Java SE 5.0 特性

叶亮 (Liang.Ye@sun.com)
Sun Microsystems

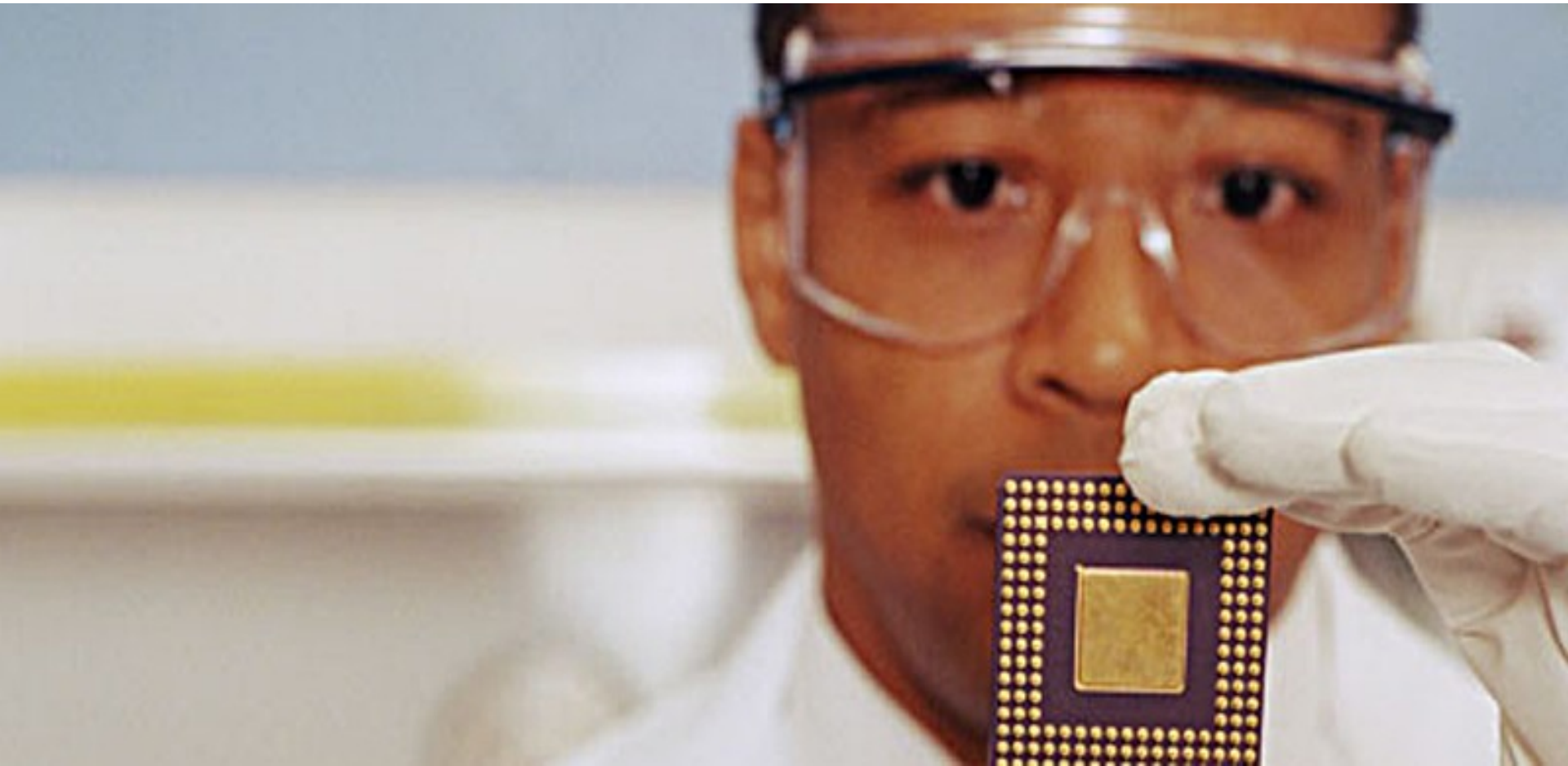
Java SE 设计思想

- 强调质量，稳定性和兼容性
  - 用户需要稳定优质的发行版本
- 支持及其广泛的应用类型
  - 从桌面应用到数据中心
- 可扩展性是工作的重点
  - 对于大堆栈，大数据量 I/O 的情况进行了优化
- 继续加入新的创新特性
- 帮助程序员更方便的开发程序
  - 更快速，更可靠

日程

- 语法变化
  - Generics, Annotation 等
- 库函数变化
  - Concurrency 工具库
- 虚拟机
  - 性能监控
- Java SE 6 新特性
- 总结和资源

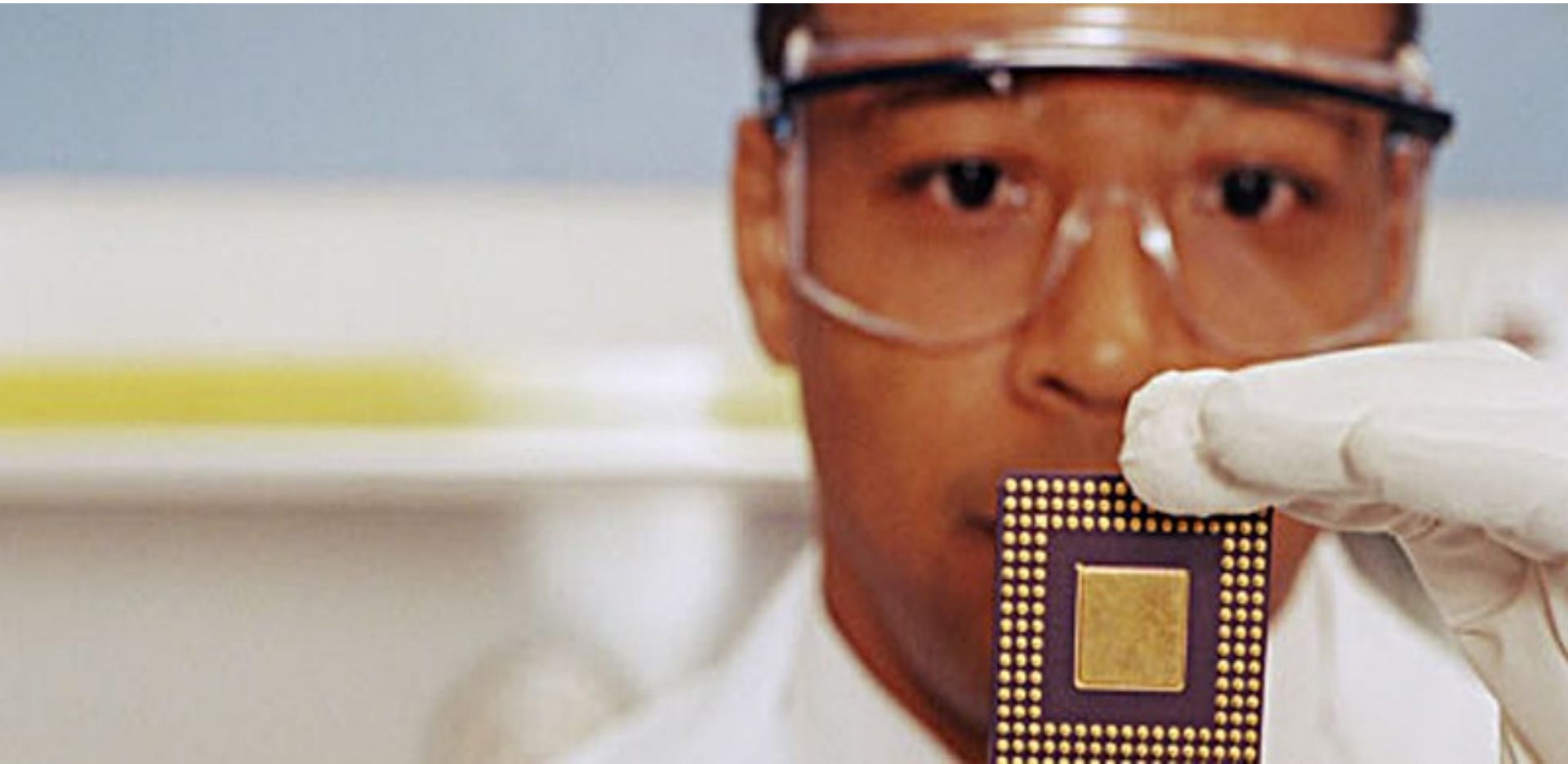
语言的更新



Java 5.0 语言上的主要变化

- 泛型 Generics
- 自动封装 Autoboxing/Unboxing
- 循环的增强
- 枚举类型
- 可变参数
- 静态导入
- 注解 (Annotation, Metadata)

泛型 Generics



问题 1

```
Vector v = new Vector();
v.add(new Integer(4));
OtherClass.expurgate(v);


...


static void expurgate(Collection c) {
  for (Iterator it = c.iterator();
it.hasNext();)
    /*  ClassCastException  */
    if (((String)it.next()).length() == 4)
      it.remove();
}
```

泛型

- 问题所在：Collection 元素类型
  - 编译器无法帮助验证类型
  - 赋值必须进行强制类型转换
  - 有可能产生运行时的错误 (**`ClassCastException`**)
- 解决办法：
  - 告诉编译器元素类型
  - 让编译器来做类型的匹配和转换
  - 保证运行成功

使用泛型修饰的对象：

- 创建与特定类型关联的泛型对象实例

```
Vector<String> x = new Vector<String>();
x.add(new Integer(5)); // 编译器错误
```

- 类型变量定义在 <> 之间
- 不同类型变量之间用逗号分隔

泛型的兼容性问题

- 与现成代码兼容

```
/*  Old class  */
public Vector getVector() { return new
Vector(); }

/*  New class  */
public Vector<String> s = oldCode.getVector();
/*  产生编译器警告，注意不是错误  */
```

问题2：

- 定义二元关系对：

```
public class PersonalInfo {
    int id; String name;
    public NumberPair(int f, String s) {
        id =f; name =s;
    }
}
public class NameInfo {
    String firstName; String secondName;
    public StringPair(String f, String s) {
        first =f; second =s;
    }
}
```

```
CarInfo{
long id, Car model
}
DressInfo{
Shirt shirt, Tie tie
}
.....
```

太“类”了！

- 泛型修饰的类

```
public class Pair<F, S> {
  F first;  S second;

  public Pair(F f, S s) {
    first = f;  second = s;
  }
}
Pair<String, String> nameinfo = new
Pair<String, String>("Liang", "Ye");
Pair<Shirt, Tie> dressinfo = new Pair<Shirt,
Tie>(new Shirt(Color.WHITE),
     new Tie(Color.DEEPBLUE));
```

问题 3：

- 如何打印任何 Collection 的内容？
  - 王二的解决方案

```
void printCollection(Collection<Object> c) {
  for (Object o : c)
    System.out.println(o);
}
```

问题 3：

- 如何打印任何 Collection 的内容？
  - 王二的解决方案

```
void printCollection(Collection<Object> c) {
  for (Object o : c)
    System.out.println(o);
}
```

- 错误！
- 如果传入一个 Collection<String> 编译器将报错
- Collection<Object> 不是所有 Collection 的父类

泛型通配符

- 用泛型通配符来对付

```
void printCollection(Collection<?> c) {
   for (Object o : c)
      System.out.println(o);
}
```

- ? 是通配符
- Collection<?>可以匹配成类型的 Collection

泛型通配符

- 用作参数的类型不能像普通对象那样继承
- 通配符可以规定类型上限

```
public void drawAll(List<? extends Shape>s) {
  ...
}

List<Circle> c = getCircles();
drawAll(c);
List<Triangle> t = getTriangles();
drawAll(t);
```

问题 4：

- 如何将任意类型的数组元素复制到相应的集合中
  - 王二的解决方案

```
static void aToC(Object[] a, Collection<?> c) {
  for (Object o : a)
    c.add(o);
}
```

泛型方法

- 问题：如何对方法使用泛型

```
static void aToC(Object[] a, Collection<?> c) {
  for (Object o : a)
    c.add(o); /*   编译器错误   */
}
```

- 错误
- ? 表示未知类型，不是任何一个确定的类型

泛型方法

- 正确的解决方法
  - 用类型参数来定义泛型方法
  - 在方法调用发生时编译器匹配具体类型

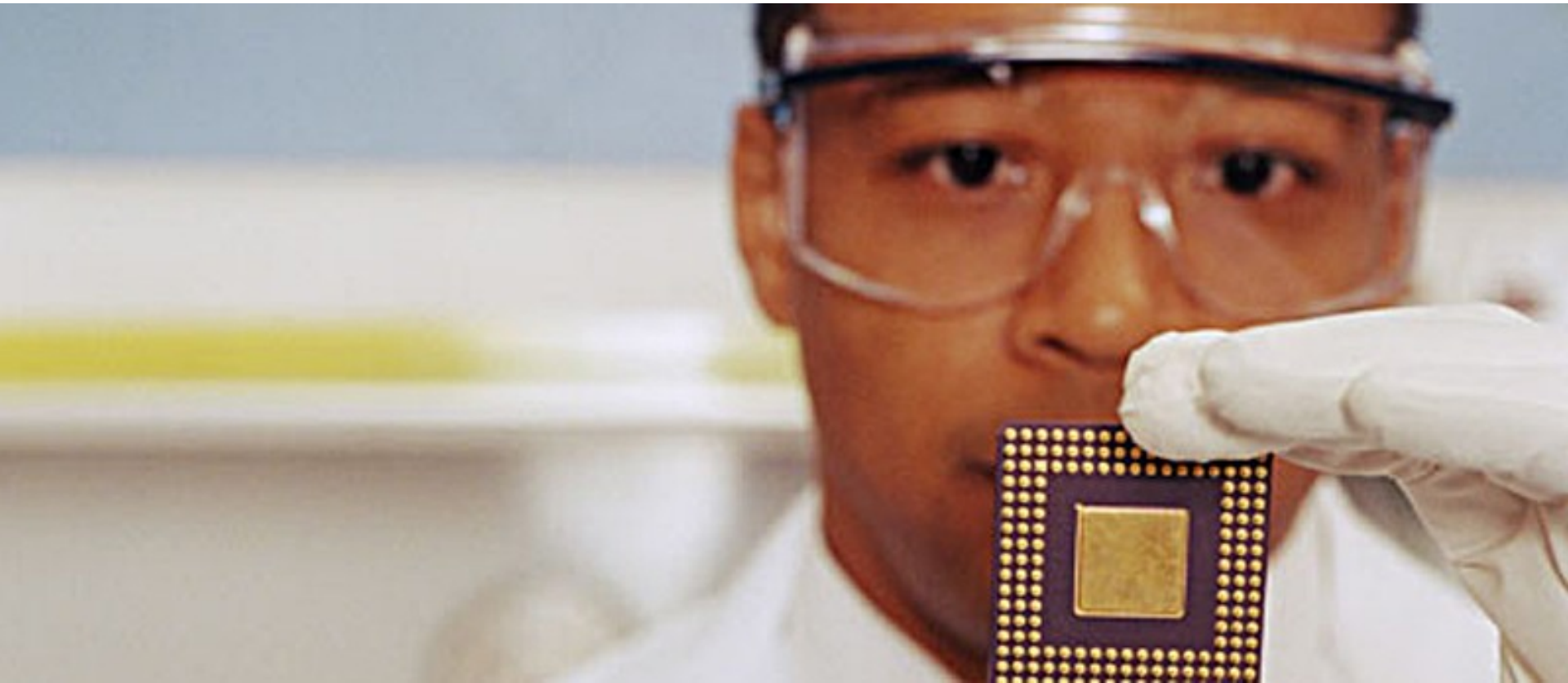
```
static <T> void aToC(T[] a, Collection<T> c) {
  for (T o : a)
    c.add(o);     /* 不会产生编译错误 */
}

String[] sa = new String[100];
Collection<Object> co = new ArrayList<Object>();
Collection<String> cs = new ArrayList<String>();
aToC(sa, cs);     /*  T 匹配为类型 String  */
aToC((Object[])sa, co);/*  T 匹配为类型 Object  */
```

类型去除 (Type Erasure)

- 编译器“去除”类型信息
  - 字节码 (Bytecodes) 与 5.0 以前版本相同
  - 安全性不受影响
  - 性能不受影响
    - 没有增进也没有减少
  - 类型信息保存在 ClassFile 中

```
Vector<String> vs = new Vector<String>();
Vector<Integer> vi = new Vector<Integer>();
vs.getClass() == vi.getClass();  /* true or false? */
```

自动装箱，For 循环增强，枚举类型及其它



基本类型的自动装箱

- 问题：
  - 基础类型与其包装类之间的相互转换
  - 比如当需要把基础类型加入集合的时候
- 解决方案：让编译器帮忙！

```
Byte byteObj = 22;          // 包装转换
int i = byteObj             // 解包转换

ArrayList al = new ArrayList();
al.add(22);  //   包装转换
```

For 循环增强 (foreach)

- 问题
  - 集合遍历容易出错
  - 通常情况下，iterator 只被用来得到一个元素
- 解决方法： 让编译器帮忙！
  - 新的 for 循环句法：

```
for (variable : collection)
```

  - 对集合与数组起作用

Loop 循环增强举例

- 早先的代码

```
void cancelAll(Collection c) {
  for (Iterator i = c.iterator(); i.hasNext();/*!
*/){
    TimerTask task = (TimerTask)i.next();
    task.cancel();
  }
}
```

- 如今的代码

```
void cancelAll(Collection<TimerTask> c) {
  for (TimerTask task : c)
    task.cancel();
}
```

类型安全的枚举类型

- 问题
  - 变量需要限定在一定的取值范围内
  - 比如，扑克牌花色只有四种
- 解决方案：引入新类型的类
  - 枚举类型定义了 public 的固定的常量元素
  - 新的关键字：enum
  - 可以用在 Switch 的情况

枚举类型：1

```
public enum Suit { spade, diamond, club, heart };
public enum Value { ace, two, three, four, five,
                    six, seven, eight, nine, ten,
                    jack, queen, king };

List<Card> deck = new ArrayList<Card>();

for (Suit suit : Suit.values())
  for (Value value : Value.values())
    deck.add(new Card(suit, value));

Collections.shuffle(deck);
```

想想 **Java 1.4** 要用多少行代码？！

枚举类型：2

```
public enum TrafficLight {
  RED(30), AMBER(10), GREEN(40);

  private final int duration;

  TrafficLight(int duration) {
    this.duration = duration;
  }

  public int duration() {
    return duration;
  }
}
```

枚举类型：3

```
public enum TrafficLight {
  RED(30)
    { public TrafficLight next() { return GREEN; }},
  AMBER(10)
    { public TrafficLight next() { return RED; }},
  GREEN(40)
    { public TrafficLight next() { return AMBER; }};
  private final int duration;

  TrafficLight(int duration) {
    this.duration = duration;
  }
  public int getDuration() { return duration; }

  public abstract TrafficLight next();
}
```

可变参数

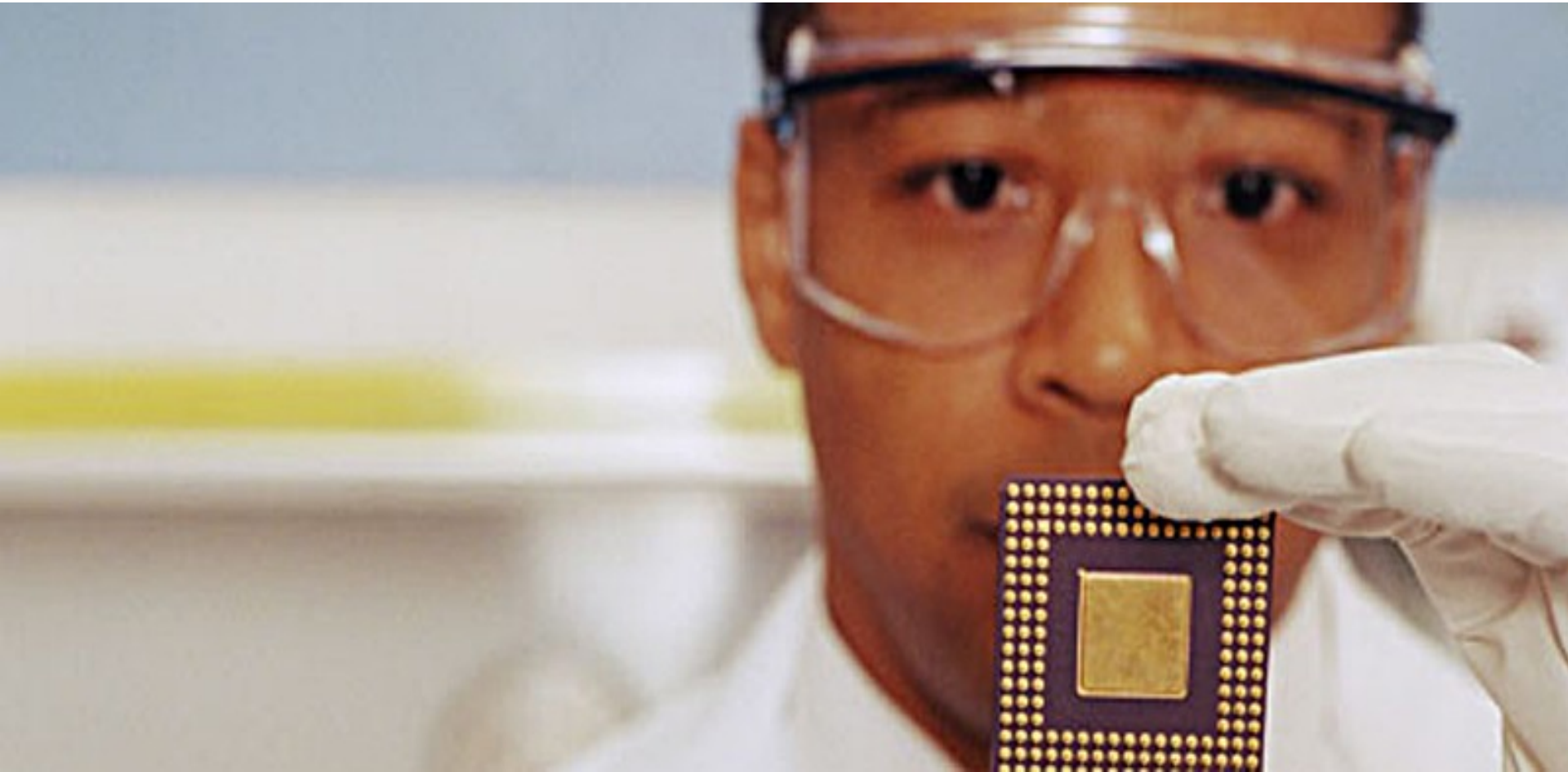
- 问题
  - 含有可变参数个数的方法
  - 可以通过数组来传递，但是不方便
  - 看看 **java.text.MessageFormat**
- 解决方法：让编译器帮你！
  - 新的语法：

```
public static void printf
(String fmt, Object... args);
```

静态导入

- 问题：
  - 外部静态引用必须要引用类名
- 解决方法：新的导入语法
  - `import static TypeName.Identifier;`
  - `import static Typename.*;`
  - 静态方法和枚举类型同样适用
    比如`Math.sin(x)`变成 `sin(x)`

注解 (Annotation)



Remote Interface 是多余的！

- 老代码

```
public interface PingIF implements java.rmi.Remote {
    public void foo() throws java.rmi.RemoteException;
}

public class Ping implements PingIF {
    public void foo() {}
    public void localfoo() {}
}
```

- 新的代码

```
public class Ping {
    public @Remote void foo() {}
    public void localfoo() {}
}
```

Metadata (JSR-175)

- 注解为代码提供额外的信息 (MetaData)
- JSR 175 为 Java 代码中的注解提供标准
- 代码中的注解由各种工具去解读
  - 编译器
  - IDE
  - Runtime 工具

标准 annotation - java.lang

- Override
- Deprecated
- SupressWarnings

给代码加上注解

- 标注 annotation
  - `public int @BetaVersion getValue()`
- 单值 annotation
  - `@Copyright(value = "Sun Microsystems")`
  - `@Copyright("Sun Microsystems")`
    - 只有当参数名称是 Value 的时候才可以直接使用
- 一般的 annotation
  - `@Author(@Name(first="fred",`
`last="bloggs"))`
  - `@Conributors({"fred", "joe", "bill"})`

定义 Annotations

- 跟接口定义类似
  - @interface
- 定义包含参数，这些参数在使用注解的时候被赋上具体值
- 可以设置缺省值
- 对于只有一个参数的注解， 参数名称 'value' 有特殊功用

注解的注解

- @Retention
  - 表示注解的信息要保存多久
  - **enum RetentionPolicy**
    - **SOURCE, CLASS, RUNTIME**
- @Target
  - 限定注解的作用范围
  - **enum ElementType**
    - **TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, ANNOTATION_TYPE, PACKAGE**

注解的注解

- Documented
- Inherited

注解举例

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Accessor {
   String variableName();
   String variableType() default "String";
}
```

使用上面定义的 Annotation:

```
@Accessor(variableName = "myVariable")
public String myVariable;
```

用反射得到 Annotation

- 标注 Annotation

```
boolean isBeta =
  MyClass.class.isAnnotationPresent(BetaVers
  ion.class);
```

- 单值 Annotation

```
String copyright =
  MyClass.class.getAnnotation
  (Copyright.class).value();
```

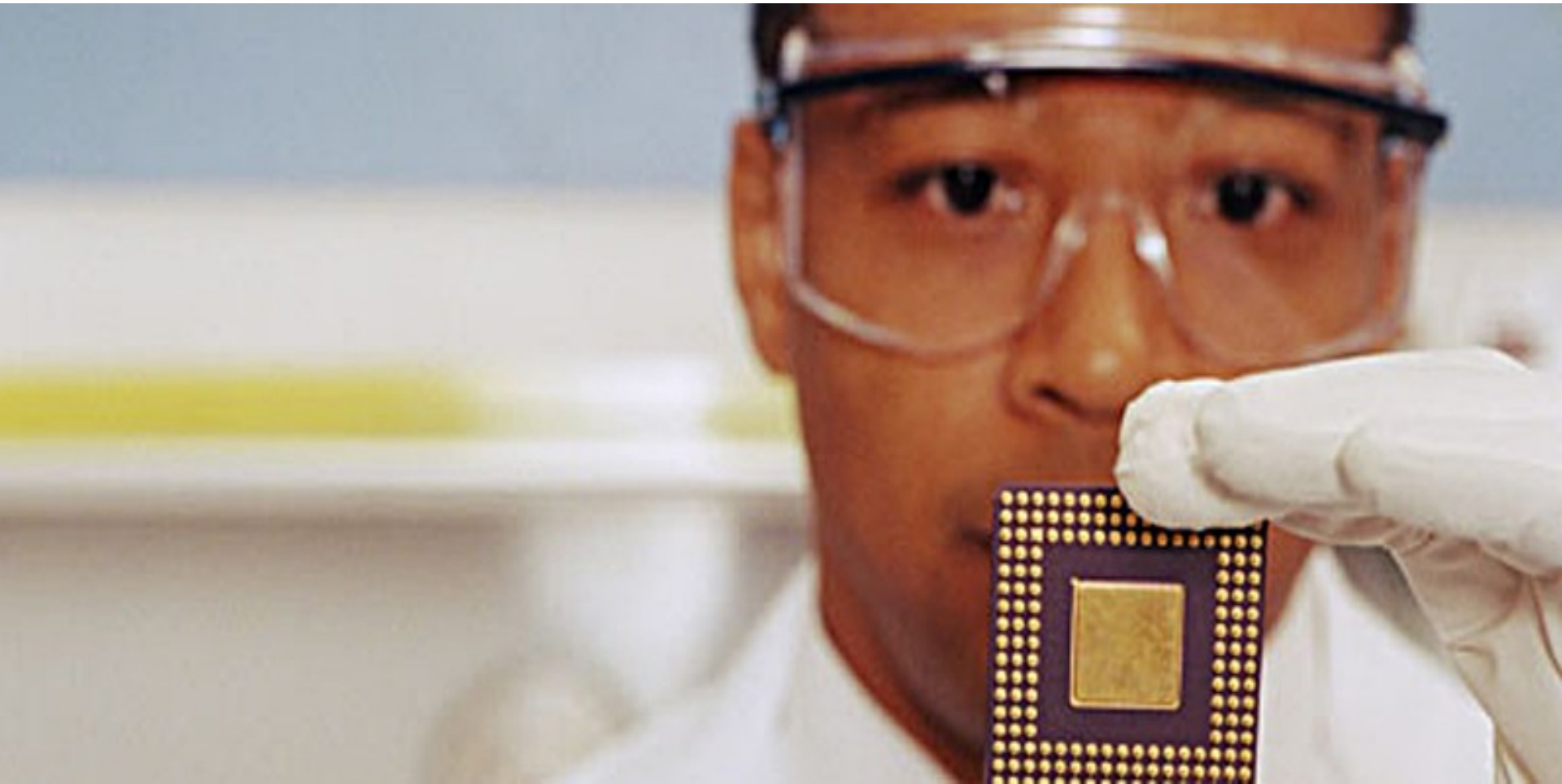
- 一般 Annotation

```
Name author =
  MyClass.class.getAnnotation(Author.class).
  value();
String first = author.getFirst();
String last = author.getLast();
```

Annotation Processing Tool

- 由 Annotation 修饰的代码文件生成多个文件
  - **apt** 是一个注解的解析工具
  - 生成的文件可以是多个源程序，发布配置文件等等
- 使用 Annotation 处理器
  - 可以用 **com.sun.mirror** 包开发 Annotation 处理器
  - **process()** 方法处理 annotations
  - **Filer** 创建多个派生文件
- **apt** 可以编译派生文件

库变化



格式化 I/O 和扫描程序

- Printf 倍受广大 C/C++程序员喜爱
  - 功能强大，使用方便
- Java 5.0 终于也有了自己的 printf!

```
out.printf("%-12s is %2d long", name, l);
out.printf("value = %2.2F", value);
```

  %n 是平台无关的换行标志
- 同时提供一个方便的扫描 API: 把文本转化成基本类型或者 String

```
Scanner s = new Scanner(System.in);
int n = s.nextInt();
```

多线程工具库：JSR-166

- 目标
  - 像集合包装数据一样包装多线程操作
  - 在高端 Server 应用上效率超过 C 的实现
  - 提供一些基本的多线程编程工具类
    - **wait()**, **notify()** and **synchronized** 太太太原始了
  - 增加了 Java 应用程序的可扩展性、可读性、线程安全性和运算性能

Thread.UncaughtExceptionHandler

- 不再需要实现 ThreadGroup
- 定义单个 Thread 的异常处理
- Thread.setUncaughtExceptionHandler
- Thread.setDefaultUncaughtExceptionHandler

Blocking Queue

- 为多线程程序提供线程安全的集合操作
- ArrayBlockingQueue 是最简单的实现
- 方法:
  - **put()***
  - **offer()** [非阻塞]
  - **Peek()** [取走但并不移除]
  - **take()***
  - **poll()** [非阻塞和定时阻塞]
    - add()
    - remove()

Blocking Queue 举例 : 1

```
private BlockingQueue<String> msgQueue;

public Logger(BlockingQueue<String> mq) {
  msgQueue = mq;
}
public void run() {
  try {
    while (true) {
      String message = msgQueue.take();
      /*   在日志中记录消息   */
    }
  } catch (InterruptedException ie) {
    /*  Handle  */
  }
}
```

Blocking Queue 举例 : 2

```
private ArrayBlockingQueue messageQueue =
  new ArrayBlockingQueue<String>(10);

Logger logger = new Logger(messageQueue);

public void run() {
  String someMessage;
  try {
    while (true) {
      /*  做一些处理  */
      /*  如果没有空间则阻塞  */
      messageQueue.put(someMessage);
    }
  } catch (InterruptedException ie) { }
}
```

BlockingQueue

- ArrayBlockingQueue
  - 固定大小、查询速度快
- LinkedBlockingQueue
  - 无限制、以 linked list 为基础
- PriorityBlockingQueue
  - take() 时排序、 comparator
- DelayQueue
- SynchronousQueue

java.util.concurrent.TimeUnit

- Enum:
  SENCONDS 、MILLISECONDS 、MICR
  OSECONDS 、NANOSECONDS
- long convert(long duration, TimeUnit unit)
- Void sleep(long timeout)
- timedJ oin(),timedWait(),toMicros().....

多线程工具库：JSR-166

- Executor 接口取代对 Thread 的直接使用
  - **`ExecutorService`** 接口
- Executors factory 工具类
  - **`ThreadPool`**
  - **`SingleThreadPool`**
  - **`PriorityThreadExecutor`**
- **`Callable`** 和 **`Future`**
- **`Semaphore`**
- **`BlockingQueue`**
- **`Atomic`**

Executor

- 不要用
  `new Thread(Runnable r).start();`
  - 创建一个 **`Executor`** 然后调用 **`execute()`**
- 新的线程工具类提供了一种干净的实现方法
  `Executor` 可以通过 **`ThreadPool`**, 或者 **`PriorityThreadPool`** 或者 **`SingleThreadExecutor`** 来创建

Executors

- 结束线程的方法被简化了
  - `pool.shutdown();`
- Executors 类提供了 factory 方法
  - `newSingleThreadExecutor();`
  - `newFixedThreadPool(int size);`
  - `newCachedThreadPool();`
  - `newScheduledThreadPool();`

Thread Pool 举例

```
class WebService {
  public static void main(String[] args) {
    Executor pool = Executors.newFixedThreadPool(7);
    ServerSocket socket = new ServerSocket(999);

    for (;;) {
      final Socket connection = socket.accept();
      pool.execute(new Runnable() {
        public void run() {
          new Handler().process(connection);
        }
      });
    }
  }
}
```

Callables 和 Futures

- **Callable** 接口提供了一种从其它线程得到结果或者异常的方法
  - 实现 **call()** 方法，而不是 **run()**
- **Callable** 提交到 **Executor**
  - 调用 **submit()** 而不是 **execute()**
  - 返回一个 **Future** 对象
- 通过对 **Future** 对象 **get()** 方法的调用得到结果
  - 如果结果已经产生了，则调用返回
  - 如果结果尚未产生，则调用方法的线程阻塞

Callable 举例

```
class CallableExample implements
    Callable<String> {

  public String call() {
    String result = null;

    /*   干点什么，然后产生一个结果   */

    return result;
  }
}
```

Future 举例

```
ExecutorService es =
  Executors.newSingleThreadExecutor();

Future<String> f =
  es.submit(new CallableExample());

/*   同时干点别的什么   */

try {
  String callableResult = f.get();
} catch (InterruptedException ie) {
  /*  Handle  */
} catch (ExecutionException ee) {
  /*  Handle  */
}
```

Locks

- Lock 接口
  - 比 synchronized 扩展性更强
  - 没有自动 unlocking
  - 用 **`tryLock()`** 调用尝试非阻塞的锁获取
- ReentrantLock
  - Lock 的一种具体实现
  - 握有该锁的线程可以调用多次 **`lock()`** 而不被阻塞

ReadWriteLock

- 两个 Lock 分别控制读写
  - Locks 是内部类
  - 如果没有线程握有 WriteLock，多个线程可以同时得到 ReadLock
  - 只有一个线程可以得到 WriteLock
  - 获取 Lock 的方法

```
rwl.readLock().lock();
rwl.writeLock().lock();
```

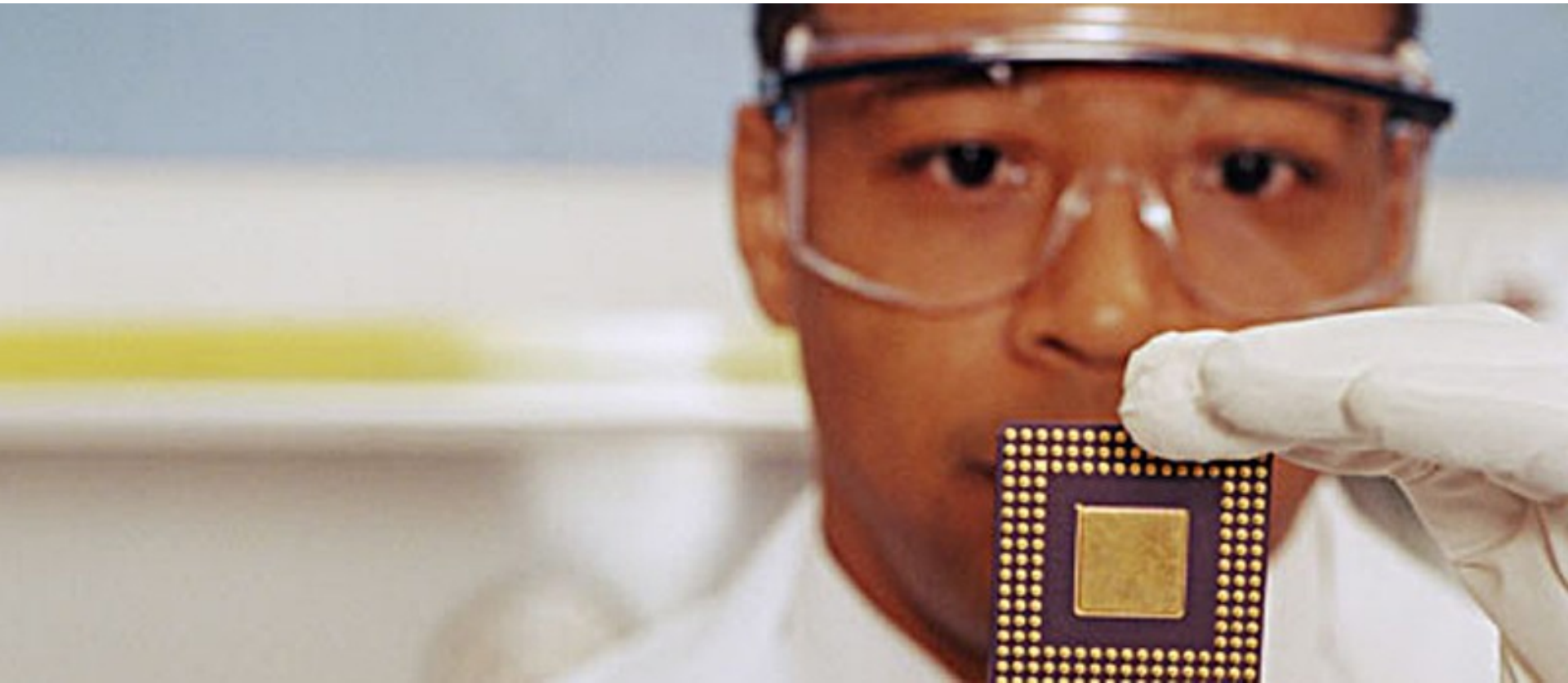
Semaphores

- 通常用来控制对一定数量的资源池的访问
- 根据需要限定访问的资源的数目来创建 Semaphore 对象
- 试图访问资源的线程调用 **aquire()**
  - 如果 semaphore 值大于零马上返回
  - 如果是零，则阻塞至其它线程调用 **release()** 方法
  - **aquire()** 和 **release()** 使线程安全的原子操作

Semaphore 举例

```
private Semaphore available;
private Resource[] resources;
private boolean[] used;

public Resource(int poolSize) {
  available = new Semaphore(poolSize);
  /*  初始化 resource pool  */
}
public Resource getResource() {
  try { available.aquire() } catch (IE) {}
  /*  返回  resource  */
}
public void returnResource(Resource r) {
  /*  把 resource 返回到池中  */
  available.release();
}
```

虚拟机



VM 的主要变化

- 类数据共享
  - 改善启动时间 （比原来快 30%）
  - 减少内存占用空间
  - -Xshare:on, -Xshare:dump
- 线程优先级的改进 (JSR-133)
- 严重问题的处理
  - **-XX:OnError="gcore %p; dbx - -%p"**
- 简化 stack trace 访问
  - **`Thread.getStackTrace()`**

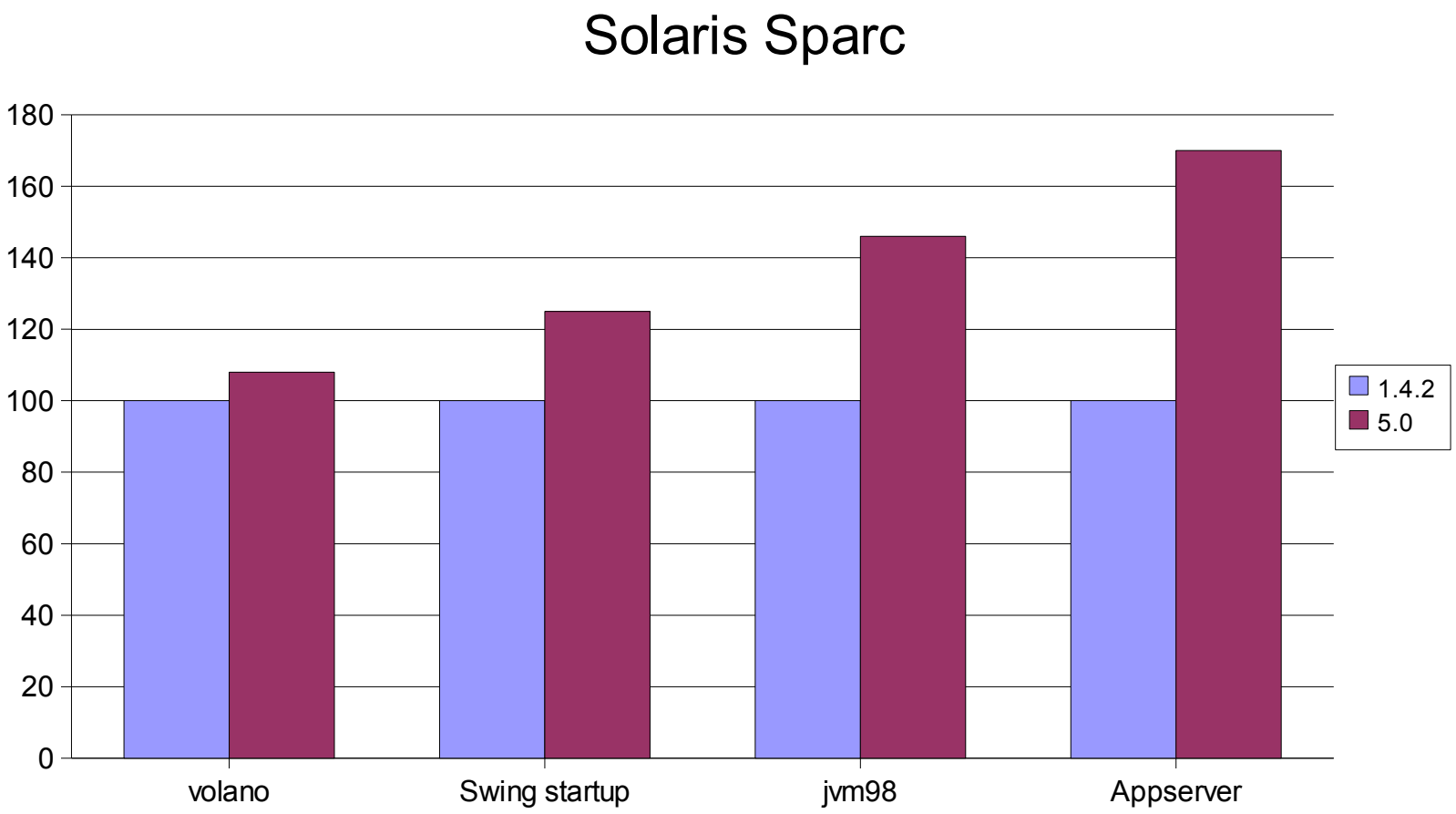
服务器版本

- 自动检测硬件配置
- 2 CPU, 2GB 内存 （不包括 windows）
  - 采用服务器版的 compiler
  - 采用并行垃圾回收器
  - 初始化堆大小为物理内存的 1/64 （最多 1GB）
  - 最大堆大小为物理内存的 1/4 （最多 1G）

JVM 调优 (Ergonomics)

- 最大间隔时间
  - -XX:MaxGCPauseMillis=<n>
  - 只是大概值，不是一个保证值
  - GC 自动调正参数尽量达到这一目标
  - 可以影响应用程序的吞吐量 (Throughput)
- 吞吐量目标
  - -XX:GCTimeRatio=<n>
  - GC 运行的比例 =1 / (1 + n)
  - 例如 -XX:GCTimeRatio=19 (5% 时间进行 GC)

性能改善


Solaris Sparc
180
160
140
120
100
80
60
40
20
0
volano
Swing startup
jvm98
Appserver
1.4.2
5.0

监控和管理

- Java 平台 RAS 的关键元素 (可靠性 Reliability, 可用性 Availability, 易用性 Serviceability)
- 特性
  - JVM 操作和 JMX 集成
  - 监控和管理 APIs
  - 工具
- JVMTI 替代 JVMPI
  - 性能分析改善
  - Bytecode 级别的操作

集成的 JMX (JSR-003)

- 标准的操作方式 (instrumenting)
  - JMX 内置的 Mbean server
    - JVM 启动时自动打开
  - 用于检测 JVM 各个方面特性的 MXBeans
- 内置 SNMP 监测
- 可以同现有的 J2EE 应用服务器一起工作
- 支持远程管理 (JSR-160)

监控和管理 APIs

- **`java.lang.management`**
- JVM 内置的 MXBeans
- 通过 API 可以访问
  - 载入系统中的类数， 运行线程数
  - 线程状态，连接的属性，堆栈跟踪
  - 垃圾回收统计
  - 内存使用情况
  - VM 运行时间，系统配置，输入参数

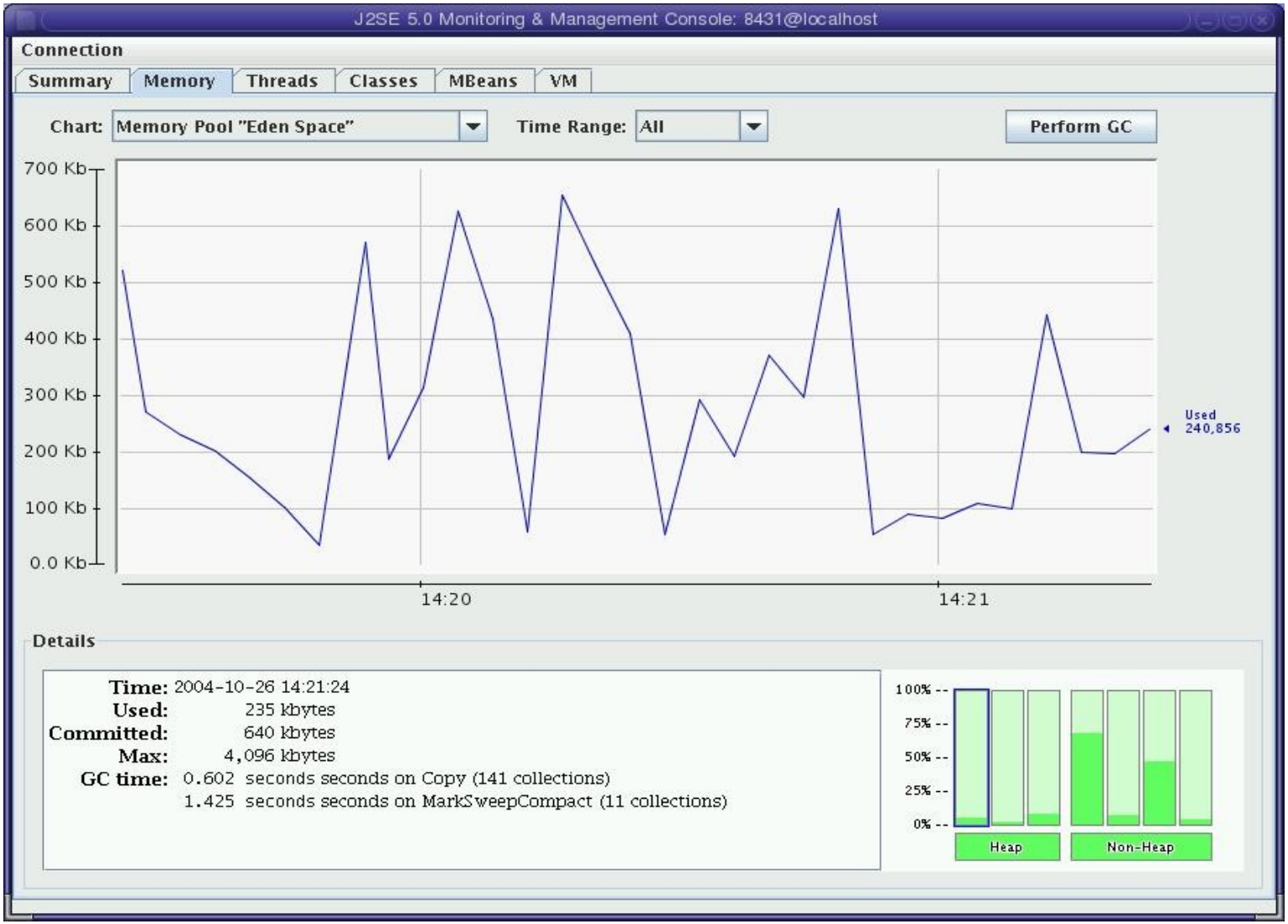
监控工具

- jinfo
  - JVM 或 crash dump 的配置信息
- jmap
  - JVM 或 crash dump 的很多有用的内存使用信息
- jstack
  - 打印出 VM 或 crash dump 的所有线程的堆栈跟踪
- 以上这些命令在 Windows 上不支持

监控工具

- jstat
  - 垃圾回收状态统计，动态编译，class loader
  - 不支持 Windows 98, ME
  - 不支持采用 FAT32 文件格式的 Windows NT, 2000, XP
- jps
  - 列出所有 HotSpot VMs 的进程号

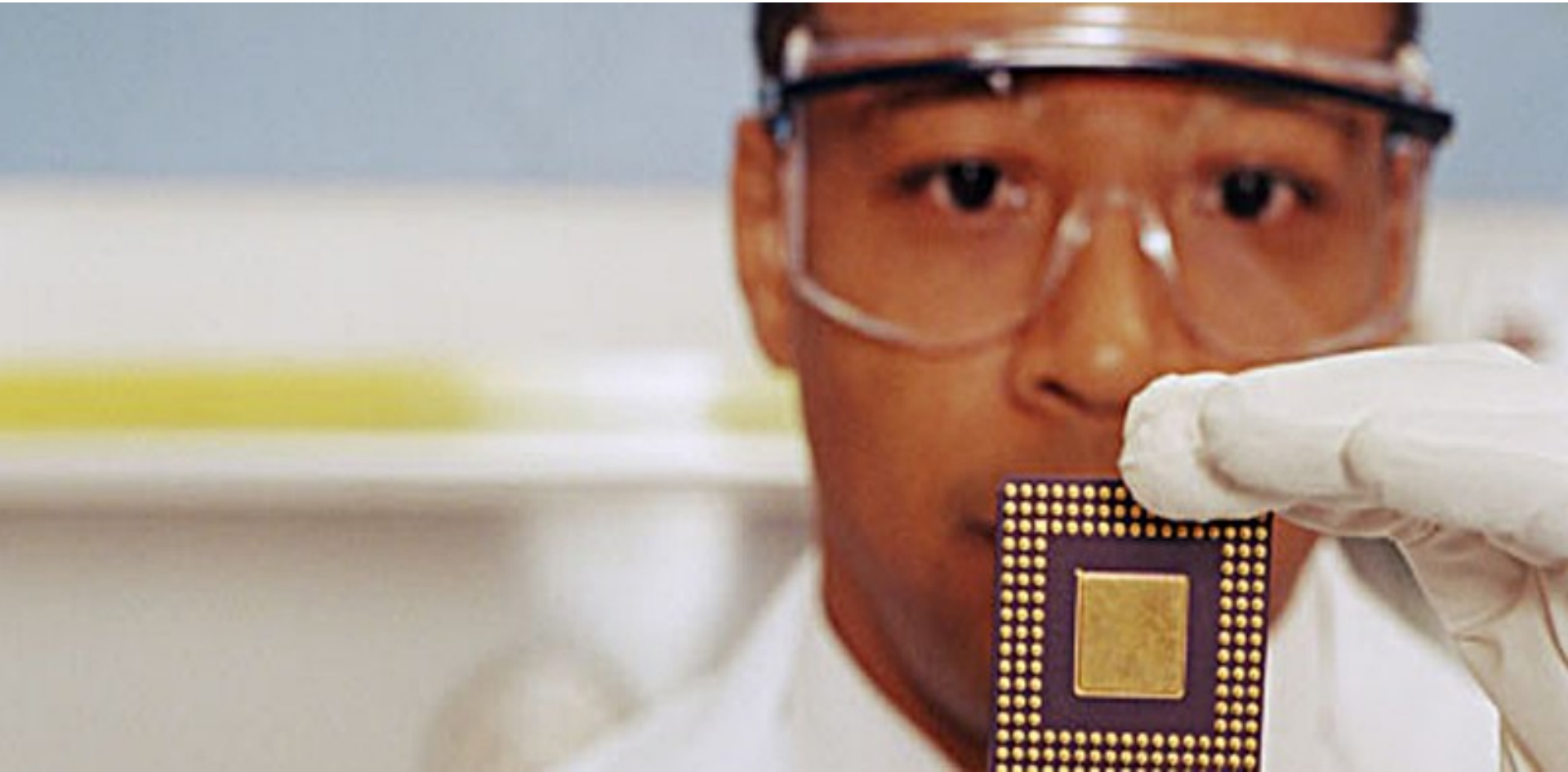
监控工具 : jconsole



兼容性和移植

- Java 5.0 类无法运行在早期版本的 JVM 上
- **enum** 现在是保留字
  - 不能再被用于变量名
- java.net.Proxy
  - 可能跟 java.lang.reflect.Proxy 冲突
- JAXP 微小的变动
  - 3 级 DOM, Xerces 替换了 Crimson
- 更多的信息参见
  - http://java.sun.com/j2se/1.5.0/compatibility.html

总结以及资源



资源

- www.jcp.org
  - JSR-014 泛型 (Generics)
  - JSR-166 多线程工具 (Concurrency utilities)
  - JSR-175 注解 (Annotation)
  - JSR-201 枚举类型，For循环，静态导入
- java.sun.com/j2se
- java.sun.com/j2se/1.5.0/compatibility.html
- j2se.dev.java.net
- www.netbeans.org

谢谢！

Sun Microsystems