JDT embraces Type Annotations

Java 8 ready

Stephan Herrmann
GK Software
Eclipse and Java™ 8

• **Java 8 features supported:**
  - JSR308 - Type Annotations.
  - JEP120 - Repeating Annotations.
  - JEP118 - Method Parameter Reflection.
  - JSR269 - Pluggable Annotation Processor API & javax.lang.model API enhancements for Java 8.
  - JSR 335 – Lambda Expressions
    • Lambda expressions & method/constructor references
    • Support for “code carrying” interface methods
    • Enhanced target typing / new overload resolution & type inference
`\lambda` (JSR 335)
Stephan Herrmann: JDT Embraces Type Annotations - EclipseCon North America 2014

Tomorrow 17:00 to 18:00:
Grand Peninsula C

JDT embraces lambda expressions

- Srikanth [IBM India]
- Noopur Gupta [IBM India]
- Stephan Herrmann

λ (JSR 335)
@ (JSR 308)
Annotations in more Places

- **Java 5: annotate declarations**
  - ElementType: packages, classes, fields, methods, locals ...

- **Java 8: annotate types**
  - ElementType.TYPE_USE
  - ElementType.TYPE_PARAMETER

So what?
Why Care About Types?

- **Dynamically typed languages**
  - anything goes
  - but may fail at runtime
  - e.g.: “method not understood”

- **Type = Constraints on values**
  
- **To statically detect anomalies**
  
  - missing capability
  - incompatible assignment
  - undeclared capability

```java
dog1 = new Dog();
dog1.bark();
dog2 = new Object();
dog2.bark();
```
Why Care About Types?

- Dynamically typed languages
  - anything goes
  - but may fail at runtime
  - e.g.: “method not understood”
- Type = Constraints on values
  To statically detect anomalies
  - missing capability
  - incompatible assignment
  - undeclared capability

Annotate source code:
- make assumptions explicit
- convince the compiler that code is safe

Dog dog1 = new Dog();
dog1.bark();

Object dog2 = new Object();
dog2.bark();

Dog dog3 = new Object();
dog3.bark();

Object dog4 = new Dog();
dog4.bark();
Why Care About Types?

- **Constraint checking avoids errors**
  - No Such Method / Field
    - Basic statically typed OO
  - ClassCastException
    - Generics
  - ??Exception
    - SWTException("Invalid thread access")
    - ...
    - NullPointerException

```java
void letemBark(Vector dogs) {
    Dog aDog = (Dog) dogs.get(0);
    aDog.bark();
}
```
Let the Type System Handle Nullity

- **Ideally**: Java would force explicit choice
  - String  definitely a String, never null
  - String?  either a String or null
  - Type system ensures: no dereferencing of null

- **Nullity as a language feature?**
  - Heavy weight, incompatible change
  - Language change for each new constraint?
Pluggable Type System

• Make it easier to add new constraints
  – Only one new syntax for all kinds of constraints

• Make it easier to add new type checkers
  – Checker Framework (Michael Ernst – U of Washington)

• Examples
  – @NonNull
  – @Interned equals(==, equals)
  – @Immutable value cannot change (Java 5 ?)
  – @ReadOnly value cannot change via this reference
  – @UI code requires to run on the SWT UI thread
Can't Java 5 Do All This?

- We've been lying about the method result
  - but we can't lie about everything, e.g.:

```java
void letemBark(@NonNull List<Dog> dogs) {
    dogs.get(0).bark();
}
```

NPE?
void good() {
    @NonNull List<@NonNull String> l1 = new ArrayList<>();
    l1.add("Hello");
    for (String elem : l1)
        System.out.println(elem.toUpperCase());

    @NonNull List<@Nullable String> l2 = new ArrayList<>();
    l2.add(null);
    for (String unknown : l2)
        if (unknown != null)
            System.out.println(unknown.toUpperCase());
}

void bad(List<String> unknown, List<@Nullable String> withNulls) {
    @NonNull List<@NonNull String> l1 = new ArrayList<>();
    l1.add(null);
    String canNull = withNulls.get(0);
    System.out.println(canNull.toUpperCase());
}

l1 cannot contain null elements

l2 can contain null elements

Null type mismatch: required '@NonNull String' but the provided value is null
Annotated Generics

```java
void good() {
    @NonNull List<@NonNull String> l1 = new ArrayList<>();
    l1.add("Hello");
    for (String elem : l1)
        System.out.println(elem.toUpperCase());

    @NonNull List<@Nullable String> l2 = new ArrayList<>();
    l2.add(null);
    for (String unknown : l2)
        if (unknown != null) System.out.println(unknown.toUpperCase());
}

void bad(List<String> unknown, List<@Nullable String> withNulls) {
    @NonNull List<@NonNull String> l1 = new ArrayList<>();
    l1.add(null);
    String first = l1.get(0);
    if (first == null) return;

    l1 = unknown;
    l1 = withNulls;
    String canNull = withNulls.get(0);
    System.out.println(canNull.toUpperCase());
}
```

- **l1** cannot contain **null** elements
- **l2** can contain **null** elements

Null comparison always yields false: The variable `first` cannot be null at this location
Annotated Generics

```java
void good() {
    @NonNull List<@NonNull String> l1 = new ArrayList<>();
l1.add("Hello");
    for (String elem : l1)
        System.out.println(elem.toUpperCase());

    @NonNull List<@Nullable String> l2 = new ArrayList<>();
l2.add(null);
    for (String unknown : l2)
        if (unknown != null)
            System.out.println(unknown.toUpperCase());
}
```

```java
void bad(List<String> unknown, List<@Nullable String> withNulls) {
    @NonNull List<@NonNull String> l1 = new ArrayList<>();
l1.add(null);
    String first = l1.get(0);
    if (first == null) return;

    l1 = unknown;
l1 = withNulls;
    String canNull = withNulls.get(0);
    System.out.println(canNull.toUpperCase());
}
```

Null type safety (type annotations): The expression of type 'List<String>' needs unchecked conversion to conform to '@NonNull List<@NonNull String>'

l1 cannot contain null elements

l2 can contain null elements
Annotated Generics

```java
void good() {
    @NonNull List<? @NonNull String> l1 = new ArrayList<>();
    l1.add("Hello");
    for (String elem : l1)
        System.out.println(elem.toUpperCase());

    @NonNull List<? @NonNull String> l2 = new ArrayList<>();
    l2.add(null);
    for (String unknown : l2)
        if (unknown != null)
            System.out.println(unknown.toUpperCase());
}

void bad(List<String> unknown, List<? @Nullable String> withNulls) {
    @NonNull List<? @NonNull String> l1 = new ArrayList<>();
    l1.add(null);
    String first = l1.get(0);
    if (first == null) return;

    l1 = unknown;
    l1 = withNulls;
    String l1 = l1.get(0);
    System.out.println(l1.toUpperCase());
}
```

- `l1` cannot contain `null` elements
- `l2` can contain `null` elements
- Null type mismatch (type annotations): required '@NonNull List<@NonNull String>' but this expression has type 'List<@Nullable String>'
Annotated Generics

```java
void good() {
    @NonNull List<String> l1 = new ArrayList<>();
    l1.add("Hello");
    for (String elem : l1)
        System.out.println(elem.toUpperCase());

    @NonNull List<String> l2 = new ArrayList<>();
    l2.add(null);
    for (String unknown : l2)
        if (unknown != null)
            System.out.println(unknown.toUpperCase());
}

void bad(List<String> unknown, List<String> withNulls) {
    @NonNull List<String> l1 = new ArrayList<>();
    l1.add(null);
    String first = l1.get(0);
    if (first == null) return;

    l1 = unknown;
    l1 = withNulls;

    String canNull = withNulls.get(0);
    System.out.println(canNull.toUpperCase());
}
```

- l1 cannot contain null elements
- l2 can contain null elements
- Potential null pointer access: The variable canNull may be null at this location
Theorie for API design

• **Terminologie**
  - Type parameter, type variable, type argument, type bound
  - Covariance, contravariance, invariance

• **Choosing the right level of genericity**
  - Always @NonNull, always @Nullable?
    • easier for the library developer
  - Should clients be able to choose?
    • more widely usable
Generic API

• The classic declaration
  – Unconstrained type parameter:
    • public interface List<T> { ... }

• Client side
  – Free to choose the type argument:
    • List<@NonNull Person>
    • List<@Nullable Person>

• Implementer
  – No knowledge about type variable T
  – Must assume the worst
    • need to check on dereference
    • cannot assign null to a T variable

Caveat:
Strict checking not yet implemented
Type Bounds

- **Implementer needs more knowledge**
  - constrain the type
    - class X <T extends Comparable> { ... }
  - constrain the nullness
    - class X <T extends @NonNull Object> { ... }
  - client can provide same or more specific type
    - @NonNull Object <: @Nullable Object
API Methods

- **What's the contract?**
  - `abstract 0 apply (@Nullable I arg);`

- **Callers**
  - can pass null

- **All implementers**
  - must accept null
  - **cannot** override `@Nullable → @NonNull`
  - **could** override `@NonNull → @Nullable`
    - contravariant parameters
    - covariant return
What does it cost?

- How many additional annotations?
  - will code become unreadable due to null annotations?
- We have 2 strong mechanisms to alleviate the burden

Demo Time
Caveat: Arrays

- Semantics are changing from Java 7 to Java 8
More Type Annotations
JavaUI

• Research by Colin Gordon et al
  – Statically check that
    • code needing access to the SWT display is called from the UI thread
  – Is the approach safe?
JavaUI

- Research
  - Can we statically check that code needing access to the SWT display is called from the UI thread
- Is the approach safe?

Fig. 2. $\lambda_{\mathsf{UI}}$ runtime expression syntax and operational semantics.
JavaUI

- Research by Colin Gordon et al
  - To statically check that
    - code needing access to the SWT display is called from the UI thread
  - Is the approach safe?
  - Is it practical?
### JavaUI

<table>
<thead>
<tr>
<th>Role</th>
<th>Annotation</th>
<th>Target</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Effects</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@SafeEffect</td>
<td>Method</td>
<td>Marks a method as safe to run on any thread (default)</td>
</tr>
<tr>
<td></td>
<td>@UIEffect</td>
<td>Method</td>
<td>Marks a method as callable only on the UI thread</td>
</tr>
<tr>
<td></td>
<td>@PolyUIEffect</td>
<td>Method</td>
<td>Marks a method whose effect is polymorphic over the receiver type’s effect parameter</td>
</tr>
<tr>
<td><strong>Defaults</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@UIType</td>
<td>Type Decl.</td>
<td>Changes the default method effect for a type’s methods to @UIEffect</td>
</tr>
<tr>
<td></td>
<td>@UIPackage</td>
<td>Package</td>
<td>Changes the default method effect for all methods in a package to @UIEffect</td>
</tr>
<tr>
<td></td>
<td>@SafeType</td>
<td>Type Decl.</td>
<td>Changes the default method effect for a type’s methods to @SafeEffect (useful inside a @UIPackage package)</td>
</tr>
<tr>
<td><strong>Polymorphism</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@PolyUIType</td>
<td>Type Decl.</td>
<td>Marks an effect-polymorphic type (which takes exactly one effect parameter)</td>
</tr>
<tr>
<td><strong>Instantiating Polymorphism</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@Safe</td>
<td>Type Use</td>
<td>Instantiates an effect-polymorphic type with the @SafeEffect effect (also used for monomorphic types, considered subtypes of @Safe Object)</td>
</tr>
<tr>
<td></td>
<td>@UI</td>
<td>Type Use</td>
<td>Instantiates an effect-polymorphic type with the @UIEffect effect</td>
</tr>
<tr>
<td></td>
<td>@PolyUI</td>
<td>Type Use</td>
<td>Instantiates an effect-polymorphic type with the @PolyUIEffect effect (the effect parameter of the enclosing type)</td>
</tr>
</tbody>
</table>

Table 1. JavaUI annotations.
JavaUI

- **Research by Colin Gordon et al**
  - To statically check that
    - code needing access to the SWT display is called from the UI thread
  - Is the approach safe?
  - Is it practical?
    - Evaluated against 8 programs / plugins – 90,000+ UI LOC
    - Found 8 real defects
    - Extensive assessment of study results
  - Does it need JSR 308?
    - For full expressiveness: yes
    - But much can already be done with SE5 annotations
TypeBinding Backstage Story

aka “Symbol”

- Type bindings are “interned”
  - OK to use ==
- Broken by encoding type annotations in type bindings
- Solution
  - Find/replace == comparisons for T <: TypeBinding
  - Tweak our compiler report affected locations
  - Plan: publish the tweak, controlled by @Uninterned
Status

• Null Annotations
  – org.eclipse.jdt.annotation_2.0.0
    • @NonNull, @Nullable specify Target(TYPE_USE)
    • @NonNullByDefault: more fine tuning

• Null Analysis (per compiler option)
  – Nullness is an integral part of the type system
  – Fine tuned defaults only in Luna
  – TODO: Strict checking against type variables
Status

- **Null Annotations**
- **Null Analysis (per compiler option)**
- **Planned: @Uninterned**
  - Detect accidental comparison using == or !=
- **Proposed: @UiEffect, @Ui ...**
  - by [Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman]
  - SWT: bye, bye, “Invalid thread access”
Status

- Null Annotations
- Null Analysis (per compiler option)
- Planned: @Uninterned
- Proposed: @UiEffect, @Ui …
- JDT/UI
  - OK: completion, refactoring, etc.
  - TODO: show in hover
  - TODO: update / add more quickfixes
Status

- Null Annotations
- Null Analysis (per compiler option)
- Planned: @Uninterned
- Proposed: @UiEffect, @Ui ...
- JDT/UI

Do You care about Types?
Dramatis personæ

- Olivier Thomann
- Andy Clement
- Michael Rennie
- Walter Harley
- Jesper S. Møller
- Stephan Herrmann
- Dani Megert
- Markus Keller
- Jay Arthanareeswaran
- Anirban Chakarborty
- Manoj Palat
- Shankha Banerjee
- Manju Mathew
- Noopur Gupta
- Deepak Azad
- Srikanth Sankaran

• Release: TODAY!!

http://download.eclipse.org/eclipse/downloads