Java 1.6 Annotations

ACCU 2007

Tony Barrett-Powell
tony.barrett-powell@oracle.com

Introduction

- An exploration of Java annotations
  - Concepts – what are annotations?
  - Java 1.6 – changes related to annotations
  - Using annotations – a worked example
  - Some other approaches
- Me
  - A long time C++/Java programmer
  - At Oracle developing BI products
  - tony.barrett-powell@oracle.com

Introduction

- Before we start
  - Java?
  - Java 1.5/1.6?
  - Annotations?

What are annotations?

- Java <1.5
  - Comments
    - Invariants
    - Pre and Post Conditions, for example not null
    - Doxygen
  - Existing metadata
    - transient
    - Marker interfaces, for example Serializable
    - @deprecated
    - final

What are annotations?

- Java 5 (JDK 1.5)
  - JSR 175 A Metadata Facility for the Java Programming Langauge
    - Adds annotations
      - Are defined by annotation types
      - Applied to declarations
    - Provides a small number of built-in annotations
    - Provides a processing tool - APT

What are annotations?

- Marker Annotations in use:

```
void accelerate(@NotNull Force amount) { ... }
private @Name String owner;
@Value class Address { ... }
```

- Built-in Annotations in use:

```
@Deprecated void accelerate(int amount);
@Override  boolean equals(Object other);
@SuppressWarnings(...)
```

What are annotations?

- Parameterised Annotations in use:

  void accelerate(@Range(min=0, max=50) Force amount) { ... }

  private @Name(length=20) String owner;

  @Help(url="speedcalculations.html") float calculateSpeed();

- Single parameters can use default member value:

  private @Name(20) String owner;

  @Help("speedcalculations.htm") float calculateSpeed();

What are annotations?
Annotation Types

- Declared using @ (a)nnotation (t)ype
  - Much like interfaces
- Restricted return types on methods:
  - Built-in types, String, Class, Enum or Annotation
  - Arrays of the above

What are annotations?
Annotation Types

- public @interface Name {
  int length();
  }
- public @interface Help {
  String url();
  String proxy() default “”; // default value = “”
  }
- Usage:

  @Name(length=20) ...

  @Help(url=”somehelpfile.html”); // default value used for proxy

What are annotations?
value member

- public @interface Name {

  int value();

  }
- Usage:

  @Name(20) ...

What are annotations?
Nested Annotations

- public @interface Postcode {

```
    int value();
}
public @interface Address {
    int houseNumber();
    Postcode postcode();
}
```

- Usage:

```
@Address(houseNumber=50, postcode=@Postcode(”BS1 2NG”)) ...
```

What are annotations?
Package Info

- package-info.java

```
/**
  Package summary.
  More info about the package.
*/

// package annotations go here
@SomeAnnotation

package accu2007.example.package;
```

- Replaces package.html

What are annotations?
Meta Annotations

- @Target
  - Where the annotation can be used
  - ANNOTATION_TYPE, PACKAGE, TYPE, METHOD, CONSTRUCTOR, FIELD, PARAMETER, LOCAL_VARIABLE
- @Inherited
  - Whether an annotation on a class element is inherited by a subclass.

What are annotations?
Meta Annotations

- @Retention
  - How long the annotation information is retained
  - SOURCE, CLASS, RUNTIME
  - Default is SOURCE
- @Documented
  - Whether the usage of an annotation appears in the class javadoc

What are annotations? What are they for?

- Main uses:
  - Frameworks, JUnit, EJB3, etc
  - Runtime information
  - Static analysis
    - For example, look for unchecked null access
    - Integrated into an IDE
- Other uses:
  - Generative programming?
  - Standards enforcement?

What are annotations? Limitations

- An annotation:
  - Cannot change the way a program executes
  - Cannot inherit from other annotations
- Annotation processors:
  - Cannot modify existing files

Changes in 1.6

- APT replaced by Annotation Processor API
  - Annotations processed by javac
  - APT deprecated
- A new syntax tree API
- New built-in annotations added
  - Annotation Processing
  - Web services, etc.

Changes in 1.6
Annotation Processing API

- Used to provide a standard API for annotation processors to be used by a processing tool, nominally the compiler
- Annotation processing is performed in “rounds”
  - Initially the processing tool determines which annotations the processor supports
  - A processor will be asked to process it's annotations (or a subset) and the classes available from a previous round
    - The processing tool provides the Filer and Messager
    - A single instance of the processor is created and this instance is used for all round processing
    - An annotation can be “claimed” by the processor
  - Once a processor is used for a round it will be used for all subsequent rounds

Changes in 1.6
Annotation Processing API

- In package javax.annotation.processing
- Main class is:
  - AbstractProcessor provides the template for bespoke processors.
- And main interfaces
  - RoundEnvironment provides information about a round of processing
  - ProcessingEnvironment provides access to the processing environment, to write to files, or provide messages
  - Filer provides the API for writing files
  - Messager provides the API for writing processing tool messages

Changes in 1.6
Annotation Processor

- Example processor:

```
@SupportedAnnotationTypes("*")
@SupportedSourceVersion(SourceVersion.RELEASE_6)

public class NoOpProcessor extends AbstractProcessor {

  public boolean process(Set<? extends TypeElement> annotations,
                         RoundEnvironment roundEnv) {
    processingEnv.getMessager().printMessage(NOTE, "Processor running.");
    return false;  // not claiming the annotation
  }
}
```

Changes in 1.6
Annotation Processor

- Running the example processor:

  javac NoOpProcessor.java
  javac -processor NoOpProcessor Foo.java

  The output will be:

  Note: Processor running.

- The message will be produced for each round of processing.

Changes in 1.6
Writing new classes

- The annotation

```
public @interface WriteClass {}
```

- The source

```
@WriteClass public class Bar {}
```

- The processor shell

```
@SupportedAnnotationTypes("WriteClass")
@SupportedSourceVersion(RELEASE_6)
public class FileWriterProc extends AbstractProcessor {
  public boolean process(Set<? extends TypeElement> annotations,
                         RoundEnvironment roundEnv) {
    // next slide
  }
}
```

Changes in 1.6
Writing new classes

- The process() contents

```
Filer filer = processingEnv.getFiler();
Messager messager = processingEnv.getMessager();
Elements elementUtils = processingEnv.getElementUtils();
if (!roundEnv.processingOver()) {
  TypeElement WriteFileEement = elementUtils.getTypeElement("WriteClass");
  if (!roundEnv.getElementsAnnotatedWith(WriteFileElement).isEmpty()) {
    try {
      PrintWriter pw = new PrintWriter(filer.createSourceFile("Foo"));
      pw.println("public class Foo {}");
      pw.close();
    } catch (IOException ex) { // do something useful with the exception }
  }
}
return true; // annotation claimed, this is the only processor for the annotation
}
```

Changes in 1.6
Writing new classes

- Running the processor:

```
> javac -XprintRounds -processor FooWriterProc Bar.java
Round 1:                          <-- File written in this round
  input files: {Bar}
  annotations: [WriteFile]
  last round: false
Round 2:                          <-- File compiled in this round
  input files: {Foo}
  annotations: []
  last round: false
Round 3:
  input files: {}
  annotations: []
  last round: true
```

Changes in 1.6
The java model packages

- javax.lang.model.*
  - Provides a model of the java programming language
  - Used by annotation processing but not limited to this
    - Also used by the compiler API (JSR 199)
  - Is a representation of the source
    - Not bytecode, though a decompiled version could be used.
    - Allows use of visitors to simplify navigation of tree

Changes in 1.6
The java model packages

- javax.lang.model.*
  - Follows a mirror based design (http://bracha.org/mirrors.pdf)
    - Separation of types from reflection meta-level information
    - This is distinct from the reflection model in java.lang.*
  - Models the difference between elements and types: i.e. static element java.util.Set vs types java.util.Set, java.util.Set<T> and java.util.Set<String>
    - Separate packages for Elements and Types

Changes in 1.6
Example static analysis

- Determine if a class that overrides equals() also overrides hashCode()
- Example class:

```
public class Example {
  public Example(String value) { ... }

  @Override
  public boolean equals(Object o) { ... }

  private final String _value;
}
```

Changes in 1.6
Example static analysis

- Processor shell:

```
@SupportedAnnotationTypes("*")
@SupportedSourceVersion(RELEASE_6)
public class ExampleProcessor extends AbstractProcessor {
  public boolean process(Set<? extends TypeElement> annotations,
                         RoundEnvironment roundEnv) {
    Messager messager = processingEnv.getMessager()
    for (Element e : roundEnv.getElementsAnnotatedWith(Override.class)) {
      if (e.getKind() != ElementKind.METHOD) { continue; }
       // next slide
    }
    return false;
  }
}
```

Changes in 1.6
Example static analysis

- Process content:

```
TypeElement clazz = (TypeElement) e.getEnclosingElement();
List<? extends Element> elements = clazz.getEnclosedElements();
boolean hasHashCode = false;
boolean hasEquals = false;
for (Element element : elements) {
  if (element.getKind() == ElementKind.METHOD) {
    ExecutableElement method = (ExecutableElement)element;
    if (method.getSimpleName().contentEquals("equals"))
        hasEquals = true;
    if (method.getSimpleName().contentEquals("hashCode"))
       hasHashCode = true;
  }
} ...
```

Changes in 1.6
Example static analysis

- Process content (continued):

```
if ((hasEquals == true) && (hasHashCode == false)) {
  messager.printMessage(
      Diagnostic.Kind.WARNING,
      "Implements equals() but not hashCode()", clazz);
}
return false;  // do not claim this annotation
}
```

- Compiler output:

```
Example.java:3: warning: Implements equals() but not hashCode()
public class Example {
       ^
```

Changes in 1.6
Using Visitors

- The Processor API allows the use of visitors to perform processing on specific elements
- These visitors are defined in package javax.lang.model.util
- Process method becomes:

```
public boolean process(Set<? extends TypeElement> annotations,
                       RoundEnvironment roundEnv) {
   Messager messager = processingEnv.getMessager();
   for (Element element :
        roundEnv.getElementsAnnotatedWith(Override.class)) {
     OverrideVisitor visitor = new OverrideVisitor();
     element.accept(visitor, messager);
   }
   return false;  // do not claim this annotation
}
```

Changes in 1.6
Using Visitors

- An implementation of the javax.lang.model.util visitor is required:

```
public class OverrideVisitor extends SimpleElementVisitor6<Void, Messager>
{
  public Void visitExecutable(ExecutableElement element,
                              Messager messager)   {
    if (element.getKind() == ElementKind.METHOD) {
      ... // as previous example
    }
    return defaultAction(element, messager);
  }
}
```

Changes in 1.6
Using Visitors

- Looking at the API of SimpleElementVisitor6:

```
@SupportedSourceVersion(RELEASE_6)
public class SimpleElementVisitor6<R, P> extends
                                        AbstractElementVisitor6<R, P> {
    protected final R DEFAULT_VALUE;
    protected SimpleElementVisitor6();
    protected SimpleElementVisitor6(R defaultValue);
    protected R defaultAction(Element e, P p);
    public R visitPackage(PackageElement e, P p);
    public R visitType(TypeElement e, P p);
    public R visitVariable(VariableElement e, P p);
    public R visitExecutable(ExecutableElement e, P p);
    public R visitTypeParameter(TypeParameterElement e, P p);
}
```

Worked Example
A member constraint

- This annotation (and the processing) provides a constraint on a class string member or method
  - It is contrived to show annotation usage and processing (in most forms)
  - It is simpler just to implement a standard OO version
  - The constraint will be in the form of a regex, and is a postcode, but is by no means correct (lack of room for the full regex for a postcode)
  - On with the example

Worked Example
The annotation

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

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.METHOD})
public @interface StringConstraint {
    // a regex to constrain a string value
    String value() default ".?";
}
```

Worked Example
The Address class

```
public final class Address
{
  public Address(String houseNumber, String postcode) {
     ...
  }

  public String getPostcode() {
    return _postcode;
  }


  ...
  private @StringConstraint("^([A-Z][A-Z][0-9]?[0-9] [0-9][A-Z][A-Z])")
                              String _postcode;
}
```

Worked Example
Java 1.4 version

- _postcode member of Address now:

```
  private final PostCode _postcode;
}
```

- New Postcode class:

```
public PostCode {
  public PostCode(String postCode) {
    String constraint = "^([A-Z][A-Z][0-9]?[0-9] [0-9][A-Z][A-Z])";
    Pattern p = Pattern.compile(constraint);
    Matcher m = p.matcher(postcode);
    if (!m.find()) {
      throw new IllegalArgumentException("Not a postcode");
    ...
  }
  ...
```

Worked Example
Processing

- Without some form of processing the annotation will have no effect
- Processing can be performed at:
  - Runtime
  - Class Loading/Post Compilation
  - Compilation/Pre-compilation
  - This list is the same list as Java 1.5, only the specifics of compilation/pre-compilation have changed
- We'll consider all of these in the following examples

Worked Example
Reflection Processor

- Processing is performed at runtime
- Use reflection to find annotation
  - Use the isAnnotationPresent()/getAnnotation() methods
  - On Class, Method or Field
- Processing could be placed in client or a decorator
  - Though a client is not forced to do this

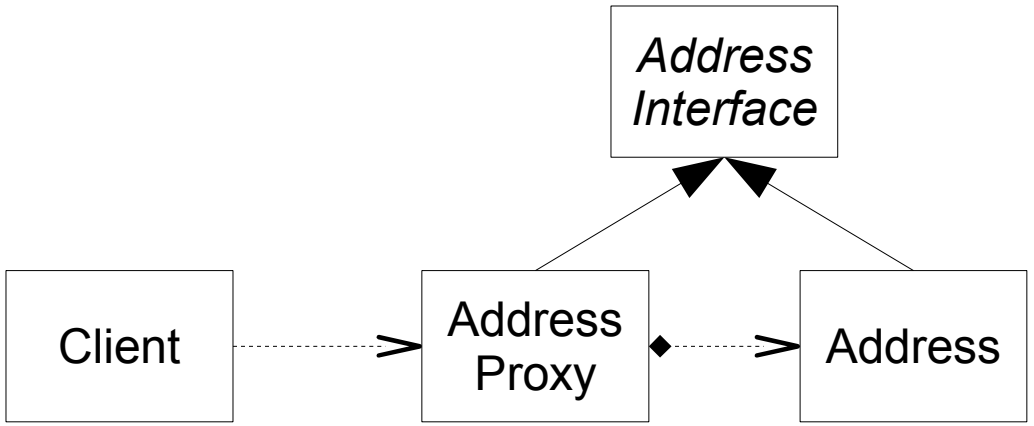
Worked Example
Reflection Processor

- Simple reflection code for annotation:

```
...
for (Field f : Address.class.getDeclaredFields()) {
  f.setAccessible(true);
  if (f.isAnnotationPresent(StringConstraint.class)) {
    StringConstraint sc = f.getAnnotation(StringConstraint.class);
    String constraint = sc.value();
    String postcode = (String)f.get(address);
    Pattern p = Pattern.compile(constraint);
    Matcher m = p.matcher(postcode);
    if (!m.find()) {  // throw an exception }
  }
}
```

Worked Example
Dynamic Proxy Processor

- A runtime processor
- Uses reflection to process annotation
- Wrap the Address class in a dynamic proxy
- Need to introduce an Interface on which to base the dynamic proxy



Worked Example
Dynamic Proxy Processor

- Dynamic Proxy shell:

```
class AddressProxy implements java.lang.reflect.InvocationHandler {
  public static Object newInstance(Object obj) {
    return java.lang.reflect.Proxy.newProxyInstance(
    obj.getClass().getClassLoader(),
    obj.getClass().getInterfaces(),
    new AddressProxy(obj));
  }

  private AddressProxy(Object obj) {
    this.obj = obj;
  }
  // next slide
}
```

Worked Example
Dynamic Proxy Processor

- Dynamic Proxy invoke method:

```
public Object invoke(Object proxy, Method m, Object[] args)
        throws Throwable {
  Object result;
  try {
    result = m.invoke(obj, args);
    Method om = obj.getClass().getMethod(m.getName());
    if (f.isAnnotationPresent(StringConstraint.class)) {
      StringConstraint sc = f.getAnnotation(StringConstraint.class)
      // as in the reflection code
    }
  } catch (InvocationTargetException e) { throw e.getTargetException(); }
  return result;
}
```

Worked Example
Dynamic Proxy Processor

- Usage:

```
AddressInterface address =
    (AddressInterface) AddressProxy.newInstance(
        new Address("138", "BS2 0JA"));
String postcode = address.getPostcode();
```

- Still requires cooperation of the client
- The dynamic proxy could be generated by an annotation processor

Worked Example
Byte Code Modification

- Processing can be performed on the bytecode
  - Post compilation
    - A separate phase in the build chain
  - Class Loading (a Custom Class Loader)
    - Need to modify a class before it is loaded, a class cannot normally be unloaded
  - Instrumentation API
    - Allows a class modification before main() is executed
- There are a number of libraries to facilitate this:
  - BCEL
  - ASM
  - JAssist

Worked Example
Byte Code Modification

- Accessor method can be modifed from:

```
@StringConstraint("^([A-Z][A-Z][0-9]?[0-9] [0-9][A-Z][A-Z])");
public void setPostCode(String postCode) {
  _postCode = postCode;
}
```

- to:

```
public void setPostCode(String postCode) {
  Pattern p = Pattern.compile("^([A-Z][A-Z][0-9]?[0-9] [0-9][A-Z][A-Z])");
  Matcher m = p.matcher(postCode);
  if (!m.find()) {  // throw an exception }
  _postCode = postCode;
}
```

Worked Example
Byte Code Modification

- Plugging into the Instrumentation API:

```
public class ExampleAgent implements ClassFileTransformer {
  public byte[] transform(ClassLoader loader, String cname, Class class,
                          ProtectionDomain domain, byte[] bytes)
                          throws IllegalClassFormatException {
    try {
      // find annotation usage
      ClassWriter writer = new ClassWriter(false);
      // write modifications
      return writer.toByteArray();
    } catch (IllegalStateException e) { // throw exception }
    return null;
  }
  public static void premain(String arglist, Instrumentation inst) {
       inst.addTransformer(new ExampleAgent());
  }}
```

Worked Example
Byte Code Modification (ASM)

- ASM writing changes to a method:

```
public class ExampleGenerator extends ClassAdapter {
  public MethodReplacer(ClassVisitor cv, String mname, String mdesc) { ... };

  public void visit(int version, int access, String name, String signature, String
                    superName, String[] interfaces) { ... }

  public MethodVisitor visitMethod(int access, String name, String desc, String
                                                  signature, String[] exceptions) {
    // write changes to method here
    return super.visitMethod(access, newName, desc, signature, exceptions);
  }
}
```

Worked Example
Byte Code Modification

- Issues:
  - Complex
  - Maintainability
  - Choosing the right tool/API
- Accessor method is modified with no impact (or expectations) on the client

Worked Example Compilation

- Use Annotation Processor to generate extra code:
  - Cannot modify the existing code
  - Can generate new base class or sub class
    - Cooperation required from the class using the annotation
    - Reference to class that doesn't yet exist
      - All sorted out at the end of annotation processing

Worked Example
Compilation Processor

- Modified Address class (to use generated base class):

```
public final class Address extends AddressBase
{
  public Address(String houseNumber, String postcode) { ... }

  public void setPostcode(String postcode) {
    validateConstraint_postcode(postcode);
    _postcode = postcode;
  }

  ...
  private final @StringConstraint("^([A-Z][A-Z][0-9]?[0-9] [0-9][A-Z][A-Z])")
                              String _postcode;
}
```

Worked Example
Compilation Processor

```
public boolean process(Set<? extends TypeElement> annotations,
                       RoundEnvironment roundEnv) {
  for (Element e : roundEnv.getElementsAnnotatedWith(StringConstraint.class)) {
    if (e.getKind() != ElementKind.FIELD) { continue; }
    String name = capitalize(e.getSimpleName().toString());
    TypeElement clazz = (TypeElement) e.getEnclosingElement();
    try {
      Filer filer = processingEnv.getFiler()
      JavaFileObject f = filer.createSourceFile(clazz.getQualifiedName() + "Base");
      Writer w = f.openWriter();
      // write abstract class
    } catch (IOException x) { };
  }
  return true;
}
```

Worked Example
Compilation Processor

- Generated code

```
...
public abstract class AddressBase {
    protected AddressBase() {}

    protected final String validateConstraint_postcode(java.lang.String value) {
      Pattern p = Pattern.compile("^([A-Z][A-Z][0-9]?[0-9] [0-9][A-Z][A-Z])");
      Matcher m = p.matcher(value);
      if (!m.find()) { throw new IllegalArgumentException("Constraint failed"); }
      return value;
    }
}
```

Worked Example
Compilation Processor

- Compiler output:

```
Address.java:3: cannot find symbol
symbol: class AddressBase
public class Address extends AddressBase
                             ^
```

- This isn't an error, the file isn't generated when this message is created
- The next round will compile both Address and the generated AddressBase
- The annotation could allow the name of the base class and the name of the method to be specified

Other Approaches

- The annotation processor is the most straight forward (annotation processing) mechanism to use with the least drawbacks
- Some tools are available which allow changes to the source files using a JSR 269 like approach:
  - Spoon (http://spoon.gforge.inria.fr/)
  - Jackpot (http://jackpot.netbeans.org/)
- The Compiler API could be used (JSR 199) to change the byte code being written from the compiler

Summary

- We have covered:
  - Annotations
    - Syntax
    - Intent
  - The new 1.6 annotation features
  - Processing annotations
    - Runtime, Load and Compile time options
    - The limitations of “standard” approaches
  - Some tools to considers which overcome these limitations

References

- JSR 269: Pluggable Annotation Processing API (http://jcp.org/en/jsr/detail?id=269)
- JSR 199: Java Compiler API (http://jcp.org/en/jsr/detail?id=199)
- JSR 175: A Metadata Facility for the Java Programming Language (http://jcp.org/en/jsr/detail?id=175)
- ASM (http://asm.objectweb.org/)
- BCEL (http://jakarta.apache.org/bcel/)
- JAssist (http://jassistant.sourceforge.net/)
- Mirror-based design (http://bracha.org/mirrors.pdf)
- Spoon (http://spoon.gforge.inria.fr/)
- Jackpot (http://jackpot.netbeans.org/)
- Benedict Heal – What are Java annotations for? (http://www.benedictheal.com/java/index.htm)

A final puzzle

- A couple of years ago Benedict Heal presented a session on annotations (when they were added in Java 5).
- He closed with a “what happens now?” slide, here is the code he presented:

```
@persistent( table="t1")
    @vars(   @var(name="age",
                    type="Integer", range=("1-99")),
                @var(name="address", type="MyString",
                    checkerClass="com.MyChecker"))
    @methods({
          @method(name="m1", body="if .. {}"),
          @method(name="m2", body="while..") }

class Thing {  }
```

How would you write the annotation processor(s) for this?

Java 1.6 Annotations

ACCU 2007

Tony Barrett-Powell
tony.barrett-powell@oracle.com

Welcome to this set of slides. The notes should cover the basic points, beyond the slides, I will discuss during the session. The intent is to provide a fuller resource than is possible than just the slides alone.

Introduction

- An exploration of Java annotations
  - Concepts – what are annotations?
  - Java 1.6 – changes related to annotations
  - Using annotations – a worked example
  - Some other approaches
- Me
  - A long time C++/Java programmer
  - At Oracle developing BI products
  - tony.barrett-powell@oracle.com

This session looks at java annotations, starting with introductory material and then following with a look at the new Java APIs and finally how to process annotations.

Benedict Heal presented a similar session at ACCU2005 when annotations were introduced into Java 5, though covering more of the reason for the addition of annotations. I cover a reasonable amount of the same ground to provide context to those with minimal exposure to annotation processing.

I have been writing software since 1990 mainly in C/C++ and Java, with a few diversions along the way. I currently work for Oracle focused on the Business Intelligence product suite.

Introduction

- Before we start
  - Java?
  - Java 1.5/1.6?
  - Annotations?

This session contains a decent number of code examples, therefore it helps if you can read the java. I try to highlight the interesting parts of the code examples which reflect the current subject, but I do not want to dwell on syntax.

What are annotations?

- Java <1.5
  - Comments
    - Invariants
    - Pre and Post Conditions, for example not null
    - Doxygen
  - Existing metadata
    - transient
    - Marker interfaces, for example Serializable
    - @deprecated
    - final

Annotations are deemed to be metadata, data about data. Before the introduction of annotations into Java with version 1.5 there were already lots of metadata present in the source code, though some of this was usable by tools or the compiler, but most of this was really for the human audience.

What are annotations?

- Java 5 (JDK 1.5)
  - JSR 175 A Metadata Facility for the Java Programming Langauge
    - Adds annotations
      - Are defined by annotation types
      - Applied to declarations
    - Provides a small number of built-in annotations
    - Provides a processing tool - APT

Java 5 added the ability to define annotations and to process them before compile time. Thus it was possible to define metadata directly in the source which would allow this information to be process in some way.

The standard annotation processor was APT (Annotation Processing Tool) and this was used before the compiler in the tool chain.

A small number of built in annotations were defined which were enforce by the compiler rather than the APT.

What are annotations?

- Marker Annotations in use:

```
void accelerate(@NotNull Force amount) { ... }
private @Name String owner;
@Value class Address { ... }
```

- Built-in Annotations in use:

```
@Deprecated void accelerate(int amount);
@Override  boolean equals(Object other);
@SuppressWarnings(...)
```


Marker annotations are much like existing interface such as Serializable they do not define any semantics beyond their presence. They do not define any additional information. An example of such an annotation is NotNull, its presence is sufficient to define that the element cannot be null.

The built-in annotations understood by the compiler allow useful information to be define, or allow the author to controller the warnings generated by the compiler (useful for generics).

What are annotations?

- Parameterised Annotations in use:

```
void accelerate(@Range(min=0, max=50) Force amount) { ... }
private @Name(length=20) String owner;
@Help(url="speedcalculations.html") float calculateSpeed();
```

- Single parameters can use default member value:

```
private @Name(20) String owner;
@Help("speedcalculations.htm") float calculateSpeed();
```


Parametrized annotations allow addition information to be specified in the declaration to control the processing of the annotation. For example a range annotation could accept the low and high limits for the range and this could be enforced by the processing.

If an annotation accepts a single parameter the default value parameter can be used and so does not need to be named in the annotation usage.

What are annotations?
Annotation Types

- Declared using @ (a)nnotation (t)ype
  - Much like interfaces
- Restricted return types on methods:
  - Built-in types, String, Class, Enum or Annotation
  - Arrays of the above

Annotations are defined like interfaces in source code with an additional @ symbol (@ -> at -> Annotation Type).

The annotation can have methods, but these are restricted in the possible return types, to built-in types, String, Class, Enum or another Annotation (or arrays of these).

What are annotations?
Annotation Types

- public @interface Name {
  int length();
  }
- public @interface Help {
  String url();
  String proxy() default “”; // default value = “”
  }
- Usage:

  @Name(length=20) ...

  @Help(url=”somehelpfile.html”); // default value used for proxy

Here are a few examples of simple annotation types.

What are annotations?
value member

- public @interface Name {

  int value();

  }
- Usage:

  @Name(20) ...

The default value member can be used in the annotation definition and the type of this variable is determined by the definition.

The usage of the annotation allows the name of the value being defined to be skipped.

What are annotations?
Nested Annotations

- public @interface Postcode {

```
    int value();
}
public @interface Address {
    int houseNumber();
    Postcode postcode();
}
```

- Usage:

```
@Address(houseNumber=50, postcode=@Postcode("BS1 2NG")) ...
```


It is possible to nest an annotation in another, thus allowing reuse of annotations. Though there is no support for inheritance.

What are annotations?
Package Info

- package-info.java

```
/**
  Package summary.
  More info about the package.
*/

// package annotations go here
@SomeAnnotation

package accu2007.example.package;
```

- Replaces package.html

Annotations can be applied to packages, so a mechanism was introduced in Java 1.5 to allow annotations to be attached to the package. This also changes the way that package documentation is defined, it can be defined in package-info.java rather than package.html, even though this is still supported.

What are annotations?
Meta Annotations

- @Target
  - Where the annotation can be used
  - ANNOTATION_TYPE, PACKAGE, TYPE, METHOD, CONSTRUCTOR, FIELD, PARAMETER, LOCAL_VARIABLE
- @Inherited
  - Whether an annotation on a class element is inherited by a subclass.

Meta-annotations are annotations applied to other annotations. There are a number of these provided for control of annotations and their usage. For example the target meta-annotation allows where the annotation can be used to be controlled by the author of the annotation. This is especially important for annotations that have processors.

What are annotations?
Meta Annotations

- @Retention
  - How long the annotation information is retained
  - SOURCE, CLASS, RUNTIME
  - Default is SOURCE
- @Documented
  - Whether the usage of an annotation appears in the class javadoc

Further meta-annotations control how far into the processing chain the annotation information is maintained and whether it appears in the javadoc.

What are annotations?
What are they for?

- Main uses:
  - Frameworks, JUnit, EJB3, etc
  - Runtime information
  - Static analysis
    - For example, look for unchecked null access
    - Integrated into an IDE
- Other uses:
  - Generative programming?
  - Standards enforcement?

Annotations provide the mechanism for many uses. The changes in EJB and JUnit have been driven by the availability of annotations. Further uses that are appearing are the ability to define bespoke static analysis and control of coding standards.

There is now a JSR for static analysis JSR 305, Annotations for Java Software Defect Detection.

See JSR 305: Annotations for Software Defect Detection (http://jcp.org/en/jsr/detail?id=305)

What are annotations?
Limitations

- An annotation:
  - Cannot change the way a program executes
  - Cannot inherit from other annotations
- Annotation processors:
  - Cannot modify existing files

It is worth noting the limitations of annotations. The main rule of annotations is that they cannot change the way a program executes. This doesn't mean that their presence and use by a processor will not change the execution of a program, but the existence of an annotation without some form of processing will not change the execution, just as if it wasn't there.

Another limitation is that the annotation processing provided by the standard tools do not allow existing source files to be modified.

Changes in 1.6

- APT replaced by Annotation Processor API
  - Annotations processed by javac
  - APT deprecated
- A new syntax tree API
- New built-in annotations added
  - Annotation Processing
  - Web services, etc.

In Java 1.6 the existing APT has been replaced with an API for annotation processors. These processors can be directly invoked from the Java compiler. In addition to the new annotation processor API, an additional API to navigate the AST of the Java source has been introduced.

New built-in annotations have been added for the annotation processors as well as other supported technologies, such as web services.

Changes in 1.6
Annotation Processing API

- Used to provide a standard API for annotation processors to be used by a processing tool, nominally the compiler
- Annotation processing is performed in “rounds”
  - Initially the processing tool determines which annotations the processor supports
  - A processor will be asked to process it's annotations (or a subset) and the classes available from a previous round
    - The processing tool provides the Filer and Messager
    - A single instance of the processor is created and this instance is used for all round processing
    - An annotation can be “claimed” by the processor
  - Once a processor is used for a round it will be used for all subsequent rounds

The annotation processing API is the new standard API for all annotation processors. Thus all tools that allow annotation processors to be executed will now use this API to invoke them. The standard tool is the Java compiler which provides 2 ways to define the processors to be executed during the compilation of the source code.

Changes in 1.6
Annotation Processing API

- In package javax.annotation.processing
- Main class is:
  - AbstractProcessor provides the template for bespoke processors.
- And main interfaces
  - RoundEnvironment provides information about a round of processing
  - ProcessingEnvironment provides access to the processing environment, to write to files, or provide messages
  - Filer provides the API for writing files
  - Messager provides the API for writing processing tool messages

19

The main class that author's of annotation processors will use will be the AbstractProcessor, which can be overridden by bespoke processors by providing the method process(...). All other utility methods are provided by the AbstractProcessor.

The RoundEnvironment provides information to the processor about the current round of annotation processing, see later slides for more details.

The ProcessingEnvironment provides access to file and system output APIs to allow the processor to create and modify files and to provide diagnostic messages to the user.

Changes in 1.6
Annotation Processor

- Example processor:

```
@SupportedAnnotationTypes("*")
@SupportedSourceVersion(SourceVersion.RELEASE_6)

public class NoOpProcessor extends AbstractProcessor {

  public boolean process(Set<? extends TypeElement> annotations,
                         RoundEnvironment roundEnv) {
    processingEnv.getMessager().printMessage(NOTE, "Processor running.");
    return false;  // not claiming the annotation
  }
}
```


This simple processor does nothing more than send a message to the user, but it shows how simple a processor can be using the new API.

You can see some useful detail here, the annotation which controls which annotation this processor supports, and which release of the Java language this processor supports.

The process method returns false, this indicates that the annotation is not claimed, that is, the annotation may be processed by other annotations. If the annotation is claimed by a processor, it is not passed to other annotation processors. It is good practice when allowing the ability to process all annotations that these annotations are not claimed.

Changes in 1.6
Annotation Processor

- Running the example processor:

  javac NoOpProcessor.java
  javac -processor NoOpProcessor Foo.java

  The output will be:

  Note: Processor running.

- The message will be produced for each round of processing.

This shows how the annotation processor is used, first it needs to be compiled. Then it can be used by the compiler, the location of the processor can be defined by using the -processor javac argument.

Another approach to providing the compiler where to find processors is to create a META_INF/services/javax.annotation.processing.Processor file. This file lists each processor (one per line) and this needs to be on the classpath, or can be provided using the -processorpath javac agument.

Changes in 1.6
Writing new classes

- The annotation

```
public @interface WriteClass {}
```

- The source

```
@WriteClass public class Bar {}
```

- The processor shell

```
@SupportedAnnotationTypes("WriteClass")
@SupportedSourceVersion(RELEASE_6)
public class FileWriterProc extends AbstractProcessor {
  public boolean process(Set<? extends TypeElement> annotations,
                         RoundEnvironment roundEnv) {
    // next slide
  }
}
```


It is possible to create new classes using an annotation processor, even though it is not possible to modify existing ones using the standard tools.

This example shows a very simple processor that creates a new class Foo. The annotation used is a marker annotation and this is applied to a class Bar.

The annotation processor, supports only the WriteClass annotation. The body of the process method is on the next slide.

Changes in 1.6
Writing new classes

- The process() contents

```
Filer filer = processingEnv.getFiler();
Messager messager = processingEnv.getMessager();
Elements elementUtils = processingEnv.getElementUtils();
if (!roundEnv.processingOver()) {
  TypeElement WriteFileEement = elementUtils.getTypeElement("WriteClass");
  if (!roundEnv.getElementsAnnotatedWith(WriteFileElement).isEmpty()) {
    try {
      PrintWriter pw = new PrintWriter(filer.createSourceFile("Foo"));
      pw.println("public class Foo {}");
      pw.close();
    } catch (IOException ex) { // do something useful with the exception }
  }
 }
 return true; // annotation claimed, this is the only processor for the annotation
}
```


The ability to create files is obtained through the Filer interface and an instance of this is obtained from the ProcessingEnvironment.

The processing environment provides a utility class for handling processing of annotation elements, this is used to obtain the WriteClass annotations available in the processing round.

If a WriteClass is found a PrintWriter created based on a new file obtained from the Filer and the new class is created. This will be compiled in the next round of processing.

The example claims the annotation as this is the only processor for this type and we want to avoid writing the file more than once.

Changes in 1.6
Writing new classes

- Running the processor:

```
> javac -XprintRounds -processor FooWriterProc Bar.java
Round 1:                    <-- File written in this round
  input files: {Bar}
  annotations: [WriteFile]
  last round: false
Round 2:                    <-- File compiled in this round
  input files: {Foo}
  annotations: []
  last round: false
Round 3:
  input files: {}
  annotations: []
  last round: true
```


The usage shows a useful extended compiler switch, printRounds, this shows information about the annotation processing rounds during the compilation process.

You can see in the output that the WriteFile annotation is processed in the first round and that the output file Foo is compiled in the second round.

The information about whether the round is the last round can be obtained from the RoundEnviroment using the processingOver() method.

Changes in 1.6
The java model packages

- javax.lang.model.*
  - Provides a model of the java programming language
  - Used by annotation processing but not limited to this
    - Also used by the compiler API (JSR 199)
  - Is a representation of the source
    - Not bytecode, though a decompiled version could be used.
    - Allows use of visitors to simplify navigation of tree

New in Java 1.6 are the javax.lang.model packages, these provide the APIs that allow exploration of the java source in an appropriate environment, such as the compiler annotation processing.

This API is also used by the new compiler API added to Java 1.6.

The API provides representation of the source, so this is not the same as APIs that allow class files to be processed. The source representation includes all the artefacts in the source code including those that are lost during compilation, such as type parameters on containers.

It is worth noting that this API will change if new constructs are added to the language, thus the API is version specific.

Changes in 1.6
The java model packages

- javax.lang.model.*
  - Follows a mirror based design (http://bracha.org/mirrors.pdf)
    - Separation of types from reflection meta-level information
    - This is distinct from the reflection model in java.lang.*
  - Models the difference between elements and types: i.e. static element java.util.Set vs types java.util.Set, java.util.Set<T> and java.util.Set<String>
    - Separate packages for Elements and Types

The model uses a mirror based design approach, this basically separates the class and objects form the meta-level information for these elements. Thus for each class or object in a system a mirror with the meta-level information will be available This approach improves the separation of concerns in a language. This is different from the existing APIs in the java language as the meta-level information is available from the elements in the type system.

In the javax.lang.model.* packages elements and types are modelled separately, for example the element java.util.Set can be different distinct types in the source, i.e. java.util.Set, java.util.Set<T> and java.util.Set<String>.

Changes in 1.6
Example static analysis

- Determine if a class that overrides equals() also overrides hashCode()
- Example class:

```
public class Example {
  public Example(String value) { ... }

  @Override
  public boolean equals(Object o) { ... }

  private final String _value;
}
```


This example shows how the java model api can be used to perform some simple static analysis. This static analysis is performed using an annotation processor and so allows this static analysis to be performed during compilation.

This example ensures that classes which override equals also override hashCode.

Changes in 1.6
Example static analysis

- Processor shell:

```
@SupportedAnnotationTypes("*")
@SupportedSourceVersion(RELEASE_6)
public class ExampleProcessor extends AbstractProcessor {
  public boolean process(Set<? extends TypeElement> annotations,
                         RoundEnvironment roundEnv) {
    Messager messager = processingEnv.getMessager()
    for (Element e : roundEnv.getElementsAnnotatedWith(Override.class)) {
      if (e.getKind() != ElementKind.METHOD) { continue; }
       // next slide
    }
    return false;
  }
}
```

28

The shell of the processor shows that the annotation processor will operate on all annotations, and because of this does not claim any. A messager is obtained to allow the output of the analysis (if any) to be output to the user.

The processor gets the a method that is annotated with the Override annotation. It would be reasonable to expect equals methods in a class to have this annotation. Though if this is not the case it is possible to add this information using tools such as ASM.

Changes in 1.6
Example static analysis

- Process content:

```
TypeElement clazz = (TypeElement) e.getEnclosingElement();
List<? extends Element> elements = clazz.getEnclosedElements();
boolean hasHashCode = false;
boolean hasEquals = false;
for (Element element : elements) {
  if (element.getKind() == ElementKind.METHOD) {
    ExecutableElement method = (ExecutableElement)element;
    if (method.getSimpleName().contentEquals("equals"))
        hasEquals = true;
    if (method.getSimpleName().contentEquals("hashCode"))
       hasHashCode = true;
  }
} ...
```


For the method found to be annotated with Override, the processor navigates to the enclosing class and then determines whether the class has either an equals method or a hashCode method.

Changes in 1.6
Example static analysis

- Process content (continued):

```
if ((hasEquals == true) && (hasHashCode == false)) {
  messager.printMessage(
      Diagnostic.Kind.WARNING,
      "Implements equals() but not hashCode()", clazz);
}
return false;  // do not claim this annotation
}
```

- Compiler output:

```
Example.java:3: warning: Implements equals() but not hashCode()
public class Example {
       ^
```

30

Now it is possible for the processor to report whether a class defines equals without defining hashCode using the information gathered during iteration over a classes methods. The condition is reported as a warning, but could be an error, if required.

As the processor is processing Override, a general annotation, the processor should not claim it. This would be equally true of a processor that processes all annotations.

The output from the processor gives the programming enough information to fix the problem as the class context is output by the Messager using the clazz parameter.

Changes in 1.6
Using Visitors

- The Processor API allows the use of visitors to perform processing on specific elements
- These visitors are defined in package javax.lang.model.util
- Process method becomes:

```
public boolean process(Set<? extends TypeElement> annotations,
                       RoundEnvironment roundEnv) {
  Messager messager = processingEnv.getMessager();
  for (Element element :
       roundEnv.getElementsAnnotatedWith(Override.class)) {
    OverrideVisitor visitor = new OverrideVisitor();
    element.accept(visitor, messager);
  }
  return false;  // do not claim this annotation
}
```

31

The javax.lang.model api defines a set of visitor implementations that can be used within a processor (as well as the compiler api), which allows for a call-back approach. This can greatly simplify the processor code.

So we can re-write the static analysis example using the visitor approach. The process method is very much simpler as the static processing is now performed by the OverrideVisitor class.

Changes in 1.6
Using Visitors

- An implementation of the javax.lang.model.util visitor is required:

```
public class OverrideVisitor extends SimpleElementVisitor6<Void, Messager>
{
  public Void visitExecutable(ExecutableElement element,
                              Messager messager)   {
    if (element.getKind() == ElementKind.METHOD) {
      ... // as previous example
    }
    return defaultAction(element, messager);
  }
}
```


The SimpleElementVisitor6 is a generic class that provides a default action for all the visit methods. The author of a Visitor provides types for the return and a parameter. The return type can be void and this is defined using the java.lang.Void class, as can the parameter.

The author need only override the visit methods required, and so in this example only overrides the visitExecutable.

Changes in 1.6
Using Visitors

- Looking at the API of SimpleElementVisitor6:

```
@SupportedSourceVersion(RELEASE_6)
public class SimpleElementVisitor6<R, P> extends
                                        AbstractElementVisitor6<R, P> {
    protected final R DEFAULT_VALUE;
    protected SimpleElementVisitor6();
    protected SimpleElementVisitor6(R defaultValue);
    protected R defaultAction(Element e, P p);
    public R visitPackage(PackageElement e, P p);
    public R visitType(TypeElement e, P p);
    public R visitVariable(VariableElement e, P p);
    public R visitExecutable(ExecutableElement e, P p);
    public R visitTypeParameter(TypeParameterElement e, P p);
}
```


The SimpleElementVisitor6 supports Java 6 source code, it is safe to assume that this class will be deprecated in the next version of Java.

The javadoc for this class states that for the next version of Java any additional methods on the ElementVisitor interface to support changes in source code will mean that the method visitUnknown (from AbstractElementVisitor) will be invoked for these new constructs.

Worked Example
A member constraint

- This annotation (and the processing) provides a constraint on a class string member or method
  - It is contrived to show annotation usage and processing (in most forms)
  - It is simpler just to implement a standard OO version
  - The constraint will be in the form of a regex, and is a postcode, but is by no means correct (lack of room for the full regex for a postcode)
  - On with the example

The following section provides a looks at how an annotation can be processed using all (well most) mechanisms available. The example is contrived (to some extent) to allow the consideration of the viability of an approach.

It will be much easier (and correct) to actually write the code using a simple OO approach, but that wouldn't show how we can process annotations, though I will show this solution just for completeness.

Worked Example
The annotation

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

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.METHOD})
public @interface StringConstraint {
   // a regex to constrain a string value
   String value() default ".?";
}
```


This is a simple annotation to define a regex value for a String. The retention is RUNTIME, to allow us to explore run-time processing. The element types are restricted to fields and methods to reduce the scope of the processor handling. The default for the value is “?”, which means “any string”.

Worked Example
The Address class

```
public final class Address
{
  public Address(String houseNumber, String postcode) {
     ...
  }

  public String getPostcode() {
    return _postcode;
  }

  ...
  private @StringConstraint("^([A-Z][A-Z][0-9]?[0-9] [0-9][A-Z][A-Z])")
                              String _postcode;
}
```


Here is the outline of a class which uses the annotation, the Address class. This uses the annotation to define the format of the postcode field. Of course I have missed out the rest of the code which determines the rest of the information, such as the street name, etc from the house number and post code.

It is worth noting that the format is very simplistic definition of the UK postcode, and something more like:

“^([A-PR-UWYZ0-9][A-HK-Y0-9][AEHMNPRTVXY0-9]?[ABEHMNPRVWXY0-9]? {1,2}[0-9][ABD-HJLN-UW-Z]{2}|GIR 0AA)$"

is required, but that wouldn't fit on the page :)

I can almost hear you think that why not just have a Postcode class that handles this?

Worked Example
Java 1.4 version

- _postcode member of Address now:

```
  private final PostCode _postcode;
}
```

- New Postcode class:

```
public PostCode {
  public PostCode(String postCode) {
    String constraint = "^([A-Z][A-Z][0-9]?[0-9] [0-9][A-Z][A-Z])";
    Pattern p = Pattern.compile(constraint);
    Matcher m = p.matcher(postcode);
    if (!m.find()) {
      throw new IllegalArgumentException("Not a postcode");
    ...
  }
  ...
```


It is immediately apparent that a standard OO solution is best, that adding a Postcode type will provide the required enforcement of the constraint and the annotation is not needed.

But that wouldn't help us look at annotation processing, so we will note that this example is contrived and carry on.

Worked Example
Processing

- Without some form of processing the annotation will have no effect
- Processing can be performed at:
  - Runtime
  - Class Loading/Post Compilation
  - Compilation/Pre-compilation
  - This list is the same list as Java 1.5, only the specifics of compilation/pre-compilation have changed
- We'll consider all of these in the following examples

We covered some of these points earlier, but it is worth re-iterating them now. The presence of an annotation does not change the way the code executes. It is only through processing that any change can occur.

There are a number of points in the build, execution cycle that this processing can occur and the following slides examine these options.

Worked Example
Reflection Processor

- Processing is performed at runtime
- Use reflection to find annotation
  - Use the isAnnotationPresent()/getAnnotation() methods
  - On Class, Method or Field
- Processing could be placed in client or a decorator
  - Though a client is not forced to do this

Starting with runtime processing, the first option is to use reflection to find the annotation and perform some processing on it. This, of course, required that the retention of the annotation is set to RUNTIME, otherwise it will have been discarded before this point.

The isAnnotationPresent() and getAnnotation() meta information methods are available through reflection and this it is possible to find and use the StringConstraint annotation, from our example.

Reflection could be performed in the client of the Address class, or through some form of delegate. Though in either case the client needs to understand that the annotation could be present and write code that deals with this.

Worked Example
Reflection Processor

- Simple reflection code for annotation:

```
...
for (Field f : Address.class.getDeclaredFields()) {
  f.setAccessible(true);
  if (f.isAnnotationPresent(StringConstraint.class)) {
    StringConstraint sc = f.getAnnotation(StringConstraint.class);
    String constraint = sc.value();
    String postcode = (String)f.get(address);
    Pattern p = Pattern.compile(constraint);
    Matcher m = p.matcher(postcode);
    if (!m.find()) {  // throw an exception }
  }
}
```

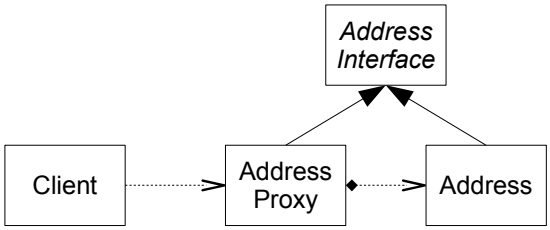

The actual processing code iterates through the fields of the Address class looking for the StringConstraint annotation. If this is found the constraint is obtained from the field and the value of the constraint is used to perfrom the regex test. The value is not defined in the meta-information but is obtained from the instance of the annotation.

This code does test the current value of the field, so it would be required that the programming use this code after the field had been modified. Thus the best exception to throw would probably be an IllegalStateException.

Note that the setAccessible() method call will only work if the security manager allows this, thus it is possible that the code will fail to access a private field.

Worked Example
Dynamic Proxy Processor

- A runtime processor
- Uses reflection to process annotation
- Wrap the Address class in a dynamic proxy
- Need to introduce an Interface on which to base the dynamic proxy



41

Using the above reflection example we can use a dynamic proxy to wrap u the processing, this provides less impact on the client code as only the creation point will need to understand that a dynamic proxy is being used, and this could be hidden in a factory.

In order to use a dynamic proxy an interface for the Address class should be provided, not a bad thing.

Worked Example
Dynamic Proxy Processor

- Dynamic Proxy shell:

```
class AddressProxy implements java.lang.reflect.InvocationHandler {
  public static Object newInstance(Object obj) {
    return java.lang.reflect.Proxy.newProxyInstance(
    obj.getClass().getClassLoader(),
    obj.getClass().getInterfaces(),
    new AddressProxy(obj));
  }

  private AddressProxy(Object obj) {
    this.obj = obj;
  }
  // next slide
}
```


This slide shows the infrastructure of the dynamic proxy implementation. A new instance of the Address class is created and this uses the dynamic proxy's private constructor to keep a reference to the class being wrapped. So far so standard dynamic proxy.

Worked Example
Dynamic Proxy Processor

- Dynamic Proxy invoke method:

```
public Object invoke(Object proxy, Method m, Object[] args)
        throws Throwable {
  Object result;
  try {
    result = m.invoke(obj, args);
    Method om = obj.getClass().getMethod(m.getName());
    if (f.isAnnotationPresent(StringConstraint.class)) {
      StringConstraint sc = f.getAnnotation(StringConstraint.class)
      // as in the reflection code
    }
  } catch (InvocationTargetException e) { throw e.getTargetException(); }
  return result;
}
```


We provide an invoke method which performs the reflective lookup of the annotation and the processing required. In this case the code demonstrates how to look for the annotation on methods of the Address class, for example an accessor for the postcode.

Worked Example
Dynamic Proxy Processor

- Usage:

```
AddressInterface address =
   (AddressInterface) AddressProxy.newInstance(
      new Address("138", "BS2 0JA"));
String postcode = address.getPostcode();
```

- Still requires cooperation of the client
- The dynamic proxy could be generated by an annotation processor

The client code only needs to understand that a dynamic proxy is being used at the point of creation. Thus will have less impact than the reflection solution if this is used directly in the client code.

However this is still worse than the standard OO solution which has no impact on the client code whatsoever.

It is possible for the dynamic proxy to be generated by an annotation processor during compilation.

Worked Example
Byte Code Modification

- Processing can be performed on the bytecode
  - Post compilation
    - A separate phase in the build chain
  - Class Loading (a Custom Class Loader)
    - Need to modify a class before it is loaded, a class cannot normally be unloaded
  - Instrumentation API
    - Allows a class modification before main() is executed
- There are a number of libraries to facilitate this:
  - BCEL
  - ASM
  - JAssist

Another option is to modify the byte code, this could be performed at class loading time or as a post complication step.

If the class loading option is used this would require that the class is modified the first time it is loaded, as it is generally not possible for a class to be reloaded into the type system. There are 2 possible avenues to allow a class to be modified during class loading, either through the creation of a custom class loader or through the use of the instrumentation API. I will not provide many details of this here, but this is easy enough to find information on with a quick internet search.

There are a number of frameworks which allow the processing and modification of byte code. I would suggest having a look at all of these before deciding on which one to use, as they are all slightly different, but basically provide the same outcome. You should find one that suits your needs and preferred API style.

Worked Example
Byte Code Modification

- Accessor method can be modifed from:

```
@StringConstraint("^([A-Z][A-Z][0-9]?[0-9] [0-9][A-Z][A-Z])");
public void setPostCode(String postCode) {
  _postCode = postCode;
}
```

- to:

```
public void setPostCode(String postCode) {
  Pattern p = Pattern.compile("^([A-Z][A-Z][0-9]?[0-9] [0-9][A-Z][A-Z])");
  Matcher m = p.matcher(postCode);
  if (!m.find()) {  // throw an exception }
  _postCode = postCode;
}
```


What would we want to do with byte code modification? We would want to insert the check into a mutator such as a set method. This code follows the basic regex check performed in the reflection code and so is simple enough to create.

This would allow the a zero impact approach on the client, it does not need to understand how the constraint is processed, the constraint is just enforced at runtime.

Worked Example
Byte Code Modification

- Plugging into the Instrumentation API:

```
public class ExampleAgent implements ClassFileTransformer {
  public byte[] transform(ClassLoader loader, String cname, Class class,
                          ProtectionDomain domain, byte[] bytes)
                          throws IllegalClassFormatException {
    try {
      // find annotation usage
      ClassWriter writer = new ClassWriter(false);
      // write modifications
      return writer.toByteArray();
    } catch (IllegalStateException e) { // throw exception }
    return null;
  }
  public static void premain(String arglist, Instrumentation inst) {
       inst.addTransformer(new ExampleAgent());
  }}
```


For the byte code processing to be performed when the class is loaded the instrumentation API is a good option, this code show s the outline of the code required to plug into this framework.

The API defines a static premain(...) which as the name suggests is executed before main. The code loaded at the start of the program can then be modified by the ExampleAgent, using the preferred byte code modification framework.

It may be possible, if not now but in the future, to do this using the compiler API, by defining the source code and compiling this in memory and returning the byte code from memory. Though of course the byte code supplied to the ClassFileTransformer would need to be decompiled first in order to be modified, a possibly more difficult task than modifying the byte code itself.

Worked Example
Byte Code Modification (ASM)

- ASM writing changes to a method:

```
public class ExampleGenerator extends ClassAdapter {
  public MethodReplacer(ClassVisitor cv, String mname, String mdesc) { ... };

  public void visit(int version, int access, String name, String signature, String
                    superName, String[] interfaces) { ... }

  public MethodVisitor visitMethod(int access, String name, String desc, String
                                   signature, String[] exceptions) {
    // write changes to method here
    return super.visitMethod(access, newName, desc, signature, exceptions);
  }
}
```


This example shows the outline of an ASM adapter which allows modification of the byte code. The API is simple, using the visitor approach, and thus allows the author of the code to only write code for the elements that need to be modified.

This is isn't covered in great detail because this is a talk all by itself, but there are a decent number of resources available online if you want to consider this approach further.

Worked Example
Byte Code Modification

- Issues:
  - Complex
  - Maintainability
  - Choosing the right tool/API
- Accessor method is modified with no impact (or expectations) on the client

It is obvious that byte code modification is a complex area and so needs a good level of understanding to achieve annotation processing. It is good to think of the byte code as a new language, it is not Java after all, so one would be required to understand the JVM byte code language to some extent in order to successfully modify it. The frameworks available do make this as easy as possible.

Whilst the byte code is very stable, it is worth noting that it does change from time to time, for example in Java 5, so future maintenance would have to be considered and more importantly how many people have the expertise to write this code.

Finally it does provide a good alternative to the standard OO approach, though it does seem a lot harder.

Worked Example
Compilation

- Use Annotation Processor to generate extra code:
  - Cannot modify the existing code
  - Can generate new base class or sub class
    - Cooperation required from the class using the annotation
    - Reference to class that doesn't yet exist
      - All sorted out at the end of annotation processing

The other stage of the build cycle where we can perform processing is the compilation stage. Thus we can add code that allow s the constraint to be enforced. The one problem we have to overcome is that we cannot modify the existing code using the standard tools. There is a simple way around this problem, we can define a abstract base class to contain the constraint check and generate this during compilation.

This requires knowledge in the Address class that such a base class will be created and code will need to be added that uses the constraint checks, this may cause problems in IDEs, though something like eclipse which has annotation processor support should be able to handle this.

Worked Example
Compilation Processor

- Modified Address class (to use generated base class):

```
public final class Address extends AddressBase
{
  public Address(String houseNumber, String postcode) { ... }

  public void setPostcode(String postcode) {
    validateConstraint_postcode(postcode);
    _postcode = postcode;
  }

  ...
  private final @StringConstraint("^([A-Z][A-Z][0-9]?[0-9] [0-9][A-Z][A-Z])")
                          String _postcode;
}
```

51

So we can modify the Address class to use a to be generated base class.

The base class would introduce a protected method, validateConstraint_postcode() that performs the checking, and this could be added to the constructor and any mutator methods.

Worked Example
Compilation Processor

```
public boolean process(Set<? extends TypeElement> annotations,
                       RoundEnvironment roundEnv) {
  for (Element e : roundEnv.getElementsAnnotatedWith(StringConstraint.class)) {
    if (e.getKind() != ElementKind.FIELD) { continue; }
    String name = capitalize(e.getSimpleName().toString());
    TypeElement clazz = (TypeElement) e.getEnclosingElement();
    try {
      Filer filer = processingEnv.getFiler()
      JavaFileObject f = filer.createSourceFile(clazz.getQualifiedName() + "Base");
      Writer w = f.openWriter();
      // write abstract class
    } catch (IOException x) { };
  }
  return true;
}
```

52

The processor for this would be as our earlier example to generate a new class, using a Filer to open a new file for the base class and generating the contents from the annotation information.

The cycle processing would be able to include this new file in the next cycle and allow the Address class to be compiled correctly.

Worked Example
Compilation Processor

- Generated code

```
...
public abstract class AddressBase {
   protected AddressBase() {}

   protected final String validateConstraint_postcode(java.lang.String value) {
     Pattern p = Pattern.compile("^([A-Z][A-Z][0-9]?[0-9] [0-9][A-Z][A-Z])");
     Matcher m = p.matcher(value);
     if (!m.find()) { throw new IllegalArgumentException("Constraint failed"); }
     return value;
   }
}
```


This slide shows what the generated code would look like, it looks very similar to the earlier reflection example.

In this example the name of the method is generated from the name of the field to which the constraint is applied. The name of the class is derived from the name of the class in which the constraint exists.

It would be better if the annotation could allow the name of the base class and the name of the method to be defined by the client code, so that the class that using the annotation has control when the processor causes a compilation error because it uses a name that is already in use.

Worked Example
Compilation Processor

- Compiler output:

```
Address.java:3: cannot find symbol
symbol: class AddressBase
public class Address extends AddressBase
                             ^
```

- This isn't an error, the file isn't generated when this message is created
- The next round will compile both Address and the generated AddressBase
- The annotation could allow the name of the base class and the name of the method to be specified

As you remember from earlier in the slides the dependency on a class that does not exist in the first round of annotation processing will cause a message to be displayed in the compiler output, even though this does not eventually lead to a compilation error.

Other Approaches

- The annotation processor is the most straight forward (annotation processing) mechanism to use with the least drawbacks
- Some tools are available which allow changes to the source files using a JSR 269 like approach:
  - Spoon (http://spoon.gforge.inria.fr/)
  - Jackpot (http://jackpot.netbeans.org/)
- The Compiler API could be used (JSR 199) to change the byte code being written from the compiler

The annotation processor does seem the simplest mechanism to use (if you discount the standard OO approach). But there are some issues with this approach used, what if the class that uses the annotation already has a base class?

It would be better to be able to modify the existing code during annotation processing and write this back before compilation. There are libraries which provide this functionality, Spoon and Jackpot. Spoon is the more general tool as this is not tied to an IDE, as Jackpot is, though it is worth looking at both of these tools anyway. Both of the implementations are already using, or are moving towards a JSR 269 approach, with extensions, so the code can be navigated and modified using the javax.lang.model classes. The modification is provided through extensions to the APIs.

Another approach might be to use the compiler API, which is alreay used by at least one project for scripting languages on the JVM, allowing the script source to be dynamically compiled to byte code. I haven't had time to look into this approach, so I have not included it in this session, but it is worth investigation.

Summary

- We have covered:
  - Annotations
    - Syntax
    - Intent
  - The new 1.6 annotation features
  - Processing annotations
    - Runtime, Load and Compile time options
    - The limitations of “standard” approaches
  - Some tools to considers which overcome these limitations

A lot of material has been covered, but a lot has been skipped over. It is a large subject and I think to fully explore the whole subject of annotation processing would be ambitious in one session. Though, hopefully, I have covered enough ground to provide you a good place to start from.

References

- JSR 269: Pluggable Annotation Processing API (http://jcp.org/en/jsr/detail?id=269)
- JSR 199: Java Compiler API (http://jcp.org/en/jsr/detail?id=199)
- JSR 175: A Metadata Facility for the Java Programming Language (http://jcp.org/en/jsr/detail?id=175)
- ASM (http://asm.objectweb.org/)
- BCEL (http://jakarta.apache.org/bcel/)
- JAssist (http://jassistant.sourceforge.net/)
- Mirror-based design (http://bracha.org/mirrors.pdf)
- Spoon (http://spoon.gforge.inria.fr/)
- Jackpot (http://jackpot.netbeans.org/)
- Benedict Heal - What are Java annotations for? (http://www.benedictheal.com/java/index.htm)

These are some of the specifications, frameworks and tools I have mentioned in the slides, you may want to have a look at least a few of these.

There is certainly more information appearing on the internet that considers annotation processing, though not a huge amount yet covering the new Java 6 features.

A final puzzle

- A couple of years ago Benedict Heal presented a session on annotations (when they were added in Java 5).
- He closed with a “what happens now?” slide, here is the code he presented:

```
@persistent( table="t1")
   @vars(  @var(name="age",
                  type="Integer", range=("1-99")),
           @var(name="address", type="MyString",
                  checkerClass="com.MyChecker"))
   @methods({
        @method(name="m1", body="if .. {}"),
        @method(name="m2", body="while..") }

class Thing {  }
```

How would you write the annotation processor(s) for this?

You can find Benedict's slides at http://www.benedictheal.com/java/annotations/annotations.ppt