Introduction to Annotations

- Annotations
- Processing Annotations

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

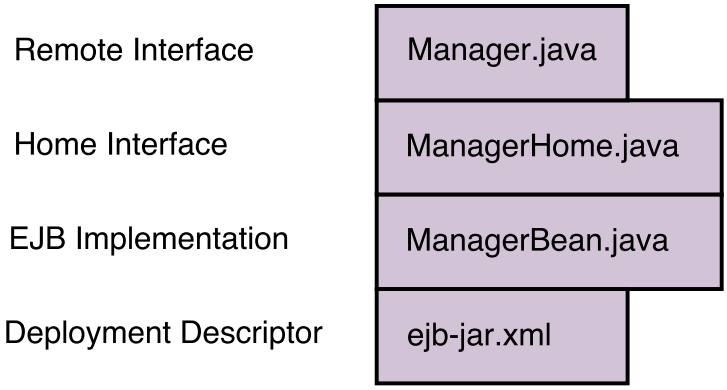
JDK 5 Programming

- JDK 5 introduces several new programming techniques that make Java coding easier and less error-prone, including:
  - Annotations
  - Enhanced *for* loop
  - Generic types
  - Formatted input and output
- In general, these techniques are **not** backward compatible with earlier-version Java Virtual Machines

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Metadata

- Annotations, also known as **metadata**, let you avoid writing companion classes, interfaces and configuration files
- For example, in EJB2 (prior to JDK 5), a simple session EJB required: two interfaces, an implementation class and an XML deployment descriptor

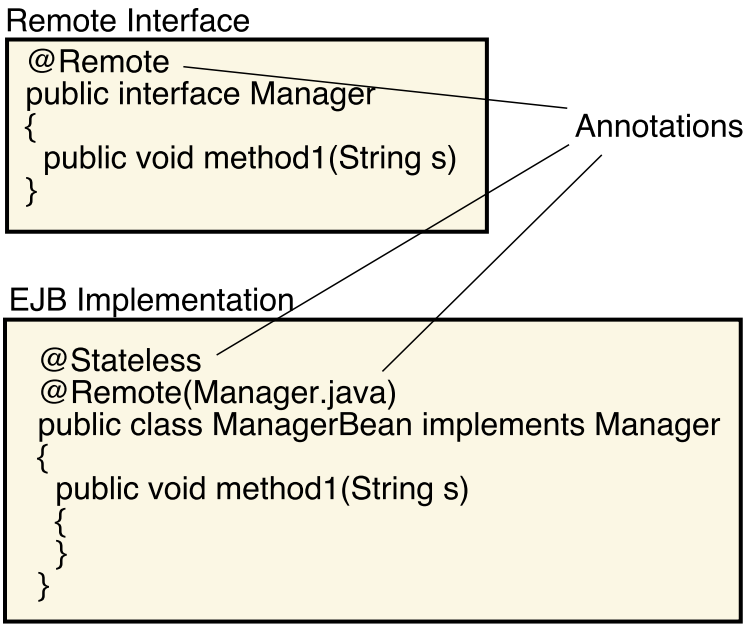


One common complaint against enterprise Java is the proliferation of accessory files that you need to create. For example, in EJB2, even the simplest session enterprise bean requires that you create and maintain four separate files.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Annotations

- In EJB3, which requires JDK 5, a simple session EJB requires only two files



EJB3 is much simpler than EJB2, since you can supply the required metadata as JDK 5 annotations rather than writing separate accessory files such as deployment descriptors.

The EJB3 container (runtime) processes the annotations and extracts the metadata.

The annotations shown here are specific to EJB3 and wouldn't be helpful if your not writing an EJB.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Where Can You Put Annotations?

- You can annotate the following Java syntax components:
  - Class
  - Field
  - Method
  - Method parameter

```
1    @Annotation1
2    public class Test
3    {
4       @Annotation2
5       private int myfield;
6
7       @Annotation3
8       public void myMethod(
9          @Annotation4 int x)
10      {
11      }
12   }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Standard Annotations

- JDK 5 defines a few **standard** annotations:

- @Override
- @Deprecated
- @SuppressWarnings
- @Retention
- @Target

The @Override annotation lets you explicitly indicate that you're overriding a method and helps avoid the case where you think you are overriding, but really aren't, normally because you didn't exactly match the argument signature in your overridden method.

The @Deprecated annotation is an alternative to the JavaDoc tag with a similar name and function. It's advantage over the JavaDoc tag is that it's available at runtime via reflection.

The @SuppressWarnings annotation lets you selectively turn off compiler warnings.

The @Retention annotation is a "meta-annotation" -- it lets an annotation author specify how long a custom annotation's data remains available

The @Target annotation is another meta-annotation -- it lets an annotation author specify what Java element (e.g. class or method) is meant to carry a custom annotation

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Standard Annotations Example

- By convention, annotations should precede other modifiers such as *public* or *static*

```
1    public class TestAnnotations
2    {
3      @SuppressWarnings("unchecked")
4      public void method1(String s)
5      {
6        ArrayList list = new ArrayList();
7        list.add("Hello");
8      }
9
10     @Override
11     public String toString()
12     {
13       return "My String";
14     }
15   }
```

Here we show a small program that uses annotations. In "method1()", we'd normally get a compiler warning in JDK 5 since the ArrayList isn't the generic form of that collection. To avoid the warning, we can use the annotation as shown in line 3.

The @Override annotation in line 10 lets us explicitly indicate that we are overriding the "toString()" method from our superclass. By putting it there, we would get a compiler ERROR if we didn't override the method correctly, for example if our "toString()" method took a parameter. Without the annotation, the compiler will silently let you write such a "toString()" method, but it wouldn't act as an override of "toString()" from the superclass. Such bugs are difficult to find.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Annotation Elements

- Annotations can define **elements**, which act like parameters or properties of the annotation
- There are three possibilities for an annotation's elements:
    - Annotations with no elements (aka **Marker** annotations)
    - Annotations with a single element
    - Annotations with multiple elements

```
1    @Override
2    @Override()
3
4    @SuppressWarnings("unchecked")
5    @SuppressWarnings(value="unchecked")
6
7    @MyAnnotation(p1="abc", p2="def")
```

If an annotation accepts no elements (parameters), you can omit the parenthesis as shown in line 1.

If an annotation accepts a single element, the element MUST be named "value". Note that you can omit the "value=" text as shown in line 4.

If an annotation accepts multiple elements, you must use the full syntax as shown in line 7.

Note that annotations can also accept arrays of elements; to provide the array, use curly braces "{ . . . }".

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Custom Annotations

- You can create your own annotations, which define a standard Java interface, with restrictions
- EJB3 and other frameworks define their own custom annotations

```
1    public @interface MyAnnotation
2    {
3      String p1(); // define element p1
4      String p2(); // define element p2
5    }
```

- To define an element, write a method in the annotation interface
- Such methods can accept no parameters and must return a primitive, String, Class, enum or other annotation
- The return value can also be an array of a supported type

Here we show a simple custom annotation that accepts two elements. Note the odd syntax of defining the interface that comprises the annotation.

To define an annotation's elements (parameters), write methods that adhere to the restrictions shown here.

If the annotation has a single element, its method must be named "value".

Remember that annotations represent metadata and thus the elements provide additional data.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Annotation Rentention Policies

- An annotation's **retention policy** indicates how long the annotation's elements should be retained
- Possible values:
  - **RetentionPolicy.SOURCE** The compiler discards the annotation
  - **RetentionPolicy.CLASS** The annotation is put in the compiled class, but may not be retained at runtime (default value)
  - **RetentionPolicy.RUNTIME** The annotation is available at runtime via reflection

```
1    @Retention(RetentionPolicy.RUNTIME)
2    public @interface MyAnnotation
3    {
4      String p1();
5      String p2();
6    }
```

Some annotations, especially those just for documentation, don't need to appear in the compiled class, while others are meant to be processed at runtime by frameworks or other custom programs.

The retention policy gives the annotation author control over how long the annotation is retained. Note that you specify the policy by writing the @Retention annotation.

RetentionPolicy is an enum defined in the java.lang.annotation package.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Default Element Values

- Optionally, you can define elements with **default values** that will automatically be supplied if the annotation user doesn't provide them

```
1    @Retention(RetentionPolicy.RUNTIME)
2    public @interface MyAnnotation
3    {
4      String p1();
5      String p2();
6      String p3() default "Parameter 3";
7    }
```

```
1    @MyAnnotation(p1="abc", p2="def")
2    public void method2()
3    {
4    }
5
6    @MyAnnotation(p1="abc", p2="def", p3="ghi")
7    public void method3()
8    {
9    }
```

To define an annotation element with a default value, use the default keyword.

In the second listing, we show two uses of the annotation: line 1 shows using the default, while line 6 demonstrates overriding the default.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Processing Annotations

- Custom annotations are ignored by the Java compiler and JVM
- To process annotations, you can either write a custom program that uses **reflection** to retrieve the annotations or use the JDK's **annotation processing tool**

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Sample Reflection-Based Annotation Processing

```
 1    package jdk5;
 2
 3    import java.lang.reflect.Method;
 4
 5    public class ProcessAnnotations
 6    {
 7      public static void main(String[] args)
 8        throws Exception
 9      {
10        Class<?> cls = TestAnnotations.class;
11        Method[] methods = cls.getDeclaredMethods();
12
13        for(Method m : methods)
14        {
15          System.out.println("Processing method: "
16           + m.getName());
```

Here we start a sample program that uses the reflection API to get a list of methods on a class that uses annotations. In line 13, we start a loop that iterates over all of the methods in the class.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Sample Reflection-Based Annotation Processing, cont'd

```
17          if (m.isAnnotationPresent(MyAnnotation.class))
18          {
19            MyAnnotation ma =
20              m.getAnnotation(MyAnnotation.class);
21
22            System.out.printf("P1: %s\n", ma.p1());
23            System.out.printf("P2: %s\n", ma.p2());
24            System.out.printf("P3: %s\n", ma.p3());
25          }
26        }
27      }
28    }
```

```
Processing method: method1
Processing method: method2
P1: abc
P2: def
P3: Parameter 3
Processing method: method3
P1: abc
P2: def
P3: ghi
Processing method: toString
```


Continuing the listing, in line 17, we check to see if the current method is decorated with our custom annotation. If so, retrieve the annotation object and then retrieve the annotation's elements (properties) and display them to the console.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- About annotations

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum