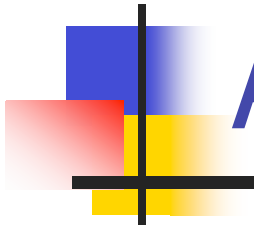


# Annotated Java

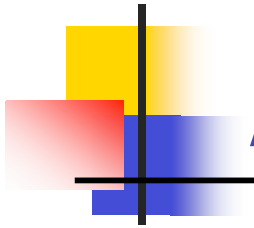
## Annotations in J2SE 5.0

Dave Landers

BEA Systems, Inc.

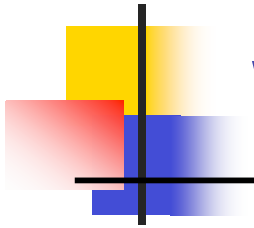
Colorado Software Summit

bea™



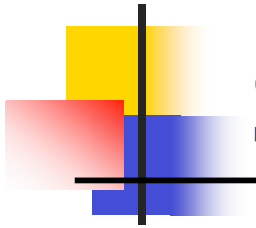
# Agenda

- Overview
- Annotations built in to J2SE 5.0
- Defining Annotations
- Meta-Annotations
- Using Annotations
  - Reflection
  - apt
- Misc Topics



# What Are Annotations?

- New Java modifier
  - Sits with public, static, final, *etc.* in your source
- You (or someone) defines the annotations
  - What they mean or do
- Do not affect semantics of the class itself
  - Unlike other Java modifiers
- May affect semantics of things *using* the class
  - How code is handled by tools and libraries
    - Code generation, runtime options, containers, *etc.*



# Simple Example

- The annotation definition

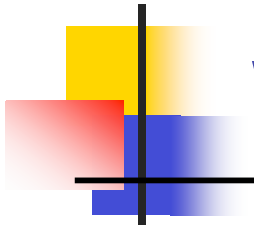
```
@interface FixMe { String value(); }
```

  - Defines a "Fix Me" annotation
  - Has one String attribute (value)

- The Usage

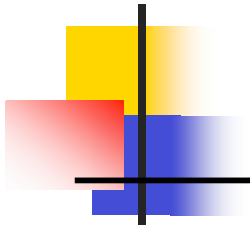
```
@FixMe( "Missing method body" )
  public void theMethod() { }
```

  - Adds FixMe annotation as modifier to the method
  - To be used at build- or run-time
    - Automated tests might print report of all @FixMe's



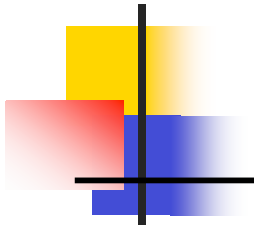
# What Can Be Annotated?

- Any program element
  - Package
    - package-info.java
  - Types
    - Class, Interface, Enum definition, Annotation Type
  - Method, Constructor, Field, Enum constant, Method parameter
  - Local Variable declaration



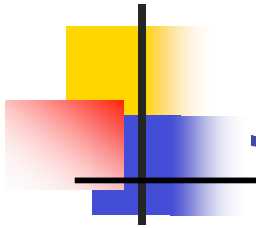
# Why Would I Use This?

- EJB, Web Services, *etc.*
  - Replace or supplement descriptors with annotations
  - Code generate boring classes/interfaces
    - Annotated EJB implementation to generate Home, Remote, *etc.*
    - Annotated Web Service implementation to generate Servlet binding, JAX-RPC interfaces, *etc.*
- Your use case for generating code from annotated classes/interfaces
  - JavaBeans, Logger utilities, Debugging classes, *etc.*
- Recognizing special methods, classes, *etc.* at runtime
  - Test methods, plug-in points, UI elements, *etc.*
  - AOP crosscuts



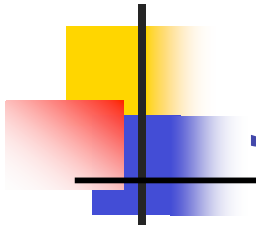
# Compare with XDoclet, *etc.*

- XDoclet is source-only (JavaDoc) annotations
  - Useful for build-time source code processing
- Annotations are:
  - Modifiers, not documentation
    - Part of the code
  - Strongly typed
    - @interface
  - Can persist in the class file
    - Don’t need source code to be useful
  - Can persist at runtime
    - Processing at run-time or deploy-time



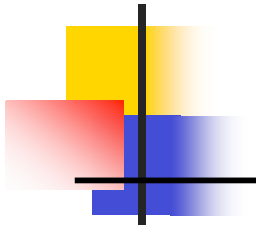
# J2SE Built-In Annotations

- Since this is so useful, there must be lots of new annotations in J2SE 5.0
  - Right?
- Well…
  - @Override
  - @Deprecated
  - @SupressWarnings



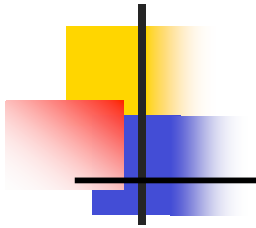
# J2SE Built-in Annotations

- All are used by javac
  - java.lang.*
- @Deprecated
  - Like javadoc's @deprecated
    - Without support for comments, replacement APIs
  - RUNTIME retention policy
    - Allows inspection at runtime, if annotation is used
- @SuppressWarnings( { "unchecked", "deprecated" } )
  - Compiler should ignore specified warnings
    - For example: Suppress type safety warnings on field definition when not using generics
    - Could be very handy for asserting that you know what you're doing
  - Not currently implemented by javac (Bug ID 4986256)
    - And *what* to ignore is compiler-specific (not documented by SuppressWarnings)



# @Override Annotation

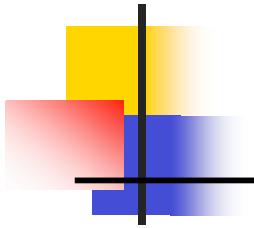
- Very useful
- Asserts that you intend to override a method in a superclass
- Compiler will fail if not actually overriding
  - Without this would silently create method with new signature
- Checks Override *vs.* Overload
- Checks for "missing" methods in base class



# @Override example

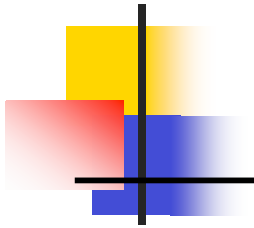
```
class Base {
    void m(Type2 a, Type1 b) { }
}

class Sub extends Base {
  @Override void m(Type1 a, Type2 b) {…}
}
```

```
Sub.java:6: method does not override a method from its
   superclass
  @Override void m(Type1 a, Type2 b)
                  ^
```



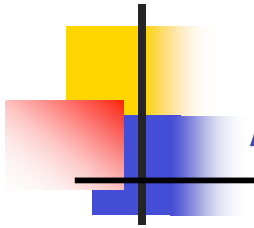
# Kinds of Annotations

- Marker annotations
  - Have no attributes
  - @Override
  - @Deprecated
  - @Preliminary
- Single Value annotations
  - @Copyright( "2004, Dave Landers" )
  - @SuppressWarnings({"unchecked", "deprecation" })
    - Single value is a String[]
- Multi-valued annotations
  - @Review( reviewer="Landers", date="4/1/2004", comment="Close stream in finally block" )



# Defining Annotations

- Defined as @interface
- Compile into java class files
- Automatically extends java.lang.annotation.Annotation
  - You don't write "extends Annotation"
  - Extending Annotation does *not* make an annotation, only @interface marks an annotation
- Annotations can have attributes
  - No-argument methods on the @interface
  - Types can be Primitives, String, enums, other annotations, or arrays of these types
  - Can have default values

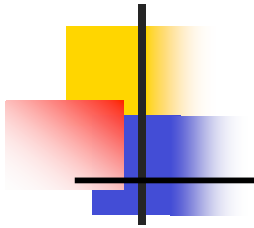


# Annotation Definitions

```
@interface Review {
 String reviewer() default “[unknown]”;
 String date() default “0/0/00”;
 String comment();
}
```

- Usage

```
@Review( reviewer=“Landers”,
         comment=“Does not say hello” )
  public void helloWorld() { }
```



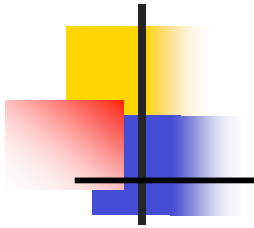
# Single Value Annotations

- Shortcut for Annotations with single attribute
  - Method named value()

```
@interface Copyright {
   String value();
}
```

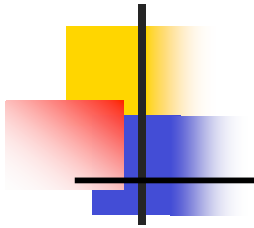
- Usage - don't need the attribute name

```
@Copyright( "2004, Dave Landers" )
   public class OriginalWork { }
```



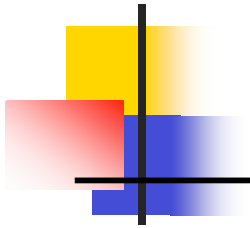
# Code Break

- Example Annotation Definitions
  - SimpleAnnotations.java



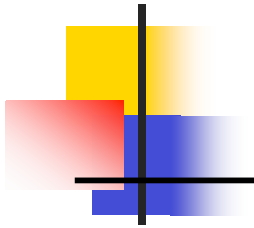
# Meta-Annotations

- Annotations used when defining Annotations
- Specify how the Annotation can be used used
  - Defined in java.lang.annotation.*
  - @Documented
  - @Inherited
  - @Target
  - @Retention



# Meta-Annotations

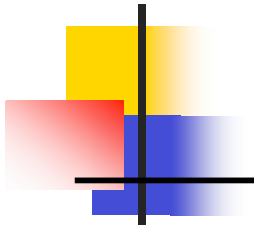
- @Documented
  - Javadoc should be generated when this annotation is applied to an element
  - Is the use of the annotation part of the public API?
- @Inherited
  - Does the annotation get applied to subclasses or only to the base type?
  - Only works on classes
    - Not overriden Methods
    - Not Interfaces



# @Target Meta-Annotation

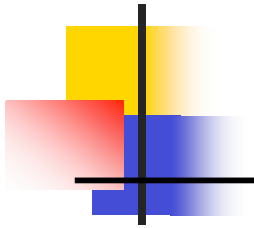
- Where the annotation can be used
  - What kind of source elements
  - Default is all

```
public @interface Target {
   ElementType[] value();
}
public enum ElementType { ANNOTATION_TYPE,
  CONSTRUCTOR, FIELD, LOCAL_VARIABLE,
  METHOD, PACKAGE, PARAMETER, TYPE }
```



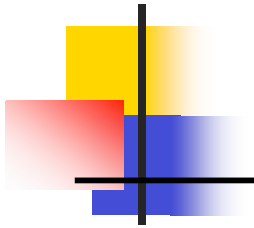
# @Target Element Types

- ANNOTATION_TYPE
  - A meta-annotation
- TYPE
  - Class, Interface, Annotation, or enum
- CONSTRUCTOR, FIELD, METHOD
  - Field also includes enum constants
- LOCAL_VARIABLE
  - Tools like apt can't currently access this
- PARAMETER
  - Method parameter
- PACKAGE
  - Package annotations go in package-info.java



# @Target Usage

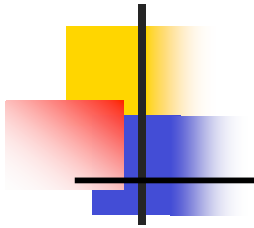
```
import static java.lang.annotation.ElementType.*;

@Target({TYPE, CONSTRUCTOR, PARAMETER})
  public @interface Marker { }

@Marker class Foo {                    // OK
  @Marker public Foo() { }             // OK
  @Marker int x;                       //   No
  @Marker public m(                    //   No
        @Marker int param ) {          // OK
    @Marker int variable;              //   No
  }
}
```



# @Retention Meta-Annotation

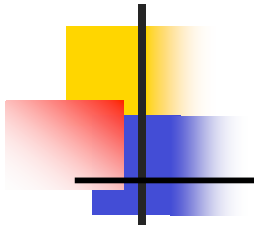
- Where is the annotation retained
  - Where can the annotation be accessed and used
  - Default is CLASS

```
public @interface Retention {
   RetentionPolicy value();
}
public enum RetentionPolicy {
   SOURCE, CLASS, RUNTIME };
```



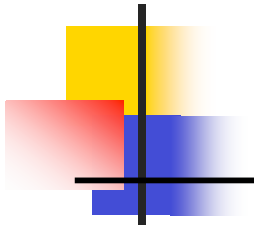
# Retention Policies

- SOURCE
  - Discarded by the compiler
- CLASS
  - Retained by compiler to class file, may be discarded by VM
- RUNTIME
  - Retained in class file and by VM
  - Can be accessed with reflection



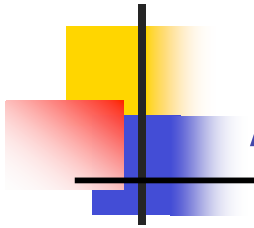
# @Retention Usage

```
@import java.lang.annotation.RetentionPolicy;

@Retention( RetentionPolicy.RUNTIME )
  public @interface Marker{ }
```



# Code Break

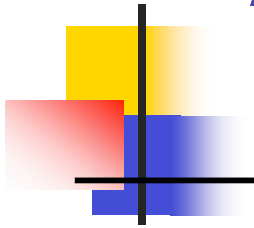
- Example Annotations and Meta-Annotations
  - @FixMe
  - @ToDo
  - Simple code using @FixMe and @ToDo



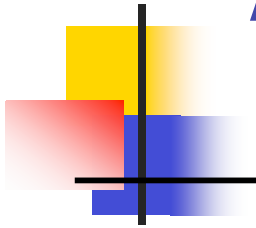
# Accessing Annotations

- Annotations are not much use unless you can access them and use them
- Where can we access Annotations?
  - Potentially any phase of
    - Develop
    - Build
    - Test
    - Deploy
    - Run

# Accessing Annotations Develop … Build



- IDE or other Development tools
  - Use annotations to mark special things like design patterns
    - @Singleton, @Decorator, @Bean …
  - Tools could help you get it right, *etc.*
- Compiler
  - javac recognizes java.lang.* annotations
    - @Deprecated, @Override, (@SuppressWarnings)

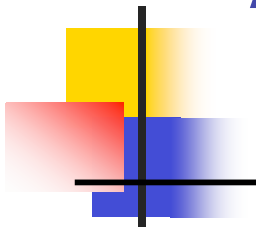


# Accessing Annotations … Build

- Build Tools
  - Access source-level annotations
  - Using apt or doclet
  - Usually generates code or other support files from annotations
  - Examples:
    - Generate BeanInfo classes from annotated Beans
    - Generate deployment descriptor from annotated EJB

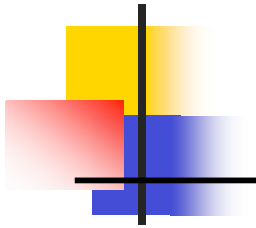
# Accessing Annotations … Build … Deploy

- Post-Processing Tools
  - Access class-level annotations by scanning class files*?*
  - Or runtime annotations with reflection
  - Similar function to Build tools
- Deploy-time Processing
  - Container responding to runtime annotations
  - ClassLoader accessing class-level annotations*?*
  - Dynamic class generation, plug-ins, *etc.*
  - Examples:
    - Dynamic generation of EJB descriptor information from annotated EJB
    - ClassLoader generates BeanInfo class when annotated Bean is loaded



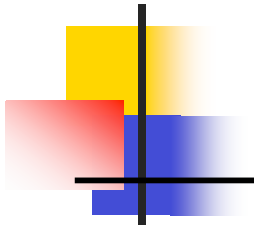
# Accessing Annotations … Run

- Runtime Processing
  - Factory or Proxy adds behavior based on annotations
  - Framework code looks for annotations
  - Examples:
    - Annotations to mark unit test methods



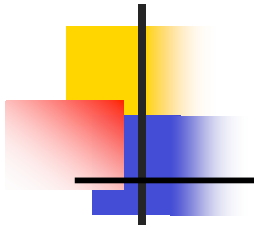
# Reflection and Annotations

- Annotation needs @Retention( RUNTIME )
- Class, Constructor, Field, Method, Package:
  - `boolean isAnnotationPresent( Class<? extends Annotation> a)`
  - `<T extends Annotation> T getAnnotation( Class<T> a )`
  - `Annotation[] getAnnotations()`
  - `Annotation[] getDeclaredAnnotations()`
    - Ignores inherited annotations
  - `Method.getParameterAnnotations()`



# Code Break

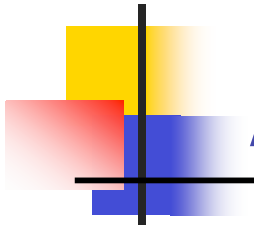
- Example using reflection
  - FixMeReporter
  - NoBrokenCodeClassLoader



# Processing Source Annotations

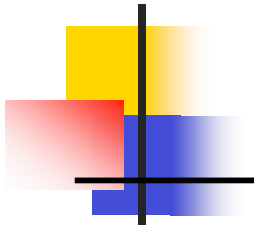
- apt
  - Annotation Processing Tool
  - JDK tool for processing source annotations
  - Cleaner model of source and types than doclet
  - Supports recursive processing of generated files
    - Can generate code containing annotations
  - Multiple processors (*vs.* single doclet)

  - http://java.sun.com/j2se/1.5.0/docs/guide/apt/index.html



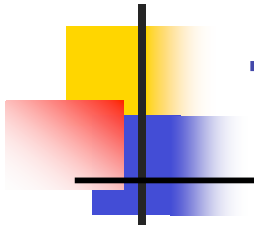
# Annotations *vs.* Doclet

- Better, more up-to-date model of Java type system
  - Including Generics
- Annotation processors run based on all annotations present in code
  - Rather than single "-doclet" switch
  - Potentially multiple processors
  - Recursive
    - Generated code can contain annotations
  - Compile generated code (javac)
- Limitations
  - No processing of local variable annotations



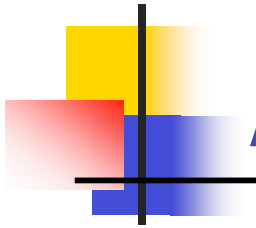
# Using apt

- Write an AnnotationProcessorFactory
  - That creates an AnnotationProcessor
- Include tools.jar in apt's classpath
  - apt -classpath …tools.jar…
- Invoke apt
  - Much like javac
  - Will compile any code generated by annotation processors



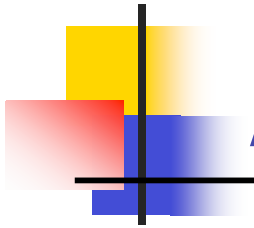
# The mirror packages

- com.sun.mirror.apt
  - Interface with the apt tool
- com.sun.mirror.declaration
  - Models representing declarations in the source
    - Field, Class, Method, *etc.*
- com.sun.mirror.types
  - Models representing types in the source
  - Usages (or invocations) of the declarations
- com.sun.mirror.util
  - Utilities for processing declarations and types



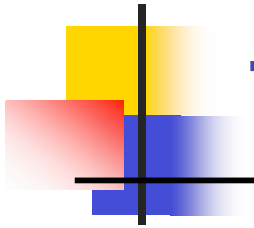
# AnnotationProcessorFactory

- public Collection<String> supportedAnnotationTypes();
  - Return annotations supported by this Factory
  - Can be “com.foo.*” or “*”
- public Collection<String> supportedOptions();
  - Return options recognized by this Factory
    - apt -Afoo -Abar=3 …
- public AnnotationProcessor getProcessorFor(
  Set<AnnotationTypeDeclaration> atds,
  AnnotationProcessorEnvironment env );
  - Return an AnnotationProcessor for the types and environment described by the arguments



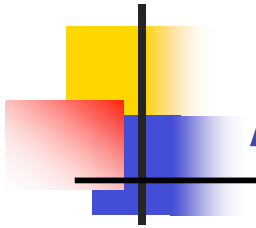
# AnnotationProcessor

- public void process();
  - Do something directly in this method
  - Or use Visitors from com.sun.mirror.util.*
  - Will usually use the environment from the AnnotationProcessorFactory
    - Iterate through Types being processed



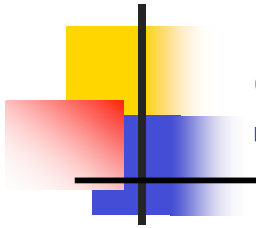
# The apt Tool

- apt [options] sourcefiles… [@files]
  - sourcefiles
    - File(s) to process
  - @files
    - File(s) listing source files or other options



# Apt Options

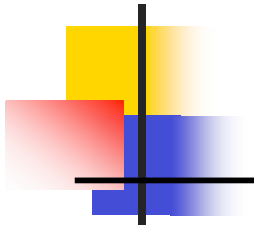
- -classpath, -sourcepath, -d
  - Shared by apt and javac
- Other javac options
  - Passed to javac
- -s *dir*
  - Where processor-generated source files go
- -nocompile
  - Do not compile generated source
- -print
  - Do no processing or compilation, just print specified types
- -A[*key*[=*val*]]
  - Options passed to annotation processors
- -factorypath *path*
  - Where to find annotation processor factories
  - If used, classpath is not searched
- -factory *classname*
  - Annotation processor factory to use
  - Bypasses default discovery process



# Specifying Processors to Run

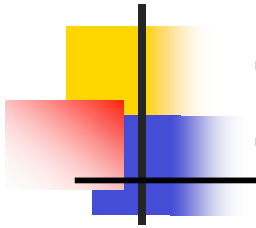
- Single processor
  - apt … -factory foo.MyAPF foo/bar/*.java
- Multiple or automatic processing
  - Annotation processor factories *in a jar*
  - Jar also contains

    META-INF/services/com.sun.mirror.apt.AnnotationProcessorFactory
    - Text file containing classnames of processor factories
      - ✓ One per line
    - Jar in apt's classpath or factorypath



# Code Break

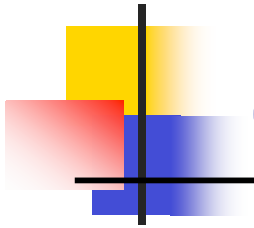
- Example using apt
  - CodeReportAPF
  - LoggerAPF



# Interfaces *vs.* Annotations

- Annotations can replace Interfaces in some cases
  - But is this a good idea?
- Interface indicates desired *capability*
  - Interfaces are a language-based mechanism
  - Strongly typed
- Annotation indicates desired *attributes*
  - Annotations are a tool- or library-based mechanism
    - Source, Class, or Runtime
  - Not mandatory

# Marker Interfaces or Annotations



- Can sometimes be annotations
  - Serializable *vs.* @Serializable
    - No methods, just a statement of behavior
    - However: how do you write this using annotations?
      - ✓ void saveToFile( Serializable object );
  - Bean *vs.* @Bean
    - There is no real interface-level semantics for a bean, just following a pattern
    - An annotation could be useful
      - ✓ Code can do special things for something that declares itself to be a @Bean
      - ✓ Build could generate BeanInfo based on annotations

# More Interfaces or Annotations

- Interface

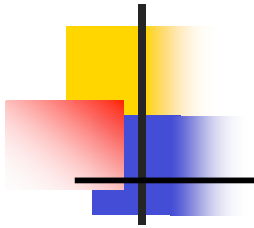
```
class MyAP implements AnnotationProcessorFactory {
  Collection<String> supportedAnnotationTypes() {
    return Arrays.asList("FixMe","ToDo","Review");
  } … }
```

  - Interface forces implementation to provide the method

- Annotation

```
@SupportedAnnotationTypes({"FixMe","ToDo","Review"})
  class MyAP2 implements AnnotationProcessorFactory
  {…}
```

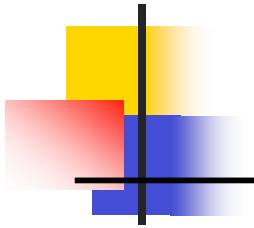
  - No way to enforce that MyAP2 has the annotation



# Limitations

- No inheritance of annotations
  - ✗ `@interface FixMe extends ToDo { … }`
  - ➢ Use Meta-Annotations and apt to "inject" behavior ???

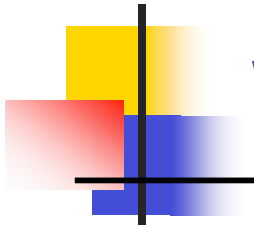
```
@Target({ANNOTATION_TYPE})
  @interface Extends { String value() }

@Extends( "ToDo" )
  @interface FixMe { … }
```



# Limitations

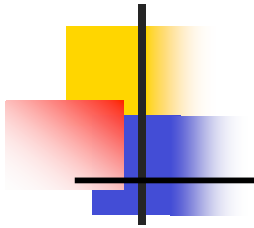
- No way to add simple behavior
  - ✗
```
@interface FixMe { …
    public String toString() { … }
```
  - ➢ Write such behavior in associated helper class

```
Helper.getInstance( fixMe ).toString();
```



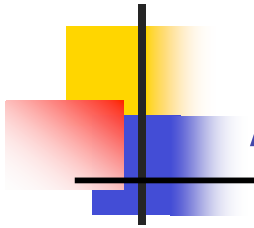
# What's Missing?

- More standard annotations - watch JSR-250
  - For J2SE things
    - Beans, GUI elements, *etc.*
  - For J2EE components
    - Are coming, but vendors will likely roll-their-own until JSRs jell
- Apt integrated into javac
  - More automatic, less dependence on build sequence
- Apt and mirror packages are in com.sun.*
  - Not java.*
- Runtime overrides of annotations
  - Why recompile to change an attribute?
  - Beehive



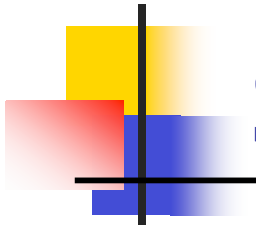
# Standard Annotations

- JSRs for J2EE, EJB3, WebApp, Web Services, *etc.*
  - Generate all those required, boring, repetitive interfaces and descriptors from a single implementation class
    - Remote, Home, Local, *etc.* EJB interfaces, ejb-jar.xml
    - Taglib TLD descriptor
    - JAX-RPC interfaces, descriptors
    - Web Services - JSR-181
  - Already tools to do this (EjbGen / XDoclet / *etc.*)
    - Annotations move the tagging from documentation to source
      - ✓ More formal
    - Annotations extend processing ability to the container
      - ✓ Deploy-time *vs.* Build-time



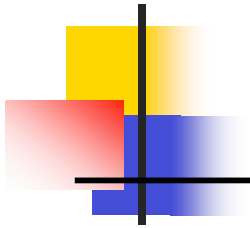
# Annotation Users

- Cedric's TestNG
  - Mark test methods using Annotations, not name patterns
  - Annotations to inject properties, *etc.*
- Beehive
  - Annotation-driven programming model
    - Controls (Annotated JavaBeans)
    - Web Services (JSR-181)
    - NetUI: Struts, XMLBeans, Controls, JSF
- More…



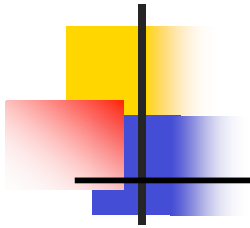
# Summary

- Annotations are modifiers
- Annotations do not affect class semantics
  - Need build- or run-time tools, libraries for this
- Cool Things
  - Annotations at runtime
  - @Override
  - apt
- Try to find and use standard annotations
  - Rather than always rolling your own
- Experiment and have fun



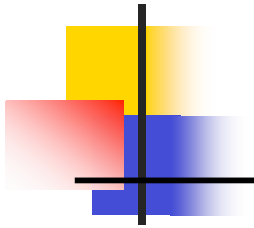
# References

- Sun's Annotation overview
  - http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html
- APT docs
  - http://java.sun.com/j2se/1.5.0/docs/guide/apt/index.html
- JSR-250: Common Annotations for Java
  - http://jcp.org/en/jsr/detail?id=250



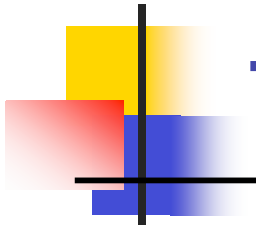
# More References

- Annotations in Tiger, Brett McLaughlin
  - http://www-106.ibm.com/developerworks/library/j-annotate1
- Aspect-Oriented Annotations, Bill Burke
  - http://www.onjava.com/pub/a/onjava/2004/08/25/aoa.html
- Beehive
  - http://incubator.apache.org/projects/beehive.html
- TestNG
  - http://beust.com/testng/



# Other Related Sessions

- Mark Reinhold
  - The Rest of Tiger
    - Other J2SE 5.0 features
- Donald Smith
  - Caging the Tiger
    - Persistence, EJB3



# The End

- Please fill out the evaluations
- Example code available
  - On the conference CDROM
  - http://boulderites.bea.com/~landers
    - References there, too

dave.landers@bea.com