Annotation Types

this slide usually left blank

Annotations in Java

- Added back in Java 5
  - now widely used
- Modifiers that can be added to code
  - package declarations
  - type declarations
  - constructors
  - methods
  - fields
  - parameters
  - variables
  - other annotations
- Standard annotations supplied
  - also framework for user defined annotations
  - access to annotations for processing tools

Syntax of Annotations

- General format:

```
@annotation ( name="value", … )
```

- "Marker" annotations have no parameters
  - parentheses may be omitted
- Many annotations take parameters
  - string valued parameters
  - more complex annotations may take arrays
  - annotations can take annotations as parameters
- Annotation is an instance of an "annotation type"

Defining a New Annotation Type

- Defined using `@interface`
  - compile to a .class file
- Applied using '@' followed by name of annotation

Provisional.java

```
/**
 * Indicate a method or class is under development
 * and  may change
 */
public @interface Provisional {

}
```

No arguments to annotation, so type body is empty.

```
public class Person {
  …
  @Provisional
  public Date getDob( ) {
    return null;
  }
  …
}
```

Annotations with Parameters

- Annotation parameter represented as a method
  - type of parameter is return type of method
- Annotation must now supply value for parameter
  - as name = "value"

Provisional.java

```
/**
 * Indicate a method or class is under development
 * and may change
 */
public @interface Provisional {
  String developer();  // Name of developer
}
```

```
public class Person {
  @Provisional (developer = "George")
  public Date getDob( ) {
    return null;
  }
  …
}
```

Parameter name/value must be specified here

Annotations with Parameters

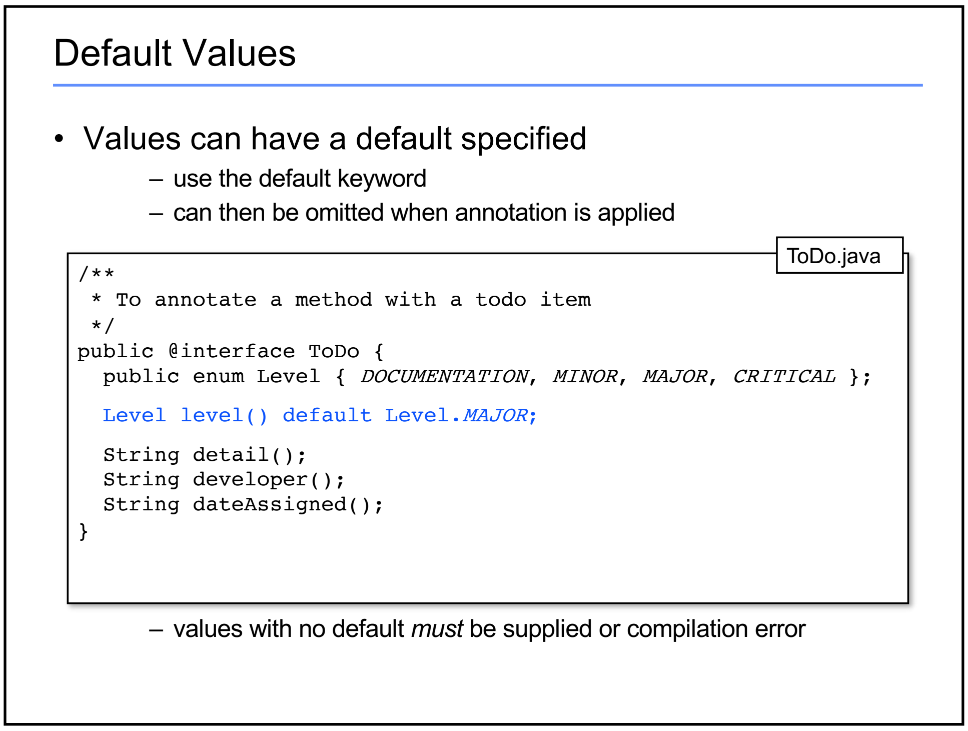
- If parameter is called *value*, name can be omitted

Provisional.java

```
/**
 * Indicate a method or class is under development
 * and may change
 */
public @interface Provisional {
  String value();  // Name of developer
}
```

```
public class Person {
  @Provisional ("George")
  public Date getDob( ) {
    return null;
  }
  …
}
```

Parameter name/value must be specified here



Default Values

- Values can have a default specified
  - use the default keyword
  - can then be omitted when annotation is applied

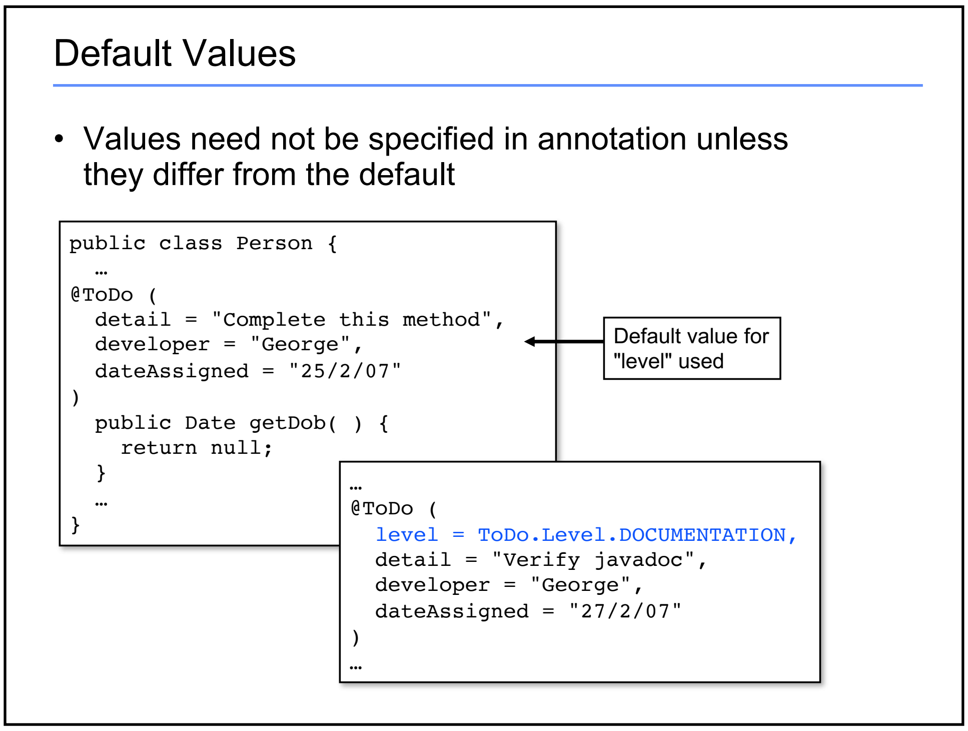
ToDo.java

```
/**
 * To annotate a method with a todo item
 */
public @interface ToDo {
  public enum Level { DOCUMENTATION, MINOR, MAJOR, CRITICAL };

  Level level() default Level.MAJOR;

  String detail();
  String developer();
  String dateAssigned();
}
```

  - values with no default *must* be supplied or compilation error



Default Values

- Values need not be specified in annotation unless they differ from the default

```
public class Person {
  …
@ToDo (
  detail = "Complete this method",
  developer = "George",
  dateAssigned = "25/2/07"
)
  public Date getDob( ) {
    return null;
  }
  …
}
```

Default value for "level" used

```
…
@ToDo (
  level = ToDo.Level.DOCUMENTATION,
  detail = "Verify javadoc",
  developer = "George",
  dateAssigned = "27/2/07"
)
…
```

Annotating Annotation Types

- Meta-annotations?
- Allows annotation to be better targeted
  - compiler can check for correct usage
- `@Target` meta-annotation
  - specifies entities to which annotation can be applied

```
import java.lang.annotation.*;
@Target ( { ElementType.TYPE,
   ElementType.METHOD,
   ElementType.CONSTRUCTOR,
   ElementType.ANNOTATION_TYPE } )
public @interface ToDo {
  …
}
```

```
package java.lang.annotation;
public enum ElementType {
      TYPE,
      FIELD,
      METHOD,
      PARAMETER,
      CONSTRUCTOR,
      LOCAL_VARIABLE,
      ANNOTATION_TYPE,
      PACKAGE
}
```

Annotation Retention

- Set using the `@Retention` meta-annotation
- Specifies whether or not annotation information is retained after compilation

```
package java.lang.annotation;
public enum RetentionPolicy {
  SOURCE,   // Annotation is discarded by the compiler
  CLASS,    // Annotation stored in class file but ignored
            // by the VM (default)
  RUNTIME   // Annotation stored in class file and read by the VM
}
```

```
@Retention ( RetentionPolicy.SOURCE )
public @interface SuppressWarnings {
  …
}
```

Documenting Annotation Types

- Use the `@Documented` meta-annotation
  - instructs javadoc to process annotation type
  - and include in javadoc
- Requires `RetentionPolicy.RUNTIME`
  - since javadoc processes class files

```
import java.lang.annotation.*;
@Documented
@Retention ( RetentionPolicy.RUNTIME )
public @interface ToDo {
  …
}
```

Inheriting Annotations

- `@Inherited` meta-annotation
  - marker
- Specifies that annotation is inherited by subclasses
  - default is not to inherit

```
import java.lang.annotation.*;
@Documented
@Inherited
@Retention ( RetentionPolicy.RUNTIME )
public @interface Provisional  {
  …
}
```

```
@Provisional ( … )
public class BaseClass {
  …
}
```

```
public class SubClass
             extends BaseClass {
  …
}
```

`Provisional` annotation now available in `SubClass`

Working with Annotations

- Many frameworks process annotation information at runtime
  - examples include Spring and Hibernate
- Can use reflection APIs
  - to retrieve details of annotations
  - requires `RetentionPolicy.RUNTIME`
  - not always efficient
- Can process source files for annotations
  - perform additional validation
  - generate configuration files automatically
  - generate new types from existing source code
  - requires source code

Using the Reflection APIs

- `getAnnotation()` checks for specific annotation
  - returns instance of annotation type or null
- `getDeclaredAnnotations()` returns all annotations

```
import java.lang.reflect.*;
public class AnnotationTest {
  public static void main(String[] args) {
    String dev = args[1];
    try {
      for (Method m : Class.forName(args[0]).getMethods()) {
        ToDo td = m.getAnnotation(ToDo.class);
        if ( td != null && dev.equals(td.developer())) {
          System.out.print( m.getName()
                          + " ToDo: "
                          + td.level().toString() );
```

Using the Reflection APIs

```
          System.out.println(" by "
                            + td.developer()
                            + " as of date "
                            + td.dateAssigned() );
        }
      }
    } catch ( ClassNotFoundException ce ) {
      …
    }
  }
}
```

```
$ java AnnotationTest Person George
getAge ToDo: DOCUMENTATION by George as of date 27/2/15
setAge ToDo: DOCUMENTATION by George as of date 27/2/15
getDob ToDo: MAJOR by George as of date 27/2/15
```

An Annotation Processor

- Called by the compiler to handle source annotations

```
@SupportedSourceVersion(SourceVersion.RELEASE_8)
@SupportedAnnotationTypes("*")
public class CodeAnalyzerProcessor extends AbstractProcessor {
  @Override
  public boolean process( Set<? extends TypeElement> annotations,
                          RoundEnvironment env ) {

    for ( Element e : env.getRootElements() ) {
      System.out.println( "Root element: "
                         + e.getSimpleName()
                         + " [ "
                         + e.getKind().toString()
                         + " ] ");
    }
```

An Annotation Processor

```
    System.out.println("Looking for @Override...");
    for ( Element e : env.getElementsAnnotatedWith(
                             java.lang.Override.class) ) {
      System.out.println( "Annotated element: "
                           + e.getSimpleName()
                           + " [ "
                           + e.getKind().toString()
                           + " ] ");
    }
    return true;
  }
}
```

Installing the Processor

- Command line invocation possible

```
$ javac –processor CodeAnalyzerProcessor TestClass.java
```

```
public class TestClass {
  private int num;
  private String str;
  @Override
  public boolean equals( Object o ) {
    if ( this == o )
      return true;
    if ( (o == null) || !(o.getClass() == this.getClass()) )
      return false;
    TestClass tObj = (TestClass)o;
    return ( (num == tObj.num) && (str.equals(tObj.str)));

  }
}
```

```
Root element: TestClass [ CLASS ]
Looking for @Override...
Annotated element: equals [ METHOD ]
Looking for @Override...
Done
```

The Java Compiler API

- Available since Java 6
- Allows compiler to be invoked from within a program
  - options can be passed
  - diagnostics can be captured
  - annotation processing can be specified
  - AST available (read only) if required
- `javax.tools` package
- Allows source code annotations
  - to be processed programmatically

The Java Compiler API

- Works with a file manager
- Every compiler has an associated StandardFileManager
  - this is the native (or built-in) file manager
  - supplies file to be compiled
  - can supply your own
  - done by forwarding onto custom file manager

```
JavaFileManager fileManager =
    new ForwardingJavaFileManager(stdFileManager) {
        public void flush() {
            System.out.println("Starting flush");
            super.flush();
           System.out.println("Finished flush");
        }
};
```

The Java Compiler API

- Compilation Tasks
  - generated by the compiler using getTask

```
CompilationTask compiler.getTask(
    Writer out,
    JavaFileManager fileManager,
    DiagnosticListener<? super JavaFileObject> diagnosticListener,
    Iterable<String> options,
    Iterable<String> classes,
    Iterable<? extends JavaFileObject> compilationUnits)
```

  - out - Writer for output from the compiler; System.err if null
  - fileManager - if null use the compiler's standard filemanager
  - diagnosticListener - a diagnostic listener; if null use the compiler's default method for reporting diagnostics
  - options - compiler options, null means no options
  - classes - names of classes to be processed by annotation processing, null means no class names
  - compilationUnits - the compilation units to compile

Invoking the Java Compiler

```
public class RunCompiler {

  private final static String sourceDir =
                              directory for source files;
  private final static String binRoot   =
                              directory for compiled classes;
  private final static String [] compilerOptions =
                              new String[] {"-d", binRoot};

  public static void main(String[] args) {

    JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
    StandardJavaFileManager fileManager =
                compiler.getStandardFileManager(null, null, null);
```

Invoking the Java Compiler

```
    Iterable<? extends JavaFileObject> compilationUnits =
           fileManager.getJavaFileObjects( sourceDir
                                            + "TestClass.java" );

    CompilationTask task = compiler.getTask(null,
                                  fileManager, null,
                                  Arrays.asList(compilerOptions),
                                  null, compilationUnits);

    LinkedList<AbstractProcessor> processors =
                       new LinkedList<AbstractProcessor>();
    // Add an annotation processor to the list
    processors.add(new CodeAnalyzerProcessor());
    // Set the annotation processor to the compiler task
    task.setProcessors(processors);
    task.call();
    System.out.println("Done");
  }
}
```

Going Further

- Further analysis is possible
- AST for class is made available
  - compiler tree API
  - java.lang.model… packages
- Allows more sophisticated validation of source code
  - check idioms or patterns rather than language rules