- **new** 的作用：**分配空间；调用构造；返回引用**

scanner

- Scanner的作用：通过分隔符模式将输入分解为标记，默认情况下该分隔符模式与空白匹配。
- 通过Scanner 类的 `next()` 与 `nextLine()` 方法获取输入的字符串，在读取前我们一般需要使用 `hasNext` 与 `hasNextLine` 判断是否还有输入的数据
- **Scanner**

```
Scanner scanner = new Scanner(System.in);
while(scanner.hasNextDouble()){
    double x = scanner.nextDouble();
}
```

- **转换大小写**:

```
String original = "Hello, World!";
String uppercase = original.toUpperCase();
String lowercase = original.toLowerCase();
```

- **替换字符串**:

```
String original = "Hello, World!";
String replaced = original.replace("World", "Java");
```

- **分割字符串**:

```
String original = "one,two,three";
String[] parts = original.split(",");
```

- **连接字符串**:

```
String part1 = "Hello, ";
String part2 = "World!";
String combined = part1 + part2;
```

- **格式化字符串**:

```
int number = 42;
String formatted = String.format("The answer is %d", number);
```

例

next

```
import java.util.Scanner;

public class ScannerDemo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        // 从键盘接收数据

        // next方式接收字符串
        System.out.println("next方式接收：");
        // 判断是否还有输入
        if (scan.hasNext()) {
            String str1 = scan.next();
            System.out.println("输入的数据为：" + str1);
        }
        scan.close();
    }
}
```

```
$ javac ScannerDemo.java
$ java ScannerDemo
next方式接收：
runoob com
输入的数据为：runoob
```

nextLine

```
import java.util.Scanner;

public class ScannerDemo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        // 从键盘接收数据

        // nextLine方式接收字符串
        System.out.println("nextLine方式接收：");
        // 判断是否还有输入
        if (scan.hasNextLine()) {
            String str2 = scan.nextLine();
            System.out.println("输入的数据为：" + str2);
        }
        scan.close();
    }
}
```

```
$ javac ScannerDemo.java
$ java ScannerDemo
nextLine方式接收：
runoob com
输入的数据为：runoob com
```

nextInt

```
import java.util.Scanner;

public class ScannerDemo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        // 从键盘接收数据
        int i = 0;
        float f = 0.0f;
        System.out.print("输入整数：");
        if (scan.hasNextInt()) {
            // 判断输入的是否是整数
            i = scan.nextInt();
            // 接收整数
            System.out.println("整数数据：" + i);
        } else {
            // 输入错误的信息
            System.out.println("输入的不是整数！");
        }
        System.out.print("输入小数：");
        if (scan.hasNextFloat()) {
            // 判断输入的是否是小数
            f = scan.nextFloat();
            // 接收小数
            System.out.println("小数数据：" + f);
        } else {
            // 输入错误的信息
            System.out.println("输入的不是小数！");
        }
        scan.close();
    }
}
```

```
$ javac ScannerDemo.java
$ java ScannerDemo
输入整数：12
整数数据：12
输入小数：1.2
小数数据：1.2
```

匿名对象

- 匿名对象：无管理者（无栈内存引用指向它）
- 使用场景：**只需要进行一次方法调用 / 作为参数**传递给函数

```
Student stu = new Student();//stu是对象，名字是stu
new Student();//这个也是一个对象，但是没有名字，称为匿名对象
new Student().show(); //匿名对象方法调用
```

静态

静态成员变量：类加载时就被装载和分配，不依赖于对象而存在 => 通过 类名.变量名 访问
静态成员方法：可以通过 **类名.函数名 调用，只能访问类的静态成员**
静态代码块：仅能定义在类中，一般用于初始化类的静态变量

语句执行顺序

第一次 new 时：**（1）初始化有显式初始化的静态成员变量；（2）顺序执行静态代码块；**（3）初始化有显式初始化的非静态成员变量；（4）顺序执行非静态代码块；（5）调用构造。
之后再 new 时：（1）初始化有显式初始化的非静态成员变量；（2）顺序执行非静态代码块；（3）调用构造。

equals 和 ==

- 比较**对象**的equals和==是等价的，判断**是不是引用的同一个对象。**
- **String**的**equals**只看**字符串内容**是否相等，而==还得看是不是**同一个对象。**（覆盖了Object类的equals()方法，java.io.File、java.util.Date、包装类（如java.lang.Integer和java.lang.Double类等）同理）

```
String str1=new String("Hello");
String str2=new String("Hello");

System.out.println(str1==str2); //打印false
System.out.println(str1.equals(str2)); //打印true
```

```
Integer a = 1;
Integer b = 1;
System.out.println(a == b); // 输出 true
```

这行输出 `true` 是因为 `Integer` 有一个缓存机制，对于 `-128` 到 `127` 之间的整数，`Integer` 会缓存这些值的实例。当你使用 `Integer a = 1;` 和 `Integer b = 1;` 这样的方式创建对象时，`a` 和 `b` 实际上指向了缓存中的同一个 `Integer` 实例。因此，`a == b` 比较的是同一个实例的引用，结果为 `true`。

访问控制修饰符

访问控制分四种类别：

- 公开 `public` 对外公开。
- 受保护 `protected` 向**子类**以及**同一个包中的类**公开。
- 默认 向**同一个包中的类**公开。
- 私有 `private` 只有类本身可以访问，不对外公开

修饰符	同一个类	同一个包	子类	整体
`private`	yeah			
`default`	yeah	yeah		
`protected`	yeah	yeah	yeah	
`public`	yeah	yeah	yeah	yeah

protected

1 包内可见

2 子类可见（子类和父类在同一个包：通过自己访问、通过父类访问。在不同包：仅可通过自己访问。）

若子类与父类**不在同一包**中，那么在子类中

- **子类实例**可以访问其从父类继承而来的 `protected` 方法
- 不能访问**父类实例**的 `protected` 方法
- 不能通过**另一个子类引用**访问共同基类的 `protected` 方法。

```
// 父类
package com.protectedaccess.parentpackage;

public class Parent{
        protected String protect = "protect field";
        protected void getMessages(){
                System.out.println("i am parent");
        }
}
```

```
// 不同包下，可以访问其从父类继承而来的`protected`方法
package com.protectedaccess.parentpackage.sonpackage1;
import com.protectedaccess.parentpackage.Parent;
public class son1 extends Parent {
    public static void main(String[] args) {
        Son1 son1 = new Son1();
        son1.getMessage();  // 输出：i am parent
```

```
    }
    private void message() {
        getMessage();  // 如果子类重写了该方法，则输出重写方法中的内容
        super.getMessage();  // 输出父类该方法中的内容
    }
}
```

```
// 不同包下，不能访问**父类实例**的`protected`方法
package com.protectedaccess.parentpackage.sonpackage1;
import com.protectedaccess.parentpackage.Parent;
public class son1 extends Parent {
        public static void main(String[] args) {
                Parent parent1 = new Parent();
                // parent1.getMessage(); // 错误
                Parent parent2 = new Parent();
                // parent2.getMessage(); // 错误
        }
}
```

```
// 不同包下，不能通过**另一个子类引用**访问共同基类的`protected`方法。
package com.protectedaccess.parentpackage.sonpackage2;
import com.protectedaccess.parentpackage.Parent;
public class son2 extends Parent {

}

package com.protectedaccess.parentpackage.sonpackage1;
import com.protectedaccess.parentpackage.Parent;
import com.protectedaccess.parentpackage.sonpackage2.Son2;
public class son1 extends Parent {
    public static void main(String[] args) {
        Son2 son2 = new Son2();
        // son2.getMessage();  // 错误
    }
}
```

3 protected 的 static 成员对所有子类可见。

对于`protected`修饰的静态变量，无论是否同一个包，在子类中均可直接访问；在不同包的非子类中则不可访问。

```
// 父类
package com.protectedaccess.parentpackage;

public class Parent{
```

```
        protected String protect = "protect field";
        protected static void getMessages(){
                System.out.println("i am parent");
        }
}
```

```
// 无论是否同一个包，在子类中均可直接访问
package com.protectedaccess.parentpackage.sonpackage1;
import com.protectedaccess.parentpackage.Parent;

public class son3 extends Parent {
    public static void main(String[] args) {
        Parent.getMessage();    // 输出：i am parent
    }
}
```

```
// 在不同包的非子类中则不可访问
package com.protectedaccess.parentpackage.sonpackage1;
import com.protectedaccess.parentpackage.Parent;

public class son4 {
    public static void main(String[] args) {
        // Parent.getMessage();    // 错误
    }
}
```

继承

继承（Inheritance）

继承虽好，但也破坏了封装性。子类(派生类)拥有父类(基类/超类)的**所有属性和方法**
(不过私有的属性和方法不能直接访问)

构造方法不能继承

子类构造方法必须最先调用父类构造。

new 出子类对象时，构造方法按照继承链从上往下依次调用。

变量隐藏：子类隐藏父类的同名变量。

继承：super vs this

共性

（1）出现在实例方法和构造方法中
（2）不能出现在静态方法中
（3）大部分情况下是可以省略的
（4）this(), super() 只能出现在构造方法的第一行（若没有 自动补上无参构造方法）

super

（1）当在子类对象中，子类想访问父类的东西，可以使用“super.”的方式访问
（2）当子类构造方法的第一行执行super()无参数方法，那么父类中一定要有无参数构造方法
（当一个类的构造方法是显式的有参数的，会替代原本的无参构造方法，这个时候如果子类执行super()无参方法就会报错，一个好的习惯是要补上无参的构造方法）

继承：方法重写 / 覆盖（Override）

条件

（1）子类中方法的返回值必须与父类的相同、或是其子类。
（2）**方法名、形参列表**完全相同。
（3）**访问权限**不能更低，只能相同或更高。
（4）抛出异常的范围不能更大。

注意事项

（1）**private 方法**不能继承，所以不能覆盖，只有隐藏。
（2）**构造方法**不能继承，所以不能覆盖。
（3）**静态方法**不存在覆盖，只有隐藏。
（4）**final 修饰**的是最终方法，不能覆盖。

子类构造函数

1. 构造函数不能被继承：
   `Child` 类不会继承 `Parent` 类的构造函数，而是需要定义自己的构造函数。
2. 无参子类构造函数的编写：子类可以通过 `super()` **显方式**调用父类无参的构造函数，也可以**隐式**调用

3. 有参子类构造函数的编写：初始化父类的成员变量；初始化子类的成员变量；必须**显式**调用父类有参构造函数
4. `this` 和 `super` 调用必须是第一条可以执行语句

子类对象的生成

1. 创建子类对象时，子类总是按**层次结构从上到下**的顺序调用所有超类的构造函数。如果继承和组合联用，要先构造基类的构造函数，然后调用组合对象的构造函数（组合按照声明的顺序调用）。
2. 如果父类没有不带参数的构造方法，则在子类的构造方法中必须明确的告诉调用父类的某个带参数的构造方法，通过**super关键字**，这条语句还必须出现在构造方法的第一句。
3. 子类创建对象时，子类的构造方法总是**先调用父类**的某个构造方法，完成**父类部分的创建**；然后再调用**子类**自己的构造方法，完成**子类部分的创建。**
4. 如果子类的构造方法没有明显地指明使用父类的哪个构造方法，子类就调用父类的**不带参数**的构造方法 。
5. 子类在创建一个子类对象时，不仅子类中声明的成员变量被分配了内存，而且父类的所有的成员变量也都分配了内存空间。

隐藏和覆盖

- 变量隐藏：在子类对父类的继承中，如果子类的成员变量和父类的成员变量**同名**，此时称为子类隐藏（override）了父类的成员变量。
  - 子类若要引用父类的同名变量。要用 `super` 关键字做前缀加圆点操作符引用, 即 `super.变量名`。
- 方法覆盖：名称、参数、返回类型与父类方法完全相同
  - **私有方法、静态方法不能被覆盖**
  - 用 `fianl` 声明的成员方法是最终方法，**最终方法不能被子类覆盖**（重写）
- 方法隐藏：名称相同

例

```
public class other {
    public static void main(String[] args) {
        Subclass s = new Subclass();
        System.out.println(s.x + " " + s.y + " " + s.z);
        System.out.println(s.method());
        Base b2 = s;         // 正确
        Base b3 = (Base)s;   // 正确
        System.out.println(b2.x + " " + b2.y + " " + b2.z);
        System.out.println(b2.method());
    }
}

class Base {
    int x = 1;
```

```
    static int y = 2;
    int z = 3;

    int method() {
        return x;
    }
}

class Subclass extends Base {
    int x = 4;
    int y = 5;
    static int z = 6;

    int method() {
        return x;
    }
}
```

【输出】

```
4 5 6
4
1 2 3
4
```

```
class Planet{
    public static void hide(){
        System.out.println("The hide method in Planet");
    }
    public void override(){
        System.out.println("The override method in Planet");
    }
}

public class Earth extends Planet{
    public static void hide(){
        System.out.println("The hide method in Earth");
    }
    public void override(){
        System.out.println("The override method in Earth");
    }
    public static void main(String[] args) {
        Earth myEarth = new Earth();
        Planet myPlanet = (Planet) myEarth;
        myPlanet.hide();
        myPlanet.override();
    }
}
```

```
【输出】
The hide method in Planet
The override method in Earth
```

1. `myPlanet.hide();` 调用的是 `Planet` 类的静态方法 `hide`。由于静态方法是绑定到类而不是对象的，所以即使 `myPlanet` 引用的是一个 `Earth` 类型的对象，它仍然会调用 `Planet` 类的 `hide` 方法。因此，输出是 “The hide method in Planet”。
2. `myPlanet.override();` ：由于 `myPlanet` 是一个 `Planet` 类型的引用，但它指向的是一个 `Earth` 类型的对象，它将调用 `Earth` 类中覆盖的 `override` 方法。这是因为 `override` 是一个非静态方法，Java 使用动态绑定（也称为运行时绑定）来确定调用哪个方法。如果 `myPlanet` 实际上指向的是一个 `Earth` 对象，那么调用 `override` 方法时，会调用 `Earth` 类中的版本，输出是 “The override method in Earth”。

多态

- **静多态**：在**编译**时决定调用哪个方法；**方法重载、方法隐藏**
- **动多态**：在**运行**时才能确定调用哪个方法；**方法覆盖**
  - 3个条件：**继承、覆盖、向上转型**（必须由父类的引用指向派生类的实例，并且通过父类的引用调用被覆盖的方法）
  - **方法覆盖(Override)**
    - 方法名、参数个数、参数类型及参数顺序必须一致
    - **异常抛出范围**：子类 $\leq$ 父类
    - **访问权限**：子类 $\geq$ 父类
    - **私有方法、静态方法不能被覆盖**，如果在子类出现了同签名的方法，那是方法隐藏；
- **多态中成员变量编译运行看左边，多态中成员方法编译看左边，运行看右边**

抽象类

- 抽象类
  - 抽象类引用：虽然**不能实例化抽象类**，但可以**创建它的引用**。因为Java支持多态性，允许通过父类引用来引用子类的对象。
  - 如果一个类里有抽象的方法，则这个类就必须声明成抽象的。**但一个抽象类中却可以没有抽象方法。**
- 抽象方法
  - 不能被**private、final或static**修饰。

接口

- 接口：不相关类的功能继承。
  - 只包含**常量**（所有变量默认 `public static final`）和**方法**（默认 `public abstract`）的定义，没有方法的实现。（但一般不包含变量）
  - **没有构造方法**

    - 一个类可以实现多个接口；如果类没有实现接口的全部方法。需要被定义成 `abstract` 类
    - 接口可以继承，而且可以多重继承
        - 同一个函数只能实现一次
        - 不同接口的同名变量相互隐藏
        - 接口变量和类中成员同名时，存在作用域问题
- 关键词 `interface` `implements`
- 接口回调
    - 把实现某一接口的类创建的**对象引用**赋给该接口声明的**接口变量**
    - 该接口变量就可以**调用被类实现的接口中的方法。**
    - 即：
      `接口变量 = 实现该接口的类所创建的对象；`
      `接口变量.接口方法([参数列表])`;

例

```
interface Runner {
        //接口 1
        public void run();
}

interface Swimmer {
        //接口 2
        public void swim();
}

abstract class Animal {
        public abstract void eat();
}

class Person extends Animal implements Runner,Swimmer { //继承类，实现接口
        public void run() {
                System.out.println("我是飞毛腿,跑步速度极快!");
        }
        public void swim(){
                System.out.println("我游泳技术很好,会蛙泳、自由泳、仰泳、蝶泳 ... ");
        }
        public void eat(){
                System.out.println("我牙好胃好,吃啥都香!");
        }
}

public class InterfaceTest{
```

```
        public void m1(Runner r) { r.run(); } //接口作参数
        public void m2(Swimmer s) {s.swim();} //接口作参数
        public void m3(Animal a) {a.eat();} //抽象类引用
        public static void main(String args[]){
                InterfaceTest t = new InterfaceTest();
                Person p = new Person();
                t.m1(p); //接口回调
                t.m2(p); //接口回调
                t.m3(p); //接口回调
        }
}
```

抽象类与接口

- 区别
  - 接口中的成员**变量和方法**只能是 `public` 类型的，而抽象类中的成员变量和方法可以处于各种访问级别。
  - 接口中的**成员变量**只能是 `public`、`static` 和 `final` 类型的，而在抽象类中可以定义各种类型的实例变量和静态变量。
  - 接口中没有**构造方法**，抽象类中有构造方法。接口中所有方法都是抽象方法，抽象类中可以有，也可以没有抽象方法。抽象类比接口包含了更多的实现细节。
  - 抽象类是某一类事物的一种抽象，而接口不是类，它**只定义了某些行为**；
    例如，“生物”类虽然抽象，但有“狗”类的雏形，接口中的run方法可以由狗类实现，也可以由汽车实现。
  - 在语义上，接口表示更高层次的抽象，声明系统对外提供的服务。而抽象类则是各种具体类型的抽象。

upcasting 和 downcasting

向上转型 upcasting

- 向上转型：父类类型 变量名 = new 子类类型();
  如：Person p = new Student();

- 上转型对象的使用(父类有的就能访问，没有的不能访问，且儿子的优先级更高)
  - 上转型对象可以**访问子类继承或隐藏的成员变量**，也可以**调用子类继承的方法或子类重写的实例方法。**
  - 如果子类重写了父类的某个实例方法后，当用上转型对象调用这个实例方法时一定是**调用了子类重写的实例方法。**
  - 上转型对象**不能操作子类新增**的成员变量；**不能调用子类新增**的方法。

向下转型 downcasting

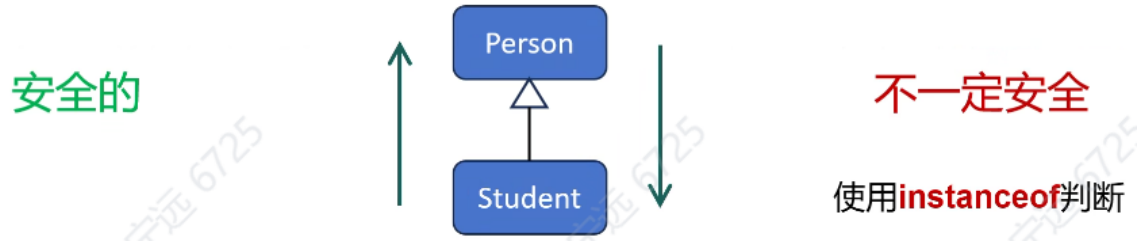
- 向下转型(映射)：一个**已经向上转型的子类对象**可以使用强制类型转换的格式，**将父类引用转为子类引用**，这个过程是向下转型。
  子类类型 变量名 = (子类类型) 父类类型的变量;

如：Person p = new Student();
Student stu = (Student) p

- 如果是**直接创建父类对象，是无法向下转型的**，能过编译，但运行时会产生异常
  如：Person p = new Peron();
  Student stu = (Student) p

instanceof 操作符

- `instanceof` 操作符用于判断一个引用类型所**引用的对象是否是一个类的实例。**
- 形式如下：`obj instanceof ClassName` 或者 `obj instanceof InterfaceName`
- `a instanceof X`，当 X 是 **A类/A类的直接或间接父类/A类实现的接口**时，表达式的值为true



Object类

`public final Class<?> getClass(){}`

- 该方法用于获取对象运行时的字节码类型，得到该对象的**运行时的真实类型。**
- 通常用于判断**两个引用中实际存储对象类型**是否一致。
- 最主要应用：该方法属于Java的反射机制，其返回值是Class类型，例如 Class c = obj.getClass();。通过对象c，
  - 获取所有**成员方法，每个成员方法都是一个Method对象。** `Method[] methods = cls.getDeclaredMethods();`
  - 获取所有**成员变量，每个成员变量都是一个Field对象。** `Field[] fields = cls.getDeclaredFields();`
  - 获取所有**构造函数，构造函数则是一个Constructor对象。** `Constructor<?>[] constructors = cls.getDeclaredConstructors();`

`equals()` 与 `hashCode`

equals为true与hashCode相同的关系?

- 如果两个对象的equals()结果为**true**，那么这两个对象的**hashCode()一定相同**；
- 两个对象的**hashCode()结果相同**，并**不能代表**两个对象的equals()一定为true

`public String toString(){}`

默认的 `toString()` 输出**包名**加**类名**和**堆上的首地址**

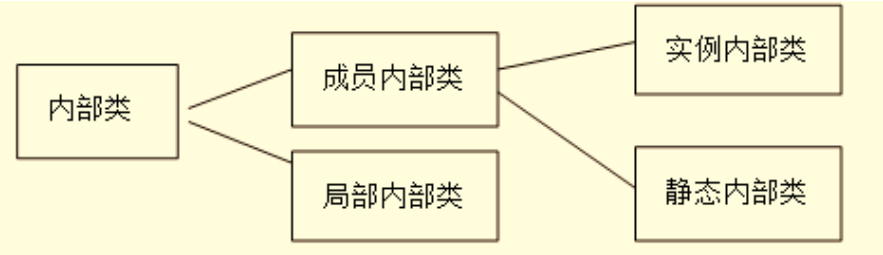
`Objects` 与 `Object` 区别

- **Objects是Object 的工具类。**被final修饰不能被继承，拥有私有的构造函数。此类包含static实用程序方法，用于**操作对象或在操作前检查某些条件。**
  - **null** 或 null方法；
  - 用于计算**一堆对象的混合哈希代码**；
  - 返回对象的**字符串**（会对null进行处理）；
  - 比较两个对象，以及检查索引或子范围值是否超出范围

final

- 定义类头时，**abstract和final不能同时使用**
- 访问权限为**private的方法默认为final**的
- final既可以修饰**简单数据**类型，也可以修饰**复合数据**类型。
  - 简单数据类型：值不能再变
  - 符合数据类型：引用不能再变，值可以改变
- final常量可以在**声明的同时赋初值**，也可以在**构造函数**中
- 常量既可以是**局部常量**，也可以是**类常量和实例常量。**如果是类常量，在**数据类型前加static修饰**（由所有对象共享）。如果是实例常量，就**不加static修饰。**
  1. 局部常量（Local Constant）： 局部常量是在**方法、构造器或代码块内部**定义的常量
  2. 类常量（Class Constant）： 类常量是在**类的静态初始化块或静态成员变量**中定义的常量。 `public static final`
  3. 实例常量（Instance Constant）： 实例常量是在类的**非静态成员变量**中定义的常量。 `public final`

内部类



实例内部类

在创建实例内部类的实例时，外部类的实例必须已经存在，例如要创建InnerTool类的实例，必须先创建Outer外部类的实例
两种语法：

- `Outer.InnerTool tool=new Outer().new InnerTool();`
- `Outer outer=new Outer();`
  `Outer.InnerTool tool =outer.new InnerTool();`
  以下代码会导致编译错误：`Outer.InnerTool tool=new Outer.InnerTool();`

- 在内部类中，可以**直接访问外部类的所有成员**，包括成员变量和成员方法。
- 实例内部类的实例**自动持有外部类的实例的引用。**

例

```
public class A {
    private int a1;
    public int a2;
    static int a3;
    public A(int a1, int a2) {
        this.a1 = a1;
        this.a2 = a2;
    }
    protected int methodA() {
        return a1 * a2;
    }
    class B{  //内部类
        int b1=a1;  //直接访问private的a1
        int b2=a2;  //直接访问public的a2
        int b3=a3;  //直接访问static的a3
        int b4=new A(3,4).a1;  //访问一个新建的实例A的a1
        int b5=methodA();  //访问methodA()方法
      }
      public static void main(String args[]){
        A.B b=new A(1,2).new B();
        System.out.println("b.b1="+b.b1);  //打印b.b1=1
        System.out.println("b.b2="+b.b2);  //打印b.b2=2
        System.out.println("b.b3="+b.b3);  //打印b.b3=0
        System.out.println("b.b4="+b.b4);  //打印b.b4=3
        System.out.println("b.b5="+b.b5);  //打印b.b5=2
      }
}
```

静态内部类

- 静态内部类的实例**不会自动持有外部类的特定实例的引用**
- 在创建内部类的实例时，**不必创建外部类的实例。**

```
class AA {
    public static class B {
        int v;
    }
}

class Tester {
    public void test() {
        AA.B b = new AA.B();
        b.v = 1;
    }
}
```

- 客户类可以通过**完整的类名直接访问静态内部类的静态成员。**

局部内部类

- 局部内部类只能在**当前方法**中使用。
- 局部内部类和实例内部类一样，可以**访问外部类的所有成员**
- 此外，局部内部类还可以访问**函数中的最终变量或参数**(final)

内部类的用途

- 封装类型：如果**一个类只能由系统中的某一个类访问**，可以定义为该类的内部类。
  - 顶层类只能处于**public和默认**访问级别
  - 而成员内部类可以处于public、protected、默认和private四个访问级别。
- 直接访问外部类的成员
- 回调外部类的方法

  回调实质上是指**一个类(Sub)尽管实际上实现了某种功能(调节温度)，但是没有直接提供相应的接口，客户类可以通过这个类的内部类(Closure)的接口(Adjustable)来获得这种功能。而这个内部类本身并没有提供真正的实现，仅仅调用外部类的实现(adjustTemperature)。**

```
public class Sub extends Base {
  private int temperature;

  private void adjustTemperature(int temperature){
    this.temperature=temperature;
  }

  private class Closure implements Adjustable{
    public void adjust(int temperature){
      adjustTemperature(temperature);
    }
  }
  public Adjustable getCallBackReference(){
    return new Closure();
  }
}
```

- 以下代码演示客户类使用Sub类的调节温度的功能：

```
public class TestClass {
    public static void main(String[] args){
        //调节温度
        Sub sub=new Sub();
        Adjustable ad=sub.getCallBackReference();
        ad.adjust(15);

        //调节速度
        sub.adjust(350);
    }
}
```

接口回调

内部类的文件命名

对于每个内部类，Java编译器会生成独立的.class文件。这些类文件的命名规则如下：

- 成员内部类：外部类的名字$内部类的名字
- 局部内部类：外部类的名字$数字和内部类的名字

- 匿名类：外部类的名字$数字

```
public class AAAAAA {
    static class B {
    } // 成员内部类，对应A$B.class
    class C { // 成员内部类，对应A$C.class
        class D {
        } // 成员内部类，对应A$C$D.class
    }
    public void method1() {
        class E {
        } // 局部内部类1，对应A$1E.class

        B b = new B() {
        }; // 匿名类1，对应A$1.class
        C c = new C() {
        }; // 匿名类2，对应A$2.class
    }
    public void method2() {
        class E {
        } // 局部内部类2，对应A$2E.class
    }
}
```

Java编译器编译以上程序，会生成以下类文件：

A.class

A$B.class

A$C.class

ACD.class

A$1E.class

A$1.class

A$2.class

A$2E.class

匿名类

- 匿名类必须继承自一个具体的**类**或实现一个**接口**。
- 匿名类就是没有名字的类，是将类和类的方法定义在一个表达式范围里。
- 匿名类本身**没有构造方法**，但是会**调用父类的构造方法**。
- 匿名内部类将内部类的定义与生成实例的语句合在一起，并省去了类名以及关键字“class”,”extends”和“implements”等

```java
public class AAAAA {
    AAAAA(int v) {
        System.out.println("another constructor");
    }
    AAAAA() {
        System.out.println("default constructor");
    }
    void method() {
        System.out.println("from AAAAA");
    };
    public static void main(String args[]) {
        new AAAAA().method(); // 打印from AAAAA
        AAAAA a = new AAAAA() { // 匿名类
            void method() {
                System.out.println("from anonymous");
            }
        };
        a.method(); // 打印from anonymous
    }
}
```

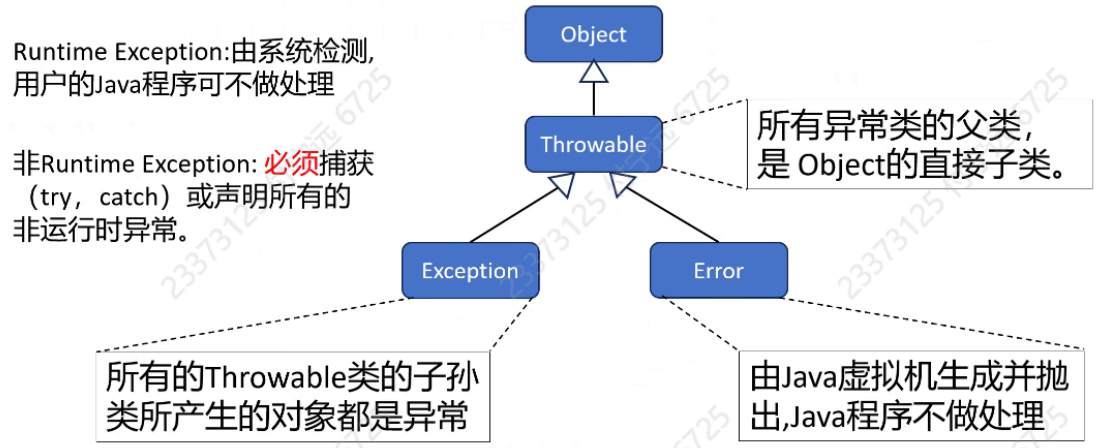
```
default constructor
from AAAAA
default constructor
from anonymous
```

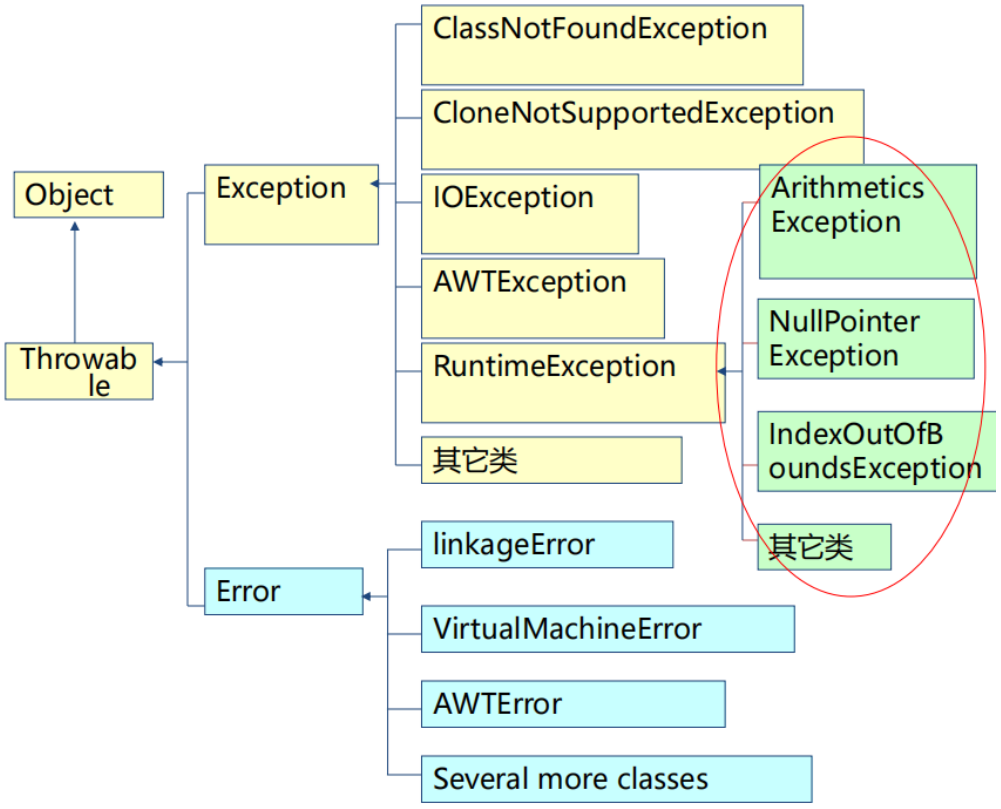
UML

继承：空心三角形 实线
实现：空心三角形 虚线
关联（拥有）：普通箭头 实线
依赖（使用）：普通箭头 虚线

异常处理

异常处理





- Runtime Exception：**编译时不可监测的异常**，由系统检测，用户的Java程序可不做处理，系统将它们交给缺省的异常处理程序。
- **非Runtime Exception**：**编译时可以监测的异常**，Java编译器要求Java程序**必须**捕获或声明所有的非运行时异常，可以通过try-catch或throws处理。
- 非受检异常 (unchecked exception)： Runtime Exception 及其子类、Error 及其子类。
  - 只能在程序执行时被检测到，**不能在编译时被检测到**；
  - 程序可不处理，交由系统处理。
- 受检异常 (checked exception)：除了非受检异常之外的异常（即其他的异常类都是可检测的类）
  - 这些异常在**编译时**就能被java编译器所检测到异常。

- 必须采用 **throws 语句**或者 **try-catch** 方式处理异常

throws在方法声明处声明抛出特定异常。（只抛出不处理，交给调用该方法的方法进行处理，若一直不处理，则交给系统处理）

- 声明异常的格式

```
<访问权限修饰符><返回值类型><方法名>(参数列表) throws 异常列表{}
```

- **当父类中的方法没有throws，则子类重写此方法时也不可以throws。**若重写方法中出异常，必须采用try结构处理。
- 重写方法不能抛出比被重写方法**范围更大**的异常类型，子类重写方法也**可以不抛出**异常。

try-catch捕获并处理异常。（处理）

- `try{}` 中执行 `return` ，`finally` 语句仍然执行，在 `return` 前执行
- `try{}` 中执行 `exist(0)`，`finally` 语句不执行
- 一般 `finally` 写**释放资源**的部分（打开水龙头后无论水龙头能不能使最后都要关上水龙头）
- `catch(异常名1|异常名2|异常名3 变量)` 中异常必须是**同级**关系；

`try-with-resourse`

- 资源：所有实现Closeable的类，如流操作，socket操作，httpclient等
- 带资源的try语句（try-with-resource）的最简形式为：

```
try(Resource res = xxx){ // 可指定多个资源
        work with res
}
```

- 处理规则
  - 凡是实现了AutoCloseable接口的类，在try()里声明该类实例的时候，在try结束后，**close方法都会被调用**，这个动作会**早于finally里调用的方法。**
  - 不管是否出现异常，try()里的实例都会被调用；
  - close方法**越晚声明**的对象，会**越早被close**掉。

throw抛出具体的异常。（自定义异常）

- throw抛出**用户自定义异常。**
- 用户定义的异常必须由**用户自己抛出** `<throw><异常对象>` `throw new MyException`
- 程序会在throw语句处立即终止，转向 try…catch 寻找异常处理方法。
- 语句格式

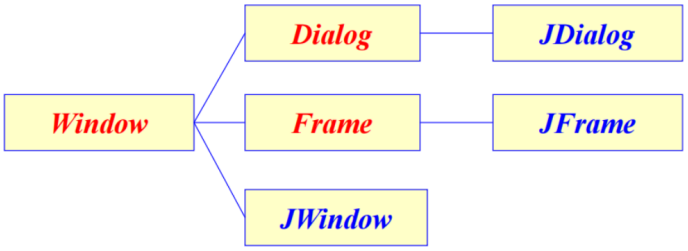
```
<class> <自定义异常名> extends <Exception>{
        public String tostring(){
                return "myException";
        }
}
```

图形界面

小结(事件处理)

- 确认触发的事件，取得事件类（如ActionEvent）的名字，并删掉其中的“Event”，加入“Listener”；
- 实现上面的接口，针对想要捕获的事件编写方法代码；
- 为事件处理器（监听者接口）创建一个对象，让自己的组件和方法完成对它的注册。

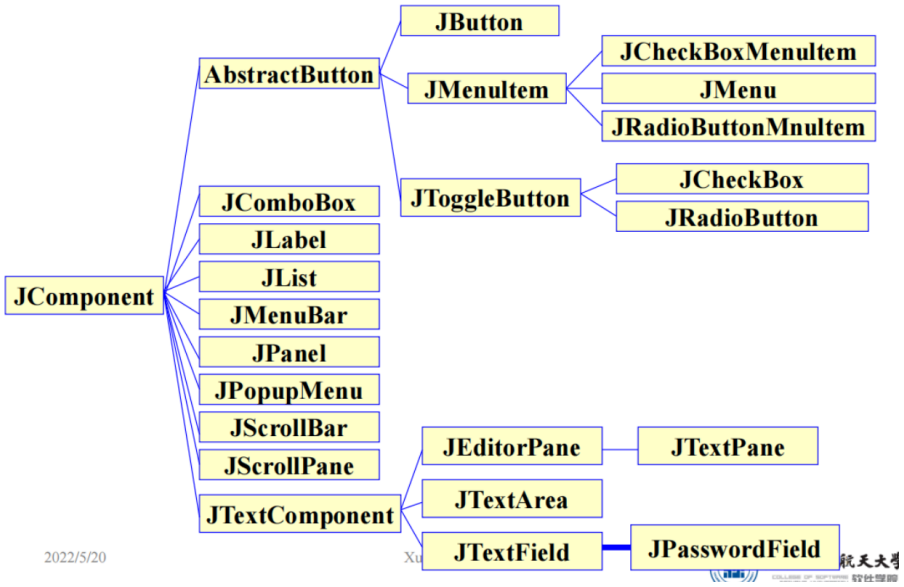
小结：*JFrame and Windows*



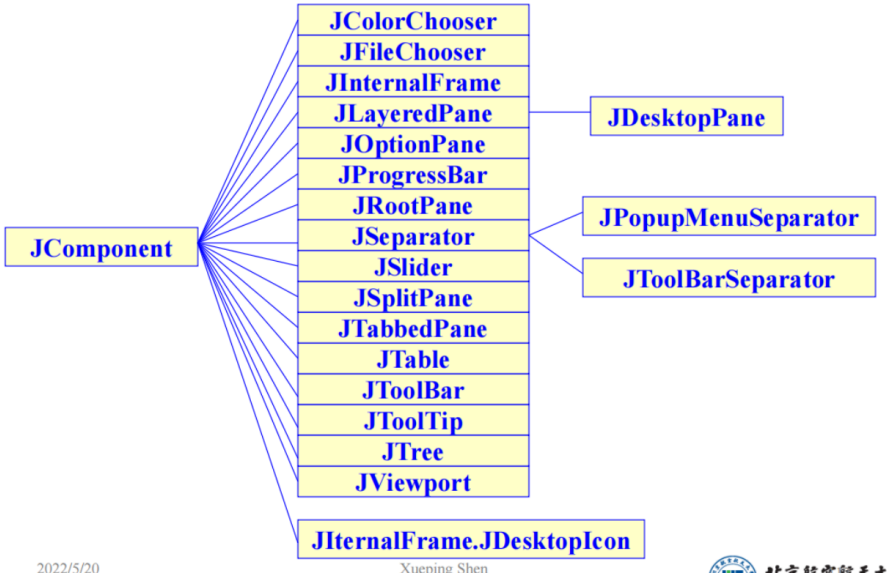
小结：创建容器与组件基本步骤

- 创建顶层容器（常用的为窗口对象）
- 创建内容面板，设置其背景颜色，设置其布局管理器
- 创建普通面板，设置其背景颜色，设置其位置、大小，设置其布局管理器
- 创建组件，设置其背景颜色，设置其位置、大小、字体等
- 将面板添加到窗口，将组件添加到指定面板
- 创建事件监听器，实现事件接口方法
- 给事件源注册监听器

小结：*Component Hierarchy—AWT Similar*



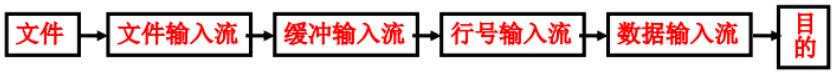
小结：*ComponentHierarchy—New and Expanded Components*



Java IO

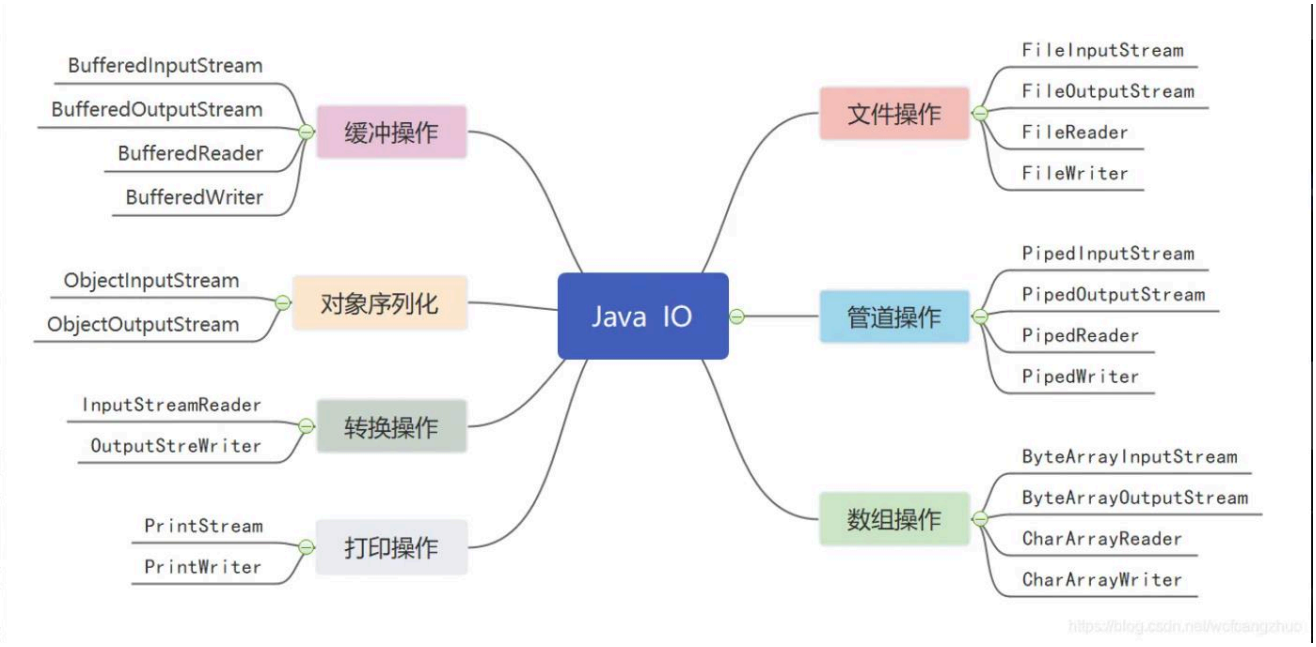
- 输入流：输入数据流只能读,不能写
  - 字节流：Java中的输入数据流(字节流)都是抽象类**InputStream**的子类；
  - 字符流：Java中的输入数据流(字符流)都是抽象类**Reader**的子类；
- 输出流：输出数据流只能写,不能读
  - 字节流:java中的输出数据流(字节流)都是抽象类**OutputStream**的子类；
  - 字符流:java中的输出数据流(字符流)都是抽象类**Writer**的子类；
- 字节流可以操作所有类型的文件；
- 字符流只能操作**纯文本文件**；

- 在Java中有数据传输的地方都用到I/O流(通常是文件,网络,内存和标准输入输出等)
- Java 的 IO 流主要包括输入、输出两种 IO 流，每种输入、输出流有可分为字节流和字符流两大类：
  - 字节流以字节为单位来处理输入、输出操作
  - 字符流以字符为单位来处理输入、输出操作
- 众多的流对象协同工作



1. 读写的时候字节流是按字节读写，字符流按字符读写。
2. 字节流适合所有类型文件的数据传输，因为计算机字节（Byte）是电脑中表示信息含义的最小单位。字符流只能够处理纯文本数据，其他类型数据不行，但是字符流处理文本要比字节流处理文本要方便。
3. 在读写文件需要对内容按行处理，比如比较特定字符，处理某一行数据的时候一般会选择字符流。
4. 只是读写文件，和文件内容无关时，一般选择字节流。

- File, File(Input/Output)Stream, RandomAccessFile是处理本地文件的类。
- Data(Input/Output)Stream是一个过滤流的子类,借此可以读写各种基本数据, 在文件和网络中经常使用.如: readByte, writeBoolean等。
- Buffered(Input/Output)Stream的作用是在数据送到目的之前先缓存,达到一定数量时再送到目的,以提高程序的运行效率。
- Piped(Input/Output)Stream适合于一个处理的输出作为另一个处理的输入的情况。



IO NIO NIO2

- IO流（同步、阻塞）
- NIO（同步、非阻塞）:NIO(NEW IO)用到块，效率比IO高很多
  三个组件:
  - Channels（通道）：**流是单向的，Channel是双向的**，既可以读又可以写，Channel可以进行异步的读写，对Channel的读写必须通过**buffer**对象
  - Buffer（缓冲区）
  - Selector（选择器）：Selector是一个对象，它可以注册到很多个Channel上，监听各个Channel上发生的事件，并且能够根据事件情况决定Channel读写。这样，通过一个线程管理多个Channel，就可以处理大量网络连接了
- NIO2(异步、非阻塞)：AIO(Asynchronous IO)

同步与异步

- 同步:是一种可靠的有序运行机制，进行同步操作时，后续的任务须**等待当前调用返回**，才会进行下一步；
- 异步：后续任务**不需要等待当前调用返回**，通常依靠事件、回调等机制来实现任务间次序关系；

阻塞与非阻塞

- 阻塞：在进行读写操作时，当前线程会处于阻塞状态，**无法从事其他任务**。只有当条件就绪才能继续；
- 非阻塞：不管IO操作是否结束，**直接返回**，相应操作在后台继续处理

多线程

多线程

BETTER!!!

Thread	Runnable
简单易用	编码较复杂
java不支持多继承，因此有局限性	可以避免单继承的限制

适合资源的共享

更推荐TimeUnit.xxxx.sleep

Sleep	yield
使线程转入阻塞状态	使线程转入runnable状态
不会考虑线程的优先级	相同优先级或更高的线程运行机会

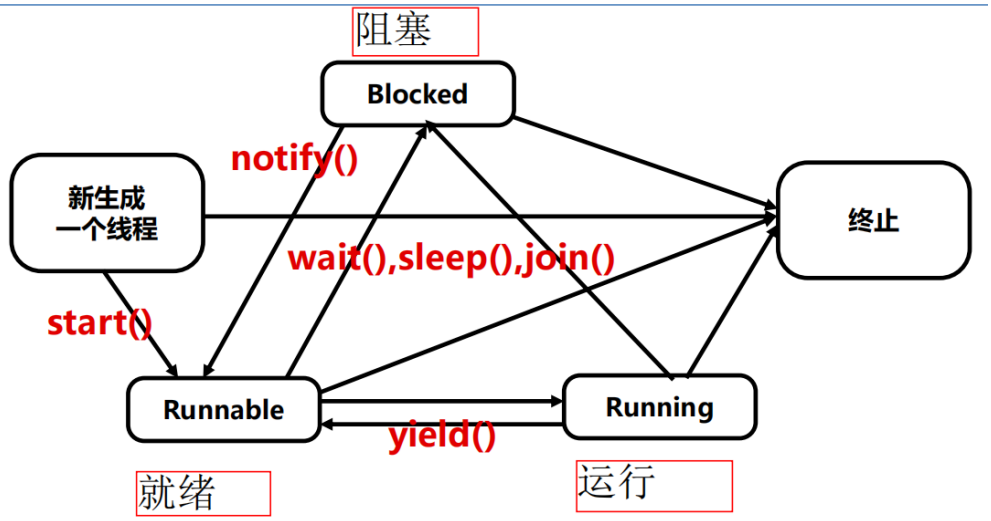
runnable局限性：run()返回值是void；不允许抛出任何已检查的异常——callable借口既有返回值又能抛出异常

wait, notify, notifyAll必须在已经持有锁的情况下执行,所以它们只能出现在**synchronized**作用的范围内，也就是出现在**synchronized**修饰的方法中。

wait的作用:**释放已持有的锁**,进入等待队列. **sleep不释放锁**

notify的作用:**随机唤醒**wait队列中的一个线程并把它移入锁申请队列

notifyAll的作用:唤醒wait队列中的所有的线程并把它们移入锁申请队列



因为Java线程的调度不是分时的，所以你必须确保你的代码中的线程会**不时地给另外一个线程运行的机会**。有三种方法可以做到一点：

- 让处于运行状态的线程调用 **Thread.sleep()** 方法。
- 让处于运行状态的线程调用 **Thread.yield()** 方法。
- 让处于运行状态的线程调用另一个线程的 **join()** 方法。

sleep与yield

- 这两个方法都是静态的实例方法。
- sleep()会有**中断异常抛出**，而yiled()不抛出任何异常。
- sleep()方法具有更好的**可移植性**，因为yield()的实现还取决于底层的操作系统对线程的调度策略。
- 对于yield()的主要用途是在**测试阶段**，**人为的提高程序的并发性能，以帮助发现一些隐藏的并发错误**，当程序正常运行时，则不能依靠yield方法提高程序的并发行能。

wait与sleep

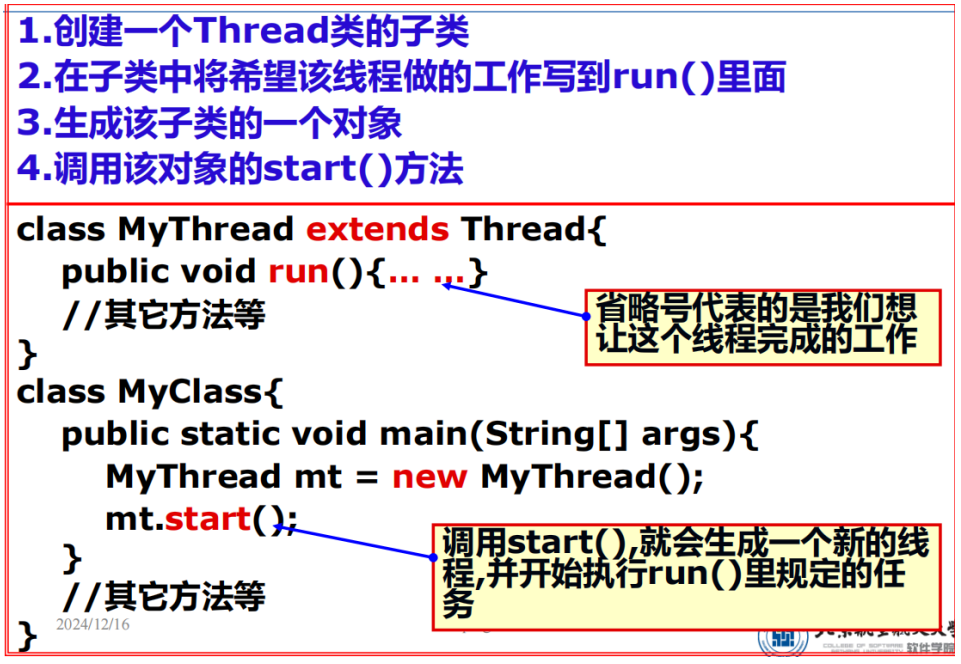
- wait,notify和notifyAll都是与同步相关联的方法,只有在synchronized方法中才可以用。在不同步的方法或代码中则使用sleep()方法使线程暂时停止运行

join

作用：使当前正在运行的线程暂停下来，**等待指定的时间后**或**等待调用该方法的线程结束后**，再恢复运行

创建用户多线程的步骤 4

法1



法2

法3

实现callable接口

1. **创建一个实现Callable接口的类。**
2. 重写类中的**Callable接口的call()方法。**
3. **通过传递实现了Callable接口的类对象来创建FutureTask对象。**
4. **通过传递创建的FutureTask对象在主类中创建一个Thread对象。**
5. 调用**Thread对象的start()方法。**
6. **使用FutureTask对象的get()方法获取可调用类返回的结果。**

```
CallableThread callableThread = new CallableThread();
FutureTask<String> futureTask = new FutureTask<>(callableThread);
Thread thread = new Thread(futureTask);
thread.start();
String result = futureTask.get();
System.out.println("执行结果= " + result);
```

使用 FutureTask.get()方法获取执行结果

借助FutureTask，因为 Thread类没有接受Callable的构造函数。

网络编程

网络编程

- C/S，B/S
- URI，URL，TCP，UDP
- Socket
- …

了解基本概念即可，不会考太难。

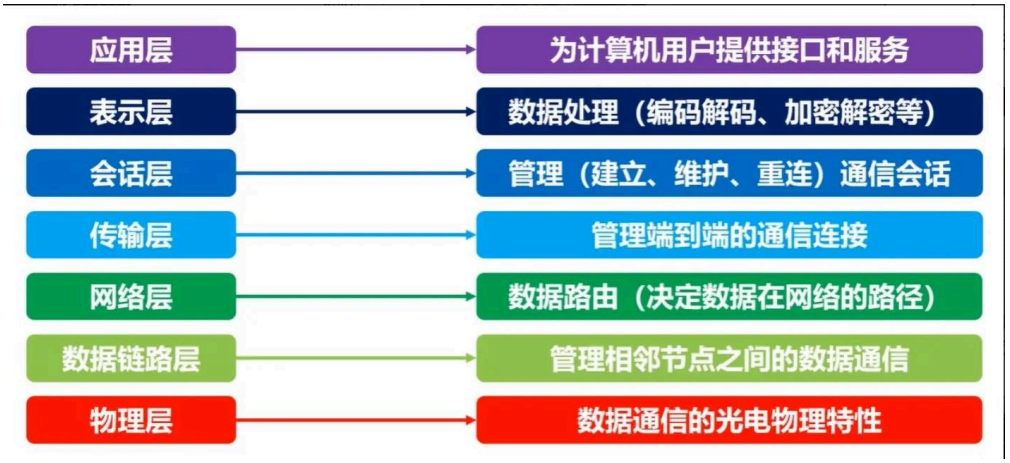
计算机网络工作模式

- 客户机/服务器模式(Client/Server C/S)
  一共两种
  - **数据库服务器端**，客户端**通过数据库连接访问服务器端的数据**；
  - （本讲内容）**Socket服务器端**，服务器端的程序通过Socket与客户端的程序通信。
    另，socket服务器端为“**传输层**”，BS模式为“应用层”
- 浏览器/服务器模式（Browser/Server）

网络通信协议与接口

- 网络通信协议：计算机网络中实现通信必须有一些约定
- 网络通信接口：为了使两个结点之间能进行对话，必须在他们之间建立通信工具(即**接口**)，使彼此之间能进行信息交换，接口包括两部分：
  - 硬件装置:实现结点之间的信息传递。

- 软件装置:规定双方进行通信的约定协议。



URI 包含 URL 和 URN

URI（Uniform Resource Identifier，统一资源**标识**符）用于唯一地**标识**资源，无论是通过名称、位置还是两者兼有。
URL（Uniform Resource Locator，统一资源**定位**符）是URI的一个子集，它提供了资源的**定位**信息，即如何访问资源，但不直接提供资源的名称。
URN（Uniform Resource Name）是URI的另一个子集，它提供了资源的**名称**，但不提供如何定位或访问资源的信息。URN是持久的、与位置无关的标识符。

TCP/IP

- TCP/IP是一组在Internet网络上的不同计算机之间进行通信的协议的总称；从下往上可视为4层结构：*物理层、网络层、传输层和应用层。*
- TCP/IP由应用层的HTTP、FTP、SMTP和传输层的TCP（传输控制协议）、UDP（用户数据报协议）以及网络层的IP（Internet协议）等一系列协议组成。

TCP和UDP

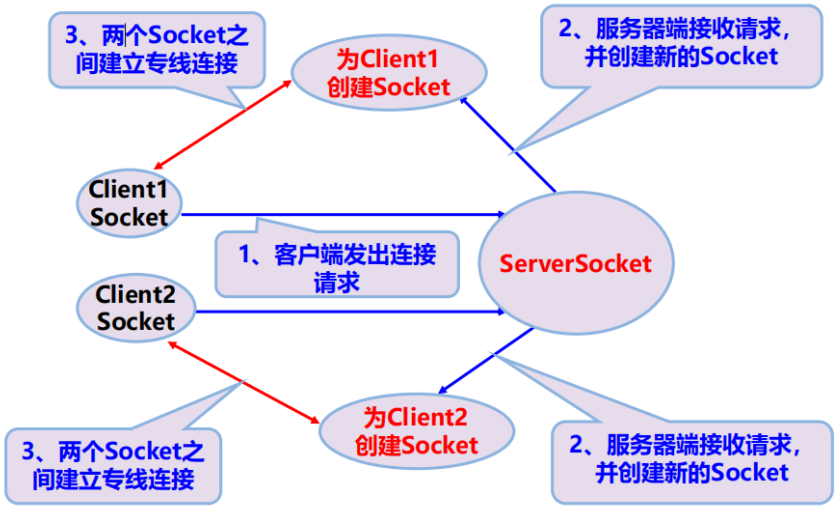
- UDP与TCP
  - TCP，传输控制协议(Transmission Control Protocol)，是面向连接的通信协议。
    - 使用TCP协议进行数据传输时,两个进程之间会建立一个连接,数据以流的形式顺序传输。
  - UDP，用户数据协议(User Datagram Protocol)，是无连接通信协议。
    - 使用UDP协议进行数据传输时,两个进程之间不建立特定的连接,不对数据到达的顺序进行检查。
- 在互联网上进行数据传输,多用TCP和UDP协议，它们传输的都是一个byte stream/字节型的数据流

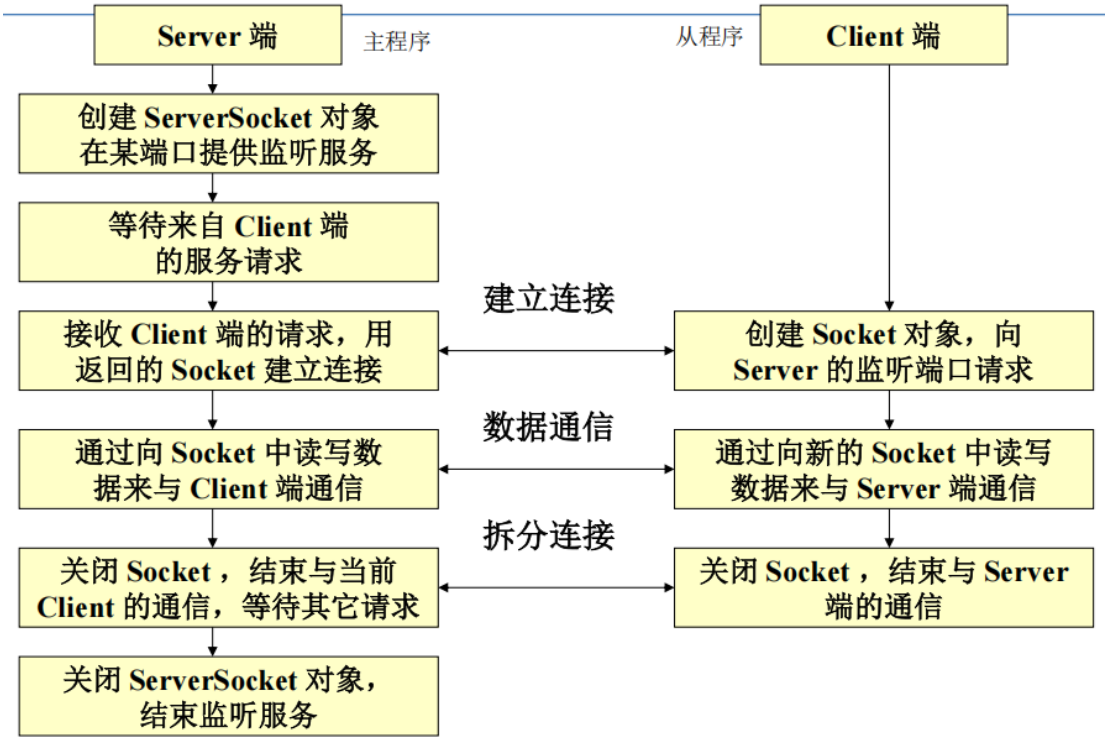
更加稳定可靠，常用来传文件

效率高但容易丢东西

- Socket（套接字）是网络通信的一个端点，它提供了一种机制，允许应用程序通过网络发送和接收数据。
- **Socket与TCP/UDP**：Socket是应用程序与TCP/UDP协议之间的接口，应用程序通过Socket来使用TCP或UDP提供的服务。
- **TCP与UDP**：TCP和UDP都是传输层协议，它们为Socket提供了不同的数据传输服务。TCP提供可靠的服务，而UDP提供高效但不可靠的服务。
- **流式Socket**：用于TCP连接，提供可靠、有序、双向的数据流。
- **数据报Socket**：用于UDP连接，提供不可靠、无序、双向的数据报服务

TCP网络程序的工作原理





在TCP网络连接上传递对象

- ObjectInputStream和ObjectOutputStream可以用来包装底层网络字节流，TCP服务器和TCP客户端之间就可以传递对象类型的数据，实现从底层输入流中读取对象类型的数据和将对象类型的数据写入到底层输出流。
- RMI(remote method invocation)编程：是java进行分布式编程的基础。

UDP网络程序

- 用户数据报协议UDP（user datagram protocol）是一个无连接的、发送独立数据包的协议，它不保证数据按顺序传送和正确到达。
- 数据报Socket又称为UDP套接字，它无需建立、拆除连接，而是直接将信息打包传向指定的目的地，使用简单，占用资源少，适合于断续、非实时通信。
- 利用UDP通信的两个程序是平等的，没有主次之分，两个程序的代码可以完全一样。

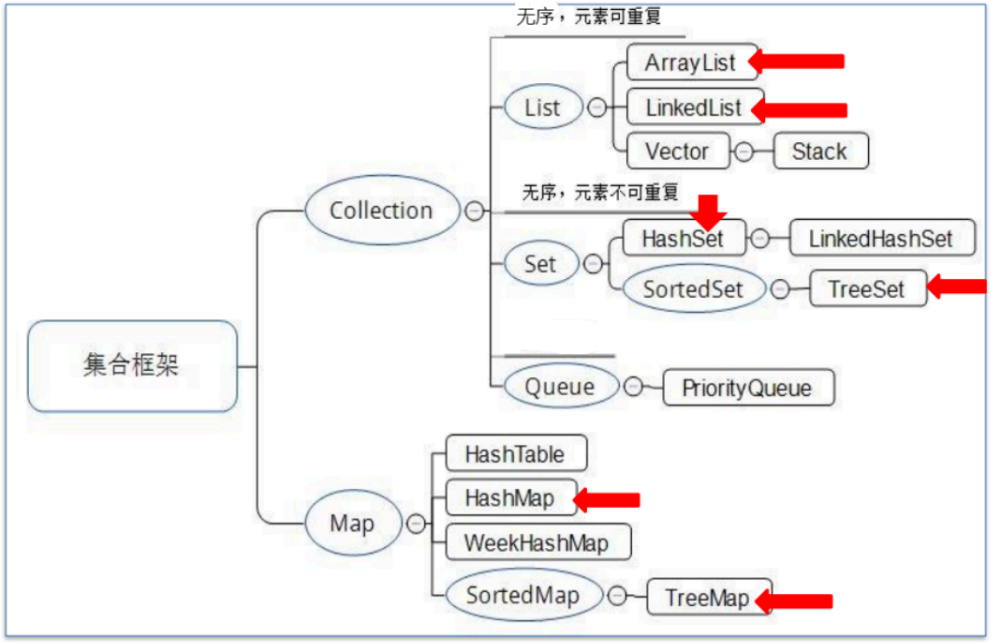
所有集合类都位于 java.util 包下。Java的集合类主要由两个接口派生而出：Collection 和 Map

Set、List 和 Map 可以看做集合的三大类：

Set 接口继承 Collection，无序集合，集合中的元素不可以重复

List 接口继承 Collection，允许重复，插入序

Map 集合中保存 key-value 对形式的元素



普通数组的定义：`int[] a = new int[10];`

ArrayList：无序，可重复，长度可变，**遍历元素和随机访问元素效率较高**

ArrayList

ArrayList<String> sites = new ArrayList<String>();

添加元素：sites.add(“str”)

访问元素：sites.get(index)

修改元素：sites.set(2, "Wiki");

删除元素：sites.remove(index);

使用Collections排序：Collections.sort(sites); // 默认数字/字母排序

数组大小：`site.size()`

LinkedList：无序，可重复，FIFO，**插入删除元素效率较高**

方法名	说　明
void　addFirst(Object o)	在列表的首部添加元素
void　addLast(Object o)	在列表的末尾添加元素
Object getFirst()	返回列表中的第一个元素
Object getLast()	返回列表中的最后一个元素
Object removeFirst()	删除并返回列表中的第一个元素
Object removeLast()	删除并返回列表中的最后一个元素

```
public class Test3 {
    public static void main(String[] args) {
        Dog ououDog = new Dog("欧欧", "雪娜瑞");
        Dog yayaDog = new Dog("亚亚", "拉布拉多");
        Dog meimeiDog = new Dog("美美", "雪娜瑞");
        Dog feifeiDog = new Dog("菲菲", "拉布拉多");

        LinkedList<Object> dogs = new LinkedList<Object>();
//      LinkedList<Dog> dogs = new LinkedList<Dog>();
        dogs.add(ououDog);
        dogs.add(yayaDog);
        dogs.addFirst(meimeiDog);  //// 添加meimeiDog到指定位置
        dogs.addLast(feifeiDog); // 添加feifeiDog到指定位置

        System.out.println("共计有" + dogs.size() + "条狗狗。");

        System.out.println("分别是：");
        for (int i = 0; i < dogs.size(); i++) {
            Dog dog = (Dog) dogs.get(i);
            System.out.println(dog.getName() + "\t" + dog.getStrain());
        }
```

```
Dog dogFirst= (Dog)dogs.getFirst();
System.out.println("第一条狗狗昵称是"+dogFirst.getName() );

Dog dogLast= (Dog)dogs.getLast();
System.out.println("最后一条狗狗昵称是"+dogLast.getName());

dogs.removeFirst();
dogs.removeLast();
System.out.println("共计有" + dogs.size() + "条狗狗。");

System.out.println("分别是：");
for (int i = 0; i < dogs.size(); i++) {
    Dog dog = (Dog) dogs.get(i);
    System.out.println(dog.getName() + "\t" + dog.getStrain());
}
```

HashSet：无序，不可重复

HashSet

HashSet<String> sites = new HashSet<String>();

添加元素：sites.add(“str”)

判断元素是否存在：sites.contains("success")

删除元素：sites.remove("success");

使用Collections排序：先转成List再用ArrayList的方法排序

List<String> sortedList = new ArrayList<>(sites);

*也可以是key-value的，比如HashSet<String, String>，第一个做key，第二个做value

HashMap：无序，键（Key）不能重复，值（Value）可以重复

方法名	说　明
Object put(Object key, Object val)	以“键-值对”的方式进行存储
Object get (Object key)	根据键返回相关联的值，如果不存在指定的键，返回null
Object remove (Object key)	删除由指定的键映射的“键-值对”
int size()	返回元素个数
Set keySet ()	返回键的集合
Collection values ()	返回值的集合
Boolean containsKey (Object key)	如果存在由指定的键映射的“键-值对”，返回true

重写排序

如果 `a` 是 list：`Collection.sort(a)`

如果 `a` 是普通数组：`Arrays.sort(a)`

```
class Person implements Comparable<Person>{
    String name;
    int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
@Override
    public int compareTo(Person person) {
        return this.age - person.age;
    }
}
```

```
import java.util.Arrays;
public class Test {
   public static void main(String[] args) {
       Person[] persons
= new Person[]{new Person("tom", 20), new Person("jack", 12)};
       System.out.print("排序之前： ");
       for (Person p : persons) {
          System.out.print("name:" + p.name + ", age:" + p.age + " ");
       }
       // 排序之后
       Arrays.sort(persons);
       System.out.println("排序之后:");
       for (Person p : persons) {
          System.out.print("name:" + p.name + ", age:" + p.age + " ");
       }
```


例

1. 关于Java类LinkedList的特点，下面描述正确的是（）_____
A 查询快
B 增删快
C 元素不重复
D 元素自然排序

正确答案: B

10. 下面Java中关于List、Set的说法正确的是（）_____
A List 接口存储一组唯一，有序的对象
B Set 接口存储一组唯一，有序的对象
C Set可以允许插入多个null值
D ArrayList和Vector都继承自List

正确答案: D

设计原则(7个) 7

SOLID合成复用

1. S **Single Responsibility Principle (单一职责原则)**：每个类只干一件事
2. O **Open/Closed Principle (开闭原则)**：对扩展开放对修改关闭；用抽象类和接口而不是if-else
3. L **Liskov Substitution Principle (里氏代换原则)**：（子类能够替换父类对象）子类不能改变父类的方法
4. I **Interface Segregation Principle (接口隔离原则)**：把总接口拆分成多个接口（防止有的类不需要某个功能但被迫实现）
5. D **Dependency Inversion Principle (依赖倒转原则)**：细节(更具体的东西，如email通信)实现抽象(更抽象的东西，如通信)而不是抽象拥有细节
6. **迪米特法则**：一个软件实体尽量少的与其他实体发生相互作用（找中介）
7. **合成复用原则**：少用继承，用组合/聚合代替继承

设计模式

工厂方法模式的核心是把类的**实例化延迟到其子类**
被造的东西有个接口、工厂有个接口，被造的东西和工厂分别实现这两个接口，然后工厂类
`public Vehicle createVehicle() { return new Car();}`
适配器模式的核心是将一个类的接口**转换**成客户希望的另外一个接口
两个接口：原来的东西和新加的东西。类来实现新加的东西的接口。Adapter实现原来的东西的接口，同时拥有新加的东西的接口。
装饰模式的核心是**动态**地给对象添加一些额外的职责。
具体组件继承抽象组件；抽象装饰继承抽象组件，同时拥有抽象组件；具体装饰继承抽象装饰，有装饰函数；调用时 `Bird bird = new Sparrow; bird = new`

`birdDecorator(bird)`

外观模式的核心是通过为多个复杂的子系统提供一个**一致的接口**，而使这些子系统**更加容易被访问**的模式。

外观角色拥有子系统123；客户角色依赖外观角色。

策略模式的核心是定义**一系列算法**,把它们一个个**封装**起来,并且使它们可**相互替换**。本模式使得算法可独立于使用它的客户而变化。

上下文拥有抽象策略，同时在方法体内调用策略的算法；具体策略实现抽象策略

访问者模式的核心是在**不改变各个元素的类**的前提下定义作用于这些元素的**新操作**。

元素、访问者接口；具体元素实现元素接口；具体访问者实现访问者接口；具体元素和集体访问者相互关联；元素接受访问者的访问后访问者访问元素

责任链模式的关键是将用户的请求**分派给许多对象**。

处理者接口规定具体处理者**处理用户的请求的方法**以及具体处理者**设置后继对象**的方法；具体处理者实现处理者接口；使用时先设置后继对象在调用第一个处理者

观察者模式的核心是当一个对象改变状态时，所有依赖于它的对象都会得到通知并自动更新。

被观察者存了一个list表示观察者，观察者存了自己观察的对象。当被观察者发生变化时，通知观察者，观察者更新数据并展示出来

原型模式

原型借口实现 `cloneable`，具体原型实现原型接口，实现 `clone` 方法 `return (TextDocument) super.clone();` ；调用时 `TextDocument copiedDocument = (TextDocument) originalDocument.clone()`

- **饿汉式**：类加载时就创建实例，像是一个饥饿的人急于吃东西。
- **懒汉式**：使用时才创建实例，像是一个懒惰的人等到需要时才行动。

饿汉式

是否 Lazy 初始化：否

是否多线程安全：是

常用，但容易产生垃圾对象。

优点：没有加锁，执行效率会提高。

缺点：类加载时就初始化，浪费内存。

特点：避免了多线程的同步问题

懒汉式

是否 Lazy 初始化：是

是否多线程安全：否

不支持多线程。因为没有加锁 synchronized