Adding Type Metadata with Annotations



Jim Wilson
MOBILE SOLUTIONS DEVELOPER & ARCHITECT
@hedgehogjim blog.jwhh.com

Overview



The need for type metadata

Using annotations

Declaring custom annotations

Accessing annotations

Annotation target and retention

Simplifying element setting

Annotation element types

Class-cross reference



The Need to Express Context and Intent

Programs do not stand alone

- They fit into a larger picture
- Incorporates developer's assumption
 - About the type system
 - About the toolset
 - About the execution environment



The Need to Express Context and Intent

Programs incorporate context and intent

- Type system solves much of this issue
- But standard type system isn't enough



Type Expressing Intent

```
public class TxWorker implements Runnable {
  protected BankAccount account;
  protected char txType;
  protected int amt;
  public TxWorker(BankAccount account, char txType, int amt) { . . . }
  public void run() {
    if (txType == 'w')
      account.withdrawal(amt);
    else if (txType == 'd')
      account.deposit(amt);
```

Type Unable to Express Intent

```
Implied extension
                                       of Object class
public class BankAccount
  private final String id;
  private int balance = 0;
  public BankAccount(String id, int balance) {...}
  public String getId() {...}
  public synchronized int getBalance() {...}
  public synchronized void deposit(int amount) {...}
  public synchronized void withdrawal(int amount) {...}
  public String toStrong() {
    return String.format(getId() + ": " + getBalance());
```

The Need to Express Context and Intent

We need a way to extend the type system

- We often try to supplement manually
 - Code comments
 - Documentation
 - Just isn't enough
- We need a structured solution
 - Allows tools to act on context & intent



Annotations

- Special types that act as metadata
 - Applied to a specific target
- Have no direct impact on target
 - Do not change target's behavior

Annotations must be interpreted

- Tools
- Execution environments
- Any program



Annotations in code

- Name always preceded by @ when used
- Placed directly before target
- Allowable targets vary with annotation



Expressing Intent with Override Annotation

```
public class BankAccount {
 private final String id;
 private int balance = 0;
 public BankAccount(String id, int balance) {...}
 public String getId() {...}
 public synchronized int getBalance() {...}
 public synchronized void deposit(int amount) {...}
 public synchronized void withdrawal(int amount) {...}
                                         Compiler looks for
 @Override
                                       methods marked with
 public String toStrong() {
                                          this annotation
    return String.format(getId() + ": " + getBalance());
                                         Verifies there is a method
                                       with matching signature that
                                            can be overridden
```

Annotations and the core Java platform

- Types to create annotations
- Has only a few annotations

Widely used by other tools/environments

- Java EE
- XML processors such as JAXP
- Your code??



Common Java core platform annotations

- Most affect compilation
 - Override
 - Deprecated
 - SuppressWarnings



```
class Doer {
    @Deprecated
    void doItThisWay() { ... }
    void doItThisNewWay() { ... }
}
```

```
@SuppressWarnings("deprecation")
class MyWorker {
 @SuppressWarnings("deprecation")
 void doSomeWork() {
    Doer d = new Doer();
    d.doItThisWay();
  @SuppressWarnings("deprecation")
  void doDoubleWork() {
    Doer d1 = new Doer();
    Doer d2 = new Doer();
    d1.doItThisWay();
    d2.doItThisWay();
```

You can create custom annotations

- Acts as custom meta data
- Provides same capabilities as built-in



Flexible work dispatch system

- Executes worker classes against targets

Worker type requirements

- Has a no-argument constructor
- Implements TaskWorker interface
 - Our custom interface

Worker threading requirements

- Can create own thread
- Can be run on app's thread pool
- Preference indicated with annotation



Annotations are a special kind of interface

- Usage is much more restricted
 - Can't be explicitly implemented
- Implicitly extend Annotation interface
- Interface behavior not initially apparent



Declaring annotations similar to interfaces

- Use interface keyword
 - Preceded by an @ symbol
- Declarations allow same modifiers
- Declarations allowed in same places

```
public @interface WorkHandler {
}
```



Annotations can optionally have elements

- Associate values within annotation
- Declared as methods
- Setting is similar to fields

```
public @interface WorkHandler {
  boolean useThreadPool();
}
```



```
@WorkHandler(useThreadPool = false)
public class AccountWorker implements Runnable, TaskWorker {
 BankAccount ba;
 public void setTarget(object target) { ... }
 public void doWork() {
    Thread t = new Thread(
      HighVolumeAccount.class.isInstance(ba) ? (HighVolumeAccount)ba : this);
    t.start();
 public void run() {...}
```

Accessing

Annotations available through reflection

- Call getAnnotation on type/member
 - Accepts Class of annotation
- Returns reference to annotation interface
 - Null if does not have annotation of requested type



Accessing Annotations

```
void startWork(String workerTypeName, Object workerTarget) throws Exception {
 Class<?> workerType = Class.forName(workerTypeName);
  TaskWorker worker = (TaskWorker) workerType.newInstance();
  worker.setTarget(workerTarget);
      ExecutorService pool = Executors.newFixedThreadPool(5);
   worker.doWork();
```

Accessing Annotations

```
void startWork(String workerTypeName, Object workerTarget) throws Exception {
  Class<?> workerType = Class.forName(workerTypeName);
  TaskWorker worker = (TaskWorker) workerType.newInstance();
  worker.setTarget(workerTarget);
  WorkHandler wh = workerType.getAnnotation
  if(wh == null)
    // throw exception 🗲
  if(wh.useThreadPool()
    pool.submit(
      public void run() {
        worker.doWork();
  else
    worker.doWork();
```

Annotation Retention

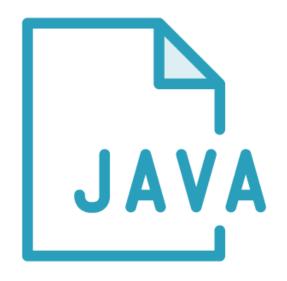
Annotations can specify availability

- Part of annotation declaration
- Indicated with Retention annotation
 - Accepts RetentionPolicy value

```
@Retention( . . . )
public @interface WorkHandler { . . . }
```

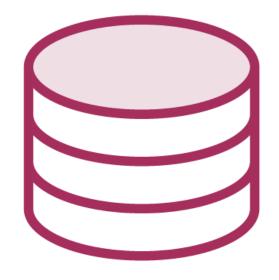


RetentionPolicy Values



SOURCE

Exist only in source file Discarded by compiler



CLASS

Compiled into class file Discarded by runtime



RUNTIME

Loaded into runtime
Accessible with reflection



Annotation Retention

```
@Retention(RetentionPolicy.RUNTIME)
public @interface WorkHandler {
  boolean useThreadPool();
}
```



Annotation Retention

```
void startWork(String workerTypeName, Object workerTarget) throws Exception {
 Class<?> workerType = Class.forName(workerTypeName);
  TaskWorker worker = (TaskWorker) workerType.newInstance();
  worker.setTarget(workerTarget);
  WorkHandler wh = workerType.getAnnotation(WorkHandler.class);
  if(wh == null)
    // throw exception
  if(wh.useThreadPool())
    pool.submit(new Runnable() {
      public void run() {
        doWork();
  else
    worker.doWork();
```

Annotation Target

```
@WorkHandler(useThreadPool = false)
public class AccountWorker implements Runnable, TaskWorker {
  @WorkHandler(useThreadPool = false)
  BankAccount ba;
  @WorkHandler(useThreadPool = false)
  public void setTarget(object target) { ... }
  public void doWork() {
    @WorkHandler(useThreadPool = false)
    Thread t = new Thread(
      HighVolumeAccount.class.isInstance(ba) ? (HighVolumeAccount)ba : this);
    t.start();
  public void run() {...}
```

Annotation Target

Annotations can narrow allowable targets

- Part of annotation declaration
- Indicated with Target annotation
 - Accepts ElementType value
- Can support multiple targets
 - Use array notation

Target(ElementType.CONSTRUCTOR)

```
Target( { ElementType.TYPE, ElementType.METHOD } )
```



Annotation Target

```
@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)
public @interface WorkHandler {
  boolean useThreadPool();
}
```



Annotation Target

```
@WorkHandler(useThreadPool = false)
public class AccountWorker implements Runnable, TaskWorker {
  BankAccount ba;
  public void setTarget(object target) { ... }
  public void doWork() {
    Thread t = new Thread(
      HighVolumeAccount.class.isInstance(ba) ? (HighVolumeAccount)ba : this);
    t.start();
  public void run() {...}
```

Simplifying Element Setting

Elements can be setup to simplify setting

- Handle common cases
- Element default values
- Element assignment shorthand



Element Default Values

Elements can be declared with a default

- Use default keyword
- Can still explicitly set if desired



Element Default Values

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface WorkHandler {
  boolean useThreadPool() default true;
}
```

```
@WorkHandler
public class AccountWorker implements Runnable, TaskWorker { ... }
```



Element Assignment Shorthand

Can exclude element name when setting

- Must be setting only one element
- Element name must be value



Element Assignment Shorthand

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface WorkHandler {
  boolean useThreadPool() default true;
}
```

```
@WorkHandler(false)
public class AccountWorker implements Runnable, TaskWorker { ... }
```



Valid Annotation Element Types

String Primitive type Enum **Annotation** Class<?>

Can also be an array of any of these types



Annotation Class<?> Element

```
BankAccount acct1 = new BankAccount();
startWork("com.jwhh.utils.AccountWorker", acct1);
```

```
@ProcessedBy(AccountWorker.class)
public class BankAccount {
  public BankAccount(String id) {...}
  public BankAccount(String id, int balance) {...}
  // other members elided
}
```

```
BankAccount acct1 = new BankAccount();
startWorkSelfContained(acct1);
```



Annotation Class<?> Element

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface ProcessedBy {
   Class<?> value();
}
```

```
@ProcessedBy(AccountWorker.class)
public class BankAccount {
  public BankAccount(String id) {...}
  public BankAccount(String id, int balance) {...}
  // other members elided
}
```

Annotation Class<?> Element

```
BankAccount acct1 = new BankAccount();
startWorkSelfContained(acct1);
```

```
void startWorkSelfContained(Object workerTarget) throws Exception {
 Class<?> targetType = workerTarget.getClass();
  ProcessedBy pb = targetType.getAnnotation(ProcessedBy.class);
 Class<?> workerType = pb.value();
  TaskWorker worker = (TaskWorker) workerType.newInstance();
  // Remainder of code just like startWork method . . .
 // . . .
```

Summary



Programs incorporate context and intent

- Standard type system isn't always enough
- Sometimes need metadata

Annotations act as metadata

- Annotations are a special kind of interface
- Do not change target behavior
- Must be interpreted



Summary



Can declare custom annotations

- Similar to declaring interfaces
- Use interface keyword preceded by @
- Set retention to control availability
- Set target to narrow use

Annotations accessed with reflection

- Use getAnnotation method of target



Summary



Annotations can optionally have elements

- Associate values with annotation
- Declared as methods
- Setting is similar to fields
- Can associate a default value
- Element name value provides shorthand

