# UPCOMING JAVA™ PROGRAMMING LANGUAGE FEATURES

Alex Buckley
Spec lead, Java Language
Sun Microsystems

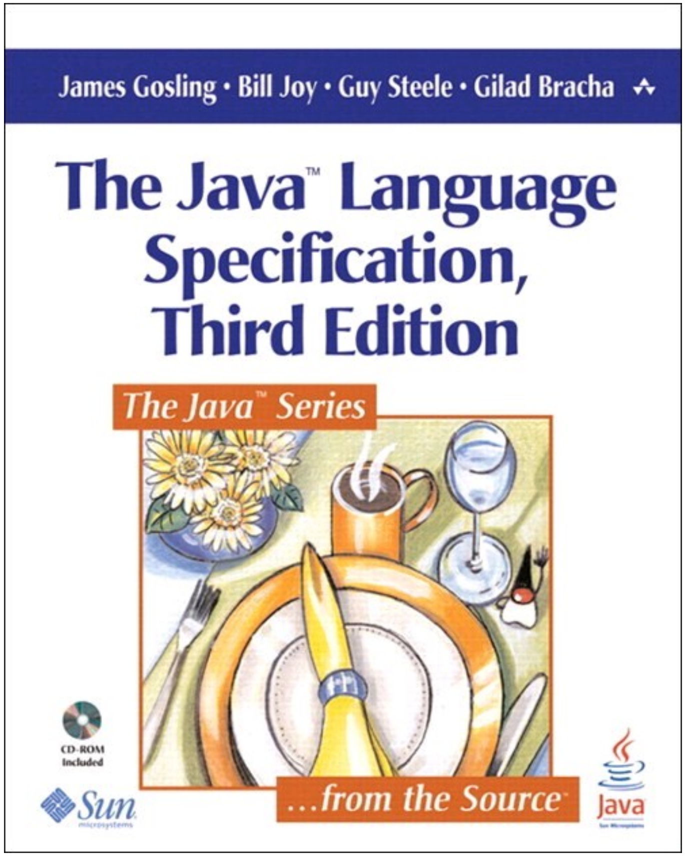
Neal Gafter
Software Engineer
Google

Michael D. Ernst
Associate Professor
MIT

Discover how Sun evolves the Java™ programming language and what lessons we have learned.

Learn about plans for annotations in the Java Platform, Standard Edition 7, and how OpenJDK™ affects language features.

# Who we are



James Gosling • Bill Joy • Guy Steele • Gilad Bracha

The Java™ Language Specification, Third Edition

*The Java™ Series*

CD-ROM Included

Sun microsystems

*…from the Source*

Java

Google Calendar BETA

MIT CSAIL

Massachusetts Institute of Technology

# Disclaimer

- The information in this presentation concerns forward-looking statements based on current expectations and beliefs about future events that are inherently susceptible to uncertainty and changes in circumstances etc etc etc etc

# Outline

- The art and science of language evolution
- Some language features under consideration
- JSR 308: Annotations on Java Types
- Long-term evolution
- OpenJDK & The Java Programming Language

# Outline

- The art and science of language evolution
- Some language features under consideration
- JSR 308: Annotations on Java Types
- Long-term evolution
- OpenJDK & The Java Programming Language

# Language evolution

- We're often asked: "Why don't you add X?"
- The assumption is that adding features is always good
- Why is this?

# Applications v. Languages

- A good application is *rich*
  - Applications compete on basis of completeness
  - User cannot do X until the application supports it
  - Features *rarely* interact with each other
  - Conclusion: <u>More features are better</u>
- A good language is *pure*
  - Languages are all Turing-complete
  - User can always do X;  the question is how elegantly
  - Features *often* interact with each other
  - Conclusion: <u>Fewer, more regular features are better</u>

# Fewer, more regular features are better

- The barrier to entry is very high
  - We need compelling reasons to add a feature
  - If in doubt, leave it out
  - "Just say no, until threatened with bodily harm" - James Gosling
  - Encourage creativity with the language, not in it
- Of all the features we could add, which should we add?

# Three pre-conditions for a Java language feature

- Respect the past
- Respect the future
- Respect the model

# 1/3: Respect the past

- Programs written in the Java language are strategic assets
- Adding a feature can break code
  - Sorry about **`assert`** and **`enum`**
  - Restricted keywords (**`module`**) are backward-compatible
- Removing a feature can break code
  - **`private protected`** in JDK™ version 1.0
- Changing a feature can break code
  - JLS2 required a trailing \n if last line in source was a comment
  - javac did not; "fixing" it to agree with JLS2 would break code
  - JLS3 was loosened

- A feature must be compatible with existing code

# 2/3: Respect the future

- Leave room for syntax to breathe
  - E.g. Nested modules not supported now, could be in future
- Syntax/semantics of a new feature should not conflict with syntax/semantics of an existing or potential feature

# Keyword parameters

- Keyword parameters exist in annotations:

```
@Point(x=3, y=4)
```

- The obvious syntax for keyword parameters at method call is:

```
new Point(x=3, y=4)
```

- But that already has a meaning, so the syntax would be:

```
new Point(x:3, y:4)
```

- = in annotations is inconsistent with = in expressions
- Annotations should probably have used : to align with future keyword parameters at method call
- A feature must allow consistent evolution

# 3/3: Respect the model

- A language reflects a computational model
  - Simula: Object orientation → Classes
  - CLU: Data abstraction → Interfaces
  - Erlang: Inter-process communication → Actors
- The Java language has a simple computational model
  - High-level ("General-purpose, concurrent, class-based, object-oriented")
  - Civilized relationship to APIs (java.lang.String, java.lang.Throwable)
  - Aligned with the JVM (Accessibility, inheritance, dynamic linking)
- Evolution can make a language more regular within its model
  - Improves consistency
  - E.g. Strings in switch

# Respecting the model

- A more abstract model encourages a more abstract language
- A more abstract language encourages more abstract programs
  - If you can only express 'procedure', you will not program 'objects'
  - If you can only express 'error code', you will not program 'exceptions'
  - If you can only express 'thread', you will not program 'actors'
- A more abstract program is easier to understand and maintain
- Evolution can make a language reflect a more abstract model
  - Improves expressiveness

# Four principles that recognize the Java model

- Encourage high-level practices
- Covet clarity
- Prefer static typing
- Isolate the language from APIs

# 1/4: A program's meaning is hidden by accidental complexity

- Fred Brooks, "No Silver Bullet"

> "What does a high-level language accomplish?
> It frees a program from much of its accidental complexity.
>
> An abstract program consists of conceptual constructs:
> operations, datatypes, sequences, and communication.
>
> The concrete machine program is concerned with
> bits, registers, conditions, branches, channels, disks, and such.
>
> To the extent that the high-level language embodies the constructs wanted in the abstract program ... it eliminates a whole level of complexity that was never inherent in the program at all."

- <u>Principle 1: Encourage high-level practices</u>

# 2/4: A program is read more often than it is written

- Clear code is easiest to read
  - Directly expresses the programmer's intent in solving the problem
  - Uses the language idiomatically (respects the computational model)
  - Minimizes implementation artifacts (accidental complexity)

- Language abstractions are the primary enabler of clear code
  - Java: Interfaces, exceptions (OO model)
  - Scala: Traits, pattern matching (OO/Functional model)
  - Erlang: Message passing (Communication model)

- Principle 2: Covet clarity

# 3/4: A static type system increases confidence in code

- Static typing proves the absence of (some) bugs at compile-time
- Testing and dynamic typing can only prove the presence of bugs
- Formal documentation is complete; narrative text is not
- Most Java keywords are about static typing

abstract
assert
boolean
break
byte
case
catch
char
class
const
continue
default
do
double
else
enum
extends
final
finally
float
for
if
goto
implements
import
instanceof
int
interface
long
native
new
package
private
protected
public
return
short
static
strictfp
super
switch
synchronized
this
throw
throws
transient
try
void
volatile
while

- Principle 3: Prefer static typing

# 4/4: A language is more widespread than its APIs

- One language, many APIs
- APIs come and go but a language is forever
  - APIs are deprecated far more than language features
  - A language will be compromised by linkage to a deprecated API
  - The JLS used to define java.lang
- <u>Principle 4: Isolate the language from APIs</u>

# Recap: Principles for Java language evolution

- Encourage high-level practices
  - Do the right thing
- Covet clarity
  - Do the thing right
- Prefer static typing
  - Stay safe
- Isolate the language from APIs
  - Stay abstract

# Outline

- The art and science of language evolution
- Some language features under consideration
- JSR 308: Annotations on Java Types
- Long-term evolution
- OpenJDK & The Java Programming Language

# Multi-catch

-
  ```
  try { ... }
  catch (X1 e) { foo(); }
  catch (X2 e) { foo(); }
  catch (X3 e) { bar(); }
  ```

- A longstanding request is to allow catching `X1` and `X2` together, without resorting to catching `Exception` itself

-
  ```
  try { ... }
  catch (X1,X2 e) { foo(); }
  catch (X3    e) { bar(); }
  ```

- A disjunctive type `X1,X2` represents `X1` or `X2`
  - The members of `e` are those of a common superclass

# Safe re-throw

```
void m() throws X1,X2 {
  try { /* Something that can throw X1,X2 */ }
  catch (Throwable e) {
    logger.log(e);
    throw e; // Error: Unreported exception Throwable
  }
}
```

- We want to express we're rethrowing the exception in the `try{}`

```
void m() throws X1,X2 {
  try { /* Something that can throw X1,X2 */ }
  catch (final Throwable e) {
    logger.log(e);
    throw e; // Compiles OK; can throw X1,X2
  }
}
```

# Modular programming in Java today

- Packages
  - Package names are hierarchical
  - Package membership is not

- Access control
  - Types shared across packages must be made public
  - Hope no-one finds your "internal" packages
  - Rely on comments/documentation to describe "official" APIs

- Interfaces
  - Not always desirable to have all members be public

# A typical package hierarchy

```
org/
   netbeans/
           core/
               Debugger.class
               ...
               utils/
                    ErrorTracker.class
                    ...
               wizards/
                    JavaFXApp.class
                    ...
           addins/
                 ...
```

# Classes in different packages need to collaborate

```
org/
   netbeans/
           core/
               Debugger.class
               ...
               utils/
                    ErrorTracker.class
                    ...
               wizards/
                    JavaFXApp.class
                    ...
           addins/
                 ...
```

# org.netbeans.core is an obvious "unit"

```
org/
   netbeans/
           core/
               Debugger.class
               ...
               utils/
                    ErrorTracker.class
                    ...
               wizards/
                    JavaFXApp.class
                    ...
           addins/
                 ...
```

# org.netbeans.core is a conceptual "module"

```
org/
   netbeans/
           core/
               Debugger.class
               ...
               utils/
                    ErrorTracker.class
                    ...
               wizards/
                    JavaFXApp.class
                    ...
           addins/
                 ...
```

# Modules in the Java language

Module concept in the language

```
//  org/netbeans/core/Debugger.java
module  org.netbeans.core;
package org.netbeans.core;
public  class Debugger {
    ... new ErrorTracker() ...
}
```

# Modules in the Java language

Module concept in the language

One module has many packages

Module access specified in the language

```
//  org/netbeans/core/Debugger.java
module  org.netbeans.core;
package org.netbeans.core;
public  class Debugger {
    ... new ErrorTracker() ...
}

//  org/netbeans/core/utils/ErrorTracker.java
module  org.netbeans.core;
package org.netbeans.core.utils;
module  class ErrorTracker {
    module int getErrorLine() { ... }
}
```

# Modules in the Java language

Module concept in the language

One module has many packages

```
//  org/netbeans/core/Debugger.java
module  org.netbeans.core;
package org.netbeans.core;
public  class Debugger {
    ... new ErrorTracker() ...
}
```

Module access specified in the language

```
//  org/netbeans/core/utils/ErrorTracker.java
module  org.netbeans.core;
package org.netbeans.core.utils;
module  class ErrorTracker {
    module int getErrorLine() { ... }
}
```

Module dependencies specified in the language

```
//  org/netbeans/core/module-info.java
@Version("7.0")
@ImportModule(name="java.se.core", version="1.7+")
module org.netbeans.core;
```

# Compiling and running a Java module

- `javac org/netbeans/core/*`
- `javac org/netbeans/core/utils/*`

- `java org.netbeans.core.Debugger`

# Impact of modules in the language & VM

- **`module`** restricted keyword
- **`module-info.java`** for module-level annotations
- **`package-info.java`** can declare module membership
- Classfile attribute for module membership
- Classfile flag for "module-private" accessibility
- Module-private accessibility enforced by the Java Virtual Machine
- javadoc and javap understand modules in .java and .class files

# Outline

- The art and science of language evolution
- Some language features under consideration
- JSR 308: Annotations on Java Types
- Long-term evolution
- OpenJDK & The Java Programming Language

# JSR 308: Annotations on Java Types

Two problems with annotations in Java 1.5:

1. Syntactic limitation on annotations
   - Can only be written on declarations
2. Semantic limitation of the type system
   - Doesn't prevent enough bugs

JSR 308 addresses these problems:

- Extends Java programming language syntax to permit annotations in more locations
- Enables creation of more powerful annotation processors

# Syntactic problem: Annotations on declarations only

- Classes

```
package java.security;
@Deprecated class Signer { ... }
```

- Methods

```
@Test void additionWorks() { assert 1 + 1 == 2; }
@Override boolean equals(MyClass other) // warning
```

- Fields

```
@CommandLineArg(name="input", required=true)
private String inputFilename;
```

- Locals/statements

```
List<Object> objs = ...;
@SuppressWarnings List<String> strings = objs;
```

- Goal:  Write annotations on type uses

# JSR 308: Annotations on generics and arrays

- Generics:

```
List<@NonNull String> strings;

class UnmodifiableList<T>
    implements @Readonly List<@Readonly T> {
  ...
}
```

- Arrays are treated analogously
  - Separately annotate the element type and the array itself

# JSR 308: Annotations on local variables

```
@Interned String s = getName().intern();

@NonEmpty List<String> strings = ...;
```

- Possible to annotate local variables today but annotations are not preserved in the class file

# JSR 308: Annotations on casts

```
// Both variables are null, or neither is.
Pattern startRegex, endRegex;
...
if (startRegex != null) {
  endRegex = (@NonNull Pattern) endRegex;
  ...
}
```

---

```
Graph g = new Graph();
...  } add nodes and
...  }    edges
// Now, g will not be changed any more
@Immutable Graph g2 = (@Immutable Graph) g;
```

# JSR 308: Annotations on the receiver (`this`)

- It is possible to annotate formal parameters today

```
/** Method body does not modify jaxbElement */
void marshal(@Readonly Object jaxbElement,
             @Mutable Writer writer)
{ .. }
```

- It should be possible to annotate the receiver too

```
/** myMarshaller.marshal(myJaxb, myWriter)
  * does not modify myMarshaller           */
void marshal(@Readonly Object jaxbElement,
             @Mutable Writer writer) @Readonly
{ .. }
```

# Semantic problem: Weak type checking

- Type checking prevents many bugs
  - `int i = “JSR 308”;`
- Type checking doesn't prevent enough bugs
  - `getValue().toString(); // NullPointerException`
- Cannot express important properties about code
  - Non-null, interned, immutable, encrypted, tainted, ...

- Solution: pluggable type systems
  - Design a type system to solve a specific problem
  - Annotate your code with type qualifiers
  - Type checker warns about violations (bugs)
  - Using annotations insulates the language in case we make a mistake designing the type system

# Pluggable checkers in practice

- Scales to >200,000 LOC
- Found bugs in every codebase

- Comparison to other null dereference checkers (on a 5KLOC codebase)

| | Errors | | False | Annotations |
|---|---|---|---|---|
| | *Found* | *Missed* | warnings | written |
| JSR 308 | 8 | 0 | 4 | 35 |
| FindBugs | 0 | 8 | 1 | 0 |
| Jlint | 0 | 8 | 8 | 0 |
| PMD | 0 | 8 | 0 | 0 |

# Usability

- Programmers found the checkers easy to use
- Is it too verbose?
  - @NonNull:  1 per 75 lines
  - @Interned:  124 annotations in 220KLOC revealed 11 bugs
  - Possible to annotate part of program
  - Fewer annotations in new code
- Is it hard to build a new checker?
  - Most users don't have to
  - Basic functionality: mention annotation on command line
  - More advanced functionality: using the Checkers Framework, just override a few methods

Demo of pluggable type-checking

BOF-5031, 7.30pm tonight

# JSR 308: How to get involved

- Web search for "JSR 308" or "Annotations on Java types"
- Completely open mailing list
- Specification document
- Reference implementation (patch to OpenJDK compiler)
- Checkers Framework
  - 5 checkers built so far
  - @NonNull @Interned @Readonly @Immutable
  - Basic checker (for any annotation name)

- Go forth and prevent bugs!

# Outline

- The art and science of language evolution
- Some language features under consideration
- JSR 308: Annotations on Java Types
- Long-term evolution
- OpenJDK & The Java Programming Language

# Long-term evolution: areas of interest

- Abstraction
  - Parallel algorithms
- Component adaptation
  - Interface evolution via first-class versioning
  - Delegation / Forwarding
  - Extension methods / Scala views
- Practical structural typing
  - See Aldrich & Malayeri at ECOOP 2008
- Pluggable literal syntaxes
  - Integration with other languages via JSR-223

# Long-term evolution: areas of non-interest

- User-defined operator overloading
- Multimethods
- Macros
- Dynamic typing

- Keeping something out of the Java programming language does not imply keeping it out of the Java platform
  - JavaFX™ Script
  - JRuby
  - Jython
  - Groovy
  - Scala

# Outline

- The art and science of language evolution
- Some language features under consideration
- JSR 308: Annotations on Java Types
- Long-term evolution
- OpenJDK & The Java Programming Language

# OpenJDK & The Java Programming Language

- The OpenJDK Compiler Group discusses javac implementation
- It is not the place to discuss Java language design
- It is the place to bring spec-compliant bug fixes
- Performance improvements and better diagnostics too
- Patches must not silently change the language

Bug 4741726: allow Object+=String

- 2002: Proposed as a spec change by Neal
- 2005: Accepted in JLS3 *but not in javac*
- 2008: Michael Bailey questioned whether JLS or javac is right
- 2008: First javac fix through the OpenJDK process :-)

# OpenJDK & The Kitchen Sink Language

Oak Language Specif
JAVA
Effective Java
JAVA PUZZLERS
JAVA CONCURRENCY IN PRACTICE
imagine
experience
Beyond Java
The Java Language Specification, Third Edition
The Java Virtual Machine Specification, Second Edition
SMALLTALK-80 THE LANGUAGE AND ITS IMPLEMENTATION
Goldberg and Ro
Java Tuto
RUB
CLASS List
INT size()

James Gosling:

"Throw stuff into the kitchen sink without thinking too hard about whether or not it's a good idea.

Let folk kick the tires.

Those experiences inform the choice of which features go into the standard."

# OpenJDK & The Kitchen Sink Language

- The KSL is a virtual language: Java language + your ideas
- Host your javac patches on your blog or Web site
- Mail the OpenJDK Compiler Group with the URL
- Try to update the Java Language and VM Specs too
  - See Joe Darcy's blog on "So you want to change the Java Programming Language..."

# Signing off...

- Thanks to Josh Bloch, Joe Darcy, Jon Gibbons, Brian Goetz, Eamonn McManus

- http://blogs.sun.com/abuckley/
- http://gafter.blogspot.com/
- http://openjdk.java.net/groups/compiler/
- http://pag.csail.mit.edu/jsr308/

- "Upcoming Java Programming Language Features"
  - BOF-5031, tonight 7.30pm
- "Modularity in the Java Platform"
  - TS-6175, Wednesday 10:50 - 11:50

# THANK YOU

Alex Buckley / Neal Gafter / Michael D. Ernst

Sun / Google / MIT