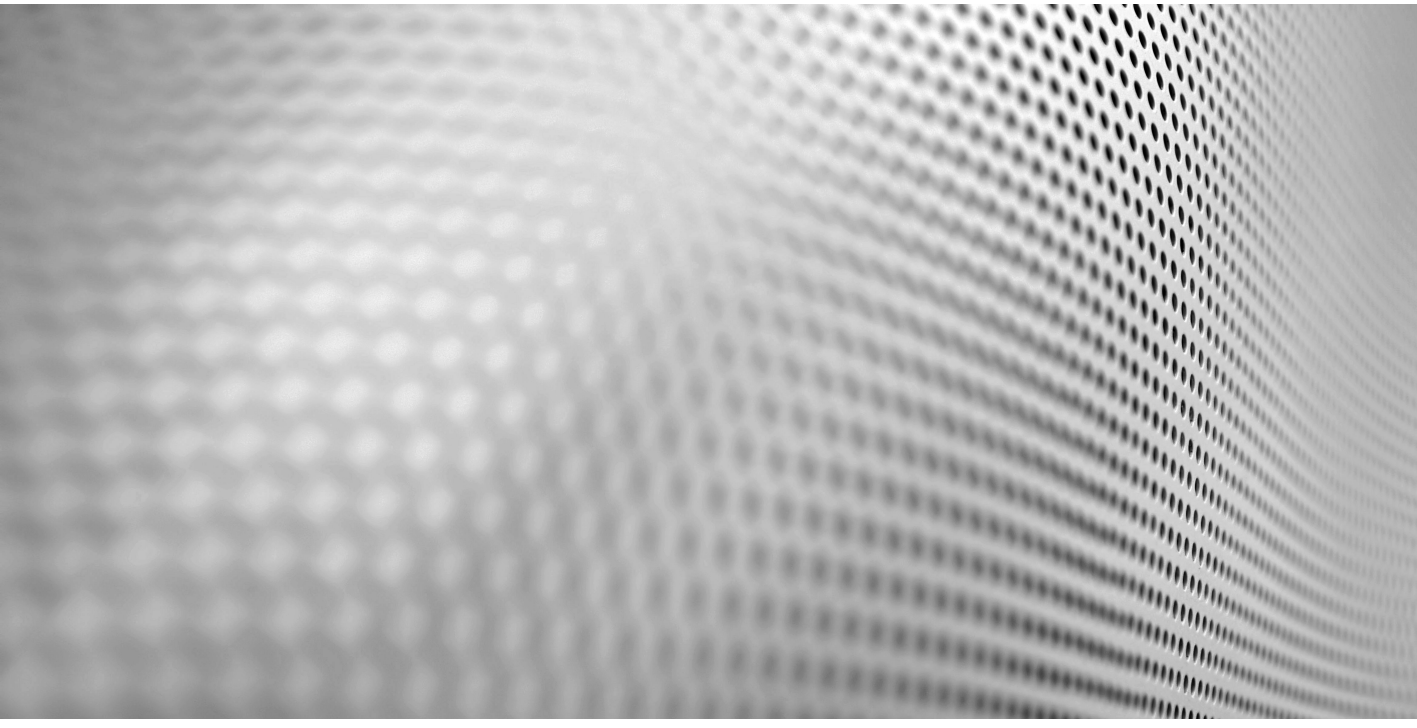

Supplement to Java: A Beginner's Guide, Seventh Edition

Two Key JDK 10 Features

This supplement to *Java: A Beginner's Guide, Seventh Edition* discusses two key features added by JDK 10. It is provided because a significant change has occurred to the way Java releases are scheduled. In the past, major Java releases were typically separated by two or more years. However, subsequent to the release of Java SE 9 (JDK 9), the time between major Java releases has been decreased. Beginning with Java SE 10 (JDK 10), a major release will occur on a strict, time-based schedule, with the time between major releases being just six months.

Each major release, now called a *feature release,* will include those features ready at the time of the release. This increased *release cadence* enables new features and enhancements to be available to Java programmers in a timely fashion. Furthermore, it allows Java to respond quickly to the demands of an ever-changing programming environment. Simply put, the faster release schedule promises to be a very positive development for Java programmers.

At the time of this writing, feature releases are scheduled for March and September of each year. As a result, JDK 10 was released in March 2018, which is six months after the release of JDK 9. The next release will be in September 2018. Again, every six months a new feature release will take place. Because of this more rapid release schedule, some releases will be specified as *long-term support* (LTS). This means that such a release will be supported for a certain length of time. Other feature releases are considered short term. At the time of this writing, both JDK 9 and JDK 10 are indicated as short term, and the September 2018 release is expected to be LTS. Consult the Java documentation for the latest information.

Java: A Beginner's Guide, Seventh Edition was updated for JDK 9. As you can probably guess, revising the book can involve a lengthy process. However, prior to the publication of the next edition of the book, of the many new features added by JDK 10, two will be of immediate interest to all Java programmers. For this reason, they are the subjects of this supplement. The first feature is called *local variable type inference*. Local variable type inference is especially important because it affects both the syntax and semantics of the Java language. The second new JDK 10 feature described here involves the changes to the JDK version number scheme. These changes support the time-based release schedule and alter the meaning of the version number elements. The version number changes also affect the **Runtime.Version** class (which encapsulates version information).

Beyond the two key features described here, JDK 10 includes other enhancements and changes, including several to the Java API. You will want to review the information and release notes provided by it in detail. Furthermore, you should examine each new six-month release carefully. There are a number of new features on the horizon. It is truly an exciting time to be a Java programmer!

Type Inference for Local Variables

Beginning with JDK 10, it is now possible to let the type of a local variable be inferred from the type of its initializer, rather than being explicitly specified. To support this new capability, the context-sensitive identifier **var** was added to Java as a reserved type name. Type inference can streamline code by eliminating the need to redundantly specify a variable's type when it can be inferred from its initializer. It can also simplify declarations in cases in which the type is difficult

to discern or cannot be denoted. (An example of a type that cannot be denoted is the type of an anonymous class.) Furthermore, local variable type inference has become a common part of the contemporary programming environment. Its inclusion in Java helps keep Java up-to-date with evolving trends in language design.

To use local variable type inference, the variable must be declared with **var** as the type name and it must include an initializer. For example, in the past you would declare a local **int** variable called **counter** that is initialized with the value 10 as shown here:

```
int counter = 10;
```

Using type inference, this declaration can now also be written like this:

```
var counter = 10;
```

In both cases, **counter** will be of type **int**. In the first case, its type is explicitly specified. In the second, its type is inferred as **int** because the initializer 10 is of type **int**.

As mentioned, **var** was added as a context-sensitive identifier. When it is used as the type name in the context of a local variable declaration, it tells the compiler to use type inference to determine the type of the variable being declared based on the type of the initializer. Thus, in a local variable declaration, **var** is a placeholder for the actual, inferred type. However, when used in most other places, **var** is simply a user-defined identifier with no special meaning. For example, the following declaration is still valid:

```
int var = 1;  // Here, var is simply a user-defined identifier.
```

In this case, the type is explicitly specified as **int** and **var** is the name of the variable being declared. Even though it is a context-sensitive identifier, there are a few places in which the use of **var** is illegal: It cannot be used as the name of a class, interface, enumeration, or annotation.

The following program puts the preceding discussion into action:

```
// A simple demonstration of local variable type inference.
class VarDemo {
  public static void main(String args[]) {

    // Use type inference to determine the type of the
    // variable named counter. In this case, int is inferred.
    var counter = 10;
    System.out.println("Value of counter: " + counter);

    // In the following context, var is not a predefined identifier.
    // It is simply a user-defined variable name.
    int var = 1;
    System.out.println("Value of var: " + var);
```

```
    // Interestingly, in the following sequence, var is used
    // as both the type of the declaration and as a variable name
    // in the initializer.
    var k = -var;
    System.out.println("Value of k: " + k);
  }
}
```

Here is the output:

```
Value of counter: 10
Value of var: 1
Value of k: -1
```

The preceding example uses **var** to declare only simple variables, but you can also use **var** to declare an array. For example:

```
var myArray = new int[10]; // This is valid.
```

Notice that neither **var** nor **myArray** has brackets. Instead, the type of **myArray** is inferred to be **int[ ]**. Furthermore, you *cannot* use brackets on the left side of a **var** declaration. Thus, both of these declarations are invalid:

```
var[] myArray = new int[10]; // Wrong
var myArray[] = new int[10]; // Wrong
```

In the first line, an attempt is made to bracket **var**. In the second, an attempt is made to bracket **myArray**. In both cases, the use of the brackets is wrong because the type is inferred from the type of the initializer.

It is important to emphasize that **var** can be used to declare a variable only when that variable is initialized. For example, the following statement is wrong:

```
var counter; // Wrong! Initializer required.
```

Also, remember that **var** can be used only to declare local variables. It cannot be used when declaring fields, parameters, or return types, for example.

Local Variable Type Inference with Reference Types

Although the preceding examples used a primitive type, there is no restriction in this regard. Local variable type inference with reference types, such as class types, is perfectly valid. Here is a simple example that declares a **String** variable called **myStr**:

```
var myStr = "This is a string";
```

Because a quoted string is used as an initializer, the type **String** is inferred.

As mentioned, one of the benefits of local variable type inference is its ability to streamline code, and it is with reference types where such streamlining is most apparent. For example, consider this declaration written the traditional way:

```
FileInputStream fin = new FileInputStream("test.txt");
```

With the use of **var**, it can now be written like this:

```
var fin = new FileInputStream("test.txt");
```

Here, **fin** is inferred to be of type **FileInputStream** because that is the type of its initializer. There is no need to explicitly repeat the type name. As a result, this declaration of **fin** is substantially shorter than writing it the traditional way. Thus, the use of **var** streamlines the declaration. In general, the streamlining attribute of local variable type inference helps lessen the tedium of entering long type names into your program.

Of course, you can also use local variable type inference with user-defined classes, as the following program illustrates:

```
// Local variable type inference with a user-defined class type.
class MyClass {
  private int i;

  MyClass(int k) { i = k;}

  int geti() { return i; }
  void seti(int k) { if(k >= 0) i = k; }
}

class VarDemo2 {
  public static void main(String args[]) {
    var mc = new MyClass(10); // Notice the use of var here.

    System.out.println("Value of i in mc is " + mc.geti());
    mc.seti(19);
    System.out.println("Value of i in mc is now " + mc.geti());
  }
}
```

Here, the type of **mc** will be **MyClass** because that is the type of the initializer. The output of the program is shown here:

```
Value of i in mc is 10
Value of i in mc is now 19
```

Local Variable Type Inference and Inheritance

It is important to not be confused about local variable type inference and inheritance hierarchies. As explained in the book, a superclass reference can refer to a derived class object, and this feature is part of Java's support for polymorphism. However, it is critical to remember that the inferred type of a variable is based on the declared type of its initializer. Therefore, if the initializer is of the superclass type, that will be the type of the variable. It does not matter if the actual object being referred to by the initializer is an instance of a derived class. For example, consider this program:

```
// When working with inheritance, the inferred type is the declared
// type of the initializer, which may not be the most derived type of
// the object being referred to by the initializer.

class MyClass {
  // ...
}

class FirstDerivedClass extends MyClass {
  int x;
  // ...
}

class SecondDerivedClass extends FirstDerivedClass {
  int y;
  // ...
}

class VarDemo3 {

  // Return some type of MyClass object.
  static MyClass getObj(int which) {
    switch(which) {
      case 0: return new MyClass();
      case 1: return new FirstDerivedClass();
      default: return new SecondDerivedClass();
   }
 }

  public static void main(String args[]) {

    // Even though getObj() returns different types of
    // objects within the MyClass inheritance hierarchy,
    // its declared return type is MyClass. As a result,
    // in all three cases shown here, the type of the
    // variables are inferred to be MyClass, even though
    // different derived types of objects are obtained.
```

```
    // Here, getObj() returns a MyClass object.
    var mc = getObj(0);

    // In this case, a FirstDerivedClass object is returned.
    var mc2 = getObj(1);

    // Here, a SecondDerivedClass object is returned.
    var mc3 = getObj(2);

    // Because the types of both mc2 and mc3 are inferred
    // as MyClass (because the return type of getObj() is
    // MyClass), neither mc2 or mc3 can access the fields
    // declared by FirstDerivedClass or SecondDerivedClass.
//    mc2.x = 10; // Wrong! MyClass does not have an x field.
//    mc3.y = 10; // Wrong! MyClass does not have an y field.
  }
}
```

In the program, a hierarchy is created that consists of three classes, at the top of which is **MyClass**. **FirstDerivedClass** is a subclass of **MyClass**, and **SecondDerivedClass** is a subclass of **FirstDerivedClass**. The program then uses type inference to create three variables, called **mc**, **mc2**, and **mc3** by calling **getObj( )**. The **getObj( )** method has a return type of **MyClass** (the superclass), but returns objects of type **MyClass**, **FirstDerivedClass**, or **SecondDerivedClass**, depending on the argument that it is passed. As the output shows, the inferred type is determined by the return type of **getObj( )** and not by the actual type of the object obtained.

Local Variable Type Inference and Generics

As explained in Chapter 13, one sort of type inference is already supported for generics through the use of **< >**, commonly called the diamond operator. However, you can also use local variable type inference with a generic class. For example, given this class:

```
class MyClass<T> {
  // ...
}
```

the following local variable declaration is valid:

```
var mc = new MyClass<Integer>();
```

In this case, the type of **mc** is inferred to be **MyClass<Integer>**. Also notice that the use of **var** results in a shorter declaration than would be the case otherwise. In general, generic type names can often be quite long and (in some cases) complicated. The use of **var** will substantially shorten such declarations.

One other point: You cannot use **var** as the name of a type parameter. For example, the following is invalid:

```
class MyClass<var> { // Wrong
```

Local Variable Type Inference in for loops and try Statements

Local variable type inference is not limited to the stand-alone declarations shown in the preceding examples. It can also be used with a **for** loop and in a **try**-with-resources statement. Let's look at each case.

Type inference can be used in a **for** loop when declaring and initializing the loop control variable in a traditional **for** loop, or when specifying the iteration variable in a for-each **for**. The following program shows an example of each case:

```
// Use type inference in a for loop.
class VarDemo4 {
  public static void main(String args[]) {

    // Use type inference with the loop control variable.
    System.out.print("Values of x: ");
    for(var x = 2.5; x < 100.0; x = x * 2)
      System.out.print(x + " ");

    System.out.println();

    // Use type inference with the iteration variable.
    int[] nums = { 1, 2, 3, 4, 5, 6};
    System.out.print("Values in nums array: ");
    for(var v : nums)
      System.out.print(v + " ");

    System.out.println();
  }
}
```

The output is shown here:

```
Values of x: 2.5 5.0 10.0 20.0 40.0 80.0
Values in nums array: 1 2 3 4 5 6
```

In this example, loop control variable **x** is inferred to be type **double** because that is the type of its initializer. Iteration variable **v** is inferred to be of type **int** because that is the element type of the array **nums**.

In a **try**-with-resources statement, the type of the resource can be inferred from its initializer. For example, the following statement is now valid:

```
try( var fin = new FileInputStream("test.txt")) {
  // ...
} catch(IOException exc) { // ... }
```

Here, **fin** is inferred to be of type **FileInputStream** because that is the type of its initializer.

Some var Restrictions

In addition to those mentioned in the preceding discussion, several other restrictions apply to the use of **var**. Only one variable can be declared at a time, a variable cannot use **null** as an initializer, and the variable being declared cannot be used by the initializer expression. Also, neither lambda expressions nor method references can be used as initializers. Although you can declare an array type using **var**, you cannot use **var** with an array initializer. For example, this is valid:

```
var myArray = new int[10]; // This is valid.
```

but this is not:

```
var myArray = { 1, 2, 3 }; // Wrong
```

Local variable type inference cannot be used to declare the exception type caught by a **catch** statement.

Updates to the JDK Version Number Scheme and Runtime.Version

With the release of JDK 10, the meaning of the JDK version number was changed to better accommodate the faster, time-based release schedule described at the start of this supplement. In the past, the JDK version number used the well-known *major.minor* approach. This mechanism did not, however, provide a good fit with the new release cadence. As a result, a different meaning was given to the elements of a version number. Beginning with JDK 10, the first four elements specify *counters,* which occur in the following order: feature release counter, interim release counter, update release counter, and patch release counter. Each number is separated by a period. However, trailing zeros, along with their preceding periods, are removed. Although additional elements may also be included, only the meaning of the first four are predefined.

The feature release counter specifies the number of the release. This counter is updated with each feature release, which (at the time of this writing) will occur every six months. To smooth the transition from the previous version scheme, the feature release counter will begin at 10. Thus, the feature release counter for JDK 10 is 10.

The interim release counter indicates the number of a release that occurs between feature releases. At the time of this writing, the value of the interim release counter will be zero because interim releases are not expected to be part of the increased release cadence. (It is defined for possible future use.) An interim release will not cause breaking changes to the

JDK feature set. The update release counter indicates the number of a release that addresses security, and possibly other problems. The patch release counter specifies a number of a release that addresses a serious flaw that must be fixed as soon as possible. With each new feature release, the interim, update, and patch counters are reset to zero.

It is useful to point out that the version number just described is a necessary component of the *version string,* but optional elements may also be included in the string. For example, a version string may include information for a pre-release version. Optional elements follow the version number in the version string.

Runtime.Version was added to the Java API by JDK 9. Although this class is not described within the book, the following discussion is included for the benefit of the interested reader. The purpose of **Runtime.Version** is to encapsulate version information pertaining to the Java runtime environment. Beginning with JDK 10, **Runtime.Version** was updated to include the following methods that support the new feature, interim, update, and patch counter values:

```
int feature( )
int interim( )
int update( )
int patch( )
```

Each returns an integer value that represents the indicated value. Here is a short program that demonstrates their use:

```
// Demonstrate Runtime.Version release counters.
class VerDemo {
  public static void main(String args[]) {
    Runtime.Version ver = Runtime.version();

    // Display individual counters.
    System.out.println("Feature release counter: " + ver.feature());
    System.out.println("Interim release counter: " + ver.interim());
    System.out.println("Update release counter: " + ver.update());
    System.out.println("Patch release counter: " + ver.patch());
  }
}
```

Runtime.Version also includes several other methods. Three are of special interest because they return optional version data, if present. They are **pre( )**, **build( )**, and **optional( )**, and they obtain pre-release information, the build number, and other optional data, respectively. If your program requires access to the Java version string, then you will want to explore **Runtime.Version** in detail.

As a result of the change to time-based releases, the following methods in **Runtime.Version** have been deprecated: **major( )**, **minor( )**, and **security( )**. Previously, these returned the major version number, the minor version number, and the security update number. These values have been superseded by the feature, interim, and update numbers, as just described.