Java Overview

        - 单元素集合
        - 排序
        - 洗牌
        - 不可变集合
        - 线程安全集合
    - 重写（Override） VS 重载（Overload）

语法

基础

classpath

- classpath 为JVM用到的一个环境变量，指示JVM如何搜索class。
- 表示一组目录的集合
    - windows下，;分割，带空格的目录用""括起来
        - C:\work\project1\bin;C:\shared;"D:\My Documents\project1\bin"
    - linux下，用:分割
        - /usr/shared:/usr/local/bin:/home/liaoxuefeng/bin
- classpath的设定
    - 系统环境变量中设置
    - 启动JVM时设置
        - java -classpath .;C:\work\project1\bin;C:\shared abc.xyz.Hello
        - -cp 简写java -cp .;C:\work\project1\bin;C:\shared abc.xyz.Hello
        - 没有设置系统环境变量，也没有传入-cp参数，那么JVM默认的classpath为.，即当前目录

jar

- jar包就是zip包,把后缀从.zip改为.jar
- jar包里的第一层目录，不能是bin，而应该是hong、ming、mr
- MANIFEST.MF
    - 特殊的/META-INF/MANIFEST.MF文件
    - 指定Main-Class和其它jar包的信息
    - JVM会自动读取这个MANIFEST.MF文件，如果存在Main-Class，我们就不必在命令行指定启动的类名
        - java -jar hello.jar

变量

- $a 合法标识符
- java 数据类型
    - 内置
        - byte 默认0
        - int 默认0
        - long 默认0L/0l
        - float 0.0f
        - double 0.0d
        - boolean false

        - char 16bit Unicode字符，最小\0 \u0000，最大65535 \uffff
        - Byte.SIZE Byte.MIN_VALUE Byte.MAX_VALUE
    - 引用类型
- java 变量类型
    - 局部变量没有默认值，声明后，必须初始化后才能用
    - 成员/实例变量
        - public 能被所有class访问
        - private 只能被该类访问
        - protectd 本类访问+子类访问+同一个package中类访问
        - default 同一个package中类访问，即使继承了父类，不在一个package也不能访问
    - 类变量
        - 类变量多声明为常量,public/private,final,static类变量

修饰符

- public类、方法、构造方法、接口能被任何其他类访问，若相互访问的public类不在同一包中，需导入相应的包
- private方法、变量、构造方法只能被所属类访问，且类和接口不能private(内部类除外)
- protected可修饰成员、构造方法、方法成员，不能修饰类(内部类除外),接口及接口的成员变量/成员方法不能protected
    - 子类基类同一包中
    - 子类基类不在同一包，子类可以访问从基类继承类的protected方法，不能访问基类实例的protected方法
- 继承规则
    - 父类声明public的方法，子类必须public
    - 父类protected，子类protected/public，不能private
    - 父类private,不能被继承
- 非访问修饰符
    - static 修饰类方法/类变量
    - final修饰类、方法、变量，final类不能被继承，final方法不能被继承类重定义，final变量为常量
        - final变量，必须显示指定初始值 ，或者在定义时初始化，或者在类的构造函数中初始化
        - final 和static一起使用创建类常量
    - abstract 创建抽象类，抽象方法，
        - 一个类不能同时abstract && final
        - 包含一个抽象方法（方法声明为abstract），则该类为抽象类
        - 抽象方法不能final/static
        - 继承抽象类的子类必须实现父类所有抽象方法，除非子类也是抽象类
        - VS interface
        - interface 没有构造函数 abstract 类有构造函数
        - interface内部任何方法不允许实现，abstract类允许有一般非abstract的方法，有具体实现
        - interface内部没有super，this变量，abstract类有
        - interface成员变量一定常数，abstract类成员变量为一般变量
        - interface所有成员public，abstract类可以任何等级
        - interfact成员变量默认public (允许继承)static（和实例无关，类本身） final（接口定义的常亮不能被修改）
    - synchronized
        - 该方法同一时间只能被一个线程访问

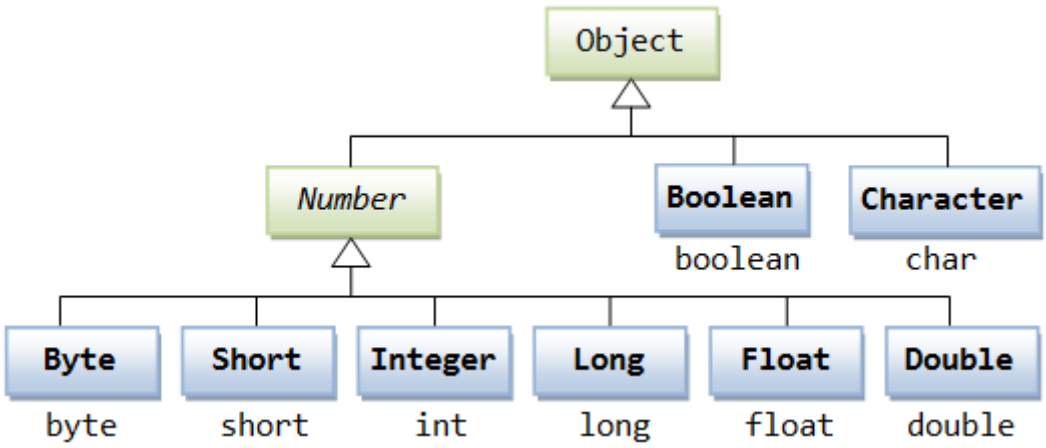
    - transient
        - 包含在定义变量的语句中，预处理类和变量的数据类型。不会被持久化
    - volatile
        - 修饰变量在每次被线程访问时，都强制从共享内存中重新读取变量值。当成员变量发生变化时，强制线程将变化值写回内存，两个不同线程总是看到某个成员变量的同一个值

运算符

- instanceof
    - 检查对象是否是一个特定类型（类类型/接口类型）
    - boolean result = name instanceof String;
- switch-case
    - switch必须为byte、short、int、char、String
    - case必须为字符常量、字面量

常用类

Number&Math类



- xxxValue
    - byteValue 以byte形式返回指定数值
- compareTo
- equals
- valueof 给定参数的原生Number对象值，参数可以原生数据类型或String,为静态方法
    - static Integer.valueof(int i)
    - static Integer.valueOf(String s)
    - static Integer.valueOf(String s,int radix) radix指定使用的进制数
- Character
    - char的封装类型
        - isLetter() 是否是字母
        - isDigit()是否数字
        - isWhitespace()是否空白字符
        - isUpperCase()是否大写字母
        - toString()返回字符字符串形式

String

- 常用函数
  - char charAt(int index)
  - int compareTo(Obejct o)
  - int compareToIgnoreCase(String str)
  - String concat(String str)
  - boolean contentEquals(StringBuffer sb) 判断二者内容相同
  - static String copyValueOf(char[] data) 将char数组转为String
  - staic String copyValueOf(char[] data,int offset,int count)
  - boolean endsWith(String suffix)
  - boolean equals(Obejct s) 只能比较两个String内容是否相同
  - int hashCode()
  - String intern() 返回字符串对象的规范化表示
    - 检查字符串池中是否有该字符串，如果存在则返回池中的该字符串的引用。否则将该字符串添加进池并返回引用
  - new String(byte[]) //将byte数组转为String
  - String replace(CharSequence target, CharSequence replacement) //替换字符串并返回新的字符串
  - String[] split(String regex)//根据正则切分

```
str="A,B,C,";
   String[] ss=str.split("\\,");
   System.out.println(Arrays.toString(ss));
```

  - trim()去除字符串首尾空白
  - strip() 去除字符串首尾空白+类似中文空格字符\u3000也会去除(trim 不会！！)
  - boolean isEmpty()
  - boolean isBlank()
  - static String join(CharSequence delimiter, CharSequence... elements) //根据间隔符将序列Join
  - static String valueOf(Object obj)//将Object 转为String
  - Integer.parseInt（）Boolean.parseBoolean("true") //将String转为其他
  - char[] toCharArray() new String(char[])//转为char数组
  - 编码转换
    - byte[] b2="Hello".getBytes("UTF-8")
    - String s1=new String(b2,"GBK")//将byte[] 转为GBK的String
    - Java的String和char在内存中以Unicode编码

StringBuffer/Builder

- 对字符串多次修改，且不产生新的对象
- String Buffer 线程安全，慢
- StringBuilder 线程不安全，快
- 对于普通的String +操作，java在编译时自动把多个连续的+操作编码为StringConcatFactory操作，运行期，StringConcatFactory 自动把字符串连接操作优化为数组复制或者StringBuilder操作
- public StringBuffer append(String s)
- public StringBuffer reverse()

- public delete(int start,int end)
- public insert(int offset,int i)在offset处，将int字符串插进去
- replace(int start,int end,String str)
- indexOf(String str) 返回第一次出现的子字符串的索引
- indexOf(String str,int fromIndex)
- int lastIndexOf()
- CharSequence subSequence(int start,int end)

StringJoiner

- 分隔符拼接数组

```
StringJoiner sj=new StringJoiner(",");
sj.add("name");
sj.add("hello");
sj.add("world");
System.out.println(sj.toString());//name,hello,world

 String[] names = {"Bob", "Alice", "Grace"};
var sj = new StringJoiner(", ", "Hello ", "!"); //指定拼接头 、拼接尾
for (String name : names) {
   sj.add(name);
}
System.out.println(sj.toString());//Hello Bob, Alice, Grace!
```

包装类型

- 将一个基本类型视为引用(引用类型)
- 引用可以null，基本类型不可以null
- auto boxing / unboxing自动装箱/拆箱
    - int i=100; Integer n=Integer.valueof(i);
    - Integer n=100; 自动调用Integer.valueOf ---- int->Integer成为自动装箱
    - int x=n; 自动调用Integer.intValue() ---- Integer->int
    - 装箱拆箱影响效率，当Integer为null时，拆箱引发NullPointerException
- 所有包装类型为不变类
    - 定义class时final
    - 每个字段final修饰，保证创建实例后无法修改字段
    - 为了保证不变类的比较，还需要正确覆写equals()和hashCode()方法

```
public final class Integer{
  private final int value;
}
```

- 包装类型比大小 equals()
    - 为了节省内存，Integer.valueOf()对于较小的数，始终返回相同的实例
- 创建Integer

    - Integer n=new Integer(100) 总是创建新的Integer实例
    - Integer n=Integer.valueOf(100) 尽可能返回缓存的实例，优选！！！
        - 能创建新对象的静态方法为静态工厂方法，优选！！！
- 进制转换
    - int x1=Integer.parseInt("1000")
    - int x2=Integer.parseInt("100",16)//16进制
    - Integer.toString(100)
    - Integer.toString(100, 36)//36进制
    - Integer.toOctalString(100)//8进制
    - Integer.toBinaryString(100)//2进制字符串
    - Integer.MAX_VALUE;
    - Long.SIZE
    - Long.BYTES
    - Byte.toUnsignedInt(x) 把一个负的byte按无符号整型转换为int
    - 把一个short按unsigned转换为int，把一个int按unsigned转换为long

java bean

- 如果class定义满足,则称为JavaBean
    - private实例字段
    - public 读写字段
        - 读写方法满足
            - public Type getXyz()
            - public void setXyz(Type value)

枚举 #archor

enum是一个class，每个枚举的值都是class实例

- 优势
    - 带有类型信息
    - 不会引用非枚举的值
    - 不同枚举类型不能比较/赋值 安全
- 同其他class 区别
    - 枚举类可以有自己构造函数，默认private
    - 只能定义出enum的实例，而无法通过new操作符创建enum的实例
    - enum继承自java.lang.Enum，且无法被继承
    - 可以将enum类型用于switch语句
    - 枚举类的字段也可以是非final类型，即可以在运行期修改，但是不推荐这样做!
- 静态方法values() 返回当前类中所有值
- 成员方法ordinal() 找到枚举常量的索引
- 成员方法valuesOf("SUN") 返回指定字符串的枚举常量

```
public class EnumTest {
 enum Color{
   RED,BLUE,GREEN;
  private Color(){
     System.out.println("constructor called for "+this.toString());
```

```
    }
    public void colorinfo(){
        System.out.println("Universe color");
    }
}

public static void main(String[] args) {
    Color cr=Color.BLUE;
    for(Color c:Color.values()){
        System.out.println(c);
    }
    System.out.println(cr.ordinal());//=1 返回索引
    Color red=Color.valueOf("RED");
    System.out.println(red);//=RED返回指定字符串值的枚举常量
}
}
```

```
public class Main {
public static void main(String[] args) {
    Weekday day = Weekday.SUN;
    if (day.dayValue == 6 || day.dayValue == 0) {
        System.out.println("Work at home!");
    } else {
        System.out.println("Work at office!");
    }
}
}

enum Weekday {
MON(1), TUE(2), WED(3), THU(4), FRI(5), SAT(6), SUN(0); //给每个枚举常量添加字 段 类似构造函数
public final int dayValue;

private Weekday(int dayValue) {
    this.dayValue = dayValue;
}
}
```

- 对枚举常量调用toString()会返回和name()一样的字符串。但是，toString()可以被覆写，而name()则不行。
- 判断枚举常量的名字，要始终使用name()方法，绝不能调用toString(),因为可能被复写 😆
- 覆写toString()的目的是在输出时更有可读性

```
enum Weekday {
    MON(1, "星期一"), TUE(2, "星期二"), WED(3, "星期三"), THU(4, "星期四"), FRI(5, "星期五"),
SAT(6, "星期六"), SUN(0, "星期日");

    public final int dayValue;
    private final String chinese;
    private Weekday(int dayValue, String chinese) {
```

```
        this.dayValue = dayValue;
        this.chinese = chinese;
    }
    @Override
    public String toString() {
        return this.chinese;
    }
}
```

BigInteger

在Java中，由CPU原生提供的整型最大范围是64位long型整数。 java.math.BigInteger就是用来表示任意大小的整数。BigInteger内部用一个int[]数组来模拟一个非常大的整数，不变类，继承自Number

- BigInteger bi = new BigInteger("1234567890");
- 对BigInteger做运算的时候，只能使用实例方法, (没有相应的操作符重载 😆)

```
BigInteger bi=new BigInteger("1234567890");
    BigInteger bi2=new BigInteger("1234567890");
    System.out.println(bi.add(bi2));
```

- 转换为有限类型
    - i.longValue() byteValue() intValue() doubleValue()
    - longValueExact() 超范围时，抛出异常ArithmeticException

BigDecimal

表示任意大小且精度准确的浮点数

- scale() 表示小数位数

```
BigDecimal d1 = new BigDecimal("123.45");
System.out.println(d1.scale()); // 2,两位小数
```

- stripTrailingZeros 将一个BigDecimal格式化为一个相等的，但去掉了末尾0的BigDecimal
- BigDecimal的scale()返回负数，例如，-2，表示这个数是个整数，并且末尾有2个0
- 对一个BigDecimal设置它的scale，如果精度比原始值低，那么按照指定的方法进行四舍五入或者直接截断

```
BigDecimal d1 = new BigDecimal("123.456789");
BigDecimal d2 = d1.setScale(4, RoundingMode.HALF_UP); // 四舍五入，123.4568
BigDecimal d3 = d1.setScale(4, RoundingMode.DOWN); // 直接截断，123.4567
```

    - 对BigDecimal做加、减、乘时，精度不会丢失，但是做除法时，存在无法除尽的情况，这时，就必须指定精度以及如何进行截断：

```
BigDecimal d1 = new BigDecimal("123.456");
BigDecimal d2 = new BigDecimal("23.456789");
BigDecimal d3 = d1.divide(d2, 10, RoundingMode.HALF_UP); // 保留10位小数并四舍五入
BigDecimal d4 = d1.divide(d2); // 报错：ArithmeticException，因为除不尽
```

- 对BigDecimal做除法的同时求余数：

```
BigDecimal n = new BigDecimal("12.345");
BigDecimal m = new BigDecimal("0.12");
BigDecimal[] dr = n.divideAndRemainder(m);
System.out.println(dr[0]); // 102
System.out.println(dr[1]); // 0.105
```

- 总是使用compareTo()比较两个BigDecimal的值，不要使用equals()！
- 使用equals()方法不但要求两个BigDecimal的值相等，还要求它们的scale()相等

数组

- public static void fill(int[] a,int val) val初始化所有a
- public static boolean equals(long[]a,long[] b) 相同顺序，相同数据，则相同
- Date
  - boolean before(Date date) 调用对象在date对象之后
  - long getTime() 返回毫秒数
  - String toString()
  -
```
//统计运行时间
long start=Sytem.currentTimeMills()
long end=Sytem.currentTimeMills();
```
  - Calender 用于设置/获取日期的特定部分
    - date=0 表示上一个月的最后一天，-1 表示上一个月的倒数第二天，一次类推月份
    -
```
Calendar c=Calender.getInstance()
c1.set(2009, 6 - 1, 12);
Calendar.DAY_OF_WEEK
Calendar.HOUR_OF_DAY //24小时 小时
Calendar.HOUR
c1.add(Calendar.DATE, 10);
 c.set(2017,1,1);  2017 1 1
 c.set(2017,1,0);  2017 0 31
 c.set(2017,0,0); 2016 11 31
 c1.set(2017, 2, -10); 2017 1 18
```

正则表达式

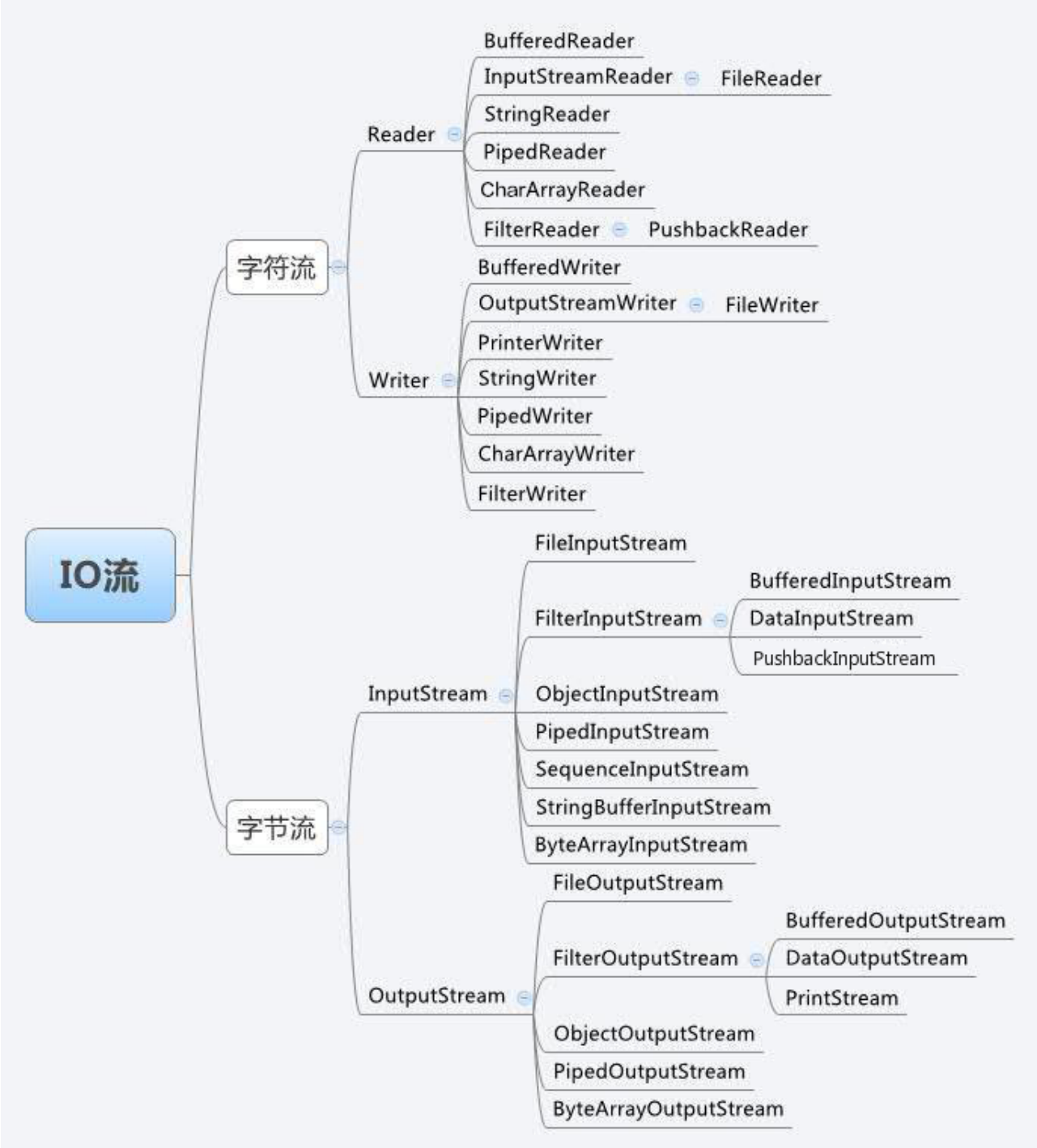
- Pattern 正则表达式的编译表示,接受正则表达式作为参数
  - Pattern m=Pattern.compile(regrex)
- Matcher对输入字符串进行解释和匹配操作
  - 调用pattern对象的matcher方法获得Matcher对象
- PatternSyntaxException 非强制异常类
- 捕获组
  - 将多个字符当一个单独单元处理，通过对括号内的字符分组来创建
  - 通过从左至右计算开括号 进行编号
  - 通过matcher的groupCount查看分组个数。
    - group(0)代表整个表达式，所以不在count内

```
  // 按指定模式在字符串查找
String line = "This order was placed for QT3000! OK?";
String pattern = "(\\D*)(\\d+)(.*)";

// 创建 Pattern 对象
Pattern r = Pattern.compile(pattern);

// 现在创建 matcher 对象
Matcher m = r.matcher(line);
if (m.find( )) {
  System.out.println("Found value: " + m.group(0) );
  System.out.println("Found value: " + m.group(1) );
  System.out.println("Found value: " + m.group(2) );
  System.out.println("Found value: " + m.group(3) );
} else {
  System.out.println("NO MATCH");
}
 $ 匹配结束
 * 零次或多次匹配前面表达式
 + 1次或多次匹配前面表达式
 {n} n>=0 匹配n次
 {n,} n>=0至少匹配n次
 {n,m}  0<=n<=m 至少n次，至多m次
 ? 当此字符紧随其他限定符(*、+、？ 、{n}、{n,m}、{n,}) 表示匹配模式非贪心
 x|y 匹配x或y
 [xyz] 匹配包含的任一字符
 [a-z]
 [^a-z] 反向范围匹配
 \d 数字匹配
 \D 非数字匹配
 \n 换行符
 \s 任意空白
 \S 任意非空白
 \w 任意字类字符，包括_ 等同[A-Za-z0-9]
 \W 任意非单词字符
 \b匹配一个字边界，即字与空格键的位置 er\b 表示以er为结尾或者其后有空格的单词
 \B
```

- Matcher索引方法
  - public int start() 返回之前匹配的初始索引
  - public int start(int group) 给定组捕获的子序列的初始索引
  - public int end() 匹配字符后的偏移量
  - public int end(int group)
  - public boolean lookingAt() 将从区域开头的输入序列与该模式匹配，不要求全部匹配
  - boolean find() 查找该模式匹配的输入序列的下一个序列
  - boolean matches() 整个区域与模式是否匹配,要求全部匹配
- Stream File IO



  - 控制台
    - 输入
      - 控制台输入由System.in完成，为获得一个绑定到控制台的字符流，将System.in包装在BufferedReader对象中

```
int read( ) throws IOException //从BufferReader对象读取一个字符，流结束返回-1
String readLine() throws IOException //读取字符串

    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
char cha;
do{
    cha=(char)br.read();
    System.out.println(cha);
}while(cha != 'q');

String str;
do{
    str = br.readLine();
    System.out.println(str);
}while(!str.equals("end"));
String strend = "end";
System.out.println(strend =="end");//true
```

- 输出 输出由print println完成，由PrintStream定义，System.out是该类的一个引用。PrintStream继承OutputStream类，且实现了write方法

```
void write(int byteval) //将byteval的低八位字节写到流中
 int b='A';
System.out.write(b);
```

面向对象

方法

- 可变参数
  - 一个方法只能指定一个可变参数，且必须是方法的最后一个参数

```
 public static void print(double... num){
if(num.length==0) {
   System.out.println("no argument");
}
for(double n:num){
   System.out.println(n);
  }
}
```

- finalize 方法
  在垃圾收集器之前调用，可用来清除回收对象

```

```

```
//常用格式
protected void finalize(){
}
```

继承

- extends 可以继承一个类，implements 可以多个接口
- 通过super接口调用父类方法/变量，尤其子类构造函数需要调用父类构造函数的时候
  - 任何类的构造方法，编译器默认会在子类的构造函数调用父类的默认构造函数，调用失败则异常
- 向上转型
  - 子类向上转型为父类
  - isAssignableFrom 判断上转型是否成立

```
Number.class.isAssignableFrom(Integer.class)
```

    - instanceof 判断一个实例是否是某个类型

```
Object n = Integer.valueOf(123);
boolean isDouble = n instanceof Double;
```

- 向下转型
  - 只有当父类原来是一个子类的引用时，转型成功（父类由子类向上转型而成）
  - 通过 instanceof 判断

```
Person p = new Student();
if (p instanceof Student) {
  // 只有判断成功才会向下转型:
  Student s = (Student) p; // 一定会成功
}
```

- 继承VS组合
  - 继承 is a
  - 组合 has a
  - 当需要做向上转型时，考虑继承

多态

- 所有类均继承自Obejct，重写override Object方法
  - String toString()
  - boolean equals(Object o)
  - int hashCode()
- final
  - 父类方法final,父类不允许子类对其方法override

    - 变量final，意味着初始化后不可修改
        - 定义时初始化
        - 在类的构造函数中初始化 !!!

接口

- 接口定义变量默认 pulibc static final
- 接口定义方法默认 public abstract
- 使用时，实例化对象永远为某个具体的子类，通过接口去引用它，因为接口比抽象类更抽象
- default方法
    - 给子类增加方法，且子类不用实现

```
interface Person {
String getName();
default void run() {
  System.out.println(getName() + " run");
}
}

class Student implements Person {
private String name;

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

public String getName() {
  return this.name;
}
}
```

包

- 包不具有继承性质
- import

```
import mr.jun.* //不包括子包的class
import static java.lang.System.*    //导入静态字段和方法
```

- 默认导入
    - 默认import当前package的其他class
    - 默认import java.lang.*

文件

- 输入
    - FileInputStream

```
void close() throws IOException{} 关闭文件输入流以及有关的系统资源
void finalize() throws IOException{} 清除与该文件的连接，确保在不再引用文件输入流时调用其 close 方法
int read()
int read(byte[] r)//读取r的size大小的字节
int available() 返回下一次对此输入流调用的方法可以不收阻塞的从此输入流读取的字节数
```

```
InputStream is=new FileInputStream("/home/bliss/sn.txt");
//      File f=new File("/home/bliss/sn.txt");
//      InputStream is2=new FileInputStream(f);
 byte[] bytes=new byte[512];
  is.read(bytes);
  int oneint = 0;
  oneint=is.read();
 System.out.println(new String(bytes));
```

- 输出
  - FileOutputStream

```
void close() throws IOException{}
void finalize()throws IOException {}
void write(int w)throws IOException{}
void write(byte[] w)
```

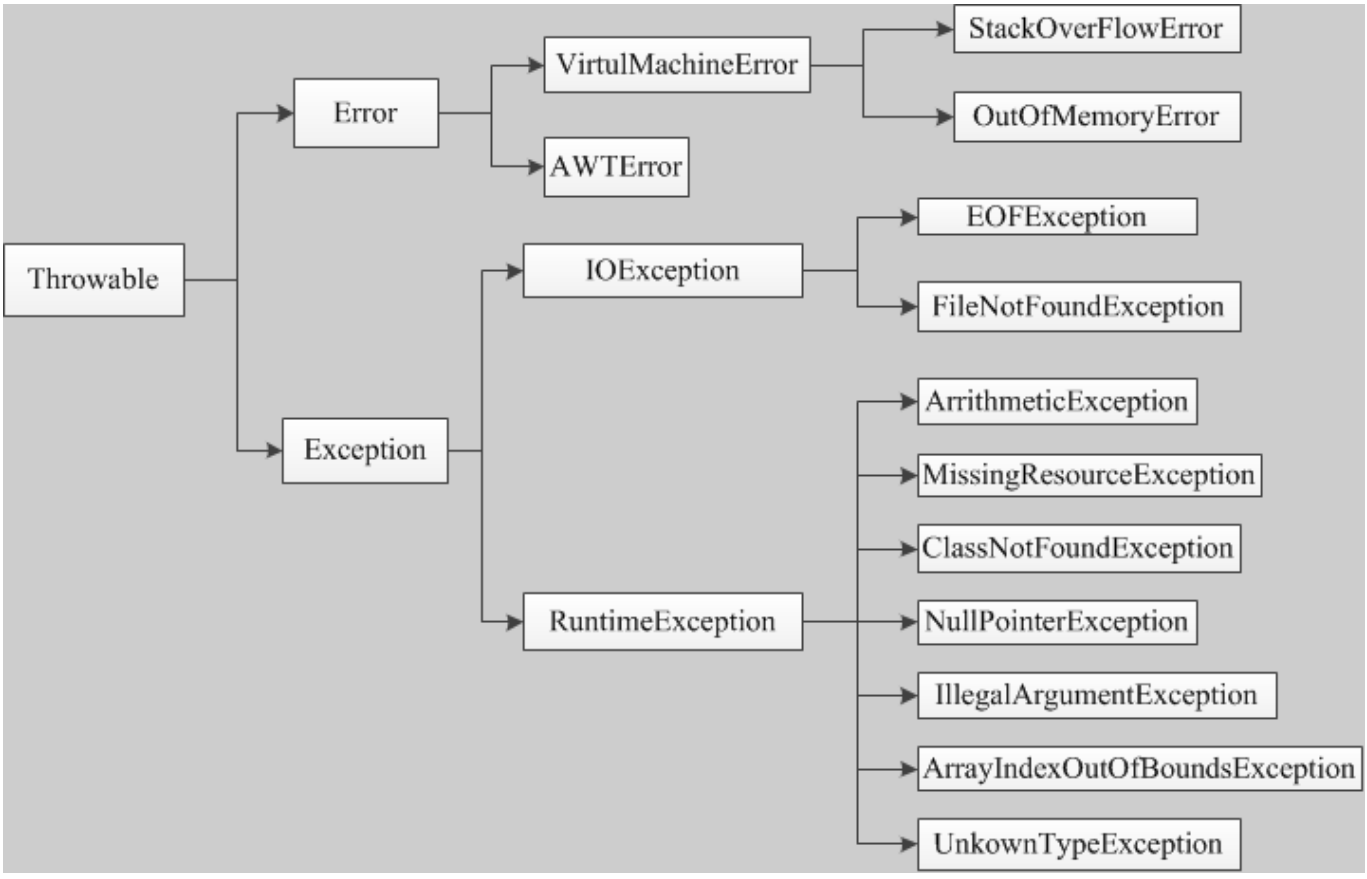
```
OutputStream os = new FileOutputStream("/home/bliss/sn.txt")
File f = new File("/home/bliss/sn.txt")
OutputStream f = new FileOutputStream(f)
```

- 目录
  - File FileReader FileWriter
    - 创建目录--File对象方法
      - boolean mkdir() 用于创建文件夹。失败 File对象指定的文件夹存在或整个路径不存在
      - boolean mkdirs()创建文件夹及其父文件夹
    - 读取目录--File
      - 目录即File对象，包含其他文件和文件夹
      - boolean isDirectory()
      - String[] list() 包含的文件以及文件夹
    - 删除目录/文件--File
      - boolean delete()
- Scanner 类

- 用于获取用户输入
- next 一定读取有效字符后才结束，可忽略有效字符前的空白，将有效字符后的空格作为结束，不能获得空白
- nextLine 以回车为结束符，可空白

```
Scanner scanner=new Scanner(System.in);
  if(scanner.hasNextLine()){
     String str=scanner.nextLine();
     System.out.println(str);
  }
  if(scanner.hasNext()){
     String str=scanner.next();
     System.out.println(str);
  }
  scanner.close();
```

异常



- 必须捕获的异常Exception 及其子类，但不包括RuntimeException及其子类，---检查性异常 不处理编译不通过
- 运行时异常
- 错误
- 所有未捕获的异常，最终也必须在main()方法中捕获，不会出现漏写try的情况。这是由编译器保证的。main()方法也是最后捕获Exception的机会
- 异常也可以不捕获，再把异常抛出 throws，也可以通过编译器检查
- 捕获异常后至少 调用printStackTrace()方法打印异常栈

```
//catch 中异常依次往下越来越抽象
try{
}catch{}
catch{}
finally{
  //一定会执行
}
  try {
     process1();
     process2();
     process3();
  } catch (IOException | NumberFormatException e) { // 多个同级异常时 IOException或
NumberFormatException
     System.out.println("Bad input");
  } catch (Exception e) {
     System.out.println("Unknown error");
  }
void withdraw(double amount) throws RemoteException,InsufficientFundsException{
}
//自定义异常
class RemoteException extends Exception{
}
```

- 断言
  调试方式，失败返回AssertionError异常，断言不能用于可恢复的程序错误，只应该用于开发和测试阶段 对于可恢复的程序错误，不应该使用断言，应该抛出异常并在上层捕获
  - assert x >= 0 : "x must >= 0";//添加一个可选的断言消息
  - JVM默认关闭断言指令，即遇到assert语句就自动忽略了，不执行
  - java -ea Main.java 启用assert
- 日志
  - java.util.logging JDK的Logging定义了7个日志级别
  - SEVERE | WARNING | INFO | CONFIG | FINE | FINER | FINEST
  - Logging系统在JVM启动时读取配置文件并完成初始化，一旦开始运行main()方法，就无法修改配置
  - 配置不太方便，需要在JVM启动时传递参数-Djava.util.logging.config.file=

    ```
    import java.util.logging.Level;
    import java.util.logging.Logger;
    Logger logger = Logger.getGlobal();
      logger.info("start process...");
      logger.warning("memory is running out...");
      logger.fine("ignored.");
      logger.severe("process will be terminated...");
    ```

  - SLF4J+Logback

反射

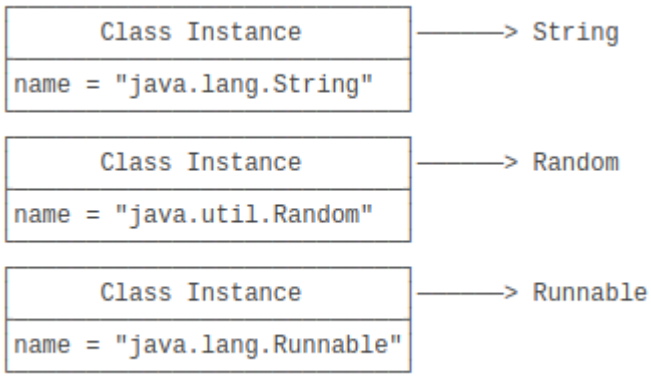
反射--程序在运行期间可以拿到一个对象的所有信息，解决在运行期间，对某个实例一无所知的情况下，如何调用其方法

Class 类

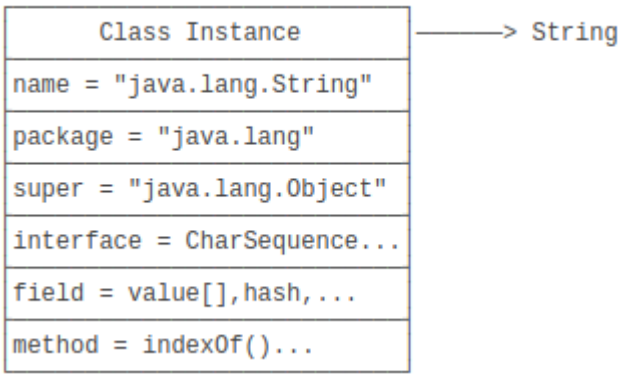
- class 本质是数据类型，没有继承关系的数据类型无法相互赋值
- class 类时JVM在执行过程中动态加载的，第一次读取到一种class类型时，将其加载进内存。
- 每加载一种class类，JVM创建一个Class 类型的实例，并将两者关联起来（Class 一个名字为Class的class类 😆😕）

```
public final class Class{
  private Class{}//只有JVM可以构造
}
```

  - 新建String类为例 ，JVM加载String时，读取String.class到内存，为String创建一个Class实例并关联
  - Class cls=new Class(String)
  - 如图JVM每个Class实例都指向一个数据类型（class或interface）



- 一个Class包含class的所有信息，**通过Class实例获得类String信息的方法为反射**



- 获取一个class类绑定的Class
  - Class cls=String.class; //通过一个class的静态变量
  - String s="hello";Class cls=s.getClass() class实例的getClass接口

- Class cls=Class.forName("java.lang.String") 根据一个class的完整类名

- Class 实例比较
  - Class 实例在JVM中是唯一的

```
Class cls1=String.class;
Class cls2=s.getClass();
boolean same=cls1==cls2;//true
```

```
 Integer n = new Integer(123);
  boolean b1 = n instanceof Integer; // true，因为n是Integer类型
  boolean b2 = n instanceof Number; // true，因为n是Number类型的子类
  boolean b3 = n.getClass() == Integer.class; // true，因为n.getClass()返回
Integer.class
  boolean b4 = n.getClass() == Number.class; // false，因为
Integer.class!=Number.class
```

  - 获取class 信息

```
 System.out.println("Class name: " + cls.getName());
System.out.println("Simple name: " + cls.getSimpleName());
if (cls.getPackage() != null) {
   System.out.println("Package name: " + cls.getPackage().getName());
}
System.out.println("is interface: " + cls.isInterface());
System.out.println("is enum: " + cls.isEnum());
System.out.println("is array: " + cls.isArray());
System.out.println("is primitive: " + cls.isPrimitive());
/**
 Class name: java.lang.String  --------String
 Simple name: String
 Package name: java.lang
 is interface: false
 is enum: false
 is array: false
 is primitive: false

Class name: [Ljava.lang.String;   ---------String[]
Simple name: String[]
is interface: false
is enum: false
is array: true
is primitive: false
**/
```

- 动态加载
  JVM在执行Java程序的时候，并不是一次性把所有用到的class全部加载到内存，而是第一次需要用到class时才加载，可以在运行期根据条件来控制加载class

```
// Commons Logging优先使用Log4j:
  LogFactory factory = null;
  if (isClassPresent("org.apache.logging.log4j.Logger")) {
    factory = createLog4j();
  } else {
    factory = createJdkLog();
  }

  boolean isClassPresent(String name) {
    try {
      Class.forName(name);
      return true;
    } catch (Exception e) {
      return false;
    }
  }
```

访问字段

更多的给工具或者底层框架使用，目的是在不知道目标实例任何信息的情况下，获取特定字段的值

- 通过Class实例获取字段信息
  - Field getField(name) 获取public，包括从父类继承的字段
  - Field getDeclaredField(name) 获取本类所有字段，包括private，不能获取继承的字段（获取的private字段，人不能访问该字段的值，除非 setAccessible(true)）
  - Field[] getFields() 获取所有public字段(包括父类)
  - Field[] getDeclaredFields()
  - 一个Field对象包含一个字段的所有信息
    - getName 返回字段名称
    - getType 字段类型 String.class
    - getModifiers() 字段修饰符 private/public/final

```
 public final class String{
   private final byte[] value;
 }

 Field f = String.class.getDeclaredField("value");
 f.getName(); // "value"
 f.getType(); // class [B 表示byte[]类型
 int m = f.getModifiers();
 Modifier.isFinal(m); // true
 Modifier.isPublic(m); // false
 Modifier.isProtected(m); // false
```

```
Modifier.isPrivate(m); // true
Modifier.isStatic(m); // false
```

- 获取字段值
  - Field.get(Object) 获取指定实例的指定字段的值

```
Object p = new Person("Xiao Ming")
Class c = p.getClass();
Field f = c.getDecalaredField("name");
f,setAccessible(true); //不管这个字段是否是public，一律通过访问
Object value = f.get(p);//获取指定实例的指定字段的值
```

  - setAccessible(true)可能会失败,如果JVM运行期存在SecurityManager，那么它会根据规则进行检查，有可能阻止setAccessible(true)
- 设置字段值
  - 修改非public字段，需要首先调用setAccessible(true)
  - Field.set(Object,Object) //指定的实例/待修改的值

```
  public class Main {
public static void main(String[] args) throws Exception {
  Person p = new Person("Xiao Ming");
  System.out.println(p.getName()); // "Xiao Ming"
  Class c = p.getClass();
  Field f = c.getDeclaredField("name");
  f.setAccessible(true);
  f.set(p, "Xiao Hong");
  System.out.println(p.getName()); // "Xiao Hong"
}
}

class Person {
  private String name;
  public Person(String name) {
    this.name = name;
  }

  public String getName() {
    return this.name;
  }
}
```

调用方法

- Method
  - 获取method
  - Method getMethod(name,class) 获取public 方法 包括从父类继承的
  - Method getDeclaredMethod(name,class)获取当前类的某个method 不包括父类

    - Method[] getMethods()
    - Method[] getDeclaredMethods()
- Method 属性
    - getName() 返回方法名称
    - getReturnType() 返回类型，一个Class类型表示 String.class
    - getParameterTypes() Class数组表示的参数类型 {String.class,int.class}
    - getModifiers() int 表示的修饰符
- 调用method
    - 通用方法
        - Method.invoke(Object,parameter)// 对object对象的method方法调用parameter参数

```
String s="hello";
Method m=String.class.getMethod("substring",int.class)//substring  int 参数
String r = (String) m.invoke(s,6);
```

    - 静态方法

```
// 获取Integer.parseInt(String)方法，参数为String:
Method m = Integer.class.getMethod("parseInt", String.class);
// 调用该静态方法并获取结果:
Integer n = (Integer) m.invoke(null, "12345");
```

- 非public 方法

```
Person p = new Person();
 Method m = p.getClass().getDeclaredMethod("setName", String.class);
 m.setAccessible(true);
 m.invoke(p, "Bob");

    class Person {
  String name;
  private void setName(String name) {
    this.name = name;
  }
}
```

- 多态
    - invoke时，会传入实际调用的对象，所以反射支持多态
- 构造方法
    - Constructor总是当前类定义的构造方法，和父类无关，因此不存在多态的问题
    - 调用非public的Constructor时，必须首先通过setAccessible(true)设置允许访问。setAccessible(true)可能会失败。
    - Class.newInstance() 只能调用该类public无参数构造方法。有参/非public no
    - getConstructor(class...) 获取某个public Constructor

- getDeclaredConstructor(class) 获取某个constructor
- getConstructors(class...)
- getDeclaredConstructor(class)

```
Person p = new Person();
Person p1 = Person.class.newInstance();

// 获取构造方法Integer(int):
  Constructor cons1 = Integer.class.getConstructor(int.class);
  // 调用构造方法:
  Integer n1 = (Integer) cons1.newInstance(123);
  System.out.println(n1);

  // 获取构造方法Integer(String)
  Constructor cons2 = Integer.class.getConstructor(String.class);
  Integer n2 = (Integer) cons2.newInstance("456");
  System.out.println(n2);
```

获取继承关系

- 获取父类Class
  - class.getSuperclass(); 一直到object父类null

```
Class i = Integer.class;
Class n = i.class.getSuperclass();
```

- 获取interface
  - class.getInterfaces 返回当前类直接实现的接口类型，不包括父类实现的接口
  - 此外，对所有interface的Class调用getSuperclass()返回的是null，获取接口的父接口要用getInterfaces
  - 没有implements接口时返回空数组

```
Class s = Integer.class;
 Class[] is = s.getInterfaces();
 for (Class i : is) {
    System.out.println(i);
 }
 //interface java.lang.Comparable
  // interface java.lang.constant.Constable
  // interface java.lang.constant.ConstantDesc
```

动态代理

- 一般接口类型变量均 向上转型并指向某个实例，动态代理可以在运行期间动态创建interface实例，不编写实现类，直接在运行期间创建某个interface实例

- 定义InvocationHandler实例，负责接口的方法调用
- 通过Proxy.newProxyInstance()创建interface实例
  - 参数 ClassLoader 接口类
  - 需要实现的接口数组，至少需要传入一个接口进去
  - 处理接口方法调用的InvocationHandler
- 将返回的Object强制转型为接口

```
  interface Hello{
   void morning(String name);
}
  public class ReflectionTest {
  public static void main(String[] args) {
     InvocationHandler handler = new InvocationHandler() {
        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
            System.out.println(method);
            if (method.getName().equals("morning")) {
               System.out.println("good morning" + args[0]);
            }
            return null;
         }
        };
        Hello hello = (Hello) Proxy.newProxyInstance(
            Hello.class.getClassLoader(),//传入classloader
            new Class[]{ Hello.class },//传入要实现的接口
            handler);//传入处理调用方法的InvocationHandler
        hello.morning("Bob");
     }
  }
```

注解

用作标注的元数据，会被编译器打包进入class文件

使用注解

编译器使用的注解

- **@Override** 检查该方法是否实现了正确覆写
- **@SuppressWarnings** 告诉编译器忽略此处警告

工具处理.class文件使用的注解

程序运行期能够读取的注解

- 加载后一直存在于JVM中
- 可以配置参数，没有指定配置的参数默认值
- 如果注解参数名称value，且只有一个参数，可以忽略参数名称

```
public class Hello{
 @Check(min=0,max=100,value=55)
 public int n;
 @Check(value=99)
 public int p;
 @Check(99) //==@Check(value=99)
 public int x;
 @Check
 public int y;
}
```

定义注解

使用@interface定义注解，注解的参数类似无参数方法，可通过default设定一个默认值

```
public @interface Report{
 int type() default 0;
 String level() default "info";
 String value() default "";
}
```

元注解

可以修饰注解的注解为元注解，一般只需要使用元注解，不需要定义

- @Target
  - 定义注解能够应用于源码的哪些位置
    - 类/接口 ElementType.Type
    - 字段 ElementType.FIELD
    - 方法 ElementType.METHOD
    - 构造方法 ElementType.CONSTRUCTOR
    - 方法参数 ElementType.PARAMETER
    - @Target定义的value为一个ElementType[]数组
  - ```
    @Target(ElementType.METHOD)//定义注解@Report 可以用在方法上
    public @interface Report{
    }
    @Target({                    //定义注解@Report 可以用在方法上或字段上
      ElementType.METHOD,
      ElementType.FIELD
    })
    public @interface Report {
    }
    ```
- @Retention 定义注解的生命周期
    ```

    - 仅编译器 RetentionPolicy.SOURCE
    - 仅class文件RetentionPolicy.CLASS
    - 运行期 RetentionPolicy.RUNTIME
    - 如果Retention 不存在，默认为RetentionPolicy.CLASS，而我们通常定义的注解都在运行期，所以一定加上@Retention(RetentionPolicy.RUNTIME)

```
@Retention(RetentionPolicy.RUNTIME)
```

- @Repeatable 定义Annotation是否可以重复

```
@Repeatable(Report.class)
@Target(ElementType.Type)
public @interface Report{
 int type() default 0;
 String level() default "info";
 String value() default "";
}
@Target(ElementType)
public @interface Reports{
 Report [] value;
}
```

```
@Report(type = 1, level = "debug")
@Report(type = 2, level = "warning") //经过@Repeatable修饰，可以添加多个@Report注解
public class Hello{}
```

- @Inherited 定义子类是否可以继承父类定义的Annotation, 仅针对@Target(ElementType.Type)类型的Annotation有效，且仅针对class的继承，interface继承无效

```
@Inherited
@Target(ElementType.Type)
public @interface Report{
 int type() defualt 0;
}

@Report(type=1)
public class Person{
}
public class Student extends Person{
}
```

总结

- @interface 定义注解

- 添加默认值
- 用元注解配置注解

```
@Target(ElementType.Type)
@Retention(RetentionPolicy.RUNTIME)
public @interface Report{
  int type() default 0;
}
```

- 必须设置@Target 和 @Retention(一般runtime)

处理注解

- SOURCE注解在编译器丢掉，由编译器使用，我们仅使用
- CLASS注解保存在class文件中，不会被加载进JVM，底层工具库使用，涉及class加载，很少用
- RUNTIME加载进JVM，可以在运行期间读取，经常使用+编写
  注解也是class，继承java.lang.annotation.Annotation，读取注解时，使用反射API

提供的使用反射API读取Annotation的方法

判断某个注解是否存在于Class,Field,Method,Constructor中

- Class.isAnnotationPresent(Class) Person.class.isAnnotationPresent(Report.class)判断@Report是否存在Person类中
- Field.isAnnotationPresent(Class)
- Method.isAnnotationPresent(Class)
- Constructor.isAnnotationPresent(Class)
  反射API读取Annotation
- Class.getAnnotation(Class)
- Field.getAnnotation(Class)
- Method.getAnnotation(Class)
- Constructor.getAnnotation(Class)

```
//两种方式读取annotation
Class cls = Person.class;
if(cls.isAnnotationPresent(Report.class)){ //判断是否存在再获取
  Report report = cls.getAnnotation(Report.class);
}
Report report = cls.getAnnotation(Report.class);//获取，再判断是否是null
if(report != null){
}
```

读取方法、字段、构造方法annotation和class类似。唯独方法参数的annotation，方法参数本身看做一个数组，同时每个参数可以定义多个注解。一次获取方法参数的所有注解用二维数组

```
public void hello(@NotNull @Range(max=5) String name, @NotNull String prefix){
 //先获取method实例
 Method m =...
 Annotation[][] annos = m.getParameterAnnotations();
 //第一个参数所有的annotation
 Annotation[] ann =annos[0];
 for(Annotation anno : ann){
  if(anno instanceof Range){
   Range r = (Range)anno;
  }
 }
}
```

注解使用

由程序决定，注解本身对逻辑没有影响

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Range {
   int min() default 0;
   int max() default 255;
}
public class Person {
   @Range(min=1, max=20)
   public String name;

   @Range(max=10)
   public String city;
}
//自己定义方法去检测参数
void check(Person person) throws IllegalArgumentException, ReflectiveOperationException {
   // 遍历所有Field:
   for (Field field : person.getClass().getFields()) {
      // 获取Field定义的@Range:
      Range range = field.getAnnotation(Range.class);
      // 如果@Range存在:
      if (range != null) {
         // 获取Field的值:
         Object value = field.get(person);
         // 如果值是String:
         if (value instanceof String) {
            String s = (String) value;
            // 判断值是否满足@Range的min/max:
            if (s.length() < range.min() || s.length() > range.max()) {
               throw new IllegalArgumentException("Invalid field: " + field.getName());
            }
         }
      }
```

```
    }
}
```

泛型

ArrayList 可以向上转型为 List
ArrayList 不可以向上转型为 List

泛型使用

```
ArrayList<String> array = new ArrayList<>();

public interface Comparable<T>{
 int compareTo(T o);
}
class Person implements Comparable<Person>{
 public int compareTo(Person other){
  return this.name.compareTo(other.name);
 }
}
```

泛型编写

泛型类型<T>不能用于静态方法，必须定义其他类型<K>将静态方法的泛型类型和实例类型的泛型类型区分开

```
public class Pair<T>{
 public T getFirst(){}
 //静态泛型方法应该使用其他类型区分
 public static <K> Pair<K> create(K first, K second){
 }
}
```

多种类型的泛型

```
public class Pair<K, V>{

}
```

擦拭法

虚拟机对泛型一无所知，编译器做工作。编译器把<T>视为Object，并根据<T>实现安全的强制转型。java的泛型是由编译器在编译时运行的，编译器内部将所有类型T视为Object，同时在需要转型时，根据T类型做强制类型转换。

```

```

```
//编辑的代码
public class Pair<T>{
  private T first;
  private T second;
}
//jvm虚拟机运行的代码 将所有泛型替换为Object
public class Pair{
  private Object first;
  private Object second;
}
```

局限性

- <T>不能是基本类型，因为实际类型为Object，Object无法持有基本类型
- 无法取得泛型的Class

```
Pair<Sintr> p1 = new Pair<>("1", "2");
Pair<Integer> p2 = new Integer<>(1, 2);
Class c1 = p1.getClass();
Class c2 = p2.getClass();
assert(c1 == c2); //true
assert(c1 == Pair.Class);//true
```

- 无法判断带泛型的类型

```
Pair<Integer> p = new Pair<>(1, 2);
if(p instanceof Pair<String>){
  //并不存在Pair<Sintr>.class 只有Pair.class
}
```

- 不能实例化T

```
public class Pair<T>{
  private T first;
  private T second;
  public Pair(){
    fist = new T(); // !!!! error
    second = new T();
  }
}
public class Pair<T>{
  private T first;
  private T second;
  public Pair(Class<T> clazz){//实例化T类型时，借助Class<T>参数
    first = clazz.newInstance();
    second = clazz.newInstance();
```

```
    }
  }
```

不恰当的覆写

定义的equals(T t)被擦拭为equals(Object t)，这个方法继承自object，编译器阻止一个实际上会被转为覆写的泛型方法，可重命名实现

```
public class Pair<T>{
  public boolean equals (T t){
    return this == t;
  }
}
```

泛型继承

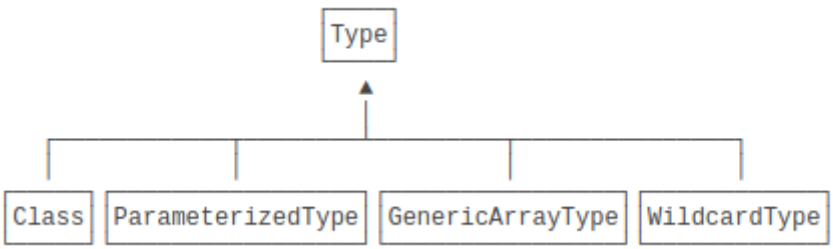
无法获取Pair的类型T

```
public class IntPair extends Pair<Integer>{

}
IntPair p =new IntPair(1, 2);//继承了泛型的子类，可以直接使用没有泛型参数
```

在继承了泛型类型时，子类可以获取父类的泛型类型，否则编译器就无法得知子类要存储的类型T

```
Class<IntPair> clazz = IntPair.class;
Type t = clazz.getGenericSuperclass();
if(t instanceof ParameterizedType){
  ParameterizedType pt = (ParameterizedType)t;
  Type[] types = pt.getActualTypeArguments();//可能有多个泛型类型
  Type firstType = types[0];
  Class<?> typeClass = (Class<?>)firstType;
  System.out.println(typeClass);//class java.lang.Integer
}
```



java实际的类型体系

extends 通配符

Pair 不是Pair的子类，无法强制类型转换

```
public class PairHelper{
 static int add(Pair<Number> p){
   Number first = p.getFirst();
   Number second = p.getSecond();
  return first.intValue() + second.intValue();
 }
}

int sum =PairHelper.add(new Pair<Number>(1, 2));// OK
int sum2 = PairHelper.add(new Pair<Integer>)(1, 2);//fail error: incompatible types: Pair<Integer>
cannot be converted to Pair<Number>
```

<? extends Number> 上界通配符

```
public class PairHelper{
 static int add(Pair< ? extends Number> p){
   Number first = p.getFirst();
   Number second = p.getSecond();
  return first.intValue() + second.intValue();
 }
}
//可以传参数 Pair<Integer> Pair<Double> 等 T为Number的子类
```

extends 用于get方法

对Pair<? extends Number>调用getfirst方法，方法实际签名为<? extends Number> getFirst()，返回值为Number的子类

```
Numbe x = p.getFirst();
Integer y = p .getFirst();//error? 无法确定具体类型，只能确保是Number的子类型
```

extends 用于set方法

```
static int add(Pair<? extends Number> p) {
      Number first = p.getFirst();
      Number last = p.getLast();
      p.setFirst(new Integer(first.intValue() + 100));
      p.setLast(new Integer(last.intValue() + 100));
      return p.getFirst().intValue() + p.getFirst().intValue();
   }

   Number first = p.getFirst();
      Number last = p.getLast();
```

```
    p.setFirst(new Integer(first.intValue() + 100)); // incompatible types: Integer cannot be converted to CAP#1
    //擦拭法，当p为Pair<Double>时，显然Pair<Double>的 setfirst不能接受Integer类型
```

方法参数签名setFirst(? extends Number)无法传递任何Number的子类给setFirst(? extends Number),除了可以setFirst(null)

extends 使用

```
int sumOfList(List<? extends Integer> list){

}
```

List<? extends Integer>表明该方法内部只会读取List元素，不会修改List元素，是一个对参数List<? extends Integer>只读的方法

extends限定T类型

定义泛型类型时，通过extends限定T类型

```
public class Pair<T extends Number>{}
```

总结

- extends 作为方法参数时，可读不可写
- extends作为泛型类时，限定泛型类型为T及其T的子类

super通配符

<? super Integer>表示

- 允许调用set(? super Integer)方法传入Integer的引用
- 不允许调用get方法获得Integer的引用
- 可写不可读

extends VS super

- <? extends T>允许调用读方法T get()获取T的引用，但不允许调用写方法set(T)传入T的引用（传入null除外）
- <? super T>允许调用方法set(T)传入T的引用，不允许调用T get()获取T的引用

```
public class Collections{
  public static <T> void copy(List<? super T> dest, List<? extends T> src){
    //将src中元素复制到dest中，编译器检查 是否满足对super的只写，extends的只读
```

```
  }
}
```

PECS原则

Producer Extends Consumer Super

无限定通配符

```
void sample(Pair<?> p){

}
static boolean isNull(Pair<?> p){
  return p.getFirst() ==null;
}
```

没有extends 同时也没有super

- 不允许调用set(T)并传入引用(null除外)
- 不允许调用T get()并获取T引用(只能获取Object引用)
- Pair<?> 为所有Pair的超类，可以用Pair替换，只能做null判断

泛型和反射

部分反射的API也是泛型

- Class<T>

  ```
  Class<String> clazz = String.class;
  ```

- Constructor<T>

  ```
  Class<Integer> clazz = Integer.class;
  Constructor<Integer> cons = clazz.getConstructor(int.class);
  Integer i = cons.newInstance(0);
  ```

泛型数组

- 可以声明带泛型的数组，不能new创建

  ```
  Pair<String>[] ps = null;//ok
  Pair<String>[] ps = new Pair<String>[2];//fail
  ```

- 通过强制类型转换实现带泛型的数组

```
Pair<String>[] ps = (Pair<String>[])new Pair[2];
Pair<String>[] ps2 = (Pair<String>[])Array.newInstance(String.class, 2);
```

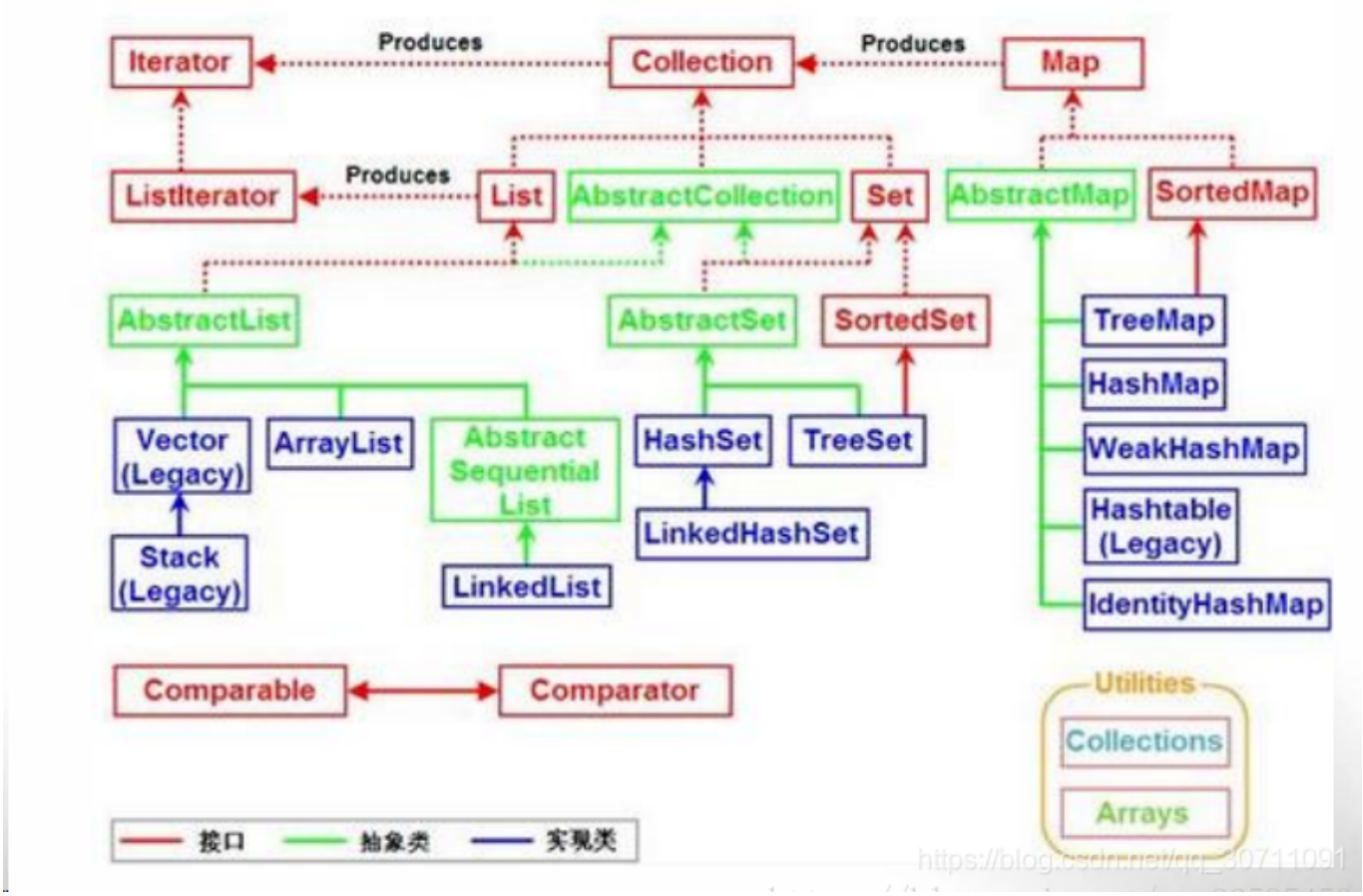
- 泛型数组的使用

```
Pair[] arr = new Pair[2];
Pair<String>[] ps = (Pair<String>[])arr;
arr[0] = new Pair<Integer>(1, 2);
Pair<String> p = ps[0];// cast error
//安全使用时，扔掉arr的引用
Pair<String>[] ps = (Pair<String>[])new Pair[2];
ps.getClass() == Pair[].class//true
```

```
T[] createArray(Class<T> cls){
 return (T[])Array.newInstance(cls, 5);
}
//通过可变参数创建泛型数组
public class ArrayHelper{
static <T> T[] asArray(T.. objs){
 return obj;
}
}
```

如果在方法内创建了泛型数组，最好不要将它返回给外部使用

集合



集合接口和实现分离，支持泛型，分为Collection和Map，细分为

- List
- Set
- Map 遗留的集合类，不应继续使用
- Hashtable 线程安全的Map实现
- Vector 线程安全的List实现
- Stack 基于Vector实现的LIFO栈
- Enumeration 被Iterator取代

List

-----	ArrayList	LinkedList
获取指定元素	快	从头开始查找
添加元素到末尾	快	快
指定位置添加、删除	需要移动元素	不需要移动
内存占有	少	多

创建

- List<Integer> list = new ArrayList<>()
- List<Integer> list = new LinkedList<>()
- List<Integer> list = List.of(1,2,3)//不能null

遍历

- Iterator

```
Iterator<String> it = list.iterator();
while(it.hasNext(){
  System.out.println(it.Next());
}
```

- for each

```
for(String str : list){
  System.out.println(str);
}
```

同Array互转

- Object[] array = list.toArray()//返回Object[]
- Integer[] array = list.toArray(new Integer[3])//

```
List<Integer> list = List.of(1, 2);
Number[] array = list.toArray(new Number[3]);//可以向上转型！！！
```

- List<Integer> list = List.of(array);//!! List.of 返回只读类型的list
- List list = Array.asList(array);

equals方法

要正确使用List的contains() indexof()方法，放入的实例必须正确覆写equals()方法，否则可以不用

- 自反性，对于非null的x来说，x.equals(x)为true
- 对称性，对于非null的x，y来说，x.equals(y)为true，y.equals(x)也为true
- 传递性，对于非null的x,y,z，x.equals(y)为true，y.equals(x)为true，那么x.equals(x)也为true
- 一致性，对于非null的x,y来说，只要x，y状态不变，x.equals(y)总是一致的返回true或false
- 对非null的比较，x.equals(null)永远返回false
  对于引用字段比较用equals，对于基本类型字段用==
  可调用Object.equals(a,b)静态方法，省去对null的判断

1. 先确定实例相等逻辑
2. 用instanceof判断传入的待比较的Object是不是当前类型，如果是，则继续，否则false
3. 对引用类型用Object.equals，对基本类型用==

Map

Map中不存在重复的Key，相同的Key会把原有的Key-Value替换

遍历Map

遍历时，不能假设输出的key为有序的

- 遍历Key map.keySet()返回不重复的key集合

```
for(String key : map.keySet()){

}
```

- 遍历key和value

```
for(Map.Entry<String, String> entry : map.entrySet()){
  String key = entry.getKey();
  String value = entry.getValue();
}
```

equals 和hashcode

正确使用Map

- 作为key对象覆写equals()方法，相等的两个key实例调用equals()返回true
  - 对引用对象调用Object.equals，对基本类型调用==
- 作为key对象覆写hashcode()方法
  - 如果两个对象相等，则hashcode()必须相等
  - 如果两个对象不等，则hashcode()尽量不相等

```
@Override
int hashcode(){
  int h =0;
  h = 31*h + firstname.hashCode();
 h = 31*h + secondname.hashCode();
 h = 31*h +age;
 return h;
}
```

同样为了NPE问题，可以调用Object.hash方法，如果有数组作为参数，Arrays.hashCode(array)包起来作为参数传给Object.hash。----https://stackoverflow.com/questions/29955291/why-objects-hash-returns-different-values-for-the-same-input

```
int hashCode(){
  return Object.hash(firstname, secondname, age);
}
```

原则：
equals()用到的用于比较的每一个字段，都在hashCode()中用于计算，equals()中没有用到的字段，一定不要在hashCode()中计算。 hash内部使用了数组，通过计算key的hashCode()定位value的索引。

- hashmap 初始化时默认数组大小16，任何key，无论hashCode()多大，都通过int index = key.hashCode() & 0x0f,将索引定位在0-15
- 如果添加元素超过16，HashMap内部自动扩充，每次扩充一倍，后需要重新计算hashCode对应的索引 int index = key.hashCode() & 0X1f
- 频繁扩充影响性能，最好创建时指定大小,HashMap内部数组大小为 2^n
    - Map<String, Integer> map = new HashMap<>(10000);实际大小为16384
- hash冲突时，每个索引处存放一个List包含多个Entry，
    - 内部通过key实际找到的为List<Entry<String , Person>>,之后再遍历这个List

EnumMap

当key为Enum类型时，EnumMap内部以非常紧凑的数组存储value，根据enum类型的key直接定位到内部数组的索引，不需要计算hashCode(),效率高，空间省

```
Map<DayOfWork, String> map = new EnumMap<>(DayOfWork.class);
map.put(DayOfWork.MONDAY, "周一");
map.get(DayOfWork.MONDAY);
```

TreeMap

HashMap内部key无序，遍历Key时，顺序不可预测。对key会排序的接口Map为SortedMap，实现类为TreeMap。SortedMap保证以key的顺序，放入的key必须实现Comparable接口。或者在创建时定义个排序算法，传入Comparator类

```
Map<Person, Integer> map = new TreeMap<>(new Comparator<Person>(){
 public int compare(Person p1, Person p2){
 }
});
```

comparable VS comparator

```
 package java.lang;
public interface Comparable<T> {
   public int compareTo(T o);
}
```

```
package java.util;
public interface Comparator<T> {
  int compare(T o1, T o2);
```

```
  boolean equals(Object obj);
 //..... many interface
}
```

Comparable对实现它的每个类的对象进行整体排序，这个接口需要类本身实现，需要在设计类之处就实现该接口类。

- 实现Comparable接口的类的List/数组，通过Collections.sort或者Arrays.sort进行排序
- 也可作为有序映射TreeMap或有序集合TreeSet中的元素，而不需要指定比较器

```
public class Person1 implements Comparable<Person1>{
 @Override
 public int compareTo(Person1 o){
  return this.age - o.age;
 }
}
    Person1 person1 = new Person1("zzh",18);
    Person1 person2 = new Person1("jj",17);
    Person1 person3 = new Person1("qq",19);

    List<Person1> list = new ArrayList<>();
    list.add(person1);
    list.add(person2);
    list.add(person3);

    System.out.println(list);
    Collections.sort(list);
```

当类不可修改，有需要对其进行排序时，需要用到comparator

```
pulibc final class Person2{

}
```

final修饰，无法再implements Comparable,在类的外部使用Comparator接口

```
    Person2 p1 = new Person2("zzh",18);
    Person2 p2 = new Person2("jj",17);
    Person2 p3 = new Person2("qq",19);
    List<Person2> list2 = new ArrayList<Person2>();
    list2.add(p1);
    list2.add(p2);
    list2.add(p3);
    System.out.println(list2);
    Collections.sort(list2,new Comparator<Person2>(){

      @Override
```

```
        public int compare(Person2 o1, Person2 o2)
        {
         if(o1 == null || o2 == null)
          return 0;
           return o1.getAge()-o2.getAge();
        }
      });
      System.out.println(list2);
```

总结
Comparable 是排序接口；若一个类实现了 Comparable 接口，就意味着 “该类支持排序”。而 Comparator 是比较器；我们若需要控制某个类的次序，可以建立一个 “该类的比较器” 来进行排序

使用Properties

配置文件Key-value一般都是String-String的，默认以properties扩展名

读

```
# setting.properties
last_open_file=/data/hello.txt
auto_save_interval=60
```

```
String f = "setting.properties"
Properties props = new Properties();
props.load(new java.io.FileInputStream(f));
//propes.load(getClass().getResourceAsStream("/common/setting.properties")); 多次load会覆盖之前的
String filepath = props.getProperty("last_ioen_file")//不存在 返回Null
props.getProperty("auto_save", "60"));//默认值
```

写

```
Properties props = new Properties();
props.setProperty("url","www.baidu.com");
props.setProperty("l","1");
props.store(new FileOutputStream("C:/conf/setting.properties"),"this is writed annotation");
```

编码

早起规定.properties编码ascii，中文时用name=\u4e2d\u6587表示，JDK9后，.properties可以使用UTF-8编码。load(InputStream)总是以ASCII编码读取字节流，乱码。load(Reader)使用字符流,已经在内存中以char表示，不涉及编码问题

```
props.load(new FileReader("settings.properties", StandardCharsets.UTF_8));
```

Set

存储不重复的元素集合

- boolean add(E e)
- boolean remove(Object e)
- boolean contains(Object e) 常用HashSet为HashMap的简单封装，理解为只存储key的HashMap

```
public class HashSet<E> implements Set<E>{
  private HashMap<E, Object> map = new HashMap<>();
  private static final Object PRESENT = new Object();
  public boolean add(E e){
    return map.put(e, PRESENT) == null;
  }
  public boolean contains(Object o){
    return map.contains(o);
  }
  public boolean remove(Object o){
    return map.remove(o) == PRESENT;
  }
}
```

类似SortedMap TreeMap，存在SortedSet TreeSet，key的排序顺序，TreeSet需要key的类实现Comparable接口，或者创建TreeSet时传入Comparator对象。

Queue

避免将null传入Queue，实现类有AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, SynchronousQueue

desc	throws Exception	false或null
队列长度		int size()
添加到队尾	boolean add(E e)	boolean offer(E e)
取队首元素并删除	E remove()	E poll()
取队首元素不删除	E element()	E peek()

```
Queue<String> q = new LinkedList<>();
List<String> list = new LinkedList<>();//LinkedList既实现了List,也实现了Queue
```

PripriryQueue

即有顺序的Queue,要求存放的元素implements Comparable接口，或者在创建PriorityQueue时传入Comparator对象。
PriorityQueue和Queue的区别在于，它的出队顺序与元素的优先级有关，对PriorityQueue调用remove()或poll()方法，返回的总是优先级最高的元素

```
Queue<User> q =new PriorityQueue<>(new UserComparator());

class UserComparator implements Comparator<User>{
 public int compare(User u1, User u2){

 }
}
```

Deque

双端队列,实现类有ArrayDeque和LinkedList,避免把null添加到队列

desc	Queue	Deque
添加到队尾	add(E e)/offer(E e)	addLast(E e)/offerLast(E e)
取队首不删除	E element()/E peek()	E getFirst()/E peekFirst()
取队首并删除	E remove()/E poll()	E removeFirst()/E pollFirst()
添加到队首	None	addFirst(E e)/offerFirst(E e)
取队尾不删除	None	E getLast()/E peekLast()
取队尾并删除	None	E getFirst()/E peekFirst()

Stack

Deque模拟的Stack

- 压栈 push(E)
- 取栈顶弹出 pop(E)
- 取栈顶不弹出 peek(E)

Iterator

for each 通过Iterator改写成了普通的for循环。 自定义集合类的for each循环，条件

- 实现Iterable接口，按要求返回一个Iterator对象
- 用Iterator对象迭代集合内部数据

在编写Iterator的时候，我们通常可以用一个内部类来实现Iterator接口，这个内部类可以直接访问对应的外部类的所有字段和方法。例如，上述代码中，内部类ReverseIterator可以用ReverseList.this获得当前外部类的this引用，然后，通过这个this引用就可以访问ReverseList的所有字段和方法。

```
ReverseList<String> rlist = new ReverseList<>();
rlist.add("apple");

class ReverseList<T> implements Iterable<T>{
 private List<T> list = new ArrayList<>();
 public void add(T t){
  list.add(T);
 }
 @Override
 public Iterator<T> iterator(){
  return new ReverseIterator(list.size());
 }
 class ReverseIterator implements Iterator<T>{
  int index;
  ReverseIterator(int index){
   this.index = index;
  }
  @Override
  public boolean hasNext(){
   return index > 0;
  }
  @Override
  public T next(){
   index --;
   return ReverseList.this.list.get(index);
  }
 }
}
```

Iterator进行迭代好处

- 对任何集合统一
- 调用者无需对集合内部结构了解
- 集合类返回的Iterator对象知道如何迭代 ??

Collections

位于java.util包中，提供了静态方法，便于对集合操作

创建空集合

返回的空集合为不可变对象，无法添加元素

- List emptyList();
- Map<K, V> emptyMap()
- Set emptySet() 也可利用集合提供的of接口

```
List<String> list1 = List.of();
List<String> list2 = Collections.emptyList();
```

单元素集合

也是不可变

- List singletonList(T o)
- Map<K, V>singletonMap(K key, V value);
- Set singleton(T o) 同样也可以用List.of(T o)

排序

Collections 对List排序，会改动List元素位置，传入可变的List

```
Collections.sort(list);
```

洗牌

Collections.shuffle(list);

不可变集合

提供了一组方法把可变集合封装成不可变集合,继续对原始的可变List进行增删是可以的，并且，会直接影响到封装后的“不可变”List,如果我们希望把一个可变List封装成不可变List，那么，返回不可变List后，最好立刻扔掉可变List的引用，这样可以保证后续操作不会意外改变原始对象，从而造成“不可变”List变化

- 封装成不可变List：List unmodifiableList(List<? extends T> list)
- 封装成不可变Set：Set unmodifiableSet(Set<? extends T> set)
- 封装成不可变Map：Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> m)

```
List<String> mutable = new ArrayList<>();
 mutable.add("apple");
 mutable.add("pear");
 // 变为不可变集合:
 List<String> immutable = Collections.unmodifiableList(mutable);
 // 立刻扔掉mutable的引用:
 mutable = null;
```

线程安全集合

从Java 5开始，引入了更高效的并发集合类,所以上述这几个同步方法已经没有什么用了。

- 变为线程安全的List：List synchronizedList(List list)
- 变为线程安全的Set：Set synchronizedSet(Set s)
- 变为线程安全的Map：Map<K,V> synchronizedMap(Map<K,V> m)

重写（Override） VS 重载（Overload）

- Override
  - 重写方法不能抛出新的异常或者比被重写方法方法更加宽泛的异常
  - 声明为static的方法不能重写，但是可以再次声明
  - final方法不能重写
- Overload
  - 被重载的方法可以声明新的异常或更广的检查异常
  - 被重载的方法可以改变访问修饰符
  - 被重载的方法可以改变返回类型