Java 基础知识

一、数据类型

1. 基本类型

Byte/8, char/16, short/16, int/32, float/32, long/64, double/64, Boolean/~

boolean 只有两个值：true、false，可以使用 1 bit 来存储，但是具体大小没有明确规定。JVM 会在编译时期将 boolean 类型的数据转换为 int，使用 1 来表示 true，0 表示 false。JVM 支持 boolean 数组，但是是通过读写 byte 数组来实现的。

2. 包装类型

每个基本类型都有其对应的包装类型，基本类型和包装类型之间的赋值通过自动装箱和拆箱来完成。

```
Integer x = 2;     // 装箱 调用了 Integer.valueOf(2)
int y = x;         // 拆箱 调用了 X.intValue()
```

3. 缓存池

new Integer(123) 每次都会新建一个缓存对象

Integer.valueOf(123) 会使用缓存池中的对象，多次调用会获得同一个对象的引用。

```
Integer x = new Integer(123);
Integer y = new Integer(123);
System.out.println(x == y);    // false
Integer z = Integer.valueOf(123);
Integer k = Integer.valueOf(123);
System.out.println(z == k);   // true
```

valueOf() 函数会首先判断值是否在缓存池中，如果在的话就直接调用缓存池的内容。

在 java8 中，Integer 的缓存池大小默认为-128 ~ 127

编译器会在自动装箱的过程中调用 valueOf 的方法

```
Integer m = 123;
Integer n = 123;
System.out.println(m == n); // true
```

二、String

1. 概览

String 被声明为 final，不可继承。Java8 中，string 内部通过 char 数组来储存数据。

Java9 之后，string 改用 byte 数组来储存字符串。

被声明为 final 后，内部没有改变 value 的函数，保证了 string 不可变。

2. 不可变的好处

a. 可以缓存 hash 值。

因为 string 的 hash 值经常被使用，例如用 string 做 HashMap 的 key，因为不可变的特性使得 hash 值也不会改变，因此只需要做一次计算。

b. String Pool

如果一个 string 已经被创建过了，那么就会从 string pool 中获得引用，只有当 string 是不可变的时候才可以使用 string pool。

c. 安全性

String 经常被用作参数，保证了参数不可变，

d. 线程安全

String 的不可变性天然具备线程安全

3. String, StringBuffer and StringBuilder

1. 可变性

String 不可变

StringBuffer 和 StringBuilder 可变

2. 线程安全

   String 不可变，因此是线程安全的

   StringBuilder 不是线程安全的

   StringBuffer 是线程安全的，内部使用 synchronized 进行同步

4. String Pool

字符串常量池（String Pool）保存着所有字符串字面量（literal strings），这些字面量在编译时期就确定。不仅如此，还可以使用 String 的 intern() 方法在运行过程将字符串添加到 String Pool 中。

当一个字符串调用 intern() 方法时，如果 String Pool 中已经存在一个字符串和该字符串值相等（使用 equals() 方法进行确定），那么就会返回 String Pool 中字符串的引用；否则，就会在 String Pool 中添加一个新的字符串，并返回这个新字符串的引用。

下面示例中，s1 和 s2 采用 new String() 的方式新建了两个不同字符串，而 s3 和 s4 是通过 s1.intern() 方法取得同一个字符串引用。intern() 首先把 s1 引用的字符串放到 String Pool 中，然后返回这个字符串引用。因此 s3 和 s4 引用的是同一个字符串。

5.

三、运算

1. 参数传递

Java 的参数是值传递而不是引用传递

2. 隐式类型转换

Java 不能直接向下进行隐式类型转换，但使用+=或者++的时候可以

3. Switch

从 java7 开始，可以在 switch 判断句中使用 string 对象

```
String s = "a";
switch (s) {
    case "a":
        System.out.println("aaa");
        break;
    case "b":
        System.out.println("bbb");
        break;
}
```

4.

四、关键字

1. Final

a. 声明数据为常量，可以是编译时常量，也可以是在运行时被初始化后不能被改变的常量.

   对于基本类型，final 使数值不变。引用类型，引用不改变。

b. 对于方法而言，final 方法不能被子类重写

c. Final 类不能被继承

2. Static

a. 静态变量

   又称为类变量，也就是说这个变量属于类的，类所有的实例都共享静态变量，可以直接通过类名来访问它。静态变量在内存中只存在一份。

b. 静态方法

   静态方法在类加载的时候就存在了，它不依赖于任何实例。所以静态方法必须有实现，也就是说它不能是抽象方法。只能访问所属类的静态字段和静态方法，方法中不能有 this 和 super 关键字，因此这两个关键字与具体对象关联。

c. 静态语句块

   静态语句块在类初始化时运行一次。

d. 静态内部类

非静态内部类依赖于外部类的实例，也就是说需要先创建外部类实例，才能用这个实例去创建非静态内部类。而静态内部类不需要。

```
public class OuterClass {

    class InnerClass {
    }

    static class StaticInnerClass {
    }

    public static void main(String[] args) {
        // InnerClass innerClass = new InnerClass(); // 'OuterClass.this' cannot be referenced fr
        OuterClass outerClass = new OuterClass();
        InnerClass innerClass = outerClass.new InnerClass();
        StaticInnerClass staticInnerClass = new StaticInnerClass();
    }
}
```

e. 静态导包

在使用静态变量和方法时不用再指明 ClassName，从而简化代码，但可读性大大降低

f. 初始化顺序

静态变量和静态语句块优先于实例变量和普通语句块，静态变量和静态语句块的初始化顺序取决于它们在代码中的顺序。

存在继承的情况下，初始化顺序为：

- 父类（静态变量、静态语句块）
- 子类（静态变量、静态语句块）
- 父类（实例变量、普通语句块）
- 父类（构造函数）
- 子类（实例变量、普通语句块）
- 子类（构造函数）

五、Object 通用方法

1. Equals ()

a. 等价关系

I 自反性

```
x.equals(x); // true
```

II 对称性

```
x.equals(y) == y.equals(x); // true
```

III 传递性

```
if (x.equals(y) && y.equals(z))
    x.equals(z); // true;
```

IV 一致性

多次调用 equals() 方法结果不变

```
x.equals(y) == x.equals(y); // true
```

V 与 null 的比较

对任何不是 null 的对象 x 调用 x.equals(null) 结果都为 false

```
x.equals(null); // false;
```

b. 等价与相等

- 对于基本类型，== 判断两个值是否相等，基本类型没有 equals() 方法。
- 对于引用类型，== 判断两个变量是否引用同一个对象，而 equals() 判断引用的对象是否等价。

```
Integer x = new Integer(1);
Integer y = new Integer(1);
System.out.println(x.equals(y)); // true
System.out.println(x == y);      // false
```

c. 实现

- 检查是否为同一个对象的引用，如果是直接返回 true；
- 检查是否是同一个类型，如果不是，直接返回 false；
- 将 Object 对象进行转型；
- 判断每个关键域是否相等。

2. HashCode ()

hashCode() 返回哈希值，而 equals() 是用来判断两个对象是否等价。等价的两个对象散列值一定相同，但是散列值相同的两个对象不一定等价，这是因为计算哈希值具有随机性，两个值不同的对象可能计算出相同的哈希值。

在覆盖 equals() 方法时应当总是覆盖 hashCode() 方法，保证等价的两个对象哈希值也相等。

HashSet 和 HashMap 等集合类使用了 hashCode() 方法来计算对象应该存储的位置，因此要将对象添加到这些集合类中，需要让对应的类实现 hashCode() 方法。

下面的代码中，新建了两个等价的对象，并将它们添加到 HashSet 中。我们希望将这两个对象当成一样的，只在集合中添加一个对象。但是 EqualExample 没有实现 hashCode() 方法，因此这两个对象的哈希值是不同的，最终导致集合添加了两个等价的对象。

```
EqualExample e1 = new EqualExample(1, 1, 1);
EqualExample e2 = new EqualExample(1, 1, 1);
System.out.println(e1.equals(e2)); // true
HashSet<EqualExample> set = new HashSet<>();
set.add(e1);
set.add(e2);
System.out.println(set.size());   // 2
```

理想的哈希函数应当具有均匀性，即不相等的对象应当均匀分布到所有可能的哈希值上。这就要求了哈希函数要把所有域的值都考虑进来。可以将每个域都当成 R 进制的某一位，然后组成一个 R 进制的整数。

R 一般取 31，因为它是一个奇素数，如果是偶数的话，当出现乘法溢出，信息就会丢失，因为与 2 相乘相当于向左移一位，最左边的位丢失。并且一个数与 31 相乘可以转换成移位和减法：`31*x == (x<<5)-x`，编译器会自动进行这个优化。

```
@Override
public int hashCode() {
    int result = 17;
    result = 31 * result + x;
    result = 31 * result + y;
    result = 31 * result + z;
    return result;
}
```

3. **toString ()**
4. **clone ()**
   a. cloneable

   clone() 是 Object 的 protected 方法，它不是 public，一个类不显式去重写 clone()，其它类就不能直接去调用该类实例的 clone() 方法。

```
public class CloneExample implements Cloneable {
    private int a;
    private int b;

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

   b. 浅拷贝

   拷贝对象和原始对象的引用类型引用同一个对象。

```
public class ShallowCloneExample implements Cloneable {

    private int[] arr;

    public ShallowCloneExample() {
        arr = new int[10];
        for (int i = 0; i < arr.length; i++) {
            arr[i] = i;
        }
    }

    public void set(int index, int value) {
        arr[index] = value;
    }

    public int get(int index) {
        return arr[index];
    }

    @Override
    protected ShallowCloneExample clone() throws CloneNotSupportedException {
        return (ShallowCloneExample) super.clone();
    }
}
```

```
ShallowCloneExample e1 = new ShallowCloneExample();
ShallowCloneExample e2 = null;
try {
    e2 = e1.clone();
} catch (CloneNotSupportedException e) {
    e.printStackTrace();
}
e1.set(2, 222);
System.out.println(e2.get(2)); // 222
```

   c. 深拷贝

   拷贝对象和原始对象的引用类型引用不同对象

```
    @Override
    protected DeepCloneExample clone() throws CloneNotSupportedException {
        DeepCloneExample result = (DeepCloneExample) super.clone();
        result.arr = new int[arr.length];
        for (int i = 0; i < arr.length; i++) {
            result.arr[i] = arr[i];
        }
        return result;
    }
```

d. Clone 的替代方案

使用 clone() 方法来拷贝一个对象即复杂又有风险，它会抛出异常，并且还需要类型转换。Effective Java 书上讲到，最好不要去使用 clone()，可以使用拷贝构造函数或者拷贝工厂来拷贝一个对象。

```
public class CloneConstructorExample {

    private int[] arr;

    public CloneConstructorExample() {
        arr = new int[10];
        for (int i = 0; i < arr.length; i++) {
            arr[i] = i;
        }
    }

    public CloneConstructorExample(CloneConstructorExample original) {
        arr = new int[original.arr.length];
        for (int i = 0; i < original.arr.length; i++) {
            arr[i] = original.arr[i];
        }
    }

    public void set(int index, int value) {
        arr[index] = value;
    }

    public int get(int index) {
        return arr[index];
    }
}
```

六、反射

1. 反射和类加载

a. 每个类都有一个 Class 对象，包含了与类有关的信息。当编译一个新类时，会产生一个同名的 .class 文件，该文件内容保存着 Class 对象。

b. 类加载相当于 Class 对象的加载，类在第一次使用时才动态加载到 JVM 中。也可以使用 Class.forName("com.mysql.jdbc.Driver") 这种方式来控制类的加载，该方法会返回一个 Class 对象。

c. 反射可以提供运行时的类信息，并且这个类可以在运行时才加载进来，甚至在编译时期该类的 .class 不存在也可以加载进来。

Java 反射主要提供以下功能：

- 在运行时判断任意一个对象所属的类；
- 在运行时构造任意一个类的对象；
- 在运行时判断任意一个类所具有的成员变量和方法（通过反射甚至可以调用private方法）；
- 在运行时调用任意一个对象的方法

d. Class 和 java.lang.reflect 一起对反射提供了支持，java.lang.reflect 类库主要包含了以下三个类：

- **Field**：可以使用 get() 和 set() 方法读取和修改 Field 对象关联的字段；
- **Method**：可以使用 invoke() 方法调用与 Method 对象关联的方法；
- **Constructor**：可以用 Constructor 的 newInstance() 创建新的对象。

e. 反射的主要用途

反射最重要的用途就是开发各种通用框架。很多框架（比如 Spring）都是配置化的（比如通过 XML 文件配置 Bean），为了保证框架的通用性，它们可能需要根据配置文件加载不同的对象或类，调用不同的方法，这个时候就必须用到反射，运行时动态加载需要加载的对象。

举一个例子，在运用 Struts 2 框架的开发中我们一般会在 `struts.xml` 里去配置 `Action`，比如：

举一个例子，在运用 Struts 2 框架的开发中我们一般会在 `struts.xml` 里去配置 `Action`，比如：

```
<action name="login"
                class="org.ScZyhSoft.test.action.SimpleLoginAction"
                method="execute">
            <result>/shop/shop-index.jsp</result>
            <result name="error">login.jsp</result>
        </action>
```

配置文件与 `Action` 建立了一种映射关系，当 View 层发出请求时，请求会被 `StrutsPrepareAndExecuteFilter` 拦截，然后 `StrutsPrepareAndExecuteFilter` 会去动态地创建 Action 实例。比如我们请求 `login.action`，那么 `StrutsPrepareAndExecuteFilter` 就会去解析struts.xml文件，检索action中name为login的Action，并根据class属性创建SimpleLoginAction实例，并用invoke方法来调用execute方法，这个过程离不开反射。

f. 反射的基本运用

https://www.sczyh30.com/posts/Java/java-reflection-1/

(1) 获取 class 对象，如 getClass()
(2) 判断是否为某个类的对象，isInstance()
(3) 创建示例，newInstance()
(4) 获取方法，getMethods(), getDeclaredMethods(), getMethod()
(5) 获取构造器的信息
(6) 获取成员变量的信息
(7) 调用方法，invoke()
(8) 利用反射创建数组

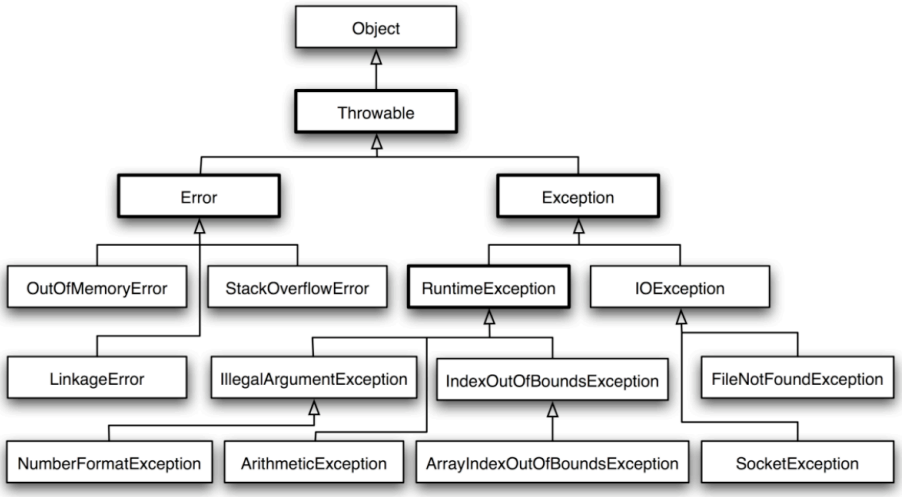
2. 反射的注意事项

反射需要消耗额外的系统资源，如果不是需要动态地创建对象，就不需要使用反射。
反射会忽略掉权限的检查，可能会因为破坏封装特性而产生安全性问题。

七、异常

Throwable 可以用来表示任何可以作为异常抛出的类，分为两种： **Error** 和 **Exception**。其中 Error 用来表示 JVM 无法处理的错误，Exception 分为两种：

- **受检异常**： 需要用 try...catch... 语句捕获并进行处理，并且可以从异常中恢复；
- **非受检异常**： 是程序运行时错误，例如除 0 会引发 Arithmetic Exception，此时程序崩溃并且无法恢复。



八、泛型