PATH: 是属于操作系统属性，定义所有课执行程序的路径。

CLASSPATH:是Java程序解释类文件时所使用的的加载路径。

当Java命令运行的时候实际上JVM会自动找到CLASSPATH的属性，然后找到此属性对应的内容，通过此内容指定的路径来加载所需要的类。

3.1 字符与字符串

|  |  |  |  |
| --- | --- | --- | --- |
| NO | 方法 | 类型 | 说明 |
| 1 | public String(char[] value) | 构造 | 将字符数组变成字符串STRING对象 |
| 2 | public String(char[] value,  int offset,  int count) | 构造 | 将部分字符数组变成String |
| 3 | public char charAt(int index) | 普通 | 返回指定索引对应的字符信息 |
| 4 | public char[] toCharArray() | 普通 | 将字符串以字符数组的形式返回 |

3.2 字节与字符串

|  |  |  |  |
| --- | --- | --- | --- |
| NO | 方法 | 类型 | 说明 |
| 1 | public String(byte[] bytes) | 构造 | 将全部字符数组变成字符串 |
| 2 | public String(byte[] bytes,  int offset,  int length) | 构造 | 部分字节数组变成字符串 |
| 3 | public byte[] getBytes() | 普通 | 将字符串变为字节数组 |
| 4 | public byte[] getBytes([String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) charsetName)  throws [UnsupportedEncodingException](file:///C:\software\jdk-8u144-docs-all\docs\api\java\io\UnsupportedEncodingException.html) | 普通 | 进行编码转换 |

3.3 字符串比较

|  |  |  |  |
| --- | --- | --- | --- |
| NO | 方法 | 类型 | 说明 |
| 1 | public boolean equals([Object](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\Object.html) anObject) | 普通 | 进行等值判断，区分大小写 |
| 2 | public boolean equalsIgnoreCase([String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) anotherString) | 普通 | 进行等值判断，不区分大小写 |
| 3 | public int compareTo([String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) anotherString) | 普通 | 判断两个字符串的大小（按照字符编码比较），次方法的返回值有如下三种结果：  ·=0：表示要比较的两个字符串内容相等  ·>0：表示大于的结果  ·<0: 表示小于的结果 |

3.4 字符串查找

|  |  |  |  |
| --- | --- | --- | --- |
| NO | 方法 | 类型 | 说明 |
| 1 | public boolean contains([CharSequence](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\CharSequence.html) s) | 普通 | 判断指定的内容是否存在 |
| 2 | public int indexOf([String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) str) | 普通 | 由前向后查找字符串的位置，如果找到则返回（第一个字母）字母的索引，如果找不到则返回-1 |
| 3 | public int indexOf([String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) str,  int fromIndex) | 普通 | 由指定位置向后查找字符串的位置，找不到返回-1 |
| 4 | public int lastIndexOf([String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) str) | 普通 | 由后向前查找字符串的位置，如果找到则返回（第一个字母）字母的索引，如果找不到则返回-1 |
| 5 | public int lastIndexOf([String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) str,  int fromIndex) | 普通 | 由指定位置向前查找字符串的位置，找不到返回-1 |
| 6 | public boolean startsWith([String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) prefix) | 普通 | 判断是否是指定的字符串开头 |
| 7 | public boolean startsWith([String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) prefix,  int toffset) | 普通 | 以指定位置开始判断是否是指定的字符串开头 |
| 8 | public boolean endsWith([String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) suffix) | 普通 | 判断是否以指定的字符串结尾 |

3.5 字符串替换

|  |  |  |  |
| --- | --- | --- | --- |
| NO | 方法 | 类型 | 说明 |
| 1 | public [String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) [**replaceAll**](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html#replaceAll-java.lang.String-java.lang.String-)([**String**](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) regex, [**String**](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) replacement) | 普通 | 用新的内容全部替换掉旧的内容 |
| 2 | public [String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) replaceFirst([String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) regex,  [String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) replacement) | 普通 | 替换首个满足条件的内容 |

3.6字符串截取

|  |  |  |  |
| --- | --- | --- | --- |
| NO | 方法 | 类型 | 说明 |
| 1 | public [String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) substring(int beginIndex) | 普通 | 从指定索引截取到结尾 |
| 2 | public [String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) substring(int beginIndex,  int endIndex) | 普通 | 从指定索引截取到结束索引 |

3.7字符串拆分

|  |  |  |  |
| --- | --- | --- | --- |
| NO | 方法 | 类型 | 说明 |
| 1 | public [String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html)[] split([String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) regex) | 普通 | 按照指定的字符串进行全部拆分 |
| 2 | public [String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html)[] split([String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) regex,  int limit) | 普通 | 按照指定的字符串进行部分拆分，最后的数组长度由limit决定（最多的长度由LIMIT决定）的，即：前面拆，后面不拆 |

如果拆分的时候只是写了一个空字符串（""不是NULL）,表示按照每一个字符进行拆分

如果是一些敏感字符（正则标记）严格来讲是拆分不了，如果真的遇见了拆分不了的情况，那么使用“//（就是一个/）”进行转译后才可以拆分。

3.8 其他方法

|  |  |  |  |
| --- | --- | --- | --- |
| NO | 方法 | 类型 | 说明 |
| 1 | public [String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) concat([String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) str) | 普通 | 字符串的链接，与‘+’类似 |
| 2 | public [String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) toLowerCase() | 普通 | 转小写 |
| 3 | public [String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) toUpperCase() | 普通 | 转大写 |
| 4 | public [String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) trim() | 普通 | 去掉字符串左右两边的空格，中间的保留 |
| 5 | public int length() | 普通 | 取得字符串的长度 |
| 6 | public [String](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\String.html) intern() | 普通 | 数据入池 |
| 7 | public boolean isEmpty() | 普通 | 判断是否是空字符串，空字符串不是null，而是“”，长度0 |
| 8 |  |  |  |
| 9 |  |  |  |

如果isEmpty()不方便可以使用””.equals(str)

STATIC关键字

·static方法不能直接访问非static属性和方法，只能访问static属性和方法。

·非static方法可以直接访问static属性和方法，不受任何限制。

主方法

·public:主方法是程序的开始，所以这个方法对任何的操作一定是可见的，那么就然是可见的那就要使用public（公共）

·static:证明此方法是由类名称调用的；

·void：主方法是一切的执行的开始点，既然是所有的开头，那么就不能够回头，执行完毕为止

·mian：是系统规定好的名称，不能够修改

·String args[]：程序运行的时候传递的参数

代码块

在程序编写之中可以直接使用“{}”定义一段语句，那么根据此部分定义的位置以及申明关键字的不同共可以分为四种：普通代码块，构造块，静态块，同步代码块（等待多线程的时候）

尽可能不使用代码块。

普通代码块：

如果一个代码块写在方法里面，那么就称它为普通代码块。

普通代码块例子：

|  |
| --- |
| public class TestBlock  {  public static void main(String[] args)  {  {//普通代码快  int num = 10;  System.out.println("num = " + num);  }  int num = 100;  System.out.println("num = " + num);  }  } |

为了防止在方法里面编写代码过多时有可能产生的变量重名，对一个代码进行局部的分割。

在以后编写的代码的时候，一个方法里面的代码别写太长。

构造块

如果一个代码块写在类里面，那么就称它为构造块。

范例：构造块

|  |
| --- |
| class Book  {  public Book(){  System.out.println("AAAA IN");  }  {//构造块  System.out.println("BBBB OUT");  }  }  public class TestBlock  {  public static void main(String[] args)  {  new Book();  new Book();  new Book();  }  } |

运行结果：

|  |
| --- |
| C:\testdemo>javac TestBlock.java  C:\testdemo>java TestBlock  BBBB OUT  AAAA IN  BBBB OUT  AAAA IN  BBBB OUT  AAAA IN |

构造块优先于构造方法执行。

如果产生了多个对象，构造块被多次执行。

静态块

如果一个代码块用STATIC进行修饰的话，就称为静态块，需要分为两种情况

情况一：在非主类中使用

|  |
| --- |
| class Book  {  public Book(){  System.out.println("AAAA IN");  }  {  System.out.println("BBBB OUT");  }  static{//静态块  System.out.println("CCCC Book static");  }  }  public class TestBlock  {  public static void main(String[] args)  {  new Book();  new Book();  new Book();  }  } |

运行结果：

|  |
| --- |
| C:\testdemo>javac TestBlock.java  C:\testdemo>java TestBlock  CCCC Book static  BBBB OUT  AAAA IN  BBBB OUT  AAAA IN  BBBB OUT  AAAA IN |

静态块优先于构造块执行，而且不管实例化多少个对象，静态块只执行一次，他的作用主要是为了类中的static变量初始化

情况二：在主类中定义静态块

|  |
| --- |
| public class TestBlock  {  static{  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  }  public static void main(String[] args)  {  System.out.println("Hello World");  }  } |

运行结果：

|  |
| --- |
| C:\testdemo>javac TestBlock.java  C:\testdemo>java TestBlock  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Hello World |

静态块优先于主方法执行

内部类

所谓的内部类指的就是在一个类的内部继续定义了其它内部结构类的情况

范例：内部类

|  |
| --- |
| class outer  {  private String msg = "hello";  class ineer  {//定义内部类  public void print(){  System.out.println(msg);  }  }  public void fun(){  new ineer().print();  }  }  public class TestA  {  public static void main(String[] args)  {  new outer().fun();  }  } |

内部类的最大的特点是：可以方便的访问外部类的私有操作。

外部类也可以通过内部类对象轻松的访问内部类的私有属性

范例：访问内部类的私有属性

|  |
| --- |
| class outer  {  private String msg = "hello";  class ineer  {  private String ms = "hi";  public void print(){  //外部类的当前对象外部类.this  System.out.println(outer.this.msg);  }  }  public void fun(){  //外部类访问内部类对象的私有属性  System.out.println(new ineer().ms);  }  }  public class TestA  {  public static void main(String[] args)  {  new outer().fun();  }  } |

内部类class文件的形式是‘outer$ineer.class’。所有的‘$’是在文件中的命名，如果换回程序中就变成了‘.’也就是说内部类的名称是‘外部类.内部类’。

内部类对象的实例化语法：

|  |
| --- |
| 外部类.内部类对象 = new 外部类().new 内部类(); |

范例：实例化内部类对象

|  |
| --- |
| class outer  {  private String msg = "hello";  class ineer  {  public void print(){  System.out.println(msg);  }  }  }  public class TestA  {  public static void main(String[] args)  {  outer.ineer oi = new outer().new ineer();  oi.print();  }  } |

内部类不可能离开外部类的实例化对象，所以一定要先实例化外部类对象以后才可以可以使用内部类对象。如果真的使用到内部类，也基本上不会像以上的操作进行的，一定是通过外部类访问内部类。

如果现在一个内部类只希望被一个外部了访问，不能被外部调用

|  |
| --- |
| class outer  {  private String msg = "hello";  private class ineer  {  public void print(){  System.out.println(msg);  }  }  }  public class TestA  {  public static void main(String[] args)  {  outer.ineer oi = new outer().new ineer();  oi.print();  }  } |

STATIC定义内部类

使用static定义的属性或者方法是不受类实例化控制的，所以如果使用了static定义的内部类，它一定不受外部类的实例化对象控制

如果一个内部类使用了static定义的话，那么这个类就变为了一个外部类，并且只能访问外部类中定义的static操作。相当于定义了一个外部类

|  |
| --- |
| class outer  {  private static String msg = "hello";  static class ineer  {  public void print(){  System.out.println(msg);  }  }  } |

实例化static内部类

|  |
| --- |
| 外部类.内部类对象 = new 外部类. 内部类(); |

|  |
| --- |
| class outer  {  private static String msg = "hello";  static class ineer  {  public void print(){  System.out.println(msg);  }  }  }  public class TestA  {  public static void main(String[] args)  {  outer.ineer oi = new outer.ineer();  oi.print();  }  } |

如果以后看见可以直接实例“类.类.类”，这是一个使用了static定义的内部类

**在方法中定义内部类**

内部类可以在任意位置上定义：类中，代码块，方法，其中方法中定义内部类是比较常见的形式。

范例：在方法里面定义内部类

|  |
| --- |
| class outer  {  private String msg = "hello";  public void fun(){  class ineer  {  public void print(){  System.out.println(Outer.this.msg);  }  }  new ineer().print();  }  }  public class TestA  {  public static void main(String[] args)  {  new outer().fun();  }  } |

范例：访问方法中定义的参数或者变量

|  |
| --- |
| class outer  {  private String msg = "hello";  public void fun(int num){  double a = 99.9;  class ineer  {  public void print(){  System.out.println(Outer.this.msg);  System.out.println(num);  System.out.println(a);  }  }  new ineer().print();  }  }  public class TestA  {  public static void main(String[] args)  {  new outer().fun(10);  }  } |

此时没有加任何修饰，方法中的内部类，可以访问方法的参数和定义的变量，只适合jdk1.8以后的版本。如果在jdk1.7以及之前的版本有一个严格的要求，方法中定义的内部类如果要访问方法的参数和定义的变量，一定要在参数和变量前加上‘final’标记

正规的代码：

|  |
| --- |
| class outer  {  private String msg = "hello";  public void fun(final int num){  final double a = 99.9;  class ineer  {  public void print(){  System.out.println(Outer.this.msg);  System.out.println(num);  System.out.println(a);  }  }  new ineer().print();  }  }  public class TestA  {  public static void main(String[] args)  {  new outer().fun(10);  }  } |

**链表的定义与使用**

**3.1链表基本概念**

链表是一种最为简单的数据结构，它的主要目的是依靠引用关系来实现多个数据的保存，那么假设要保存的数据是字符串。

范例：定义一个Node类

|  |
| --- |
| class Node  {  private String data;  private Node next;  public Node(String data){  this.data = data;  }  public void setNextNode(Node next){  this.next = next;  }  public Node getNextNode(){  return next;  }  public String getData(){  return this.data;  }  }  public class LinkDemo  {  public static void main(String[] args)  {  Node root = new Node("huochetou");  Node a = new Node("chengxiangA");  Node b = new Node("chengxiangB");  root.setNextNode(a);  a.setNextNode(b);  print(root);  // Node currentNode = root;  // while(currentNode != null){  // System.out.println(currentNode.getData());  // currentNode = currentNode.getNextNode();  // }  }  public static void print(Node node){  if(node == null){  return;  }  System.out.println(node.getData());  print(node.getNextNode());  }  } |

递归实现比较好

**3.2链表的基本实现**

|  |
| --- |
| class Node  {  private String data;  private Node next;  public Node(String data){  this.data = data;  }  public void setNextNode(Node next){  this.next = next;  }  public Node getNextNode(){  return next;  }  public String getData(){  return this.data;  }  public void addNew(Node newNode){  if(this.next == null){  this.next = newNode;  }else{  this.next.addNew(newNode);  }  }  public void printNode(){  System.out.println(this.data);  if(this.next != null){  this.next.printNode();  }  }  }  class Link  {  private Node root;  public void add(String data){  Node newNode = new Node(data);  if(root == null){  this.root = newNode;  }else{  this.root.addNew(newNode);  }  }  public void print(){  if(this.root != null){  this.root.printNode();  }  }  }  public class LinkTest2  {  public static void main(String[] args)  {  Link lk = new Link();  lk.add("hello");  lk.add("world");  lk.add("www");  lk.add("ibm");  lk.add("mdln");  lk.print();  }  } |

通过以上代码发现链表的基本特点：

·客户端的代码不用去关注具体的node以及引用关系的细节，只关注link类中提供的方法；

·Link类的主要功能是提供node对象的产生和根节点的使用。

·Node类主要负责数据的保存以及引用关系的分配。

**3.3 开发可用列表**

指的是可以使用链表实现数据的新增，修改，删除以及查询。

3.3.1 程序基本结构

修改以上程序结构，node类只能被link类使用。

这个时候内部类明显是一个最好的选择。内部类可以使用private定义，这样一个内部类只能被一个外部类使用；另一点，内部类可以方便的与外部类进行私有属性的访问。

3.3.2增加数据

|  |
| --- |
| class Link  {  private class Node  {  private String data;  private Node next;  public Node(String data){  this.data = data;  }  public void addNode(Node newNode){  if(this.next == null){  this.next = newNode;  }else{  this.next.addNode(newNode);  }  }  }  private Node root;  public void add(String data){  Node newNode = new Node(data);  if(data == null){  return;  }  if(root == null){  this.root = newNode;  }else{  this.root.addNode(newNode);  }    }  public int size(){  return this.count;  }  } |

3.3.3取得链表长度

|  |
| --- |
| class Link  {  private class Node  {  private String data;  private Node next;  public Node(String data){  this.data = data;  }  public void addNode(Node newNode){  if(this.next == null){  this.next = newNode;  }else{  this.next.addNode(newNode);  }  }  }  private Node root;  private int count = 0;  public void add(String data){  Node newNode = new Node(data);  if(data == null){  return;  }  if(root == null){  this.root = newNode;  }else{  this.root.addNode(newNode);  }  count++;  }  public int size(){  return this.count;  }  } |

3.3.4判断空链表

·判断root是否为空

·判断链表长度是否为0

|  |
| --- |
| public boolean isEmpty(){  return this.count == 0;  } |

3.3.5内容查询

|  |
| --- |
| public boolean containsNode(String data){  if(data.equals(this.data)){  return ture;  }else{  if(this.next != null){  return this.next.containsNode(data);  }else{  return false;  }  }  }  public boolean contains(String data){  if(data == null || this.root ==null){  return false;  }  Return this.root.containsNode(data);  } |

3.3.6根据索引取得数据

|  |
| --- |
| 内部类  public String getNode(int index){  if(Link.this.foot ++ == index){  return this.data;  }else{  return this.next.getNode(index);  }  }  外部类  public String get(int index){  if(index > this.count){  return null;  }  this.foot =0;  return this.root.getNode(index);  } |

3.3.7修改指定索引的内容

|  |
| --- |
| 内部类  public void setNode(int index,Object data){  if(Link.this.foot ++ == index){  this.data = data;  }else{  this.next.setNode(index,data);  }  }  外部类  public void set(int index,String data){  if(index > this.count){  return;  }  this.foot =0;  this.root.setNode(index,data);  } |

3.3.8删除链表数据（Public void remove(数据类型变量)）

·删除的是root节点，处理在link类中

·删除的是非root节点，处理在Node类中

|  |
| --- |
| 内部类Node  public void removeNode(Node previous,String data){  if(data.equals(this.data)){  previous.next = this.next;  }else{  this.next.removeNode(this,data);  }  }  外部类Link  public void remove(String data){  if(this.contains(data)){  if(data.equals(this.root.data)){  this.root = this.root.next;  }else{  this.root.next.removeNode(this.root,data);  }  count--;  }  } |

3.3.9将链表转换为数组对象（public 数据类型 [] toArray()）

Link类的定义

|  |
| --- |
| public String[] toArray(){  if (this.root == null)  {  return null;  }  this.foot = 0;  this.retArray = new String[this.count];  this.root.toArrayNode();  return this.retArray;  } |

在Node类中处理数组数据的保存

|  |
| --- |
| public void toArrayNode(){  Link.this.retArray[Link.this.foot ++] = this.data;  if(this.next != null){  this.next.toArrayNode();  }  } |

3.4使用链表

|  |  |  |  |
| --- | --- | --- | --- |
| NO | 方法 | 类型 | 描述 |
| 1 | public void add(数据类型变量) | 普通 | 向列表之中增加新的数据 |
| 2 | public int size() | 普通 | 取得链表中元素的个数 |
| 3 | public boolean isEmpty() | 普通 | 判断是否是空链表 |
| 4 | Public Boolean contains(数据类型变量) | 普通 | 判断某一个数据是否存在 |
| 5 | Public 数据类型 get(int index) | 普通 | 根据索引取得数据类型 |
| 6 | Public void set(int index,数据类型变量) | 普通 | 使用新的内容替换掉指定索引的旧内容 |
| 7 | Public void remove(数据类型变量) | 普通 | 删除指定数据，如果是对象则要进行对象比较 |
| 8 | public 数据类型 [] toArray() | 普通 | 将链表以对象数组的形式进行返回 |

**继承性**

继承的实现

继承用extends实现

|  |
| --- |
| Class 子类 extends 父类{} |

子类也被称为派生类，父类也被称为基类（超类，super class）

继承优点：

·子类可以直接将父类中的操作继续使用，即代码重用。

·子类可以继续扩充属于自己的标准。

3.3继承的限制

利用extends关键之在大多数情况下不需要考虑到继承的限制（前提：按照标准格式开发）但是事实上由于要限制用户的使用，所以针对继承也有一定的要求。

1. Java不允许多重继承，但是允许多层继承。

注：希望一个子类同事继承多个父类时，可以使用多层继承

|  |
| --- |
| Class A{}  Class B extends A{}  Class C extends B{} |

相当于C是B的子类，C是A的孙子类，对于多层继承并没有层数的限制，但是一般开发中不要超过3层继承关系。

1. 子类在继承父类的时候严格来讲会继承父类的所有操作。但是对于所有的私有变量是属于隐式继承，对于所有的非私有变量是属于显示继承。
2. 在子类对象构造之前，一定会调用父类的构造（默认调用父类的无参构造），以保证父类的对象先实例化再实例化子类对象。

那么对于子类而言，相当于隐含了一个“super()”

疑问：super()和this()都放在首行，你又说，如果子类构造没有编写super()的话，会自动使用一个super()调用父类的无参构造，如果写上了“this()”,是否表示子类无法调用父类的无参构造？

通过代码的验证：super()和this()不能同时存在；不管子类怎么折腾，都存在一个永恒的前提：子类构造调用前一定要执行父类的构造。为父类对象初始化后再给子类对象初始化。

**覆写**

3.1 方法的覆写

当子类定义了与父类方法名称相同，参数的类型及个数，返回值相同的方法是，我们就说发生了覆写。

覆写的使用原则:（被动）

如果发现父类中的方法名称功能不足（不适用与本子类），但又必须使用这个方法名称时，就需要采用覆写这个概念实现。

被子类覆写的方法不能够拥有比父类更严格的访问权限。

如果

如果父类中的方法使用private申明，那么这个方法对子类来说是不可见的，就算子类定义了一个鱼子完全相同的并且符合覆写要求的方法，那么也不能发生覆写。这时候子类定义的这个方法相当于子类自己的一个方法。

关于super.方法()与this.方法()的区别：

使用“this.方法()”会首先查找本类中是否存在要有调用的方法名称，如果存在则直接，如果不存在则查找父类中的方法，如果有就调用，如果没有回发生编译时的错误。

使用“super.方法()”明确的表示不是调用子类的方法，直接调用父类中指定的方法

面试题：请解释重载和覆写的区别（解释overload和override的区别）

|  |  |  |  |
| --- | --- | --- | --- |
| No | 区别 | 重载 | 覆写 |
| 1 | 英文单词 | Overloading | Override |
| 2 | 发生范围 | 发生在一个类里面 | 发生在继承关系中 |
| 3 | 定义 | 方法名称相同，参数的类型及个数不同 | 方法名称相同，参数的类型及个数相同，方法返回值相同 |
| 4 | 权限 | 重载没有权限的限制 | 被覆写的方法不能让拥有比父类更严格的访问权限 |

在使用overloading的时候返回值能否不同？

在发生重载的时候，返回值可以不同，但考虑到程序设计的统一性，重载时，尽量保持方法的返回值类型相同。

属性的覆盖（了解）

如果子类定义了和父类完全相同的属性名称时，就称为属性的覆盖

This与 super 的区别？

|  |  |  |  |
| --- | --- | --- | --- |
| No | 区别 | This | Super |
| 1 | 功能 | 调用本类的构造，本类的方法，本类的属性 | 调用父类的构造，父类的方法，父类的属性 |
| 2 | 形式 | 先查找本类是否存在有指定的调用结构，如果有直接使用，如果没有调用父类的定义 | 不查找子类，直接调用父类的定义 |
| 3 | 特殊 | 表示本类的当前对象 | - |

实例：

TestArray.java

FINAL关键字

1. 使用final定义的类不能够再有子类
2. 使用final定义的方法不能被子类所覆写
3. 使用final定义的变量就成为了常量，常量必须在定义时设置好内容，并且不能被修改

常量名大写

全局常量：public static final声明的全局常量

**多态性**

对于对象的转型，给出以下的经验总结:

·80%的情况下都只会使用向上转型，因为可以得到参数类型的统一，方便与我们的程序设计；

|-子类定义的方法大部分情况下以父类的方法名称为标准进行覆写，不要过多的扩充方法

·5%的情况下会使用向下转型，调用子类的特殊方法

·15%的情况下是不转型的

|  |
| --- |
| Classcastexception |

此时出现的一场表示的是类转换异常，指的的两个没有关系的类对象强制发生向下转型时所带来的异常。所以向下转型时存在有风险的。

为了保证转型的顺利进行，java中提供了一个关键字 instanceof

|  |
| --- |
| 对象instanceof 类返回boolean型 |

某个对象是某个类的实例，返回true，否则返回false；

**抽象类的定义以及使用**

抽象方法是没有方法体“{}”，抽象类和抽象方法要使用abstarct修饰。

抽象类的使用原则如下：

·抽象类必须有子类，即：每一个抽象类一定要被子类所继承；

·抽象类的子类（子类不是抽象类）必须要覆写抽象类之中的全部抽象方法（强制子类覆写）；

·抽象类的对象实例化需要依靠子类完成，采用向上转型的方式处理。

抽象类的相关限制

1. 抽象类由于会存在一些属性，那么一定会存在构造方法，目的：为属性初始化，并且子类对象实例化的时候依然满足先执行父类的构造函数在执行子类的构造函数。
2. 抽象类不能够使用final修饰
3. 外部抽象类不能使用static声明，而内部的抽象类允许使用static声明，使用static声明的内部类相当于一个外部抽象类，继承的时候使用“外部类.内部类”的形式表示类名称。
4. 在任何情况下，如果要执行类中static声明的方法是，都可以不需要对象直接使用。抽象类也是这样的
5. 有些时候，由于抽象类中需要一个特定的系统子类操作，所以可以忽略掉外部子类。

|  |
| --- |
| abstract class A  {//定义一个抽象类  public abstract void print();  public static class B extends A{//内部抽象类  public void print(){//覆写抽象类的方法  System.out.println("hello");  }  }  //这个方法不受实例化对象的控制  public static A getInstance(){  return new B();  }  }  public class Test  {  public static void main(String[] args)  {  //此时取得抽象类对象的时候完全不需要知道B这个子类的存在  A a = new A.getInstance();  a,print();  }  } |

这样的设计在系统类库之中比较常见，目的：为用户隐藏不需要知道的子类

3.3抽象类的应用—模板设计

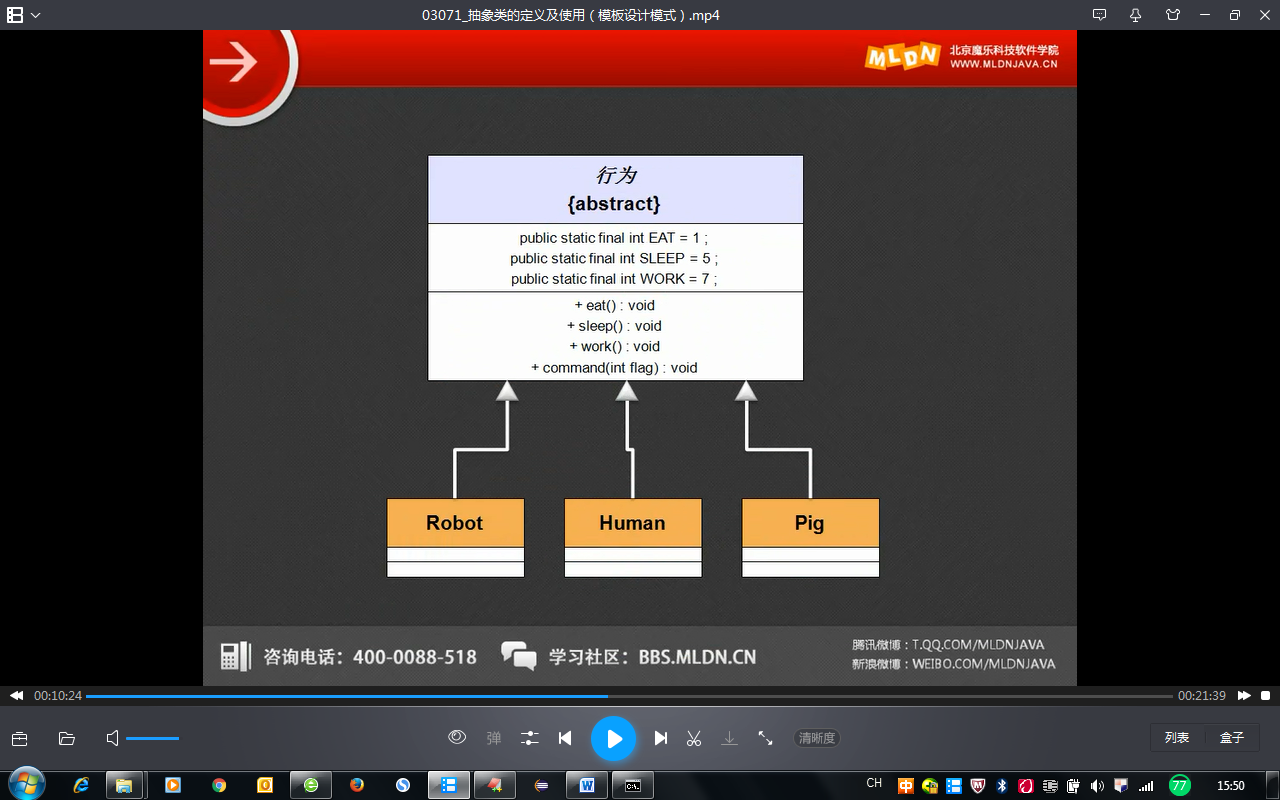
例如，现在有三类事物：

·机器人：充电，工作

·人：吃饭，工作，睡觉

·猪：吃饭，睡觉

要求可以实现以上的操作控制，即：可以任意的控制人，机器人，猪的操作行为（充电，工作，吃饭，睡觉）



**借口的定义和使用**

如果一个类之中只是由抽象方法和全局常量所组成的，那么在这种抢矿下不会讲其定义为抽象类，而是定义为借口，所以所谓的接口严格来讲就属于特殊的类。只有抽象方法和全局变量，没有构造方法。

接口的使用原则：

·接口必须要有子类，但是此时一个子类可以使用implements实现多个接口。

·接口的子类（如果不是抽象类），必须覆写接口中所有的方法。

·接口的对象可以使用子类对象的向上转型进行实例化操作。

范例：实现接口

|  |
| --- |
| interface A  {  public static final String MSG = "HELLO";  public abstract void print();  }  interface B  {  public abstract void get();  }  class X implements A,B  {  public void print(){  System.out.println("A jiekou !");  }  public void get(){  System.out.println("B jiekou !");  }  }  public class TestInterfaceA  {  public static void main(String[] args)  {  X x = new X();  A a = x;  B b = x;  a.print();  b.get();  x.print();  x.get();  }  } |

先extends继承父类在implements实现接口

一个抽象类只能继承一个抽象类，但是反过来一个接口可以使用extends同时继承多个接口（接口不能狗继承抽象类）

范例：观察接口的多继承概念

|  |
| --- |
| interface A  {  public abstract void getA();  }  interface B  {  public abstract void getB();  }  interface C extends A,B  {  public abstract void getC();  }  class X implements C  {  public abstract void getA(){}  public abstract void getB(){}  public abstract void getC(){}  } |

从继承关系上来看抽象类的限制要比接口严格：

·一个抽象类只能够继承一个抽象的父类，而接口没有这个限制；

·一个子类只能够继承一个抽象类，而却可以实现多个接口；

接口是解决单继承局限问题。

虽然从概念上来看接口只能够由抽象方法和全局变量组成,但是所有的内部结构不受到这些要求的限制，也就是说在接口里面可以定义普通内部类，抽象内部类，内部接口。

范例：在接口中定义抽象类

|  |
| --- |
| interface A  {  public abstract void getA();  abstract class B  {  public abstract void funB();  }  }  class X implements A  {  public void getA(){};  class Y extends B  {  public void funB(){} //内部抽象类的子类  }  } |

范例：在一个接口内部如果使用了static去定义一个内部接口表示是一个外部接口。

|  |
| --- |
| interface A  {  public abstract void getA();  static interface B //外部接口  {  public abstract void funB();  }  }  class X implements A.B  {  public void getB(){}  } |

接口在实际的开发之中有三大核心作用：

* 定义不同层之间的操作标准
* 表示一种操作的能力
* 表示将服务器端的远程方法视图暴露给客户端。

**接口的实际应用—标准定义**

范例：定义USB标准

|  |
| --- |
| interface USB  {  public void start();  public void stop();  } |

范例：定义电脑

|  |
| --- |
| class Computer  {  public void plugin(USB usb){  usb.start();  usb.stop();  }  } |

程序里面的标准就是用接口来定义的

**接口的应用—工厂设计模式**

|  |
| --- |
| interface Fruit  {  public void eat();  }  class Apple implements Fruit  {  public void eat(){  System.out.println("\*\*\* chi ping guo");  }  }  public class TestFactory  {  public static void main(String[] args)  {  Fruit f = new Apple();  f.eat();  }  } |

要确认一个代码是否真的好，有这么几个标准:

·客户端调用简单，不需要关注具体的细节；

·客户端之外的代码修改，不影响用户的使用，即，用户可以不用去关心程序是否变更

本次的程序语法上没有问题，但是问题出现关键字“NEW”上

对于Fruit有可能产生多个子类

|  |
| --- |
| class Oragne implements Fruit  {  public void eat(){  System.out.println("\*\*\* chi ju zi");  }  } |

现在的客户端上要想得到新的子类对象，需要修改代码:

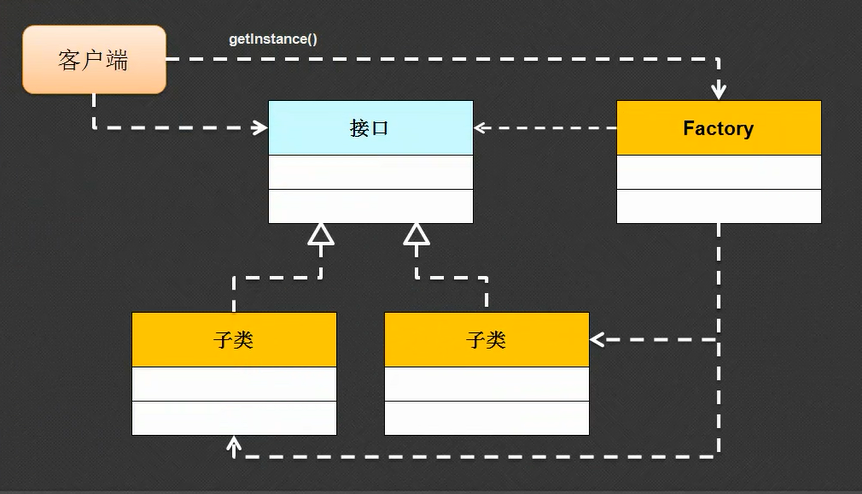
|  |
| --- |
| public class TestFactory  {  public static void main(String[] args)  {  Fruit f = new Oragne();  f.eat();  }  } |

发现如果现在直接在客户端上产生实例化对象，那么每一次更换对象，都需要修改客户端上的逻辑控制代码，这样的做法明显是不好的。

范例：增加一个过渡

|  |
| --- |
| class Factory  {  public static Fruit getInstance(String className){  if("apple".equals(className)){  return new Apple();  }else if("orange".equals(className)){  return new Oragne();  }else{  return null;  }  }  }  public class TestFactory  {  public static void main(String[] args)  {  Fruit f = Factory.getInstance("orange");  f.eat();  }  } |

现在的客户端不会看见具体的子类，因为所有的接口对象都是通过Factory类取得的没如果日后要扩充新的Fruit子类对象，则只需要修改Factory类即可，但客户端的调用不会发生变化。

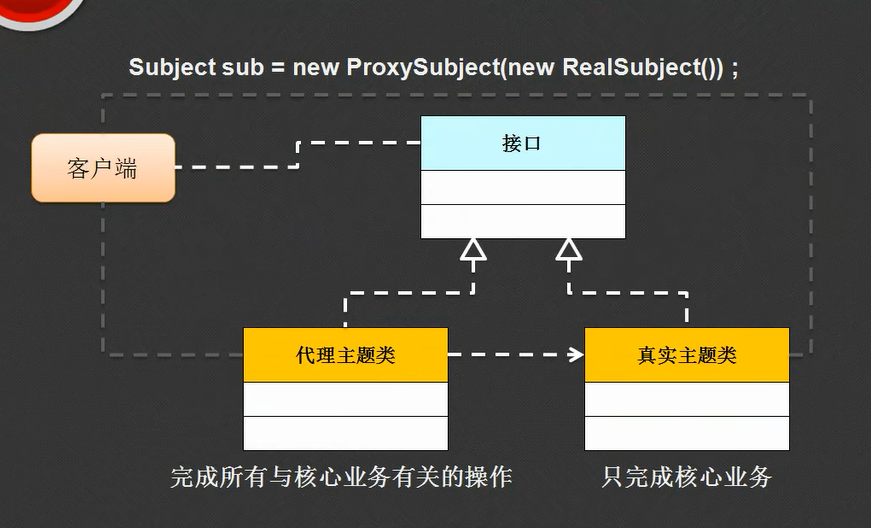


面试题：请编写一个Factory程序

**3.4 代理设计模式(Proxy)**

|  |
| --- |
| interface Subject  {  public abstract void fun();  }  class RealSubject implements Subject  {  public void fun(){  System.out.println("RealSubject");  }  }  class ProxySubject implements Subject  {  private Subject subject;    public ProxySubject(Subject subject){  this.subject = subject;  }  public void prepare(){  System.out.println("\*\*\*before\*\*\*");  }  public void fun(){  this.prepare();  this.subject.fun();  this.destroy();  }  public void destroy(){  System.out.println("\*\*\*after\*\*\*");  }  }  public class TestProxy  {  public static void main(String[] args)  {  Subject sub = new ProxySubject(new RealSubject());  sub.fun();  }  } |

代理设计模式的精髓在于有一个主题操作接口（可能有多种方法），核心的业务主要完成核心主题，而代理主题完成所有与核心主题有关的辅助性操作。



**3.5接口与抽象类的区别：**

|  |  |  |  |
| --- | --- | --- | --- |
| No | 区别 | 抽象类 | 接口 |
| 1 | 关键字 | abstract class | interface class |
| 2 | 组成 | 构造方法，普通方法，抽象方法，static方法，常量，变量 | 抽象方法，全局常量 |
| 3 | 子类使用 | class 子类 extends 抽象类 | class 子类 implements 接口，接口。。。 |
| 4 | 关系 | 抽象类可以实现多个接口 | 接口不能继承抽象类，却可以继承多个父类接口 |
| 5 | 权限 | 各种权限 | 只能够使用public权限 |
| 6 | 限制 | 单继承局限 | 没有单继承局限 |
| 7 | 子类 | 抽象类和接口必须有子类，子类必须覆写全部的抽象方法 | |
| 8 | 实例化对象 | 依靠子类对象的向上转型实现实例化 | |

经过比较发现，抽象类中支持的功能绝对要比接口中多，但是只有一点不好，单继承局限，所以这一点掩盖了所有抽象类的有点。即，抽象类和接口都可以使用的时候，有限考虑接口。

一个不成文的参考:

·在进行某些公共操作的时候一定要定义接口；

·有了接口就需要利用子类完善方法；

·如果自己写的接口，那么绝对不要使用关键字NEW直接实例化接口子类，使用工厂类完成；

**Object类**

3.1Object类的基本定义

Object类是所有类的父类，那么最大的好处在于，利用Object类可以接受全部类的对象，因为可以向上自动转型。

问题：为什么在Object类中定义一个无参的构造方法？

既然Object类是所有类的父类，那么对象实例化的时候，子类构造方法一定要调用父类的无参构造。

从严格意义上来说（一般不遵守），任何一个简单Java类都应该覆写Object类中的如下三个方法：

·取得对象信息：public String toString();

·对象比较：public boolean equals([Object](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\Object.html) obj);

·取得对象HASH码：public int hashCode();

3.2 取得对象信息：toString

在进行对象输出的时候，输出操作会自动调用对象中的toString()方法将对象变成呢字符串后自动输出，默认情况下Object类中的toString()方法为了适应所有对象的输出，所以只输出对象的编码。

3.3对象比较：qeuals()

|  |
| --- |
| public boolean equals(Object obj){  if (this == obj)  {  return true;  }  if (obj ==null)  {  return false;  }  if (!(obj instanceof Book))  {  return false;  }  Book book = (Book) obj;  if (this.title.equals(book.title)  && this.price == book.price)  {  return true;  }  return false; |

3.4Object类可以接受一切引用类型

Object类是所有类的父类，所以Object类的对象可以接受所有类的对象，可是除了类对象以外，Object类连数组和接口对象也可以接受。

范例：接受数组数据

|  |
| --- |
| public class testDemo  {  public static void main(String[] args)  {  Object obj = new int[]{1,2,3};  if (obj instanceof int[])  {  int data[] = (int[])obj;  for (int i = 0;i<data.length ;i++ )  {  System.out.println(data[i]);  }  }  }  } |

范例：Object类接受接口对象

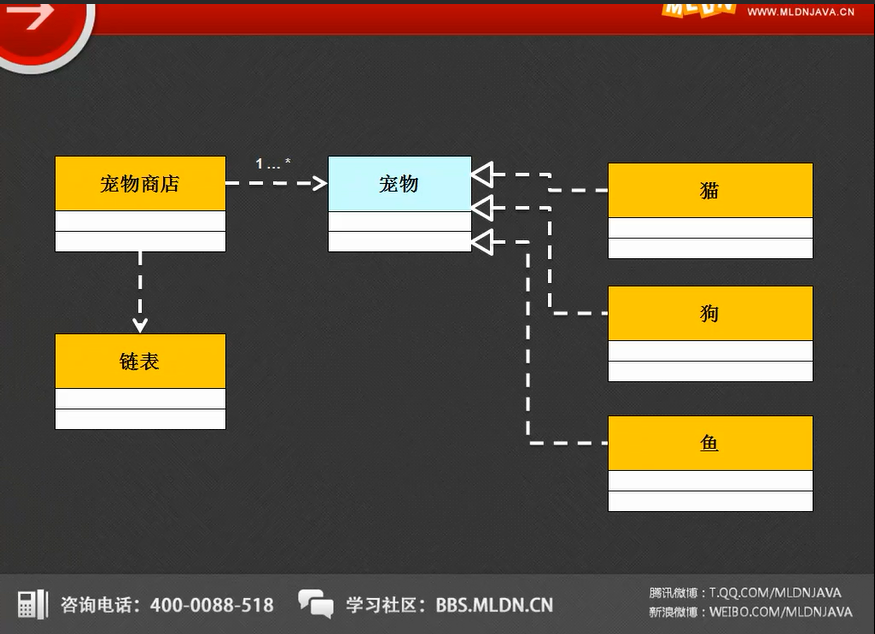
|  |
| --- |
| interface A  {  public abstract void fun();  }  class B implements A  {  public void fun(){  System.out.println("hi");  }  }  public class testDemo  {  public static void main(String[] args)  {  A a = new B();  Object obj = a;  A t = (A)obj;  t.fun();  }  } |

3.5完善链表

|  |
| --- |
| class Link  {  private class Node  {  private Object data;  private Node next;  public Node(Object data){  this.data = data;  }  public void addNode(Node newNode){  if(this.next == null){  this.next = newNode;  }else{  this.next.addNode(newNode);  }  }  public boolean containsNode(Object data){  if(data.equals(this.data)){  return true;  }else{  if(this.next != null){  return this.next.containsNode(data);  }else{  return false;  }  }  }  public Object getNode(int index){  if(Link.this.foot ++ == index){  return this.data;  }else{  return this.next.getNode(index);  }  }  public void setNode(int index,Object data){  if(Link.this.foot ++ == index){  this.data = data;  }else{  this.next.setNode(index,data);  }  }  public void removeNode(Node previous,Object data){  if(data.equals(this.data)){  previous.next = this.next;  }else{  this.next.removeNode(this,data);  }  }  public void toArrayNode(){  Link.this.retArray[Link.this.foot ++] = this.data;  if(this.next != null){  this.next.toArrayNode();  }  }  }  private Node root;  private int count = 0;  private int foot;  private Object retArray[];  public void add(Object data){  Node newNode = new Node(data);  if(data == null){  return;  }  if(root == null){  this.root = newNode;  }else{  this.root.addNode(newNode);  }  count++;  }  public int size(){  return this.count;  }  public boolean isEmpty(){  return this.count == 0;  }  public boolean contains(Object data){  if(data == null || this.root ==null){  return false;  }  return this.root.containsNode(data);  }  public Object get(int index){  if(index >= this.count){  return null;  }  this.foot =0;  return this.root.getNode(index);  }  public void set(int index,Object data){  if(index >= this.count){  return;  }  this.foot =0;  this.root.setNode(index,data);  }  public void remove(Object data){  if(this.contains(data)){  if(data.equals(this.root.data)){  this.root = this.root.next;  }else{  this.root.next.removeNode(this.root,data);  }  count--;  }  }  public Object[] toArray(){  if (this.root == null)  {  return null;  }  this.foot = 0;  this.retArray = new Object[this.count];  this.root.toArrayNode();  return this.retArray;  }  }  public class LinkTest4  {  public static void main(String[] args)  {  Link all = new Link();  all.add("a");  all.add("b");  all.add("c");  all.remove("a");  Object[] obj = all.toArray();  for (int i = 0;i<obj.length ;i++ )  {  String s = (String)obj[i];  System.out.println(s);  }  }  } |

综合实战：宠物商店

实现一个宠物商店的模型，一个宠物商店可以保存多个宠物的信息（名字，年龄），可以实现宠物的上架，下架，模糊查询的功能。



范例:定义Pet接口

|  |
| --- |
| interface Pet  {  public abstract String getName();  public abstract int getAge();  } |

范例：定义PetShop类

|  |
| --- |
| class PetShop  {  private Link pets = new Link();  public void add(Pet pet){  this.pets.add(pet);  }  public void delete(Pet pet){  this.pets.remove(pet);  }  public Link sreach(String keyWord){  Link result = new Link();  Object obj[] = this.pets.toArray();  for (int i = 0;i<obj.length ;i++ )  {  Pet p = (Pet) obj[i];  if (p.getName().contains(keyWord))  {  result.add(p);  }  }  return result;  }  } |

范例：添加Cat子类

|  |
| --- |
| class Cat implements Pet  {  private String name;  private int age;  public Cat(String name,int age){  this.name =name;  this.age = age;  }  public String getName(){  return this.name;  }  public int getAge(){  return this.age;  }  public boolean equals(Object anObject){  if(this == anObject){  return true;  }  if (anObject == null)  {  return false;  }  if (!(anObject instanceof Cat))  {  return false;  }  Cat c = (Cat)anObject;  if (this.name.equals(c.name)  &&this.age == c.age)  {  return true;  }  return false;  }  public String toString(){  return "cat name : " + this.name + ", age : " + this.age;  }  } |

范例：添加Dog子类

|  |
| --- |
| class Dog implements Pet  {  private String name;  private int age;  public Dog(String name,int age){  this.name =name;  this.age = age;  }  public String getName(){  return this.name;  }  public int getAge(){  return this.age;  }  public boolean equals(Object anObject){  if(this == anObject){  return true;  }  if (anObject == null)  {  return false;  }  if (!(anObject instanceof Dog))  {  return false;  }  Dog c = (Dog)anObject;  if (this.name.equals(c.name)  &&this.age == c.age)  {  return true;  }  return false;  }  public String toString(){  return "dog name : " + this.name + ", age : " + this.age;  }  } |

范例：测试

|  |
| --- |
| public class LinkTest5  {  public static void main(String[] args)  {  PetShop ps = new PetShop();  ps.add(new Cat("wangmao",10));  ps.add(new Cat("jingmao",11));  ps.add(new Cat("leimao",12));  ps.add(new Dog("wangdog",13));  ps.add(new Dog("jingdog",14));  ps.add(new Dog("leidog",15));  ps.delete(new Dog("jingdog",14));  Link all = ps.sreach("wang");  Object[] obj = all.toArray();  for (int i = 0;i < obj.length ;i++ )  {  System.out.println(obj[i]);  }  }  } |

**匿名内部类**

为什么需要存在匿名内部类？观察一下代码

|  |
| --- |
| interface Message  {  public abstract void fun();  }  class MessageImpl implements Message  {  public void fun(){  System.out.println("hello!");  }  }  public class TestDemoA  {  public static void main(String[] args)  {  fun(new MessageImpl());  }  public static void fun(Message msg){  msg.fun();  }  } |

一个抽象类和接口需要有子类，子类需要覆盖所有的抽象方法。

可是如果说现在的MessageImpl只使用唯一的一次。那么还有必要将其单独定义为一个类吗？所以这个时候就可以采用匿名内部类的方式进行代码的简化。

|  |
| --- |
| interface Message  {  public abstract void fun();  }  public class TestDemoA  {  public static void main(String[] args)  {  fun(new Message(){  public void fun(){  System.out.println("Hello!");  }  });  }  public static void fun(Message msg){  msg.fun();  }  } |

使用匿名内部类的时候有个前提：必须基于接口或抽象类的应用。

注意点：如果匿名内部类定义在了方法里面，方法的参数或变量要被匿名内部类访问，那么必须加上final关键字（JDK1.8以后此要求被改变）

**基本数据类型的包装类**

包装类又分为两种子类型：

·对象型包装类（Object直接子类）：Character、Boolean。

·数值型包装类（Number直接子类）：Byte、Short、Integer、Float、Double。Long

Number是一个抽象类，里面一共定义了六个操作方法：byteValue()、shortValue()、intValue()、longValue()、floatValue()、doubleValue()。

**3.2、装箱与拆箱**

现在已经存在有基本数据类型与包装类，那么这两种变量间的转换就通过以下方式定义：

·装箱操作：将基本数据类型变为包装类的形式。

|-每个包装类的构造方法都可以接受各自数据类型的变量

·拆箱操作：从包装类之中取出被包装的数据；

|-利用Number类中提供的一系列的：xxxValue()方法完成。

范例：使用int 和Integer

|  |
| --- |
| public class test  {  public static void main(String[] args)  {  Integer i = new Integer(10);  int temp = i.intValue();  System.out.println(temp \* 2);  }  } |

范例：以boolean和Boolean（不是Number子类）

|  |
| --- |
| public class test  {  public static void main(String[] args)  {  Boolean i = new Boolean(false);  boolean temp = i.booleanValue();  System.out.println(temp);  }  } |

在JDK1.5之前能够使用的操作都是以上形式的代码，但是从JDK1.5之后，Java为了方便开发提供了自动装箱与自动拆箱的机制，并且可以直接利用包装类的对象进行数学计算。

范例：观察自动装箱和自动拆箱

|  |
| --- |
| public class test  {  public static void main(String[] args)  {  Integer i = 10; //自动装箱  int temp = i; //自动拆箱  i++; //包装类直接进行数学计算  System.out.println(temp \* i);  }  } |

注意：在Integer类对象上发现可以直接赋予内容，也可以使用构造方法，有什么区别呢？

|  |
| --- |
| public class test  {  public static void main(String[] args)  {  Integer obja = 10; //直接复制  Integer objb = 10; //直接复制  Integer objc = new Integer(10);  System.out.println(obja == objb); //true  System.out.println(obja == objc); //false  System.out.println(objb == objc); //false  System.out.println(obja.equals(objc)); //true  }  } |

在使用包装类的时候很少会利用构造方法完成，几乎都是直接复制（这一点跟String相同），但是在判断内容是都相等的时候请一定记住使用equals()方法。

提示：此时Object可以统一天下了

Object可以接受一切的引用数据类型，但是由于存在自动的装箱机制，那么Objec也可以存放基本类型了。

·流程：基本数据类型→自动装箱（成为对象）→向上转型为Object

|  |
| --- |
| public class test  {  public static void main(String[] args)  {  Object obj = 10;  //object不可能直接向下转型为int  int temp = (Integer)obj; //向下变为Integer而后自动拆箱  System.out.println(temp \* 2); //true  }  } |

**3.3 数据类型转换（核心）**

使用包装类最多的情况实际上是他的数据类型转换功能上，在包装类里面提供有将String型数据变为基本数据类型的方法，使用几个代表的类做说明：

·Integer类：public static int parseInt([String](file:///D:\jdk-8u144-docs-all\docs\api\java\lang\String.html) s)

·Double类：public static double parseDouble(String s)

·Boolean类：public static boolean parseBoolean(String s)

特别需要注意的是Character类里面并不存在有字符串变为字符的方法，因为String类有一个charAt()的方法，可以根据索引取出字符内容。

范例：将字符串变为int型数据

|  |
| --- |
| public class test  {  public static void main(String[] args)  {  String str = new String("123");  int temp = Integer.parseInt(str);  System.out.println(temp \* 2);  }  } |

此时实现了字符串变为基本数据类型的操作。但是在这样的转换过程之中请一定要注意：被转换的字符串一定要由数字组成！

范例：错位代码

|  |
| --- |
| public class test  {  public static void main(String[] args)  {  String str = new String("1a3");  int temp = Integer.parseInt(str);  System.out.println(temp \* 2);  }  } |

|  |
| --- |
| Exception in thread "main" java.lang.NumberFormatException: For input string: "1  a3" |

范例：double型

|  |
| --- |
| public class test  {  public static void main(String[] args)  {  String str = new String("1.3");  double temp = Double.parseDouble(str);  System.out.println(temp \* 2);  }  } |

|  |
| --- |
| public class test  {  public static void main(String[] args)  {  String str = new String("13");  double temp = Double.parseDouble(str);  System.out.println(temp \* 2);  }  } |

范例：观察Boolean

|  |
| --- |
| public class test  {  public static void main(String[] args)  {  String str = "true";  boolean temp = Boolean.parseBoolean(str);  if(temp){  System.out.println("\*\*\* 满足条件 \*\*\*");  }else{  System.out.println("\*\*\* 不满足条件 \*\*\*");  }  }  } |

在Boolean进行转换的过程里面，如果要转换的字符串不是true或者false，那么将统一按照false进行处理。

现在既然实现了字符串变为基本数据类型的操作，那么也一定可以实现基本数据类型变为字符串的操作，对于此类操作有两种做法：

·操作一：任何基本数据类型与字符创用了“+”操作之后都表示变为字符串

|  |
| --- |
| public class test  {  public static void main(String[] args)  {  int num =100;  String str = num + "";  System.out.println(str.replaceAll("0","9"));  }  } |

这样的操作虽然可以简单的完成，但是会存在有垃圾问题。

·操作二：public static String valueOf(数据类型变量)

|  |
| --- |
| public class test  {  public static void main(String[] args)  {  int num =100;  String str = String.valueOf(num);  System.out.println(str.replaceAll("0","9"));  }  } |

这样的转换不会产生垃圾，所以在开发是往往会使用这种方式完成转换

**包的定义及使用**

包指的是一个程序的目录，在最早的时候如果要开发一个程序，只需要定义一个\*.java文件，在这个文件里面编写所需要的类文件。而在编译之后程序将直接保存在跟目录下，而利用包可以实现同一个程序的拆分，即：可以根据不同的要求将代码保存在不同的目录下。

如果要定义包可以使用package关键字完成。

范例：定义包

|  |
| --- |
| package cn.mldn.demo; //包  public class Hello  {  public static void main(String args[]){  System.out.println("Hello World !");  }  } |

不要手工创建这些目录，应该使用命令自动生成：

·打包编辑：javac–d . Hello.java;

|-“-d”:生成目录，根据package的定义生成目录；

|- “.”：设置保存的路径，如果为“.”表示在当前所在路径下生成；

·在解释程序的时候不要进入到包里面解释程序，应该在包外面输入类的完成名称（包.类）

|-输入: javacn.mldn.demo.Hello

以后所有的类都一定要定义在包之中，那么完整的类名称就是“包.类”

**3.2 、包的导入**

范例：定义一个 cn.mldn.util.Message类

|  |
| --- |
| package cn.mldn.util;  class Message  {  public void print(){  System.out.println("Hello World!");  }  } |

范例：定义一个cn.mldn.test.TestMessage的类

|  |
| --- |
| package cn.mldn.test;  import cn.mldn.util.Message;  public class TestMessage  {  public static void main(String[] args)  {  Message msg = new Message();  msg.print();  }  } |

对于程序的编译应该采用先编译Message在编译TestMessage。

|  |
| --- |
| javac -d . Message.java |
| javac -d . TestMessage.java在编译本程序的时候出现了一下的错误提示： |
| TestMessage.java:2: 错误: Message在cn.mldn.util中不是公共的; 无法从外部程序包中  对其进行访问  import cn.mldn.util.Message; |

总结：关于public class和class声明类的完整区别

·public class：文件名必须与类名称保持一致，在一个\*.java文件里面只能够有一个public class声明，如果一个类需要被不同的包所访问，那么一定要定义为public class；

·class：文件名称可以与类名称不一致，并且一个\*.java文件里面可以有多个class定义，编译后会形成多个\*.class文件。如果一个类使用的是class定义，那么表示这个类只能够被本包所访问。

范例：修改Message.java类的定义

|  |
| --- |
| package cn.mldn.util;  public class Message  {  public void print(){  System.out.println("Hello World!");  }  } |

Java编译器考虑到了作为大型程序开发的时候有可能会存在有多个\*.java文件互相引用的情况，为了解决这个问题，提供了通配符“\*”这个操作。javac -d . \*.java

但是发现在导入包的时候也出现了一个问题，如果使用一个包中类的时候要编写“import 包.类”，那么如果说现在要使用一个包中多个类的时候，那么肯定要重复编写“import 包.类”，这样是很麻烦的，此时可以使用“import 包.\*”

|  |
| --- |
| package cn.mldn.test;  import cn.mldn.util.\*  public class TestMessage  {  public static void main(String[] args)  {  Message msg = new Message();  msg.print();  }  } |

使用“包.\*”指的并不是全部导入，而是只导入程序里面所需要的类，所以不需要去考虑性能的问题。

有可能同一个代码里面会同时导入不同包，并且这不同的包里面有可能会存在同名类。

那么在这种情况下为了可以明确的找到所需要的类，可以再使用类的时候加上包名称

**3.3、系统常用包**

·java.lang：包含了String、Object、Integer等类，从JDK1.1开始此包自动导入；

·java.lang.reflect：反射开发包；

·java.util：java的工具包，提供了大量的工具类，像链表；

·java.util.regex：正则工具包；

·java.text：国际化处理程序包；

·java.io：进行输入输出处理以及文件操作；

·java.net：网络编程开发包；

·java.sql：数据库程序开发包；

·java.applet：Applet程序开发包（已经不用了）；

|-applet指的是在网页上嵌套的程序，可以使用Applet做一些动态效果

|  |
| --- |
| package cn.mdln.demo;  import java.Applet.\*;  import java.awt.\*;  public class MyApplet extends Applet  {  public void print(Graphics g){  r.drawLine(10,10,100,100);  }  } |

编译之后并不能表示其可以正常的使用，要需要编写Html代码进行引用

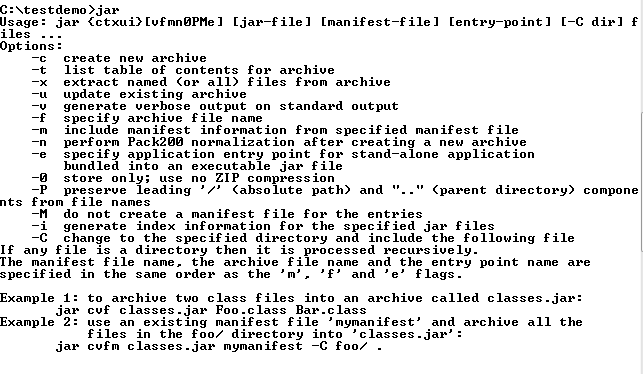
|  |
| --- |
| <applet code ="cn.mdln.demo.MyApplet" width = "300" height = "300" /> |

解释：关于Applet与Application的区别？

Applet是java在网页上嵌套的程序，是采用绘图的方式完成的显示，而Application是在主方法中运行，通过命令行执行。随着时间的发展Applet不在使用了（如果要实现同样的功能，都是用Canvas绘图 html5）.

·java.awt、javax.swing：图形界面的开发包，主要功能是进行单机版下横须界面编写的;

**3.4、jar命令**



范例：定义一个Message.java文件

|  |
| --- |
| package cn.mldn.util;  public class Message  {  public void print()  {  System.out.println("Hello World!");  }  } |

随后打包编译此文件：javac -d . Message.java

此时会兴盛“包.类”的形式。将包的代码压缩：jar -cvf my.jar cn

生成的my.jar文件并不能够直接使用，必须配置CLASSPATCH才可以加载。

|  |
| --- |
| SET CLASSPATH = .;c:\mydemo\my.jar |

以后的开发中会使用到大量的第三方的jar文件，那么所有的jar文件必须配置CLASSPATH,否则不能使用。

最简单的配置方式在环境变量中设置



**访问控制权限**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| No | 范围 | private | default | protected | public |
| 1 | 在同一类中 | √ | √ | √ | √ |
| 2 | 在同一包的不同类 |  | √ | √ | √ |
| 3 | 在不同包的子类 |  |  | √ | √ |
| 4 | 在不同包的非子类 |  |  |  | √ |

除了public之外，对于封装可以使用 private、pretected、defualt，只不过一般不考虑使用defualt。

Protected演示

范例：定义cn.mldn.demoa.A类

|  |
| --- |
| package cn.mldn.demoa;  public class A  {  protected static String info = "hello";  } |

范例：定义cn.mldn.demob.B类

|  |
| --- |
| package cn.mldn.demob;  import cn.mldn.demoa.A;  public class B extends A  {  public void print()  {  System.out.println("A: " + super.info);  }  } |

范例：测试

|  |
| --- |
| package test;  import cn.mldn.demob.B;  public class Test  {  public static void main(String[] args)  {  new B().print();  }  } |

如果在test中直接输出A类中的info属性

范例：错误的访问

|  |
| --- |
| package test;  import cn.mldn.demoa.A;  public class Test  {  public static void main(String[] args)  {  A a = new A();  System.out.println(a.info);  }  } |

权限的选择，给出以下的建议：

·声明属性就使用private

·声明方法就是用public

命名要求

·类名称每一个单词的首字母大写，其余字母小写，例如：StudentInfo

·属性名称第一个单词字母小写，而后每个单词首字母大写，例如：studentName

·方法名称属性名称第一个单词字母小写，而后每个单词首字母大写，例如：getName().

·常量名称使用大写字母表示，例如：MSG;

·包名称使用小写字母，例如：cn.mldn.demo;

**构造方法私有化**

3.1、单例设计模式（Singleton）

在正常情况下，如果给了你一个类，那么只有通过产生对象之后才可以操作这个类。

范例：观察如下代码

|  |
| --- |
| class Singleton  {  public void print(){  System.out.println("Hello");  }  }  public class TestDemo  {  public static void main(String[] args)  {  Singleton sln = null;  sln = new Singleton();  sln.print();  }  } |

现在Singleton类里面存在有构造方法，如果没有明确定义一个构造的话，会自动在编译时生成一个无参的，什么也不做的构造方法。即，一个类至少会保留一个构造方法。

范例：修改Singleton类定义

|  |
| --- |
| class Singleton  {  private Singleton(){}//构造方法私有化  public void print(){  System.out.println("Hello");  }  } |

一旦构造方法私有化了，那么在外部将无法直接通过关键字new来进行对象的实例化操作了。

范例：错误的代码

|  |
| --- |
| public class TestDemo  {  public static void main(String[] args)  {  Singleton sln = null;  **sln = new Singleton();**  sln.print();  }  } |

整个程序要如何修改才可以实现在TestDemo类里面得到Singleton类的实例化呢

分析步骤：

1. 构造防范上使用了private声明，那么就表示这个构造方法只能在类的内部使用，既然如此，可以直接在类的内部实例化一个对象。

|  |
| --- |
|  |

1. 现在的instance在Singleton里面只是一个普通的类属性，所有的普通类属性必须在类产生实例化对象之后才可以使用，是否存在有一种方式，可以让这个类属性不受Singleton类实例化对象的控制呢？

如果使用了static声明instance属性，那么就表示可以在一个类没有产生实例化对象的时候直接使用该属性

|  |
| --- |
| class Singleton  {  static Singleton instance = new Singleton();  private Singleton(){}  public void print(){  System.out.println("Hello");  }  }  public class TestDemo  {  public static void main(String[] args)  {  Singleton sln = null;  sln = Singleton.instance;  sln.print();  }  } |

1. 在一个类定义的时候应该首先想到的就是类中的属性进行封装

|  |
| --- |
| private static Singleton instance = new Singleton(); |

1. 而一旦封装之后如果要想访问此属性只能够通过getter方法，那么就需要提供一个getter方法同样不受到Singleton的实例化对象的控制，继续使用static属性。

|  |
| --- |
| class Singleton  {  static Singleton instance = new Singleton();  private Singleton(){}  public static Singleton getInstance(){  return instance;  }  public void print(){  System.out.println("Hello");  }  }  public class TestDemo  {  public static void main(String[] args)  {  Singleton sln = null;  sln = Singleton.getInstance();  sln.print();  }  } |

代码意义：

如果说现在要想控制一个类中实例化对象的产生个数，那么首先要锁定的就是类中的构造方法，因为实例化任何新对象都要使用构造方法。如果构造方法被锁了，那么就自然无法产生新的实例化对象了

可是既然需要时一个实例化对象，那么就可以在类的内部使用static方式来定义一个公共的对象，并且通过static方法返回唯一的一个对象，这样外部不管有多少次调用，那么最终一个类只能产生唯一的一个对象，这样的设计就属于单例设计模式（Singleton）

1. Instance用final修饰

|  |
| --- |
| class Singleton  {  private static final Singleton instance = new Singleton();  private Singleton(){}  public static Singleton getInstance(){  return instance;  }  public void print(){  System.out.println("Hello");  }  }  public class TestDemo  {  public static void main(String[] args)  {  Singleton sln = null;  sln = Singleton.getInstance();  sln.print();  }  } |

面试题：请编写一个Singleton程序，请解释Singleton的特点？

|  |
| --- |
| class Singleton  {  private static final Singleton INSTANCE= new Singleton();  private Singleton(){}  public static Singleton getInstance(){  return instance;  }  public void print(){  System.out.println("Hello");  }  }  public class TestDemo  {  public static void main(String[] args)  {  Singleton sln = null;  sln = Singleton.getInstance();  sln.print();  }  } |

程序特点：构造方法私有化，在类的内部定义static属性与方法，利用static方法取得本类的实例化对象，这样一来不管外部会产生多少个Singleton类的对象，但是本质上永远只有唯一的一个实例化对象。

可是对于单例设计模式有两种形式：饿汉式，懒汉式。

在之前所编写的单例实际上就属于饿汉式的应用，在Singleton类定义的时候就已经准备好了一个Singleton类的实例化对象INSTANCE,而并没有关心这个对象有没有使用。

而懒汉式的最大特点就在于它是在第一次使用的时候才进行实例化操作。

范例：懒汉式

|  |
| --- |
| class Singleton  {  private static final Singleton INSTANCE;  private Singleton(){}  public static Singleton getInstance(){  if (INSTANCE == null) //此时没有实例化  {  INSTANCE = new Singleton(); //实例化对象  }  return instance;  }  public void print(){  System.out.println("Hello");  }  }  public class TestDemo  {  public static void main(String[] args)  {  Singleton sln = null;  sln = Singleton.getInstance();  sln.print();  }  } |

**3.2、多例设计模式**

多例设计模式，可以让一个类产生指定多个实例化对象。

例如：现在要定义一个表示一周时间数的类，这个类只能取七个对象。

例如：现在要求定义有个表示性别的类，这个类只能够取两个对象

范例：表示性别的类

|  |
| --- |
| class Sex  {  private String title;  private static final Sex MALE = new Sex("男");  private static final Sex FEMALE = new Sex("女");  private Sex(String title){  this.title = title;  }  public static Sex getInstance(int ch)  {  switch(ch){  case 1:  return MALE;  case 2:  return FEMALE;  default:  return null;  }  }  public String toString(){  return this.title;  }  }  public class TestDemo  {  public static void main(String[] args)  {  Sex s = Sex.getInstance(1);  System.out.println(s);  }  } |

在JDK1.7之前，SWITH只能够利用int或char进行判断，但是正因为如果纯粹是数字或者字符意义不明确，所以增加了String的支持。

|  |
| --- |
| class Sex  {  private String title;  private static final Sex MALE = new Sex("男");  private static final Sex FEMALE = new Sex("女");  private Sex(String title){  this.title = title;  }  public static Sex getInstance(String ch)  {  switch(ch){  case "man":  return MALE;  case "woman":  return FEMALE;  default:  return null;  }  }  public String toString(){  return this.title;  }  }  public class TestDemo  {  public static void main(String[] args)  {  Sex s = Sex.getInstance("woman");  System.out.println(s);  }  } |

如果现在不希望使用String在switch语句（这个习惯实际上不好），那么可以引入一个标记的接口。

|  |
| --- |
| class Sex  {  private String title;  private static final Sex MALE = new Sex("男");  private static final Sex FEMALE = new Sex("女");  private Sex(String title){  this.title = title;  }  public static Sex getInstance(int ch)  {  switch(ch){  case 1:  return MALE;  case 2:  return FEMALE;  default:  return null;  }  }  public String toString(){  return this.title;  }  }  interface Choose  {  public int MAN = 1;  public int WOMAN = 2;  }  public class TestDemo  {  public static void main(String[] args)  {  Sex s = Sex.getInstance(Choose.MAN);  System.out.println(s);  }  } |

以上代码如果自己看是没有什么问题的，但是给别人看就看不懂了

不管是单例设计还是多例设计的核心是：构造方法私有化！！！

**异常的捕获及处理**

**3.1、异常额产生**

异常是导致程序中断执行的一种指令流，异常一旦出现并且没有进行合理处理的话，那么程序就将中断执行。

范例：产生异常

|  |
| --- |
| public class TestDemo  {  public static void main(String[] args)  {  System.out.println("start");  System.out.println("10/0:"+(10/0));  System.out.println("end");  }  } |

|  |
| --- |
| Start  Exception in thread "main" java.lang.ArithmeticException: / by zero  at TestDemo.main(TestDemo.java:6) |

一旦产生异常之后发现产生异常的语句以及之后的语句不再执行，默认情况下进行异常信息的输出，然后自动结束程序的执行。

**3.2、处理异常**

如果要想进行异常的处理，在java之中提供了三个关键字：try、catch、finally，而这三个关键字的语法如下：

|  |
| --- |
| try{  //有可能出现异常的语句  }[catch (异常类型对象){  //异常处理 ;  }catch (异常类型对象){  //异常处理 ;  }catch (异常类型对象){  //异常处理 ;  }......] [finally {  不管是否出现异常，都执行的统一代码  }] |

对于以上的操作组合：try…catch、try…catch…finally、try…finally。

范例：try…catch异常处理

|  |
| --- |
| public class TestDemo  {  public static void main(String[] args)  {  System.out.println("start");  try{  System.out.println("10/0:"+(10/0));  }catch(ArithmeticException e){  System.out.println("\*\*\*\*\*\* 出现异常\*\*\*\*\*\*\*");  }  System.out.println("end");  }  } |
| Start  \*\*\*\*\*\* 出现异常\*\*\*\*\*\*\*  End |

由于使用了异常处理，这样即使程序中出现了异常，发现程序也可以正常执行结束了。

出现异常的目的是为了解决异常，所以为了能够处理异常，可以使用异常类中提供的printStackTrace()方法进行异常的完整输出。

|  |
| --- |
| public class TestDemo  {  public static void main(String[] args)  {  System.out.println("start");  try{  System.out.println("10/0:"+(10/0));  }catch(ArithmeticException e){  e.printStackTrace();  }  System.out.println("end");  }  } |
| Start  java.lang.ArithmeticException: / by zero  at TestDemo.main(TestDemo.java:7)  end |

发现打印的异常信息是很完整的。

范例：还可以使用try…catch…finally

|  |
| --- |
| public class TestDemo  {  public static void main(String[] args)  {  System.out.println("start");  try{  System.out.println("10/0:"+(10/0));  }catch(ArithmeticException e){  e.printStackTrace();  }finally  {  System.out.println("### 不管是否出现异常我都执行 ###");  }  System.out.println("end");  }  } |
| Start  java.lang.ArithmeticException: / by zero  at TestDemo.main(TestDemo.java:7)  ### 不管是否出现异常我都执行 ###  End |

|  |
| --- |
| public class TestDemo  {  public static void main(String[] args)  {  System.out.println("start");  try{  System.out.println("10/2:"+(10/2));  }catch(ArithmeticException e){  e.printStackTrace();  }finally  {  System.out.println("### 不管是否出现异常我都执行 ###");  }  System.out.println("end");  }  } |
| Start  10/2:5  ### 不管是否出现异常我都执行 ###  End |

|  |
| --- |
| public class TestDemo  {  public static void main(String[] args)  {  System.out.println("start");  try{  int x= Integer.parseInt(args[0]);  int y= Integer.parseInt(args[1]);  System.out.println("x/y:"+(x/y));  }catch(ArithmeticException e){  e.printStackTrace();  }finally  {  System.out.println("### 不管是否出现异常我都执行 ###");  }  System.out.println("end");  }  } |

以上的程序将由用户输入操作数据，于是可能存在如下情况出现：

·用户执行的时候不输入参数（java TestDemo）未处理

|  |
| --- |
| Start  ### 不管是否出现异常我都执行 ###  Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0  at TestDemo.main(TestDemo.java:7) |

·用户输入的参数不是数据（java TestDemo a b）未处理

|  |
| --- |
| Start  ### 不管是否出现异常我都执行 ###  Exception in thread "main" java.lang.NumberFormatException: For input string: "a  "  at java.lang.NumberFormatException.forInputString(NumberFormatException.  java:65)  at java.lang.Integer.parseInt(Integer.java:580)  at java.lang.Integer.parseInt(Integer.java:615)  at TestDemo.main(TestDemo.java:7) |

·被除数为0（java TestDemo 10 0）已处理

范例：加入多个catch

|  |
| --- |
| public class TestDemo  {  public static void main(String[] args)  {  System.out.println("start");  try{  int x= Integer.parseInt(args[0]);  int y= Integer.parseInt(args[1]);  System.out.println("x/y:"+(x/y));  }catch(ArithmeticException e){  e.printStackTrace();  }catch(ArrayIndexOutOfBoundsException e){  e.printStackTrace();  }catch(NumberFormatException e){  e.printStackTrace();  }finally  {  System.out.println("### 不管是否出现异常我都执行 ###");  }  System.out.println("end");  }  } |

以上的异常都已经知道了，你还让他出现，这绝对是技术问题。

**3.3异常的处理流程**

首先来观察两个异常类的继承结构

|  |  |
| --- | --- |
| ArithmeticException | NumberFormatException |
| java.lang.Object  java.lang.Throwable  java.lang.Exception  java.lang.RuntimeException  java.lang.ArithmeticException | java.lang.Object  java.lang.Throwable  java.lang.Exception  java.lang.RuntimeException  java.lang.IllegalArgumentException  java.lang.NumberFormatException |

发现所有的异常都是Throwable类的子类。而在Throwable下有两个子类。

（面试题：请解释ERROR和EXCEPTION的区别）

·Error：只的是JVM错误，即：此时的程序还没有执行，用户无法处理

·Exception：程序运行中产生的异常，用户可以处理。

所谓的异常处理指的就是所有Exception以及它的子类。



1、当程序在运行的过程之中出现了异常后，jvm自动根据异常的类型实例化一个与之类型匹配的异常类对象。（此处用户不用去关心new，由系统自动处理）

2、产生了异常对象之后会判断当前的语句上是否存在有异常处理，如果没有异常处理，那么就交给jvm进行默认的异常处理。处理的方式：输出异常信息，而后结束程序调用；

3、如果此时存在有异常的捕获操作，那么会有try语句来捕获的产生的异常实例化对象，而后与try语句后会跟每一根catch语句进行比较如果现在又符合的捕获类型，则使用当前catch的与来进行异常的处理，如果不匹配，则向下匹配其他的catch语句。

4、不管最后异常处理是否能够匹配，那么都要向后执行，如果程序中存在有finally语句，那么就先执行finally中的代码，但是执行完毕后需要根据之前的catch匹配结果来决定如何执行，如果之前已经成功的捕获了异常，那么就继续执行finally之后的代码，如果之前没有成功的捕获异常，那么就将此异常交给JVM进行默认异常处理（输出异常信息，结束程序执行）

整个过程就好比方法重载一样。根据catch后面的参数类型进行匹配，但是所有java对象都存在自动的向上转型的操作支持，也就是说如果要真的匹配类型，简单的做法匹配Eception就够了。

|  |
| --- |
| public class TestDemo  {  public static void main(String[] args)  {  System.out.println("start");  try{  int x= Integer.parseInt(args[0]);  int y= Integer.parseInt(args[1]);  System.out.println("x/y:"+(x/y));  }catch(Exception e){  e.printStackTrace();  }finally  {  System.out.println("### 不管是否出现异常我都执行 ###");  }  System.out.println("end");  }  } |

此时所有的异常都使用了Exception进行处理，所以在程序之中不用再去关系到底使用哪一个异常。

两点说明：

·在捕获异常的时候，捕获范围大的异常一定要放在捕获范围小的异常之后，否则编译报错

·虽然直接捕获Exception比较方便，但是这样也不好，因为所有的异常按照同样的方式处理，在一些要求严格的项目里面，异常一定要分开处理比较好。

**3.4、throws关键字**

Throws用于方法的声明上，指的是当方法之中出现异常后交由被调用出来进行处理。

范例：throws

|  |
| --- |
| class MyMath  {//由于存在throws，那么就表示此方法里面产生的异常交给别调用处进行处理  public static int div(int x, int y) throws Exception{  return x/y;  }  } |

范例：调用以上的方法

|  |
| --- |
| public class TestDemo  {  public static void main(String[] args)  {  try{  System.out.println(MyMath.div(10,2));  }catch(Exception e){  e.printStackTrace();  }  }  } |

调用了具有throws声明的方法之后，那么不管操作是否出现异常，都必须使用try..catch来进行异常处理。

可是程序之中主方法也属于方法，那么主方法上能否继续使用throws抛出异常呢

|  |
| --- |
| public class TestDemo  {  public static void main(String[] args) throws Exception  {//表示此异常产生以后，会直接通过主方法抛出  System.out.println(MyMath.div(10,2));  }  } |

在主方法上如果继续抛出异常，那么这个异常就交给JVM进行处理也就是采用默认的处理方式，输出异常信息。而后结束程序调用。

主方法上不要加上throws，因为程序如果出错了，也希望可以正常的结束

**3.5throw关键字**

在程序之中可以直接使用throw手工的抛出一个异常类的实例化对象。

|  |
| --- |
| public class TestDemo  {  public static void main(String[] args)  {  try{  throw new Exception("自己定义的异常！");  }catch(Exception e){  e.printStackTrace();  }  }  } |

面试题：请解释throw和throws的区别？

·throw：指的是在方法之中人为抛出一个异常类对象（这个异常类对象可能是自己实例化或者是抛出已存在的）

·throws：在方法的声明上使用，表示此方法在调用的时候必须处理异常

**3．6、重要的代码类型：异常处理标准格式**

现在要求定义一个div()方法，要求：被处理之前打印信息，被处理之后打印信息，如果调用过程中出现异常，则交给被调用处进行处理。

范例：给出代码不出错的情况

|  |
| --- |
| class MyMath  {  public static int div(int x, int y){  int result = 0;  System.out.println("Start");  result = x/y;  System.out.println("End");  return result;  }  }  public class TestDemo  {  public static void main(String[] args)  {  System.out.println(MyMath.div(10,2));  }  } |

以上的方法我们应该进行一些合理的处理，如果出现异常了应该交给被调用处处理，那么方法上应该加上throws抛出异常

|  |
| --- |
| class MyMath  {  public static int div(int x, int y) throws Exception{  int result = 0;  System.out.println("Start");  result = x/y;  System.out.println("End");  return result;  }  }  public class TestDemo  {  public static void main(String[] args)  {  try{  System.out.println(MyMath.div(10,2));  }catch(Exception e){  e.printStackTrace();  }    }  } |

如果以上的代码真的出错了呢，需要在方法中作如下处理

|  |
| --- |
| class MyMath  {  public static int div(int x, int y) throws Exception{  int result = 0;  System.out.println("Start");  try{  result = x/y;  }catch(Exception e){  throw e; //继续抛异常  }finally{  System.out.println("End");  }  return result;  }  }  public class TestDemo  {  public static void main(String[] args)  {  try{  System.out.println(MyMath.div(10,0));  }catch(Exception e){  e.printStackTrace();  }    }  } |

以上代码是异常处理的标准格式。实际上以上的代码还可以缩减，把catch省略

|  |
| --- |
| class MyMath  {  public static int div(int x, int y) throws Exception{  int result = 0;  System.out.println("Start");  try{  result = x/y;  }finally{  System.out.println("End");  }  return result;  }  }  public class TestDemo  {  public static void main(String[] args)  {  try{  System.out.println(MyMath.div(10,0));  }catch(Exception e){  e.printStackTrace();  }    }  } |

**3.7、RuntimeException类**

观察以下代码

|  |
| --- |
| public class TestDemo  {  public static void main(String[] args)  {  int temp = Integer.parseInt("100");  }  } |

观察下parseInt()方法的定义

·方法：public static int parseInt(String s) throws NumberFormatException

此时parseInt()抛出了NumberFormatException，按照道理来说，要进行强制性的异常捕获，可是现在并没有这种强制性的要求，来观察下NumberFormatException继承结构

|  |
| --- |
| java.lang.Object  java.lang.Throwable  java.lang.Exception  java.lang.RuntimeException→运行时异常  java.lang.IllegalArgumentException  java.lang.NumberFormatException |

Java中为了方便用户代码的编写，专门提供了一种RuntimeException类，这种异常类的最大特点是：程序在编译的时候不会强制性要求用户处理异常，用户可以根据自身的需要选择性的处理，如果没有处理又发生了异常，将交给JVM进行默认的处理。即：RuntimeException类的子类，可以由用户根据需求选择性的进行处理。

面试题：请解释Exception和RuntimeException的区别。请列举出个常见的RuntimeException

·Exception是RuntimeException的父类；

·Exception的异常必须要被处理，RuntimeException的异常可以选择性的处理；

常见的RuntimeException：[ArithmeticException](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\ArithmeticException.html)， [NullPointerException](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\NullPointerException.html)， [ClassCastException](file:///C:\software\jdk-8u144-docs-all\docs\api\java\lang\ClassCastException.html)

**3.8、assert关键字（了解）**

assert关键字是JDK1.4的时候引入的，其主要的功能是进行断言。

在Java中的断言值得是程序执行到某行代码处时一定是预期的结果

范例：观察断言

|  |
| --- |
| public class TestDemo  {  public static void main(String[] args)  {  int num = 10;  //...  assert num == 20:"num的内容不是20";  System.out.println("num = " + num);  }  } |

默认情况下断言是不影响程序运行的，就是说在java解释程序的时候，断言是默认不起作用的。

启用断言：java -ea TestDemo

|  |
| --- |
| C:\mydemo>java -ea TestDemo  Exception in thread "main" java.lang.AssertionError: num bu shi 20  at TestDemo.main(TestDemo.java:7) |

**类图描述**

3.1类图

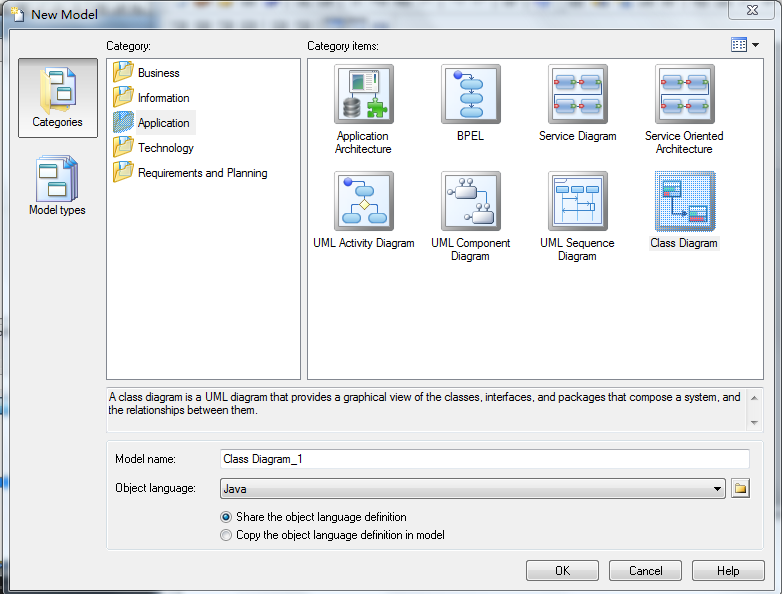
如果要想描述类图一般有三个组成结构：

·第一层：类名称，如果是抽象类使用斜体字

·第二层：描述类中的属性，对于属性肯定要封装如果是封装使用“-”表示

|-public（+）、protected（#）、private（-）

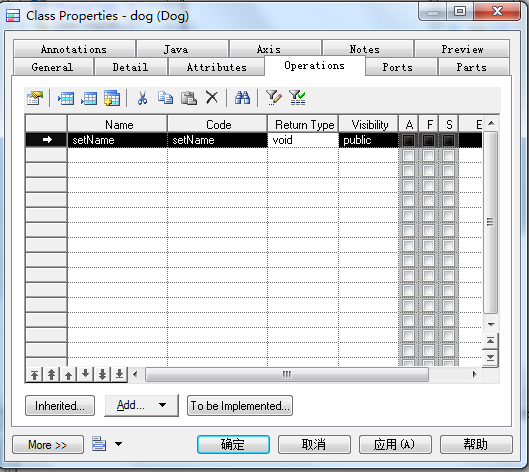
·第三层：类中定义的方法



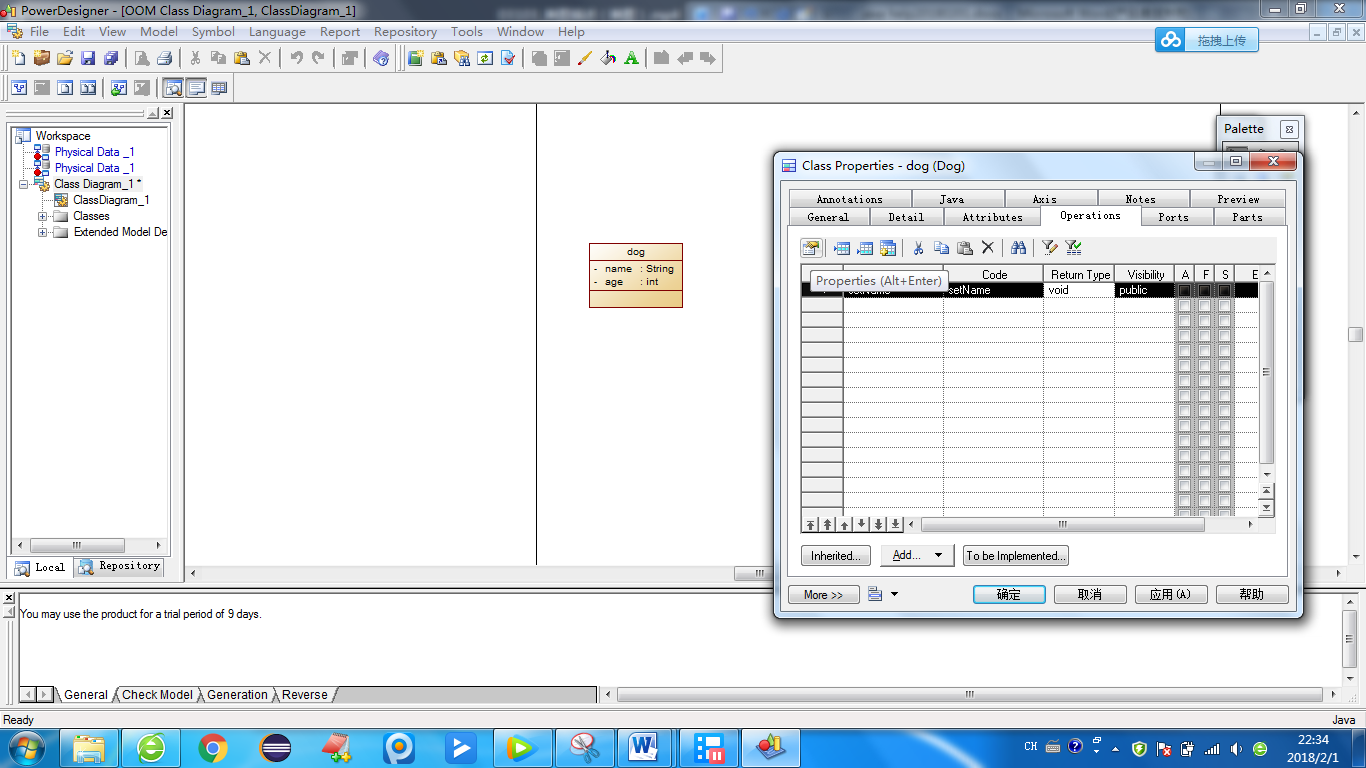
属性：

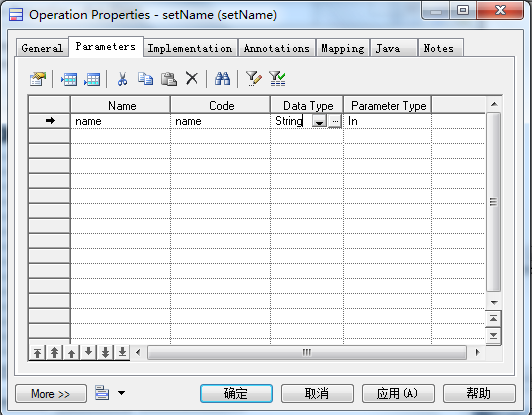


方法：



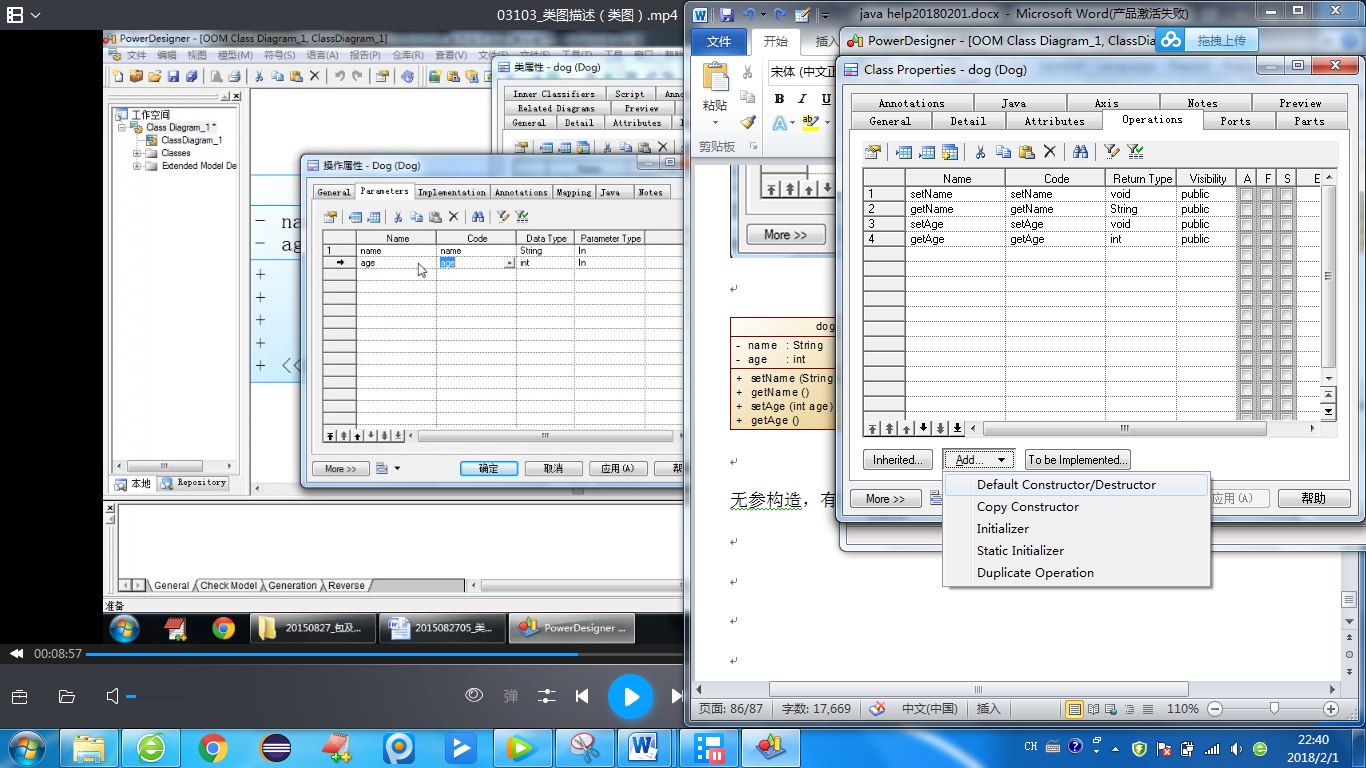
设置方法的参数

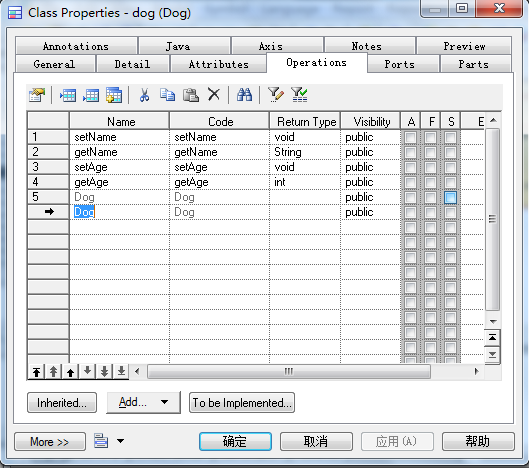






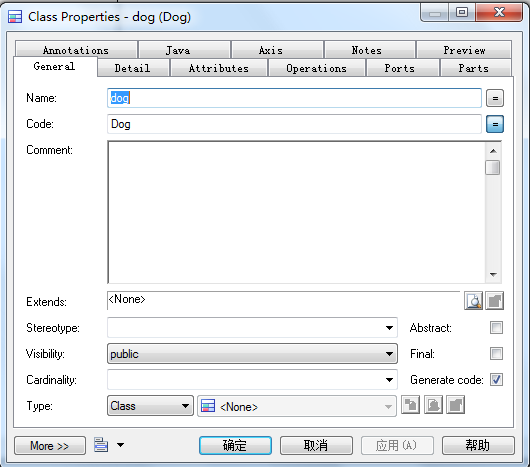
无参构造，有参构造







抽象类：





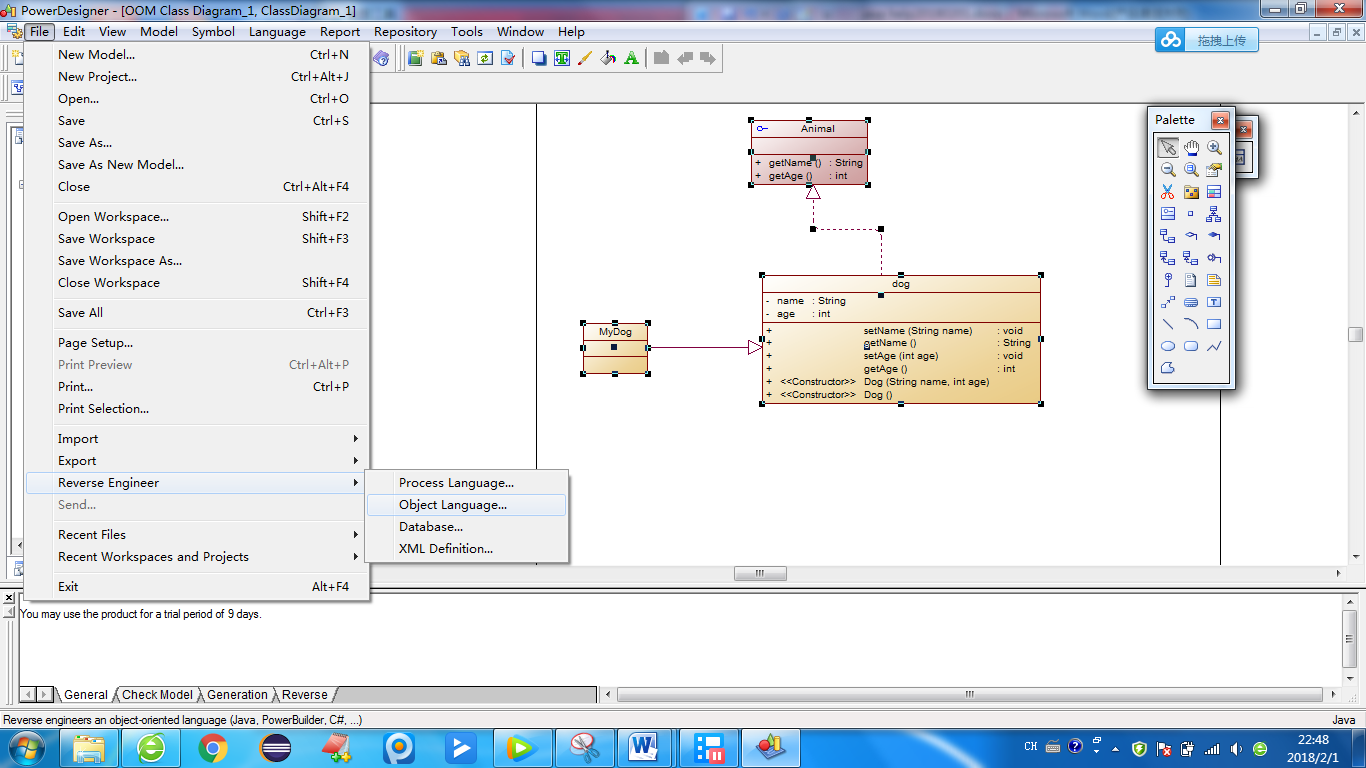
接口的实现虚线

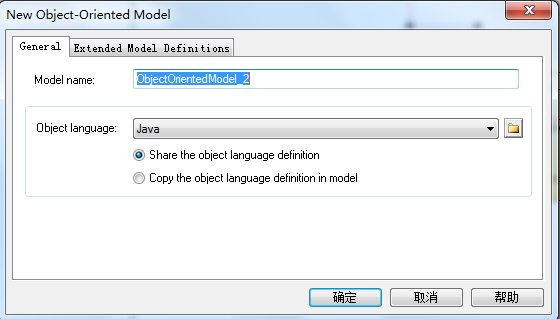


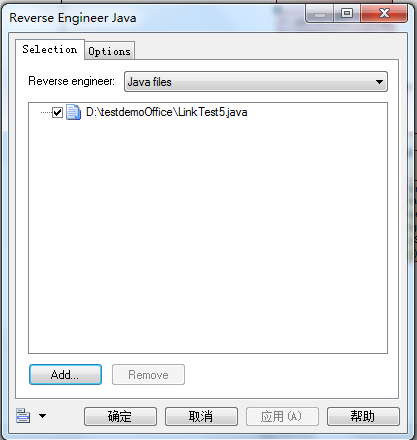
继承实线



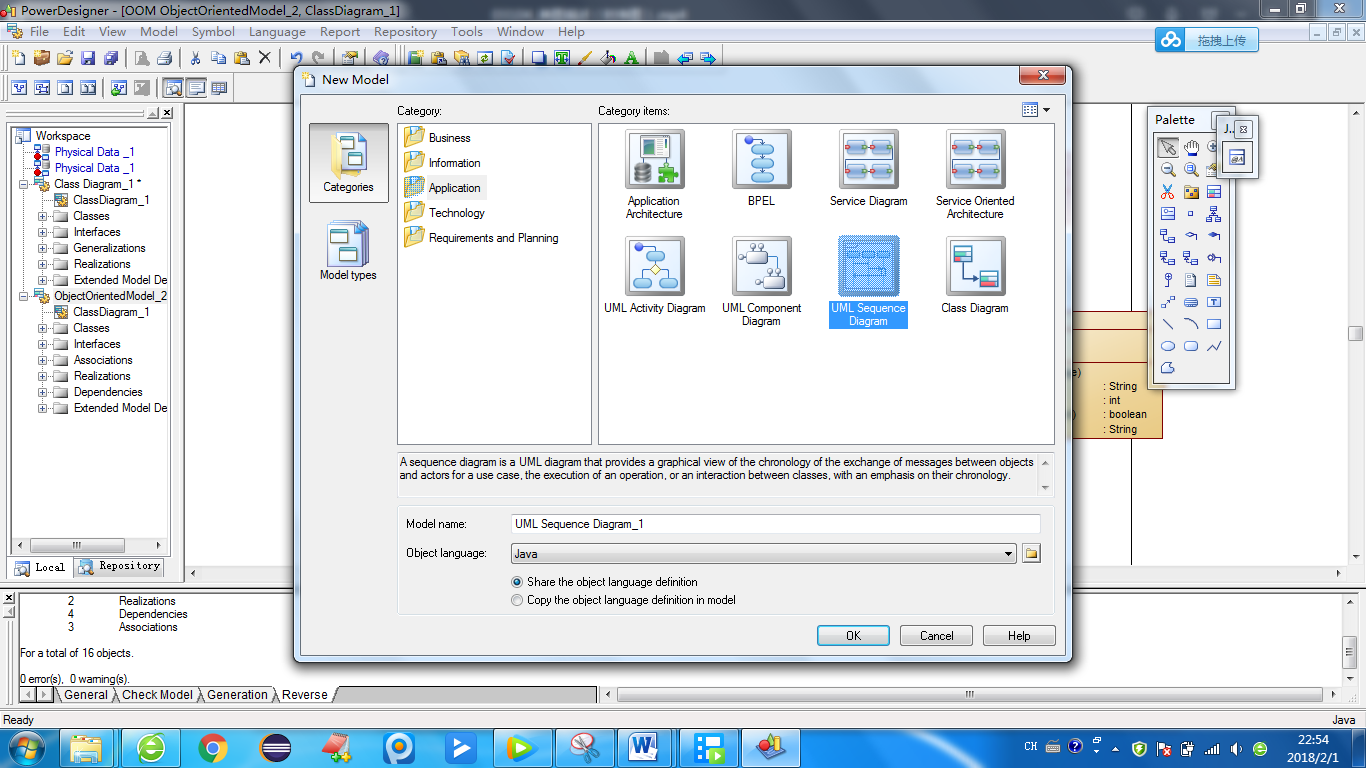
由代码生成图







时序图新建





**Eclipse基本使用**

在2004年之前在整个行业内有两套开发的架构：

·解释顺序：操作系统+数据库+中间件+开发工具

·皇家级：aix+DB2+WAS+WSAD;

·杂牌级：UNIX/LINUX + ORACLE +BEA WebLogic +JBuilder；

在2006年开始出现了一下的一套架构：

·免费级：Linux+MySql+Tomact +Eclipse

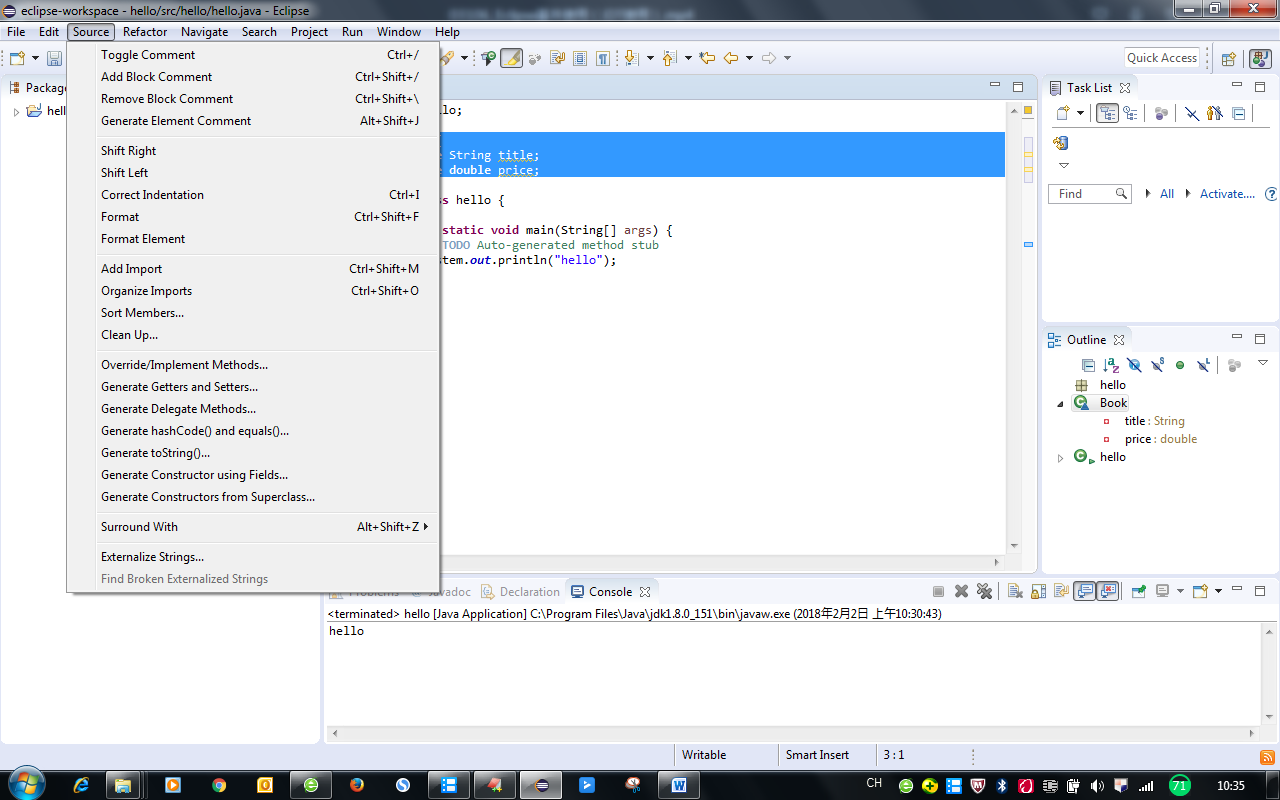
Eclipse包含一下的几个部分：JDT、JUNIT、CVS客户端、插件开发、GIT客户端。

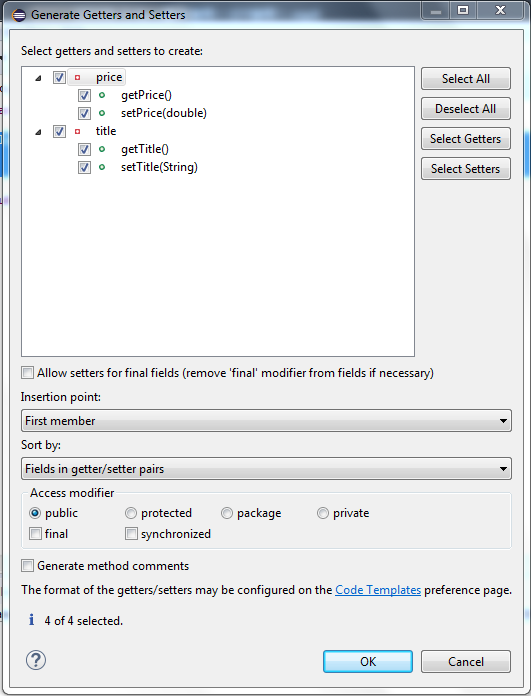
Eclipse的所有配置都是以工作区为主的，也就是说每一个工作区都有自己独立的配置

**3.2、JDT的使用**

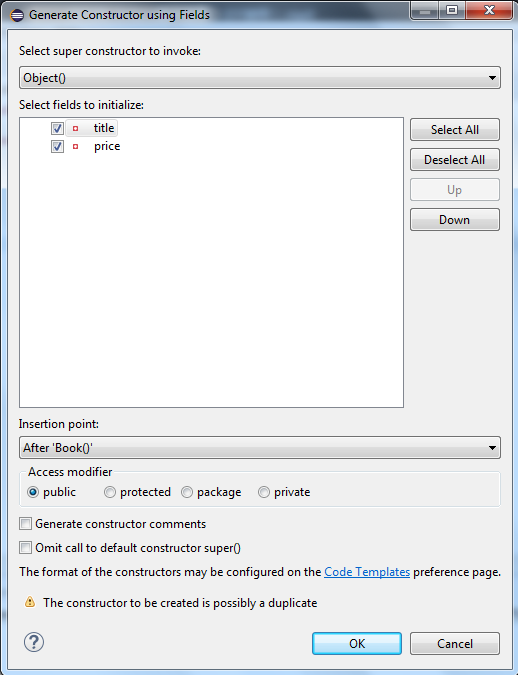
|  |
| --- |
| **class** Book{  **private** String title;  **privatedouble**price;  } |

如果此时要编写一个简单Java类，则需要生成构造，getter，setter等方法。

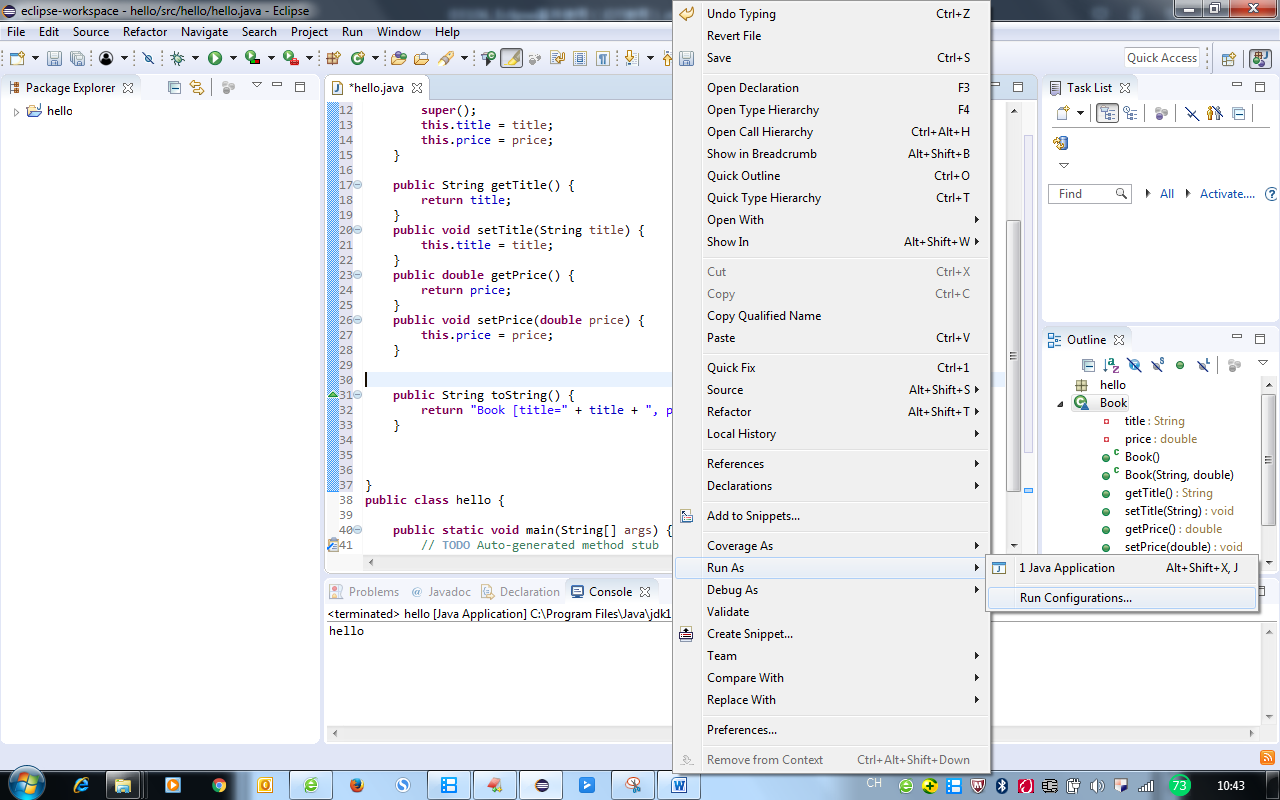


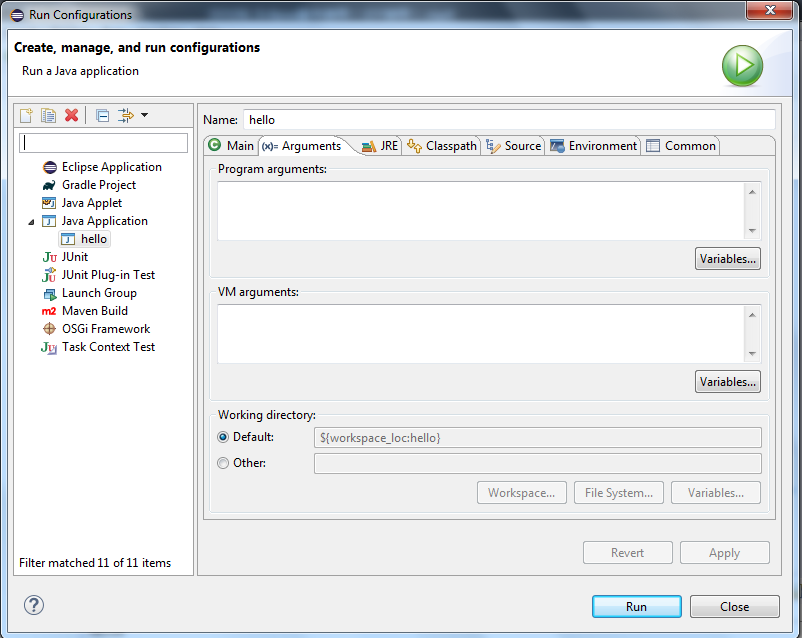


自动生成构造方法



在E陈丽萍色里面如果要想配置程序的初始化参数会比较麻烦。





Eclipse快捷键：

·ALT+ /：进行代码的提示（sysout、man）；

·CTRL +1:为错误的代码给出纠正方案；

·CTRL + SHOFT +O：组织导入，导入其他包的类；

·CTRL+D：删除当前行代码；

·CTRL + /：使用单行注释；

·CTRL + H ：强力搜索；

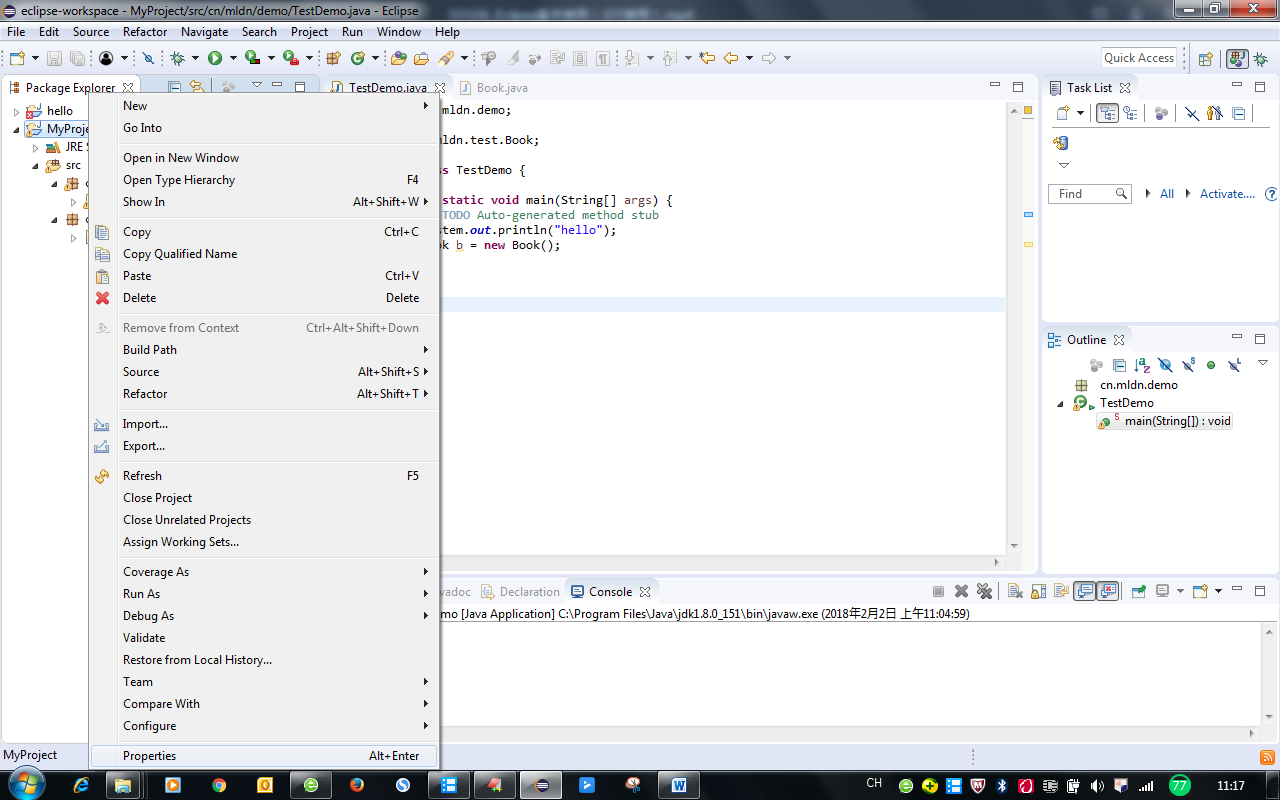
·CTRL + SHIFT + ↓：复制当前行代码

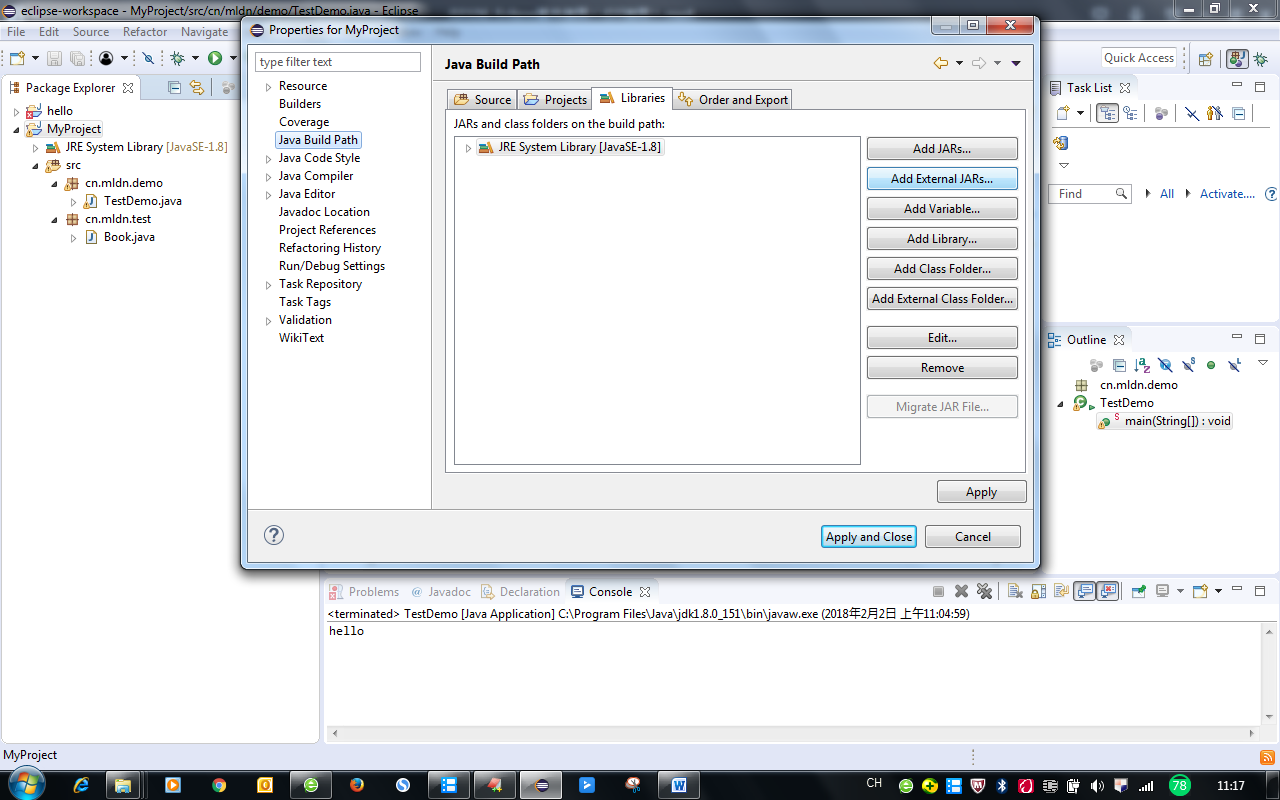
·CTRL + SHIFT + L；全部快捷键列表

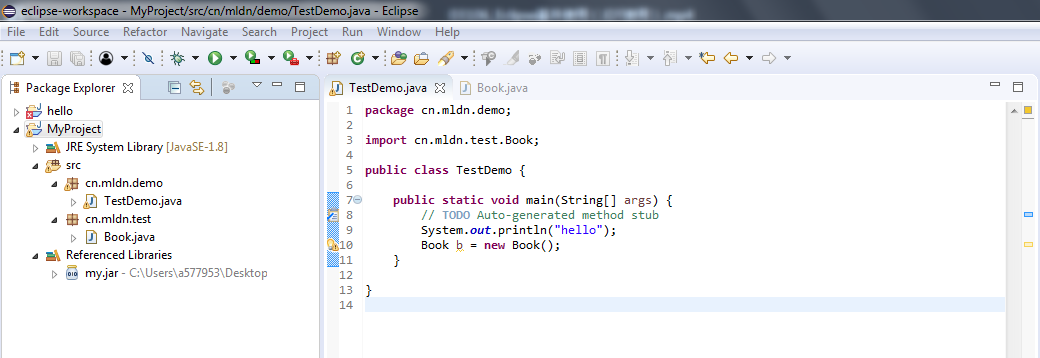
在Eclipse里面还提供有debug功能

有时候Eclipse打包的jar文件无法使用，建议还是使用jat命令进行打包。

如果Eclipse的项目要想使用一个jar文件，那么配置CLASSPATH无效。只能够根据羡慕属性进行配置。







3.3、junit的使用（重点）

Junit是一种测试工具。软件测试而言分为两种测试：

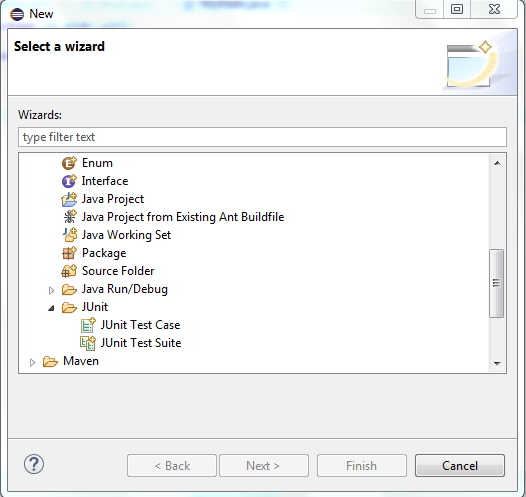
·黑盒测试：针对于功能测试，看不见代码。

·白盒测试：针对性能进行测试，算法的调整。

Junit就是一个use case（用例测试）的测试工具

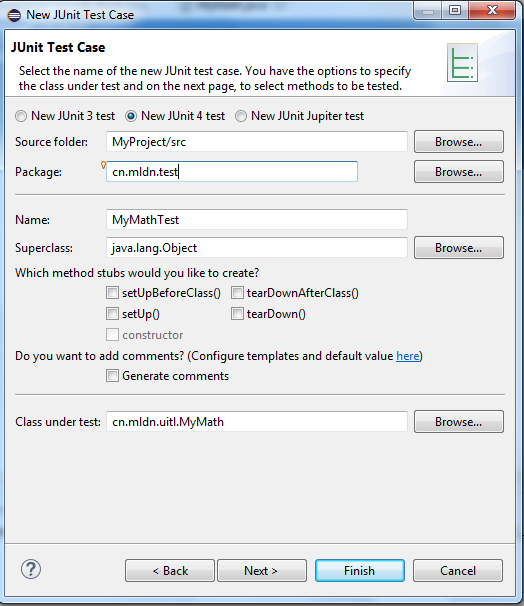
选中这个类：

|  |
| --- |
| **package** cn.mldn.uitl;  **publicclass** MyMath {  **publicstaticint** div(**int**x, **int**y ) **throws** Exception{  **int**temp = 0;  temp = x/y;  **return**temp;  }  } |

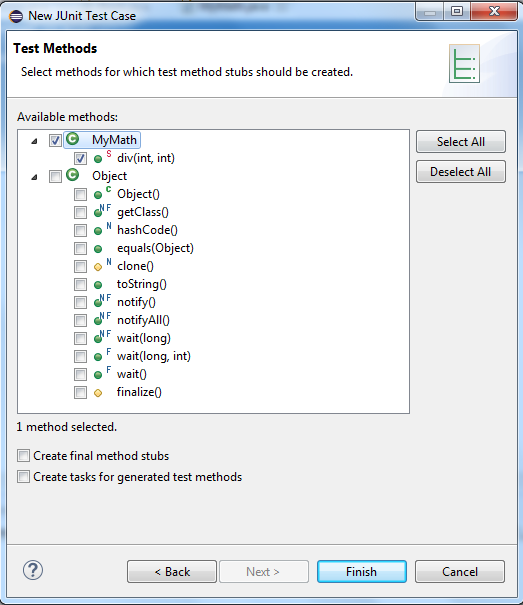


Test case：一个测试用例;

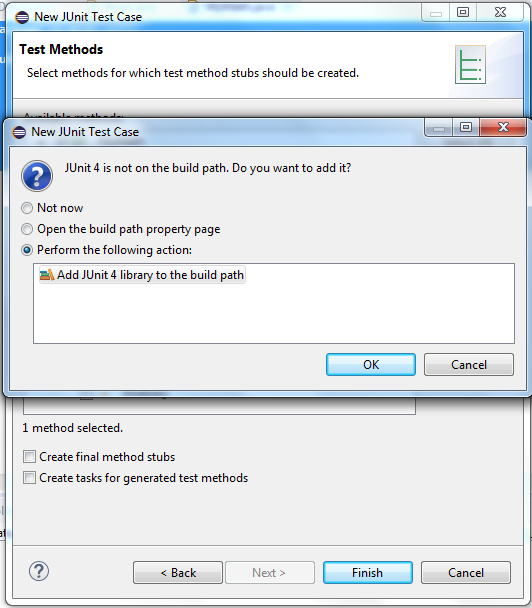
Test Suite:一组测试用例；



针对div方法进行测试：



Junit是一个第三方的开发包，所以使用前需要导入此包



|  |
| --- |
| package cn.mldn.test;  import static org.junit.Assert.\*;  import org.junit.Test;  import cn.mldn.uitl.MyMath;  import junit.framework.TestCase;  public class MyMathTest {  @Test  public void testDiv() {  try {  TestCase.assertEquals(MyMath.div(10, 2), 5);  } catch (Exception e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  } |

对于junit的测试结果：

·Green Bar：测试通过；

·Read Bar：测试失败；

**可变参数**

如果说现在要求设计一个方法，这个方法可以接受任意多个整形数据的相加。

范例：最初的解决方案；

|  |
| --- |
| **package** cn.mldn.demo;  **public class** TestDemo {  **publicstaticvoid** main(String[] args) {  // **TODO** Auto-generated method stub  System.***out***.println(*add*(**newint**[]{1,2,3}));  System.***out***.println(*add*(**newint**[]{10,20}));  }  /\*\*  \* 实现任意多个整形数据的相加操作  \* **@param** data 由于要接受多个整形数据，所以使用数组完成接受  \* **@return**多个整形数据的累加结果  \*/  **publicstaticint** add(**int**[] data) {  **int**sum= 0 ;  **for**(**int**i = 0 ; i<data.length;i++){  sum += data[i];  }  **return**sum;    }  } |

以上的diamante之所以使用数组，是因为多个参数方法上无法描述，所以利用数组整合多个参数，但是严格来讲这样的实现并不标准。要求是可以接受任意多个整形数据：

理想的调用形式：add(1,2,3)、add(10,20)、add(10,10,10,20,20,20,…)；

这一功能从JDK1.5之后正式的登录到了Java之中，它主要实在方法上使用，其定义形式:

|  |
| --- |
| [public | protected |private] [static] [final]  [abstract] 返回值类型方法名称(参数类型...变量){  [return [返回值];]  } |

此时诶出的参数不在是一个内容，而是多个内容，但是尽管参数的定义形式变了，可是参数的访问却没有改变，也是就是说在进行参数的访问时候按照数组进行操作。

|  |
| --- |
| **package** cn.mldn.demo;  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  // **TODO** Auto-generated method stub  System.***out***.println(*add*(**newint**[]{1,2,3}));  System.***out***.println(*add*(**newint**[]{10,20}));    System.***out***.println(*add*(1,2,3));  System.***out***.println(*add*(10,20));  System.***out***.println(*add*());  }  /\*\*  \* 实现任意多个整形数据的相加操作  \* **@param** data 由于要接受多个整形数据，所以使用数组完成接受  \* **@return**多个整形数据的累加结果  \*/  **publicstaticint** add(**int** ... data) {  **int**sum= 0 ;  **for**(**int**i = 0 ; i<data.length;i++){  sum += data[i];  }  **return**sum;    }  } |

在大部分的开发情况下，应该要求参数的个数是准确的，所以对于这样的开发往往不会用于应用型的开发上，可能用于一些程序相关系统类的设计使用上。

**Foreach循环**

有人会认为以上的输出需要使用索引会比较麻烦。

从JDK1.5之后增加的foreach循环形式就可以取消掉索引的操作形式，语法如下：

|  |
| --- |
| for(类型 变量：数组|集合){  //每一次循环会自动的将数组的内容设置给变量  } |

范例：观察增强型的for循环

|  |
| --- |
| **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  // **TODO** Auto-generated method stub  **int**data[] = **newint**[] {1,2,3,4,5,6};  **for**(**int**x : data) {  System.***out***.println(x);  }  }  } |

**静态导入**

如果说现在某一个类重定义的方法全部都属于static型的方法，那么其他类要引用此类事必须要使用“类名称.方法”进行使用。

|  |
| --- |
| **package** cn.mldn.uitl;  **publicclass** MyMath {  **publicstaticint** div(**int**x, **int**y ) **throws** Exception{  **int**temp = 0;  temp = x/y;  **return**temp;  }  **publicstaticint** add(**int**x ,**int**y ) {  **return**x + y;  }  } |

其他类使用：

|  |
| --- |
| **publicstaticvoid** main(String[] args) {  // **TODO** Auto-generated method stub  System.***out***.println("add : " + MyMath.*add*(10, 20));  **try** {  System.***out***.println("div : " + MyMath.*div*(10, 2));  } **catch** (Exception e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  } |

如果在主类中定义的是static方法，那么可以直接调用static方法。而现在的MyMath类里面都是static方法，那么觉得前面加上类名称就多余，于是从JDK1.5之后开始增加了静态导入

|  |
| --- |
| **package** cn.mldn.demo;  //将MyMath类中的全部static方法导入，这些方法就好比在主类中定义的static方法一样  **importstatic** cn.mldn.uitl.MyMath.\*;  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  // **TODO** Auto-generated method stub  //直接使用方法名称进行访问  System.***out***.println("add : " + *add*(10, 20));  **try** {  System.***out***.println("div : " + *div*(10, 2));  } **catch** (Exception e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  } |

**泛型**

**3.1、泛型的引出**

现在要求定义一个表示坐标的操作类（Point），在这个类里面要求保存有以下几种坐标：

·保存数字：X =10、 y=20；

·保存小数：x = 10.2、y=20.3；

·保存字符串：x=东经20度、y=北纬15度。

现在这个Point类设计的关键就在于x于y这两个变量的类型设计上.。必须有一种类型可以保存一张三种数据类型，那么首先想到的是Object类型

·int：int自动装箱为Integer，Integer向上转型为Object；

·double：double自动装箱为Double，Double向上转型为Object；

·String：String直接向上转型为Object；

范例：代码的初期设计如下：

|  |
| --- |
| **class** Point{  **private** Object x;  **private** Object y;  **publicvoid** setX(Object x) {  **this**.x = x;  }  **public** Object getX() {  **return**x;  }  **publicvoid** setY(Object y) {  **this**.y = y;  }  **public** Object getY() {  **return**y;  }  } |

下面重复的演示三个程序，分别使用不同的数据类型。

范例：在Point类里面保存整型数据

|  |
| --- |
| **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  // **TODO** Auto-generated method stub  Point p = **new** Point();  p.setX(10);  p.setY(20);  **int**x = (Integer)p.getX();  **int**y = (Integer)p.getY();  System.***out***.println("x : " + x + ",y : " + y);  }  } |

范例：使用小数Double

|  |
| --- |
| **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  // **TODO** Auto-generated method stub  Point p = **new** Point();  p.setX(10.8);  p.setY(20.3);  **double**x = (Double)p.getX();  **double**y = (Double)p.getY();  System.***out***.println("x : " + x + ",y : " + y);  }  } |

范例：使用String型

|  |
| --- |
| **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  // **TODO** Auto-generated method stub  Point p = **new** Point();  p.setX("东经100度");  p.setY("北纬20度");  String x = (String)p.getX();  String y = (String)p.getY();  System.***out***.println("x : " + x + " ，y : " + y);  }  } |

此时的代码已经利用了Object数据类型解决了一切的开发问题，可是解决的关键靠的是Object，失败的关键也是Object。

范例：错误的程序

|  |
| --- |
| **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  // **TODO** Auto-generated method stub  Point p = **new** Point();  p.setX("东经100度");  p.setY(10);  String x = (String)p.getX();  String y = (String)p.getY();  System.***out***.println("x : " + x + " ，y : " + y);  }  } |

存的时候是存的Integer，取得时候使用的是String，两个没有任何关系的类之间要发生强制的类型转换，就一定会发生，ClassCastException

|  |
| --- |
| Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String  at cn.mldn.demo.TestDemo.main(TestDemo.java:28) |

向上转型的核心目的在于统一操作的参数上，而向下转型的目的是操作子类定义的特殊功能，可是现在问题我们发现向下转型时意见非常不安全的操作，那么这一操作应该在代码运行之前就已经能够自动的排查出来是最好的选择。但是之前的数据做不到

从JDK1.5开始增加了泛型技术，而泛型技术的核心意义在于：类在定义的时候，可以使用一个标记，次标记就表示类中属性或方法参数的类型标记，在使用的时候才动态的设置类型。

|  |
| --- |
| //此时设置的T在Point类定义上只表示一个标记，在使用的时候需要为其设置具体的类型  **class** Point<T>{//Type = T,是有一个类型  **private** T x;//此属性的类型不知道，有Point类使用时动态的决定  **private** T y;//此属性的类型不知道，有Point类使用时动态的决定  **publicvoid** setX(T x) {  **this**.x = x;  }  **publicvoid** setY(T y) {  **this**.y = y;  }  **public** T getX() {  **return**x;  }  **public** T getY() {  **return**y;  }  } |

在使用Point类的时候才取设置标记的内容，也就是设置了类中的属性额类型

范例：设置String

|  |
| --- |
| **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  // **TODO** Auto-generated method stub  Point<String>p = **new** Point<String>();  p.setX("东经100度");  //如果设置的数据是错误的，我们在编译的时候就自动排查了  p.setY("北纬20度");  //由于接受的类型就是String，所以不需要向下转型了  String x = p.getX();  String y = p.getY();  System.***out***.println("x : " + x + " ，y : " + y);  }  } |

返现使用了泛型之后，所有类中属性的类型都是动态设置的，而所有使用泛型标记的方法参数类型也都繁盛改变没这样就相当于避免了向下转型的问题，从而解决了类转换的安全隐患。

但是需要特别说明的是，如果要想使用泛型，那么他能够采用的类型只能够是类，即，不能是基本类型，只能够是引用类型。

对于泛型有两点说明：

·如果在使用泛型或者是接口的时候，没有设置泛型具体类型，那么会出现变异时的警告，同时为了保证程序不出错，所有的泛型就将使用Object表示；

·从JDK1.7开始可以简化声明泛型

|  |
| --- |
| **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  // **TODO** Auto-generated method stub  Point<String> p = **new** Point<>();//实例化的泛型可以省略  p.setX("东经100度");  //如果设置的数据是错误的，我们在编译的时候就自动排查了  p.setY("北纬20度");  //由于接受的类型就是String，所以不需要向下转型了  String x = p.getX();  String y = p.getY();  System.***out***.println("x : " + x + " ，y : " + y);  }  } |

**3.2、通配符**

观察如下程序

|  |
| --- |
| **package** cn.mldn.demo;  **class** Message<T>{  **private** T msg;  **publicvoid** setMsg(T msg) {  **this**.msg = msg;  }  **public** T getMsg() {  **return**msg;  }  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  Message<String>m = **new** Message<String>();  m.setMsg("Hello");  *fun*(m);  }  **publicstaticvoid** fun(Message<String>temp) {  System.***out***.println(temp.getMsg());  }  } |

以上的代码为Messgae类设置的泛型类型，如果设置的是其他类型呢

Fun()方法里面接受的“Message<String>”那么就不能够使用了，并且fun()方法不能够针对于不同的泛型进行重载，因为方法的重载认的只是参数的类型，于泛型不管。

解决方法一：不设置方法参数的泛型

|  |
| --- |
| **package** cn.mldn.demo;  **class** Message<T>{  **private** T msg;  **publicvoid** setMsg(T msg) {  **this**.msg = msg;  }  **public** T getMsg() {  **return**msg;  }  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  Message<String>m1 = **new** Message<String>();  Message<Integer>m2 = **new** Message<Integer>();  m1.setMsg("Hello");  m2.setMsg(10);  *fun*(m1);  *fun*(m2);  }  **publicstaticvoid** fun(Messagetemp) {  System.***out***.println(temp.getMsg());  }  } |

此时在fun防范上依然存在警告信息，因为只要不设置泛型，就一定会存在警告信息。可能存在的问题：

|  |
| --- |
| **package** cn.mldn.demo;  **class** Message<T>{  **private** T msg;  **publicvoid** setMsg(T msg) {  **this**.msg = msg;  }  **public** T getMsg() {  **return**msg;  }  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  Message<Integer>m2 = **new** Message<Integer>();  m2.setMsg(10);  *fun*(m2);  }  **publicstaticvoid** fun(Messagetemp) {//如果不设置泛型就是Object  temp.setMsg("hello");//设置为String型  System.***out***.println(temp.getMsg());  }  } |

所以我们现在最应该解决的是：需要有一种方式可以接受一个类的任意的泛型类型，但是不可以修改只能够取出，那么就可以使用“?”来描述。

|  |
| --- |
| **package** cn.mldn.demo;  **class** Message<T>{  **private** T msg;  **publicvoid** setMsg(T msg) {  **this**.msg = msg;  }  **public** T getMsg() {  **return**msg;  }  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  Message<Integer>m2 = **new** Message<Integer>();  m2.setMsg(10);  *fun*(m2);  }  **publicstaticvoid** fun(Message<?>temp) {//不能够设置但是可以取出  System.***out***.println(temp.getMsg());  }  } |

在“?”通配符基础上还会有两个子的通配符：

·？extends 类：设置泛型上限，可以在声明上和方法参数上使用；

|-？extends Number :意味着可以设置Number或者Number的子类（Integer、double…）

·？Super 类：设置泛型下线，方法参数上使用；

|-? Super String:意味着只能够设置String或者是他的父类Object。

范例：设置泛型的上限

|  |
| --- |
| **package** cn.mldn.demo;  **class** Message<T **extends**Number>{  **private** T msg;  **publicvoid** setMsg(T msg) {  **this**.msg = msg;  }  **public** T getMsg() {  **return**msg;  }  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  Message<Integer>m2 = **new** Message<Integer>();  m2.setMsg(10);  *fun*(m2);  }  **publicstaticvoid** fun(Message<? **extends**Number>temp) {//不能够设置但是可以取出  System.***out***.println(temp.getMsg());  }  } |

如果设置了非Number或者子类时，会编译出错

范例：设置泛型的下限

|  |
| --- |
| **package** cn.mldn.demo;  **class** Message<T>{  **private** T msg;  **publicvoid** setMsg(T msg) {  **this**.msg = msg;  }  **public** T getMsg() {  **return**msg;  }  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  Message<String>m2 = **new** Message<String>();  m2.setMsg("hello");  *fun*(m2);  }  **publicstaticvoid** fun(Message<? **super** String>temp) {//不能够设置但是可以取出  System.***out***.println(temp.getMsg());  }  } |

3.3、泛型接口

**如果是接口在前面加上字母I，例如：IMessage；**

**如果是抽象类在前面加上Abstract，例如：AbstractMessage；**

**如果是普通类直接编写，例如：Message；**

范例：定义一个泛型接口

|  |
| --- |
| **interface** IMessage<T>{  **publicvoid** print(T t);  } |

在接口上必须要定义其相应的子类，所以来说要定义子类有两种形式：

形式一：在子类继续设置泛型：

|  |
| --- |
| **class**MessageImpl<T>**implements** IMessage<T>{  **publicvoid** print(T t) {  System.***out***.println(t);    }    }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  IMessage<String>msg = **new**MessageImpl<String>();  msg.print("hello");  }  } |

形式二：在子类不设置泛型，而为父类口明确的定义一个泛型类型：

|  |
| --- |
| **class** MessageImpl **implements** IMessage<String>{  **publicvoid** print(String t) {  System.***out***.println(t);  }  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  IMessage<String>msg = **new** MessageImpl();  msg.print("hello");  }  } |

**3.4、泛型方法**

泛型方法不一定非要定义在支持泛型的类里面。在之前所编写的所有存在有泛型的方法都是在泛型支持类里面定义的。

范例：泛型方法定义

|  |
| --- |
| **package** cn.mldn.demo;  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  String str = *fun*("Hello");  System.***out***.println(str.length());  }  **publicstatic**<T> T fun(T t){  **return**t;  }  } |

能够看懂泛型方法的标记就行了。

1. 泛型解决的死向下转型所带来的安全隐患，其核心的组成就是在声明类或者接口的时候不设置参数或属性的类型；
2. “?”可以接受任意的泛型类型，只能够取出，不能够修改。

**枚举**

多例设计模式：

构造方法私有化，而后在类的内部提供有若干个实例化对象，并且通过static方法返回

范例：定义一个颜色基色的多例

|  |
| --- |
| **package** cn.mldn.demo;  **class** Color{  **private** String title;  **privatestaticfinal** Color ***RED*** = **new** Color("红色");  **privatestaticfinal** Color ***GREEN*** = **new** Color("绿色");  **privatestaticfinal** Color ***BULE*** = **new** Color("蓝色");  **private** Color(String title){  **this**.title = title;  }  **publicstatic** Color getInstance(**int**ch){  **switch** (ch){  **case** 1:  **return*RED***;  **case** 2:  **return*GREEN***;  **case** 3:  **return*BULE***;  **default**:  **returnnull**;  }  }    **public** String toString(){  **returnthis**.title;  }  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  Color red = Color.*getInstance*(1);  System.***out***.println(red);  }  } |

枚举严格来讲各个语言都支持（除了2005年之前的Java），也就是说在2005年之前，要是想定义枚举都会采用如上的代码形式问题完成。

从2005年之后，增加了枚举的概念，同事设置了一个新的enum的关键字。

范例：定义枚举

|  |
| --- |
| **package** cn.mldn.demo;  **enum** Color{//定义好了枚举  ***RED***,***GREEN***,***BLUE***;  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  Color red = Color.***RED***;  System.***out***.println(red);  }  } |

可以发现使用枚举之后可以完全简化的替代多例设计模式。

但是需要说明的是，严格来讲枚举并不是一个新的功能，在JAVA里面虽然使用了enum关键字定义了枚举，但是使用enum定义的枚举就相当于是一个继承了Enum类而已。

|  |
| --- |
| public abstract class Enum<E extends Enum<E>>  extends Object  implements Comparable<E>, Serializable |

Enum是一个抽象类。里面的构造方法如下：

|  |
| --- |
| protected Enum(String name,  int ordinal) |

Enum类的构造方法依然是被封装的，所以也属于构造方法私有化的应用范畴，所有的多例设计模式前提：构造方法私有化。

在Enum类里面定义了两个方法：

·取得枚举的索引：public final int ordinal()

·取得枚举的名字：public final String name()

除了以上支持的方法之外，使用enum关键字定义的枚举类里面还有一个values()方法，这个方法将枚举对象以对象数组的形式全部返回。

|  |
| --- |
| **package** cn.mldn.demo;  **enum** Color{//定义好了枚举  ***RED***,***GREEN***,***BLUE***;  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  **for**(Color c :Color.*values*()){  System.***out***.println(c.ordinal() + " - " + c.name());  }  }  } |

面试题：请解释enum和Enum的区别？

·enum是一个关键字，而Enum是一个抽象类；

·使用enum定义的枚举就相当于一个类继承了Enum这个抽象类。

**3.2、定义其他结构**

·枚举中定义的方法不能够使用public声明；

·枚举对象必须放在首行，随后才可以定义属性，构造，普通方法。

范例：扩展枚举功能

|  |
| --- |
| **package** cn.mldn.demo;  **enum** Color{  ***RED***("红色"),***GREEN***("绿色"),***BLUE***("蓝色");  **private** String title;  **private** Color(String title) {  **this**.title = title;  }  **public**String toString() {  **returnthis**.title;  }  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  **for**(Color c: Color.*values*()) {  System.***out***.println(c);  }  }  } |

此时与之前定义的多例设计模式操作方式就完全相同，而且代码更加的简单。

枚举还可以实现接口

范例：枚举实现接口

|  |
| --- |
| **package** cn.mldn.demo;  **interface** Message{  **publicabstract** String getTitle();  }  **enum** Color **implements** Message{  ***RED***("红色"),***GREEN***("绿色"),***BLUE***("蓝色");  **private** String title;  **private** Color(String title) {  **this**.title = title;  }  **public** String toString() {  **returnthis**.title;  }  **public** String getTitle() {  **returnthis**.title;  }  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  Message msg = Color.***RED***;  System.***out***.println(msg.getTitle());  }  } |

枚举里面每一个对象后面使用匿名内部类的形式实现抽象方法。

范例：实现接口的另一个方式

|  |
| --- |
| **package** cn.mldn.demo;  **interface** Message{  **publicabstract** String getTitle();  }  **enum** Color **implements** Message{  ***RED***("红色"){  **public** String getTitle() {  **return**"自己的" + **this**;  }  },***GREEN***("绿色"){  **public** String getTitle() {  **return**"自己的" + **this**;  }  },***BLUE***("蓝色"){  **public** String getTitle() {  **return**"自己的" + **this**;  }  };  **private** String title;  **private** Color(String title) {  **this**.title = title;  }  **public** String toString() {  **returnthis**.title;  }  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  Message msg = Color.***RED***;  System.***out***.println(msg.getTitle());  }  } |

在枚举里面可以直接定义抽象方法。此时每一个枚举对象必须分别覆写抽象方法

范例：定义抽象方法并且覆写

|  |
| --- |
| **package** cn.mldn.demo;  **enum** Color{  ***RED***("红色"){  **public** String getTitle() {  **return**"自己的" + **this**;  }  },***GREEN***("绿色"){  **public** String getTitle() {  **return**"自己的" + **this**;  }  },***BLUE***("蓝色"){  **public** String getTitle() {  **return**"自己的" + **this**;  }  };  **private** String title;  **private** Color(String title) {  **this**.title = title;  }  **public**String toString() {  **returnthis**.title;  }  **publicabstract** String getTitle();  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  System.***out***.println(Color.***RED***.getTitle());  }  } |

**3.3、枚举的实际作用**

枚举可以在swith上使用

|  |
| --- |
| **package** cn.mldn.demo;  **enum** Color{  ***RED***,***GREEN***,***BLUE***;  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  Color c = Color.***RED***;  **switch**(c) {  **case*RED***:  System.***out***.println("红色");  **break**;  **case*GREEN***:  System.***out***.println("绿色");  **break**;  **case*BLUE***:  System.***out***.println("蓝色");  **break**;  **default** :  System.***out***.println("null");  }  }  } |

|  |
| --- |
| **package** cn.mldn.demo;  **enum** Sex{  ***MALE***("男"),***FEMALE***("女");  **private** String title;  **private** Sex(String title) {  **this**.title = title;  }  **public** String toString() {  **returnthis**.title;  }  }  **class** Person{  **private** String name;  **privateint**age;  **private** Sex sex;  **public** Person(String name,**int**age,Sex sex) {  **this**.name = name;  **this**.age = age;  **this**.sex = sex;  }  **public** String toString() {  **return**"name : " + **this**.name + " , age ； " + **this**.age + " , sex ：" + **this**.sex;  }  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  System.***out***.println(**new** Person("aaa",10,Sex.***FEMALE***));  }  } |

**Annotation**

对于软件程序的开发实际上经过了三个发展过程：

·第一个过程：将所有与配置相关的内容直接写到代码之中；

·第二个过程：将配置与程序代码独立，即：程序运行的时候根据配置文件进行操作；

|-最严重的问题：一个项目里面配置文件过多，根本就无法查询错误；

·第三个过程：配置信息对用户而言无用，而且胡乱的修改还会导致程序的错误。可以将配置信息写回到程序里面，但是利用一些明显的标记来区分配置信息和程序。

Annotation是JDK1.5最大的特色，利用注解的形式来实现程序的不同功能实现。

在Java SE里面支持自定义Annotation开发，并且提供了三个最为常用的基础Annotation：@Override、@Deprecated、@Suppress Warnings。

**3.1、准确覆写@Override**

想要覆写toString但是写成了tostring，这样的错误无法再编译的时候发现，只能在测试的时候发现。

所以此时为了告诉编译器，tostring()应该是覆写的方法，那么就可以加上“@Override”，明确的告诉编译器这个方法应该是覆写来的，如果不是要报错。

|  |
| --- |
| **package** cn.mldn.demo;  **enum** Sex{  ***MALE***("男"),***FEMALE***("女");  **private** String title;  **private** Sex(String title) {  **this**.title = title;  }  **public** String toString() {  **returnthis**.title;  }  }  **class** Person{  **private** String name;  **privateint**age;  **private** Sex sex;  **public** Person(String name,**int**age,Sex sex) {  **this**.name = name;  **this**.age = age;  **this**.sex = sex;  }  @Override  **public** String toString() {  **return**"name : " + **this**.name + " , age ； " + **this**.age + " , sex ：" + **this**.sex;  }  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  System.***out***.println(**new** Person("aaa",10,Sex.***FEMALE***));  }  } |

**3.2、声明过期操作@Deprecated**

范例：申明过期操作

|  |
| --- |
| **package** cn.mldn.demo;  **class** Book{  @Deprecated  **publicvoid**~~fun~~() {}  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  Book b = **new** Book();  b.~~fun~~();  }  } |

**3.3、压制警告 @Suppress Warnings**

范例：观察警告：

|  |
| --- |
| **package** cn.mldn.demo;  **class** Book<T>{  **private** T title;  **publicvoid** setTitle(T title) {  **this**.title = title;  }  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  Book b = **new**Book();  b.setTitle("Hello");  }  } |

那么如果说现在是开发者故意留下的警告信息，但是又不希望其总是提示警告。就可以选择压制警告。

范例：压制警告

|  |
| --- |
| **package** cn.mldn.demo;  **class** Book<T>{  **private** T title ;  **publicvoid** setTitle(T title) {  **this**.title = title;  }  **public** T getTitle() {  **return**title;  }  }  **publicclass** TestDemo {  @SuppressWarnings({ "rawtypes", "unchecked" })  **publicstaticvoid** main(String[] args) {  Book b = **new** Book();  b.setTitle("Hello");  }  } |

现在相当于取消了警告的提示信息

**接口定义增强**

如果说现在有某一个接口，这个接口随着时间的发展已经产生了2W个子类。突然有一天发现，这个几口里面的方法不足，应该在增加一个方法，而针对于所有不同的子类，这个方法的功能实现是完全一样的。按照最初的做法，应该在每一个子类上都覆写这个新的方法，那么就要修改2w个子类。

所以为了解决的问题，允许在接口里面定义普通方法了，但是如果要定义普通方法就必须明确的使用default来进行定义了。

|  |
| --- |
| **package** cn.mldn.demo;  **interface** IMessage{  **publicabstractvoid** print();//这是一个借口里面原本定义的方法  **defaultvoid** fun(){//在接口里面定义一个普通的方法  System.***out***.println("goodbye");  }  }  **class** MessageImpl **implements** IMessage{  **publicvoid** print(){  System.***out***.println("hello world!");  }  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  IMessage msg = **new** MessageImpl();  msg.fun();//此方法是在接口中直接定义的  }  } |

除了使用default定义方法以外，还可以使用static定义方法，一旦使用了static定义了方法以后，就意味着这个方法只能使用类名称调用

范例：定义static方法

|  |
| --- |
| **package** cn.mldn.demo;  **interface** IMessage{  **publicabstractvoid** print();//这是一个借口里面原本定义的方法  **defaultvoid** fun(){//在接口里面定义一个普通的方法  System.***out***.println("goodbye");  }  **staticvoid** get(){  System.***out***.println("直接有接口调用");  }  }  **class** MessageImpl **implements** IMessage{  **publicvoid** print(){  System.***out***.println("hello world!");  }  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  IMessage msg = **new** MessageImpl();  msg.fun();//此方法是在接口中直接定义的  IMessage.*get*();  }  } |

在JDK1.8里面有一个最重要的概念：内部类访问方法参数的时候可以不加上final关键字

所有出现的这些新特性，完全打破了java已有的代码组成

**Lamda表达式**

Lamda属于函数式变成的概念，那么为什么需要函数式的变成呢？

如果要想清楚函数式编程的产生目的，那么必须通过匿名内部类来分析。

范例：传统的匿名内部类

|  |
| --- |
| **package** cn.mldn.demo;  **interface** IMessage{  **publicabstractvoid** print();//这是一个借口里面原本定义的方法  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  *fun*(**new** IMessage(){  **publicvoid** print(){  System.***out***.println("hello");  }  });  }  **publicstaticvoid** fun(IMessage msg){  msg.print();  }  } |

实际上整个代码之中，如果是fun（）方法，最终需要的只是一个输出而已，但是由于java的开发的结构性完成性的要求，搜易不得不在这个核心的语句上嵌套更多的内容。

以上的做法要求的实在过于严谨了，所以在JDK1.8时代引入了函数式的编程，可以简化以上代码

范例：使用Lamda表达式

|  |
| --- |
| **package** cn.mldn.demo;  **interface** IMessage{  **publicabstractvoid** print();//这是一个借口里面原本定义的方法  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  *fun*(()-> System.***out***.println("hello"));  }  **publicstaticvoid** fun(IMessage msg){  msg.print();  }  } |

整个操作里面匿名内部类只是进行一行语句的输出，所以此时使用了Lamda表达式可以非常轻松的实现输出要求。

对于Lamda语法有三种形式：

·（参数）->单行语句；

·（参数）->{单行语句}；

·（参数）->表达式；

范例：观察有参的单行语句

|  |
| --- |
| **package** cn.mldn.demo;  **interface** IMessage{  **publicabstractvoid** print(String str);//这是一个借口里面原本定义的方法  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  //首先要定义出此表达式里面需要接收的变量，单行语句直接进行输出  *fun*((s)-> System.***out***.println(s));  }  **publicstaticvoid** fun(IMessage msg){  msg.print("Hello");//设置参数的内容  }  } |

范例：编写多行语句

|  |
| --- |
| **package** cn.mldn.demo;  **interface** IMessage{  **publicabstractvoid** print(String str);//这是一个借口里面原本定义的方法  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  //首先要定义出此表达式里面需要接收的变量，单行语句直接进行输出  *fun*((s)-> {  s = s.toUpperCase();  System.***out***.println(s);});  }  **publicstaticvoid** fun(IMessage msg){  msg.print("Hello");//设置参数的内容  }  } |

如果说现在代码里面只是一个简答的计算表达式，那么操作也很容易

范例：编写一个表达式

|  |
| --- |
| **package** cn.mldn.demo;  **interface** IMessage{  **publicabstractint** add(**int**x ,**int**y);//这是一个借口里面原本定义的方法  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  //首先要定义出此表达式里面需要接收的变量，单行语句直接进行输出  *fun*((s1,s2)->s1 + s2);  }  **publicstaticvoid** fun(IMessage msg){  System.***out***.println(msg.add(10,20));  }  } |

如果要写ruturn，如下

|  |
| --- |
| fun((s1,s2)-> {return s1 + s2;}); |

如果现在只是一个表达式，那么进行操作的返回，还是不写return比较合适。是多行的时候才可以考虑写上return

利用lamda表达式最终解决的问题：避免了匿名内部类定义过多无用的操作。

**方法引用**

一直以来都是知识在对象上能否发现引用的身影，而对象引用的特点：不同的对象可以操作同一块内容。而所谓的方法引用就是指为一个方法设置别名，相当于一个方法定义了不同的名字。

方法引用在java8之中一共定义了四种形式：

·引用静态方法：类名称 ::static方法名称

·引用某个对象的方法：实例化对象 :: 普通方法；

·引用特定类型的方法：特定类 ::普通方法；

·引用构造方法：类名称 :: new。

范例：引用静态方法

·在String类里面有一个valueof()方法：public static String valueOf(int x);

|  |
| --- |
| **package** cn.mldn.demo;  /\*\*  \*  \* 实现方法的引用接口  \*  \* **@param**<P>引用方法的参数类型  \* **@param**<R>引用方法的返回类型  \*/  **interface** IMessage<P,R>{  **public** R zhuanhuan(P p);  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  //即：将String.valueOf()方法变为了IMessage接口里面的zhuanhuan()方法  IMessage<Integer,String>msg = String :: *valueOf*;  String str = msg.zhuanhuan(1000);  System.***out***.println(str.replaceAll("0", "9"));  }  } |

范例:普通方法引用

|  |
| --- |
| **package** cn.mldn.demo;  /\*\*  \*  \* 实现方法的引用接口  \*  \* **@param**<P>引用方法的参数类型  \* **@param**<R>引用方法的返回类型  \*/  **interface** IMessage<R>{  **public** R upper();  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  //String类的toUpperCase()定义：public String toUpperCase()  //这个方法没有参数，但是有返回值，并且这个方法一定要在有实例化对象的情况下才可以调用  //"hello"字符串是String类的实例化对象，所以可以直接调用toUpperCase()方法  //将toUpperCase()函数的应用交给了IMessage接口  IMessage<String>msg = "hello" :: toUpperCase;  String str = msg.upper();//降档于"hello".toUpperCase()  System.***out***.println(str);  }  } |

通过两个代码演示应该已经发现了，如果要实现函数的引用，那么必须要有接口，而且最为关心的，接口里面需要只存在一个方法。

为了保证被引用接口里面只能够由一个方法，那么就需要增加一个注解的声明。

|  |
| --- |
| **package** cn.mldn.demo;  /\*\*  \*  \* 实现方法的引用接口  \*  \* **@param**<P>引用方法的参数类型  \* **@param**<R>引用方法的返回类型  \*/  //此为函数接口，只能够定义一个方法  **interface** IMessage<R>{  **public** R upper();  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  //String类的toUpperCase()定义：public String toUpperCase()  //这个方法没有参数，但是有返回值，并且这个方法一定要在有实例化对象的情况下才可以调用  //"hello"字符串是String类的实例化对象，所以可以直接调用toUpperCase()方法  //将toUpperCase()函数的应用交给了IMessage接口  IMessage<String>msg = "hello" :: toUpperCase;  String str = msg.upper();//降档于"hello".toUpperCase()  System.***out***.println(str);  }  } |

在进行方法引用的过程里面还有另一种形式的引用（它需要特定类的对象支持），正常情况下如果使用了“类 :: 方法”，引用的一定是类中的静态方法，但是这种形式也可以引用普通方法

例如：在String类里面有一个方法：public int compareTo(String anotherString)

如果要进行比较的话，比较的形式：字符串1对象.compareTo(String 字符串2对象)，也就是说如果要引用这个方法，就要准备出两个参数。

范例：引用特定类的方法

|  |
| --- |
| **package** cn.mldn.demo;  /\*\*  \*  \* 实现方法的引用接口  \*  \* **@param**<P>引用方法的参数类型  \* **@param**<R>引用方法的返回类型  \*/  @FunctionalInterface//此为函数接口，只能够定义一个方法  **interface** IMessage<P>{  **publicint** compare(P p1,P p2);  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  IMessage<String>msg = String :: compareTo;  System.***out***.println(msg.compare("A", "B"));  }  } |

与之前相比，方法引用前不在需要定义对象，而是可以理解为将我们的对象定义在参数上。

范例：引用构造方法

|  |
| --- |
| **package** cn.mldn.demo;  /\*\*  \*  \* 实现方法的引用接口  \*  \* **@param**<P>引用方法的参数类型  \* **@param**<R>引用方法的返回类型  \*/  @FunctionalInterface//此为函数接口，只能够定义一个方法  **interface** IMessage<C>{  **public** C create(String t , **double**p);  }  **class** Book{  **private** String title;  **privatedouble**price;  **public** Book(String title,**double**price){  **this**.title = title;  **this**.price = price;  }  @Override  **public** String toString() {  **return**"书名： " + **this**.title + " , 价格：" + **this**.price;  }  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  IMessage<Book>msg = Book :: **new**;//引用构造方法  //调用的虽然是create()，但是这个方法引用的是Book类的构造  Book book = msg.create("java", 20.2);  System.***out***.println(book);  }  } |

**内建函数式接口**

对于方法的引用，严格来讲都需要定义一个接口，可是不管如何操作，实际上有可能操作的接口只有四种，在JDK1.8里面提供了一个包: java.util.function

1、功能型接口（Function）：public interface Function<T,R> {public R apply(T t);}

|-此接口需要接收一个参数，并且返回一个处理结果；

2、消费型接口（Consumer）：public interface Consumer<T> {public void accept(T t);}

|-此接口只是负责接收数据（引用数据是不需要返回），并且不返回处理结果

3、供给型接口（Supplier）：public interface Supplier<T>{public T get();}

|-此接口不接受参数，可以返回结果

4、断言型接口（Predicate）：public interface Predicate<T>{public boolean test(T t);}

|-进行判断操作使用；

所有在JDK1.8之中由于存在有以上的四个功能接口，所以一般很少会有用户去定义新的函数式接口。

范例：观察功能型接口—接受参数并且返回一个处理结果

·String类中有一个方法：public boolean startsWith(String prefix)

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.function.Function;  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  Function<String,Boolean>fun = "##hello" :: startsWith;  System.***out***.println(fun.apply("#"));  }  } |

范例：消费型接口

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.function.Consumer;  **class** MyDemo{  **publicvoid** print(String str){  System.***out***.println(str);  }  }  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  Consumer<String>cons = **new** MyDemo() :: print;  cons.accept("hello");  }  } |

范例：供给型接口

·引用String类的toUpperCase()方法：public String toUpperCase()

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.function.Supplier;  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  Supplier<String>sup = "hello" :: toUpperCase;  System.***out***.println(sup.get());  }  } |

范例：断言型接口

·String里面有一个equalsIgnoreCase()方法

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.function.Predicate;  **publicclass** TestDemo {  **publicstaticvoid** main(String[] args) {  Predicate<String>pre = "hello" :: equalsIgnoreCase;  System.***out***.println(pre.test("Hello"));  }  } |