1. 参数传递

1.1 类名作为形参和返回值（应用）

- 1、类名作为方法的形参

  方法的形参是类名，其实需要的是该类的对象

  实际传递的是该对象的【地址值】

- 2、类名作为方法的返回值

  方法的返回值是类名，其实返回的是该类的对象

  实际传递的，也是该对象的【地址值】

- 示例代码：

```
class Cat {
    public void eat() {
        System.out.println("猫吃鱼");
    }
}
class CatOperator {
    public void useCat(Cat c) { //Cat c = new Cat();
        c.eat();
    }
    public Cat getCat() {
        Cat c = new Cat();
        return c;
    }
}
public class CatDemo {
    public static void main(String[] args) {
        //创建操作类对象，并调用方法
        CatOperator co = new CatOperator();
        Cat c = new Cat();
        co.useCat(c);

        Cat c2 = co.getCat(); //new Cat()
        c2.eat();
    }
}
```

1.2 抽象类作为形参和返回值（理解）

- 抽象类作为形参和返回值
  - 方法的形参是抽象类名，其实需要的是该抽象类的子类对象
  - 方法的返回值是抽象类名，其实返回的是该抽象类的子类对象
- 示例代码：

```
abstract class Animal {
    public abstract void eat();
}
class Cat extends Animal {
    @Override
    public void eat() {
        System.out.println("猫吃鱼");
    }
}
class AnimalOperator {
    public void useAnimal(Animal a) { //Animal a = new Cat();
        a.eat();
    }
    public Animal getAnimal() {
        Animal a = new Cat();
        return a;
    }
}
public class AnimalDemo {
    public static void main(String[] args) {
        //创建操作类对象，并调用方法
        AnimalOperator ao = new AnimalOperator();
        Animal a = new Cat();
        ao.useAnimal(a);

        Animal a2 = ao.getAnimal(); //new Cat()
        a2.eat();
    }
}
```

1.3 接口名作为形参和返回值（理解）

- 接口作为形参和返回值
  - 方法的形参是接口名，其实需要的是该接口的实现类对象
  - 方法的返回值是接口名，其实返回的是该接口的实现类对象
- 示例代码：

```
interface Jumpping {
    void jump();
}
class JumppingOperator {
    public void useJumpping(Jumpping j) { //Jumpping j = new Cat();
        j.jump();
    }
    public Jumpping getJumpping() {
        Jumpping j = new Cat();
        return j;
    }
}
class Cat implements Jumpping {
    @Override
    public void jump() {
        System.out.println("猫可以跳高了");
    }
}
public class JumppingDemo {
    public static void main(String[] args) {
        //创建操作类对象，并调用方法
        JumppingOperator jo = new JumppingOperator();
        Jumpping j = new Cat();
        jo.useJumpping(j);

        Jumpping j2 = jo.getJumpping(); //new Cat()
        j2.jump();
    }
}
```

2. 内部类

2.1 内部类的基本使用（理解）

- 内部类概念
  - 在一个类中定义一个类。举例：在一个类A的内部定义一个类B，类B就被称为内部类
- 内部类定义格式
  - 格式&举例：

```
/*
    格式：
    class 外部类名{
        修饰符 class 内部类名{

        }
    }
*/

class Outer {
    public class Inner {

    }
}
```

- 内部类的访问特点
  - 内部类可以直接访问外部类的成员，包括私有
  - 外部类要访问内部类的成员，必须创建对象
- 示例代码：

```
/*
    内部类访问特点：
        内部类可以直接访问外部类的成员，包括私有
        外部类要访问内部类的成员，必须创建对象
 */
public class Outer {
    private int num = 10;
    public class Inner {
        public void show() {
            System.out.println(num);
        }
    }
    public void method() {
        Inner i = new Inner();
        i.show();
    }
}
```

2.2 成员内部类（理解）

- 成员内部类的定义位置
  - 在类中方法，跟成员变量是一个位置
- 外界创建成员内部类格式
  - 格式：外部类名.内部类名 对象名 = 外部类对象.内部类对象;
  - 举例：Outer.Inner oi = new Outer().new Inner();
- 成员内部类的推荐使用方案
  - 将一个类，设计为内部类的目的，大多数都是不想让外界去访问，所以内部类的定义应该私有化，私有化之后，再提供一个可以让外界调用的方法，方法内部创建内部类对象并调用。

- 示例代码：

```
class Outer {
    private int num = 10;
    private class Inner {
        public void show() {
            System.out.println(num);
        }
    }
    public void method() {
        Inner i = new Inner();
        i.show();
    }
}
public class InnerDemo {
    public static void main(String[] args) {
        //Outer.Inner oi = new Outer().new Inner();
        //oi.show();
        Outer o = new Outer();
        o.method();
    }
}
```

2.3 局部内部类（理解）

- 局部内部类定义位置
  - 局部内部类是在方法中定义的类
- 局部内部类方式方式
  - 局部内部类，外界是无法直接使用，需要在方法内部创建对象并使用
  - 该类可以直接访问外部类的成员，也可以访问方法内的局部变量
- 示例代码

```
class Outer {
    private int num = 10;
    public void method() {
        int num2 = 20;
        class Inner {
            public void show() {
                System.out.println(num);
                System.out.println(num2);
            }
        }
        Inner i = new Inner();
        i.show();
    }
}
public class OuterDemo {
    public static void main(String[] args) {
        Outer o = new Outer();
        o.method();
    }
}
```

2.4 匿名内部类（应用）

- 匿名内部类的前提
  - 存在一个类或者接口，这里的类可以是具体类也可以是抽象类
- 匿名内部类的格式
  - 格式：new 类名 ( ) { 重写方法 } new 接口名 ( ) { 重写方法 }
  - 举例：

```
new Inter(){
    @Override
    public void method(){}
}
```

- 匿名内部类的本质
  - 本质：是一个继承了该类或者实现了该接口的子类匿名对象
- 匿名内部类的细节
  - 匿名内部类可以通过多态的形式接受

```
Inter i = new Inter(){
  @Override
    public void method(){

    }
}
```

- 匿名内部类直接调用方法

```
interface Inter{
    void method();
}

class Test{
    public static void main(String[] args){
        new Inter(){
            @Override
            public void method(){
                System.out.println("我是匿名内部类");
            }
        }.method(); // 直接调用方法
    }
}
```

2.4 匿名内部类在开发中的使用（应用）

- 匿名内部类在开发中的使用
  - 当发现某个方法需要，接口或抽象类的子类对象，我们就可以传递一个匿名内部类过去，来简化传统的代码
- 示例代码：

```
interface Jumpping {
    void jump();
}
class Cat implements Jumpping {
    @Override
    public void jump() {
        System.out.println("猫可以跳高了");
    }
}
class Dog implements Jumpping {
    @Override
    public void jump() {
        System.out.println("狗可以跳高了");
    }
}
class JumppingOperator {
    public void method(Jumpping j) { //new Cat();   new Dog();
        j.jump();
    }
}
class JumppingDemo {
    public static void main(String[] args) {
        //需求：创建接口操作类的对象，调用method方法
        JumppingOperator jo = new JumppingOperator();
        Jumpping j = new Cat();
        jo.method(j);

        Jumpping j2 = new Dog();
        jo.method(j2);
        System.out.println("--------");

        // 匿名内部类的简化
        jo.method(new Jumpping() {
            @Override
            public void jump() {
                System.out.println("猫可以跳高了");
            }
        });
        // 匿名内部类的简化
        jo.method(new Jumpping() {
            @Override
            public void jump() {
                System.out.println("狗可以跳高了");
            }
        });
    }
}
```

3. 常用API

3.1 Math（应用）

- 1、Math类概述
  - Math 包含执行基本数字运算的方法
- 2、Math中方法的调用方式
  - Math类中无构造方法，但内部的方法都是静态的，则可以通过 **类名.进行调用**
- 3、Math类的常用方法

方法名　方法名	说明
public static int　abs(int a)	返回参数的绝对值
public static double ceil(double a)	返回大于或等于参数的最小double值，等于一个整数
public static double floor(double a)	返回小于或等于参数的最大double值，等于一个整数
public　static int round(float a)	按照四舍五入返回最接近参数的int
public static int　max(int a,int b)	返回两个int值中的较大值
public　static int min(int a,int b)	返回两个int值中的较小值
public　static double pow (double a,double b)	返回a的b次幂的值
public　static double random()	返回值为double的正值，[0.0,1.0)

3.2 System（应用）

- System类的常用方法

方法名	说明
public　static void exit(int status)	终止当前运行的　Java　虚拟机，非零表示异常终止
public　static long currentTimeMillis()	返回当前时间(以毫秒为单位)

- 示例代码
  - 需求：在控制台输出1-10000，计算这段代码执行了多少毫秒

```java
public class SystemDemo {
    public static void main(String[] args) {
        // 获取开始的时间节点
        long start = System.currentTimeMillis();
        for (int i = 1; i <= 10000; i++) {
            System.out.println(i);
        }
        // 获取代码运行结束后的时间节点
        long end = System.currentTimeMillis();
        System.out.println("共耗时：" + (end - start) + "毫秒");
    }
}
```

3.3 Object类的toString方法（应用）

- Object类概述
  - Object 是类层次结构的根，每个类都可以将 Object 作为超类。所有类都直接或者间接的继承自该类，换句话说，该类所具备的方法，所有类都会有一份
- 查看方法源码的方式
  - 选中方法，按下Ctrl + B
- 重写toString方法的方式
  - 1. Alt + Insert 选择toString
  - 2. 在类的空白区域，右键 -> Generate -> 选择toString
- toString方法的作用：
  - 以良好的格式，更方便的展示对象中的属性值
- 示例代码：

```
class Student extends Object {
    private String name;
    private int age;

    public Student() {
    }

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
public class ObjectDemo {
    public static void main(String[] args) {
        Student s = new Student();
        s.setName("林青霞");
        s.setAge(30);
        System.out.println(s);
        System.out.println(s.toString());
    }
}
```

- 运行结果：

```
Student{name='林青霞', age=30}
Student{name='林青霞', age=30}
```

3.4 Object类的equals方法（应用）

- equals方法的作用
  - 用于对象之间的比较，返回true和false的结果
  - 举例：s1.equals(s2); s1和s2是两个对象
- 重写equals方法的场景
  - 不希望比较对象的地址值，想要结合对象属性进行比较的时候。
- 重写equals方法的方式
  - 1. alt + insert 选择equals() and hashCode()，IntelliJ Default，一路next，finish即可
  - 2. 在类的空白区域，右键 -> Generate -> 选择equals() and hashCode()，后面的同上。
- 示例代码：

```java
class Student {
    private String name;
    private int age;

    public Student() {
    }

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public boolean equals(Object o) {
        //this -- s1
        //o -- s2
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Student student = (Student) o; //student -- s2

        if (age != student.age) return false;
        return name != null ? name.equals(student.name) : student.name == null;
    }
}
public class ObjectDemo {
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.setName("林青霞");
        s1.setAge(30);

        Student s2 = new Student();
        s2.setName("林青霞");
        s2.setAge(30);

        //需求：比较两个对象的内容是否相同

        System.out.println(s1.equals(s2));
```

```
        }
    }
```

3.5 冒泡排序原理（理解）

- 冒泡排序概述
    - 一种排序的方式，对要进行排序的数据中相邻的数据进行两两比较，将较大的数据放在后面，依次对所有的数据进行操作，直至所有数据按要求完成排序
- 如果有n个数据进行排序，总共需要比较n-1次
- 每一次比较完毕，下一次的比较就会少一个数据参与

3.6 冒泡排序代码实现（理解）

- 代码实现

```java
/*
    冒泡排序：
        一种排序的方式，对要进行排序的数据中相邻的数据进行两两比较，将较大的数据放在后面，
        依次对所有的数据进行操作，直至所有数据按要求完成排序
 */
public class ArrayDemo {
    public static void main(String[] args) {
        //定义一个数组
        int[] arr = {24, 69, 80, 57, 13};
        System.out.println("排序前：" + arrayToString(arr));

        // 这里减1，是控制每轮比较的次数
        for (int x = 0; x < arr.length - 1; x++) {
            // -1是为了避免索引越界，-x是为了调高比较效率
            for (int i = 0; i < arr.length - 1 - x; i++) {
                if (arr[i] > arr[i + 1]) {
                    int temp = arr[i];
                    arr[i] = arr[i + 1];
                    arr[i + 1] = temp;
                }
            }
        }
        System.out.println("排序后：" + arrayToString(arr));

    }

    //把数组中的元素按照指定的规则组成一个字符串：[元素1, 元素2, ...]
    public static String arrayToString(int[] arr) {
        StringBuilder sb = new StringBuilder();
        sb.append("[");
        for (int i = 0; i < arr.length; i++) {
            if (i == arr.length - 1) {
                sb.append(arr[i]);
            } else {
                sb.append(arr[i]).append(", ");
            }
        }
        sb.append("]");
        String s = sb.toString();
        return s;
    }
}
```

3.7 Arrays（应用）

- Arrays的常用方法

方法名	说明
public static String toString(int[] a)	返回指定数组的内容的字符串表示形式
public static void sort(int[] a)	按照数字顺序排列指定的数组

- 工具类设计思想

  1、构造方法用 private 修饰

  2、成员用 public static 修饰