一些小细节

运算符

关系运算符

测试对象等价

1. ==和equals

```
Integer n1 = 47;
Integer n2 = 47;
System.out.println(n1 == n2);
System.out.println(n1 != n2);
System.out.println(n1.equals(n2));
```

输出结果

```
true
false
true
```

但是：

```
Integer n1 = 128;
Integer n2 = 128;
System.out.println(n1 == n2);
System.out.println(n1 != n2);
System.out.println(n1.equals(n2));
```

输出结果

```
false
true
true
```

原因：因为 Integer 内部维护着一个 IntegerCache 的缓存，默认缓存范围是 [-128, 127]，所以 [-128, 127] 之间的值用 == 和 != 比较也能能到正确的结果；而equals方法比较的是对象的内容是否相同

逻辑运算符

1. &&、||、!三个逻辑运算符在Java中不适用于非布尔值（如int、float等）
2. Java只有当你使用 System.out.printf() 或 System.out.format() 时可以用 %n 作为换行符。对于 System.out.println()，我们仍然必须使用 \n；如果你使用 %n，println() 只会输出 %n 而不是换行符。
3. 如下代码会报错：

```
float f4 = 1e-43
```

因为1e-43在Java中会被解释为double，所以需要用如下代码修正：

```
float f4 = 1e-43f
```

移位运算符

1. 只能操作整型
2. <<左移，低位补0；>>右移，值正时高位补0，值负时高位补1；>>>右移，无论正负高位补0
3. char、byte和short会被提升为int，前面补0，重新赋值时会被截断只取低位
4. 移位long的结果还是long

字符串运算符

1. Java并未实现运算符重载
2. String开头的表达式，后续所有运算对象都必须是字符串

```
int x = 0, y = 1, z = 2;
String s = "x, y, z ";
System.out.println(s + x + y + z);
System.out.println(s + (x + y + z));
```

输出结果分别为

```
x, y, z 012
x, y, z 3
```

3. 当变量 x 和 y 都是布尔值，则 x=y 是一个逻辑表达式，意义和x==y相同

截断和舍入

1. 从 float 和 double 转换为整数值时，小数位将被截断。若你想对结果进行四舍五入，可以使用 java.lang.Math 的 round() 方法
2. 如果我们对小于 int 的基本数据类型（即 char、byte 或 short）执行任何算术或按位操作，这些值会在执行操作之前类型提升为 int
3. Java没有sizeof
4. Java会溢出，并且没有报错或者警告、异常

控制流

1. else if 并非新关键字，它仅是 else 后紧跟的一条新 if 语句
2. 推荐使用for-in语法
3. Java的break <label>;语句可以用来跳出多层循环
4. 注意如下语句，此处 Random.nextInt() 将产生 0~25 之间的一个随机 int 值，它将被加到 a 上。这表示 a 将自动被转换为 int 以执行加法。为了把 c 当作字符打印，必须将其转型为 char；否则，将会输出整数

```
int c = rand.nextInt(26) + 'a';
```

5. switch可以选择字符串
6. Math.random() 的结果集范围包含 0.0 ，不包含 1.0。 在数学术语中，可用 [0,1) 来表示

初始化和清理

1. Java可以通过参数列表的不同来区分两个相同命名的方法，甚至可以根据参数列表中的参数顺序来区分不同的方法
2. Java不能通过方法名和返回值区分方法
3. 类的设计者通过构造器保证每个对象的初始化
4. 如果你创建一个类，类中没有构造器，那么编译器就会自动为你创建一个无参构造器

```
class Bird {}
public class DefaultConstructor {
    public static void main(String[] args) {
        Bird bird = new Bird(); // 默认的
    }
}
```

但是一旦你显式地定义了构造器（无论有参还是无参），编译器就不会自动为你创建无参构造器：

```
class Bird2 {
    Bird2(int i) {}
    Bird2(double d) {}
}
public class NoSynthesis {
    public static void main(String[] args) {
        //- Bird2 b = new Bird2(); // No default
        Bird2 b2 = new Bird2(1);
        Bird2 b3 = new Bird2(1.0);
    }
}
```

此时调用new Bird2()会报错

5. **this** 关键字只能在非静态方法内部使用。当你调用一个对象的方法时，**this** 生成了一个对象引用。你可以像对待其他引用一样对待这个引用。如果你在一个类的方法里调用该类的其他方法，不要使用 **this**，直接调用即可，**this** 自动地应用于其他方法上了
6. **this** 关键字只用在一些必须显式使用当前对象引用的特殊场合。例如，用在 **return** 语句中返回对当前对象的引用。因此在相同的对象上可以轻易地执行多次操作，例如

```
x.increment().increment().increment();
```

7. 只能通过 **this** 调用一次构造器。另外，必须首先调用构造器，否则编译器会报错。

```
Flower(int petals) {
    petalCount = petals;
    System.out.println("Constructor w/ int arg only, petalCount = " +
petalCount);
}

Flower(String s, int petals) {
    this(petals);
    //- this(s); // Can't call two!
    this.s = s; // Another use of "this"
    System.out.println("String & int args");
}

Flower() {
    this("hi", 47);
    System.out.println("no-arg constructor");
```

```
}
```

8. static的含义
    1. **static** 方法中不会存在 **this**
    2. 不能在静态方法中调用非静态方法（反之可以）
    3. 静态方法是为类而创建的，不需要任何对象
9. 垃圾回收
    1. 在 C++ 中，对象总是被销毁的（在一个 bug-free 的程序中），而在 Java 中，对象并非总是被垃圾回收，或者换句话说：**对象可能不被垃圾回收；垃圾回收不等同于析构；垃圾回收只与内存有关！**
    2. 例如：对象在创建的过程中会将自己绘制到屏幕上。如果不是明确地从屏幕上将其擦除，它可能永远得不到清理。如果在 `finalize()` 方法中加入某种擦除功能，那么当垃圾回收发生时，`finalize()` 方法被调用（不保证一定会发生），图像就会被擦除，要是"垃圾回收"没有发生，图像则仍会保留下来。
    3. 在 Java 中，没有用于释放对象的 **delete**，因为垃圾回收器会帮助你释放存储空间，如果希望进行除释放存储空间之外的清理工作，还是得明确调用某个恰当的 Java 方法
    4. 记住，无论是"垃圾回收"还是"终结"，都不保证一定会发生。如果 Java 虚拟机（JVM）并未面临内存耗尽的情形，它可能不会浪费时间执行垃圾回收以恢复内存。
    5. `@Override`：`@` 意味着这是一个注解，注解是关于代码的额外信息。在这里，该注解告诉编译器这不是偶然地重定义在每个对象中都存在的 `finalize()` 方法——程序员知道自己在做什么。编译器确保你没有拼错方法名，而且确保那个方法存在于基类中。
10. 类的每个基本类型数据成员保证都会有一个初始值。下面的程序可以验证这类情况，并显示它们的值：

```java
public class InitialValues {
    boolean t;
    char c;
    byte b;
    short s;
    int i;
    long l;
    float f;
    double d;
    InitialValues reference;

    void printInitialValues() {
        System.out.println("Data type Initial value");
        System.out.println("boolean " + t);
        System.out.println("char[" + c + "]");
        System.out.println("byte " + b);
        System.out.println("short " + s);
        System.out.println("int " + i);
        System.out.println("long " + l);
        System.out.println("float " + f);
        System.out.println("double " + d);
        System.out.println("reference " + reference);
    }

    public static void main(String[] args) {
        new InitialValues().printInitialValues();
    }
}
```

输出是（char的值为0，所以显示空白；在类里定义一个对象引用时，如果不将其初始化，那么引用就会被赋值为 **null**）

```
Data type Initial value
boolean false
char[NUL]
byte 0
short 0
int 0
long 0
float 0.0
double 0.0
reference null
```

11. 参数不能是未初始化的类成员变量
12. 构造器初始化
    1. 自动初始化在构造器初始化之前执行
    2. 初始化顺序

```
class Window {
    Window(int marker) {
        System.out.println("Window(" + marker + ")");
    }
}

class House {
    Window w1 = new Window(1); // Before constructor

    House() {
        // Show that we're in the constructor:
        System.out.println("House()");
        w3 = new Window(33); // Reinitialize w3
    }

    Window w2 = new Window(2); // After constructor

    void f() {
        System.out.println("f()");
    }

    Window w3 = new Window(3); // At end
}

public class OrderOfInitialization {
    public static void main(String[] args) {
        House h = new House();
        h.f();
    }
}
```

输出：

```
Window(1)
Window(2)
Window(3)
House()
Window(33)
f()
```

13. 初始化过程：

    1. 初始化的顺序先是静态对象（如果它们之前没有被初始化的话），然后是非静态对象，从输出中可以看出。要执行 `main()` 方法，必须加载 **StaticInitialization** 类，它的静态属性 **table** 和 **cupboard** 随后被初始化，这会导致它们对应的类也被加载，而由于它们都包含静态的 **Bowl** 对象，所以 **Bowl** 类也会被加载。因此，在这个特殊的程序中，所有的类都会在 `main()` 方法之前被加载。

```java
class Bowl {
    Bowl(int marker) {
        System.out.println("Bowl(" + marker + ")");
    }

    void f1(int marker) {
        System.out.println("f1(" + marker + ")");
    }
}

class Table {
    static Bowl bowl1 = new Bowl(1);

    Table() {
        System.out.println("Table()");
        bowl2.f1(1);
    }

    void f2(int marker) {
        System.out.println("f2(" + marker + ")");
    }

    static Bowl bowl2 = new Bowl(2);
}

class Cupboard {
    Bowl bowl3 = new Bowl(3);
    static Bowl bowl4 = new Bowl(4);

    Cupboard() {
        System.out.println("Cupboard()");
        bowl4.f1(2);
    }

    void f3(int marker) {
        System.out.println("f3(" + marker + ")");
    }

    static Bowl bowl5 = new Bowl(5);
}

```

```
public class StaticInitialization {
    public static void main(String[] args) {
        System.out.println("main creating new Cupboard()");
        new Cupboard();
        System.out.println("main creating new Cupboard()");
        new Cupboard();
        table.f2(1);
        cupboard.f3(1);
    }

    static Table table = new Table();
    static Cupboard cupboard = new Cupboard();
}
```

输出：

```
Bowl(1)
Bowl(2)
Table()
f1(1)
Bowl(4)
Bowl(5)
Bowl(3)
Cupboard()
f1(2)
main creating new Cupboard()
Bowl(3)
Cupboard()
f1(2)
main creating new Cupboard()
Bowl(3)
Cupboard()
f1(2)
f2(1)
f3(1)
```

1. 假设有个名为 **Dog** 的类：
    1. 即使没有显式地使用 **static** 关键字，构造器实际上也是静态方法。所以，当首次创建 **Dog** 类型的对象或是首次访问 **Dog** 类的静态方法或属性时，Java 解释器必须在类路径中查找，以定位 **Dog.class**。
    2. 当加载完 **Dog.class** 后（后面会学到，这将创建一个 **Class** 对象），有关静态初始化的所有动作都会执行。因此，静态初始化只会在首次加载 **Class** 对象时初始化一次。
    3. 当用 `new Dog()` 创建对象时，首先会在堆上为 **Dog** 对象分配足够的存储空间。
    4. 分配的存储空间首先会被清零，即会将 **Dog** 对象中的所有基本类型数据设置为默认值（数字会被置为 0，布尔型和字符型也相同），引用被置为 **null**。
    5. 执行所有出现在字段定义处的初始化动作。
    6. 执行构造器。你将会在"复用"这一章看到，这可能会牵涉到很多动作，尤其当涉及继承的时候。
14. 继承的初始化

```
class Insect {
    private int i = 9;//在Insect()构造函数执行前执行显式初始化（因为在构造函数前面）
    protected int j;//执行默认初始化

```

```
    Insect() {
        System.out.println("i = " + i + ", j = " + j);
        j = 39;
    }

    private static int x1 = printInit("static Insect.x1 initialized");

    static int printInit(String s) {
        System.out.println(s);
        return 47;
    }
}

public class Beetle extends Insect {
    private int k = printInit("Beetle.k.initialized");//在Beetle()构造函数执行前，Insect基类初始化完成以后执行显式初始化

    public Beetle() {
        System.out.println("k = " + k);
        System.out.println("j = " + j);
    }

    private static int x2 = printInit("static Beetle.x2 initialized");

    public static void main(String[] args) {
        System.out.println("Beetle constructor");
        Beetle b = new Beetle();
    }
}
```

输出

```
static Insect.x1 initialized
static Beetle.x2 initialized
Beetle constructor
i = 9, j = 0
Beetle.k initialized
k = 47
j = 39
```

15. 静态变量

    1. 静态变量total在构造器里面需要指定Human.total++才可以对static的++，否则默认是this.total++，然后会报错

```
class Human {
    static int total;
    static{
        total = 2;//Adam and Eve
    }
    int age;
    boolean gender;
    Human(){ //default constructor
        Human.total++;
        age = 0;
        gender = false;
    }
```

```
    ...
}
```

16.
```
Object girlfriend;
{
    girlfriend = new Dog();
    ...
}
```

这样可以进行不止于new的复杂的初始化过程

17.
```
static void printArray(Object... args)
```

...代表可以有任意多的参数

```
public static void main(String[] args) {
    // Can take individual elements:
    printArray(47, (float) 3.14, 11.11);
    printArray("one", "two", "three");
    // Or an array:
    printArray((Object[]) new Integer[] {1, 2, 3, 4});
    printArray(); // Empty list is OK
}
```

输出结果:

```
47 3.14 11.11
one two three
1 2 3 4
```

18. 区分静态实例初始化和非静态实例初始化

```
class Mug {
    Mug(int marker) {
        System.out.println("Mug(" + marker + ")");
    }
}

public class Mugs {
    Mug mug1;
    Mug mug2;
    { // [1]
        mug1 = new Mug(1);
        mug2 = new Mug(2);
        System.out.println("mug1 & mug2 initialized");
    }

    Mugs() {
        System.out.println("Mugs()");
    }

    Mugs(int i) {
        System.out.println("Mugs(int)");
    }

```

```
    public static void main(String[] args) {
        System.out.println("Inside main()");
        new Mugs();
        System.out.println("new Mugs() completed");
        new Mugs(1);
        System.out.println("new Mugs(1) completed");
    }
}
```

输出结果

```
Inside main()
Mug(1)
Mug(2)
mug1 & mug2 initialized
Mugs()
new Mugs() completed
Mug(1)
Mug(2)
mug1 & mug2 initialized
Mugs(int)
new Mugs(1) completed
```

```

19. 数组的创建确实是在运行时进行的，不是编译器指定的

## 封装

1. 包名package：这个包必须位于包名指定的目录中，该目录必须在以 CLASSPATH 开始的目录中可以查询到。例如：package onjava语句指明本文件应该位于onjava目录下，其它java文件可以用import引用它

2. 使用导入，是为了提供一种管理命名空间的机制，管理、防止类名冲突

3. default访问权限：赋予成员默认包访问权限，不用加任何访问修饰符，然后将其他类放在相同的包内。这样，其他类就可以访问该成员

4. 当你定义一个具有包访问权限的类时，你可以在类中定义一个 public 构造器，编译器不会报错

   ```java
   package hiding.packageaccess;

   class PublicConstructor {
       public PublicConstructor() {}
   }
```

但是实际上你不能从包外访问到这个 **public** 构造器。编译会得到错误信息

```
enum Performance {
    GOOD, BAD
}
public class Human{
    Performance performance;
}
```
```

3. 子类可以使用父类的public和protected方法，不能使用private方法

复用

1. 组合、继承和委托
2. 强合成：部分的生命期不能比整体还要长（如：Human和Heart和Liver的关系）
3. 弱合成：
    1. 聚合：部分可独立存在（如：车和轮子、引擎的关系，他们都可以独立存在）
    2. 关联：对象可以向另一个对象通过某种方式发送消息（如：人和工作场所）
4. 继承：继承并不只是复制基类的接口。当你创建派生类的对象时，它包含基类的子对象。这个子对象与你自己创建基类的对象是一样的。只是从外部看，基类的子对象被包装在派生类的对象中。
    1. 如果没有无参数的基类构造函数，或者必须调用具有参数的基类构造函数，则必须使用 **super** 关键字和适当的参数列表显式地编写对基类构造函数的调用，并且对基类构造函数的调用必须是派生类构造函数中的第一个操作
5. 委托：有两个对象参与处理同一个请求，接受请求的对象将请求委托给另一个对象来处理

```
public class Car {
    public Window[] windows = new Window[4];
}

public class Tesla extends Car{
    public void openWindow(int i){
        this.windows[i].open(); //delegate
    }
}

Car hisCar = new Tesla();

//自动式玻璃
hisCar.openWindow(1);
```

把窗户的打开委托给Window的open()成员，而不是Tesla车自己打开

6. `final`关键字：
    1. 和static一起修饰变量：不会再改了，且可以通过类名访问
        1. 对基本类型，**final** 使数值恒定不变，而对于对象引用，**final** 使引用恒定不变，不能指向新的对象，但对指向的对象本身则可以修改
    2. `final`参数：在方法中不能改变参数指向的对象或基本变量
    3. 类中所有的 **private** 方法都隐式地指定为 **final**，但试图覆盖一个 **private** 方法（隐式是 **final** 的）时，看上去奏效，而且编译器不会给出错误信息：因为你没有覆写方法，实际上你创建了新的方法
    4. 父类成员函数`final`：该方法不能被子类重写
    5. `final`类：类不能被继承
7. 类的代码在首次使用时加载，准确地说，一个类当它任意一个 **static** 成员被访问时，就会被加载。
8. 在加载过程中，编译器注意到有一个基类，于是继续加载基类。不论是否创建了基类的对象，基类都会被加载。如果基类还存在自身的基类，那么第二个基类也将被加载，以此类推。
9. 根基类（例子中根基类是 **Insect**）的 **static** 的初始化开始执行，接着是派生类，以此类推。这点很重要，因为派生类中 **static** 的初始化可能依赖基类成员是否被正确地初始化。

多态

1. 多态是继承的产物

```
class Creature extends Being{
    public void eat(){  System.out.println("eating"); }
}
class Human extends Creature{
    @Override
    public void eat{ System.out.println("cooking...eating") }
}
class Woman extends Human {
    @Override
    public void eat{ System.out.println("cooking...photoing...eating");
}
}

Being you = new Woman();
you.eat(); //注意这里是虚函数，动态绑定，运行时根据虚函数表决定具体指向哪个对象
```

输出

```
cooking...photoing...eating
```

2. 多态形成的条件：继承、重写、父类引用指向子类对象——动态绑定
3. 陷阱：

```
public class PrivateOverride {
    private void f() {
        System.out.println("private f()");
    }

    public static void main(String[] args) {
        PrivateOverride po = new Derived();
        po.f();
    }
}

class Derived extends PrivateOverride {
    public void f() {
        System.out.println("public f()");
    }
}
```

输出结果：`private f()`，而不是`public f()`。如果使用`@Override`，那么编译器会输出报错

4. 陷阱：

```
class Super {
    public int field = 0;

    public int getField() {
        return field;
    }
}
```

```
8
9  class Sub extends Super {
10     public int field = 1;
11
12     @Override
13     public int getField() {
14         return field;
15     }
16
17     public int getSuperField() {
18         return super.field;
19     }
20 }
21
22 public class FieldAccess {
23     public static void main(String[] args) {
24         Super sup = new Sub(); // Upcast
25         System.out.println("sup.field = " + sup.field +
26                            ", sup.getField() = " + sup.getField());
27         Sub sub = new Sub();
28         System.out.println("sub.field = " + sub.field +
29                            ", sub.getField() = " + sub.getField()
30                            + ", sub.getSuperField() = " +
   sub.getSuperField())
31     }
32 }
```

输出结果

```
1 sup.field = 0, sup.getField() = 1
2 sub.field = 1, sub.getField() = 1, sub.getSuperField() = 0
```

5. static的话就不具有多态性
6. 构造器调用顺序：

> 1. 分配给对象的存储空间都被初始化为0
> 2. 基类构造器被调用。这个步骤被递归地重复，这样一来类层次的顶级父类会被最先构造，然后是它的派生类，以此类推，直到最底层的派生类。（如果父类构造器中方法被子类重写`@Override`，那么直接调用子类函数==>我们不应当调用类中的任何方法）
> 3. 按声明顺序初始化成员。
> 4. 调用派生类构造器的方法体。

7. 使用继承表达行为的差异，使用属性表达状态的变化。

接口

1. 某些行为不确定：抽象类

```
1 abstract class Human {
2     public abstract void meetLouisVuitton();
3 }
4 class Man extends Human {
5     @Override
6     public void meetLouisVuitton(){
7         pass();
8     }
```

```
}
class Woman extends Human {
    @Override
    public void meetLouisVuitton(){
        enter();
    }
}
```

2. 如果创建一个继承抽象类的新类并为之创建对象，那么就必须为基类的**所有**抽象方法提供方法定义。如果不这么做（可以选择不做），新类仍然是一个抽象类，编译器会强制我们为新类加上 **abstract** 关键字。
3. 所有行为不确定：接口

```
public interface Communicate{
    public String talkTo(String message);
}
public Man extends Human implements Communicate{
    ...
    public String talkTo(String message){
        return process(message);
    }
}
public Woman extends Human implements Communicate{
    ...
    public String talkTo(String message){
        return "我不听我不听我不听";
    }
}
```

4. Java 8 开始允许接口包含默认方法和静态方法
5. 和类一样，需要在关键字 **interface** 前加上 **public** 关键字（但只是在接口名与文件名相同的情况下），否则接口只有包访问权限，只能在接口相同的包下才能使用它。
6. 编译器用方法名、参数列表来区分方法，不通过返回类型区分方法
7. 为什么Java不支持多继承多个父类但支持实现多个接口?
8. 为什么接口中的成员变量必须是`public static final`?
    1. 因为必须满足公有化、标准化、规范化
9. 抽象类和接口

特性	接口	抽象类
组合	新类可以组合多个接口	只能继承单一抽象类
状态	不能包含属性（除了静态属性，不支持对象状态）	可以包含属性，非抽象方法可能引用这些属性
默认方法和抽象方法	不需要在子类中实现默认方法。默认方法可以引用其他接口的方法	必须在子类中实现抽象方法
构造器	没有构造器	可以有构造器
可见性	隐式 **public**	可以是 **protected** 或友元

内部类

1. 为什么用内部类？内部类是一种非常有用的特性，因为它允许你把一些逻辑相关的类组织在一起，并控制位于内部的类的可见性
    1. 如前所示，你实现了某类型的接口，于是可以创建并返回对其的引用。
    2. 你要解决一个复杂的问题，想创建一个类来辅助你的解决方案，但是又不希望这个类是公共可用的。
2. 一个内部类的对象能访问其外部对象的所有成员，而不需要任何特殊条件。此外，内部类还拥有其外部类的所有元素的访问权。
3. 使用.this和.new
    1. 如果你需要生成对外部类对象的引用，可以使用外部类的名字后面紧跟**.this**

```java
public class DotThis {
    void f() { System.out.println("DotThis.f()"); }

    public class Inner {
        public DotThis outer() {
            return DotThis.this;
            // A plain "this" would be Inner's "this"
        }
    }

    public Inner inner() { return new Inner(); }

    public static void main(String[] args) {
        DotThis dt = new DotThis();
        DotThis.Inner dti = dt.inner();
        dti.outer().f();
    }
}
```

    输出`DotThis.f()`

    2. 有时你可能想要告知某些其他对象，去创建其某个内部类的对象。要实现此目的，你必须在 **new** 表达式中提供对其他外部类对象的引用，这是需要使用 **.new** 语法

```java
public class DotNew {
    public class Inner {}
    public static void main(String[] args) {
        DotNew dn = new DotNew();
        DotNew.Inner dni = dn.new Inner();
    }
}
```

    3. 在拥有外部类对象之前是不可能创建内部类对象的。这是因为内部类对象会暗暗地连接到建它的外部类对象上。但是，如果你创建的是嵌套类（静态内部类），那么它就不需要对外部类对象的引用。
4. 一个定义在方法中的类：方法之外不能访问，但也不意味着方法执行完毕，类不可用
5. 一个定义在作用域内的类，此作用域在方法的内部。不意味着该类的创建有条件，而是已经与别的类一起编译过了
6. 一个实现了接口的匿名类。
7. 一个匿名类，它扩展了没有默认构造器的类。
8. 一个匿名类，它执行字段初始化。

9. 一个匿名类，它通过实例初始化实现构造（匿名内部类不可能有构造器）。
10. 如果不需要内部类对象与其外部类对象之间有联系，那么可以将内部类声明为 **static**，这通常称为嵌套类。
11. 普通的内部类对象隐式地保存了一个引用，指向创建它的外部类对象。然而，当内部类是 **static** 的时，就不是这样了。嵌套类意味着：
    1. 要创建嵌套类的对象，并不需要其外部类的对象。
    2. 不能从嵌套类的对象中访问非静态的外部类对象。
    3. 普通内部类的字段与方法，只能放在类的外部层次上，所以普通的内部类不能有 **static** 数据和 **static** 字段，也不能包含嵌套类。但是嵌套类可以包含所有这些东西：
12. 嵌套类可以作为接口的一部分。你放到接口中的任何类都自动地是 **public** 和 **static** 的

```java
public interface ClassInInterface {
    void howdy();
    class Test implements ClassInInterface {
        @Override
        public void howdy() {
            System.out.println("Howdy!");
        }
        public static void main(String[] args) {
            new Test().howdy();
        }
    }
}
```

    输出 `Howdy!`
13. 一个内部类被嵌套多少层并不重要——它能透明地访问所有它所嵌入的外部类的所有成员，即使是 private
14. **为什么需要内部类？每个内部类都能独立地继承自一个（接口的）实现，所以无论外部类是否已经继承了某个（接口的）实现，对于内部类都没有影响。**

SOLID原则

SRP 单一职责原则

1. 一个类仅有一个引起他变化的原因
2. 但是如果严格遵守这个原则，就会产生很多小类

OCP 开放封闭原则

1. 对扩展开放，对修改封闭
2. OCP是OO design的核心

LSP 里氏替换法则

1. 子类型必须可以替换它们的基类型。
2. 子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法。
3. 子类中可以增加自己特有的方法。
4. 当子类的方法重载父类的方法时，方法的前置条件（即方法的形参）要比父类方法的输入参数更宽松。
5. 当子类的方法实现父类的抽象方法时，方法的后置条件（即方法的返回值）要比父类更严格。

ISP 接口隔离原则

1. 不应强迫客户依赖于他们不使用的方法。
2. 客户程序应该仅依赖于它们实际调用的方法。
3. 方法：把胖类的接口分解为多个特定于客户程序的接口。
4. 目标：高内聚，低耦合

DIP 依赖倒置原则

1. 高层次模块不应依赖于低层次模块。两者都应该依赖于抽象。抽象不应该依赖细节。细节应该依靠抽象。
2. 任何变量都不应该持有一个指向具体类的指针或引用
3. 任何类都不应该从具体类派生
4. 任何方法都不应该覆写它的任何基类中的已经实现了的方法
5. 例外：可以依赖稳定的具体类，比如`String`

CARP 合成/聚合复用原则

1. 聚合表示“拥有”关系或者整体与部分的关系
2. 合成是一种强得多的“拥有”关系——部分和整体的生命周期是一样的。
3. 优点：
    1. 黑箱复用
    2. 每一个新的类可以将焦点集中在一个任务上
    3. 可以在运行时动态进行
4. 缺点：系统中会有较多的对象需要管理
5. 区分HAS-A 和 IS-A

LoD 信息隐藏

1. 创建有弱耦合的类，类之间的耦合越弱，就越有利于复用。
2. 一个类不应当public自己的属性，而应当提供取值和赋值的方法让外界间接访问自己的属性。
3. 在类的设计上，只要有可能，一个类应当设计成不变类。
4. 在对其它对象的引用上，一个类对其它对象的引用应该降到最低。

类加载和自省

1. 问题：需要知道某个泛化引用的具体类型——RTTI
2. `Class` 对象就是用来创建该类所有"常规"对象的。Java 使用 `Class` 对象来实现 RTTI，即便是类型转换这样的操作都是用 `Class` 对象实现的。每当我们编写并且编译了一个新类，就会产生一个 `Class` 对象（更恰当的说，是被保存在一个同名的二进制 `.class` 文件中）。为了生成这个类的对象，Java 虚拟机 (JVM) 先会调用"类加载器"子系统把这个类加载到内存中。
3. Java 程序在它开始运行之前并没有被完全加载，很多部分是在需要时才会加载。所有的类都是第一次使用时动态加载到 JVM 中的，当程序创建第一个对类的静态成员的引用时，就会加载这个类。
    1. 其实构造器也是类的静态方法，虽然构造器前面并没有 `static` 关键字。所以，使用 `new` 操作符创建类的新对象，这个操作也算作对类的静态成员引用。
4. 使用 `newInstance()` 来创建的类，必须带有无参数的构造器。通过Java 的反射API，可以用任意的构造器来动态地创建类的对象。
5. JVM
    1. JVM至少包括如下三个部分：
        1. 类加载系统（Class Loader Subsystem）

    2. 执行时数据区域（Runtime Data Area）
    3. 执行引擎（Execution Engine）
2. JVM Languages: Java、Clojure、Groovy、JRuby、Jython、Kotlin、Scala

6. 每一个类都持有其对应的`Class`类的对象的引用，其中包含着与类相关的信息，`Object`类中的`getClass()`能让我们获取到这个对象——RTTI，在不破坏抽象的原则下获得对象的实际类型。
7. Java虚拟机运行你的代码的第一步，就是先加载你的.class文件中的类型信息到内存中。
    1. 加载(Loading)。由**类加载器(Class Loaders)**执行，查找字节码，创建一个`Class`对象。
    2. 链接(Linking)。验证字节码，为静态域分配存储空间，如果必需的话，会解析这个类创建的对其他类的所有引用（比如说该类持有`static`域）。
    3. 初始化(Initializing)。如果该类有超类，则对其初始化，执行静态初始化器和静态初始化块。
8. Class Loader有哪些?
    1. Bootstrap Class Loader
    2. Extension class loader
        1. Java9以后Extension class loader改为Platform class loader。
    3. Application class loader
9. 请问： 输出中 $ 代表什么?

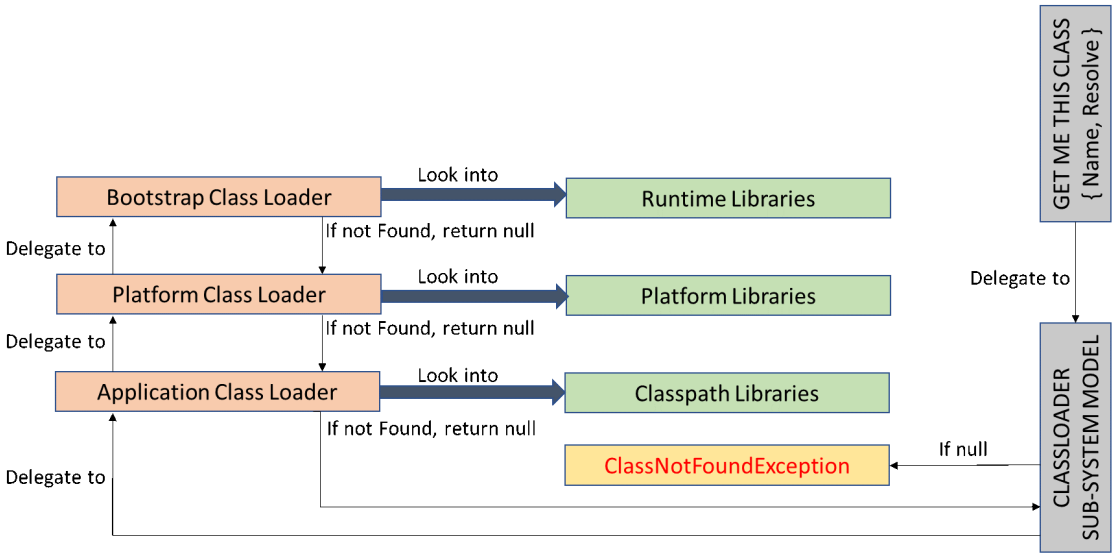
```
import java.util.ArrayList;
import java.sql.Array;
public class PrintClassLoader {
    public static void main(String[] ags)  {
        System.out.println("Classloader of this class:" +
PrintClassLoader.class.getClassLoader());
        System.out.println("Classloader of Array:" +
Array.class.getClassLoader());
        System.out.println("Classloader of ArrayList:" +
ArrayList.class.getClassLoader());
    }
}
```

输出

```
Classloader of this
class:jdk.internal.loader.ClassLoaders$AppClassLoader@55054057
Classloader of
Array:jdk.internal.loader.ClassLoaders$PlatformClassLoader@2626b418
Classloader of ArrayList:null
```

因为ArrayList是Bootstrap ClassLoader加载的，所以最后一个是`null`，同时`$`代表内部类

10. 双亲委派机制（Delegation Model）



11. If the class isn't already loaded, it delegates the request to the **parent class loader**. This process happens recursively.
12. If the parent class loader doesn't find the class, only then the child class will try to look for classes in the file system itself.
13. If unsucessful, it throws exception/error.
14. 什么时候JVM要求寻找一个class？当`new`一个对象的时候
15. 静态方法`Class.forName()`可以让我们对于某个类不进行创建就得到它的`Class`对象的引用
    1. 缺点：可能有异常，且会触发类的static子句（静态初始化块）
    2. 替换：类字面常量`Class c = Circle.class;`，支持编译时检查，所以不会抛出异常;不会触发类的static子句（静态初始化块），更简单更安全更高效
16. 类型检查：`instanceof`的形式包括`isInstance`和`instanceof`
    1. `public boolean isInstance(Object obj)`
        1. `Class.isInstance()`方法提供了一种动态测试对象类型的方法。
    2. `instanceof`
        1. `instanceof`有一个严格的限制：只可以将它与命名类型进行比较，而不能与`Class`对象作比较。
17. `isInstance()`和`instanceof`结果应该是一样的
18. 例如：

```
```java
public class FamilyVsExactType {
    static void test(Object x) {
        System.out.println(
              "Testing x of type " + x.getClass());
        System.out.println(
              "x instanceof Base " + (x instanceof Base));
        System.out.println(
              "x instanceof Derived " + (x instanceof Derived));
        System.out.println(
              "Base.isInstance(x) " + Base.class.isInstance(x));
        System.out.println(
              "Derived.isInstance(x) " +
              Derived.class.isInstance(x));
        System.out.println(
              "x.getClass() == Base.class " +
```
```

```
                (x.getClass() == Base.class));
        System.out.println(
                "x.getClass() == Derived.class " +
                (x.getClass() == Derived.class));
        System.out.println(
                "x.getClass().equals(Base.class)) "+
                (x.getClass().equals(Base.class)));
        System.out.println(
                "x.getClass().equals(Derived.class)) " +
                (x.getClass().equals(Derived.class)));
    }

    public static void main(String[] args) {
        test(new Base());
        test(new Derived());
    }
}
```

```
Testing x of type class typeinfo.Base
x instanceof Base true
x instanceof Derived false
Base.isInstance(x) true
Derived.isInstance(x) false
x.getClass() == Base.class true
x.getClass() == Derived.class false
x.getClass().equals(Base.class)) true
x.getClass().equals(Derived.class)) false
Testing x of type class typeinfo.Derived
x instanceof Base true
x instanceof Derived true
Base.isInstance(x) true
Derived.isInstance(x) true
x.getClass() == Base.class false
x.getClass() == Derived.class true
x.getClass().equals(Base.class)) false
x.getClass().equals(Derived.class)) true
```

令人放心的是，`instanceof` 和 `isInstance()` 产生的结果相同， `equals()` 和 `==` 产生的结果也相同。但测试本身得出了不同的结论。与类型的概念一致，`instanceof` 说的是“你是这个类，还是从这个类派生的类？”。而如果使用 `==` 比较实际的`Class` 对象，则与继承无关 —— 它要么是确切的类型，要么不是。

19. 泛化的`class`引用

    1. ```
       Class<Integer> genericIntClass = int.class;
       genericIntClass = Integer.class; // 同一个东西
       // genericIntClass = double.class; // 非法
       Class intClass = int.class;
       intClass = double.class;//合法
       ```

    2. 乍一看，下面的操作好像是可以的：

       ```
       Class<Number> geenericNumberClass = int.class;
       ```
       ```

这看起来似乎是起作用的，因为 `Integer` 继承自 `Number`。但事实却是不行，因为 `Integer` 的 `Class` 对象并不是 `Number` 的 `Class` 对象的子类（这看起来可能有点诡异，我们将在泛型这一章详细讨论）。

3.
```
Class<? extends Number> bounded = int.class;
bounded = double.class;
bounded = Number.class;
```

做范围限定

4.
```
Class<FancyToy> ftClass = FancyToy.class;
// Produces exact type:
FancyToy fancyToy = ftClass.newInstance();
Class<? super FancyToy> up = ftClass.getSuperclass();
// This won't compile:
// Class<Toy> up2 = ftClass.getSuperclass();
// Only produces Object:
```

super只能是Object类，不能是基类，因为具有含糊性

20. 自省：
    1. 问题：以上使用的RTTI都具有一个共同的限制：在编译时，编译器必须知道所有要通过RTTI来处理的类。但有的时候获取了一个对象引用，然而其对应的类并不在你的程序空间中（e.g. 网络、磁盘文件）
    2. 自省∈反射，反射除了检查，还可以控制改变
        1. 反射的核心是 JVM 在**运行时**才动态加载类或调用方法/访问属性，它不需要事先（写代码的时候或编译期）知道运行对象是谁
            1. 在运行时判断任意一个对象所属的类；
            2. 在运行时构造任意一个类的对象；
            3. 在运行时判断任意一个类所具有的成员变量和方法
            4. 在运行时调用任意一个对象的方法

异常

1. 当抛出异常后，有几件事会随之发生：首先，同 Java 中其他对象的创建一样，将使用 new 在堆上创建异常对象。然后，当前的执行路径（它不能继续下去了）被终止，并且从当前环境中弹出对异常对象的引用。此时，异常处理机制接管程序，并开始寻找一个异常处理程序继续执行程序。它的任务是将程序从错误状态中恢复，以使程序能要么换一种方式运行，要么继续运行下去。
2.
```
e.printStackTrace();
```

信息会被输出到标准错误流。

3. 异常声明：Java 提供了相应的语法（并强制使用这个语法），使你能以礼貌的方式告知客户端程序员某个方法可能会抛出的异常类型，然后客户端程序员就可以进行相应的处理。这就是异常说明，它属于方法声明的一部分，紧跟在形式参数列表之后。
4. `printStackTrace()`：

```
public class WhoCalled {
    static void f() {
// Generate an exception to fill in the stack trace
        try {
            throw new Exception();
        } catch(Exception e) {
```

```
            for(StackTraceElement ste : e.getStackTrace())
                System.out.println(ste.getMethodName());
        }
    }
    static void g() { f(); }
    static void h() { g(); }
    public static void main(String[] args) {
        f();
        System.out.println("*******");
        g();
        System.out.println("*******");
        h();
    }
}
```

输出：

```
f
main
*******
f
g
main
*******
f
g
h
main
```

一直从当前往main抛出异常。如果没有被捕获，那么就终止program

5. 重新抛出异常：会把异常抛给上一级环境中的异常处理程序，同一个 try 块的后续 catch 子句将被忽略。此外，异常对象的所有信息都得以保持，所以高一级环境中捕获此异常的处理程序可以从这个异常对象中得到所有信息。

```
catch(Exception e) {
    System.out.println("An exception was thrown");
    throw e;
}
```

    1. `printStackTrace()`：显示的是原来异常抛出点的调用栈信息，而非重新抛出点的信息
    2. `fillInStackTrace()`：返回一个`Throwable`对象，它是通过把当前调用栈信息填入原来那个异常对象而建立的。
6. 精准的重新抛出异常：在 Java 7 之前，如果捕捉到一个异常，重新抛出的异常类型只能与原异常完全相同。这导致代码不精确，Java 7修复了这个问题。可以不抛出BaseException，而抛出更具体的DerivedException
7. Catch有顺序：An error occurs if the super-type is arranged before the sub-type.如下代码，只会执行`SuperException`

```
try {
    throw new SubException();
} catch (SuperException superRef) { ...
} catch (SubException subRef) {
    ...// never be reached
} // an INVALID catch ordering
```

1. 当抛出异常时，异常处理系统会按照“最近的”处理程序的写入顺序查看它们。当它找到匹配项时，该异常被视为已处理，不再进行进一步的搜索。
2. 派生类对象将与基类的处理程序匹配。

8. 代码中只有 RuntimeException（及其子类）类型的异常可以被忽略，因为编译器强制要求处理所有受检查类型的异常。

```
public class NeverCaught {
    static void f() {
        throw new RuntimeException("From f()");
    }
    static void g() {
        f();
    }
    public static void main(String[] args) {
        g();
    }
}
```

这样，即使没有`try...catch`，照样会捕获异常，直接输出到`System.err`流

1. `RuntimeException`, `Error` and their subclasses are known as **unchecked exceptions**.
    1. 注意：Java的继承关系：Exception和Error都继承Throwable，然后Throwable继承Object
    - In most cases, unchecked exceptions reflect programming logic errors that are **not recoverable**.
        - For example: `NullPointerException`, `IndexOutOfBoundsException`

9. 使用`finally`进行清理，`finally`子句总能运行：
    1. 要把除内存之外的资源恢复到它们的初始状态时，就要用到 finally 子句。这种需要清理的资源包括：已经打开的文件或网络连接，在屏幕上画的图形，甚至可以是外部世界的某个开关
    2. 在异常没有被当前的异常处理程序捕获的情况下，异常处理机制也会在跳到更高一层的异常处理程序之前，执行 finally 子句：

```
class FourException extends Exception {}
public class AlwaysFinally {
    public static void main(String[] args) {
        System.out.println("Entering first try block");
        try {
            System.out.println("Entering second try block");
            try {
                throw new FourException();
            } finally {
                System.out.println("finally in 2nd try block");
            }
        } catch(FourException e) {
            System.out.println(
                    "Caught FourException in 1st try block");
```

```
        } finally {
            System.out.println("finally in 1st try block");
        }
    }
}
```

输出为

```
Entering first try block
Entering second try block
finally in 2nd try block
Caught FourException in 1st try block
finally in 1st try block
```

3. `return`之前会执行`finally`，保证重要的清理工作仍然执行

```
try {
    System.out.println("Point 1");
    if(i == 1) return;
} finally {
    System.out.println("Performing cleanup");
}
```

输出

```
Point 1
Performing cleanup
```

4. 丢失异常：

```
try {
    throw new RuntimeException();
} finally {
    // Using 'return' inside the finally block
    // will silence any thrown exception.
    return;
}
```

如果运行这个程序，就会看到即使方法里抛出了异常，它也不会产生任何输出。

10. 构造器

1. 在创建需要清理的对象之后，立即进入一个 try-finally 语句块：例如

```
NeedsCleanup nc1 = new NeedsCleanup();
try {
    // ...
} finally {
    nc1.dispose();
}
```

2. 构造器注意`finally`每一次都会执行清理代码，但是对象或许还没成功创建，所以不能用`finally`：

```java
try {
    in = new BufferedReader(new FileReader(fname));
    // Other code that might throw exceptions
} catch(FileNotFoundException e) {
    System.out.println("Could not open " + fname);
    // Wasn't open, so don't close it
    throw e;
} catch(Exception e) {
    // All other exceptions must close it
    try {
    in.close();
    } catch(IOException e2) {
        System.out.println("in.close() unsuccessful");
    }
    throw e; // Rethrow
} finally {
    // Don't close it here!!!
}
```

11. Try-With-Resources

    1. Java7引入。原来：

```java
try {
    in = new FileInputStream(
        new File("MessyExceptions.java"));
    int contents = in.read();
    // Process contents
} catch(IOException e) {
    // Handle the error
} finally {
    if(in != null) {
        try {
            in.close();
        } catch(IOException e) {
            // Handle the close() error
        }
    }
}
```

    现在：

```java
try(
    InputStream in = new FileInputStream(
        new File("TryWithResources.java"))
) {
    int contents = in.read();
    // Process contents
} catch(IOException e) {
    // Handle the error
}
```

    无论你如何退出 try 块（正常或通过异常），和 之前finally 子句等价的代码都会被执行，并且不用编写那些杂乱而棘手的代码

    2. 原理： try-with-resources 定义子句中创建的对象（在括号内）必须实现 `java.lang.AutoCloseable` 接口，这个接口只有一个方法：`close()`。

3. 退出`try`块会调用对象的 close() 方法，并以与创建顺序相反的顺序关闭。因为在这种情况下，Second 对象可能依赖于 First 对象，因此如果 First 在第 Second 关闭时已经关闭。Second 的 close() 方法可能会尝试访问 First 中不再可用的某些功能。

```
class Reporter implements AutoCloseable {
    String name = getClass().getSimpleName();
    Reporter() {
        System.out.println("Creating " + name);
    }
    public void close() {
        System.out.println("Closing " + name);
    }
}
class First extends Reporter {}
class Second extends Reporter {}
public class AutoCloseableDetails {
    public static void main(String[] args) {
        try(
                First f = new First();
                Second s = new Second()
        ) {
        }
    }
}
```

输出

```
Creating First
Creating Second
Closing Second
Closing First
```

4. 如果其中一个构造函数抛出异常怎么办?

```
// exceptions/ConstructorException.java
class CE extends Exception {}
class SecondExcept extends Reporter {
    SecondExcept() throws CE {
        super();
        throw new CE();
    }
}
public class ConstructorException {
    public static void main(String[] args) {
        try(
                First f = new First();
                SecondExcept s = new SecondExcept();
                Second s2 = new Second()
        ) {
            System.out.println("In body");
        } catch(CE e) {
            System.out.println("Caught: " + e);
        }
    }
}
```

输出为

```
Creating First
Creating SecondExcept
Closing First
Caught: CE
```

正如预期的那样，First 创建时没有发生意外，SecondExcept 在创建期间抛出异常。请注意，不会为 SecondExcept 调用 close()，因为如果构造函数失败，则无法假设你可以安全地对该对象执行任何操作，包括关闭它。由于 SecondExcept 的异常，Second 对象实例 s2 不会被创建，因此也不会有清除事件发生。

5. 但如果在`try`主体中抛出异常

```
class Third extends Reporter {}
public class BodyException {
    public static void main(String[] args) {
        try(
                First f = new First();
                Second s2 = new Second()
        ) {
            System.out.println("In body");
            Third t = new Third();
            new SecondExcept();
            System.out.println("End of body");
        } catch(CE e) {
            System.out.println("Caught: " + e);
        }
    }
}
```

输出为

```
Creating First
Creating Second
In body
Creating Third
Creating SecondExcept
Closing Second
Closing First
Caught: CE
```

注意Third对象永远不会被清除，因为它不是在资源规范头中创建的，所以它没有被保护。

6. 准则：如果可以，你应当始终使用 try-with-resources。这个特性有助于生成更简洁，更易于理解的代码。

12. 异常匹配：异常处理系统会按照代码的书写顺序找出“最近”的处理程序。找到匹配的处理程序之后，它就认为异常将得到处理，然后就不再继续查找。派生类的对象也可以匹配其基类的处理程序。
    1. 如果把捕获基类的 catch 子句放在最前面，以此想把派生类的异常全给“屏蔽”掉，编译器会发现派生类的catch永远不会执行而**报错**，所以应该把派生类异常放在基类前面

集合

1. ArrayList通过抛出*运行时异常*，所以不会不小心因数组越界而引发错误

1. 尖括号括起来的是*类型参数*（可能会有多个）
2. 当使用 **ArrayList** 的 `get()` 方法来取出你认为是 **Apple** 的对象时，得到的只是 **Object** 引用，必须将其转型为 **Apple**。然后需要将整个表达式用括号括起来，以便在调用 **Apple** 的 `id()` 方法之前，强制执行转型。否则，将会产生语法错误。

```
ArrayList apples = new ArrayList();
for(int i = 0; i < 3; i++)
    apples.add(new Apple());
// No problem adding an Orange to apples:
apples.add(new Orange());
for(Object apple : apples) {
    ((Apple) apple).id();
```

3. 菱形语法：在 Java 7 之前，必须要在两端都进行类型声明，如`ArrayList<Apple> apples = new ArrayList<Apple>();`，但是后来有了类型推断以后，就可以简化成`ArrayList<Apple> apples = new ArrayList<>()`

2. 类库的基本接口
    1. **集合（Collection）**：一个独立元素的序列，这些元素都服从一条或多条规则。**List** 必须以插入的顺序保存元素， **Set** 不能包含重复元素， **Queue** 按照*排队规则*来确定对象产生的顺序
    2. **映射（Map）**： 一组成对的“键值对”对象，允许使用键来查找值。 **ArrayList** 使用数字来查找对象。 **map** 允许我们使用一个对象来查找另一个对象，它也被称作*关联数组*（associative array）或者称作*字典*（dictionary），因为它将对象和其它对象关联在一起。
3. 向上转型：如`List<Apple> apples = new LinkedList<>();`。但**LinkedList** 具有 **List** 接口中未包含的额外方法，而 **TreeMap** 也具有在 **Map** 接口中未包含的方法。如果需要使用这些方法，就不能将它们向上转型为更通用的接口。
4. `Collections.addAll()`运行更快，很容易构建一个不包含元素的 **Collection**，`Arrays.asList()`输出List，底层实现是数组，没法调整大小，无法`add()`或`remove()`
5. 一些初步的注意点
    1. **ArrayList** 和 **LinkedList** 都是 **List** 的类型。两者之间的区别不仅在于执行某些类型的操作时的性能，而且 **LinkedList** 包含的操作多于 **ArrayList** 。
    2. **HashSet**， **TreeSet** 和 **LinkedHashSet** 是 **Set** 的类型。 **HashSet** 是检索元素的最快方法；**TreeSet** 将按比较结果的升序保存对象； **LinkedHashSet**，它按照被添加的先后顺序保存对象。
    3. **Map**（也称为*关联数组*）使用*键*来查找对象。键和值保存在 **HashMap** 中的顺序不是插入顺序，因为 **HashMap** 实现使用了非常快速的算法来控制顺序。 **TreeMap** 通过比较结果的升序来保存键， **LinkedHashMap** 在保持 **HashMap** 查找速度的同时按键的插入顺序保存键。
    4. 没必要使用 **Vector**, **Hashtable**, and **Stack**
6. `List`
    1. 对象的equals方法比较的是对象本身一不一样，String的equals方法是如果两个 **String** 的内容相同，则这两个 **String** 相等。因此，为了防止出现意外，请务必注意 **List** 行为会根据`equals()`行为而发生变化。
    2. 对于 **List**，有一个重载的`addAll()`方法可以将新列表插入到原始列表的中间位置，而不是仅能用 **Collection** 的`addAll()`方法将其追加到列表的末尾。
    3. 基本操作
        1. `add()`用于插入元素
        2. `get()`用于随机访问元素
        3. `iterator()`获取序列上的一个 **Iterator**
        4. `stream()`生成元素的一个 **Stream**

7. `ArrayList`
    1. 擅长随机访问元素，但在 **List** 中间插入和删除元素时速度较慢
    2. 是不是永远不要在`ArrayList`中插入和删除元素？不，应该看看你的 **List** 实现（发现此类瓶颈的最佳方式是使用分析器 profiler）
    3. Vector与ArrayList一样，也是通过数组实现的，不同的是它支持线程的同步，即某一时刻只有一个线程能够写Vector，避免多线程同时写而引起的不一致性，但实现同步需要很高的花费，因此，访问它比访问ArrayList慢。
    4. vector是线程（Thread）同步（Synchronized）的，所以它也是线程安全的，而Arraylist是线程异步（Asynchronized）的，是不安全的。如果不考虑到线程的安全因素，一般用Arraylist效率比较高。
    5. 如果集合中的元素的数目大于目前集合数组的长度时，vector增长率为目前数组长度的100%,而arraylist增长率为目前数组长度的50%。如过在集合中使用数据量比较大的数据，用vector有一定的优势。
    6. If a thread-safe implementation is not needed, it is recommended to use `ArrayList` in place of Vector.但实际上即使是并发编程，我们也不推荐使用`vector`，而使用`java.util.concurrency`包中的东西
8. `LinkedList`
    1. 底层是双向链表
    2. 通过代价较低的在 **List** 中间进行的插入和删除操作，提供了优化的顺序访问。 **LinkedList** 对于随机访问来说相对较慢，但它具有比 **ArrayList** 更大的特征集
    3. `getFirst()` 和 `element()` 是相同的，它们都返回列表的头部（第一个元素）而并不删除它，如果 **List** 为空，则抛出 **NoSuchElementException** 异常。 `peek()` 方法与这两个方法只是稍有差异，它在列表为空时返回 **null** 。
    4. `removeFirst()` 和 `remove()` 也是相同的，它们删除并返回列表的头部元素，并在列表为空时抛出 **NoSuchElementException** 异常。 `poll()` 稍有差异，它在列表为空时返回 **null** 。
    5. `addFirst()` 在列表的开头插入一个元素。
    6. `offer()` 与 `add()` 和 `addLast()` 相同。 它们都在列表的尾部（末尾）添加一个元素。
    7. `removeLast()` 删除并返回列表的最后一个元素。
    8. 队列和堆栈的行为是通过 **LinkedList** 提供的。
9. `iterator`迭代器
    1. 通常被称为“轻量级对象”，创建的代价很小
    2. 只能单向移动
    3. 只能
        1. 使用 `iterator()` 方法要求集合返回一个 **Iterator**。 **Iterator** 将准备好返回序列中的第一个元素。
        2. 使用 `next()` 方法获得序列中的下一个元素。
        3. 使用 `hasNext()` 方法检查序列中是否还有元素。
        4. 使用 `remove()` 方法将迭代器最近返回的那个元素删除。
    4. 若只想遍历，而不想修改List，那么用for-in语法更简洁
    5. `display()`不包含任何有关它所遍历的序列的类型信息，可以让迭代器遍历完整个序列
    6.
```java
public interface Iterator<E>{
  boolean hasNext();
  E next();
  void remove(); //optional
}
```

10. `ListIterator`： **Iterator** 只能向前移动，而 **ListIterator** 可以双向移动。`previous()`，`hasPrevious()`方法
11. `Stack`：Java 1.0 中附带了一个 **Stack** 类，结果设计得很糟糕（为了向后兼容，我们永远坚持 Java 中的旧设计错误）。Java 6 添加了 **ArrayDeque**，其中包含直接实现堆栈功能的方法
12. `Set`：查找通常是 **Set** 最重要的操作，因此通常会选择 **HashSet** 实现，该实现针对快速查找进行了优化
    1. **TreeSet** 将元素存储在红-黑树数据结构中，对元素进行排序
    2. **HashSet** 使用散列函数，用哈希表
    3. **LinkedHashSet** 因为查询速度的原因也使用了散列，但是使用了链表来维护元素的插入顺序
    4.

Set 类型	约束
Set(interface)	添加到 **Set** 中的每个元素必须是唯一的，否则，**Set** 不会添加重复元素。添加到 **Set** 的元素必须至少定义 `equals()` 方法以建立对象唯一性。 **Set** 与 **Collection** 具有完全相同的接口。 **Set** 接口不保证它将以任何特定顺序维护其元素。
HashSet*	注重快速查找元素的集合，其中元素必须定义 `hashCode()` 和 `equals()` 方法。
TreeSet	由树支持的有序 **Set**。这样，就可以从 **Set** 中获取有序序列，其中元素必须实现 **Comparable** 接口。
LinkedHashSet	具有 **HashSet** 的查找速度，但在内部使用链表维护元素的插入顺序。因此，当在遍历 **Set** 时，结果将按元素的插入顺序显示。元素必须定义 `hashCode()` 和 `equals()` 方法。

13. `HashSet`：
    1. 早期 Java 版本中的 **HashSet** 产生的输出没有可辨别的顺序。这是因为出于对速度的追求，**HashSet** 使用了散列
    2. **HashSet** 提供最快的查询速度
14. `SortedSet`：元素保证按排序规则顺序， **SortedSet** 接口中的以下方法可以产生其他功能：
    1. `Comparator comparator()`：生成用于此 **Set** 的**Comparator** 或 **null** 来用于自然排序。
    2. `Object first()`：返回第一个元素。
    3. `Object last()`：返回最后一个元素。
    4. `SortedSet subSet(fromElement, toElement)`：使用 **fromElement**（包含）和 **toElement**（不包括）中的元素生成此 **Set** 的一个视图。
    5. `SortedSet headSet(toElement)`：使用顺序在 **toElement** 之前的元素生成此 **Set** 的一个视图。
    6. `SortedSet tailSet(fromElement)`：使用顺序在 **fromElement** 之后（包含 **fromElement**）的元素生成此 **Set** 的一个视图。
15. `Map`：
    1. 可以将集合组合起来以快速生成强大的数据结构。例如，假设你正在追踪有多个宠物的人，只需要一个 **Map<Person, List>** 即可
    2. **Map** 可以返回由其键组成的 **Set**，由其值组成的 **Collection**，或者其键值对的 **Set**
    3. `hashCode()` 是根类 **Object** 中的一个方法，因此所有 Java 对象都可以生成哈希码。**HashMap** 获取对象的 `hashCode()` 并使用它来快速搜索键。
    4. `HashMap`查询、插入、删除高效
    5. `TreeMap`排序、按序遍历高效

6. `LinkedHashMap` JDK 1.4以后才有，扩展Hashmap，用链表满足顺序访问map的entries
7.

Map 实现	描述
HashMap*	基于哈希表的实现。（使用此类来代替 **Hashtable** 。）为插入和定位键值对提供了常数时间性能。可以通过构造方法调整性能，这些构造方法允许你设置哈希表的容量和装填因子。
LinkedHashMap	与 **HashMap** 类似，但是当遍历时，可以按插入顺序或最近最少使用（LRU）顺序获取键值对（可以轻松创建一个能够定期清理以节省空间的程序）。只比 **HashMap** 略慢，一个例外是在迭代时，由于其使用链表维护内部顺序，所以会更快些。
TreeMap	基于红黑树的实现。当查看键或键值对时，它们按排序顺序（由 **Comparable** 或 **Comparator** 确定）。 **TreeMap** 的侧重点是按排序顺序获得结果。 **TreeMap** 是唯一使用 `subMap()` 方法的 **Map** ，它返回红黑树的一部分。
WeakHashMap	一种具有 *弱键* （weak keys） 的 **Map** ，为了解决某些类型的问题，它允许释放 **Map** 所引用的对象。如果在 **Map** 外没有对特定键的引用，则可以对该键进行垃圾回收。
ConcurrentHashMap	不使用同步锁定的线程安全 **Map** 。这在第二十四章 并发编程 一章中讨论。
IdentityHashMap	使用 `==` 而不是 `equals()` 来比较键。仅用于解决特殊问题，不适用于一般用途。

16. `SortedMap`：（由 **TreeMap** 或 **ConcurrentSkipListMap** 实现），键保证按排序顺序，这允许在 **SortedMap** 接口中使用这些方法来提供其他功能：
    1. `Comparator comparator()` ：生成用于此 **Map** 的比较器， **null** 表示自然排序。
    2. `T firstKey()` ：返回第一个键。
    3. `T lastKey()` ：返回最后一个键。
    4. `SortedMap subMap(fromKey, toKey)` ：生成此 **Map** 的视图，其中键从 **fromKey**（包括），到 **toKey** （不包括）。
    5. `SortedMap headMap(toKey)` ：使用小于 **toKey** 的键生成此 **Map** 的视图。
    6. `SortedMap tailMap(fromKey)` ：使用大于或等于 **fromKey** 的键生成此 **Map** 的视图。
17. `Queue`：
    1. 并发编程中特别重要
    2. **LinkedList** 实现了 **Queue** 接口，并且提供了一些方法以支持队列行为，因此 **LinkedList** 可以用作 **Queue** 的一种实现。如下代码是合法的：

    ```
    1 Queue<Integer> queue = new LinkedList<>();
    ```

    3. `offer()` 是与 **Queue** 相关的方法之一，它在允许的情况下，在队列的尾部插入一个元素，或者返回 **false** 。
    4. `peek()` 和 `element()` 都返回队头元素而不删除它，但是如果队列为空，则 `element()` 抛出 **NoSuchElementException** ，而 `peek()` 返回 **null** 。
    5. `poll()` 和 `remove()` * 都删除并返回队头元素，但如果队列为空，`poll()` 返回 **null** ，而 `remove()` 抛出 **NoSuchElementException** 。

18. `PriorityQueue`
    1. **PriorityQueue** 确保在调用 `peek()` ， `poll()` 或 `remove()` 方法时，获得的元素将是队列中优先级最高的元素。
    2. 允许重复
19. `Collection`
    1. 如果实现了 **Collection** ，就必须实现 `iterator()` 和 `size()` 方法
    2. `Collections.shuffle()` 方法不会影响到原始数组，而只是打乱了 **shuffled** 中的引用。之所以这样，是因为 `randomized()` 方法用一个 **ArrayList** 将 `Arrays.asList()` 的结果包装了起来。如果这个由 `Arrays.asList()` 生成的 **List** 被直接打乱，那么它将修改底层数组

```
import java.util.*;

public class ModifyingArraysAsList {
  public static void main(String[] args) {
    Random rand = new Random(47);
    Integer[] ia = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    List<Integer> list1 =
      new ArrayList<>(Arrays.asList(ia));
    System.out.println("Before shuffling: " + list1);
    Collections.shuffle(list1, rand);
    System.out.println("After shuffling: " + list1);
    System.out.println("array: " + Arrays.toString(ia));

    List<Integer> list2 = Arrays.asList(ia);
    System.out.println("Before shuffling: " + list2);
    Collections.shuffle(list2, rand);
    System.out.println("After shuffling: " + list2);
    System.out.println("array: " + Arrays.toString(ia));
  }
}
/* Output:
Before shuffling: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
After shuffling: [4, 6, 3, 1, 8, 7, 2, 5, 10, 9]
array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Before shuffling: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
After shuffling: [9, 1, 6, 3, 7, 2, 5, 10, 4, 8]
array: [9, 1, 6, 3, 7, 2, 5, 10, 4, 8]
*/
```

20. `Iterable` 和 `for-in`
    1. *for-in*是所有 **Collection** 对象的特征，因为 Java 5 引入了一个名为 **Iterable** 的接口，该接口包含一个能够生成 **Iterator** 的 `iterator()` 方法。*for-in* 使用此 **Iterable** 接口来遍历序列。因此，如果创建了任何实现了 **Iterable** 的类，都可以将它用于 *for-in* 语句中
    2. 许多类都是 **Iterable** ，主要包括所有的 **Collection** 类（但不包括各种 **Maps** ）
    3. *for-in* 语句适用于数组或其它任何 **Iterable** ，但这并不意味着数组肯定也是个 **Iterable**
    4. 可以设计反向迭代器

```
//n=4
i=2 n%i==0 flag=False
i=3 n%i!=0 flag=True
flag==True
print('No')
```

泛型

1. Generic Programming
2. 使用*类型参数*，用尖括号括住，放在类名后面。然后在使用这个类时，再用实际的类型替换此类型参数。

```
public class Position<T extends Creature>{
    private T holder;
}
```

3. 在代码中避免（**casts**）的一种方法，将运行时错误（通常是**ClassCastException**）转换为编译时错误。
4. 有了泛型，你可以很容易地创建元组，令其返回一组任意类型的对象。

```
public class Tuple2<A, B> {
    public final A a1;
    public final B a2;
    public Tuple2(A a, B b) { a1 = a; a2 = b; }
    public String rep() { return a1 + ", " + a2; }

    @Override
    public String toString() {
        return "(" + rep() + ")";
    }
}
```

5. 泛型方法：

```
public class GenericMethods {
    public <T> void f(T x) {
        System.out.println(x.getClass().getName());
    }

    public static void main(String[] args) {
        GenericMethods gm = new GenericMethods();
        gm.f("");
        gm.f(1);
        gm.f(1.0);
        gm.f(1.0F);
        gm.f('c');
        gm.f(gm);
    }
}
```

对于泛型类，必须在实例化该类时指定类型参数。使用泛型方法时，通常不需要指定参数类型，因为编译器会找出这些类型。 这称为 *类型参数推断*。因此，对 `f()` 的调用看起来像普通的方法调用，并且 `f()` 看起来像被重载了无数次一样。它甚至会接受 **GenericMethods** 类型的参数。

6. 变长参数与泛型方法：

```java
@SafeVarargs
public static <T> List<T> makeList(T... args) {
    List<T> result = new ArrayList<>();
    for (T item : args)
        result.add(item);
    return result;
}
```

注意`SafeVarargs`注解保证我们不会对变长参数列表进行任何修改，这是正确的，因为我们只从中读取。如果没有此注解，编译器将无法知道这些并会发出警告。

7. 编译时刻和运行时类型信息（这里基本类型无法作为类型参数。不过 Java 5 具备自动装箱和拆箱的功能，可以很方便地在基本类型和相应的包装类之间进行转换。）

```java
public class Holder<T> {
    private T obj;
    public void set(T obj){ this.obj = obj; }
    public T get(){ return obj; }
    public static void main(String[] args){
        Holder<Integer> holder = new Holder<>();
        holder.set(1);//自动装箱，让int型的1变成Integer的1
        //holder.set("Abc"); // error
        Integer obj = holder.get(); //无须cast
    }
}
```

即运行时并不知道holder中放的是整型，类型信息被擦掉了，只有编译时知道。用`javap -v -p -s -sysinfo -constants Holder.class`编译，查看`set`和`get`相关的内容：

```
public void set(T);
    descriptor: (Ljava/lang/Object;)V
...
public T get();
    descriptor: ()Ljava/lang/Object;
```

`set`传入的和`get`返回的都是Object类型，而不是T类型。

8. 擦除：Java泛型的实现方式就是将类型参数用边界类型替换，在上面的例子中就是把T用Disk替换。这种实现方式看上去就像是**把具体的类型**（某种硬盘，机械的或者是固态的），**擦除到了边界类型**（它们的父类Disk）。如果用了`extends`，

```java
public class Computer<T extends Disk>{
    private T disk;    // 运行时disk是Disk类型
    Computer(T disk){
        disk = disk;
    }
}
```

运行时就是Disk类型了

1. 区别：5的只能确定T一定是Object，6只能确定T一定是Disk，这就是擦除，且只会擦除到边界类型
2. 再例如：Java 泛型是使用擦除实现的。这意味着当你在使用泛型时，任何具体的类型信息都被擦除了，你唯一知道的就是你在使用一个对象。因此，`List<String>`和`List<Integer>`

在运行时实际上是相同的类型。它们都被擦除成原生类型 `List`。String和Integer在代码内部，只是用作参数占位符的标识符而已

9. 泛型类型只有在静态类型检测期间才出现，在此之后，程序中的所有泛型类型都将被擦除，替换为它们的非泛型上界。例如， `List<T>` 这样的类型注解会被擦除为 **List**，普通的类型变量在未指定边界的情况下会被擦除为 **Object**。

    1. 这两个在编译器编译以后是一样的

```
ArrayList<Integer> intList = new ArrayList<>();
ArrayList rawList = new ArrayList();

System.out.println(intList.getClass().getSimpleName());
System.out.println(rawList.getClass().getSimpleName());
```

    2. 以下会出现编译错误

```
public class Holder<T> {
    private T obj;
    public void set(T obj){ this.obj = obj; }
    public T get(){ return obj; }
    public void testT(Object arg){
        if (arg instanceof T){ ... } //编译错误
        T var = new T(); //编译错误
        T[] array = new T[100]; //编译错误
        }
    }
}
```

    泛型不能用于显式地引用运行时类型的操作中，例如转型、**instanceof** 操作和 **new** 表达式。因为所有关于参数的类型信息都丢失了，你只是看起来拥有参数的类型信息而已。

10. T存在的意义：

```
public class Holder<T> {
    private T obj; //在编译时，该类中的所有的T都会被替换为边界类型Object。
    public void set(T obj){ this.obj = obj; }
    public T get(){ return obj; }
    public static void main(String[] args){
        Holder<Integer> holder = new Holder<>();
        //编译器会检查实参是不是一个Integer，虽然这里的1是int类型，
        //但是因为自动包装机制的存在，他会被转化为一个Integer，因此能够通过类型检查。
        holder.set(1);
        //编译器也会进行类型检查，并且自动插入一个Object类型到Integer类型的转型操作。
        Integer obj = holder.get();
    }
}
```

11. 对泛型的处理全部集中在编译期，在编译时，编译器会执行如下操作。

    - 会将泛型类的类型参数都用*边界类型*替换。
    - 对于传入对象给方法形参的指令，编译器会执行一个类型检查，看传入的对象是不是类型参数所指定的类型。
    - 对于返回类型参数表示对象的指令，也会执行一个类型检查，还会插入一个自动的向下转型，将对象从边界类型向下转型到类型参数所表示的类型。

12. 如果真的想生成泛型对象？利用设计模式的工厂模式，或者使用RTTI

13. 泛型继承先写类型再写接口（因为Java不允许多继承，可以多接口但不能多类型）

协变与逆变

1. 协变与逆变：例子，**每年都考**

```
class Fruit{}
class Apple extends Fruit{}

public class NonConvariantGeneric {
    List<Fruit> flist = new ArrayList<Apple>(); //编译错误
}
```

因为如下理解：

```
//现在我定义一个“水果盘子”，逻辑上水果盘子当然可以装苹果。
//但实际上Java编译器不允许这个操作。会报错，“装苹果的盘子”无法转换成“装水果的盘子”。
Plate<Fruit> p=new Plate<Apple>(new Apple()); //编译错误!
```

但是这样就可以

```
//Plate<? extends Fruit>和Plate<Apple>最大的区别就是：
//Plate<? extends Fruit>是Plate<Fruit>以及Plate<Apple>的基类。
Plate<? extends Fruit> p=new Plate<Apple>(new Apple());
// a list of any type that's inherited from Fruit
List<? extends Fruit> flist = new ArrayList<Apple>();
```

但是

```
Plate<? extends Fruit> p=new Plate<Apple>(new Apple());
//不能存入任何元素
p.set(new Fruit());    //Error，因为 set() 的参数也是"? extends Fruit"，意味着它可以是任何事物，编译器无法验证“任何事物”的类型安全性。
p.set(new Apple());    //Error
//读取出来的东西只能存放在Fruit或它的基类里。
Fruit newFruit1=p.get();
Object newFruit2=p.get();
Apple newFruit3=p.get();    //Error
```

虽然看起来p指向`new Plate<Apple>(new Apple())`，但是执行到`p.set()`的时候，编译器并不知道p实际上指向什么（例如可能不指向`Apple`，而指向`GreenApple`，这时候只能存入`GreenApple`，而不能存入其它），为了安全起见，因此不能存入任何元素：即不能确定放入`new Fruit()`或者`new Apple()`是不是安全的，因此会报错。同时，get出来的东西我们肯定可以确定至少是一个`Fruit`或者它的基类，不能是`Fruit`的子类

但是

```
Plate<? super Fruit> p=new Plate<Fruit>(new Fruit());
//存入元素正常
p.set(new Fruit());
p.set(new Apple());
//读取出来的东西只能存放在Object类里。
Apple newFruit3=p.get();    //Error
Fruit newFruit1=p.get();    //Error
Object newFruit2=p.get();
```

第6、7行：因为get出来的只能是`Fruit`的父类，所以可能是`Food`，甚至是`Object`，所以不能get出来`Apple`，`Fruit`，或者`Food`，只能是`Object`，最基本的类型。

1. 通配符`<? extends Fruit>`
    1. ```
       List<? extends Fruit> flist = new ArrayList<>();
       // Compile Error: can't add any type of object:
       // flist.add(new Apple());
       // flist.add(new Fruit());
       // flist.add(new Object());
       flist.add(null); // Legal but uninteresting
       // We know it returns at least Fruit:
       Fruit f = flist.get(0);
       ```
    2. ```
       List<? extends Fruit> flist = Arrays.asList(new Apple());
       Apple a = (Apple) flist.get(0); // No warning
       flist.contains(new Apple()); // Argument is 'Object'
       flist.indexOf(new Apple()); // Argument is 'Object'
       ```
    3. 可以看到，尽管`add()`接受一个泛型参数类型的参数，但`contains()`和`indexOf()`接受的参数类型是**Object**。因此当你指定一个`ArrayList<? extends Fruit>`时，`add()`的参数就变成了"**? extends Fruit**"。从这个描述中，编译器无法得知这里需要**Fruit**的哪个具体子类型，因此它不会接受任何类型的**Fruit**。如果你先把**Apple**向上转型为**Fruit**，也没有关系——编译器仅仅会拒绝调用像`add()`这样参数列表中涉及通配符的方法。

       `contains()`和`indexOf()`的参数类型是**Object**，不涉及通配符，所以编译器允许调用它们。这意味着将由泛型类的设计者来决定哪些调用是“安全的”，并使用**Object**类作为它们的参数类型。为了禁止对类型中使用了通配符的方法调用，需要在参数列表中使用类型参数。
2. 区别：`List`、`List<?>`和`List<? extends Object>`？
    1. 编译器很少关心使用的是原生类型还是`<?>`。在这些情况中，`<?>`可以被认为是一种装饰，但是它仍旧是很有价值的，因为，实际上它是在声明：“我是想用Java的泛型来编写这段代码，我在这里并不是要用原生类型，但是在当前这种情况下，泛型参数可以持有任何类型。”即：`List<?>`看起来等价于`List<Object>`，而**List**实际上也是`List<Object>`
        1. 但是：原生**List**将持有任何类型的组合，而`List<?>`将持有具有某种具体类型的同构集合
3. 问题：
    1. 任何基本类型都不能作为类型参数，例如：`int`必须转换（自动装箱）为`Integer`才可以
        1. 注意自动装箱不适用于数组
    2. 一个类不能实现同一个泛型接口的两种变体：因为擦除的原因，这两个变体会成为相同的接口

       ```
       interface Payable<T> {}
       class Employee implements Payable<Employee> {}
       class Hourly extends Employee implements Payable<Hourly> {}
       ```

Hourly 不能编译，因为擦除会将 `Payable<Employe>` 和 `Payable<Hourly>` 简化为相同的类 **Payable**，这样，上面的代码就意味着在重复两次地实现相同的接口。

3. 重载：不能用传入参数定义方法的重载

```
public class UseList<W,T>{
    void f(List<T> v){}
    void f(List<W> v){}
}
```

如上例，从编译器的角度说，因为擦除，所以重载方法产生了相同的类型签名：假设W和T传入的是相同的参数，那么`f`就重定义报错而非重载了。编译器的"擦除"能够识别到它

自限定的类型Self-bounded Type

1. 很多很多场景都用到，例如`Enum<E extends Enum<E>>`。
2. 首先是一个例子：

```
public class BasicHolder<T> {
    T element;
    void set(T arg) { element = arg; }
    T get() { return element; }
    void f() {
        System.out.println(element.getClass().getSimpleName());
    }
}

class Subtype extends BasicHolder<Subtype> {}

public class CRGWithBasicHolder {
    public static void main(String[] args) {
        Subtype st1 = new Subtype(), st2 = new Subtype();
        st1.set(st2);
        Subtype st3 = st1.get();
        st1.f();
    }
}
/* Output:
Subtype
*/
```

基类用导出类替代其参数。这意味着泛型基类变成了一种其所有导出类的公共功能的模版，但是这些功能对于其所有参数和返回值，将使用导出类型。也就是说，在所产生的类中将使用确切类型而不是基类型。因此，在**Subtype** 中，传递给 `set()` 的参数和从 `get()` 返回的类型都是确切的 **Subtype**。

大概意思就是运用了自限定以后，Subtype对象本身的类型是Subtype，且Subtype对象继承而来的成员、方法形参、方法返回值都是Subtype，这样Subtype对象只能和Subtype对象交互。

因此，自限定：强制要求将正在定义的类当作参数传递给基类，保证类型参数必须与正在被定义的类相同。自限定限制只能强制作用于继承关系。如果使用自限定，就应该了解这个类所用的类型参数将与使用这个参数的类具有相同的基类型。这会强制要求使用这个类的每个人都要遵循这种形式。

```
//forcing the generic to be used as its own bound argument
class SelfBounded<T extends SelfBounded<T>>
```

3. 观察以下结果：

```
class SelfBounded<T extends SelfBounded<T>> {
    T element;
    SelfBounded<T> set(T arg) {
        element = arg;
        return this;
    }
    T get() { return element; }
}

class A extends SelfBounded<A> {}
class B extends SelfBounded<A> {} // Also OK

class C extends SelfBounded<C> {
    C setAndGet(C arg) {
        set(arg);
        return get();
    }
}

class D {}
// Can't do this:
// class E extends SelfBounded<D> {}
// Compile error:
//   Type parameter D is not within its bound

// Alas, you can do this, so you cannot force the idiom:
class F extends SelfBounded {}

public class SelfBounding {
    public static void main(String[] args) {
        A a = new A();
        a.set(new A());
        a = a.set(new A()).get();
        a = a.get();
        C c = new C();
        c = c.setAndGet(new C());
    }
}
```

正如你在 B 类的定义中所看到的，还可以从使用了另一个 **SelfBounded** 参数的 **SelfBounded** 中导出，尽管在 **A** 类看到的用法看起来是主要的用法。对定义 **E** 的尝试说明不能使用不是 **SelfBounded** 的类型参数。 遗憾的是， **F** 可以编译，不会有任何警告，因此自限定惯用法不是可强制执行的。

自限定限制只能强制作用于继承关系。如果使用自限定，就应该了解这个类所用的类型参数将与使用这个参数的类具有相同的基类型.

泛型异常

注解

基本的annotation

1. `@Override` — 限定重写父类方法
    1. 如果父类没有这个方法，编译器会报错：“错误: 方法不会覆盖或实现超类型的方法”
2. `@Deprecated` — 标示已过时
3. `@SuppressWarning` — 抑制编译器警告
    1. 该注解有方法value()属性,可支持多个字符串参数，用户指定忽略哪种警告：`@SupressWarning(value={"uncheck","deprecation"}`
4. `@SafeVarargs`：在 Java 7 中加入用于禁止对具有泛型varargs参数的方法或构造函数的调用方发出警告。
5. `@FunctionalInterface`：Java 8 中加入用于表示类型声明为函数式接口。

元注解

1. | 注解 | 解释 |
   | --- | --- |
   | @Target | 表示注解可以用于哪些地方。可能的 **ElementType** 参数包括：**CONSTRUCTOR**：构造器的声明；**FIELD**：字段声明（包括 enum 实例）；**LOCAL_VARIABLE**：局部变量声明；**METHOD**：方法声明；**PACKAGE**：包声明；**PARAMETER**：参数声明；**TYPE**：类、接口（包括注解类型）或者 enum 声明 |
   | @Retention | 表示注解信息保存的时长。可选的 **RetentionPolicy** 参数包括：**SOURCE**：注解将被编译器丢弃；**CLASS**：注解在 class 文件中可用，但是会被 VM 丢弃；**RUNTIME**：VM 将在运行期也保留注解，因此可以通过反射机制读取注解的信息。 |
   | @Documented | 将此注解保存在 Javadoc 中 |
   | @Inherited | 允许子类继承父类的注解 |
   | @Repeatable | 允许一个注解可以被使用一次或者多次（Java 8）。 |
2. 例如：

   ```
   @Target(value=METHOD)
   @Retention(value=SOURCE)
   public @interface Override {

   }
   ```

   其中`@interface`定义了`Override`是一个Annotation类型，或者叫元数据(meta-data)。`@Target`和`@Retetion`是对`@Override`的注解，称之为元注解(注解的注解)。`@Target`定义你的注解可以应用在哪里（例如是类TYPE，域FIELD，方法METHOD还是参数PATAMETER等）。`@Retention`定义了注解在哪里可用，在源代码中（SOURCE），class文件（CLASS）中或者是在运行时（RUNTIME）。
3. `@Target`

```java
package java.lang.annotation;
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Target {
    /**
     * Returns an array of the kinds of elements an annotation type can be applied to.
     * @return an array of the kinds of elements an annotation type can be applied to
     */
    ElementType[] value();
}
```

1. `Target`也是一个注解类型，在其内部定义了方法`ElementType[] value();`返回值就是`@Target(ElementType.METHOD)`中的`ElementType.METHOD`，也就是注解的属性，是一个`ElementType`枚举。
2. `Target`注解本身也有一个`@Target`元注解，这个`@Target`元注解属性是`ElementType.ANNOTATION_TYPE`，也就是说`Target`注解只能用作元数据(注解)的注解，所以叫它元注解。
3. `ElementType`：这个枚举其实就是定义了注解的适用范围，在`Override`注解中，`@Target`的属性是`ElementType.METHOD`,所以`Override`这个注解只能用于注解Method（方法）。

4. `@Retention`：

```java
package java.lang.annotation;
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Retention {
    /**
     * Returns the retention policy.
     * @return the retention policy
     */
    RetentionPolicy value();
}
```

1. `RetentionPolicy`：`@Retention`注解则定义了注解的保留范围，如:在源代码、CLASS文件或运行时保留。超出`@Retention`定义的属性，注解将被丢弃。

5. `@Documented`：如果一个注解定义了`@Ducumented`，在javadoc生成API文档时，被这个注解标记的元素在文档上也会出现该注解。命令行生成javadoc的示例：

```
javadoc B.java
```

6. `@Inherited`：只对`@Target`为`ElementType.TYPE`(类、接口、枚举)有效，并且只支持类元素。使用了`@Inherited`的注解修饰在一个class上，可以保证继承它的子类也拥有同样的注解。

自定义注解

1.
```java
@Documented
@Target(ElementType.METHOD)
@Inherited
@Retention(RetentionPolicy.RUNTIME)
public @interface AuthorAnno{
    //以下无参方法实际上定义的是注解的属性
    //其返回值可以是所有基本类型、String、Class、enum、Annotation或以上类型的数组
    String name();
    //default关键字可以定义该属性的默认值，如果没有指定默认值在使用注解时必须显示指定属性值
    String website() default "hello";
    int revision() default 1;
}

```

2. 注解可以进行解析：在运行时获得代码的注解信息——在测试的时候测试框架可以读到你的哪个方法是用来做测试的

```java
@Documented
@Target(ElementType.METHOD)
@Inherited
@Retention(RetentionPolicy.RUNTIME)
public @interface AuthorAnno{
    //以下无参方法实际上定义的是注解的属性
    //其返回值可以是所有基本类型、String、Class、enum、Annotation或以上类型的数组
    String name();
    //default关键字可以定义该属性的默认值，如果没有指定默认值在使用注解时必须显示指定属性值
    String website() default "hello";
    int revision() default 1;
}

public class AnnotationDemo {
    @AuthorAnno(name="lvr", website="hello", revision=1)
    public static void main(String[] args) {
        System.out.println("I am main method");
    }

    @SuppressWarnings({ "unchecked", "deprecation" })
    @AuthorAnno(name="lvr", website="hello", revision=2)
    public void demo(){
        System.out.println("I am demo method");
    }
}
```

注解解析：

```java
public class AnnotationParser {
    public static void main(String[] args) throws SecurityException, ClassNotFoundException {
        Method[]  demoMethod = AnnotationDemo.class.getMethods();

        for (Method method : demoMethod) {

```

```
            if (method.isAnnotationPresent(AuthorAnno.class)) {
                AuthorAnno annotationInfo =
method.getAnnotation(AuthorAnno.class);
                System.out.println("method: "+ method);
                System.out.println("name= "+ annotationInfo.name() +
                        " , website= "+ annotationInfo.website()
                       + " , revision= "+annotationInfo.revision());
            }
        }
    }
}
```

测试

1. 简单示例：

```
public class Math {
    public int factorial(int n) throws Exception { // 阶乘函数
        if (n < 0) {
            throw new Exception("负数没有阶乘");
        } else if (n <= 1) {
            return 1;
        } else {
            return n * factorial(n - 1);
        }
    }
}
```

编写一个测试类：

```
import org.junit.Test;
import static org.junit.Assert.*;

public class MathTest {
    @Test
    public void testFactorial() throws Exception {
        assertEquals(120, new Math().factorial(5));
    }
}
```

- 导入了`org.junit.Test`和`org.junit.Assert.*`，后者是静态导入
- `testFactorial`是在要测试的方法名`factorial`前加个`test`
- 所有测试方法返回类型必须为void且无参数。
- 一个测试方法之所以是个测试方法是因为`@Test`这个注解
- `assertEquals`的作用是判断两个参数是否相等
- JUnit4包含一堆`assertXX`方法，这些`assertXX`统称为断言

运行`java -cp ... org.junit.runner.JUnitCore MathTest`命令即可

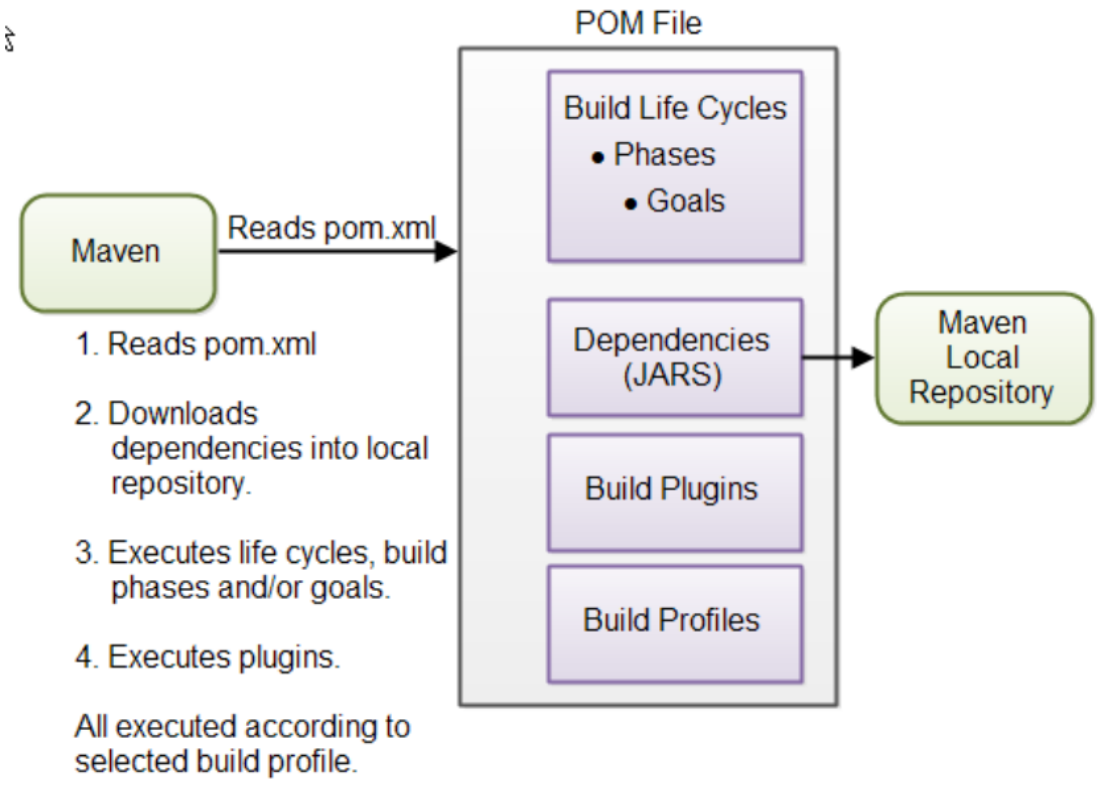
2. 使用注解来定义测试规则：
    1. 一些标注：
        1. `@Test`：把一个方法标记为测试方法
            1. `excepted`属性用来测试异常，例如`@Test(expected = Exception.class)`
            2. `timeout`属性用来测试性能，例如`@Test(timeout = 2000)`

2. `@Before`：每一个测试方法执行前自动调用一次
3. `@After`：每一个测试方法执行完自动调用一次
4. `@BeforeClass`：所有测试方法执行前执行一次，在测试类还没有实例化就已经被加载，所以用`static`修饰
5. `@AfterClass`：所有测试方法执行完执行一次，在测试类还没有实例化就已经被加载，所以用`static`修饰
6. `@Ignore`：暂不执行该测试方法

2. `@BeforeClass`和`@AfterClass`**在类被实例化前（构造方法执行前）就被调用了，而且只执行一次**，通常用来初始化和关闭资源
3. `@Before`和`@After`和在每个`@Test`执行前后都会被执行一次
4. `@Test`标记一个方法为测试方法，被`@Ignore`标记的测试方法不会被执行
5. 每个测试方法执行前**都会**重新实例化测试类

构建

1. Maven阶段
    1. **validate**: 验证项目是否正确，所有必要的信息是否可用
    2. **compile**: 编译项目的源代码
    3. **test**: 使用合适的单元测试框架测试编译的源代码。这些测试不应该要求打包或部署代码
    4. **package**: 将编译好的代码打包成可分发的格式，比如JAR。
    5. **integration-test**: 必要时处理包并将其部署到可以运行集成测试的环境中
    6. **verify**: 运行任何检查以验证包是否有效并符合质量标准
    7. **install**: 将包安装到本地存储库中，作为本地其他项目的依赖项使用
    8. **deploy**: 在集成或发布环境中完成，将最终包复制到远程存储库，以便与其他开发人员和项目共享。
2. Maven工作：



3. Maven POM文件的作用：

1. Maven POM文件（Project Object Model）是描述项目资源的XML文件。这包括源代码、测试源等所在的目录、项目的外部依赖项（JAR文件）等。
2. POM文件描述了要构建什么，但通常不描述如何构建它。如何构建它取决于Maven构建阶段和目标。不过，如果需要的话，可以在Maven构建阶段插入自定义操作（目标）。

输入输出

> **注意**：Java 8 函数式编程中的 `Stream` 类和这里的 I/O stream 没有任何关系。这又是另一个例子，如果再给设计者一次重来的机会，他们将使用不同的术语。

1. 流的特性：FIFO，顺序存储，只读或只写。
    1. 特例：`RandomAccessFile`，不在`InputStream`或者`OutputStream`继承体系里，直接继承自`Object`，所以它可以既读又写
    2. Note: 在 Java 1.4 中，`RandomAccessFile`的大多数功能（但不是全部）都被 NIO 中的内存映射文件（mmap）取代
2. 问题：

```
Class<? extends InputStream> InClass=System.in.getClass();
Class<? extends PrintStream> OutClass=System.out.getClass();
System.out.println("in 所在的类为："+InClass.toString());
System.out.println("out 所在的类为："+OutClass.toString());
```

   的结果？

```
in 所在的类为：class java.io.BufferedInputStream
out 所在的类为：class java.io.PrintStream
```

3. 字节流 vs. 字符流的最佳实践：使用带有Text I/O的字符流

```
FileInputStream fis = new FileInputStream(file);
InputStreamReader isr = new InputStreamReader(fis, "UTF-8");
FileOutputStream fos = new FileOutputStream(newFilePath);
OutputStreamWriter osw = new OutputStreamWriter(fos, "UTF-8");
```

4. 缓冲流的最佳实践：用Buffered Streams包装字节流或字符流

```
try (BufferedReader in = new BufferedReader(
        new FileReader(filename))) {
    return in.lines()
             .collect(Collectors.joining("\n"));
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

    1. 限制：一旦要使用`readLine()`，我们就不应该用`DataInputStream`，而应该使用`BufferedReader`。除了这种情况之外的情形中，`DataInputStream`仍是 I/O 类库的首选成员。
5. 读写中文的例子：

```
File firstFile = new File("chineseIn.txt");
File secondFile = new File("chineseOut.txt");
BufferedReader in = null;
BufferedWriter out = null;
```

```
try {
    in = new BufferedReader(new InputStreamReader(new
FileInputStream(firstFile), "UTF-8"));
    out = new BufferedWriter(new OutputStreamWriter(new
FileOutputStream(secondFile), "UTF-8"));
    String line = "";
    while((line = in.readLine())!=null){
      System.out.println(line);
      out.write(line+"\r\n");
    }
}
catch (FileNotFoundException e) {
    System.out.println("File is not found!");
}
catch (IOException e) {
    System.out.println("Read or write Exception!");
}
finally{
    //TODO: clean up
}
```

BufferedReader、BufferedWriter是操作字符的，InputStreamReader和OutputStreamWriter是字节流到字符流的桥接器，FileInputStream和FileOutputStream是文件字节输入输出流

6. try-with-resources:

```
public static void main(String[] args) throws FileNotFoundException,
IOException {
    try(FileInputStream input = new FileInputStream("test.txt")) {
        int data = input.read();
        while(data != -1){
            System.out.print((char) data);
            data = input.read();
        }
    }
}
```

7. Java IO的设计模式：装饰器模式

输入流类型

1. 数据源包括：
    1. 字节数组；
    2. `String`对象；
    3. 文件；
    4. “管道”，工作方式与实际生活中的管道类似：从一端输入，从另一端输出；
    5. 一个由其它种类的流组成的序列，然后我们可以把它们汇聚成一个流；
    6. 其它数据源，如 Internet 连接。
2. `InputStream`类型：

类	功能	构造器参数	如何使用
`ByteArrayInputStream`	允许将内存的缓冲区当做 `InputStream` 使用	缓冲区，字节将从中取出	作为一种数据源：将其与 `FilterInputStream` 对象相连以提供有用接口
`StringBufferInputStream`	将 `String` 转换成 `InputStream`	字符串。底层实现实际使用 `StringBuffer`	作为一种数据源：将其与 `FilterInputStream` 对象相连以提供有用接口
`FileInputStream`	用于从文件中读取信息	字符串，表示文件名、文件或 `FileDescriptor` 对象	作为一种数据源：将其与 `FilterInputStream` 对象相连以提供有用接口
`PipedInputStream`	产生用于写入相关 `PipedOutputStream` 的数据。实现“管道化”概念	`PipedOutputSteam`	作为多线程中的数据源：将其与 `FilterInputStream` 对象相连以提供有用接口
`SequenceInputStream`	将两个或多个 `InputStream` 对象转换成一个 `InputStream`	两个 `InputStream` 对象或一个容纳 `InputStream` 对象的容器 `Enumeration`	作为一种数据源：将其与 `FilterInputStream` 对象相连以提供有用接口
`FilterInputStream`	抽象类，作为“装饰器”的接口。其中，“装饰器”为其它的 `InputStream` 类提供有用的功能。见表 I/O-3	见表 I/O-3	见表 I/O-3

输出流类型

1. 输出目标：字节数组（但不是 `String`）、文件或管道
2. `OutputStream` 类型

类	功能	构造器参数	如何使用
`ByteArrayOutputStream`	在内存中创建缓冲区。所有送往“流”的数据都要放置在此缓冲区	缓冲区初始大小（可选）	用于指定数据的目的地：将其与 `FilterOutputStream` 对象相连以提供有用接口
`FileOutputStream`	用于将信息写入文件	字符串，表示文件名、文件或 `FileDescriptor` 对象	用于指定数据的目的地：将其与 `FilterOutputStream` 对象相连以提供有用接口
`PipedOutputStream`	任何写入其中的信息都会自动作为相关 `PipedInputStream` 的输出。实现“管道化”概念	`PipedInputStream`	指定用于多线程的数据的目的地：将其与 `FilterOutputStream` 对象相连以提供有用接口
`FilterOutputStream`	抽象类，作为“装饰器”的接口。其中，“装饰器”为其它 `OutputStream` 提供有用功能。见表 I/O-4	见表 I/O-4	见表 I/O-4

对象序列化

1. Java序列化是**不能**序列化***static***变量的，因为其保存的是**对象的状态**，而static变量保存在**全局数据区**，在对象未实例化时就已经生成，属于**类的状态。**
2. `private static final long serialVersionUID`的作用？表示序列化版本标识符的静态变量
3. Serializable的代码怎么写?

```
public class SerialCtl implements Serializable {
    private String a;
    private transient String b;
    public SerialCtl(String aa, String bb) {
        a = aa;  b = bb;
    }
    public String toString() {
        return a + "\n" + b;
    }
    private void writeObject(ObjectOutputStream stream) throws
IOException {
        stream.defaultWriteObject();
    }
    private void readObject(ObjectInputStream stream) throws
IOException, ClassNotFoundException {
        stream.defaultReadObject();
    }
    public static void main(String[] args) throws IOException,
ClassNotFoundException {
        SerialCtl sc = new SerialCtl("Test1", "Test2");
        System.out.println("Before:\n" + sc);
        ByteArrayOutputStream buf= new ByteArrayOutputStream();
        ObjectOutputStream o = new ObjectOutputStream(buf);
        o.writeObject(sc);
        ObjectInputStream in = new ObjectInputStream(new
ByteArrayInputStream(buf.toByteArray()));
        SerialCtl sc2 = (SerialCtl)in.readObject();
```

```
        System.out.println("After:\n" + sc2);
    }
}
```

结果：

```
Before:
Test1
Test2
After:
Test1
null
```

4. **Serializable和Externalizable**，*考试考过！*
    1. Serializable是标识接口，实现该接口，无需重写任何方法
    2. Serializable提供了两种方式进行对象的序列化：（1）采用默认序列化方式，将非transient和非static的属性进行序列化；（2）编写readObject和writeObject完成部分属性的序列化
    3. Externalizable接口的实现方式一定要有默认的无参构造函数
    4. 采用Externalizable无需产生序列化ID（serialVersionUID），而Serializable接口则需要
    5. 相比较Serializable, Externalizable序列化、反序列更加快速，占用相比较小的内存
5. Serializable和Externalizable：Externalizable 接口继承了 Serializable 接口，同时增添了两个方法：writeExternal() 和 readExternal()。
    1. 对于 Serializable 对象，对象完全以它存储的二进制位为基础来构造，而不调用构造器。而对于一个 Externalizable 对象，所有**普通的默认构造器**都会被调用（包括在字段定义时的初始化），然后调用 readExternal() 必须注意这一点--所有默认的构造器都会被调用，才能使 Externalizable 对象产生正确的行为。
    2. ```
        private int i;
        private String s; // No initialization
        public Blip3() {
            System.out.println("Blip3 Constructor");
            // s, i not initialized
        }
        public Blip3(String x, int a) {
            System.out.println("Blip3(String x, int a)");
            s = x;
            i = a;
            // s & i initialized only in non-no-arg constructor.
        }
        @Override
        public String toString() { return s + i; }
        @Override
        public void writeExternal(ObjectOutput out)
                throws IOException {
            System.out.println("Blip3.writeExternal");
            // You must do this:
            out.writeObject(s);
            out.writeInt(i);
        }
        @Override
        public void readExternal(ObjectInput in)
                throws IOException, ClassNotFoundException {
            System.out.println("Blip3.readExternal");
            // You must do this:
            s = (String)in.readObject();
        ```
        ```

```
29      i = in.readInt();
30  }
```

假如不在 readExternal() 中初始化 s 和 i，s 就会为 null，而 i 就会为零。

3. 我们不仅需要在 writeExternal() 方法（没有任何默认行为来为 Externalizable 对象写入任何成员对象）中将来自对象的重要信息写入，还必须在 readExternal() 方法中恢复数据。
4. Externalizable 对象在默认情况下不保存它们的任何字段

6. transient关键字：有一种办法可以防止对象的敏感部分被序列化，就是将类实现为 Externalizable，如前面所示。这样一来，没有任何东西可以自动序列化，并且可以在 writeExternal() 内部只对所需部分进行显式的序列化。但如果实现为Serializable，可以用 transient（瞬时）关键字逐个字段地关闭序列化
7. Externalizable的替代：向Serializable中添加下述方法

```
1  private void writeObject(ObjectOutputStream stream) throws IOException
2  private void readObject(ObjectInputStream stream) throws IOException,
   ClassNotFoundException
```

注意两个方法都要是`private`的。这样，在调用 ObjectOutputStream.writeObject() 时，会检查所传递的 Serializable 对象，看看是否实现了它自己的 writeObject()。如果是这样，就跳过正常的序列化过程并调用它的 writeObiect()。readObject() 的情形与此相同。

8. 持久化：如果我们查看 JDK 文档，就会发现 Class 是 Serializable 的，因此只需直接对 Class 对象序列化，就可以很容易地保存 static 字段。在任何情况下，这都是一种明智的做法。

新I/O

ByteBuffer

1. 有且仅有 **ByteBuffer**（字节缓冲区，保存原始字节的缓冲区）这一类型可直接与通道交互
2. 旧式 I/O 中的三个类分别被更新成 **FileChannel**（文件通道），分别是：**FileInputStream**、**FileOutputStream**，以及用于读写的 **RandomAccessFile** 类。
    1. **FileChannel** 的操作相当基础：操作 **ByteBuffer** 来用于读写，并独占式访问和锁定文件区域。
    2. **NIO** 的目标是快速移动大量数据，因此 **ByteBuffer** 的大小应该很重要
3. 将基本类型数据插入 **ByteBuffer** 的最简单方法就是使用 `asCharBuffer()`、`asShortBuffer()` 等方法获取该缓冲区适当的“视图”（View），然后调用该“视图”的 `put()` 方法。

```
1  ByteBuffer bb = ByteBuffer.allocate(BSIZE);
2  // 自动分配 0 到 ByteBuffer:
3  int i = 0;
4  while(i++ < bb.limit())
5      if(bb.get() != 0)
6          System.out.println("nonzero");
7  System.out.println("i = " + i);
8  bb.rewind();
9  // 保存和读取 char 数组:
10 bb.asCharBuffer().put("Howdy!");
11 char c;
12 while((c = bb.getChar()) != 0)
13     System.out.print(c + " ");
14 System.out.println();
15 bb.rewind();
```

```
// 保存和读取 short:
bb.asShortBuffer().put((short)471142);
System.out.println(bb.getShort());
bb.rewind();
// 保存和读取 int:
bb.asIntBuffer().put(99471142);
System.out.println(bb.getInt());
bb.rewind();
// 保存和读取 long:
bb.asLongBuffer().put(99471142);
System.out.println(bb.getLong());
bb.rewind();
// 保存和读取 float:
bb.asFloatBuffer().put(99471142);
System.out.println(bb.getFloat());
bb.rewind();
// 保存和读取 double:
bb.asDoubleBuffer().put(99471142);
System.out.println(bb.getDouble());
bb.rewind();
```

输出

```
i = 1025
H o w d y !
12390
99471142
99471142
9.9471144E7
9.9471142E7
```

4. **ByteBuffer** 以“高位优先”形式存储数据；通过网络发送的数据总是使用“高位优先”形式。我们可以使用 **ByteOrder** 的 `order()` 方法和参数 **ByteOrder.BIG_ENDIAN** 或 **ByteOrder.LITTLE_ENDIAN** 来改变它的字节存储次序。

缓冲区数据操作

1. 例如，要将字节数组写入文件，使用 **ByteBuffer.**`wrap()` 方法包装字节数组，使用 `getChannel()` 在 **FileOutputStream** 上打开通道，然后从 **ByteBuffer** 将数据写入 **FileChannel**。
2. **ByteBuffer** 是将数据移入和移出通道的唯一方法，我们只能创建一个独立的基本类型缓冲区，或者使用 `as` 方法从 **ByteBuffer** 获得一个新缓冲区。也就是说，不能将基本类型缓冲区转换为 **ByteBuffer**。但我们能够通过视图缓冲区将基本类型数据移动到 **ByteBuffer** 中或移出 **ByteBuffer**。

内存映射文件

1. 为了读写，我们从 **RandomAccessFile** 开始，获取该文件的通道，然后调用 `map()` 来生成 **MappedByteBuffer** ，这是一种特殊的直接缓冲区。你必须指定要在文件中映射的区域的起始点和长度—这意味着你可以选择映射大文件的较小区域。

```
try(
    RandomAccessFile tdat =
        new RandomAccessFile("test.dat", "rw")
) {
    MappedByteBuffer out = tdat.getChannel().map(
        FileChannel.MapMode.READ_WRITE, 0, length);
    for(int i = 0; i < length; i++)
        out.put((byte)'x');
    System.out.println("Finished writing");
    for(int i = length/2; i < length/2 + 6; i++)
        System.out.print((char)out.get(i));
}
```

文件锁

1. 通过调用 **FileChannel** 上的 `tryLock()` 或 `lock()`，可以获得整个文件的 **FileLock**（**SocketChannel**、**DatagramChannel** 和 **ServerSocketChannel** 不需要锁定，因为它们本质上是单进程实体；通常不会在两个进程之间共享一个网络套接字）。

```
try(
    FileOutputStream fos =
        new FileOutputStream("file.txt");
    FileLock fl = fos.getChannel().tryLock()
) {
    if(fl != null) {
        System.out.println("Locked File");
        TimeUnit.MILLISECONDS.sleep(100);
        fl.release();
        System.out.println("Released Lock");
    }
} catch(IOException | InterruptedException e) {
    throw new RuntimeException(e);
}
```

2. `tryLock()` 是非阻塞的。它试图获取锁，若不能获取（当其他进程已经持有相同的锁，并且它不是共享的），它只是从方法调用返回。

   `lock()` 会阻塞，直到获得锁，或者调用 `lock()` 的线程中断，或者调用 `lock()` 方法的通道关闭。使用 **FileLock.**`release()` 释放锁。

3. `tryLock(long position, long size, boolean shared)` 和 `lock(long position, long size, boolean shared)` 可以锁住文件的一部分，但是如果文件大小发生变化，具有固定大小的锁不会发生变化。而零参数锁定方法锁定整个文件，即使它在增长。

4. 当 JVM 退出或关闭获取锁的通道时，锁会自动释放，但是你也可以显式地调用 **FileLock** 对象上的 `release()`，如上所示。

Lambda表达式

1. `Comparator` 中的

```
int compare(T o1,
            T o2)
```

   方法，必须保持自反性、传递性和同质性

```
sgn(compare(x, y)) == -sgn(compare(y, x))//自反性
((compare(x, y)>0) && (compare(y, z)>0)) => compare(x, z)>0//传递性
compare(x, y)==0 => sgn(compare(x, z))==sgn(compare(y, z)) for all z//同质性
```

在Java8之前要重载`Comparator`接口的`compare`函数，可以用匿名类实现：

```
Comparator<Employee> compareById = new Comparator<Employee>() {
    @Override
    public int compare(Employee o1, Employee o2) {
        return o1.getId().compareTo(o2.getId());
    }
};
```

但Java8之后：

```
Comparator<Employee> compareById = (Employee o1, Employee o2) ->
                                    o1.getId().compareTo( o2.getId() );
```

可以用函数式程序设计进行表达。

2. Imperative和Declarative
    1. Imperative: is a style of programming where you program the algorithm with control flow and explicit steps.
    2. Declarative: is a style of programming where you declare what needs be done without concern for the control flow.
        1. Functional programming: is a declarative programming paradigm that treats computation as a series of functions and avoids state and mutable data to facilitate concurrency
3. lambda表达式

```
(parameters) -> expression
或
(parameters) ->{ statements; }
```

规则：

- 一个Lambda表达式可以有零个或多个参数
    - 参数类型既可以明确声明，也可以根据上下文来推断。例如`(int a)`与`(a)`等效
    - 所有参数需包含在圆括号内，参数之间用逗号相隔。例如：`(a, b)`或`(int a, int b)`或`(String a, int b, float c)`
    - 空圆括号代表参数集为空。例如：`() -> 42`；当只有一个参数，且其类型可推导时，圆括号`()`可省略。例如：`a -> return a*a`
- Lambda 表达式的主体可包含零条或多条语句
    - 若只有一条语句，花括号`{}`可省略。匿名函数的返回类型与该主体表达式一致
    - 若包含一条以上语句，则表达式必须包含在花括号`{}`中（形成代码块）。匿名函数的返回类型与代码块的返回类型一致，若没有返回则为空

例如

```
(int a, int b) -> {  return a + b; }
() -> System.out.println("Hello World");
(String s) -> { System.out.println(s); }
() -> 42
() -> { return 3.1415 };
```

4. 函数式接口(Functional Interface)就是一个有且仅有一个抽象方法，但是可以有多个非抽象方法的接口。函数式接口可以被隐式转换为 lambda 表达式。
    1. 例如：java.lang.Runnable就是一种函数式接口，在Runnable接口中只声明了一个方法void run()。因此，可以使用 Lambda 表达式和方法引用作为 **Runnable**

```
new Thread(
      () -> System.out.println("lambda")
    ).start();
```

5. 定义自己的函数式接口：`@FunctionalInterface`是 Java 8 新加入的一种接口，用于指明该接口类型声明是根据 Java 语言规范定义的函数式接口。

```
@FunctionalInterface
public interface WorkerInterface {
   public void doSomeWork();
}

public class WorkerInterfaceTest {
    public static void execute(WorkerInterface worker) {
worker.doSomeWork(); }
    public static void main(String [] args) {
        execute(new WorkerInterface() {
            @Override
            public void doSomeWork() {
                System.out.println("Worker invoked using Anonymous
class");
            }
        });
        execute(() -> System.out.println("Worker invoked using Lambda
expression"));
    }
}
```

6. `java.util.function.Function`

```
Interface Function<T,R>
```

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

```
R apply(T t)
```

Applies this function to the given argument.

并发编程

1. Runnable是线程吗?

```
public class MainThread {
    public static void main(String[] args) {
        LiftOff launch = new LiftOff();
        launch.run(); //is it a thread?
    }
}
```

不，Runnable接口仅仅定义任务

```
Runnable r = () -> { task code };
```

因此：`Thread`对象像是运载火箭，`Runnable`的实现对象就是一个荷载（payload）：

```
Thread t = new Thread(new LiftOff());//把任务装进线程里
```

2. 线程池：

```
ExecutorService exec = Executors.newCachedThreadPool();
for (int i = 0; i < 5; i++)
    exec.execute(new LiftOff());
exec.shutdown();
```

`CachedThreadPool`：根据需要创建新线程的线程池，如果现有线程没有可用的，则创建一个新线程并添加到池中，如果有被使用完但是还没销毁的线程，就复用该线程

3. 为什么需要线程池?

   在面向对象编程中，创建和销毁对象是很费时间的，因为创建一个对象要获取内存资源或者其它更多资源。在Java中更是如此，虚拟机将试图跟踪每一个对象，以便能够在对象销毁后进行垃圾回收。所以提高服务程序效率的一个手段就是尽可能减少创建和销毁对象的次数，特别是一些很耗资源的对象创建和销毁。

4. Callable，Future：获得异步执行的任务的结果

   1. `Callable<V>`：封装异步运行的任务，有返回值

      ```
      public interface Callable<V> {
          V call() throws Exception;
      }
      ```

   2. `Future<V>`：保存异步计算的结果

      ```
      V get() //阻塞，直到计算完成
      V get(long timeout, TImeUnit unit)
      void cancel(boolean mayInterrupt)
      boolean isCancelled()
      boolean isDone()
      ```

   3. 例子：

      ```
      class MyCallable implements Callable<String>{
          @Override
          public String call() throws Exception {
              System.out.println("做一些耗时的任务...");
              Thread.sleep(5000);
      ```

```
        return "OK";
    }
}

public class FutureSimpleDemo {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executorService = Executors.newCachedThreadPool();
        Future<String> future = executorService.submit(new MyCallable());
        System.out.println("do something...");
        System.out.println("得到异步任务返回结果: " + future.get());
        System.out.println("Completed!");
    }
}
```

当调用`Future`的`get()`方法以获得结果时，当前线程就开始阻塞，直到`call()`方法结束返回结果。

5. yield和sleep
    1. yield方法会临时暂停当前正在执行的线程，来让有同样优先级的正在等待的线程有机会执行
    2. 如果没有正在等待的线程，或者所有正在等待的线程的优先级都比较低，那么该线程会继续运行
    3. 执行了yield方法的线程什么时候会继续运行由线程调度器来决定，不同的厂商可能有不同的行为
    4. yield方法不保证当前的线程会暂停或者停止，但是可以保证当前线程在调用yield方法时会放弃CPU
    5. sleep指定休眠时间，休眠结束被调用
6. 获取当前线程：`public static Thread currentThread()`
7. join：`final void join() throws InterruptedException`，等待所调用线程结束，即某个线程在另一个线程t上调用`t.join()`，此线程将被挂起，直到目标线程t结束才恢复
8. Synchronized和ReentrantLock区别：

synchronized	**Reentrantlock**
使用Object本身的wait、notify、notifyAll调度机制	与Condition结合进行线程的调度
显式的使用在同步方法或者同步代码块	显式的声明指定起始和结束位置
托管给JVM执行，不会因为异常、或者未释放而发生死锁	手动释放锁

9. volatile关键字：为实例字段的同步访问提供了一种免锁机制。
    1. 如果声明一个字段为volatile，那么编译器和虚拟机就知道该字段可能被另一个线程并发更新。更确切地说，对volatile变量的每次读操作都会直接从计算机的主存中读取，而不是从CPU缓存中读取；同样，每次对volatile变量的写操作都会直接写入到主存中，而不仅仅写入到CPU缓存里。
10. 单例模式：
    1. 线程不安全：

```
public class LazySingleton {
```

```
    private static LazySingleton instance = null;
    protected LazySingleton(){
        System.out.println("Singleton's consturct method is invoked. " +
                "This method should not be public");
    }
    //is it thread-safe? how to?
    public static LazySingleton getInstance(){
        if (instance == null){
            instance = new LazySingleton();
        }
        return instance;
    }
    public void operation(){
        System.out.println("LazySignleton.operation() is executed");
    }
}
```

2. 利用synchronized实现线程安全：

```
public class ThreadSafeSingleton {
    private static ThreadSafeSingleton instance = null;
    protected ThreadSafeSingleton(){
        System.out.println("Singleton's consturct method is invoked. " +
                "This method should not be public");
    }
    public static synchronized ThreadSafeSingleton getInstance(){
        if (instance == null){
            instance = new ThreadSafeSingleton()
        }
        return instance;
    }
    public void operation(){
        System.out.println("ThreadSafeSingleton.operation() is executed");
    }
}
```

3. 利用双检锁机制实现线程安全：

```
public static ThreadSafeSingleton getInstance(){
    if (instance == null){
        synchronized (ThreadSafeSingleton.class){
            if(instance == null){
                instance = new ThreadSafeSingleton();
            }
        }
    }
    return instance;
}
```

11. ThreadLocal：实现线程本地存储的功能，原理：有一个ThreadLocalMap对象，这个对象存储了一组以TreadLocal.threadLocalHashCode为键，以本地线程变量为值的K-V值对。示例：

```
private static final ThreadLocal<Integer> value = new
ThreadLocal<Integer>() {
    @Override
    protected Integer initialValue() {
        return 0;
    }
};
```

这样各个线程的value值是相互独立的，本线程的累加操作不会影响到其他线程的value。

12. wait和notify：
    1. `Waiter`：等待其它的线程调用notify方法以唤醒线程完成处理。注意等待线程必须通过加synchronized同步锁拥有Message对象的监视器。

```
public class Waiter implements Runnable{
    private Message msg;
    public Waiter(Message m){
        this.msg=m;
    }
    @Override
    public void run() {
        String name = Thread.currentThread().getName();
        synchronized (msg) {
            try{
                System.out.println(name+" waiting to get notified
at time:"+System.currentTimeMillis());
                msg.wait();
            }catch(InterruptedException e){
                e.printStackTrace();
            }
            System.out.println(name+" waiter thread got notified at
time:"+System.currentTimeMillis());
            //process the message now
            System.out.println(name+" processed: "+msg.getMsg());
        }
    }
}
```

    2. `Notifier`：处理Message对象并调用notify方法唤醒等待Message对象的线程。注意synchronized代码块被用于持有Message对象的监视器。

```
public class Notifier implements Runnable {
    private Message msg;
    public Notifier(Message msg) {
        this.msg = msg;
    }
    @Override
    public void run() {
        String name = Thread.currentThread().getName();
        System.out.println(name+" started");
        try {
            Thread.sleep(1000);
            synchronized (msg) {
                msg.setMsg(name+" Notifier work done");
                //msg.notify();  //只能唤醒一个Waiter线程
                msg.notifyAll();  //可以唤醒所有的Waiter线程
```

```
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

13. wait和sleep的区别：
    1. 调用wait方法时，线程在等待的时候会释放掉它所获得的monitor，但是调用Thread.sleep()方法时，线程在等待的时候仍然会持有monitor或者锁，wait方法应在同步代码块中调用，但是sleep方法不需要；
    2. Thread.sleep()方法是一个静态方法，作用在当前线程上；但是wait方法是一个实例方法，并且只能在其他线程调用本实例的notify()方法时被唤醒
    3. sleep：暂停线程一段特定时间；线程通信：wait

设计模式

1. 指导设计模式的三个概念：复用、接口与实现分离、解耦
2. 遇到过的设计模式：
    1. `Collections`：迭代器模式
    2. Java IO：装饰器模式
    3. Java GUI：组合模式、默认适配器模式、观察者模式
3. 设计模式分类：创建型、结构型、行为型

创建型设计模式

1. 简单工厂、工厂方法、抽象工厂、建造者模式、原型模式、单例模式
2. 单例模式：
    1. 懒汉式，线程不安全：

```
public class LazySingleton {
    private static LazySingleton instance = null;
    protected LazySingleton(){
        System.out.println("Singleton's consturct method is
invoked. " +
                "This method should not be public");
    }
    //is it thread-safe? how to?
    public static LazySingleton getInstance(){
        if (instance == null){
            instance = new LazySingleton();
        }
        return instance;
    }
    public void operation(){
        System.out.println("LazySignleton.operation() is
executed");
    }
}

```

    2. 饿汉式，线程安全，无锁，效率高，但类加载时即初始化，没有lazy loading，浪费内存：

```
public class EagerSingleton {
    //is it thread-safe?
    private static final EagerSingleton instance = new EagerSingleton() ;
    private EagerSingleton() {}
    public static EagerSingleton getInstance(){
        return instance ;
    }
    public void operation(){
      System.out.println("EagerSignleton.operation() is executed");
    }
}
```

注意final static的变量会被编译器放在常量池。它基于 classloader 机制避免了多线程的同步问题，不过，instance 在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用 getInstance 方法， 但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化 instance 显然没有达到 lazy loading 的效果。

3. 懒汉式双重校验锁，线程安全，在getInstance() 的性能对应用程序很关键时性能好：

```
public class ThreadSafeSingleton {
    private static ThreadSafeSingleton instance = null;
    protected ThreadSafeSingleton(){
        System.out.println("Singleton's consturct method is invoked. " +
                "This method should not be public");
    }
    //double-check locking
    public static ThreadSafeSingleton getInstance(){
        if (instance == null){
            synchronized (ThreadSafeSingleton.class){
                if(instance == null){
                    instance = new ThreadSafeSingleton();
                }
            }
        }
        return instance;
    }
    public void operation(){
        System.out.println("ThreadSafeSingleton.operation() is executed");
    }
}
```

4. 内部类，线程安全，延迟加载，线程安全，也减少了内存消耗：

```java
public class LazyInitHolderSingleton {
    private static class SingletonHolder{
        private static LazyInitHolderSingleton instance = new LazyInitHolderSingleton();
    }
    private LazyInitHolderSingleton(){}
    public static LazyInitHolderSingleton getInstance(){
        return SingletonHolder.instance;
    }
    public void operation(){
        System.out.println("LazyInitHolderSignleton.operation() is executed");
    }
}
```

5. 枚举，线程安全：

```java
public enum EnumSingleton {
    uniqueInstance;

    public void operation(){
        System.out.println("EnumSingleton.operation() is executed");
    }
}
```

从Java1.5开始支持; 无偿提供序列化机制; 可以防止多次实例化，即使在面对复杂的序列化或者反射攻击的时候。

这种实现方式还没有被广泛采用，但这是实现单例模式的最佳方法。它更简洁，自动支持序列化机制，绝对防止多次实例化。

这种方式是 Effective Java 作者 Josh Bloch 提倡的方式，它不仅能避免多线程同步问题，而且还自动支持序列化机制，防止反序列化重新创建新的对象，绝对防止多次实例化。不过，由于JDK1.5 之后才加入 enum 特性，用这种方式写不免让人感觉生疏，在实际工作中，也很少用。不能通过 reflection attack 来调用私有构造方法。

结构型设计模式

1. 结构模式描述如何将类或者对象结合在一起形成更大的结构
    - 类的结构模式：结构型模式使用继承机制来组合接口或实现。
    - 对象的结构模式：结构型对象模式描述了如何对一些对象进行组合，从而实现新功能的一些方法。可以在运行时刻改变对象的组合关系。
2. 适配器模式、桥接器模式、组合、装饰器、门面、享元、代理
3. 代理模式
    - Remote Proxy：为一个对象在不同的地址空间提供局部代表
    - Virtual Proxy：根据需要创建开销很大的对象
    - Protection Proxy：控制对原始对象的访问，用于对象应该有不同访问权限的时候
    - Smart Reference：取代了简单的指针，在访问对象时执行一些附加操作—引用计数，加锁，将第一次引用的持久对象装入内存...

行为型设计模式

1. 行为模式是对在不同的对象之间划分责任和算法的抽象化。行为模式不仅仅是关于类和对象的，而且关注它们之间的**通信模式**。
    - 类的行为模式：使用继承关系在几个类之间分配行为
    - 对象的行为模式：使用对象的聚合来分配行为
2. 责任链模式、命令模式、解释器模式、迭代器模式、中介模式、备忘录模式、观察者模式、状态模式、策略模式、模板方法、访问者模式
3. 责任链模式：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。
    1. 优点：
        1. 降低耦合度：对象仅需知道请求会被“正确”地处理。接收者和发送者都没有对方的明确信息
        2. 增强了给对象指派职责的灵活性
    2. 缺点：不保证被接受
4. 命令模式：将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；抽象出待执行的动作以参数化某对象，可以代替“回调”函数
5. 迭代器模式：提供一种方法顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。
    1. 考虑：Iterable!

    ```
    public interface Iterator<E>{
          boolean hasNext();
          <E> next();
          void remove();
    }
    ```

    2. 不同的实现
        1. 宽接口 VS. 窄接口
            - 宽接口：一个聚集的接口提供了可以用来修改聚集元素的方法
            - 窄接口：一个聚集的接口没有提供修改聚集元素的方法
        2. 白箱聚集 VS. 黑箱聚集
            - 白箱聚集：聚集对象为所有对象提供同一个接口(宽接口)
            - 黑箱聚集：聚集对象为迭代子对象提供一个宽接口，而为其它对象提供一个窄接口。同时保证聚集对象的封装和迭代子功能的实现。
6. 观察者模式：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。