Annotations & Reflection

CSE260, Computer Science B: Honors
Stony Brook University

http://www.cs.stonybrook.edu/~cse260
Annotations and Reflection?

- Features of some languages
- Programmer conveniences
- Useful in checking inheritance
- Alternate development modes
Annotations

• Remember `@Override`
• Remember JUnit?
  • `@Before`, `@After`, `@Test`
  • `@` is Java’s notation for the start of an annotation
  • like `@author` for javadoc
• What are they?
  • metadata
  • provide data about a program
What are annotations used for?

- Information for the compiler
  - detect errors
  - suppress warnings.
- Compile-time and deployment-time processing
  - for IDEs and other tools
  - generate code, XML files, etc.
- Runtime processing
  - some annotations are used at runtime.
Annotations can have field names and data

```java
@Author(
    name = "Benjamin Franklin",
    date = "3/27/2003"
)
class MyClass()

@SuppressWarnings(value = "unchecked")
void myMethod() { ... }
```
Where can annotations be used?

- Declarations of classes, fields, methods, etc.
- Java SE 8 also has type annotations:
  - Class instance creation expression:
    ```java
    new @Interned MyObject();
    ```
  - Type cast:
    ```java
    myString = (@NonNull String) str;
    ```
  - implements clause:
    ```java
    class UnmodifiableList<T> implements @Readonly List<@Readonly T> {
    ...
    }
    ```
  - Thrown exception declaration:
    ```java
    void monitorTemperature() throws @Critical TemperatureException {
    ...
    }
    ```
Why do we care about annotations?

- Tools love to use them
  - JUnit
  - Javadoc
- Web-related Tools:
  - Java Persistence API (JPA)
    - describes the management of relational data in applications
- Application Servers
Annotations Look-Up

- Scattered in the Java API. Examples:

- Via cheat sheets:
Annotations

- Annotation Types Used by the Java Language
- The predefined annotation types defined in java.lang are @Deprecated, @Override, and @SuppressWarnings.
  - @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.

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

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

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

- `@SuppressWarnings` annotation tells the compiler to suppress specific warnings that it would otherwise generate.

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

- Define the annotation type:

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

- After the annotation type is defined, you can use annotations of that type:

```java
@ClassPreamble ( 
    author = "John Doe",
    date = "3/17/2002",
    currentRevision = 6,
    lastModified = "4/12/2004",
    lastModifiedBy = "Jane Doe",
    // Note array notation
    reviewers = {"Alice", "Bob", "Cindy"} 
  )

public class Generation3List extends List2 { 
    // class code goes here
}
```
Declaring an Annotation Type

- To make the information in @ClassPreamble appear in Javadoc-generated documentation, when you define the annotation:

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

    // Annotation element definitions

}
```

Declaring an Annotation Type
Reflection

- A powerful programming feature
  - requires the ability to examine or modify the runtime behavior of applications running in the Java virtual machine.
  - i.e. dynamically examine classes and objects

- Should be used only by developers who have a strong grasp of the fundamentals of the language.

- Can enable applications to perform operations which would otherwise be impossible.
Reflection

• Call methods at runtime that you didn’t know existed at compile time.

• Isn’t that polymorphism?
  • No, polymorphism uses inheritance and knows the overridden method signatures

• At runtime:
  • ask a Class what methods it has
  • call one of those methods
Reflection Uses

• Extensibility Features
  • dynamically use classes not known at compile time
  • plug-ins, add-ons, etc.
  • complete flexibility

• Class Browsers and Visual Development Environments
  • i.e. display class properties
    • think the visual debugger

• Debuggers and Test Tools

• Watch class values change
It all starts with the `Class` class:
- Every object in Java is a member of a class.
- How do we get an object’s `Class`?
  - `getClass()` method inherited from `Object`. Ex:
    ```java
    Class c = "Hello".getClass();
    ```
  - Using `Class.forName` and a string. Ex:
    ```java
    Class c2 = Class.forName("java.lang.String");
    ```
    - can throw `ClassNotFoundException`
  - Other methods:
    - `getSuperclass`
    - `getDeclaredClasses`
      - returns an array of `Class` object members declared by the class, but excludes inherited classes
        ```java
        Class cls = Class.forName("ClassDemo");
        Class[] classes = cls.getDeclaredClasses();
        ```
    - `getEnclosingClass`
      - Returns the outer class of an inner class (or null if none)
# The Class class has useful methods

## Class Methods for Locating Fields

<table>
<thead>
<tr>
<th>Class Method</th>
<th>List of members?</th>
<th>Inherited members?</th>
<th>Private members?</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDeclaredField()</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>getField()</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>getDeclaredFields()</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>getFields()</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

## Class Methods for Locating Methods

<table>
<thead>
<tr>
<th>Class Method</th>
<th>List of members?</th>
<th>Inherited members?</th>
<th>Private members?</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDeclaredMethod()</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>getMethod()</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>getDeclaredMethods()</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>getMethods()</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

## Class Methods for Locating Constructors

<table>
<thead>
<tr>
<th>Class Method</th>
<th>List of members?</th>
<th>Inherited members?</th>
<th>Private members?</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDeclaredConstructor()</td>
<td>no</td>
<td>N/A(^1)</td>
<td>yes</td>
</tr>
<tr>
<td>getConstructor()</td>
<td>no</td>
<td>N/A(^1)</td>
<td>no</td>
</tr>
<tr>
<td>getDeclaredConstructors()</td>
<td>yes</td>
<td>N/A(^1)</td>
<td>yes</td>
</tr>
<tr>
<td>getConstructors()</td>
<td>yes</td>
<td>N/A(^1)</td>
<td>no</td>
</tr>
</tbody>
</table>

\(^1\) N/A: Not Applicable
Fields

• Has a type and value
• Type is a **Class**
• Get/Set data via `get/set` methods
• Other useful classes
  • **Method**
  • **Constructor**
Drawbacks of Reflection

• Performance Overhead
  • dynamic type resolution is expensive
  • certain Java virtual machine optimizations skipped
  • should be avoided in hot spots

• Security Restrictions
  • requires a runtime permission which may not be present when running under a security manager.
    • can’t be used with Applets

• Exposure of Internals
  • allows code to perform operations that would be illegal in non-reflective code
    • accessing private fields and methods
  • can result in unexpected side-effects