Inner Classes Specification

The newest release of the Java language allows classes to be defined in any scope. This paper specifies how the language has been extended to permit this, and shows how Java programmers can benefit from the change.

For more up-to-date and detailed information about the Java language, platform, and development environment, refer to the JavaSoft web site `http://java.sun.com/products/JDK/1.1/`.

Java is developed by JavaSoft, an operating company of Sun Microsystems, Inc.

Contents

What are top-level classes and inner classes?

In previous releases, Java supported only *top-level classes*, which must be members of packages. In the 1.1 release, the Java 1.1 programmer can now define *inner classes* as members of other classes, locally within a block of statements, or (anonymously) within an expression.

Here are some of the properties that make inner classes useful:

- The inner class's name is not usable outside its scope, except perhaps in a qualified name. This helps in structuring the classes within a package.
- The code of an inner class can use simple names from enclosing scopes, including both class and instance members of enclosing classes, and local variables of enclosing blocks.

Inner classes result from the combination of block structure with class-based programming, which was pioneered by the programming language Beta. Using block structure with inner classes makes it easier for the Java programmer to connect objects together, since classes can be defined closer to the objects they need to manipulate, and can directly use the names they need. With the removal of restrictions on the placement of classes, Java's scoping rules become more regular, like those of classical block structured languages such as Pascal and Scheme.

In addition, the programmer can define a class as a `static` member of any top-level class. Classes which are `static` class members and classes which are package members are both called top-level classes. They differ from inner classes in that a top-level class can make direct use only of its own instance variables. The ability to nest classes in this way allows any top-level class to provide a package-like organization for a logically related group of secondary top-level classes, all of which share full access to private members.

Inner classes and nested top-level classes are implemented by the compiler, and do not require any changes to the Java Virtual Machine. They do not break source or binary compatibility with existing Java programs.

All of the new nested class constructs are specified via transformations to Java 1.0 code that does not use inner classes. When a Java 1.1 compiler is producing Java virtual machine bytecodes, these bytecodes must represent the results of this (hypothetical) source-to-source transformation, so that binaries produced by different Java 1.1 compilers will be compatible. The bytecodes must also be tagged with certain attributes to indicate the presence of any nested classes to other Java 1.1 compilers. This is discussed further below.

Example: A simple adapter class

Consider the design of an *adapter class*, which receives method invocations using a specified interface type on behalf of another object not of that type. Adapter classes are generally required in order to receive events from AWT and Java Bean components. In Java 1.1, an adapter class is most easily defined as an inner class, placed inside the class which requires the adapter.

Here is an incomplete class `FixedStack` which implements a stack, and is willing to enumerate the elements of the stack, from the top down:

```
public class FixedStack {
    Object array[];
    int top = 0;
    FixedStack(int fixedSizeLimit) {
        array = new Object[fixedSizeLimit];
    }

    public void push(Object item) {
        array[top++] = item;
    }
    public boolean isEmpty() {
        return top == 0;
    }
    // other stack methods go here...

   /** This adapter class is defined as part of its target class,
     *  It is placed alongside the variables it needs to access.
     */
    class Enumerator implements java.util.Enumeration {
        int count = top;
        public boolean hasMoreElements() {
            return count > 0;
        }
        public Object nextElement() {
            if (count == 0)
                throw new NoSuchElementException("FixedStack");
            return array[--count];
        }
    }
    public java.util.Enumeration elements() {
        return new Enumerator();
    }
}
```

The interface `java.util.Enumeration` is used to communicate a series of values to a client. Since `FixedStack` does not (and should not!) directly implement the `Enumeration` interface, a separate adapter class is required to present the series of elements, in the form of an `Enumeration`. Of course, the adapter class will need some sort of access to the stack's array of elements. If the programmer puts the definition of the adapter class inside of `FixedStack`, the adapter's code can directly refer to the stack object's instance variables.

In Java, a class's non-`static` members are able to refer to each other, and they all take their meaning relative to the current instance `this`. Thus, the instance variable `array` of `FixedStack` is available to the instance method `push` and to the entire body of the inner class `FixedStack.Enumerator`. Just as instance method bodies "know" their current instance `this`, the code within any inner class like `Enumerator` "knows" its *enclosing instance*, the instance of the enclosing class from which variables like `array` are fetched.

One of the ways in which the `FixedStack` example is incomplete is that there is a race condition among the operations of the FixedStack and its `Enumerator`. If a sequence of pushes and pops occurs between calls to `nextElement`, the value returned might not be properly related to previously enumerated values; it might even be a "garbage value" from beyond the current end of the stack. It is the responsibility of the programmer to defend against such race conditions, or to document usage limitations for the class. This point is discussed later. One defense against races looks like this:

```
public class FixedStack {
    ...
    synchronized public void push(Object item) {
        array[top++] = item;
    }
    class Enumerator implements java.util.Enumeration {
        ...
        public Object nextElement() {
            synchronized (FixedStack.this) {
                if (count > top)  count = top;
                if (count == 0)
                 throw new NoSuchElementException("FixedStack");
                return array[--count];
            }
        }
```

The expression `FixedStack.this` refers to the enclosing instance.

Example: A local class

When a class definition is local to a block, it may access any names which are available to ordinary expressions within the same block. Here is an example:

```
Enumeration myEnumerate(final Object array[]) {
    class E implements Enumeration {
        int count = 0;
        public boolean hasMoreElements()
            { return count < array.length; }
        public Object nextElement()
            { return array[count++]; }
    }
    return new E();
}
```

For the moment, we say nothing about *how* this code works, but Java's rules of scoping and variable semantics precisely require *what* this code does. Even after the method `myEnumerate` returns, `array` can still be used by the inner object; it does not "go away" as in C. Instead, its value continues to be available wherever that value is required, including the two methods of `E`.

Note the final declaration. Local `final` variables such as `array` are a new feature in 1.1. In fact, if a local variable or parameter in one class is referred to by another (inner) class, it *must* be declared `final`. Because of potential synchronization problems, there is by design no way for two objects to share access to a changeable local variable. The state variable `count` could not be coded as a local variable, unless perhaps it were changed a one-element array:

```
Enumeration myEnumerate(final Object array[]) {
  final int count[] = {0}; // final reference to mutable array
    class E implements Enumeration {
        public boolean hasMoreElements()
            { return count[0] < array.length; }   ...
```

(Sometimes the combination of inheritance and lexical scoping can be confusing. For example, if the class `E` inherited a field named `array` from `Enumeration`, the field would hide the parameter of the same name in the enclosing scope. To prevent ambiguity in such cases, Java 1.1 allows inherited names to hide ones defined in enclosing block or class scopes, but prohibits them from being used without explicit qualification.)

Anonymous classes

In the previous example, the local class name `E` adds little or no clarity to the code. The problem is not that it is too short: A longer name would convey little additional information to the maintainer, beyond what can be seen at a glance in the class body. In order to make very small adapter classes as concise as possible, Java 1.1 allows an abbreviated notation for local objects. A single expression syntax combines the definition of an *anonymous* class with the allocation of the instance:

```
Enumeration myEnumerate(final Object array[]) {
    return new Enumeration() {
        int count = 0;
        public boolean hasMoreElements()
            { return count < array.length; }
        public Object nextElement()
            { return array[count++]; }
    };
}
```

In general, a `new` expression (an instance creation expression) can end with a class body. The effect of this is to take the class (or interface) named after the `new` token, and subclass it (or implement it) with the given body. The resulting anonymous inner class has the same meaning as if the programmer had defined it locally, with a name, in the current block of statements.

Anonymous constructs like these must be kept simple, to avoid deeply nested code. When properly used, they are more understandable and maintainable than the alternatives—named local classes or top-level adapter classes.

If an anonymous class contains more than a line or two of executable code, then its meaning is probably not self-evident, and so a descriptive local name should be given to either the class or (via a local variable) to the instance.

An anonymous class can have initializers but cannot have a constructor. The argument list of the associated `new` expression (often empty) is implicitly passed to a constructor of the superclass.

As already hinted, if an anonymous class is derived from an interface *I*, the actual superclass is `Object`, and the class implements *I* rather than extending it. (Explicit `implements` clauses are illegal.) This is the only way an interface name can legally follow the keyword `new`. In such cases, the argument list must always be null, to match the constructor of the actual superclass, `Object`.

How do inner classes work?

Inner class code is typically defined relative to some enclosing class instance, so the inner class instance needs to be able to determine the enclosing instance.

The JavaSoft Java 1.1 compiler arranges this by adding an extra `private` instance variable which links the inner class to the enclosing class. This variable is initialized from an extra argument passed to the inner class constructor. That argument, in turn, is determined by the expression which creates the inner class instance; by default it is the object doing the creation.

The Java 1.1 Language Specification specifies that the name of a type which is a class member, when transformed into Java 1.0 code for the purpose of generating Java virtual machine bytecodes, consists of the fully qualified name of the inner class, except that each '`.`' character following a class name is replaced by a '`$`'. In addition, each inner class constructor receives the enclosing instance in a prepended argument. Here is how the transformed source code of the `FixedStack` example might look:

```
public class FixedStack {
    ... (the methods omitted here are unchanged)
    public java.util.Enumeration elements() {
        return new FixedStack$Enumerator(this);
    }
}

class FixedStack$Enumerator implements java.util.Enumeration {
    private FixedStack this$0; // saved copy of FixedStack.this
    FixedStack$Enumerator(FixedStack this$0) {
        this.this$0 = this$0;
        this.count = this$0.top;
    }

    int count;
    public boolean hasMoreElements() {
        return count > 0;
    }
    public Object nextElement() {
        if (count == 0)
            throw new NoSuchElementException("FixedStack");
        return this$0.array[--count];
    }
}
```

Anyone who has already programmed with Java or C++ adapter classes has written code similar to this, except that the link variables must be manually defined and explicitly initialized in top-level adapter classes, whereas the Java 1.1 compiler creates them automatically for inner classes.

When the `Enumerator` needs to refer to the `top` or `array` fields of the enclosing instance, it indirects through a `private` link called `this$0`. The spelling of this name is a mandatory part of the transformation of inner classes to the Java 1.0 language, so that debuggers and similar tools can recognize such links easily. (Most programmers are happily unaware of such names.)

(Note: There is a limitation in some implementations of Java 1.1, under which the initialization of `this$0` is delayed until after any superclass constructor is run. This means that up-level references made by a subclass method may fail if the method happens to be executed by the superclass constructor.)

References to local variables

A class definition which is local to a block may access local variables. This complicates the compiler's job. Here is the previous example of a local class:

```
Enumeration myEnumerate(final Object array[]) {
    class E implements Enumeration {
        int count = 0;
        public boolean hasMoreElements()
            { return count < array.length; }
        public Object nextElement() {
            { return array[count++]; }
    }
    return new E();
}
```

In order to make a local variable visible to a method of the inner class, the compiler must copy the variable's value into a place where the inner class can access it. References to the same variable may use different code sequences in different places, as long as the same value is produced everywhere, so that the name consistently appears to refer to the same variable in all parts of its scope.

By convention, a local variable like `array` is copied into a `private` field `val$array` of the inner class. (Because `array` is `final`, such copies never contain inconsistent values.) Each copied value is passed to the inner class constructor as a separate argument of the same name.

Here is what the resulting transformed code looks like:

```
    Enumeration myEnumerate(final Object array[]) {
        return new MyOuterClass$19(array);
    }
...
class MyOuterClass$19 implements Enumeration {
    private Object val$array[];
    int count;
    MyOuterClass$19(Object val$array[])
        { this.val$array = val$array; count = 0; }
    public boolean hasMoreElements()
        { return count < val$array.length; }
    public Object nextElement()
        { return val$array[count++]; }
}
```

A compiler may avoid allocating an inner class field to a variable, if it can determine that the variable is used only within the inner class constructors.

Notice that a class defined by a block, like `E`, is not a member of its enclosing class, and so it cannot be named outside of its block. This is the same scoping restriction as applies to local variables, which also cannot be named outside of their blocks. In fact, any class contained in a block (whether directly or inside an intervening local class) cannot be named outside the block. All such classes are called *inaccessible*. For purposes of linking, the compiler must generate a unique externally visible name for every inaccessible class. The overall form of these names is a class name, followed by additional numbers or names, separated by `$` characters.

Also, variable names synthesized by the compiler beginning with `this$` and `val$` must follow the usage patterns described here.

These names and conventions must be recognized by 1.1-compliant tools, and are strongly suggested for most compilation purposes. They are discussed further in the section on binary compatibility.

It must be emphasized that these oddly-named “`this$`“ and “`val$`“ fields and extra constructor arguments are added by the compiler to the generated bytecodes, and cannot be directly referenced by Java source code. Likewise, bytecode-level class names like `MyOuterClass$19` cannot be used by source code (except under pre-1.1 compilers, which know nothing of inner classes).

Why does Java need inner classes?

From the very beginnings of Java, its designers have recognized the need for a construct like a “method pointer,” which (in all its various forms) amounts to a handle on an individual block of code which can be used without reference to the object or class containing the code. In languages (like C or Lisp) where functions are free standing, independent of objects, function pointers serve this role. For example, these pointers often serve to connect a “callback” or “event” in one module to a piece of code in another. In a more object oriented style, Smalltalk has “blocks,” which are chunks of code that behave like little objects. As with C or Lisp function pointers, Smalltalk blocks can be used to organize complex control flow patterns, such as iteration over collections.

In Java, the same complex control flow patterns, including event management and iteration, are expressed by classes and interfaces. Java uses interfaces with one method where other languages might use separate “function types.” The Java programmer creates the equivalent of a callback or a Smalltalk block by wrapping the desired code in an adapter class which implements the required interface. With inner classes, the notation for adapters is about as simple as that of Smalltalk blocks, or of inner functions in other languages. However, since classes are richer than functions (because they have multiple entry points), Java adapter objects are more powerful and more structured than function pointers.

So, whereas C, Lisp, and Smalltalk programmers use variations of “method pointers” to encapsulate chunks of code, Java programmers use objects. Where other languages have specialized function types and notations to encapsulate behavior as functions, Java has only class and interface types. In Java, “the class is the quantum of behavior.” One benefit of this approach is simplicity and stability for the Java Virtual Machine, which needs no special support for inner classes or function pointers.

Without inner classes, Java programmers can create callbacks and iterators by means of adapter classes defined at top-level, but the notation is so clumsy as to be impractical. By means of inner classes, Java programmers can write concise adapter classes which are coded precisely where they are needed, and operate directly on the internal variables and methods of a class or a block.

Thus, inner classes make adapter classes practical as a coding style. In the future, inner classes will also be more efficient than equivalent top-level adapter classes, because of increased opportunities for optimization, especially of (externally) inaccessible classes.

Why anonymous classes?

An anonymous class is an abbreviated notation for creating a simple local object "in-line" within any expression, simply by wrapping the desired code in a "`new`" expression.

As noted previously, not every inner class should be anonymous, but very simple "one-shot" local objects are such a common case that they merit some syntactic sugar.

Anonymous classes are useful for writing small encapsulated "callbacks," such as enumerations, iterators, visitors, etc. They are also helpful for adding behavior to objects which already have names, such as AWT components (to which anonymous event handlers are added), and threads. In both cases, an intervening class name can detract from the clarity of the code.

Several other languages from which Java derives inspiration, such as Smalltalk and Beta, offer similar shorthands for anonyous objects or functions.

What about dynamic typing and computed selectors ("`perform`")?

In order to support the construction of robust and secure systems, Java is statically typed. In other languages, callbacks sometimes take a form which is untyped, or dynamically typed. C callbacks usually work with an untyped "client data" address, while Smalltalk classes sometimes plug into each other by means of symbolic method references computed at run time, which are passed to an interpretive "`perform`" method.

The closest equivalent to a C `void*` pointer in Java is a reference of type `Object`. As in C, it is possible to program in Java with such "untyped" references. A generic "argument" field in an event descriptor might be an undifferentiated `Object`, as is the element type of `java.util.Vector`. Coding with untyped references is sometimes a workable technique, despite the execution costs of dynamic type checking, but the lack of static declarations can make programs hard to understand and maintain.

Also, some applications for "method pointer" constructs, such as application builders or the Java Beans component framework, have needed the ability to invoke a method of a computed name on an arbitrary object. This capability is provided by the Java Core Reflection API, `java.lang.reflect`, a new Java 1.1 API.

How do inner classes affect the idea of `this` in Java code?

Recall that the code of a top-level class *T* can make use of the *current instance*, either directly by means of the keyword `this`, or indirectly, by naming one of the instance members of *T*. The idea of a current instance has been extended, so that inner classes have more than one current instance. This allows the code of an inner class *C* to execute relative to an enclosing instance of its outer class, as well as the current instance of *C*.

In our first example of adapter classes, the `Enumerator` code had two current instances, the enclosing `FixedStack` and the `Enumerator` object itself. More generally, within the entire body of an inner class *C*, there is a current instance for *C*, another current instance for every inner class enclosing *C*, and finally a current instance for the innermost enclosing top-level class. (A `static` class is a top-level class, not an inner class. Therefore, our enumeration of current instances stops at the first `static` keyword.)

Top-level classes do not have multiple current instances. Within the non-`static` code of a top-level class *T*, there is one current instance of type *T*. Within the `static` code of a top-level class *T*, there are no current instances. This has always been true of top-level classes which are package members, and is also true of top-level classes which are `static` members of other top-level classes. It is as if all package members have been implicitly `static` all along.

In all classes, each current instance can be named explicitly or can play an implicit part when its members are used. Any current instance can be referred to by explicitly qualifying the keyword `this` as if it were a name: `FixedStack.this`, `Enumerator.this`, etc. (This notation is always effective, since the language prohibits an inner class from having the same name as any of its enclosing classes.) As always, the innermost current instance can be named with the unqualified keyword `this`.

Remember that an instance variable reference *m* has the same meaning as a field reference expression `this.`*m*, so the current instance is implicitly used. In a given piece of code, all members of all current classes are in scope and usable (except for name hiding by intervening scopes). The simple name of a member is implicitly qualified by the current instance in whose class the name was found. (All `static` members of enclosing classes are always usable.)

In particular, Java code can directly use all methods of all current instances. Class scoping does not influence overloading: If the inner class has one `print` method, the simple method name `print` refers to that method, not any of the ten `print` methods in the enclosing class.

Enclosing classes and instantiation

It is sometimes useful to speak of an *enclosing instance* of an inner class *C*. This is defined as any current instance within *C*, other than the instance of *C* itself. Every instance of *C* is permanently associated with its enclosing instances.

When a constructor is invoked, the new object must be supplied with an enclosing instance, which will become a current instance for all the code executed by the new object. (If there are layers of enclosing instances, the innermost is required, and it in turn determines all the others.) The enclosing instance may be specified explicitly, by qualifying the keyword `new`:

```
class Automobile {
    class Wheel {
        String hubcapType;
        float radius;
    }

    Wheel leftWheel = this. new Wheel();
    Wheel rightWheel = this. new Wheel();
    Wheel extra;

    static void thirdWheel(Automobile car) {
        if (car.extra == null) {
            car.extra = car. new Wheel();
        }
        return car.extra;
    }
}

class WireRimWheel extends Automobile.Wheel {
    WireRimWheel(Automobile car, float wireGauge) {
        car. super();
    }
}
```

A subclass of an inner class *C* must pass an enclosing instance to *C*'s constructor. This may be done, as just shown, by explicitly qualifying the keyword `super`. By default, a current instance of the caller becomes the enclosing instance of a new inner object. In an earlier example, the expression `new Enumerator()` is equivalent to the explicitly qualified `this.new Enumerator()`. This default is almost always correct, but some applications (such as source code generators) may need to override it from time to time.

Do inner classes affect the correct synchronization of Java code?

An inner class is part of the implementation of its enclosing class (or classes). As such, it has access to the `private` members of any enclosing class. This means that the programmer must be aware of the possibility of concurrent access to state stored in `private` variables, and ensure that non-`private` methods are correctly synchronized. Sometimes this just means that the enclosing method needs to be declared with the `synchronized` keyword.

When more than one object is involved, as with `FixedStack` and its enumerator, the programmer must choose which instance to synchronize upon, and write an explicit `synchronized` statement for the enclosing instance:

```
public Object nextElement() {
    ...
    synchronized (FixedStack.this) {
        return array[--count];
    }
}
```

There is no special relation between the `synchronized` methods of an inner class and the enclosing instance. To synchronize on an enclosing instance, use an explicit `synchronized` statement.

When writing multi-threaded code, programmers must always be aware of potential asynchronous accesses to shared state variables. Anonymous inner classes make it extremely easy to create threads which share private fields or local variables. The programmer must take care either to synchronize access to these variables, or to make separate copies of them for each thread. For example, this `for`-loop needs to make copies of its index variable:

```
for (int ii = 0; ii < getBinCount(); ii++) {
   final int i = ii; // capture a stable copy for each thread
    Runnable r = new Runnable() {
        public void run() { processBin(i); }
    };
    new Thread(r, "processBin("+i+")").start();
}
```

It is a common mistake to try to use the loop index directly within the inner class body. Since the index is not `final`, the compiler reports an error.

Can a nested class be declared `final`, `private`, `protected`, or `static`?

All the existing access protection and modification modes apply in a regular fashion to types which are members of other classes. Classes and interfaces can be declared `private` or `protected` within their enclosing classes.

A class which is local to a block is not a member, and so cannot be `public`, `private`, `protected`, or `static`. It is in effect private to the block, since it cannot be used outside its scope.

Access protection never prevents a class from using any member of another class, as long as one encloses the other, or they are enclosed by a third class.

Any class (if it has a name) can be declared `final` or `abstract`, and any accessible non-`final` named class or interface can serve as a supertype. A compiler may also change a class to be `final` if it can determine that it has no subclasses, and that there is no way for subclasses to be added later. This is possible when a `private` or block-local class has no subclasses in its scope.

Members that can be marked `static`

The `static` declaration modifier was designed to give programmers a way to define *class methods* and *class variables* which pertain to a class as a whole, rather than any particular instance. They are "top-level" entities.

The `static` keyword may also modify the declaration of a class *C* within the body of a top-level class *T*. Its effect is to declare that *C* is also a top-level class. Just as a class method of *T* has no current instance of *T* in its body, *C* also has no current instance of *T*. Thus, this new usage of `static` is not arbitrary.

As opposed to top-level classes (whether nested or not), inner classes cannot declare any `static` members at all. To create a class variable for an inner class, the programmer must place the desired variable in an enclosing class.

It is helpful at this point to abuse the terminology somewhat, and say, loosely, that the `static` keyword always marks a "top-level" construct (variable, method, or class), which is never subject to an enclosing instance.

This shows why an inner class cannot declare a `static` member, because the entire body of the inner class is in the scope of one or more enclosing instances.

While the C language allows block-local `static` variables, the same effect can be obtained in Java, more regularly and maintainably, by defining the desired long-lived variable in the scope which corresponds to the required lifetime.

How do inner classes affect the organization of the Java Virtual Machine?

There are no changes to the class file format as processed by the Java Virtual Machine, or to the standard class libraries. The new features are implemented by the compiler. The organization of the resulting bytecodes is specified with enough precision that all 1.1-conforming compilers will produce binary compatible class files.

A single file of Java source code can compile to many class files. Although this is not a new phenomenon, the power of the inner class notation means that the programmer can end up creating a larger number of class files with relatively less code. In addition, adapter classes tend to be very simple, with few methods. This means that a Java program which uses many inner classes will compile to many small class files. Packaging technologies for such classes process them reasonably efficiently. For example, the class file for the example class `FixedStack.Enumeration` occupies about three quarters of a kilobyte, of which about 40% is directly required to implement its code. This ratio is likely to improve over time as file formats are tuned. The memory usage patterns in the virtual machine are comparable.

Class name transformations

Names of nested classes are transformed as necessary by the compiler to avoid conflicts with identical names in other scopes. Names are encoded to the virtual machine by taking their source form, qualified with dots, and changing each dot '`.`' after a class name into a dollar sign '`$`'. (Mechanical translators are allowed to use dollar signs in Java.)

When a class name is `private` or local to a block, it is globally inaccessible. A compiler may opt to code such an inaccessible name by using an accessible enclosing class name as a prefix, followed by a '`$`' separator and a locally unique decimal number. Anonymous classes must be encoded this way.

So, an inner class `pkg.Foo.Bar` gets a run-time name of `pkg.Foo$Bar`, or perhaps something like `pkg.Foo$23`, if Bar is a `private` member or local class. Implementations must conform to the format of names, even globally inaccessible ones, so that debuggers and similar tools can recognize them.

Any class file which defines or uses a transformed name also contains an attribute (as supported by the 1.0 file format) recording the transformation. These attributes are ignorable by the virtual machine and by 1.0 compilers. The format of this attribute is described in the section on binary compatibility.

Names of generated variables and methods

As we have seen previously, if an inner class uses a variable from an enclosing scope, the name expression will be transformed, into a reference either to a field of an enclosing instance, or to a field of the current instance which provides the value of a `final` local variable. A reference to an enclosing instance, in turn, is transformed into a reference to a field in a more accessible current instance. These techniques require that the compiler synthesize hidden fields in inner classes.

There is one more category of compiler-generated members. A `private` member *m* of a class *C* may be used by another class *D*, if one class encloses the other, or if they are enclosed by a common class. Since the virtual machine does not know about this sort of grouping, the compiler creates a local protocol of access methods in *C* to allow *D* to read, write, or call the member *m*. These methods have names of the form `access$0`, `access$1`, etc. They are never public. Access methods are unique in that they may be added to *enclosing* classes, not just inner classes.

All generated variables and methods are declared in a class file attribute, so that the 1.1 compilers can prevent programs from referring to them directly.

Security implications

If an inner class *C* requires access to a `private` member *m* of an enclosing class *T*, the inserted access method for *m* opens up *T* to illegal access by any class *K* in the same package. There at present are no known security problems with such access methods, since it is difficult to misuse a method with package scope. The compiler can be instructed to emit warnings when it creates access methods, to monitor the creation of possible loopholes.

If a class *N* is a `protected` member of another class *C*, then *N*'s class file defines it as a `public` class. A class file attribute correctly records the protection mode bits. This attribute is ignored by the current virtual machine, which therefore will allow access to *N* by any class, and not just to subclasses of *C*. The compiler, of course, will correctly diagnose such errors, because it looks at the attribute. This is not a security hole, since malicious users can easily create subclasses of *C* and so gain access to *N*, `protected` or not.

Likewise, if a class is a `private` member of another class, its class file defines it as having package scope, and an attribute declares the true access protection, so that 1.1 compilers can prevent inadvertant access, even within the package.

How does the Java Language Specification change for inner classes?

There are few significant changes, since the new features primarily relax restrictions in the existing language, and work out new implications for the old design. The key change is that types can now have types as members. (But type names can't contain instance expressions.) The basic definitions of scope, name scoping, member naming, and member access control are unchanged.

Here are the extensions to the class body and block syntax:

> *ClassMemberDeclaration, InterfaceMemberDeclaration:*
>   *...*
>   *ClassDeclaration*
>   *InterfaceDeclaration*
>
> *BlockStatement:*
>   *...*
>   *ClassDeclaration*

A type which is a type member is inherited by subtypes, and may be hidden in them by type declarations of the same name. (Types are never "virtual.") Members which are types may be declared `private` or `protected`.

A non-`static` member class, or a class defined by a block or expression, is an *inner* class. All other classes are *top-level*. Inner classes may not declare `static` members, `static` initializers, or member interfaces. Package members are never `static`. But a class which is a member of a top-level class may be declared `static`, thereby declaring it also to be a top-level class. Interfaces are always `static`, as are their non-method members.

A class may not have the same simple name as any of its enclosing classes.

The keyword `this` can be qualified, to select one of possibly several current instances. (Inner classes have two or more current instances.)

> *PrimaryNoNewArray:*
>   *...*
>   *ClassName* `. this`

The syntax for class instance creation extended to support anonymous classes and enclosing instances:

> *ClassInstanceCreationExpression:*
>   `new` *TypeName* `(` *ArgumentList*$_{opt}$ `)` *ClassBody*$_{opt}$
>   *Primary* `. new` *Identifier* `(` *ArgumentListOpt* `)` *ClassBody*$_{opt}$

A `new` expression may define an anonymous class by specifying its body. Independently, the type of a `new` expression may specified as the simple name of an inner class, if an instance of the immediately enclosing class is given as a qualifying expression before the keyword `new`. The qualifying instance becomes the enclosing instance of the new object. A corresponding qualification of `super` allows a subclass constructor to specify an enclosing instance for a superclass which is an inner:

ExplicitConstructorInvocation: ...
    Primary `. super (` *ArgumentList*$_{Opt}$ `) ;`

If an inner class is constructed by an unqualified `new` or `super` expression, the enclosing instance will be the (innermost) current instance of the required type.

Some of the detailed descriptions of name binding in the 1.0 Java Language Specification require amendment to reflect the new regularity in lexical scoping. For example, a simple variable name refers to the innermost lexically apparent definition, whether that definition comes from a class or a block. The same is true for simple type names. The grammar for a qualifier name (i.e., an *AmbiguousName*) is extended to reflect the possibility of class names qualifying other type names. The initial simple name in a qualified type name is taken to be a class name if a class of that name is in scope; otherwise it is taken to be a package name, as in Java 1.0.

Any inherited member *m* of a subclass *C* is in scope within the body of *C*, including any inner classes within *C*. If *C* itself is an inner class, there may be definitions of the same kind (variable, method, or type) for *m* in enclosing scopes. (The scopes may be blocks, classes, or packages.) In all such cases, the inherited member *m* hides the other definitions of *m*. Additionally, unless the hidden definition is a package member, the simple name *m* is illegal; the programmer must write *C*`.this.`*m*.

Nested classes of all sorts (top-level or inner) can be imported by either kind of `import` statement. Class names in `import` statements must be fully package qualified, and be resolvable without reference to inheritance relations. As in Java 1.0, it is illegal for a class and a package of the same name to co-exist.

A `break` or `continue` statement must refer to a label within the immediately enclosing method or initializer block. There are no non-local jumps.

The checking of definite assignment includes classes defined by blocks and expressions, and extends to occurrences of variables within those classes. Any local variable used but not defined in an inner class must be declared `final`, and must be definitely assigned before the body of the inner class.

Other changes in the Java 1.1 language

The Java 1.1 language includes four additional extensions which fill small holes in the language, and make certain kinds of APIs easier to use.

Instance initializers

The static initializer syntax is extended to support instance initialization also:

ClassBodyDeclaration:
    Block

Initialization code introduced without the `static` keyword is executed by every constructor, just after the superclass constructor is called, in textual order along with any instance variable initializations.

An instance initializer may not return, nor throw a checked exception, unless that exception is explicitly declared in the `throws` clause of each constructor. An instance initializer in an anonymous class can throw any exceptions.

Instance initializers are useful when instance variables (including blank finals) must be initialized by code which must catch exceptions, or perform other kinds of control flow which cannot be expressed in a single initializer expression. Instance initializers are required if an anonymous class is to initialize itself, since an anonymous class cannot declare any constructors.

Anonymous array expressions

The array allocation syntax is extended to support initialization of the elements of anonymous arrays. Just as a named array can be initialized by a brace-enclosed list of element expressions, an array creation expression can now be followed by such a brace-enclosed list. In both cases, the array type is not allowed to include any dimension expressions; the dimension is computed by counting the number of element expressions. Here is the new syntax:

ArrayCreationExpression:
    `new` *Type Dims ArrayInitializer*

The equivalence of the following two statements illustrates the new syntax:

```
T v[] = {a};
T v[] = new T[] {a};
```

Class literals

PrimaryNoNewArray:
  ...
  Type `. class`
  `void . class`

A *class literal* is an expression consisting of the name of a class, interface, array, or primitive type followed by a '`.`' and the token `class`. It evaluates to an object of type `Class`, the class object for the named type (or for void).

For reference types, a class literal is equivalent to a call to `Class.forName` with the appropriate string, except that it does not raise any checked exceptions. (Its efficiency is likely to be comparable to that of a field access, rather than a method call.) The class literal of a reference type can raise `NoClassDefFoundError`, in much the same way that a class variable reference can raise that error if the variable's class is not available.

The class literal of a primitive type or void is equivalent to a static variable reference to a pre-installed primitive type descriptor, according to this table:

```
boolean.class    ==    Boolean.TYPE
char.class       ==    Character.TYPE
byte.class       ==    Byte.TYPE
short.class      ==    Short.TYPE
int.class        ==    Integer.TYPE
long.class       ==    Long.TYPE
float.class      ==    Float.TYPE
double.class     ==    Double.TYPE
void.class       ==    Void.TYPE
```

Java APIs which require class objects as method arguments are much easier to use when the class literal syntax is available. Note that the compiler is responsible for taking into account the ambient `package` and `import` statements when processing the *TypeName* of a class literal.

The older usage of `Class.forName` requires the programmer to figure out the desired package prefix and write it in a class name string. The difficulty of getting the string spelled right becomes greater in the presence of inner classes, since their names (as processed by `Class.forName`) are encoded with '`$`' characters instead of dots.

Note that a class literal never contains an expression, only a type name.

Blank finals and final local variables

A *blank final* is `final` variable declaration (of any kind) which lacks an initializer. A blank final must be assigned an initial value, at most once.

The definite assignment rules are extended to record variables which are "definitely unassigned," and an assignment to a blank final is prohibited unless the final is definitely unassigned before the assignment statement. Subsequently, it is definitely assigned, and, being a `final`, it cannot be re-assigned along the same execution path.

The definite unassignment rules take into account back-branches of loops, so that a variable occurrence in a loop body may not be definitely unassigned if the loop makes an assignment which can reach the occurrence via a back-branch. The definite assignment checks work as if the first iteration of the loop had been unrolled into an `if` statement.

A blank final class variable must be definitely assigned by a `static` initializer (in the same class). This is the only context where class variables are checked for definite assignment.

A blank final instance variable must be definitely assigned by a non-`static` initializer, or else by every constructor. These are the only contexts in which definite assignment checking is done on instance variables. Within these contexts, an assignment to `this.`*V* is recognized as performing an assignment to the name *V* for purposes of definite assignment checking.

Local variables and parameters of all sorts can now be declared `final`:

LocalVariableDeclaration:
    Modifiers$_{Opt}$ *Type VariableDeclarators*

FormalParameter:
    Modifiers$_{Opt}$ *Type VariableDeclaratorId*

Such a variable is subject to the usual definite assignment rules governing local variables. In addition, it cannot be assigned to, except for initialization.

A method parameter or catch formal parameter may be declared `final`. This has no effect on the method signature or the caught exception type. Within the body of the method or catch, the parameter may not be assigned to.

The `final` declaration modifier may be used to make local variables and parameters available to inner classes.

What are the new binary compatibility requirements for Java 1.1 classes?

In order to binary ensure compatibility between bytecodes output for Java 1.1 compilers from different vendors, and to ensure proper applicability of debuggers and similar tools to those bytecodes, Java makes certain requirements on the form of the bytecodes produced. This section describes the requirements, new in Java 1.1, which pertain to the implementation of various kinds of inner and nested top-level classes.

Bytecode names of classes and interfaces

Instances of the Java Virtual Machines, and Java bytecodes, refer to reference types by means of *bytecode names* which differ in detail from the names used in Java source code. The bytecode name of a package member *T* is defined as the name of the package, with every '`.`' replaced by '`/`', followed (if the package name is not null) by another '`/`', and then by the simple name of *T*. The bytecode name of *T* also serves as a prefix for the bytecode name of every class defined within the body of *T*.

The bytecode name of a class *C* which is a non-`private` member of another class, and which is not contained (directly or indirectly) in any block or `private` class, is defined as the bytecode name of the immediately-enclosing class, followed by '`$`', followed by the simple name of *C*.

All other classes are called *inaccessible*. No inaccessible class *N* can ever be referenced by the code of any other compilation unit. Thus, as long as the name of *N* is chosen by the compiler in such as way as not to conflict with any other class in the same compilation unit, the name will be globally unique, because (as required previously) its prefix is unique to the package member in which it occurs.

For the sake of tools, there are some additional requirements on the naming of an inaccessible class *N*. Its bytecode name must consist of the bytecode name of an enclosing class (the immediately enclosing class, if it is a member), followed either by '`$`' and a positive decimal numeral chosen by the compiler, or by '`$`' and the simple name of *N*, or else by both (in that order). Moreover, the bytecode name of a block-local *N* must consist of its enclosing package member *T*, the characters '`$1$`', and *N*, if the resulting name would be unique.

The string produced by the `getName` method of `Class` is derived, in all of these cases, from the bytecode name, by replacing '`/`' by '`.`'. There is no attempt to "clean up" the name to make it resemble Java source code.

The class attribute `InnerClasses`

The bytecode output of a Java 1.1 compiler may refer (via `CONSTANT_Class` entries) to bytecode names of classes or interfaces which are not package members. If so, the bytecodes must also contain an class attribute called `InnerClasses` which declares the encoding of those names. This attribute contains an array of records, one for each encoded name:

```
InnerClasses_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 number_of_classes;
  {
    u2 inner_class_info_index;   // CONSTANT_Class_info index
    u2 outer_class_info_index;   // CONSTANT_Class_info index
    u2 inner_name_index;         // CONSTANT_Utf8_info index
    u2 inner_class_access_flags; // access_flags bitmask
  } classes[number_of_classes]
}
```

Each array element records a class with an encoded name, its defining scope, its simple name, and a bitmask of the originally declared, untransformed access flags. If an inner class is not a member, its `outer_class_info_index` is zero. If a class is anonymous, its `inner_name_index` is zero.

If a class *C* was declared `protected`, the `public` access flag bit is cleared in its `InnerClasses` record, even though it is set in *C*'s `access_flags` field.

If the `outer_class_info_index` of a record refers to a class *E* which itself is not a package member, then an earlier record of the same `InnerClasses` attribute must describe *E*.

If a class has members which are types, it must have an `InnerClasses` attribute, with a record for each of the types. The rules already given imply that a class which is not a package member has an `InnerClasses` attribute which has a record for it and all of its enclosing classes, except the outermost.

These rules ensure that compilers and debuggers can correctly interpret bytecode names without parsing them, and without opening additional files to examine inner class definitions. Compilers are allowed to omit `InnerClasses` records for inaccessible classes, but they are encouraged to include records for all classes, especially when the code is being compiled for use with a debugger.

The member attribute `Synthetic`

As discussed previously, the compiler synthesizes certain hidden fields and methods in order to implement the scoping of names. These fields are `private` unless noted otherwise, or they are at most of package scope.

Java 1.1 compilers are required, when producing bytecodes, to mark any field or member not directly defined in the source code with an attribute named `Synthetic`. (At present, the length must be zero.) This will allow other compilers to avoid inadvertant source-level references to non-private hidden members, and will allow tools to avoid displaying them unnecessarily.

(A corresponding mechanism for declaring a local variable to be `Synthetic` may also be introduced.)

Java 1.1 compilers are strongly encouraged, though not required, to use the following naming conventions when implementing inner classes. Compilers may not use synthetic names of the forms defined here for any other purposes.

A synthetic field pointing to the outermost enclosing instance is named `this$0`. The next-outermost enclosing instance is `this$1`, and so forth. (At most one such field is necessary in any given inner class.) A synthetic field containing a copy of a constant *v* is named `val$`*v*. These fields are `final`.

All these synthetic fields are initialized by constructor parameters, which have the same names as the fields they initialize. If one of the parameters is the innermost enclosing instance, it is the first. All such constructor parameters are deemed to be synthetic. If the compiler determines that the synthetic field's value is used only in the code of the constructor, it may omit the field itself, and use only the parameter to implement variable references.

A non-`private final` synthetic method which grants access to a private member or constructor has a name of the form `access$`*N*, where *N* is a decimal numeral. The organization of such access protocols is unspecified.

Debuggers and similar tools which are 1.1 compatible must recognize these naming conventions, and organize variable displays and symbol tables accordingly. Note that tools may need to parse these names. Compilers are strongly encouraged to use these conventions, at least by default.

Implementations of the Java Virtual Machine may verify and require that the synthetic members specified here are defined and used properly. It is reasonable to exploit the nature of synthetic members by basing optimization techniques on them.

Further Example: Sample AWT code

The 1.1 version of the Java Abstract Window Toolkit provides a new event handling framework, based on "event listener" interfaces, to which the programmer must write callback objects. The callbacks amount to a flexible new layer between the GUI and the application's data structures. These adapters must be subclassed to hook them up to the application. The point of this is to avoid the need to subclass the GUI components themselves, or to write complicated `if/else` and `switch` statements to interpret event codes.

This design requires that the adapter classes be simpler to write and maintain than the corresponding `if/else` and `switch` code! This is where inner classes become important.

Here is an typical example of AWT event handling code. It uses a named inner class `App.GUI` to organize the GUI code, and anonymous adapters to tie individual GUI components to the application's methods:

```
public class App {
    void search() { ...do search operation...}
    void sort() { ...do sort operation ... }
    static public void main(String args[]) {
        App app = new App(args);
       GUI gui = app.new GUI();   // make a new GUI enclosed by app
    }
    class GUI extends Frame {   // App.GUI is enclosed in an App.
        public GUI() {
            setLayout(new FlowLayout());
            Button b;
            add(b = new Button("Search"));
            b.setActionListener(
                new ActionAdaptor() {
                    public void actionPerformed(ActionEvent e) {
                        search();     // App.this.search()
                    }
                }
            );
            ... build a Sort button the same way ...
           pack(); show();
        }
    }
    ...
}
```

Further Example: An API with coordinated inner classes

Sometimes a class-based API will include as an essential feature secondary classes or interfaces. These latter can be structured quite naturally as `static` inner classes of the main class.

To see an example of this, imagine a hypothetical utility `Sort` with an interface `Comparer` which virtualizes the comparison operation, and a handful of standard reusable comparison implementations. (This example has a flaw: `Comparer` is generic enough to stand alone.) The code might look like this:

```
public class Sorter {
    public interface Comparer {
        /** Returns <0 if x < y, etc. */
        int compare(Object x, Object y);
    }
    public static void sort(Object keys[], Comparer c) {...}
    public static void sort(Object keys[], Comparer c,
                            Object values[]) {...}
    public static void sort(String keys[], Object values[])
        { sort(keys, stringComparer, values); }

    public static class StringComparer implements Comparer {
        public int compare(Object x, Object y) {
            if (x == null)  return (y == null) ? 0 : -1;
            if (y == null)  return 1;
            return x.toString().compareTo(y.toString());
        }
    }
    public static final Comparer stringComparer
            = new StringComparer();

    public static class LongComparer implements Comparer {
            ... long lx = ((Number)x).longValue(); ...
    }
    public static final Comparer longComparer
            = new LongComparer();

   /** Compose 2 comparisons, presumably on distinct sub-keys. */
   public static class CombinedComparer implements Comparer {...}
   public static Comparer combine(Comparer c1, Comparer c2) {...}
    ...
}
```

Further Example: Multi-threaded task partitioning.

It is sometimes useful to parallelize a task with independent sub-tasks, by assigning each sub-task to a thread. This can make the whole task finish sooner, if multiple processors are involved. (This will be the case, if the sub-tasks perform network traffic.) In interactive Java programs, multithreading is also used to enable partial results from sub-tasks to be pushed through to the end user, while slower sub-tasks continue to grind away.

The code below (based on an example from Doug Lea) shows a very simple way to control the rendering of several pictures in such a way that each picture is delivered to a displayer as soon as it's ready, but the original requester blocks until all rendering is finished. Each sub-task is coded by an anonymous implementation of `Runnable` which is at the heart of each thread.

```
public class GroupPictureRenderer {
  private PictureRenderer renderer;
  private PictureDisplayer displayer;
  ...

  public Picture[] render(final byte[][] rawPictures)
                    throws InterruptedException {
    Thread workers[] = new Thread[rawPictures.length];
    final Picture results[] = new Picture[rawPictures.length];
    // start one thread per rendering sub-task
    for (int ii = 0; ii < rawPictures.length; ii++) {
      final int i = ii;   // capture ii for each new thread
      Runnable work = new Runnable() {
            public void run() {
              results[i] = renderer.render(rawPictures[i]);
              displayer.display(results[i]);
            }
      };
      workers[i] = new Thread(work, "Renderer");
      workers[i].start();
    }
    // all threads are running; now wait for them all to finish
    for (int i = 0; i < workers.length; i++)
      workers[i].join();
    // give all the finished pictures to the caller, too:
    return results;
  }
}
```