



Annotations

Decorating Your Code



Annotations

- Annotations are used to store meta-information
 - Data for the compiler, VM or container about what code is intended for and how it can or should be used
 - For example a method is used for testing, requires a transaction or can be used as the basis for a Web Service
- Typically meta-info has been supplied in XML files
 - More properly described as deployment descriptors
 - This has fragmented the way in which components are developed, sacrificing simplicity for reuse potential
 - Not all the information in the XML needs to be there
- Ad-hoc solutions do place meta-info beside the code
 - XDoclet generates code based on JavaDoc style comments



The Syntax of Annotations

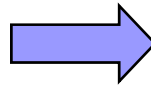
- Annotations are denoted by the '@' symbol
 - For example '@Test', @ThreadSafe and '@WebService'
- Information can be included with the annotation
 - For example '@Test(order = 1)' or '@WebService("SOAP")'
 - These are referred to as 'Annotation Elements'
- We say that code is 'decorated' with annotations
 - Packages, classes, members and variables can all be decorated
- The language defines a small number of annotations
 - Typically they will be defined by the JCP, other standards bodies and creators of open source and proprietary frameworks



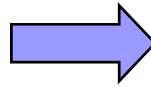
Creating Annotations

- Annotations are declared using '@interface'
 - They contain 'elements' which resemble method declarations
- All annotations implicitly extend the 'Annotation' interface
 - Declared in the package 'java.lang.annotation'
- An annotations use is specified via 'meta-annotations'
 - Annotations which are only used to decorate other annotations
 - '@Target' defines where the annotation can be used
 - This is one of the values of the 'ElementType' enumeration
 - '@Retention' defines when the annotation can be accessed
 - This is one of the values of the 'RetentionPolicy' enumeration

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestCase {
    String value() default "Anonymous";
}
```



```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {
    int order() default 0;
    boolean ignore() default false;
}
```



```
@TestCase
public class MyTestOne {
    public void op1() {
        System.out.println("op1 called");
    }
    @Test
    public void op2() {
        System.out.println("op2 called");
    }
    @Test(order = 2)
    public void op3() {
        System.out.println("op3 called");
    }
    @Test(order = 1, ignore = true)
    public void op4() {
        System.out.println("op4 called");
    }
}
```



Annotations and Reflection

- There are three retention policies
 - With 'SOURCE' annotations only exist in the source code
 - The annotation information is discarded by the compiler
 - With 'CLASS' annotations are compiled into the class file
 - But may not be accessible via reflection
 - With 'RUNTIME' annotations are in the class file and can be investigated via reflection
- Reflection has been expanded in 1.5
 - 'Field', 'Method' etc... now implement 'AnnotatedElement'
 - This contains methods for querying and obtaining annotations



```
// Version one of TestRunner – minus error handling and test ordering
```

```
public class TestRunner {  
    public TestRunner(Class<?> klass) {  
        if(klass.isAnnotationPresent(TestCase.class)) {  
            this.klass = klass;  
        }  
    }  
    public void runTests() {  
        Object instance = klass.newInstance();  
        for(Method m: findTestMethods()) {  
            m.invoke(instance,new Object[]{});  
        }  
    }  
    private List<Method> findTestMethods() {  
        List<Method> testMethods = new ArrayList<Method>();  
        for(Method m: klass.getMethods()) {  
            Annotation note = m.getAnnotation(Test.class);  
            if(note != null) {  
                testMethods.add(m);  
            }  
        }  
        return testMethods;  
    }  
    private Class klass;  
}
```



Annotation Elements

- Annotation elements are made up of:
 - The type of the element
 - This must be a primitive type, String, Class or enum
 - Arrays of the above are also allowed
 - The name of the element
 - An empty set of brackets
 - Elements cannot take parameters
 - An optional default value
- Elements are set using 'name=value' syntax
 - If a value is specified without a name it is assumed to initialize an element called 'value', otherwise a compiler error occurs



Using Annotation Elements

```
public void printTestDetails(PrintStream out) {
    String name = ((TestCase)klass.getAnnotation(TestCase.class)).value();
    out.println("Test methods in [" + name + "] are:");
    for(Method m: findTestMethods()) {
        printTestMethod(out, m);
    }
}
private void printTestMethod(PrintStream out, Method m) {
    Test t = m.getAnnotation(Test.class);
    out.print("\t");
    out.print(m.getName());
    out.print("] with order ");
    out.print(t.order());
    if(t.ignore()) {
        out.println(" to be ignored");
    } else {
        out.println(" to be executed");
    }
}
```



Built In Annotations

- Java 1.5 defines four meta-annotations
 - The '@Target' and '@Retention' attributes
 - '@Documented' marks an annotation as needing documentation
 - '@Inherited' marks an annotation as applying to subclasses
- It also defines three standard annotations
 - '@Deprecated' marks a feature as deprecated
 - '@Override' indicates a method should override a method of a base class, otherwise there is a compiler error
 - '@SuppressWarnings' causes a compiler not to issue warnings
 - The ID's of the warnings are passed as a parameter
 - Unfortunately the ID's are compiler specific



Validating Annotations in JSE 5/6

- Validating annotations is problematic
 - You can decorate your code with incompatible annotations
 - This is not discovered till the annotation reading tool examines your compiled class (and possibly not even then)
- Annotation validation should be supported by 'javac'
 - This can only be done by library writers supplying plug-in classes that enforce the appropriate rules
- Processing is available in both JSE5 and JSE6
 - JSE5 ships with the 'apt' command line utility
 - JSE6 provides an API for writing annotation processors
 - E.g. `javac -processor demos.MyProcessor *.java`



Generics in Java

Using Parameritized Types



Introducing Generics

- Generic code represents algorithms which stay the same regardless of the data they are used on
 - For example the store, retrieve and sort methods of a List work the same way regardless of what is placed in it
- Java already lets you write generic code
 - By using references of type 'java.lang.Object'
 - However this requires casts to keep the compiler happy
- Java 1.5 introduces full support for generic code
 - Classes and methods can be declared with type parameters
 - For example 'List<Date> list = new LinkedList<Date>()'
 - This removes the need for casts and increases type safety



An Example Generic Type

```
public class Pair<T,U> {
    public Pair(T first, U second) {
        super();
        this.first = first;
        this.second = second;
    }
    public T getFirst() {
        return first;
    }
    public U getSecond() {
        return second;
    }
    private T first;
    private U second;
}
```

```
Pair<Integer,Double> tst1 = new Pair<Integer,Double>(1,2.3);
Pair<Integer,Double> tst2 = new Pair<Integer,Double>(4,5.6);
Pair<Integer,String> tst3 = new Pair<Integer,String>(4,"abc");
Pair<String, Double> tst4 = new Pair<String,Double>("def",5.6);

boolean r1 = haveSameFirst(tst1,tst3);
boolean r2 = haveSameFirst(tst2,tst3);
boolean r3 = haveSameSecond(tst2,tst4);
boolean r4 = haveSameSecond(tst1,tst4);

System.out.printf("Results are %b, %b, %b, %b\n",r1,r2,r3,r4);
```

```
<T> boolean haveSameFirst(Pair<T,?> p1, Pair<T,?> p2) {
    return p1.getFirst().equals(p2.getFirst());
}
<T> boolean haveSameSecond(Pair<?,T> p1, Pair<?,T> p2) {
    return p1.getSecond().equals(p2.getSecond());
}
```



Generics Compared to C# and C++

- The implementation of Generics in Java is very different from that found in other languages
 - Generics in Java is entirely implemented by the compiler
 - The extra type info is erased and never given to the VM
- In C# and C++ new versions of the code are created
 - Every time 'LinkedList<T>' is used with a new value of 'T' a separate copy of the code is created and then used
- In Java the same implementation is always used
 - So the 'java.lang.Object' version of 'LinkedList' is the only one that ever exists and is run inside the Virtual Machine



Generics and Collections

- Generics is the major change to collections in Java 1.5
 - V1.5 collections are easy once you understand type parameters

```
private static void demoGenericList() {
    List<String> strList = new LinkedList<String>();
    strList.add("abc");
    strList.add("def");
    strList.add("ghi");
    strList.add("jkl");
    strList.add("mno");
    strList.add("pqr");
    strList.add("stu");

    System.out.println("LinkedList<String> contents are: ");
    for(String str : strList) {
        System.out.println(str);
    }
}
```



Generic Maps and Sets

```
private static void demoGenericMap() {
    Map<String,Employee> staff = new HashMap<String,Employee>();

    staff.put("ABC123",new Employee("ABC123","Joe Bloggs",20000));
    staff.put("DEF456",new Employee("DEF456","Fred Bloggs",30000));
    staff.put("GHI789",new Employee("GHI789","Dave Bloggs",40000));
    staff.put("JKL012",new Employee("JKL012","Peter Bloggs",50000));

    System.out.println("Map<String,Employee> contents are: ");

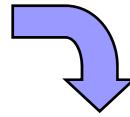
    Set<Map.Entry<String,Employee>> contents = staff.entrySet();
    for(Map.Entry<String,Employee> entry : contents) {
        System.out.println(entry.getKey() + " indexes " + entry.getValue());
    }
}
```



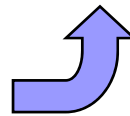
Creating Generic Types

- Using generics in your own code is straightforward
 - Simply introduce type parameters where required
- Some aspects of generics are non-intuitive
 - For example consider ‘void func(ArrayList<Employee> param)’
 - The call ‘func(new ArrayList<Manager>())’ will fail
 - Wildcards were introduced to cope with these situations:
 - ‘void func(ArrayList<?> param);’
 - Any type of ArrayList can be passed in ‘param’
 - ‘void func(ArrayList<? extends Employee> param);’
 - An ArrayList of any subtype of Employee can be passed
 - ‘void func(ArrayList<? super Manager> param);’
 - An ArrayList of any supertype of Manager can be passed

```
//Generic interface for lists
public interface List<T> {
    int size();
    void clear();
    void add(T value);
    void addAll(T ...values);
    T get(int index);
}
```



```
//Generic implementation of the list interface
public class LinkedList<T> implements List<T> {
    public void add(T value) {
        size++;
        if(listIsEmpty()) {
            first = new Node<T>(value,null);
        } else {
            Node<T> current = first;
            while(current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(new Node<T>(value,current));
        }
    }
    private Node<T> first;
    private int size;
}
```



```
//Generic node class with sample methods
class Node<T> {
    public Node<T> getNext() {
        return next;
    }
    public void setNext(Node<T> other) {
        next = other;
        other.previous = this;
    }
    public T getPayload() {
        return payload;
    }
    private Node<T> next;
    private Node<T> previous;
    private T payload;
}
```

```

public static void main(String[] args) {
    ArrayList<Employee> list1 = new ArrayList<Employee>();
    ArrayList<Manager> list2 = new ArrayList<Manager>();
    ArrayList<Director> list3 = new ArrayList<Director>();

    func1(list1);
    //func1(list2); //Will not compile
    //func1(list3); //Will not compile

    func2(list1);
    func2(list2);
    func2(list3);

    //func3(list1); //Will not compile
    func3(list2);
    func3(list3);

    func4(list1);
    func4(list2);
    //func4(list3); //Will not compile
}

```

```

class Employee {
}
class Manager extends Employee {
}
class Director extends Manager {
}

```

```

public static void func1(ArrayList<Employee> param) {
    System.out.println(param.size());
}
public static void func2(ArrayList<?> param) {
    System.out.println(param.size());
}
public static void func3(ArrayList<? extends Manager> param) {
    System.out.println(param.size());
}
public static void func4(ArrayList<? super Manager> param) {
    System.out.println(param.size());
}

```