
ORACLE®

ORACLE®

Project Coin: Small Language Changes for JDK 7 & JSR 334: Small Language Changes for Java SE 7

Joseph D. Darcy
Java Platform Group

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions.
The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

coin, *n*. A piece of small change
coin, *v*. To create new language

“Making things programmers do everyday easier.”

Outline

- Background
- Overview of new language features
- Demo of features in NetBeans
- Developing the features
- Q & A

From *Evolving the Java™ Language*, JavaOne 2005

- **Java Language Principles**
  - Reading is more important than writing
  - Code should be a joy to read
  - The language should not hide what is happening
  - Code should do what it seems to do
  - Simplicity matters
  - A clear semantic model greatly boosts readability
  - Every “good” feature adds more “bad” weight
  - Sometimes it is best to leave things out

Evolving the Java™ Language, JavaOne 2005, cont.

- **One language: with same meaning everywhere**
- **We will evolve the Java Language but cautiously, with a long-term view**
  - we want Java to be around in 2030
  - we can't take a slash-and-burn approach
  - “first do no harm”
- **We will add a few selected features periodically**
  - aimed at developer productivity
  - while preserving clarity and simplicity

Project Coin Today

- OpenJDK Project:
  http://openjdk.java.net/projects/coin/
- JSR 334
  http://www.jcp.org/en/jsr/detail?id=334
  EDR specification on the way!
- Download JDK 7 binary snapshot builds from
  http://jdk7.dev.java.net/
- Current JDK 7 schedule is to
  ship in second half of 2011

Project Coin Features in JDK 7 builds

- Binary literals and underscores in literals
- Strings in switch
- Varargs warnings
- Diamond
- Multi-catch and more precise rethrow
- **`try`**-with-resources
  (formerly known as Automatic Resource Management, ARM)

Project Coin Tomorrow?

- Collections support?
  - Collection literals?
  - Support for [] access?
- Large arrays?
- Unsigned integer literals?
- Multi-line strings??
- Your favorite feature???

Coin Constraints

- *Small* language changes
  - Small in specification, implementation, testing
  - No new keywords!
  - Wary of type system changes
  - No JVM changes
- Coordinate with larger language changes
  - Project Lambda
  - Modularity
- One language, one **`javac`**
  - Interplay and interactions

The Features

A Java Riddle

What is special about the **`int`** value

```
1346704470
```

What is special about the **`int`** value

```
1_346_704_470
```

What is special about the **`int`** value

`0x50451456`

What is special about the **`int`** value

```
0x5045_1456
```

What is special about the **`int`** value

```
0x50_45_14_56
```

What is special about the **`int`** value

```
0b01010000010001010001010001010110
```

What is special about the **`int`** value

```
0b0101_0000_0100_0101_0001_0100_0101_0110
```

What is special about the **`int`** value

```
0b0101_0000_0100_0101_0001_0100_0101_0110
```

From the lsb, bit positions set are
2, 3, 5, 7, 11, 13, 17, 19…

What is special about the **`int`** value

```
0b0101_0000_0100_0101_0001_0100_0101_0110
```

From the lsb, bit positions set are
2, 3, 5, 7, 11, 13, 17, 19…

The bits set are the prime bit positions!

Strings in Switch

- When do you use a switch statement?
  - Many alternatives
- Case labels include
  - Integral *constants*
  - Enum constants
- But strings can be constants too!

```
int monthNameToDays(String s, int year) {
    if(s.equals("April") ||
       s.equals("June") ||
       s.equals("September") ||
       s.equals("November"))
       return 30;
    if(s.equals("January") ||
       s.equals("March") ||
       s.equals("May") ||
       s.equals("July") ||
       s.equals("August") ||
       s.equals("December"))
       return 31;
     if(s.equals("February"))
       ...
     else
       ...
     }
}
```

```
int monthNameToDays(String s, int year) {
    if(s == "April" ||
       s == "June" ||
       s == "September" ||
       s == "November")
       return 30;
    if(s == "January" ||
       s == "March" ||
       s == "May" ||
       s == "July" ||
       s == "August" ||
       s == "December")
       return 31;
     if(s == "February")
       ...
     else
       ...
     }
}
```

```
int monthNameToDays(String s, int year) {
    switch(s) {
      case "April":
      case "June":
      case "September":
      case "November":
          return 30;
      case "January":
      case "March":
      case "May":
      case "July":
      case "August":
      case "December":
          return 31;
      case "February":
          ...
      default
          ...
      }
}
```

```
int monthNameToDays(String s, int year) {
    switch(s) {
     case "April":       case "June":
     case "September":   case "November":
         return 30;
     case "January":     case "March":
     case "May":         case "July":
     case "August":      case "December":
         return 31;
     case "February":
         ...
     default
        ...
     }
}
```

Varargs warnings

- Is anything wrong with calling
  - `Arrays.asList(T... a)`
  - `Collections.addAll(Collection<? super T> c, T... elements)`
  - `EnumSet.of(E first, E... rest)`
- No!

```
class Test {
  public static void main(String... args) {
    List<List<String>> monthsInTwoLanguages =
      Arrays.asList(Arrays.asList("January",
                                  "February"),

                    Arrays.asList("Gennaio",
                                  "Febbraio"));
  }
}
```

```
class Test {
  public static void main(String... args) {
    List<List<String>> monthsInTwoLanguages =
      Arrays.asList(Arrays.asList("January",
                                  "February"),

                    Arrays.asList("Gennaio",
                                  "Febbraio"));
  }
}
```

```
Test.java:7: warning:
[unchecked] unchecked generic array creation
  for varargs parameter of type List<String>[]
          Arrays.asList(Arrays.asList("January",
                        ^
1 warning
```

Heap Pollution – JLSv3 4.12.2.1

- For example, a variable of type **`List<String>[]`** might point to an array of Lists where the Lists did not contain strings
- Reports possible locations of **`ClassCastExceptions`** at runtime
- A consequence of erasure and lack of reification

```
class Test {
  public static void main(String... args) {
    List<List<String>> monthsInTwoLanguages =
      Arrays.asList(Arrays.asList("January",
                                  "February"),

                    Arrays.asList("Gennaio",
                                  "Febbraio"));
  }
}
```

```
Test.java:7: warning:
[unchecked] unchecked generic array creation
  for varargs parameter of type List<String>[]
          Arrays.asList(Arrays.asList("January",
                        ^
1 warning
```

But nothing bad happens!

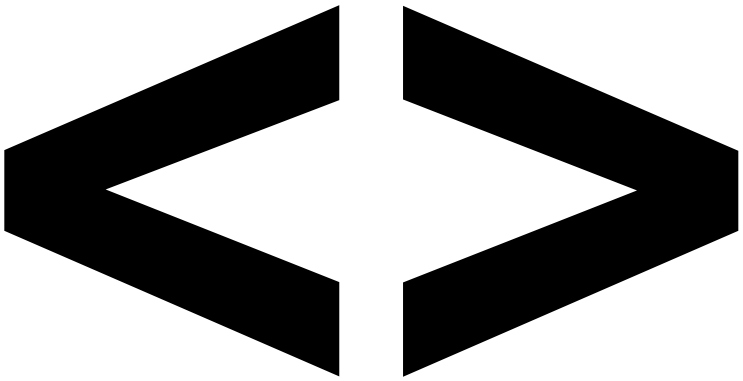
- Arrays created by the compiler for varargs are properly formed
- Well-behaved methods just iterate over the elements
- Unfriendly to warn at every call site
- *Declaration* is problematic

Varargs Warnings Revised

- New mandatory compiler warning at suspect varargs method declarations
- By applying an annotation at the declaration, warnings at the declaration *and call sites* can be suppressed
- New "`@SafeVarargs`" annotation in `java.lang`
  - Compiler will trust, may verify
  - Warnings or errors if annotation applied improperly

```
class Test {
  public static void main(String... args) {
    List<List<String>> monthsInTwoLanguages =
      {{"January", "February"},
       {"Gennaio", "Febbraio"}};
}
```

A possible future with Collection literals.



Pre-generics

```
List list =
   new ArrayList();
```

With Generics

```
List<String> list =
   new ArrayList<String>();
```

With diamond

```
List<List<String>> list =
   new ArrayList<>();
```

```
List<List<List<String>>> list =
   new ArrayList<List<List<String>>>();
```

```
List<List<List<String>>> list =
   new ArrayList<>();
```

```
List<List<List<List<String>>>> list =
  new ArrayList<List<List<List<String>>>>();
```

```
List<List<List<List<String>>>> list =
   new ArrayList<>();
```

```
List<List<List<List<List<String>>>>> list =
  new ArrayList<List<List<List<List<String>>>>>();
```

```
List<List<List<List<List<String>>>>> list =
  new ArrayList<>();
```

Multi-catch with More Precise Rethrow

```
try {
  // Reflective operations calling Class.forName,
  // Class.newInstance, Class.getMethod,
  // Method.invoke, etc.
} catch(ClassNotFoundException cnfe) {
  log(cnfe);
  throw cnfe;
} catch(InstantiationException ie) {
  log(ie);
  throw ie;
} catch(NoSuchMethodException nsme) {
  log(nsme);
  throw nsme;
} catch(InvocationTargetException ite) {
  log(ite);
  throw ite;
}
```

A tempting, but troublesome alternative

```
try {
  // Reflective operations calling Class.forName,
  // Class.newInstance, Class.getMethod,
  // Method.invoke, etc.
} catch(Exception e) {
  log(e);
  throw e;
}
```

Exception by-catch

```
try {
   // Reflective operations calling Class.forName,
   // Class.newInstance, Class.getMethod,
   // Method.invoke, etc.
} catch(Exception e) {
   log(e);
   throw e;
}
```

Catches both checked and unchecked exceptions

Reduced by-catch

```
try {
   // Reflective operations calling Class.forName,
   // Class.newInstance, Class.getMethod,
   // Method.invoke, etc.
} catch(RuntimeException e) {
   …
} catch(Exception e) {
   log(e);
   throw e;
}
```

Better.

Multi-catch

```
try {
  // Reflective operations calling Class.forName,
  // Class.newInstance, Class.getMethod,
  // Method.invoke, etc.
} catch(ClassNotFoundException |
        InstantiationException |
        NoSuchMethodException  |
        InvocationTargetException e) {
  log(e);
  throw e;
}
```

More Precise Rethrow

```
try {
  // Reflective operations calling Class.forName,
  // Class.newInstance, Class.getMethod,
  // Method.invoke, etc.
} catch(ClassNotFoundException |
        InstantiationException |
        NoSuchMethodException  |
        InvocationTargetException e) {
  log(e);
  throw e;
}
```

More More Precise Rethrow

```
try {
  // Reflective operations calling Class.forName,
  // Class.newInstance, Class.getMethod,
  // Method.invoke, etc.
} catch(ReflectiveOperationException e) {
  log(e);
  throw e;  // Means ClassNotFoundException or …
}
```

Still More More Precise Rethrow

```
void foo() throws ClassNotFoundException …
try {
  // Reflective operations calling Class.forName,
  // Class.newInstance, Class.getMethod,
  // Method.invoke, etc.
} catch(ReflectiveOperationException e) {
  log(e);
  throw e;  // Means ClassNotFoundException or …
}
```

More precise rethrow

- Under **`-source 7`**, enabled by default for **`final`** and *effectively final* catch parameters
- From *quantitative analysis*, $99\frac{44}{100}\%$ of catch parameters are **`final`** or effectively final
- Changing meaning of **`throw`**
  - Stops compilation of contrived legal programs, *but*
  - Compilation breakage not observed in practice analyzing 9+ million loc in several dozens projects
- *Disjunctive* catch parameters are implicitly final
  - Eases fuller support for disjunctive types in the future

`try`-with-resources (Automatic Resource Management)

- Let's say you want to copy an input stream to an output stream…

```
InputStream  in  = new FileInputStream(src);
OutputStream out = new FileOutputStream(dest);

byte[] buf = new byte[8192];
int n;

while ((n = in.read(buf)) >= 0)
  out.write(buf, 0, n);
```

```
InputStream  in  = new FileInputStream(src);
OutputStream out = new FileOutputStream(dest);

byte[] buf = new byte[8192];
int n;

while ((n = in.read(buf)) >= 0)
  out.write(buf, 0, n);
```

```
InputStream  in  = new FileInputStream(src);
OutputStream out = new FileOutputStream(dest);
try {
  byte[] buf = new byte[8192];
  int n;

  while ((n = in.read(buf)) >= 0)
    out.write(buf, 0, n);
} finally {
  out.close();
  in.close();
}
```

```
InputStream in = new FileInputStream(src);
try {
  OutputStream out = new FileOutputStream(dest) ;
  try {
    byte[] buf = new byte[8192];
    int n;

    while ((n = in.read(buf)) >= 0)
      out.write(buf, 0, n);
  } finally {
    out.close();
  }
} finally {
  in.close();
}
```

```
InputStream in = new FileInputStream(src);
try {
  OutputStream out = new FileOutputStream(dest) ;
  try {
    byte[] buf = new byte[8192];
    int n;

    while ((n = in.read(buf)) >= 0)
      out.write(buf, 0, n);
  } finally {
    out.close();
  }
} finally {
  in.close();
}
```

What if an exception occurs here?

```
InputStream in = new FileInputStream(src);
try {
  OutputStream out = new FileOutputStream(dest) ;
  try {
    byte[] buf = new byte[8192];
    int n;

    while ((n = in.read(buf)) >= 0)
      out.write(buf, 0, n);
  } finally {
    out.close();
  }
} finally {
  in.close();
}
```

Can get another exception here!

```
InputStream in = new FileInputStream(src);
try {
  OutputStream out = new FileOutputStream(dest) ;
  try {
    byte[] buf = new byte[8192];
    int n;

    while ((n = in.read(buf)) >= 0)
      out.write(buf, 0, n);
  } finally {
    out.close();
  }
} finally {
  in.close();
}
```

Could even get a third exception here!

Considerations

- First exception thrown is most likely to be informative
- Exception from a **`close`** method should propagate, unless there is already in incoming exception
- Don’t want to lose all record of a *suppressed* exception
- The additional code to implement this doesn’t fit on a slide anymore

```
InputStream  in  = new FileInputStream(src);
OutputStream out = new FileOutputStream(dest);

byte[] buf = new byte[8192];
int n;

while ((n = in.read(buf)) >= 0)
  out.write(buf, 0, n);
```

```
try(InputStream  in  = new FileInputStream(src);
    OutputStream out = new FileOutputStream(dest)) {

    byte[] buf = new byte[8192];
    int n;

    while ((n = in.read(buf)) >= 0)
      out.write(buf, 0, n);
}
```

How sweet it is

- Compiler desugars **`try`**-with-resources into nested **`try-finally`** blocks with variables to track exception state
- Suppressed exceptions are recorded for posterity using a new facility of **`Throwable`**
- API support in JDK 7
  - New superinterface **`java.lang.AutoCloseable`**
  - All **`AutoCloseable`** and by extension **`java.io.Closeable`** types usable with **`try`**-with-resources
  - JDBC 4.1 retrofitted as **`AutoCloseable`** too

More informative backtraces

```
java.io.IOException
        at Suppress.write(Suppress.java:19)
        at Suppress.main(Suppress.java:8)
        Suppressed: java.io.IOException
                at Suppress.close(Suppress.java:24)
                at Suppress.main(Suppress.java:9)
        Suppressed: java.io.IOException
                at Suppress.close(Suppress.java:24)
                at Suppress.main(Suppress.java:9)
```

To update your code to use with `try`-with-resources

- All **`Closeables`** are already useable!
- If a type has a no-arg **`public void close()`** method, implement **`AutoCloseable`** or **`Closeable`** as appropriate
- Use an annotation processor to find types to retrofit: “Project Coin: Bringing it to a Close(able)” http://blogs.sun.com/darcy/entry/project_coin_bring_close

The long arm of checked exceptions

- In the desugaring of

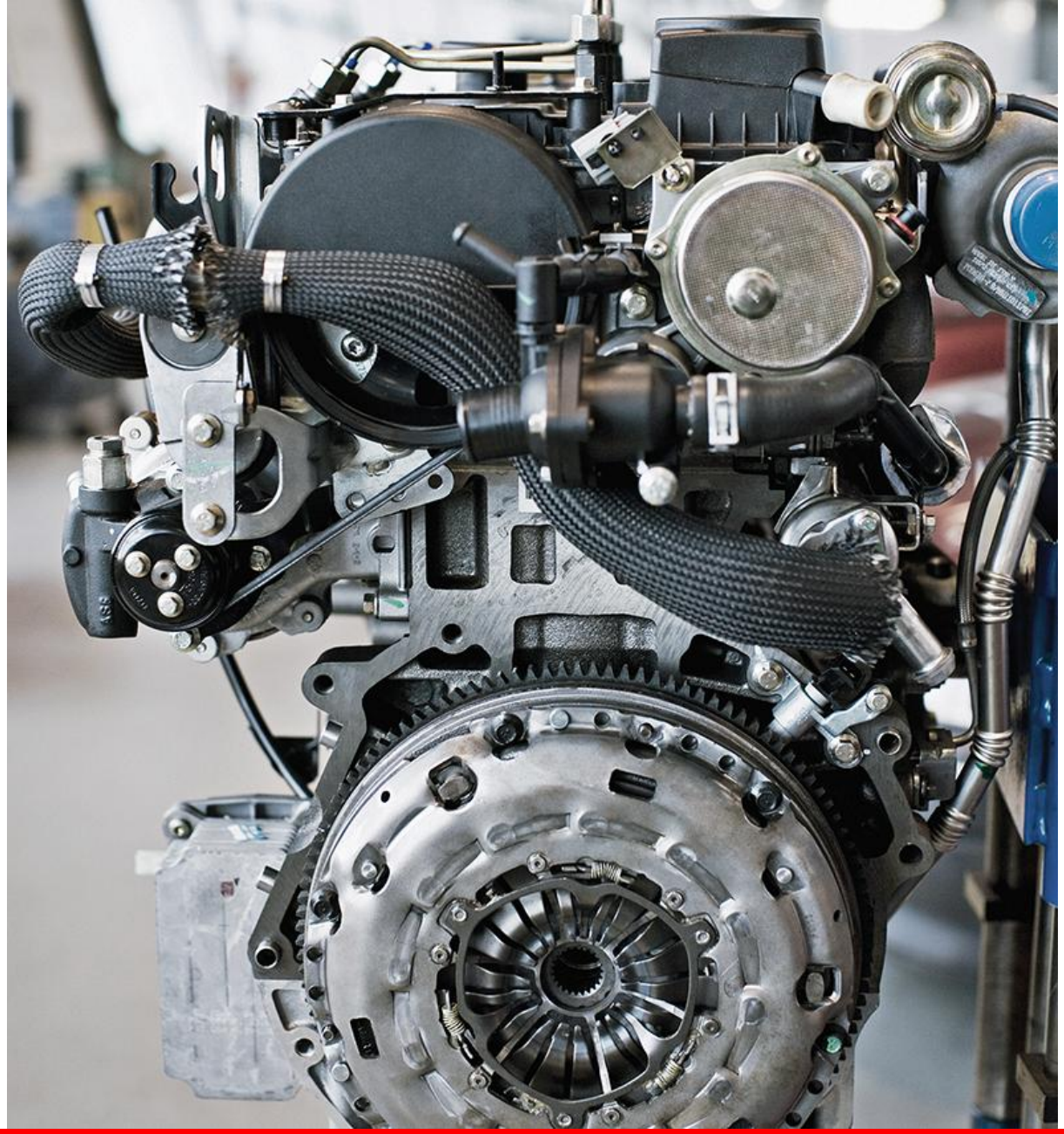
  ```
  try(new MyIoClose(){…})
  ```

  what should be the type of the synthetic variable?

  ```
  AutoCloseable #ac = new MyIoClose() {…}
  ```

- Just need to call the `close` method, right?
  - But what exceptions can the `close` method throw?
  - Use the most precise type possible to avoid overly broad exception inference
- Should this variant without a variable be dropped?
- Feedback through usage can help!

Demo



Developing the features

- Straightforward to use, less obvious to develop!

So you want to change the language…

- Update the Java Language Spec.
- Compiler Implementation
- Essential library support
- Write tests
- Update the JVM Spec.
- Future language evolution
- Update the JVM and class file tools
- Update JNI
- Update the reflective APIs
- Update serialization
- Update javadoc output
- Kinds of compatibility

Updating the Java Language Specification

- Syntax
- Type system
- Method resolution
- Flow analysis, e.g. definite assignment
- Memory model
- Total length of JLSv3: 647 pages
  - Chapter on Lexical structure ends on page 32
  - Syntax chapter is 12 pages
  - Syntax is less than 6% of the JLS!

Writing language change unit/regression tests

- Negative tests:
  - Invalid source files are rejected with expected error messages referencing the proper source locations
- Positive tests:
  - Valid source is compiled.
  - Proper modeling of the new language construct.
  - Resulting class files are structurally well-formed.
  - Resulting class files follow compiler-specific idioms.
  - Resulting class files run have correct operational semantics.

Strings in switch specification change

JLS §14.11 The switch Statement
"The type of [the switch] *Expression* must be **`char`**, **`byte`**, **`short`**, **`int`**, **`Character`**, **`Byte`**, **`Short`**, **`Integer`**, **`String`**, or an enum type (§8.9), or a compile-time error occurs."

Strings in switch Project Coin proposal form

PROJECT COIN SMALL LANGUAGE CHANGE PROPOSAL FORM v 1.0

AUTHOR(S): Joseph D. Darcy

OVERVIEW
Provide a two sentence or shorter description of these five aspects of the feature:
FEATURE SUMMARY: Should be suitable as a summary in a language tutorial.

Add the ability to switch on string values analogous to the existing ability to switch on values of the primitive types.

MAJOR ADVANTAGE: What makes the proposal a favorable change?

More regular coding patterns can be used for operations selected on the basis of a set of constant string values; then testing of the new construct should be obvious to casual developers.

MAJOR BENEFIT: Why is the platform better if the proposal is adopted?

Potentially better performance for string-based dispatch code.

MAJOR DISADVANTAGE: There is always a cost.

Some increased implementation and testing complexity for the compiler.

ALTERNATIVES: Can the benefits and advantages be had some way without a language change?

No; chained if-then-else tests for string equality are potentially expressive and introducing an enum for its switchable constants, one per string value of interest, would add another type to a program without good cause.

EXAMPLES
Show us the code.
SIMPLE EXAMPLE: Show the simplest possible program utilizing the new feature.

```
String s = ...
switch(s){
  case "foo":
    processFoo(s);
    break;
}
```

ADVANCED EXAMPLE: Show advanced usage(s) of the feature.

```
String s = ...
switch(s){
case "quux":
  processBaz(s);
  // fall-through
case "foo":
case "bar":
  processFooOrBar(s);
  break;
case "baz":
  processBaz(s);
  // fall-through
default:
  processDefault(s);
  break;
}
```

DETAILS
SPECIFICATION: Describe below the proposal's effect on the grammar, type system, and meaning of expressions and statements in the Java Programming Language as well as any other known impacts.

The lexical grammar is unchanged. String is added to the set of types valid for a switch statement in JLS section 14.11. Since Strings are already excluded in the definition of constant expressions, JLS 5 section 15.28, the switch label production does not need to be augmented. The existing restrictions in 14.11 on no duplicate labels, at most one default, no null labels, etc. all apply to Strings as well. The type system is unchanged. The defined assignment analysis of switch statement, JLS 5 section 16.2.9, is unchanged as well.

COMPILATION: How would the feature be compiled to class files?

One way to support this change would be to augment the JVM's lookupswitch instruction to operate on String values; however, that approach is not recommended or necessary. It would be possible to translate the switch to equivalent if-then-else code, but that would require unnecessary equality comparisons which are potentially expensive. Instead, a switch should count on a pre-existing method for

integer (or long) function values computed from the string. The exact natural choice for this function is String.hashCode, but other functions could also be used either alone or in conjunction with hashCode. (The specification of String.hashCode is a convenient detail at this point.) If all the string labels have different lengths, String.length() could be used instead of hashCode. Generally, a String.equals() check will be needed to verify the candidate string's identity in addition to the evaluation of the preceeding function because small testing inputs could evaluate to the same result.

A single case label, a single case label with a default, and two case labels can be special-cased to just equality checks without function evaluations. If there are collisions in String.hashCode on the set of case labels in a switch block, a different function without collisions on that set of inputs should be used; for example (length() + hashCode<<13) + s.length() is another candidate function.

Here are desugarings to currently legal Java source for the two examples above where the default hashcode doesn't collide:

```
// Simple Example
if (s.equals("foo")) { // cause NPE if s is null
  processFoo(s);
}
```

```
// Advanced example
{  // new scope for synthetic variables
  boolean $take_default = false;
  boolean $fallthrough = false;
  $default_label: {
    switch(s.hashCode()) { // cause NPE if s is null
      case 1402663: // "quux".hashCode():
        if (s.equals("quux")){
          $take_default = true;
          break $default_label;
        }
        processBaz(s);
        $fallthrough = true;
      case 101974: // "foo".hashCode()
      if (!$fallthrough && s.equals("foo")) {
          $take_default = true;
          break $default_label;
        }
        $fallthrough = true;
      case 97299: // "bar".hashCode():
      if (!$fallthrough && s.equals("bar")) {
          $take_default = true;
          break $default_label;
        }
        processFooOrBar(s);
        break;

      case 97307: // "baz".hashCode():
        if (s.equals("baz")){
          $take_default = true;
          break $default_label;
        }
        processBaz(s);
        $fallthrough = true;

      default:
        $take_default = true;
        break $default_label;
    }
  }
  if ($take_default)
    processDefault(s);
}
```

In the advanced example, the boolean "fallthrough" variable is needed to track whether a fall-through has occurred so the string equality checks can be skipped. If there are no fall-throughs, this variable can be removed. Likewise, if there is no default label in the original code, the $take_default variable is not needed and a simple "break" can be used instead.

In a translation directly to bytecode, the synthetic state variables can be replaced with goto's; expressing this in pseudo-Java source with goto:

```
// Advanced example in pseudo-Java with goto
switch(s.hashCode()) { // cause NPE if s is null
case 1402663: // "quux".hashCode():
  if (s.equals("quux")){
    goto $default_label;
  goto $foo$fooOrBar$code_label;

case 101974: // "foo".hashCode()
  if (s.equals("foo")){
    goto $default_label;
  goto $foo$fooOrBar$code_label;

case 97299: // "bar".hashCode()
  if (s.equals("bar")){
    goto $default_label;
  goto $foo$fooOrBar$code_label;

case 97307: // "baz".hashCode()
  if (s.equals("baz")){
    goto $default_label;
  processBaz(s);

default:
$default_label:
  processDefault(s);
  break;
}
```

Related to compilation, a compiler's existing diagnostics around falling through switches, such as Java's -Xlint:fallthrough option and @SuppressWarnings("fallthrough"), should work identically on switch statements based on strings.

TESTING: How can the feature be tested?

Generating various simple and complex uses of the new structure and verifying the proper execution paths occur; combinations to test include switch statements with and without fall-throughs, with and without collisions in the hash codes, and with and without default labels.

LIBRARY SUPPORT: Are any supporting libraries needed for the feature?

No.

REFLECTIVE APIS: Do any of the various reflective APIs need to be updated? This list of reflective APIs includes but is not limited to core reflection (java.lang.Class and java.lang.reflect.*), java.lang.model.*, the doclet API, and JPDA.

Only reflective APIs that model statements in the source language might be affected. None of core reflection, java.lang.model.*, the doclet API, and JPDA model statements; therefore, they are unaffected. The tree API in javac, http://java.sun.com/javase/6/docs/jdk/api/javac/tree/index.html, does model statements, but the existing API for switch statements is general enough to model the revised language without any API changes.

OTHER CHANGES: Do any other parts of the platform need be updated too? Possibilities include but are not limited to JVM, serialization, and output of the javadoc tool.

No.

MIGRATION: Sketch how a code base could be converted, manually or automatically, to use the new feature.

Look for sequences of if ("constant string".equals(foo)) or if (foo.equals("constant string")) and replace accordingly.

COMPATIBILITY
BREAKING CHANGES: Are any previously valid programs now invalid? If so, list one.

All existing programs remain valid.

EXISTING PROGRAMS: How do source and class files of earlier platform versions interact with the feature? Can any new overloading occur? Can any new overriding occur?

The semantics of existing class files and legal source files are unchanged by this feature.

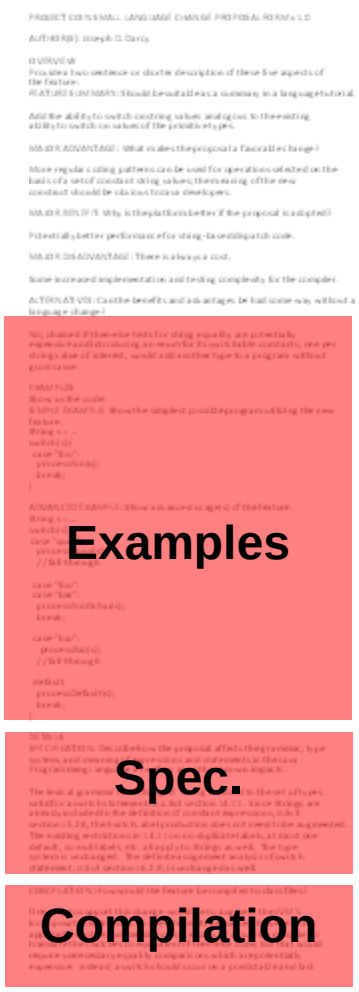
REFERENCES
EXISTING BUGS: Please include a list of any existing Sun bug ids related to this proposal.

5013360: Using Strings and Objects in Switch case statements.
http://bugs.sun.com/view_bug.do?bug_id=5013360

URL FOR PROTOTYPE (optional):

No prototype at this time.

Strings in switch Project Coin proposal form



Strings in switch, implementation and tests

Strings in switch, implementation only

Impl.

Impl., cont.

Making a hash of it

```
// Sugared
switch(s) {
  case "a":
  case "b":
  case "c":
    return 10;

  case "d":
  case "e":
  case "f":
    return 20;
  ...
}
```

```
// Desugared
int $t = -1;
switch(s.hashCode()) {
 case 0x61: // "a".hashCode()
   if(s.equals("a")) $t = 1; break;
 case 0x62:
   if(s.equals("b")) $t = 2; break;
 case 0x63:
   if(s.equals("c")) $t = 3; break;
 ...
}
switch($t) {
 case 1: case 2: case 3:
   return 10;

 case 4: case 5: case 6:
   return 20;
 ...
}
```

Making a hash of it

```
// Sugared
switch(s) {
  case "a":
  case "b":
  case "c":
    return 10;

  case "d":
  case "e":
  case "f":
    return 20;
  ...
}
```

```
// Desugared
int $t = -1;
switch(s.hashCode()) {
 case 0x61: // "a".hashCode()
   if(s.equals("a")) $t = 1; break;
 case 0x62:
   if(s.equals("b")) $t = 2; break;
 case 0x63:
   if(s.equals("c")) $t = 3; break;
 ...
}
switch($t) {
 case 1: case 2: case 3:
   return 10;

 case 4: case 5: case 6:
   return 20;
 ...
}
```

How to make a diamond

- *Type inference* has the compiler figure out types rather than the programmer writing them out
- The type argument for diamond, "**<>**", is inferred by the compiler
- Diamond reapplies existing type inference features to infer types parameters in constructor calls
- Similar to inference for generic methods:
  `public static <T> List<T> asList(T... a)`

What's in the box?

```
… = new Box<>(42);
```

What's in the box?

```
public class Box<T> {
    private T value;

    public Box(T value) {
        this.value = value;
    }

    T getValue() {
        return value;
    }
}
```

```
Box<>(42);
```

Types on the left…

```
Box<Integer> box
Box<Number> box
Box<Object> box
Box<?> box
Box<? extends Comparable<?>> box
…
```

```
… = new Box<>(42);
```

Pick a type for the right, but not just any type

```
Box<Integer> box
Box<Number> box
Box<Object> box
Box<?> box
Box<? extends Comparable<?>> box
…
```

```
… = new Box<>(42);
```

```
                                  Integer
                                   Number
                                   Object
                              Comparable<?>
  Object & Comparable<? extends …>
                                        …
```

Two inference schemes, “Simple” and “Complex”

- Simple, types parameters from:
  - Assignment context (where available)
- Complex, type parameters from:
  - Assignment context (where available) *plus*
  - Actual arguments to the constructor

Simple algorithm

```
Box<? extends Number> b = new Box<>(42)
```

Integer
Number
Object
Comparable<?>
Object & Comparable<? extends…>
…

Complex algorithm

```
Box<? extends Number> b = new Box<>(42)
```

Integer
Number
Object
Comparable<?>
Object & Comparable<? extends…>
…

A distinction with a difference

Simple: `Box<Number> b = new Box<>(42)`

Complex: `Box<Number> b = new Box<>(42)`

```
Integer
Number
Object
Comparable<?>
Object & Comparable<? extends…>
…
```

A distinction with a difference

Simple: `Box<Number> b = new Box<>(42)`

Complex: `Box<Number> b = new Box<>(42)`

```
incompatible types
Box<Number> b = new Box<>(42);
                    ^
  required: Box<Number>
  found:    Box<Integer>
1 error
```

Method contexts and algorithms

```
void m(Box<Integer> box){…}
```

Simple:

```
m(new Box<>(42))
```

Complex:

```
m(new Box<>(42))
```

```
Integer
Number
Object
Comparable<?>
Object & Comparable<? extends…>
…
```

Method contexts and algorithms

```
void m(Box<Integer> box){…}
```

Simple:

```
m(new Box<>(42))
```

```
method m cannot be applied to given types;
{ m(new Box<>(42)); }
  ^
  required: Box<Integer>
  found: Box<Object>
1 error
```

Comp…2))

Language design for the real world

- Sometimes the simple algorithm is more useful,
  but
  other times the complex algorithm is more useful
- What to do?
  - Is either one any good?
  - How to choose between them?
- Generate some data!
- *Quantitative* language design

Experimental Methodology, Summer 2009

- Find relevant large code bases (millions of lines of code)
  - OpenJDK
  - Tomcat
  - NetBeans
- Create and run *diamond finder*
- *Measure* effectiveness of algorithms
- Interpret results and decide

Per code base

	OpenJDK	Tomcat	NetBeans
Total `new`'s	104,138	6,048	94,768
Generics `new`'s	5,076	153	12,010
Simple Success	4,409	148	10,670
Complex Success	4,533	148	11,085

Analysis

	OpenJDK	Tomcat	NetBeans
Total `new`'s	104,138	6,048	94,768
Generics `new`'s	5,076	153	12,010
Simple Success	4,409	148	10,670
Complex Success	4,533	148	11,085

- Nontrivial fraction of constructor calls are to generic classes
- Of constructor calls to generic classes, in ***90%*** of cases the type parameters are successfully inferred
  - Simple infers in one 90% subset
  - Complex infers in a slightly different 90% subset
- Therefore, either algorithm would be effective
- Given equal effectiveness, what other criteria to use?

A way ahead to break the tie

- Neither algorithm is always better than the other
- Neither algorithm is a *subset* of the other
  - Picking one algorithm in JDK *N* and the other in JDK (*N+1*) would mean that some code that compiled in JDK *N* would stop compiling in JDK (*N+1*)
- Decision today constrains decisions tomorrow
- Originally integrated the simple algorithm…

A rising tide lifts all boats

- … later switched to the complex algorithm because
  - The complex algorithm reuses more inference machinery in the spec and implementation
  - More maintainable, implicit bug fixes for free
  - Better evolution properties
- Since the experiment, anticipate beneficial interactions with future inference improvements
  - Target typing in Project Lambda

A surprise: why is this disallowed?

- Using a more sophisticated inference scheme can be problematic sometimes

```
List<?> arg = ...;
new Box<>(arg);
```

A surprise: why is this disallowed?

```
List<?> arg = ...;
new Box<>(arg);
```

```
cannot infer type arguments for Box<>;
new Box<>(arg);
        ^
  reason: type argument List<CAP#1>
  inferred for Box<> is not allowed in this context
```

Types pre-JDK 5

- Primitive Types
- Reference Types

Types in JDK 5

- Primitive Types
- Reference Types
- Type-variables: **`class Box<X>`**
- Wildcards: **`? extends Number`**
- Captured-types: **`#103 capture-of ? extends Number`**
- Intersection types: **`Object & Comparable<?>`**

Types in JDK 7

- Primitive Types
- Reference Types
- Type-variables: **`class Box<X>`**
- Wildcards: **`? extends Number`**
- Captured-types: **`#103 capture-of ? extends Number`**
- Intersection types: **`Object & Comparable<?>`**
- Disjunctive types: **`IOException | SQLException`**

Expressible vs. Denotable

Expressible *and* Denotable

- Primitive Types
- Reference Types
- Type-variables: `class Box<X>`
- Wildcards: `? extends Number`
- Captured-types: `#103 capture-of ? extends Number`
- Intersection types: `Object & Comparable<?>`
- Disjunctive types: `IOException | SQLException`

Expressible vs. Denotable

Expressible, *non*-Denotable

- Primitive Types
- Reference Types
- Type-variables: `class Box<X>`
- Wildcards: `? extends Number`
- Captured-types: `#103 capture-of ? extends Number`
- Intersection types: `Object & Comparable<?>`
- Disjunctive types: `IOException | SQLException`

Expressible vs. Denotable

Expressible, *partially*-Denotable

- Primitive Types
- Reference Types
- Type-variables: `class Box<X>`
- Wildcards: `? extends Number`
- Captured-types: `#103 capture-of ? extends Number`
- Intersection types: `Object & Comparable<?>`
- Disjunctive types: `IOException | SQLException`

How is `T` inferred?

```
<T> List<T> asList(T... t){…}

List<?> arg = ...;
Arrays.asList(arg);
```

Don’t mistreat captured types!

```
<T> List<T> asList(T... t)

List<?> arg = ...;
Arrays.asList(arg);
```

T == List<#capture of ?>

How to break a diamond

```
List<?> arg = ...;
new Box<>(arg);
```

Not just a copy…

```
List<?> arg = ...;
new Box<List<?>>(arg);
```

Not just a copy…

```
List<?> arg = ...;
new Box<List<capture of ?>>(arg);
```

Even worse…

```
List<?> arg = ...;
new Box<List<capture of ?>>(arg) {...};
```

How does this get compiled?

```
List<?> arg = ...;
new a$1(arg);
```

Anonymous classes translate into a new class file with a full set of attributes.

```
class a$1 extends Box<List<capture of ?>>{...}
```

How does this get compiled?

This signature cannot be represented in the class file! Problematic for core reflection and separate compilation

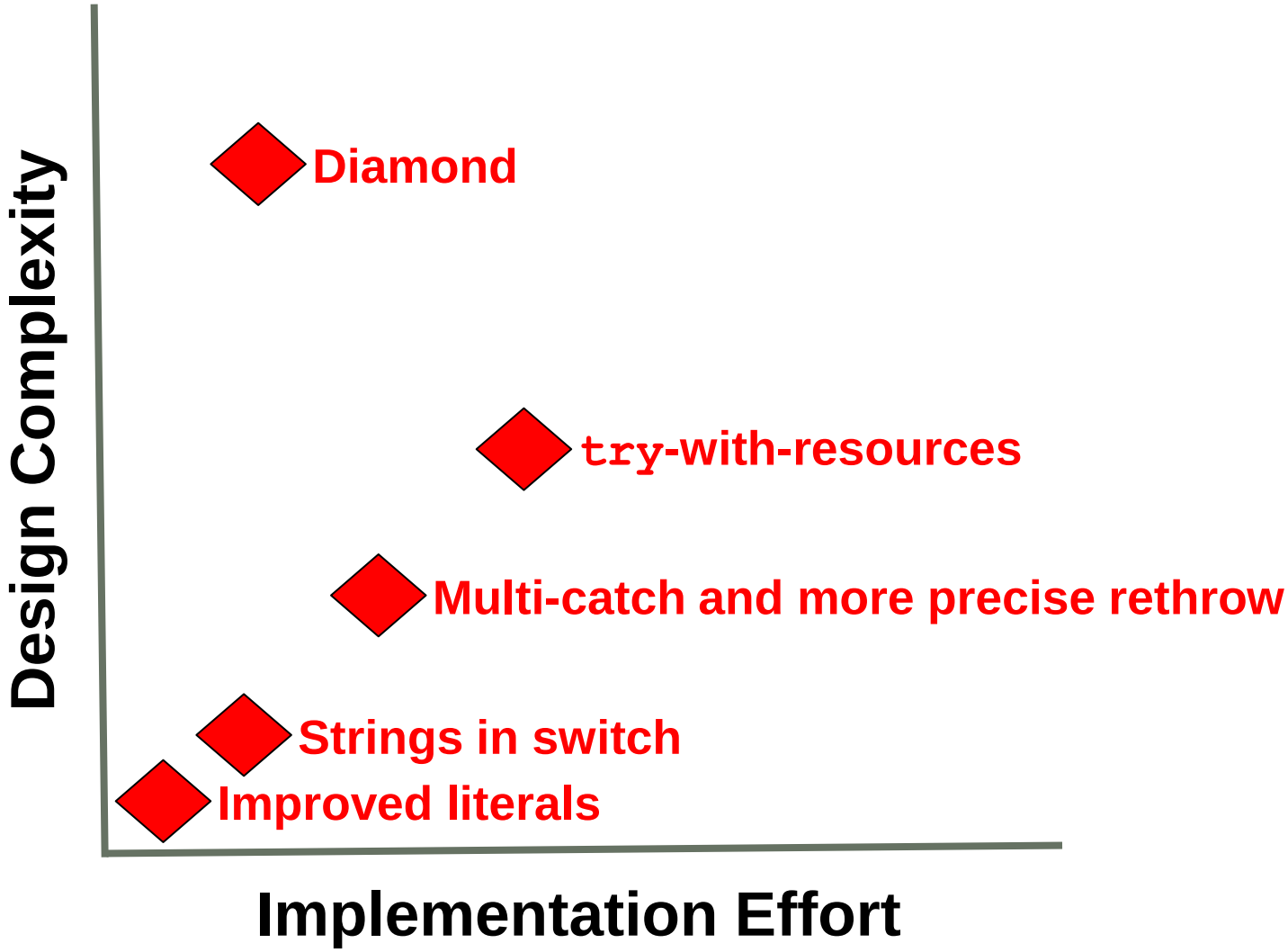
```
class a$1 extends Box<List<capture of ?>>{...}
```

Therefore, disallow non-denotable types in diamond inference.

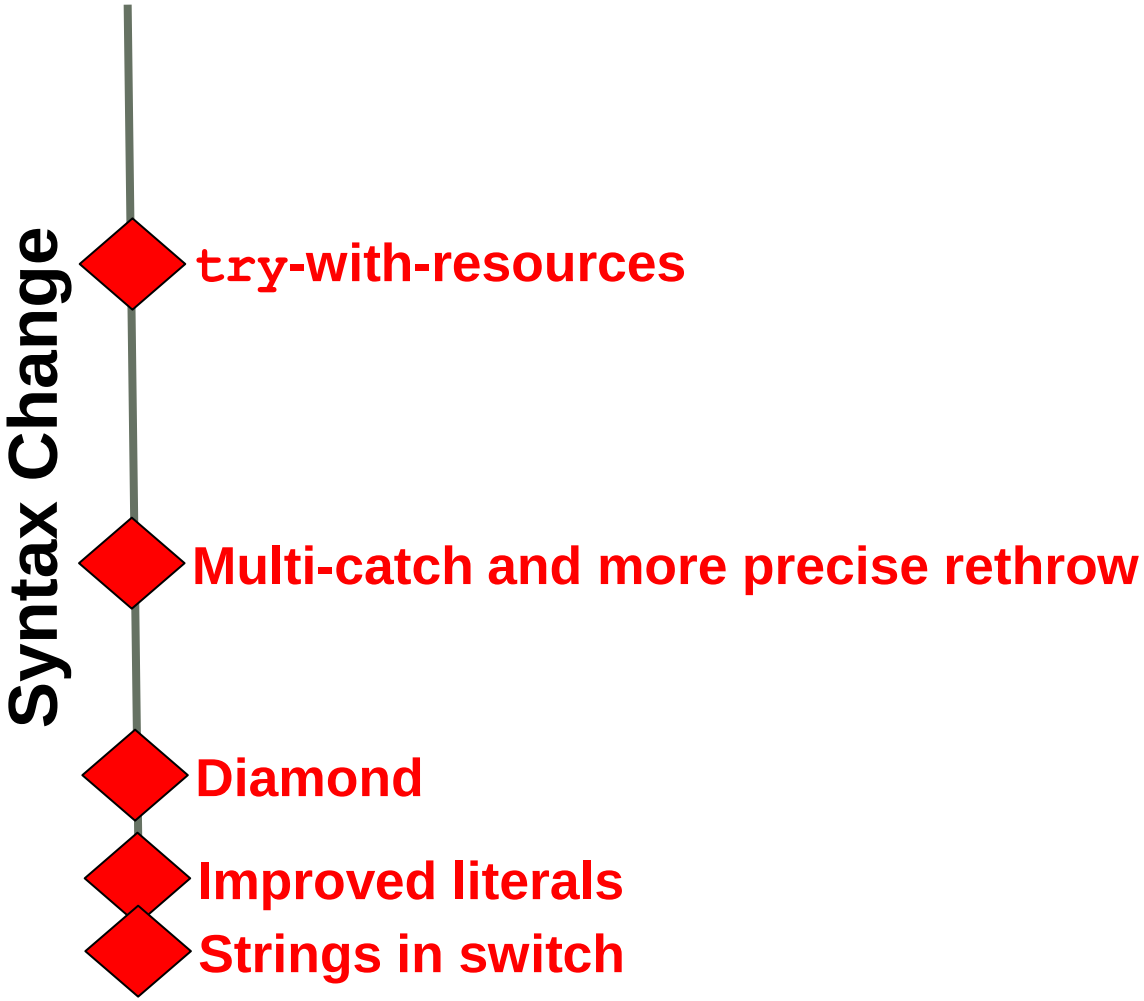
Lesson: keeping the future safe from the past

- Language features over time
  - Anonymous class in JDK 1.1 (1997)
  - Generics in JDK 5 (2004)
  - Diamond in JDK 7 builds (2009)
- Seemingly unrelated features can have deep semantic interactions!

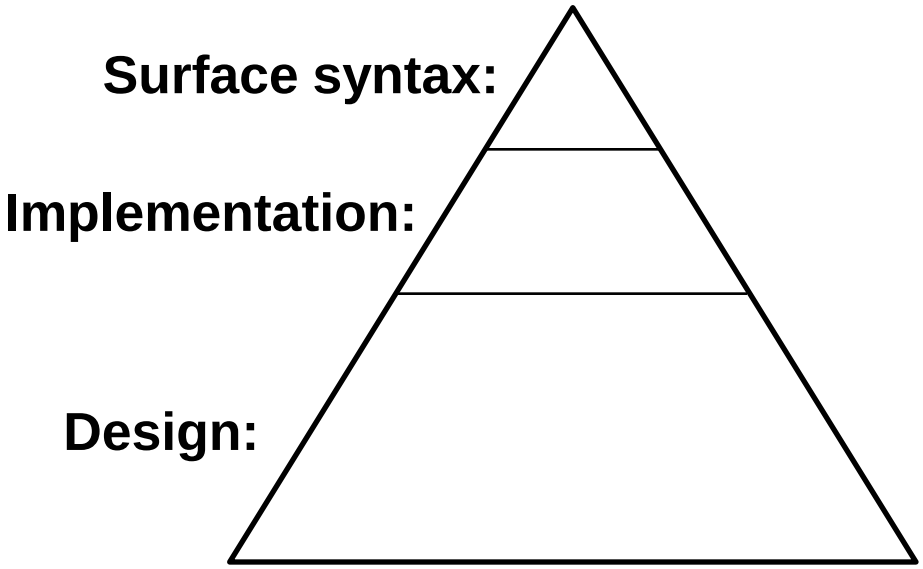
Sizing up the new features



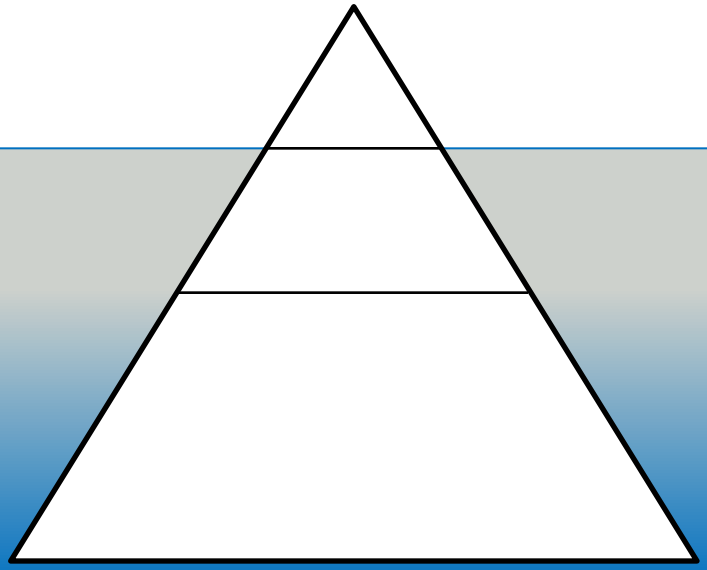
Sizing up the *syntax* of new features



Where does the effort go?



An iceberg!



Summary

- Coin features affect the *bodies* of methods, not their signatures
- Rounding off sharp corners of generics
  - Diamond
  - Varargs warnings
- Increase % of code for non-exceptional circumstances
  - Multi-catch
  - **`try`**-with-resources
- Consistency and clarity
  - Strings in switch
  - Literal improvements

Conclusions

- Features easier to use than to develop!
- Expect increasing use of quantitative design
- Tooling support important along the way
- Project Coin features
  - Remove superfluous text making programs more *readable*
  - Encourage writing programs that are more *reliable*
  - Play well with past and future changes
- Features ready to try, please give us feedback!
  http://jdk7.dev.java.net/
- JSR 334 EDR Draft available soon!

http://www.jcp.org/en/jsr/summary?id=334
http://openjdk.java.net/projects/coin
http://jdk7.dev.java.net/

Q & A

http://blogs.sun.com/darcy


ORACLE®

Appendix

Finding more dead code

```
try {
   throw new DaughterOfFoo();
} catch (Foo e) {
   try {
      throw e; // before, judged to throw Foo,
               // now throws DaughterOfFoo
   } catch (SonOfFoo anotherException) {
      // Reachable?
   }
}
```

How to infer?

- Multiple ways to perform type inference
  - What constraints are added?
  - Where do the constraints come from?
  - What context and locations are examined?
- What properties should an inference scheme have?
  - Effective
  - Consistent, few corner cases and interactions
  - Long term evolution

Example: captured-types

Compiler turns top-level wildcard into synthetic type-variables with upper/lower bounds

This process is known as *capture conversion*

When?

- Method conversion
- Member access

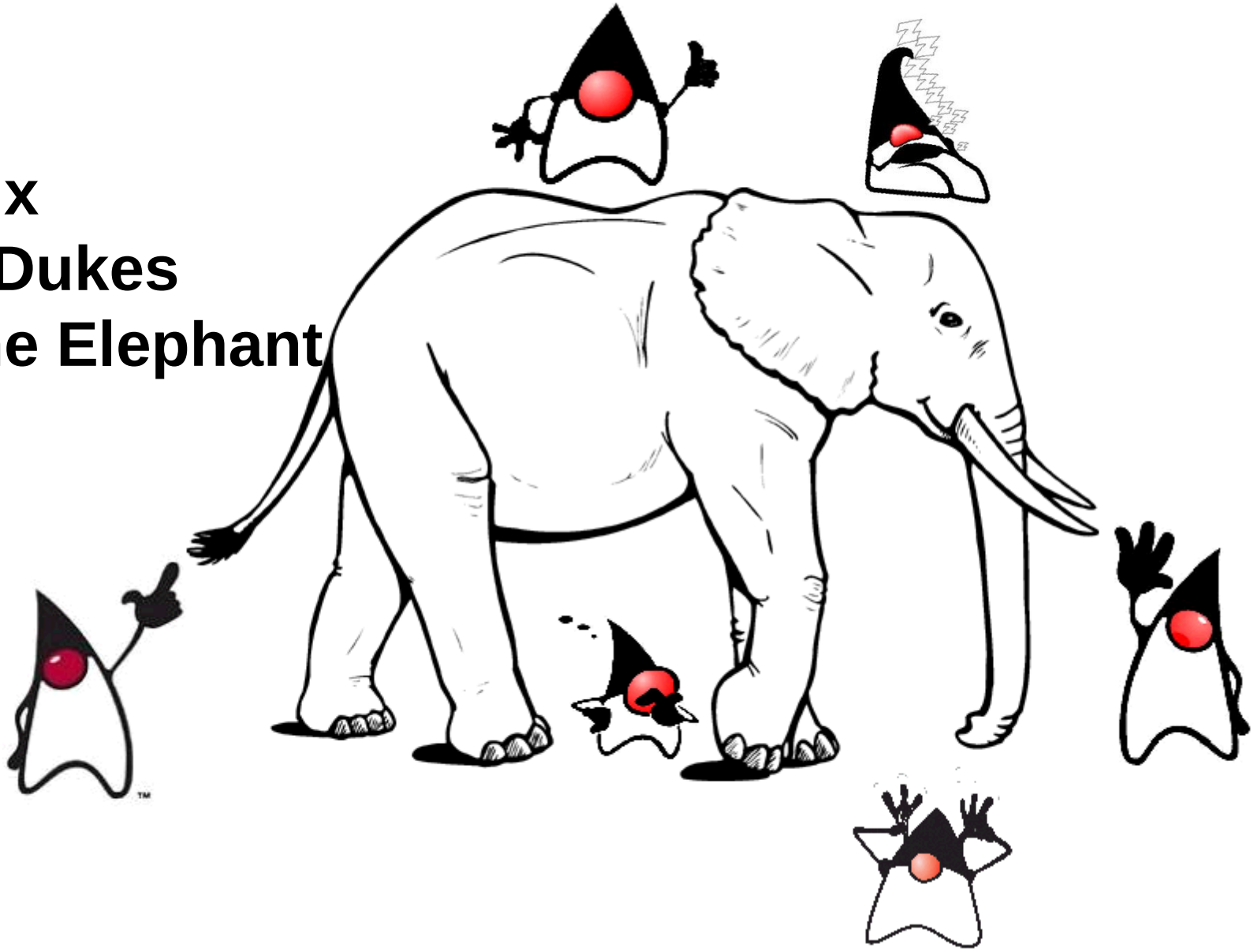
Non-trivial impact on method type-inference...

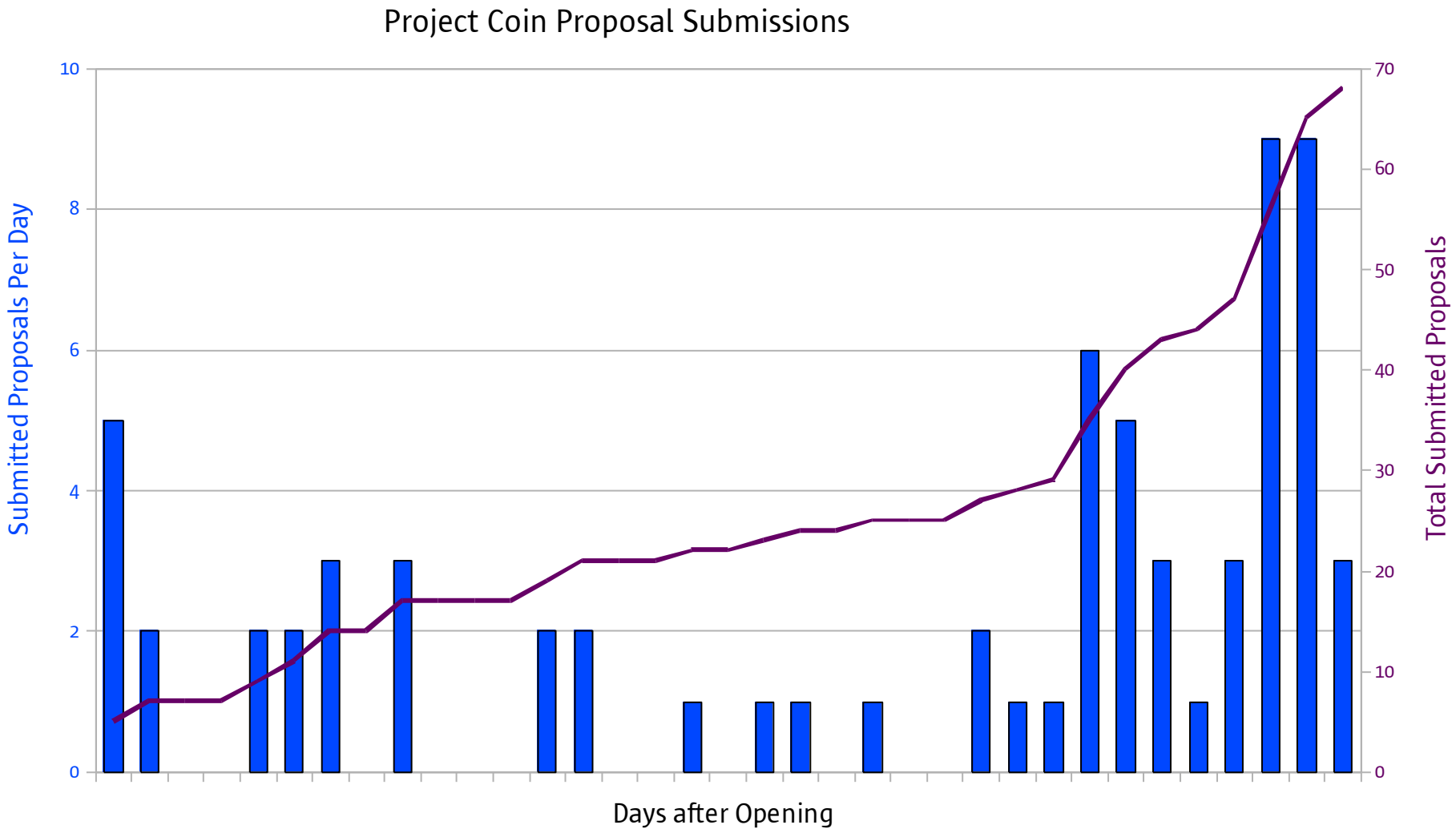
...and hence on diamond!

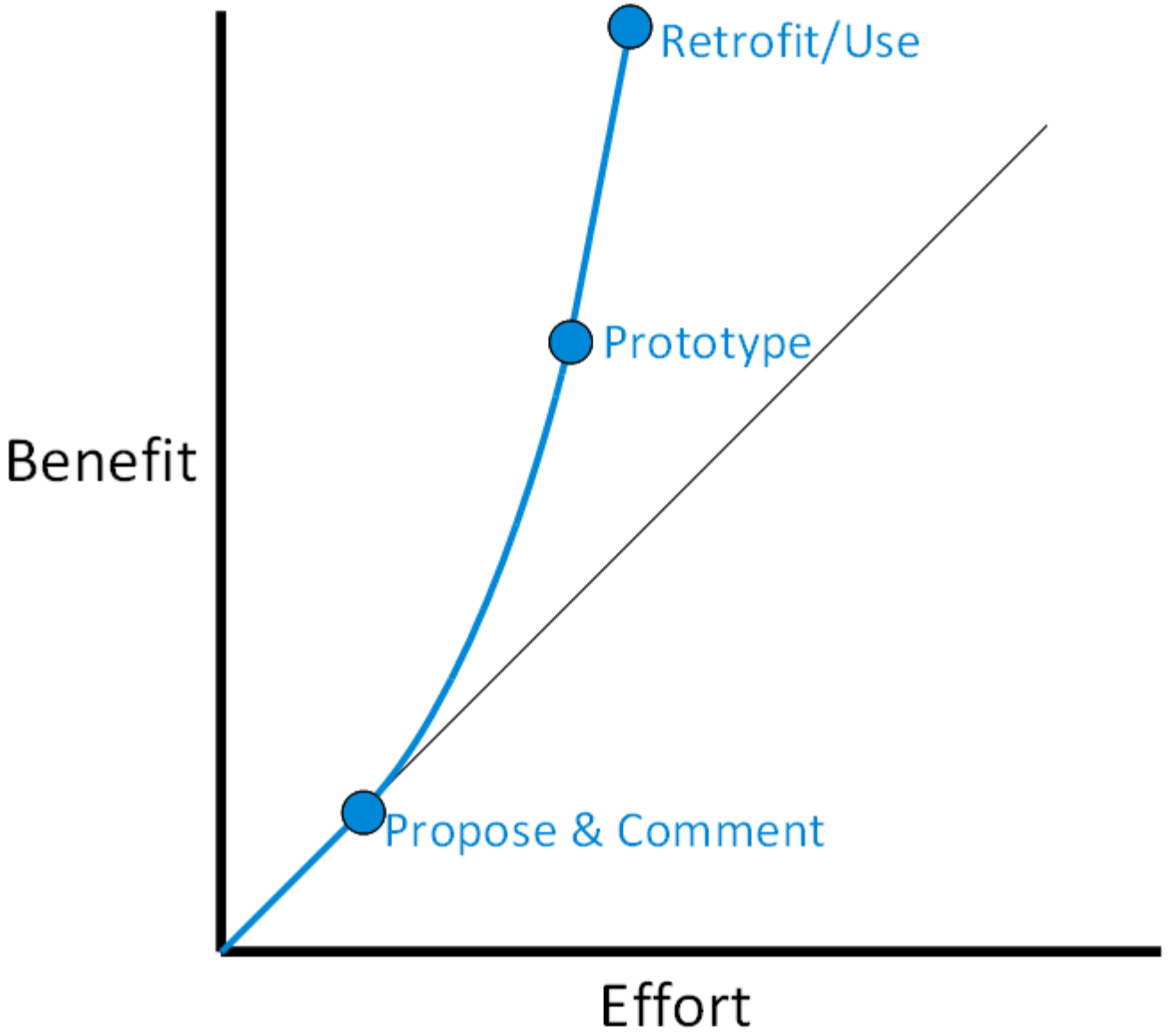
Net Present Value

- Good language changes today are better than the same good changes tomorrow!

The Six Blind Dukes and the Elephant




Project Coin Proposal Submissions
Submitted Proposals Per Day
Total Submitted Proposals
Days after Opening
0
2
4
6
8
10
0
10
20
30
40
50
60
70


Retrofit/Use
Prototype
Benefit
Propose & Comment
Effort

Sizing up JDK 5 Language changes

Normal maintenance:	**Tiny**
Hexadecimal floating-point literals	**Very small**
static import	**Small**
`for-each` loop	**Small**
`enum` types	**Medium**
Autoboxing and unboxing	**Medium**
Annotation types	**Large**
Generics	**Huge**