Chapter 3

The `static` Modifier, `this`, and `super`

Chapter Contents

3.1 Introduction

The `static` keyword is the opposite of `this` and `super`. Where one exists the others do not. Consequently, this chapter has two major divi-

sions. The first is an elaborate definition of the `static` keyword. The second is about the `this` and `super` keywords. To my knowledge this is the first Java book to present these subjects together and in their own chapter.

The realization that these subjects are so closely related and of sufficient weight to justify their inclusion in a separate chapter came about as a result of my efforts to define the `static` keyword. A defining moment in that effort was when I realized the utter futility of trying to find a single definition. Just as there is no one definition for expressions, there is no one definition for the `static` keyword. The meaning of `static` depends upon the entity, which is a field, method, or class. This chapter includes sections on `static` fields, `static` methods, and `static` classes in which the `static` keyword is defined for that particular kind of entity.

3.2 The `static` Modifier

In early releases of the JVM (now called the classic JVM), one could point to the Method Area as a definition of `static`. In a classic JVM main memory is divided into the Method Area and the heap. Anything defined in the Method Area of a classic JVM is by definition `static`. This includes, notably, the internal table of loaded classes and interfaces (which are basically class files after they are loaded into a JVM), method dispatch tables (from which the Method Area derives its name), and class variables. The Method Area of a classic JVM is literally more "static" than the heap. Early implementations of the JVM did not even unload classes. Once a class was loaded it remained in memory until the JVM was exited.

I retain this discussion of the Method Area for historical reasons. A familiarity with the Method Area helps to understand the use of `static` as a keyword. The newer HotSpot JVM is completely object-oriented. Run-time data structures used exclusively by the JVM are now allocated as objects on the heap. What makes these objects unique is that they are neither created by a programmer nor directly referenced in application programs. They are part of the JVM implementation. The point is that there is no Method Area in the latest implementations of the JVM. This requires a new definition of `static`. It can no longer be defined in terms of memory usage.

In a section entitled "Members that can be marked `static`" in the original *Inner Classes Specification*, John Rose begins by explaining how the `static` modifier was used before inner classes were introduced to the language.

> 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.[1]

The `static` keyword is used in the declaration of *top-level entities*. Table 3-1 includes a complete list of top-level entities in the column marked `static`. Initialization blocks are executed in special initialization methods. Therefore, we can eliminate them from this discussion. That leaves the following **top-level entities**:

- Class variables (including interface constants)
- Class methods
- All interfaces
- Classes that are either implicitly or explicitly declared `static`

Interfaces are excluded from this discussion because they are always either implicitly or explicitly `static`. That leaves only the following three kinds of top-level entities that can be either `static` or non-`static`:

- Fields
- Methods
- Classes

Table 3-1: Terms for Class Body Declarations

Entity	`static`	non-`static`
Field	Class variable	Instance variable
Initialization block	`static` initialization block	Instance initialization block
Constructor		Constructor
Method	Class methods	Instance method
Interface	Nested interface	
Class	Nested top-level class	Inner member class

1. John Rose, *Inner Classes Specification* (Mountain View: Sun Microsystems, 1997), "Can a nested class be declared final, private, protected, or static?"

The meaning of the `static` keyword in relation to each of these three kinds of top-level entities is the subject of the following three subsections.

The remainder of this section discusses the one thing that all top-level entities have in common.

The value of a `static` field can be accessed, a utility method can be invoked, and a nested top-level class can be instantiated without having to instantiate the class in which they are declared.

The `Character` class has all three kinds of top-level entities. For example,

```
class Test {
    public static void main(String[] args) {
       int i = Character.MAX_VALUE;
       System.out.println(i);
       System.out.println(Character.toLowerCase('A'));
    }
}
```

Executing this program prints

```
65535
a
```

Nowhere in this program is the `Character` class instantiated. Nor is the `Character` class instantiated anywhere in the example of the `Character.Unicode` nested top-level class in 1.2 White Space. All you have to do to access the value of a `static` field, to invoke a utility method, or to instantiate a nested top-level class is to qualify the name of the `static` member with the name of the class or interface in which it is declared.

The following entities do not exist unless you first instantiate the class in which they are declared:

- Instance variables
- Instance methods
- Inner classes

For example,

```
class Test {
   public static void main(String[] args) {
       Character c = new Character('a');
       System.out.println(c.hashCode());
   }
}
```

Executing this program prints the character code for the letter `a`, which is `97`. The `hashCode()` method is an instance method. It cannot be invoked until the `Character` class has been instantiated. The `c` variable is the "particular instance" referred to in the above quotation from the *Inner Classes Specification*. This is the fundamental difference between `static` and non-`static` entities.

3.2.1 `static` Fields

An object in memory is little more than a list of instance variables, the values of which collectively represent the state of the object at any given point in time. The value of an instance variable can be different for every instance of the class because instance variables are used as a template when creating objects. Every time the template is used to stamp out a new object, a copy is made of each of the instance variables. You cannot reference an instance variable without first implicitly or explicitly naming an object. Class variables are entirely different. The value of a class variable is the same for all instances of the class.

Class variables have only one value for the entire class.

That is the essential meaning of the `static` keyword in relation to fields. Here is a playful example:

```
class Human {
    public static final String GOD = "good";
    private char sex;
    private int age;
    private float height;
    private float weight;
    private Fingerprint[] fingerprints;
    private DNA dna;
}
```

No matter how many `Human` objects are created, there is only one `GOD`. The value of `GOD` is always `good`, unless hidden in subclasses by the declaration of a different `static` field with the same name.

That "class variables have only one value" is only part of the story. The real question is how `static` fields are used. Answering this question is more difficult than one may think. My analysis is as follows:

- Convenience constants and enumerated types

- True mathematical constants
- Program variables (that is, non-`public` variables declared `static` so they can be used in more than one method)
- One-of-a-kind, system-wide values
- Take-a-number variables (rare)

I believe this analysis is complete but welcome any comments from readers. The list is partly based on the idea of *inconstant constants*. Constants should be either true mathematical constants such as `Math.PI` or what are referred to as *convenience constants* in API docs. Convenience constants take the place of enumerated types, which Java does not support (at least not at the language level).

Non-`public` variables used in more than one method are just a programmer using the `static` keyword to scope a variable to the type in which it is declared. This is a scope issue in application programs, utility classes, and other nonobject-oriented uses of reference data types. That use of `static` fields is entirely different from analyzing the uses of class variables declared `public`. Class variables declared `public` are very special.

In a `public` type, `static` variables declared `public` have a de facto system-wide scope.

Combined with the fact that class variables only have one value, you have one value with a system-wide scope, or a system-wide value. The defining characteristic of a system-wide value is that there is only one such value for the entire system. True mathematical constants, convenience constants, enumerated types, and take-a-number variables are different kinds of system-wide values. The following subsection discusses some one-of-a-kind system-wide values.

3.2.1.1 One-of-a-Kind System-Wide Values

One-of-a-kind **system-wide values** may be either references to objects or primitive data types. Here are some examples from the `java.lang.System`, `java.io.File`, and `java.util.Locale` classes of one-of-a-kind system-wide values that reference objects:

```
private static Properties props;
private static SecurityManager security = null;
private static String tmpdir;
private static Locale defaultLocale;
```

There is only one system `Properties` object, just as there is only one `SecurityManager`, one temporary directory, and one default `Locale`. Here are some more examples from the `File` class of one-of-a-kind system-wide values that are primitive data types:

```
public static final char separatorChar;
public static final char pathSeparatorChar;
```

There is only one file and path separator for any given system.

3.2.1.2 Take-a-Number Variables

I named these variables after the take-a-number dispensers at the deli counter of your local grocery store. They are usually `int` type `static` fields. For example, threads have either names or numbers. Threads that do not have names are referred to as anonymous threads. They are assigned a number in `Thread` class constructors by invoking the `nextThreadNum` method. Here are the relevant declarations:

```
private static int threadInitNumber;
private static synchronized int nextThreadNum() {
   return threadInitNumber++;
}
```

The `static` field `threadInitNumber` is an example of a take-a-number variable.

3.2.2 `static` Methods

The meaning of `static` in the declaration of a class method is that there is no current object. That is why using the `this` keyword in class methods generates a compiler error. I defer that discussion, however, until 3.4.1 The Current Object (`this`). There is only one point I would like to make in this section.

Any `public static` method is a **utility method** regardless of the class in which it is declared.

A utility method is comparable to a function in procedure-oriented programming languages. By this I mean a named sequence of statements that,

when executed, may also return a value. There is nothing object-oriented about `static` methods because there is no current object. They do not need to be declared in a utility class in order to be utility methods. Some utility methods such as `Math.pow(double a, double b)` are declared in utility classes, but most are not.

Why declare utility methods in the body of a reusable object or other non-utility class? In a word, encapsulation. They require access to the non-`public` members of the class in which they are declared. For example, the `ClassLoader.getSystemResource(String name)` utility method in the `java.util` package does not load anything; it only returns an instance of the `URL` class. Why not declare `ClassLoader.getSystemResource(String name)` in a utility class (or even the `URL` class)? Doing so would have created encapsulation problems for the `ClassLoader` class. The utility method in question invokes a `private` method in the `ClassLoader` class.

Sometimes utility methods are at best only logically related to objects of the class in which they are declared. Consider, for example, the `Integer.parseInt(String s)` method. The implementation of this method does not reference any of the fields or methods declared in the `Integer` class. Why not declare `parseInt(String s)` in the `String` class? The `String` class is a reasonable choice because `Integer.parseInt(String s)` parses a `String` object and also invokes the `length()` and `charAt(int index)` methods declared in the `String` class. In this sense it has a stronger affinity to the `String` class than it does to the `Integer` class. The `length()` and `charAt(int index)` methods, however, are `public`. There are, therefore, no encapsulation issues. Because the string argument is supposed to be an integer (in the range of an `int`), the `Integer` class was deemed a logical place to declare this utility method.

3.2.3 `static` Classes

This section is a summary of 2.7 The Definition of Top-Level Classes and is included here for the sake of completeness. The meaning of the `static` keyword in the declaration of a class is the difference between a top-level class and an inner class. As stated in the *JLS*, the effect of the `static` keyword in the declaration of a nested class "is to declare that

[the nested class] is not an inner class."[2] Package members are implicitly `static` and therefore top-level. Nested classes are top-level only if they are implicitly or explicitly declared `static`; otherwise they are inner classes.

To say that a class is top-level or inner is only part of the story. You have to understand the difference between a top-level class and an inner class. That difference is summarized in the following sentence.

> There is only one current instance in the non-`static` code of a top-level class, whereas inner classes always have at least two current instances depending on how deeply nested the inner class is (except for orphaned local and anonymous classes declared in a `static` context).

Notice how this parallels the "class variables have only one value" definition of a `static` field. Multiple current instances are discussed in 3.6 Multiple Current Instances (a.k.a. Levels). The definition of top-level class in that section is much more meaningful than simply saying that the class is implicitly or explicitly `static`.

3.3 The Definition of `static` Context

Many a compiler error message includes the term **`static` context**. For example,

```
class Test {
  int instanceVariable;
  public static void main(String[] args) {
      System.out.println(instanceVariable);
  }
}
```

Attempting to compile this program generates the following compiler error:

```
Test.java:4: non-static variable instanceVariable cannot be referenced from a static context
      System.out.println(instanceVariable);
                         ^
1 error
```

2. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *The Java Language Specification, Second Edition* (Boston: Addison-Wesley, 2000), §8.5.2, "Static Member Type Declarations."

The term `static` *context* has a very precise definition in the *JLS*. There are four `static` contexts in the *JLS*:

- `static` method
- `static` initialization block
- Class variable initializer
- Argument expressions in explicit constructor invocation expressions

The only surprise here is argument expressions in explicit constructor invocation expressions. The constructor body that begins with the explicit constructor invocation expression is not a `static` context, just the argument expressions in the first line of code. For example,

```
class Superclass {
    Superclass(int i) {}
    Superclass(int i) {}
}

class Test extends Superclass {
  int instanceVariable;
  Test() {
      this(instanceVariable);
  }
  Test(int i) { }
}
```

Attempting to compile this program generates the following compiler error:

```
Test.java:9: cannot reference instanceVariable before supertype constructor has
been called
      this(instanceVariable);
           ^
1 error
```

If `this` is changed to `super`, almost the exact same error message is generated.

There are at least two other `static` contexts besides those included in the *JLS*. Qualified access involving type names is a `static` context. That is just another way of saying that only class variables and interface constants can be accessed using the `TypeName.fieldName` general form of field access expressions, and only class methods can be invoked using the `TypeName.methodName` general form method invocation expressions. The other `static` context not mentioned in the *JLS* is `static` context in an inner class hierarchy, which is critically important in

terms of the default qualifying instance. The following section explains in part why I separate containment and inner class hierarchies.

3.3.1 The `static` Context in an Inner Class Hierarchy

An inner member class can be instantiated anywhere in an inner class hierarchy using the default qualifying instance. It is important to remember, however, that the definition of an inner class hierarchy as given in 2.12.2 Inner Class Hierarchies does not include any nested top-level classes or interfaces declared in the body of the top-level class at the top of the inner class hierarchy.

See what happens when I try to instantiate `InnerMemberClass` in a nested top-level class:

```
public class Top  {
    public static class NestedTopLevelClass {
        InnerMemberClass inner = new InnerMemberClass();
    }
    class InnerMemberClass {}
}
```

Attempting to compile this class generates the following compiler error:

```
Top.java:3: non-static variable this cannot be referenced from a static context
        InnerMemberClass inner = new InnerMemberClass();
                                 ^
1 error
```

Insofar as the inner class hierarchy is concerned, anywhere in the body of a nested top-level class or interface is a `static` *context*.

3.4 The `this` and `super` Keywords

The and `super` keywords cannot be used in a `static` context. That includes the `static` code numbered ①, ②, and ③ in Figure 2-1. Everywhere else in a compilation unit (numbers ❹, ❺, ❻, and ❼ in the same figure, as well as inner classes) the `this` keyword represents a complex mechanism that allows a Java programmer to reference the current object. I refer to that mechanism as the **`this` mechanism**. What follows is a brief overview of how the `this` mechanism works.

Instance methods cannot be invoked without first implicitly or explicitly naming an object. That object is known as the **target object**. The target object (as with all objects) has what is called an *object header*, which

includes a reference to the `Class` object. The first thing that happens when an instance method is invoked is that the JVM uses the `Class` object referenced in the header of the target object to locate the method dispatch table.[3] After searching the method dispatch table to find the matching method, the bytecodes for that method are executed. Figure 3-1 is a diagram of the `this` mechanism.

The JVM passes a reference to the target object to instance methods much like it passes a reference to the array in which command-line arguments are stored to the `main(String[] args)` method in an application program. Instead of storing that reference in the `args` parameter, however, the reference to the target is always stored in the first component of the local variable array. The local variable array is an array in which the JVM allocates local variables and parameters during the execution of a method. Supposing that array were called `locals`, the `this` keyword is always a reference to `locals[0]`, which is more commonly referred to as *local variable zero*.

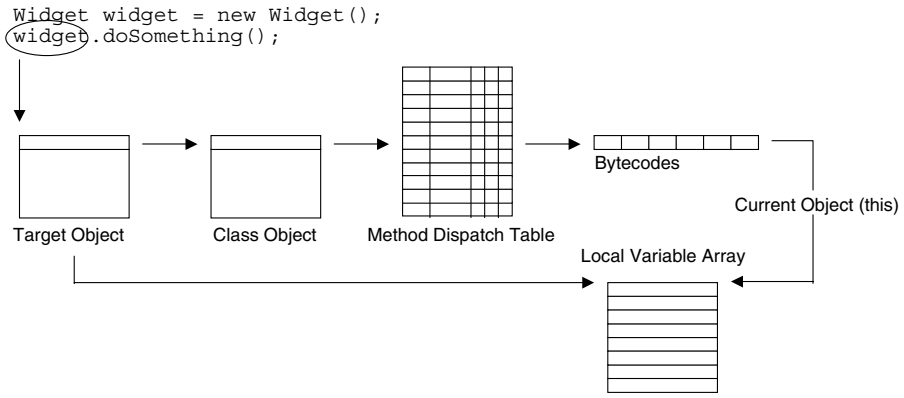


Figure 3-1: The `this` mechanism.

3. While most, if not all, JVM use object headers, they are nevertheless implementation defined. As always, I base my description of the JVM on what I understand to be the latest Sun implementations. I think it is more important to describe one JVM in detail rather than just saying something is implementation defined and leaving it at that. In this case, however, it is reasonable to assume that some JVM implementations do not use the `Class` object to find the method dispatch table. An obvious alternative would be to store a reference to the method dispatch table directly in the object header, thus eliminating a layer of indirection. That, however, increases the size of every object in the system. In general, JVM implementations strive for the smallest memory footprint possible.

Class methods are entirely different. Class methods are almost always invoked using the `TypeName.methodName` general form of a method invocation expression, where `TypeName` is the name of the class in which the method is declared. The JVM uses that type name to go directly to the `Class` object. There is no object involved in locating the method dispatch table for a class method. Moreover, there is no target object to pass to the method invoked.

In the body of the instance method invoked the target object is known as the current object. There are two names for the object involved in invoking an instance method or constructor because that object is seen from two different perspectives. On one hand a client programmer is invoking an instance method. On the other hand is the programmer responsible for implementing that method. Figure 3-2 shows the object from both perspectives. Although the `this` mechanism is completely hidden from both programmers, you know it is there because you are able to reference the current object using the `this` keyword. Think about that for a moment. Unlike every other entity in a compilation unit, there is no declaration for `this`.

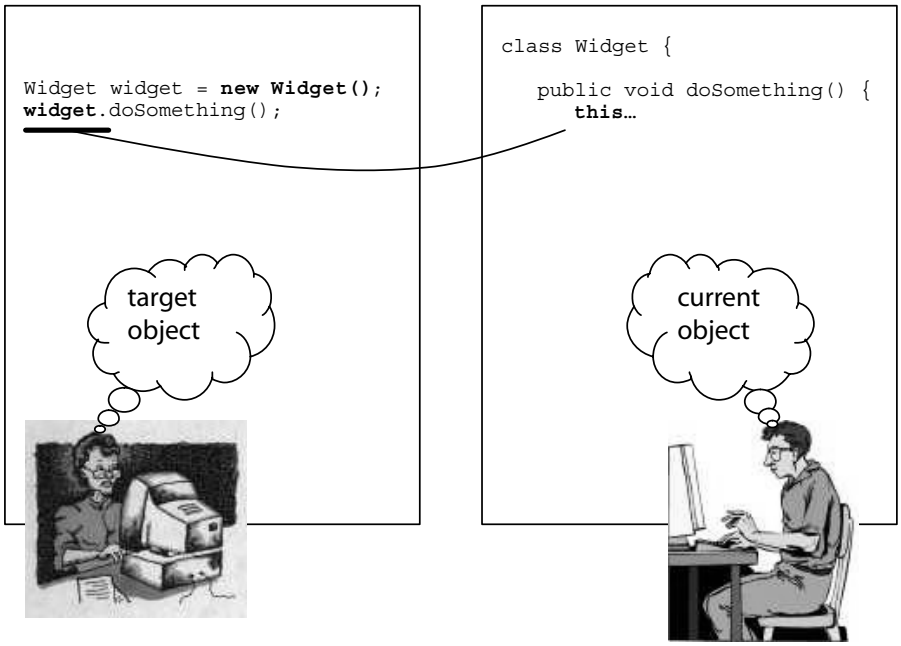


Figure 3-2: Target object and current object are the same.

The `this` keyword is one of the most important mechanisms in the whole of the JVM. Understanding how this mechanism works is something like pulling the curtain back on the Wizard of Oz. I strongly encourage all readers to pursue an uncompromising understanding of the `this` keyword.

3.4.1 The Current Object (`this`)

The current object is always an instance of the class in which the `this` keyword appears. A programmer must think abstractly about an entire class of objects when writing instance methods. However, instance methods are always invoked with respect to a particular instance of that class. From the perspective of the programmer defining the state and behavior of an object, that instance is referred to as either the **current object** or **current instance** (both terms are acceptable). The current object is an abstraction of every object for which the instance method will ever be executed. Because those objects do not exist when the instance method is written, they obviously cannot be referred to by name. Thus, the generic `this` keyword is used in source code to refer to the current object.

3.4.1.1 `this` is Polymorphic, `super` is Not

The compile-time type of the `this` keyword is always the class type in which the keyword appears. At run time, however, `this` is polymorphic. If the member in which the `this` keyword appears is inherited by a subclass, the class of the object referenced by the `this` keyword is the subclass. For example,

```
class Test  {
  public static void main(String[] args) {
     Subclass sub = new Subclass();
     sub.print();
  }
  void print() {
      System.out.println(this.getClass().getName());
  }
}
class Subclass extends Test { }
```

Executing this program prints `"Subclass"`. This much you would expect because the `print()` method is inherited.

The strange thing about the fact that `this` is polymorphic is overridden instance methods and constructors. Constructors are not members of a class and are not, therefore, inherited by subclasses. Nor are overridden

instance methods. Nevertheless, the `this` keyword is polymorphic in both. For example,

```
class Superclass {
  void print() {
      System.out.println(this.getClass().getName());
  }
}
class Test extends Superclass {
  public static void main(String[] args) {
      new Test().print();
  }
  void print() {
     super.print();
  }
}
```

Executing this program prints `"Test"`. The reason for this is explained in detail in the next section. It is not hard to understand what is happening if you know how `super` is implemented in a Java compiler.

The method invoked using the `super` keyword is always found in the method dispatch table of the direct superclass because a special machine instruction is used. The only question is *what reference to pass the instance method invoked*. The answer is to pass a reference to the current object using the `this` keyword, which is polymorphically also an instance of the superclass. Think of `super` as one part of the complex `this` mechanism, all of which is hidden from view.

When using the `super` keyword to invoke an overridden instance method, `this` is implicitly used as the target object.

In simpler terms, using `super` is the same as using `this`. That is why the `super` keyword cannot be used in class variable initializers, `static` initialization blocks, or class methods.

Here is an example of a polymorphic `this` keyword in a constructor:

```
class Test  {
  Test() {
      System.out.println(this.getClass().getName());
  }
  public static void main(String[] args) {
     Subclass sub = new Subclass();
  }
}
class Subclass extends Test { }
```

Executing this program prints "Subclass". The name of the class printed in the Test() constructor is Subclass because a Subclass object is being created, not a Test object.

The class of the object referenced by the this keyword in constructors is always the class of the object under construction.

The explanation for this is exactly the same as when using the super keyword to invoke an overridden superclass method. There is no such thing as a constructor in the JVM. Constructors are incorporated into the instance initialization methods named <init>. When subclasses are instantiated, the design of the JVM is such that one of the instance initialization methods (which may correspond to the default constructor) of the direct superclass is always invoked. Again, the only question is *what reference to pass the instance method invoked*. This time the answer is to pass a reference to the object under construction, which, again, is an instance of the superclass.

3.4.2 The Direct Superclass (super)

As explained in the *JLS*, the super keyword is implemented in Java compilers "as if" it were replaced with a cast of the this keyword. The type name in the cast operator is always the direct superclass of the class in which the super keyword appears. As much as I would like to include an example of how the compiler implements the super keyword using this explanation, doing so within the body of an overriding instance method (which is the primary use of the super keyword) is impossible. For example,

```
class Superclass {
  void print() {
      System.out.println(Superclass.class.getName());
  }
}
class Test extends Superclass {
  public static void main(String[] args) {
      new Test().print();
  }
  void print() {
      ((Superclass)this).print();
  }
}
```

Executing this program results in infinite recursion. The `print` method in the `Test` class is no different than if it were written

```
void print() {
    print();
}
```

The infinite recursion is much more obvious when written like this. That is why the *JLS* must say the `super` keyword is implemented *as if* it were a cast of the `this` keyword. The point of such an example would be to show that the following two lines of code are identical:

```
super.print();
((Superclass)this).print();
```

They are identical for field access expressions or method invocation expressions that invoke anything other than an overridden instance method. For example,

```
class Superclass {
  void print() {
      System.out.println(Superclass.class.getName());
  }
}
class Test extends Superclass {
  public static void main(String[] args) {
      new Test().whatever();
  }
  void whatever() {
      ((Superclass)this).print();
      super.print();
  }
}
```

This program completes normally and prints

```
Superclass
Superclass
```

The difference between a programmer typing `((Superclass)this).print()` in the body of the overriding print method and what the compiler does is the invocation mode. The invocation mode for `((Superclass)this).print()` is `virtual`, which corresponds to the `invokevirtual` machine instruction. The `virtual` invocation mode uses the run-time class of the object referenced to determine which method dispatch table to search, not the compile-time type of the variable or other expression used as the target object in the method invocation expression. Therefore, the `virtual` invocation mode always invokes an overriding method in the class of the object referenced, which explains the infinite recursion in the example at the top of this section.

Casting the variable or other expression used as the target object in such a method invocation expression has no effect.

You'd think that things like invocation modes would be discussed in the *JVMS*, but they are not. There is not a single reference to *invocation mode* anywhere in either edition of the *JVMS*. Invocation modes are *JLS* bytecode mnemonics mentioned only once in the entire *JLS*, in 15.12.3 Compile-Time Step 3: Is the Chosen Method Appropriate?

Invocation modes are implicit in the machine instruction used to invoke a method.

The *JVMS* uses different bytecode mnemonics. For example, the `super` invocation mode in the *JLS* corresponds to the `invokespecial` machine instruction, which is the *JVMS* mnemonic for bytecode 183.[4] It is a special machine instruction that always uses the method dispatch table of the direct superclass. It is used to implement the `super` keyword, invoke `private` instance methods (that cannot be overridden), and to invoke `<init>` methods when compiling class instance creation expressions.

3.5 Practical Uses of the `this` and `super` Keywords

This section *introduces* the reader to the practical uses of the `this` and `super` keywords. I stress the word *introduce* because fully understanding the practical uses of the `this` and `super` keywords is a sign that you have "arrived" in terms of your understanding of the Java programming language. It is not a subject that can be completely addressed in one chapter. The main goal of Table 3-2 is to show the reader that there are a finite number of practical uses for the `this` and `super` keywords. As presented in that table, there are exactly four uses of both and they closely parallel one another, making them even easier to learn. If you become acquainted with these uses and are able to recall some or all of them, then I have achieved the goal of this chapter. It is my belief that defining the boundaries of something that must be eventually learned in much greater detail is useful.

4. I use a fixed font for both *JLS* invocation modes and the corresponding *JVMS* machine instructions. That is just my preference. The `invokespecial` instruction was named `invokenonvirtual` prior to the JDK 1.0.2 release. I do not know the story behind the name change.

Doing so provides a kind of mental picture frame that serves to make the learning process more manageable. In this case the open question of how the `this` and `super` keywords are used is reduced from an infinite unknown to four possibilities. The remainder of this section includes simple examples of each of the uses in Table 3-2.

Table 3-2: Uses of the `this` and `super` Keywords

Description of Use	`this`	`super`
Primary use	A reference to the current object.	Invoking overridden superclass methods from within the body of the overriding subclass.
Accessing shadowed or hidden fields and hidden class methods	When a local variable or parameter is shadowing the name of a field in non-`static` code, the general form `this.fieldName` is used to refer to the hidden field. This is normally done only in constructors. If a class variable is shadowed in a `static` context, it must be referenced using the `TypeName.fieldName` general form.	When a field or class method declared in the current class is hiding a superclass member, the `super.fieldName` and `super.methodName` general forms are used in non-`static` code to refer to the hidden members. If class variables or class methods are hidden in a `static` context, the `TypeName.fieldName` and `TypeName.methodName` general forms must be used.
Explicit constructor invocations	The `this` keyword is used as the first statement in a constructor body to explicitly invoke another constructor in the same class.	The `super` keyword is used as the first statement in a constructor body to explicitly invoke a constructor in the direct superclass.
Miscellaneous uses	Fields are sometimes qualified with the `this` keyword in non-`static` code as a matter of style to differentiate them from local variables and parameters. To achieve the same effect in a `static` context, class variables and interface constants must be qualified with their type name.	The `super.fieldName` general form of the field access expression can be used to access a superclass field when the simple name of the field would be ambiguous because an interface constant with the same name was inherited.

The Java programming language is designed so that the `this` keyword is implicit in most of the contexts in which it is needed. However, `this` must be explicitly coded when referencing the current object in the following three contexts:

- As an argument expression in a method invocation expression or class instance creation expression
- As a return value in a `return` statement
- As the expression in a `synchronized` statement

This is what I characterize as the primary use of the `this` keyword. Examples of this use of the `this` keyword are as simple as `return this`. Used in these contexts the `this` keyword is sometimes referred to as the **`this` reference**.

The primary use of the `super` keyword is to invoke overridden superclass instance methods from within the body of the overriding subclass. Were it not for the `super` keyword, doing so would be impossible. Except for the `super.methodName` general form invoked from within the body of the overriding subclass, access to the overridden instance method in the superclass is impossible given a reference to an instance of the overriding subclass. Were that not so it would be possible to defeat the design of the subclass in which the overriding method is declared. For example, the following program does everything it can to invoke the overridden `print()` method in `T2` given only a reference to a `T3` class object:

```
class T1 {
    String print() { return "NEVER HAPPEN"; }
}

class T2 extends T1 {
    String print() { return "YES"; }
}

class T3 extends T2 {
   String print() { return "NO"; }
   void test() {
       System.out.println("print() = " + print());
       System.out.println("super.print() = " + super.print());
       System.out.println("((T2)this).print() = " + ((T2)this).print());
       System.out.println("((T1)this).print() = " + ((T1)this).print());
       System.out.println();
    }
}

public class Test {
  public static void main(String[] args) {
```

```
        T3 t3 = new T3();
        t3.test();
    /* Try casting the t3 variable */
        System.out.println("t3.print() = " + t3.print());
        System.out.println("((T2)t3).print() = " + ((T2)t3).print());
        System.out.println("((T1)t3).print() = " + ((T1)t3).print());
    /* Try converting t3 to a superclass type and then invoking
        the print() method */
        T2 t2 = t3;
        System.out.println("t2.print() = " + t2.print());
        T1 t1 = t3;
        System.out.println("t1.print() = " + t1.print());
      }
}
```

Executing this program prints

```
print() = NO
super.print() = YES
((T2)this).print() = NO
((T1)this).print() = NO

t3.print() = NO
((T2)t3).print() = NO
((T1)t3).print() = NO
t2.print() = NO
t1.print() = NO
```

The best way to remember this is that when a subclass overrides a superclass instance method, the `super.methodName` general form is "the only way back."

Note that, although the method invoked by the `super.methodName` general form is a member of the direct superclass, it may be declared in any of the superclasses in the class hierarchy. For example, if the `print()` method were removed from the `T2` class in the example above, a `super.print()` method invocation in the `T3` class would invoke the `print()` method declared in the `T1` class, and `NEVER HAPPEN` would be printed.

The `this` and `super` keywords are also used to access hidden or shadowed fields and hidden class methods. The following program is an example of accessing hidden fields:

```
class Superclass {
  static String msg = "hidden field";
}
public class Test extends Superclass {
  String msg = "shadowed field";
  public static void main(String[] args) {
      /* instantiate Test in order to invoke an instance method */
      new Test().print();
  }
  void print() {
```

```
        String msg = "local variable";
        System.out.println(msg);
        System.out.println(this.msg);
        System.out.println(super.msg);
    }
}
```

Executing this program prints

```
local variable
shadowed field
hidden field
```

Hiding is generally discouraged in the Java programming language. The same could be said of local variables or parameters shadowing fields except for one important exception. Constructor parameters usually shadow the name of the instance variable to which they are assigned. This programmer convention makes coding constructors much easier because you do not have to think of names for all of the constructor parameters. For example,

```
class Widget {
     int x;
     Widget(int x) {
         this.x = x;
     }
}
```

Note that if `this` were not used to qualify the instance variable `x`, `x = x` would assign the value of the constructor parameter to itself.

One special use of the `this` and `super` keywords is in explicit constructor invocations. Explicit constructor invocations invoke either another constructor in the same class (the `this` keyword) or a superclass constructor (the `super` keyword). They are always the first statements in a constructor. For example,

```
class Superclass {
    Superclass(String s) {
        System.out.println(s);
    }
}
public class Test extends Superclass {
    Test() {
        this("Test(String s)");
        System.out.println("Test()");
    }
    Test(String s) {
        super("Superclass(String s)");
        System.out.println(s);
    }
    public static void main(String[] args) {
```

```
        new Test();
    }
}
```

Executing this program prints

```
Superclass(String s)
Test(String s)
Test()
```

The other uses of the `this` and `super` keywords are described as *miscellaneous* in Table 3-2. They include qualifying members of the current class when doing so is unnecessary. For example,

```
public class Test {
    static int x;
    public static void main(String[] args) {
      /* instantiate Test in order to invoke an instance method */
      new Test().print();
    }
    void print() {
      System.out.println(this.x);
    }
}
```

The `x` field does not need to be qualified. The simple name of a field or method is implicitly qualified with the `this` keyword. For whatever reason, programmers sometimes choose to make that qualification explicit. To achieve the same effect in a `static` context a type name is used to qualify the simple name of a class variable or class method. For example,

```
public class Test {
    static int x;
    public static void main(String[] args) {
      System.out.println(Test.x);
    }
}
```

Again, the `x` field does not have to be qualified like this.

Finally, there is one very obscure use of the `super` keyword. The simple name of a field is ambiguous if inherited from both a superclass and superinterface. For example,

```
interface Superinterface { double x = 3.3; }
class Superclass { int x = 3; }
class Test extends Superclass implements Superinterface {
    public static void main(String[] args) {
      /* instantiate Test in order to invoke an instance method */
      new Test().print();
    }
    void print() {
       {System.out.println(x);}     //COMPILER ERROR
    }
}
```

Attempting to compile this program generates the following compiler error:

```
Test.java:9: reference to x is ambiguous, both variable x in Superclass and variable x in
Superinterface match
       {System.out.println(x);}     //COMPILER ERROR
                           ^
1 error
```

If the reference to `x` in `Subclass` is changed to `super.x`, the same code compiles and prints `3`.

3.5.1 Using `super` to Reference Members in Different Packages

A member must be accessible for the general form `super.fieldName` or `super.methodName` to compile. A superclass member declared `private` or that has default access is not accessible from subclasses declared in different packages. Therefore, such a member cannot be accessed using the general form `super.fieldName` or `super.methodName`. For example,

```
package com.javarules.examples;
public class Superclass  {
   int x;
   int getX() {return x;}
}
```

```
import com.javarules.examples.Superclass;
class Test extends Superclass {
    public static void main(String[] args) {
      /* instantiate Test in order to invoke an instance method */
      new Test().print();
    }
    void print() {
       System.out.println(super.x);           //COMPILER ERROR
       System.out.println(super.getX());      //COMPILER ERROR
    }
}
```

`Superclass` and `Test` are members of different packages. (`Test` is a member of the unnamed package.) Attempting to compile this program generates the following compiler errors:

```
Test.java:8: x is not public in com.javarules.examples.Superclass; cannot be accessed from
outside package
      System.out.println(super.x);           //COMPILER ERROR
                         ^
Test.java:9: getX() is not public in com.javarules.examples.Superclass; cannot be accessed
               from outside package
      System.out.println(super.getX());      //COMPILER ERROR
                         ^
```

```
2 errors
```

In other words, there is nothing special about the `super` keyword that would allow access to an otherwise inaccessible superclass member.

3.6 Multiple Current Instances (a.k.a. Levels)

The term *current instance* is synonymous with *current object*, which is another name for the `this` keyword. I prefer *current instance* in this context. As with the discussion of the five kinds of classes and interfaces in the last chapter, the discussion of multiple current instances in this chapter is a result of my decision not to have a separate chapter on inner classes.

This section is more or less a continuation of 2.12.2 Inner Class Hierarchies in that the idea of *levels* (as in *top-level*) in an inner class hierarchy is defined.

The members of an inner class cannot be accessed unless two or more objects are first created. For example,

```
public class Dog {
    public class Tail {
        public void wag () {…}
    }
}
```

To create a dog and then wag its tail requires three lines of code:

```
Dog dog = new Dog();
Dog.Tail tail = dog.new Tail();
tail.wag();
```

From an interface design perspective, the problem here is that most dogs come with their tails already attached. Notice that the `Tail` class must be instantiated using a reference to a *particular instance* of the `Dog` class. This is referred to as a *qualified* class instance creation expression, which is discussed in the following subsection. There are two levels of code involved here. One corresponds to the `Dog` class and the other to the `Tail` class. Both of these classes must be instantiated before you can wag the tail of a dog. The inner class hierarchy for this example is shown in Figure 3-3. The lines in an inheritance hierarchy connect subclasses to their direct superclass. In an inner class hierarchy lines are used to connect inner classes to their innermost enclosing class. In other words, the innermost enclosing class in an inner class hierarchy is just like a direct superclass in an inheritance hierarchy. There is only one innermost enclos-

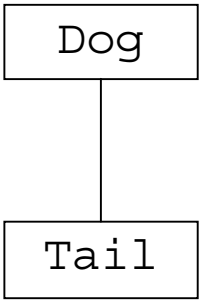


Figure 3-3: The `Dog` inner class hierarchy.

ing class, and there is a line connecting the inner class to the innermost enclosing class.

`Dog` is the top-level class. The second level is the `Tail` class, which has two current instances. A third level would have three current instances and so on. Therefore, level numbers correspond to the number of objects that must be created before a member at that level can be accessed. The following example is used throughout this section as well as in some of the subsections that follow:

```
public class Test  {

    interface Printable {
        void print();
    }

    class M1 {}

    class M2 {
        class M1 {
            Printable getPrint() {
                return new Printable() {
                    public void print() {
                        System.out.println(this.getClass().getName());
                        System.out.println(Test.this.getClass().getName());
                        System.out.println(Test.M2.this.getClass().getName());
                        System.out.println(Test.M2.M1.this.getClass().getName());
                    }
                };
            }
        }
        M1 m1 = new M1();
    }
    static M2.M1 m1 = new Test().new M2().new M1(); /* TOTALLY WEIRD CODE */

    public static void main(String[]  args) {
       /* instantiate Test so that an instance method can be invoked */
       new Test().print();
```

```
    }
    void print() {
       M2 m2 = new M2();
       Printable p = m2.m1.getPrint();
       p.print();
    }
}
```

The `M` in the type names stands for *member type*. Executing this program prints

```
Test$1
Test
Test$M2
Test$M2$M1
```

The inner class hierarchy for this example is shown in Figure 3-4.

Notice the following line of code:

```
static M2.M1 m1 = new Test().new M2().new M1();
```

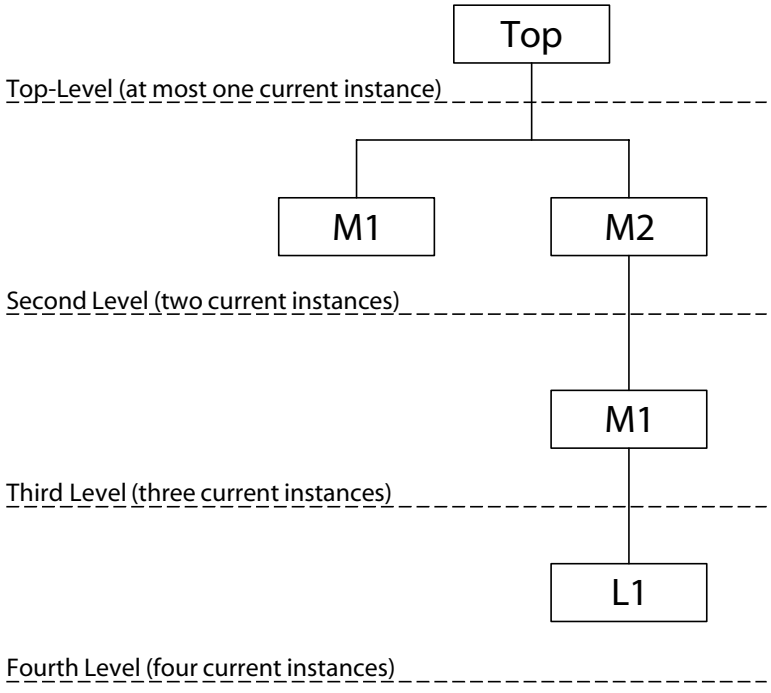


Figure 3-4: Multiple current instances (a.k.a. levels).

This is an example of using qualified class instance creation expressions to instantiate a deeply nested inner member class. It shows that three objects must be created before the anonymous class in the `print()` method of `Test.M2.M1` can be instantiated. Java programmers are normally shielded from the complexity of qualified class instance creation expressions for reasons that are explained in 3.6.3.1 Default Qualifying Instances.

Another indication of multiple current instances in this program are the following qualified and unqualified `this` keywords used in the body of the anonymous class:

```
this
Test.this
Test.M2.this
Test.M2.M1.this
```

They clearly show that there are four current instances in the body of the anonymous class. Qualified `this` keywords are discussed in 3.6.2 Qualifying the `this` Keyword.

The difference between top-level classes and inner classes is that there is only one current instance in the non-`static` code of a top-level class, whereas inner classes always have at least two current instances depending on how deeply nested the inner class is (except for orphaned local and anonymous classes declared in a `static` context). The reason that there is only one current instance is that there is no enclosing instance. That is what defines a top-level class. John Rose says as much in the following quotation from the *Inner Classes Specification*.

> 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.[5]

Elsewhere in the *Inner Classes Specification* he says,

> 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

5. Rose, *Inner Classes Specification* (Mountain View: Sun Microsystems, 1997), "Can a nested class be declared `final`, `private`, `protected`, or `static`?"

> top-level classes which are package members, and is also true of top- level classes which are `static` members of other top-level classes.[6]

This observation is fully supported by the concept of inner class hierarchies as presented in this book because at the top level of an inner class hierarchy there is at most one current instance. The difference is that the current instance in a top-level class cannot be referenced in `static` contexts. In other words, there are current instances everywhere except in `static` contexts, which can only be found in top-level classes. Current instances are what make a programming language such as Java object-oriented.

3.6.1 A Note About Deeply Nested Types

The definition of **deeply nested** is anything at level three or below. Deeply nested local and anonymous classes are common. Deeply nested member types, however, are unusual. Nevertheless, there are no restrictions whatsoever on the placement of inner classes. For example,

- An inner member class can enclose a local or anonymous class (normal inner class usage)
- A local class can enclose an inner member class or an anonymous class (unusual coding)
- An anonymous class can enclose an inner member class or a local class (highly questionable coding)

Nor is there a limit on the depth to which inner classes can be nested. For example, an anonymous class can enclose an inner member class, which in turn can enclose a local class that encloses yet another anonymous class, and so on.

Programmers should guard against this *Alice in Wonderland* world of unlimited possibilities. Local and anonymous classes especially should be kept simple. By that I mean nested types generally should not be declared in local and anonymous classes. John Rose specifically warns against this in the *Inner Classes Specification* when he warns:

> ...avoid deeply nested code.

6. Rose, "How do inner classes affect the idea of this in Java code?"

Nested types are typically no more complex than the relationships suggested in Figure 3-5. Half of the local and anonymous classes in this figure are deeply nested (level three and below). Typically only local and anonymous classes are deeply nested, but because they always use the default qualifying instance, you are not usually aware of the fact that they are deeply nested. Figure 3-5 is intended primarily to show that the other

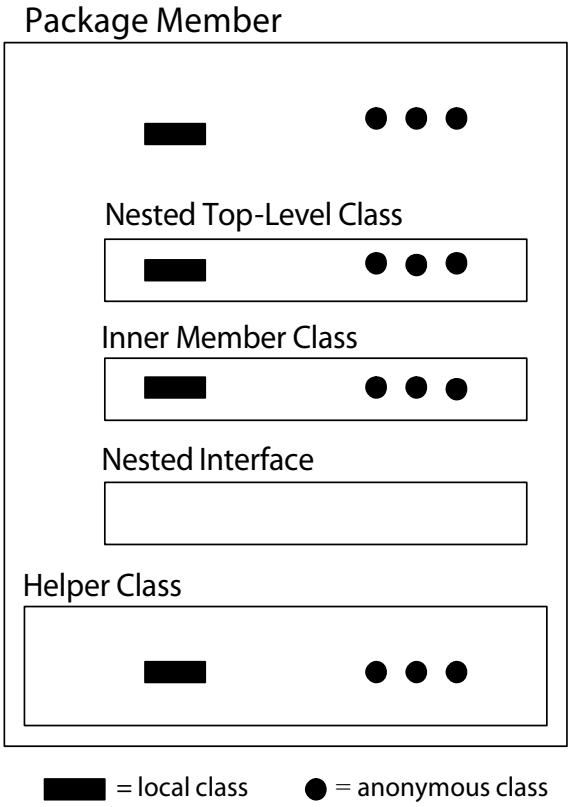


Figure 3-5: Typical uses of nested types.

nested types (nested top-level classes and interfaces and inner member classes) are generally not nested more than one level deep. In other words, most uses of nested types are conservative. This is not my opinion but based upon a very time-consuming and thorough study of nested types in the core API.

3.6.2 Qualifying the `this` Keyword

An unqualified `this` keyword always refers to the current instance of the class in which the `this` keyword appears. In an inner class hierarchy current instances are referred to as either the **innermost current instance** or an **enclosing instance**. To refer to enclosing instances the `this` keyword must be qualified by the name of the corresponding enclosing class. The following `nextElement()` method from an implementation of the `Enumerator` interface includes an example of a qualified `this` keyword:

```
public Object nextElement() {
    synchronized (Vector.this) {
           if (count < elementCount) {
               return elementData[count++];
           }
    }
    throw new java.util.NoSuchElementException("Enumerator");
}
```

An unqualified `this` keyword in the `nextElement()` method would be a reference to the `Enumerator` object. The `nextElement()` method uses `Vector.this` to lock the enclosing `Vector` object while an element is being accessed.

As can be seen in this example, the uses of qualified `this` keywords are the same as those of an unqualified `this` keyword. Another one of those uses is to reference a field shadowed by a local variable or parameter. Likewise, a qualified `this` can be used to reference a field shadowed by an unqualified `this`. For example,

```
class Test {
     class Superclass {
         String s = "inheritance takes precedence over scope";
     }
     public static void main (String[] args) {
           EnclosingClass enclose = new EnclosingClass();
           EnclosingClass.Subclass inner = enclose.new Subclass();
           inner.print();
     }
}

class EnclosingClass extends Test {
      String s = "this is the enclosing class";
      class Subclass extends Superclass {
           void print() {
               System.out.println(this.s);
               System.out.println(EnclosingClass.this.s);
           }
      }
}
```

Executing this program prints

```
inheritance takes precedence over scope
this is the enclosing class
```

If the simple name `s` is used in the `Subclass`, the following compiler error is generated:

```
Test.java:18: s is inherited from Test.Superclass and hides variable in outer class Enclosing-
Class.  An explicit 'this' qualifier must be used to select the desired instance.
                System.out.println(s);
                                   ^

1 error
```

In other words, the simple name `s` is ambiguous in the `InnerClass`.

3.6.2.1 Special Restriction on Nested Type Names

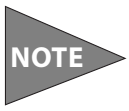
Nested types cannot have the same name as an enclosing type. For example,

```
class Top  {
    class Inner {
        class Inner { }
    }
}
```

Attempting to compile this class generates the following compiler error:

```
Top.java:3: Top.Inner is already defined in Top
        class Inner { }
        ^
1 error
```

The same error message would be generated if nested top-level classes were used instead of inner member classes. This restriction applies to all nested types. Using class names to qualify the `this` keyword works precisely because of this restriction on nested type names.



3.1

The remainder of this chapter discusses qualifying the `new` and `super` keywords. As mentioned in the last chapter, while qualifying the `super` keyword has been called "the pathological case," the truth is that the `new` keyword is not being explicitly qualified either. I will go out on a limb and say that 99.9 percent of the time you do not need to know this stuff. For that reason, I earnestly recommend that inexperienced programmers bypass these sections. They have been deliberately placed at the bottom of the chapter for this very reason. You can always come back and read these sections after having coded your first inner member class.

3.6.3 Qualifying the `new` Keyword

Class instance creation expressions are either qualified or unqualified. Usually they are unqualified. Unqualified class instance creation expressions begin with the `new` keyword. For example,

```
new Object()
```

The general form of a qualified class instance creation expression is as follows:

```
primaryExpression.new Identifier ( ArgumentList_opt ) ClassBody_opt
```

The primary expression must evaluate to an instance of the innermost enclosing class. In the context of a qualified class instance creation expression, an instance of the innermost enclosing class is referred to as the **qualifying instance** in this book. There is also a default qualifying instance, which is the subject of the next section.

Only inner member classes can be instantiated using a qualified class instance creation expression, which explains why they are so unusual. Top-level classes are only used in unqualified class instance creation expressions because they have no enclosing instance. Local and anonymous classes always use the default qualifying instance. An unqualified class instance creation expression for an inner member class also defaults to the current instance of the innermost enclosing class. As John Rose states,

> 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 [emphasis added].[7]

I do not want to make too much of this quotation, but "source code generators" is a long way from mainstream business applications. If you consider that quotation and the general tenor of the following one, also from the original *Inner Classes Specification*, it appears obvious to me that John Rose did not envision inner member classes being instantiated outside of the class in which they are declared.

> The inner class's name is not usable outside its scope, except perhaps in a qualified name.[8]

7. Rose, "How do inner classes affect the idea of this in Java code?"
8. Rose, "What are top-level classes and inner classes?"

Inner classes are declared close to where they are used. Qualified class instance creation expressions imply that an inner member class is being used far from where it is declared. This is an inherent contradiction. That is why qualified class instance creation expressions are so unusual.

The following example is repeated from 3.6 Multiple Current Instances (a.k.a. Levels):

```
public class Dog {
    public class Tail {
        public void wag () {…}
    }
}
```

The innermost enclosing class of `Tail` is `Dog`. Therefore, in a qualified class instance creation expression for the `Tail` class, the primary expression must evaluate to an instance of the `Dog` class. Here is the rest of the code from that example:

```
Dog dog = new Dog();
Dog.Tail tail = dog.new Tail();
tail.wag();
```

The variable `dog` is the primary expression. As can be seen, the variable `dog` evaluates to an instance of the `Dog` class.

Java programmers are usually shielded from the complexity of qualified class instance creation expressions for reasons that are explained in the following subsection.

3.6.3.1 Default Qualifying Instances

Unqualified class instance creation expressions for inner classes always default to the current instance of the innermost enclosing class. This is called the **default qualifying instance**. It is precisely this language feature that shields Java programmers from the inherent complexity of inner classes. For example,

```
class Top {
    public Top() {
        InnerMemberClass inner = new InnerMemberClass(); //implicit
        System.out.println(inner.getClass().getName());
    }
    class InnerMemberClass {}
}
```

Now consider the following program:

```
class Test {
    public static void main(String[] args) {
        new Top();
    }
}
```

Executing this program prints

```
Top$InnerMemberClass
```

The `Top` class could have just as easily been written using the `this` keyword to qualify the class instance create expression. For example,

```
class Top {
    public Top() {
        InnerMemberClass inner = this.new InnerMemberClass();  //explicit
        System.out.println(inner.getClass().getName());
    }
    class InnerMemberClass {}
}
```

The only difference in this implementation is that the default qualifying instance is made explicit by use of the `this` keyword.

The default qualifying instance of an inner member class is not always the unqualified `this`. For example,

```
public class Test  {
    class M1 { }
    class M2 {
        M1 m1 = new M1();  //implicit
    }
    public static void main(String[]  args) {
       /* instantiate Test so that an instance method can be invoked */
       new Test().print();
    }
    void print() {
       M2 m2 = new M2();
       System.out.println(m2.m1.getClass().getName());
    }
 }
```

Executing this program prints `"Test$M1"`. The compiler first determined that `Test` was the innermost enclosing class of `M1` and then used `Test.this` as the default qualifying instance. The following program produces exactly the same results:

```
public class Test  {
    class M1 { }
    class M2 {
        M1 m1 = Test.this.new M1();  //explicit
    }
    public static void main(String[]  args) {
       /* instantiate Test so that an instance method can be invoked */
       new Test().print();
    }
    void print() {
       M2 m2 = new M2();
       System.out.println(m2.m1.getClass().getName());
    }
 }
```

The ability of Java compilers to determine the innermost enclosing class and then use the appropriate current instance is what makes it possible to instantiate an inner member class anywhere in the inner class hierarchy in which it is declared without having to use a qualified class instance creation expression.

3.6.4 Qualifying the `super` Keyword

Qualifying the `super` keyword in an explicit constructor invocation is analogous to qualifying the `new` keyword in a class instance creation expression for an inner member class. The general form of a qualified superclass explicit constructor invocation is

```
primaryExpression.super (ArgumentList_opt);
```

A compiler error is generated if the `primaryExpression` does not evaluate to an instance of the innermost enclosing class of the direct superclass.

Inner member classes are sometimes extended by the subclasses that inherit them. Doing so is well within the limits of normal inner class usage. For example,

```
class Top {
     class Inner {
         String s;
         Inner(String s) {
             this.s = s;
         }
     }
}
```

```
class Test extends Top {
      class InnerSubclass extends Inner {
          InnerSubclass(String s) {
              super(s);
          }
          void print() {
              System.out.println(s);
          }
     }
     public static void main (String[] args) {
         /* instantiate Test so that an instance method can be invoked */
         new Test().print();
     }
     void print() {
         InnerSubclass innersub = new InnerSubclass("Hello World!");
         innersub.print();
     }
}
```

Executing this program prints "Hello World!". These are inner classes at their best. A Java programmer who extends an inner member class in this fashion is shielded from a ton of complexity.

So why would you want to qualify the `super` keyword? That is a good question. Although doing so is technically possible, a qualified `super` keyword is a rare bird indeed. Suppose `Test` did not extend `Top`. For example,

```
class Top {
     class Inner {
         String s;
         Inner(String s) {
             this.s = s;
         }
     }
}

class Test {
      class InnerSubclass extends Top.Inner {
          InnerSubclass(String s) {
              super(s);
          }
          void print() {
              System.out.println(s);
          }
      }
     public static void main (String[] args) {
         /* instantiate Test so that an instance method can be invoked */
         new Test().print();
     }
     void print() {
         InnerSubclass topinnersub = new InnerSubclass("Hello World!");
         topinnersub.print();
     }
}
```

Other than putting the classes in the same compilation unit and dropping the `extends` clause in the declaration of `Test`, nothing has changed. Attempting to compile this example, however, generates the following compiler error:

```
Test.java:13: no enclosing instance of type Top is in scope
              super(s);
              ^
1 error
```

The problem here is that there is no instance of `Top` being created that can be used in creating the `Inner` class. In the previous example there was. `Test` extended `Top` and was therefore an instance of the `Top` class.

If for some reason you ever find a need to code like this, what you need to do is pass the `Test` constructor an instance of the `Top` class that can be used to qualify the `super` keyword. For example,

```
class Top {
     class Inner {
         String s;
         Inner(String s) {
             this.s = s;
         }
     }
}

class Test {
      class InnerSubclass extends Top.Inner {
          InnerSubclass(Top top, String s) {
              top.super(s);
          }
          void print() {
              System.out.println(s);
          }
      }
     public static void main (String[] args) {
         /* instantiate Test so that an instance method can be invoked */
         new Test().print();
     }
     void print() {
         Top top = new Top();
         InnerSubclass topinnersub = new InnerSubclass(top, "Hello World!");
         topinnersub.print();
     }
}
```

The program now compiles and prints "`Hello World!`" as before.