CS5233
Components – Models and Engineering
(Komponententechnologien)
Master of Science (Informatik)

Anotations
Annotations

Annotations – Relevant JSRs

- **JSR 175**  
  Gilad Bracha: *A Metadata Facility for the Java Programming Language*  
  supported by Java 5

- **JSR 250**  
  Rajiv Mordani: *Common Annotations for the Java Platform*  
  supported by Java 6

- **JSR 269**  
  Joe Darcy: *Pluggable Annotation Processing API*  
  supported by Java 6

- **JSR 308**  
  Alex Buckley, Michael Ernst: *Annotations on Java Types*  
  supported by Java 7 deferred to Java 8 (or later)
Annotations

Annotations – JSR 175

*Meta-Information for programs placed into the source code – since Java 1.5*

- annotations are not part of the program logic
- they may be evaluated:
  - at compile time by the compiler
  - at build / deployment time by annotation processors
  - at runtime by the program using reflection

- Annotations may be
  - build-in annotations
  - user defined annotations

- Annotations are placed before the program element they annotate:
  - packages
  - classes, interfaces
  - fields, methods, constructors
  - parameters, variables

- Syntax
  
  ```java
  @<Name-of-annotation> [ <parameters> ]
  ```

```java
public class AClass extends BClass {
    @Deprecated
    void oldMethod() {
        // do something in an old
        // fashioned way
    }

    @Override
    void method() {
        // do something in a different
        // way
    }

    @SuppressWarnings("unchecked")
    void f() {
        List l = new ArrayList<Integer>();
    }
}
```

*a class with annotated methods*
Annotations

Annotating without Annotations

Pre Java-1.5 annotation techniques:

– marker interfaces
– modifiers
– structured comments

Ad-hoc mix of language elements and programming conventions

```java
import java.io.Serializable;

public class AClassInOldStile implements Serializable {
    volatile boolean b = false;

    /**
     * @deprecated
     */
    @Deprecated
    void oldMethod() {
        ... do something in an old fashioned way
    }
    ...
}
```
Annotations

Standard Annotations

predefined in java.lang

- @Deprecated
- @Override
- @SuppressWarnings
Annotations

Standard Annotations

predefined in java.lang

- @Deprecated
  - may annotate classes, interfaces, methods
  - discourages programmers to use the element
  - no advantage compared to the structured @deprecated-comment (annotation is a marker annotation, i.e. without parameters, so it can't explain neither reason nor degree of deprecation – don't use)
Annotations

Standard Annotations

predefined in java.lang

- @Override
  - annotates methods
  - indicates that the annotated method overrides a declaration in a supertype
  - may be used if the overridden method is abstract / defined in an interface (since 1.6)
  - should be used if the overridden method is not abstract
  - is checked by the compiler
  - may be used and checked by IDEs (e.g. Create warnings, enforce a certain interpretation of “may” and “should”)

Annotations

Standard Annotations

predefined in java.lang / Example @Override

class Base {
    int f() { return 0; }
    private int g() { return 0; }
}

class Derived extends Base {
    // missing @Override detected by the IDE
    // (if configured appropriately)
    int f() { return 1; }

    // spurious @Override, detected by the compiler
    @Override
    int g() { return 1; }
}

missing @Override detected by the IDE

Erroneous @Override detected by the compiler
Annotations

Standard Annotations

predefined in java.lang

- @SuppressWarnings( warning-category )
  @SuppressWarnings({ list of warning-categories })

  - annotates classes, methods, statements
  - indicates that the compiler should suppress within the annotated element all warnings in the warning categories or any of the warning categories.

  - Warning categories
    - Examples: "all", "deprecation", "unchecked", "serial", ...
    - General: all warning categories that may be set using javac -Xlint

      may be suppressed using the annotation (The IDE will handle this)

```java
class C {
  @SuppressWarnings({"unchecked", "unused"})
  private int g() {
    List l = new ArrayList();
    return 0;
  }
}
```
Annotations

Standard Annotations / JSR 250 Annotations

defined in package javax.annotation

– Java 6 comprises annotations defined in package javax.annotation
  ▪ @Generated
  ▪ @PostConstruct
  ▪ @PreDestroy
  ▪ @Ressource
  ▪ @Ressources

– intended usage
  ▪ Enterprise frameworks
  ▪ encode common concepts in unique ways across different frameworks

– javax.annotation annotations are defined in
  JSR-250: Common Annotations for the Java® Platform
Annotations

Standard Annotations / JSR 250 Annotations

defined in package `javax.annotation`

- `@Generated`
  - mark generated source code
  - may annotate any language element
  - not part of the class-code
  - must name the code generator that created the code
  - may contain date of creation and comment string

```java
@Generated("com.sun.xml.rpc.AProcessor")
public interface StockQuoteService extends java.rmi.Remote {
    this.context = context;
}
```

demote (from JSR 250)
Annotations

Standard Annotations / JSR 250 Annotations

defined in package `javax.annotation`

– `@PostConstruct`

- mark a non-static void-method that does not throw a checked exception
- usage:
  live-cycle related Action on a “container-component”
  marks begin-of-live action

JSR 250:

“The `PostConstruct` annotation is used on a method that needs to be executed:

- after dependency injection is done
- to perform any initialization.

This method MUST be invoked before the class is put into service.

This annotation MUST be supported on all classes that support dependency injection.

The method annotated with `PostConstruct` MUST be invoked even if the class does not request any resources to be injected.”
Annotations

Standard Annotations / JSR 250 Annotations

defined in package javax.annotation

– @PreDestroy

- mark a non-static void-method that does not throw a checked exception
- usage:
  live-cycle related Action on a “container-component”
  marks end-of-live action

JSR 250:

“The PreDestroy annotation is used on methods
  ➢ as a callback notification
  ➢ to signal that the instance is in the process of being removed by the container.

*The method annotated with PreDestroy is typically used to release resources that it has been holding.*”
Annotations

Standard Annotations / JSR 250 Annotations

defined in package `javax.annotation`

- `@Resource`
- `@Resources`
  - mark field, a method or a class
  - usage: a marked element will be provided via resource injection by the framework / container
Usage

- **Inversion of control (Hollywood principle)**
  
  Framework / container - instead of user provided classes - manage flow of control in an application.
  
  Application has to provide callable ("callback") classes that conform to given conventions.

- **Dependency injection a special case of inversion of control**
  
  Application entities depend on other entities. The framework / container resolves dependencies by “injecting” information into dependent entities.
  
  Simply: the framework links the application entities.
  
  This is may be viewed as an instance of inversion of control because not the application entity looks for what it needs (normal), but the framework provides it with what it needs (inverted).

- **Resource injection**
  
  Dependency injection where the dependence is on a resource.
Annotations

Defining Annotations

– “normal” application code should not define annotations
– they may be defined for use in frameworks or classes of applications
– Examples:

```java
@interface MyAnnotation {}

@MyAnnotation
class annotatedClass {···}
```

Marker annotation:
Annotation without elements

```java
@interface MyValuedAnno {
    String value() default "default value";
}

@MyValuedAnno
class annotatedClass1 {···}

@MyValuedAnno(value="special value 1")
class annotatedClass2 {···}

@MyValuedAnno("special value 2")
class annotatedClass3 {···}
```

Single element annotation:
Annotation with one element.
Element may have default value.

Annotations are defined using the keyword `@interface`.
Element definitions look like parameterless methods.
Default elements must be named value.
Annotations

Defining Annotations

– Examples cont.:

```java
@interface MyMultiValuedAnno {
    String val1() default "default value";
    int val2() default 42;
}

@MyMultiValuedAnno
class annotatedClass1 {···}

@MyMultiValuedAnno(val1="Hello", val2=11)
class annotatedClass2 {···}
```

Multi-element annotation:
Annotation with several elements

```java
@interface MyArrayValuedAnno {
    String[] value() default {"default String", "42"};
}

@MyArrayValuedAnno
class annotatedClass1 {···}

@MyArrayValuedAnno({"Hello", "11"})
class annotatedClass2 {···}
```

Single-element annotation with array value:
Annotation with array elements

Annotations may have elements only of restricted types: String, Class, enum, Annotation, one-dimensional arrays of these types.
All values must be compile-time computable. The value null is not allowed.
Annotations

Annotations : java.lang.annotation

- Language features + library support
- Package java.lang.annotation provides library support for annotations:

  - **Interface** java.lang.annotation.Annotation  
    
    *all annotations are extensions of this interface, however they may not be defined as extensions of this interface*

    ```java
    interface MyAnno  
        extends java.lang.annotation.Annotation { }

    @MyAnno
    class Aclass { }
    ```

    Type mismatch: cannot convert from MyAnno to Annotation

    ```java
    package packP;
    @interface MyAnno { }
    public class Main { 
        public static void main(String[] args) { 
            System.out.println(MyAnno.class);
            for (Class<?> c : MyAnno.class.getInterfaces()) { 
                System.out.println(c);
            }
        }
    }
    ```

    interface packP.MyAnno  
    interface java.lang.annotation.Annotation
Annotations

Annotations : java.lang.annotation

Package java.lang.annotation provides library support for annotations:

- **Annotation** java.lang.annotation.**Target**
  meta-annotation to designate annotate-able elements

- **Enum** java.lang.annotation.**ElementType**
  enumeration of all annotate-able elements:
  ElementType.**TYPE**, ElementType.**FIELD**, ElementType.**METHOD**, 
  ElementType.**PARAMETER**, ElementType.**CONSTRUCTOR**,
  ElementType.**LOCAL_VARIABLE**, ElementType.**ANNOTATION_TYPE**,
  ElementType.**PACKAGE**

```java
import java.lang.annotation.**ElementType**;
import java.lang.annotation.**Target**;

@**Target**(ElementType.**METHOD**)
@interface **MyMethodAnno** { }

class AnnotatedClass { 
  @**MyMethodAnno**
  void f() { 
    System.**out**.println("I am an annotated method");
  }
}
```

Example
Annotations

Annotations: java.lang.annotation

Package java.lang.annotation provides library support for annotations:

- **Annotation** java.lang.annotation.Retention
  Meta-annotation to designate a retention policy for the annotated annotation

- **Enum** java.lang.annotation.RetentionPolicy
  Enumeration of all retention policies:
  
  - RetentionPolicy.CLASS
    the annotation should be placed into the class code and thus available to the JVM. It is not available at runtime. (Default value)
  
  - RetentionPolicy.RUNTIME
    the annotation should be placed into the class code and retained by the JVM and thus be available at runtime
  
  - RetentionPolicy.SOURCE
    the annotation should not be placed into the class code and thus be available neither to the JVM, nor to the running code.
Annotations

Annotation java.lang.annotation.Retention / Examples

```java
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.SOURCE)
@interface MyAnno {}

@MyAnno
class AnnotatedClass {}
```

definitions of the classes and annotations

```java
package packP;
import java.lang.annotation.Annotation;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
@interface MyRTAnno { String value() default "Hugo"; }

@Retention(RetentionPolicy.CLASS)
@interface MyClassAnno { String value() default "Egon"; }

@MyRTAnno
@MyClassAnno
class AClass {}

public class Main {
    public static void main(String[] args) {
        for (Annotation a : Aclass.class.getAnnotations())
            System.out.println("Annotation of AClass: @packP.MyRTAnno(value=Hugo)" + a);
        for (Annotation a : MyRTAnno.class.getAnnotations())
            System.out.println("Annotation of MyRTAnno: @java.lang.annotation.Retention(value=RUNTIME)" + a);
        for (Annotation a : MyClassAnno.class.getAnnotations())
            System.out.println("Annotation of MyClassAnno: @java.lang.annotation.Retention(value=CLASS)" + a);
    }
}
```
Annotations : java.lang.annotation

Package java.lang.annotation provides library support for annotations:

- **Annotation** java.lang.annotation.Documented
  
  *meta-annotation to designate annotations that are to be documented by javadoc or related tools*

```
@Documented
@interface MyDocumentedAnno {}

@MyDocumentedAnno
class AnnotatedClass {...}
```

Example
## Annotations

Package `java.lang.annotation` provides library support for annotations:

- **Annotation** `java.lang.annotation.Inherited` meta-annotation to denote annotations that are inherited (only for classes)

```java
package packP;
import java.lang.annotation.Annotation;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
@Inherited
@interface MyAnno {}

@MyAnno
class BaseClass {}

class DerivedClass extends BaseClass {}

public class Main {
    public static void main(String[] args) {
        for (Annotation a : BaseClass.class.getAnnotations()) {
            System.out.println("Annotation of BaseClass: " + a);
        }
        for (Annotation a : DerivedClass.class.getAnnotations()) {
            System.out.println("Annotation of Derived: " + a);
        }
    }
}
```

**Example 1**
Annotations

Annotation java.lang.annotation.Inherited / Example 2

```java
package packP;

import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Inherited
@interface Car {
    String manufacturer() default "Opel";
}

@Car
class Astra {
    ...
}

@Car(manufacturer="BMW")
class BMW {
    ...
}

class Mini extends BMW {
    ...
}

public class Main {
    public static void main(String[] args) {
        Astra astra = new Astra(1992, "TLE");
        Mini mini = new Mini(2008, "Carla");

        for (Annotation a : astra.getClass().getAnnotations()) {
            System.out.println("Annotation of astra: " + a);
        }

        for (Annotation a : mini.getClass().getAnnotations()) {
            System.out.println("Annotation of mini: " + a);
        }
    }
}
```

Example 2
Annotations

Accessing Annotations

Annotations may be accessed:

– **within the source code**
  - by code processing tools
  - by the compiler

– **within the class code**
  - by the JVM (if retention at least = CLASS)
  - by a running program (if Retention = RUNTIME)
    - the application itself
    - a framework / container
Accessing Annotations at Runtime

Annotations may be accessed at runtime via reflection.

Interface `java.lang.reflect.AnnotatedElement`

*represents an annotated element of the program currently running in this VM*

- is implemented by all annotate-able elements
  (with a special treatment of `packages`)

- provides 4 methods to access annotation-information at runtime
  
  - `<T extends Annotation> T getAnnotation(Class<T> annotationClass)`
    Returns this element's annotation for the specified type if such an annotation is present, else null.
  
  - `Annotation[] getAnnotations()`
    Returns all annotations present on this element.
  
  - `Annotation[] getDeclaredAnnotations()`
    Returns all annotations that are directly present on this element. (Inherited Anno. ignored)
  
  - `boolean isAnnotationPresent(Class<? extends Annotation> annotationClass)`
    Returns true if an annotation for the specified type is present on this element else false.
Accessing annotations at runtime / Example

```java
package annoP;

import java.lang.annotation.*;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@interface MyAnno {
  String base() default "NoName";
}

@MyAnno(base="Base")
class Base {
}

class Derived extends Base {
}

public class AccessAno {
  public static void main(String[] args) {
    System.out.println("Get @MyAnno = " + Derived.class.getAnnotation(MyAnno.class));

    MyAnno myAnno = (MyAnno) Derived.class.getAnnotation(MyAnno.class);
    String basename = myAnno.base();
    System.out.println("base name of Derived = " + basename);

    System.out.println("Annotations: ");
    for (Annotation anno: Derived.class.getAnnotations()) {
      System.out.println("\t" + anno);
    }
    System.out.println("Declared Annotations: ");
    for (Annotation anno: Derived.class.getDeclaredAnnotations()) {
      System.out.println("\t" + anno);
    }
  }
}
```

Get @MyAnno = @annoP.MyAnno(base=Base)
base name of Derived = Base
Annotations:
  @annoP.MyAnno(base=Base)
Declared Annotations:
 Annotations

Type Annotations / JSR 308 Annotations

introduced with Java 7, specified in JSR 308

- JSR 175 / JSR 250 Annotation Syntax:
  class-, method-, field-, variable-declarations may be annotated
- JSR 308
  extension of the annotation syntax:
  also type usage may be annotated
- Example

```java
public static void main(String[] args) {
    List<String> strings = new LinkedList<String>();
    strings.add("Hello");
    strings.add(null);
    f(strings);
}
```

External tool should detect this violation!
Compile-time Annotation Processing

Accessing / Processing Annotations in Sources

– **(Java 5) APT Annotation Processing Tool**
  handle annotations in a source file while the code is compiled
  consists of
  - a tool: `apt`
  - a mirror API: package `sun.com.mirror`
    introduced with Java 5, *deprecated* with Java 6

– **JSR 269 Pluggable Annotation Processing API**
  handle annotations via the compiler API introduced with Java 6.
Compile-time Annotation Processing

Pluggable Annotation Processing

- **Annotations**
  - are processed at **compile time** by the **compiler** (or other tools)
  - processing may result in: **compiler messages** (warnings, errors), **new source code, new byte code** ......

- **Annotations are processed** by
  
  Annotation processors
  
  that are passed to the **compiler**
  
  and **invoked** by the compiler (thus processors are **compiler plugins**)

- **Processor activation**
  
  candidate processors are set:
  
  - using a **compiler option**
    
    `javac -processor MyAnnoProcessorClass MyAnnotatedClass.java`
  
  - using the **service lookup facility**
    
    e.g. through provider-configuration files named
    
    META-INF/services/javax.annotation.processing.Processor
    
    that contains the names of all processors (each in a line)
  
  - using a **search path**
    
    defined by option **-processorpath** if present or else equal to classpath if not defined
Essential steps

- Define annotations that are to be processed
- Create a processor: typically by extending `javax.annotation.processing.AbstractProcessor`
- Create source code that contains annotations that are to be processed
- Compile source code with processor-aware compiler
Compile-time Annotation Processing

Example / define an annotation that is to be processed

```java
package pack269;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface MyAnnotation {
}
```
Compile-time Annotation Processing

Example / define an annotation processor (1)

```java
package pack269;
import java.util.Set;
import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.RoundEnvironment;
import javax.lang.model.element.TypeElement;
import javax.annotation.processing.SupportedAnnotationTypes;
import javax.annotation.processing.SupportedSourceVersion;
import javax.lang.model.SourceVersion;

@SupportedAnnotationTypes("pack269.MyAnnotation")
@SupportedSourceVersion(SourceVersion.RELEASE_7)
public class MyProcessor extends AbstractProcessor {

    @Override
    public boolean process(Set<? extends TypeElement> annotations,
                            RoundEnvironment roundEnv) {
        return false;
    }
}
```

This initial version doesn’t do anything.

@SupportedAnnotationTypes states which annotation(s) will be processed (“*”, package names are possible)
@SupportedSourceVersion states which version of source files are processed by this processor
return value
true: this processors “claims” the annotations, subsequent processors will not deal with them, false: subsequent processors may be asked to process them.
annotations
annotation types to be processed
roundEnv contains information about the current processing round
Example / First annotation processor

```java
package pack269;
import ... ...

@SupportedAnnotationTypes("pack269.MyAnnotation")
@SupportedSourceVersion(SourceVersion.RELEASE_7)
public class MyProcessor extends AbstractProcessor {

    @Override
    public boolean process(Set<? extends TypeElement> annotations,
                            RoundEnvironment roundEnv) {

        for (TypeElement typeElement: annotations) {
            this.processingEnv.getMessager().printMessage(Diagnostic.Kind.NOTE,
                                                        "processing annotations=" + annotations + ", " +
                                                        "roundEnv=" + roundEnv + ", " +
                                                        "typeElement=" + typeElement);
        }
        return false;
    }
}
```

```
package aPackage;
import pack269.MyAnnotation;
@MyAnnotation
public class AClass {}
```

`j`dk1.7.0/bin/javacc pack269/*.java
jk1.7.0/bin/javacc -processor pack269.MyProcessor aPackage/AClass.java

Note: processing annotations=[pack269.MyAnnotation],
roundEnv=[errorRaised=false, rootElements=[aPackage.AClass],
processingOver=false],
typeElement=pack269.MyAnnotation
Compile-time Annotation Processing

```java
this.processingEnv.getMessage() // access to the processing environment – i.e. the compiler – and its message facility

annotations
   a set of TypeElements: represent the annotations to be processed

roundEnv
   from the round environment we can get the class being compiled as rootElement:
   roundEnv.getRootElements()
```
Compile-time Annotation Processing

Example / Find annotated elements

```java
@SupportedAnnotationTypes("pack269.MyAnnotation")
@SupportedSourceVersion(SourceVersion.RELEASE_7)
public class MyProcessor extends AbstractProcessor {
    @Override
    public boolean process(Set<? extends TypeElement> annotations,
                            RoundEnvironment roundEnv) {
        for (TypeElement typeElement : annotations) {
            for (Element element : roundEnv.getElementsAnnotatedWith(typeElement)) {
                this.processingEnv.getMessager().printMessage(Diagnostic.Kind.WARNING,
                                                               "Annotated class found ", element);
            }
        }
        return false;
    }
}
```

```java
package aPackage;
import pack269.MyAnnotation;
@MyAnnotation
public class AClass {
    class InnerCass {}
}
```

```
javac -processor pack269.MyProcessor aPackage/AClass.java
aPackage/AClass.java:6: warning: Annotated class found
aPackage/AClass.java:6:  ^

aPackage/AClass.java:8: warning: Annotated class found
    class InnerCass {
    ^
```

Element / TypeElement

`javax.lang.model.element.Element` models a static program element

`javax.lang.model.element.TypeElement` models a class or interface

```java
package aPackage;
import pack269.MyAnnotation;
@MyAnnotation
public class AClass {
    @MyAnnotation
    class InnerClass {}
}
```

This code is part of a Processor, i.e. it is "meta-code"!

`typeElement` iterates over the types within the processed code, among them MyAnnotation.

`element` iterates over all static program elements of the processed code, among them are AClass and InnerClass.


**Example / Generate (and compile!) source code (1)**

```java
package pack269;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface GenAnno {}
```

Define Annotation for Classes that should trigger user defined code generation at compile time.

```java
package aPackage;
import pack269.GenAnno;
@GenAnno
public class GClass {}
```

This class should trigger code generation.

```java
package blub;
public class Blub {
    public void sayHello () {
        System.out.println("Hello World");
    }
}
```

The code to be generated.
Compile-time Annotation Processing

Example / Generate (and compile!) source code (2)

```java
package pack269;

import java.io.IOException;
import java.io.Writer;
import java.util.Set;
import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.Filer;
import javax.annotation.processing.Messager;
import javax.annotation.processing.RoundEnvironment;
import javax.annotation.processing.SupportedAnnotationTypes;
import javax.annotation.processing.SupportedSourceVersion;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.Element;
import javax.lang.model.element.TypeElement;
import javax.tools.Diagnostic;
import javax.tools.FileObject;

@SupportedAnnotationTypes("pack269.GenAnno")
@SupportedSourceVersion(SourceVersion.RELEASE_7)
public class GenAnnoProcessor extends AbstractProcessor {

    ... Next Slide ...
}
```
Example / Generate (and compile!) source code (3)

```java
@Override
public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {

    Filer filer = this.processingEnv.getFiler();
    Messager messager = this.processingEnv.getMessager();

    for (TypeElement typeElement : annotations) {
        for (Element element : roundEnv.getElementsAnnotatedWith(typeElement)) {
            messager.printMessage(Diagnostic.Kind.NOTE, "Annotated class found ", element);
            try {
                FileObject file = filer.createSourceFile("blub.Blub");
                Writer writer = file.openWriter();
                writer.write("package blub; 
" + "public class Blub" + " { 
" + "public void sayHello () {
" + "System.out.println(" + "Hello World" + ");" + "
" + "}");
                writer.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
    return false;
}
```
Example / Generate (and compile!) source code (4)

It is even possible to use the generated code in the class that triggers generation!

```
package aPackage;

import pack269.GenAnno;

@GenAnno
public class GClass {
    public static void main(String[] args) {
        blub.Blub b = new blub.Blub();
        b.sayHello();
    }
}
```

```
javac pack269/*.java
javac -processor pack269.GenAnnoProcessor aPackage/GClass.java
aPackage/GClass.java:6: Note: Annotated class found
public class GClass {
^  
java aPackage.GClass
Hello World
```
Compile-time Annotation Processing

javafx.lang.model

Packages and classes used to model (static) language elements of Java.

Elements

javafx.lang.model.element.*Element*

Elements model the static structure of the program, i.e. packages, classes, methods and variables.

Types

javafx.lang.model.type.*Type*

Mirror classes for types: Types model the statically defined types and their constraints in a program, i.e. types, generic type parameters, generic type wildcards.

TypeMirror

javafx.lang.model.type.TypeMirror

Type mirrors are the basic interface of all type mirror classes.

Compile-time Annotation Processing

Example / Work with type mirrors

```java
@SupportedAnnotationTypes("pack269.MyClassAnno")
@SupportedSourceVersion(SourceVersion.RELEASE_6)
public class ClassCheckProcessor extends AbstractProcessor {
    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {
        Messager messager = this.processingEnv.getMessager();

        for (TypeElement typeElement: annotations) {
            for (Element element : roundEnv.getElementsAnnotatedWith(typeElement)) {
                // get the type defined by this element
                TypeMirror typeMirror = element.asType();
                messager.printMessage(Diagnostic.Kind.NOTE, "Annotated element has type: " + typeMirror, element);

                // check whether it is a declared type (class or interface)
                if (typeMirror instanceof DeclaredType) {
                    DeclaredType declaredType = (DeclaredType) typeMirror;
                    String argInfo = "";
                    for (TypeMirror typeArgument: declaredType.getTypeArguments()) {
                        argInfo = argInfo + " " + typeArgument.toString();
                    }
                    messager.printMessage(Diagnostic.Kind.NOTE, "Annotated class: " + typeMirror + " Args: " + argInfo, element);
                }
            }
        }
        return false;
    }
}
```

This processor prints some information about annotated elements of compiled classes when passed to the compiler.
Compile-time Annotation Processing

Example / Compile-time check using user code (1)
Task: implement compile-time checking of annotated classes

```java
package checked_processes;
import processChecker.Process;
public class CheckMe {
    @Process class P1<T> implements Runnable {
        @Override
        public void run() {  }
    }
    @Process class P2 implements Runnable {
        @Override
        public void run() {  }
    }
    @Process class P3<T> {
        public void run() {  }
    }
    @Process interface X<T> {
        int f();
    }
}
```

@Process should be restricted to:
- classes
- that have a type parameter and
- implement Runnable
The compiler should check this restriction.

```java
package processChecker;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface Process {
}
```

Error: Not a class
Error: No type parameter
Error: Does not implement Runnable
Compile-time Annotation Processing

Example / Compile-time check using user code (2a) : Write Processor

```java
package processChecker;

import java.util.List;
import java.util.Set;

import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.RoundEnvironment;
import javax.annotation.processing.SupportedAnnotationTypes;
import javax.annotation.processing.SupportedSourceVersion;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.Element;
import javax.lang.model.element.TypeElement;
import javax.lang.model.type.DeclaredType;
import javax.lang.model.type.TypeMirror;
import javax.tools.Diagnostic;

@SupportedAnnotationTypes("processChecker.Process")
@SupportedSourceVersion(SourceVersion.RELEASE_6)

public class ProcessDefinitionCheckProcessor extends AbstractProcessor {
    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {
        . . .
    }
}
```
Example / Compile-time check using user code (2b) : Write Processor

```java
for (TypeElement typeElement : annotations) {
    for (Element element : roundEnv.getElementsAnnotatedWith(typeElement)) {
        //this.processingEnv.getMessager().printMessage(
        //    Diagnostic.Kind.NOTE, "Process found", element);
        TypeMirror typeMirror = element.asType();
        if (!(typeMirror instanceof DeclaredType)) {
            this.processingEnv.getMessager().printMessage(
                Diagnostic.Kind.ERROR, "Processes have to be Classes", element);
        } else {
            DeclaredType declaredType = (DeclaredType) typeMirror;
            if (!declaredType.asElement().getKind().isClass()) {
                this.processingEnv.getMessager().printMessage(
                    Diagnostic.Kind.ERROR, "Processes have to be Classes", element);
            } else if (declaredType.getTypeArguments().size() != 1) {
                this.processingEnv.getMessager().printMessage(
                    Diagnostic.Kind.ERROR, "Processes have to have a type parameter", element);
            } else {
                ...next slide...
            }
        }
    }
    return true;
}
```
Example / Compile-time check using user code (2c) : Write Processor

TypeElement elemAsTypeElem = (TypeElement) element;
boolean isRunnable = false;
List<? extends TypeMirror> ifs = elemAsTypeElem.getInterfaces();

for (TypeMirror tm: ifs) {
    if ( ((TypeElement)((DeclaredType)tm)
            .asElement())
            .getQualifiedName()
            .contentEquals("java.lang.Runnable")) {
        isRunnable = true;
    }
}

if (!isRunnable) {
    this.processingEnv.getMessager().printMessage(
        Diagnostic.Kind.ERROR, "Processes must be Runnable", element);
}
Compile-time Annotation Processing

Example / Compile-time check using user code (3) : Test

`:~/EclipseWorkspaces/Komponenten-11/Test_269/src> javac //
   -processorpath /home/thomas/EclipseWorkspaces/Komponenten-11/Test_269/bin //
   -processor processChecker.ProcessDefinitionCheckProcessor checked_processes/CheckMe.java

checked_processes/CheckMe.java:14: Processes have to have a type parameter
   class P2 implements Runnable {
^  
checked_processes/CheckMe.java:20: Processes must be Runnable
   class P3<T> {
^  
checked_processes/CheckMe.java:26: Processes have to be Classes
   interface X<T> {
^  
3 errors
Compile-time Annotation Processing

Example / Compile-time check using user code (4a) : Use with Eclipse

If we want to use the checker when developing “processes” with eclipse we have to define the processor as a (Java-) service.

First move the annotation and the processor to a new project and export it as jar.

Create new project and move package processChecker to this project.
Of course CheckMe.java now is erroneous because the annotation @Process is unknown.

Export it as jar.

`thomas@karlchen:~/tmp> ls processchecker.jar`
`processchecker.jar`
Example / Compile-time check using user code (4b) : Use with Eclipse

Extract jar and add files and folder according to the java service specification (see `java.util.ServiceLoader<S>` ) then jar it again.

```
~/tmp/processchecker>
jar -cvf ../processchecker.jar
META-INF/services/javax.annotation.processing.Processor
processChecker/Process.class
processChecker/ProcessDefinitionCheckProcessor.class
```
Compile-time Annotation Processing

Example / Compile-time check using user code (4c) : Use with Eclipse

Add jar as external lib to the project containing classes that are to be checked.
Compile-time Annotation Processing

Example / Compile-time check using user code (4d) : Use with Eclipse

Configure project properties to activate annotation processing

![Factory Path](attachment:image.png)

- Resource
  - Builders
  - Java Build Path
- Java Code Style
- Java Compiler
  - Annotation Processing
    - Factory Path
- Java Editor
  - Javadoc Location
  - Project Facets
  - Project References
  - Run/Debug Settings
- Task Repository
  - Task Tags
- Validation
Compile-time Annotation Processing

Example / Compile-time check using user code (4e) : Use with Eclipse

Eclipses now compiles with your annotation processing plug-in activated!