

Modifications to Java Since Version 1.0, Proposed Modifications to Java, and Titanium Compatibility With Java

Amir Kamil, Titanium Group, UC Berkeley

July 1, 2003

The Java language has been modified substantially since the original 1.0 version, and additional modifications are planned in the future. These changes, unfortunately, are not documented in a central location on Sun's website. Therefore, for the benefit of Java and Titanium developers, the following is a compilation¹ of the various changes to Java, with overviews of each modification, and the status of Titanium compatibility with each new feature.

1 Modifications to Java Since 1.0

The Java language was changed substantially between versions 1.0 and 1.1, with the introduction of enclosed classes. Other, less significant features, such as instance initializers and the `assert` keyword, were also added in Java 1.1 and later versions.

1.1 Lexical Changes

Very few changes to the lexical structure of Java have been made since version 1.0. Two keywords have been added, `strictfp` and `assert`.

1.1.1 `strictfp`

The `strictfp` keyword was introduced in Java 1.2 as a modifier. It forces methods it modifies to adhere exactly to the IEEE 754 standard for all floating point operations. It can also be applied to classes, in which case all floating point operations in the class will be strictly IEEE 754. Individual variables cannot be declared `strictfp`.

1.1.2 `assert`

Until Java 1.4, the language contained no built-in facilities for assertions, prompting many programmers to write their own assertion methods. Java 1.4 introduced `assert` as a keyword, making it incompatible with previous code that used it as an identifier.

Two `assert` constructs were introduced, both of which throw `AssertionErrors` when the assertion fails. The first is

```
assert [boolean expr];
```

This construct throws an `AssertionError` if the given boolean expression evaluates to `false`. The second construct is

¹The information contained in this document is accurate to my knowledge as of the date of its writing.

```
assert x != 0;
assert repOk() : "Representation failure";
```

Figure 1: The two `assert` constructs.

```
assert [boolean expr] : [expr];
```

where the second expression must have a value (cannot be `void`). This construct, before throwing an `AssertionError`, evaluates the second expression, converts it to a `String`, and packages it into the `AssertionError`. Examples of using these constructs are given in figure 1.

1.2 New Constructs

A flurry of new constructs was introduced in Java 1.1, but since then only the `assert` construct (described above) has been added. The most significant new Java 1.1 construct was the enclosed class. Additional constructs were instance initializers, anonymous array expressions, and class literals.

1.2.1 Enclosed Classes

The most significant addition in Java 1.1 was the various types of *enclosed classes*². These classes are defined inside another class, called the *enclosing class*. Enclosed classes are mostly useful for organization, though the fact that they have access to private members of the enclosing class is also useful.

1.2.1.1 Nested Classes

The simplest type of enclosed class is the *nested class*. Such a class is defined within another class, with the class definition either modified by the `static` keyword or implicitly `static`. There are only minor differences between nested and normal classes. A nested class has access to private members of its enclosing class, and the enclosing class has access to private members of the nested class. References to the nested class follow the usual rules for `static` members, i.e. must be preceded by the enclosing class's name followed by the dot operator. As with other `static` members, the nested class can be accessed by its simple name within the definition of the enclosing class. Figure 2 shows an example of a nested class.

Since a nested class is a `static` member of the enclosing class, an instance of the nested class is not associated with any instance of the enclosing class. As such, the nested class can be instantiated without any instances of the enclosing class in existence.

1.2.1.2 Inner Classes

An inner class is similar to a nested class. It is defined as a nested class but without the `static` keyword modifying the definition. References to an inner class follow the same rule as references to a nested class, and the access rules between the inner class and the enclosing class are the same as well. Unlike a nested class, however, an instance of an inner class always has an associated instance of its enclosing class (like all instance members). Thus an inner class cannot be instantiated from a `static` context in the enclosing class or at all from an external class, except through an instance of the enclosing class. The syntax for instantiating an inner class through an instance of its enclosing class is `[enclosing instance].new [inner class constructor]`. Figure 3 shows an example of defining and using an inner class.

Since an instance of an inner class has an associated instance of its enclosing class, it is possible for the inner class to access members of the enclosing class directly as if they were its own. The reverse, of course,

²The Java language specification uses the term *nested classes* to refer to these types of classes. However, this term is overloaded to mean enclosed classes that are declared `static`, so we will not use it as a synonym for enclosed class.

```

class Outer {
    static class Nested {
        void baz(Outer out) {
            int b = out.x; // allowed access to private member in Outer
            ...
        }
        private int y;
    }
    static void foo() {
        Nested n = new Nested(); // can be accessed by simple name
        int a = n.y; // allowed access to private member in Nested
        ...
    }
    private int x;
}

class SomeOtherClass {
    void bar() {
        Outer.Nested n = new Outer.Nested(); // cannot be accessed by simple name
        ...
    }
}

```

Figure 2: Definition and access rules for nested classes.

```

class Outer {
    class Inner {
        void baz(Outer out) {
            int b = x; // allowed direct access to private member in Outer
            ...
        }
        private int y;
    }
    void foo() { // not static since cannot access inner class from static context
        Inner in = new Inner(); // can be accessed by simple name
        int a = in.y; // allowed access to private member in Inner
        ...
    }
    private int x;
}

class SomeOtherClass {
    void bar() {
        Outer.Inner in; // cannot be accessed by simple name
        in = new Outer().new Inner(); // must be created through instance of Outer
        ...
    }
}

```

Figure 3: Definition and access rules for inner classes.

```

class Outer {
    void foo(final int z, int w) {
        class Local {
            void baz(Outer out) {
                int b = x; // allowed direct access to private member in Outer
                ...
            }
            private int v = w; // illegal; w not final
            private int y = z; // allowed access to final parameter
        }
        Local l = new Local();
        int a = l.y; // allowed access to private member of Local
        ...
    }
    void foo() {
        Local l = new Local(); // illegal; not in scope
        ...
    }
    private int x;
}

class SomeOtherClass {
    void bar() {
        Outer.Local in; // illegal; not in scope
        ...
    }
}

```

Figure 4: Defining and using a local class.

is not true since an multiple instances of the inner class can be associated with a single instance of the outer class.

Certain restrictions on members of inner classes exist. For example, it is illegal to define `static` members of inner classes with the exception of compile-time constants. See the Java language specification for more details.

1.2.1.3 Local Classes

Local classes are like inner classes, except that they are defined within a method, constructor, or initializer. As such, the scope rules for a local class definition follow the same rules as local variables, i.e. the scope of the definition is the set of statements that it precedes in the block in which it is defined. Besides scoping rules, local classes for the most part follow the rules for inner classes. The exception is that instances of a local class defined within a `static` context do not have associated instances of the enclosing class, and therefore cannot directly access the enclosing class's instance variables.

An addition complication is introduced when considering local classes. Unlike with inner and nested classes, local variables and parameters of the method or constructor in which the local class is defined are accessible by the class definition. However, Java restricts such access to `final` local variables and parameters. Attempting to access non-`final` variables is a semantic error.

```

class Outer {
    void foo() {
        Object obj = new Object() {
            public String toString() {
                ...
            }
        };
        ...
    }
}

```

Figure 5: An example of an anonymous class.

```

new String[] {"Hi", "how", "are", "you"};

```

Figure 6: An anonymous array of strings.

1.2.1.4 Anonymous Classes

If a local class is to be used only once, it can instead be written as an *anonymous class* instead, in order to prevent cluttering up the namespace. Anonymous classes are defined as part of a class instance creation expression. The syntax is `new [constructor] [class body]` where the constructor is of the supertype of the anonymous class. An example of such a definition is in figure 5.

Anonymous classes follow the rules for local classes, except that an anonymous class may not declare a constructor. Therefore they can only be initialized with instance initializers.

1.2.2 Instance Initializers

With the introduction of anonymous classes, it was necessary to introduce a mechanism for initializing an instance of such a class, since they were not allowed to have constructors. The *instance initializer* was added as a result. This is similar to a static initializer, except that there is not `static` keyword preceding the initializer, and that the instance initializer is executed every time an object is created. Instance initializers must be able to complete normally, and if it is possible for an initializer to throw a checked exception, all constructors must declare it.

1.2.3 Anonymous Array Expressions

In Java 1.0, it was impossible to create anonymous arrays in Java. The *anonymous array expression* construct was introduced in Java 1.1. The syntax is similar to an array literal, except that the literal must be preceded by `new T[]` where `T[]` is the array type. Figure 6 gives an example of constructing an anonymous array of strings.

1.2.4 Class Literals

Java 1.1 also introduced *class literals*. The syntax of a class literal is `T.class` where `T` is a type. This is almost equivalent to `Class.forName("T")`, except that primitive types are allowed in a class literal. Class literals of the form `p.class` where `p` is a primitive type are the equivalent of `P.TYPE` where `P` is the corresponding boxed type, and returns a `Class` object representing the primitive type, not its boxed counterpart.

1.3 Other New Features

1.3.1 Local and Parameter `final` Variables

Starting from Java 1.1, local variables and parameters can be declared `final`. Local `final` variables can only be assigned to once, and `final` parameters cannot be reassigned.

1.3.2 Blank `final` Variables

Also starting from Java 1.1, `final` variables need not be assigned at declaration. Such variables are called *blank finals*. Blank `final` instance variables must be assigned to exactly once by each constructor, and `static` variables that are blank finals must be assigned to by a `static` initializer. Local blank finals need not be assigned to.

2 Proposed Modifications to Java

In addition to the modifications currently implemented in Java 1.4, there are a few more modifications under considerations. The most anticipated of these is the introduction of templates. Related to templates is the concept of variance. Unrelated but also under consideration are concise object-array literals.

2.1 Templates

Templates³ have long been lacking in Java, and they are likely to finally be introduced in Java 1.5. The probable implementation is specified in JSR 14 and is based on the GJ language. Currently, there is a prototype compiler based on Java 1.3 that purports to implement this specification. However, some features in the compiler are undocumented in the specification, and some aspects of the specification appear to be broken in the compiler. Below is an overview of the specification and the language accepted by the compiler.

2.1.1 Syntax

2.1.1.1 Definition

A template definition is the same as a definition for a non-generic type, except that the type name is followed by a list of formals enclosed by `<` and `>`. Formals in a list are separated by commas. Formals can be bounded (§2.1.3.1). The scope of a formal is the entire class definition, including the formals list itself. Figure 7 shows examples of template definitions.

Template formals can be used anywhere a type is required, except for the cases mentioned below. However, it is unclear whether or not a template formal is allowed in a class instance creation expression. Currently, the prototype compiler does not accept this syntax. Thus `new T()`, where `T` is a template formal, currently results in a compile-time error.

2.1.1.2 Parameterization

A template is parameterized by following its simple name with a list of parameters enclosed by `<` and `>`. This parameterization can be used anywhere a type is required, such as variable declarations and instantiations. An example of this is given in figure 8.

2.1.2 Treatment of Primitive Types

The JSR 14 specification forbids the use of primitives as parameters to templates. The reason appears to be that allowing them as parameters would require a change to the Java virtual machine.

³Also known as *generic types* and *parameterized types*. We will use the three terms interchangeably.

```

class Vector<T> {
    T[] elems;
    int size;
    ...
    void add(T x) {
        ...
    }
    T get(int n) {
        return elems[n];
    }
}

class Two<A, B> {
    A first() {...}
    B second() {...}
}

```

Figure 7: Examples of template definitions.

```

Vector<String> vs = new Vector<String>();
Two<Object, Object> = new Two<Object, Object>();

```

Figure 8: Template parameterization.

2.1.3 Template Formals and Parameters

2.1.3.1 Bounding

Template formals can be followed by a set of *bounds*, the **extends** keyword followed by a list of types delimited by **&**. Only the first type in this list can be a class, the rest must be interfaces. Figure 9 shows the syntax for type bounds.

The semantic effect of bounding is that in any instantiation of a template, a given parameter must be a subtype of each of the bounds of the corresponding formal. The **Object** class is the implicit bound if none is given. The bounds are also used in statically checking the template definition (§2.1.4).

2.1.3.2 Parameters

Primitives are not allowed as parameters to generic types. Otherwise, all types that meet the required bounds, including generic types, can be used as parameters.

2.1.4 Code Generation and Static Checking

It appears as though generic types are practically the same as non-generic types, except that the former introduces implicit casts when necessary. Thus there appears to be little difference between generic and non-generic types when generating code.

```

class Bounded<A extends Number&Comparable&Cloneable> {...}

```

Figure 9: Example of a bounded formal. **Bounded** can only be instantiated with a class that is a subtype of **Number**, **Comparable**, and **Cloneable**.

```

class Foo<T> {
    void bar(T t1, T t2) {
        int n = t1.compareTo(t2); // illegal since Object has no compareTo()
        ...
    }
}

```

Figure 10: A template that would be legal in C++ (modulo syntax) but is illegal in Java due to semantic checking rules.

```

class Generic<T> {...}
class Child1 extends Generic {...}
class Child2<T> extends Generic<T> {...}
class Child3 extends Generic<String> {...}

```

Figure 11: Inheritance of generic types.

In order for this scheme to work, generic types must be statically checked with certain assumptions about the template formals. Specifically, the template is semantically checked with the assumption that each template formal defines all the members in its bounding types, no more and no less. Thus the template definition in figure 10 is illegal, since `Object` does not define a `compareTo()` method. It is irrelevant whether or not the template will only be used with classes that define this method. This differs from C++ templates, in which each instantiation of a template with distinct parameters generates a new version of the generic class with the formals replaced by the parameters, which is then statically checked.

2.1.5 Inheritance

The JSR 14 specification allows inheritance of generic types. This includes extension of raw types, of generic types parameterized by a template formal, and generic types with a given parameterization. Figure 11 gives examples of these inheritance types. Generic interfaces can be defined and implemented as well.

2.1.6 Exceptions

Generic exceptions are disallowed by the JSR 14 specification. Exceptions are allowed as template parameters, and template formals can be thrown and used in a `throws` clause. However, template formals cannot be used in a `catch` clause.

2.1.7 Raw Types

Raw types are parameterized types that are not given any parameters. For example, `Vector` is the raw type for the generic class `Vector<T>`. Raw types are mostly allowed to support legacy code and appear to be somewhat equivalent to the generic type parameterized by the bounds of each of its formals (e.g. `Vector` is roughly equivalent to `Vector<Object>`). However, variables of a raw type can be assigned any parametric instances of the generic type, and vice versa. See the JSR 14 specification for rules concerning raw types.

2.1.8 Generic Methods

The JSR 14 specification refers to *generic methods*, methods that are parameterized. The syntax is similar to parameterization of classes, except that the type parameter section comes before the return type of the method. The type parameter can include bounds. The example given in the JSR 14 specification is shown in figure 12.

```

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);
            }
        }
    }
}
}

```

Figure 12: Example of generic methods.

```

Vector<Integer> v = new Vector<Integer>;
int i = 4;
Integer i2 = i; // automatic boxing not allowed
v.add(i); // automatic boxing allowed
i = v.get(0); // automatic unboxing not allowed
i = (int) v.get(0); // automatic unboxing not allowed

```

Figure 13: Boxing and unboxing of primitive types.

Unfortunately, it appears that generic methods are not implemented correctly in the prototype compiler. The compiler disallows bounded parameters, and does not allow the method call syntax `<E>method()` used in figure 12.

2.1.9 Automatic Boxing of Primitives

One feature that is implemented in the JSR 14 prototype compiler is automatic boxing of primitives. This feature is not documented in the JSR 14 specification, or anywhere else as far as I know, and its implementation is incomplete. Specifically, the only case that I know of in which it automatically boxes primitives is when you use the unboxed type with a generic class parameterized by the boxed type. Automatic unboxing is not implemented. Figure 13 shows the various cases of boxing and unboxing and the legality of each case. Casts between the boxed and unboxed types are still not allowed.

2.2 Variance

According to the JSR 14 template specification, instantiations of a generic type that are parameterized by distinct types are unrelated, even if the parameter types are. For example, `Vector<String>` is in no way related to `Vector<Object>`. Consider the code in figure 14. It should be legal to pass in a `Vector<String>` to the `addAll()` method of `Vector<Object>`. Yet this is illegal, since `Vector<String>` is not a subtype of `Vector<Object>`.

We would like the `addAll()` call in figure 14 to be allowed, so we need to change the parameter type of `addAll()`. The only variable types that can hold either generic instance are `Object`, which is not very useful, and the raw type `Vector`, which is unsafe. What we want instead, is a variable that can hold all `Vector<T'>` such that `T'` is a subtype of `T`, the template formal. Using *variance*, we can accomplish this.

There are different types of variance depending on the operations, reading or writing, to be done on the variant type.

```

class Vector<T> {
    void addAll(Vector<T> v) {
        for (int i = 0; i < v.size(); i++) {
            add(v.get(i));
        }
    }
    void add(T x) {...}
    ...
}

void foo() {
    Vector<Object> v1 = new Vector<Object>();
    Vector<String> v2 = new Vector<String>();
    ...
    v1.addAll(v2); // Illegal
}

```

Figure 14: Generic instances parameterized by related types are unrelated, and therefore cannot be assigned to each other.

2.2.1 Covariance

If only read operations are required from the variant type, then *covariance* can be used. The syntax for a covariant type is `[template name].<+[parameter]>`. For example, `Vector<+Set>` is covariant in `Set`. A variable of type `Vector<+Set>` can reference any `Vector<T>` where `T` is a subtype of `Set`. Using covariance, we can rewrite the `addAll()` of `Vector` to accept more general arguments. The rewritten code is in figure 15.

2.2.2 Contravariance

On the other hand, if the necessary operation on the variant type is writing, then *contravariance* can be used. The syntax is the same as for covariance, except that the `+` is replaced by a `-`. A variable of type `Vector<-Set>` can reference any `Vector<T>` such that `T` is a supertype of `Set`. Using this syntax, we can write the `fill()` method of the `Collections` class, as in figure 16.

2.2.3 Bivariance

If neither reads nor writes need be done, then *bivariance* can be used to range over all parameterizations of a generic type. The syntax for a bivariant type is like the syntax of a normal parameterized type, but with the parameter replaced by `*`. For example, a variable of type `Vector<*>` ranges over all `Vector<T>`. Bivariance is limited in use since neither reads nor writes can be done. Examples of operations that can be called on a `Vector<*>` are the `isEmpty()` and `size()` methods.

2.2.4 Invariance

The normal syntax for generic types represent *invariant* types. For completeness, invariant types can be explicitly specified using the same syntax as covariant types, but with the `+` replaced by `=`. Both reads and writes are allowed on invariant types.

2.2.5 Arrays

Java arrays, unfortunately, are not type safe. This is mostly due to the fact that, unlike with templates, arrays of related types are themselves related. For example, `String[]` is a subtype of `Object[]`. Thus the

```

class Vector<T> {
    void addAll(Vector<+T> v) { // covariant argument
        for (int i = 0; i < v.size(); i++) {
            add(v.get(i));
        }
    }
    void add(T x) {...}
    ...
}

void foo() {
    Vector<Object> v1 = new Vector<Object>();
    Vector<String> v2 = new Vector<String>();
    Vector<+Object> v3;
    ...
    v1.addAll(v2); // Now legal
    v3 = v2; // legal
}

```

Figure 15: Covariance can be used to interchange generic types parameterized by related types.

```

class Collections {
    ...
    static <T> void fill(List<-T> l, T x) {
        for (int i = 0; i < l.size(); i++) {
            l.set(i, x);
        }
    }
}

```

Figure 16: A (slow) implementation of the `Collections.fill()` method using contravariance.

```
Object[] arr = new String[10]; // OK since String[] <= Object[]
arr[0] = new Integer(-1); // OK at compile-time since Integer <= Object
// Not OK at runtime since Integer not <= String
```

Figure 17: Normal Java arrays are type unsafe.

```
List[+] arr1 = new Vector[10]; // legal since Vector <= List
List[-] arr2 = new Collection[10]; // legal since Collection >= List
Vector[=] arr3 = new Vector[10]; // legal Vector = Vector
List[*] arr4; // illegal; bivariant arrays not allowed
arr1 = new Collection[10]; // illegal since Collection not <= List
arr2 = new Vector[10]; // illegal since Vector not >= List
arr3 = new Collection[10]; // illegal since Collection not = Vector
```

Figure 18: Variant arrays.

code in figure 17 is accepted by the compiler, but results in a runtime error. Such arrays are called *dynamic arrays*, since they need to be checked at runtime.

Variance can also be used with arrays in order to achieve static type safety. A variant array is denoted by $T[x]$ where x is one of the variance symbols $+$, $-$, and $=$. The rules for variant arrays are analogous to those of variant generics.

Note that bivariant arrays are not allowed. This is because something like `Number[*]` doesn't make much sense, since the `Number` type is irrelevant if the variable can range over all arrays. A similar effect can be achieved by using `Object[+]`, though this cannot range over arrays of primitives.

Unlike with templates, the `=` symbol is not optional for invariant arrays, since dynamic arrays are not invariant.

Variance symbols are only used with variable declarations. They are unneeded when allocating arrays, since the type of the allocated array is known. Thus, array allocations always create invariant arrays.

2.2.6 Incompatibilities With Nonvariant Java

The variance rules were designed to retain compatibility with non-generic Java code. However, the rules do result in incompatibilities with nonvariant generic code. Dynamic arrays of template formals are not allowed, nor are dynamic arrays of generic types. Assigning arrays of generic types to dynamic arrays types is allowed, though unsafe and deprecated.

```
Vector<String>[] arr; // dynamic array of generic not allowed
Object[] arr2 = new Vector<String>[10]; // deprecated, unsafe
arr[0] = new Vector<Integer>(); // allowed at both compile-time and runtime;
// runtime system doesn't have enough info to catch

class Foo<T> {
    T[] arr3; // dynamic array of formal not allowed
    T[=] arr4; // variant array of formal allowed
}
```

Figure 19: Legal and illegal nonvariant arrays.

```
String name, city;
int age;
void introduce() {
    printf("Hello, my name is %s. I am %d years old. I live in %s.\n",
          {name, age, city});
}
static void printf(String fmt, Object[] args) {
    ...
}
```

Figure 20: Example of using object-array literals.

2.2.7 Specification and Future Implementation

Variance is not described in the JSR 14 specification. The specification for variance tags along with the prototype compiler supporting templates, which also supports variance. Thus it remains to be seen whether or not variance will be implemented in a future version of Java.

2.3 Concise Object-Array Literals

Another feature under consideration is *concise object-array literals*, proposed under JSR 65. At the present time, array literals are only allowed as the optional initialization part of a variable declaration, or as part of an anonymous array expression. In the proposals, array literals not part of either of these two expressions would be allowed, and would be of type `Object[]`. Primitive elements would be automatically boxed.

The main purpose of introducing concise object-array literals is to circumvent the fact that Java does not allow variable-arguments methods. Using array literals, extra arguments can be concisely package into an `Object` array. For example, a C-style formatted printing function would take in a format string and an `Object` array as arguments. Figure 20 shows how this would be done.

A small problem with this scheme is that it does not allow you to call `printf()` with only a single string as the argument. As it is in figure 20, you would have to pass an empty array in as the second argument (`{}`). An easy way around this is to overload the function, adding a version that only takes in a string.

Currently, I know of no prototype compiler that implements concise object-array literals. It remains to be seen whether or not this feature will be added in a future Java version.

3 Titanium Compatibility With Java

The Titanium language is essentially a superset of Java 1.0, so most programs written in Java 1.0 should compile and run using the Titanium compiler. The most notable departure from Java 1.0 is that it does not support dynamically allocated threads. As such, any Java program or library (e.g. AWT) that uses threads will not work with Titanium.

3.1 Compatibility With Java 1.4

Since Titanium is based on Java 1.0, it does not implement most of the features introduced in later Java versions. The exception is blank, local, and parameter `final` variables. Parameter `final` variables are correctly implemented in Titanium. Blank and local `final` variables are in Titanium but their implementation is broken. Specifically, Titanium accepts the syntax but does not implement the semantics, i.e. such variables can be reassigned to. Correct Java programs that use them should work correctly in Titanium, barring interference from malicious programs.

```

template<class T> class List {
    void add(T elem) {...}
    void addAll(List other) {...} // 'List' equivalent to 'template List<T>'
    T get(int n) {...}
}

```

Figure 21: A template definition.

```

template List<String> strs = new template List<String>();

```

Figure 22: An example of a template instantiation.

All features in Java 1.4 will probably make their way into Titanium to some extent. However, the `strictfp` modifier may be allowed but ignored, and the bugs with `final` variables may not be fixed.

3.2 Compatibility With Proposed Java Modifications

3.2.1 Templates

Templates have been in Titanium since its inception, long before their (as of now, future) introduction into Java. As such, Titanium's version of templates differs greatly from Java's, some cases beneficial and some not.

3.2.1.1 Syntax

Titanium templates deviate from Java templates even in the syntax used to define and instantiate them. Titanium uses a more C++-like syntax than Java, with template definitions preceded by `template<[parameters]>`. Instantiation of a template is similar to Java, except that the template name is preceded by the `template` keyword. Figures 21 and 22 exemplify these rules.

Within a template definition, the current instantiation of the template can be referred to by the unparameterized template name. This differs from Java, in which that syntax refers to the raw template type. Titanium has no such raw types.

It is likely that we will change Titanium's template syntax to match Java's, as Java's is simpler and cleaner. It is not likely that we will add support for raw types into Titanium.

3.2.1.2 Treatment of Primitive Types

Primitive and *immutable* (see Titanium language specification) types are allowed as parameters to templates in Titanium. This is the biggest advantage of Titanium templates over Java templates, as it allows much more efficient data structures when storing primitive values. This feature will definitely stay in Titanium.

3.2.1.3 Template Formals and Parameters

Titanium does not allow template formals to be bounded. Due to the difference in static checking of templates between Java and Titanium, this is no great loss.

In addition to allowing types as template formals, Titanium also accepts primitive values as formals. The syntax for definition is similar, except that `<class [type]>` is replaced by `<[type] [identifier]>`. Such templates can only be instantiated with constants, and the instantiation constant is bound to the identifier given in the template formals. Figure 23 shows an example of defining and using this type of template. The built-in `Point<N>`, `Domain<N>`, and `RectDomain<N>` classes are defined like this.

```

template<int n> class TInt {
    int value() {
        return n;
    }
}

void uselessExample() {
    template TInt<5> ti5 = new template TInt<5>();
    System.out.println(ti5.value());
}

```

Figure 23: Defining and using a template parameterized by a primitive value.

```

template<class T> class Bar {
    void baz(T x) {
        x.foo(); // illegal in Java since Object does not define foo()
    }
}

```

Figure 24: A legal template in Titanium that would be illegal in Java.

Titanium also disallows using templates as parameters to other templates. However, as mentioned above, it does allow primitives to be used as parameters, unlike Java.

Bounded formals are unlikely to make their way into Titanium, though we may allow templates to be parameters to other templates.

3.2.1.4 Code Generation and Static Checking

Templates in Titanium are statically checked as in C++. Each use of templates with different parameters results in an entire new class being generated, with references to the template formals replaced by the actual parameter types used. This new class is then statically checked. As a result, the template in figure 24 is legal in Titanium but can only be used with types that define a method `foo()`, while it would be illegal in Java (even ignoring the syntactic differences), since the implicit `Object` bound does not define the method.

Since Titanium's method of static checking allows more general template usage than Java's does, it is likely to remain the same.

3.2.1.5 Inheritance

Unlike Java, Titanium disallows all forms of inheritance of generic types. This includes extending or implementing a parameterized type, an instantiation of a parameterized type, or a raw template type. As such, while it is allowed to define generic interfaces, they can never be implemented.

We may in the future extend Titanium to allow classes to inherit generic types, though not raw types.

3.2.1.6 Exceptions

In Titanium, it is legal to define and use generic exceptions as you would normal exceptions, though they are illegal in Java. Like in Java, it is possible to throw template formals. Unlike in Java, it is also possible to catch template formals in Titanium. The legal exception usage in Titanium is a superset of the legal usage in Java.

Feature	Version	Titanium Status
<code>strictfp</code>	1.2	Likely syntactically accepted in future
<code>assert</code>	1.4	Likely implemented in future
Enclosed Classes	1.1	Likely implemented in future
Instance Initializers	1.1	Likely implemented in future
Anonymous Arrays	1.1	Likely implemented in future
Class Literals	1.1	Likely implemented in future
Blank/Local <code>finals</code>	1.1	Broken implementation, unlikely to be fixed
Templates	1.5+	Incompatible implementation, compatibility likely increased in future
Variance	1.5+	Unlikely to be implemented in future
Concise <code>Object []</code> Literals	1.5+	Implementation likely to wait until Java implementation

Table 1: Summary of new Java features.

3.2.1.7 Raw Types

As mentioned above, Titanium does not have raw types, and there are currently no plans to introduce them.

3.2.1.8 Generic Methods

Titanium does not have generic methods. There does not appear to be much motivation to add them.

3.2.1.9 Automatic Boxing of Primitives

There is not automatic boxing of primitives in Titanium, and no motivation to add it since Titanium templates allow primitive types as parameters.

3.2.2 Variance

Titanium does not implement variance and as such suffers from the same issues as non-variant Java with regards to templates and inheritance. At the moment, there are no plans to implement variance, as the concept and rules appear to be too confusing to be useful to most programmers.

3.2.3 Concise Object-Array Literals

Concise object-array literals are not currently in Titanium, though they may be implemented in a future Titanium version. We will likely wait until they are introduced into an official Java version before adding them to Titanium. If we do implement them, we will need to determine what to do with immutables, since they do not have corresponding boxed types. One (suboptimal) solution is to not allow immutables to be used in object-array literals.

4 Summary of New Features

Table 1 summarizes the new features introduced since Java 1.0, proposed modifications, and Titanium's current status regarding compatibility. Table 2 details Titanium's compatibility status with Java templates.

5 References

- [1] Gosling, James, et al. *Java Language Specification, Second Edition*. Addison-Wesley, 2000.

Feature	Titanium Status
Syntax	Incompatible implementation, likely made compatible in future
Formals/Parameters	No support for bounding or templates as parameters, future compatibility undecided
Inheritance	No support for extending templates, may be added in future
Exceptions	Support more general than Java's
Raw Types	Unsupported, unlikely to be in future
Generic Methods	Unsupported, unlikely to be in future
Automatic Boxing	Unsupported, unlikely to be in future

Table 2: Titanium compatibility with Java templates.

- [2] *JDK 1.1 New Feature Summary*. Sun Microsystems. 1998.
[available via <http://java.sun.com/products/archive/jdk/1.1/index.html>]
- [3] *J2SE 1.2 Platform Compatibility*. Sun Microsystems. 2003.
[<http://java.sun.com/j2se/1.2/compatibility.html>]
- [4] *J2SE 1.3 Platform Compatibility*. Sun Microsystems. 2003.
[<http://java.sun.com/j2se/1.3/compatibility.html>]
- [5] *J2SE 1.4.1 Platform Compatibility*. Sun Microsystems. 2003.
[<http://java.sun.com/j2se/1.4.1/compatibility.html>]
- [6] *Proposed Changes to the Java Language Specification*. Sun Microsystems. 7 March 2000.
[<http://java.sun.com/docs/books/jls/jls-proposed-changes.html>]
- [7] Bracha, Gilad, et al. “Adding Generics to the Java Programming Language: Participant Draft Specification.” Technical Report, Sun Microsystems, 2001.
- [8] Sun Microsystems, et al. “Variance in the Java Programming Language.” Technical Report, Sun Microsystems, 2003.