包

声明包

包名的规范

> 包名应尽量使用纯小写英文字母组成 .作为路径分隔符使用 所以.不能用在开头或结尾位置

```
//声明包必须是java源文件中的第一行非注释性代码
package 包名;
```

> 如果类文件 直接属于源文件夹 则无需在类文件中 声明包
>
> idea中 需要通过 标记来区分项目中的文件夹作用 右键菜单中 有 mark directory as-->sources root

导包

> 一个java的源文件 可以导入多个其他类的引用 导包是为了 降低代码的编写复杂度而设计的

```
public class Test{
    public static void main(String[] args){
        //未进行导包时的写法
        java.util.Random ran=new java.util.Random();
    }
}

//当前类中使用的Random 就是java.util.Random
import java.util.Random;
public class Test{
    public static void main(String[] args){
        //进行导包时的写法
        Random ran=new Random();
    }
}
```

> 一个java源文件 访问自己所在包中的其他类文件 无需导包 跨包访问 才需要进行导包操作
>
> 如果在同一个源文件中 需要访问来自其他包的 同名类文件 最多只能导入一个包 另一个则需要使用完整的 包名.类名进行访问

```
//导入包
import java.util.Date;
//import java.sql.Date;//导致Date引用不明 出现报错
public class Test {
    public static void main(String[] args) {
        Date d1=new Date();//有导包的Date 实例化对象
        java.sql.Date d2=new java.sql.Date(d1.getTime()); //没有导包的Date 实例化对象
    }
}
```

权限修饰符

修饰符	同一 个类中	同一个包中其他类	不同包下的子类	不同包下的无关类
private	√			
缺省	√	√		
protected	√	√	√	
public	√	√	√	√

final 修饰符的应用

> final 代表终态,最终 是一个较为常用的 成员修饰符 也可以用于修饰类 与abstract 属于 矛盾修饰符

1. final 修饰 类

> 被final修饰的类 称为终态类/最终类 此种类 表现出 无法被其他类 继承的效果 Object 是继承树的根 final修饰的类 是继承树的末梢节点

```
//系统核心类库中String就是典型的最终类 类中含有大量功能实现 方法 为了避免子类重写影响方法功能 使用 final修饰
public final class String .....{
    //字符串的大量方法
}
//Math工具类也是一个终态类 含大量运算相关方法
public final class Math {
    //数学计算的大量方法
}
```

2. final修饰成员方法

> 被final 修饰的方法 称为 终态方法 该方法 不能被子类重写覆盖 相较于final 修饰类 具有更好的可控性

```
public class Arrays{
    //该类中的一部分方法使用了final修饰 拒绝子类重写覆盖 另一些方法 没有使用final 可以重写
}
```

3. final 修饰变量

3.1 final修饰成员变量

> 被修饰的成员变量 称为常量 有且仅有一次赋值的机会 不允许重赋值 且必须在声明的第一时间完成赋值

3.2 final修饰局部变量

> 被修饰的成员变量 称为常量 有且仅有一次赋值的机会 不允许重赋值 声明与赋值可以分步完成 赋值可以延后

常量的命名规则规范

> 常量命名 应尽量全部使用大写的英文字母来组成 如果有多个单词 则使用 _ 进行单词的分割

常量应用场景1：使用常量 替代字面量(宏替换) 来提高代码的可阅读性

```
public static final String MOVE_UP="w";
public static final String MOVE_DOWN="s";
public static final String MOVE_LEFT="a";
public static final String MOVE_RIGHT="d";
//case 只能搭配字面量使用  常量可以替换字面量
public static void move(String key){
    switch (key){
        case MOVE_UP:
            System.out.println("角色向上移动");
            break;
        case MOVE_DOWN:
            System.out.println("角色向下移动");
            break;
        case MOVE_LEFT:
            System.out.println("角色向左移动");
            break;
        case MOVE_RIGHT:
            System.out.println("角色向右移动");
            break;
    }
}
```

常量应用场景2: 用于存储参与程序运算 且需要避免误改数值的数据

```
//Math类中 存储圆周率PI 使用的就是常量
public final class Math{
    //...
    public static final double PI = 3.14159265358979323846;
    //...
}
//字符串类中 存储字符数据的数组  也是常量
public final class String{
    //...
    private final char value[];
    //...
}
```

> 引用类型数据使用final 是代表变量空间中的地址不能改动 还是 地址所指向的存值空间中的数据不能改动

```
final Student stu=new Student(); //使用final 修饰引用类型变量
// 对象中的属性可以随意修改 调整 重赋值
stu.name="a";
stu.name="b";
stu.age=30;
stu.age=40;
//stu=new Student();//报错  不能更改变量中存储的地址
```

枚举

> 枚举的本质就是一个继承自 java.lang.Enum类型的子类 创建时需要将kind 选为enum类型 通过反编译可以看到枚举的本质 枚举本质就是一个 类的 多例模式 预设好了多个该类的对象给用户选择使用 但由于构造方法是私有的 用户无法生成新的对象

```
public enum Gender {
    MAN,WOMAN
}
```

经过反编译查看

```
public final class com.Gender extends java.lang.Enum<com.Gender> {
    public static final com.Gender MAN=new com.Gender();//在对象中通过属性存储了"MAN"
    public static final com.Gender WOMAN=new com.Gender(); //在对象中通过属性存储了"WOMAN"
    //返回所有常量组成的数组  方法实现无法观察
    public static com.Gender[] values(){
        return new com.Gender[]{MAN,WOMAN};
    }
    //将字符串转为Gender枚举类型   方法实现无法观察
    public static com.Gender valueOf(java.lang.String){
        //return String-->Gender;
    }
    static {
        //....
    };
}
```

枚举在switch中使用

```
public enum MOVE {
    UP,DOWN,LEFT,RIGHT
}

public class Test{
    public static void main(String[] args) {
        String m="UP";//DOWN LEFT RIGHT
        switch (MOVE.valueOf(m.toUpperCase())){
            case UP:break;
            case DOWN:break;
            case LEFT:break;
            case RIGHT:break;
        }
    }
}
```

抽象方法和抽象类

什么时候使用抽象方法

1. 多个子类 对于同一个方法 的实现各不相同 提取任意子类的方法到父类中都不合适 就可以考虑父类定义抽象方法

```
public class Pig extends Animal{
    @Override
    public void call(){
        System.out.println("哼哼哼");
    }
}

public class Sheep extends Animal {
    @Override
    public void call(){
        System.out.println("咩咩咩");
    }
}
public class Dog extends Animal {
    @Override
    public void call() {
        System.out.println("汪汪汪");
    }
}
```

2. 父类分析后 应该具备某个方法 但父类本身的抽象性决定了这个方法难以实现 此时就应该设计抽象方法

```
public abstract class Animal {
    //抽象方法 要避免用户调用  只要用户无法实例化Animal对象 那就无法调用该方法
    public abstract void call();
}
```

抽象方法的定义语法

```
权限修饰符 abstract 返回值类型 方法名(形参列表);
```

> 1. abstract 与 final和 static都不能同时使用 具有**互斥性**
> 2. 包含抽象方法的类 一定是抽象类 抽象类不一定包含抽象方法
> 3. 抽象类中定义的抽象 方法 在 子类中必须重写实现 除非子类也是抽象类

类与抽象类的对比

类的类别	成员	能否实例化	子类是否重写方法
普通的类	属性 方法 构造器 代码块 内部类	可以实例化生成对象	子类自行选择

类的类别	成员	能否实例化	子类是否重写方法
抽象类	属性 方法 构造器 代码块 内部类 抽象方法	不能实例化生成对象	抽象方法子类必须重写

模版方法模式

模板方法模式实现步骤

1、定义一个抽象类。

2、定义2个方法，一个是模板方法：把相同代码放里面去，不同代码定义成抽象方法

3、子类继承抽象类，重写抽象方法。

多态

> 多态是面向对象的三大特征之一 多态就是指 对象拥有多种不同的形态 同一个父类类型的不同子类对象 对同一个父类方法 进行了不同的重写实现 表现出不同的形态

多态的实现有三个条件

1. 必须有类的继承关系 或 类与接口的实现关系
2. 必须要有方法的重写
3. 要有装箱 父类对象指向子类引用 里氏替换原则

多态实现后 成员访问特点

> 父类类型 变量= new 子类类型();
>
> 变量名.方法名();

方法调用：编译看左边，运行看右边。 父类定义了方法 才可以通过调用语法的编译 子类方法对父类方法进行了重写 运行时执行的是子类方法的主体代码

变量调用：编译看左边，运行也看左边。（注意）