1. Java 程序的组成

1.1 Java 程序是由各种类组成

Java 语言是一种分布式面向对象语言，对于对象概念中的类、对象、继承、封装、多态、接口和包等具有很好的支撑。

2. 基础语法

2.1 基础数据类型

一共有八种基本数据数据类型

类型	存储需求	解释
int	4字节	整数型，通常情况下使用
short	2字节	用于特定应用场合，例如底层文件处理或者需要控制占用内存空间量的大数组
long	8字节	长整型数值，可加后缀l或者L（114514L）
byte	1字节	同short
float	4字节	只需要单精度或者存储大量数据的库，可加后缀F或者f，没有后缀的F的浮点数（例如2.72）默认为double

类型	存储需求	解释
double	8字节	数值精度是float两倍（绝大多数情况使用double），可以加后缀D或者d（1.14D）
char	16字节	字符类型，若要使用字符串不建议使用char做字符序列
boolean	1字节	返回true，false布尔值，java中，0值和非0值无法直接代表false和true

特殊的浮点数：

正无穷大、负无穷大、NaN（不是一个数字），表达为溢出和错误，NaN与其它数字和自己比较结果为false

Void 不是基础数据类型

2.2 引用数据类型

引用数据类型存放的值为指向目标函数的引用

除去基本数据类型，其它的类型都是引用数据类型，自己定义的class类都是引用类型，可以像基本类型使用。数组、类、接口被称作引用数据类型。

String, StringBuffer, ArrayList, HashMap等都是

2.3 变量

1、函数中定义的变量为 **局部变量** 作用域从什么位置到语句块末尾。

2、类中声明的成员变量分为

对象成员变量

对象成员变量储存在对象中，生存期与对象一致，用过对象访问，可访问性用

`public` 、 `private` 和 `protected` 等修饰符访问

（这三个修饰符区别看 3.）

类成员变量

类成员变量声明用 `static` 关键词，可以在一个类中的多个方法使用。

```
static int value;
```

3、类常量

使用关键字 `static final` 定义一个类常量

```
public static final double Value = 114514;
```

类常量定义位于main方法的外部，同一个类的其它方法中也可以使用这个常量。如果常量被声明为public，其它类的方法也可以使用这个常量。

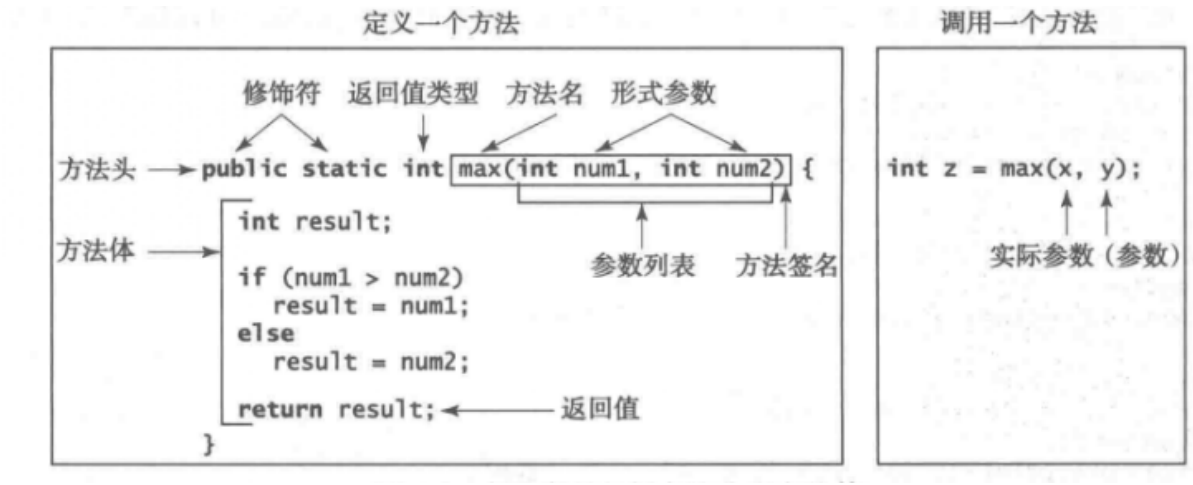
2.4 函数

2.4.1 定义函数（方法）

对象成员函数代表对具体对象的操作，需要先创建对象在用过对象引用调用函数。

静态函数通过 `static` 关键字修饰，不依赖对象直接调用

方法定义：



例子：

```
public class Testmax{
  public static void main(String[] args){
    int i = 5;
    int j = 2;
    int k = max(i, j);
    System.out.println("The maximum of " + i + " and " + j "
is " + k);
  }
  // 比较两个数最大值
  public static int max(int num1n int num2){
    return num1>num2?num1:num2;
  }
}
```

2.4.2函数的参数传递：

函数的参数传递采用值传递的方法，调用函数方法是，按照**参数顺序匹配。**

基本类型的传递（`int`、`double`等）

引用类型的传递（`StringBuilder`、 自己创建的类）

注意!

因为Java不能直接操作指针，所以说像无法像C++一样实现swap（）方法

```
// c++ 实现swap操作
template<class T>
void swap(T &a, T &b){
  T temp = a;
  a = b;
  b = temp;
}
//可以实际交换a,b
```

```
// java中无法直接实现swap
public static void swap(int a, int b){
  int temp = a;
  a = b;
  b = temp;
}
// 实际上并没有交换a,b ...
```

解决方法有很多，可以在网上找到很多方法解决

2.4.3 函数重载：

函数名称相同，参数列表不同。在调用函数时，根据实参与函数参数列表的匹配情况调用

编译器会挑出具体执行的方法，通过用各种方法给出的参数类型与特定方法调用所使用的值类型进行匹配来挑选出相应的方法。如果没有找到匹配的参数，会产生编译错误

例子：

```
public class TestMethodOverLoading{
  public static void main(String[] agrs){
    System.out.println(max(3, 4));
    System.out.println(max(3.0, 4.0));
    System.out.println(max(3.0, 5.4, 10.14));
  }
  public static int max(int num1, int num2){
    return num1>num2?num1:num2;
  }
  public static double max(double num1, double num2){
    return num1>num2?num1:num2;
  }
  public static double max(double num1, double num2, double
num3){
    return max(max(num1, num2), num3);
  }
}
```

2.5 字符串

2.5.1 三种不同方法

`String` 是 Java 中的对象。

Java中创建处理字符串的有

`String` 、 `StringBuffer` 和 `StringBuilder`

`String` 类是不可变类，一旦一个String对象被创建以后，包含在对象中的字符序列是不可变的。

`StringBuffer` 可变字符串对象

`StringBuilder` 类是可变字符串对象，StringBuilder实现了线程安全功能，性能更高，如果需要创建内容可变的字符串对象，则应该优先考虑到使用 `StringBuilder` 类

操作少量的数据使用 String。 单线程操作大量数据使用 StringBuilder。 多线程操作大量数据使用 StringBuffer。

2.5.2 String

实例：

```
String s = "";//空字符串
String greeting = "Hello";
```

字串：

```
String greeting = "hello";
String s = greeting.substring(0,3);//Hel
//范围左闭右开
```

拼接：

```
String expletive = "Exletive";
String PG13 = "deleted";
String message = expletive + PG13;
```

是否相等：

```
String greeting = "hello";
"heLlO".equals(greeting);
"heLlO".equalsIgnoreCase(greeting);//不区分大小写
```

注意!

不要使用 `==` 比较两个字符串是否相等，这个运算符只能确定两个字符串是否放在同一个位置上。

API

方法	作用
length()	返回字符串字符数
charAt(int index)	返回字符串s中指定位置的字符
concat(String s1)	将本字符串和字符串s1连接，返回一个新字符串
toUpperCase()	返回一个新字符串，其中所有的字母大写
toLowerCase()	返回一个新字符串，其中所有的字母小写
trim()	返回一个新字符串，去掉两边空白字符
compareTo()	按照字典顺序，比较两个字符串，返回一个负数，正数，0
subString(int beginIndex, int endIndex)	返回一个截取的字符串
equals()	比较两个字符串是否相等
equalsIgnoreCase()	忽略大小写比较是否相等

2.5.3 StringBuilder

实例：

```
StringBuilder builder = new StringBuilder();
```

添加：

```
builder.append("hello");
```

得到一个String对象：

```
String str = builder.toString();
```

API

方法	作用
length()	返回长度
append(String str)	添加内容，返回this
setCharAt(int index, char c)	将index位置设置为c
insert(int index, String str)	在index位置插入str
delete(int startIndex, int endIndex)	删除一段的内容
toString()	返回成String类型

2.6 标准输入输出

2.6.1 Scanner输入

实例：

```
Scanner in = new Scanner(System.in)
  //这里以System.in举例，后面核心技术部分会展开
```

Sanner()的api

方法	作用
Scanner(inputStream in)	用给定的输入流创建一个Scanner对象
nextLine()	读取输入的下一行
next()	读取输入的下一个单词（以空格为分隔）
nextint()	读取并转换下一个表示整数
nextDouble()	读取并转换下一个表示浮点数
hasNext()	检查输入中是否还有其它单词
hasNextInt()	检查是否还有表示整数的下一个字符序列
hasNextDouble()	检查是否还有表示浮点数的下一个字符序列

注意!

若输入的一个字符，可以这么来写

```
Scanner input = new Scanner(System.in);
char c = input.next().charAt(0);
```

2.6.2 输出

```
System.out.println(...)//换行输出
System.out.print(...)//不换行输出
System.out.printf("...", ...)//格式化输出
```

具体的输入输出流和文件的输出输入看 **Java核心技术部分**。

2.7 数组

数组是一种数据结构，支持随机访问，本身也是对象，有一些对象操作，数组的属性length等

注意!

length是数组属性，直接这么写

```
int[] arr = new int[100];
arr.length;
```

2.7.1 数组初始化

```
//声明整型数组
int[] arr;
```

这条语句只是声明了变量a，没有将a初始化为一个真正的数组

```
int[] arr = new int[100];
//字符串长度可以传入变量，但是长度是固定的!
```

数组初始化器，也就是后面直接跟值

```
int[] arr = {1, 2, 4, 5};
object o = {new object(), new object()};
```

2.7.2 数组访问

通过下标随机访问

循环

for-each遍历

```
for(int temp: arr)
  System.out.println(temp);
```

注意一点，数组取出一个元素临时放在temp中，所以说这样子写是不会改变arr里的元素的

```
for(int temp : arr)
  temp = 114514;
```

数组作为参数传递，传递的是数组的引用

```
public static void change(int[] arr){
  for(int i = 0;i<arr.length;i++)
    arr[i] = i;
}
```

匿名数组

```
new int[]{1,2,3,4,5,6};
```

数组拷贝

```
int[] copiedNumber = Array.copyOf(copiedNumbers,
copiedNumbers.length)
```

3. 类与对象

3.1 类结构

Java中类的定义包含修饰符（public，final等），类定义关键字（class，interface，enum等），类名。

类中包含有属性（字段，成员变量）、构造器和成员方法（操作）

一个简单的类的例子

```
public class SimpleCircle{
  private double radius;
  SimpleCircle(){
    this.radius = 0;
  }
  SimpleCircle(double radius){
    this.radius = radius;
  }
  public void setRadius(double newradius){
    this.radius = newradius;
  }
  public double getRadius(){
    return radius;
```

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

构造方法

每个类可以用多个构造器

构造方法必须具备和所在类相同的名字

构造方法没有返回值类型

构造方法是在创建一个对象使用new操作符时调用

不要在构造函数中设置局部变量

对象

对象是类的实例，每个对象都保留着当前的特征状态，也就是说每实例一个类，就是一个单独的对象。

对于Circle 对象c 的值存档是一个引用

所以说

```
Circle c1 = new Circle();
Circle c2 = new Circle();
c1 = c2;
//变量c1 和 c2 指向同一个对象
```

例子:

```
//Emloyee 类
public class Employee
{
  private String name;
  private double salary;
  private LocalDate hireDay;

  public Employee(String name, double salary, int year, int
month, int day){
    this.name = name;
    this.salary = salary;
    hirDay = LocalData.of(year, month, day);
  }

  public String getName()
  {
    return name;
  }
  public double getSalary()
  {
    return salary;
  }
  public LocalDate getHireDay()
  {
    return hirday;
  }
  public void raiseSalary(double byPercent){
    double raise = salary * byPercent / 100;
    salary += rasie;
  }

}
```

主函数:

```
public class void main(String[] args)
{
  Employee[] staff = new Employee[3];

  staff[0] = new Employee("Carl Cracker", 75000, 1987, 12,
15);
  staff[1] = new Employee("Harry Hacker", 50000, 1989, 10,
1);
  staff[2] = new Employee("Tony Tester", 40000, 1990, 3,
15);

  for(Employee e : staff)
    e.raiseSalary(5);
  for(Employee e : staff)
    System.out.println("Name: " + e.getName() + " ,Salary "
+ e.getSalary() + " , hireDay: " + e.getHireDay());
}
```

3.2 类的封装

将类的属性和操作封装到一个单元中。

3.3 静态域和静态方法

如果将域定义为static， 每一个类中只有一个只有的域，而每一个对象对域所有的实例域都有自己的一份拷贝。

3.3.1 静态变量

静态变量被类的所有对象所共享。静态变量存储到一个公共的内存地址上，如果某一个对象修改了静态变量的值，那么同一个类的所有对象都会受到影响。

```
public class Circle{
  public static double pi = 3.14;
  ....
}

public static void main(String[] args){
  Circle c1 = new Circle();
  Circle c2 = new Circle();
  Circle c3 = new Circle();
  c1.pi = 3.1415;
  //c1,c2,c3 的所有pi都变成了3.1415
}
```

3.3.2 静态方法

对于非静态类，它们是通过引用变量来访问的。静态方法可以通过既可以通过引用方法来调用，有还可以直接通过类名来调用。

静态变量和方法还可以在不创建对象的情况下访问。

3.4 包

Java可以使用包package将类组成起来，使用包的原因是为了确保类名的唯一性。如果两个程序员都建立了一个Employee类，只要将类放置于不同的包中，就不会冲突。

（类比于C++里的名字空间）

`java.util` 、 `java.lang` 叫做一个包

使用 `instanceof`

`instanceof` 判断对象是否属于特定的类或者子类

5. 继承

继承就是子类复用父类代码的过程。Java支持引用类型转化，即将子类型引用转换为父类型引用

（引用类型转换，对象本身类型不会转换）

Java只支持单继承，子类只有一个父类

一些术语

超类，基类，父类，都是属于存在的类

子类，孩子类，派生类就是继承出的新类

5.1 继承的方法

使用关键字extend来继承

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

```
class A{
  public void methodOfA(){
    System.out.println("Method fo A");
  }
}

class B extend A{
  public void methodOfB(){
    System.out.println("Method of A");
  }
}
```

注意一点

```
A a = new B();
a.methodOfB();
```

这个程序是错误的，虽然是a是引用了B，但是A中是没有包含methodOfB字段的，（B继承于A，包含了A中的方法，反之则不没有），所以说a无调用子类中的方法。

5.2 super关键字

子类继承它的父类的所有**可访问的**数据域和方法，不能继承父类的构造方法

super构造器可以调用父类的构造器和方法

方法为

```
super();//无参构造器
super(参数);//有参构造器
```

语句 super() 和 super (arguments) 必须出现在子类构造方法的第一行，这是显式调用父类构造方法的唯一方式

super关键字调用父类方法

`super.方法名.(参数)`

比如下面的例子

```
public class Employee{
  public double getSalary(){
    return salary;
  }
  ...
}

public class Manager extends Employee{
  public double getSalary(){
    baseSalary = super.getSalary
    return bonus + baseSalary;
  }
}
```

因为子类也定义了一个getSalary，如果不用super来区分是子类还是父类的getSalary方法，会导致程序无限递归。

5.3 覆盖方法（方法重写）

父类中的有些方法对子类不一定适用，需要提供一新方法覆盖。

对用方法时调用的是实际的引用类型

```
Employee e = new Emloyee();
e.getSalary();//调用的是Employee里的getSalary方法
```

```
Employee e = new Manager();
e.getSalary();//调用的是Manager里面的getSalary方法
```

注意:

仅当实例方法是可访问时，他才能被覆盖

与实例方法一样，静态方法也能被继承，但是不能被覆盖

5.4 对象转换

对象的引用可以类型转换成另外一种对象的引用

例如

```
Object o = new Student();
```

或者:

```
object o = new obeject();
student b = (student)o;
```

总是可以将一个子类的实例转换为一个父类的变量，称为向上转换 ,因为子类的实例永远是它的父类的实例。当把一个父类的实例转换为它的子类变量（称为向下转换 时，必须使用转换记号 “(子类名)” 进行显式转换，向编译器表明你的意图。

5.5 instanceof 运算符

nstanceof通过返回一个布尔值来指出，这个对象是否是这个特定类或者是它的子类的一个实例

```
object myObject = new Circle()
  if(myObejct instanceof Circle)
    ...
```

6. 接口和多态

6.1 多态

一个对象变量，可以指示多种实际类型的现象被称为多态。在运行是能够自动地选择调用哪个方法的现象为动态绑定

例如

```
Employee e;
e = new Employee(..);
e = new Manager(..);
//动态绑定的例子
```

或者这样子情况下

```
public class PolymorphismDemo{
public static void main(String[] args){
  displayObject(new CircleFromSimpleGeometricObject(1,
"red", false));
  displayObject(new RectangleFromSimpleGeometricObject(1, 1,
"black", true));
}
  public static void displayObject(SimpleGeometricObject
object){
    ...
  }
}
```

```
Object o = new GeometricObject();
System.out.println(o.toString());
```

其中，Object是java中所有类的始祖，java中每个类都是有它扩展而来。

o可以引用所有类

6.2 抽象类

抽象类不可以用于创建对象。抽象类可以包含抽象方法，这些方法将在具体的子类中实现。

注意以下几点

抽象方法不能包含在非抽象类中

不能使用 new 操作符从一个抽象类创建一个实例

包含抽象方法的类必须是抽象的

即使子类的父类是具体的

6.2.1 抽象类的一个例子

使用abstract建立抽象类，抽象方法

```
//person 抽象类
public abstract class Person
{
  public abstract String getDescription();
  private String name;

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

  public String getName(){
    return name;
  }
}
```

```
// employee类，继承自Person
public class Employee extends Person
{
    private double salary;
    private LocalDate hirDay;

    public Employee(String name, double salary, int year,
int month, int day){
    super(name);
    this.salary = salary;
    hireDay = LocalDate.of(year, month, day);
    }
```

```
    public double getSalary(){
    return salary;
    }

    public LocalDate getHireDay(){
    return hireDay;
    }

    public String getDescription(){
    return String.format("an employee with a salary of
%.2f", salary);
    }

    public void rasieSalary(double byPercent){
    double rasie = salary * byPercent / 100;
    salary += raise;
    }
}
```

```
//Student 类, 继承自 Person
public class Student extends Person
{
    private String major;

    public Student(String name, String major){
    super(name);
    this.major = major;
    }

    public String getDescription(){
    return "a student major in " + major;
    }
}
```

```
//main 中调用
public static void main(String[] args){
  Person[] people = new Person[2];
  people[0] = new Employee("Harry Hacker", 50000, 1989, 10,
1);
  people[1] = new Student("Maria Morris", "computer
science");
  for(Person p : people)
    System.out.println(p.getName() + ", " +
p.getDesrciption());
}
```

6.2.2 抽象Number类

Number 类是教值包装类、Biglnteger 以及 BigDecimal 的抽象父类

包含方法有

byteValue()

shortValue()

intValue()

longValue()

floatValue()

doubleValue()

6.3 接口

接口和抽象类很相似，，但是它的目的是指明相关或者不相关类的多个对象的共同行为，**只包含常量和抽象方法。**

注意：

1.接口不是类，不能new运算符实例化一个接口

2.可以声明接口变量 `InterfaceName variable`

3.可以使用interface检查一个对象是否实现了某个特定的接口

4.接口也可以继承

```
public interface Moveable
{
  void move(double x, double y);
}
//可以以此为基础拓展接口
public interface Powered extends Moveable
{
    double milesPerGallon();
}
```

使用接口的原因，可以实现多重继承

6.3.1 接口例子

接口用关键字interface实现，接口实现使用 implement

```
//Comparable接口
public interface Comparable<T>
{
    int compareTo(T other);
}
```

```
//Employee 实现 ComparaTo
public class Employee implements Comparable<Employee>
{
    private String name;
    private double salary;

    public Employee(String name, double salary){
    this.name = name;
    this.salary = salary;
    }

    public String getName(){
    return name;
    }

    public double getSalary(){
    return salary;
    }

    public void rasieSalary(double byPercent){
    double raise = salary * byPercent / 100;
    salary += raise;
    }

    public int compareTo(Employee other)
    {
        return Double.compare(salary, other.salary);
    }
}
```

6.3.2 lambda 表达式

对于只有一个接口方法的接口，这种接口称为函数式接口，可以提供一个lambda表达式。

lambda 表达式语法

lambda表达式就是一个代码块，以及必须传入代码的变量规范

lambda表达式形式：

参数， 箭头（->）以及一个表达式，表达式可以放在{}中

```
(String first, String second)->
{
  if(first.length() < second.length())
    return -1;
  else if (first.length() > second.length())
    return 1;
  else return 0;
}
```

没有参数的情况下

```
 () ->{
  int i = 100;
  while(i){
    System.out.println(i);
    --i;
  }
}
```

函数式接口：

```
public interface IntConsumer
{
  void accept(int value);
}

public static void repeat(int n, IntConsumer action){
  for (int i = 0; i < n; i++)
    action.accept(i);
}

//可以这么调用
repeat(10, i->System.out.println("Contdown: " + (9 - i)));
```

6.3.3 抽象类与接口表达式的辨析

	变量	构造方法	方法
抽象类	无限制	子类通过构造方法链调用构造方法，抽象类不能用new实例化	无限制
接口	所有的变量必须是public static final	没有构造方法。接口不能用new实例化	所有方法必须是公有的抽象实例方法

类的扩展做单一继承，但是允许使用接口做多重扩展

7. 泛型与类安全

泛型编程提供编译时类型安全。

7.1 简单的泛型类和泛型方法

```
public class Pair<T>
{
    private T first;
    private T second;

    public Pair(){
        first = null;
        second = null;
    }

    public Pair(T first, T second){
        this.first = first;
        this.first = second;
    }

    public T getFirst(){
        return first;
    }

    public T getSecond(){
        return second;
    }

    public void setFirst(T newValue){
        first = newValue;
    }

    public void setSecond(T newValue){
        second = newValue;
    }
}
```

可以使用多个变量

```
public class Pair<T, U>{...}
//其中的T，U使用不同的类型
```

使用时：

```
public class PairTest1
{
  public static void main(String[] args){
    String[] words = {"Mary", "had", "a", "little", "lamb"};
    Pair<String> mm = ArrayAlg.minmax(words);
    System.out.println("min = " + mm.getFirst());
    System.out.println("max = " + mm.getSecond());
  }
}

class ArrayAlg
{
  public static Pair<String> minmax(String[] a){
    if (a == null || a.length == 0)
      return null;
    String min = a[0];
    String max = a[0];
    for(int i = 1;i < a.length; i++){
      if (min.compareTo(a[i]) > 0)
        min = a[i];
      if (max.compareTo(a[i]) < 0)
        max = a[i];
    }
    return new Pair<>(min, max);
  }
}
```

7.2 类型变量的限定

例如如下代码

```
class ArrayAlg
{
  public static <T> T min(T[] a){
    if (a == null || a.length == 0)
      return null;
    T smallest = a[0];
    for (int i = 1; i < a.length ; i++)
      if (smallest.compareTo(a[i]) > 0)
        smallest = a[i];
    return smallest;
  }
}
```

泛型可以传入任意类型，但是不是所有的类型都有compareTo方法，需要对类型T进行限定

```
public static <T extends Comparable> T min(T[] a) ...
```

现在泛型的min方法只能被实现了Comparable接口的类调用

多个限定用 & 分隔

```
<T extends Comparable & Serializable>
```

7.3 泛型注意的几点

1、不能用基本参数类型实例化类型参数

比如

上例中的

```
min<double>
min<int>
```

这样子是错误的

得使用

```
min<Double>
min<Integer>
```

2、运行是类型查询只适用于原始类型

比如

```
if (a instanceof Pair<String>)
if (a instanceof Pair<T>)
//这段程序是错误的
```

查询对象是否属于某个泛型类型时，用instanceof会报错

用getClass时，返回的原始类型

```
Pair<String> stringPair = ..;
Pair<Employee> employeePair = ....;
stringPair.getClass() == employeePair.getClass()// 他们是相同的
```

3、不能创建参数化类型数组

比如

```
Pair<String>[] arr = new Pair<String>[10];
//这段程序是错误的
```

7.4 通配符

通配符类型中，允许类型参数变化

例如

`Pair<? extends Employee>`

表示任何泛型Pair类型，它的参数类型是Employee的子类，如Pair<Manager> 但不是 Pair<STring>

例如

```
public static void printBuddies(Pair<Employee> p){
  Employee first = p.getFirst();
  Employee second = p.getSecond();
  System.out.prinln(first.getName() + " and " +
second.getName() + " are buddies.");
}
//这样子写的话，pair<Manager>不能传入这个方法
```

解决方法就是使用通配符

```
public static void printBuddies(Pair<? extend Employee> p)
  ...
```

7.4.1 通配符的限定

通配符限定与变量限定类似，还可以指定一个超类（父类）型限定

例子：

```
public static void minmaxBonus(Manager[] a, Pair<? super
Manager> result){
  if (a.length == 0)
    return;
  Manager min = a[0];
  Manager max = a[0];
  for(int i = 1; i < a.length ;i++){
    if(min.getBonus() > a[i].getBonus())
      mix = a[i];
    if(max.getBonus() < a[i].getBonus())
      max = a[i];
  }
  result.setFirst(min);
  result.setSecond(max);
}
```

这样子写可以接受任何适当的Pair

Pair<Employee> 甚至 Pair<Object>

甚至还能这么写

```
public static <T extends Comparable<? super T>> T min(T[] a)
{
  ....
}
```

8. 对象关系

8.1 关联

它使一个类知道另一个类的属性和方法。关联可以是双向的，也可以是单向的。在Java语言中，关联关系一般使用成员变量来实现

例子

单向关联：

```
//在Student中使用了Course类
public class Student{
  private Course[] courseList;
  public void addCourse(Course s){
     ....
  }
}
```

```
public class Course{
  private Student[] classList;
  private Faculty faculty;

  public void addStudent(Student s){
    ...
  }

  public void setFaculty(Faculty faculty){
    ...
  }
}
```

```
//在Faculty中使用了Course类
public class Faculty{
  private Course[] courseList;
  public void addCourse(Course c){
    ...
  }
}
```

双向关联：

```
public class Class{
  private Teacher[] teachList;
  ....
}
public class Teacher{
  private Class[] classList
}
```

自己关联：

```
public class Node{
  private Node subNode;
}
```

8.2 依赖

表示一个类依赖于另一个类的定义，一种非常弱、临时性的关系，体现为局域变量、方法的形参，或者对静态方法的调用

```
public class People{
  public void read(Book book){
    ....
  }

  public void eat(Food food){
    ....
  }

}
```

```
public class Food{
  ...
}
```

```
public class Book{
  ...
}
```

8.3 聚合

聚合是整体和个体之间的关系

聚合是动态聚集, 部分的实例化过程在整体外进行，然后通过某种方式注入给整体

而聚合的整体和部分之间在生命周期上没有什么必然的联系

例子:

```
public class School {
  private List<Student> students;
  ....
}
```

```
public class Student{
    ....
}
```

8.4 组合

比聚合关系更强，整体与部分的生命周期是一致，

组合关系中，整体与部分是不可分的，整体的生命周期结束也就意味着部分的生命周期结束

```
//聚集类
public class Student{
  private Name name;
  private Address address;
}
```

```
//被聚集类 1
public class Name{
  ...
}
```

```
// 被聚集类 2
public class Address{
  ...
}
```

9.原始类型装箱与拆箱

9.1 包装类

在Java中，为8种原始数据类型设计了8种包装类，用于方便使用

注意

1、包装类为final类型，无法继承

2、被包装的值为final类型，初始化后不能被修改

3、无构造函数

例如将int包装成Integer类，将double包装成Double，将char包装成Character类

包装类有：

`Integer`、`Short`、`Long`、`Byte`、`Float`、`Double`、`Boolean`、`Character`

使用

```
Integer intObject = new Integer(2);
Integer intObject = 2;
```

包装类方法Double 举例

方法名	返回值
Double(double value)	
Double(String s)	
byteValue()	byte
shortValue()	short
intValue()	int
longValue()	long
floatValue()	float
doubleValue()	double
compareTo(Double o)	int
toString()	String
valueOf(String s)	Double
valueOf(String s, int radix)	Double
parseDouble(String s)	double

方法名	返回值
parseDouble(String s, int radix)	double

9.2 装箱和拆箱

可以通过Integer类装箱

```
Integer intObject = new Integer(1000);
Integer intObject2 = 1000;
Integer.valueOf(1000);
```

通过intValue等拆箱

```
int n = intObject.intValue();
int n2 = intObject2;
```

```
String str = "1000";
int x = Integer.parseInt(str);//转化为int
float x2 = Integer.parseFloat(str)//转化为float
```

10. 枚举类

枚举类型通过enum关键字定义，是一种特殊类型，里面包含着实例

1、对象预先设定

2、构造器私有

3、枚举类父类为Enum<E>，其中就包含了toString, Valueof（），values（） 方法

4、比较两个枚举类，使用 ==

例子：

```
public enum Size
{
  SMALL("S"),MEDIUM("M"),LARGE("L"),EXTRA_LARGE("XL");

  private String abbr;
  public Size(String abbr){
    this.abbr = abbr;
  }
  public String getAbbreviation(){
    return abbr;
  }
}
```

```
public static void main(String[] args){
  Scanner in = new Scanner(System.in);
  System.out.println("Enter a size: (SAMLL, MEDIUM, LARGE,
EXTRA_LARGE)");
  String input = in.next().toUpperCase();
  Size size = Enum.ValueOf(Size.class, input);
  System.out.println("size = ", size);
  System.out.println("abbreviation = ",
size.getAbbreviation());
}
```

11.内部类与匿名类

11.1 内部类

定义在类中的类

1、处于代码结构考虑，内部类有较好的隐藏性

2、内部类既可以访问自身的数据域，又可以方法创建它的外围类对象的数据域

3、方便回调

例子:

```
public class TalkingClock
{
  private int interval;
  private boolean beep;

  public TalkingClock(int interval, boolean beep){
    this.interval = interval;
    this.beep = beep;
  }

  public void start(){
```

```
    ActionListener listener = new TimePrinter();
    Timer t = new Timer(interval, listener);
    t.start();
  }

  public class TimePrinter implements ActionListener
  {
    public void actionPerformed(Action event){
      System.out.println("the time is " + new Date());
      if (beep) Toolkit.getDefaultTookit().beep();
    }
  }
}
```

11.2 匿名类

假如只创建这个类的一个对象，不命名，这种类叫做匿名内部类。

匿名内部类一般还是用lambda表达式

例子：

```
//与上面Talking类似
class TalkingClock{
  ...
    public void start(int interval, boolean beep){
    ActionListener listener = new ActionListener(){
      public void actionPerformed(ActionEvent event){
        System.out.println("The time is " + new Date());
        if(beep) Toolkit.getDefaultTookit().beep();
      }
    };
    Timer t = new Time(interval, listener);
    t.start();
  }
```

```
}
```

但是一般的使用lambda表达式

```
public void start(int interval, boolean beep){
  Timer t = new Timer(interval, event ->{
    System.out.println("The time is " + new Date());
    if (beep) Tookit.getDefaultTookit().beep();
  });
  t.start();
}
```

12. 反射与动态编程

反射，用于编写动态操纵Java代码的程序

1、在运行时分析类

2、在运行时检查对象

3、利用class，Constructor，Field，Method等反射库中的类实现动态编程

反射机制主要提供了以下功能

1、运行时判断任意对象所属的类

2、运行时构造任意一个类的对象

3、运行时判断任意一个类所具有的成员变量和方法

4、运行时调用任意一个对象的方法

5、生成动态代理

12.1 获得Class实例

Class 类的一个实例表示java的一种数据类型，包括类，家口，枚举，注解，数组，基本数据类型和void，每一个都有一个class（）和getClass（）方法获得class实例

例子：

```
Class class1 = String.class;//获得String的Class
Integer num = 114;
CLass class2 = num.getClass();//获得对象的Class
```

一些API

方法	说明
getName()	返回类名
isInterface()	是否为接口
isArray()	是否为数组
getSuperclass()	返回父类
isPrimitive()	是否是基本类型

12.2 reflect

判断对于类内部的

```
Class class1 = Class.forName("java.lang.String");//动态加载类的运行时对象
Method[] methods = class1.getDeclaredMethods();//或许成员方法集合
for(Method method : method){
  System.out.print(Modifoer.toString(method.getModifiers()));//返回权限修饰符
  System.out.print(method.getReturnType().getName());// 返回返回值类型名称
  System.out.println(method.getName());//返回方法名
}
```

12.3 访问方法

12.3.1 访问构造方法

创建一个Constructor对象，然后利用Constructor对象的方法访问

```
Constructor[] contructors = 类名.getConstructors;
```

如果是访问指定的构造方法，需要根据该构造方法的入口参数的类型来访问

`getConstructors()`

`getConstructor(Class<?>…parameterTypes)`

`getDeclaredConstructors()`

`getDeclaredConstructor(Class<?>...parameterTypes)`

一些常用方法

方法	说明
isVarArgs()	查看该构造方法是否允许带可变数量的参数
getParameterTypes()	获取该构造方法各个参数的类型
getExceptionTypes()	取该构造方法可能抛出的异常类型
newInstance(object initargs)	通过该构造方法利用指定参数创建一个该类型的对象
setAccessiable(boolean flag)	如果该构造方法的权限为 private，默认为不允许通过反射利用 netlnstance()方法创建对象。如果先执行该方法，并将入口参数设置为 true，则允许创建对象
getModifier()	获得可以解析出该构造方法所采用修饰符的整数

Modifier类常用方法

方法	说明
isStatic(int mod)	是否为静态
isPublic(int mod)	是否为public
isProtected(int mod)	是否为protected
isPrivate(int mod)	是否为private
isFinal(int mod)	是否为final
toString	

12.4 访问成员变量

创建一个Field对象，然后利用Field对象的方法访问

```
Field[] objectFields = 类名.getField();
```

同样，可以指定返回的成员变量

`getFields()`

`getField(String name)`

`getDeclaredFields()`

`getDeclaredField(String name)`

常用方法

方法	说明
getName()	获得该成员变量的名称
getTpye()	获取表示该成员变量的 Class 对象
get(Object obj)	获得指定对象 obj 中成员变量的值，返回值为 Object 类型
set(Object obj, Object value)	将指定对象 obj 中成员变量的值设置为 value
getInt(Object obj, int i)	获得指定对象 obj 中成员类型为 int 的成员变量的值
setlnt(Object obj, int i)	将指定对象 obj 中成员变量的值设置为 i
setFloat(Object obj, float f)	将指定对象 obj 中成员变量的值设置为 f
getBoolean(Object obj)	获得指定对象 obj 中成员类型为 boolean 的成员变量的值
setBoolean(Object obj, boolean b)	将指定对象 obj 中成员变量的值设置为 b

方法	说明
getFloat(Object obj	获得指定对象 obj 中成员类型为 float 的成员变量的值
setAccessible(boolean flag)	此方法可以设置是否忽略权限直接访问 private 等私有权限的成员变量
getModifiers()	获得可以解析出该方法所采用修饰符的整数