



J2SE 5.0 Update: The Roar Of The Tiger

Simon Ritter

Technology Evangelist



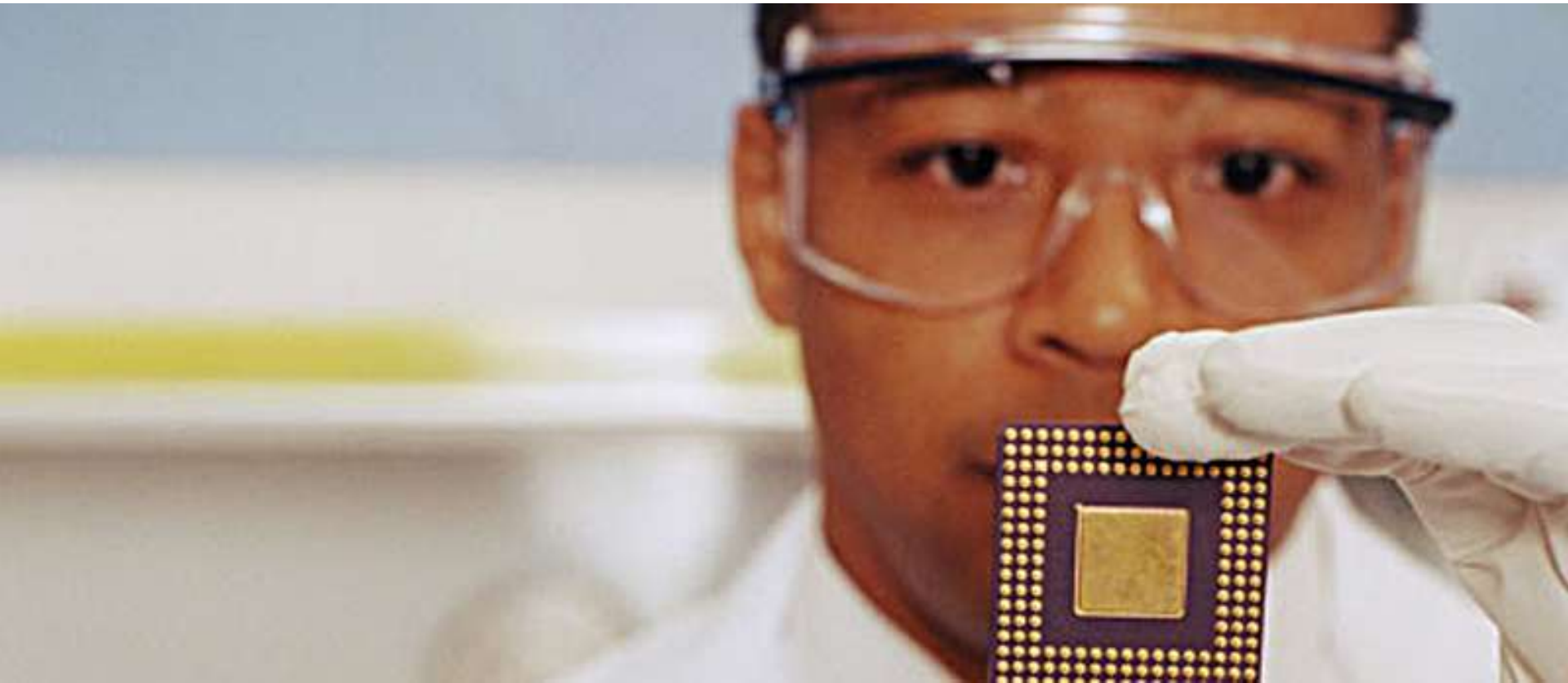
Agenda

- J2SE 5.0 Design Themes
- Language/Compiler Changes
 - Generics & Metadata
- Library API Changes
 - Concurrency utilities
- Virtual Machine
 - Monitoring & performance
- NetBeans for development
- Conclusions and Resources

J2SE 5.0 Design Themes

- Focus on quality, stability, compatibility
 - customers need rock solid releases
- Support a wide range of application styles
 - “from desktop to data centre”
- Big emphasis on scalability
 - exploit big heaps, big I/O, big everything
- Continuing to deliver great new features
 - In an evolutionary way
- Ease of development
 - Faster, cheaper, more reliable

Language Changes



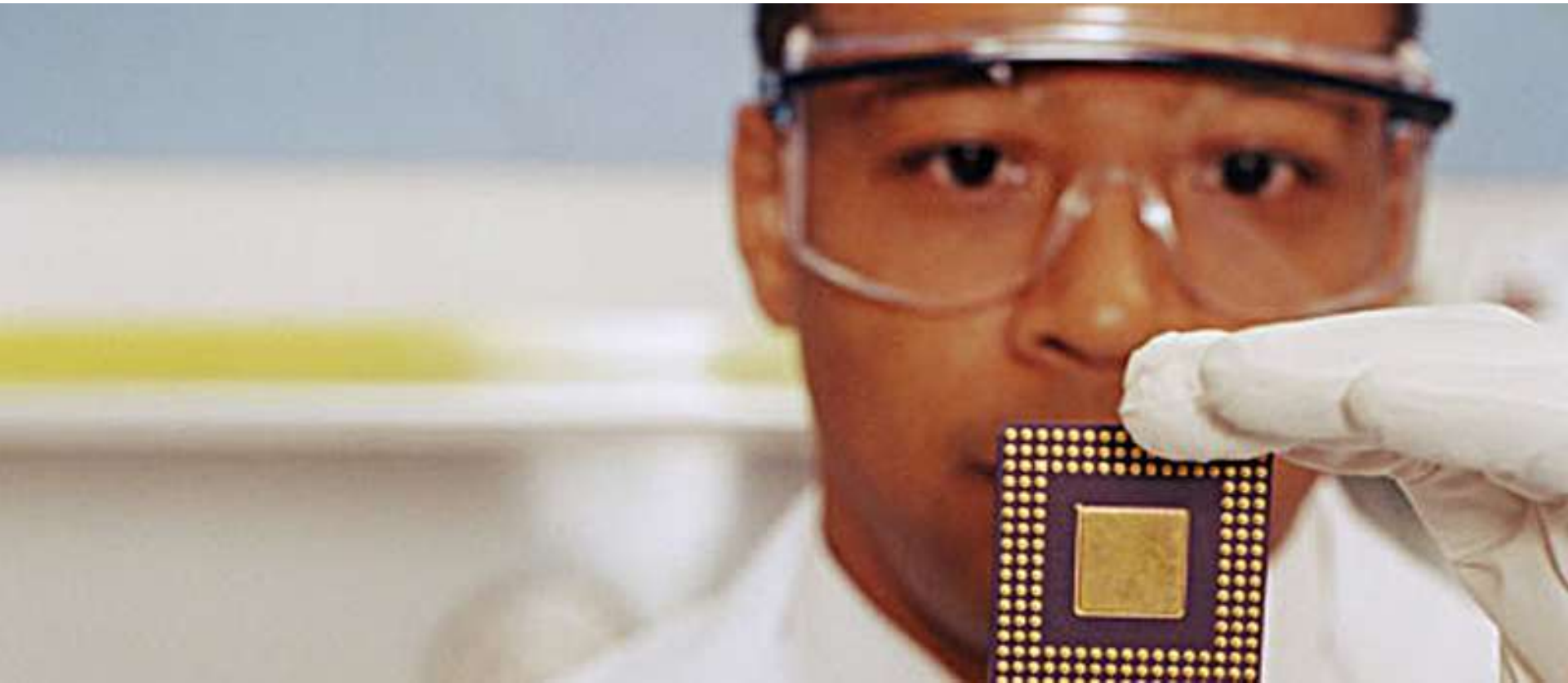
Java Language Changes

- JDK 1.0
 - Initial language, very popular
- JDK1.1
 - Inner classes, new event model
- JDK 1.4
 - Assertions (minor change)
- JDK 5.0
 - Biggest changes to language since release 1.0

Seven Major New Features

- Generics
- Autoboxing/Unboxing
- Enhanced for loop (“foreach”)
- Type-safe enumerations
- Varargs
- Static import
- Metadata

Generics



The Problem (Pre-J2SE 5.0)

```
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();  
    }  
}
```

Generics

- Problem: Collection element types
 - Compiler is unable to verify types
 - Assignment must use the cast operator
 - This can generate runtime errors
(**ClassCastException**)
- Solution:
 - Tell the compiler what type the collection is
 - Let the compiler fill in the cast
 - Guaranteed to succeed *

Generic Class Instantiation

```
ReferenceType TypeArgumentsopt Identifier =  
new ReferenceType TypeArgumentsopt ( ) ;
```

- ReferenceType must be a generic class to have TypeArguments specified
- A generic class instantiated without TypeArguments is called a raw type

Generic Type Arguments

TypeArguments ::= < TypeArgumentList >

TypeArgumentList ::= TypeArgument
TypeArgumentList , TypeArgument

ActualTypeArgument ::= ReferenceType | **Wildcard**

Wildcard ::= ? WildcardBounds_{opt}

WildcardBounds ::= **extends** ReferenceType
super ReferenceType

Using Generic Classes: 1

- Instantiate a generic class to create type specific object
- Example

```
Vector<String> x = new Vector<String>();  
x.add(new Integer(5)); // Compiler error!
```

```
Vector<Integer> y = new Vector<Integer>();  
return x.getClass() == y.getClass(); // ?
```

- Type argument cannot be Throwable or subclass of Throwable

Using Generic Classes: 2

- Class can have multiple type parameters
- Hashmap example:

```
HashMap<String, Mammal> map =  
    new HashMap<String, Mammal>();  
map.put("wombat",  
    new Mammal("wombat"));  
  
Mammal w = map.get("wombat");
```

Generics & Compatability

- Platform compatability, not language
- Raw types for existing code
 - Generic type instantiated with no type arguments

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

```
/* New class */  
public Vector<String> s = oldCode.getVector();  
/* Generates compiler warning, not error */
```

Generic Class Definition

ClassDeclaration ::= ClassModifier_{opt} **class** Identifier

TypeParameters_{opt} Super_{opt} Interfaces_{opt} ClassBody

TypeParameters ::= < TypeParameterList >

TypeParameterList ::= TypeParameterList, TypeParameter
 TypeParameter

TypeParameter ::= TypeVariable TypeBound_{opt}

TypeBound ::= **extends** ClassOrInterfaceType AdditionalBoundList_{opt}

AdditionalBoundList ::= AdditionalBound AdditionalBoundList
 AdditionalBound

AdditionalBound ::= **&** InterfaceType

Generic Classes

- Add type parameters in class definition
- Typically use one capital letter for types
- Use anywhere in class that type is required

```
public class Pair<F, S> {  
    F first;  S second;  
  
    public Pair(F f, S s) {  
        first = f;  second = s;  
    }  
}
```

Generic Classes

```

public class Seq<T> {
    T head;
    Seq<T> tail;

    public Seq(T h, Seq<T> t) { head = h; tail = t; }
    boolean isEmpty() { return tail == null; }

    class Zipper<S> {
        Seq<Pair<T,S>> zip(Seq<S> that) {
            if (this.isEmpty() || that.isEmpty())
                return new Seq<Pair<T,S>>();
            else
                return new Seq<Pair<T,S>>(
                    new Pair<T,S>(this.head, that.head),
                    this.tail.zip(that.tail));
        }
    }
}

```

Generic Classes

```
public class Client {
    Seq<String> strs =
        new Seq<String>("a",
            new Seq<String>("b", new Seq<String>()));

    Seq<Number> nums =
        new Seq<Number>(new Integer(1),
            new Seq<Number>(new Double(1.5),
                new Seq<Number>()));

    Seq<String>.Zipper<Number> zipper =
        strs.new Zipper<Number>();

    Seq<Pair<String, Number>> combined =
        zipper.zip(nums);
    ...
}
```

Wildcards

- Method to print contents of any Collection?

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

Wildcards

- Method to print contents of any Collection?

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

- Wrong!
- Passing a Collection of type String will give a compiler error
 - A Collection with type parameter Object is NOT the supertype of all types of Collections

Wildcards

- Correct way:

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

- ? is the wildcard type
- Collection<?> means Collection of unknown

Bounded Wildcards

- Parameterised types do not have inheritance like objects
- A wildcard can be specified with an upper bound

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

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

Generic Methods

- Problem: How to make method generic
- Wrong solution:

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

- Remember ? means unknown type, not any type

Generic Methods

- Correct solution
 - Make method generic with type parameter
 - Compiler infers type when method called

```
static <T> void aToC(T[] a, Collection<T> c) {  
    for (T o : a)  
        c.add(o);    /* No compiler error */  
}
```

```
String[] sa = new String[100];  
Collection<Object> co = new ArrayList<Object>();  
Collection<String> cs = new ArrayList<String>();  
aToC(sa, cs);    /* T inferred to be String */  
aToC(sa, co);    /* T inferred to be Object */
```

Generics With Bound Types

- Interfaces can be generic

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

- Type can be bound to interface

```
class Byte implements Comparable<Byte> {  
    public int compareTo(Byte that) {  
        return this.value - that.value;  
    }  
}
```

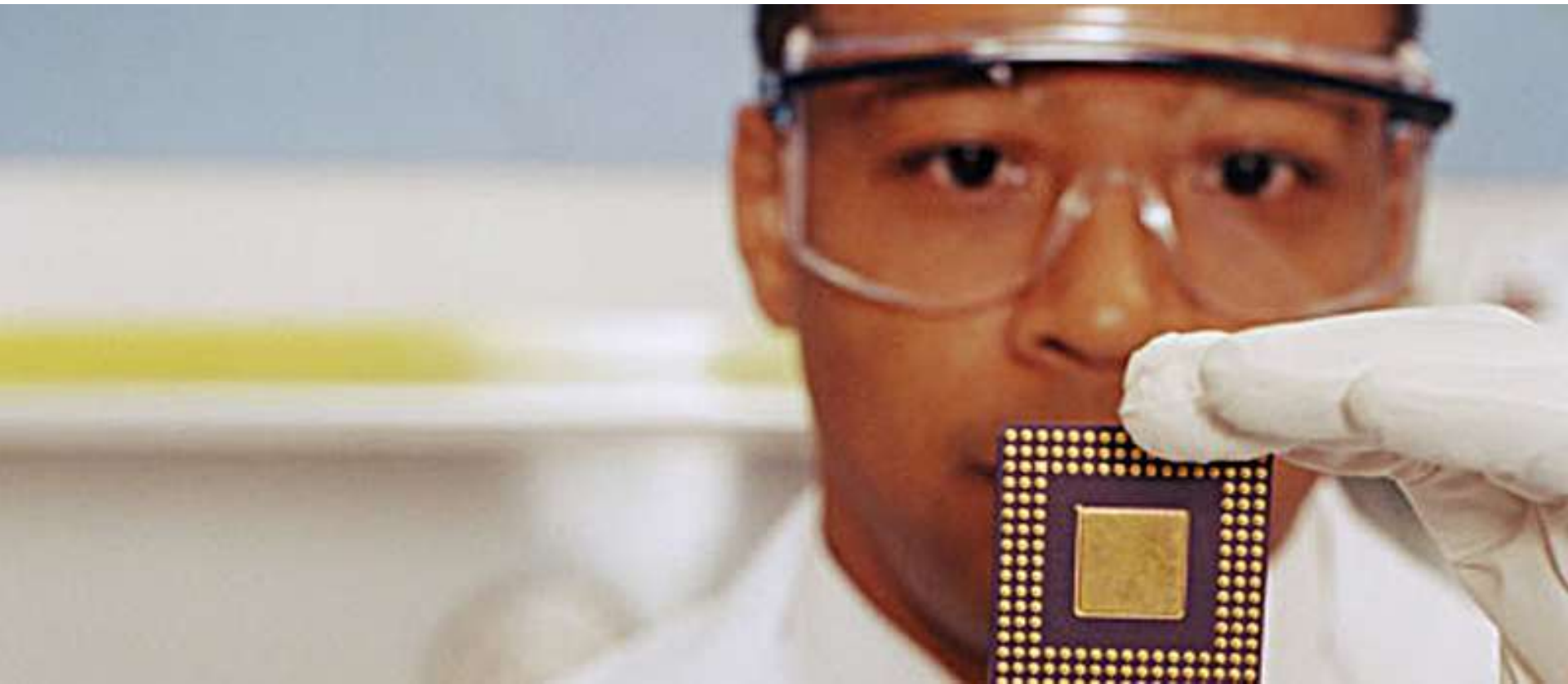
```
class Collections {  
    public static <A implements Comparable<A>> A  
        max(Collection <A> c) { ... }
```

Bounded Wildcards (Again)

- Wildcards do not use type inference
- Wildcards can have lower bounds
- Example: generic copy

```
class CopyUtil {  
    public <T>void copy(List<T> src,  
                        List<? super T> dest) {  
        for (T o : src)  
            dest.add(o);  
    }  
}
```

Auto-Boxing, Enhanced For Loop, Enumerations & More



Autoboxing of Primitive Types

- Problem:
 - Conversion between primitive types and wrapper objects (and vice-versa)
 - Needed when adding primitives to a collection
- Solution: Let the compiler do it

```
Byte byteObj = 22;           // Boxing conversion  
int i = byteObj             // Unboxing conversion
```

```
ArrayList al = new ArrayList();  
al.add(22); // Boxing conversion
```

Enhanced for Loop (foreach)

- Problem:
 - Iterating over collections is tricky
 - Often, iterator only used to get an element
 - Iterator is error prone
(Can occur three times in a for loop)
 - Can produce subtle runtime errors
- Solution: Let the compiler do it
 - New for loop syntax
for (variable : collection)
 - Works for Collections and arrays

Enhanced for Loop Example

- Old code

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

- New Code

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

Type-safe Enumerations

- Problem:
 - Variable needs to hold limited set of values
 - e.g. card suit can only be spade, diamond, club, heart
- Solution: New type of class declaration
 - enum type has public, self-typed members for each enum constant
 - New keyword, enum
 - Works with switch statement

Enumeration Example: 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);
```

Think how much JDK1.4 code this would require!

Enumeration Example: 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;
    }
}
```

Enumeration Example: 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();
}
```

Varargs

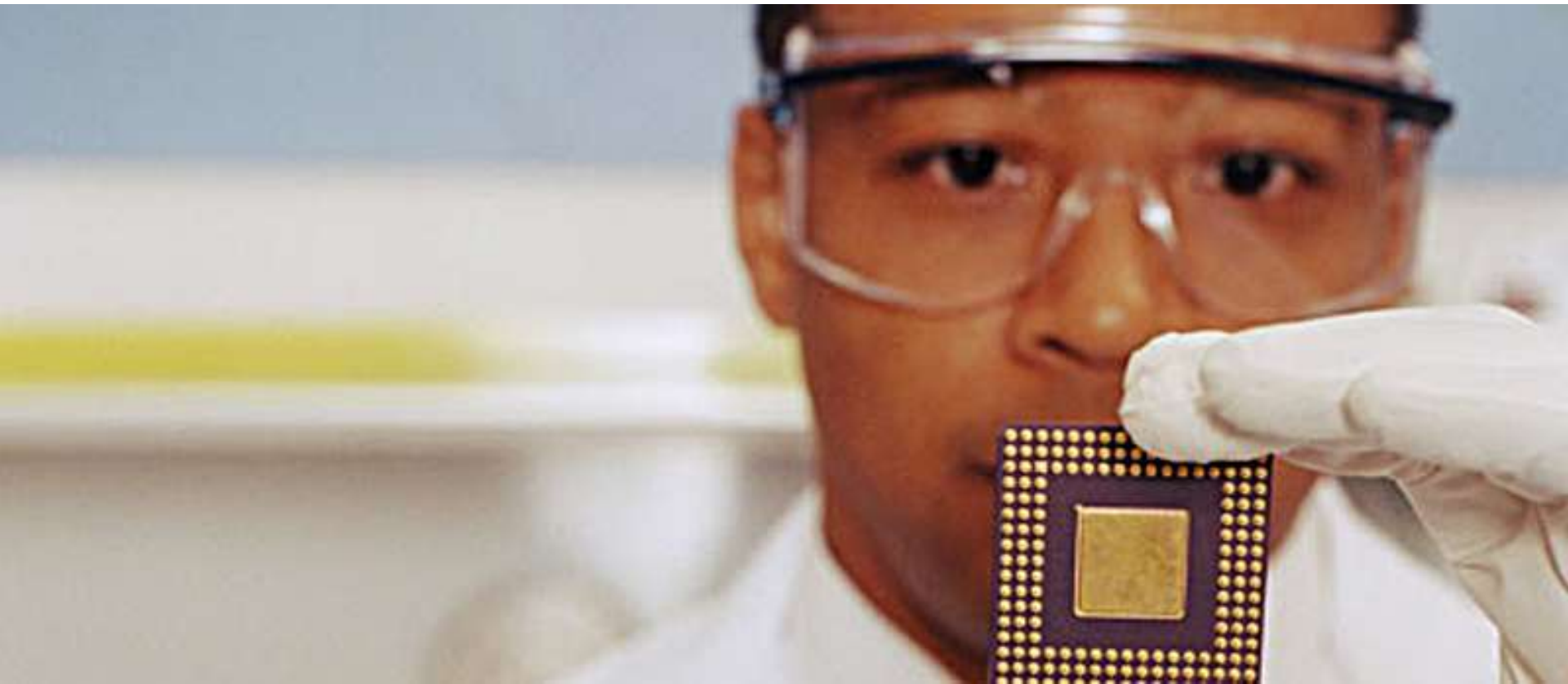
- Problem:
 - To have a method that takes a variable number of parameters
 - Can be done with an array, but not nice
 - Look at `java.text.MessageFormat`
- Solution: Let the compiler do it for you
 - New syntax:

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

Static Imports

- Problem:
 - Having to fully qualify every static referenced from external classes
- Solution: New import syntax
 - `import static TypeName.Identifier;`
 - `import static Typename.*;`
 - Also works for static methods and enums
e.g `Math.sin(x)` becomes `sin(x)`

Metadata



Metadata (JSR-175)

- Provide standardised way of adding annotations to Java code
- Like **Serializable** interface, javadoc comments and Xdoclets, but better
- Annotations are used by tools that work with Java code:
 - Compiler
 - IDE
 - Runtime tools

Defining Annotations

AnnotationType ::= @ interface Identifier AnnotationBodyType

AnnotationBodyType ::= { AnnotationElements_{opt} }

AnnotationElements ::= AnnotationElements AnnotationElement
AnnotationElement

AnnotationElement ::=
 AbstractMethodModifiers_{opt} Type Identifier () DefaultValue_{opt}
 ConstantDeclaration
 ClassDeclaration
 InterfaceDeclaration
 EnumDeclaration
 AnnotationType
 ;

DefaultValue ::= default ElementValue

Defining Annotations

- Defined like an interface
 - @interface
- Definition contains parameters that can be specified when using the annotation
- Default values can be provided
- For annotations with one parameter, the name 'value' is special

Meta-Annotations

- **@Retention**
 - How long is annotation information kept
 - **Enum RetentionPolicy**
 - **SOURCE, CLASS, RUNTIME**
- **@Target**
 - Restrictions on use of this annotation
 - **Enum ElementType**
 - **TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, ANNOTATION_TYPE, PACKAGE**

Annotation Example

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

Annotating Code

- Marker annotation
 - `public int @BetaVersion getValue()`
- Single value annotation
 - `@Copyright(value = "Sun Microsystems")`
 - `@Copyright("Sun Microsystems")`
 - This can only be used if member is called value
- Normal annotation
 - `@Author(@Name(first="fred",
last="bloggs"))`
 - `@Contributors({"fred", "joe", "bill"})`

Reflection and Metadata

- Marker annotation

```
boolean isBeta =  
    MyClass.class.isAnnotationPresent  
        (BetaVersion.class);
```

- Single value annotation

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

- Normal annotation

```
Name author = MyClass.class.getAnnotation  
    (Author.class).value();
```

```
String first = author.getFirst();
```

```
String last = author.getLast();
```

Metadata Example: JAX-RPC

- Old Code

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

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

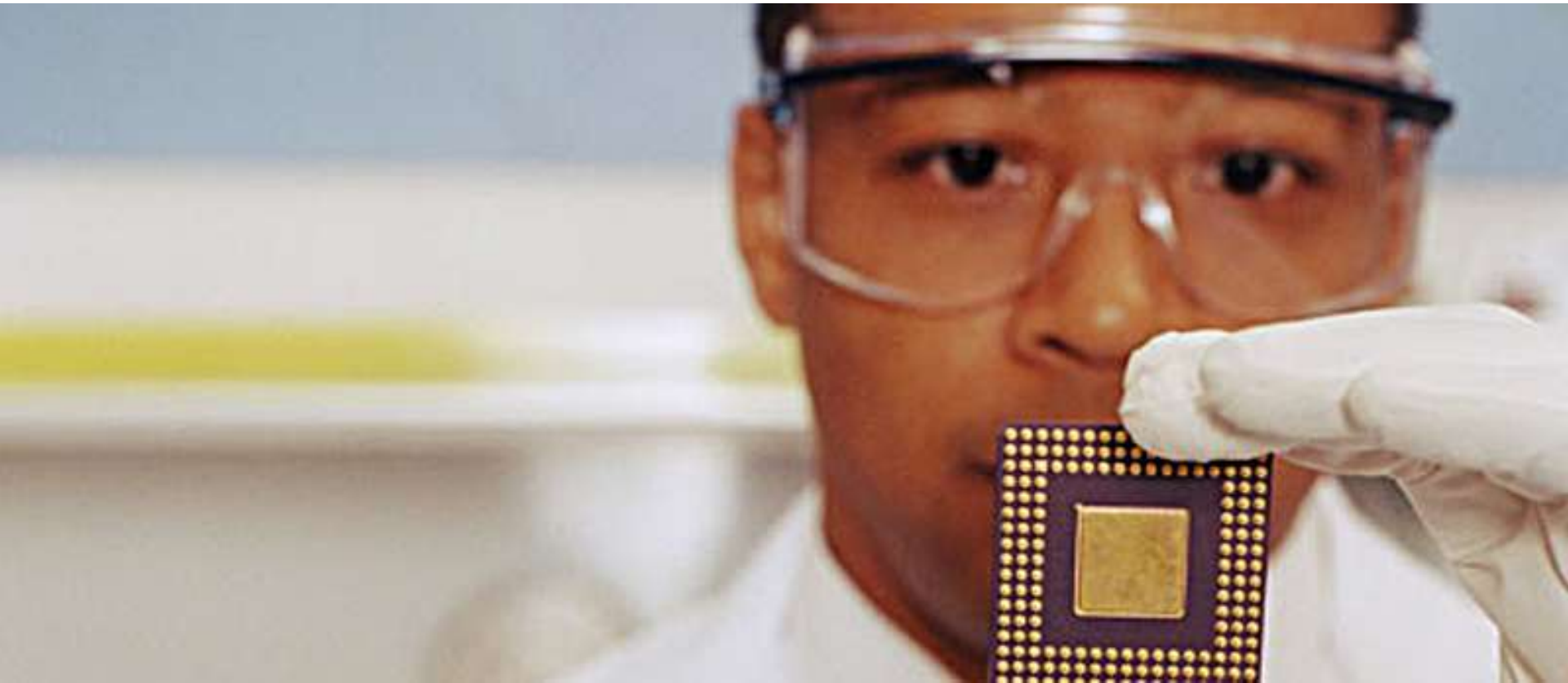
- New Code

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

Annotation Processing Tool

- Annotated base file generates derived files
 - **apt** is tool to generate derived files
 - Derived files can be new source files, deployment descriptor, etc
- Uses annotation processors
 - Processors can be developed using the **com.sun.mirror** packages
 - **process()** method processes specific annotations
 - Access to **File** to create derived files
- **apt** can compile the derived files if required

Library Changes



Simple Formatted I/O & Scanner

- Printf is popular with C/C++ developers
 - Powerful, easy to use
- Finally adding printf to J2SE 5.0!

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

`%n` is platform independant newline
- Also a simple scanning API: convert text into primitives or Strings

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

Concurrency Utils: JSR-166

- Goals
 - Do for concurrency what Collection did for data structures
 - Beat C performance in high-end server applications
 - Provide a set of basic concurrency building blocks
 - **wait()**, **notify()** and **synchronized** are too primitive
 - Enhance scalability, performance, readability and thread safety of Java applications

Concurrency Utilities: JSR-166

- Executor interface to replace direct use of Thread
 - **ExecutorService** interface
- Executors factory utility class
 - **ThreadPool**
 - **SingleThreadPool**
 - **PriorityThreadPool**
- **Callable** and **Future**
- **Semaphore**
- **BlockingQueue**
- **Atomic**

Executors

- Don't use
`new Thread(Runnable r).start();`
 - Create an **Executor** and call `execute()`
- This provides a clean way to change the mechanism without changing lots of code
 - **Executor** can come from a **ThreadPool**, a **PriorityThreadPool** or **SingleThreadExecutor**

Executors

- Simplified way to shutdown threads
 - `pool.shutdown();`
- Executors class provides factory methods
 - `newSingleThreadExecutor();`
 - `newFixedThreadPool(int size);`

Thread Pool Example

```
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);
                }
            });
        }
    }
}

class Handler { void process(Socket s); }
```

Callables and Futures

- **Callable** interface provides way to get a result or **Exception** from a separate thread
 - implement `call()` method rather than `run()`
- **Callable** is submitted to **Executor**
 - call `submit()` not `execute()`
 - returns a **Future** object
- When result is required retrieve using `get()` method of **Future** object
 - If result is ready it is returned
 - If result is not ready calling thread will block

Callable Example

```
class CallableExample implements
    Callable<String> {

    public String call() {
        String result = null;

        /* Do some work and create a result */

        return result;
    }
}
```

Future Example

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

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

```
/* Do some work in parallel */
```

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

Locks

- Lock interface
 - More extensive locking operations than synchronized
 - No automatic unlocking
 - Non-blocking access using **tryLock()**
- ReentrantLock
 - Concrete implementation of Lock
 - Holding thread can call **lock()** multiple times and not block

ReadWriteLock

- Has two locks controlling read and write access
 - Locks are inner classes
 - Multiple threads can acquire the read lock if no threads have a write lock
 - Only one thread can acquire the write lock
 - Methods to access locks

```
rw1.readLock().lock();  
rw1.writeLock().lock();
```

Semaphores

- Typically used to restrict access to fixed size pool of resources
- New Semaphore object is created with same count as number of resources
- Thread trying to access resource calls **acquire()**
 - Returns immediately if semaphore count > 0
 - Blocks if count is zero until **release()** is called by different thread
 - **acquire()** and **release()** are thread safe atomic operations

Semaphore Example

```
private Semaphore available;  
private Resource[] resources;  
private boolean[] used;  
  
public Resource(int poolSize) {  
    available = new Semaphore(poolSize);  
    /* Initialise resource pool */  
}  
public Resource getResource() {  
    try { available.acquire() } catch (IE) {}  
    /* Return resource */  
}  
public void returnResource(Resource r) {  
    /* Return resource to pool */  
    available.release();  
}
```

Blocking Queue

- Provides thread safe way for multiple threads to manipulate collection
- `ArrayBlockingQueue` is simplest concrete implementation
- Full set of methods
 - `put()`
 - `offer()` [non-blocking]
 - `peek()`
 - `take()`
 - `poll()` [non-blocking and fixed time blocking]

Blocking Queue Example: 1

```
private BlockingQueue<String> msgQueue;

public Logger(BlockingQueue<String> mq) {
    msgQueue = mq;
}

public void run() {
    try {
        while (true) {
            String message = msgQueue.take();
            /* Log message */
        }
    } catch (InterruptedException ie) {
        /* Handle */
    }
}
```

Blocking Queue Example: 2

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

```
Logger logger = new Logger(messageQueue);
```

```
public void run() {  
    String someMessage;  
    try {  
        while (true) {  
            /* Do some processing */  
  
            /* Blocks if no space available */  
            messageQueue.put(someMessage);  
        }  
    } catch (InterruptedException ie) { }  
}
```

Atomics

- **java.util.concurrent.atomic**
 - Small toolkit of classes that support lock-free thread-safe programming on single variables
- Not guaranteed to be non-blocking
 - Newer processors have compare-and-set instructions built in

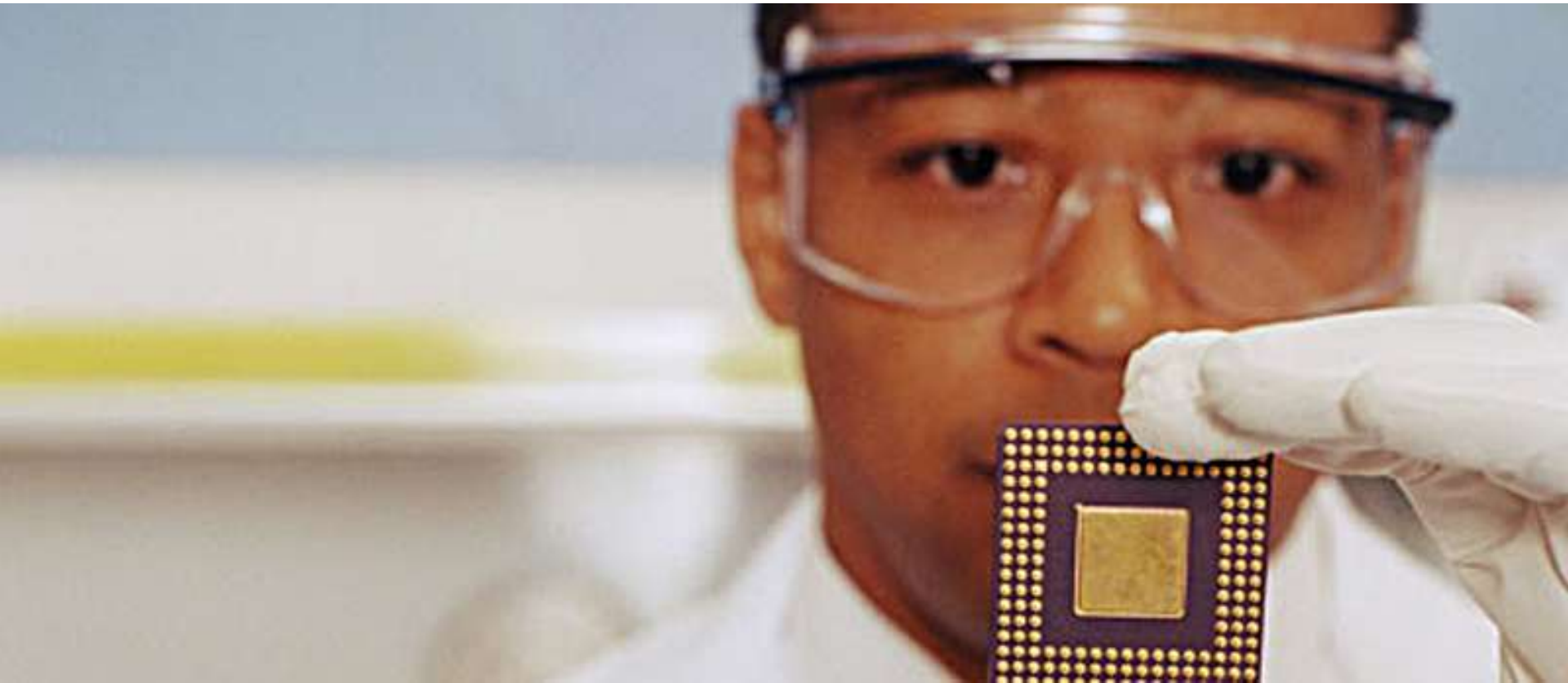
```
AtomicInteger balance = new AtomicInteger(0);
```

```
public int deposit(integer amount) {  
    return balance.addAndGet(amount);  
}
```

Desktop Client

- Additional accessibility support
- Continue to improve deployment
 - Upgrades to Java Plug-in and Java WebStart
- Continued quality & performance work
 - Java2D will exploit OpenGL
 - XAWT: lightweight AWT on Xlib
- Improved GUI look-and-feel support
 - Gnome skins support
 - Better Windows look and feel
 - Revised (but compatible!) Java look and feel

Virtual Machine



Main VM Changes

- Class data sharing
 - Improved startup time (up to 30% faster)
 - Reduced memory footprint
 - -Xshare:on, -Xshare:dump
- Thread priority changes (JSR-133)
- Fatal error handling
 - -XX:OnError="gcore %p; dbx - -%p"
- Simplified stack trace access
 - **Thread.getStackTrace()**

Server Class Machine

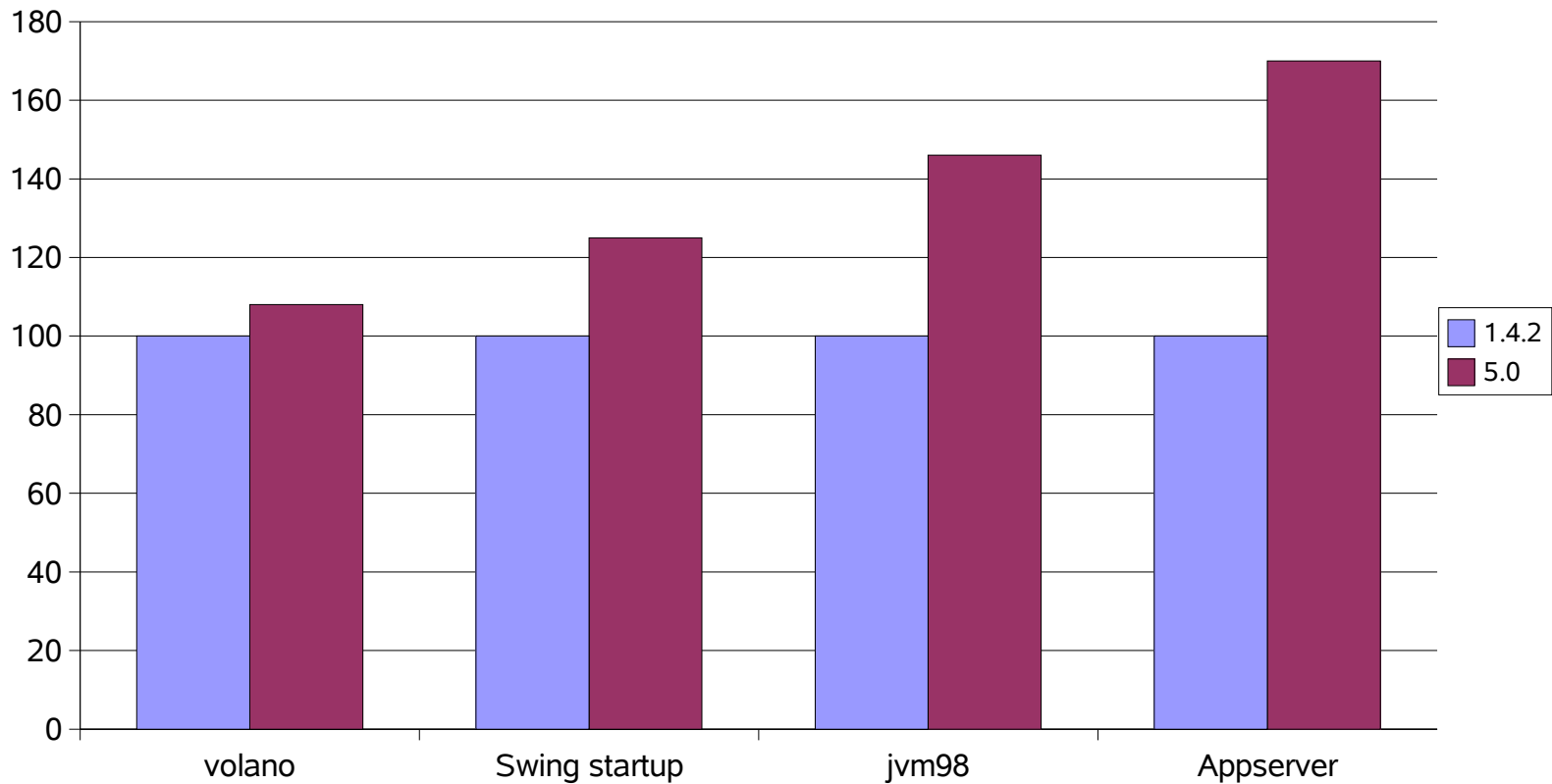
- Auto-detected
- 2 CPU, 2GB mem (except windows)
 - Uses server compiler
 - Uses parallel collector
 - Initial heap size is 1/64 of physical memory up to 1GB
 - Max heap size is 1/4 of physical memory up to 1GB

JVM Self Tuning (Ergonomics)

- Maximum pause time goal
 - `-XX:MaxGCPauseMillis=<n>`
 - This is a hint, not a guarantee
 - GC will adjust parameters to try and meet goal
 - Can adversely effect applicaiton throughput
- Throughput goal
 - `-XX:GCTimeRatio=<n>`
 - Percentage GC Time = $1 / (1 + n)$
 - e.g. `-XX:GCTimeRatio=19` (5% of time in GC)

Performance Improvement

Solaris Sparc



Monitoring & Management

- Key component of RAS in the Java platform (Reliability, Availability, Serviceability)
- Features
 - JVM instrumentation and integrated JMX
 - Monitoring and management APIs
 - Tools
- JVMTI replaces JVMPI
 - Improved performance analysis
 - Bytecode level instrumentation

Integrated JMX (JSR-003)

- Standard way of instrumenting
 - Mbean server built into JVM
 - Switch on when starting JVM
 - MXBeans for different JVM functionality
- SNMP monitoring built in
- Works with existing J2EE app servers
- Support for remote management (JSR-160)

Monitoring & Management APIs

- `java.lang.management`
- MXBeans in JVM
- API access to
 - number of classes loaded, threads running
 - Thread state, contention stats, stack traces
 - GC statistics
 - memory consumption, low memory detection
 - VM uptime, system properties, input arguments
 - On-demand deadlock detection

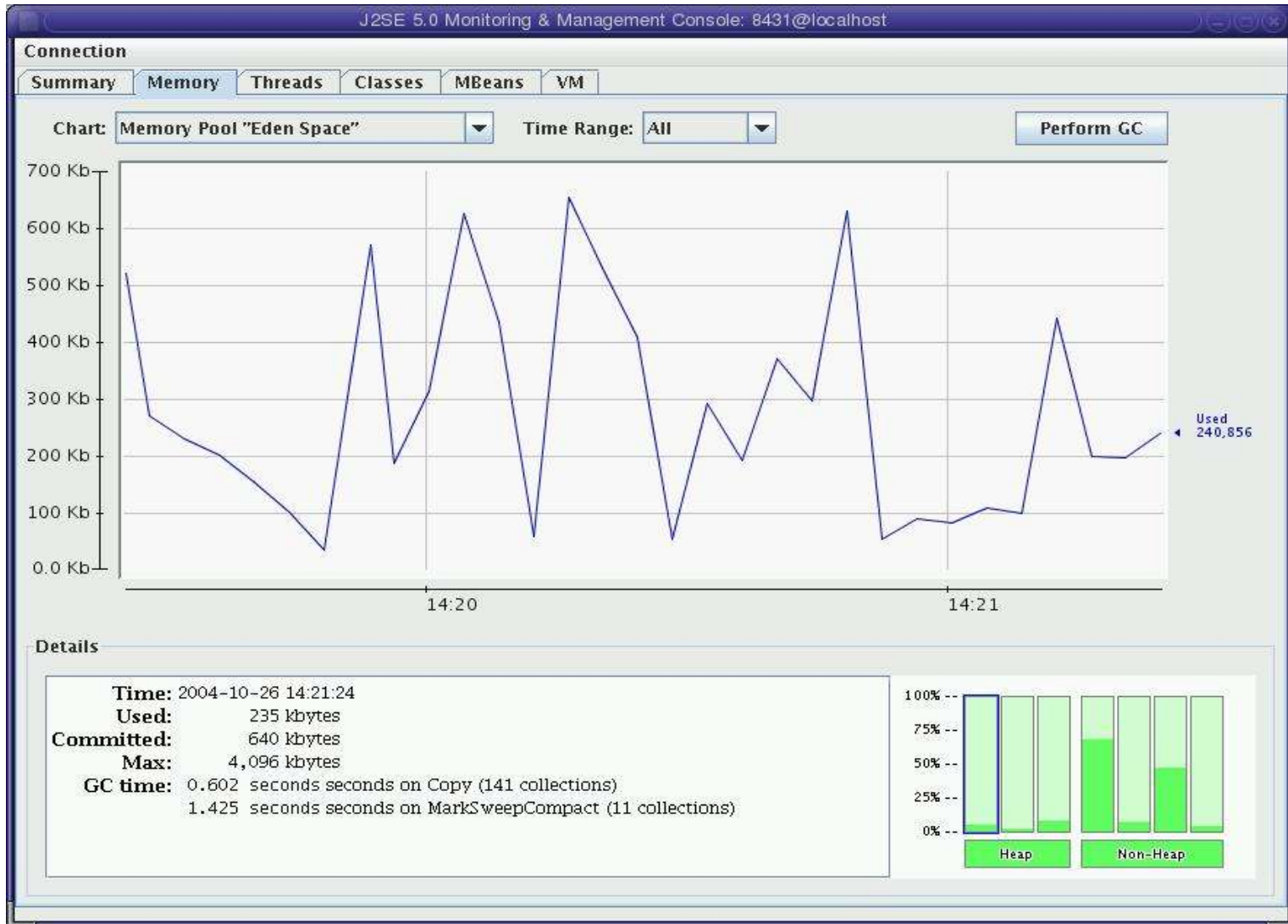
Monitoring Tools

- jinfo
 - Config info from running JVM or crash dump
- jmap
 - Lots of useful memory information on running JVM or crash dump
- jstack
 - Prints stack traces for all threads in running VM or stack trace
- These are not supported on Windows

Monitoring Tools

- `jstat`
 - Statistics of GC, dynamic compilation, class loader
 - Not available on Windows 98, ME
 - Not available on Windows NT, 2000, XP if using FAT32 filesystem
- `jps`
 - Process ids of all instrumented HotSpot VMs

Monitoring Tools: jconsole



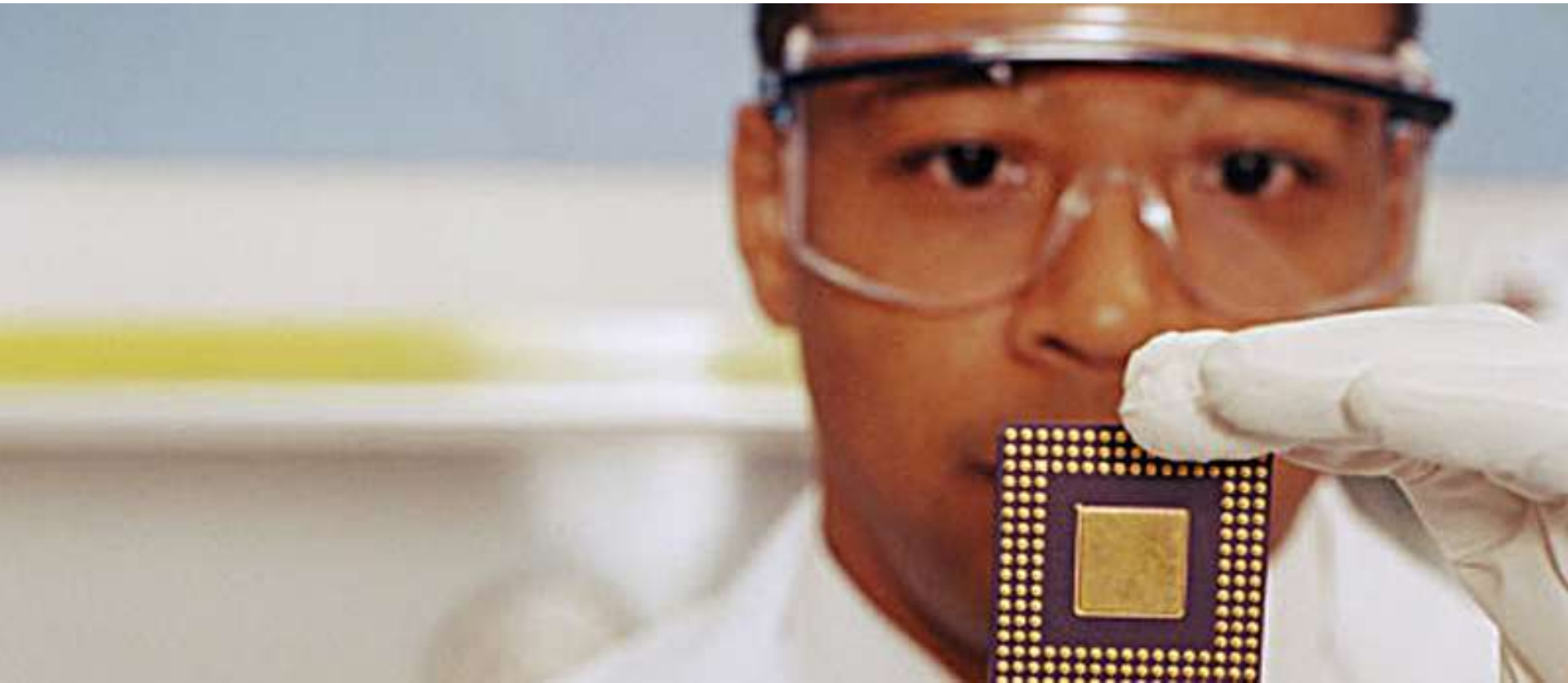
Compatability and Porting

- J2SE 5.0 classes will not run on earlier JVMs
- **enum** is now a keyword
 - Cannot be used as a variable identifier
- `java.net.Proxy`
 - Can clash with `java.lang.reflect.Proxy`
- JAXP minor differences
 - DOM Level 3, Xerces rather than Crimson
- For full details see
 - java.sun.com/j2se/1.5.0/compatability.html

J2SE: The Future

- J2SE 6: Codename “Mustang”
- More community based development
 - j2se.dev.java.net
 - Reference implementation still created through JCP
- Source code released under Java Research License
 - Designed for universities and researchers
 - Simpler and more relaxed terms than SCSL
- Get involved!

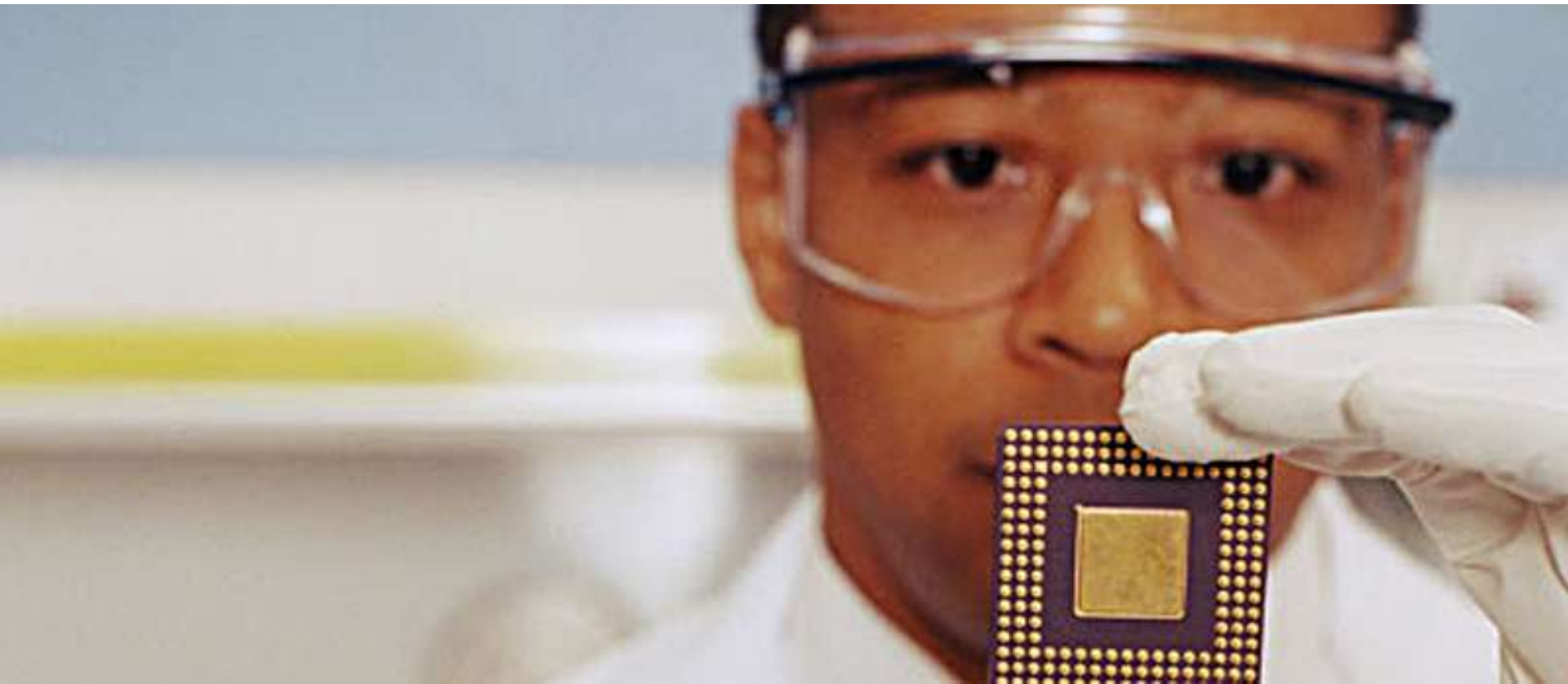
New in NetBeans



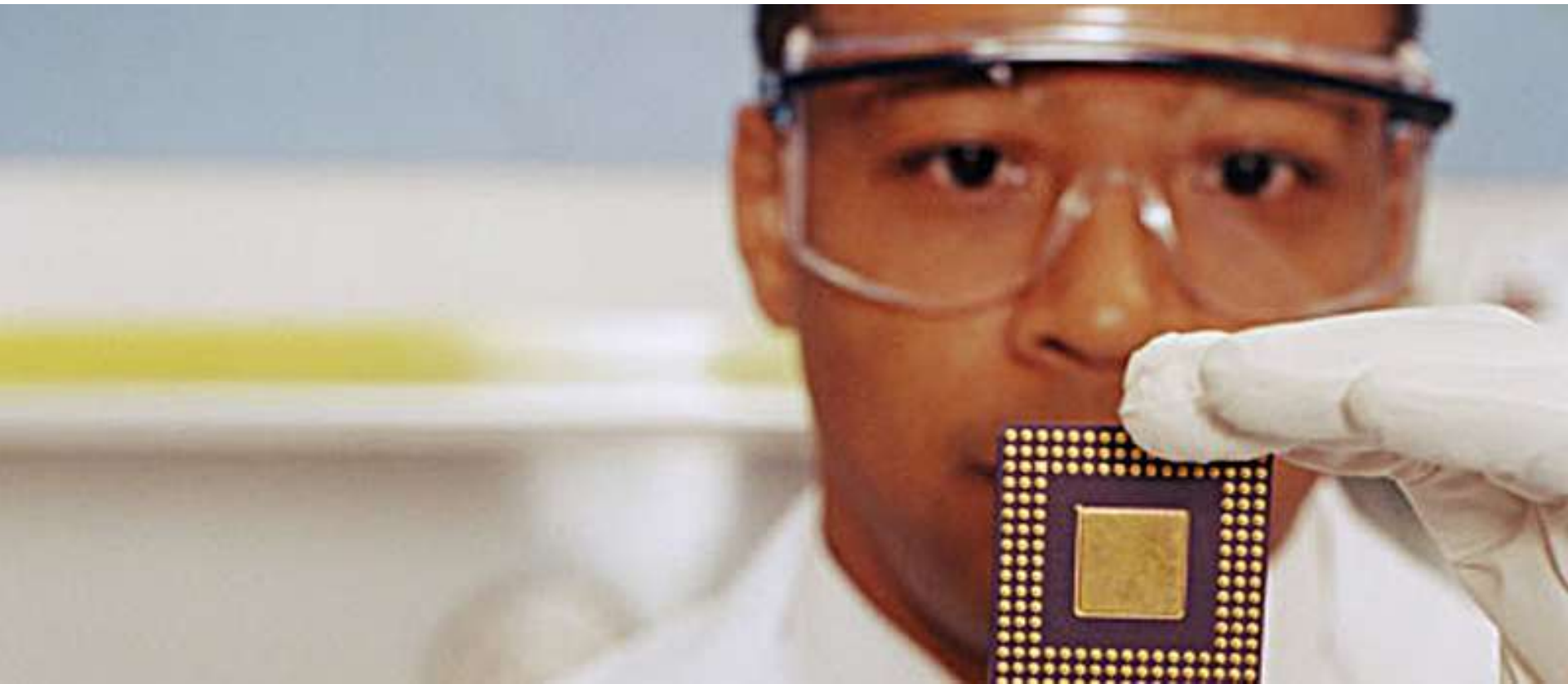
NetBeans 4.0 Highlights

- Full support for J2SE 5.0
 - Colouring of new syntax
- Ant based project system
- Refactoring
- Web based application improvements
- Mobility
 - Support for MIDP 2.0 development

Demo



Conclusions & Resources



J2SE 5.0

- Over one hundred new features
- Lots of power
- Take time to learn the details
- Use the new features wherever you can
- Upgrade to the latest JVM wherever possible

Resources

- www.jcp.org
 - JSR-014 Generics
 - JSR-166 Concurrency utilities
 - JSR-175 Metadata facility
 - JSR-201 Enums, Autoboxing, For loop, Static import
- java.sun.com/j2se
- java.sun.com/j2se/1.5.0/compatibility.html
- j2se.dev.java.net
- www.netbeans.org



J2SE 5.0 Update

simon.ritter@sun.com

