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
• 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
  *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 \texttt{int} value

\texttt{0x5045 \_ 1456}
What is special about the `int` value

0x50 45 14 56
What is special about the int value

0b01010000010001010001010110
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!
```java
text
```
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 "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 ClassCastException 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

```java
List list = new ArrayList();
```
With Generics

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

```java
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<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<>();
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

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

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

```java
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();
}
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();
}
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
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
  ```java
  try (new MyIoClose() {…})
  ```
  what should be the type of the synthetic variable?
  ```java
  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!
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
Strings in switch Project Coin proposal form

Examples

Compilation, cont.

Different APIs

Migration

Compatibility

References, prototype?
Strings in switch, implementation and tests
Strings in switch, implementation only
// 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:
  ```java
  public static <T> List<T> asList(T... a)
  ```
What’s in the box?

... = new Box<>(42);
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

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

```
Integer
Number
Object
Comparable<?>
Object & Comparable<? extends...
...
```
Complex algorithm

Box<? extends Number> b = new Box<> (42)
A distinction with a difference

Simple: \( \text{Box<Number> b = new Box<>}(42) \)

Complex: \( \text{Box<Number> b = new Box<>}(42) \)
          \[
\begin{align*}
\text{Integer} \\
\text{Number} \\
\text{Object} \\
\text{Comparable<??>}
\end{align*}
\]
          \[
\text{Object & Comparable<?? extends...>}
\]
          ...
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

```java
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

```java
void m(Box<Integer> box){...}
```

Simple:  

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

Complex:  

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

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

<table>
<thead>
<tr>
<th></th>
<th>OpenJDK</th>
<th>Tomcat</th>
<th>NetBeans</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total <strong>new’s</strong></td>
<td>104,138</td>
<td>6,048</td>
<td>94,768</td>
</tr>
<tr>
<td>Generics <strong>new’s</strong></td>
<td>5,076</td>
<td>153</td>
<td>12,010</td>
</tr>
<tr>
<td>Simple Success</td>
<td>4,409</td>
<td>148</td>
<td>10,670</td>
</tr>
<tr>
<td>Complex Success</td>
<td>4,533</td>
<td>148</td>
<td>11,085</td>
</tr>
</tbody>
</table>
## Analysis

<table>
<thead>
<tr>
<th></th>
<th>OpenJDK</th>
<th>Tomcat</th>
<th>NetBeans</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total <em>new’s</em></td>
<td>104,138</td>
<td>6,048</td>
<td>94,768</td>
</tr>
<tr>
<td>Generics <em>new’s</em></td>
<td>5,076</td>
<td>153</td>
<td>12,010</td>
</tr>
<tr>
<td>Simple Success</td>
<td>4,409</td>
<td>148</td>
<td>10,670</td>
</tr>
<tr>
<td>Complex Success</td>
<td>4,533</td>
<td>148</td>
<td>11,085</td>
</tr>
</tbody>
</table>

- 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

```java
List<?> arg = ...;
new Box<>(arg);
```
A surprise: why is this disallowed?

```java
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

- 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

- 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

- 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, `partially`-Denotable
How is \( T \) inferred?

\[
<T> \text{ List}<T> \text{ asList}(T\ldots t)\{\ldots\}
\]

\[
\text{List}<??> \text{ arg} = \ldots;
\text{Arrays.asList(}\text{arg})\;
\]
Don’t mistreat captured types!

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

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

```java
T == List<#capture of ?>
```
How to break a diamond

List<??> arg = ...;
new Box<> (arg);
Not just a copy...

```java
List<?> arg = ...;
new Box<List<?>>(arg);
```
Not just a copy...

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

```java
List<?> arg = ...;
new Box<List<capture of ?>>>(arg) { ... };
```
How does this get compiled?

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

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

Anonymous classes translate into a new class file with a full set of attributes.
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)
  - Diamond in JDK 7 builds (2009)

- Seemingly unrelated features can have deep semantic interactions!
Sizing up the new features

- Improved literals
- Implementation Effort
- Design Complexity
- Strings in switch
- Multi-catch and more precise rethrow
- try-with-resources
- Improved literals
- Diamond
Sizing up the *syntax* of new features

- try-with-resources
- Multi-catch and more precise rethrow
- Diamond
- Improved literals
- Strings in switch
Where does the effort go?

Surface syntax:

Implementation:

Design:
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://openjdk.java.net/projects/coin
http://jdk7.dev.java.net/

Q & A

http://blogs.sun.com/darcy
Finding more dead code

```java
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

Submitted Proposals Per Day

Total Submitted Proposals
Sizing up JDK 5 Language changes

Normal maintenance:
Hexadecimal floating-point literals
static import
for-each loop
enum types
Autoboxing and unboxing
Annotation types
Generics