Introduction

- Why annotations?
  - Enhance ease-of-development
  - Shift some code generation from programmer to compiler
- What are annotations?
  - Meta-tags
  - Can be applied at various levels
    - Package
    - Classes
    - Methods
    - Fields

@SuppressWarnings

```
public class Test_Deprecated
{
  @Deprecated
  public void doSomething()
 {
    System.out.println("Testing annotation name: 'Deprecated'");
 }
}

public class TestAnnotations
{
  public static void main(String arg[]) throws Exception {
      new TestAnnotations().doSomeTestNow();
  }
  @SuppressWarnings({"deprecation"})
   public void doSomeTestNow()
  {
    Test_Deprecated t2 = new Test_Deprecated();
    t2.doSomething();
  }

}
```

Java Annotations

Introduction

- Sun's Definition:
  - "It (annotation-based development) lets us avoid writing boilerplate code under many circumstances by enabling tools to generate it from annotations in the source code. This leads to a declarative programming style where the programmer says what should be done and tools emit the code to do it."
- Annotations can direct program behaviors through:
  - Source code
  - Compiler
  - Runtime (VM)

Annotation Types

- Single-Element: Provide a single value only

declaration →

```
public @interface MyAnnotation
{
    String value();
}
```

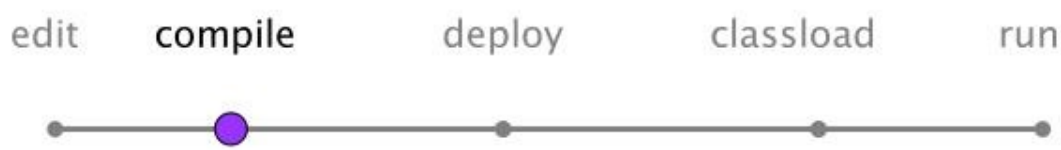
usage →

```
@MyAnnotation ("What to do")
 public void myMethod() { ... }
```

In-Class Example

- Annotating Event Handlers at runtime using Reflection

Annotation processing at compile time



- Standard Annotations
  - @Deprecated
  - @SuppressWarnings
  - @Override
  - @PostConstruct
  - @PreDestroy
  - @Resource
  - @Generated

Annotation Types

- Multi-value: Provide multiple data members

declaration →

```
public @interface MyAnnotation {
    String doSomething();
    int count;
    String date();
}
```

usage →

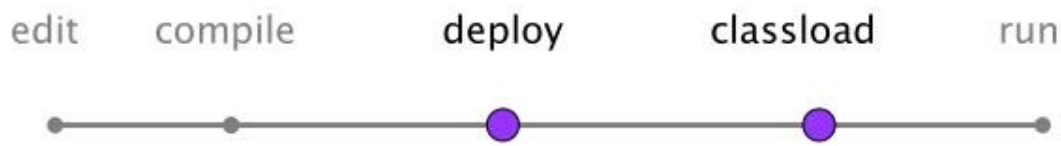
```
@MyAnnotation (doSomething="What
  to do", count=1, date="09-09-
  2005")
 public void myMethod() { ... }
```

Defining Annotation Type

- Annotation Declaration: @ + interface + annotation name
- Annotation Type Methods
  - No parameters
  - No throws clauses
  - Return types
    - Primitives
    - String
    - Class
    - Enum
    - An annotation type
    - Array of the above types
  - May be declared with defaults

```
public @interface MyAnnotation
{
    String doSomething() default "nothing";
}
```

Processing annotations on class files



- Bytecode enhancement based on annotations
  - Libraries like BCEL to read and write class files
- Bytecode Engineering Example (Core Java, pp. 926-934)

Annotation Types

declaration (MyAnno.java) →

```
package p;

public @interface MyAnno {}
```

usage (MyClass.java) →

```
import p.MyAnno;

@MyAnno class MyClass {}
```

- Marker: No Elements, except the annotation name
- Declaration is like declaring a normal type.
  - But notice the '@' in the declaration
- A marker annotation is the simplest type of annotation
- No member values – just presence or absence of the annotation

Annotations as tags/comments

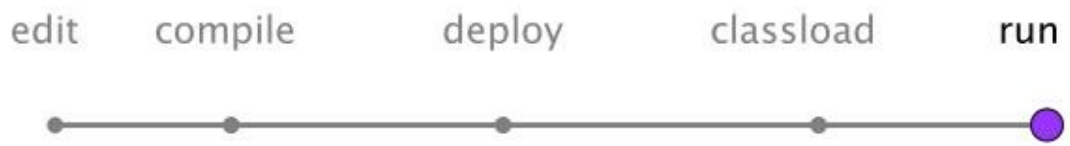
- Annotations as "standardized comments" – e.g., @Deprecated, versus "/* don't use this any more */"
  - Harder to mis-spell, easier to search, and less ambiguous.
- Defined entities (@deprecated) in javadoc are pretty good; but @depracated in javadoc fails silently.
- Not only an effective human-readable marker, but compiler also generates warning when you use a deprecated item.

@Deprecated

```
public class Test_Deprecated
{
  @Deprecated
  public void doSomething()
 {
     System.out.println("Testing annotation name: 'Deprecated'");
 }
}
```

```
public class TestAnnotations
{
  public static void main(String arg[]) throws Exception {
        new TestAnnotations();
  }
  public TestAnnotations() {
       Test_Deprecated t2=new Test_Deprecated();
       t2.doSomething(); // Generates warning
}
```

Reading annotations at runtime (JUnit 4)



- JUnit 4 test runner finds annotated classes, instantiates them, executes the annotated methods
- Test case classes don't need to subclass TestCase

```
@Test(expected = IndexOutOfBoundsException.class)
public void empty() {
    List l = new ArrayList<Object>();
    l.get(0);  // should throw exception
}
```

How are annotations used?

- There are use cases throughout the development cycle
  - Capabilities and challenges different at each point
- Many ways to read, and act upon, an annotation
  - Human-readable in source code
  - Built-in support in IDE
  - Annotation processing during compilation
  - Class file bytecode readers (BCEL)
  - Reflection at runtime

Where do you get annotations?

- Write your own
  - but non-standard annotations are of limited use, in practice, because of the investment required to write tooling that uses them.
- Industry standards (org.apache, com.bea, …)
  - Like APIs, annotations often start out proprietary and then become standardized even if the implementation stays proprietary.
- Built into the Java language (java.lang)

Package annotations

```
// file package-info.java:
@Deprecated
package p;
// no other contents in file
```

- Example use case: deprecate an entire package with @Deprecated
- But packages usually have multiple declarations! By convention, annotate only one of them, in a file named "package-info.java".
  - Analogous to package-info.html for javadoc
  - Because this name contains a dash, it is not a legal identifier; so, cannot contain a primary type.

Using an annotation: member restrictions

```
@A(null)                  // can't pass null value
@B(3+4)                   // ok to compute constants
@C(this.getClass())       // can't eval at runtime
class X {}
```

- Member values cannot be null (but can be an empty array or String)
- Values must be constant expressions
  - I.e., computed statically at compile time

Annotation types look like interfaces...

```
@interface ScreenFormat {
    enum COLOR { RED, BLUE, GREEN, BLACK }
    COLOR background() default BLACK;

    static final int SCREEN_DPI = 72;
    @interface VideoDevice {
        String name(); // name of device
        int dpi() default SCREEN_DPI; // resolution
    }
    VideoDevice[] supportedDevices();
}
```

- Implicitly extend interface java.lang.annotation.Annotation
  - defines equals(), hashCode(), toString(), and annotationType()
- Can declare constants, enums, and inner types.
- In bytecode, an annotation type *is* an interface, with a flag.

How do you annotate code?

```
@A class X {
    @A @B("quux") public void foo(@C x) { ... }
    @B private String s;
}
```

- Syntactically, annotations are modifiers, like "final".
- Annotations can be applied to any declaration: types (including enums and annotation types), fields, constructors, methods, parameters, enum constants, packages, and local variables.
  - Roughly speaking, the same things that you'd javadoc.
  - JSR-308 seeks to extend the set of things that can be annotated.
- Can put multiple annotations on one element, but they must each be of a different annotation type

Quickly, what's an annotation?

declaration →

```
public @interface Author {
    String name();
    int year();
}
```

usage →

```
@Author(name = "Brett Whiskers",
        year = 2008)
 class MyClass {}
```

- Program metadata – decorations on ordinary Java code.
- Like javadoc comments, but with syntax and strong types.
- Meant to be both human- and machine-readable.
- Note difference between "annotation" and "annotation type."

Built-in annotations

```
@Deprecated class Y {
    public abstract int foo();
}

class X extends Y {
    @SuppressWarnings("unchecked") List numbers;
    @Override public int foo() { ... }
}
```

- Defined in java.lang; support built into the compiler or IDE.
- @Deprecated warns when deprecated item is used
- @SuppressWarnings turns off compiler warnings
  - There is no standard list of suppressible warnings ☹
- @Override warns if a method is not truly an override
  - avoid subtle errors, e.g., equals(MyClass f) vs. equals(Object o)
  - `@Override` applies to methods in superclasses and implemented interface methods

Reflecting on annotations at runtime

```
@interface MaxLength { int value(); }

class ValidatingMethodCaller {
  String validate(java.lang.reflect.Method m, …) {
    MaxLength maxAnno = m.getAnnotation(MaxLength.class);
    String s = (String)m.invoke(…);
    if (maxAnno != null && s.length() > maxAnno.value() {
      throw new ValidationException("exceeded max length");
    }
    return s;
  }
}
```

- Annotations have to explicitly be given @Retention(RUNTIME).
- Reflection is about the only way to create an in-memory instance of an annotation type (because annotations are interfaces).

Built-in annotations *for annotations (Meta-annotations)*

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE, ElementType.FIELD })
@Documented
@Inherited
public @interface MyAnno { }
```

- @Retention: does MyAnno get compiled into class file, and does it get loaded into the VM so it can be reflected on? Default is CLASS.
- @Target: to which elements can MyAnno be applied?
- @Documented: will MyAnno be mentioned in javadoc of the classes or fields it is present on? (Is it part of the API contract?)
- @Inherited: if MyAnno is present on a class, is it inherited by subclasses?

The built-in meta-annotations control how the tools (compiler, javadoc, VM) will treat an annotation.