Java常识

JVM

> 对于 Java 代码，刚开始都是`被编译器编译成字节码文件`，`然后字节码文件会被交由 JVM 解释执行`，所以可以说 Java 本身是一种`半编译半解释执行的语言`。
>
> 当JIT编译启用时（默认是启用的），JVM读入`.class文件解释后`，将其`发给JIT编译器`。JIT编译器`将字节码编译成本机机器代码`。
>
> 通常`Javac将程序源码编译`，`转换成java字节码`，`JVM通过解释字节码将其翻译成相应的机器指令`，逐条读入，逐条解释翻译。
> 经过解释运行，其运行速度必定会比可运行的二进制字节码程序慢。为了提高运行速度，引入了JIT技术。
>
> 在执行时JIT会把翻译过的机器码保存起来，已备下次使用，因此从理论上来说，采用该JIT技术能够，能够接近曾经纯编译技术。

- 字节码文件是与平台无关的二进制码，执行时由解释器解释成本地机器码。字节码由JVM解释成机器码

char

- 2字节，unicode

main

- 必须是public static的
- 如果有两个main，不会报错，需要选择一个作为入口

Java基础

类型转换

- 不能对boolean类型进行类型转换。
- 不能把对象类型转换成不相关类的对象。
- 在把容量大的类型转换为容量小的类型时必须使用强制类型转换。
- 转换过程中可能导致溢出或损失精度，例如：

```
int i =128;
byte b = (byte)i;
```

因为 byte 类型是 8 位，最大值为127，所以当 int 强制转换为 byte 类型时，值 128 时候就会导致溢出。

- 浮点数到整数的转换是通过舍弃小数得到，而不是四舍五入，例如：

```
(int)23.7 == 23;
(int)-45.89f == -45
```

- ```
  int a = 2147483647*2;
  System.out.println(a);
  long b = 2147483647*2;
  System.out.println(b);
  ```

  2147483647 = 2^31 - 1 左移一位变成-2

  下面这个右侧其实还是int型，如果要得到long，需要用2147483647L

switch

- switch 语句中的变量类型可以是： byte、short、int 或者 char。从 Java SE 7 开始，switch 支持字符串 String 类型了，同时 case 标签必须为字符串常量或字面量。

  switch 语句可以拥有多个 case 语句。每个 case 后面跟一个要比较的值和冒号。

toString

- System.out.println(xx)，括号里面的“xx”如果不是String类型的话，就自动调用xx的toString()方法
- System.out.println(4+8+"Hello,world!");
- System.out.println("Hello,world!"+4+8);

常量池

- new Integer(1) 和Integer a = 1不同，前者会创建对象，**存储在堆中**，而后者因为在-128到127的范围内，不会创建新的对象，而是从IntegerCache中获取的。那么Integer a = 128, 大于该范围的话才会直接通过new Integer（128）创建对象，进行装箱。
- ```
  1.public class Test {
  2.
  3.  public static void main(String[] args) {
  4.
  5.      Integer i = new Integer(128);
  6.      Integer i2 = 128;
  7.
  8.      System.out.println(i == i2);
  9.
  10.     Integer i3 = new Integer(127);
  11.     Integer i4 = 127;
  12.     System.out.println(i3 == i4);
  13.
  14.     Integer i5 = 128;
  15.     Integer i6 = 128;
  16.     System.out.println(i5 == i6);
  17.
  18.     Integer i7 = 127;
  19.     Integer i8 = 127;
  20.     System.out.println(i7 == i8);
  21. }
  22.}
  ```

  输出的结果为
  1.false
  2.false
  3.false
  4.true

  其中第三个，128不在常量池范围（-128-127）

- 所有整数变量都有常量池，但是Double等没有

数组

- 首先必须声明数组变量，才能在程序中使用数组。下面是声明数组变量的语法：

```
dataType[] arrayRefVar;   // 首选的方法

或

dataType arrayRefVar[];  // 效果相同，但不是首选方法
```

注意: 建议使用 **dataType[] arrayRefVar** 的声明风格声明数组变量。 dataType arrayRefVar[] 风格是来自 C/C++ 语言 ，在Java中采用是为了让 C/C++ 程序员能够快速理解java语言。

- 长度是a.length，是成员变量，不是方法，String有length()方法
- 程序运行中不能改变数组的大小
- Java允许创建不规则数组，即Java多维数组中各行的列数可以不同。
- Java没有int a[2]={1,2}这样的声明方法;

final

- 当用final修饰一个类时，表明这个类不能被继承。final类中的成员变量可以根据需要设为final，但是要注意final类中的所有成员方法都会被隐式地指定为final方法。
- 定义为final的方法不能被重写。定义为final的方法执行效率要高于非final方法。如果一个父类的某个方法被设置为private修饰符，子类将无法访问该方法，自然无法覆盖该方法，所以一个定义为private的方法隐式被指定为final类型，这样无须将一个定义为private的方法再定义为final类型。
- final变量，如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象。

序列化

- 一个类的对象要想序列化成功，必须满足两个条件：该类必须实现 java.io.Serializable 接口。该类的所有属性必须是可序列化的。如果有一个属性不是可序列化的，则该属性必须注明是transient。

```
Employee e = null;
try
{
   FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");
   ObjectInputStream in = new ObjectInputStream(fileIn);
   e = (Employee) in.readObject();
   in.close();
   fileIn.close();
}catch(IOException i)
{
   i.printStackTrace();
   return;
}catch(ClassNotFoundException c)
{
   System.out.println("Employee class not found");
   c.printStackTrace();
   return;
}
```

```
Employee e = new Employee();
e.name = "Reyan Ali";
e.address = "Phokka Kuan, Ambehta Peer";
e.SSN = 11122333;
e.number = 101;
try
{
    FileOutputStream fileOut =
    new FileOutputStream("/tmp/employee.ser");
    ObjectOutputStream out = new ObjectOutputStream(fileOut);
    out.writeObject(e);
    out.close();
    fileOut.close();
    System.out.printf("Serialized data is saved in /tmp/employee.ser");
```

import/package

- Java 提供了包机制，用于区别类名的命名空间。
- 同一个包中的类名字是不同的，不同的包中的类的名字是可以相同的，当同时调用两个不同包中相同类名的类时，应该加上包名加以区别。
- ```
  package pkg1[. pkg2[. pkg3…]];
  ```

  例如,一个Something.java 文件它的内容

  ```
  package net.java.util;
  public class Something{
     ...
  }
  ```

  那么它的路径应该是 **net/java/util/Something.java** 这样保存的。 package(包) 的作用是把不同的 java 程序分类保存，更方便的被其他 java 程序调用。

- 为了能够使用某一个包的成员，我们需要在 Java 程序中明确导入该包。使用 "import" 语句可完成此功能。

  在 java 源文件中 import 语句应位于 package 语句之前，所有类的定义之前，可以没有，也可以有多条，其语法格式为：

  ```
  import package1[.package2…].(classname|*);
  ```

  如果在一个包中，一个类想要使用本包中的另一个类，那么该包名可以省略。

- 对于按需类型导入(import java.io.*;)，有人误解为导入一个包下的所有类，其实不然，看名字就知道，他只会按需导入，也就是说它并非导入整个包，而仅仅导入当前类需要使用的类。

  了解以上原理之后，我们可以得出这样的结论：按需类型导入是绝对不会降低Java代码的执行效率的，但会影响到Java代码的编译速度。
  ```

- import static 导入静态方法

访问控制符

- **default** (即默认，什么也不写）: 在同一包内可见，不使用任何修饰符。使用对象：类、接口、变量、方法。
- **private** : 在同一类内可见。使用对象：变量、方法。 **注意：不能修饰类（外部类）**
- **public** : 对所有类可见。使用对象：类、接口、变量、方法
- **protected** : 对同一包内的类和所有子类可见。使用对象：变量、方法。 **注意：不能修饰类（外部类）。**

访问控制

修饰符	当前类	同一包内	子孙类(同一包)	子孙类(不同包)	其他包
public	Y	Y	Y	Y	Y
protected	Y	Y	Y	Y/N（说明）	N
default	Y	Y	Y	N	N
private	Y	N	N	N	N

标识符

- 只能由英文字母（A~Z）或（a~z）、下划线（_）、美元符号（$）和数字（0~9）组成，且开头不能为数字。
- 区分大小写！
- 无长度限制！
- _3_ √
- $aa √
- java 中没有sizeof 运算符，所有类型的长度和表示都是固定的
- java 不用goto和const作关键字，二者是java的保留字
- then 不是关键字，sizeof，null，native都不是关键字
- 关键字和保留字有区别！

Java中关键字（keyword）和保留字（reservedword）
Keyword ： Java的关键字对java的编译器有特殊的意义，他们用来表示一种数据类型，或者表示程序的结构等。
Reserved word:保留字是为java预留的关键字，他们虽然现在没有作为关键字，但在以后的升级版本中有可能作为关键字。
如下列表:

类

构造方法

- 构造方法不能被继承，构造函数名必须与类名一致
- 无返回值无void
- 构造方法可以被重载，不可以重写
- 构造方法不能是静态的
- 一般构造函数可加可不加public,如果加上public,就代表此类可以对外开放,其他的类可以实例化该对象。如果不加public,则默认的修饰词是protected,代表只对同包开放。除了这两个,还可以添加private和default

静态代码块

- 执行优先级高于非静态的初始化块，它会在类初始化的时候执行一次，执行完成便销毁，它仅能初始化类变量，即static修饰的数据成员。
- 非静态：执行的时候如果有静态初始化块，先执行静态初始化块再执行非静态初始化块，在每个对象生成时都会被执行一次，它可以初始化类的实例变量。非静态初始化块会在构造函数执行时，在构造函数主体代码执行之前被运行。

静态成员

- static是不允许用来修饰局部变量
- 通常，静态变量常用final关键来修饰，表示通用资源或可以被所有的对象所使用。
- 如果静态变量未被私有化，可以用"类名.变量名"的方式来使用。
- 注意静态变量初始化的时间和顺序

静态方法

- 静态方法只能访问类的静态变量，或调用类的静态方法。
- 可以被继承，但是不能被重写，如果父子类静态方法名相同，则会隐藏父类方法。

  Father f = new Son(); f.staticMethod();
  Father f = null; f.staticMethod();
  这两个都会调用父类的方法，因为静态方法是类的方法和具体对象没关系，和直接用类名调用一样。
- 静态方法是编译时绑定的，方法重写是运行时绑定的。
- 因为静态方法是独立于实例存在的，所以static里不能用this和super

继承/子父类

- 重写：存在于子父类中，方法名、方法参数、返回值全部相同
- 子类所有的构造函数默认调用父类的无参构造函数（其实是默认省略掉了一行代码：super();）。省略掉的这行super()代码可以自行添加到构造函数的第一行（必须是第一行，否则报错）
- 如果父类定义了有参构造函数，此时子类的构造函数中第一行必须显式调用父类定义的某个有参数构造函数。

泛型

-

```java
public class MaximumTest
{
   // 比较三个值并返回最大值
   public static <T extends Comparable<T>> T maximum(T x, T y, T z)
   {
      T max = x; // 假设x是初始最大值
      if ( y.compareTo( max ) > 0 ){
         max = y; //y 更大
      }
      if ( z.compareTo( max ) > 0 ){
         max = z; // 现在 z 更大
      }
      return max; // 返回最大对象
   }
   public static void main( String args[] )
   {
      System.out.printf( "%d, %d 和 %d 中最大的数为 %d\n\n",
                  3, 4, 5, maximum( 3, 4, 5 ) );

      System.out.printf( "%.1f, %.1f 和 %.1f 中最大的数为 %.1f\n\n",
                  6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );

      System.out.printf( "%s, %s 和 %s 中最大的数为 %s\n","pear",
         "apple", "orange", maximum( "pear", "apple", "orange" ) );
   }
}
```

- 类型通配符一般是使用?代替具体的类型参数。例如 **List** 在逻辑上是**List,List** 等所有List<具体类型实参>的父类。
- ```java
  public static void getUperNumber(List<? extends Number> data) {
        System.out.println("data :" + data.get(0));
    }
  ```

  这里规定了是针对所有派生自Number的类，也就是上限是Number，则String等不可以

  类型通配符下限通过形如 **List**来定义，表示类型只能接受Number和他的父类类型，如 Object 类型的实例。
- List<?> 赋值后，不能往里面添加或删除元素

特殊的类

接口

- 一个接口可以有多个方法。
- 接口文件保存在 .java 结尾的文件中，文件名使用接口名。
- 接口的字节码文件保存在 .class 结尾的文件中。
- 接口相应的字节码文件必须在与包名称相匹配的目录结构中。
- 接口不能用于实例化对象。
- 接口没有构造方法。
- 接口中所有的方法必须是抽象方法。
- 接口不能包含成员变量，除了 static 和 final 变量。
- 接口支持多继承
- 接口中每一个方法也是隐式抽象的,接口中的方法会被隐式的指定为 **public abstract**（只能是 public abstract，其他修饰符都会报错）。接口中所有方法默认都是public，至于为什么要是public，原因在于如果不是public，那么只能在同个包下被实现，可访问权限就降低很多了，那么在实现类中，实现的类相当于子类，子类的访问权限是不能比父类小的，而在java中一个类如果没有权限的修饰符，默认是friendly(同一个包内的其它类才可访问)，所以在实现类中一定要写public。
- 接口中可以含有变量，但是接口中的变量会被隐式的指定为 **public static final** 变量（并且只能是 public，用 private 修饰会报编译错误）。
- 类在实现接口的方法时，必须显示地使用public修饰符
  ```

抽象类

- Java没有virtual关键字，和C++不要搞混了
- abstract，不可以实例化
- 抽象方法：类包含一个特别的成员方法，该方法的具体实现由它的子类确定，那么可以在父类中声明该方法为抽象方法。Abstract 关键字同样可以用来声明抽象方法，抽象方法只包含一个方法名，而没有方法体。抽象方法没有定义，方法名后面直接跟一个分号，而不是花括号。继承这个类的子类必须实现该方法或者声明自己是抽象类。
- 抽象类和接口区别：
  - 接口是公开的，不能有私有的方法或变量，接口中的所有方法都`没有方法体`，通过关键字`interface`实现。
  - 抽象类是可以有私有方法或私有变量的，通过把类或者类中的方法声明为`abstract`来表示一个类是抽象类，被声明为抽象的方法`不能包含方法体`。子类实现方法必须含有相同的或者更低的访问级别(public->protected->private)。抽象类的子类为父类中`所有抽象方法`的具体实现，否则也是抽象类。‘
  - 实现接口的关键字为`implements`，继承抽象类的关键字为`extends`。一个类**可以实现多个接口**，但一个类只能继承一个抽象类。所以，使用接口可以间接地实现多重继承。

静态内部类

- 不能用static修饰顶级类（top level class）。只有内部类可以为static。
- 静态内部类不能直接访问外部类的非静态成员，但可以通过new 外部类().成员的方式访问
- 内部静态类访问外部类的静态成员和方法不需要有指向外部类的引用。但非静态内部类需要持有对外部类的引用。
- 非静态内部类能够访问外部类的静态和非静态成员。静态类不能访问外部类的非静态成员。他只能访问外部类的静态成员和方法。
- 一个非静态内部类不能脱离外部类实体被创建，也就是说需要先new一个外部的对象，一个非静态内部类可以访问外部类的数据和方法，因为他就在外部类里面。
- 外部类只能通过类名访问静态内部类的静态方法和变量

匿名类

```
class Polygon {
   public void display() {
      System.out.println("在 Polygon 类内部");
   }
}

class AnonymousDemo {
   public void createClass() {
```

```
      // 创建的匿名类继承了 Polygon 类
      Polygon p1 = new Polygon() {
         public void display() {
            System.out.println("在匿名类内部。");
         }
      };
      p1.display();
   }
}
```

-
```
interface Polygon {
   public void display();
}

class AnonymousDemo {
   public void createClass() {

      // 匿名类实现一个接口
      Polygon p1 = new Polygon() {
         public void display() {
            System.out.println("在匿名类内部。");
         }
      };
      p1.display();
   }
}
```

- 匿名内部类不能有构造方法
- 匿名内部类不能定义任何静态成员、方法和类。
- 匿名内部类不能是抽象类。（因为在创建匿名内部类的时候，会立即创建匿名内部类的对象。从而也可知匿名内部类必须实现它的抽象父类或者接口里包含的所有抽象方法）
- 匿名内部类不能是public,protected,private,static。
- 内部类可以访问外部类变量
- An anonymous inner class can access final variables in any enclosing scope.

枚举类

多态

- **多态存在的三个必要条件**：继承、重写、父类引用指向子类对象：Parent p = new Child();
- 多态只对方法有效，变量还是以“指针”内部的为准

-
```
Animal a = new Cat();   // 向上转型
a.eat();                // 调用的是 Cat 的 eat
Cat c = (Cat)a;         // 向下转型
c.work();        // 调用的是 Cat 的 work
```

容器

Linkedlist

-

public E remove(int index)	删除指定位置的元素。
public E poll()	删除并返回第一个元素。

这个太离谱了

-

public boolean add(E e)	链表末尾添加元素，返回是否成功，成功为 true，失败为 false。
public void add(int index, E element)	向指定位置插入元素。

标准类库

Math

- .random: 返回 0<= random < 1
- .floor 地板
- .ceil 天花板

Random

Random r1 = new Random();

需要说明的是：你在创建一个Random对象的时候可以给定任意一个合法的种子数，种子数只是随机算法的起源数字，和生成的随机数的区间没有任何关系。如下面的Java代码：

- ```
  Random rand =new Random(25);
  int i;
  i=rand.nextInt(100);
  ```

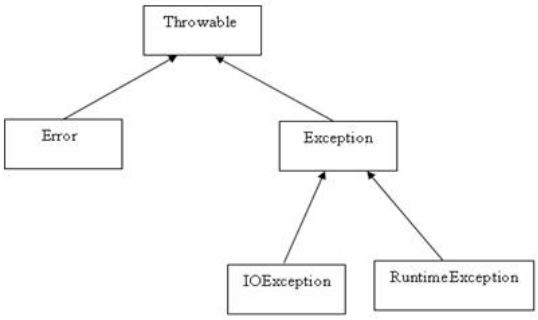
初始化时25并没有起直接作用（注意：不是没有起作用）,rand.nextInt(100);中的100是随机数的上限,产生的随机数为0-100的整数,不包括100。

对于种子相同的Random对象，生成的随机数序列是一样的。

异常

异常分类

- **检查性异常：**最具代表的检查性异常是用户错误或问题引起的异常，这是程序员无法预见的。例如要打开一个不存在文件时，一个异常就发生了，这些异常在编译时不能被简单地忽略。
- **运行时异常：** 运行时异常是可能被程序员避免的异常。与检查性异常相反，运行时异常可以在编译时被忽略。
- **错误：** 错误不是异常，而是脱离程序员控制的问题。错误在代码中通常被忽略。例如，当栈溢出时，一个错误就发生了，它们在编译也检查不到的。



finally

finally 关键字用来创建在 try 代码块后面执行的代码块。无论是否发生异常，finally 代码块中的代码总会被执行。

try catch
  ```

- catch 不能独立于 try 存在。
- 在 try/catch 后面添加 finally 块并非强制性要求的。try 代码后不能既没 catch 块也没 finally 块。try, catch, finally 块之间不能添加任何代码。
- 前后的catch块，后面的不能被前面的包含，否则会编译错误

定义异常

- 所有异常都必须是 Throwable 的子类。
- 如果希望写一个检查性异常类，则需要继承 Exception 类。
- 如果你想写一个运行时异常类，那么需要继承 RuntimeException 类。

可以像下面这样定义自己的异常类：

```
class MyException extends Exception{
}
```

又继承Exception 类来创建的异常类是检查性异常类。

下面的 InsufficientFundsException 类是用户定义的异常类，它继承自 Exception。

一个异常类和其它任何类一样，包含有变量和方法。

throws

如果一个方法没有捕获到一个检查性异常，那么该方法必须使用 throws 关键字来声明。throws 关键字放在方法签名的尾部。

也可以使用 throw 关键字抛出一个异常，无论它是新实例化的还是刚捕获到的。

下面方法的声明抛出一个 RemoteException 异常：

```
import java.io.*;
public class className
{
  public void deposit(double amount) throws RemoteException
  {
    // Method implementation
    throw new RemoteException();
  }
  //Remainder of class definition
}
```

一个方法可以声明抛出多个异常，多个异常之间用逗号隔开。

IO

标准流Stream

Java 的控制台输入由 System.in 完成。

为了获得一个绑定到控制台的字符流，你可以把 System.in 包装在一个 BufferedReader 对象中来创建一个字符流。

下面是创建 BufferedReader 的基本语法：

```
BufferedReader br = new BufferedReader(new
                    InputStreamReader(System.in));
```

BufferedReader 对象创建后，我们便可以使用 read() 方法从控制台读取一个字符，或者用 readLine() 方法读取一个字符串。

- FileOutputStream是用byte为单位进行读写

GUI

container

- Swing组件的窗体通常与组件和容器相关，所以在JFrame对象创建完成后，需要调用getContentPane()方法将窗体转换为容器，然后在容器中添加组件或设置布局管理器。

layout manager

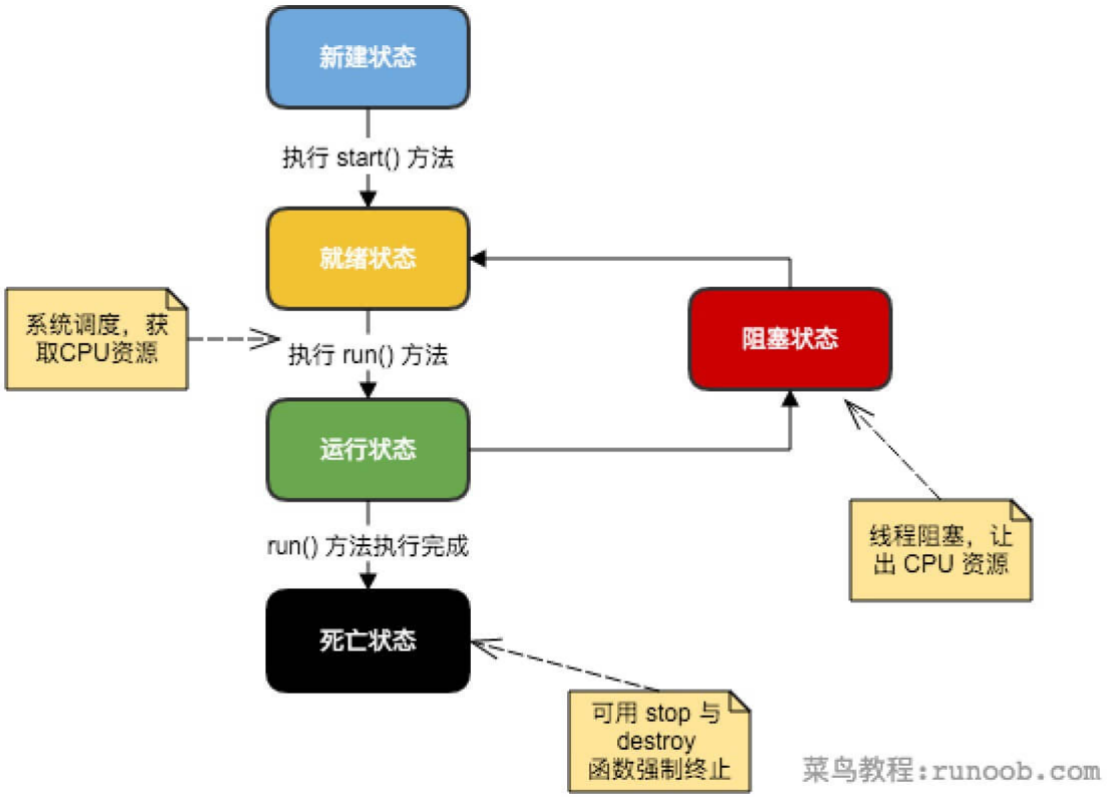
- **BorderLayout**： 分为东、南、西、北、中心 这五个方位。北和 南的组件可以在水平方向上拉伸；而 东和 西 的组件可以在垂直方向上拉伸；中心的组件可同时在水平和垂直方向上拉伸，从而填充所有剩余空间。BorderLayout是RootPaneContainer(JInternalFrame、JDialog、JFrame、JWindow)的默认布局管理器。
- **FlowLayout**： 该布局是从左到右，中间放置，一行放不下就换到另外一行。因此它一行能放置多少组件取决于窗口的宽度。注意它默认组件是居中对齐，可以通过FlowLayout(intalign)函数来指定对齐方式。FlowLayout是JPanel的默认的布局管理器
- 使用setLayout(null)可以变成绝对布局，消除布局管理器

线程

synchronized

- synchronized关键字可以作为函数的修饰符，也可作为函数内的语句，也就是平时说的同步方法和同步语句块。如果 再细的分类，synchronized可作用于instance变量、object reference（对象引用）、static函数和class literals(类名称字面常量)身上。
- 无论synchronized关键字加在方法上还是对象上，它取得的锁都是对象，而不是把一段代码或函数当作锁——而且同步方法很可能还会被其他线程的对象访问。
- 每个对象只有一个锁（lock）与之相关联。
- **synchronized关键字的作用域有二种：**
    - 某个对象实例内，synchronized aMethod(){}可以防止多个线程同时访问这个对象的synchronized方法（如果一个对象有多个synchronized方法，只要一个线 程访问了其中的一个synchronized方法，其它线程不能同时访问这个对象中任何一个synchronized方法）。这时，不同的对象实例的 synchronized方法是不相干扰的。也就是说，其它线程照样可以同时访问相同类的另一个对象实例中的synchronized方法；
    - 某个类的范围，synchronized static aStaticMethod{}防止多个线程同时访问这个类中的synchronized static 方法。它可以对类的所有对象实例起作用。也就是说这个锁是加在这个类上的。

生命周期



Runnable

- ```
  class RunnableDemo implements Runnable {
     private Thread t;
     private String threadName;

     RunnableDemo( String name) {
        threadName = name;
        System.out.println("Creating " +  threadName );
     }

     public void run() {
        System.out.println("Running " +  threadName );
        try {
           for(int i = 4; i > 0; i--) {
              System.out.println("Thread: " + threadName + ", " + i);
              // 让线程睡眠一会
              Thread.sleep(50);
           }
        }catch (InterruptedException e) {
           System.out.println("Thread " +  threadName + " interrupted.");
        }
        System.out.println("Thread " +  threadName + " exiting.");
     }
  ```
- Thread(new RunnableDemo(Object args[])).start()

Thread

- 创建一个线程的第二种方法是创建一个新的类，该类继承 Thread 类，然后创建一个该类的实例。继承类必须重写 run() 方法，该方法是新线程的入口点。它也必须调用 start() 方法才能执行。

RTTI

getClass getName
  ```

- 注意getName需要带上包名

Socket和数据库

函数式编程和流式计算

lambda 条件

lambda表达式可以代替只有一个抽象方法的接口。只有一个抽象方法的接口叫"函数式接口".

Arrays.sort(words, (first, second) -> { first.length() - secon.length() })

函数式接口FI

函数式接口，有如下特点：
- 有且只有一个抽象方法
- 可以有多个静态方法
- 可以有多个default方法(默认方法)
- 可以有多个Object的public的抽象方法

但是不能只有一个Object类的Public方法

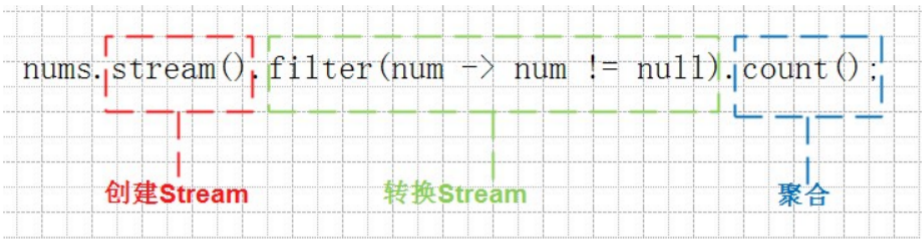
```
@FunctionalInterface
public interface Func {
    Object clone();
    void run();
}
```

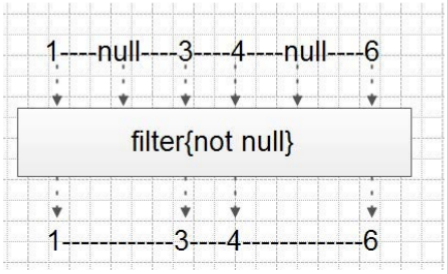
不行，要Object的public方法，而clone是protected的。。

Stream

- 原始版本的Iterator，用户只能一个一个的遍历元素并对其执行某些操作；
- 高级版本的Stream，用户只要给出需要对其包含的元素执行什么操作，比如“过滤掉长度大于10的字符串”、“获取每个字符串的首字母”等，具体这些操作如何应用到每个元素上，就给Stream就好了！

```java
//Lists是Guava中的一个工具类
List<Integer> nums = Lists.newArrayList(1,null,3,4,null,6);
nums.stream().filter(num -> num != null).count();
```

- 



Stream使用基本步骤

1. 创建Stream；
2. 转换Stream，每次转换原有Stream对象不改变，返回一个新的Stream对象（**可以有多次转换**）；
3. 对Stream进行聚合（Reduce）操作，获取想要的结果；

- IntStream

```java
// Fig. 17.5: IntStreamOperations.java
// Demonstrating IntStream operations.
import java.util.Arrays;
import java.util.stream.IntStream;

public class IntStreamOperations
{
   public static void main(String[] args)
   {
      int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};

      // display original values
      System.out.print("Original values: ");
      IntStream.of(values)
               .forEach(value -> System.out.printf("%d ", value));
      System.out.println();

      // count, min, max, sum and average of the values
      System.out.printf("%nCount: %d%n", IntStream.of(values).count());
      System.out.printf("Min: %d%n",
         IntStream.of(values).min().getAsInt());
      System.out.printf("Max: %d%n",
         IntStream.of(values).max().getAsInt());
      System.out.printf("Sum: %d%n", IntStream.of(values).sum());
      System.out.printf("Average: %.2f%n",
         IntStream.of(values).average().getAsDouble());

      // sum of values with reduce method
      System.out.printf("%nSum via reduce method: %d%n",
         IntStream.of(values)
                  .reduce(0, (x, y) -> x + y));

      // sum of squares of values with reduce method
      System.out.printf("Sum of squares via reduce method: %d%n",
         IntStream.of(values)
                  .reduce(0, (x, y) -> x + y * y));
```

```java
// Predicate that returns true for salaries in the range $4000-$6000
Predicate<Employee> fourToSixThousand =
   e -> (e.getSalary() >= 4000 && e.getSalary() <= 6000);

// Display Employees with salaries in the range $4000-$6000
// sorted into ascending order by salary
System.out.printf(
   "%nEmployees earning $4000-$6000 per month sorted by salary:%n");
list.stream()
    .filter(fourToSixThousand)
    .sorted(Comparator.comparing(Employee::getSalary))
    .forEach(System.out::println);

// Display first Employee with salary in the range $4000-$6000
System.out.printf("%nFirst employee who earns $4000-$6000:%n%s%n",
   list.stream()
       .filter(fourToSixThousand)
       .findFirst()
       .get());
```

```java
// Fig. 17.5: IntStreamOperations.java
// Demonstrating IntStream operations.
import java.util.Arrays;
import java.util.stream.IntStream;

public class IntStreamOperations
{
   public static void main(String[] args)
   {
      int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};

      // display original values
      System.out.print("Original values: ");
      IntStream.of(values)
               .forEach(value -> System.out.printf("%d ", value));
      System.out.println();

      // count, min, max, sum and average of the values
      System.out.printf("%nCount: %d%n", IntStream.of(values).count());
      System.out.printf("Min: %d%n",
         IntStream.of(values).min().getAsInt());
      System.out.printf("Max: %d%n",
         IntStream.of(values).max().getAsInt());
      System.out.printf("Sum: %d%n", IntStream.of(values).sum());
      System.out.printf("Average: %.2f%n",
         IntStream.of(values).average().getAsDouble());

      // sum of values with reduce method
      System.out.printf("%nSum via reduce method: %d%n",
         IntStream.of(values)
                  .reduce(0, (x, y) -> x + y));

      // sum of squares of values with reduce method
      System.out.printf("Sum of squares via reduce method: %d%n",
         IntStream.of(values)
                  .reduce(0, (x, y) -> x + y * y));
```

```java
// Predicate that returns true for salaries in the range $4000-$6000
Predicate<Employee> fourToSixThousand =
   e -> (e.getSalary() >= 4000 && e.getSalary() <= 6000);

// Display Employees with salaries in the range $4000-$6000
// sorted into ascending order by salary
System.out.printf(
   "%nEmployees earning $4000-$6000 per month sorted by salary:%n");
list.stream()
    .filter(fourToSixThousand)
    .sorted(Comparator.comparing(Employee::getSalary))
    .forEach(System.out::println);

// Display first Employee with salary in the range $4000-$6000
System.out.printf("%nFirst employee who earns $4000-$6000:%n%s%n",
   list.stream()
       .filter(fourToSixThousand)
       .findFirst()
       .get());
```