Java Review

目录:

Java语言基础

- **Java 简介**
  - **Java的特点:**
    - 面向对象(OOP)

      > 支持面向对象编程语法
      >
      > 是使用广泛的面向对象语言之一

- 跨平台(WORA)

  > 使用Java虚拟机(Java Virtual Machine, or JVM) 和 Java bytecode 实现

- 类C语法

  > 去除了指针
  >
  > 自动内存管理和垃圾回收(GC)

Java编写：

- 流程

  > 编写源代码
  >
  > 编译：javac
  >
  > 运行：java

- 可能出现的错误

  > 编译时错误 Compile-time error
  >
  > 运行时错误 Run-time error
  >
  > 逻辑错误 Logic error

Java类型

基本类型

Primitive type	Size (bits)	Minimum	Maximum	Wrapper type	Default
boolean	-	-	-	Boolean	false
char	16	Unicode 0	Unicode $2^{16}-1$	Character	'\u0000'(null)
byte	8	-128	$+127$	Byte	(byte)0
short	16	-2^{15}	$+2^{15}-1$	Short	(short)0
int	32	-2^{31}	$+2^{31}-1$	Integer	0
long	64	-2^{63}	$+2^{63}-1$	Long	0L
float	32	IEEE 754	IEEE 754	Float	0.0f
double	64	IEEE 754	IEEE 754	Double	0.0d
void	-	-	-	Void	-

> 注：
>
> 1. boolean只有`true`和`false`两种取值，而对于boolean类型的大小，并没有给出精确的定义，对于不同的虚拟机可能会出现8 bits或者32 bits
> 2. Java种float和double均按照IEEE 754标准存储，其上下限与IEEE 754标准完全一致，即：

类型	有效数字	最小正正规数	最大正数
float	**约**7位	约1.175×10^{-38}	约$3.402 \times 10^{+38}$
double	**约**16位	约2.225×10^{-308}	约$1.797 \times 10^{+308}$

需要注意的是，有效数字并不是单指小数点后，而是整个数的有效数字。浮点数的精度是有限的。[IEEE754标准: 三, 为什么说32位浮点数的精度是"7位有效数" - 知乎 (zhihu.com)]

3. **Java使用Unicode字符集的UTF-16格式，而不是ASCII字符集。**所以Java的char并不是8 bits，而是16 bits
4. 由于Java使用JVM运行，所以（除boolean以外），在不同的操作系统中，基本类型不会有区别。（事实上boolean除了可能size有变化以外，也不会有任何区别）
5. Java没有unsigned
6. 注意不可以用数字直接代替boolean,例如

   ~~`if(1),if(0)`~~

   是错误的，应该为

   `if(true),if(false)`

- **可变不可变类型**
  - **不可变类型(Immutable)**
    - 类型的对象一旦创建就不能被改变
    - 如`String,Integer,Float`等
  - **可变类型(Mutable)**
    - 对象可以被操作修改
  - **不可变类型的优点**
    - 简单、易用、安全
  - **不可变类型的缓存池**

    > - boolean values true and false
    > - all byte values
    > - short values between -128 and 127
    > - int values between -128 and 127
    > - char in the range \u0000 to \u007F
    > - existed String

    在使用这些基本类型对应的**封装类**时，如果该数值范围在缓存池范围内，就可直接引用缓存池的对象，否则创建一个新的对象。

    可以使用缓存池对不可变类型对象的构造进行加速的原因是**不可变类型本质上是不可变的**，所以即使引用已经存在的对象也不会造成其他影响。

    例：

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

```
        String a = new String("A String");
        String b = new String("A String");

        String c = "Another String";
        String d = "Another String";

        Integer e = 114;
        Integer f = 114;

        Integer g = 514;
        Integer h = 514;

        System.out.println(a==b);           //false
        System.out.println(a.equals(b));    //true

        System.out.println(c==d);           //true
        System.out.println(c.equals(d));    //true

        System.out.println(e==f);           //true
        System.out.println(e.equals(f));    //true

        System.out.println(g==h);           //false
        System.out.println(g.equals(h));    //true
    }
}
```

注意：使用new来生成对象时一定会新建新对象，而不会采用缓冲池。

- **数组**
  - **初始化：**
    - 静态初始化

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

    - 动态初始化

```
int []a = new int[5];
MyType []m = new MyType[3];

int []a = new int[] {1,2,3,4,5};
MyType []m = new MyType[] {
    new MyType(),
    new MyType(),
    new MyType()
};
```

    - 多维数组

```
int [][]a = new int[2][3];
```

- **数组的特性：**

  > **数组是对象**，是一种特殊的对象。
  >
  > 可以使用`arrayName.length`来获得数组的长度，可见`length`是它的一个数据成员。

- **类**

  - **定义：**

```
class Mytype{
    int i;
    double d;
    char c;                 //Fields

    void setD(double x);
    double getD();          //Methods
}
```

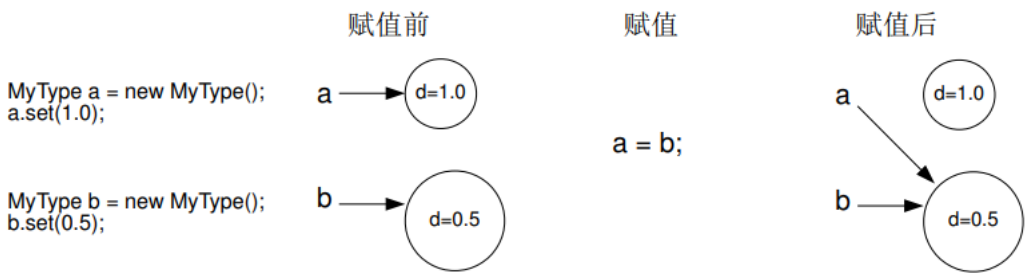
  - **构造对象，访问对象：**

```
public class Main{
    public static void main(String []args){
        MyType a = new MyType();
        int b = a.i;
        a.setD(10.0);
        double c = a.getD();
    }
}
```

- **Java 引用**

  - 引用(Reference)
    - 对象的名字
    - 同一个对象可以有不同的名字
    - 和对象的关系可以类比遥控器和电视机的关系

```
MyType m = new MyType();
MyType n = m;           //引用赋值
n.set(1.0);
System.out.println(m.d);
```



  - 引用和指针

> Java标准并没有指定引用应该如何实现
> 绝大多数Java内部使用指针实现引用

- 引用是受限的指针

> 不允许引用直接运算，不允许强制转换*
> （多态是特殊情况）
> 对象有引用，而**基本类型不是对象**，基本类型的封装是对象，所以对于基本类型来说不存在引用

- **强调：我们所用到的所谓“引用数据类型”，或者我们用来指代对象的变量，实际上都是*引用*而已，所以从这个层面上讲，Java只有值传递。在传递引用时，相当于复制了一个引用而已，而不是复制了一个对象（对于基本类型来讲，只是复制了一个值传递进去而已，所以无论是引用还是基本数据都，都是只传值。）**
- ## 创建对象

> - 使用构造函数构造对象
>
> ```
>     MyType m = new MyType();
> //   类型 变量名      构造函数
> ```
>
> - 构造函数可以被重载
> - 所有类如果没有自行重载，则会有一个无参数的默认构造函数。
> - ### 函数重载
>
>   - 函数名相同，参数类型/数量不同
>   - 优点：接口简洁，统一
>
> ```
> public class Printer{
>     void print(int x){
>         System.out.println("print an integer:" + x);
>     }
>     void print(MyType m){
>         System.out.println("print a MyType:" + m.get());
>     }
> }
> ```
>
>   - 说明Java中区分不同的函数不能看函数名，还要看参数列表（包括参数的类型和顺序）和返回类型。
>   - 例如`print()`和`println()`方法就使用重载来适应各种类型。

void	println() Terminates the current line by writing the line separator string.
void	println(boolean x) Prints a boolean value and then terminates the line.
void	println(char x) Prints a character and then terminates the line.
void	println(char[] x) Prints an array of characters and then terminates the line.
void	println(double x) Prints a double-precision floating-point number and then terminates the line.
void	println(float x) Prints a floating-point number and then terminates the line.
void	println(int x) Prints an integer and then terminates the line.
void	println(long x) Prints a long integer and then terminates the line.
void	println(Object x) Prints an Object and then terminates the line.
void	println(String x) Prints a String and then terminates the line.

- **重载规则:**
  - 被重载的方法必须改变参数列表(参数个数或类型不一样);
  - 被重载的方法可以改变返回类型;
  - 被重载的方法可以改变访问修饰符;
  - 被重载的方法可以声明新的或更广的检查异常;
  - 方法能够在同一个类中或者在一个子类中被重载。
  - 无法以返回值类型作为重载函数的区分标准。

cr. Java 重写(Override)与重载(Overload) | 菜鸟教程 (runoob.com)

Java操作与控制

- 操作符、逻辑操作和表达式类C
- 表达式的值为boolean，和C不同（也即不能使用0,1指代false, true）
- 相等判断

> C 中使用`==`直接判断值是否相等，而对Java而言，对基本类型来说是比较它们的值，对对象而言是比较它们的引用（引用的地址）
>
> 所以Java提供了`equals()`方法，默认情况下与直接使用==相同，但是可以自行重写，并且Java的一些封装类中也已经重写了该方法，以提供我们认知中的正确的相等判断。

- 三目操作类C(`a == b?1:0;`)
- `String`可以使用`+`连接

```
String s = "hello";
String r = "world";
String t = s + r;
System.out.println(t); // helloworld
```

- **强制转换**
  - 基本类型:

    - 格式: `int a  = (int)1.0f`
    - 如果转换是安全的，则可以隐式的自动转换（如int -> double）
    - 如果转换不安全（会损失精度），则需要自行显式转换（如double - > int）
    - boolean不能强转

> char → int, byte → short, short → int, int → long, long → float, float → double（安全顺序）

- 自定义类

> 一般只有父类与子类之间会进行转换

- 条件(if-else)，循环(while, do-while, for) 与跳转(return, break, continue, switch)类C
- **For-Each**

> - 本质上是语法糖
> - 原理是利用要遍历的对象的迭代器进行迭代
>
> ```
> for (Integer integer : arrayList){
>     System.out.println(integer);
> }
> ```
>
> 经反编译后得到，实质上被编译为
>
> ```
> Iterator arrayIterator = arrayList.iterator();
> while(arrayInterator.hasNext()){
>     Integer integer = (Integer)arrayIterator.next();
>     System.out.println(integer);
> }
> ```

Java 包

包 (package)

> - 由多个类组成
> - 共享一个命名空间(namespace)
>
> > 同一个包中类的名字不能相同
> >
> > 不同的包中类的名字可以相同
>
> - 使用包
>
> ```
> import java.util.ArrayList;
>
> public class Test{
>     public static void main(String[] args) {
>         ArrayList<Integer> list = new ArrayList<>();//直接使用
>         list.add(114514);
>     }
> }
> ```
>
> - 创建包
>
> ```
> package myPackage;
> //yourCodes
> ```
>
> 使用`package`语句创建包，包的结构与文件目录结构一致。

- 可以打包成 jar 包，将*目录*打包成*单文件*，方便使用

Java 访问控制

- 访问控制：控制类的数据和方法是否能被访问，以及能被谁访问
- 共有四种：
  - package access
    - 同一个包中的类可以访问
    - 其他包中的类不能访问
    - 没有标识符
    - 如果没有package声明是哪个包，则默认当前目录下的所有java文件同属一个包
    - 对class修饰，每个java源文件中除去public class以外，其他class都是package access
  - public
    - 所有包中的类都可以访问
    - 对class修饰，每个java源文件文件都包含一个public class，该class的名字应与该java源文件的文件名相同
  - private
    - 除了该类自身以外，所有类都不能访问该成员
    - 构造函数被标识为private，则外部无法直接创建该类的对象

      设计模式之单件模式：

      ```
      public class MyType {
          private int i;
          private double d;
          private char c;
          public void set(double x) { d = x;}
          public double get() { return d; }
          private MyType(int i1, double d1, char c1){
              i = i1; d = d1; c = c1;
          }
          private static MyType instance= null;
          public static MyType getInstance(){
              If (instance == null)
              Instance = new MyType(1, 1.0, ‘a’);
              return instance;
          }
      }
      ```

      只能创建一个对象的类。设计中应避免使用。
  - protected
    - 只有该类和其子类可以访问该成员

Modifier	Class	Package	Subclass	World
public	T	T	T	T
protected	T	T	T	F
no modifier(p.a.)	T	T	F	F
private	T	F	F	F

- 应善用访问控制，对类进行**封装**，即在满足需求的情况下，接口尽量简单，尽量只提供接口，在可能的情况下尽量使用private，尽可能隐藏细节。
- **final关键字：**

  - final 数据：编译时常数，一旦被赋值就不能被修改
  - final 引用：一旦被赋值就不能再指向其他对象，**但对象本身并不受影响**
  - final成员在定义时可以不给初始值，但必须在构造函数中初始化
  - final 方法：不能被重写
  - final class：不能被继承

Java面向对象

面向对象编程概述

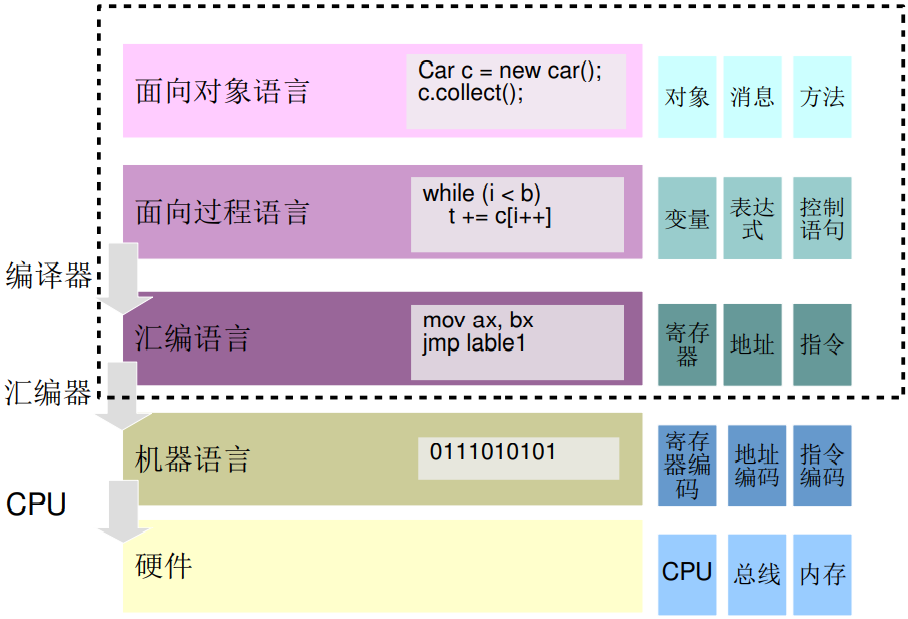
- **从实际问题到计算模型：**
  - 对于给定的某实际问题（如计算两个向量的内积）

    面向过程编程需要定义函数，实现函数并控制计算流程。问题是每个数组参数都需要一个长度参数，向量的实现方式也不确定。抽象程度不高。

    面向对象编程是从对象本身出发，首先将问题转化为不同的对象，再考虑对对象本身进行变换。并且，不同的对象可以有不同的功能，对象之间也可以传递消息。**抽象程度更高。**
- **面向对象语言：**
  - 编程语言直接提供了对对象的支持

    定义、构造对象，对对象的操作，对象提供的服务以及消息传递方法等
  - 提高了问题的抽象程度，缩短了实际问题到计算机算法的距离。 *(图源ybwu.org，如侵删)*

程序语言的抽象层级



- **面向对象编程要素：**

  - 任何事物都是对象
  - 程序为一些对象间的相互协作
  - 一个对象可以包含另一个对象
  - 每个对象都有类型
  - 同一类型的对象接收相同类型的消息，提供相同类型的服务

- **对象：**

  - **对象的基本要素：状态、行为和类型**
  - 对象的状态(State)

    - 每一个对象都有自己的状态
    - 程序可以改变一组对象的状态

  - **对象的接口(Interface)**

    - 对象向外界提供的服务，“行为”
    - 接口的实现
    - **隐藏实现的细节**（封装，Encapsulation，梦中情码就是所有接口完美的封装）
    - *In any relationship, it's important to have boundaries that are respected by all parties involved.* **不该看的不看**

  - 对象的类型 (Type, class)

    同类型的对象就是一组行为相同但可能状态不同的对象。

    可类比基本类型，但是更加多元化。

    类型或类是图纸，对象是按照图纸制造的机器。

  - 对象存储位置

    - 对象本身存储在堆中
    - 基本数据类型和局部变量在栈中

- new的作用相当于malloc

- static

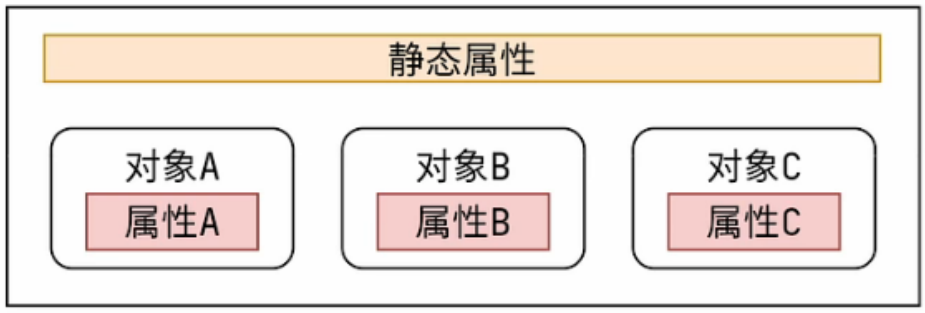
  静态方法 (static methods)

  - 不用创建对象即可被调用的方法
  - 在定义时使用`static`关键字
  - 也被称为**类方法** (class methods)

  静态数据 (static data)

  - 类似于静态方法，不依赖于类的实例化
  - 也被称为**类数据** (class data)

  简单来说，静态的方法和数据就是属于类的方法和数据，不且不能依赖于类的实例化。



(图源【每天一个技术点】static关键字原来还有这么多用法哔哩哔哩bilibili，如侵删)

- this关键字

  - 含义：在类的**非静态方法**中，返回调用该方法的对象的**引用**
  - 可以被用作其他函数方法的参数，也可以在构造函数中使用this关键字调用构造函数（只出现在构造函数第一行，只能调用一个构造函数）

```
public class MyType {
    int i;
    double d;
    char c;
    void set(double x) { d = x; }
    double get() { return d; }
    MyType(double d) {this.d = d;}
    MyType(int i) {this.i = i;}
    MyType(int i, double d, char c){
        this(d);
        this.i = i; // can not use this(i) again
        this.c = c;
    }
    public static void main(String [ ]args) {
        MyType m = new MyType();
        m.set(1);
    }
}
```

  编译器在调用类的非静态方法时，隐式地增加了参数一个this，而静态方法则不会。所以实际上静态与非静态环境的区别，是是否与具体对象绑定，也即可以通过能否使用this关键字区分。

- # Java 面向对象操作
  - ## 类的复用
    - Has-a关系，组合(Composition)

      > class B 中包含有 class A类型的数据成员
      >
      > 例：

      ```
      /*
      已有的类：
      class Engine;    //引擎
      class Wheel;     //轮胎
      class Clutch;    //离合
      */
      class Car{
       Engine engine;
       Wheel wheels[4];
       Clutch clutch;
      }
      ```

    - Is-a关系，继承(Inheritance)

      > class B中不仅带有 class A 所有的数据和方法成员，同时还增加了新的成员，或者修改原有的成员
      >
      > 例：

      ```
      public class Animal {
       String name;
       public Animal(){

       }
       public Animal(String name){
           this.name = name;
       }
       void eat(){
           System.out.println("Animal eating.");
       }
       public static void main(String[] args) {
           Cat cat = new Cat("meow");
           Animal animal = new Animal("I don't know");

           cat.eat();
           animal.eat();
       }
      }
      class Cat extends Animal{
       public Cat(){
           super();
       }
       public Cat(String name){
           super(name);
       }
      ```

```
    @Override
    void eat(){
        System.out.println("Cat eating.");
    }
}
```

- 新类包含已有类的方法和数据，并可修改
- 子类有父类的所有方法和数据
- 子类可以定义新的方法和数据
- 子类可以更新父类的方法，称为**重写(override)**

继承实质上是子类包含一个父类的对象作为数据成员，使用super关键字作为该父类对象的引用。可以通过super关键字来调用父类的方法。

- 构造函数

  > 调用父类带参数的构造函数，且必须出现在子类构造函数的首行

每个类都是Object class的子类，其包含有`toString() equals()`等方法。

函数重写 (Override)

> **方法的重写规则**
>
> - 参数列表与被重写方法的参数列表必须完全相同。
> - 返回类型与被重写方法的返回类型可以不相同，但是必须是父类返回值的派生类（java5 及更早版本返回类型要一样，java7 及更高版本可以不同）。
> - 访问权限不能比父类中被重写的方法的访问权限更低。例如：如果父类的一个方法被声明为 public，那么在子类中重写该方法就不能声明为 protected。
> - 父类的成员方法只能被它的子类重写。
> - 声明为 final 的方法不能被重写。
> - 声明为 static 的方法不能被重写，但是能够被再次声明。
> - 子类和父类在同一个包中，那么子类可以重写父类所有方法，除了声明为 private 和 final 的方法。
> - 子类和父类不在同一个包中，那么子类只能够重写父类的声明为 public 和 protected 的非 final 方法。
> - 重写的方法能够抛出任何非强制异常，无论被重写的方法是否抛出异常。但是，重写的方法不能抛出新的强制性异常，或者比被重写方法声明的更广泛的强制性异常，反之则可以。
> - 构造方法不能被重写。
> - 如果不能继承一个类，则不能重写该类的方法。
>
> *cr. [Java 重写(Override)与重载(Overload) | 菜鸟教程 (runoob.com)]*

多态

Upcasting

> - 对于一般的两个不同类型的对象来说，相互转换类型是不可以的，因为二者没有关联。
> - 但对于子类和父类的对象来说，子类是可以向上转型(Upcasting)为父类对象的，也就是说父类的引用可以指向子类对象。**因为子类拥有父类所有的数据和方法。**

- 也就是说，如果在某个地方需要某个类的对象，那么对其子类对象也同样可以适用。并且这种Upcasting是安全的。
- Downcasting

  > 当且仅当转换的引用确实指向子类对象时才能进行。

- 同一**基类**的不同子类可以被视为同一类型（基类），也即可以放宽类型一致性。这样，如果我们需要针对某一类的所有子类的接口，只需要声明为该基类即可适用所有子类，以简化接口。

同一个对象，在不同的阶段可以灵活的表现出多种对应的状态（类型），这就是多态。

动态绑定（Dynamic Binding）

- 静态绑定

  - 函数的调用在**编译之后**就已确定
  - 也叫 early binding
  - 优点是快速，易于debug；缺点是接口繁琐

- 动态绑定

  - 函数的调用在**运行时**才能确定
  - 也叫 late binding
  - 优点是接口简洁；缺点是函数调用需要额外开销，debug困难。

- Java 中的所有方法都采用动态绑定，除了final和static
- **数据成员不使用动态绑定**

```
public class Test{
    public static void main(String[] args){
        ClassA a = new ClassA();
        ClassB b = new ClassB();
        System.out.println(a.a);//1
        System.out.println(b.a);//2
        /*
        ClassA b = new ClassB();
        System.out.println(b.a);
        输出为1
        */
    }
}
class ClassA{
    int a = 1;
    public void methodA(){
        System.out.println("ClassA.methodA");
    }
}
class ClassB extends ClassA{
    int a = 2;
    @Override
    public void methodA(){
        System.out.println("ClassB.methodA");
    }
}
```

- 构造函数
  - 初始化顺序
    - 分配内存空间，默认初始化
    - （递归）初始化父类
    - 静态成员初始化（若首次创建该类对象）
    - 数据成员初始化（按照定义顺序）
    - 调用构造函数
  - 构造函数中避免使用将被重写的函数

```
public class Test{
    public static void main(String[] args){
        ClassA a = new ClassA();
        ClassB b = new ClassB();
        /*
        输出为
        ClassA.methodA
        ClassB.methodA
        ClassB.mehtodA
        */
    }
}
class ClassA{
    int a = 1;
    public ClassA(){
        methodA();
    }
    public void methodA(){
        System.out.println("ClassA.methodA");
    }
}
class ClassB extends ClassA{
    int a = 2;
    public ClassB(){
        methodA();
    }
    @Override
    public void methodA(){
        System.out.println("ClassB.methodA");
    }
}
```

- 多态适用于协变返回值，即被重写的函数返回值可以是原函数的子类

Java进阶内容

抽象类

- 对于一个基类而言，如果**所有**的子类都将重写某一方法，那么在该基类中**实现**该方法是否还有必要?

抽象方法 (abstract method)

> - 仅提供方法的名称，参数和返回值
> - 没有具体实现
> - 使用`abstract`关键字

```
abstract class ClassA{
    public abstract void methodA(int a);
}
```

抽象类 (abstract class)

> - **包含**抽象方法的类称为抽象类
> - 抽象类是不完整的类（缺失抽象方法的实现）
> - 抽象方法需要在子类中重写后才有意义
> - 所以不能**直接**创建抽象类的对象
> - 若子类没有重写父类中的抽象方法，子类仍为抽象类
> - 抽象类中可以有数据成员，也可以有正常实现的方法

接口

定义

> - “所有方法都是抽象方法的类”
> - 所有方法都只有方法的名称，参数和返回值，而不实现方法
> - 接口没有代码重用，仅仅保留了**Upcasting和多态**
> - 接口需要被实现，实现某接口的类必须重写所有接口中定义的方法
> - 所有实现该接口的类都有接口提供的方法，也就是说，任何使用该接口类型的方法，都可以使用他的任何一种实现。类似某种协议，所以称之为接口(Interface)

```
interface Instrument{
    void play(int note);
    String what();
}
class Stringed implements Instrument{
    public void play(int note){
        System.out.println("Stringed played");
    }
    public String what(){
        return "Stringed.";
    }
}
```

> - 接口的所有方法默认为public
> - 所有数据默认为final static，所以可以用来定义常量

实现多个接口（多继承问题）

- **多继承问题 (Diamond Problem):**

```
public class DiamondProblem{
    public static void main(String []args){
        Cog cog = new Cog();
        cog.say();
    }
}
interface Cat{
    public void say();
}
interface Dog{
    public void say();
}
class Cog implements Cat, Dog{
    @Override
    public void say(){
        System.out.println("Meof");
    }
}
/*
如果换成普通类，会有什么问题？
class Cat{
    public void say(){
        System.out.println("Meow");
    }
}
class Dog{
    public void say(){
        System.out.println("Woof")
    }
}
*/
```

多继承的决议问题

> 多继承可以分为**声明**多继承和**实现**多继承两个层面，Java支持声明多继承，也即是支持实现多个接口。原因是，即使多个接口中有相同的方法声明，最终，方法还是在实现这些接口的类中实现的，所以不会造成决议问题。而实现多继承则会产生决议问题：除非特别指明，否则机器无法得知使用哪个父类的方法。

- Java中规定每个类只能有一个（除Object之外的）普通类或抽象类作为基类，但可以实现多个接口。
- 实现多个接口的意义在于保证规范，以及最大限度利用Upcasting和多态机制。

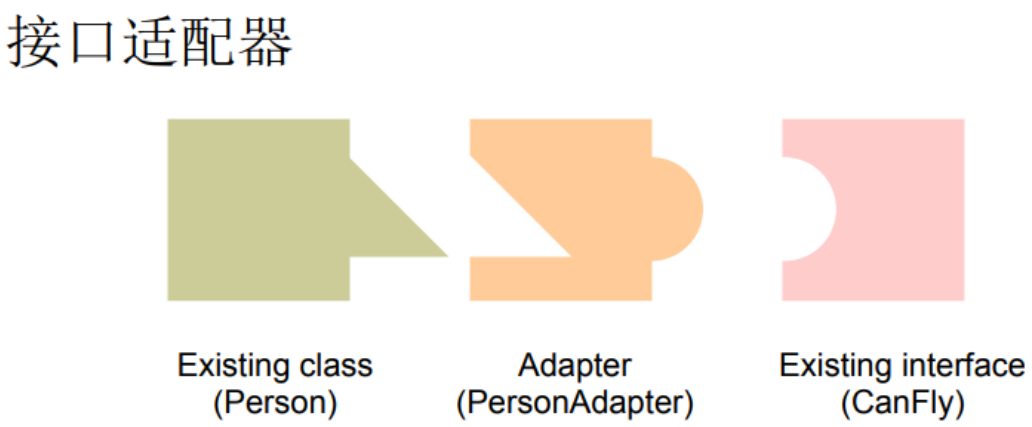
拓展接口

- 使用 `extends` 关键字对接口进行拓展

```
interface Monster{
    void menace();
}
interface DangerousMonster extends Monster{
    void destroy();
}
interface Lethal{
    void kill();
}
interface Vampire extends DangerousMonster, Lethal{
    void drinkblood();
}
```

- **接口适配器 (Adapter)**
  - 已有方法f，参数类型为Interface1
  - 假设类A已经存在，但并未实现Interface1接口
  - 希望f能处理A的对象

```
interface CanFly{
    void fly();
}
class Person{
    public void walk(){};
    public void buyTicket(){};
    public void takeFlight(){};
}
class PersonAdapter implements CanFly{
    private Person p;
    public PersonAdapter(Person p){
        this.p = p;
    }
    public void fly(){
        p.buyTicket();
        p.takeFlight();
    }
}
public class Adventure{
    public static void travel(CanFly c){
        c.fly();
    }
    public static void main(String []args){
        Person p = new Person();
        PersonAdapter pd = new PersonAdater(p);
        travel(pd);
    }
}
```



（图源bwu.org，如侵删）

- 工厂模式：设计模式的一种，当构造对象比较繁琐时，可以增加一层包装。

内部类

- 定义在一个类的内部
- 与组合不同

```
class Outer{
    //your code
    class Inner{
        //your code
    }
}
```

- 内部类可以帮助隐藏实现细节，帮助组织代码
- 若要返回内部类的引用，则需使用`OuterClassName.InnerClassName`
- 内部类与外部类的关系

  - 内部类的对象隐含了一个引用，指向包含它的外部类对象
  - 内部类对象能够访问该外部对象的所有成员和方法
  - 使用`OuterClassName.this`在内部类中访问外部类对象的引用
  - 内部类对象的创建：

    在外部类的方法中：可以直接创建

    其他地方：`OuterClassObject.new`

```
public class Outer{
 class Inner{

 }
 public static void main(String []args){
     Outer o = new Outer();
     Outer.Inner i = o.new Inner();
 }
}
```

- 内部类通常实现某个接口或继承某个类（例如实现Iterable的一般方式是定义一个内部类实现对应的Iterator）
- private的内部类可以完全隐藏内部类，外界仅仅知道接口，但不知道内部类的存在。

其他内部类

- 定义在方法中的内部类
  - 也被称为local inner class
  - 在方法之外，该类不可见
- 定义在其他任意作用域中的内部类，在该作用域之外不可见
- 匿名内部类
  - 没有名字的内部类，必须继承某个类或实现某个接口
  - 没有构造函数，必须同时定义和创建对象
  - 使用外部变量对匿名函数类数据成员初始化时，外部变量需要是final

```
public class Parcel{

    public Contents contents(){
        return new Contents() {
            // anonymous inner class definition
            private int i = 11;
            public int value() {return i;}
        };
    }

    public static void main(String []args){
        Parcel p = new Parcel();
        Contents c = p.contents();
    }
}
```

```
public interface Contents{
    int value();
}
```

“ 创建一个实现 Contents 的匿名类”

语法解释
1. “;” 为 return 语句的分号
2. 在 return 语句中定义匿名类
   - 实现 Contents 接口
   - 花括号内部
3. 创建一个该匿名类的对象
   - new Content () { }

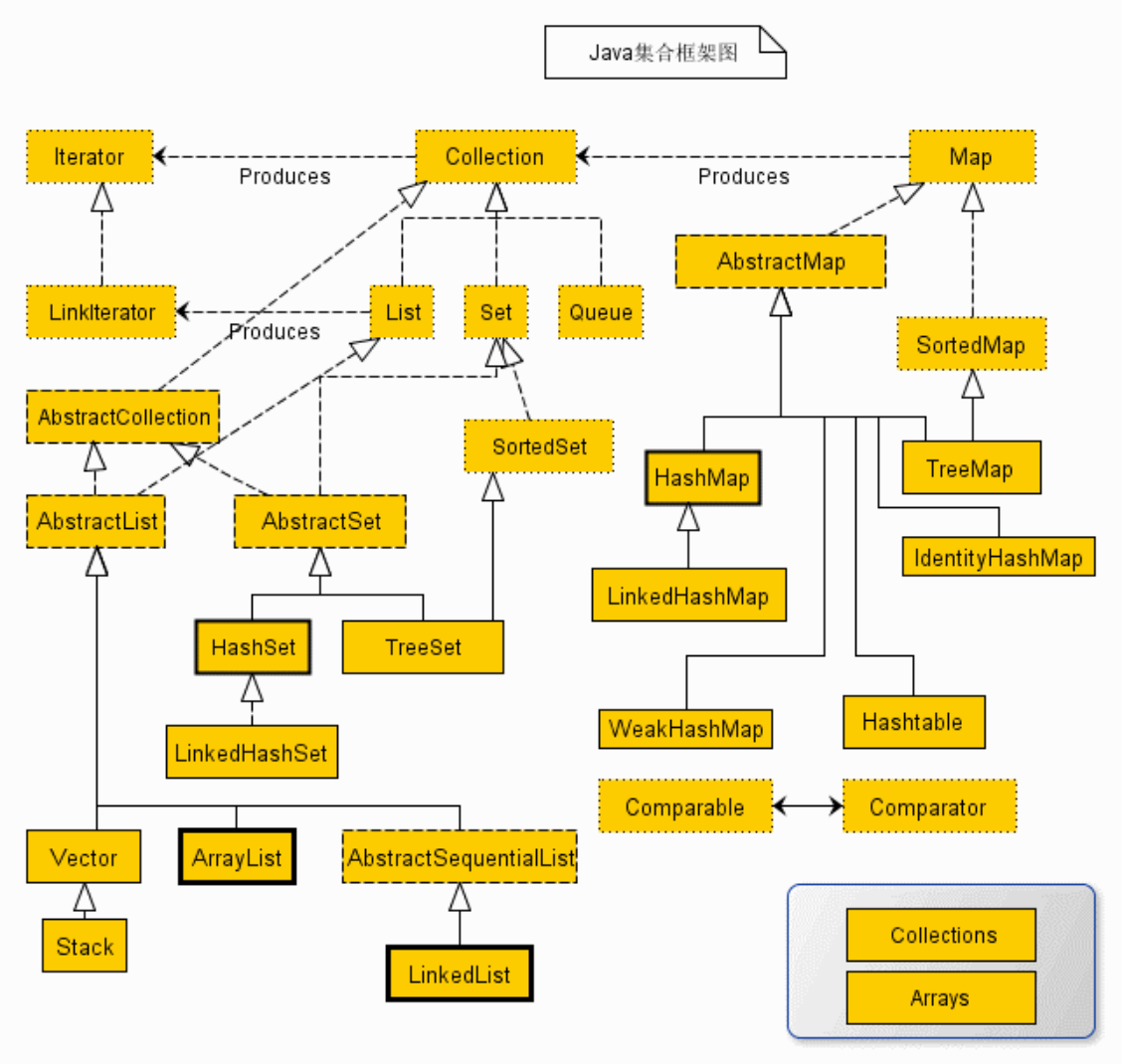
图源ybwu.org，如侵删

- 嵌套类
  - 静态的内部类
  - 不需要外部类的对象即可创建
  - 接口中的内部类默认都是静态内部类，也即嵌套类

内部类的作用

- 多继承：可以通过多个内部类继承多个类/抽象类/接口

容器



图源[Java 集合框架 | 菜鸟教程 (runoob.com)](), 如侵删

- ## 容器简介

  > - 使用数组组织对象的缺点是长度不可变，也无法体现数组元素之间的关系。
  > - 容器提供了更加灵活的组织对象的方式（例如动态添加，删除等）
  > - 有List, Set, Queue, Map等
  > - 位于java.util中

  - **List:**

    > - 一列有序的对象（数组、链表）
    >
    > ```
    > import java.util.*;
    >
    > ArrayList a = new ArrayList();
    > LinkedList b = new LinkedList();
    > ```
    >
    > - 实现了List接口

  - **Set:**

    > - 集合，没有重复元素

```
import java.util.*;
HashSet a = new HashSet();
TreeSet b = new TreeSet();
```

- 实现了Set接口

- **Queue:**

  - 队列，先进先出 FIFO(First In First Out)
  - 有入队(enqueue),出队(dequeue)两种操作
  - 可以用来作任务调度

```
import java.util.*;

LinkedList a = new LinkedList();
PriorityQueue b = new PriorityQueue();
```

  - 实现了Queue接口

- **Map:**

  - 包含一组组 Key-Value 对
  - Key 不重复，Value可以重复

```
import java.util.*;

HashMap a = new HashMap();
```

  - 实现了Map接口

泛型

- 我们既需要容器中只存放同一类型的对象，又需要不必为每种类型都单独创建一种容器。此时泛型就出现了
- 容器可以存放的类型为Object，也即是任何类型的对象都可以被放入容器中，而容器的类型只能在运行时确定

  `ArrayList<T> a =  new ArrayList<T>();`
- 使用泛型能保证类型安全，同时支持Upcasting
- 对基本类型，只能使用其对应的封装类

```
import java.util.*;

ArrayList<Integer> a = new ArrayList<Integer>();
for(int i=0;i<10;++i){
    a.add(i);//自动装箱autoboxing
}
```

容器接口

Collection接口

> - 用于存放一组对象
> - List接口：对象按照插入顺序排列容器中的对象
> - Set接口：容器中不能有重复的对象
> - Queue接口：按“队列”规则插入或删除对象

Map接口

> - 用于存放一组“键-值对”(key-value pair)
> - 也被称为字典(dictionary)

List

> - List接口定义的方法：
>
>> - `add()`：添加元素
>> - `remove()`：删除元素
>> - `get(int i)`：返回第i个位置的元素
>> - `size()`：返回元素数量（容器大小）
>
> - 构造函数：
>
>> - ArrayList:
>>
>> ```
>> ArrayList<E>();
>> ArrayList<E>(int initialCapacity);
>> ArrayList<E>(Collection<E> c);
>> ```
>>
>> - LinkedList:
>>
>> ```
>> LinkedList<E>();
>> LinkedList<E>(Collection<E> c);
>> ```
>
> - 迭代器：
>
>> - 用于遍历访问Collection中的元素的工具对象
>> - 若要用for-each遍历自定义类，则需要实现iterable，iterable要求提供其迭代器，因而要新建一个类实现iterator
>> - List接口提供了ListIterator，可以用来双向遍历
>> - 拓展：
>>
>>> 作为迭代器，要有时空一致性，也即：
>>>
>>> 1. 在A处定义的迭代器，在B处应能正常使用
>>> 2. 在T时刻定义的迭代器，在T'时刻应能正常使用
>>>
>>> 同时，迭代器也应该具备独立性和隔离性：
>>>
>>> 1. 独立性：不同迭代器遍历元素时互不影响
>>> 2. 隔离性：如果集合增删元素，不能影响到**已有的**迭代器

> > > 所以如果要使用增强for循环(for-each)，需要实现iterable接口，因为每次使用都要用到不同的迭代器，但它们之间相互独立，互不干扰
> >
> > 但这只是满足了前提，具体实践会遇到一些问题，例如：
> >
> > - 在迭代的过程中增删元素
> >
> > 此时迭代器很有可能不能够正常地继续迭代，这也与迭代器`hasNext()`的具体实现有关，此时就需要一些机制来避免迭代器迭代产生错误。
> >
> > 例如，ArrayList中使用变量`modcount`来记录操作的次数。在生成一个新的迭代器时记录下当前的`modcount`，每次迭代时检查`modcount`是否变化，如果变化则说明容器已经被增删过，此时采用快速失败Fail-fast机制，直接抛出异常。
> >
> > 但也并不是无法在迭代时对容器进行操作，可以使用其它方法实现，例如Copy On Write，COW

- LinkedList

  > - 实现了List接口和Queue接口
  > - 提供了更多的方法，如`add(), remove(), element(), offer(), poll(), peek()`
  > - 应用：实现Stack(Last In First Out, LIFO)

Set

> - Set接口定义的方法：
>
>   > - `add(Object o), addAll(Collection<E> c)`：添加元素
>   > - `remove(Object o), removeAll(Collection<E> c)`：删除元素
>   > - `contains(Object o)`：是否包含元素o
>   > - `iterator()`：返回迭代器
>   > - `size()`：返回元素数量（容器大小）
>   > - `toArray()`：转换成数组
>
> - HashSet
>
>   > - 特点：快速（增删查），无序
>
> - TreeSet
>
>   > - 特点：速度较慢（增删查），有序
>
> - LinkedHashset
>
>   > - 特点：速度快，按插入顺序排列

Queue

> - 规则：先入先出
> - 接口提供的方法：
>
>   > - `offer(Object o), add(Object o)`：将对象加入队列尾部
>   > - `poll(), remove()`：弹出位于队首的对象
>   > - `peek(), element()`：返回位于队首的对象，并不删除

- **PriorityQueue**
  - 优先级队列
  - 每次出队时，选择优先级最高的对象
  - 队列中的对象可以比较优先级
  - 普通队列也可以看作是优先级为加入队列时刻的优先级队列
  - 自定义优先级：

    构造函数：

    ```
    PriorityQueue<E>(int initialCapicity, Comparator<E>
    comparator);
    ```

    - Comparator接口：
      - 定义两个元素的优先级关系
      - 包含方法`compare(E e1, E e2)`（类似C中的cmp函数）

        Compare返回：

        - 正，e1<e2
        - 负，e1>e2
        - 零，e1=e2

    ```
    PriorityQueue<Character> rqc = new
    PriorityQueue<Character>(10,
        new Comparator<Character>(){
            public int compare(Character c1, Character
    c2){
                if(c1 > c2) return -1;
                else if(c1 < c2) return 1;
                else return 0;
            }
        });//使用匿名类构造Comparator
    ```

Map

- Map接口定义的方法：
  - `put(K key,V value)`：存入键值对
  - `get(K key)`：返回键对应的值
  - `containsKey(Object key)`：是否包含键key
  - `containsValue(Object value)`：是否包含值Value
  - `keySet()`：返回键组成的Set
  - `values()`：返回值组成的Collection

异常处理

- ## Java错误处理流程

  > - 某方法中发现错误
  > - **中断**当前方法的执行
  > - **创建 / 捕捉 Exception** 类对象
  > - **跳转**到相应的异常处理代码段
  > - 在代码段中**处理**该异常

- ## 抛出异常

  > - 需要自行检查错误条件
  > - 使用throw关键字抛出异常
  >
  > ```
  > if(t == null) throw new NullPointerException();
  > ```
  >
  > - 含义为发生了一个异常，需要合适的异常处理模块处理，异常的具体信息存储在一个 Exception 对象中

- ## 处理异常

  > - try-catch
  >
  > ```
  > try{
  >     //可能会抛出异常的代码
  > }
  > catch (ExceptionType1) {
  >     //处理类型为"ExceptionType1"的异常
  > }
  > catch (ExceptionType2) {
  >     //处理类型为"ExceptionType2"的异常
  > }
  > catch (ExceptionType3) {
  >     //处理类型为"ExceptionType3"的异常
  > }
  > ```
  >
  > - 一旦发生异常立即跳转捕获

- ## 异常对象

  > - Exception 类的子类
  > - Exception 类的方法
  >   - `toString()`
  >   - `printStackTrace()`
  > - 通常情况下不需要重写Exception类中的任何方法

- ## 分离式异常处理

  > - 在不同的方法中完成正常代码与错误代码的隔离
  > - 方法中只抛出异常，而不进行处理，将处理交由调用者
  > - 使用throws关键字标识该方法可能会抛出何种类型的异常
  >
  > ```
  > bar() throws ExceptionType1, ExceptionType2{
  >     //your code
  > ```

```
    throw new ExceptionType1();
    //your code
    throw new ExceptionType2();
}

foo(){
    try{
        //your code
        bar();
    }
    catch (ExceptionType1 e){
        //your code
    }
    catch (ExceptionType2 e){
        //your code
    }
}
```

- 编译器保证如果方法使用了throws关键字，则在调用处必须要处理相应的异常
- 如果不包含throws关键字而发生了异常，默认抛出RuntimeException 类型的异常
- 捕获到的异常也可以再次被抛出，交由调用者的调用者处理

Java标准异常

- 都是Exception类的子类
- 大多数通过名字表明含义（如IOException, RuntimeException, SQLException等）
- Java提供了许多现成的异常，也会自动抛出一些异常，例如
  - 数组越界时抛出ArrayOutOfBoundException
  - 访问空对象时抛出NullPointerException
  - 除以0时自动跑出ArithmeticException
- 抛出RuntimeException通常表示程序有bug，因为不会主动抛出。通常情况下不需要catch Runtime Exception，它会由main函数自动catch并调用printStackTrace()

finally关键字

- 无论try是否有异常抛出，都会执行
- 无论是否有对应的catch语句，都会执行
- 作用：帮助保证一致性，简化代码（例如保证在某函数运行结束，某对象一定处于某状态）

I/O

比较碎且多，所以省略（绝不是因为懒），建议仔细阅读PPT和课本