Adding Generics to the Java Programming Language: Participant Draft Specification

Gilad Bracha, Sun Microsystems
Norman Cohen, IBM
Christian Kemper, Inprise
Steve Marx, WebGain
Martin Odersky, EPFL
Sven-Eric Panitz, Software AG
David Stoutamire, Sun Microsystems
Kresten Thorup, Aarhus University
Philip Wadler, Lucent Technologies

April 27, 2001

1 Summary

We propose to add generic types and methods to the Java programming language.

The main benefit of adding genericity to the Java programming language lies in the added expressiveness and safety that stems from making type parameters explicit and making type casts implicit. This is crucial for using libraries such as collections in a flexible, yet safe way.

The proposed extension is designed to be fully backwards compatible with the current language, making the transition from non-generic to generic programming very easy. In particular, one can retrofit existing library classes with generic interfaces without changing their code.

The present specification evolved from the GJ proposal which has been presented and motivated in a previous paper [BOSW98].

The rest of this specification is organized as follows. Section 2 explains how parameterized types are declared and used. Section 3 explains polymorphic methods. Section 4 explains how parameterized types integrate with exceptions. Section 5 explains what changes in Java's expression constructs. Section 6 explains how thee extended language translated into the class file format of the Java Virtual Machine. Section 7 explains how generic type information is stored in classfiles. Where possible, we follow the format and conventions the Java Language Specification (JLS) [GJSB96].

2 Types

There are two new forms of types: *parameterized types* and *type variables.*

2.1 Type Syntax

A parameterized type consists of a class or interface type C and a parameter section $\langle T_1, \ldots, T_n \rangle$. C must be the name of a parameterized class or interface, the types in the parameter list $\langle T_1, \ldots, T_n \rangle$ must match the number of declared parameters of C, and each actual parameter must be a subtype of the formal parameter's bound types.

In the following, whenever we speak of a class or interface type, we include the parameterized version as well, unless explicitly excluded.

A type variable is an unqualified identifier. Type variables are introduced by parameterized class and interface declarations (Section 2.2) and by polymorphic method declarations (Section 3.1).

Syntax (see JLS, Sec. 4.3 and 6.5)

```
ReferenceType        ::= ClassOrInterfaceType
                      |  ArrayType
                      |  TypeVariable

TypeVariable         ::= Identifier

ClassOrInterfaceType ::= ClassOrInterface TypeArgumentsOpt

ClassOrInterface     ::= Identifier
                      |  ClassOrInterfaceType . Identifier

TypeArguments        ::= < ReferenceTypeList >

ReferenceTypeList    ::= ReferenceType
                      |  ReferenceTypeList , ReferenceType
```

Example 1 Parameterized types.

```
Vector<String>
Seq<Seq<A>>
Seq<String>.Zipper<Integer>
Collection<Integer>
Pair<String,String>

// Vector<int> -- illegal, primitive types cannot be parameters
// Pair<String> -- illegal, not enough parameters
// Pair<String,String,String> -- illegal, too many parameters
```

2.2 Parameterized Type Declarations

A parameterized class or interface declaration defines a set of types, one for each possible instantiation of the type parameter section. All parameterized types share the same class or interface at runtime. For instance, the code

```
Vector<String> x = new Vector<String>();
Vector<Integer> y = new Vector<Integer>();
return x.getClass() == y.getClass();
```

will yield `true`.

Syntax (see JLS, Secs. 8.1, 9.1)

```
ClassDeclaration     ::= ClassModifiersOpt class Identifier TypeParametersOpt
                         SuperOpt InterfacesOpt ClassBody

InterfaceDeclaration ::= InterfaceModifiersOpt interface Identifier TypeParametersOpt
                         ExtendsInterfacesOpt InterfaceBody

TypeParameters       ::= < TypeParameterList >

TypeParameterList    ::= TypeParameterList , TypeParameter
                      |  TypeParameter

TypeParameter        ::= TypeVariable TypeBoundOpt

TypeBound            ::= extends ClassOrInterfaceType AdditionalBoundListopt

AdditionalBoundList  ::= AdditionalBound AdditionalBoundList
                      | AdditionalBound

AdditionalBound      ::= & InterfaceType
```

The type parameter section follows the class name and is delimited by `< >` brackets. It defines one or more type variables that act as parameters. Type parameters have an optional bound $T\&I_1\&...\&I_n$. The bound consists of a class or interface type T and possibly further interface types $I_1, \ldots, I_n$. If no bound is given for a type parameter, `java.lang.Object` is assumed. The erasures of all constituent types of a bound must be pairwise different. The order of types in a bound does not matter, except for the requirement that a class type may only appear in first position. For instance, swapping two interfaces in a bound yields the same bound.

If a type parameter X has more than one bound, then it is a compile-time error to reference a member of an object whose declared type is X, unless that member is an accessible member of class `java.lang.Object`.

This restriction avoids possibly ambiguous member references. In such cases, it is always possible to cast the object to the bound type explicitly and then reference the desired member. If the type variable has a single bound, no such restriction applies.

The scope of a type parameter is all of the declared class, except any static members or initializers, but including the type parameter section itself. Therefore, type parameters can appear as parts of their own bounds, or as bounds of other type parameters declared in the same section.

Example 2 Mutually recursive type variable bounds.

```
interface ConvertibleTo<A> {
   A convert();
}
class ReprChange<A implements ConvertibleTo<B>,
                 B implements ConvertibleTo<A>> {
   A a;
```

```
    void set(B x) { a = x.convert(); }
    B get() { return a.convert(); }
 }
```

Parameterized declarations can be nested inside ther declarations.

Example 3 Nested parameterized class declarations.

```
 class Seq<A> {
    A head;
    Seq<A> tail;
    Seq() { this(null, null); }
    boolean isEmpty() { return tail == null; }
    Seq(A head, Seq<A> tail) { this.head = head; this.tail = tail; }
    class Zipper<B> {
       Seq<Pair<A,B>> zip(Seq<B> that) {
      if (this.isEmpty() || that.isEmpty())
              return new Seq<Pair<A,B>>();
      else
              return new Seq<Pair<A,B>>(
                  new Pair<A,B>(this.head, that.head),
              this.tail.zip(that.tail));
       }
    }
 }
 class Client {{
    Seq<String> strs =
       new Seq<String>("a", new Seq<String>("b", new Seq<String>()));
    Seq<Number> nums =
       new Seq<Number>(new Integer(1),
                       new Seq<Number>(new Double(1.5),
                                       new Seq<Number>()));
    Seq<String>.Zipper<Number> zipper = strs.new Zipper<Number>();
    Seq<Pair<String,Number>> combined = zipper.zip(nums);
 }}
```

Since the throw and catch mechanism of the JVM works only with unparameterized classes, we require that parameterized types may not inherit directly or indirectly from `java.lang.Throwable`.

2.3 Handling Consecutive Type Parameter Brackets

Consecutive type parameter brackets `<` and `>` do not need to be separated by white-space. This leads to a problem in that the lexical analyzer will map the two consecutive closing angle brackets in a type such as `Vector<Seq<String>>` to the right-shift symbol `>>`. Similarly, three consecutive closing angle brackets would be recognized as a unary right-shift symbol `>>>`. To make up for this irregularity, we refine the grammar for types and type parameters as follows.

Syntax (see JLS, Sec. 4)

```
 ReferenceType        ::= ClassOrInterfaceType
```

```
                        |  ArrayType
                        |  TypeVariable

ClassOrInterfaceType ::= Name
                        |  Name < ReferenceTypeList1

ReferenceTypeList1   ::= ReferenceType1
                        |  ReferenceTypeList , ReferenceType1

ReferenceType1       ::= ReferenceType >
                        |  Name < ReferenceTypeList2

ReferenceTypeList2   ::= ReferenceType2
                        |  ReferenceTypeList , ReferenceType2

ReferenceType2       ::= ReferenceType >>
                        |  Name < ReferenceTypeList3

ReferenceTypeList3   ::= ReferenceType3
                        |  ReferenceTypeList , ReferenceType3

ReferenceType3       ::= ReferenceType >>>

TypeParameters       ::= < TypeParameterList1

TypeParameterList1   ::= TypeParameter1
                        |  TypeParameterList , TypeParameter1

TypeParameter1       ::= TypeParameter >
                        |  TypeVariable extends ReferenceType2
                        |  TypeVariable implements ReferenceType2
```

2.4 Subtypes, Supertypes, Member Types

In the following, assume a class declaration C with parameters $A_1, ..., A_n$ which have bounds $B_1, ..., B_n$. That class declaration defines a set of types $C\langle T_1, ..., T_n\rangle$, where each argument type T_i ranges over all types that are subtypes of all types listed in the corresponding bound. That is, for each bound type S_i in B_i, T_i is a subtype of

$$S_i[A_1 := T_1, ..., A_n := T_n] \ .$$

Here, $[A := T]$ denotes substitution of the type variable A with the type T.

The definitions of subtype and supertype are generalized to parameterized types and type variables. Given a class or interface declaration for $C\langle A_1, ..., A_n\rangle$, the *direct supertypes* of the parameterized type $C\langle A_1, ..., A_n\rangle$ are

- the type given in the extends clause of the class declaration if an extends clause is present, or `java.lang.Object` otherwise, and
- the set of types given in the implements clause of the class declaration if an implements clause is present.

The direct supertypes of the type $C\langle T_1, ..., T_n\rangle$ are $D\langle U_1\theta, ..., U_k\theta\rangle$, where

- $D\langle U_1, ..., U_k\rangle$ is a direct supertype of $C\langle A_1, ..., A_n\rangle$, and
- θ is the substitution $[A_1 := T_1, ..., A_n := T_n]$.

The direct supertypes of a type variable are the types listed in its bound. The *supertypes* of a type are obtained by transitive closure over the direct supertype relation. The *subtypes* of a type T are all types U such that T is a supertype of U.

Subtyping does not extend through parameterized types: T a subtype of U does not imply that $C\langle T\rangle$ is a subtype of $C\langle U\rangle$.

To support translation by type erasure, we impose the restriction that a class or type variable may not at the same time be a subtype of two interface types which are different parameterizations of the same interface. Hence, every superclass and implemented interface of a parameterized type or type variable can be augmented by parameterization to exactly one supertype. Here is an example of an illegal multiple inheritance of an interface:

```
class B implements I<Integer>
class C extends B implements I<String>
```

A consequence of the parameterized types concept is that now the type of a class member is no longer fixed, but depends on the concrete arguments substituted for the class parameters. Here are the relevant definitions. Assume again a class or interface declaration of C with parameters $A_1, ..., A_n$.

- Let M be a member declaration in C, whose type as declared is T. Then the type of M in the type $C\langle T_1, ..., T_n\rangle$ is $T[A_1 := T_1, ..., A_n := T_n]$.
- Let M be a member declaration in D, where D is a class extended by C or an interface implemented by C. Let $D\langle U_1, ..., U_k\rangle$ be the supertype of $C\langle T_1, ..., T_n\rangle$ that corresponds to D. Then the type of M in $C\langle T_1, ..., T_n\rangle$ is the type of M in $D\langle U_1, ..., U_k\rangle$.

2.5 Raw Types

To facilitate interfacing with non-generic legacy code, it is also possible to use as a type the erasure of a parameterized class without its parameters. Such a type is called a *raw type*. Variables of a raw type can be assigned from values of any of the type's parametric instances. For instance, it is possible to assign a `Vector<String>` to a `Vector`. The reverse assignment from `Vector` to `Vector<String>` is unsafe from the standpoint of the generic semantics (since the vector might have had a different element type), but is still permitted in order to enable interfacing with legacy code. In this case, a compiler will issue a warning message that the assignment is deprecated.

The superclasses (respectively, interfaces) of a raw type are the raw versions of the superclasses (interfaces) of any of its parameterized instances.

The type of a member declaration M in a raw type C is its erased type (see Section 6.1). However, to make sure that potential violations of the typing rules are always flagged, some accesses to members of a raw type will result in "unchecked" warning messages. The rules for generating unchecked warnings for raw types are as follows:

- A method call to a raw type generates an unchecked warning if the erasure changes the argument types.
- A field assignment to a raw type generates an unchecked warning if erasure changes the field type.

No unchecked warning is required for a method call when the argument types do not change (even if the result type and/or throws clause changes), for reading from a field, or for a class instance creation of a raw type.

The supertype of a class may be a raw type. Member accesses for the class are treated as normal, and member accesses for the supertype are treated as for raw types. In the constructor of the class, calls to `super` are treated as method calls on a raw type.

Example 4 Raw types.

```
class Cell<A>
  A value;
  Cell (A v) { value=v; }
  A get() { return value; }
  void set(A v) { value=v; }
}

Cell x = new Cell<String>("abc");
x.value;          // OK, has type Object
x.get();          // OK, has type Object
x.put("def");     // deprecated
```

3 Generic Methods

3.1 Method Declarations

Syntax (See JLS 8.4)

```
MethodHeader   ::= MethodModifiersOpt TypeParametersOpt ResultType MethodDeclarator
                   ThrowsOpt

ResultType     ::= VOID
                |  Type
```

Method declarations can have a type parameter section like classes have. The parameter section precedes the result type of the method.

Example 5 Generic methods.

```
static <Elem> void swap(Elem[] a, int i, int j) {
   Elem temp = a[i]; a[i] = a[j]; a[j] = temp;
}

<Elem implements Comparable<Elem>> void  sort(Elem[] a) {
   for (int i = 0; i < xs.length; i++)
      for (int j = 0; j < i; j++)
         if (a[j].compareTo(a[i]) < 0) <Elem>swap(a, i, j);
}

class Seq<A> {

   <B> Seq<Pair<A,B>> zip(Seq<B> that) {
      if (this.isEmpty() || that.isEmpty())
         return new Seq<Pair<A,B>>();
      else
         return new Seq<Pair<A,B>>(
            new Pair<A,B>(this.head, that.head),
            this.tail.<B>zip(that.tail));
   }
}
```

It is illegal to declare two methods with the same name and the same argument types in a class. The definition of "having the same argument types" is extended to generic methods as follows:

Two method declarations M and N *have the same argument types* if either none of them has type parameters and their argument types agree, or they have the same number of type parameters, say $\langle A_1, ..., A_n\rangle$ for M and $\langle B_1, ..., B_n\rangle$ for N, and after renaming each occurrence of a B_i in $N's$ type to A_i the bounds of corresponding type variables are the same and the argument types of M and N agree.

3.2 Overriding

The definition of overriding is adapted straightforwardly to parameterized types:

A class or interface $C\langle A_1, ..., A_n\rangle$ may contain a declaration for a method with the same name and the same argument types as a method declaration in one of the supertypes of $C\langle A_1, ..., A_n\rangle$. In this case, the declaration in C is said to (directly) override the declaration in the supertype.

This specification requires that the result type of a method is a subtype of the result types of all methods it overrides. This is more general than previous specifications of the Java programming language, which require the result types to be identical. See Section 6.2 for an implementation scheme to support this generalization.

Example 6 The following declarations are legal according to the present specification, yet illegal according to the JLS.

```
class C implements Cloneable {
   C copy() { return (C)clone(); }
   ...
}
class D extends C implements Cloneable {
   D copy() { return (D)clone(); }
   ...
}
```

The relaxed rule for overriding also allows one to relax the conditions on abstract classes implementing interfaces. The new rule is as follows.

A class can inherit multiple methods with the same name and signature, provided they are all abstract, the class does not override or implement them, and one of the methods has a return type which is a subtype of the return type of each of the other methods.

By contrast, under the current JLS, all return types are required to be equal.

4 Exceptions

To enable a direct mapping into the class file format of the JVM, type variables are not allowed in `catch` clauses, but they are allowed in `throws` lists.

Example 7 Generic Throws Clause.

```
interface PrivilegedExceptionAction<E extends Exception> {
  void run() throws E;
}

class AccessController {
  public static <E extends Exception>
  Object doPrivileged(PrivilegedExceptionAction<E> action) throws E
  { ... }
}

class Test {
  public static void main(String[] args) {
    try {
      AccessController.doPrivileged(
        new PrivilegedExceptionAction<FileNotFoundException>() {
          public void run() throws FileNotFoundException
          {... delete a file  ...}
        });
    } catch (FileNotFoundException f) {...} // do something
  }
}
```

5 Expressions

5.1 Class Instance Creation Expressions

A class instance creation expression for a parameterized class consists of the fully parameterized type of the instance to be created and arguments to a constructor for that type.

Syntax (see JLS, Sec. 15.9)

```
ClassInstanceCreationExpression ::= new ClassOrInterfaceType TypeArgumentsOpt
                                    ( ArgumentListOpt ) ClassBodyOpt
                                 |  Primary.new Identifier TypeArgumentsOpt
                                    ( ArgumentListOpt ) ClassBodyOpt
```

Example 8 Class instance creation expressions.

```
new Vector<String>();

new Pair<Seq<Integer>,Seq<String>>(
   new Seq<Integer>(new Integer(0), new Seq<Integer>()),
   new Seq<String>("abc", new Seq<String>()));
```

5.2 Array Creation Expressions

The element type in an array creation expression is a fully parameterized type. Creating an array whose element type is a type variable generates an "unchecked" warning at compile-time.

Syntax (see JLS, Sec. 15.10)

```
ArrayCreationExpression ::= new PrimitiveType DimExprs DimsOpt
                         |  new ClassOrInterfaceType DimExprs DimsOpt
                         |  new PrimitiveType Dims ArrayInitializer
                         |  new ClassOrInterfaceType Dims ArrayInitializer
```

Example 9 Array creation expressions.

```
new Vector<String>[n]
new Seq<Character>[10][20][]
```

5.3 Cast Expressions

The target type for a cast can be a parameterized type.

Syntax (see JLS, Sec. 15.16)

```
CastExpression ::= ( PrimitiveType DimsOpt ) UnaryExpression
                |  ( ReferenceType ) UnaryExpressionNotPlusMinus
```

The usual rules for casting conversions (Spec, Sec. 5.5) apply. Since type parameters are not maintained at run-time, we have to require that the correctness of type parameters given in the target type of a cast can be ascertained statically. This is enforced by refining the casting conversion rules as follows:

A value of type S can be cast to a parameterized type T if one of the following two conditions holds:

- T is a subtype of S, and there are no other subtypes of S with the same *erasure* (see Section 6.1) as T.
- T is a supertype of S. The restriction on multiple interface inheritance in Section 2.2 guarantees that there will be no other supertype of S with the same erasure as T.

Note that even when parameterized subtypes of a given type are not unique, it will always be possible to cast to the raw type given by their common erasure.

Example 10 Assume the declarations

```
class Dictionary<A,B> extends Object { ... }
class Hashtable<A,B> extends Dictionary<A, B> { ... }

Dictionary<String,Integer> d;
Object o;
```

Then the following are legal:

```
(Hashtable<String,Integer>)d   // legal, has type: Hashtable<String,Integer>
(Hashtable)o                   // legal, has type: Hashtable
```

But the following are not:

```
(Hashtable<Float,Double>)d     // illegal, not a subtype
(Hashtable<String,Integer>)o   // illegal, not unique subtype
```

5.4 Type Comparison Operator

Type comparison can involve parameterized types. The rules of casting conversions, as defined in Section 5.3, apply.

Syntax (see JLS, Sec. 15.20.2)

```
RelationalExpression ::= ...
                      |  RelationalExpression instanceof ReferenceType
```

Example 11 Type comparisons.

```
class Seq<A> implements List<A> {
   static boolean isSeq(List<A> x) {
      return x instanceof Seq<A>
   }
   static boolean isSeq(Object x) {
      return x instanceof Seq
   }
   static boolean isSeqArray(Object x) {
      return x instanceof Seq[]
   }
}
```

Example 12 Type comparisons and type casts with type constructors.

```
class Pair<A, B> {

   A fst; B snd;

   public boolean equals(Object other) {
      return
     other instanceof Pair &&
     equals(fst, ((Pair)other).fst) &&
     equals(snd, ((Pair)other).snd);
```

```
    }

    private boolean equals(Object x, Object y) {
       return x == null && y == null || x != null && x.equals(y);
    }
}
```

5.5 Generic Method Invocation

Generic method invocations do not have special syntax. Type parameters of generic methods are elided; they are inferred from value parameters according to the rules given in Section 5.6.

Syntax (See JLS 15.12)

```
MethodInvocation ::= MethodExpr ( ArgumentListOpt )
MethodExpr       ::= MethodName
                  |  Primary . Identifier
                  |  super . Identifier
                  |  ClassName.super . Identifier
```

Example 13 Generic method calls. (see Example 5)

```
swap(ints, 1, 3)
sort(strings)
strings.zip(ints)
```

5.6 Type Parameter Inference

Type parameters of a call to a generic method are inferred from the call's value parameters. To make inference work, we make use of the null type (See JLS 4.1). The null type cannot be written in a Java program, but we shall denote it by `*` in this specification. The null type is a subtype of every reference type. We postulate that the subtyping relationship between `*` and reference types is promoted through constructors: If T and U are identical types except that where T has occurrences of `*` U has occurrences of other reference types, then T is a subtype of U.

Type parameter inference inserts the most specific type parameters such that

1. Each actual argument type is a subtype of the corresponding formal argument type,
2. No type variable that occurs more than once in the method's result type is instantiated to a type containing `*`.

It is an error if most specific parameter types do not exist or are not unique.

Example 14 Type parameter inference.

Assume the generic method declarations:

```
static <A> Seq<A> nil() { return new Seq<A>(); }
static <A> Seq<A> cons(A x, Seq<A> xs) { return new Seq<A>(x, xs); }
```

Then the following are legal expressions:

```
cons("abc", nil())                                // of type:    Seq<String>
cons(new IOException(), cons(new Error(), nil())) // of type:    Seq<Throwable>
nil();                                            // of type:    Seq<*>
cons(null, nil());                                // of type:    Seq<*>
```

The second restriction above needs some explanation. Note that a covariance rule applies to types containing `*`. For any type context TC, type U, $TC[*]$ is a subtype of $TC[U]$. General covariance leads to unsound type systems, so we have to argue carefully that our type system with its restricted form of covariance is still sound. The soundness argument goes as follows: since one cannot declare variables of type $TC\langle * \rangle$, all one can do with a value of that type is assign or pass it once to a variable or parameter of some other type. There are now three possibilities, depending on the variable's type:

- The variable's type is an unparameterized supertype of the raw type TC. In this case the assignment is clearly sound.
- The variable's type is $TC'\langle T \rangle$ for some type T and supertype TC' of TC. Now, the only value of `*` is `null`, which is also a value of every reference type T. Hence, any value of type $TC\langle * \rangle$ will also be a value of type $TC'\langle T \rangle$, so the assignment is sound.
- The variable is a parameter p whose type is a type variable, A. Then code that accesses p in the method body is insensitive to the type of the actual parameter, so the method body itself cannot give rise to type errors. Furthermore, by restriction (2.) above, the method's formal result type will contain at most one occurrence of A, so the actual type of the method application is again of the form $TC'\langle * \rangle$, where TC' is a type context.

Without restriction (2.) type soundness would be compromised, as is shown by the following example.

Example 15 An unsafe situation for type parameter inference.

```
Pair<A,A> duplicate(A x) { return new Pair<A,A>(x, x); }

void crackIt(Pair<Seq<String>,Seq<Integer>> p) {
   p.fst.head := "hello";
   Integer i = p.snd.head;
}

crackIt(duplicate(cons(null, nil()))); // illegal!
```

This will effectively assign a String to an Integer. The problem is that the `duplicate` method returns the same value under two type parameters which then get matched against different types. I.e.

```
duplicate(cons(null, nil()))
```

has type

```
Pair<Seq<*>,Seq<*>>
```

but the two `Seq<*>` parameters really stand for the same object, hence it is unsound to widen these types to different `Seq` types. The compiler will report an error for the call

```
  crackIt(duplicate(cons(null, nil())));
```

The error message will state that an uninstantiated type parameter appears several times in the result type of a method.

6 Translation

In the following we explain how programs involving genericity are translated to JVM bytecodes. In a nutshell, the translation proceeds by erasing all type parameters, mapping type variables to their bounds, and inserting casts as needed. Some subtleties of the translation are caused by the handling of overriding.

6.1 Translation of Types

As part of its translation, a compiler will map every parameterized type to its type erasure. *Type erasure* is a mapping from (possibly generic) types to non-generic types. We write $|T|$ for the erasure of type T. The erasure mapping is defined as follows.

- The erasure of a parameterized type $T\langle T_1, ..., T_n\rangle$ is $|T|$.
- The erasure of a nested type $T.C$ is $|T|.C$.
- The erasure of an array type $T[\,]$ is $|T|[\,]$.
- The erasure of a type variable is
  - if the variable has a class type among its bounds, the erasure of that class type;
  - otherwise, if the bound consists of interface types only, the interface among the erasures of all interface types which has the least canonical name (see JLS 6.7), using lexicographic ordering.
- The erasure of every other type is the type itself.

6.2 Translation of Methods

Each method $T\,m(T_1, ..., T_n)\,\mathbf{throws}\,S_1, ..., S_m$ is translated to a method with the same name whose return type, argument types, and thrown types are the erasures of the corresponding types in the original method. In addition, if a method m of a class or interface C is inherited in a subclass D, a *bridge method* might need to be generated in D. The precise rules are as follows.

- If $C.m$ is directly overridden by a method $D.m$ in D, and the erasure of the return type or argument types of $D.m$ differs from the erasure of the corresponding types in $C.m$, a bridge method needs to be generated.

- A bridge method also needs to be generated if *C.m* is not directly overridden in *D*, unless *C.m* is abstract.

The type of the bridge method is the type erasure of the method in the base class or interface *C*. In the bridge method's body all arguments to the method will be cast to their type erasures in the extending class *D*, after which the call will be forwarded to the overriding method *D.m* (if it exists) or to the original method *C.m* (otherwise). No special handling of changes in erasures of thrown types are required, since throws clauses are not checked at load time or run time.

Example 16 Bridge methods.

```
class C<A> { abstract A id(A x); }
class D extends C<String> { String id(String x) { return x; } }
```

This will be translated to:

```
class C { abstract Object id(Object x); }
class D extends C {
   String id(String x) { return x; }
   Object id(Object x) { return id((String)x); }
}
```

Note that the translation scheme can produce methods with identical names and argument types, yet with different result types, all declared in the same class. Here's an example:

Example 17 Bridge methods with the same parameters as normal methods.

```
class C<A> { abstract A next(); }
class D extends C<String> { String next() { return ""; } }
```

This will be translated to:

```
class C { abstract Object next(); }
class D extends C<String> {
   String next/*1*/() { return ""; }
   Object next/*2*/() { return next/*1*/(); }
}
```

A compiler would reject that program because of the double declaration of `next`. But the bytecode representation of the program is legal, since the bytecode always refers to a method via its the full signature and therefore can distinguish between the two occurrences of `next`. Since we cannot make the same distinction in the Java source, we resorted to indices in /* ... */ comments to make clear which method a name refers to.

The same technique is used to implement method overriding with covariant return types[1].

Example 18 Overriding with covariant return types.

[1]covariant return types were at some time before version 1.0 part of the Java programming language but got removed later

```
class C { C dup(){...} }
class D extends C { D dup(){...} }
```

This translates to:

```
class C { C dup(); }
class D {
   D dup/*1*/(){...}
   C dup/*2*/(){ return dup/*1*/(); }
}
```

Since our translation of methods erases types, it is possible that different methods with identical names but different types are mapped to methods with the same type erasure. Such a case is considered to be an error in the original source program and must be rejected by a Java compiler. There are three rules to prevent signature clashes caused by the translation. Say a method declaration M indirectly overrides a method declaration M' if there is a (possibly empty) path of method declarations

$$M = M_0, M_1, ..., M_n = M'$$

such that M_i overrides M_{i+1}. Then one must have:

Rule 1 Methods declared in the same class with the same name must have different erasures.

Rule 2 If a method declaration M in class C has the same name and type erasure as a method declaration M' in a superclass D of C then M in C must indirectly override M' in D.

Rule 3 If a method declaration M in a superclass D of C has the same name and type erasure as a method declaration M' in an interface I implemented by C then there must be a method declaration M'' in C or one of its superclasses that indirectly overrides M in D and implements M' in I.

Example 19 Name clash excluded by Rule 2.

```
class C<A> { A id (A x) {...} }
class D extends C<String> {
   Object id(Object x) {...}
}
```

This is illegal since `C.id` and `D.id` have the same type erasure, yet `D.id` does not override `C.id`. Hence, Rule 2 is violated.

Example 20 Name clash excluded by Rule 3.

```
class C<A> { A id (A x) {...} }
interface I<A> { A id(A x); }
class D extends C<String> implements I<Integer> {
   String id(String x) {...}
   Integer id(Integer x) {...}
}
```

This is also illegal, since `C.id` and `I.id` have the same type erasure yet there is no single method in `D` that indirectly overrides `C.id` and implements `I.id`. Hence, Rule 3 is violated.

6.3 Translation of Expressions

Expressions are translated according to the JLS except that casts are inserted where necessary. There are two situations where a cast needs to be inserted.

1. Field access where the field's type is a type parameter. Example:

```
class Cell<A> { A value; A getValue(); }
...
String f(Cell<String> cell) {
   return cell.value;
}
```

Since the type erasure of `cell.value` is `java.lang.Object`, yet `f` returns a `String`, the return statement needs to be translated to

```
return (String)cell.value;
```

2. Method invocation, where the method's return type is a type parameter. For instance, in the context of the above example the statement

```
String x = cell.getValue();
```

needs to be translated to:

```
String x = (String)cell.getValue();
```

7 The `Signature` Classfile Attribute

Classfiles need to carry generic type information in a backwards compatible way. This is accomplished by introducing a new "Signature" attribute for classes, methods and fields. The structure of this attribute is as follows:

```
"Signature" (u4 attr-length, u2 signature-index)
```

When used as an attribute of a method or field, a signature gives the full (possibly generic) type of that method or field. When used as a class attribute, a signature indicates the type parameters of the class, followed by its supertype, followed by all its interfaces.

The type syntax in signatures is extended to parameterized types and type variables. There is also a new signature syntax for formal type parameters. The syntax extensions for signature strings are as follows:

Syntax

```
MethodOrFieldSignature ::= TypeSignature

ClassSignature         ::= ParameterPartOpt super_TypeSignature interface_TypeSignatures

TypeSignatures         ::= TypeSignatures TypeSignature
```

```
                        |

TypeSignature          ::= ...
                        |  ClassTypeSignature
                        |  MethodTypeSignature
                        |  TypeVariableSignature

ClassTypeSignature     ::= 'L' Ident TypeArgumentsOpt ';'
                        |  ClassTypeSignature '.' Ident ';' TypeArgumentsOpt

MethodTypeSignature    ::= TypeArgumentsOpt '(' TypeSignatures ')'
                           TypeSignature ThrowsSignatureListOpt

ThrowsSignatureList    ::= ThrowsSignature ThrowsSignatureList
                        |  ThrowsSignature

ThrowsSignature        ::= '^' TypeSignature

TypeVariableSignature  ::= 'T' Ident ';'

TypeArguments          ::= '<' TypeSignature TypeSignatures '>'

ParameterPart          ::= '<' ParameterSignature ParameterSignatures '>'

ParameterSignatures    ::= ParameterSignatures ParameterSignature
                        |

ParameterSignature     ::= Ident ':' bound_TypeSignature
```

References

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. OOPSLA'98*, October 1998.

[GJSB96] James Gosling, Bill Joy, Guy Steele and Gilad Bracha. *The Java Language Specification, Second Edition*. Java Series, Sun Microsystems, 2000. ISBN 0-201-31008-2.