Annotations

- Definition from Encarta Dictionary
- adding of notes: the addition of explanatory or critical comments to a text
- explanatory note: an explanatory or critical comment that has been added to a text

Java Annotations

- An annotation, in the Java computer programming language, is a special form of syntactic metadata that can be added to Java source code.
- Classes, methods, variables, parameters and packages may be annotated. Unlike Javadoc tags, Java annotations are reflective in that they are embedded in class files generated by the compiler and may be retained by the Java VM to be made retrievable at run-time.
- Annotations provide data about a program that is not part of the program itself.
- They have no direct effect on the operation of the code they annotate.
- Annotations have a number of uses, among them:
- **Information for the compiler** — Annotations can be used by the compiler to detect errors or suppress warnings.
- **Compiler-time and deployment-time processing** — Software tools can process annotation information to generate code, XML files, and so forth.
- **Runtime processing** — Some annotations are available to be examined at runtime.

Annotations can be applied to a program's declarations of classes, fields, methods, and other program elements.

- The annotation appears first, often (by convention) on its own line, and may include *elements* with named or unnamed values:

```
@Author(name = “Benjamin Franklin”, date = “3/27/2003”)
class MyClass() { }
```

or

```
@SuppressWarnings(value = "unchecked")
void myMethod() { }
```

- If there is just one element named "value," then the name may be omitted, as in:

```
@SuppressWarnings("unchecked")
void myMethod() { }
```

- Also, if an annotation has no elements, the parentheses may be omitted, as in:

```
@Override
void mySuperMethod() { }
```

Documentation - Many annotations replace what would otherwise have been comments in code.

- Suppose that a software group has traditionally begun the body of every class with comments providing important info

```
public class Generation3List extends Generation2List
{
  // Author: John Doe
   // Date: 3/17/2002
  // Current revision: 6
  // Last modified: 4/12/2004
  // By: Jane Doe
  // Reviewers: Alice, Bill, Cindy

  // class code goes here
}
```

- To add this same metadata with an annotation, you must first define the *annotation type*.

```
@interface ClassPreamble
{
  String author();
  String date();
  int currentRevision() default 1;
  String lastModified() default "N/A";
  String lastModifiedBy() default "N/A";
  String[] reviewers(); // Note use of array
}
```

- The annotation type definition looks somewhat like an interface definition where the keyword interface is preceded by the @ character (@ = "AT" as in Annotation Type).
- The body of the annotation definition contains *annotation type element* declarations, which look a lot like methods. Note that they may define optional default values.
- Once the annotation type has been defined, you can use annotations of that type, with the values filled in, like this:

```
@ClassPreamble (
    author = "John Doe",
    date = "3/17/2002",
    currentRevision = 6,
    lastModified = "4/12/2004",
    lastModifiedBy = "Jane Doe"
    reviewers = {"Alice", "Bob", "Cindy"}
)
public class Generation3List extends Generation2List
{
    // class code goes here
}
```

Javadoc-generated documentation

- To make the information in @ClassPreamble appear in Javadoc-generated documentation, you must annotate the @ClassPreamble definition itself with the @Documented annotation:

```
import java.lang.annotation.*; // import this to use @Documented
@Documented
@interface ClassPreamble
{
  // Annotation element definitions
}
```

Annotations Used by the Compiler

- There are three annotation types that are predefined by the language specification itself:
  - @Deprecated,
  - @Override
  - @SuppressWarnings.

@Deprecated

- the @Deprecated annotation indicates that the marked element is deprecated and should no longer be used. The compiler generates a warning whenever a program uses a method, class, or field with the @Deprecated annotation.
- When an element is deprecated, it should also be documented using the Javadoc @deprecated tag, as shown in the following example.
- The use of the "@" symbol in both Javadoc comments and in annotations is not coincidental—they are related conceptually.
- Also, note that the Javadoc tag starts with a lowercase "d" and the annotation starts with an uppercase "D".

```
// Javadoc comment follows
 /*
  * @deprecated
  * explanation of why it was deprecated
  */
@Deprecated
static void deprecatedMethod() { }
```

@Override

- the @Override annotation informs the compiler that the element is meant to override an element declared in a superclass.

```
// mark method as a superclass method
// that has been overridden
@Override
int overriddenMethod() { }
```

- While it's not required to use this annotation when overriding a method, it helps to prevent errors.
- If a method marked with @Override fails to correctly override a method in one of its superclasses, the compiler generates an error.

@SuppressWarnings

- the @SuppressWarnings annotation tells the compiler to suppress specific warnings that it would otherwise generate. In the example below, a deprecated method is used and the compiler would normally generate a warning. In this case, however, the annotation causes the warning to be suppressed.

```
// use a deprecated method and tell
// compiler not to generate a warning
@SuppressWarnings("deprecation")
void useDeprecatedMethod() {
objectOne.deprecatedMethod(); //deprecation warning - suppressed
}
```

- Every compiler warning belongs to a category. The Java Language Specification lists two categories: "deprecation" and "unchecked." The "unchecked" warning can occur when interfacing with legacy code written before the advent of generics.. To suppress more than one category of warnings, use the following syntax:

```
@SuppressWarnings({"unchecked", "deprecation"})
```

Annotation Processing

- The more advanced uses of annotations include writing an annotation processor that can read a Java program and take actions based on its annotations.
- It might, for example, generate auxiliary source code, relieving the programmer of having to create boilerplate code that always follows predictable patterns.
- To facilitate this task, release 5.0 of the JDK includes an annotation processing tool, called apt. In release 6 of the JDK, the functionality of apt is a standard part of the Java compiler.
- To make annotation information available at runtime, the annotation type itself must be annotated with @Retention(RetentionPolicy.RUNTIME), as follows:

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

@Retention(RetentionPolicy.RUNTIME)
@interface AnnotationForRuntime {
  // Elements that give information
  // for runtime processing
}
```

Mapping with Annotations

- In previous lectures, we discussed the need to create mappings between the database model and the object model
- Mappings can be created as separate XML files, or as Java 5 annotations inline with the source code for your POJOs.
- Prior to annotations, the only way to create mappings was through XML files.
- Although using annotations is the newest way to define mappings, it is not automatically the best way to do so.
- We will briefly discuss the drawbacks and benefits of annotations before discussing when and how to apply them.

Cons of Annotations

- Using annotations immediately restricts your code to a Java 5 environment. This immediately rules out the use of annotations for some developers, as some application servers do not yet support this version of the JVM. Even when there are no technical reasons why a current JVM could not be used, many shops are quite conservative in the deployment of new technologies.
- If you are migrating from a Hibernate 2 environment or an existing Hibernate 3 environment, you will already have XML-based mapping files to support your code base. All else being equal, you will not want to re-express these mappings using annotations just for the sake of it
- If you are migrating from a legacy environment, you may not want to alter the preexisting POJO source code, in order to avoid contaminating known-good code with possible bugs.
- If you do not have the source code to your POJOs (because it has been lost, or because it was generated by an automated tool), you may prefer the use of external XML-based mappings to the decompilation of class files to obtain Java source code for alteration.
- Maintaining the mapping information as external XML files allows the mapping information to be changed to reflect business changes or schema alterations without forcing you to rebuild the application as a whole.
- Hibernate 3 support for annotation-based mappings is not yet as mature as its support for XML-based mapping files. For example, while you can still make appropriate foreign key relationships for use in schema generation, you cannot generally name the foreign keys used.

Pros of Annotations

- First, and perhaps most persuasively, we find annotations-based mappings to be far more intuitive than their XML-based alternatives, as they are immediately in the source code along with the properties that they are associated with.

- Partly as a result of this, annotations are less verbose than their XML equivalents, as evidenced by the contrast between Listings in the following slide.

A Minimal Class Mapped Using Annotations Vs. A Minimal Class Mapped Using XML

```
@Entity
public class Sample
{
    @Id
    private Integer id;
    private String name;

    //setters

    //getters
}
```

```
<hibernate-mapping>
  <class name="Sample">
    <id type="int" column="id">
      <generator class="native"/>
    </id>

    <property name="name" type="string"/>
  </class>
</hibernate-mapping>
```