learning java

20190610

1. ~~Preface~~
2. ~~Introduction to Objects~~

20190611

1. ~~Everything Is an object~~
2. Operators

- if you want to compare the actual contents of an object for equivalence? You must use the special method equals( ) that exists for all objects (not primitives, which work fine with == and !=)

20190803

第二章 Java语言基础

- 小数默认为double类型，如果要给float类型赋小数值的话，需要在小数后面加上一个f

  float = 3.14f

- 包装类的一些静态方法可以实现不同数据类型的转化，例如整数和字符串的互相转化:

  int a = Integer.parseInt("123");

  String s = String.valueOf(123);

- 可以在循环语句前加上一个lab，然后用break+lab的形式跳出循环、或者continue+lab的形式短路循环：

```
p:  for(int i=1; i<=10 ;i++)
    {
        for(int j=1; j<=10; j++)
        {
            if(j == 5)
            {
                break p;
            }
            System.out.println(j);
        }
    }
```

- java中return语句只能用在一个方法体的最后，如果在return之后仍有可执行的语句，则会出现编译错误
- 对于对象数组，可以这样初始化：

```
Integer results[] = {new Integer(3), new Integer(5)};
float f4[] = new float[]{1.0f, 2.0f, 3.0f};
String[] dogs = new String[]{new String("Fido"), new String("Spike")};
```

20190804

面向对象（上）

- 不能对构造方法指定类型，它有隐含的返回值，该值由系统内部使用，如果指定了相应的类型，则该方法就不是构造方法
- 当构造方法中的参数名与域变量名相同时，此时在构造方法中需要用this关键字来区分，或者使用下划线避免同名
- 静态代码块：一个类中可以使用不包含在任何方法体中的静态代码块。当类被装载时，静态代码块被执行且只被执行一次：

```
class StaticCodeBlock{
    static int value;
    static{
        value = 3;
        System.out.println("value = " + value);
    }
    public static void main(String[] args){

    }
}
```

- 单件模式是设计模式的一种，它确保一个类有且仅有一个对象，可以这样实现：

```
public class FighterPlane{
    private String name;
    private int missileNum;
    private FighterPlane(String _name, int _missileNum){
        name = _name;
        missileNum = _missileNum;
    }
    public static FighterPlane getInstance(String _name, int _missileNum){
        if(fp == null){
            fp = new FighterPlane(_name, _missileNum);
            return fp;
        }
    }
}
```

20190805

- 继承时父类的属性统统被复制到子类中，包括父类中的private成员，但是子类对象内部无法直接访问，必须通过父类接口方法访问
- 重载的多种方法之间往往存在一定的调用关系，即一个方法写有实现功能，其他方法采用委托方式进行调用。

```
public AudioClip getAudioClip(URL url){
    ... //真正的实现代码
}
public AudioClip getAudioClip(URL url, String name){
    ... //其他代码
    //通过重新构造一个新的URL对象，之后调用上面的同名方法
    return getAudioClip(new URL(url, name))
}
```

- 子类定义的方法与父类名称相同（大小写完全匹配），但参数不同，不是覆盖，二是重载；如果名称，参数相同，返回值不同，则编译不能通过
- 同名的非static和非static方法之间不能覆盖，方法前有final修饰符时，此方法不能在子类中进行覆盖

20190807

面向对象（下）

- this可以指代当前对象，而super没有类似功能，即没有指代父类对象的功能；子类和父类定义了同名变量，则子类对象中将父类定义的同名域变量隐藏，子类如果使用父类的x，则必须采用“super.x”的形式，覆盖的父类方法同理
- 一个类的若干个重载的构造方法之间可以相互调用，且必须使用关键字this来调用重载的其他构造方法，最大限度地提高对已有代码的复用程度,构造方法的调用必须为构造方法中的第一句

```
class AddClass{
    public int x = 0, y = 0, z = 0;
    AddClass(int x){
        this.x = x;
    }
    AddClass(int x, int y){
        this(x);
        this.y = y;
    }
    AddClass(int x, int y, int z){
        this(x, y);
        this.z = z;
    }
}
```

- 同理，子类可以调用父类的构造方法

```
public class SonAddClass extend AddClass{
    int a = 0, b = 0, c = 0;
    SonAddClass(int x){
        super(x);
        a = x + 7;
    }
    SonAddClass(int x, int y){
        super(x, y);
        a = x + 5;
        b = y + 7;
    }
    SonAddClass(int x, int y, int z){
        super(x, y, z);
        a = x + 3;
        b = y + 3;
        c = z + 3;
    }
}
```

20190814

- this和super如果存在，则位于第一句，并且它们两个不能同时存在
- 成员方法中的变量如果不赋初值，则系统会提示变量没有初始化，但类的域变量则不进行提示，因为在于类生成对象时，可以进行默认初始化，类的静态属性如果不赋值，也会被默认初始化
- 接口抽象方法的访问限制符如果为默认（没有修饰符）或public，则类再实现时必须显式地使用public修饰符，否则将被系统警告缩小了接口定义方法的访问控制范围
- 如果实现某接口的类不是抽象类，则它必须实现接口中所有的抽象方法

```
interface Washer{
    public abstruct void startUp();
    public abstruct void letWaterIn();
    public abstruct void washClothes();
}

interface RoseBrand implenments Washer{
    public void startUp(){
        System.out.println("startUp");
    }
    public void letWaterIn(){
        System.out.println("letWaterIn");
    }
    publuc void washClothes(){
        System.out.println("letWaterOut");
    }
}
```

- 接口声明能引用所有实现类对象，抽象类声明能引用所有具体类对象，因此在面向对象设计当中，应追求面向抽象类和接口编程，而不要面向具体类编程
- equals方法是object的方法，因此所有类对象都可以利用它进行引用比较，判断是否指向同一对象，且equals的传入参数必定是对象，对象不能和基本数据类型相比较
- 发生数据成员隐藏的情况下，父类声明和子类声明引用了同一个子类对象，但父类引用访问的是隐藏的成员，而子类引用访问的是新的成员；但如果子类覆盖了父类的同名方法，则父类声明引用子类对象时调用的不是父类的方法体内容，而是子类方法体中的内容，因为java中方法没有隐藏的概念，从这个意义上讲，java方法覆盖时，父类中的方法都相当于c++中被virtual修饰的方法
- 匿名内部类只能用到一个实例，不能定义任何静态成员和方法，且不能被protected、private、public等修饰

```
abstruct class Anonymity{
    abstruct public void fun1();
}
public class Outer{
    public static void main(String[] args){
        new Outer().callInner(new Anonymity(){
            public void fun1(){
                System.out.println("匿名类测试");
            }
        })
    }
    public callInner(Anonymity a){
        a.fun1();
    }
}

//上述代码中new Outer().callInner是使用了匿名对象
```

20190815

异常

- try-catch-finally的执行过程当中相当于“switch-case”，即一个catch执行，其他catch就不执行，有时如不需要对异常进行细致分类处理，则可统一为一个“catch(Exception e){}”

20190816

java常用类库与工具

- String的特点是一旦赋值，便不能改变其指向的对象，如果更改，则会指向一个新的对象。且编译器会自动优化字符串引用的指向

```
String t = "Hello"
String s = "Hello"
//此时s和t会被优化指向同一个对象
String t = "Hello"
String s = new String("Hello")
//这样写s和t才会指向不同的对象
```

- String 用 "==" 进行比较时比较的是引用，而用equals()比较时比较的是值, equalsIgnoreCase()方法忽略大小写比较
- String常用方法：

```
String.concat(String str) //将str附加在字符串后面
String.valueOf() //将其他的基本数据类型转换为String
String.charAt(int index) //得到字符串中下标为index的元素
String.toUpperCase/toLowerCase() //返回字符串大小写转换之后的字串地址
```

- StringBuffer对象可以调用其他方法动态地进行增加、插入、修改和删除操作，且不用像数组那样事先指定大小，从而实现多次插入

```
String s = "a" + "b" + "c";
自动被编译器优化为
String s = new
StringBuffer().append("a”).append("b").append("c").toString();
```

##20190820

- Calendar类是队时间操作的主要类。要得到其对象引用，不能使用new，而要调用其静态方法getInstance(),之后再利用相应对象方法

```
public class GetCurrentTime{
        public static void main(String[] args) {
                Calendar cd = Calendar.getInstance();
                Date d = cd.getTime();
                System.out.println(d.toString());
        }
}
```

- 格式化日期。主要是使用SimpleDateFormat,其对象的format方法是将Date转为指定日期格式的String，而parse方法是将String转为Date

```
import java.text.SimpleDateFormat;
import java.util. *;
public class GetCurrentTime{
```

```
        public static void main(String[] args) {
                SimpleDateFormat sdf1 = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
                System.out.println(sdf1.format(new Date()));
                /*2019-08-20 19:30:39*/
        }
}
```

-