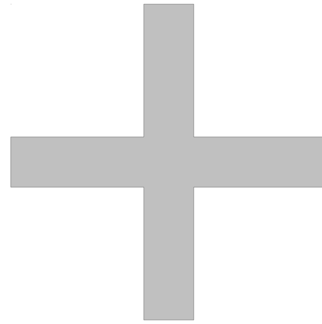
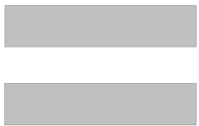


JDT embraces Type Annotations



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

λ

(JSR 335)

λ

(JSR

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

JDT embraces lambda expressions

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

@

(JSR 308)

Annotations in more Places

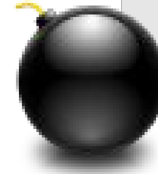
- Java 5: annotate declarations
  - ElementType: packages, classes, fields, methods, locals …
- Java 8: annotate types
  - ElementType.TYPE_USE
  - ElementType.TYPE_PARAMETER

Why Care About Types?

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

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

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



- Type = Constraints on values

  To *statically* detect anomalies
  - missing capability
  - incompatible assignment
  - undeclared capability

```
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?

- Dynamically typed languages
  - anythi
  - but m
  - e.g.: “r

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

- Type = Constraints on values

  To *statically* detect anomalies
  - missing capability
  - incompatible assignment
  - undeclared capability

```
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**

```
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?

```
@Target(ElementType.PARAMETER)
@interface NonNull5 {}

void java5(@NonNull5 String arg);
```

arg is qualified to be non-null

```
@Target(ElementType.TYPE_USE)
@interface NonNull8 {}

void java8(@NonNull8 String arg);
```

String is qualified to be non-null

```
@Target(ElementType.METHOD)
@interface NonNull5 {}

@NonNull5 String java5();
```

java5 is qualified to be non-null

```
@Target(ElementType.TYPE_USE)
@interface NonNull8 {}

@NonNull8 String java8();
```

String is qualified to be non-null

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

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

NPE?

Annotated Generics

```
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());
}
```

l1 cannot contain **null** elements

l2 can contain **null** elements

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

   l1 = unknown;
   l1 = withNulls;

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

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

Annotated Generics

```
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());
}
```

l1 cannot contain **null** elements

l2 can contain **null** elements

```
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 =
   l1 =

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

⚠ Null comparison always yields false:
The variable first cannot be null at this location

Annotated Generics

```
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());
}
```

l1 cannot contain **null** elements

l2 can contain **null** elements

```
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 =

   Strin
   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>'

Annotated Generics

```
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());
}
```

l1 cannot contain **null** elements

l2 can contain **null** elements

```
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;

   Strin
   Syste
}
```

Null type mismatch (type annotations):
required '@NonNull List<@NonNull String>'
but this expression has type 'List<@Nullable String>'

Annotated Generics

```
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());
}
```

l1 cannot contain **null** elements

l2 can contain **null** elements

```
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());
}
```

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 O 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

```
void test (@NonNull String [] stringsOrNulls) {
    System.out.println(stringsOrNulls[0]);
}
```

array of nonnull elements

the array can still be null ⇒ NPE

```
void test (String @NonNull [] stringsOrNulls) {
    System.out.println(stringsOrNulls[0]);
}
```

nonnull array

NPE- safe (but may print “null”)

- 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

- Rese
  - Ca
    - UI thread
  - Is t

Expressions	$e ::= \ldots \mid \ell$	Values	$v ::= \ell \mid n \mid () \mid (\lambda_\xi (x:\tau)\ e)$
Heaps	$H\ :\ \text{Location} \rightharpoonup \text{Effect} * \text{Value}$	Heap Type	$\Sigma\ :\ \text{Location} \rightharpoonup \text{Effect} * \text{Type}$
Machine	$\sigma ::= \langle H, \overline{e}, \overline{e} \rangle$	Machine Type	$\Omega ::= \langle \Sigma, \overline{\tau}, \overline{\tau} \rangle$
Optional New Thread	$O\ :\ \text{option}\ (\text{Expression} * \text{Effect})$		

$\boxed{\sigma \to \sigma}$

$$\text{E-UI1}\ \frac{H, e \to_{\mathsf{ui}} H', e', --}{\langle H, e\ \overline{e}, \overline{e_{bg}} \rangle \to \langle H', e'\ \overline{e}, \overline{e_{bg}} \rangle} \qquad \text{E-UI2}\ \frac{H, e \to_{\mathsf{ui}} H', e', (e_{new}, \mathsf{safe})}{\langle H, e\ \overline{e}, \overline{e_{bg}} \rangle \to \langle H', e'\ \overline{e}, \overline{e_{bg}}\ e_{new} \rangle}$$

$$\text{E-UI3}\ \frac{H, e \to_{\mathsf{ui}} H', e', (e_{new}, \mathsf{ui})}{\langle H, e\ \overline{e}, \overline{e_{bg}} \rangle \to \langle H', e'\ \overline{e}\ e_{new}, \overline{e_{bg}} \rangle} \qquad \text{E-NEXTUI}\ \frac{}{\langle H, v\ e\ \overline{e}, \overline{e_{bg}} \rangle \to \langle H', e\ \overline{e}, \overline{e_{bg}} \rangle}$$

$$\text{E-DROPBG}\ \frac{}{\langle H, \overline{e_{ui}}, \overline{e_{bg}}\ v\ \overline{e'_{bg}} \rangle \to \langle H', \overline{e_{ui}}, \overline{e_{bg}}\ \overline{e'_{bg}} \rangle} \qquad \text{E-BG1}\ \frac{H, e \to_{\mathsf{safe}} H', e', --}{\langle H, \overline{e_{ui}}, \overline{e_{bg}}\ e\ \overline{e'_{bg}} \rangle \to \langle H', \overline{e_{ui}}, \overline{e_{bg}}\ e\ \overline{e'_{bg}} \rangle}$$

$$\text{E-BG2}\ \frac{H, e \to_{\mathsf{safe}} H', e', (e_n, \mathsf{safe})}{\langle H, \overline{e_{ui}}, \overline{e_{bg}}\ e\ \overline{e'_{bg}} \rangle \to \langle H', \overline{e_{ui}}, \overline{e_{bg}}\ e\ \overline{e'_{bg}}\ e_n \rangle} \qquad \text{E-BG3}\ \frac{H, e \to_{\mathsf{safe}} H', e', (e_n, \mathsf{ui})}{\langle H, \overline{e_{ui}}, \overline{e_{bg}}\ e\ \overline{e'_{bg}} \rangle \to \langle H', \overline{e_{ui}}\ e_n, \overline{e_{bg}}\ e\ \overline{e'_{bg}} \rangle}$$

$\boxed{H, e \to_\xi H, e, O}$

$$\text{E-SPAWN}\ \frac{}{H, \mathsf{spawn}\{e\} \to_{\mathsf{safe}} H, (), (e, \mathsf{safe})} \qquad \text{E-ASYNC}\ \frac{}{H, \mathsf{asyncUI}\{e\} \to_{\mathsf{safe}} H, (), (e, \mathsf{ui})}$$

$$\text{E-REF1}\ \frac{H, e \to_\xi H', e', O}{H, \mathsf{ref}_{\xi'}\ e \to_\xi H', \mathsf{ref}_{\xi'}\ e', O} \qquad \text{E-REF2}\ \frac{\ell \notin \mathsf{Dom}(H)}{H, \mathsf{ref}_{\xi'}\ v \to_{\mathsf{safe}} H[\ell \mapsto (\xi', v)], \ell, --}$$

$$\text{E-DEREF1}\ \frac{H, e \to_\xi H', e', O}{H, !e \to_\xi H', !e', O} \qquad \text{E-DEREF2}\ \frac{H(\ell) = (\xi, v)}{H, !\ell \to_\xi H, v, --}$$

$$\text{E-APP1}\ \frac{H, e_1 \to_\xi H', e_1', O}{H, e_1\ e_2 \to_\xi H', e_1'\ e_2, O} \qquad \text{E-APP2}\ \frac{H, e \to_\xi H', e', O}{H, v\ e \to_\xi H', v\ e', O} \qquad \text{E-APP3}\ \frac{}{H, (\lambda_\xi (x:\tau)\ e)\ v \to_\xi H, e[x/v], --}$$

$$\text{E-ASSIGN1}\ \frac{H, e_1 \to_\xi H', e_1', O}{H, e_1 \leftarrow e_2 \to_\xi H', e_1' \leftarrow e_2, O} \qquad \text{E-ASSIGN2}\ \frac{H, e \to_\xi H', e', O}{H, \ell \leftarrow e \to_\xi H', \ell \leftarrow e', O}$$

$$\text{E-ASSIGN3}\ \frac{H(\ell) = (\xi, v')}{H, \ell \leftarrow v \to_\xi H[\ell \mapsto (\xi, v)], (), --} \qquad \text{E-SUBEFFECT}\ \frac{H, e \to_{\mathsf{safe}} H', e', O}{H, e \to_{\mathsf{ui}} H', e', O}$$

Fig. 2. λ_{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

- Res
  - T ... UI thread
  - Is
  - Is

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

Table 1. Java$_{UI}$ 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

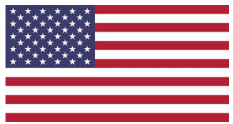
Status

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

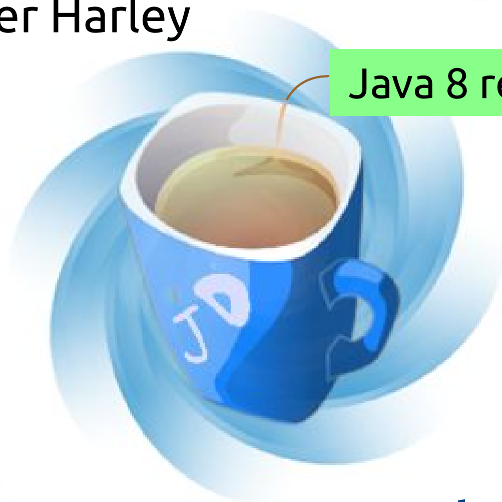
Do You care about Types?

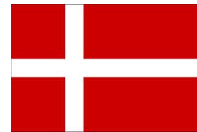
Dramatis personæ

- Olivier Thomann
- Andy Clement
- Michael Rennie



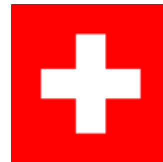
- Walter Harley





- 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