JEP 447: Statements before super(...) (Preview)

Author	Archie Cobbs & Gavin Bierman
Owner	Archie Cobbs
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	22
Component	specification / language
Discussion	amber dash dev at openjdk dot java dot net
Relates to	JEP 482: Flexible Constructor Bodies (Second Preview)
Reviewed by	Brian Goetz
Endorsed by	Brian Goetz
Created	2023/01/20 17:33
Updated	2024/05/08 12:55
Issue	8300786

Summary

In constructors in the Java programming language, allow statements that do not reference the instance being created to appear before an explicit constructor invocation. This is a preview language feature.

Goals

- Give developers greater freedom to express the behavior of constructors, enabling the more natural placement of logic that currently must be factored into auxiliary static methods, auxiliary intermediate constructors, or constructor arguments.
- Preserve the existing guarantee that constructors run in top-down order during class instantiation, ensuring that code in a subclass constructor cannot interfere with superclass instantiation.
- Do not require any changes to the Java Virtual Machine. This Java language feature relies only on the current ability of the JVM to verify and execute code that appears before explicit constructor invocations within constructors.

Motivation

When one class extends another, the subclass inherits functionality from the superclass and can add functionality by declaring its own fields and methods. The initial values of fields declared in the subclass can depend upon the initial values of fields declared in the superclass, so it is critical to initialize fields of the superclass first, before fields of the subclass. For example, if class B extends class A then the fields of the unseen class Object must be initialized first, then the fields of class A, then the fields of class B.

Initializing fields in this order means that constructors must run from the top down: A constructor in a superclass must finish initializing the fields declared in that class before a constructor in a subclass is run. This is how the overall state of an object is initialized.

It is also critical to ensure that fields of a class are not accessed before they are initialized. Preventing access to uninitialized fields means that constructors must be constrained: The body of a constructor must not access fields declared in its own class or any superclass until the constructor in the superclass has finished.

To guarantee that constructors run from the top down, the Java language requires that in a constructor body, any explicit invocation of another constructor must appear as the first statement; if no explicit constructor invocation is given, then one is injected by the compiler.

To guarantee that constructors do not access uninitialized fields, the Java language requires that if an explicit constructor invocation is given, then none of its arguments can access the current object, `this`, in any way.

These requirements guarantee top-down behavior and no-access-before-initialization, but they are heavy-handed because they make several idioms that are used in ordinary methods difficult, or even impossible, to use in constructors. The following examples illustrate the issues.

Example: Validating superclass constructor arguments

Sometimes we need to validate an argument that is passed to a superclass constructor. We can validate the argument after the fact, but that means potentially doing unnecessary work:

```
public class PositiveBigInteger extends BigInteger {

    public PositiveBigInteger(long value) {
        super(value);               // Potentially unnecessary work
        if (value <= 0)
            throw new IllegalArgumentException("non-positive value");
    }

}
```

It would be better to declare a constructor that fails fast, by validating its arguments before it invokes the superclass constructor. Today we can only do that in-line, using an auxiliary `static` method:

```
public class PositiveBigInteger extends BigInteger {

    public PositiveBigInteger(long value) {
        super(verifyPositive(value));
    }

    private static long verifyPositive(long value) {
        if (value <= 0)
            throw new IllegalArgumentException("non-positive value");
        return value;
    }

}
```

This code would be more readable if we could include the validation logic directly in the constructor. What we would like to write is:

```
public class PositiveBigInteger extends BigInteger {

    public PositiveBigInteger(long value) {
        if (value <= 0)
            throw new IllegalArgumentException("non-positive value");
        super(value);
    }

}
```

Example: Preparing superclass constructor arguments

Sometimes we must perform non-trivial computation in order to prepare arguments for a superclass constructor, resorting, yet again, to auxiliary methods:

```
public class Sub extends Super {

    public Sub(Certificate certificate) {
        super(prepareByteArray(certificate));
    }

    // Auxiliary method
    private static byte[] prepareByteArray(Certificate certificate) {
        var publicKey = certificate.getPublicKey();
        if (publicKey == null)
            throw new IllegalArgumentException("null certificate");
        return switch (publicKey) {
            case RSAKey rsaKey -> ...
            case DSAPublicKey dsaKey -> ...
            ...
            default -> ...
        };
    }

}
```

The superclass constructor takes a `byte` array argument, but the subclass constructor takes a `Certificate` argument. To satisfy the restriction that the superclass constructor invocation must be the first statement in the subclass constructor, we declare the auxiliary method `prepareByteArray` to prepare the argument for that invocation.

This code would be more readable if we could embed the argument-preparation code directly in the constructor. What we would like to write is:

```
public Sub(Certificate certificate) {
        var publicKey = certificate.getPublicKey();
        if (publicKey == null)
            throw new IllegalArgumentException("null certificate");
        final byte[] byteArray = switch (publicKey) {
            case RSAKey rsaKey -> ...
            case DSAPublicKey dsaKey -> ...
            ...
            default -> ...
        };
        super(byteArray);
    }
```

Example: Sharing superclass constructor arguments

Sometimes we need to compute a value and share it between the arguments of a superclass constructor invocation. The requirement that the constructor invocation appear first means that the only way to achieve this sharing is via an intermediate auxiliary constructor:

```
public class Super {

    public Super(F f1, F f2) {
        ...
    }

}

public class Sub extends Super {

    // Auxiliary constructor
    private Sub(int i, F f) {
        super(f, f);                // f is shared here
        ... i ...
    }

    public Sub(int i) {
        this(i, new F());
    }

}
```

In the public Sub constructor we want to create a new instance of a class F and pass two references to that instance to the superclass constructor. We do that by declaring an auxiliary private constructor.

The code that we would like to write does the copying directly in the constructor, obviating the need for an auxiliary constructor:

```
public Sub(int i) {
        var f = new F();
        super(f, f);
        ... i ...
    }
```

Summary

In all of these examples, the constructor code that we would like to write contains statements before an explicit constructor invocation but does not access any fields via `this` before the superclass constructor has finished. Today these constructors are rejected by the compiler, even though all of them are safe: They cooperate in running constructors top down, and they do not access uninitialized fields.

If the Java language could guarantee top-down construction and no-access-before-initialization with more flexible rules then code would be easier to write and easier to maintain. Constructors could more naturally do argument validation, argument preparation, and argument sharing without doing that work via clumsy auxiliary methods or constructors. We need to move beyond the simplistic syntactic requirements enforced since Java 1.0, that is, "`super(..)` or `this(..)` must be the first statement", "no use of `this`", and so forth.

Description

We revise the grammar for constructor bodies (JLS §8.8.7) to read:

```
ConstructorBody:
    { [BlockStatements] }
    { [BlockStatements] ExplicitConstructorInvocation [BlockStatements] }
```

The block statements that appear before an explicit constructor invocation constitute the *prologue* of the constructor body. The statements in a constructor body with no explicit constructor invocation, and the statements following an explicit constructor invocation, constitute the *epilogue*.

Pre-construction contexts

As to semantics, the Java Language Specification classifies code that appears in the argument list of an explicit constructor invocation in a constructor body as being in a *static context* (JLS §8.1.3). This means that the arguments to such a constructor invocation are treated as if they were in a `static` method; in other words, as if no instance is available. The technical restrictions of a static context are stronger than necessary, however, and they prevent code that is useful and safe from appearing as constructor arguments.

Rather than revise the concept of a static context, we define a new, strictly weaker concept of a *pre-construction context* to cover both the arguments to an explicit constructor invocation and any statements that occur before it. Within a pre-construction context, the rules are similar to normal instance methods, except that the code may not access the instance under construction.

It turns out to be surprisingly tricky to determine what qualifies as accessing the instance under construction. Let us consider some examples.

To start with an easy case, any unqualified `this` expression is disallowed in a pre-construction context:

```
class A {

    int i;

    A() {
        this.i++;                   // Error
        this.hashCode();            // Error
        System.out.print(this);     // Error
        super();
    }

}
```

For similar reasons, any field access, method invocation, or method reference qualified by `super` is disallowed in a pre-construction context:

```
class D {
    int i;
}

class E extends D {

    E() {
        super.i++;                  // Error
        super();
    }

}
```

In trickier cases, an illegal access does not need to contain a `this` or `super` keyword:

```
class A {

    int i;

    A() {
        i++;                        // Error
        hashCode();                 // Error
        super();
    }

}
```

More confusingly, sometimes an expression involving `this` does not refer to the current instance but, rather, to the enclosing instance of an inner class:

```
class B {

    int b;

    class C {

        int c;

        C() {
            B.this.b++;             // Allowed - enclosing instance
            C.this.c++;             // Error - same instance
            super();
        }

    }

}
```

Unqualified method invocations are also complicated by the semantics of inner classes:

```
class Outer {

    void hello() {
        System.out.println("Hello");
    }

    class Inner {

        Inner() {
            hello();                // Allowed - enclosing instance method
            super();
        }

    }

}
```

The invocation `hello()` that appears in the pre-construction context of the `Inner` constructor is allowed because it refers to the enclosing instance of `Inner` (which, in this case, has the type `Outer`), not the instance of `Inner` that is being constructed (JLS §8.8.1).

In the previous example, the `Outer` enclosing instance was already constructed, and therefore accessible, whereas the `Inner` instance was under construction and therefore not accessible. The reverse situation is also possible:

```
class Outer {

    class Inner {
    }

    Outer() {
        new Inner();                // Error - 'this' is enclosing instance
        super();
    }

}
```

The expression `new Inner()` is illegal because it requires providing the `Inner` constructor with an enclosing instance of `Outer`, but the instance of `Outer` that would be provided is still under construction and therefore inaccessible.

Similarly, in a pre-construction context, class instance creation expressions that declare anonymous classes cannot have the newly created object as the implicit enclosing instance:

```
class X {

    class S {
    }

    X() {
        var tmp = new S() { };      // Error
        super();
    }

}
```

Here the anonymous class being declared is a subclass of S, which is an inner class of X. This means that the anonymous class would also have an enclosing instance of X, and hence the class instance creation expression would have the newly created object as the implicit enclosing instance. Again, since this occurs in the pre-construction context it results in a compile-time error. If the class S were declared `static`, or if it were an interface instead of a class, then it would have no enclosing instance and there would be no compile-time error.

This example, by contrast, is permitted:

```
class O {

    class S {
    }

    class U {

        U() {
            var tmp = new S() { };  // Allowed
            super();
        }

    }

}
```

Here the enclosing instance of the class instance creation expression is not the newly created `U` object but, rather, the lexically enclosing `O` instance.

A `return` statement may be used in the epilogue of a constructor body if it does not include an expression (i.e. `return;` is allowed, but `return e;` is not). It is a compile-time error if a `return` statement appears in the prologue of a constructor body.

Throwing an exception in a prologue of a constructor body is permitted. In fact, this will be typical in fail-fast scenarios.

Unlike in a static context, code in a pre-construction context may refer to the type of the instance under construction, as long as it does not access the instance itself:

```
class A<T> extends B {

    A() {
        super(this);                // Error - refers to 'this'
    }

    A(List<?> list) {
        super((T)list.get(0));      // Allowed - refers to 'T' but not 'this'
    }

}
```

Records

Record class constructors are already subject to more restrictions than normal constructors (JLS §8.10.4). In particular:

- Canonical record constructors may not contain any explicit constructor invocation, and
- Non-canonical record constructors must invoke an alternative constructor (a `this(...)` invocation), and may not invoke a superclass constructor (a `super(...)` invocation).

These restrictions remain, but otherwise record constructors will benefit from the changes described above, primarily in that non-canonical record constructors will be able to contain statements before explicit alternative constructor invocations.

Enums

Currently, `enum` class constructors may contain explicit alternative constructor invocations but not superclass constructor invocations. Enum classes will benefit from the changes described above, primarily in that their constructors will be able to contain statements before explicit alternative constructor invocations.

Testing

We will test the compiler changes with existing unit tests, unchanged except for those tests that verify changed behavior, plus new positive and negative test cases as appropriate.

We will compile all JDK classes using the previous and new versions of the compiler and verify that the resulting bytecode is identical.

No platform-specific testing should be required.

Risks and Assumptions

The changes we propose above are source- and behavior-compatible. They strictly expand the set of legal Java programs while preserving the meaning of all existing Java programs.

These changes, though modest in themselves, represent a significant change in the long-standing requirement that a constructor invocation, if present, must always appear as the first statement in a constructor body. This requirement is deeply embedded in code analyzers, style checkers, syntax highlighters, development environments, and other tools in the Java ecosystem. As with any language change, there may be a period of pain as tools are updated.

Dependencies

This Java language feature depends on the ability of the JVM to verify and execute arbitrary code that appears before constructor invocations in constructors so long as that code does not reference the instance under construction. Fortunately, the JVM already supports a more flexible treatment of constructor bodies:

- Multiple constructor invocations may appear in a constructor provided on any code path there is exactly one invocation;
- Arbitrary code may appear before constructor invocations so long as that code does not reference the instance under construction except to assign fields; and
- Explicit constructor invocations may not appear within a `try` block, i.e., within a bytecode exception range.

These more permissive rules still ensure top-down initialization:

- Superclass initialization always happens exactly once, either directly via a superclass constructor invocation or indirectly via an alternate constructor invocation; and
- Uninitialized instances are off-limits except for field assignments, which do not affect outcomes, until superclass initialization is complete.

In other words, we do not require any changes to the Java Virtual Machine Specification.

The current mismatch between the JVM and the language is an historical artifact. Originally the JVM was more restrictive, but this led to issues with the initialization of compiler-generated fields for new language features such as inner classes and captured free variables. As a result, the specification was relaxed to accommodate compiler-generated code, but this new flexibility never made its way back up to the language.