我们希望限制可以在参数化类型 (parameterized type) 中用作类型参数的类型 (type arguments)

```
Box<Integer> integerBox = new Box<Integer>();
```

Generics, Inheritance, and Subtypes

- Stack is a composite type

```
Stack<Number> numStack = new Stack<Number>();
numStack.push(new Integer(10)); // OK, an Integer is a Number
numStack.push(new Double(10.1)); // OK, a Double is a Number
```

```

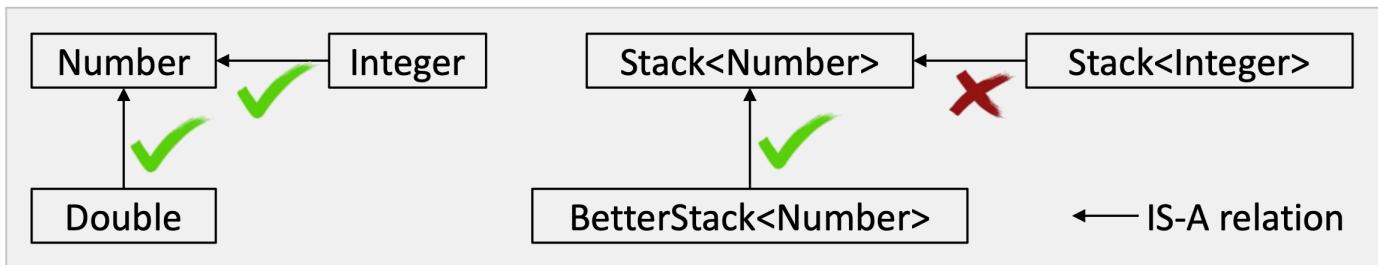
public void stackTest(Stack<Number> n) {
}

class BetterStack<T> extends Stack<T>{

}

stackTest(new Stack<Integer>()); // Error!
stackTest(new Stack<Double>()); // Error!
stackTest(new BetterStack<Number>()); // OK!

```



Wildcards(通配符)

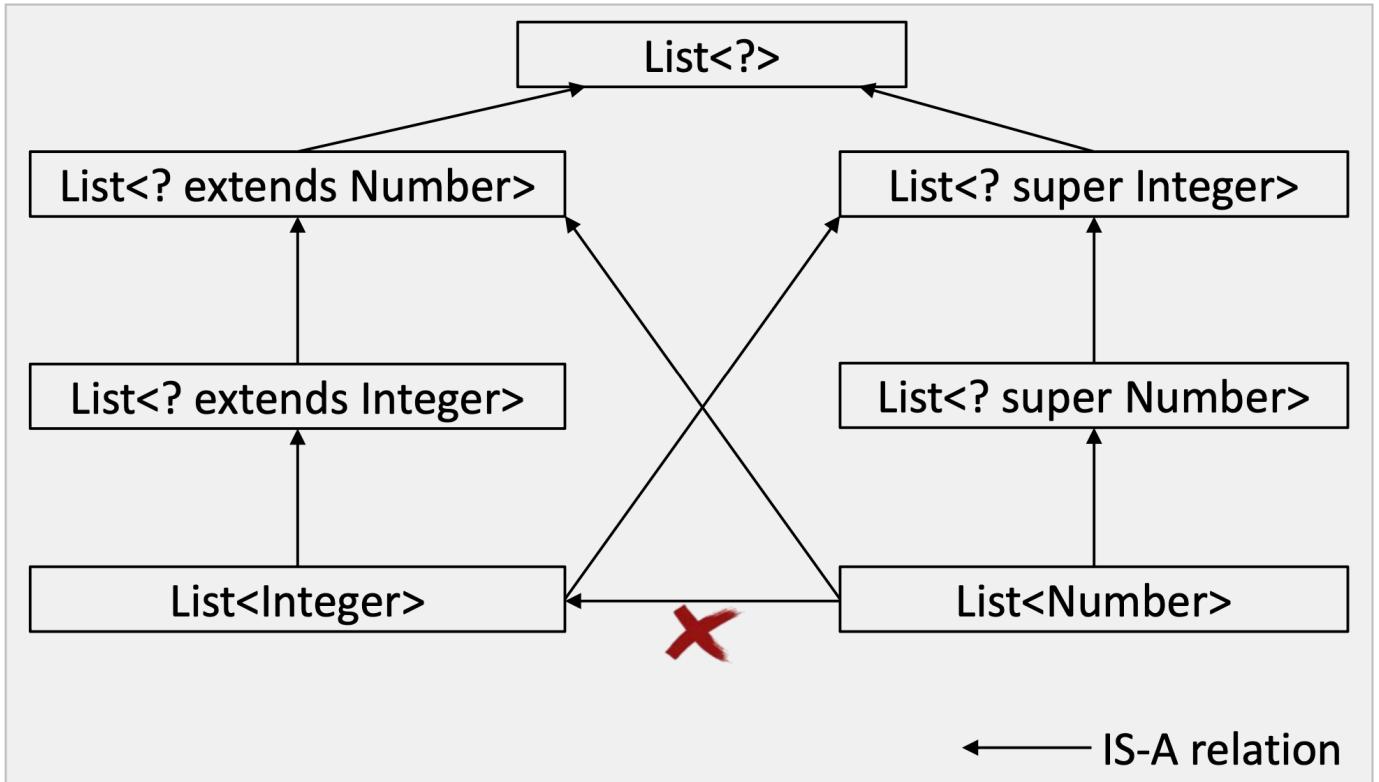
- 在通用代码(Generics)中，称为 **Wildcards(通配符)** 的问号 (?) 表示未知类型。通配符可用于多种情况：作为 **type of a parameter, field, or local variable**
- 我们可以通过使用 **Wildcards(通配符)** 更加细致的描述所需泛型的类型

通配符指南：

"in" 变量为代码提供数据 是copy(src, dest) 中的 src 参数，提供要复制的数据

"out" 变量保存在其他地方使用的数据 是copy(src, dest) 中的 dest 参数，接受数据

- 使用 `extends` 关键字，使用上限通配符定义 "in" 变量
- 使用 `super` 关键字，使用下界通配符定义 "out" 变量
- 如果可以使用 `Object` 类中定义的方法访问 "in" 变量，请使用无界通配符
- 如果代码需要以 "in" 和 "out" 变量的形式访问变量，请不要使用通配符



有三种使用 **Wildcards(通配符)** 的方式：

1. Unbounded wildcard(无限制通配符类型) 表现的类型是未知的，不是Object，是不知道什么类型。

使用 "?" 作为泛型的通配符 `List<?>`，这样可以接收所有的泛型类型的对象实例，但是不允许进行内容的修改，而只是完成内容的获取

```
public static void printList(List<Object> list) {
    for (Object elem : list)
        System.out.println(elem + " ");
    System.out.println();
}
```

`printList` 的目标是打印任何类型的列表，但它没有实现这个目标——它只打印Object实例的列表；它不能打印List, List, List, 等等，因为它们不是List的子类型。

要编写通用的`printList`方法，请使用`List<?>`：

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
```

Because for any concrete type `A`, `List<A>` is a subtype of `List<?>`, you can use `printList` to print a list of any type:

```
List<Integer> li = Arrays.asList(1, 2, 3);
List<String> ls = Arrays.asList("one", "two", "three");
printList(li);
printList(ls);
```

而在 "?" 通配符的操作基础之上，Java 又给出了两个子通配符的处理格式：

2. 上限通配符(Upper Bounded Wildcards)

【类和方法】设置泛型的上限 `? extends class`：只能够使用当前类或者当前类的子类作为泛型类型：

`List<Number>` 更有限制，只匹配类型为 Number 的 List

相比于 `List<? extends Number>` 匹配类型为 Number 或其任何子类的 List

```
// 可以使用 Number 或者是 Number 子类实现泛型类型的定义
// 使用上限通配符(Upper Bounded Wildcards) <? extends Number> 设置上限
// 不能大于Number类，可以使用他的子类：Integer, Double, Float等
```

```
// 返回数列中数字的总和：
public static double sumOfList(List<? extends Number> list) {
    double s = 0.0;
    for (Number n : list){
        s += n.doubleValue();
    }
    return s;
}

public static void main (String[] args) {
    // 下面的代码，使用一个Integer对象列表，输出sum = 6.0：
    List<Integer> li = Arrays.asList(1, 2, 3);
    System.out.println("sum = " + sumOfList(li));

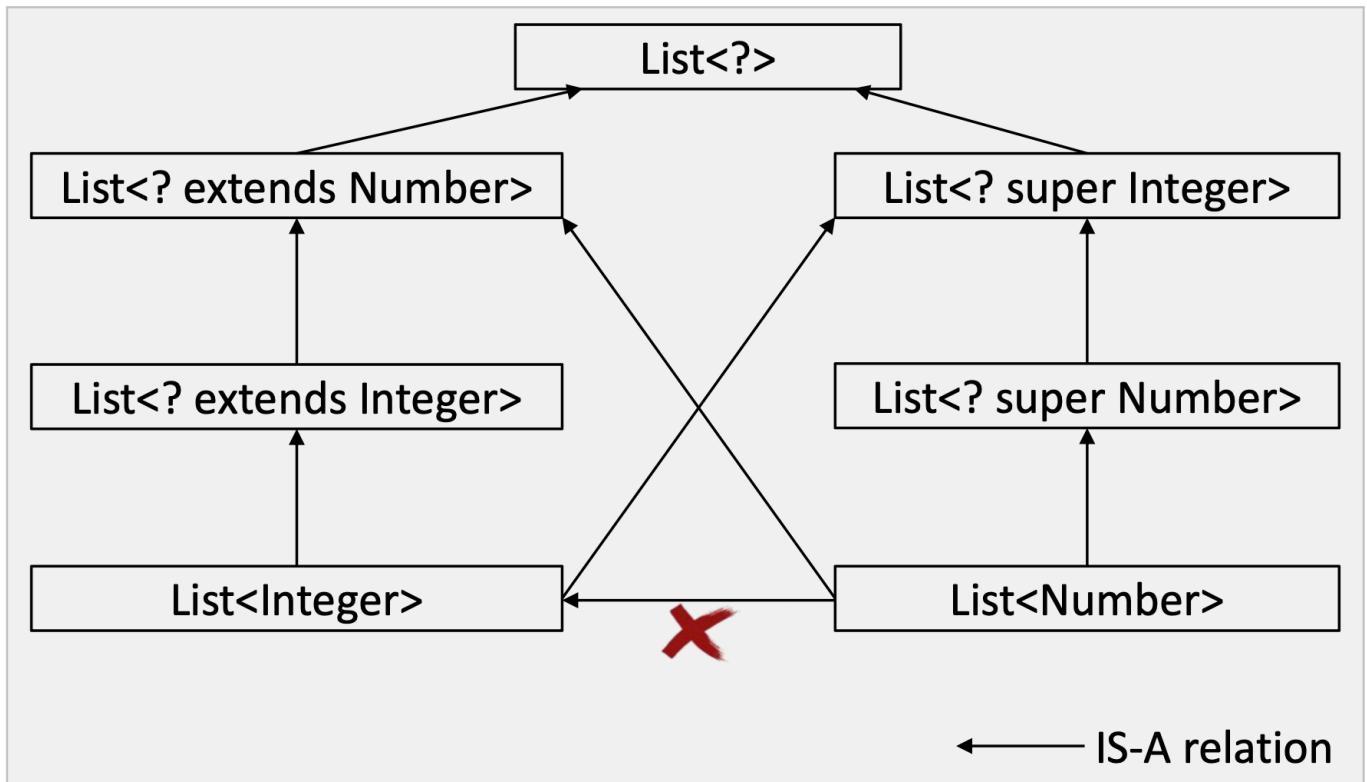
    // Double值列表可以使用相同的sumOfList方法。下面的代码打印sum = 7.0：
    List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);
    System.out.println("sum = " + sumOfList(ld));
}
```

3. 下限通配符(Lower bounded wildcard) 使用此类型或者是其父类型 `? super Integer`

设计一个可以将 `List<Integer>` `List<Number>` 和 `List<Object>` 放进列表的方法：

```
public static void addNumbers(List<? super Integer> list) {  
    for (int i = 1; i <= 10; i++) {  
        list.add(i);  
    }  
}
```

```
public static void main (String[] args) {  
    List<Integer> li  
    List<Number> ld  
    List<Object> le  
}
```

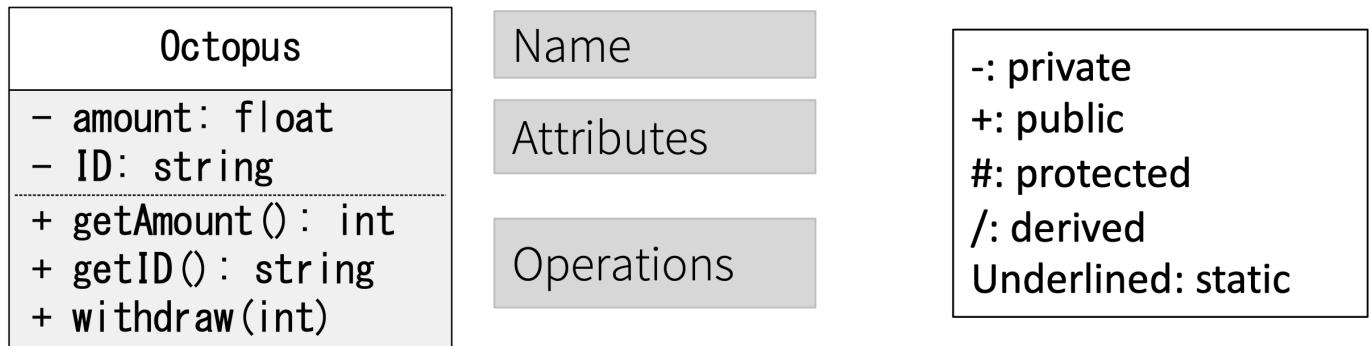


08. Unified Modeling Language (UML)

- 统一建模语言(Unified Modeling Language, UML)是一种为面向对象系统的产品进行说明、可视化和编制文档的一种标准语言，是非专利的第三代建模和规约语言。UML是面向对象设计的建模工具，独立于任何具体程序设计语言

我们来看看怎么用 UML 来表示 class: 类封装状态(属性)和行为

- Each attribute has a type
- Each operation(Method) has a signature



General relationship

Dependency

- 依赖类和独立类(模块)之间的语义连接
- 如果对一个类(模块)(服务器或目标)定义的更改可能导致对另一个类(客户端或源)的更改。



Instance-level

Association

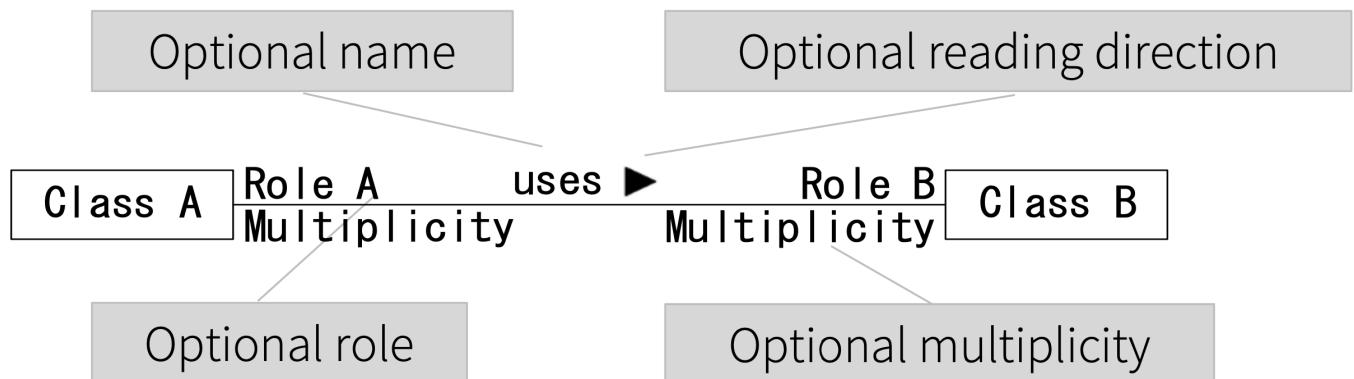
- An association 是类实例(class instances)之间的一种 关系类型(type of relation)
- 类的对象可以使用 association 进行通信(communicate)

eg.

Class A has an attribute of type B

Class A creates instances of B

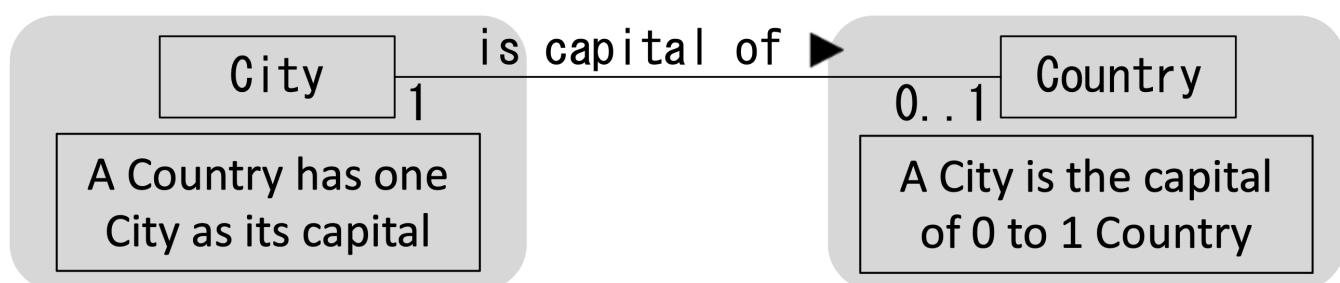
Class A received a message with argument of type B



Association Multiplicity

Multiplicity 是类中有多对象参与到关系中

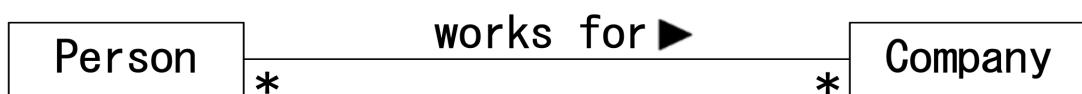
- 1-to-1



- 1-to-many



- Many-to-many



Aggregation (Special Associations)

- “part-of” relation between objects
- Component can be part of multiple aggregates
- Component can be created and destroyed independently of the aggregate



```
public class Curriculum{ List<Course> getCourses(){...}; ... }
```

Composition (Special Associations)

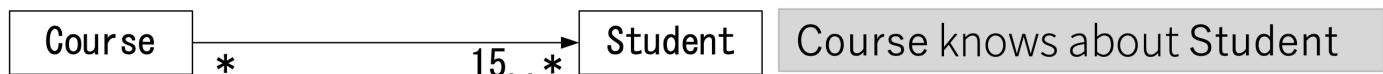
- strong aggregation
- A component can only be part of a single aggregate
- Exists only together with the aggregate



```
public class Human{ List<Leg> getLegs(){...}; ...}
```

Navigability of Association

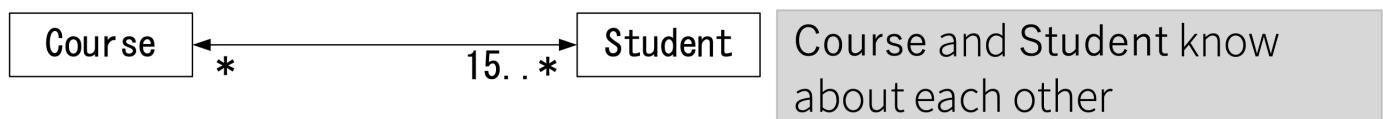
方向(Direction)表示是否可以通过该 association 访问对象



```
public class Course{ List<Student> getStudents(){...} ...}
```

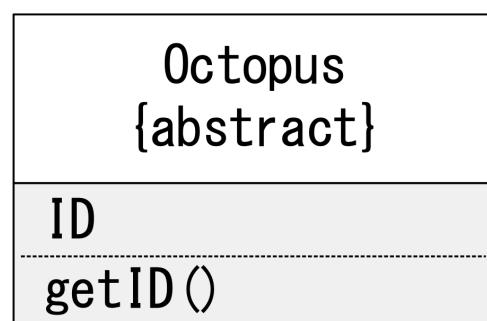
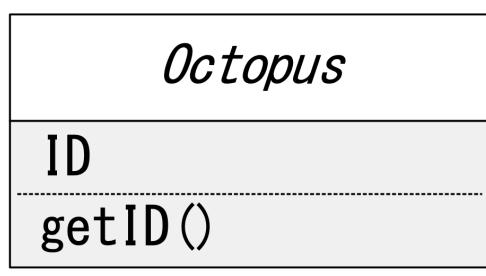


```
public class Student{ List<Course> getCourses(){...} ...}
```



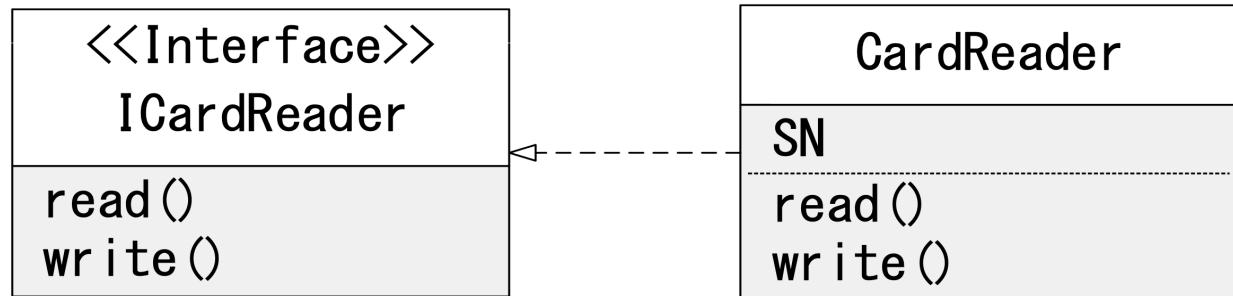
Abstract and Interface Classes

Abstract classes 具有 斜体的类名 或 {Abstract} 属性(也适用于operation)



Interface classes 用 stereotype: <> 表示

Classes implement an interface using an implementation relation

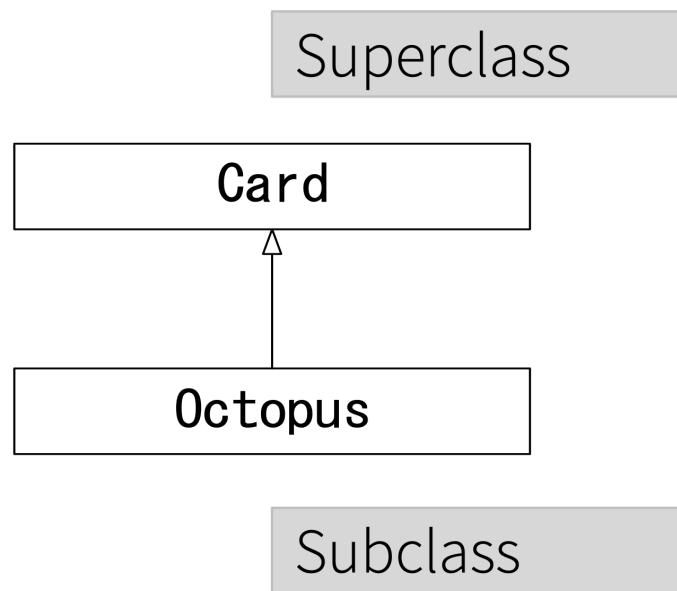


stereotype 允许设计人员扩展 vocabulary of UML, 以便从现有的模型元素中衍生(derived)/创建(create) new model elements

Class-level

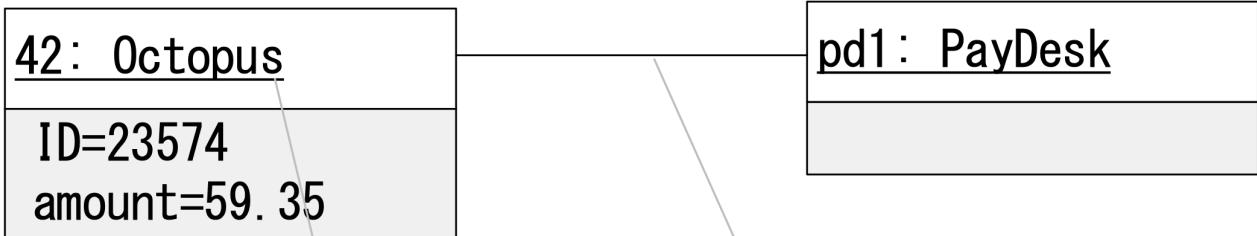
Generalization

- Generalization relation 表示一种 (“Is-A”) 关系
- Generalization 是通过继承实现的
- 子类继承父类的 attributes 和 operations
- Generalization 通过消除冗余来简化模型

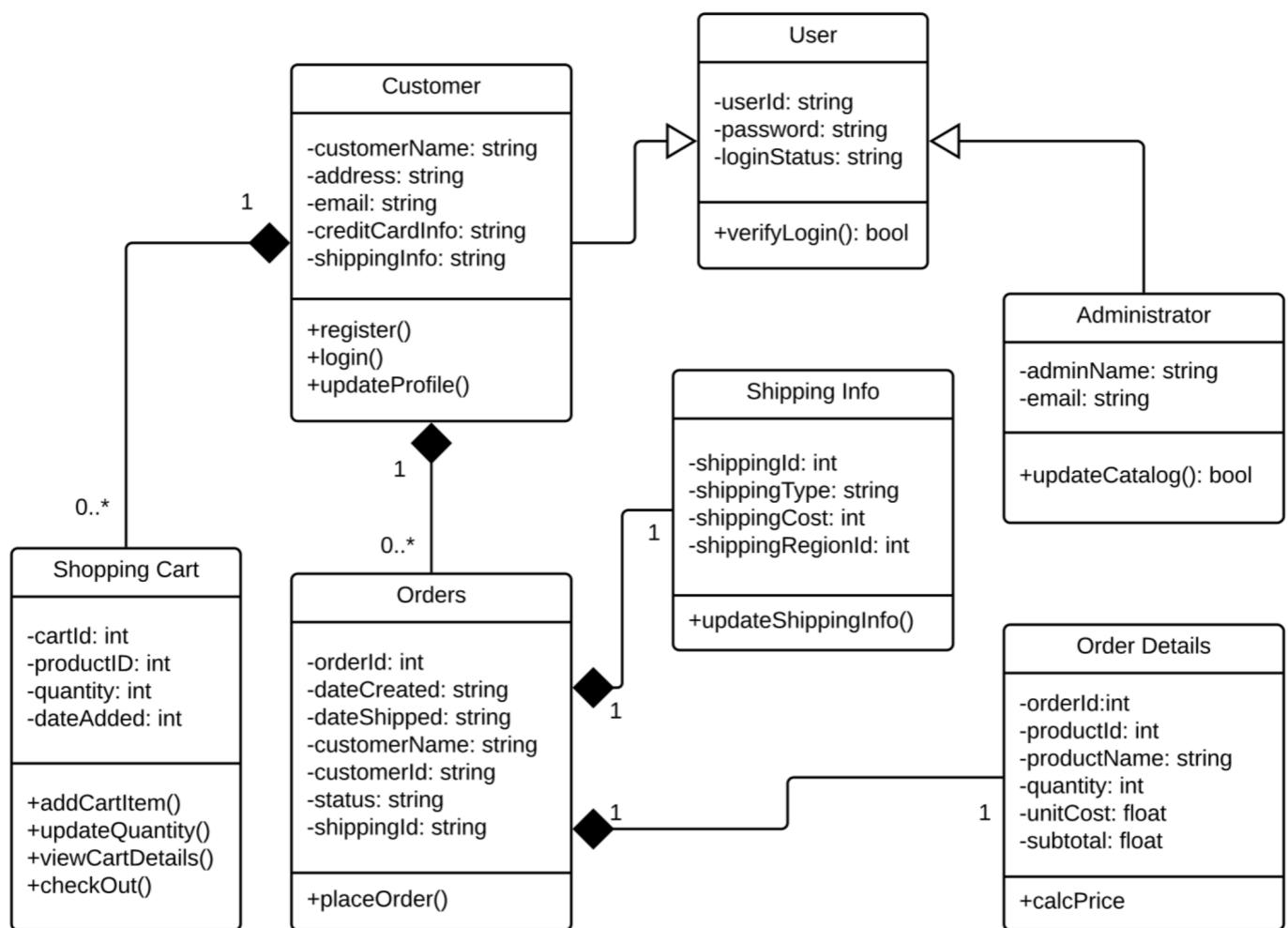


UML Object Diagrams

- An Object diagram 是在运行时表示系统的 **snapchat**
- It shows the **existing objects**, 它们在特定时间点的“属性值(attribute values)”和“关系(relations)”
- Methods 不是对象图(object diagram)的一部分

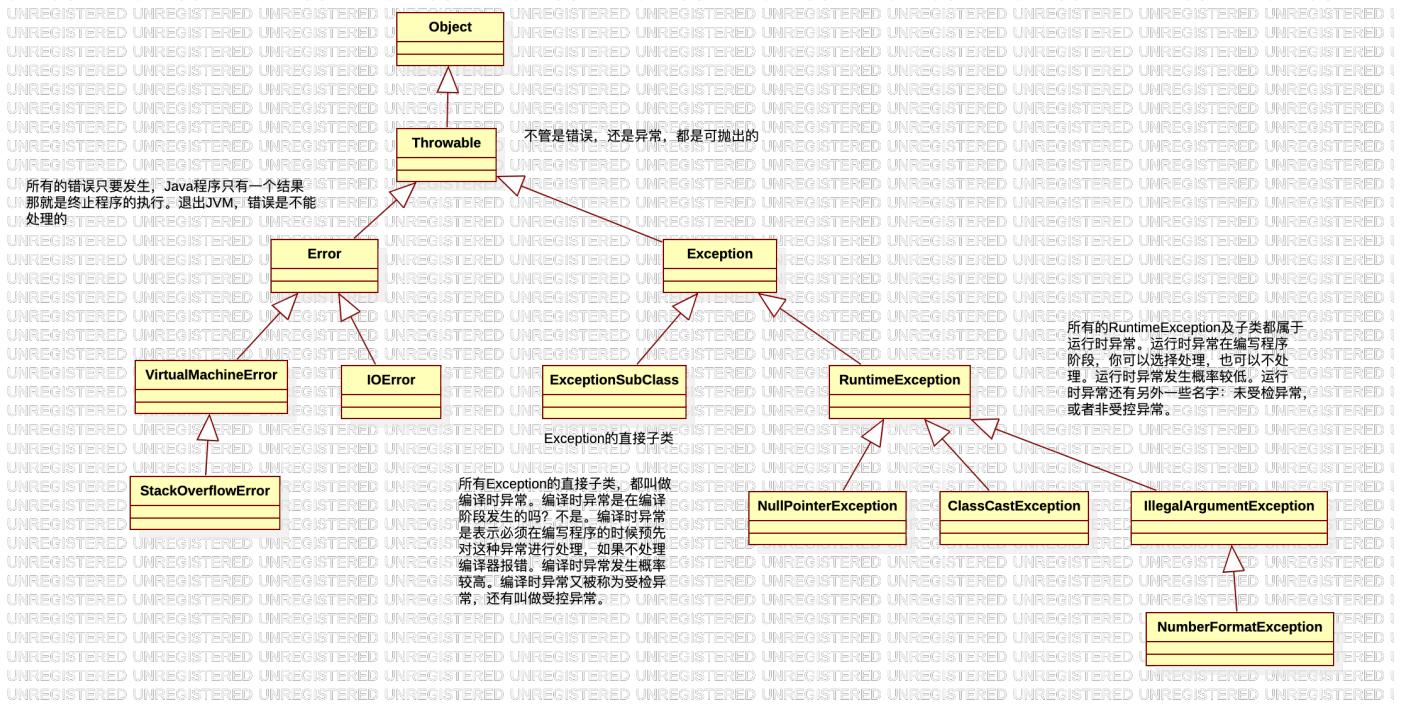


Example Class Diagram

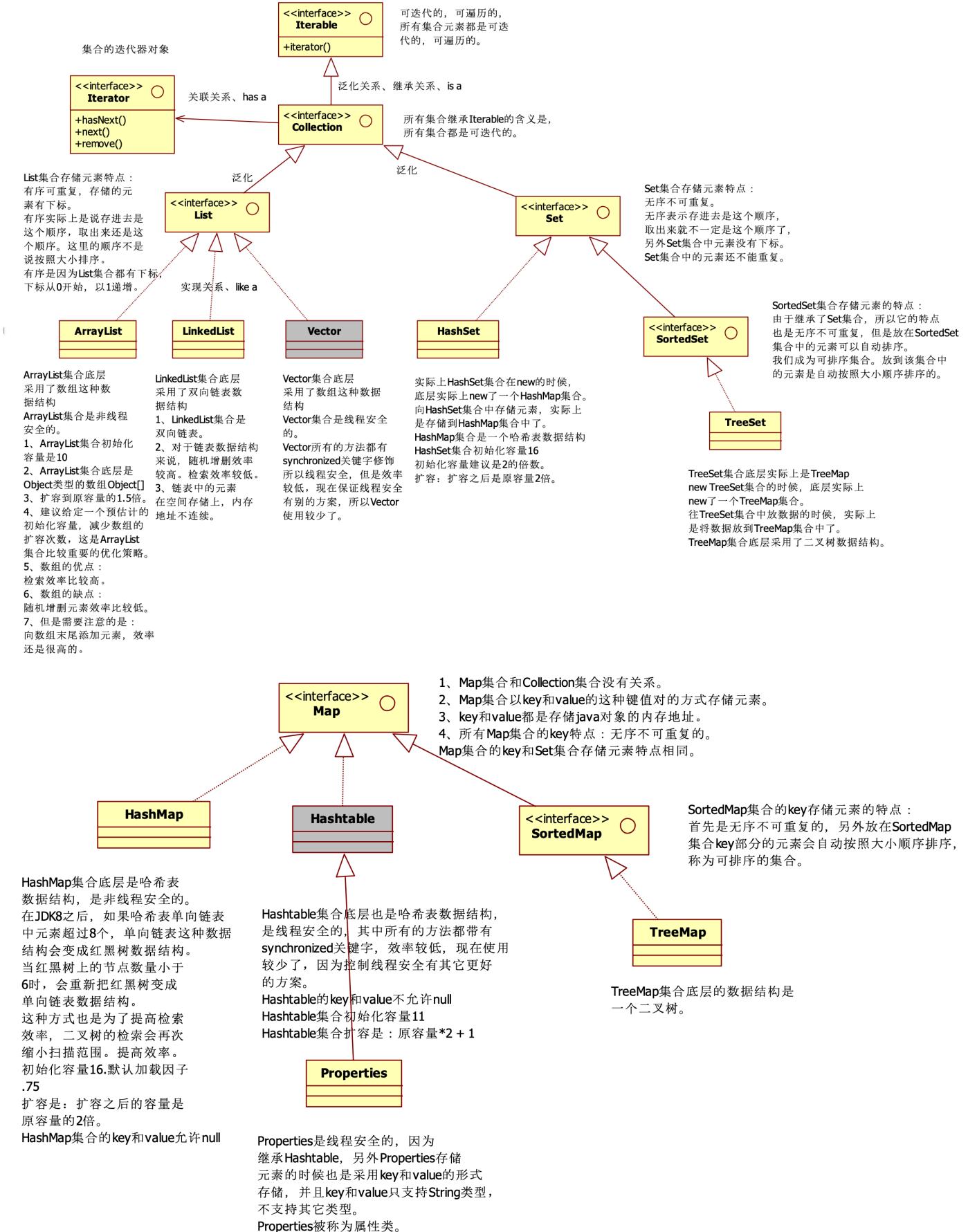


UML Example

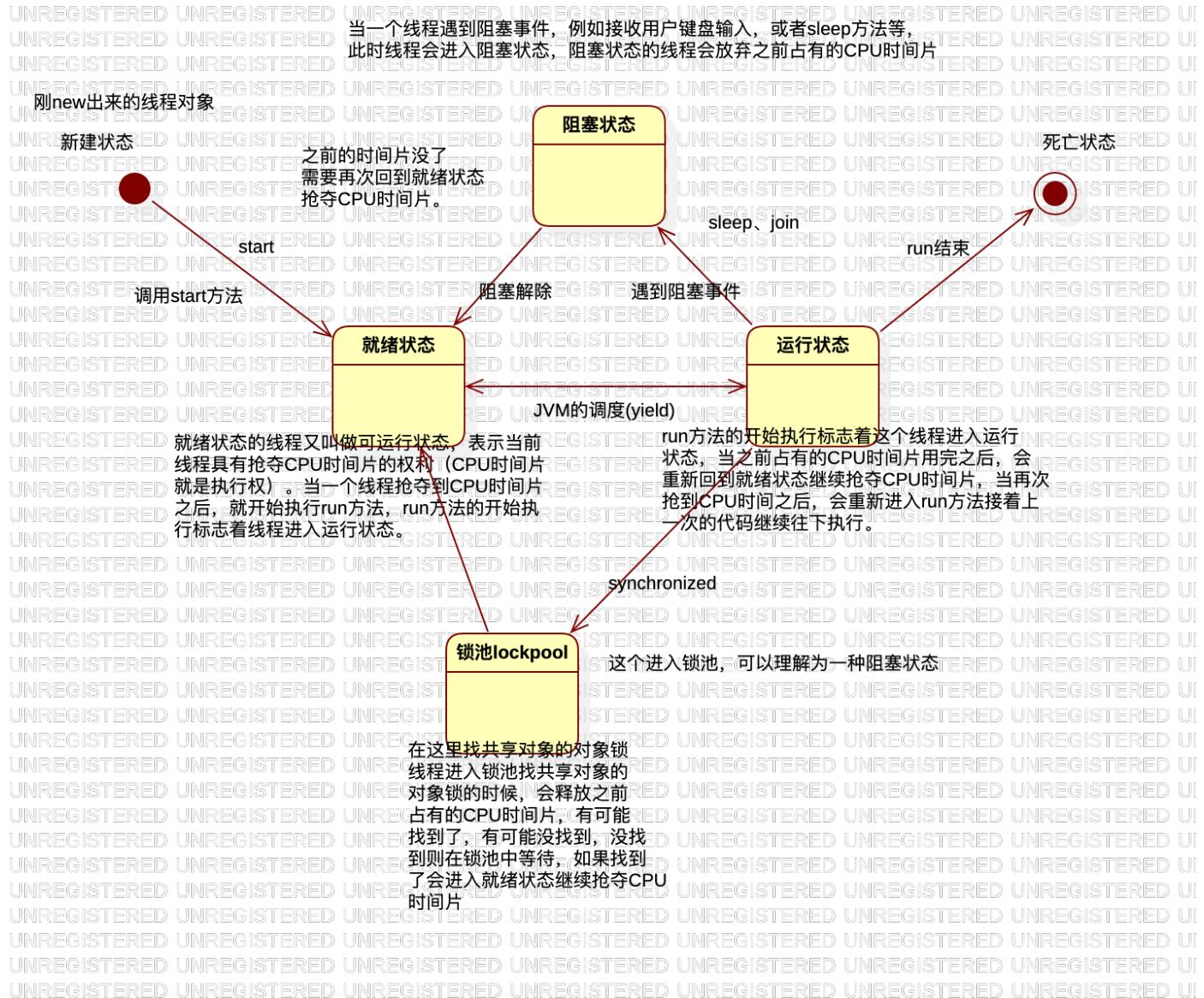
异常继承图UML:



集合继承结构图UML:



线程生命周期UML:



08. Thread

Thread Methods

Sr.No.	Method & Description
1	public void start() Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	public void run() If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
3	public final void setName(String name) Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	public final void setPriority(int priority) Sets the priority of this Thread object. The possible values are between 1 and 10.
5	public final void setDaemon(boolean on) A parameter of true denotes this Thread as a daemon thread.
6	public final void join(long millisec) The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
7	public void interrupt() Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	public final boolean isAlive() Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

前面的方法是在特定的 Thread 对象上调用的。以下方法在 Thread 类中的是静态的。调用其中一个静态方法在当前运行的线程上执行操作。

Sr.No.	Method & Description
1	public static void yield() Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.
2	public static void sleep(long millisec) Causes the currently running thread to block for at least the specified number of milliseconds.
3	public static boolean holdsLock(Object x) Returns true if the current thread holds the lock on the given Object.
4	public static Thread currentThread() Returns a reference to the currently running thread, which is the thread that invokes this method.
5	public static void dumpStack() Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

Processes and Threads

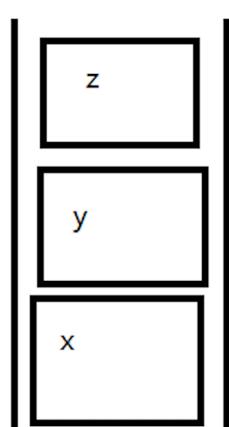
- 进程(Processes) 是一个应用程序 (1个进程是一个软件) 抽象一个正在运行的程序，包括程序计数器，寄存器，变量，...不同的进程有独立的地址空间，进程(Processes) 相当于一个 java虚拟机(JVM)
- 线程(Threads) 是一个 进程(Processes) 中的执行场景/执行单元
- 一个进程可以启动多个线程

线程A和线程B栈内存

main方法结束只代表主线程结束了，其它线程可能还在执行。

独立。不共享。

支栈：分支线程t1



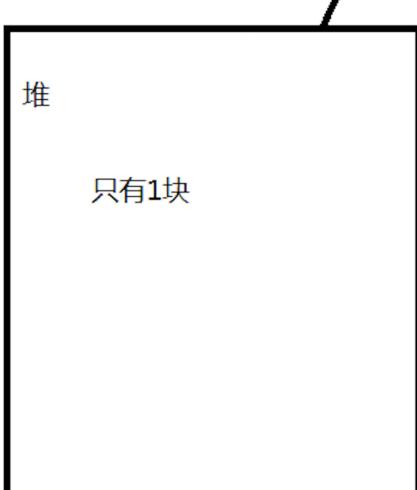
x y z三个方法在同一个线程中。

栈：线程main



主线程对应的主栈
main m1 m2三个方法在一个线程中。

多线程共享



堆
只有1块
方法区
只有1块。

进程(Processes) 可以看做是现实生活当中的公司。

线程(Threads) 可以看做是公司当中的某个员工。

阿里巴巴：**进程(Processes)**

马云CEO: 阿里巴巴的一个**线程(Threads)**

前台: 阿里巴巴的一个**线程(Threads)**

京东：**进程(Processes)**

强东: 京东的一个**线程(Threads)**

妹妹: 京东的一个**线程(Threads)**

火车站，可以看做是一个 **进程(Processes)**

火车站中的每一个售票窗口可以看做是一个 **线程(Threads)**

我在窗口1购票，你可以在窗口2购票，你不需要等我，我也不需要等你。

所以**多线程并发**可以提高效率。

- 进程A和进程B的内存独立不共享。（阿里巴巴和京东资源不会共享的！）
- 在java语言中：线程A和线程B，堆内存和方法区内存共享。但是栈内存独立，一个线程一个栈。
- 假设启动10个 **线程(Threads)**，会有10个栈空间，每个栈和每个栈之间，互不干扰，各自执行各自的，这就是**多线程并发**

什么是**多线程并发**？

t1线程执行t1的。

t2线程执行t2的。

t1不会影响t2，t2也不会影响t1。这叫做真正的**多线程并发**。

线程A：播放音乐

线程B：运行魔兽游戏

线程A和线程B频繁切换执行，人类会感觉音乐一直在播放，游戏一直在运行，给我们的感觉是同时并发的。

实现线程

java支持**多线程机制**。并且java已经将多线程实现了，我们只需要继承就行了。

实现线程有两种方式

- 写一个继承自Thread类的类
- 写一个实现Runnable接口的类

```
// 第一种方式：编写一个类，直接继承java.lang.Thread，重写run方法。  
// 定义线程类  
public class MyThread extends Thread{  
    public void run(){  
        System.out.println("主线程");  
    }  
}  
// 创建线程对象  
MyThread t = new MyThread();  
// 启动线程。  
t.start();
```

```
// 第二种方式：编写一个类，实现java.lang.Runnable接口，实现run方法。  
// 定义一个可运行的类  
public class MyRunnable implements Runnable {  
    public void run(){  
    }  
}  
// 创建线程对象  
Thread t = new Thread(new MyRunnable());  
// 启动线程  
t.start();
```

注意：第二种方式实现接口比较常用，因为一个类实现了接口，它还可以去继承其它的类，更灵活。

Concurrency(并发)

多线程也可以实现 Concurrency(并发) 操作，每个请求分配一个线程来处理。

以第一种实现线程的方式为例：编写一个类，直接继承java.lang.Thread，重写run方法。

```
// 第一种方式：编写一个类，直接继承java.lang.Thread，重写run方法。  
// 定义线程类  
class MyThread extends Thread{  
    @Override  
    public void run(){  
        for (int i = 0; i < 10; i++) {  
            System.out.println("分支线程---> " + i);  
        }  
    }  
}
```

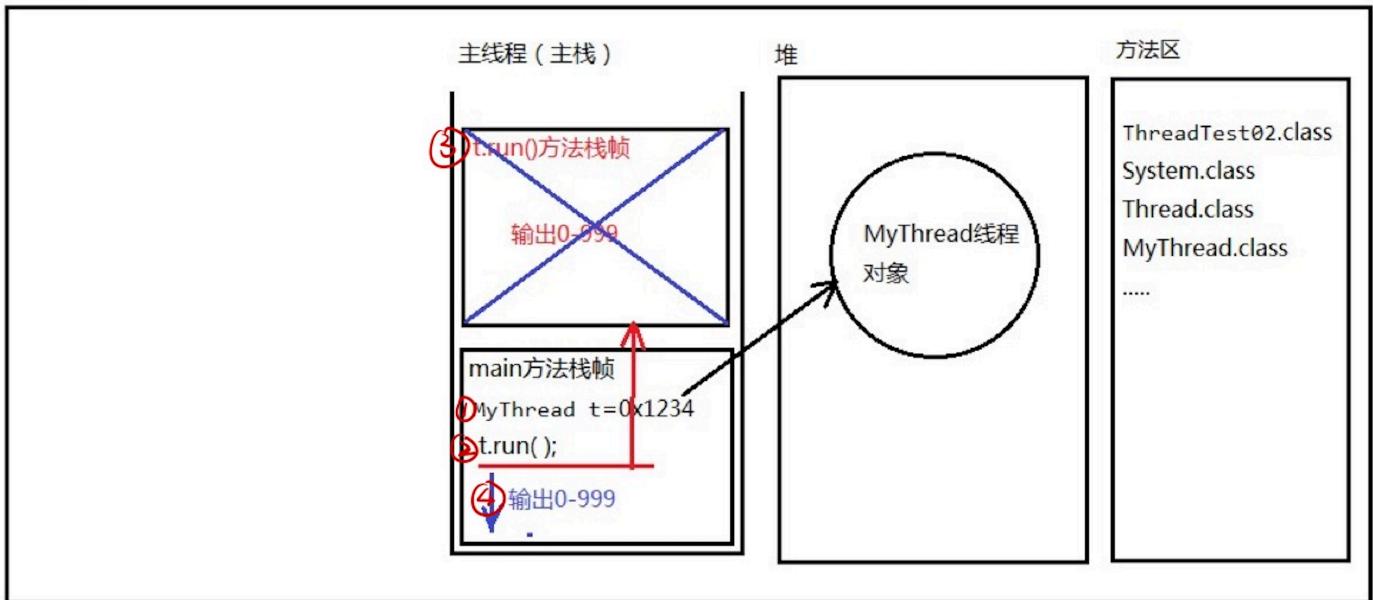
使用普通的调用：程序从上往下的同步执行，即如果第一行代码执行没有结束，第二行代码就只能等待第一行执行结束后才能结束。

```

public class ThreadTest {
    // 这里是main方法，这里的代码属于主线程，在主栈中运行。
    // 新建一个分支线程对象
    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        myThread.run(); // 普通的调用
        for (int i = 0; i < 1000; i++) {
            System.out.println("主线程---> " + i);
        }
    }
}

```

JVM



```

// Output:
分支线程---> 0 - 999
主线程---> 0 - 999

```

使用多线程API的方法 start()

- start()方法作用：启动一个分支线程，在JVM中开辟一个新的栈空间，这段代码完成之后，瞬间就结束了。
- 启动成功的线程会自动调用run方法，并且run方法在分支栈的栈底部（压栈）。
- run 方法在分支栈的栈底部，main方法在主栈的栈底部。run 和 main是平级的。
- 并发就是多个任务可以同时做，常用与任务之间比较独立，互不影响。

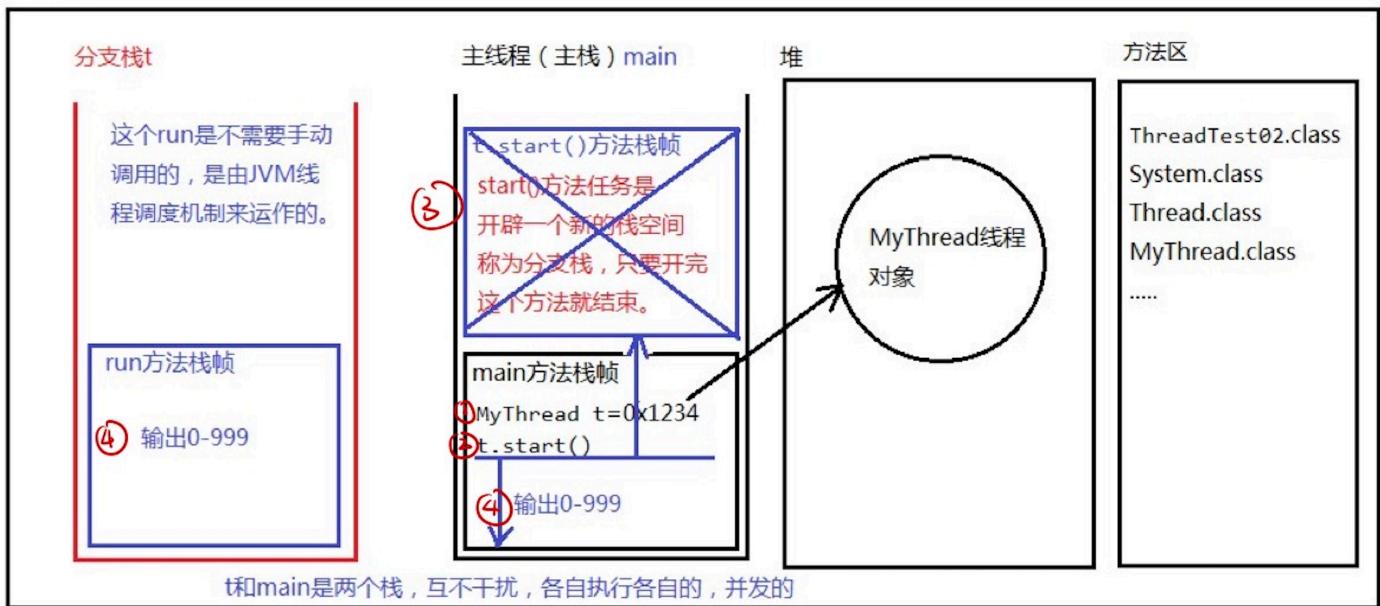
```

public class ThreadTest {
    // 这里是main方法，这里的代码属于主线程，在主栈中运行。
    //新建一个分支线程对象
    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        myThread.start(); // 启动线程，分配一个新的栈空间

        for (int i = 0; i < 1000; i++) {
            System.out.println("主线程---> " + i);
        }
    }
}

```

JVM



第三步是瞬间开始瞬间结束的

```

// Output: 结果是并发，主线程 和 分支线程 同时进行
分支线程---> 0
主线程---> 0
主线程---> 1
分支线程---> 1
主线程---> 2
分支线程---> 2
分支线程---> 3
分支线程---> 4
分支线程---> 5
主线程---> 3
...

```

以第二种实现线程的并发方式为例：编写一个类，实现java.lang.Runnable接口，实现run方法。

```

// 第二种方式：编写一个类，实现java.lang.Runnable接口，实现run方法。
// 定义一个可运行的类
class MyRunnable implements Runnable {
    public void run(){
        for (int i = 0; i < 1000; i++) {
            System.out.println("分支线程---> " + i);
        }
    }
}

```

```

public class ThreadTest {
    // 这里是main方法，这里的代码属于主线程，在主栈中运行。
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread thread = new Thread(myRunnable);
        // 也可以写成 Thread thread = new Thread(new MyRunnable());
        thread.start();
        for (int i = 0; i < 1000; i++) {
            System.out.println("主线程---> " + i);
        }
    }
}

```

```

// Output: 结果是并发，主线程 和 分支线程 同时进行
分支线程---> 0
主线程---> 0
主线程---> 1
分支线程---> 1
主线程---> 2
分支线程---> 2
分支线程---> 3
分支线程---> 4
分支线程---> 5
主线程---> 3
.....

```

void join() 合并线程，线程的调度(Coordination of Threads)

合并到当前线程中，当前线程进入阻塞，thread线程执行，直到thread线程结束。当前线程才可以继续。

```

public class ThreadTest {
    // 这里是main方法，这里的代码属于主线程，在主栈中运行。
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread thread = new Thread(myRunnable);
        // 也可以写成 Thread thread = new Thread(new MyRunnable());

```

```

        thread.start();

        try {
            thread.join(); // 两个栈之间发生等待关系，看上去好像是两个thread变成一个thread
            // 实际上是两个栈之间发生等待关系
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }

        for (int i = 0; i < 1000; i++) {
            System.out.println("主线程---> " + i);
        }
    }
}

```

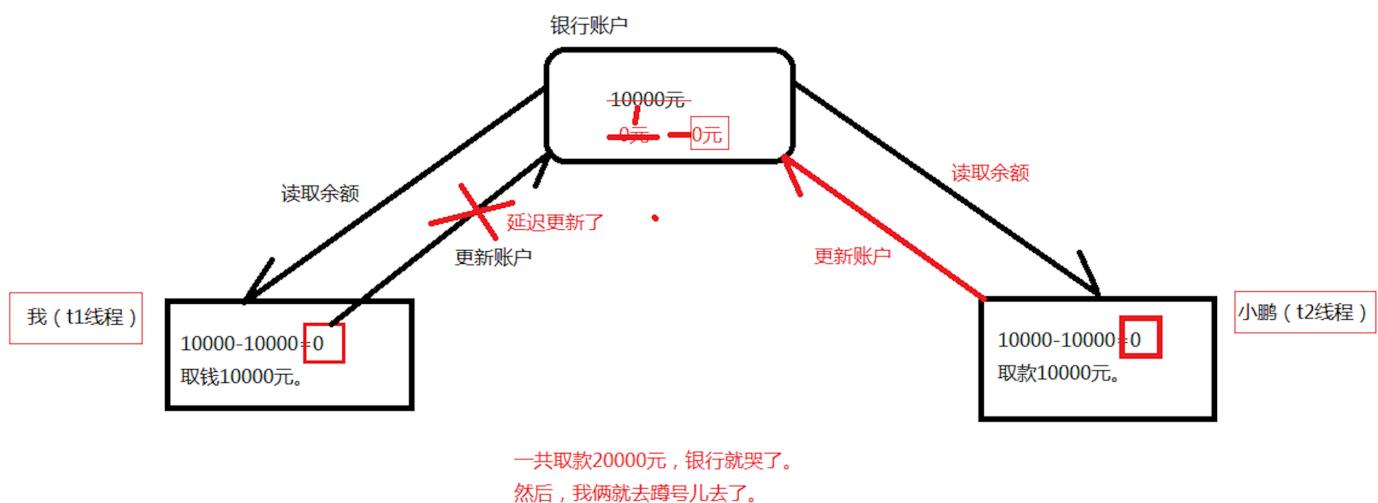
```

// Output:
分支线程---> 0 - 999
主线程---> 0 - 999

```

线程的安全问题

当多线程并发的环境下，有共享数据，并且这个数据还会被修改，此时就存在线程安全问题



线程的调度(Coordination of Threads)

线程在访问共享内存时需要 协调(coordinate)，以避免出现 争用情况(race conditions)

协调应保证访问共享资源时 互斥(mutual exclusion)

访问共享资源的一段代码称为 临界区(critical region)

```

// Shared memory
Hero h = 10; // h.health should never be negative

// critical region for the shared resource in thread 1
if(h.health > 0){ // thread 1
    h.health--;
}

// critical region for the shared resource in thread 2
h.health = 0; // thread 2

```

编写程序模拟两个线程同时对同一个账户进行取款操作

```

private String account;
private double balance;

public AccountTest(String account, double balance) {
    this.account = account;
    this.balance = balance;
}

public String getAccount() { return account; }
public void setAccount(String account) { this.account = account; }
public double getBalance() { return balance; }
public void setBalance(double balance) { this.balance = balance; }

public void withdraw (double money){
    double before = this.getBalance();
    double after = before - money;
    this.setBalance(after); // t1执行到这里了，但还没有来得及执行这行代码，t2线程进来
withdraw方法了。此时一定出问题。
}

public static void main(String[] args) {
    AccountTest a = new AccountTest("CHEN Ziyang" , 100);
    Thread t1 = new AccountThread(a);
    Thread t2 = new AccountThread(a);
    t1.setName("t1");
    t2.setName("t2");
    t1.start();
    t2.start();
}
}

```

```
class AccountThread extends Thread {  
    private AccountTest a;  
  
    public AccountThread(AccountTest a) {  
        this.a = a;  
    }  
  
    public void run(){ // 取款  
        double money = 100;  
        a.withdraw(money);  
        System.out.println(a.getAccount() + " 取款 100 成功, 还有 " + a.getBalance());  
    }  
}
```

结果会出现三种情况：

```
CHEN Ziyang 取款 50 成功, 还有 0.0  
CHEN Ziyang 取款 50 成功, 还有 50.0
```

```
CHEN Ziyang 取款 50 成功, 还有 50.0  
CHEN Ziyang 取款 50 成功, 还有 0.0
```

```
CHEN Ziyang 取款 50 成功, 还有 50.0  
CHEN Ziyang 取款 50 成功, 还有 50.0 // 出问题
```

我们模拟一下网络延迟

```
public void withdraw (double money){  
    double before = this.getBalance();  
    double after = before - money;  
  
    try {  
        Thread.sleep(1000); // 加入网络延迟  
    } catch (InterruptedException e) {  
        throw new RuntimeException(e);  
    }  
  
    this.setBalance(after);  
}
```

```
public static void main(String[] args) {
    AccountTest a = new AccountTest("CHEN Ziyang" , 100);
    Thread t1 = new AccountThread(a);
    Thread t2 = new AccountThread(a);
    t1.setName("t1");
    t2.setName("t2");
    t1.start();
    t2.start();
}
```

结果将一定是：

```
CHEN Ziyang 取款 50 成功, 还有 50.0
CHEN Ziyang 取款 50 成功, 还有 50.0 // 出问题
```

所以为了解决当多线程并发的环境下，共享数据会被修改，存在的线程安全问题

我们引入synchronized线程同步机制，使得多线程排队进行

```
synchronized (填需要排队的共享对象){
    // code
}
```

synchronized()中写什么？

- 要看你想让哪些线程同步。
- 假设有5个线程：t1、t2、t3、t4、t5
- 你只希望t1、t2、t3排队，t4、t5不需要排队。怎么办？
- 你要在()中写一个t1、t2、t3共享的对象。而这个对象对于t4、t5来说不是共享的。

```
public void withdraw (double money){
    // 同步代码块:
    synchronized (this){ // 填需要排队的共享对象，一般来说会有很多个，随便填一个就行
        double before = this.getBalance();
        double after = before - money;

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }

        this.setBalance(after);
    }
}
```

同时我们也可以将代码写成：

```
public synchronized void withdraw (double money){  
    // 就好像方法体是一个 synchronized(this) block  
    double before = this.getBalance();  
    double after = before - money;  
  
    try {  
        Thread.sleep(1000);  
    } catch (InterruptedException e) {  
        throw new RuntimeException(e);  
    }  
  
    this.setBalance(after);  
}
```

```
public static void main(String[] args) {  
    AccountTest a = new AccountTest("CHEN Ziyang" , 100);  
    Thread t1 = new AccountThread(a);  
    Thread t2 = new AccountThread(a);  
    t1.setName("t1");  
    t2.setName("t2");  
    t1.start();  
    t2.start();  
}
```

结果：

```
CHEN Ziyang 取款 50 成功, 还有 50.0  
CHEN Ziyang 取款 50 成功, 还有 0.0
```

Mutex (mutual exclusion object)

Mutex 是一个二进制变量，一次最多只能被一个线程锁定

- lock(): locks the mutex (or, obtains the lock) if it is not locked yet; otherwise suspends the execution
- unlock() : unlocks the mutex (or, releases the lock) so that other lock() operations may succeed

```
mutexForResourceA.lock();  
// critical region for resource A in thread 1  
mutexForResourceA.unlock();  
  
mutexForResourceA.lock();  
// critical region for resource A in thread 2  
mutexForResourceA.unlock();
```

lock 和 unlock操作保证 不中断(non- interrupted)

一个线程在某个时间点上可能有多个锁

Monitor = Mutex + Condition variable

Condition variable: 由 相应的Mutex 保护和调度 threads

A condition variable is 基本上是一个等待特定条件的 **container of threads**

- 一个 thread 可能无法继续(proceed), 因为它需要其他 thread 的工作。然后它可以等待并将控制权交给其他线程(yield control to other threads)
- 当一个线程执行一个其他线程可能正在等待的动作时, 它可以给它发信号并唤醒它们(中断它们的等待)。

在java语言中, 任何一个对象都有“一把锁”, 其实这把锁就是标记。(只是把它叫做锁) 相当于厕所门

100个对象, 100把锁。1个对象1把锁。

1. 假设 t1和 t2线程并发, 开始执行以下代码的时候, 肯定有一个先一个后。
2. 假设t1先放行了, 到了synchronized, 这个时候自动找“后面共享对象”的对象锁找到之后, 并占有这把锁, 然后执行同步代码块中的程序, 在程序执行过程中一直部是占有这把锁的。直到同步代码块代码结束, 这把锁才会释放。
3. 假设t1已经占有这把锁, 此时t2也遇到synchronized关键字, 也会去占有后面共享对象的这把锁, 结果这把锁被t1占有, t2只能在同步代码块外画等待t1的结束, 直到1把同步代码块执行结束了, t1会归还这把锁, 此时t2终于等到这把锁, 然后t2占有这把锁之后, 进入同步代码块执行程序。

```
public void withdraw (double money){  
    // Object o = new Object();  
    // synchronized (o){      Error! 因为 'o' 不是共享对象而是 local variable  
  
    // synchronized (null){    Error! 空指针  
  
    // synchronized ("abc"){    Ok! 因为 "abc" 在字符串常量中, 所有线程都会同步 (重  
要!!!!)  
  
    // synchronized (this){    Ok! 因为 this 就是 这个的对象 AccountTest  
    // 会同步共享 对象AccountTest 的线程  
    // 相当于以下写法, 因为this代表自身对象(默认)  
    AccountTest object = this;  
    synchronized (object){  
        double before = this.getBalance();  
        double after = before - money;  
  
        try {  
            Thread.sleep(1000);  
        } catch (InterruptedException e) {  
            throw new RuntimeException(e);  
    }  
}
```

```
    }

    this.setBalance(after);
}

}
```

结果：

```
CHEN Ziyang 取款 50 成功, 还有 50.0
CHEN Ziyang 取款 50 成功, 还有 0.0
```

Java中有三大变量？【重要的内容。】

- 实例变量：在堆中。实例变量在堆中，堆只有1个。
- 静态变量：在方法区。静态变量在方法区中，方法区只有1个。
- 局部变量：在栈中。局部变量永远都不会存在线程安全问题。因为局部变量不共享。（一个线程一个栈。）局部变量在栈中。所以局部变量永远都不会共享。

堆和方法区都是多线程共享的，所以可能存在线程安全问题。

局部变量+常量：不会有线程安全问题。

成员变量：可能会有线程安全问题。

如果使用局部变量的话：建议使用：StringBuilder。因为局部变量不存在线程安全问题。选择StringBuilder。StringBuffer效率比较低。

ArrayList是非线程安全的。

Vector是线程安全的。

HashMap HashSet是非线程安全的。

Hashtable是线程安全的。

wait(), notify(), and notifyAll()

- wait和notify方法不是线程对象的方法，是java中任何一个java对象都有的方法
- 这两个方法是Object类中自带的。
- wait方法和notify方法不是通过线程对象调用

wait()表示让正在o对象上活动的线程进入等待状态，无期限等待，直到被唤醒为止。o.wait();方法的调用，会让“当前线程（正在o对象上活动的线程）”进入等待状态。

挂起并解锁所有对象，直到某个线程对 ' o ' 执行notify或notifyAll操作(suspend and unlock all objects until some thread does a notify or notifyAll on ' o ')

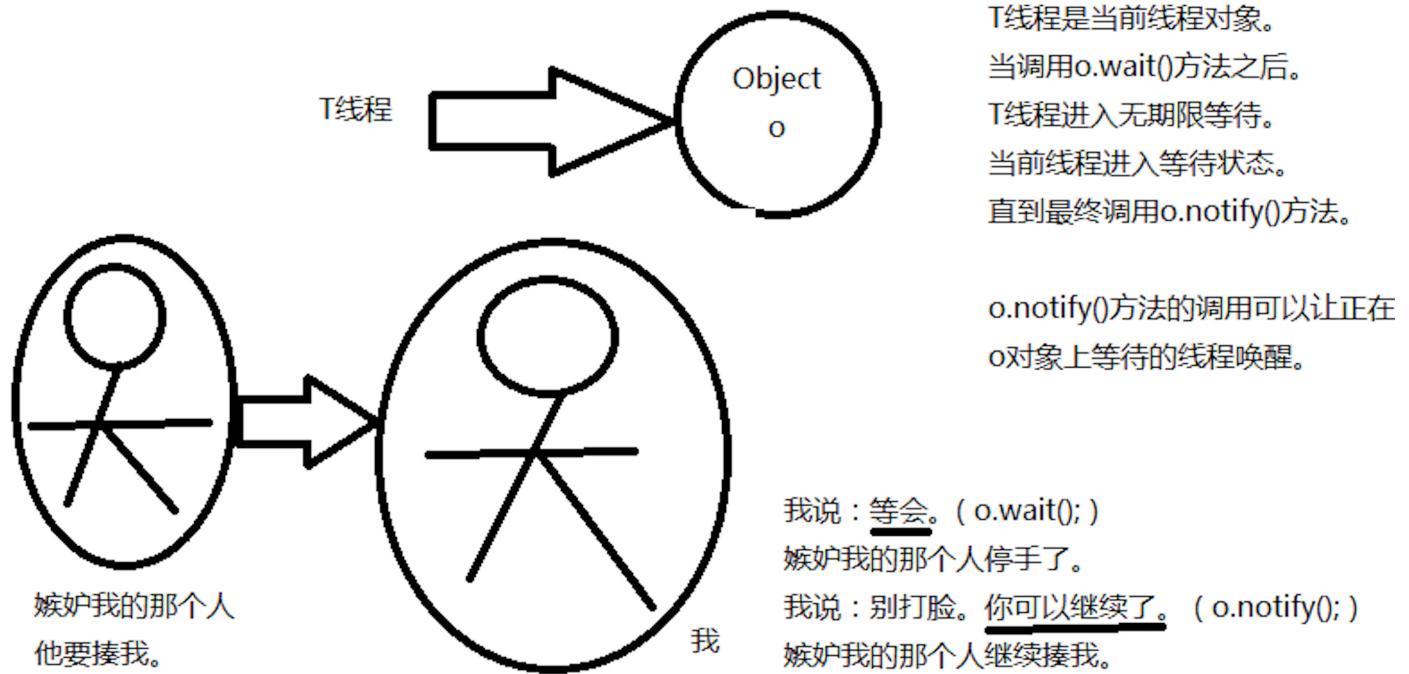
```
Object o = new Object();
o.wait(); // 表示让正在o对象上活动的线程进入等待状态，无期限等待，直到被唤醒为止
```

notify()方法表示唤醒正在o对象上等待的线程。

```
Object o = new Object();
o.notify(); // 唤醒正在o对象上等待的线程
```

notifyAll() 方法是唤醒o对象上处于等待的所有线程。

```
Object o = new Object();
o.notifyAll(); // 唤醒o对象上处于等待的所有线程
```



The Producer-Consumer

生产者和消费者模式：

两种线程，生产者和消费者，在有限大小的共享缓冲区上同时工作

➢ 生产者(Producer) 将新消息放入缓冲区 (puts new messages in the buffer)

- 如果 缓冲区(buffer) 已满，Producer 必须等到 Consumer 获取/接收一些消息 (takes some messages from the buffer)
- 生产者也发出最后一条消息 (the Producer also signals the last message)

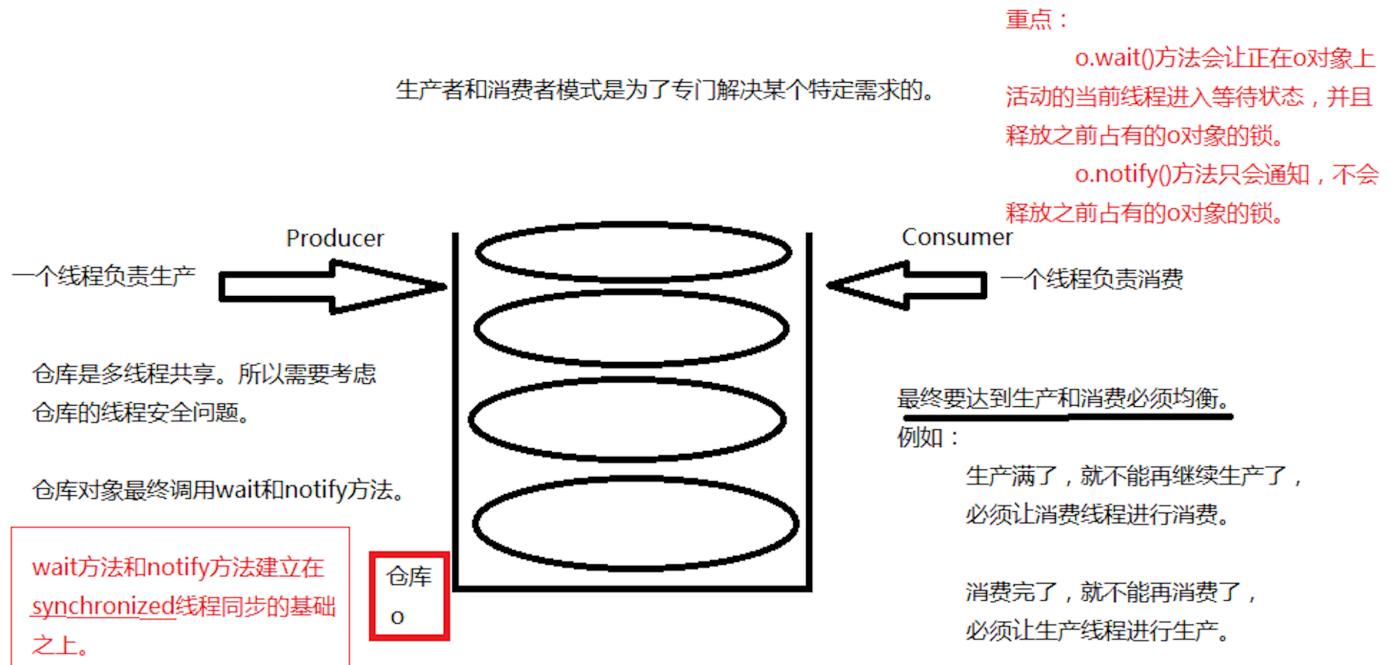
➢ 消费者(Consumer) 从缓冲区中获取消息 (takes messages from the buffer)

- 如果 缓冲区(buffer) 为空，Consumer 必须等到 Producer 发布了一些新消息 (puts new messages in the

buffer)

- 消费者在最后一条消息后终止 (the Consumer terminates after the last message)

buffer 可以理解成一个仓库：



- 使用 wait方法 和 notify方法 实现“生产者和消费者模式”

- 什么是“生产者和消费者模式”？

生产线程负责生产，消费线程负责消费。

生产线程和消费线程要达到均衡。

这是一种特殊的业务需求，在这种特殊的情况下需要使用wait方法和notify方法。

- wait和notify方法不是线程对象的方法，是普通java对象都有的方法。
- wait方法和notify方法建立在线程同步的基础之上。因为多线程要同时操作一个 buffer。有线程安全问题。
- wait方法作用：o.wait()让正在o对象上活动的 thread 进入等待状态，并且释放 thread 之前占有的o对象的锁
- notify方法作用：o.notify()让正在o对象上等待的 thread 被唤醒，只是通知，不会释放o对象上之前占有的锁

Example:

模拟这样一个需求：buffer 我们用 List 集合。List 集合中假设只能存储1个元素。1个元素就表示仓库满了。

如果List集合中元素个数是 0，就表示仓库空了。保证List集合中永远都是最多存储1个元素。

必须做到这种效果：生产1个消费1个。

```
class Producer implements Runnable { // 生产线程
    private List storage; // 仓库

    public Producer (List storage) {
        this.storage = storage;
    }

    @Override
    public void run() { // 一直生产
        while (true){ // 一直生产 (使用死循环来模拟一直生产)
    }
```

```

        synchronized (storage) { // 给仓库对象 storage 加锁
            if (storage.size() > 0) { // 大于0，说明仓库中已经有1个元素了。
                try {
                    storage.wait(); // 当前 Producer 线程进入等待状态，并且释放producer之前占有的list集合的锁。
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
            //程序能够执行到这里说明仓库是空的，可以生产
            java.lang.Object obj = new Object();
            storage.add(obj);
            System.out.println(Thread.currentThread().getName() + " ---> " + obj);
            // 唤醒 Consumer 进行消费
            storage.notify();
        }

    }
}
}

```

```

class Consumer implements Runnable { // 消费线程
    private List storage; // 仓库

    public Consumer (List storage) {
        this.storage = storage;
    }
    @Override
    public void run() { // 一直消费
        while (true){ // 一直生产（使用死循环来模拟一直生产）

            synchronized (storage) { // 给仓库对象 storage 加锁
                if (storage.size() == 0) { // 仓库是空的
                    try {
                        storage.wait(); // 当前 Consumer 线程进入等待状态，释放掉List集合的锁
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
                // 程序能够执行到此处说明仓库中有数据，进行消费。
                java.lang.Object obj = storage.remove(0);
                System.out.println(Thread.currentThread().getName() + " ---> " + obj);
                // 唤醒 Producer 进行生产
                storage.notify();
            }

        }
    }
}

```

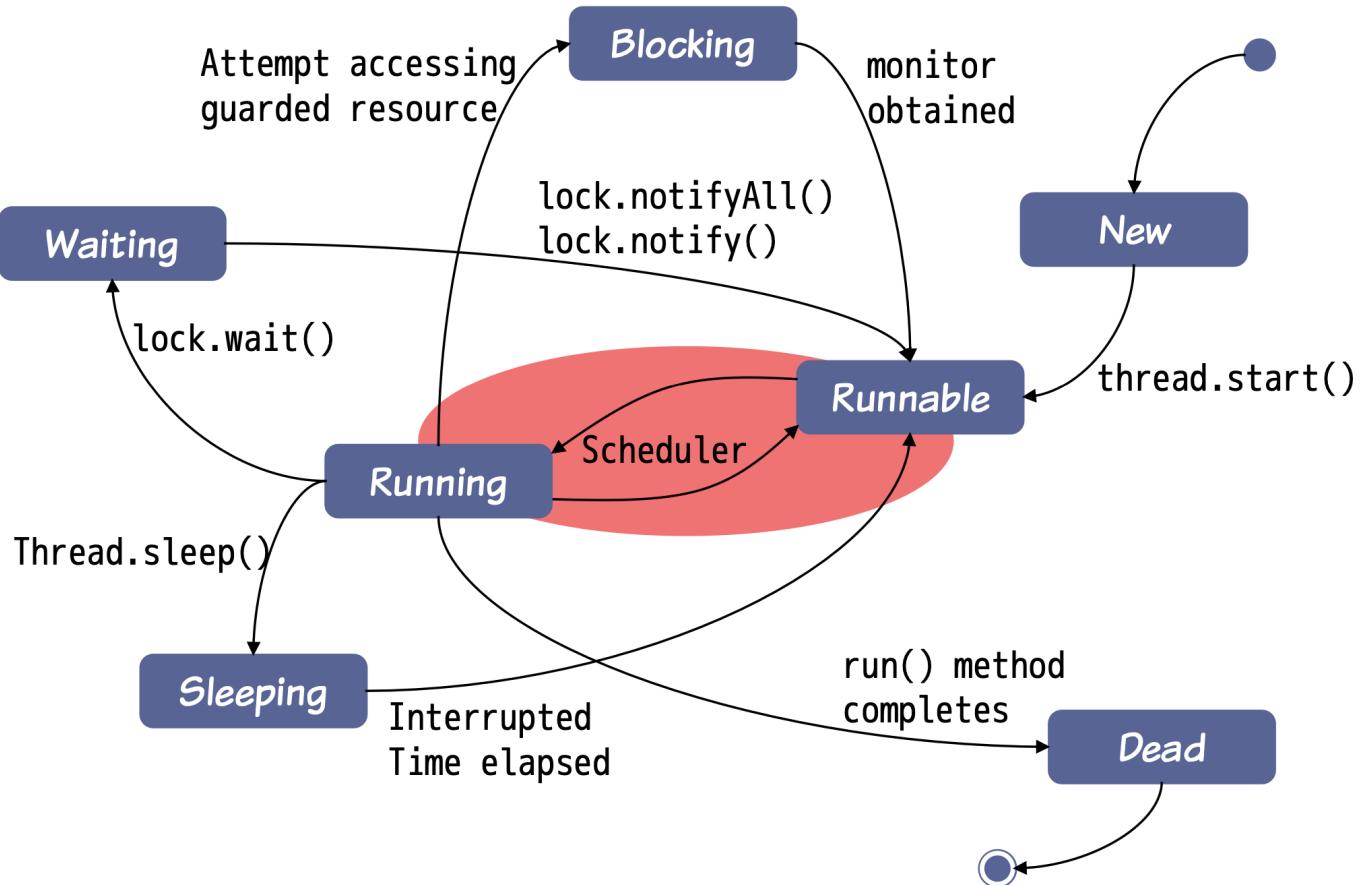
```
import java.util.ArrayList;
import java.util.List;

public class Buffer {
    public static void main(String[] args) {
        List storage = new ArrayList();
        Thread t1 = new Thread(new Producer(storage)); // 生产者线程
        Thread t2 = new Thread(new Consumer(storage)); // 消费者线程
        t1.setName("生产者线程");
        t2.setName("消费者线程");
        t1.start();
        t2.start();
    }
}
```

Output:

```
生产者线程 ---> Draft.Object@5b1dd27c
消费者线程 ---> Draft.Object@5b1dd27c
生产者线程 ---> Draft.Object@2ba256f8
消费者线程 ---> Draft.Object@2ba256f8
生产者线程 ---> Draft.Object@7f2348f2
消费者线程 ---> Draft.Object@7f2348f2
生产者线程 ---> Draft.Object@612ec01c
消费者线程 ---> Draft.Object@612ec01c
生产者线程 ---> Draft.Object@54157077
消费者线程 ---> Draft.Object@54157077
生产者线程 ---> Draft.Object@6db9e99b
消费者线程 ---> Draft.Object@6db9e99b
生产者线程 ---> Draft.Object@8385806
消费者线程 ---> Draft.Object@8385806
....
```

Thread States



守护线程 setDaemon(true) // 不考

java语言中线程分为两大类：

一类是：用户线程

一类是：守护线程（后台线程）

其中具有代表性的就是：垃圾回收线程（守护线程）。

一般守护线程是一个死循环，所有的用户线程只要结束，守护线程自动结束。

注意：主线程main方法是一个用户线程。

守护线程用在什么地方呢？

每天00:00的时候系统数据自动备份。

这个需要使用到定时器，并且我们可以将定时器设置为守护线程。

一直在那里看着，每到00:00的时候就备份一次。所有的用户线程

如果结束了，守护线程自动退出，没有必要进行数据备份了。

```

public class ThreadTest {
    public static void main(String[] args) {
        Thread t1 = new BackDataThread();
        t1.setName("备份数据的线程");
        t1.start();

        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getName() + " ---> " + i);
            try {
                Thread.sleep(1000);
            }
        }
    }
}

```

```
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

```
class BackDataThread extends Thread {
    @Override
    public void run() {
        int i = 0;
        while (true) {
            System.out.println(Thread.currentThread().getName() + " ---> " + (++i));
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

结果：

```
备份数据的线程 ---> 1
main ---> 0
备份数据的线程 ---> 2
main ---> 1
备份数据的线程 ---> 3
main ---> 2
备份数据的线程 ---> 4
main ---> 3
备份数据的线程 ---> 5
main ---> 4
备份数据的线程 ---> 6
备份数据的线程 ---> 7
备份数据的线程 ---> 8
备份数据的线程 ---> 9
备份数据的线程 ---> 10
备份数据的线程 ---> 11
备份数据的线程 ---> 12
...
...
```

使用 守护线程 setDaemon(true)

```
public class ThreadTest {
```

```

public static void main(String[] args) {
    Thread t1 = new BackDataThread();
    t1.setName("备份数据的线程");

    // 启动线程之前，将线程设置为守护线程
    t1.setDaemon(true);

    t1.start();

    for (int i = 0; i < 5; i++){
        System.out.println(Thread.currentThread().getName() + " ---> " + i);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}

```

```

class BackDataThread extends Thread {
    @Override
    public void run() {
        int i = 0;
        // 即使是死循环，但由于该线程是守护者，当用户线程结束，守护线程自动终止。
        while (true) {
            System.out.println(Thread.currentThread().getName() + " ---> " + (++i));
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}

```

结果：

```

main ---> 0
备份数据的线程 ---> 1
备份数据的线程 ---> 2
main ---> 1
备份数据的线程 ---> 3
main ---> 2
备份数据的线程 ---> 4
main ---> 3
main ---> 4
备份数据的线程 ---> 5

```

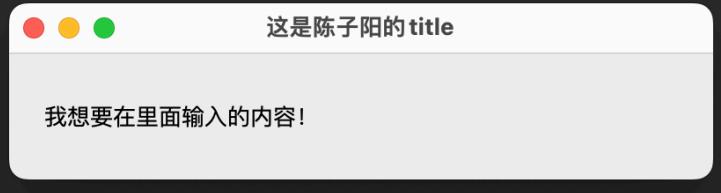
09. Swing // 不考

Swing API 包含 18 个公共包，用于构建图形用户界面 (GUI) 并为 Java 应用程序添加丰富的图形功能和交互性。

AWT vs. Swing vs. JavaFX

- AWT (Abstract Windowing Toolkit) 适用于开发简单的图形用户界面，并且可能存在特定于平台的错误。
- Swing 主要用于桌面应用程序，对 AWT 进行了扩展
- JavaFX 是一个软件平台，用于创建和交付桌面应用程序，以及可以在各种设备上运行的富互联网应用程序。
- JavaFX 平台旨在取代 AWT 和 Swing。但是 Swing API 使解释某些 OO 相关概念变得更容易

```
import javax.swing.*;
public class HelloWorldSwing {
    1 usage
    private static void createAndShowGUI() {
        //Create and set up the window.
        JFrame frame = new JFrame("这是陈子阳的title");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //Add the ubiquitous "Hello World" label.
        JLabel label = new JLabel("我想要在里面输入的内容!");
        frame.getContentPane().add(label);
        //Display the window.
        frame.setSize(400, 100);
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        // Schedule a job for the event-dispatching thread:
        // creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(()->{
            createAndShowGUI();
        });
    }
}
```



JFrame

JFrame 是一个容器，是各个组件的载体

JFrame	框架
JDialog	对话框
JOptionPane	对话框
JButton	按钮
JCheckBox	复选框
JComboBox	下拉框
JLabel	标签
JRadioButton	单选按钮
JList	显示一组条目的组件
JTextField	文本框 (一行)
JPasswordField	密码框
JTextArea	文本区域 (多行多列)

二.设置JFrame的大小

```

public void setSize(int width,int height)设置窗口的大小。
public void setLocation(int x,int y)设置窗口的位置，默认位置是(0,0)。
public void setBounds(int a,int b,int width,int height)设置窗口的初始位置是(a,b)，窗口的宽是width，高是height。

public void setVisible(boolean b)设置窗口是否可见，窗口默认是不可见的。

public void setResizable(boolean b)设置窗口是否可调整大小，默认可调整大小。

public void dispose()撤销当前窗口，并释放当前窗口所使用的资源。

public void setExtendedState(int state)设置窗口的扩展状态，其中参数state取JFrame类中的下列类常量：
MAXIMIZED_HORIZ (水平方向最大化),
MAXIMIZED_VERT (垂直方向最大化),
MAXIMIZED_BOTH (水平、垂直方向都最大化)。

```

三.设定JFrame的关闭方式

```

public void setDefaultCloseOperation(int operation)该方法用来设置单击窗体右上角的关闭图标后，程序会做出怎样的处理，其中的参数operation取JFrame类中的下列int型static常量，程序根据参数operation取值做出不同的处理：

DO NOTHING ON CLOSE (什么也不做),
HIDE ON CLOSE (隐藏当前窗口),
DISPOSE ON CLOSE (隐藏当前窗口，并释放窗体占有的其他资源),
EXIT ON CLOSE (结束窗口所在的应用程序)。

```

JDialog

继承自java.awt.Dialog类。他是从一个窗体弹出来的另外一个窗体。他和Frame类似，需要调用getContentPane将窗体转换为容器。然后在容器中设置窗体的内容。

JDialog: 可以当成JFrame使用，但必须从属于JFrame

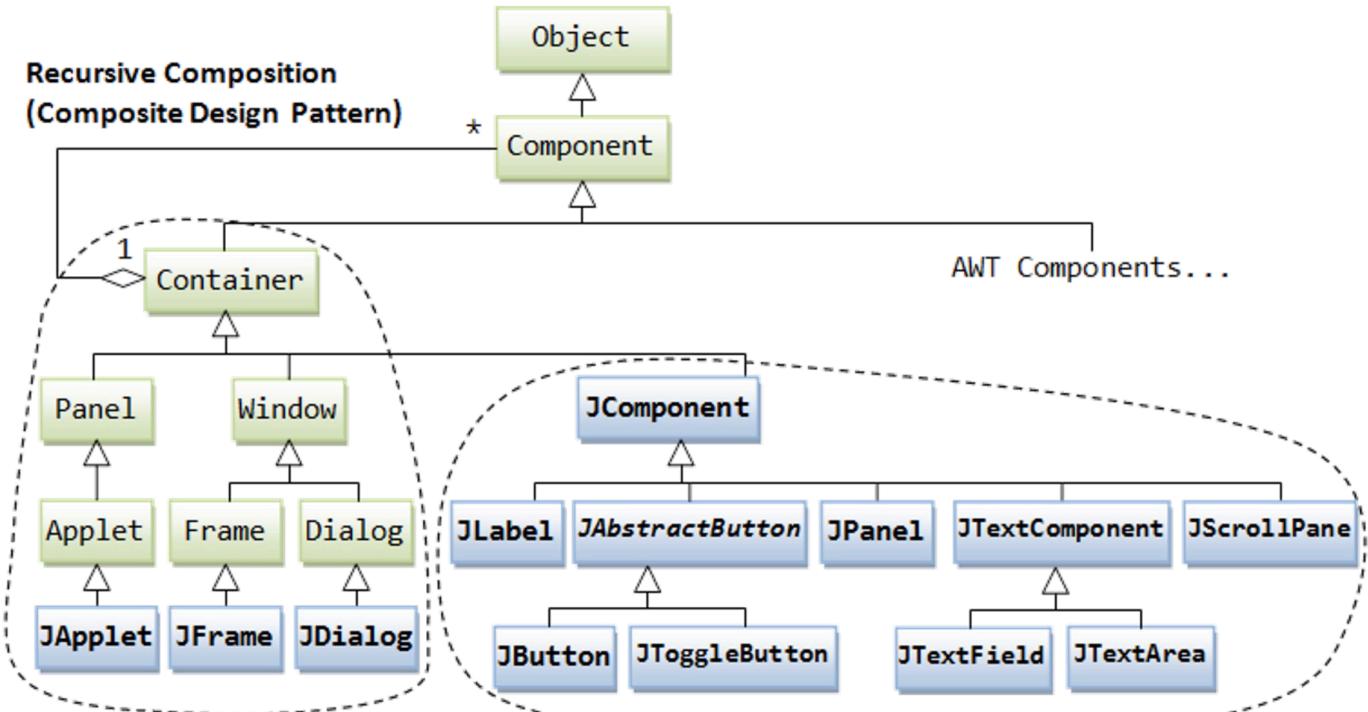
构造函数

```
JDialog();  
JDialog(Frame f); // 指定父窗口  
JDialog(Frame f, String title); // 指定父窗口 + 标题
```

Swing's Components

Types of containers:

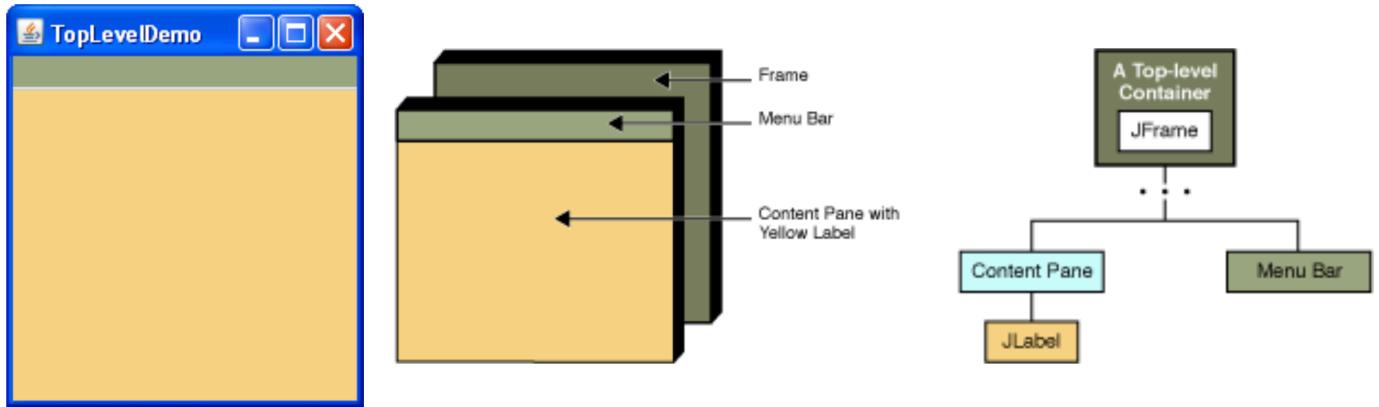
- **Top-level containers:** 每个Swing program 至少包含一个 **Top-level containers (JFrame, JDialog, and JApplet)**
- **Intermediate containers:** 用于对组件进行分组，以便将它们作为单个组件处理 (e.g. JPanel, JTabbedPane).
- **Atomic components (basic controls):** 不能包含其他组件 (e.g. JButton, JTextField)



Using Top-Level Containers

顶级容器 (JFrame、JDialog 和 JApplet) 不能添加到其他容器中

- 每个顶级容器都有一个**content pane**(内容窗格)
- 要出现在屏幕上，每个 GUI 组件必须（直接或间接）包含在顶级容器的 **content pane**(内容窗格) 中
- 每个 GUI 组件只能被顶级容器包含一次



Adding Components to the Content Pane

The **getContentPane** method

```
JLabel label = new JLabel("Hello World");
frame.getContentPane().add(label);
```

The **setContentPane** method

```
JPanel contentPane = new JPanel(...);
contentPane.add(someComponent, ...);
contentPane.add(anotherComponent, ...);
topLevelContainer.setContentPane(contentPane);
```

Layout Managers and Panel

Swing container 使用所谓的(so-called) 布局管理器(layout manager) 去排列组件

Layout managers(布局管理器) 提供了一个抽象级别来映射所有窗口系统上的用户界面 (UI)，因此 UI 可以独立于平台。

Layout managers provide a level of abstraction to map a user interface (UI) on all windowing systems, so that the UI can be platform-independent.

```
// JPanel is a secondary container
```

```
JPanel panel = new JPanel(new FlowLayout());  
panel.add(new JLabel("One"));  
panel.add(new JLabel("Two"));  
panel.add(new JLabel("Three"));
```

It's also possible to set the layout manager afterwards:

```
JPanel panel = new JPanel();  
panel.setLayout(new FlowLayout());
```



1. **FlowLayout**, **GridLayout**, and **BorderLayout** managers are obsolete. Use **MigLayout**, **GroupLayout**, or **FormLayout** to build modern GUIs.
2. We use some of the obsolete managers in our examples for their simplicity.

10. Nested Class, Local Class, Lambda Expression

Nested Classes

- 一个类可以定义为另一个类的成员
- 嵌套类(nested class)可以是 static 的 (called **static nested class**) 或 non-static (called an **inner class**)
- 就像其他类成员一样, 嵌套类可以声明为 private, public, protected, or package private

```
class A { // => A.class  
...  
}  
class B { // => B.class  
...  
}
```

```
class A { // an outer class. => A.class  
...  
class B { // a nested class. => A$B.class  
...  
}
```

- **static nested class**
- **Inner Classes (Non-Static Nested Classes)**

```
public class NestedClass {  
    public static void main(String[] args){  
        OuterClass outer = new OuterClass();  
        OuterClass.InnerClass inner = outer.new InnerClass();  
        inner.increaseValue();  
        System.out.println(inner.getValue()); // print: 1  
    }  
}
```

// 如果我们想要创建 `StaticNested` 对象 `nested`, 我们要使用外围类名访问
// `StaticNested nested = new StaticNested(); // Error!`

```
OuterClass.StaticNested nested = new OuterClass.StaticNested();
nested.method(outer); // print: 10
}

}

class OuterClass{ // // outer class
    private int x;
    public int getX(){ return x; }
    public void setX(int x) { this.x = x; }

    public class InnerClass {
        public void increaseValue() { x++; }
        public int getValue() { return getX(); }
    }
}

static class StaticNested { // static nested class
    void method(OuterClass outer) {
        // setX(10); // Error, 不能直接引用
        outer.setX(10); // OK, 只能间接引用
        System.out.println(outer.getX());
    }
}
}
```

Local Class

Lambda Expression