JAVA程序设计

1. Java简介

Java——目前最流行的编程语言之一，用于桌面应用、Web、嵌入式和移动端开发

Java SE——桌面和服务器环境开发

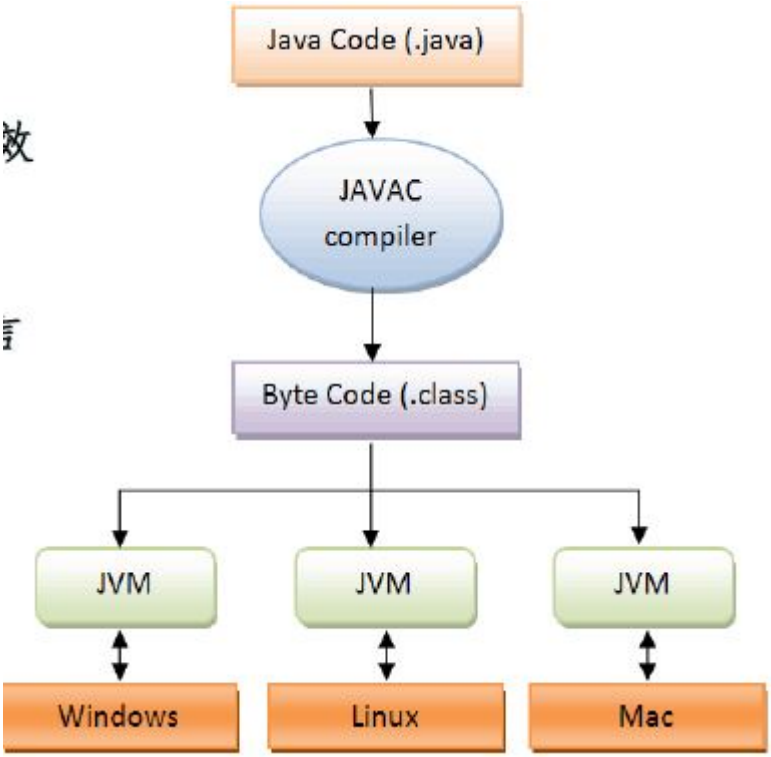
Java EE——网络和Web服务开发

Java ME——嵌入式和移动端开发

Java起源——James Gosling在SUN公司，1991开发，起初命名为“Oak”，之后更名为“Java”

运行机制——上层编译（JAVAC编译器），下层解释（JVM）

源代码(.java)→JAVAC编译器→字节码(.class)→JVM解释器→在任意系统上运行



JVM——将byte code转换为机器语言并执行

$$bytecode \rightarrow Class\ Loader \rightarrow Byte\ Code\ Verifier \rightarrow Interpreter\ and\ JIT$$

Class Loader——载入并准备

Interpreter——解释器

JIT——just-in-time Compiler，增加效率，将byte code编译为机器码

Garbage collector——自动释放对象，自动管理内存

JRE——包含运行一个Java程序的所有东西的软件包。JVM + Java class library

Java的特点——简单的面向对象编程；鲁棒性和安全性；平台无关，可移植；表现好，性能高；多线程，动态链接

JDK (Java Development Kit)——javac.exe编译器，java.exe解释器，javadoc.exe文档生成器，jdb.exe调试器

程序编辑 + 程序编译(javac) + 程序运行(java)

类（Class）——拥有共同属性和行为的对象的集合，具有层次关系

对象（Object）——现实世界中的某个具体的事物

封装的实体 = 数据 + 方法（行为）

每个对象由对象标识符唯一标识

方法（Methods）——对象的行为方式，对象与外界的接口；可以改变对象的属性，或者返回对象的属性

面向对象的抽象原理（数据抽象/封装）——提供了一种对数据和操作这些数据的算法的抽象

模块化——将复杂系统分解为若干个尽量正交独立的模块

信息隐蔽——将模块的细节部分对用户隐藏，用户只能通过受保护的接口进行访问

继承性——父类和子类间共享数据的方法，继承具有传递性

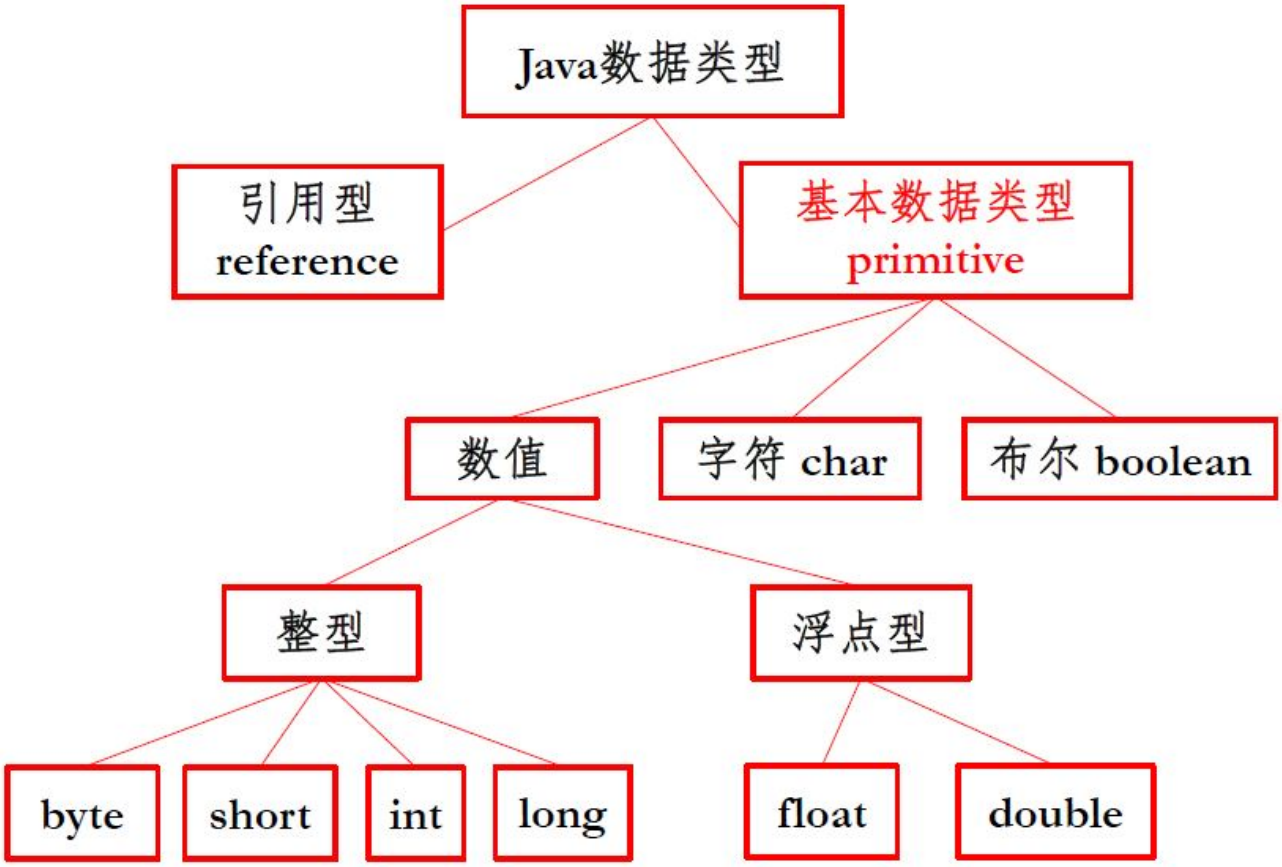
继承使得软件系统具有开放性，可以更好地进行抽象，代码重用，方便代码维护

多态性——一个实体同时具有多种形式；同一操作作用于不同的对象，由不同的解释产生不同的执行结果

程序 = 对象 + 消息

面向对象编程 = 对象 + 类 + 继承 + 通信

2. Java编程的基本组件



Java的基本数据类型具有固定的字段长度，因而有固定的表数范围

逻辑型——存储占一个byte，取值为true或false

整型——byte，short，int，long。负整数补码表示

类型	占用存储空间	表数范围
byte	1字节	-128~127
short	2字节	-2^{15}~$2^{15}-1$
int	4字节	-2^{31}~$2^{31}-1$
long	8字节	-2^{63}~$2^{63}-1$

十进制表示，正常写数字即可；八进制，用0开头；二进制，用0b开头；十六进制，用0x开头

Java默认整型常量为int，声明long型常量时可以在数字后加L

浮点型——float，double

类型	占用存储空间	表数范围
float	4字节	-3.403E38~3.403E38
double	8字节	-1.798E308~1.798E308

十进制数形式，必须包含小数点；科学计数法形式，xxxxExxx

Java默认浮点型常量为double，声明float型常量时在数字后面加f

float型——1-bit sign, 8-bit exp, 23-bit frac

double型——1-bit sign, 11-bit exp, 52-bit frac

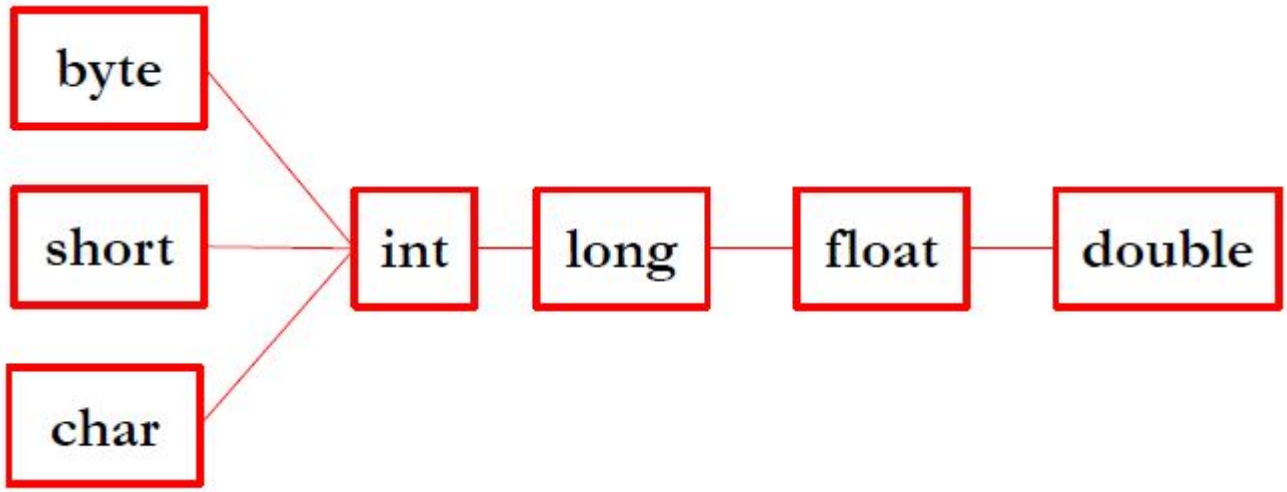
字符型——存储占2字节，char型

标识符——变量的名字。字母，数字和下划线组成；必须以字母或下划线开头；不能以数字开头

短路逻辑运算符——&&，如果第一个操作数为假则不再判断第二个；||，如果第一个操作数为真则不再判断第二个

右移—— >>带符号右移，最高位补符号位；>>>无符号右移，最高位补0

Java表达式中的数据类型转换方法



break语句——单独写"break;"可以用于终止语句块的执行；写"break Lx;"可以终止某一层语句块，在多层嵌套的时候使用

continue语句——与break类似

3. 面向对象编程

类——具有相同属性和行为的对象的抽象。成员包括字段（field）和方法（method）

字段——类的属性，又称为域

方法——类的行为，与属性有关的功能和操作

实例（instance）——类的具体的对象

构造函数——特殊的方法，用于初始化。

未定义构造函数，则系统自动添加一个无参数的构造函数，并默认该函数方法体空

类字段的默认初始化为0（int型）或者null（String型）

静态方法&静态字段——static关键字声明，所有类的实例所共享。可以通过类名访问，也可以通过实例访问，两种方法是等价的

实例方法可以访问和操作静态字段，但静态方法不能访问和操作实例字段

继承——extends关键字来实现。字段和方法的继承和添加

方法的覆盖（overriding）——子类定义与父类成员名字相同的方法，是对父类方法的重定义

构造函数——构造函数不能继承；如果子类没有显示调用父类的构造函数，则子类会在构造函数开始时自动执行父类中不带参数的构造函数；如果父类中没有定义不带参的构造函数，编译器报错

this & super——使用this调用本类的其他构造方法；使用super调用直接父类的构造方法；this和super必须在构造函数的第一条语句，并且最多只能有一条，不能同时出现this和super；如果没有调用super，编译器会自动添加上super()

abstract类——用abstract修饰的类，不能被实例化；abstract修饰的方法为抽象方法，作用为定义一个统一的接口，格式如下；抽象类中可以包含抽象方法，也可以不包含，但包含抽象方法的类必须是抽象类

```
abstract returnType abstractMethod([paramList])
```

final类——用final修饰的类，不能被继承的类，不会有任何子类；用final修饰的方法，不能被子类的方法覆盖；final字段，只能赋值一次，static final可以不设定初值，按照默认值初始化，不是静态的final字段必须且只能赋值一次（初值，或构造）

super——使用super.xxx来访问父类的字段和方法；用super调用父类的构造函数

多态——名字相同但实现不同的方法；与覆盖不同。通过参数的个数和类型来区分，返回的类型不能用于区分重写的方法。

上溯类型——子类对象可以被视为是父类的对象，但父类对象不能被视为是子类的对象

动态绑定——在运行时，会判断对象的类型并调用适当的方法

instanceof——用于判断某个对象是否是某个类的实例

访问控制符——用于保护隐私字段，规范外部代码对于类的使用

	同一个类中	同一个包中	不同包中的子类	不同包中的非子类
private	yes			
默认	yes	yes		
protected	yes	yes	yes	
public	yes	yes	yes	yes

4. 接口

接口（interface）——用来定义多个类可能具有的相似行为的一种抽象类型

接口可以被表征为不同类之间的相似行为，但不一定能够与这些类有直接的关系。可以被视作是一种规范。

接口不能被实例化，没有构造函数，只包含抽象或静态方法，只包含常量（只有static final字段）；允许多重继承，而类不可以

接口中的字段默认为static final型；接口中的方法默认为public abstract

当一个类实现一个接口时，用implements关键字，相当于“签署了这个接口规定的规范”；如果类中没有全部实现接口声明的方法，则该类必须声明为抽象类

对接口的引用——接口可以作为一种引用类型类申明变量，该变量可以指向实现了该接口的类的实例。

```
// in file 'Printable.java'
interface Printable {
    void print();
}
```

```
// in file 'Hello.java'
public class Hello implements Printable{
    void print() {
        System.out.println("Hello");
    }
}
```

```
// in file 'Byebye.java'
public class Byebye implements Printable{
    void print() {
        System.out.println("Byebye");
    }
}
```

```
// in file 'Main.java'
public class Main {
    public static void main(String args[]) {
        // declare an interface type variable
        Printable p;
        // point to the instance of Hello
        p = new Hello();
        // call the method defined by interface,
    system will print 'Hello'
        p.print();
        // now point the instance of Byebye
        p = new Byebye();
        // system will print 'Byebye'
        p.print();
    }
}
```

5. 垃圾回收

Java通过new来创建对象，垃圾回收的动作由JVM自动完成。JVM会追踪对象的引用次数，来判断是否回收，当引用为0时说明可以进行回收。

垃圾回收的时间点：应用程序空闲时，垃圾回收线程被调用；内存不足时强制调用垃圾回收线程。

System.gc()——建议JVM进行垃圾回收，但不保证一定会回收，具体的时间点仍然是由JVM来决定的。频繁调用System.gc()会降低程序运行的效率

减少垃圾回收开销的方法——减少临时对象的使用；避免在短时间内大量创建新对象；尽量使用基本类型（如尽量使用int而不是Integer）；尽量少使用静态对象变量，他们是全局变量，不会被回收

finalize()——在回收对象时清理非内存的资源，比如关闭打开的文件

可以通过覆盖finalize()方法来实现，JVM在回收对象时自动调用。

一般来说，子类的finalize方法应该调用父类的finalize方法，确保父类的清理工作正常进行

最好自己编写清除方法，而不依赖finalize()

try-finally语句块——清除方法，不会受到异常、return、continue、break的影响；当try段结束之后，一定会进入到finally段，是一种推荐的方法

```
try{
  // code and exception handling
} finally{
  // cleanup code
}
```

6. 内部类、局部类和匿名类

内部类（inner class）——定义在其他类内部的类

```
public class A{
  ...
  class B{...}
  ...
}
```

javac生成名为A$B.class的文件

内部类不能与外部类同名

内部类的使用

内部类在封装它的类内部使用时，与普通类的使用方法相同

在其他地方使用时，类名前需要冠以外部类的名字，new内部类前需要在new前冠以外部类对象名（ 外部类对象名.new 内部类名(构造函数参数) ）

内部类中使用外部类成员

内部类可以直接访问外部类的字段和方法，不受访问控制符的限制，即使是private

若内部类中具有和外部类同名的字段和方法，可以使用"外部类名.this.字段或方法”来进行访问或调用

内部类修饰符——可以使用外部类不能使用的protected，private，static等进行修饰。

嵌套类——static修饰的内部类，实际上是一种外部类

实例化static内部类时不需要在new前面加对象实例变量名；static内部类只能访问外部类的static成员，而不能访问非static字段和方法；外部类的static方法不能不带前缀地new一个非static的内部类

局部类（local class）——定义在方法中的类，不能使用public / private / protected / static修饰，但可以被final和abstract修饰。

局部类可以访问外部类的成员，可以访问该方法的局部变量。

匿名类（anonymous class）——特殊的内部类，没有类名，定义时直接生成实例，一次性使用。编译器将生成xxxx$1.class。常在实例化接口的实现类时使用

可以访问外部类的字段，可以访问外部类的本地变量，不能定义构造函数，不能定义接口。

7. 异常处理

异常——特殊的运行错误对象

一个方法的运行过程中，如果发生了异常，则这个方法生成代表该异常的一个对象，并把它交给运行时系统，运行时系统寻找相应的代码来进行处理。

throw——生成异常对象并把它提交给运行时系统的过程，抛出异常

catch——运行时系统在方法的调用栈中查找，从生成异常的方法开始回溯，知道找到包含相应异常处理的方法为止，这一过程为捕获一个异常。如果运行时系统找不到捕获异常的方法，则Java程序退出。

java.lang.Throwable——所有Java的异常类的父类，由Throwable派生出两个子类：Error和Exception

Error——由系统保留的错误

Exception——供应用程序使用的，一般性问题

Exception类

构造方法—— public Exception();

public Exception(String message);

public Exception(String message, Throwable cause);

方法—— getMessage();　　getCause();　printStackTrace();

Java中的异常处理方法——可以跟随多个catch；finally至多一个；至少包括一个catch或finally；先处理异常，再执行finally；子类异常排在父类异常前面

```
try{
  语句块
} catch(异常类变量名){
  异常处理
} catch(异常类变量名){
  异常处理
} finally{
}
```

受检的异常（checked Exception）——受检的异常必须处理，在声明方法时在方法名后面用throws xxx。

RuntimeException及其子类可以不明确处理（比如算数异常，数组越界）

在子类中如果覆盖了父类的方法，并且该方法抛出了异常，那么子类也要抛出异常；可以抛出更具体的子类异常或相同的异常，但不能抛出父类异常

重抛异常——将异常传递给调用者，以便调用者能进行处理。此时可以在catch语句块或finally语句块采用重抛。

将当前捕获的异常再次抛出。throw e;

重新生成一个异常并抛出。 throw new Exception(...);

异常链接——重新生成并抛出一个新的异常，该异常中包含了当前异常的信息。 throw new Exception("...", e);

8. 工具类

Java基础类库——java.lang核心类库，java.util实用工具，java.io标准输入输出，java.awt和java.swing GUI，java.net网络功能，java.sql数据库访问...

java.lang.Object——Java中所有类的直接或间接父类

getClass()方法——final方法，不能被重载，返回一个java.lang.Class对象

java.lang.Class.getName()以String形式返回类的名称

java.lang.newInstance()创建该类的一个新的实例

equals()方法——不能做用于基本数据类型变量，默认情况下比较a和b指向的地址是否相等；可以进行重写，来比较a和b的内容是否相等

a==b，若a和b为基本数据类型则比较a和b的值，若a和b为引用型则比较a和b指向的地址

hashCode()方法——计算并返回对象的哈希码。一旦重写equals方法，一定要重写hashCode方法。

$$obj1.equals(obj2) == true$$
$$\Rightarrow obj1.hashCode() == obj2.hasCode()$$

toString()方法——返回对象的字符串表示，通过重载toString可以用来适当的显示对象的信息进行调试。

基本数据类型的包装类——每种基本数据类型都对应一种包装类。对象中所包装的值是不可改变的，要改变的唯一方法就是重新生成新的对象

提供类一些常数，比如整数最大值，浮点数正无穷等

提供valueOf(String)，toString()方法来进行字符串转换

提供xxxValue()方法来得到所包装的基本数据类型的值

equals()方法，toString()方法均被重写

Math类——常用的数学计算

System类——提供标准I/O，运行时系统的信息等

System.getProperties()方法——获得一个Properties类的对象，包含了所有可用的系统属性信息

System.getProperty(String name)方法——获得特定的系统属性的属性值

字符串——String（创建之后不会再修改和变动），StringBuffer（创建之后允许再做更改）

String——保存不可修改的Unicode字符序列

concat, replace, replaceAll, substring, toLowerCase, toUpperCase, trim, toString方法创建并返回一个新的String对象实例

endsWith, startsWith, indexOf, lastIndexOf方法进行查找

equals, equalsIgnoreCase方法进行比较

在循环中对String对象频繁操作会带来效率问题

StringBuffer——保存可修改的Unicode字符序列

可以与String互相转换

append, insert, reverse, setCharAt, setLength可以进行修改

java. util.StringToken——对字符串进行解析和分割的类

9. 泛型和Lambda表达式

泛型——“参数化类型”，使用类型作为参数，剥离数据类型与数据操作，保证运行时的数据安全。

泛型只能是类，不能是基本数据类型

包括泛型类，泛型方法，泛型接口；不支持泛型数组

泛型方法

泛型方法

声明泛型方法,该方法中使用了两个泛型，分别为T和K。这里泛型的数量可以为任意多个

声明：public <T, K> K genericMethod(GenericClass<T> t) { ... }

该方法返回值为K类型

输入参数是一个名为GenericClass的泛型类，具体类型由T指定

调用：//构造类型为Integer的GenericClass的实例，作为genericMethod的调用参数。这里Integer对应上面函数声明的T

```
GenericClass<Integer> t;
t = new GenericClass<Integer>();
// 声明类型为Double的GenericClass的引用变量，用来记录genericMethod的调用结果。这里Double对应上面函数声明的K
GenericClass<Double> r;
r = <Integer, Double>genericMethod(t); // 注意调用的语法
```

泛型类

泛型类

这里泛型的数量可以为任意多个

```
class Key<T> {
   private T key;
   public Generic<T key> {
      this.key = key;
   }
   public T doSomething_1() {
     // 并不是泛型方法，类型在构造Key实例时已指定.对该实例来说类型不可更改
      …
   }
   public <T> T doSomething_2() {
     // 声明泛型方法、使用泛型T。
     // 注意这里T是一种全新的类型，它的作用范围仅限于doSomething_2。可以与泛型类中声明的T不是一种类型
      …
   }
   public E doSomething_3(E key) {
     // 错误，cannot resolve symbol E
      …
   }
}
```

```
public TestGeneric {

   public static void main(String[] args)
   {
     // 构造String类型的Key实例
      String s = new String("12345");
      Key<String> ks = new Key<String>(s);

     // 构造Integer类型的Key实例
      Integer i = new Integer(12345);
      Key<Integer> ki = new Key<Integer>(i);

     // 调用泛型类内部定义的泛型方法
     // String对应Key<T>中的T
     // Integer对应doSmething_2中的T
      ks.<Integer>doSomething_2();
   }

}
```

泛型类中的静态方法：如果静态方法中需要使用类型，则必须将其定义为泛型方法，因为静态方法无法访问到具体的实例的类型

```
public class A<T> {
        // 错误，静态方法在内存中独立于各个实例，无法访问各实例的类型。此时编译器会报错: cannot be referenced from static context
        public static void wrongMethod(T t) {
                …
        }
        // 正确，如果静态方法需要使用泛型，则必须定义成泛型方法
        public static <E> void correctMethod(E e) {
                …
        }
}
```

泛型接口

这里泛型的数量可以为任意多个

泛型接口

```
public interface Generator<T> { public T generate(); }
```

- 若实现泛型接口时未传入泛型实参
  - 则该实现类为泛型类，声明时需要将泛型的声明一起带入

```
class FruitGenerator<T> implements Generator<T> {
    public T generate() { return null; }
}
```

- 当实现该接口时传入泛型实参
  - 接口定义中使用泛型的地方需要替换成传入的实参类型

```
class FruitGenerator implements Generator<String> {
    private String [] fruits = new String [] {“Apple”, “Banana”, “Pear”};
    public String generate() {
        return fruits[Random().nextInt(3)];
    }
}
```

通配符“?”和上下边界——在不能确定输入类型的情况下，限定输入类型的范围

```
class A<T> { … }
A<Integer> a; // 输入类型确定为Integer
A<?> b; // 输入类型不确定, 可以是任意类型
A<? super Person> c; // 输入类型为Person的父类
A<? extends Animal> d; // 输入类型为Animal的子类
A<? extends Comparable> e; // 输入类型为接口Comparable的实现类
```

Lambda表达式——函数的一种简写方式

基本写法：(参数) -> 表达式/{语句}

```
(int x) -> x+1;
(int x) -> { return x+1; }
```

x -> x+1;

() -> { System.out.println("Hellow Lambda"); }

通常被用作为参数，作为匿名类的简写

10. 输入输出流

流——不同类型的输入输出的抽象

InputStream类，read()方法，逐字节地以二进制的原始方式读取数据

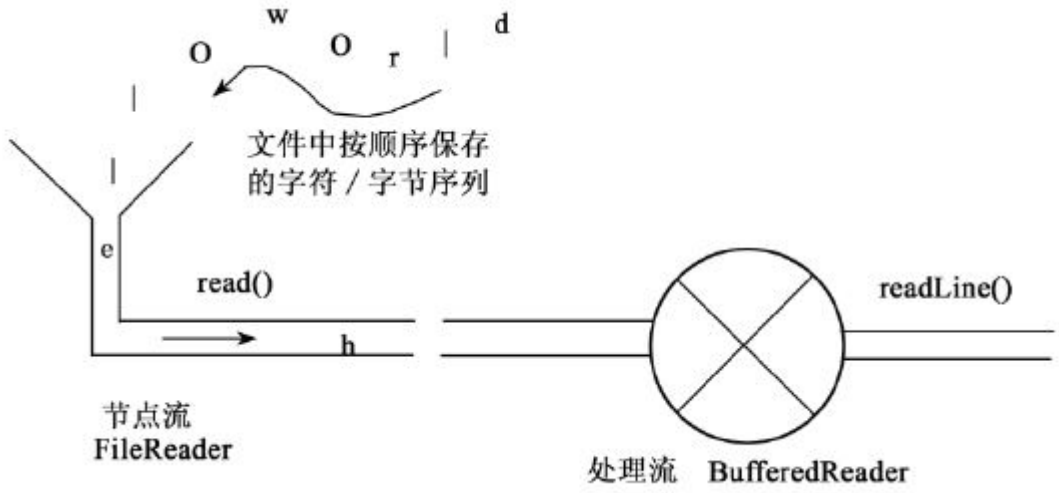
OutputStream类，write()方法，将字节写入流中

Reader类和Writer类与InputStream类和OutputStream类似，都是输入/出流，但Reader和Writer写入写出的是字符char而不是字节byte。

字节流vs字符流：字节流操作的基本单元是字节，字符流操作的基本单元是字符；字节流默认不使用缓冲区，字符流默认使用缓冲区；字节流通常用于处理二进制数据，不支持直接写入或读取字符，字符流通常用于处理文本数据，支持写入读取字符。

节点流vs处理流：节点流从或向一个特定的地方（节点）读写数据，比如FileReader；处理流是对一个已经存在的流的连接和封装，通过所封装的流的功能调用实现数据读/写功能

节点流直接与节点相连，而处理流对节点流或其他处理流进一步进行处理



流的链接——处理流的构造方法总是要带一个其他的流对象作为参数，一个流对象经过其他流的多次包装就成为流的链接

11. 多线程

程序——静态的代码

进程——程序的动态执行过程，具有私有的内存空间

线程——更小的执行单位，共享内存，单个顺序的流控制

java.lang.Thread创建多线程——继承Thread，不推荐，继承Thread之后不能再继承其他的类，程序可扩展性低

start()方法——只能调用一次，创建一个新的线程并使之可以运行

run()方法——新的线程开始其生命周期

yield()方法——暂停当前正在执行的线程并让其他同等优先级的线程开始执行

sleep()方法——使得线程休眠

```
public class ThreadExample extends Thread {
        // 重写run()方法，该线程的start()方法被调用后JVM
会自动调用该方法
        public void run() {
                System.out.println("I'm running!");
        }
}
public class Test {
        public static void main(String [] args) {
                Thread t = new ThreadExample();
                // 启动线程，t的run()方法将被调用
                t.start();
        }
}
```

继承Thread，需要重写run方法，重写的run()方法会在线程start()方法被调用后自动调用

实现Runnable接口创建多线程——将Thread类与其要处理的任务分开，允许从其他类继承

```
public class RunnableExample implements Runnable {
        public void run() {
                System.out.println("I'm running");
        }
}
public class Test {
        public static void main(String [] args) {
                 RunnableExample r = new RunnableExample();
                // 通过向Thread构造方法传递runnable对象创
建线程
                Thread t = new Thread(r);
                t.start();
        }
}
```

实现Runnable接口，作为参数传入Thread中。

Java线程的状态

新建（New）——尚未开始

就绪（Runnable）——进入线程队列排队等待CPU时间片

运行（Running）——正在运行，直至任务完成或时间片用尽

阻塞（Blocked）——认为挂起或资源被占用时终止运行；阻塞状态不能进入排队队列，阻塞消除后转入就绪状态，进入排队队列

死亡（Dead）——完成全部工作或被强制终止

线程的基本控制

stop()——通常设定标记变量的方法来决定线程是否应当终止

```
class MyThread implements Runnable {
        boolean end = false;
        public void run() {
                while (!end) {
                        System.out.println("I'm running!");
                }
        }
        public void stopRun() {
                end = true;
        }
}
```

sleep()——挂起线程的执行

join()——将一个线程加入到本线程中，本线程的执行会等待着另一线程的执行完毕

setPriority(int priority)——设置线程的优先级（MIN_PRIORITY, NORM_PRIORITY, MAX_PRIORITY），其中主线程具有普通优先级，新建县城继承其父线程的优先级

setDaemon(true)——将线程设置为守护线程

非Daemon线程（普通线程）

Daemon线程（守护线程）——运行在后台，比如GC。

主线程不能是Daemon线程，当所有非Daemon线程运行结束时，JVM强制结束所有Daemon线程并退出

多线程的同步——同时运行的线程需要共享数据，因而需要进行同步

互斥锁——保证共享数据操作的完整性，用synchronized关键字。

每个对象都对应于一个"互斥锁"标记，用来保证任意时刻都只能有一个线程访问这个对象。