



# J2SE 5.0 Update: The Roar Of The Tiger

**Sang Shin**

Java Technology Architect

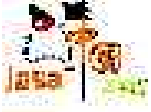
Sun Microsystems, Inc.





# Agenda

- J2SE 5.0 Design Themes
- Language Changes
  - > Generics & Metadata
- Library API Changes
  - > Concurrency utilities
- Virtual Machine
- Monitoring & Management
- Next Release: Mustang

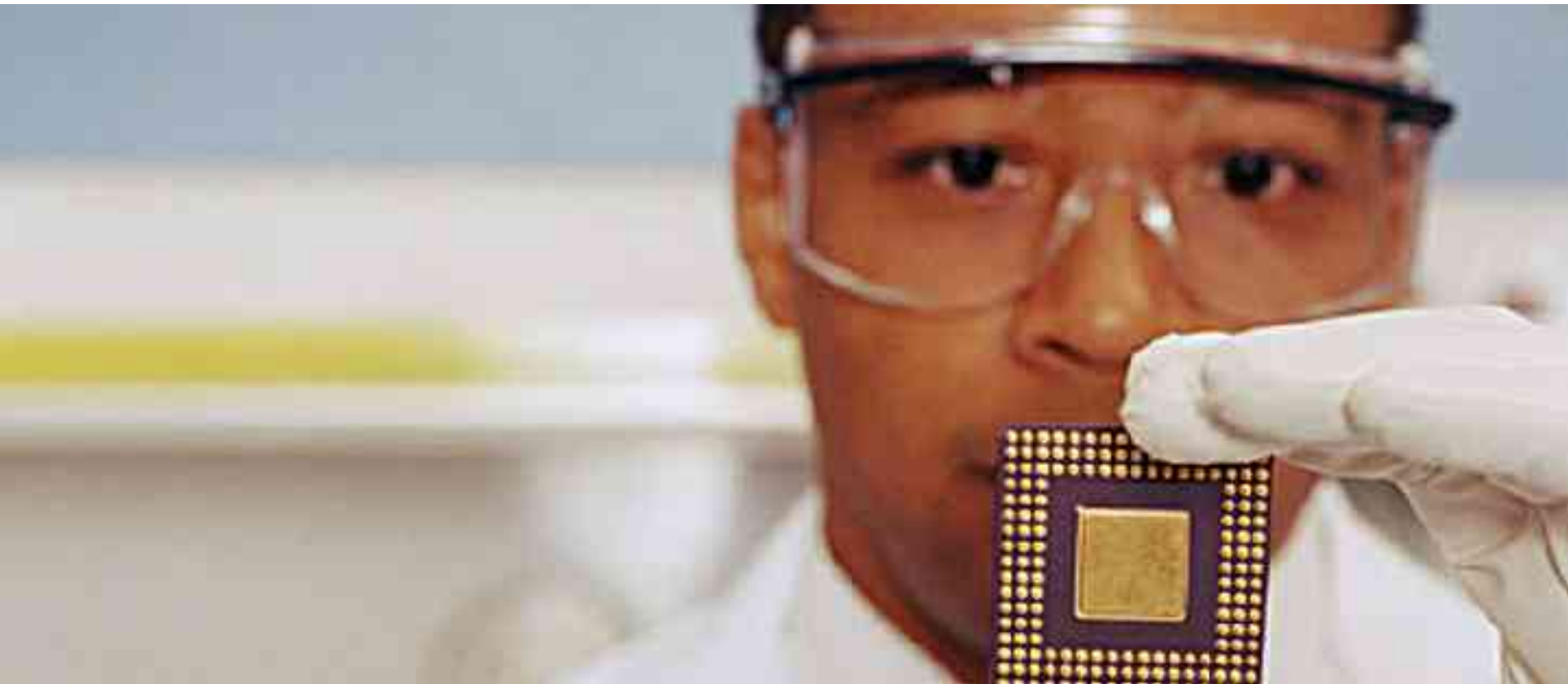


# J2SE 5.0 Design Themes

- Focus on quality, stability, compatibility
  - > Many enterprise software already run over J2SE 5.0
- Support a wide range of application styles
  - > “from desktop to data center”
- Big emphasis on scalability
  - > exploit big heaps, big I/O, big everything
- Continuing to deliver great new features
  - > Maintaining portability and compatibility
- Ease of development
  - > Faster, cheaper, more reliable



# Language Changes





# Java Language Changes

- JDK 1.0
  - > Initial language, very popular
- JDK 1.1
  - > Inner classes, new event model
- JDK 1.2, 1.3
  - > No changes at language level
- JDK 1.4
  - > Assertions (minor change)
- JDK 5.0
  - > Biggest changes to language since release 1.0

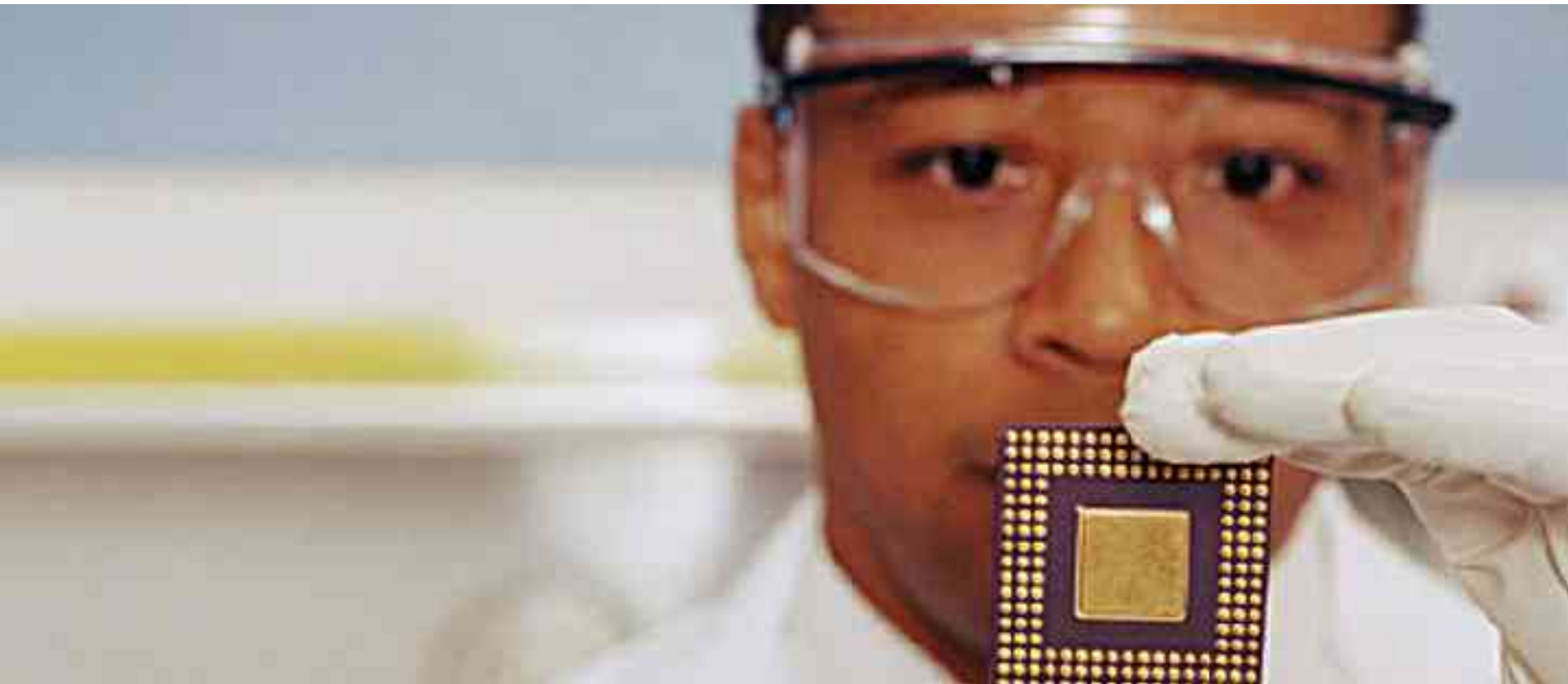


# Seven Major New Features

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



# Generics



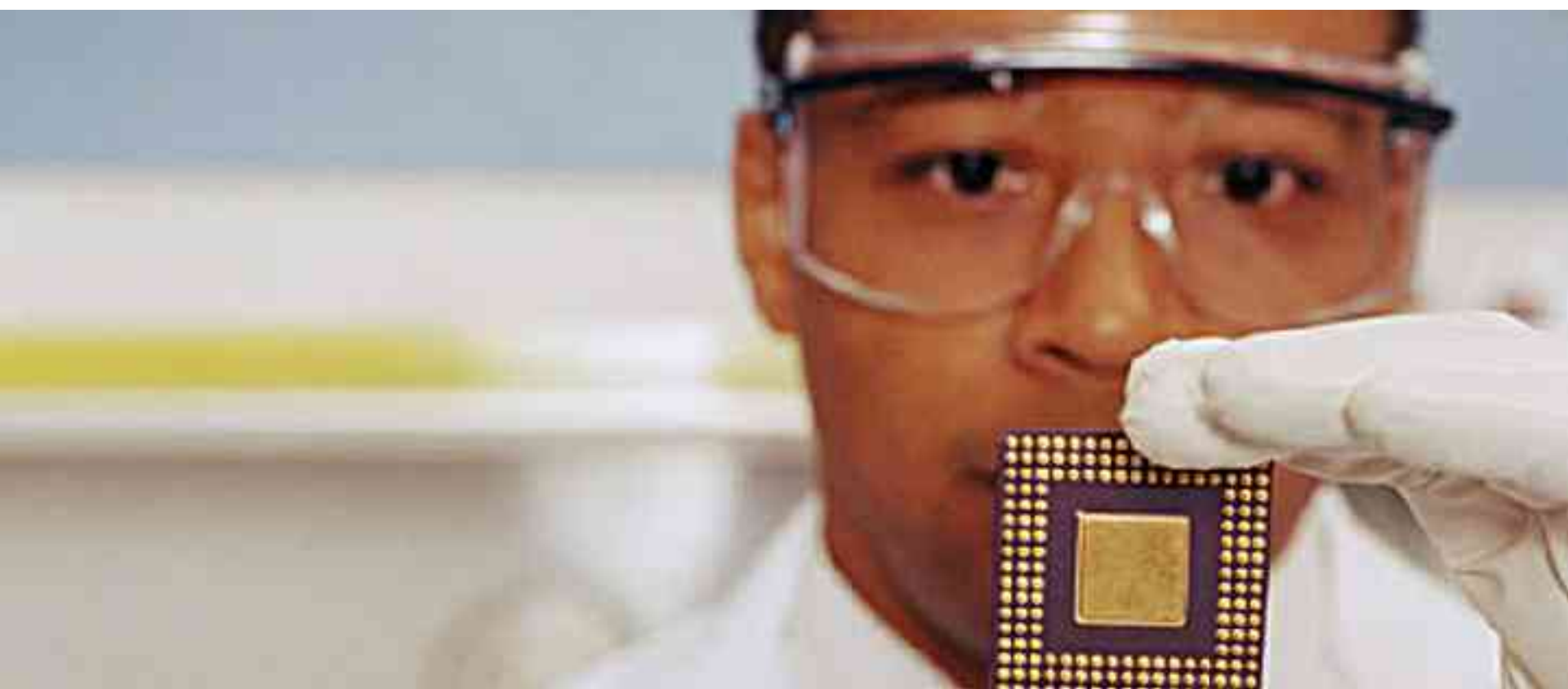


# Sub-topics of Generics

- What is and why Generics?
- Usage of Generics
- Generics and sub-typing
- Wildcard
- Defining your Generic class
- Type erasure



# Generics: What is and Why Generics?





# What is Generics?

- Generics provides abstraction over Types
  - > Classes, Interfaces and Methods can be **Parameterized** by **Types** (in the same way a Java type is parameterized by an instance of it)
- Generics makes **type safe code** possible
  - > If it compiles without any errors or warnings, then it must not raise any unexpected `ClassCastException` during runtime
- Generics provides increased readability



# Definition and Usage of Generic Class

- Definitions: LinkedList<E> has a type parameter E that represents the type of the elements stored in the list
- Usage: Replace **type parameter <E>** with concrete **type argument**, like **<Integer>** or **<MyType>**
  - > LinkedList<Integer> can store only Integer or sub-type of Integer as elements

```
LinkedList<Integer> li =  
    new LinkedList<Integer> ();  
li.add(new Integer(0));  
Integer i = li.iterator().next();
```



# Example: Definition and Usage of Parameterized List interface

```
// Definition of the Generic'ized
// List interface
interface List<E>{
    void add(E x);
    Iterator<E> iterator();
}

// Invocation (or usage) of List
// interface with concrete type parameter,
// String
List<String> ls = new ArrayList<String>(10);
```



# Why Generics? The Problem (Pre-J2SE 5.0) Code is not Type Safe

```
// Suppose you want to maintain String
// entries in a Vector.  By mistake,
// you add an Integer element.  Compiler
// does not detect this.  This is not
// type safe code.
```

```
Vector v = new Vector();
v.add(new String("valid string")); // intended
v.add(new Integer(4)); // unintended
```

...

```
// ClassCastException occurs during runtime
String s = (String)v.get(0);
```



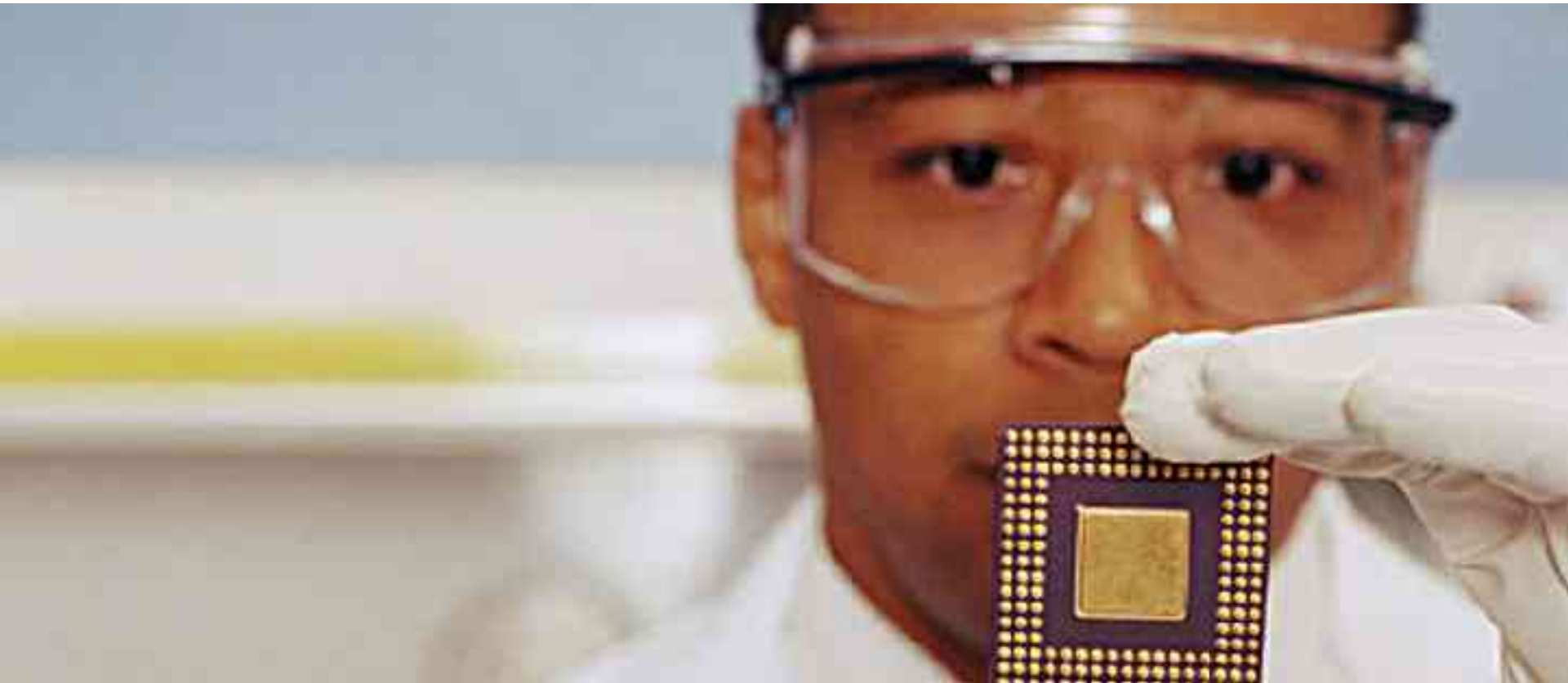
# What Problems does Generics Solve?

- Problem: Collection element types
  - > Compiler is unable to verify types
  - > Assignment must have type casting
  - > ClassCastException can occur during runtime
- Solution: Generics
  - > Tell the compiler type of the collection
  - > Let the compiler fill in the cast
  - > Example: Compiler will check if you are adding Integer type entry to a String type collection (**compile time detection of type mismatch**)



# Generics:

## Usage of Generics





# Using Generic Classes: 1

- Instantiate a generic class to create type specific object
- In J2SE 5.0, all collection classes are rewritten to be generic classes

```
Vector<String> vs = new Vector<String>();  
vs.add(new Integer(5)); // Compile error!  
vs.add(new String("hello"));  
String s = vs.get(0); // No casting needed
```



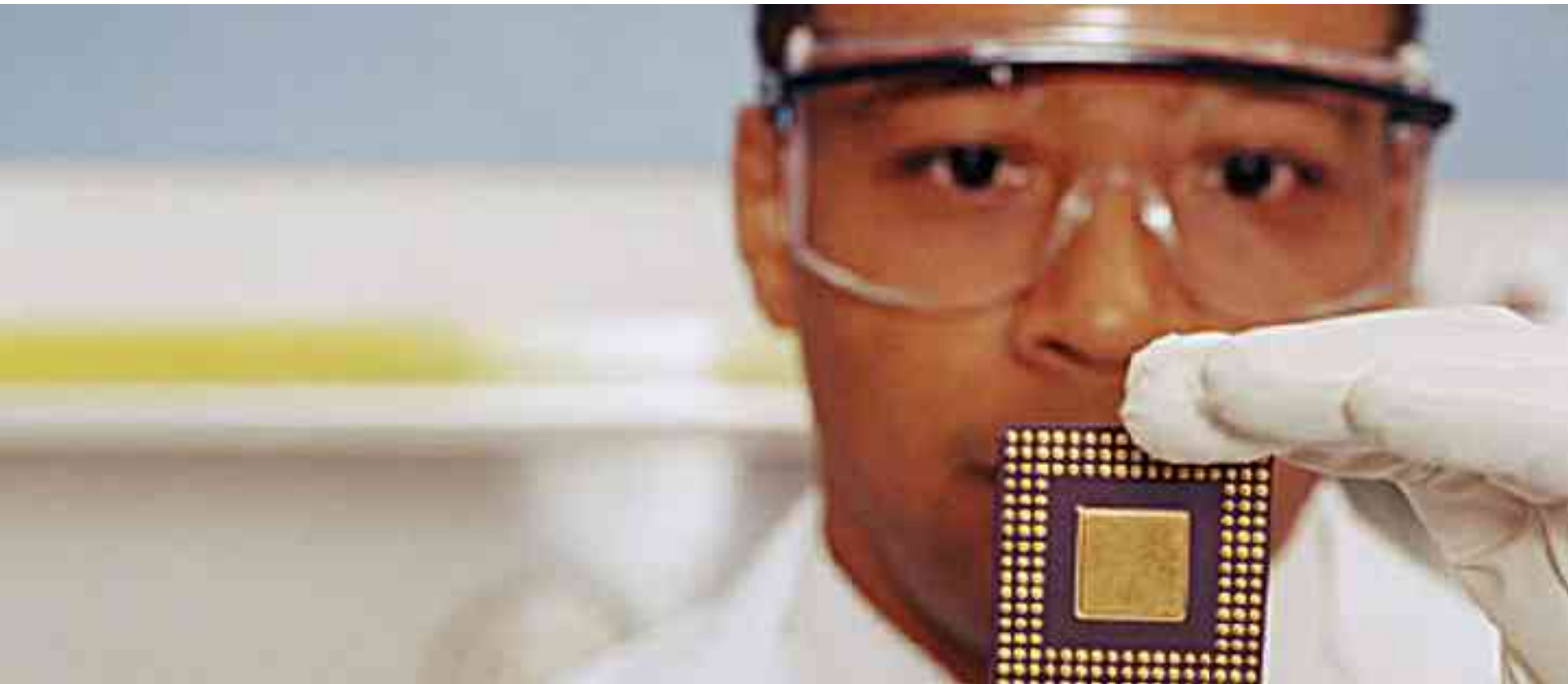
# Using Generic Classes: 2

- Generic class can have multiple type parameters
- Type argument can be a custom type

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

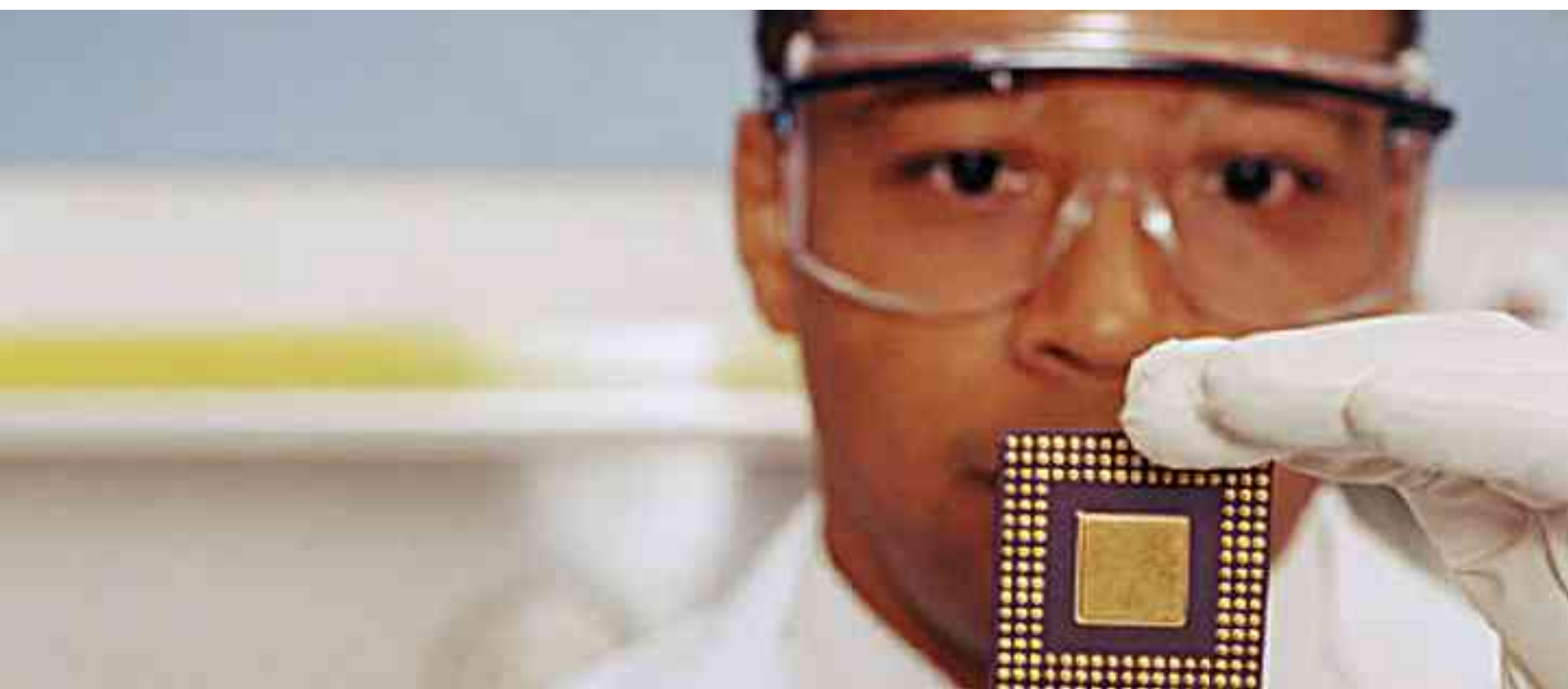


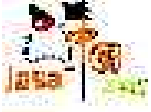
# Generics: Generics Demo using NetBeans IDE





# Generics: Sub-Typing





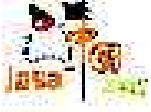
# Generics and Sub-typing

- You can do this (using pre-J2SE 5.0 Java)
  - > `Object o = new Integer(5);`
- You can even do this (using pre-J2SE 5.0 Java)
  - > `Object[] or = new Integer[5];`
- So you would expect to be able to do this (Well, you can't do this!!!)
  - > `ArrayList<Object> ao = new ArrayList<Integer>();`
  - > This is counter-intuitive at the first glance



# Generics and Sub-typing

- Why this compile error? It is because if it is allowed, `ClassCastException` can occur during runtime – **this is not type-safe**
  - > `ArrayList<Integer> ai = new ArrayList<Integer>();`
  - > `ArrayList<Object> ao = ai; // If it is allowed at compile time,`
  - > `ao.add(new Object());`
  - > `Integer i = ai.get(0); // This would result in`  
`// runtime ClassCastException`
- So there is **no inheritance relationship between type arguments** of a generic class



# Generics and Sub-typing

- The following code work
  - > `ArrayList<Integer> ai = new ArrayList<Integer>();`
  - > `List<Integer> li = new ArrayList<Integer>();`
  - > `Collection<Integer> ci = new ArrayList<Integer>();`
  - > `Collection<String> cs = new Vector<String>(4);`
- Inheritance relationship between Generic classes themselves still exist

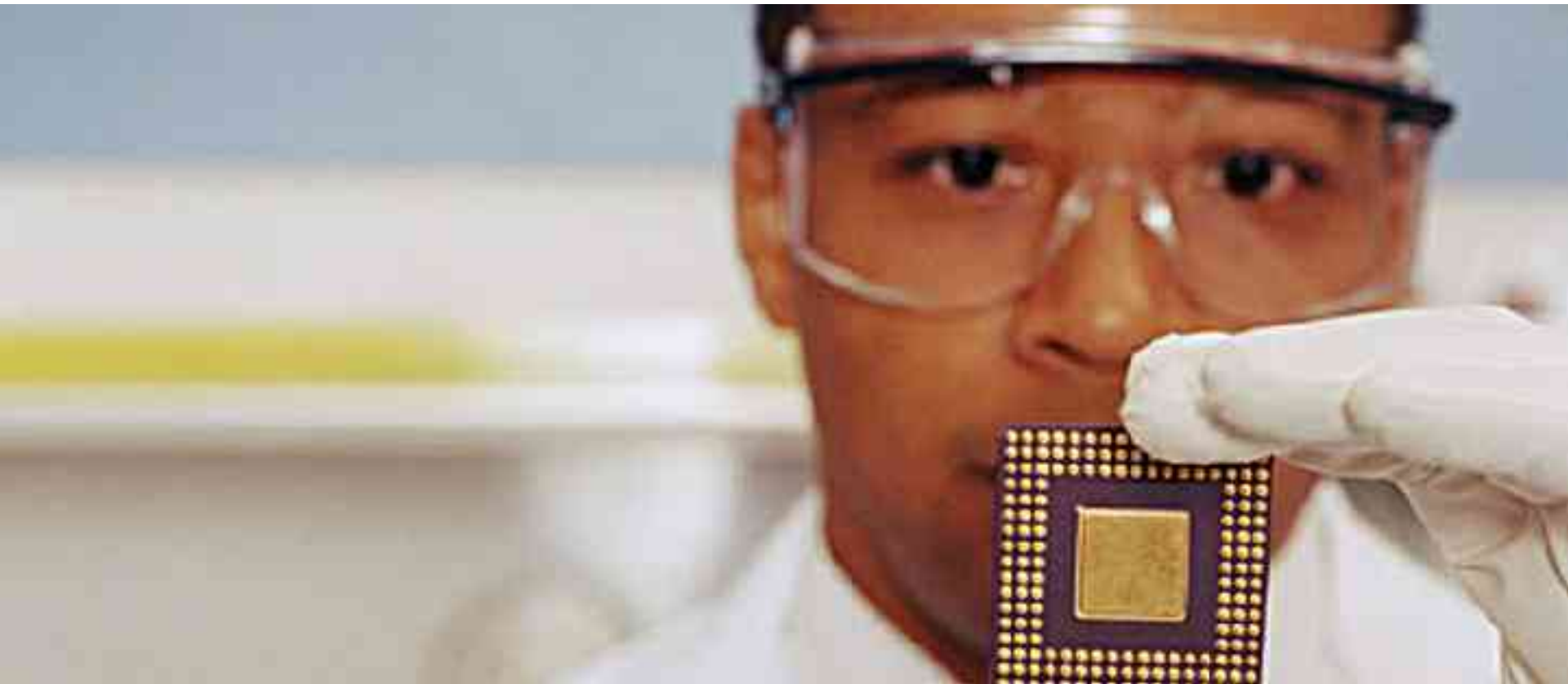


# Generics and Sub-typing

- The following code work
  - > `ArrayList<Number> an = new ArrayList<Number>();`
  - > `an.add(new Integer(5));`
  - > `an.add(new Long(1000L));`
  - > `an.add(new String("hello")); // compile error`
- Entries in a collection maintain inheritance relationship



# Generics: **Wild Card**





# Why Wildcards? Problem

- How do you write a method to print contents of any Collection – **Collection of any Type**? You now know you can't do the following:

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

```
public static void main(String[] args) {  
    List<Integer> li = new ArrayList<Integer>(10);  
    printCollection(li); // Compile error  
    Collection<String> cs = new Vector<String>();  
    printCollection(cs); // Compile error  
}
```



# Why Wildcards? Solution

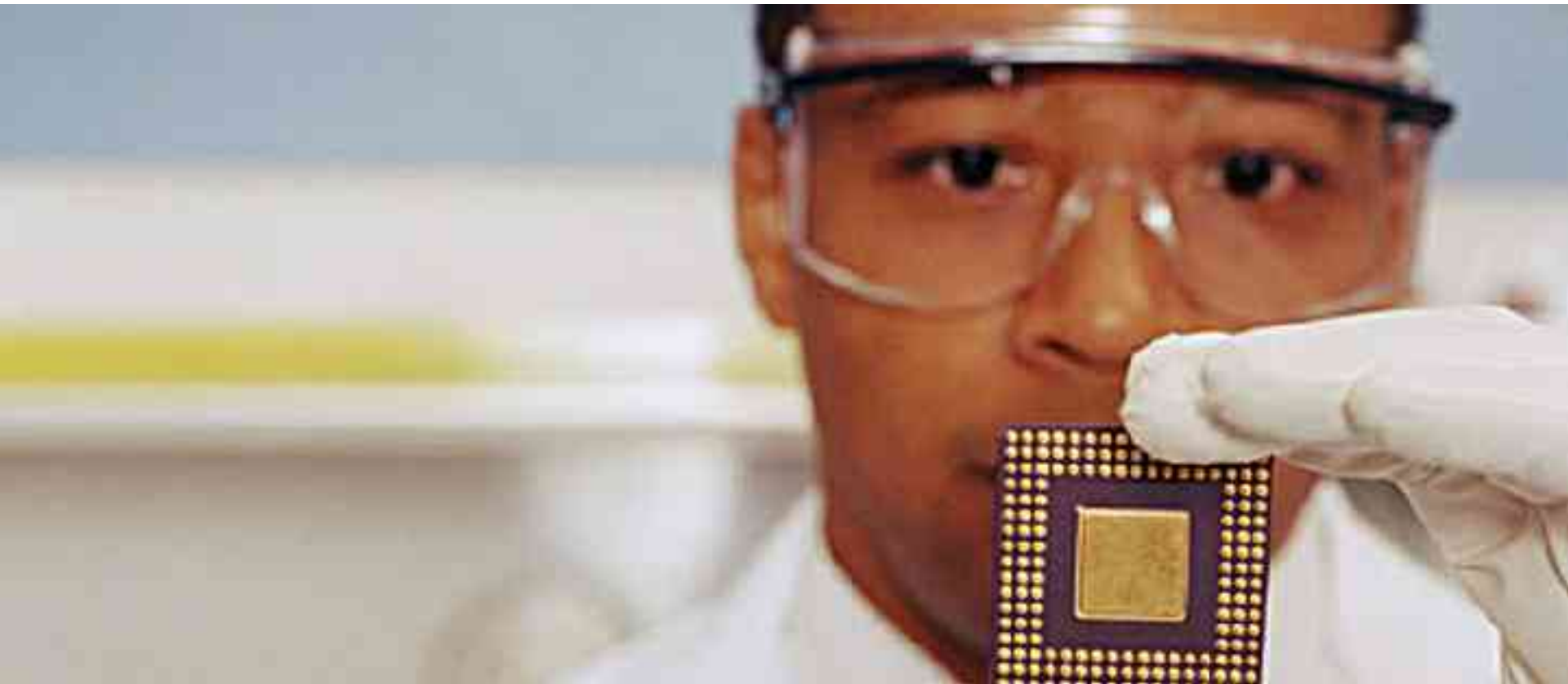
- Use Wildcard type argument `<?>`
- `Collection<?>` means **Collection of unknown type**

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

```
public static void main(String[] args) {  
    List<Integer> li = new ArrayList<Integer>(10);  
    printCollection(li); // No Compile error  
    Collection<String> cs = new Vector<String>();  
    printCollection(cs); // No Compile error  
}
```



# Generics: **Type Erasure**





# Raw Type

- Generic type instantiated with no type arguments
- Pre-J2SE 5.0 classes continue to function over J2SE 5.0 JVM as raw type

```
// Generic type instantiated with type argument  
List<String> ls = new LinkedList<String>();
```

```
// Generic type instantiated with no type  
// argument - This is Raw type  
List lraw = new LinkedList();
```



# Type Erasure

- All generic type information is removed in the resulting byte-code after compilation
- So generic type information does not exist during runtime
- After compilation, they all share same class
  - > The class that represents `ArrayList<String>`, `ArrayList<Integer>` is the same class that represents `ArrayList`



# Type Erasure Example Code: True or False?

```
ArrayList<Integer> ai = new ArrayList<Integer>();
```

```
ArrayList<String> as = new ArrayList<String>();
```

```
Boolean b1 = (ai.getClass() == as.getClass());
```

```
System.out.println("Do ArrayList<Integer> and ArrayList<String> share  
same class? " + b1);
```

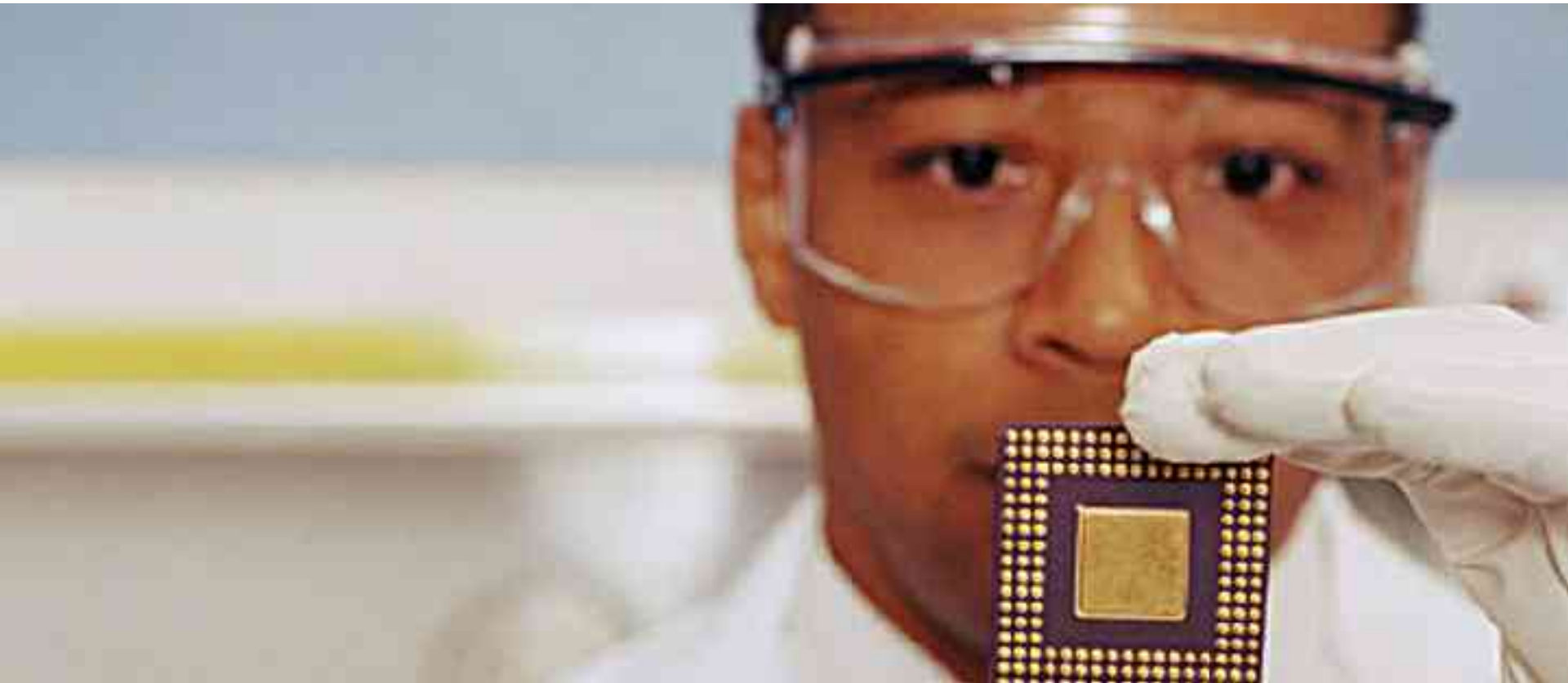


# Type-safe Code Again

- The compiler guarantees that either:
  - > the code it generates will be type-correct at run time, or
  - > it will output a warning (using Raw type) at compile time
- If your code compiles without warnings and has no casts, then you will never get a `ClassCastException`
  - > This is “type safe” code



# Autoboxing & Unboxing





# Autoboxing/Unboxing of Primitive Types

- Problem: (pre-J2SE 5.0)
  - > Conversion between primitive types and wrapper types (and vice-versa)
  - > You need manually convert a primitive type to a wrapper type before adding it to a collection

```
int i = 22;
```

```
List l = new LinkedList();
```

```
l.add(new Integer(i));
```



# Autoboxing/Unboxing of Primitive Types

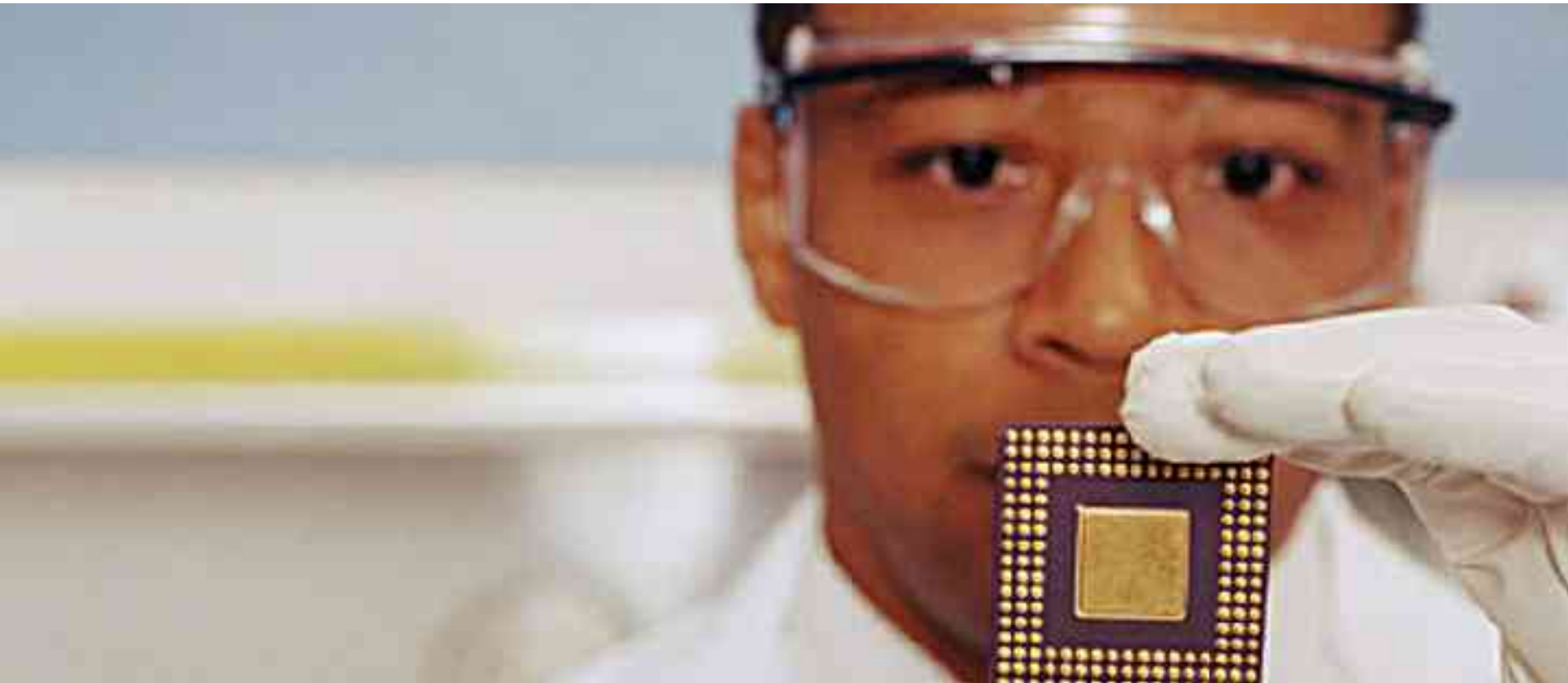
- Solution: Let the compiler do it

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

```
ArrayList<Integer> al = new ArrayList<Integer>();  
al.add(22); // Autoboxing conversion
```



# Enhanced for Loop





# Enhanced for Loop (foreach)

- Problem: (pre-J2SE 5.0)
  - > 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)
- Solution: Let the compiler do it
  - > New for loop syntax  
**for (variable : collection)**
  - > Works for Collections and arrays



# Enhanced for Loop Example

- Old code

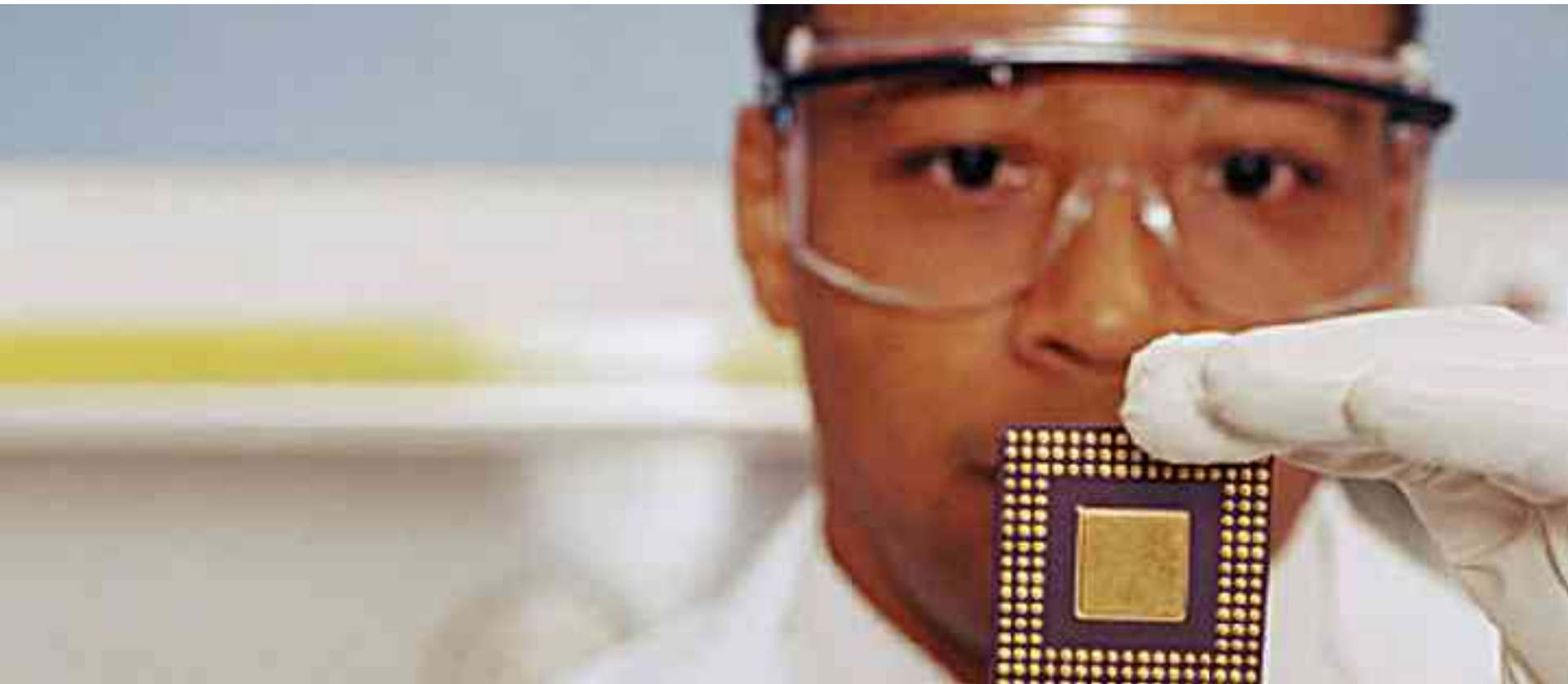
```
void cancelAll(Collection c) {  
    for (Iterator i = c.iterator(); i.hasNext(); ){  
        TimerTask task = (TimerTask)i.next();  
        task.cancel();  
    }  
}
```

- New Code

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



# Type-safe Enumerations





# Type-safe Enumerations

- Problem: (pre-J2SE 5.0) Previously, if you wanted to define an enumeration you either:
  - > Defined a bunch of integer constants
  - > Followed one of the various “type-safe enum patterns”
- Issues of using Integer constants
  - > `public static final int SEASON_WINTER = 0;`
  - > `Public static final int SEASON_SUMMER = 1;`
  - > Not type safe (any integer will pass)
  - > No namespace (SEASON\_\*)
  - > Brittleness (how do add value in-between?)
  - > Printed values uninformative (prints just int values)

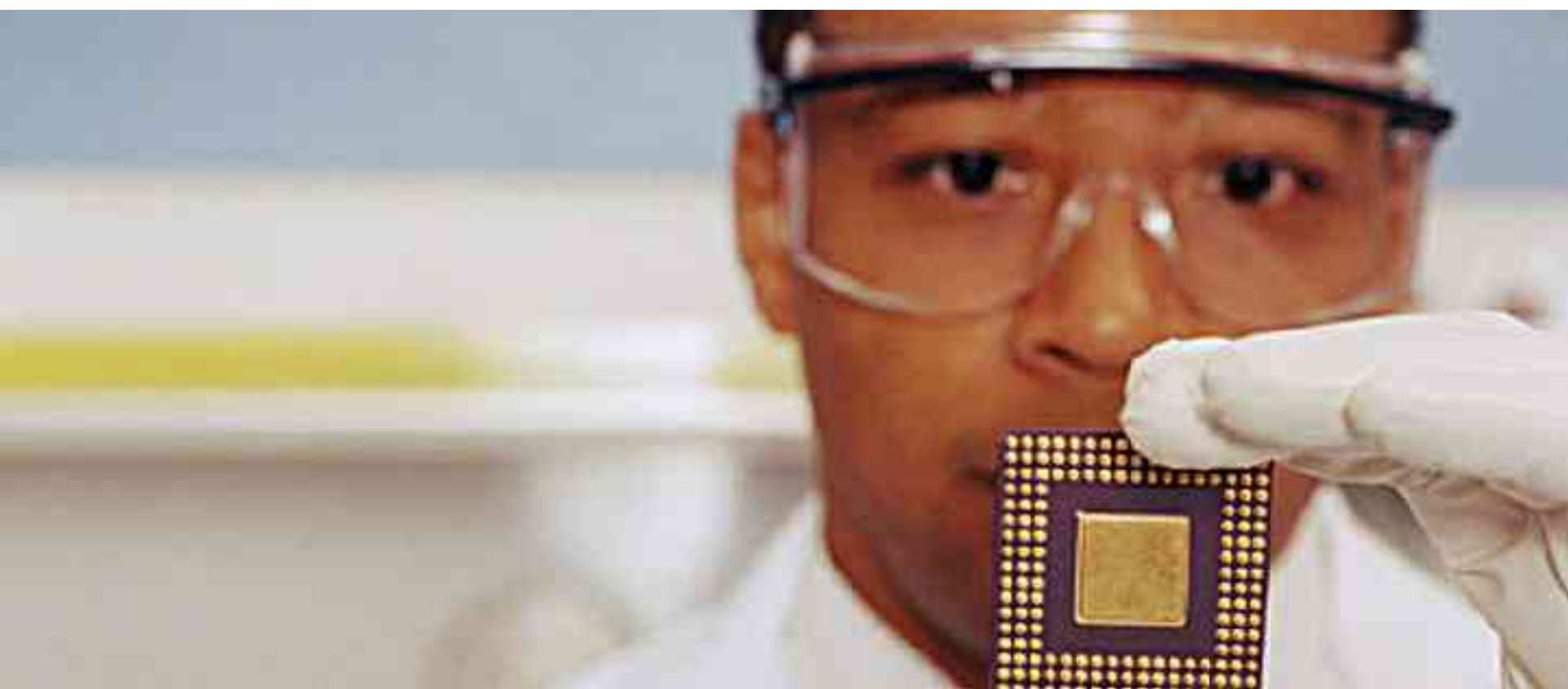


# Type-safe Enumerations

- Issues of using “type-safe enum patterns”
  - > Verbose
  - > Do not work well with switch statements
- Solution: New type of class declaration
  - > `enum` type has public, self-typed members for each enum constant
  - > New keyword, `enum`



# Varargs





# Varargs

- Problem: (in pre-J2SE 5.0)
  - > To have a method that takes a variable number of parameters
  - > Can be done with an array, but caller has to create it first
  - > Look at `java.text.MessageFormat`
- Solution: Let the compiler do it for you
  - > `public static String format`  
(String fmt, **Object...** args);
  - > Java now supports `printf(...)`

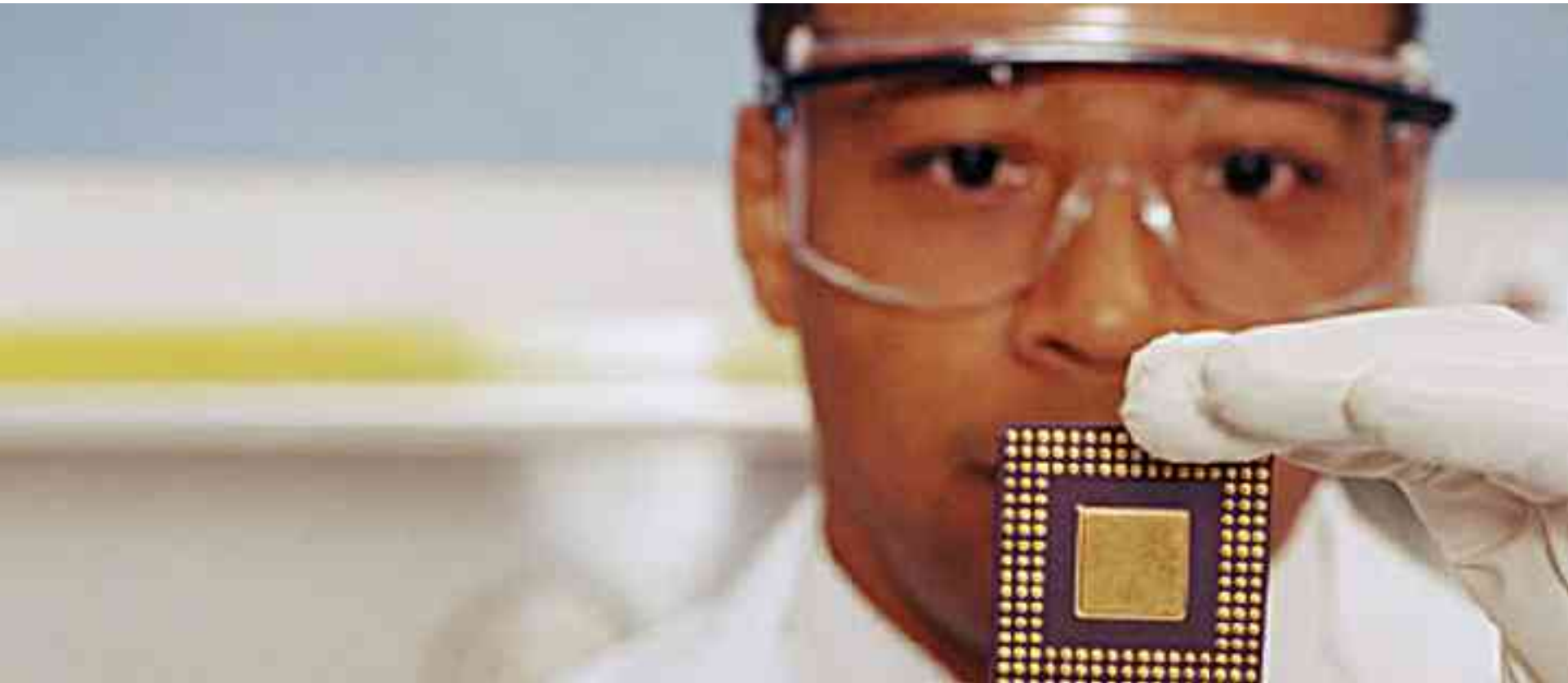


# Varargs examples

- APIs have been modified so that methods accept variable-length argument lists where appropriate
  - > `Class.getMethod`
  - > `Method.invoke`
  - > `Constructor.newInstance`
  - > `Proxy.getProxyClass`
  - > `MessageFormat.format`
- New APIs do this too
  - > `System.out.printf("%d + %d = %d\n", a, b, a+b);`



# Static Imports



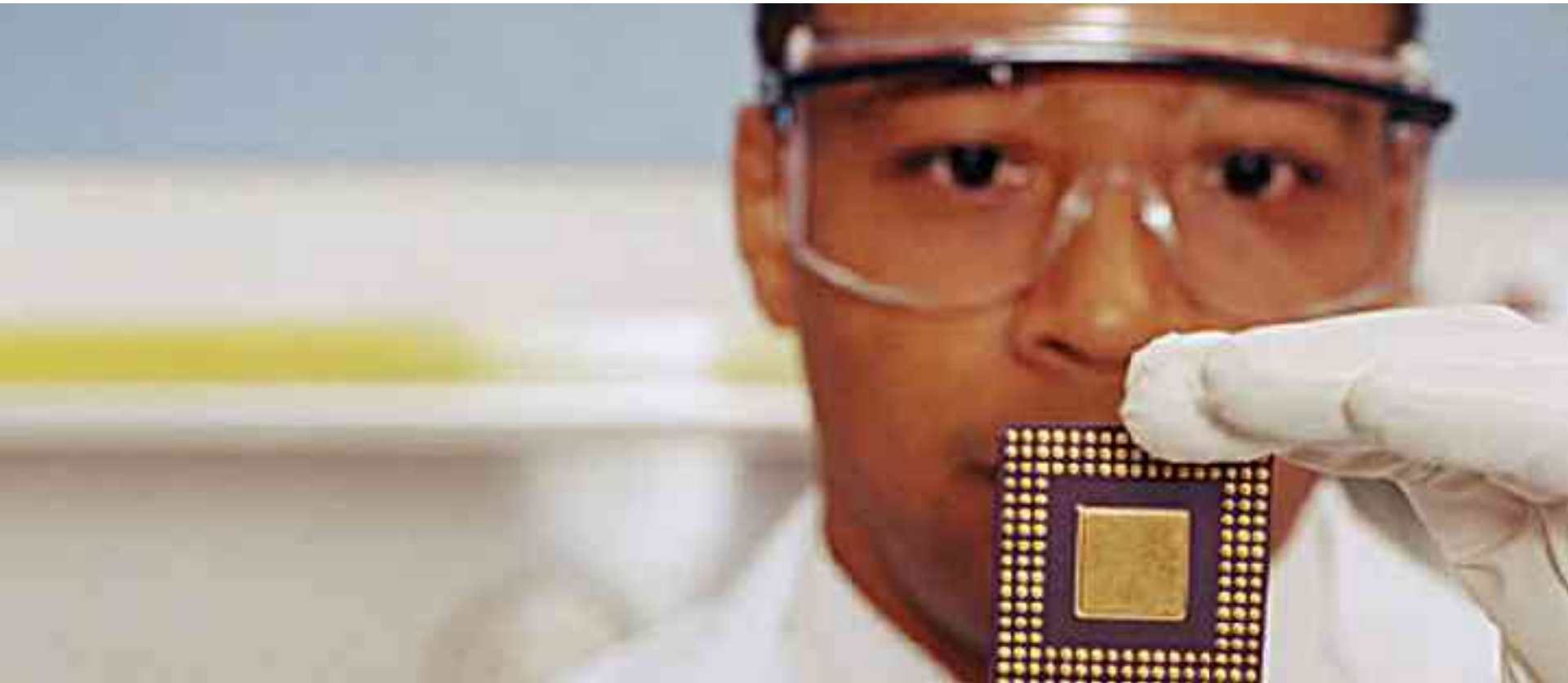


# Static Imports

- Problem: (pre-J2SE 5.0)
  - > 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)`



# Annotation(Metadata)





# Sub-topics of Annotations

- What is and Why annotation?
- How to define and use Annotations?
- 3 different kinds of Annotations
- Meta-Annotations



# How Annotations Are Used?

- Annotations are used to affect the way programs are treated by tools and libraries
- Annotations are used by tools to produce derived files
  - > Tools: Compiler, IDE, Runtime tools
  - > Derived files : New Java code, deployment descriptor, class files



# Ad-hoc Annotation-like Examples in pre-J2SE 5.0 Platform

- Ad-hoc Annotation-like examples in pre-J2SE 5.0 platform
  - > **Transient**
  - > **Serializable** interface
  - > **@deprecated**
  - > javadoc comments
  - > Xdoclet
- J2SE 5.0 Annotation provides a standard, general purpose, more powerful annotation scheme

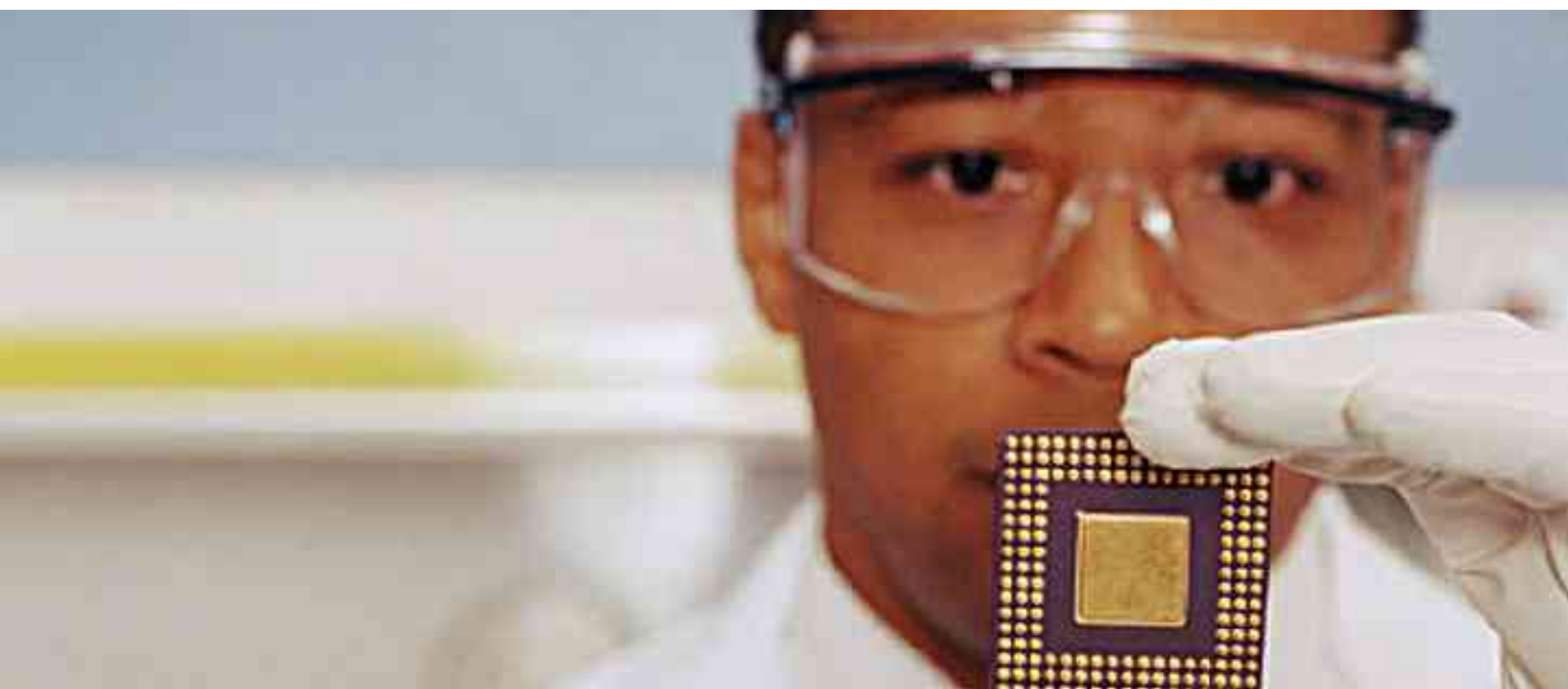


# Why Annotation?

- Enables “declarative programming” style
  - > Less coding since tool will generate the boiler plate code from annotations in the source code
  - > Easier to change
- Eliminates the need for maintaining "side files" that must be kept up to date with changes in source files
  - > Information is kept in the source file
  - > example) Eliminate the need of deployment descriptor



# How to define and use Annotations?





# How to Define Annotation Type?

- Annotation type definitions are similar to normal interface definitions
  - > An at-sign (@) precedes the **interface** keyword
  - > **Each method declaration defines an element of the annotation type**
  - > Method declarations must not have any parameters or a throws clause
  - > Return types are restricted to primitives, String, Class, enums, annotations, and arrays of the preceding types
  - > Methods can have default values



# Example: Annotation Type Definition

```
/**  
 * Describes the Request-For-Enhancement(RFE) that led  
 * to the presence of the annotated API element.  
 */  
public @interface RequestForEnhancement {  
    int id();  
    String synopsis();  
    String engineer() default "[unassigned]";  
    String date();    default "[unimplemented]";  
}
```



# How To Use Annotation

- Once an annotation type is defined, you can use it to annotate declarations
  - > class, method, field declarations
- An annotation is a special kind of modifier, and can be used anywhere that other modifiers (such as public, static, or final) can be used
  - > By convention, annotations precede other modifiers
  - > Annotations consist of an at-sign (@) followed by an annotation type and a parenthesized list of element-value pairs



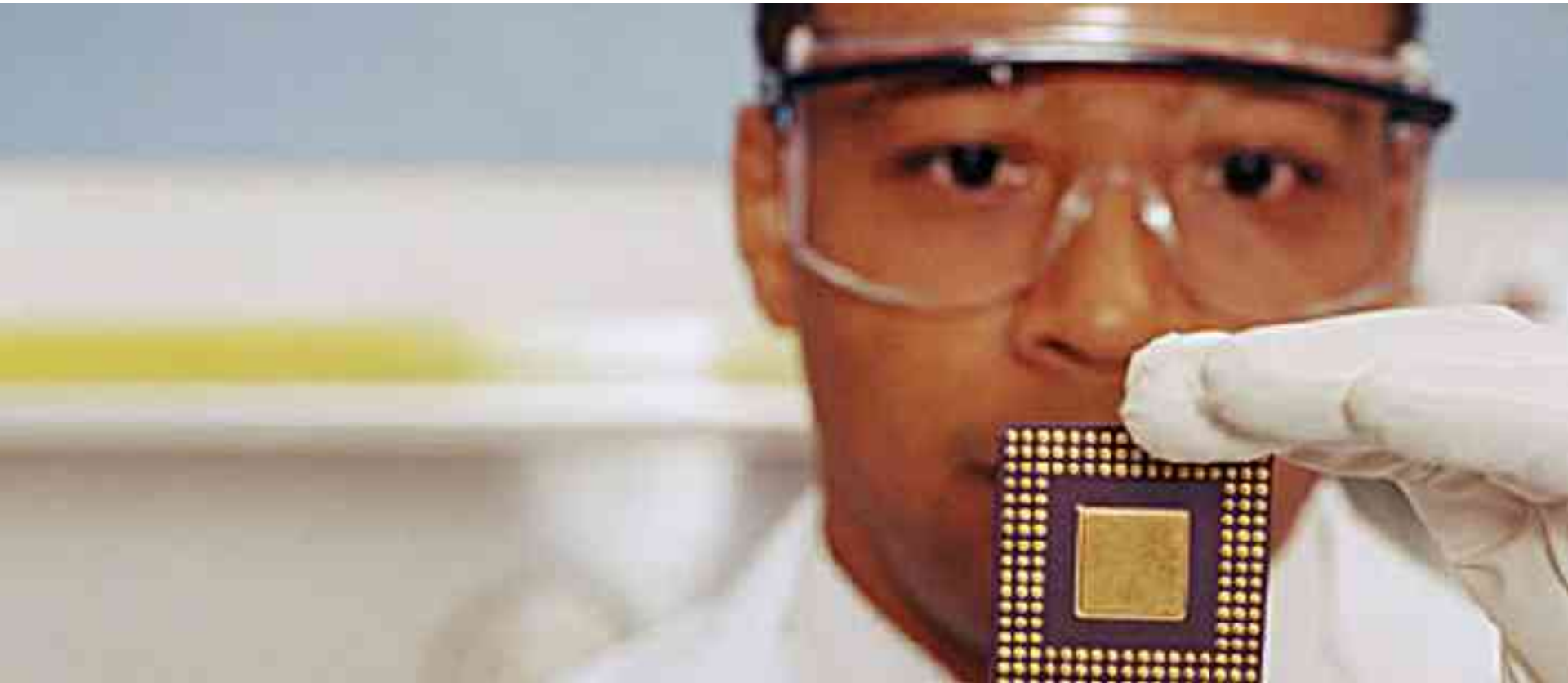
# Example: Usage of Annotation

```
@RequestForEnhancement(  
    id      = 2868724,  
    synopsis = "Enable time-travel",  
    engineer = "Mr. Peabody",  
    date    = "4/1/3007"  
)  
public static void travelThroughTime(Date destination)  
    { ... }
```

It is annotating travelThroughTime method



# 3 Different Types of Annotations





# 3 Different Kinds of Annotations

- Marker annotation
- Single value annotation
- Normal annotation



# Marker Annotation

- An annotation type with no elements
- Definition

```
/**
```

```
 * Indicates that the specification of the annotated API element  
 * is preliminary and subject to change.
```

```
*/
```

```
public @interface Preliminary { }
```

- Usage – No need to have ()

```
@Preliminary public class TimeTravel { ... }
```



# Single Value Annotation

- An annotation type with a single element
  - > The element should be named “**value**”

- Definition

```
/**  
 * Associates a copyright notice with the annotated API element.  
 */  
public @interface Copyright {  
    String value();  
}
```

- Usage – can omit the element name and equals sign (=)

```
@Copyright("2002 Yoyodyne Propulsion Systems")  
public class OscillationOverthruster { ... }
```



# Normal Annotation

- We already have seen an example
- Definition

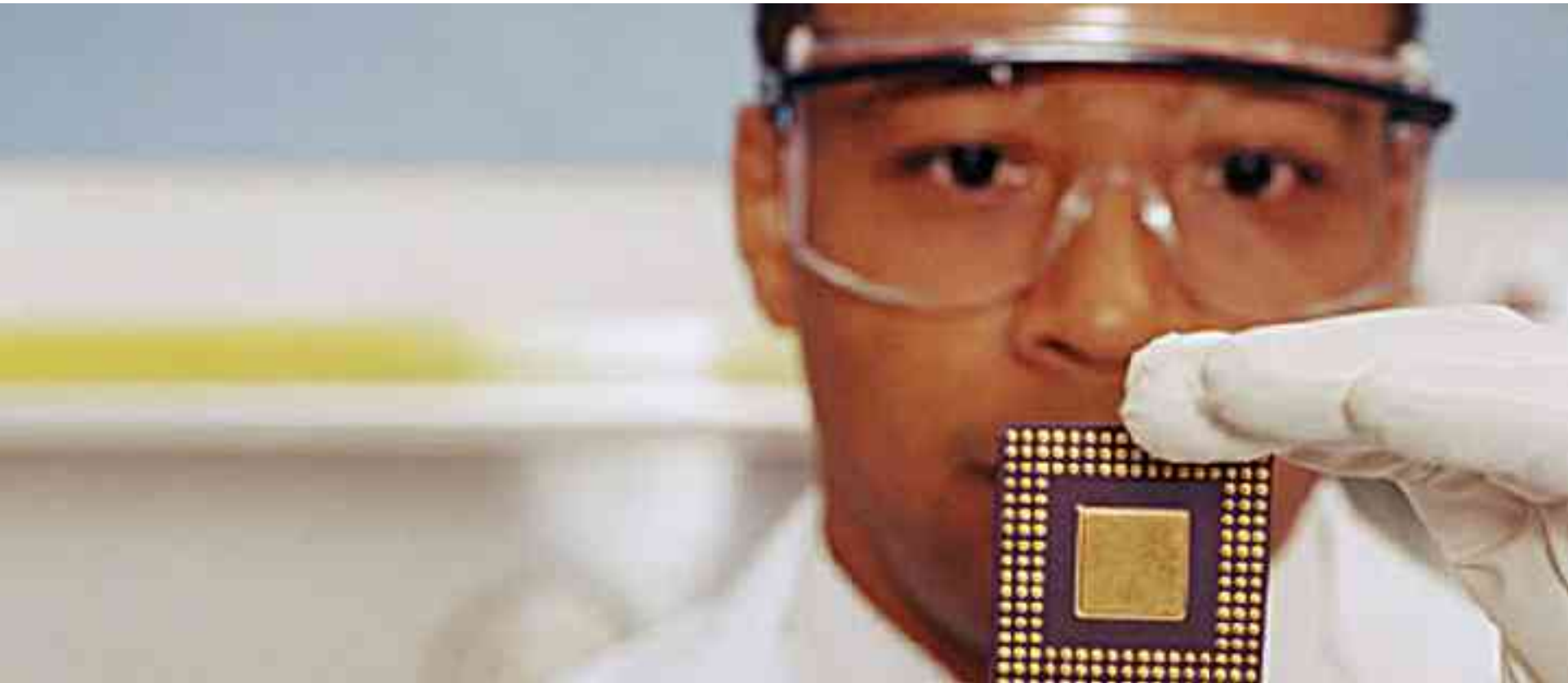
```
public @interface RequestForEnhancement {  
    int id();  
    String synopsis();  
    String engineer() default "[unassigned]";  
    String date(); default "[unimplemented]";  
}
```

- Usage

```
@RequestForEnhancement(  
    id = 2868724,  
    synopsis = "Enable time-travel",  
    engineer = "Mr. Peabody",  
    date = "4/1/3007"  
)  
public static void travelThroughTime(Date destination) { ... }
```



# Meta-Annotations





# Meta-Annotations (Used to Annotate Annotations)

- **@Retention**
  - > How long annotation information is kept
  - > **Enum RetentionPolicy**
    - > **SOURCE, CLASS (Default), RUNTIME**
- **@Target**
  - > Restrictions on use of this annotation
  - > **Enum ElementType**
    - > **TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL\_VARIABLE, ANNOTATION\_TYPE, PACKAGE**



# Example: Definition and Usage of an Annotation with Meta Annotation

## Definition of Accessor annotation

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

## Usage Example of the Accessor annotation

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



# Reflection and Metadata

- Marker annotation

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

- Single value annotation

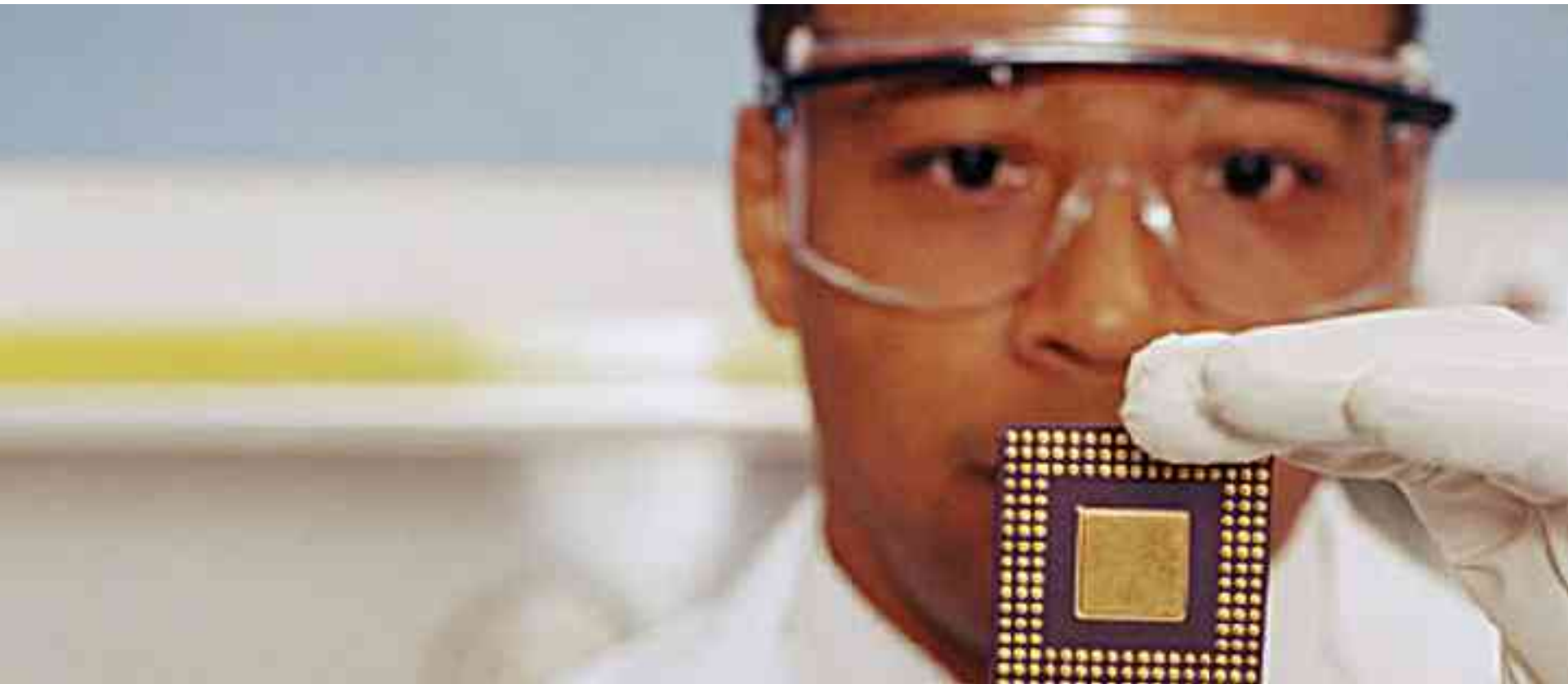
```
String copyright = MyClass.class.getAnnotation  
    (Copyright.class).value();
```

- Normal annotation

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



# Concurrent Utilities





# Concurrency Utilities: JSR-166

- Enables development of simple yet powerful multi-threaded applications
  - > Like Collection provides rich data structure handling capability
- Beat C performance in high-end server applications
- Provide richer set of concurrency building blocks
  - > `wait()`, `notify()` and `synchronized` are too primitive
- Enhance scalability, performance, readability and thread safety of Java applications



# Why Use Concurrency Utilities?

- Reduced programming effort
- Increased performance
- Increased reliability
  - > Eliminate threading hazards such as deadlock, starvation, race conditions, or excessive context switching are eliminated
- Improved maintainability
- Increased productivity

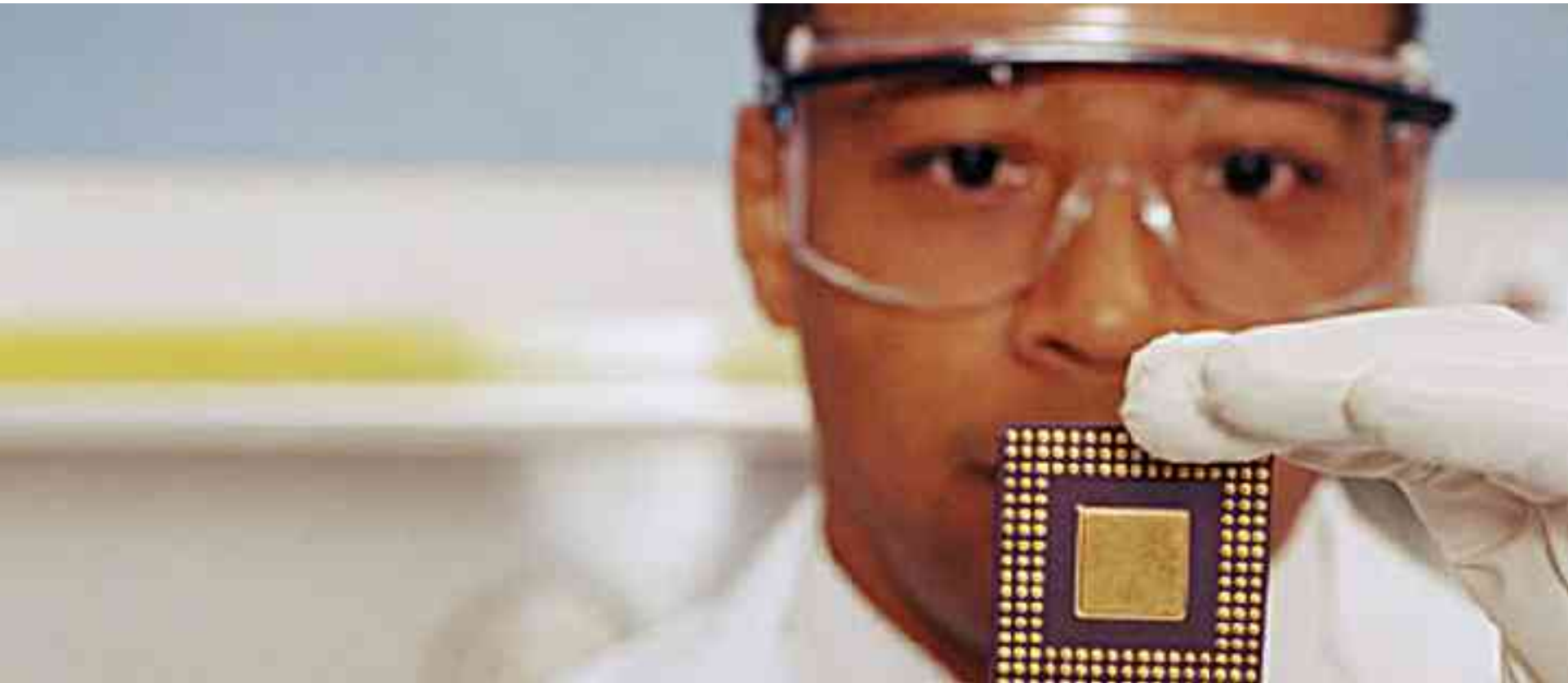


# Concurrency Utilities

- Task Scheduling Framework
- Callables and Futures
- Synchronizers
- Concurrent Collections
- Atomic Variables
- Locks
- Nanosecond-granularity timing



# Concurrent Utilities: Task Scheduling Framework





# Task Scheduling Framework

- **Executor/ExecutorsService/Executors** framework supports
  - > standardizing invocation
  - > scheduling
  - > execution
  - > control of asynchronous tasks according to a set of execution policies
- **Executor** is an interface
- **ExecutorsService** extends **Executor**
- **Executors** is factory class for creating various kinds of **ExecutorsService** implementations



# Executor Interface

- Executor interface provides a way of decoupling task **submission** from the **execution**
  - > execution: mechanics of how each task will be run, including details of thread use, scheduling
- Example

```
Executor executor = anExecutor;  
executor.execute(new RunnableTask1());  
executor.execute(new RunnableTask2());
```
- Many Executor implementations impose some sort of limitation on how and when tasks are scheduled



# Executor and ExecutorService

## ExecutorService adds lifecycle management

```
public interface Executor {
    void execute(Runnable command);
}

public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout,
                             TimeUnit unit);

    // other convenience methods for submitting tasks
}
```



# Creating ExecutorService From Executors

```
public class Executors {  
    static ExecutorService  
        newSingleThreadedExecutor();  
  
    static ExecutorService  
        newFixedThreadPool(int n);  
  
    static ExecutorService  
        newCachedThreadPool(int n);  
  
    static ScheduledExecutorService  
        newScheduledThreadPool(int n);  
  
    // additional versions specifying ThreadFactory  
    // additional utility methods  
}
```



# pre-J2SE 5.0 Code

## Web Server—poor resource management

```
class WebServer {  
  
    public static void main(String[] args) {  
        ServerSocket socket = new ServerSocket(80);  
  
        while (true) {  
            final Socket connection = socket.accept();  
            Runnable r = new Runnable() {  
                public void run() {  
                    handleRequest(connection);  
                }  
            };  
            // Don't do this!  
            new Thread(r).start();  
        }  
    }  
}
```



# Executors Example

## Web Server—better resource management

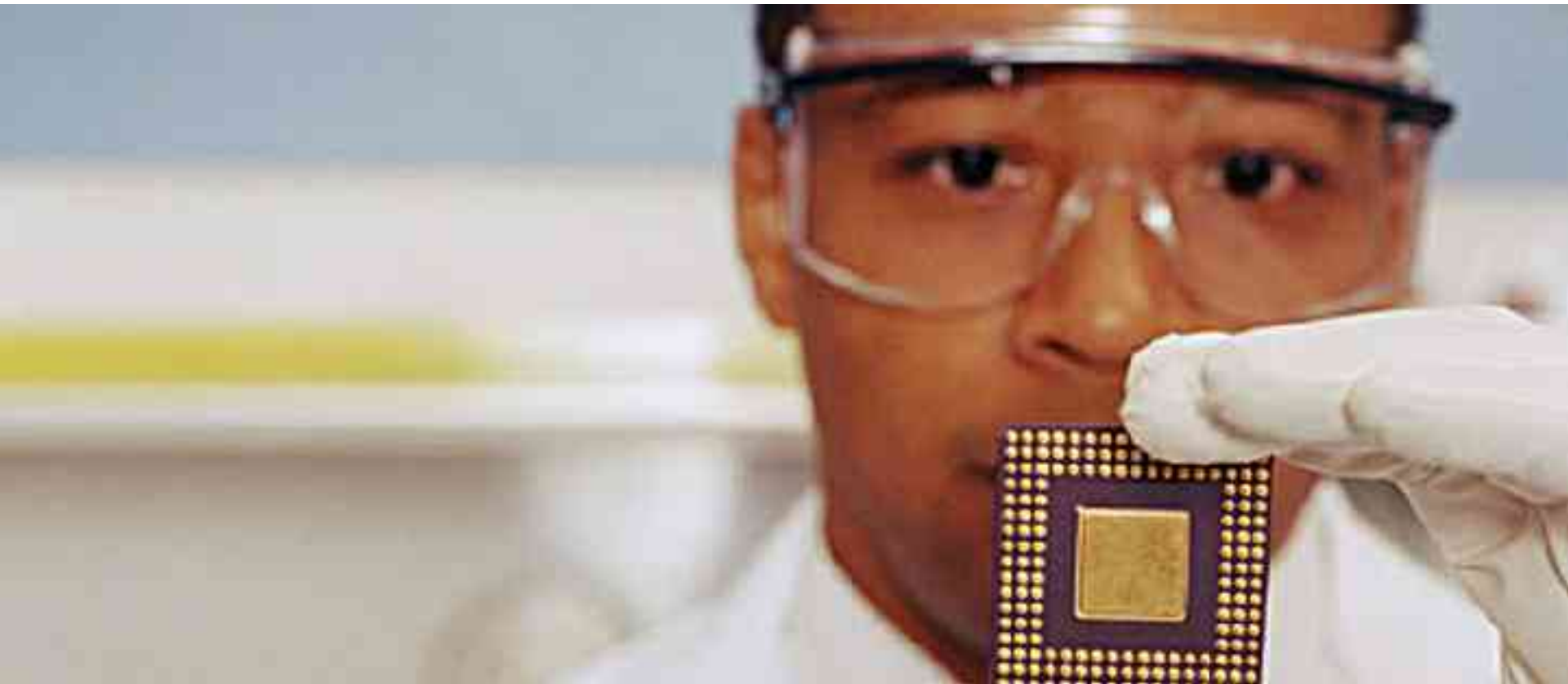
```
class WebServer {
    Executor pool =
        Executors.newFixedThreadPool(7);

    public static void main(String[] args) {
        ServerSocket socket = new ServerSocket(80);

        while (true) {
            final Socket connection = socket.accept();
            Runnable r = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            pool.execute(r);
        }
    }
}
```



# Concurrent Utilities: **Callables and Futures**





# Callable and Futures: Problem (pre-J2SE 5.0)

- If a new thread (callable thread) is started in an application, there is currently no way to return a result from that thread to the thread (calling thread) that started it without the use of a shared variable and appropriate synchronization
- This is complex and makes code harder to understand and maintain



# Callables and Futures

- Callable thread (Callee) implements **Callable** interface
  - > implement `call()` method rather than `run()`
- Calling thread (Caller) submits **Callable** object to Executor and then moves on
  - > call `submit()` not `execute()`
  - > returns a **Future** object
- Calling thread (Caller) then retrieves the result using `get()` method of Future object
  - > If result is ready, it is returned
  - > If result is not ready, calling thread will block



# Build CallableExample (This is Callee)

```
class CallableExample
    implements Callable<String> {

    public String call() {
        String result = null;

        /* Do some work and create a result */

        return result;
    }
}
```

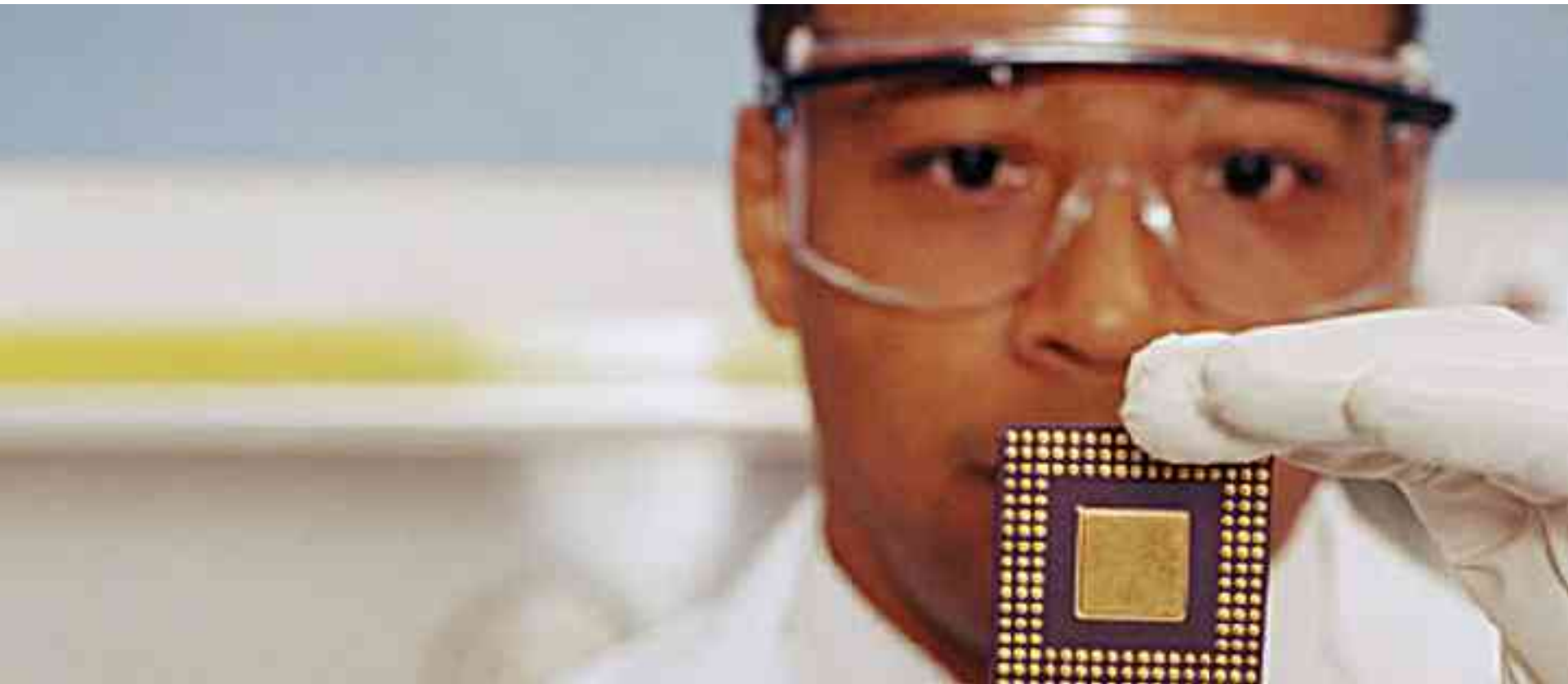


# Future Example (Caller)

```
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 */  
}
```



# Concurrent Utilities: Synchronizers





# 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

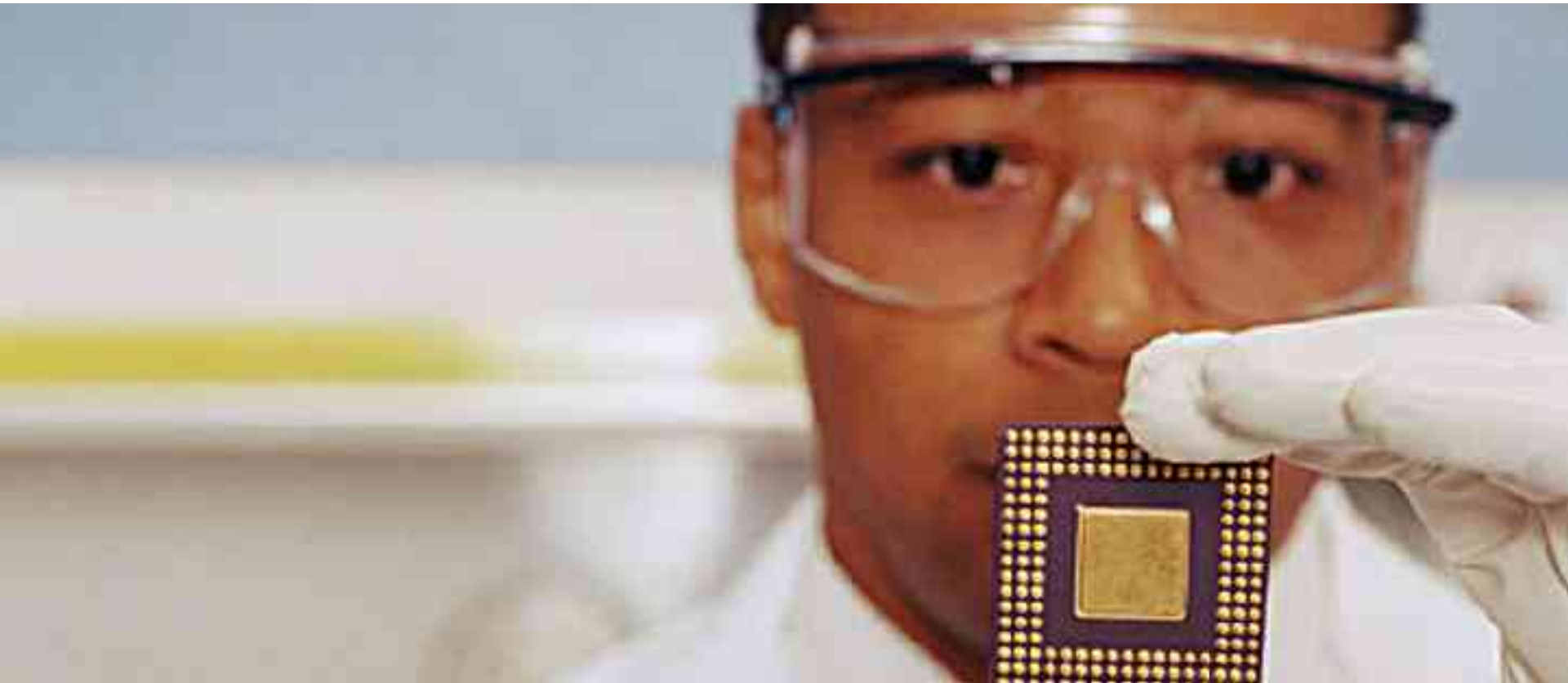


# Semaphore Example

```
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) {}  
    /* Acquire resource */  
}  
public void returnResource(Resource r) {  
    /* Return resource to pool */  
    available.release();  
}
```



# Concurrent Utilities: Concurrent Collections





# BlockingQueue Interface

- 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

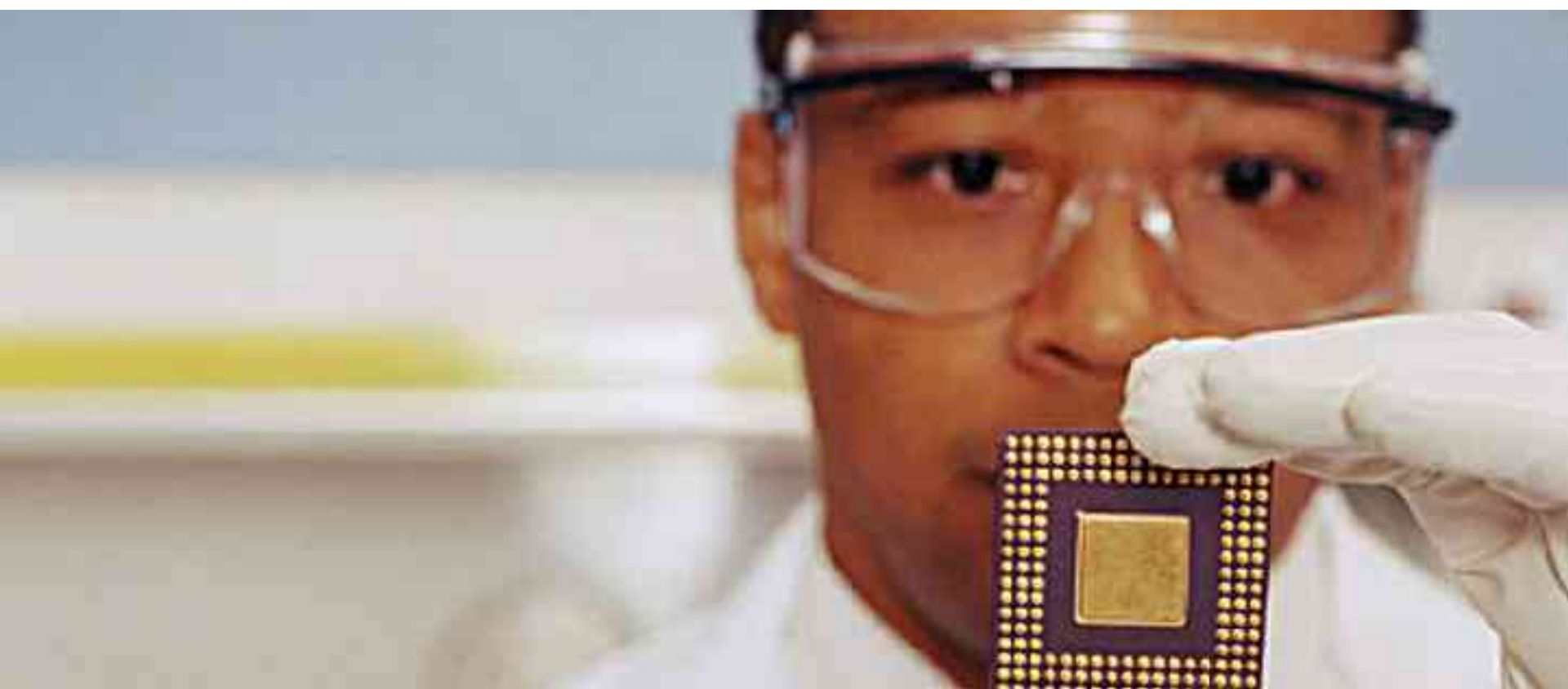
```
private ArrayBlockingQueue messageQueue =  
    new ArrayBlockingQueue<String>(10);
```

```
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) { }
```



# Concurrent Utilities: **Atomic Variables**





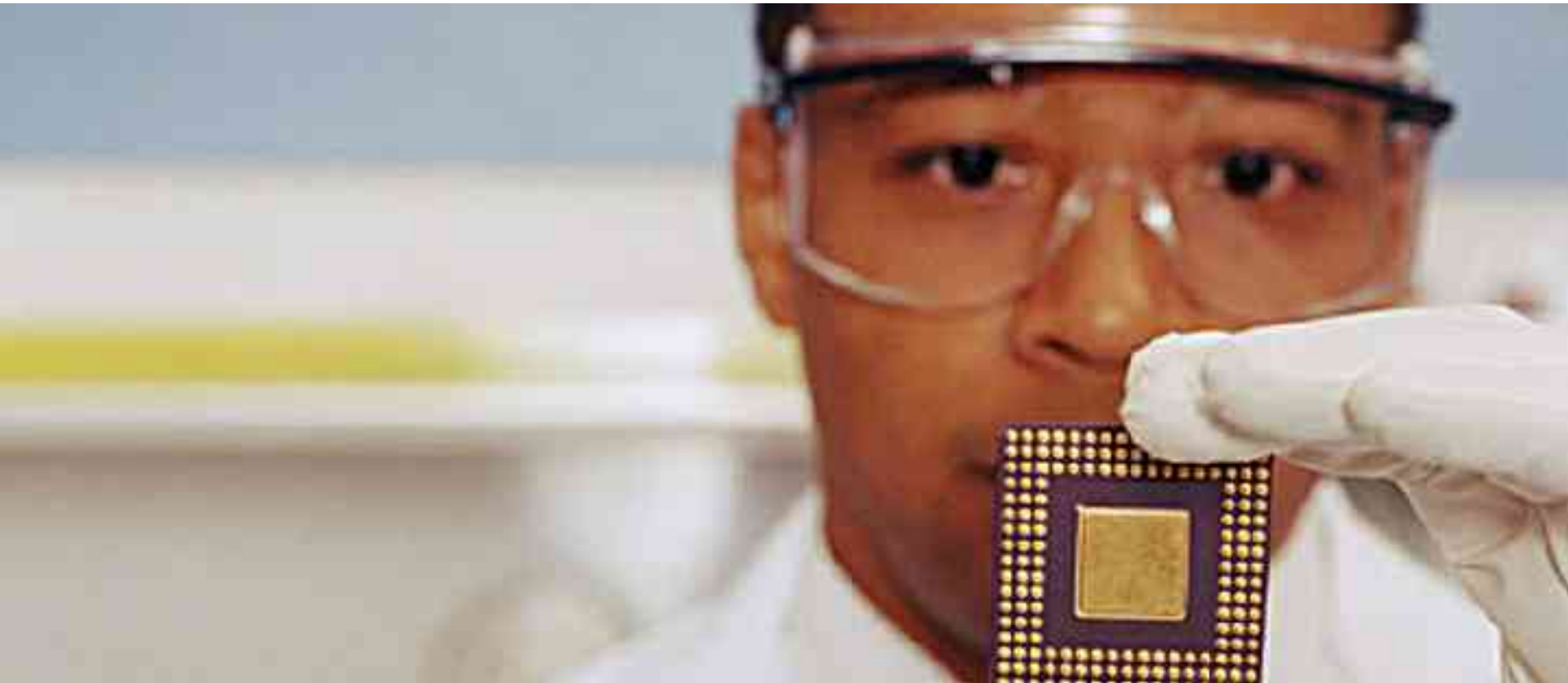
# Atomics

- **java.util.concurrent.atomic**
  - > Small toolkit of classes that support lock-free thread-safe programming on single variables

```
AtomicInteger balance = new AtomicInteger(0);  
  
public int deposit(integer amount) {  
    return balance.addAndGet(amount);  
}
```



# Concurrent Utilities: **Locks**





# Locks

- Lock interface
  - > More extensive locking operations than synchronized block
  - > No automatic unlocking – use try/finally to unlock
  - > Non-blocking access using **tryLock( )**
- ReentrantLock
  - > Concrete implementation of Lock
  - > Holding thread can call **lock( )** multiple times and not block
  - > Useful for recursive code



# ReadWriteLock

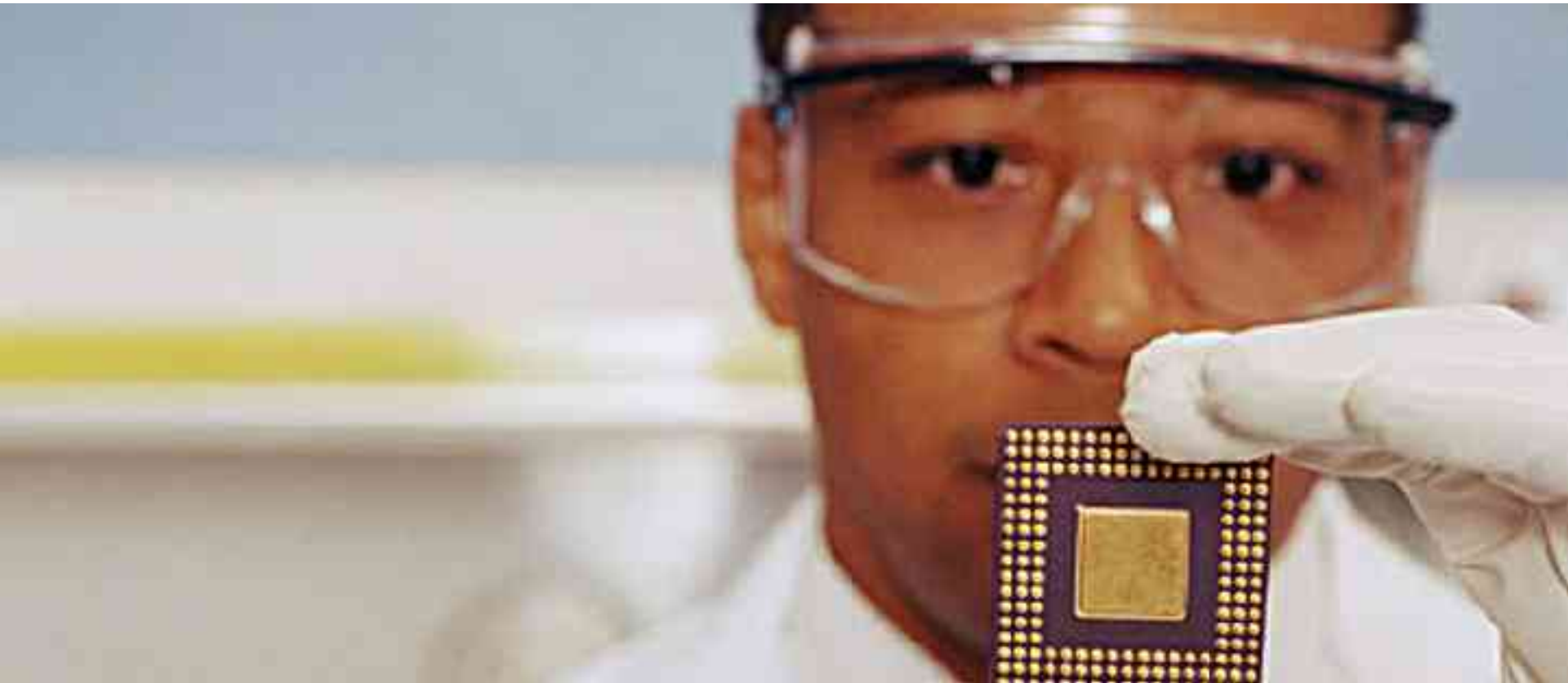
- Has two locks controlling read and write access
  - > Multiple threads can acquire the read lock if no threads have a write lock
  - > If a thread has a read lock, others can acquire read lock but nobody can acquire write lock
  - > If a thread has a write lock, nobody can have read/write lock
  - > Methods to access locks

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

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



# Formatted I/O





# Simple Formatted I/O & Scanner

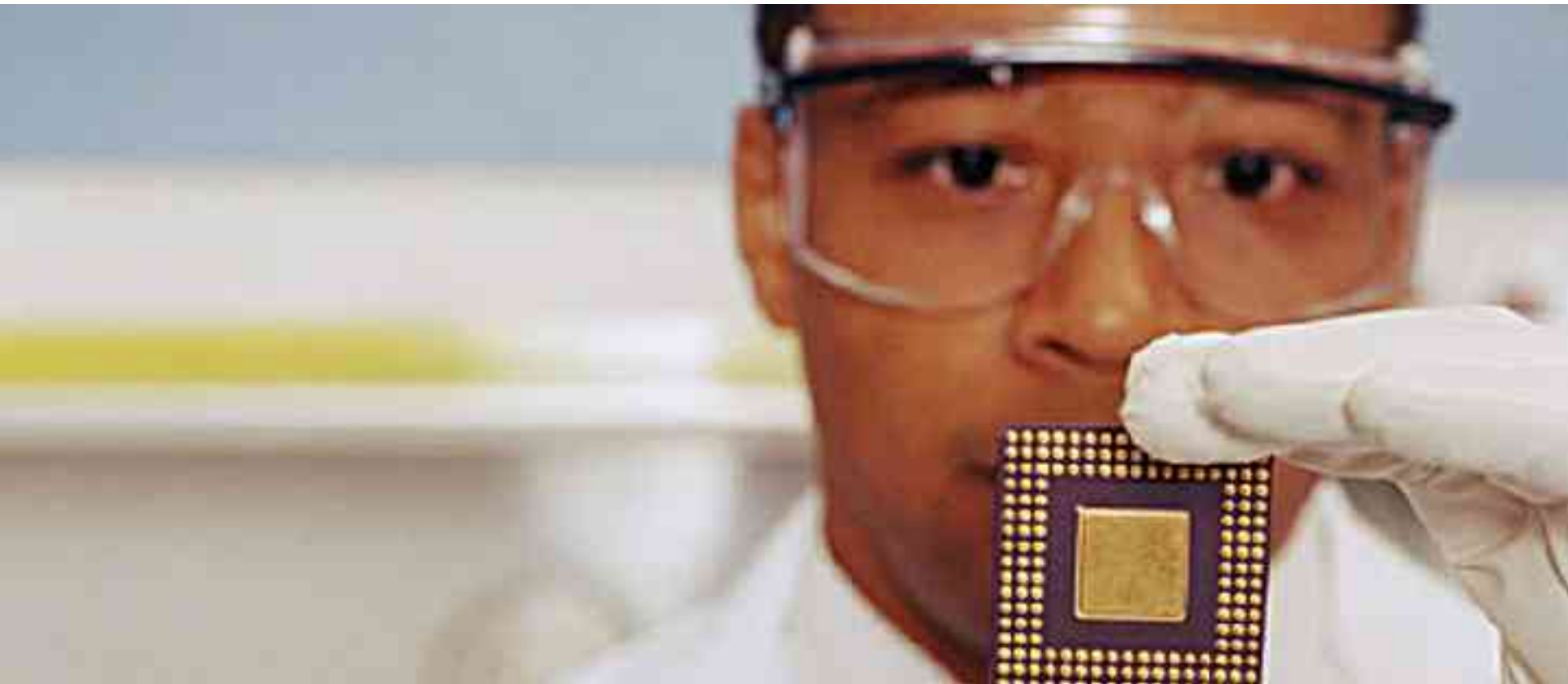
- **Printf** is popular with C/C++ developers
  - > Powerful, easy to use
- Finally adding printf to J2SE 5.0 (using varargs)

```
out.printf("%-12s is %2d long", name, l);  
out.printf("value = %2.2F", value);
```
- Also a simple scanning API: convert text into primitives or Strings

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



# Virtual Machine





# Class Data Sharing

- Improved startup time
  - > especially for small applications
  - > up to 30% faster
- Reduced memory footprint
- During JRE installation, a set of classes are saved into a file, called a "shared archive"
- During subsequent JVM invocations, the shared archive is memory-mapped in
- `-Xshare:on`, `-Xshare:off`, `-Xshare:auto`, `-Xshare:dump`



# Server Class Machine

- Auto-detected
  - > Application will use Java HotSpot Server VM
  - > Server VM starts slower but runs faster than Client VM
- 2 CPU, 2GB memory (except windows)
  - > Uses server compiler
  - > Uses parallel garbage 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



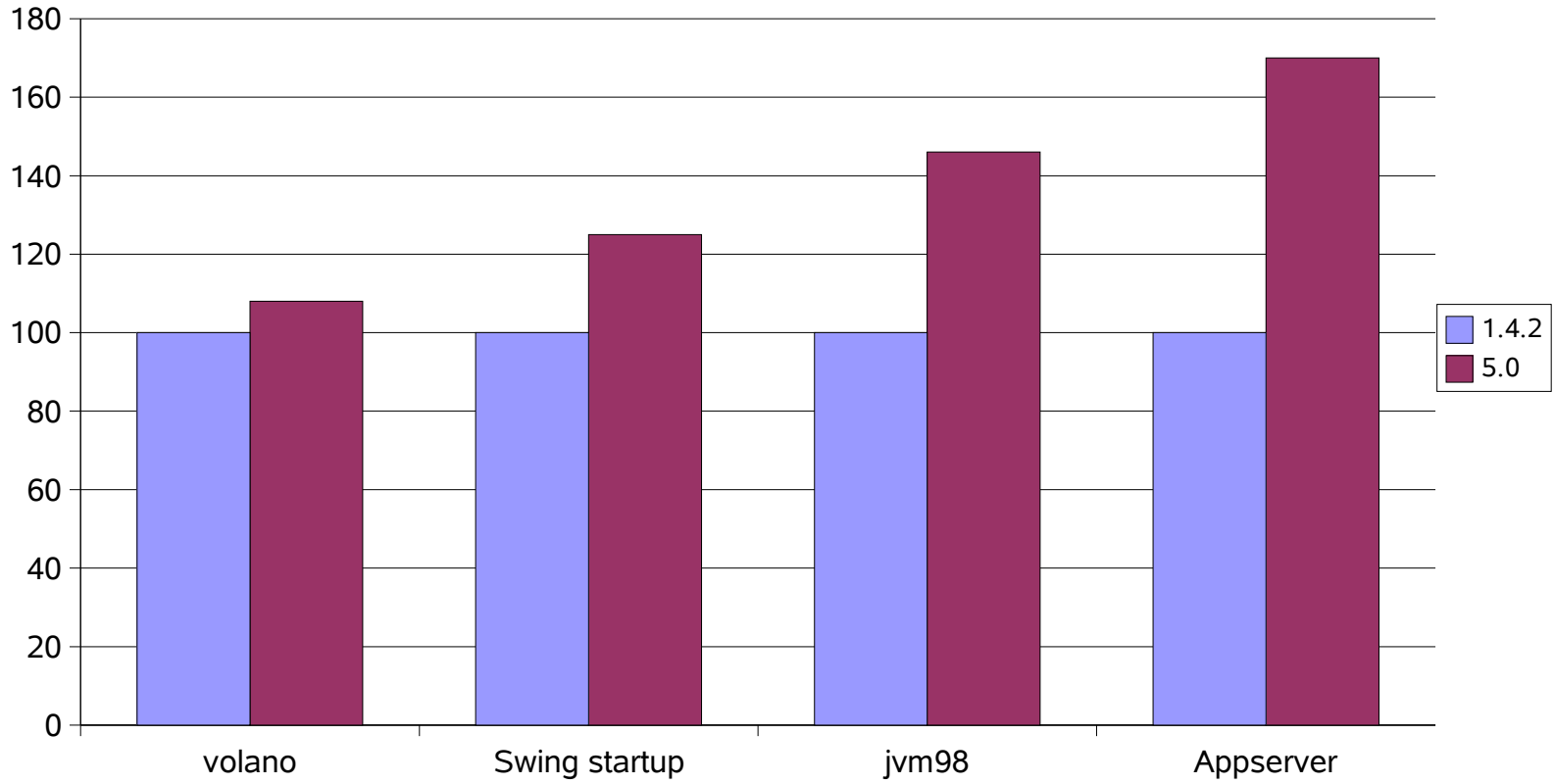
# JVM Self Tuning (Ergonomics)

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



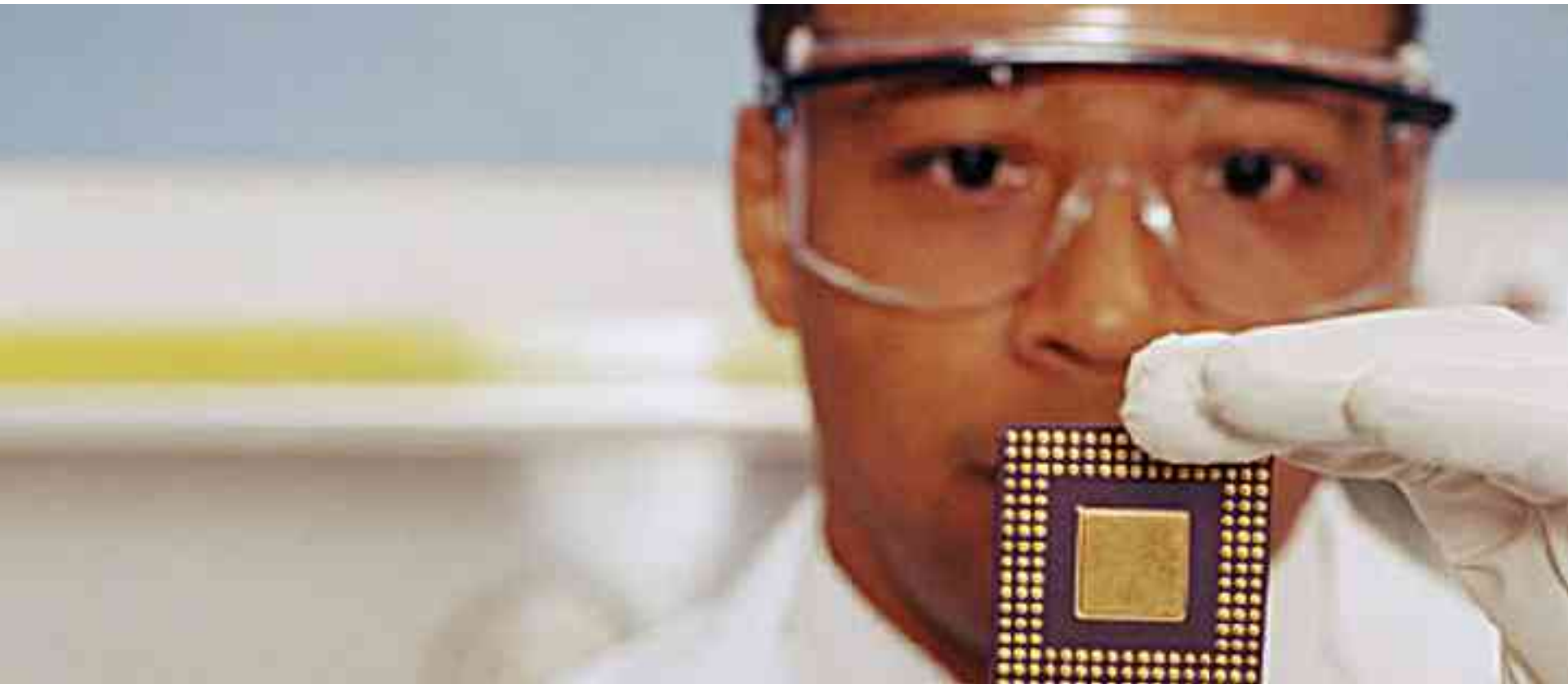
# Performance Improvement

## Solaris Sparc





# Monitoring & Management





# Monitoring & Management

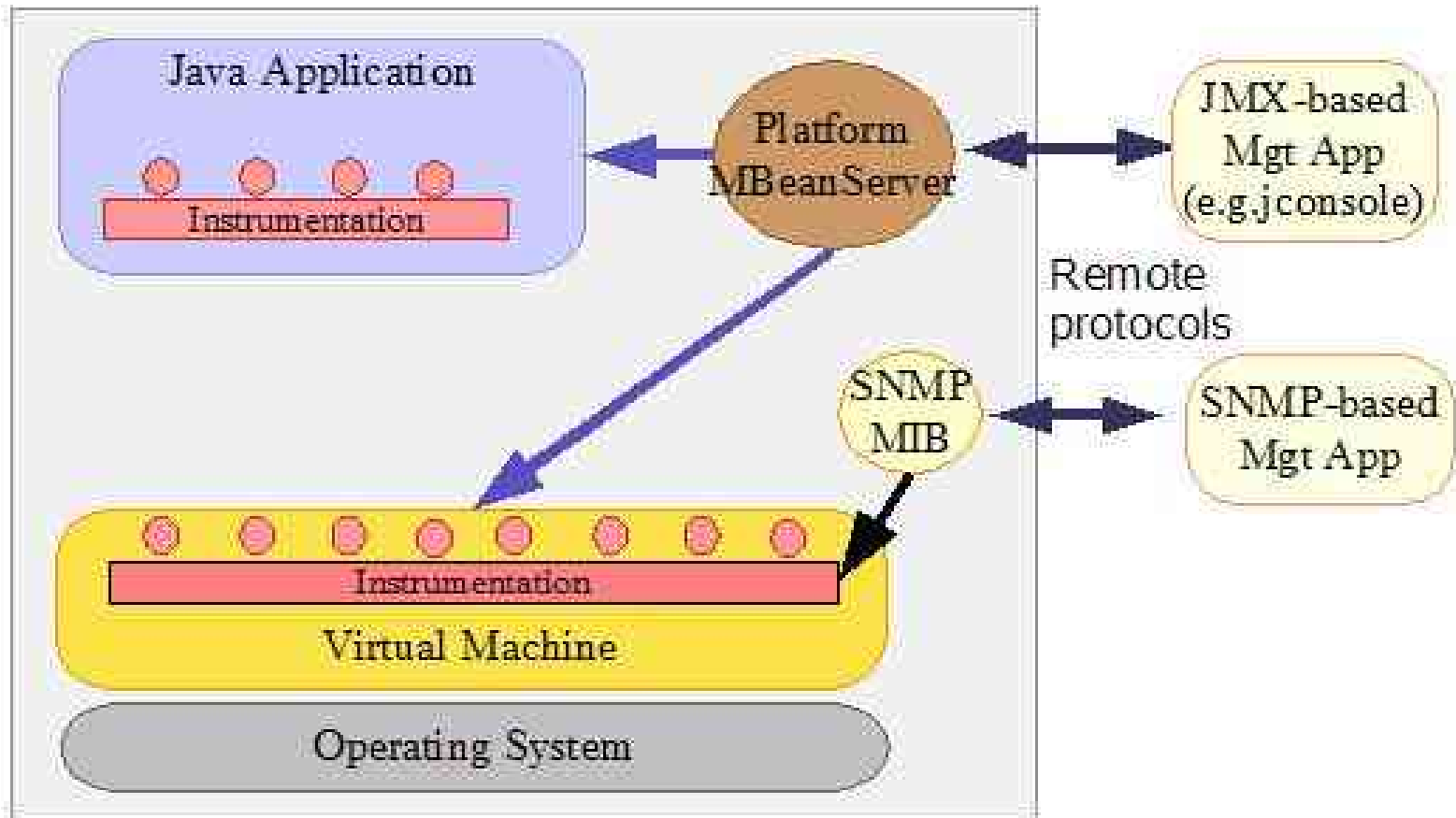
- Key component of RAS in the Java platform (Reliability, Availability, Serviceability)
- Features
  - > JVM instrumentation and integrated JMX
  - > Monitoring and management APIs
  - > Tools



# JVM TI (JVM Tool Interface)

- New native programming interface for use by development and monitoring tools
  - > Replaces JVMPI (JVM Profiler Interface) and JVMDI (JVM Debugger Interface)
- Improved performance analysis
- Java Platform Debugger Architecture uses JVM TI and provides higher-level interface
- Supports bytecode level instrumentation
  - > Provides the ability to alter the Java virtual machine bytecode instructions which comprise the target program

# J2SE 5.0 Monitoring & Management





# Integrated JMX (JSR-003): MBean

- An MBean is a managed object that follows the design patterns conforming to the JMX specification
- An MBean can represent a device, an application, or any resource that needs to be managed
- The management interface of an MBean comprises a set of readable and/or writable attributes and a set of invocable operations
- MBeans can also emit notifications when predefined events occur



# Platform Beans (MXBean's)

- Provides 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

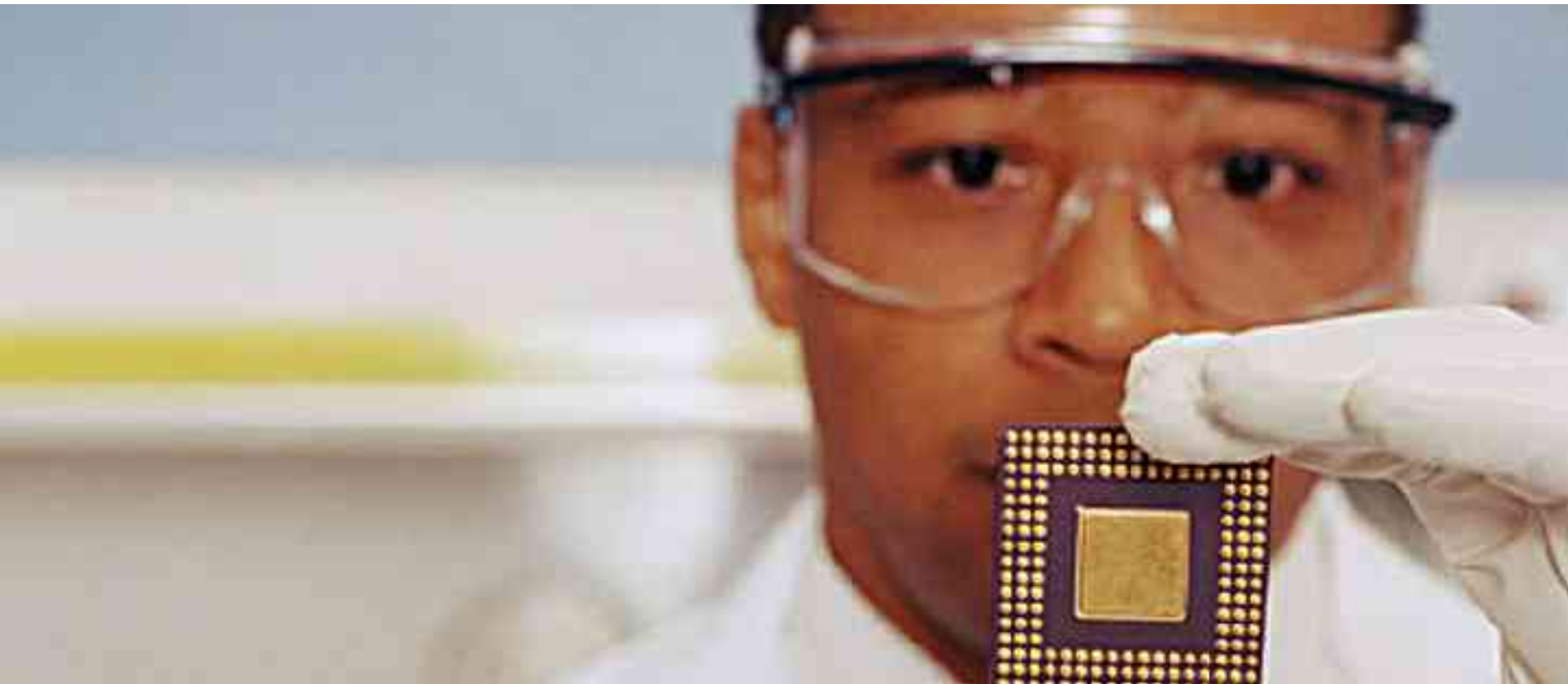


# JConsole

- JMX-compliant GUI tool that connects to a running JVM, which started with the management agent
- To start an application with the management agent for local monitoring, set the `com.sun.management.jmxremote` system property when you start the application
  - > `JDK_HOME/bin/java -Dcom.sun.management.jmxremote -jar JDK_HOME/demo/jfc/Java2D/Java2Demo.jar`
- To start JConsole
  - > `JDK_HOME/bin/jconsole`

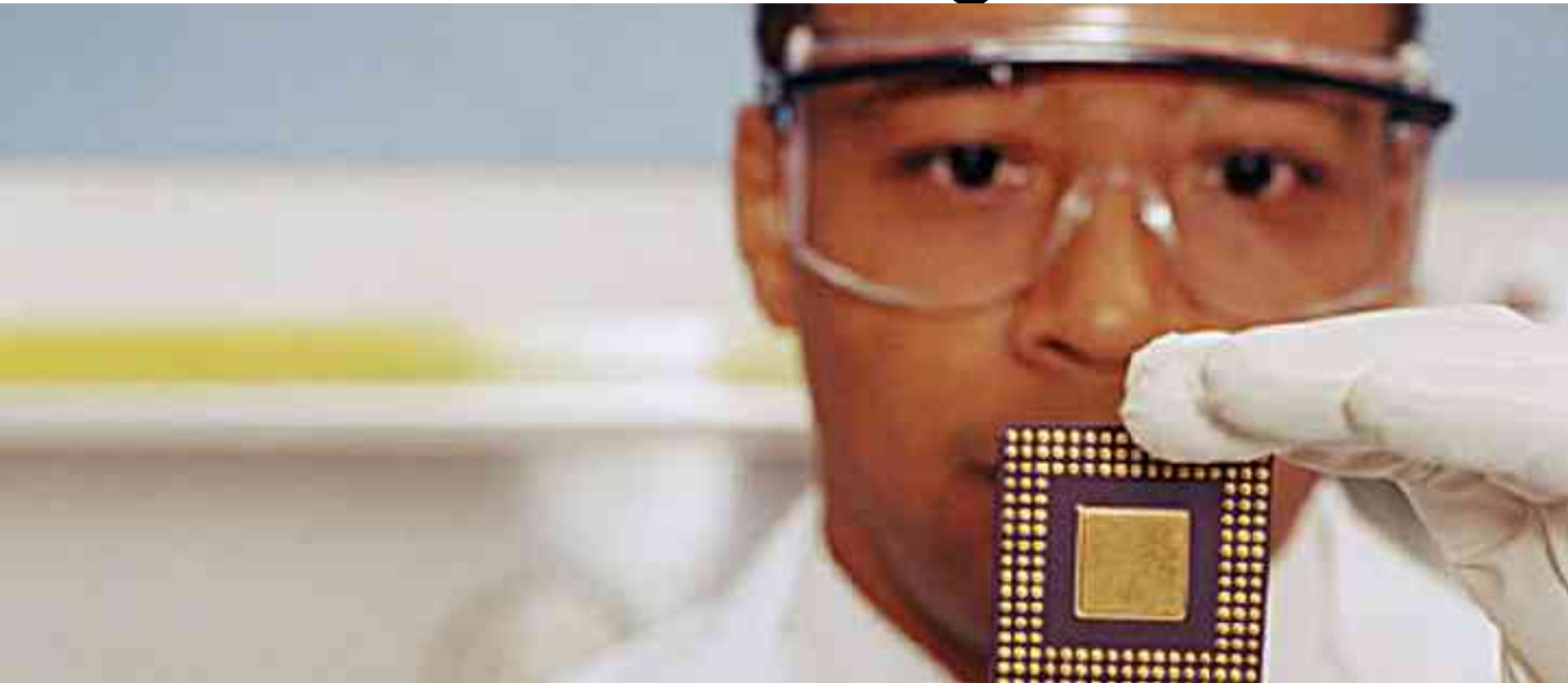


# JConsole Demo





# Next Release: J2SE 6.0 code-named Mustang





# J2SE: The Future

- J2SE 6: Codename “Mustang”
- More community based development
  - > [j2se.dev.java.net](http://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!



# Why are we doing this?

- Innovation Happens elsewhere!
- Doug Lea's work on Concurrency Utilities in J2SE 5.0
- Not about cost reduction
- Why not Open Source?
  - > Compatibility matters!



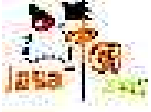
# What do developers want?

- Many discussions with individual developers especially during J2SE 5.0 development
- Four big developer themes surfaced:
  - > They want to see sources
  - > They want to be able to contribute fixes & features
  - > They want to be able to fix bugs themselves
  - > Compatibility Matters
- While specs are developed in the JCP, code matters
- Projects Peabody and GlassFish are a response to these issues



# What you can do

- Easy to download and read J2SE source code as it is being developed
- To do research to innovate and be part of new feature development
- The ability to fix bugs and deploy them internally
- Also to contribute both bug fixes and features back into the mainline J2SE releases



# Three new licenses for specific users

- The **Java Research License (JRL)** for evaluation and non-commercial use
  - > simple two page click-through license
- The **Java Internal Use License (JIUL)** to allow bug-fixing and commercial deployment inside a company – under development
- The **Java Distribution License (JDL)** for full-scale commercial use
- Q: Does looking at source code under the JRL taint me?
- A: No! (See the JRL FAQ)

**Get Involved**

- [Java-net Project](#)
  - [Request a Project](#)
  - [Project Help Wanted Ads](#)
  - [Publicize your Project](#)
  - [Submit Content](#)

**Project tools**

- [Project home](#)
- [Announcements](#)
- [Discussion forums](#)
- [Mailing lists](#)
- [Documents & files](#)
  - [Version control](#) [CVS](#)

**mustang**

Project home

If you were [registered](#) and [logged in](#), you could join this project.Summary [Mustang \(JDK 6.0\) Snapshot Releases](#)Categories [None](#)License [Java Research License \(JRL\)](#)Owner(s) [acbeck](#), [brinkley](#), [mreirhole](#), [peterkessler](#)**Description****J2SE 6.0 Snapshot Releases**

**I NEED  
YOU**

*for*

**MUSTANG  
DEVELOPMENT**



# Sun Developer Network, China

## <http://gceclub.sun.com.cn/>



- Website
  - > Product Information, FAQ, Technical Articles, and Tutorials
- Forum
  - > Sun Engineers to Answer Your Questions
- Download Center
  - > Get Any Software Product from Sun
- Community and User Group
  - > Meet Experts from Sun and Talk to Peers in Your Local Area
- Competition
  - > Excel and Gain Recognition



# Thank You!

**Sang Shin**

Java Technology Architect

Sun Microsystems, Inc.

