

The Future of Java

Bob Lee

Google Inc.



Who is Bob Lee?

- > Google engineer
- > Android core library lead
- > Guice creator
- > JSR-330 lead
- > Google's alternate EC rep
- > St. Louisan
- > Speedo model



Let's talk about...

- > Project Coin
- > JSR-330: Dependency Injection for Java

Project Coin

Small language **changes**



Currently accepted proposals

- > Strings in switch
- > Automatic Resource Management (ARM)
- > Improved generic type inference for constructors
- > Simplified varargs method invocation
- > Collection literals and access syntax
- > Better integral literals
- > JSR-292 (Invokedynamic) support

Currently accepted proposals

- > Strings in switch
- > **Automatic Resource Management (ARM)**
- > Improved generic type inference for constructors
- > **Simplified varargs method invocation**
- > Collection literals and access syntax
- > Better integral literals
- > JSR-292 (Invokedynamic) support

1. ARM

- > Automatic Resource Management
- > Helps dispose of resources
- > Proposed by Josh Bloch

Example: Parsing a file header

```
public class HeaderParser {  
    /** Parses header from the first line of file. */  
    public static Header parse(File file) throws IOException,  
        ParseException {  
        BufferedReader in = new BufferedReader(new FileReader(file));  
        Header header = parse(in.readLine());  
        in.close();  
        return header;  
    }  
  
    private static Header parse(String first) throws ParseException {  
        ...  
    }  
}
```


Example: Parsing a file header

```
public class HeaderParser {  
    /** Parses header from the first line of file. */  
    public static Header parse(File file) throws IOException,  
        ParseException {  
        BufferedReader in = new BufferedReader(new FileReader(file));  
        Header header = parse(in.readLine());  
        in.close();  
        return header;  
    }  
  
    private static Header parse(String first) throws ParseException {  
        ...  
    }  
}
```

See the problem?

If we don't reach `close()`, we leak.

```
public class HeaderParser {  
    /** Parses header from the first line of file. */  
    public static Header parse(File file) throws IOException,  
        ParseException {  
        BufferedReader in = new BufferedReader(new FileReader(file));  
        Header header = parse(in.readLine());  
        in.close();  
        return header;  
    }  
  
    private static Header parse(String first) throws ParseException {  
        ...  
    }  
}
```

finally ensures close() is always called.

```
public class HeaderParser {
    /** Parses header from the first line of file. */
    public static Header parse(File file) throws IOException,
        ParseException {
        BufferedReader in = new BufferedReader(new FileReader(file));
        try {
            return parse(in.readLine());
        } finally {
            in.close();
        }
    }

    private static Header parse(String first) throws ParseException {
        ...
    }
}
```

But what happens when `close()` throws?

```
public class HeaderParser {
    /** Parses header from the first line of file. */
    public static Header parse(File file) throws IOException,
        ParseException {
        BufferedReader in = new BufferedReader(new FileReader(file));
        try {
            return parse(in.readLine());
        } finally {
            in.close();
        }
    }

    private static Header parse(String first) throws ParseException {
        ...
    }
}
```


We could ignore the exception from `close()`.

```
public class HeaderParser {
    /** Parses header from the first line of file. */
    public static Header parse(File file) throws IOException,
        ParseException {
        BufferedReader in = new BufferedReader(new FileReader(file));
        try {
            return parse(in.readLine());
        } finally {
            try { in.close(); } catch (IOException e) { /* ignore */ }
        }
    }

    private static Header parse(String first) throws ParseException {
        ...
    }
}
```

But it's better to throw the right exception.

```
public class HeaderParser {
    /** Parses header from the first line of file. */
    public static Header parse(File file) throws IOException,
        ParseException {
        BufferedReader in = new BufferedReader(new FileReader(file));
        boolean successful = false;
        try {
            Header header = parse(in.readLine());
            successful = true;
            return header;
        } finally {
            try { in.close(); } catch (IOException e) {
                if (successful) throw e;
                else e.printStackTrace(); // let original exception propagate
            }
        }
    }

    private static Header parse(String first) throws ParseException {
        ...
    }
}
```

Equivalent code, using an ARM block.

```
public class HeaderParser {  
    /** Parses header from the first line of file. */  
    public static Header parse(File file) throws IOException,  
        ParseException {  
        try (BufferedReader in = new BufferedReader(  
            new FileReader(file))) {  
            return parse(in.readLine());  
        }  
    }  
  
    private static Header parse(String first) throws ParseException {  
        ...  
    }  
}
```

Equivalent code, using an ARM block.

```
public class HeaderParser {
    /** Parses header from the first line of file. */
    public static Header parse(File file) throws IOException,
        ParseException {
        try (BufferedReader in = new BufferedReader(
            new FileReader(file))) {
            return parse(in.readLine());
        }
    }

    private static Header parse(String first) throws ParseException {
        ...
    }
}
```

Note: Technincally, we could still leak.

Equivalent code, using an ARM block.

```
public class HeaderParser {
    /** Parses header from the first line of file. */
    public static Header parse(File file) throws IOException,
        ParseException {
        try (Reader fin = new FileReader(file);
            BufferedReader in = new BufferedReader(fin)) {
            return parse(in.readLine());
        }
    }

    private static Header parse(String first) throws ParseException {
        ...
    }
}
```

Why ARM is important

Why ARM is important

- > The JDK opens & closes resources in 110 places.

Why ARM is important

- > The JDK opens & closes resources in 110 places.
- > 74 of those can leak. *2/3rds!*

Why ARM is important

- > The JDK opens & closes resources in 110 places.
- > 74 of those can leak. *2/3rds!*
- > None suppress exceptions correctly.

Why ARM is important

- > The JDK opens & closes resources in 110 places.
- > 74 of those can leak. *2/3rds!*
- > None suppress exceptions correctly.
- > ARM reduces error-prone boilerplate.

Why ARM is important

- > The JDK opens & closes resources in 110 places.
- > 74 of those can leak. *2/3rds!*
- > None suppress exceptions correctly.
- > ARM reduces error-prone boilerplate.
- > Ideally, *all finally* blocks would work this way.

2. Simplified varargs method invocation

- > Moves varargs warnings from the caller to the callee
- > Only relevant for non-reifiable varargs types
- > Vastly reduces # of warnings
 - A warning per caller (the status quo) vs.
 - One warning on **Arrays.asList()** itself
- > Helps catch errors sooner
- > Language changes don't get much simpler.
- > Proposed by Bob Lee

Today, the compiler warns the caller.

```
/** Collects instances of T. */
abstract class Sink<T> {
    /** Adds instances to this sink. */
    abstract void add(T... a);

    /** Adds t unless it's null. */
    void addUnlessNull(T t) {
        if (t != null)
            // Warning: "uses unchecked or unsafe operations"
            add(t);
    }
}
```

What's really going on here...

```
/** Collects instances of T. */
abstract class Sink<T> {
    /** Adds instances to this sink. */
    abstract void add(T... a);

    /** Adds t unless it's null. */
    void addUnlessNull(T t) {
        if (t != null)
            // Warning: "unchecked cast"
            add((T[]) new Object[] { t });
    }
}
```

After the language change...

```
/** Collects instances of T. */
abstract class Sink<T> {
    /** Adds instances to this sink. */
    abstract void add(T... a);

    /** Adds t unless it's null. */
    void addUnlessNull(T t) {
        if (t != null)
            add(t);
    }
}
```

Before the language change

```
class BrokenSink<T> extends Sink<T> {  
    Object[] array;  
  
    @Override void add(T... a) {  
        array = a;  
    }  
  
    void violateTypeSystem() {  
        array[0] = 5;  
    }  
}
```

After the language change

```
class BrokenSink<T> extends Sink<T> {  
    Object[] array;  
  
    // Warning: "enables unsafe generic array creation"  
    @Override void add(T... a) {  
        array = a;  
    }  
  
    void violateTypeSystem() {  
        array[0] = 5;  
    }  
}
```

What does this program print?

```
class StringSink extends Sink<String> {
    final List<String> list = new ArrayList<String>();
    @Override void add(String... a) {
        list.addAll(Arrays.asList(a));
    }
    @Override public String toString() {
        return list.toString();
    }
    public static void main(String[] args) {
        Sink<String> ss = new StringSink();
        ss.addUnlessNull("seppuku");
        System.out.println(ss);
    }
}
```


What does this program print?

```
class StringSink extends Sink<String> {  
    final List<String> list = new ArrayList<String>();  
    @Override void add(String... a) {  
        list.addAll(Arrays.asList(a));  
    }  
    @Override public String toString() {  
        return list.toString();  
    }  
    public static void main(String[] args) {  
        Sink<String> ss = new StringSink();  
        ss.addUnlessNull("seppuku");  
        System.out.println(ss);  
    }  
}
```

- a) StringSink@32c41a
- b) ["seppuku"]
- c) Nothing. It throws an exception.

If you answered C, you're correct!

```
class StringSink extends Sink<String> {  
    final List<String> list = new ArrayList<String>();  
    @Override void add(String... a) {  
        list.addAll(Arrays.asList(a));  
    }  
    @Override public String toString() {  
        return list.toString();  
    }  
    public static void main(String[] args) {  
        Sink<String> ss = new StringSink();  
        ss.addUnlessNull("seppuku"); // ClassCastException!  
        System.out.println(ss);  
    }  
}
```

a) StringSink@32c41a

b) ["seppuku"]

c) Nothing. It throws ClassCastException.

Let's look at Sink again...

```
/** Collects instances of T. */
abstract class Sink<T> {
    /** Adds instances to this sink. */
    abstract void add(T... a);

    /** Adds t unless it's null. */
    void addUnlessNull(T t) {
        if (t != null)
            // Warning: "unchecked cast"
            add((T[]) new Object[] { t });
    }
}
```

After the language change

```
class StringSink extends Sink<String> {
    final List<String> list = new ArrayList<String>();
    // Warning #2: "override generates a more specific varargs
    //              type erasure"
    // Note: We don't get the first warning because String[]
    //       is reifiable.
    @Override void add(String... a) {
        list.addAll(Arrays.asList(a));
    }
    @Override public String toString() {
        return list.toString();
    }
    public static void main(String[] args) {
        Sink<String> ss = new StringSink();
        ss.addUnlessNull("seppuku");
        System.out.println(ss);
    }
}
```

Not so fast. One more loophole...

```
abstract class PlainSink<T> extends Sink<T> {  
    @Override abstract void add(T[] a);  
}
```

After the language change

```
abstract class PlainSink<T> extends Sink<T> {  
    // Warning #3: "Overrides non-reifiable varargs type with array"  
    @Override abstract void add(T[] a);  
}
```

All that work just to get...

```
public class Arrays {  
    // Warning: "enables unsafe generic array creation"  
    public static <T> List<T> asList(T... a) {  
        ...  
    }  
    ...  
}
```


So we can suppress it once and for all!

```
public class Arrays {  
    @SuppressWarnings("generic-varargs")  
    // Ensures only values of type T can be stored in elements.  
    public static <T> List<T> asList(T... a) {  
        ...  
    }  
    ...  
}
```

The moral of this story...

- > Arrays and generics don't mix.
- > Varargs should have used **List**.



JSR-330: Dependency Injection for Java

package javax.inject

Interface Summary

<u>Provider<T></u>	Provides instances of T.
---------------------------------	--------------------------

Annotation Types Summary

<u>Inject</u>	Identifies injectable constructors, methods, and fields.
<u>Named</u>	String-based <u>qualifier</u> .
<u>Qualifier</u>	Identifies qualifier annotations.
<u>Scope</u>	Identifies scope annotations.
<u>Singleton</u>	Identifies a type that the injector only instantiates once.

For example

```
class Stopwatch {  
    final TimeSource timeSource;  
    Stopwatch() {  
        timeSource = new AtomicClock();  
    }  
    void start() {  
        ...  
    }  
    long stop() {  
        ...  
    }  
}
```

We could construct the time source directly.

```
class Stopwatch {  
    final TimeSource timeSource;  
    Stopwatch() {  
        timeSource = new AtomicClock();  
    }  
    void start() {  
        ...  
    }  
    long stop() {  
        ...  
    }  
}
```

Or use a factory.

```
class Stopwatch {  
    final TimeSource timeSource;  
    Stopwatch() {  
        timeSource = DefaultTimeSource.getInstance();  
    }  
    void start() {  
        ...  
    }  
    long stop() {  
        ...  
    }  
}
```


@Inject provides the best of both worlds.

```
class Stopwatch {  
    final TimeSource timeSource;  
    @Inject Stopwatch(TimeSource injected) {  
        timeSource = injected;  
    }  
    void start() {  
        ...  
    }  
    long stop() {  
        ...  
    }  
}
```

Testing against a factory

```
public class StopwatchTest extends TestCase {  
    public void testStopwatch() {  
        MockTimeSource mts = new MockTimeSource();  
        DefaultTimeSource.setInstance(mts);  
        Stopwatch stopwatch = new Stopwatch();  
        stopwatch.start();  
        long actual = stopwatch.stop();  
        mts.verify(actual);  
    }  
}
```

Testing against a factory

```
public class StopwatchTest extends TestCase {  
    public void testStopwatch() {  
        MockTimeSource mts = new MockTimeSource();  
        DefaultTimeSource.setInstance(mts);  
        Stopwatch stopwatch = new Stopwatch();  
        stopwatch.start();  
        long actual = stopwatch.stop();  
        mts.verify(actual);  
    }  
}
```

See the problem?

Testing against a factory *the right way*

```
public class StopwatchTest extends TestCase {
    public void testStopwatch() {
        TimeSource original = DefaultTimeSource.getInstance();
        try {
            MockTimeSource mts = new MockTimeSource();
            DefaultTimeSource.setInstance(mts);
            Stopwatch stopwatch = new Stopwatch();
            stopwatch.start();
            long actual = stopwatch.stop();
            mts.verify(actual);
        } finally {
            DefaultTimeSource.setInstance(original);
        }
    }
}
```

Testing using @Inject

```
public class StopwatchTest extends TestCase {  
    public void testStopwatch() {  
        MockTimeSource mts = new MockTimeSource();  
        Stopwatch stopwatch = new Stopwatch(mts);  
        stopwatch.start();  
        long actual = stopwatch.stop();  
        mts.verify(actual);  
    }  
}
```

Scoping a dependency

`@Singleton`

```
public class AtomicClock implements TimeSource {  
    ...  
}
```

@Inject vs. the factory pattern

@Inject vs. the factory pattern

- > Unit testing is easier.

@Inject vs. the factory pattern

- > Unit testing is easier.
- > You don't need to write the factory.

@Inject vs. the factory pattern

- > Unit testing is easier.
- > You don't need to write the factory.
- > Scopes are declarative--no more hand-written DCL.

@Inject vs. the factory pattern

- > Unit testing is easier.
- > You don't need to write the factory.
- > Scopes are declarative--no more hand-written DCL.
- > Easier modularization.

@Inject vs. the factory pattern

- > Unit testing is easier.
- > You don't need to write the factory.
- > Scopes are declarative--no more hand-written DCL.
- > Easier modularization.
- > We can reuse `Stopwatch` with different time sources.

@Inject vs. the factory pattern

- > Unit testing is easier.
- > You don't need to write the factory.
- > Scopes are declarative--no more hand-written DCL.
- > Easier modularization.
- > We can reuse `Stopwatch` with different time sources.
- > Even concurrently.

@Inject vs. the factory pattern

- > Unit testing is easier.
- > You don't need to write the factory.
- > Scopes are declarative--no more hand-written DCL.
- > Easier modularization.
- > We can reuse `Stopwatch` with different time sources.
- > Even concurrently.
- > Unlike the service loader pattern...

@Inject vs. the factory pattern

- > Unit testing is easier.
- > You don't need to write the factory.
- > Scopes are declarative--no more hand-written DCL.
- > Easier modularization.
- > We can reuse `Stopwatch` with different time sources.
- > Even concurrently.
- > Unlike the service loader pattern...
- > We can verify dependencies at build time.

JSR-330 itself

JSR-330 itself

- > Most open JSR ever

JSR-330 itself

- > Most open JSR ever
- > Hosted on Google Code Hosting

JSR-330 itself

- > Most open JSR ever
- > Hosted on Google Code Hosting
- > EG mailing list is publicly readable.

JSR-330 itself

- > Most open JSR ever
- > Hosted on Google Code Hosting
- > EG mailing list is publicly readable.
- > Public observer list and issue tracker

JSR-330 itself

- > Most open JSR ever
- > Hosted on Google Code Hosting
- > EG mailing list is publicly readable.
- > Public observer list and issue tracker
- > Spec, RI and TCK are all Apache-licensed.

JSR-330 itself

- > Most open JSR ever
- > Hosted on Google Code Hosting
- > EG mailing list is publicly readable.
- > Public observer list and issue tracker
- > Spec, RI and TCK are all Apache-licensed.
- > Fastest JSR ever: proposed to final in 4.5 months!

Next step: Configuration

```
public class TimeModule extends AbstractModule {  
    public void configure() {  
        bind(Stopwatch.class);  
        bind(StopwatchWidget.class);  
        bind(TimeSource.class).to(AtomicClock.class);  
    }  
}
```


Other cool stuff

- > Modules
- > New sorting routines
- > The G1 collector
- > MapMaker



Thank you!

- > Follow @crazybob or <http://crazybob.org/>
- > Don't miss Alex's modules talk.
- > Or my references talk.
- > One more thing...

