Evolving the Java Programming Language

Neal Gafter

Overview

- The Challenge of Evolving a Language
- Design Principles
- Design Goals
- JDK7 and JDK8

Challenge: Evolving a Language

"What is it like trying to extend a mature language?" -*Brian Goetz*

Challenge: Evolving a Language

"What is it like trying to extend a *mature* language?"

ma-ture: *adjective*

1. having completed natural growth and development.
2. completed, perfected, or elaborated in full.
3. (of an industry, technology, market, etc.) no longer developing or expanding.

Challenge: Evolving a Language

Q: What is it like trying to extend a *mature* language?

A: *It is impossible, by definition.*

Challenge: Evolving a Language

Q: What is it like trying to extend a *widely deployed* language?

A: Language change is influenced by existing code and APIs.

Existing APIs affect change

Support retrofitting existing APIs:

- With compatible behavior

Existing APIs affect change

Support retrofitting existing APIs:

- With compatible behavior
- Consistent with existing design
  - Don't expose, create design flaws

Existing APIs affect change

Some case studies:

- Generics (vs erasure)
- Wildcards (vs declaration-site variance)
- Autoboxing, unboxing (vs wrappers)
- Varargs (vs overloading)
- For-each (vs Iterator)

Generics (vs erasure)

- Adding reified generics is *compatible*
- May not allow retrofitting existing code
  - Collection
  - WeakReference

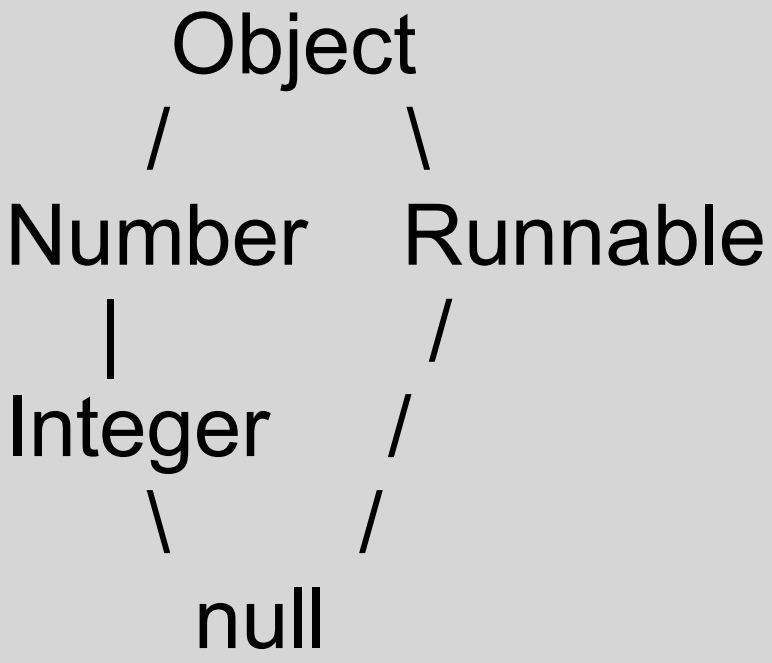
Wildcards (vs declaration-site variance)

There is a simpler alternative to *wildcards*:
declaration-site variance

- 'wildcards' appear on type definitions
- Use sites much simpler
- Can retrofit most APIs
  - but not Collection

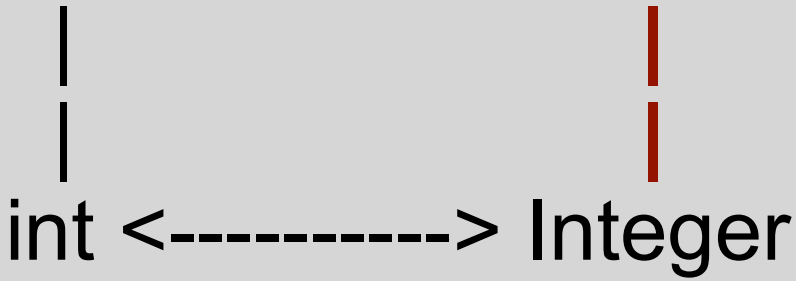
Autoboxing, unboxing

Type systems form a *lattice*



Autoboxing, unboxing

Adding conversions can break the lattice



Existing boxed types don't have this relation.

Autoboxing, unboxing

Solutions

- Introduce new boxing interfaces; or
- Patchwork specification

for-each (vs iterator)

- Iterator has a vestigal *remove* method.
- Introduce java.lang.Iterator without it?
- Cannot retrofit Collection
  - without requiring recompile

  (existing implementations don't implement iterator() that returns the new type)

Design Principles

Design Principles

- Encourage desirable practices (that might not otherwise be followed) my making them easy
  - synchronized
  - Annotations isolate configuration data
  - Generics for typesafe APIs

Design Principles

- Encourage desirable practices
- Isolate language from specific APIs

Design Principles

- Encourage desirable practices
- Isolate language from specific APIs
- Prefer reading over writing
  - clarity over conciseness

Design Principles

- Encourage desirable practices
- Isolate language from specific APIs
- Prefer reading over writing
- Prefer static typing

Design Principles

- Encourage desirable practices
- Isolate language from specific APIs
- Prefer reading over writing
- Prefer static typing
- Remain backward compatible

Short-term design Goals

- Regularize Existing Language
  - Improve diagnostics vs generics
  - Fix type inference
  - String switch
  - Limited operator overloading
  - Improved catch clauses
- Modularity

Improve Diagnostics vs Generics

```
interface Box<T> {
    T get();
    void put(T t);
}
class Tmp {
    static void test(Box<? extends Number> box) {
        box.put(box.get());
    }
}
```

```
Error: put(capture#417 of ? extends java.lang.Number)
   in Box<capture#417 of ? extends java.lang.Number>
   cannot be applied to (java.lang.Number)
        box.put(box.get());
            ^
```

Improve Diagnostics vs Generics

```
interface Box<T> {
    T get();
    void put(T t);
}
class Tmp {
    static void test(Box<? extends Number> box) {
        box.put(box.get());
    }
}
```

```
Error: cannot call put(T) as a member of in Box<? ex-
   tends java.lang.Number>
        box.put(box.get());
            ^
```

Fix Type Inference: constructors

- Today:

```
Map<String, List<String>> anagrams
  = new HashMap<String,
  List<String>>();
```

Fix Type Inference: constructors

- Proposed:

```
Map<String, List<String>> anagrams
  = new HashMap<>();
```

Fix Type Inference: arguments

- Today:

```
public <E> Set<E> emptySet() { … }
void timeWaitsFor(Set<Man> people) { … }

// * Won't compile!
timeWaitsFor(Collections.emptySet());
```

Fix Type Inference: arguments

- Today:

```
public <E> Set<E> emptySet() { … }
void timeWaitsFor(Set<Man> people) { … }

// OK
timeWaitsFor(Collections.<Man>emptySet());
```

Fix Type Inference: arguments

- Proposed:

```
public <E> Set<E> emptySet() { … }
void timeWaitsFor(Set<Man> people) { … }

// OK
timeWaitsFor(Collections.emptySet());
```

String Switch

- Today

```
static boolean booleanFromString(String s) {
    if (s.equals("true")) {
        return true;
    } else if (s.equals("false")) {
        return false;
    } else {
        throw new IllegalArgumentException(s);
    }
}
```

String Switch

- Proposed

```
static boolean booleanFromString(String s) {
    switch(s) {
      case "true":
        return true;
      case "false":
        return false;
      default:
        throw new IllegalArgumentException(s);
    }
}
```

Limited Operator Overloading

- Today

```
enum Size { SMALL, MEDIUM, LARGE }

if (mySize.compareTo(yourSize) >= 0)
    System.out.println("You can wear my pants.");
```

Limited Operator Overloading

- Proposed

```
enum Size { SMALL, MEDIUM, LARGE }

if (mySize > yourSize)
    System.out.println("You can wear my pants.");
```

Improved Catch Clauses: multi

- Today:

```
try {
    return klass.newInstance();
} catch (InstantiationException e) {
    throw new AssertionError(e);
} catch (IllegalAccessException e) {
    throw new AssertionError(e);
}
```

Improved Catch Clauses: multi

- Proposed:

```
try {
    return klass.newInstance();
} catch (InstantiationException | IllegalAccessException e) {
    throw new AssertionError(e);
}
```

Improved Catch Clauses: rethrow

- Today:

```
try {
  doable.doit(); // Throws several types
} catch (Throwable ex) {
  logger.log(ex);
  throw ex; // Error: Throwable not declared
}
```

Improved Catch Clauses: rethrow

- Proposed:

```
try {
  doable.doit(); // Throws several types
} catch (final Throwable ex) {
  logger.log(ex);
  throw ex; // OK: Throws the same several types
}
```

Others?

- Properties?
- Serialization annotations?

Long-term goals

- Regularize Existing Language
  - Reification
  - Further Generics simplification
- Concurrency support
  - Immutable data
  - Control Abstraction
  - Closures
  - Actors, etc.

JDK7

- jsrs 277 + 294 (modularity)
- Maintenance review of jsr14
- “Small” language issues
- Possibly jsr308

(limited by time, resources)

JDK8

- Reification
- Control Abstraction

Further out: Immutable data, pattern matching, further operator overloading?

Q&A