State of the Art
in
Server Side Java

Mats Henricson

Last significant update: 2005-07-06

What I will **not** talk about:

- Java ME
- Web frameworks, such as Tapestry, JSF, Wicket, WebWork, etc
- Swing and other client side technologies
- Java Plug-in, Java Web Start, etc

Areas I plan to cover in upcoming revisions of this document are:

- JBI: JSR 208
- JAXB: JSR 222
- Java EE 5.0
- NIO.2: JSR 203, scheduled for Dolphin
- STaX: JSR-173, scheduled for Java EE 5.0
- JAX-WS: JSR 224, scheduled for Java EE 5.0

State of the Art in Server Side Java

http://www.crisp.se/mats.henricson
http://www.henricson.se/mats
mats@henricson.se

I have used Java full time since 1998 and I'm a member of the JCP. In a previous life I was a member of the standardization committee for C++.

Currently I work as a server side programmer at Crisp, a team of highly experienced consultants in Stockholm. From 1999 to 2003 I worked as a Java developer in Silicon Valley, just south of San Francisco.

Please send your comments to mats@henricson.se

I will consistently use the new names for the Java profiles:

J2SE → Java SE

J2EE → Java EE

Agenda

- Annotations
- Dependency Injection
- Hibernate today
- EJB 3.0 Entity Beans
- EJB 3.0 Session and Message Driven Beans

Agenda, cont.

- AOP
- New stuff in Tiger, Mustang and Dolphin
- WS annotations
- Scripting languages
- Tools for quality assurance of software

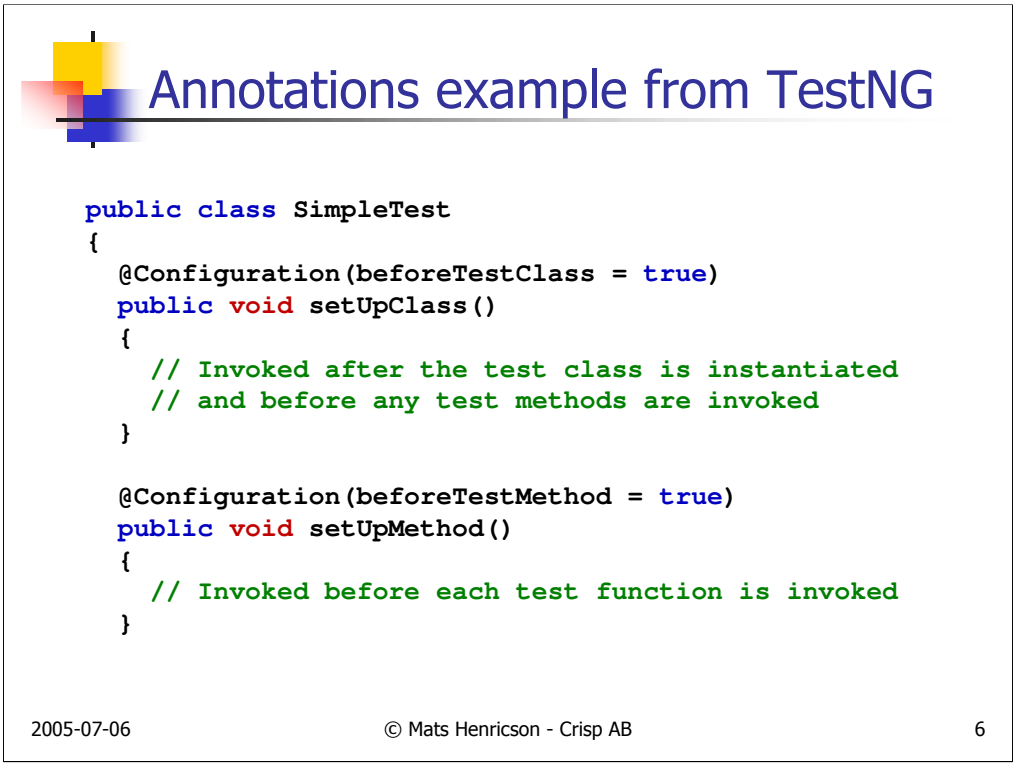
Annotations

Introduced in JDK 1.5 (Tiger)

Also called Metadata and Attributes

The most important addition to Java ever!

Annotations are to some degree replacing XML for configuration.

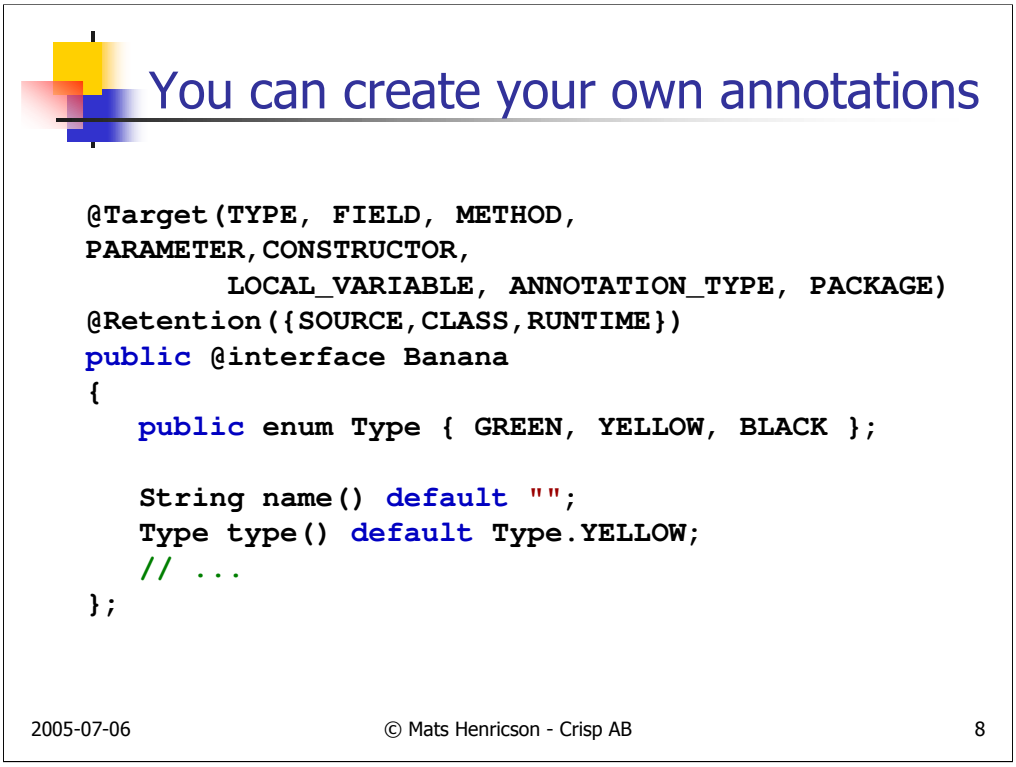


I think the use of annotations in TestNG is a good example, because everyone knows how JUnit works, so the TestNG annotations should be quite easy to understand.

Annotations example, cont.

```
  @Test(groups = {"inittest"})
  public void serverStartedOk()
  {
    // Your test code
  }

  @Test(dependsOnMethods = {"serverStartedOk"},
        groups = {"functest"})
  public void testItWorks()
  {
    // Your test code
  }
}
```



Annotations from different sources may have name conflicts, so namespaces should be used!

A tool called **apt** is used to process annotations at compile time.

Various example annotations

```
@property private String foo;
@Persistent private String street;
@Overrides public
  void mousePressed(MouseEvent e) {}
@Deprecated public void foo(int i) {}
@Asynchronous public void bar(int i) {}
@Resource int maxPercentage = 20;
@EJB private ShoppingCart myCart;
@Stateless public class Cart;
```


In no particular order.

Various example annotations, cont.

```
@Stateful public class Xyz;
@MessageDriven public class Abc;
@Entity public class OrderEntry;
@Id(generate=SEQUENCE)
    public Long getEmpNo() {return empNo;}
@Dependent public Address getAddress()
    {return address;}
@Inject protected
  void setSessionContext(SessionContext ctx)
    {this.ctx = ctx;}
```


There are a bunch of standard annotations described in JSR 250:
http://jcp.org/en/jsr/detail?id=250

Dependency Injection

Also called Inversion of Control (IoC)

The Hollywood Principle:
"Don't call me, I'll call you."

The most well known Dependency Injection framework is Spring. Others are HiveMind and PicoContainer. Limited support for Dependency Injection comes in EJB 3.0.

What is the problem?

```
public class OrderEntryImpl implements OrderEntry
{
   public void enterOrder(int customerId,
                          Order newOrder)
   {
      Customer customer = (Customer)
                  em.find("Customer", customerId);
      newOrder.setCustomer(customer);
      customer.getOrders().add(newOrder);
   }
}
```

The problem is, where does ”em” come from?

One way to solve the problem

```
public class OrderEntryImpl implements OrderEntry
{
   private EntityManager em;

   public void setEntityManager(EntityManager em)
   {
      this.em = em;
   }

   public void enterOrder(int customerId,
                          Order newOrder)
   {
      // ...
   }
}
```

Now, who calls **`setEntityManager()`** ?

With Spring, the BeanFactory calls it!

```
InputStream is =
     new FileInputStream("beans.xml");
XmlBeanFactory bf = new XmlBeanFactory(is);
OrderEntry oe =
     (OrderEntry) bf.getBean("orderEntry");
oe.enterOrder(...);
```

Wait a minute, what happened?

There are numerous ways to create a BeanFactory. The above example is just one way, and not necessarily the best.

Lets look at the beans.xml file

```
<beans>
  <bean id="orderEntry"
        class="com.acme.OrderEntryImpl">
    <property name="entityManager">
      <ref bean="myEntityManager"/>
    </property>
  </bean>
```

There is a reference to myEntityManager, so how is it defined?

beans.xml file, cont.

```
<bean id="myEntityManager"
      class="com.acme.MyEntityManager">
  <property name="dataSource">
    <ref bean="myDataSource"/>
  </property>
</bean>
```

There is a reference to myDataSource, so how is it defined?

beans.xml file, cont.

```
<bean id="myDataSource"
       class="org.apache.commons.dbcp.BasicDataSource"
       destroy-method="close">
    <property name="driverClassName">
      <value>com.mysql.jdbc.Driver</value>
    </property>
    <property name="url">
      <value>jdbc:mysql://localhost:3306/mydb</value>
    </property>
    <property name="username">
      <value>root</value>
    </property>
    <property name="password">
      <value>scott</value>
    </property>
  </bean>
</beans>
```

myDataSource has no external dependencies, so we're done!

Benefits with Dependency Injection

- A factory will improve the design by pushing us towards using interfaces
- Unit testing gets easier, since we can change dependencies just by changing the XML file
- We get thread safe creation of objects
- Pooling can be done by Spring
- We can autowire dependencies by type or name
- Reduced coupling between components

There are some drawbacks too:

1. Weak typing in XML files, so spelling mistakes shows up at runtime, not compile time.
2. Configuration spread to both Java and XML. The Java code gets cleaner with Dependency Injection, while XML configuration can be a bit complex.

Benefits with Dependency Inj., cont.

- Easier to assemble components into new applications
- The user of a class doesn't have to know if it is a singleton or not
- Classes becomes simpler to code, since we can concentrate on the important stuff, not the wiring
- The use of EJB (session beans) can be hidden to calling code, so we can "upscale" later if necessary

Summary: Dependency Injection

Howard M. Lewis Ship, the creator of Tapestry:

> *You're willing to let something else manage the death of your objects because it, the Garbage Collector, can do a better job than you can.*
>
> *Likewise, you should accept that a container, whose only responsibility is to construct and configure your objects, will do a better job of it than your own code. Embrace that fact ... and get back to interesting work!*

Hibernate

What is Hibernate?

An open source framework for mapping Java objects to database tables.

Has taken the Java world by storm.

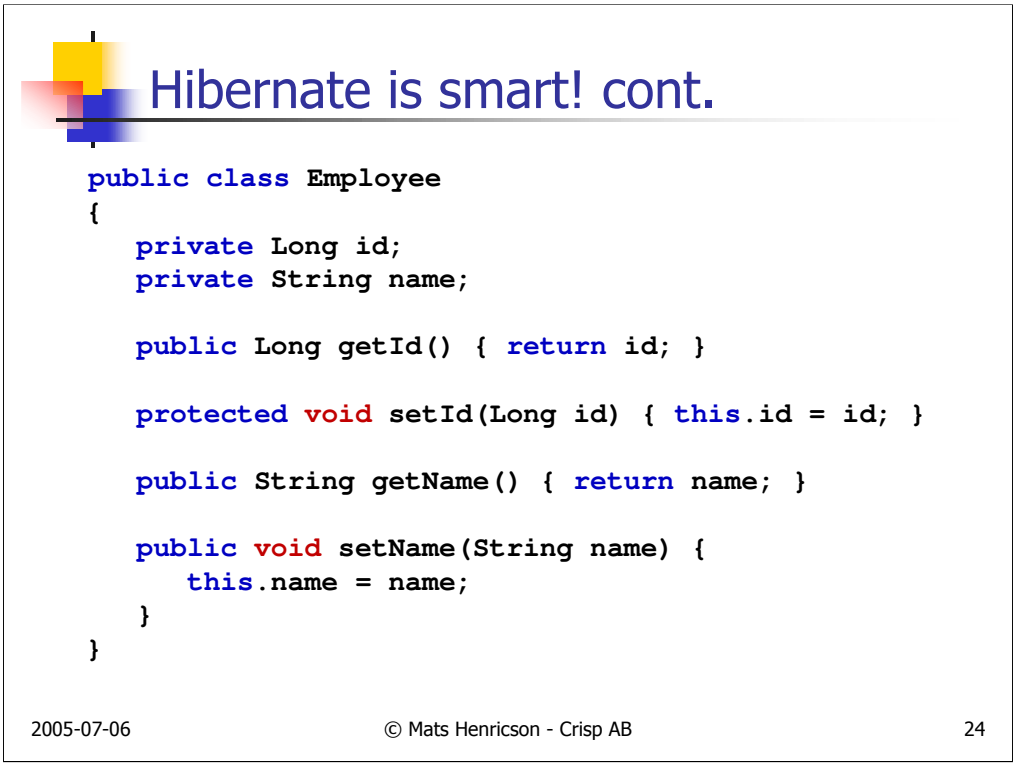
You could argue that Hibernate has been around for a long time and really isn't anything new. But many programmers across the world have not yet switched to frameworks like Hibernate for mapping relational tables to Java objects. This presentation of Hibernate also concentrates on how it works in version 3.0, to give readers a taste of what is the latest in Hibernate.

Basic Hibernate scenario

- Write a file (typically called ClassName.hbm.xml), describing which Java object property maps to which column in which table.
- The file also contains type information so that no data is lost when moved in and out of the database.

Hibernate is smart!

In many cases Hibernate can make intelligent guesses about table names, column names and data types.



Hibernate will assume the existence of a table named EMPLOYEE, containing columns with names ID and NAME. If that is correct, almost no configuration is needed.

Requirements for Hibernate POJO classes:

- Empty constructors
- Needs a primary key, preferrably with no built-in semantics. Hibernate can generate it for you.
- JavaBeans get/set methods

Hibernate features

- Works with basically all relational databases, and it is rather simple to change vendor
- Good performance
- Simple to use
- POJO's !
- Lets you avoid SQL almost entirely
- Supports inheritance and polymorphism
- Supports composition and collections
- Built in cache
- Supports clustering

It can be a bit unnerving to not have total control of how persistence works, but you'd better get used to it! Besides, it's really pretty boring.

Gives you very compact DAO classes

Especially if used with Spring or JDK 5 generics

```
public class ProductDaoImpl
   extends HibernateDaoSupport implements ProductDao
{
   public List loadProductsByType(String type)
   {
      return getHibernateTemplate().find(
        "from test.Product p where p.type=?", type);
   }
}
```

The returned List is a list of Product objects!

Guidelines for writing DAO classes:

- First write an interface, such as ProductDao, with no coupling to Hibernate
- Then write a specific ProductDaoHibernateImpl that implements ProductDao
- Use Java 5 generics or Spring to simplify the ProductDaoHibernateImpl class
- Use Spring to get the DAO objects, so that you can switch DAO implementation, for example during testing
- Write BusinessObjects (session beans) that gets objects from DAOs and does business stuff with the objects

Options for how to use Hibernate

- Start with the DB and use Middlegen to generate XML mapping files, then use hbm2java to generate JavaBeans from the XML mapping files
- Start with JavaBeans and use XDoclet to generate XML mapping files from special JavaDoc tags, then use hbm2ddl to generate the DB (more likely, update the DB by hand)
- Start with the XML mapping files and generate the DB with hbm2ddl, then use hbm2java to generate JavaBeans
- Use attributes/annotations just like EJB 3.0 (just released)

  For now, starting with JavaBeans and XDoclet is probably the best option.

Hibernate XDoclet example

```
/**
 * @hibernate.class
 *  table="CATS"
 */
public class Cat
{
   private Long id;
   private Cat  mother;
   private Set  kittens;
```

Hibernate XDoclet example, cont.

```
/**
 * @hibernate.id
 *  generator-class="native"
 *  column="CAT_ID"
 */
public Long getId()
{
   return id;
}

private void setId(Long id)
{
   this.id = id;
}
```

Hibernate XDoclet example, cont.

```
/**
 * @hibernate.many-to-one
 *   column="PARENT_ID"
 */
public Cat getMother()
{
   return mother;
}

void setMother(Cat mother)
{
   this.mother = mother;
}
```

Hibernate XDoclet example, cont.

```
/**
 * @hibernate.set
 *  inverse="true"
 *  order-by="BIRTH_DATE"
 * @hibernate.collection-key
 *  column="PARENT_ID"
 * @hibernate.collection-one-to-many
 */
public Set getKittens()
{
   return kittens;
}
void setKittens(Set kittens)
{
   this.kittens = kittens;
}
```

Typical XML mapping file example

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD//EN"
   "http://hibernate.sourceforge.net/
               hibernate-mapping-1.1.dtd">
<hibernate-mapping>
  <class name="xxx.yyy.Customer"
         table="customers">
    <id name="id" column="id" type="long">
      <generator class="hilo.long"/>
    </id>
    <property name="name"    type="string"/>
    <property name="created" type="timestamp"/>
    <property name="xxxx"    type="long"/>
  </class>
</hibernate-mapping>
```


This is just a sketch. Normally they're much longer and more complex.

EJB 3.0 Entity Beans

Will totally change the way we do EJB

Inspiration from Hibernate and Spring

Goals for EJB 3.0 Entity Beans design

- **Ease of development!**
- No interfaces needed, just the POJO with annotations!
- No more home interfaces
- Callback methods no longer necessary
- Possible to test and use outside a Javs EE server
- Created with new()
- No need to inherit or extend any specific EJB base class or interface

Goals for EJB 3.0 Entity Beans, cont.

- True and natural inheritance between entity beans, with three supported patterns for how to split data between tables
- Deployment descriptors will not be necessary, unless you want to override default values - Configuration By Exception
- Support for native SQL
- Limited support for dependency injection a la Spring, but using annotations

There are a few restrictions:

1. Must be annotated with @Entity
2. A public or protected no-args constructor is required
3. The class and its functions must not be declared final
4. Must have a serializable primary key, preferrably long or int
5. Properties must be accessed through public or protected JavaBeans get/set functions, or be protected members
6. Persistent properties must not be public
7. Collections of persistent data must be specified as java.util.Collection, java.util.Set, java.util.List or java.util.Map
8. CRUD of entity beans is through an entity manager which is accessed through JNDI or dependency injection
9. If entity beans inherit eachother, then their primary keys must be of the same type

EJB 3.0 Entity Beans features

- OneToOne, OneToMany, ManyToOne and ManyToMany relationships can be annotated directly in the Java classes
- Cascading deletes can be annotated directly for OneToOne and OneToMany relationships
- Mapping to specific tables and columns can be done directly in Java with annotations
- Callback lifecycle methods can now have any name, or even belong to a separate class, and one function can be used for several callbacks

Simple EJB 3.0 Entity Bean example

```
@Entity
public class Employee
{
  private Long id;
  private String name;

  @Id
  public Long getId() { return id; }

  protected void setId(Long id) { this.id = id; }

  public String getName() { return name; }

  public void setName(String name) {
     this.name = name;
  }
}
```


In this case, the container will assume there is an EMPLOYEE table, with columns ID and NAME. Almost no extra configuration is needed if these assumptions are true.

Lets override the table name

```
@Entity
@Table(name="EMPL")
public class Employee
{
    // ...
}
```


The container will now save Employee objects in the EMPL table.

Lets use a secondary join table

```
@Entity
@Table(name="EMPL")
@SecondaryTable(name="EMPL_DETAILS",
     pkJoin=@PrimaryKeyJoinColumn(name="EMPL_ID"))
public class Employee
{
   // ...
}
```


In this case Employee objects are saved in two tables; EMPL and EMPL_DETAILS.

Lets annotate a column

```
@Entity
public class Employee
{
  // ...
  @Column(name="name",
          nullable=false,
          length=100)
  public String getName()
  {
    return name;
  }
  // ...
}
```


Always annotate the get function

Lets use lazy loading

```
@Entity
public class Employee
{
    // ...
    @Basic(fetch=LAZY)
    public String getName()
    {
      return name;
    }
    // ...
}
```

2005-07-06 © Mats Henricson - Crisp AB 41

Lazy loading is just a hint, implementations are allowed to prefetch.

Lets use blobs and clobs

```
@Entity
public class Employee
{
  // ...
  @Lob
  @Column(name="PHOTO")
  protected JPEGImage picture;

  @Lob(fetch=EAGER, type=CLOB)
  @Column(name="REPORT")
  protected String report;
  // ...
}
```

ManyToOne example

```
@Entity
public class Employee
{
  private Department department;

  @ManyToOne
  public Department getDepartment()
  {
    return department;
  }
  // ...
}
```

ManyToOne example, cont.

```
@Entity
public class Department
{
  private Collection<Employee> employees =
                                  new HashSet();

  @OneToMany(mappedBy="department")
  public Collection<Employee> getEmployees()
  {
    return employees;
  }
  // ...
}
```

ManyToMany example

```
@Entity
public class Project
{
   private Collection<Employee> employees;

   @ManyToMany
   public Collection<Employee> getEmployees()
   {
      return employees;
   }
}
```

ManyToMany example, cont.

```
@Entity
public class Employee
{
    private Collection<Project> projects;

    @ManyToMany(mappedBy="employees")
    public Collection<Project> getProjects()
    {
        return projects;
    }
    // ...
}
```

Assumes there is a PROJECT_EMPLOYEE association table.

It takes some time to understand the default naming convention for these association tables.

Annotation of a lifecycle method

```
@Entity
public class Employee
{
   // ...
   @PrePersist
   public void beforePersistence()
   {
      // ...
   }
   // ...
}
```


Other annotations for lifecycle functions:

- @PrePersist
- @PostPersist
- @PreRemove
- @PostRemove
- @PreUpdate
- @PostUpdate
- @PostLoad

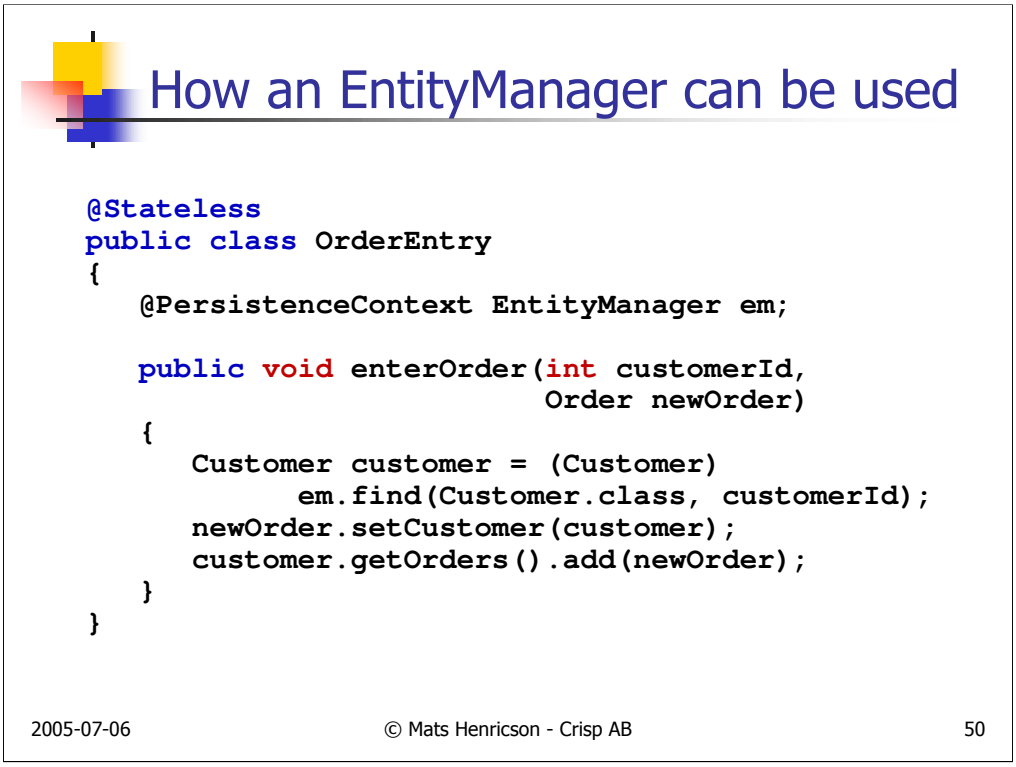
The EntityManager interface

```
public interface EntityManager
{
  // ...
  public void persist(Object entity);
  public void remove(Object entity);
  public <T> T find(Class<T> entityClass,
                    Object primaryKey);
  public void flush();
  public void refresh(Object entity);
  public boolean contains(Object entity);
  // ...
```

The EntityManager interface, cont.

```
    // ...
    public Query createQuery(String ejbqlString);
    public Query createNamedQuery(String name);
    public Query createNativeQuery(String sqlString);
    public Query createNativeQuery(String sqlString,
                                   String resultSetMapping);
    // ...

}
```



Please note how Dependency Injection gives us the EntityManager!

In an environment without a Java EE 5.0 application server, you can get an EntityManager by using factories, like this:

```
   EntityManagerFactory emf =
javax.persistence.Persistence.createEntityManagerFactory
();
   EntityManager em = emf.createEntityManager();
```

The Query interface

```
public interface Query
{
  public List getResultList();
  public Object getSingleResult();
  public int executeUpdate();
  public Query setMaxResults(int maxResult);
  public Query setFirstResult(int startPosition);
  public Query setParameter(String name,
                            Object value);
  public Query setParameter(int position,
                            Object value);
  // ...
}
```

How a Query can be used

```
public List findWithName(String name,
                         int numberOfHits)
{
   return em.createQuery(
         "SELECT c FROM Customer c" +
         " WHERE c.name LIKE :custName")
         .setParameter("custName", name)
         .setMaxResults(numberOfHits)
         .getResultList();
}
```

Goals for EJB 3.0 Session Beans design

- **Ease of development!**
- No particular EJB interfaces needs to be implemented or extended, just the POJO with annotations
- No more home interfaces
- No longer necessary to implement callback methods
- Possible to test and use outside a Java EE server
- Deployment descriptors will not be necessary, unless you want to override default values - Configuration By Exception
- Simplified exception handling
- Limited support for dependency injection a la Spring, but using annotations

There are a few basic restrictions:

1. Must be annotated with @Stateless, @Stateful or @MessageDriven
2. A public or protected no-args constructor is required
3. The class and its functions must not be declared final
4. The class must not be abstract
5. The class must be public
6. May not define the finalize() method
7. Session and message driven beans requires an interface (for message driven beans it is most likely javax.jms.MessageListener)
8. Session and message driven beans are accessed only through dependency injection or straight JNDI lookup

Simple EJB 3.0 Session Bean example

```
@Stateful
public class CartBean
        implements ShoppingCart
{
   private double total;
   private List   productCodes;

   public void shipProducts(...)
   {
      // ...
   }
}
```

Annotation of a lifecycle method

```
@Stateful
public class CartBean
        implements ShoppingCart
{
    // ...
    @PreDestroy
    public void endShopping()
    {
        // ...
    }
}
```


Lifecycle annotations:

- @PostConstruct
- @PreDestroy
- @PostActivte
- @PrePassivate

Other EJB 3.0 Session Bean features

- @Remove annotates a function to call when the container removes a stateful session bean
- @AroundInvoke annotates Interceptor functions and classes, called when business functions are called, a bit like primitive AOP
- Transaction annotations

EJB 3.0 Dependency Injection

Objects are looked up in JNDI, and inserted before any other functions are invoked on the objects.

```
// Correct type is inferred
@Resource(name = "myDb")
public DataSource customerDb;
```

2005-07-06 © Mats Henricson - Crisp AB 57

Where are objects looked up? It starts in **java:comp/env**, and then we have:

For EJB: **java:comp/env/ejb**
For web services: **java:comp/env/services**
For EntityManager's: **java:comp/env/EntityManager**

For resource manager connection factories:
- **java:comp/env/jdbc**
- **java:comp/env/jms**
- **java:comp/env/mail**
- **java:comp/env/url**
- etc

See spec for details

EJB 3.0 Dependency Injection, cont.

If type and name is correct by default

```
@Resource
public DataSource customerDb;
```

EJB object injection

```
@EJB
public ShoppingCart myShoppingCartBean;
```

EJB 3.0 Dependency Injection, cont.

Dependency Injection through setters

```
@Resource(name = "customerDb")
public void setDataSource(DataSource myDb)
{
   this.ds = myDb;
}
```

If type and name is correct by default

```
@Resource
public void setDataSource(DataSource myDb)
{
   this.ds = myDb;
}
```

Aspect Oriented Programming (AOP)

For concerns that spans "horizontally" across many modules of a system

- Doing it in OO **may** result in complex designs
- Doing it in AOP **may** result in simple designs

Examples (trivial, contrieved or real)

- Logging
- Transactions
- Authorization
- Caching
- Persistence
- Resource pooling
- Policy enforcement (e.g. object creation by factories)
- Design by contract

How are aspects weaved into code?

There are a few different approaches

- Language extensions with source generator.
  Example: AspectJ
- XML or XDoclet configuration with byte code modification by class loader.
  Example: AspectWerkz

Claimed AOP benefits

- Modularized implementation of crosscutting concerns
- Easier-to-evolve systems
- Late binding of design decisions
- More code reuse

Claimed AOP drawbacks

- Debugging gets harder, since "invisible" code is executed
- "The **goto** of Object Oriented Programming"

AspectJ example

```
// HelloWorld.java
public class HelloWorld
{
   public static void main(String args[])
   {
      HelloWorld world = new HelloWorld();
      world.greet();
   }

   public void greet()
   {
      System.out.println("Hello World!");
   }
}
```

AspectJ example, cont.

```
// MannersAspect.java
public aspect MannersAspect
{
   pointcut callSayMessage() :
            call(public void HelloWorld.greet*(..));

   before() : callSayMessage()
   {
      System.out.println("Good day!");
   }

   after() : callSayMessage()
   {
      System.out.println("Thank you!");
   }
}
```


Running the program produces this result:

Good day!
Hello World!
Thank you!

AspectWerkz example

```
// HelloWorld.java
public class HelloWorld
{
   public static void main(String args[])
   {
      HelloWorld world = new HelloWorld();
      world.greet();
   }

   public String greet()
   {
      System.out.println("Hello World!");
   }
}
```


Identical HelloWorld.java file as for AspectJ.

AspectWerks example, cont.

```
// MyAspectWithAnnotations.java
import org.codehaus.aspectwerkz.joinpoint.JoinPoint;

public class MyAspectWithAnnotations
{
  /**
   * @Before execution(* HelloWorld.greet(..))
   */
   public void beforeGreeting(JoinPoint joinPoint) {
      System.out.println("Good day!");
   }

  /**
   * @After execution(* HelloWorld.greet(..))
   */
   public void afterGreeting(JoinPoint joinPoint) {
      System.out.println("Thank you!");
   }
}
```


Running the program produces this result:

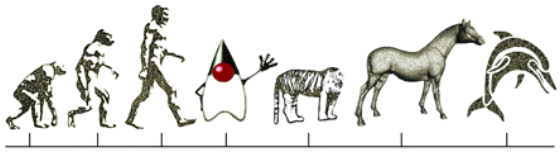
Good day!
Hello World!
Thank you!

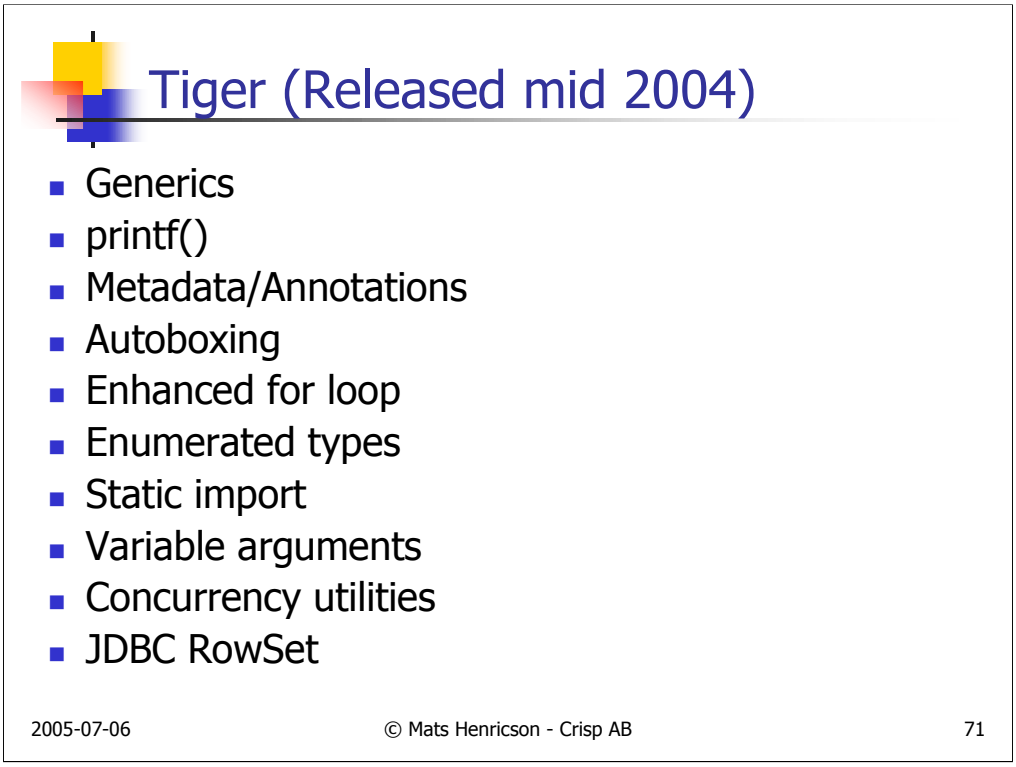
AspectJ and AspectWerkz have joined

- Are considered the leaders in this field
- The only real contender is Spring AOP
- AspectWerkz claims to be the speed champion

Tiger, Mustang and Dolphin

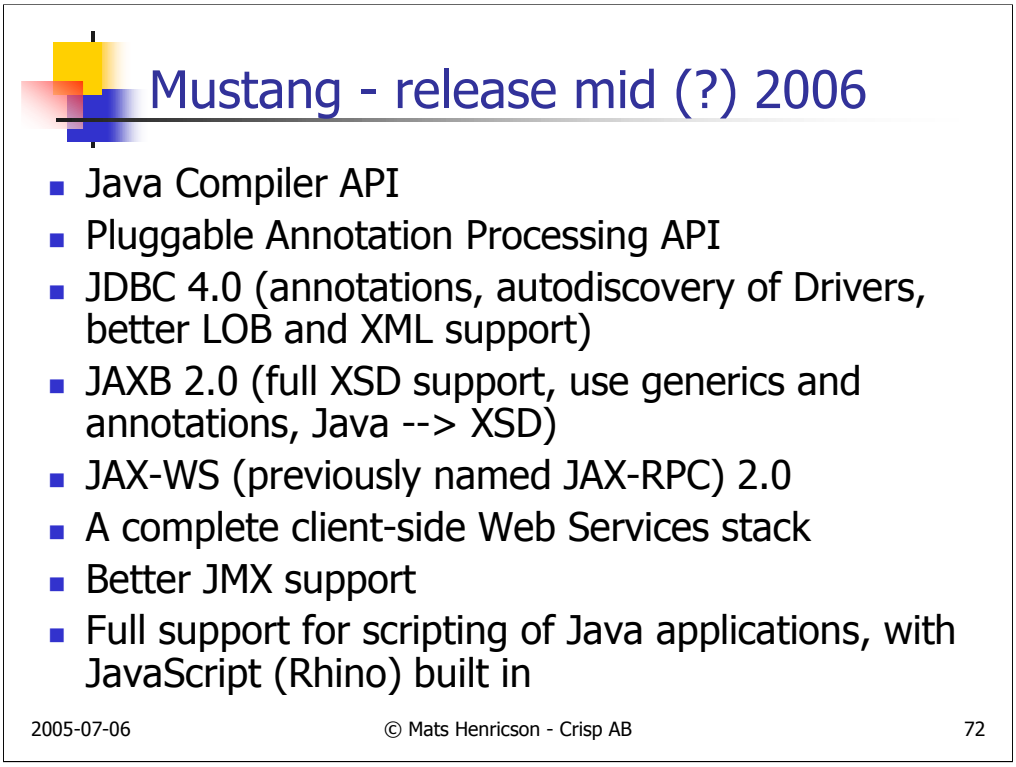
- Java SE 5.0 (Tiger)
- Java SE 6.0 (Mustang)
- Java SE 7.0 (Dolphin)





Five Favourite Features from 5.0: http://www.onjava.com/lpt/a/5799

Pretty good book about the details: Java 2 v5.0 (Tiger) New Features



Desktop Java Features in Mustang:
http://java.sun.com/developer/technicalArticles/J2SE/Desktop/mustang/index.html

Core Java Technology Features in Mustang:
http://java.sun.com/developer/technicalArticles/J2SE/Desktop/Mustang_build39.html

Dolphin - release early 2008

- Direct support for XML into the Java language?
- Upgrading the current JAR/WAR/EAR package system to make it easier to bundle and distribute complex applications: JSR 277
- Package friends?
- Add a new Java Virtual Machine instruction, targeted at "dynamic languages," such as Groovy or Python?
- Method references?
- BeanShell?

http://68.236.189.240/article/special-20050515-01.html (Requires free registration)

About adding better support for XML to Java:

http://radio.javaranch.com/val/2005/06/16/1118933294448.html

http://radio.javaranch.com/val/2005/06/23/1119534631775.html

Three types of WS annotations

- WSDL mapping annotations controls the mapping from Java to WSDL, either what the WSDL should look like, or, how Java code should be associated with existing WSDL
- Binding annotations specify network protocols and message formats
- Handler annotations specifies what handlers should run before and after the business methods themselves

Simple WS annotation example

```
@WebService
public class HelloWorldService
{
  @WebMethod
  public String helloWorld()
  {
    return "Hello World!";
  }
}
```


Requirements on WS Java code:

- Must not be an inner class
- Must not be final or abstract
- Must have a default no-args constructor
- Must not define finalize()
- WebMethod functions must be public

The @WebService annotation

```
@Target({TYPE})
public @interface WebService
{
   // Real default is ClassName
   String name() default "";

   // Real default is ClassNameService
   String serviceName() default "";

   String targetNameSpace()   default "";
   String wsdlLocation()      default "";
   String endpointInterface() default "";
};
```

The @WebMethod annotation

```
@Target({METHOD})
public @interface WebMethod
{
    // Real default is functionName
    String operationName() default "";

    String action() default "";
};
```

The @OneWay annotation

```
@Target({METHOD})
public @interface OneWay
{
};
```

Such functions must have no return value, and no INOUT or OUT parameters.

The @WebParam annotation

```
@Target({PARAMETER})
public @interface WebParam
{
   public enum Mode { IN, OUT, INOUT };

   // Real default is paramName
   String name() default "";

   String targetNameSpace() default "";
   Mode mode() default Mode.IN;
   boolean header() default false;
};
```

The @WebResult annotation

```
@Target({METHOD})
public @interface WebResult
{
  String name() default "return";
  String targetNameSpace() default "";
};
```


I think ”return” is a bad choice of name. To me, web services are about requests and responses. Please also note that the annotation is called WebResult. So, we have the names Result, Return and Response, all about the same thing. Ouch!

The @SOAPBinding annotation

```
@Target({TYPE})
public @interface SOAPBinding
{
  public enum Style { DOCUMENT, RPC };
  public enum Use { LITERAL, ENCODED };
  public enum ParameterStyle { BARE, WRAPPED };

  Style style() default Style.DOCUMENT;
  Use use()     default Use.LITERAL;
  ParameterStyle parameterStyle()
                default ParameterStyle.WRAPPED;
};
```


There are a few other annotations not described here, mostly mapping handler chains and SOAP.

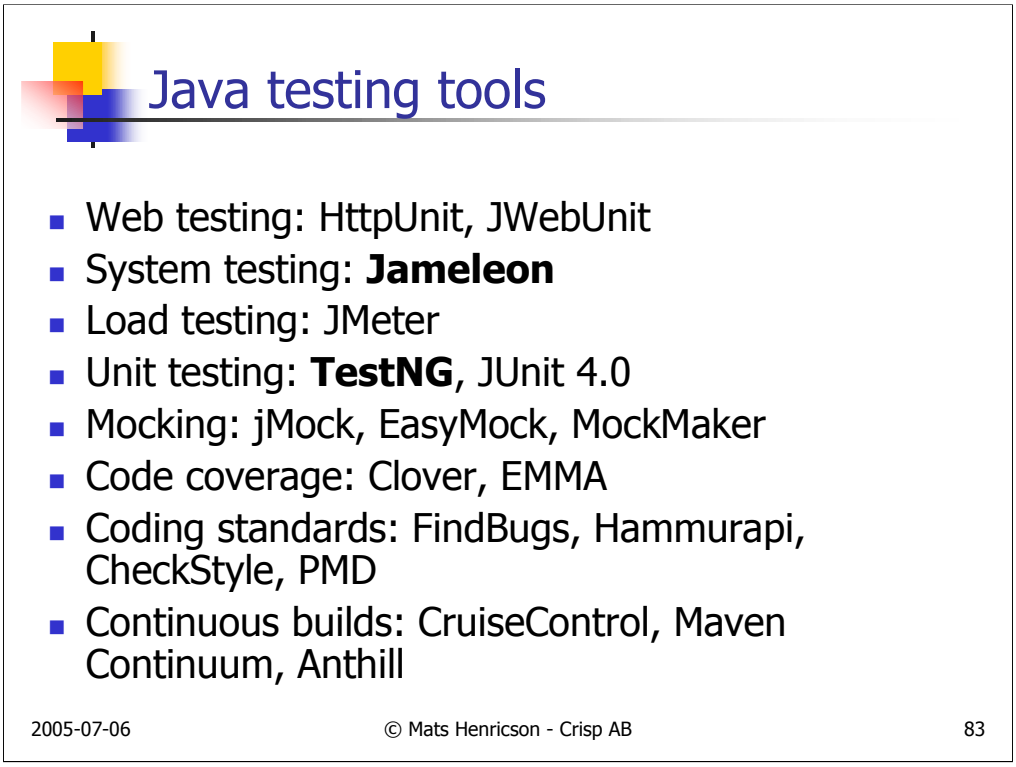
Java scripting languages

The languages with best traction right now

- Groovy - has a JSR, but bleeding edge right now
- Jython - good documentation and pretty fast
- Rhino - fast, has a debugger, and will be shipped with Java SE 6.0!
- Beanshell - slower, but good Java support, and has a JSR, probably shipped with Java SE 7.0!

Choosing a Java scripting language: Round two:
http://www.javaworld.com/javaworld/jw-03-2005/jw-0314-scripting.html

My blog:
http://www.jroller.com/page/matsh/20050613#java_scripting_language_fallout



There are also a lot of metrics tools: JavaNCSS, JDepend, OptimalJ, Dependency Finder, etc

As well as NASA's Java PathFinder tool: an explicit state software model checker for Java bytecode.

More tools:

http://www.manageability.org/blog/stuff/open-source-automated-test-tools-written-in-java

http://java-source.net/open-source/testing-tools

Jameleon

A very well designed open source tool for system or acceptance-level automated testing. It is best suited for testing of web sites, where navigation, action and validation is split into reusable building blocks. It uses JWebUnit or HttpUnit for the HTTP requests.

TestNG

Fixes many of the problems with JUnit

- Test objects are not instantiated every time a test method is executed
- Uses Java SE 5.0 annotations (or javadoc comments) for metadata
- Runtime configuration is captured in an xml file
- You can specify test groups (a 'broken' group, for example)
- You can specify test dependencies (test X requires Y and Z to be run first)
- No need to start all test functions with 'test'
- No need to extend a base class
- You can specify a timeout on a per test basis

The question is if TestNG can get the support needed from tools vendors, in the same way as JUnit. JUnit has essentially been abandonware for some time, but work is underway by Beck and Gamma to design version 4.0. It will use annotations, probably in the same way as TestNG. Noone knows when it will be finished, and if it will be backwards compatible.

http://www.theserverside.com/news/thread.tss?thread_id=34478