Adding Generics to the Java Programming Language: Public Draft Specification, Version 2.0

Gilad Bracha, Sun Microsystems
Norman Cohen, IBM
Christian Kemper, Inprise
Martin Odersky, EPFL
David Stoutamire, Sun Microsystems
Kresten Thorup, Aarhus University
Philip Wadler, Avaya Labs

June 23, 2003

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 the extended language is 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) [GJSB00].

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 an actual type parameter list $\langle T_1, \ldots, T_n \rangle$. 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 corresponding formal type 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 and constructor declarations (Sections 3.1 and 3.3).

Syntax (see JLS, Sec. 4.3 and 6.5)

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

TypeVariable         ::= Identifier

ClassOrInterfaceType ::= ClassOrInterface TypeArgumentsOpt

ClassOrInterface     ::= Identifier
                      |  ClassOrInterfaceType . Identifier

TypeArguments        ::= < ActualTypeArgumentList >

ActualTypeArgumentList    ::= ActualTypeArgument
                      |  ActualTypeArgumentList , ActualTypeArgument

ActualTypeArgument   ::= ReferenceType
                      | Wildcard

Wildcard             ::= ? WildCardBoundsOpt

WildcardBounds       ::= extends  ReferenceType
                      |  super ReferenceType
```

Example 1 Parameterized types.

```
Vector<String>
Seq<Seq<E>>
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.

If a type parameter X has more than one bound, and if different bounds have member methods with the same signature and different return types, then any reference to such members of a value of type X is ambiguous and is a compile-time error.

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<T> {
   T convert();
}
class ReprChange<A extends ConvertibleTo<B>,
                 B extends ConvertibleTo<A>> {
   A a;
   void set(B x) { a = x.convert(); }
   B get() { return a.convert(); }
}
```

Parameterized declarations can be nested inside their declarations.

Example 3 Nested parameterized class declarations.

```
class Seq<E> {
   E head;
   Seq<E> tail;
   Seq() { this(null, null); }
   boolean isEmpty() { return tail == null; }
   Seq(E head, Seq<E> tail) { this.head = head; this.tail = tail; }
   class Zipper<T> {
      Seq<Pair<E,T>> zip(Seq<T> that) {
     if (this.isEmpty() || that.isEmpty())
             return new Seq<Pair<E,T>>();
     else
             return new Seq<Pair<E,T>>(
                 new Pair<E,T>(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 Wildcards in Parameterized Types

Type arguments within a parameterized type may be replaced with *wildcards.*

Example 4 Wildcards.

```
void printCollection(Collection<?> c) {  // a wildcard collection
  for (Object o : c) {
    System.out.println(o);
  }
}
```

Wildcards are useful in situations where no specific knowledge about the type parameter is required.

Supplying a type argument of ? is a syntactic shorthand, equivalent to declaring a new, uniquely named type variable whose bound is the default bound of the corresponding formal type parameter.

Wildcard parameterized types may be used as component types of arrays.

Example 5 Wildcard parameterized types as component types of array types.

```
public Method getMethod(Class<?>[] parameterTypes) { ... }
```

Wildcards may be given explicit bounds, just like regular type variable declarations. An upper bound is signified by the syntax:

```
 ? extends B
```

, where B is the bound.

Example 6 Bounded wildcards.

```
     }
boolean addAll(Collection<? extends E> c)
```

Unlike ordinary type variables declared in a method signature, no type inference is required when using a wildcard. Consequently, it is permissable to declare lower bounds on a wildcard, using the syntax:

```
? super B
```

, where B is a lower bound.

Example 7 Lower bounds on wildcards.

```
Reference(T referent, ReferenceQueue<? super T> queue);
```

Here, the referent can be inserted into any queue whose element type is a super type of the type T of the referent.

It is a compile time error to use a wildcard as an explicit type argument to a generic method or to a constructor. It is a compile time error to declare an array whose component type is a parameterized type whose actual arguments include a bounded wildcard.

2.4 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
```

2.5 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_{ij} in B_i, T_i is a subtype of

$$S_{ij}[A_1 := T_1, ..., A_n := T_n] \ .$$

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 $D\langle U_1, ..., U_k\rangle$.

2.6 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 8 Raw types.

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

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

3 Generic Methods and Constructors

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 9 Generic methods.

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

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

class Seq<E> {
   E head;
   Seq<E> tail;
   Seq() { this(null, null); }
   boolean isEmpty() { return tail == null; }
   Seq(E head, Seq<E> tail) { this.head = head; this.tail = tail; }

   <T> Seq<Pair<E,T>> zip(Seq<T> that) {
      if (this.isEmpty() || that.isEmpty())
         return new Seq<Pair<E,T>>();
      else
         return new Seq<Pair<E,T>>(
            new Pair<E,T>(this.head, that.head),
            this.tail.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 10 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. JLS 8.4.6.4 is revised as follows:

It is possible for a class to inherit more than one method with the same signature.

It is a compile time error if a class inherits more than one concrete method with the same signature.

This can happen, if a superclass is parameteric, and it has two methods that were distinct in the generic declaration, but have the same signature in the particular instantiation used.

Otherwise, there are two possible cases:

- If one of the inherited methods is not abstract, then there are two subcases:
  - If the method that is not abstract is static, a compile-time error occurs.
  - Otherwise, the method that is not abstract is considered to override, and therefore to implement, all the other methods on behalf of the class that inherits it. A compile-time error occurs if, comparing the method that is not abstract with each of the other of the inherited methods, for any such pair, either the return type of one the methods is not a subtype of the return type of the other or one has a return type and the other is void. Moreover, a compile-time error occurs if the inherited method that is not abstract has a throws clause that conflicts with that of any other of the inherited methods.
- If all the inherited methods are abstract, then the class is necessarily an abstract class and is considered to inherit all the abstract methods. A compile-time error occurs if, for any two such inherited methods, either the return type of one the methods is not a subtype of the return type of the other or one has a return type and the other is void. (The throws clauses do not cause errors in this case.)

There might be several paths by which the same method declaration might be inherited from an interface. This fact causes no difficulty and never, of itself, results in a compile-time error.

3.3 Generic Constructors

It is possible for a constructor to be declared generic, independently of whether the class the constructor is declared in is itself generic.

Syntax (see JLS, Sec. 8.8)

```
ConstructorDeclaration ::= ConstructorModifiersOpt ConstructorDeclarator
                             ThrowsOpt ConstructorBody

ConstructorDeclarator ::= TypeArgumentsOpt SimpleTypeName(FormalParameterListOpt)
```

The type arguments to a generic constructor are governed by the same rules as those to a generic method.

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 11 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.

Constructors may also be generic themselves - that is, have type parameters specific to the constructor, not the class. In this case, the same rules apply as for generic method invocation. See section 5.6 for a detailed discussion.

Syntax (see JLS, Sec. 15.9)

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

Example 12 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 Explicit Constructor Invocations

Constructors may be invoked explicitly. Again, the same rules apply as for generic method invocation. See section 5.6.

Syntax (see JLS, Sec. 8.8.5.1)

```
ExplicitConstructorInvocation ::= TypeArgumentsOpt this ( ArgumentListOpt )
                             |  TypeArgumentsOpt super ( ArgumentListOpt )
                             |  Primary . TypeArgumentsOpt super ( ArgumentListOpt )
```

5.3 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 13 Array creation expressions.

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

5.4 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 14 Assume the declarations

```
class Dictionary<K,V> extends Object { ... }
class Hashtable<K,V> extends Dictionary<K, V> { ... }

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
```

5.5 Type Comparison Operator

Type comparison can involve parameterized types. The rules of casting conversions, as defined in Section 5.4, apply.

Syntax (see JLS, Sec. 15.20.2)

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

Example 15 Type comparisons.

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

Example 16 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);
   }
}
```

5.6 Generic Method Invocation

It is not necessary to use a special syntax to invoke a generic method. Type parameters of generic methods are almost always elided; they are then inferred according to the rules given in Section 5.6.2.

However, in some very rare cases, it may be useful to explicitly pass actual type arguments to the method, rather than relying upon the built-in inference mechanism (see section 5.6.2 below).

The syntax for method invocation is extended to support this:

Syntax (See JLS 15.12)

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

If type arguments are explicitly provided, the number of actual type arguments must be identical to the number of formal type arguments in the method declaration, and each type argument must be a subtype of the corresponding formal parameter's bound.

Example 17 Generic method calls. (see Example 9)

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

Example 18 Generic method calls with explicit type parameters (see Example 9)

```
<Integer>swap(ints, 1, 3)
<String>sort(strings)
strings.<Integer>zip(ints)
```

The convention of passing parameters before the method name matches the form of a generic method declaration. In both cases, this is made necessary by parsing constraints: with the more conventional "type parameters after method name" convention the expression `f (a<b,c>(d))` would have two possible parses. The alternative of using `[ ]` brackets for types poses other problems.

5.6.1 Overloading

Overload resolution changes in several ways due to genericity and covariant return types. In particular:

- The existing specification assumes that if there are multiple candidate methods with the same signature, they all have the same return type. This is no longer the case.
- It is possible that a particular instantiation would have several concrete methods with the same signature. This was not possible before.
- The existing specification allows for multiple abstract methods with the same signature. However, we must now ensure that all such methods have the same erasure; otherwise the call should be ambiguous.

Consequently, JLS 15.12.2.2 is modified as follows:

If more than one member method is both accessible and applicable to a method invocation, it is necessary to choose one to provide the descriptor for the run-time method dispatch. The Java programming language uses the rule that the most specific method is chosen.

The informal intuition is that one method is more specific than another if any invocation handled by the first method could be passed on to the other one without a compile-time type error.

The precise definition is as follows. Let m be a name and suppose that there are two member methods named m, each having n parameters. Suppose that the types of the parameters of one member method are $T_1, \ldots, T_n$; suppose moreover that the types of the parameters of the other method are $U_1, \ldots, U_n$. Then the first member method is *more specific* than the other if and only if T_j can be converted to U_j by method invocation conversion, for all j from 1 to n.

A method is *strictly more specific* than another if and only if it is both more specific and the signatures of the two methods are not identical. A method is said to be *maximally specific* for a method invocation if it is applicable and accessible and there is no other applicable and accessible method that is strictly more specific.

If there is exactly one maximally specific method, then it is in fact the most specific method; it is necessarily more specific than any other method that is applicable and accessible. It is then subjected to some further compile-time checks as described in 15.12.3.

It is possible that no method is the most specific, because there are two or more maximally specific methods. In this case:

- If all the maximally specific methods have the same signature, then:
  - If exactly one of the maximally specific methods is not declared abstract, it is the most specific method.
  - Otherwise, if all the maximally specific methods are declared abstract, and all of the maximally specific methods have the same erasure, then the most specific method is chosen arbitrarily among the subset of the maximally specific methods that have the most specific return type. However, the most specific method is considered to throw a checked exception if and only if that exception is declared in the throws clauses of each of the maximally specific methods.
  - Otherwise, the method invocation is is ambiguous, and a compile-time error occurs.
- Otherwise, we say that the method invocation is ambiguous, and a compile-time error occurs.

5.6.2 Type Parameter Inference

Actual type parameters of a call to a generic method are inferred as follows.

Given a generic method p, declared as

$\langle TP_1, \ldots TP_n \rangle \; R \; p(F_1 \; f_1, \ldots F_m \; f_m), (n > 0, m \geq 0)$

where

$TP_i = T_i \; extends \; U_{i1} \& \ldots U_{ih_i}, 0 \leq i \leq n, h_i \geq 0$

and a usage

$p(e_1, \ldots e_m)$ where, for $1 \leq j \leq m$ the type of e_j is E_j, then for $1 \leq i \leq n$ let A_i be the most specific non-null type such that, for $1 \leq j \leq m$ $E_j \leq F_j[T_1 := A_1 \ldots T_n := A_n]$.

It is a compile-time error if any such a type A_i does not exist or is not uniquely defined.

If, for $1 \leq i \leq n$, $A_i \neq T_i$, then A_i is the actual type parameter inferred for T_i.

Otherwise, if $A_i = T_i$, then the actual type parameter is inferred based on the context, if any, in which the result of the method is being used.

- If the method result occurs in a context where it will be subject to assignment conversion to a type S then, for $1 \leq i \leq n$, define B_i such that
  1. if $A_i \neq T_i$ then $B_i = A_i$.
  2. if $A_i = T_i$ then let B_i be the least specific type such that
     (a) B_i is less than $U_{ih}[T_1 := B_1, \ldots, T_n := B_n], 0 < h \leq h_i$.
     (b) then $R[T_1 := B_1 \ldots T_n := B_n] \leq S$.

  B_i is the actual type parameter inferred for T_i. It is a compile-time error if any such type B_i does not exist or is not uniquely defined.
- Otherwise if the method result occurs in a context where it will be subject to method invocation conversion due to an invocation of a method f with type parameters $S_1 \ldots S_k$ and a corresponding formal parameter type S then, for $1 \leq i \leq n$, define C_i such that

1. if $A_i \neq T_i$ then $C_i = A_i$.
2. if $A_i = T_i$ then let C_i be the least specific type such that
   (a) C_i is less than $U_{ih}[T_1 := C_1, \ldots, T_n := C_n], 0 < h \leq h_i$.
   (b) then either
      – $R[T_1 := C_1 \ldots T_n := C_n] \leq S$. Or
      – $R[T_1 := C_1 \ldots T_n := C_n] \not\leq S$ and the actual result type, after substituting the inferred type arguments C_i for the formal type parameters T_i, would be a subtype of the expected result type S, if each occurence of any of the type variables $S_1 \ldots S_k$ in S were replaced by an arbitrary type.

C_i is the actual type parameter inferred for T_i.

It is a compile-time error if any such type C_i does not exist or is not uniquely defined.

- Otherwise, if the method result is discarded, or if the method result occurs in a context where it is the target of a field access expression $e.m$, then the actual type parameters are inferred as if the result had been assigned to a variable of type `Object`.

Example 19 Type parameter inference.

Assume the generic method declarations:

```
static <T> Seq<T> nil() { return new Seq<T>(); }
static <T> Seq<T> cons(T x, Seq<T> xs) { return new Seq<T>(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<Object>
cons(null, nil());                                 // of type:    Seq<Object>
```

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 $T\langle T_1, \ldots, T_n\rangle$ 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 its leftmost bound.
- The erasure of every other type is the type itself.

6.2 Translation of Methods

Each method $T\,m(T_1, \ldots, T_n)$ **throws** $S_1, \ldots, S_m$ 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 20 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 21 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 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 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 22 Overriding with covariant return types.

```
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 a compile-time error in the original source program.

It is a compile time error if a type declaration T has a member method m_1 and there exists a method m_2 declared in T or a supertype of T such that all of the following conditions hold:

- m_1 and m_2 have the same name.
- m_2 is accessible from T.
- m_1 and m_2 have different signatures.
- m_1 or some method m_1 overrides (directly or indirectly) has the same erasure as m_2 or some method m_2 overrides (directly or indirectly).

More precisely, define $\mathit{overridden}(m) = \{f \mid m\ \mathit{overrides}_k\ f, k \geq 0\}$ where
$m\ \mathit{overrides}_0\ m$
$m\ \mathit{overrides}_1\ m'$ if m directly overrides or implements m'
$m\ \mathit{overrides}_n\ m'$ if $m\ \mathit{overrides}_1\ m''$ and $m''\ \mathit{overrides}_{n-1}\ m'$
$\mathit{typeErasure}(S) = \{\mathit{typeErasure}(m) \mid m \in S\}$

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

$interactingMethods(T) = \{(m_1, m_2) \mid name(m_1) = name(m_2), \exists S_1 \geq T.member(m_1, S_1), \exists S_2 \geq T.declares(S_2, m_2), S_1 \leq S_2, accessible(m_2, S_1)\}$

$\forall T.\forall(m_1, m_2) \in interactingMethods(T)$. It is a compile-time error if $typeErasure(overridden(m_1)) \cap typeErasure(overridden(m_2)) \neq \phi \wedge signature(m_1) \neq signature(m_2)$.

Example 23 A class cannot have two member methods with the same name and type erasure.

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

This is illegal since `D.id(Object)` is a member of `D`, `C<String>.id(String)` is declared in a supertype of `D` and:

- The two methods have the same name, `id`.
- `C<String>.id(String)` is accessible to `D`.
- The two methods have different signatures.
- The two methods have different erasures.

Example 24 Two different methods of a class may not override methods with the same erasure.

```
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 `D.id(String)` is a member of `D`, `D.id(Integer)` is declared in `D` and:

- the two methods have the same name, `id`.
- the two methods have different signatures.
- `D.id(Integer)` is accessible to `D`.
- `D.id(String)` overrides `C<String>.id(String)` and `D.id(Integer)` overrides `I.id(Integer)` yet the two overridden methods have the same erasure.

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 Adjustments to the Classfile Format

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.

In addition, bridge methods must be marked as such in the class file, to ensure correct compilation.

These adjustments to the classfile format are defined in the Java Virtual Machine Specification, 3rd edition draft. The relevant parts of that draft are attached to this specification as a separate document.

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.

[GJSB00] 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.