一、关键字

final

1. 数据

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；
- 对于引用类型，final 使引用不变，也就不能引用其它对象，但是被引用的对象本身是可以修改的。

```
final int x = 1;
// x = 2;  // cannot assign value to final variable 'x'
final A y = new A();
y.a = 1;
```

2. 方法

声明方法不能被子类覆盖。

private 方法隐式地被指定为 final，如果在子类中定义的方法和基类中的一个 private 方法签名相同，此时子类的方法不是覆盖基类方法，而是在子类中定义了一个新的方法。

3. 类

声明类不允许被继承。

static

1. 静态变量

静态变量在内存中只存在一份，只在类初始化时赋值一次。

- 静态变量：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；
- 实例变量：每创建一个实例就会产生一个实例变量，它与该实例同生共死。

```
public class A {
    private int x;         // 实例变量
    public static int y;  // 静态变量
}
```

2. 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（abstract）。

3. 静态语句块

静态语句块在类初始化时运行一次。

4. 静态内部类

内部类的一种，静态内部类不依赖外部类，且不能访问外部类的非静态的变量和方法。

5. 静态导包

```
import static com.xxx.ClassName.*
```

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。

6. 变量赋值顺序

静态变量的赋值和静态语句块的运行优先于实例变量的赋值和普通语句块的运行，静态变量的赋值和静态语句块的运行哪个先执行取决于它们在代码中的顺序。

```
public static String staticField = "静态变量";
```

```
static {
    System.out.println("静态语句块");
}
```

```
public String field = "实例变量";
```

```
{
    System.out.println("普通语句块");
}
```

最后才运行构造函数

```
public InitialOrderTest() {
    System.out.println("构造函数");
}
```

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）

二、Object 通用方法

概览

```
public final native Class<?> getClass()

public native int hashCode()

public boolean equals(Object obj)

protected native Object clone() throws CloneNotSupportedException

public String toString()

public final native void notify()


public final native void notifyAll()
```

```
public final native void wait(long timeout) throws InterruptedException

public final void wait(long timeout, int nanos) throws InterruptedException

public final void wait() throws InterruptedException

protected void finalize() throws Throwable {}
```

equals()

1. equals() 与 == 的区别

- 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，== 判断两个实例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
```

2. 等价关系

（一）自反性

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

（二）对称性

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

（三）传递性

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

（四）一致性

多次调用 equals() 方法结果不变

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

（五）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.euqals(null); // false;
```

3. 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 实例进行转型；
- 判断每个关键域是否相等。

```
public class EqualExample {
    private int x;
    private int y;
    private int z;

    public EqualExample(int x, int y, int z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

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

        EqualExample that = (EqualExample) o;

        if (x != that.x) return false;
        if (y != that.y) return false;
        return z == that.z;
    }
}
```

hashCode()

hasCode() 返回散列值，而 equals() 是用来判断两个实例是否等价。等价的两个实例散列值一定要相同，但是散列值相同的两个实例不一定等价。

在覆盖 equals() 方法时应当总是覆盖 hashCode() 方法，保证等价的两个实例散列值也相等。

下面的代码中，新建了两个等价的实例，并将它们添加到 HashSet 中。我们希望将这两个实例当成一样的，只在集合中添加一个实例，但是因为 EqualExample 没有实现 hasCode() 方法，因此这两个实例的散列值是不同的，最终导致集合添加了两个等价的实例。

```
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
```

理想的散列函数应当具有均匀性，即不相等的实例应当均匀分布到所有可能的散列值上。这就要求了散列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。

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;
}
```

toString()

默认返回 ToStringExample@4554617c 这种形式，其中 @ 后面的数值为散列码的无符号十六进制表示。

```
public class ToStringExample {
    private int number;

    public ToStringExample(int number) {
        this.number = number;
    }
}
```

```
ToStringExample example = new ToStringExample(123);
System.out.println(example.toString());
```

```
ToStringExample@4554617c
```

clone()

1. cloneable

clone() 是 Object 的受保护方法，这意味着，如果一个类不显式去覆盖 clone() 就没有这个方法。

```
public class CloneExample {
    private int a;
    private int b;
}
```

```
CloneExample e1 = new CloneExample();
// CloneExample e2 = e1.clone(); // 'clone()' has protected access in 'java.lang.Object'
```

接下来覆盖 Object 的 clone() 得到以下实现：

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

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

```
CloneExample e1 = new CloneExample();
try {
    CloneExample e2 = e1.clone();
} catch (CloneNotSupportedException e) {
    e.printStackTrace();
}
```

```
java.lang.CloneNotSupportedException: CloneTest
```

以上抛出了 CloneNotSupportedException，这是因为 CloneTest 没有实现 Cloneable 接口。

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

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

应该注意的是，clone() 方法并不是 Cloneable 接口的方法，而是 Object 的一个 protected 方法。Cloneable 接口只是规定，如果一个类没有实现 Cloneable 接口又调用了 clone() 方法，就会抛出 CloneNotSupportedException。

2. 深拷贝与浅拷贝

- 浅拷贝：拷贝实例和原始实例的引用类型引用同一个对象；
- 深拷贝：拷贝实例和原始实例的引用类型引用不同对象。

```
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
```

```
public class DeepCloneExample implements Cloneable {
    private int[] arr;

    public DeepCloneExample() {
        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 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;
    }
}
```

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

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];
    }
}
```

```
CloneConstructorExample e1 = new CloneConstructorExample();
CloneConstructorExample e2 = new CloneConstructorExample(e1);
e1.set(2, 222);
System.out.println(e2.get(2)); // 2
```

四、继承

访问权限

Java 中有三个访问权限修饰符：private、protected 以及 public，如果不加访问修饰符，表示包级可见。

可以对类或类中的成员（字段以及方法）加上访问修饰符。

- 成员可见表示其它类可以用这个类的实例访问到该成员；
- 类可见表示其它类可以用这个类创建对象。

protected 用于修饰成员，表示在继承体系中成员对于子类可见，但是这个访问修饰符对于类没有意义。

设计良好的模块会隐藏所有的实现细节，把它的 API 与它的实现清晰地隔离开来。模块之间只通过它们的 API 进行通信，一个模块不需要知道其他模块的内部工作情况，这个概念被称为信息隐藏或封装。因此访问权限应当尽可能地使每个类或者成员不被外界访问。

如果子类的方法覆盖了父类的方法，那么子类中该方法的访问级别不允许低于父类的访问级别。这是为了确保可以使用父类实例的地方都可以使用子类实例，也就是确保满足里氏替换原则。

字段决不能是公有的，因为这么做的话就失去了对这个字段修改行为的控制，客户端可以对其随意修改。可以使用公有的 getter 和 setter 方法来替换公有字段。

```
public class AccessExample {
    public int x;
}
```

```
public class AccessExample {
    private int x;

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }
}
```

但是也有例外，如果是包级私有的类或者私有的嵌套类，那么直接暴露成员不会有特别大的影响。

```
public class AccessWithInnerClassExample {
    private class InnerClass {
        int x;
    }

    private InnerClass innerClass;

    public AccessWithInnerClassExample() {
        innerClass = new InnerClass();
    }

    public int getValue() {
        return innerClass.x; // 直接访问
    }
}
```

抽象类与接口

1. 抽象类

抽象类和抽象方法都使用 abstract 进行声明。抽象类一般会包含抽象方法，抽象方法一定位于抽象类中。

抽象类和普通类最大的区别是，抽象类不能被实例化，需要继承抽象类才能实例化其子类。

```
public abstract class AbstractClassExample {

    protected int x;
    private int y;

    public abstract void func1();

    public void func2() {
        System.out.println("func2");
    }
}
```

```
public class AbstractExtendClassExample extends AbstractClassExample{
    @Override
    public void func1() {
        System.out.println("func1");
    }
}
```

```
// AbstractClassExample ac1 = new AbstractClassExample(); // 'AbstractClassExample' is abstract;
cannot be instantiated
AbstractClassExample ac2 = new AbstractExtendClassExample();
ac2.func1();
```

2. 接口

接口是抽象类的延伸，在 Java 8 之前，它可以看成是一个完全抽象的类，也就是说它不能有任何的方法实现。

从 Java 8 开始，接口也可以拥有默认的方法实现，这是因为不支持默认方法的接口的维护成本太高了。在 Java 8 之前，如果一个接口想要添加新的方法，那么要修改所有实现了该接口的类。

接口的成员（字段 + 方法）默认都是 public 的，并且不允许定义为 private 或者 protected。

接口的字段默认都是 static 和 final 的。

```
public interface InterfaceExample {
    void func1();

    default void func2(){
        System.out.println("func2");
    }

    int x = 123;
    // int y;               // Variable 'y' might not have been initialized
    public int z = 0;       // Modifier 'public' is redundant for interface fields
    // private int k = 0;   // Modifier 'private' not allowed here
    // protected int l = 0; // Modifier 'protected' not allowed here
```

```
    // private void fun3(); // Modifier 'private' not allowed here
}
```

```
public class InterfaceImplementExample implements InterfaceExample {
    @Override
    public void func1() {
        System.out.println("func1");
    }
}
```

```
// InterfaceExample ie1 = new InterfaceExample(); // 'InterfaceExample' is abstract; cannot be
instantiated
InterfaceExample ie2 = new InterfaceImplementExample();
ie2.func1();
System.out.println(InterfaceExample.x);
```

3. 比较

- 从设计层面上看，抽象类提供了一种 IS-A 关系，那么就必须满足里式替换原则，即子类对象必须能够替换掉所有父类对象。而接口更像是一种 LIKE-A 关系，它只是提供一种方法实现契约，并不要求接口和实现接口的类具有 IS-A 关系。
- 从使用上来看，一个类可以实现多个接口，但是不能继承多个抽象类。
- 接口的字段只能是 static 和 final 类型的，而抽象类的字段没有这种限制。
- 接口的方法只能是 public 的，而抽象类的方法可以由多种访问权限。

4. 使用选择

使用抽象类：

- 需要在几个相关的类中共享代码。
- 需要能控制继承来的方法和域的访问权限，而不是都为 public。
- 需要继承非静态（non-static）和非常量（non-final）字段。

使用接口：

- 需要让不相关的类都实现一个方法，例如不相关的类都可以实现 Compareable 接口中的 compareTo() 方法；
- 需要使用多重继承。

在很多情况下，接口优先于抽象类，因为接口没有抽象类严格的类层次结构要求，可以灵活地为一个类添加行为。并且从 Java 8 开始，接口也可以有默认的方法实现，使得修改接口的成本也变的很低。

> 深入理解 abstract class 和 interface
> When to Use Abstract Class and Interface

super

- 访问父类的构造函数：可以使用 super() 函数访问父类的构造函数，从而完成一些初始化的工作。
- 访问父类的成员：如果子类覆盖了父类的中某个方法的实现，可以通过使用 super 关键字来引用父类的方法实现。

```
public class SuperExample {
    protected int x;
    protected int y;

    public SuperExample(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void func() {
        System.out.println("SuperExample.func()");
    }
}
```

```
public class SuperExtendExample extends SuperExample {
    private int z;

    public SuperExtendExample(int x, int y, int z) {
        super(x, y);
        this.z = z;
    }

    @Override
    public void func() {
        super.func();
        System.out.println("SuperExtendExample.func()");
    }
}
```

```
SuperExample e = new SuperExtendExample(1, 2, 3);
e.func();
```

```
SuperExample.func()
SuperExtendExample.func()
```

> Using the Keyword super

覆盖与重载

- 覆盖（Override）存在于继承体系中，指子类实现了一个与父类在方法声明上完全相同的一个方法；
- 重载（Overload）存在于同一个类中，指一个方法与已经存在的方法名称上相同，但是参数类型、个数、顺序至少有一个不同。应该注意的是，返回值不同，其它都相同不算是重载。

五、String

String, StringBuffer and StringBuilder

1. 是否可变

- String 不可变
- StringBuffer 和 StringBuilder 可变

2. 是否线程安全

- 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 来同步

> String, StringBuffer, and StringBuilder

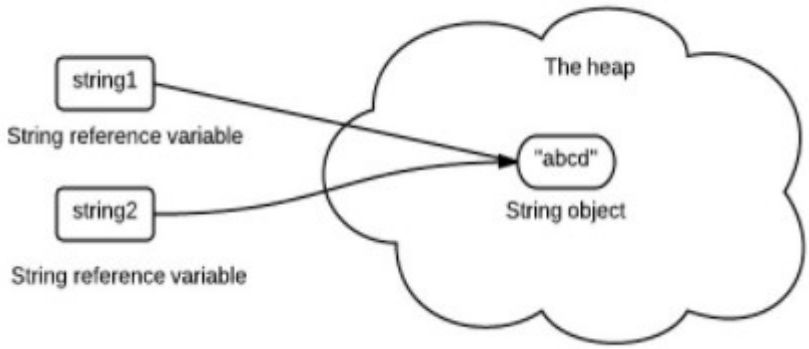
String 不可变的原因

1. 可以缓存 **hash** 值

因为 String 的 hash 值经常被使用，例如 String 用做 HashMap 的 key。不可变的特性可以使得 hash 值也不可变，因此只需要进行一次计算。

2. String Pool 的需要

如果一个 String 对象已经被创建过了，那么就会从 String Pool 中取得引用。只有 String 是不可变的，才可能使用 String Pool。



3. 安全性

String 经常作为参数，String 不可变性可以保证参数不可变。例如在作为网络连接参数的情况下如果 String 是可变的，那么在网络连接过程中，String 被改变，改变 String 对象的那一方以为现在连接的是其它主机，而实际情况却不一定是。

4. 线程安全

String 不可变性天生具备线程安全，可以在多个线程中安全地使用。

> Why String is immutable in Java?

String.intern()

使用 String.intern() 可以保证相同内容的字符串实例引用相同的内存对象。

下面示例中，s1 和 s2 采用 new String() 的方式新建了两个不同对象，而 s3 是通过 s1.intern() 方法取得一个对象引用，这个方法首先把 s1 引用的对象放到 String Poll（字符串常量池）中，然后返回这个对象引用。因此 s3 和 s1 引用的是同一个字符串常量池的对象。

```
String s1 = new String("aaa");
String s2 = new String("aaa");
System.out.println(s1 == s2);           // false
String s3 = s1.intern();
System.out.println(s1.intern() == s3);  // true
```

如果是采用 "bbb" 这种使用双引号的形式创建字符串实例，会自动地将新建的对象放入 String Poll 中。

```
String s4 = "bbb";
String s5 = "bbb";
System.out.println(s4 == s5);  // true
```

在 Java 7 之前，字符串常量池被放在运行时常量池中，它属于永久代。而在 Java 7，字符串常量池被放在堆中。这是因为永久代的空间有限，在大量使用字符串的场景下会导致 OutOfMemoryError 错误。

> What is String interning?
> 深入解析 String#intern

六、基本类型与运算

包装类型

八个基本类型：

- boolean/1
- byte/8
- char/16
- short/16
- int/32
- float/32
- long/64
- double/64

基本类型都有对应的包装类型，基本类型与其对应的包装类型之间的赋值使用自动装箱与拆箱完成。

```
Integer x = 2;     // 装箱
int y = x;         // 拆箱
```

new Integer(123) 与 Integer.valueOf(123) 的区别在于，new Integer(123) 每次都会新建一个对象，而 Integer.valueOf(123) 可能会使用缓存对象，因此多次使用 Integer.valueOf(123) 会取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
```

编译器会在自动装箱过程调用 valueOf() 方法，因此多个 Integer 实例使用自动装箱来创建并且值相同，那么就会引用相同的对象。

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

valueOf() 方法的实现比较简单，就是先判断值是否在缓存池中，如果在的话就直接使用缓存池的内容。

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

在 Java 8 中，Integer 缓存池的大小默认为 -128~127。

```
static final int low = -128;
static final int high;
static final Integer cache[];

static {
    // high value may be configured by property
    int h = 127;
    String integerCacheHighPropValue =
        sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
    if (integerCacheHighPropValue != null) {
        try {
            int i = parseInt(integerCacheHighPropValue);
            i = Math.max(i, 127);
            // Maximum array size is Integer.MAX_VALUE
            h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
        } catch( NumberFormatException nfe) {
            // If the property cannot be parsed into an int, ignore it.
        }
    }
    high = h;

    cache = new Integer[(high - low) + 1];
    int j = low;
    for(int k = 0; k < cache.length; k++)
        cache[k] = new Integer(j++);

    // range [-128, 127] must be interned (JLS7 5.1.7)
    assert IntegerCache.high >= 127;
}
```

Java 还将一些其它基本类型的值放在缓冲池中，包含以下这些：

- boolean values true and false
- all byte values

- short values between -128 and 127
- int values between -128 and 127
- char in the range \u0000 to \u007F

因此在使用这些基本类型对应的包装类型时，就可以直接使用缓冲池中的对象。

> [Differences between new Integer(123), Integer.valueOf(123) and just 123] (https://stackoverflow.com/questions/9030817/differences-between-new-integer123-integer-valueof123-and-just-123)

switch

从 Java 7 开始，可以在 switch 条件判断语句中使用 String 对象。

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

switch 不支持 long，是因为 swicth 的设计初衷是为那些只需要对少数的几个值进行等值判断，如果值过于复杂，那么还是用 if 比较合适。

```
// long x = 111;
// switch (x) { // Incompatible types. Found: 'long', required: 'char, byte, short, int,
Character, Byte, Short, Integer, String, or an enum'
//     case 111:
//         System.out.println(111);
//         break;
//     case 222:
//         System.out.println(222);
//         break;
// }
```

> Why can't your switch statement data type be long, Java?

七、反射

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。

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，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。

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。

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：

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

IDE 使用反射机制获取类的信息，在使用一个类的对象时，能够把类的字段、方法和构造函数等信息列出来供用户选择。

Advantages of Using Reflection:

- **Extensibility Features** ：An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.
- **Class Browsers and Visual Development Environments** ：A class browser needs to be able to enumerate the members of classes. Visual development environments can benefit from making use of type information available in reflection to aid the developer in writing correct code.
- **Debuggers and Test Tools** ：Debuggers need to be able to examine private members on classes. Test harnesses can make use of reflection to systematically call a discoverable set APIs defined on a class, to insure a high level of code coverage in a test suite.

Drawbacks of Reflection:

Reflection is powerful, but should not be used indiscriminately. If it is possible to perform an operation without using reflection, then it is preferable to avoid using it. The following concerns should be kept in mind when accessing code via reflection.

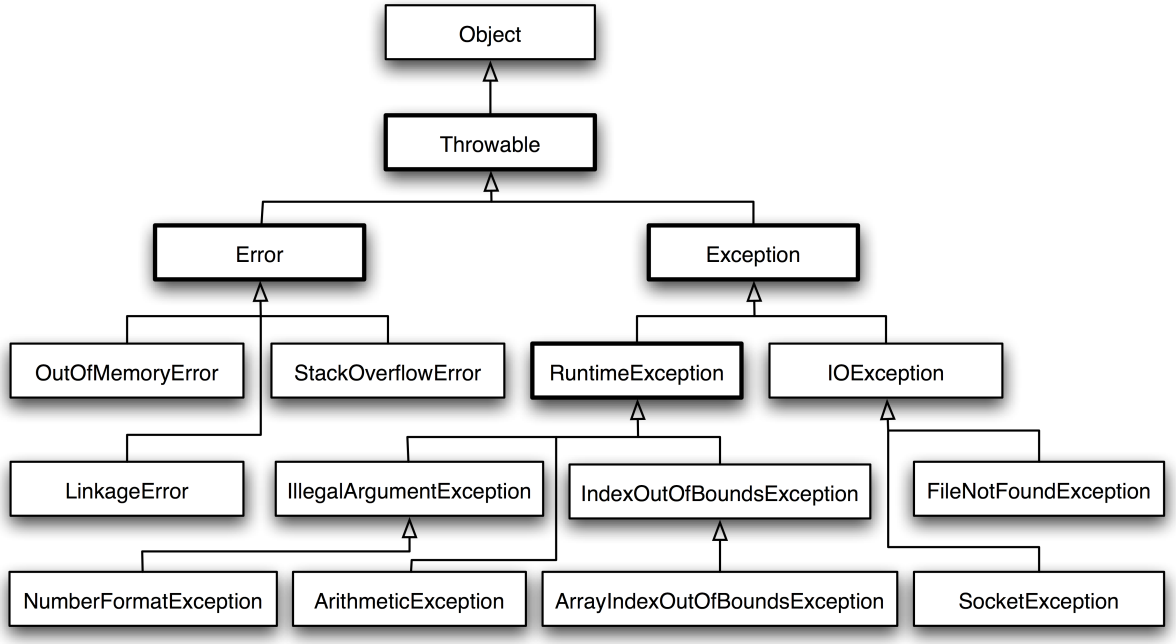
- **Performance Overhead** ：Because reflection involves types that are dynamically resolved, certain Java virtual machine optimizations can not be performed. Consequently, reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.
- **Security Restrictions** ：Reflection requires a runtime permission which may not be present when running under a security manager. This is in an important consideration for code which has to run in a restricted security context, such as in an Applet.
- **Exposure of Internals** :Since reflection allows code to perform operations that would be illegal in non-reflective code, such as accessing private fields and methods, the use of reflection can result in unexpected side-effects, which may render code dysfunctional and may destroy portability. Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.

> Trail: The Reflection API
> 深入解析 Java 反射（1）- 基础

八、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：

1. 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；
2. 非受检异常 ：是程序运行时错误，例如除 0 会引发 Arithmetic Exception，此时程序奔溃并且无法恢复。



> Java 入门之异常处理
> Java 异常的面试问题及答案 -Part 1

九、泛型

```
public class Box<T> {
    // T stands for "Type"
    private T t;
    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

> Java 泛型详解
> 10 道 Java 泛型面试题

十、注解

Java 注解是附加在代码中的一些元信息，用于一些工具在编译、运行时进行解析和使用，起到说明、配置的功能。注解不会也不能影响代码的实际逻辑，仅仅起到辅助性的作用。

> 注解 Annotation 实现原理与自定义注解例子

十一、特性

面向对象三大特性

> 封装、继承、多态

Java 各版本的新特性

New highlights in Java SE 8

1. Lambda Expressions
2. Pipelines and Streams
3. Date and Time API
4. Default Methods
5. Type Annotations
6. Nashhorn JavaScript Engine
7. Concurrent Accumulators
8. Parallel operations
9. PermGen Error Removed

New highlights in Java SE 7

1. Strings in Switch Statement
2. Type Inference for Generic Instance Creation
3. Multiple Exception Handling
4. Support for Dynamic Languages
5. Try with Resources
6. Java nio Package
7. Binary Literals, Underscore in literals
8. Diamond Syntax

> Difference between Java 1.8 and Java 1.7?
> Java 8 特性

Java 与 C++ 的区别

Java 是纯粹的面向对象语言，所有的对象都继承自 java.lang.Object，C++ 为了兼容 C 即支持面向对象也支持面向过程。

Java	C++
Java does not support pointers, templates, unions, operator overloading, structures etc. The Java language promoters initially said "No pointers!", but when many programmers questioned how you can work without pointers, the promoters began saying "Restricted pointers." Java supports what it calls "references". References act a lot like pointers in C++ languages but you cannot perform arithmetic on pointers in Java. References have types, and they're type-safe. These references cannot be interpreted as raw address and unsafe conversion is not allowed.	C++ supports structures, unions, templates, operator overloading, pointers and pointer arithmetic.
Java support automatic garbage collection. It does not support destructors as C++ does.	C++ support destructors, which is automatically invoked when the object is destroyed.
Java does not support conditional compilation and inclusion.	Conditional inclusion (#ifdef #ifndef type) is one of the main features of C++.
Java has built in support for threads. In Java, there is a `Thread` class that you inherit to create a new thread and override the `run()` method.	C++ has no built in support for threads. C++ relies on non-standard third-party libraries for thread support.
Java does not support default arguments. There is no scope resolution operator (::) in Java. The method definitions must always occur within a class, so there is no need for scope resolution there either.	C++ supports default arguments. C++ has scope resolution operator (::) which is used to to define a method outside a class and to access a global variable within from the scope where a local variable also exists with the same name.
There is no *goto* statement in Java. The keywords `const` and `goto` are reserved, even though they are not used.	C++ has *goto* statement. However, it is not considered good practice to use of *goto* statement.

Java	C++
Java doesn't provide multiple inheritance, at least not in the same sense that C++ does.	C++ does support multiple inheritance. The keyword `virtual` is used to resolve ambiguities during multiple inheritance if there is any.
Exception handling in Java is different because there are no destructors. Also, in Java, try/catch must be defined if the function declares that it may throw an exception.	While in C++, you may not include the try/catch even if the function throws an exception.
Java has method overloading, but no operator overloading. The `String` class does use the `+` and `+=` operators to concatenate strings and `String` expressions use automatic type conversion, but that's a special built-in case.	C++ supports both method overloading and operator overloading.
Java has built-in support for documentation comments (`/** ... */`); therefore, Java source files can contain their own documentation, which is read by a separate tool usually `javadoc` and reformatted into HTML. This helps keeping documentation maintained in easy way.	C++ does not support documentation comments.
Java is interpreted for the most part and hence platform independent.	C++ generates object code and the same code may not run on different platforms.

> What are the main differences between Java and C++?

JRE or JDK

- JRE is the JVM program, Java application need to run on JRE.
- JDK is a superset of JRE, JRE + tools for developing java programs. e.g, it provides the compiler "javac"

参考资料

- Eckel B. Java 编程思想[M]. 机械工业出版社, 2002.
- Bloch J. Effective java[M]. Addison-Wesley Professional, 2017.