模块化系统

目标

认识模块化的好处，如何建立模块与模块之间的访问

步骤

1. 模块化出现的背景与概念
2. 模块系统好处
3. 模块的输出与访问

模块化出现的背景与概念

jdk9之前没有解决的问题(没有模块化)：

1. **java运行环境代码臃肿、效率低**： Java9之前，每一个runtime自带开箱即用的所有编译好的平台类，这些类被一起打包到一个JRE文件叫做rt.jar。你只需将你的应用的类放到classpath中，这样runtime就可以找到，而其它的平台类它就简单粗暴的从rt.jar文件中去找。尽管你的应用只用到了这个庞大的rt.jar的一部分，这对JVM管理来说不仅增加了非必要类的体积，还增加了性能负载。Java9模块化可以按需自定义runtime!这也就是jdk9文件夹下没有了jre目录的原因！

2. **无法隐藏内部API和类型：**很难真正地对代码进行封装,系统对于不同部分的代码无法分离。 在早期我们实现封装都是需要依赖一下权限修饰符, 而权限修饰符只能修饰类、成员变量、成员方法。权限修饰符我们没法对包进行隐藏，JDK9我们通过隐藏包从而隐藏包中里面的所有类。

模块化的概念

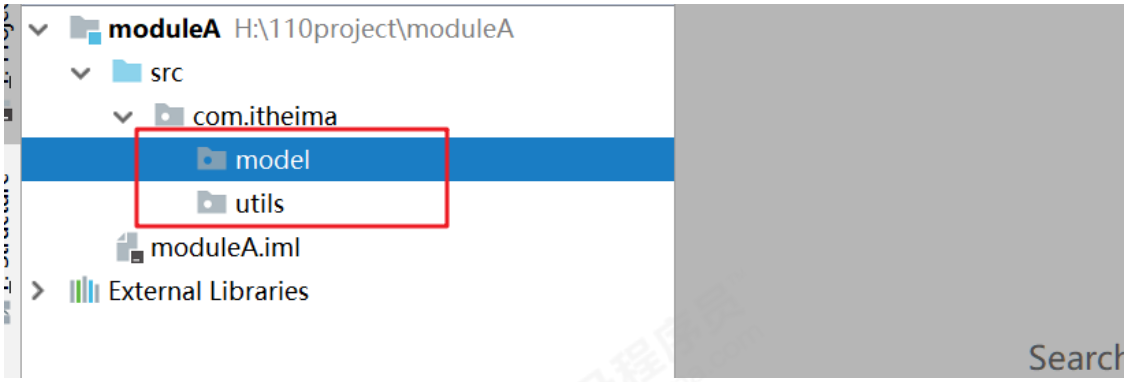
1. **模块化的目标**
   - 减少内存的开销，提高效率
   - 强封装： 每一个模块都声明了哪些包是公开的哪些包是内部的，java编译和运行时就可以实施这些规则来确保外部模块无法使用内部类型。

模块化的代码演示步骤

需求： 变量一个Module模块，然后使用ModuleB模块进行访问。

1. 创建一个ModuleA，然后创建两个包，com.itheima.utils和com.itheima.model.
2. 在utils包中创建一个ArrayUtils工具类并创建一个获取最大值的方法
3. 在modle包中创建一个Person类
4. 新建一个输出模块信息，只是输出utils包，model包对外隐藏
5. 创建一个ModuleB,然后新建一个ModuleTest类，测试使用ArrayUtils
6. 创建一个输入模块信息，并添加依赖

模块化之间的访问方式



2. 在utils包中创建一个ArrayUtils工具类并创建一个获取最大值的方法

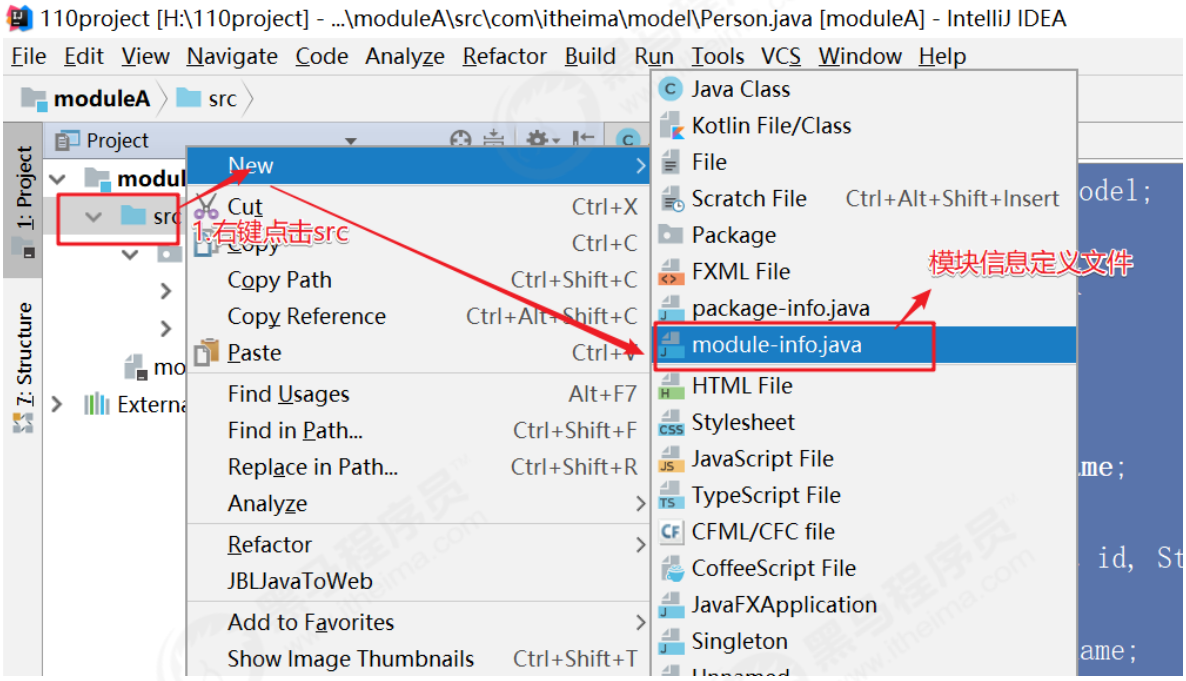
```java
package com.itheima.utils;

public class ArrayUtils {

    public static int getMax(int[] arr){
        //1.定义一个变量存储当前的最大值
        int max = arr[0];
        //2. 使用存储最大值的变量与数组中的每一个元素进行对比
        for (int i = 1; i < arr.length; i++) {
            if(arr[i]>max){
            //3. 如果发现了数组的元素比最大值变量要大，那么最大值的变量存储当前的元素
                max  = arr[i];
            }
        }
        //4. 返回最大值
        return max;
    }
}

```

3. 在modle包中创建一个Person类

```java
package com.itheima.model;

public class Person {

    private int id;

    private String name;

    public Person(int id, String name) {
        this.id = id;
        this.name = name;
    }

    @Override
    public String toString() {
```

```
                '}';
        }
}
```

4. 新建一个输出模块信息，只是输出utils包，model包对外隐藏




```
module moduleA {
    exports com.itheima.utils;
}
```

1. 创建一个ModuleB,然后新建一个ModuleTest类，测试使用ArrayUtils

```
package com.itheima.test;

public class ModuleTest {

    public static void main(String[] args) {
        int[] arr= {10,19,50,3,2};
        int max = ArrayUtils.getMax(arr);
        System.out.println("最大值："+max);
    }
}

```

2. 创建一个输入模块信息，并添加依赖

小结

1. **认识模块化的好处**
   - 提高效率
   - 可以实现包隐藏，从而实现包里面的所有类隐藏。
2. **如何建立模块与模块之间的访问**
   - 定义一个输出模块信息
   - 定义一个输入模块信息
   - 添加依赖

交互式编程： jshell工具

目标

交互式编程的作用， 如何使用jshell工具

步骤

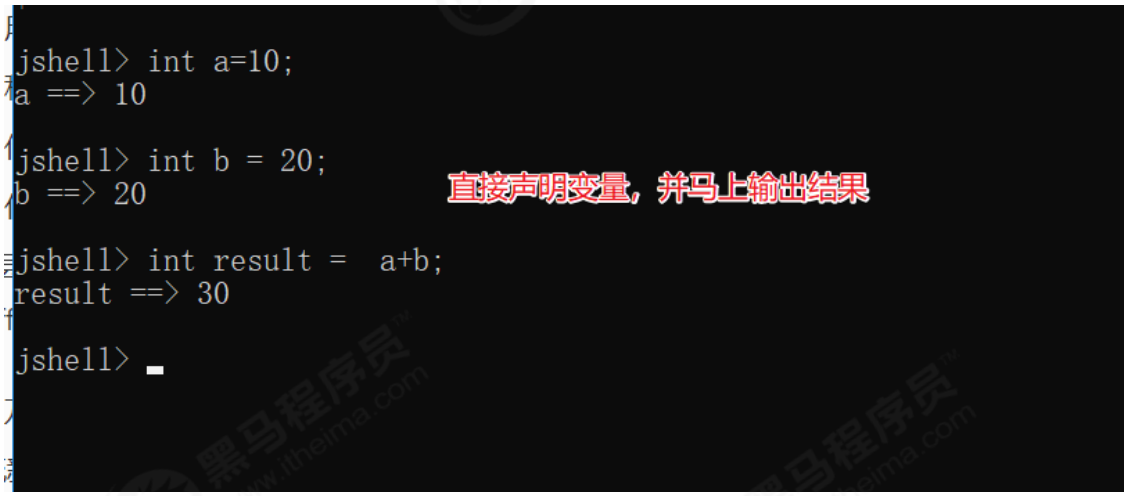
1. 交互式编程的作用
2. 如何使用jshell工具

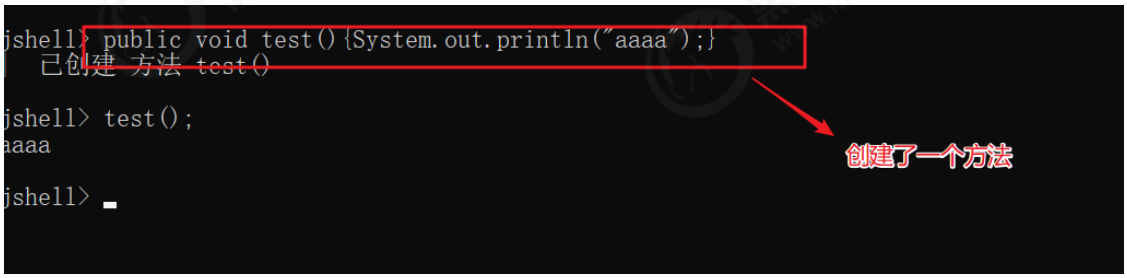
交互式编程的概念

java的编程模式是：编辑，保存，编译，运行和调试。 有时候我们需要快速看到某个语句的结果的时候，还需要写上public static void main(String[] args)这些无谓的语句，减低我们的开发效率。 JDK9引入了交互式编程，通过jshell工具即可实现，交互式编程就是指我们不需要编写类我们即可直接声明变量，方法，执行语句，不需要编译即可马上看到效果。 交互式编程的作用:即时反馈

- 打开jshell工具

- 直接声明变量、方法



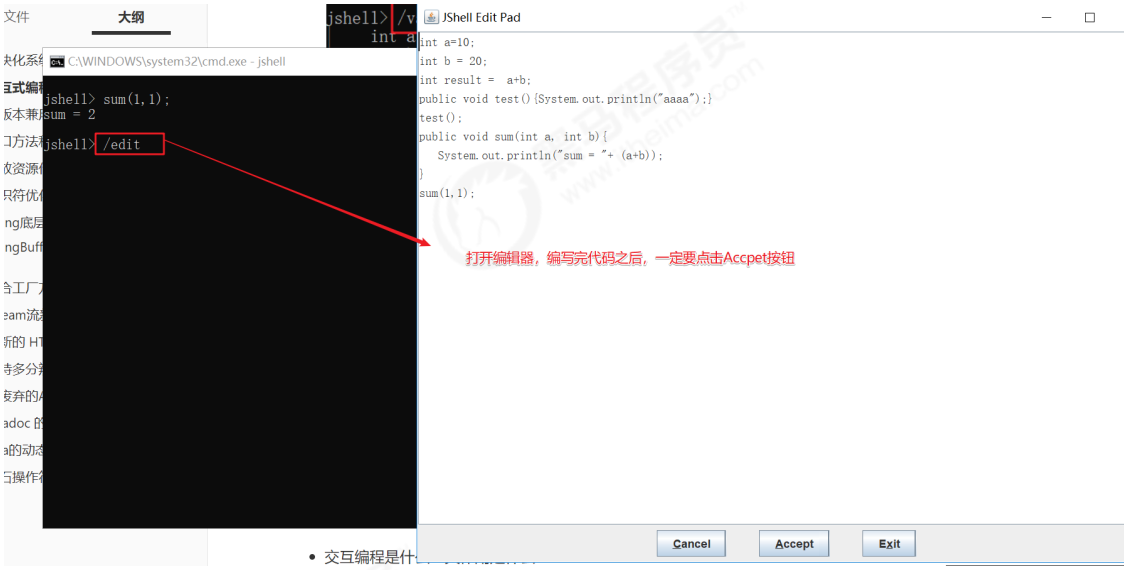


- /list 查看当前所有的代码（仅限于当前的会话，当前控制台）

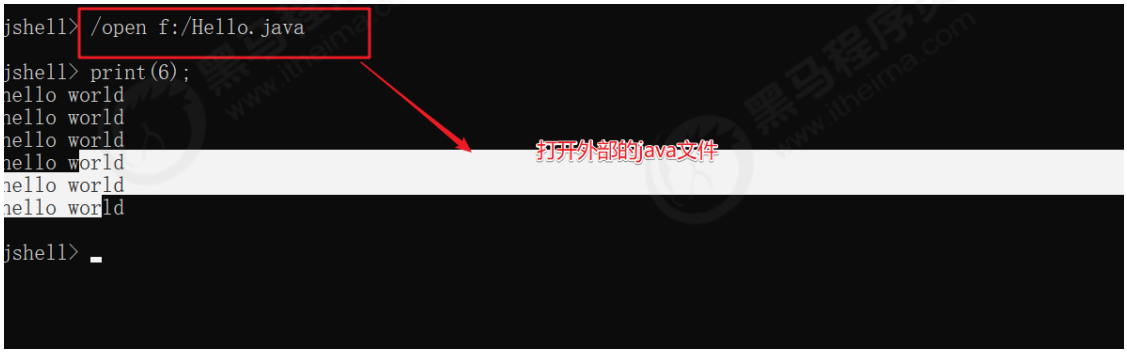
- /methods查看所有的方法

- /var 查看所有的变量

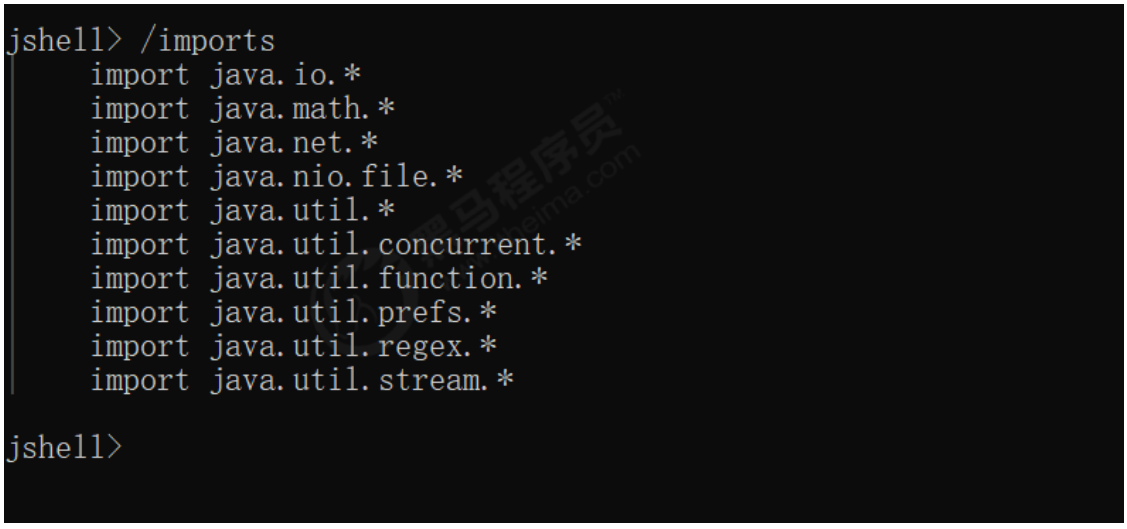
- /edit 打开编辑器



- /open 路径 执行外部的代码



- /imports 查看默认导入的包



- /exit 退出jshell工具

小结

- 交互编程是什么,其作用是什么?
  - 即时反馈, 马上能够看到语句的执行结果。

多版本兼用jar

目标

了解多版本兼用jar的作用，以及如何实现多版本jar的打包与使用

步骤

- 多版本兼用jar的作用
- 打包多版本jar
- 使用多版本jar

多版本兼用jar的作用

多版本JAR（MR JAR）可能包含同一类的多个变体,每个变体都针对特定的Java版本。 在运行时，类的正确变体将被自动加载，这取决于所使用的Java版本。这允许库作者在早期利用新的Java版本，同时保持与旧版本的兼容性。

应用场景：比如某个架构师开发了一个工具类MyUtils,该工具类里面使用了jdk9的新特性，这时候该工具在推广的时候会遇到很大的阻力，因为很多用户还没有升级jdk版本，JDK9推出了多版本兼用jar的特性就允许该架构师编写一个同类名的工具MyUtils,并在该工具类中不使用jdk9的新特性，然后两个同类名的类一起打包成为一个jar，提供给用户去使用,这时候即可根据用户当前使用的jdk版本而选择不同的工具类了。

```
util.jar
|  MyUtils.class
|
└─META-INF
    |  MANIFEST.MF
    |
    └─versions
        └─9
                MyUtils.class
```

简言之：该jar包在java 8中可以执行最上层的MyUtils.class，在Java 9中自动选择执行目录9下的MyUtils.class。

第一步

创建文件夹 java，并在该文件夹下创建 MyUtils.java 文件，代码如下：

```
package com.itheima.utils;

public class MyUtils {

    public static void print(){
        System.out.println("JDK8");
    }
}

```

第二步

创建文件夹 java9，并在该文件夹下创建 MyUtils.java 文件，代码如下：

```
package com.itheima.utils;

import java.util.stream.Stream;

public class MyUtils {

    public static void print(){
        System.out.println("JDK9");
    }
}

```

第三步

在src目录下编译成class文件：

```
javac -d out --release 8  java/com/itheima/utils/MyUtils.java
javac -d out9 --release 9 java9/com/itheima/utils/MyUtils.java
```

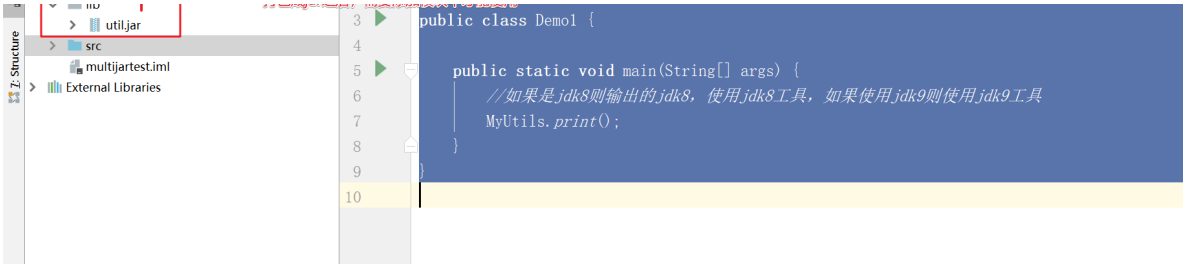
第四步

在src目录下把两个版本的class文件打成jar包

```
jar --create  --file util.jar -C out . --release 9 -C out9 .
```

第五步

新建一个模块，并把该jar导入到模块中，然后切换不同的jdk版本上去测试.



小结

- 多版本兼用jar的作用
  - 根据当前用户安装的jdk版本选择性的使用具体类。

接口方法私有化

目标

接口方法私有化的作用？ 如何在接口中定义私有化方法

学习步骤

- 清楚接口方法私有化的作用
- 如何在接口中定义私有化的方法以及调用

接口方法私有化的作用

当我们在一个接口里写多个默认方法或者静态方法的时候，可能会遇到程序重复的问题。我们可以把这些重复的程序提取出来，创建一个新的方法，用private进行修饰，这样就创造了一个只有接口可以调用的私有方法。

```
package com.itheima.dao;

public interface UserDao {

    default void methodA(){
        System.out.println("methodA...");
        System.out.println("A....");
        System.out.println("B....");
        System.out.println("C....");
    }

    default void methodB(){
        System.out.println("methodB...");
        System.out.println("A....");
        System.out.println("B....");
        System.out.println("C....");
    }
```

存在的问题： 以上代码的methodA与methodB存在着代码冗余问题，我们可以把这部分公共的方法抽取成私有的方法提供给接口内部去使用。

接口私有方法的作用：解决接口中默认方法与静态方法代码重复的问题

接口定义私有化方法

```
1
2  package com.itheima.dao;
3
4  public interface UserDao {
5
6      default void methodA(){
7          System.out.println("methodA...");
8          commons();
9      }
10
11     default void methodB(){
12         System.out.println("methodB...");
13         commons();
14     }
15
16     //定一个私有的方法，把重复部分的代码抽离出来。然后在methodA与methodB方法内部去调
   用。
17     //私有方法只能在本类中调用，这里包括接口的实现类也不能调用。
18     private void commons(){
19         System.out.println("A....");
20         System.out.println("B....");
21         System.out.println("C....");
22     }
23
24 }
25
26
```

测试代码

```
1  package com.itheima.dao.impl;
2
3  import com.itheima.dao.UserDao;
4
5  public class UserDaoImpl implements UserDao {
6
7      public static void main(String[] args) {
8          UserDaoImpl userDao = new UserDaoImpl();
9          userDao.methodA();
10         userDao.methodB();
11     }
12 }
13
14
```

小结

- 清楚接口方法私有化的目的?
  - 解决静态或者是默认方法代码重复的问题。
- 如何在接口中定义私有化的方法以及调用?
  - 在接口中使用private修饰方法即可。
  - 在方法的内部去调用。

释放资源代码优化

学习目标

- 掌握jdk9释放资源的代码格式

学习步骤

- 回顾jdk7,jdk8释放资源的代码
- 对比学习jdk9的更新操作

jdk8之前释放资源代码

```
1  package com.itheima.io;
2
3  import java.io.FileInputStream;
4  import java.io.IOException;
5
6  //回顾jdk8之前释放资源的代码
7  public class JDK7 {
8
9      public static void main(String[] args) {
10         FileInputStream fileInputStream = null;
11         try {
12             //1. 建立程序与文件的数据通道
13             fileInputStream = new FileInputStream("F:/a..txt");
14             //2. 创建字节数组缓冲区
15             byte[] buf = new byte[1024];
16             int length = 0 ;
17             //3. 读取数据，并且输出
18             while((length = fileInputStream.read(buf))!=-1){
19                 System.out.println(new String(buf,0,length));
20             }
21         } catch (IOException e) {
22             e.printStackTrace();
23         } finally {
```

```
                    fileInputStream.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }


    }
}

```

通过上述代码我们可以看到释放资源代码非常累赘， 如果释放资源较多的时候，很容易就会出现释放资源代码超过了正常业务的代码. 对此随着jdk版本的不断更新迭代，也对释放资源代码做了很大幅度的优化。

JDK8释放资源代码

```
package com.itheima.io;

import java.io.FileInputStream;
import java.io.IOException;

//回顾jdk8之前释放资源的代码
public class JDK8 {

    public static void main(String[] args) {
        //需要释放资源的流，填写在try()中
        //注意：初始化流对象的代码一定要写在try()内部中。
        try( FileInputStream fileInputStream = new
FileInputStream("F:/a.txt");){
            //1. 建立程序与文件的数据通道
            //2. 创建字节数组缓冲区
            byte[] buf = new byte[1024];
            int length = 0 ;
            //3. 读取数据，并且输出
            while((length = fileInputStream.read(buf))!=-1){
                System.out.println(new String(buf,0,length));
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

注意：JDK8开始已经不需要我们再手动关闭资源，只需要把要关闭资源的代码放入try语句中即可，但是要求初始化资源的语句必须位于try语句中

JDK9释放资源代码

```
import java.io.FileNotFoundException;
import java.io.IOException;

//回顾jdk8之前释放资源的代码
public class JDK9 {

    public static void main(String[] args) throws FileNotFoundException {
        //需要释放资源的流，填写在try()中
        FileInputStream fileInputStream = new FileInputStream("F:/a.txt");
        try(fileInputStream){
            //1. 建立程序与文件的数据通道
            //2. 创建字节数组缓冲区
            byte[] buf = new byte[1024];
            int length = 0 ;
            //3. 读取数据，并且输出
            while((length = fileInputStream.read(buf))!=-1){
                System.out.println(new String(buf,0,length));
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

小结

- **jdk9释放资源的语法格式如何?**

  try(流对象的变量|需要释放资源对象变量引用){

  }catch(){

  }

标识符优化

目标

学习jdk9对标识符的新规则

学习步骤

- 演示jdk9之前的标识符规则
- 演示jdk9标识符规则

jdk9之前

```
package com.itheima.flag;

public class Demo1 {

    public static void main(String[] args) {
        String _  = "hello";
        System.out.println(_);
    }
}

```

以上代码不会报错，允许_作为标识符

JDK9开始

```
package com.itheima.flag;

public class Demo1 {

    public static void main(String[] args) {
        String _  = "hello";
        System.out.println(_);
    }
}

```

以上代码报错，jdk9开始不允许_作为标识符

小结

- jdk9为标识符定义了什么的新规则
    - _不能作为单独的标识符。

String底层结构的变化

目标

- 了解String类底层的变化动机与具体变化

学习步骤

- 了解String底层变化的动机是什么

Motivation

The current implementation of the String class stores characters in a char array, using two bytes (sixteen bits) for each character. Data gathered from many different applications indicates that strings are a major component of heap usage and, moreover, that most String objects contain only Latin-1 characters. Such characters require only one byte of storage, hence half of the space in the internal char arrays of such String objects is going unused.

动机

string类的当前实现将字符存储在char数组中，每个字符使用两个字节（16位）。从许多不同的应用程序收集的数据表明，字符串是堆使用的主要组成部分，而且，大多数字符串对象只包含拉丁-1字符。这样的字符只需要一个字节的存储空间，因此这样的字符串对象的内部字符数组中的一半空间将被闲置。

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final char value[];

    /** Cache the hash code for the string */
```

jdk 8

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {

    /**
     * The value is used for character storage.
     *
     * @implNote This field is trusted by the VM, and is a subject to
     * constant folding if String instance is constant. Overwriting this
     * field after construction will cause problems.
     *
     * Additionally, it is marked with {@link Stable} to trust the conten
     * of the array. No other facility in JDK provides this functionality
     * {@link Stable} is safe here, because value is never null.
     */
    @Stable
    private final byte[] value;
```

jdk9

小结

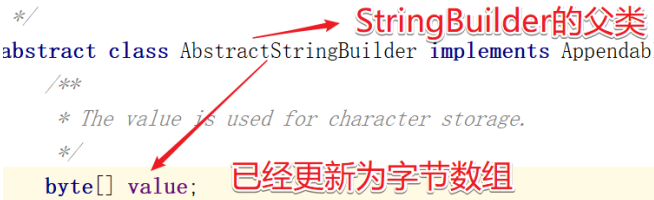
- String底层变化的动机是什么
  - 节省内存空间。
- 具体的变化有什么
  - Stirng类内部维护一个字符数组变化为维护一个字节数组。

StringBuffer与StringBuilder底层变化

由于String类底层已经发生变化，所以StringBuilder与StringBuffer底层也相应的发生了改变。

String-related classes such as AbstractStringBuilder, StringBuilder, and StringBuffer will be

```
 * @since       1.5
 */
abstract class AbstractStringBuilder implements Appendable, CharSequence {
    /**
     * The value is used for character storage.
     */
    byte[] value;

    /**
     * The id of the encoding used to encode the bytes in {@code value}.
     */
    byte coder;
```



集合工厂方法：快速创建只读集合

目标

- 学习jdk9新增的创建只读集合的方法

步骤

- 学习jdk9新增创建只读集合的方法
- 修改只读数组的异常

新增的方法

调用集合中静态方法 of()，可以将不同数量的参数传输到此工厂方法中。此功能可用于 **Set** 和 **List**，也可用于 **Map** 的类似形式。**此时得到 的集合，是不可变的**：

- List.of
- Set.of
- Map.of

方法演示

```
1  package com.itheima.collection;
2
3  import java.util.List;
4  import java.util.Map;
5  import java.util.Set;
6
7  public class CollectionDemo {
8
9      public static void main(String[] args) {
10         //1. 创建一个List的只读集合
11         List list = List.of("张三","李四","王五");
12         System.out.println("list集合的内容: "+list);
13         //2. 创建一个Set的只读集合
14         Set set = Set.of("张三","李四","王五");
15         System.out.println("set集合的内容:"+ set);
```

```
        System.out.println("map集合的内容:"+ map);

    }
}

```

常见的异常

通过of方法创建的集合对象是不能修改集合中的元素的，如果强行修改就会出现UnsupportedOperationException

```
package com.itheima.collection;

import java.util.List;
import java.util.Map;
import java.util.Set;

public class CollectionDemo {

    public static void main(String[] args) {
        //1. 创建一个List的只读集合
        List list = List.of("张三","李四","王五");
        System.out.println("list集合的内容: "+list);
        //2. 创建一个Set的只读集合
        Set set = Set.of("张三","李四","王五");
        System.out.println("set集合的内容:"+ set);
        //3. 创建一个Map的只读集合
        Map map = Map.of(1,"张三",2,"李四",3, "王五");
        System.out.println("map集合的内容:"+ map);

        //尝试修改只读集合的内容
        list.add("aa");

    }
}

```

小结

- jdk9新增的创建只读集合的方法，创建的集合具备什么特点?
  - of()
  - 特点： 只读,不能修改

Stream流新增的方法

学习目标

- 掌握Stream新增的方法以及新增方法的使用

学习步骤

新增方法的列表

- takeWhile()： *从Stream中依次获取满足条件的元素，直到不满足条件为止结束获取，只要遇到第一个不满足的条件元素马上停止获取*
- dropWhile()： *从Stream中依次删除满足条件的元素，直到不满足条件为止结束删除*
- ofNullable()： Java 8 中 Stream 不能完全为 null（一个元素不能为 null 多个元素是可以存在 null），否则会报空指针异常。而 Java 9 中的 ofNullable 方法允许我们创建一个单元素 Stream，可以包含一个非空元素，也可 以创建一个空 Stream。

方法演示

```
package com.itheima.stream;

import java.util.stream.Stream;

public class StreamDemo {

    public static void main(String[] args) {
        testNullStream();
    }

//    - takeWhile() ：从Stream中依次获取满足条件的元素，直到不满足条件为止结束获取，只要遇到第一个不满足的条件元素马上停止获取
    public static void testTakeWhile(){
        //1.创建一个流
        Stream<Integer> stream = Stream.of(10, 20, 30, 40, 50, 7, 60);
        //2 筛选符合条件的元素  条件：num<50
        stream.takeWhile(num->num<50).forEach(num->System.out.print(num+","));
    }

    //- dropWhile() ：  从Stream中依次删除满足条件的元素，直到不满足条件为止结束删除
     public static void testDropWhile(){
        //1.创建一个流
        Stream<Integer> stream = Stream.of(10, 20, 30, 40, 50, 7, 60);
        //2 删除符合条件的元素  条件：num<50
        stream.dropWhile(num->num<50).forEach(num->System.out.print(num+",")); //结果：50， 7， 60
    }

    //- ofNullable() ： Java 8 中 Stream 不能完全为 null（一个元素不能为 null 多个元素是可以存在 null ），否则会报空指针异常。而 Java 9 中的 ofNullable 方法允许我们创建一个单元素 Stream，可以包含一个非空元素，也可 以创建一个空 Stream。
    public static void testNullStream() {
        Stream<Object> stream = Stream.ofNullable(null);
        System.out.println(stream.count());
    }
}

```

小结

- takeWhile 获取符合条件的元素
- dropWhile 删除符合条件的元素
- ofNullable 允许创建只有null元素的stream对象。

全新的 HTTPClinet

目标

- 掌握HttpClient发送请求与接收响应

步骤

- 学习httpClient的基本概述
- 掌握使用HttpClient发送请求的步骤

HttpClient的基本概述

httpClient的作用就是用于获取网络资源的，Java 9 中有新的方式来处理 HTTP 调用。它提供了一个新的 HTTP 客户端（HttpClient ）， 它 将替代仅适用于 blocking 模式的 HttpURLConnection （HttpURLConnection是在HTTP 1.0的时代创建的， 并使用了协议无关的方法），并提供对 WebSocket 和 HTTP/2 的支持。

全新的HTTP客户端API可以从jdk.incubator.httpclient模块中获取。 因为在默认情况下，这个模块是不能根据 classpath 获取的，需要使 用 add modules 命令选项配置这个模块，将这个模块添加到 classpath 中。

代码演示

```
1  package url;
2
3  import jdk.incubator.http.HttpClient;
4  import jdk.incubator.http.HttpRequest;
5  import jdk.incubator.http.HttpResponse;
6
7  import java.net.URI;
8  import java.net.URISyntaxException;
9
10 //需求：使用HttpClient访问黑马服务器，得到黑马首页
11 public class HttpClientDemo {
12
13     public static void main(String[] args) throws Exception {
14         //1. 创建HttpClient的对象
15         HttpClient httpClient = HttpClient.newHttpClient();
16         //2. 创建请求的构造器
17         HttpRequest.Builder builder = HttpRequest.newBuilder(new
```

```
"heima").GET().build();
        //4. 使用HttpClient发送请求，并且得到响应的对象
        HttpResponse<String> response = httpClient.send(request,
HttpResponse.BodyHandler.asString());
        //5. 查看响应对象的信息
        System.out.println("响应状态码："+ response.statusCode());
        System.out.println("响应的信息："+response.body());
    }
}
```

小结

- HttpClient发送请求的步骤
  - 创建HttpClient的客户端
  - 创建请求构造器
  - 使用请求构造器创建请求
  - 使用客户端发送请求，并且得到响应对象
  - 查看响应的内容

支持多分辨率图片(了解)

目标

了解支持多分辨率图片

概述

在 java.awt.image 包下新增了支持多分辨率图片的API，用于支持多分辨率的图片。

1. 将不同分辨率的图像封装到一张（多分辨率的）图像中，作为它的变体。
2. 获取这个图像的所有变体。
3. 获取特定分辨率的图像变体，表示一张已知分辨率单位为 DPI 的特定尺寸大小的逻辑图像，并且这张图像是最佳的变体。。
4. 通过接口的getResolutionVariant（double destImageWidth, double destImageHeight）方法，根据分辨率获取图像。

被废弃的API(了解)

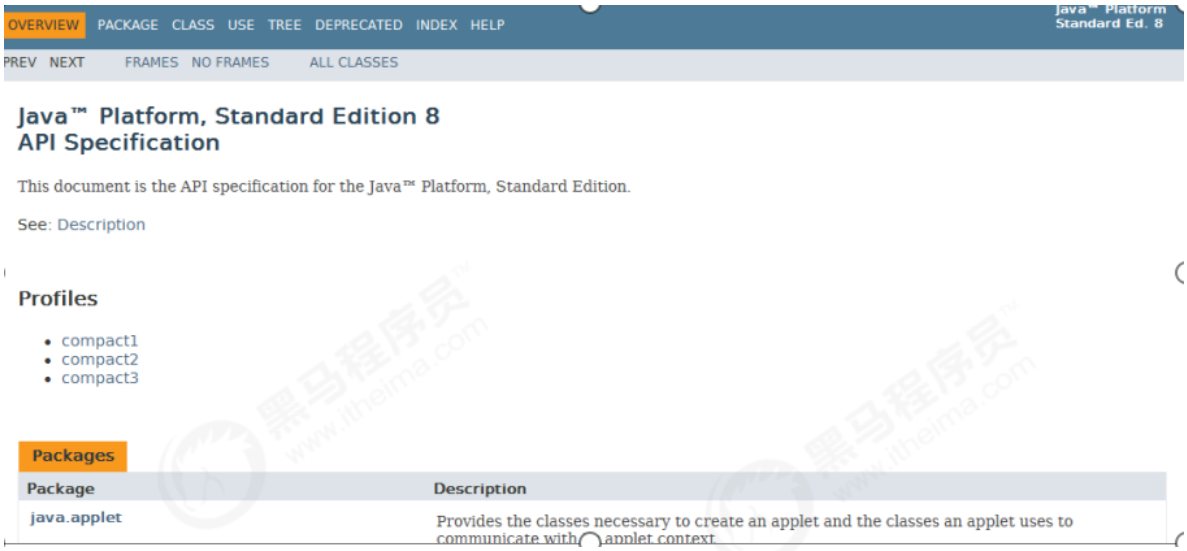
并且随着对安全要求的提高，主流浏览器已经取消对 Java 浏览器插件的支持。HTML5 的出现也进一步加速了它的消亡。开发者现在可以使用像 Java Web Start 这样的技术来代替 Applet，它可以实现从浏览器启动应用程序或者安装应用程序。

javadoc 的 HTML5 支持(了解)

JDK8 生成的java帮助文档是在 HTML4 中。而HTML4 已经是很久的标准了。

JDK9 的javadoc，现支持HTML5 标准。

下图是JDK8 API的HTML页面



java的动态编译器(了解)

动态编译器出现的目的就是为了提高编译的效率。 sjavac(smarter java compilation)最早在openjdk8中提供了初级版本，其初衷是用来加速jdk自己的编译。在9中进行过一版优化，使其更加稳定可靠，能够用来编译任意的大型java项目。sjavac在javac的基础上实现了：

- 增量编译 – 只重新编译必要的内容
- 并行编译 – 在编译期间使用多个核心

钻石操作符使用升级(了解)

jdk9开始允许匿名内部类使用与<>一起去使用。

```
1  package com.itheima.generic;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class Demo1 {
7
8      public static void main(String[] args) {
9          //jdk9之前该语句是会报错的，jdk9开始是合法的
10         List<String> list = new ArrayList<>(){};
11         list.add("aa");
12         list.add("bb");
13         list.add("cc");
14         System.out.println("集合的元素："+ list);
15     }
16 }
17
```