Metadata

Notes on Annotations

Presentation Topics

In this presentation, we will discuss:

- Metadata
- Compile Time
- Deployment
- Run Time

Metadata

- What is it?
  - Typically described as "data about data"
  - Usually provides additional information about data
  - Basic example - comments in code
  - More complex example - schema
- Why is it needed?
  - Provides additional data about data, outside of the data itself
  - Keeps data clean
  - Can be used by tools to "learn" about the data without interrogating it

Annotate

- Metadata is used to define or clarify
  - In the days before computers, metadata was the notes on the side of a page of notes
  - Metadata is used to clarify or to organize a set of data into a more useable format
- Data to define data?
  - By itself a java program can compile and run just fine without metadata
  - Adding metadata makes the java program more consistent and reliable because it lets the compiler look at the notes and make decision before running the program

When is Meatadata used

- Metadata is used to define or clarify
  - In the days before computers, metadata was the notes on the side of a page of notes
  - Metadata is used to clarify or to organize a set of data into a more useable format
- Data to define data?
  - By itself a java program can compile and run just fine without metadata
  - Adding metadata makes the java program more consistent and reliable because it lets the compiler look at the notes and make decision before running the program

When are Annotations Used

- Classes
  - Annotations can be used to categorize classes
  - @WebService defines a class that will be used across the internet as a web service
- Fields
  - Annotation can be used to define characteristics of a variable
  - @Transient defines a variable that will never be stored in a database or written to a file
- Methods
  - @Override declares to the compiler that a method has been overridden from a superclass

Interpreting Annotations

- Annotations need to be interpreted in order to be useful.
- Development time interpretation allows a development environment to provide special handlers for beans and other java objects.
- Compile time interpretation allows the compiler to precheck a program to make sure that rules are followed before reaching a critical runtime situation
- Runtime interpretation allows the JVM to make decisions wile the program is running. A common use of runtime annotations is testing: @Test

Implementation of Annotations

- Annotations are implemented in the `java.lang.annotation` package
- This package provides library support for the Java programming language annotation facility
- Uses classes to decide how to handle @annotations
- Categorizes annotations into separate functionality

Categories of Metadata

There are three categories of Metadata:

- Documentation
- Compiler checking
- Code analysis

Documentation

- Code-level documentation is the most-often-cited use for Metadata
- It is the least relevant due to Javadoc
- Javadocs should come before @Annotations

```
/**
 * Delete multiple items from the list.
 *
 * @deprecated  Not for public use.
 *    This method is expected to be retained only as a package
 *    private method.  Replaced by
 *    {@link #remove(int)} and {@link #removeAll()}
 */
@Deprecated
public synchronized void delItems(int start, int end) {
...
}
```

Compiler Checking

- Metadata can be used to define expected behavior at compile time.
- The Java compiler checks to make sure that the indicated behavior is actually happening in your code.
- Annotations such as @Override ensure that a method is actually overriding a method in a superclass.
- Compiler annotations can save hours of debugging.

Code Analysis

- Annotations make up for generics when it comes to code analysis.
- Reflection works to analyze code but can only analyze what a method is asking for not what it expects.
- A generic method may ask for `<? extends Account>` but really expect a `SavingsAccount`
- Annotations can state exactly what is expected, code analyzers relying on reflection cannot.
- Complex systems like Spssring use extensive annotation.

So What Are Annotations Really?

- What are they?
  - Metadata facility for Java
    - Allow you to provide additional data alongside Java classes
    - Similar to Javadoc “metadata” facility
    - Source code comments that usually stay in source code
  - Expanded and formalized mechanism
    - “Competes” with Doclet / XDoclet
  - Recognized by Java compiler and other tools
  - Supported by `java.lang.annotation` package

Why Do We Need Annotations?

- Additional data can be read:
  - By the compiler
  - By source-code generation tools
  - At run time
- Additional data can be used to:
  - Generate boilerplate code
  - Maintain side-file dependencies
  - Mark things for tracking purposes (like TODOs)

How Do Annotations Work?

- Annotations don’t affect program semantics.
- Annotations are not allowed to disrupt execution.
- Represented as a new type within Java language.
- Have similar syntax to Javadoc.
- Applied like modifiers.
- Have constrained lifespan.
- Detected and interpreted by compiler.

What Do Annotations Look Like?

- The @ character signals to the compiler that this is an annotation.
- The name following the @ character is the name of the annotation.

```
@Entity
```

- In this case, the annotation name is Entity.

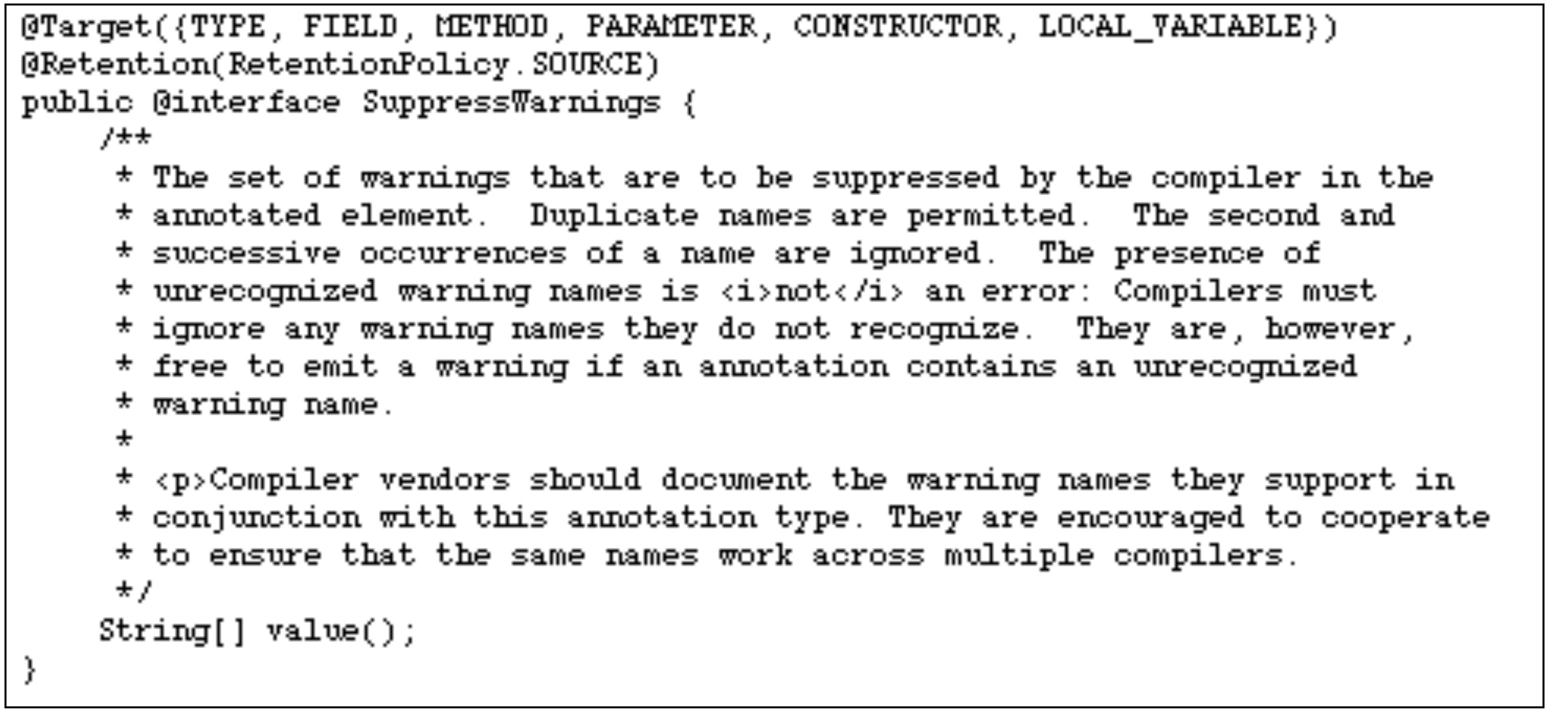
Annotation Type

- New type within language
  - `java.lang.annotation.Annotation`
  - Type can be annotated with other annotations
- Type is like an interface
  - Use `@interface` instead of interface
  - Supports methods
    - Must be declared without arguments
    - Methods can not throw `Exceptions`
  - Supports name-value-pairs (NVP)
    - Cannot have members; members defined through coding convention
    - Method name + return type define member as NVP
    - NVP can have default values (making them optional)

Annotation Syntax

- Syntax similar to Javadoc syntax
  - `@Deprecated` vs. `@deprecated`
  - `@` - represents annotation
  - `Deprecated` - represents annotation type
- Syntax more robust than Javadoc syntax
  - Can pass NVP
    - `@SupressWarnings` - no NVP passed
    - `@SuppressWarnings(value={"unchecked, fallthrough"})` - NVP passed
    - `@SuppressWarnings({"unchecked, fallthrough"})` - NVP passed; short-hand
  - Not whitespace sensitive

Annotation Example



```
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
    /**
     * The set of warnings that are to be suppressed by the compiler in the
     * annotated element.  Duplicate names are permitted.  The second and
     * successive occurrences of a name are ignored.  The presence of
     * unrecognized warning names is <i>not</i> an error: Compilers must
     * ignore any warning names they do not recognize.  They are, however,
     * free to emit a warning if an annotation contains an unrecognized
     * warning name.
     *
     * <p>Compiler vendors should document the warning names they support in
     * conjunction with this annotation type. They are encouraged to cooperate
     * to ensure that the same names work across multiple compilers.
     */
    String[] value();
}
```

Provided Annotations

- Two classifications:
  - Meta-annotations
    - Annotate annotations
    - Found in `java.lang.annotation`
    - Four main meta-annotations
    - Used to define annotation behaviors
  - Annotations
    - Core annotations
    - Found in `java.lang`; automatically imported in source
    - Three main annotations

Meta-Annotations

- `Target`
  - Identifies element applicability
  - Default / no value means applies to all elements
  - Possible values defined in `ElementType`
- `Retention`
  - Identifies lifespan of annotation
  - Three lifespans defined in `RetentionPolicy`:
    - `RetentionPolicy.SOURCE` - source only
    - `RetentionPolicy.CLASS` - source and class; not runtime
    - `RetentionPolicy.RUNTIME` - source, class, and runtime
  - Default / no value causes source-only retention
- `Documented` - something that should be documented
- `Inherited` - annotation should be carried through inheritance

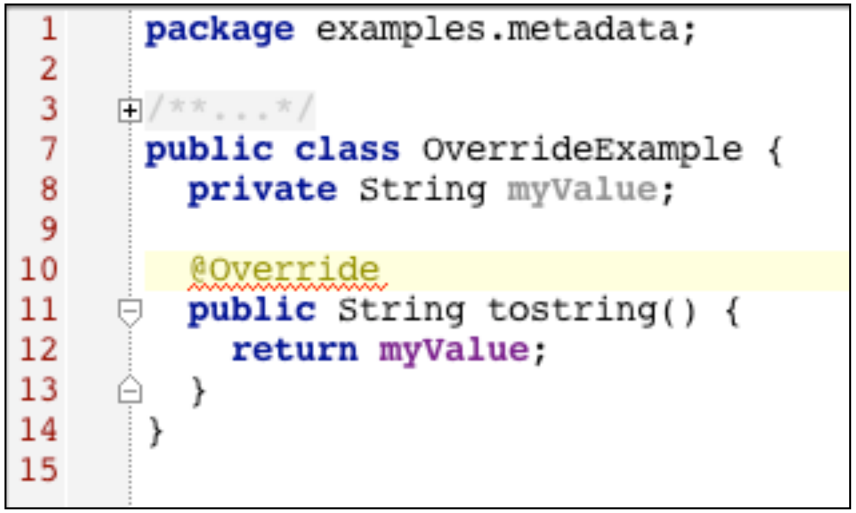
Three Categories of Annotations

- Marker annotations have no input data
  - `@Override`
  - No data, just the annotation name
- Single-value annotations provide a single data member
  - `@SuppressWarnings("unchecked")`
  - Only one argument, okay to use shortcut
  - Looks like method in Java
- Full annotations have multiple data members
  - `@MethodInfo(author = ”AOI", comments = ”Accepts Account number", date = ”June 20 2014", revision = 3)`
  - Accepts multiple types of data

Core Annotations -- `@Override`

- `@Override`
  - Used to notify compiler that method is overridden representation of inherited method
    - Causes compiler to validate overridden signature
    - Generates compiler errors if not in sync
  - `@Target(ElementType.METHOD)`
  - `@Retention(RetentionPolicy.SOURCE)`

@Override Example

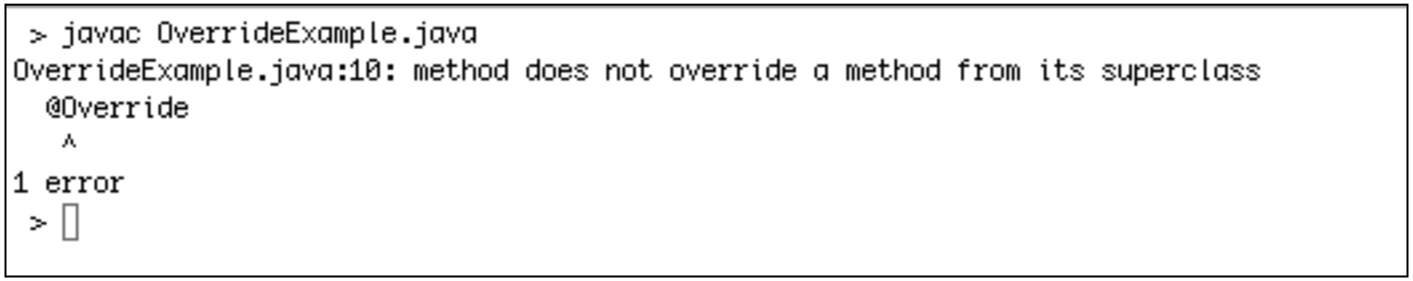


```java
package examples.metadata;

/**...*/
public class OverrideExample {
  private String myValue;

  @Override
  public String tostring() {
    return myValue;
  }
}

```



```
> javac OverrideExample.java
OverrideExample.java:10: method does not override a method from its superclass
  @Override
   ^
1 error
> 
```

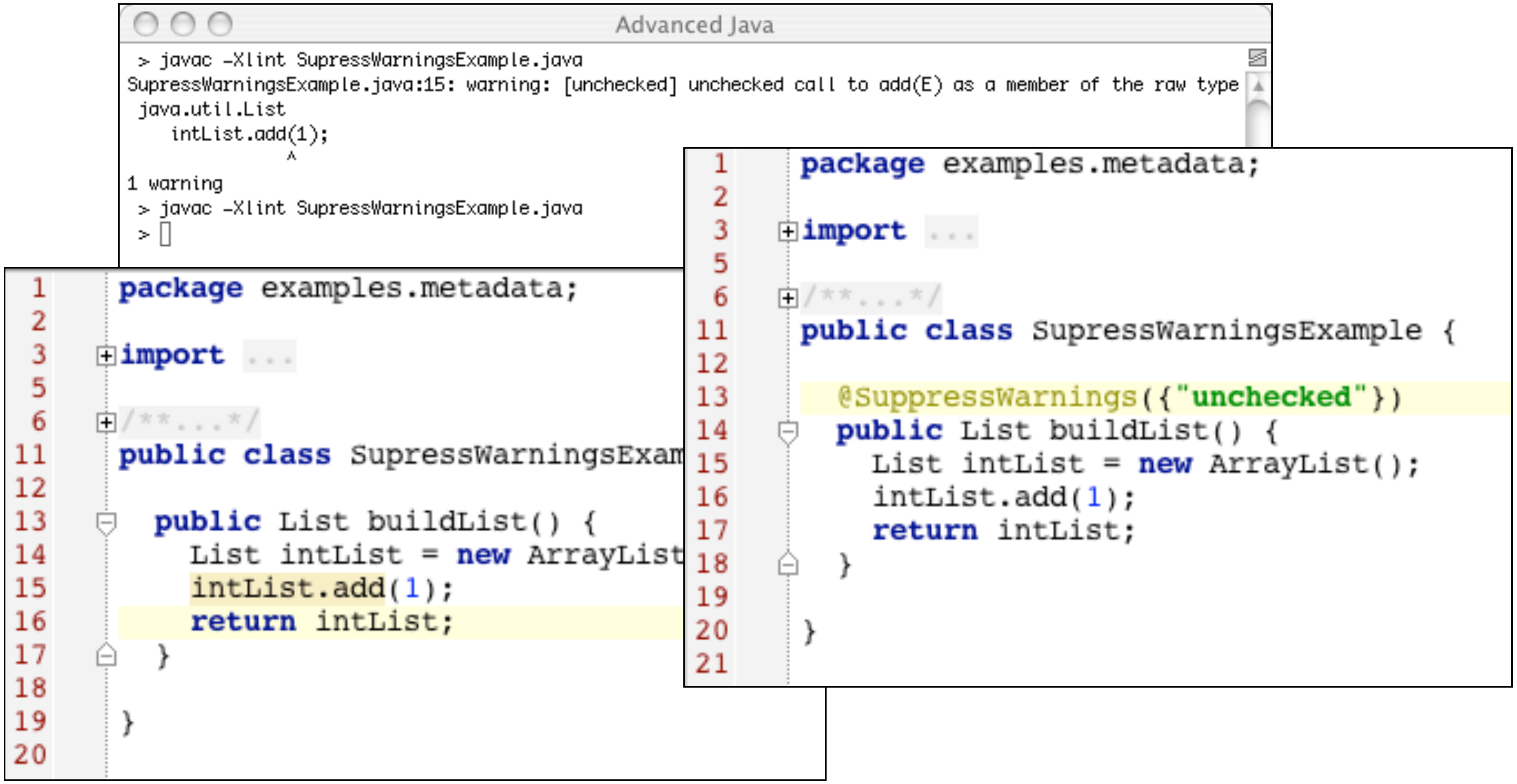
Core Annotations -- `@Deprecated`

- `@Deprecated`
  - Marker annotation similar to `@deprecated` in Javadoc
  - Used to notify compiler that use of `@Deprecated` element is discouraged
  - No `@Target` specified
  - `@Retention(RetentionPolicy.RUNTIME)`

Core Annotations – `@SupressWarnings`

- `@SupressWarnings`
  - Used to selectively turn off compiler warnings
  - Code-level alternative to `-Xlint` compiler flag
  - No `Enum` defining which warnings can be selected
  - Works in “hierarchical” manner
  - `@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})`
  - `@Retention(RetentionPolicy.SOURCE)`

@SuppressWarnings Example



Advanced Java

```
 > javac -Xlint SupressWarningsExample.java
SupressWarningsExample.java:15: warning: [unchecked] unchecked call to add(E) as a member of the raw type
 java.util.List
    intList.add(1);
               ^
1 warning
 > javac -Xlint SupressWarningsExample.java
 > 
```

```
1   package examples.metadata;
2
3   import ...
5
6   /**...*/
11  public class SupressWarningsExam
12
13     public List buildList() {
14        List intList = new ArrayList
15        intList.add(1);
16        return intList;
17     }
18
19  }
20
```

```
1   package examples.metadata;
2
3   import ...
5
6   /**...*/
11  public class SupressWarningsExample {
12
13     @SuppressWarnings({"unchecked"})
14     public List buildList() {
15        List intList = new ArrayList();
16        intList.add(1);
17        return intList;
18     }
19
20  }
21
```

Summary

- Annotations are useful in three different aspects
- Most useful as precompiled instructions to help manage code
- Annotations are built into the language as of Java 1.5
- It is possible to write customized annotations using Annotation API
- Annotations can be simple single statements or complex multi-arguments
- There is even an annotation API to annotate annotations

Creating Custom Annotations

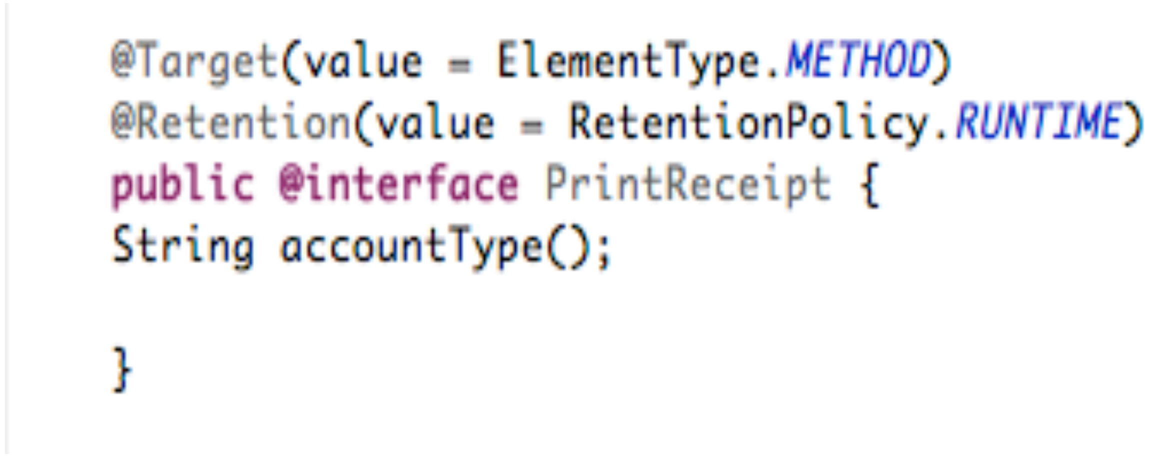
- Java includes built in Annotations
- It is possible to create custom annotations
- Custom annotations are built like a class or interface
- Use custom annotations the same way as standard annotations
- Use the reflection package to read annotations

@interface

- Annotations are creating by using the @interface before the annotation class name
- Annotations are used to define the properties of the Annotation class, for example:
  - @Target(ElementType.METHOD) defines the annotation for methods only
  - @Retention(RetentionPolicy.RUNTIME) defines the annotation as being available through the runtime of the annotated method

Typical Annotation

- This annotation is for methods and will be available during the runtime of this class

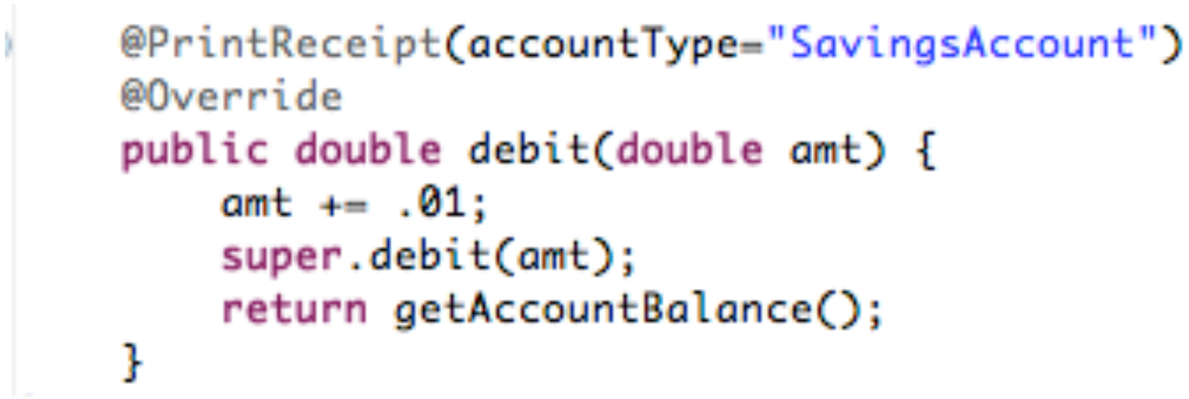


```
@Target(value = ElementType.METHOD)
@Retention(value = RetentionPolicy.RUNTIME)
public @interface PrintReceipt {
String accountType();

}
```

Marking a Method

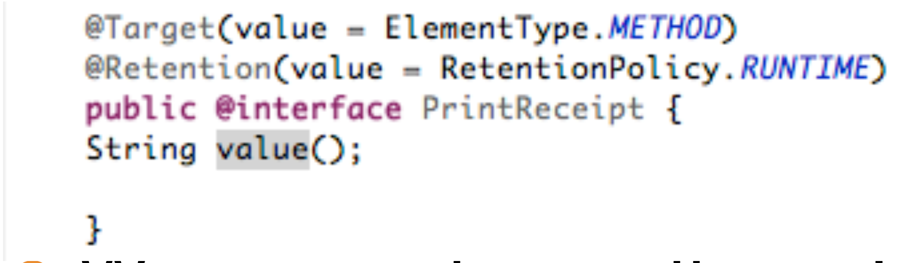
- Once the annotation is created we can mark the methods by annotating them with our custom annotation in the same we we use standard annotations:



```
@PrintReceipt(accountType="SavingsAccount")
@Override
public double debit(double amt) {
    amt += .01;
    super.debit(amt);
    return getAccountBalance();
}
```

Use value If There Is Only One Element

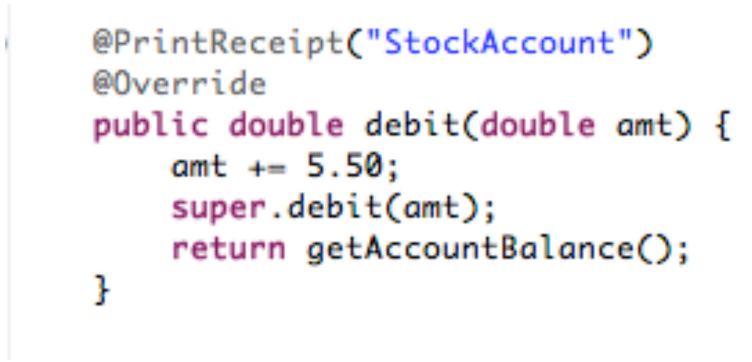
- If there is only one element defined in an annotation, name it value so it is easier to use



```
@Target(value = ElementType.METHOD)
@Retention(value = RetentionPolicy.RUNTIME)
public @interface PrintReceipt {
String value();

}
```

- We can just pass the value of the element without passing the key



```
@PrintReceipt("StockAccount")
@Override
public double debit(double amt) {
    amt += 5.50;
    super.debit(amt);
    return getAccountBalance();
}
```

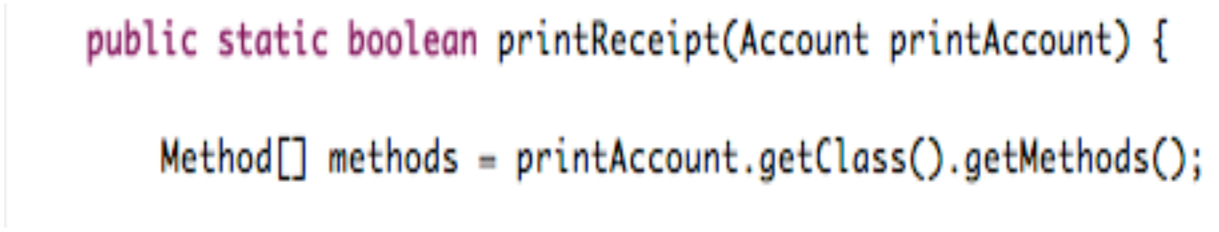
Custom Annotation Discovery

- Now that we created our annotation, we need to use reflection to find it
- The Java reflection api allows a a program to analyze a class and use the information
- @Retention value must be:

  RetentionPolicy.*RUNTIME*

Finding the Annotation

- Now that we created our annotation, we need to use reflection to find it
- First we find all the methods in the class.
- Reflection allows us to ask a class for a list of methods:

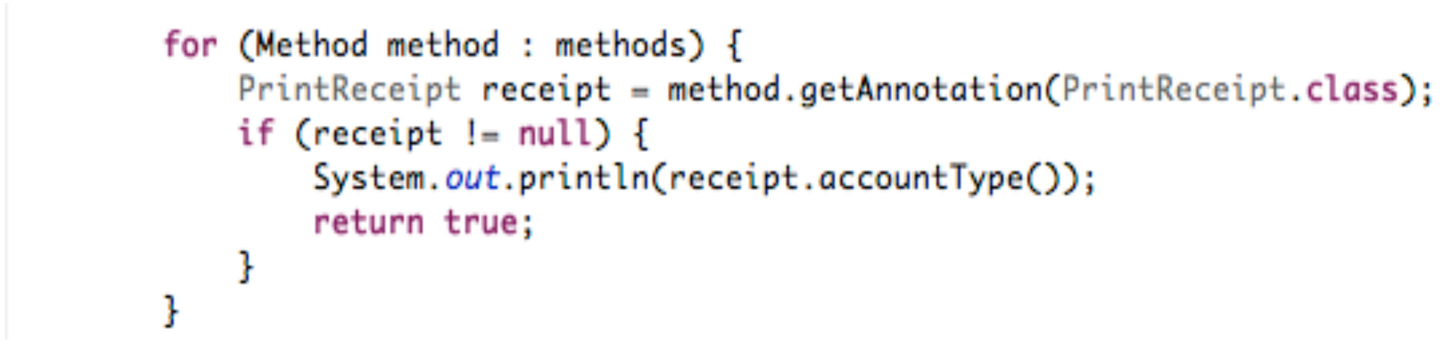


```
public static boolean printReceipt(Account printAccount) {

    Method[] methods = printAccount.getClass().getMethods();
```

Checking For Annotation Type

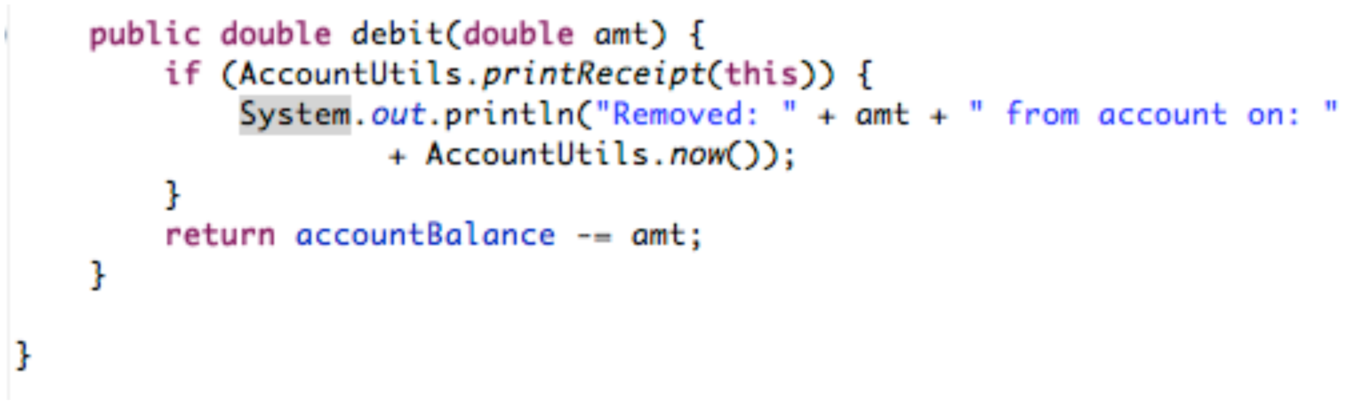
- Using the method we retrieved from the class we can look for our custom annotation
- If our annotation exist we can execute some code, in this case we print out the account type passed to the annotation and then return true



```
for (Method method : methods) {
    PrintReceipt receipt = method.getAnnotation(PrintReceipt.class);
    if (receipt != null) {
        System.out.println(receipt.accountType());
        return true;
    }
}
```

Invoking our Annotated Method

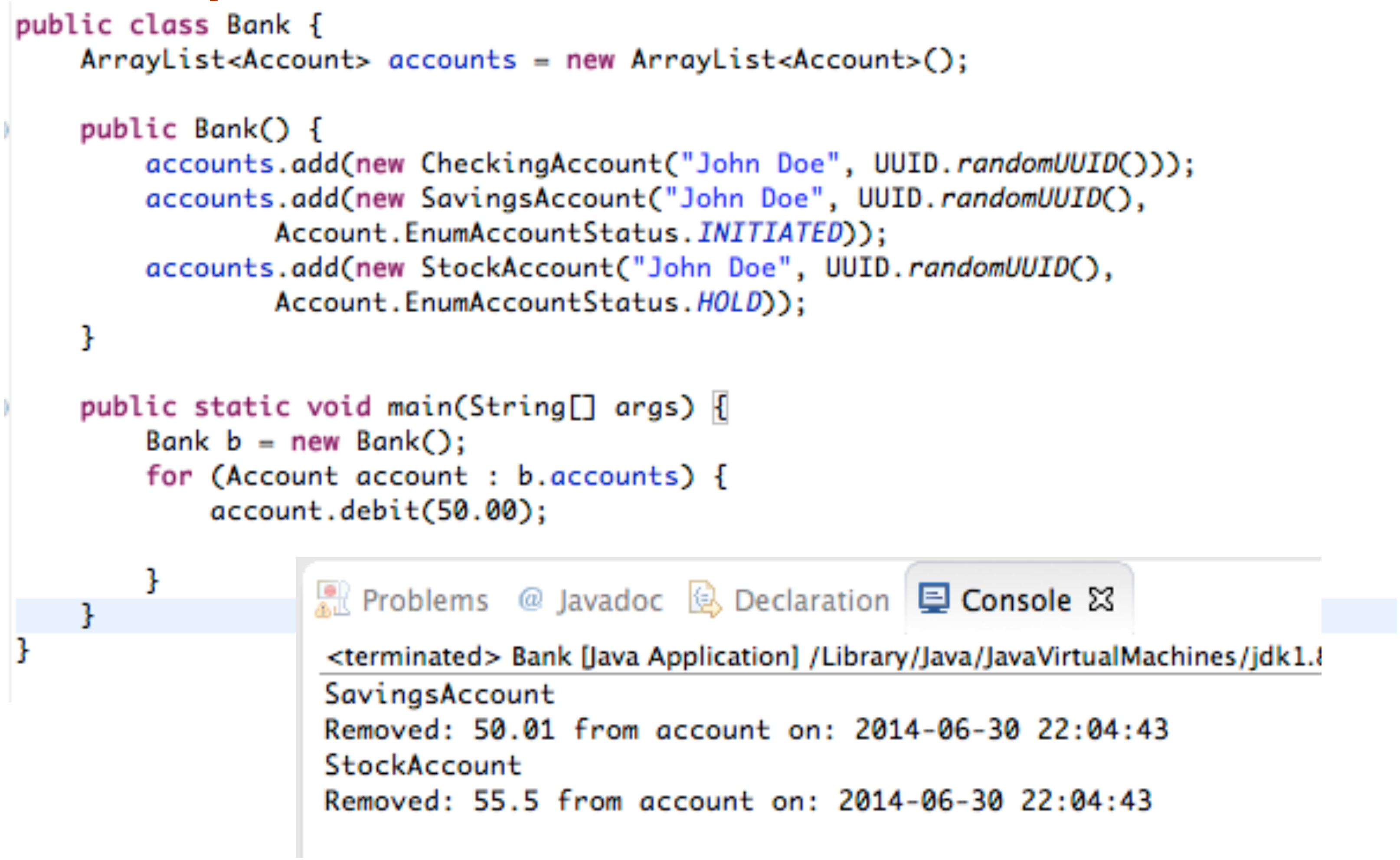
- Now that we have completed our steps:
  - Invoking method is annotated
  - We are using reflection to check for the annotation
- We can call our method and pass the calling object to check for the annotation



```
    public double debit(double amt) {
        if (AccountUtils.printReceipt(this)) {
            System.out.println("Removed: " + amt + " from account on: "
                    + AccountUtils.now());
        }
        return accountBalance -= amt;
    }

}
```

Only Annotated Methods Print a Receipt



```java
public class Bank {
    ArrayList<Account> accounts = new ArrayList<Account>();

    public Bank() {
        accounts.add(new CheckingAccount("John Doe", UUID.randomUUID()));
        accounts.add(new SavingsAccount("John Doe", UUID.randomUUID(),
                Account.EnumAccountStatus.INITIATED));
        accounts.add(new StockAccount("John Doe", UUID.randomUUID(),
                Account.EnumAccountStatus.HOLD));
    }

    public static void main(String[] args) {
        Bank b = new Bank();
        for (Account account : b.accounts) {
            account.debit(50.00);

        }
    }
}
```

```
<terminated> Bank [Java Application] /Library/Java/JavaVirtualMachines/jdk1.
SavingsAccount
Removed: 50.01 from account on: 2014-06-30 22:04:43
StockAccount
Removed: 55.5 from account on: 2014-06-30 22:04:43
```

Summary

- Custom annotations are created just like an interface
- Use custom annotations the same way as standard annotations
- If there is a single element in the custom annotation name it value
- Use the reflection package to read annotations