JEP 409: Sealed Classes

Owner	Gavin Bierman
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	17
Component	specification / language
Discussion	amber dash dev at openjdk dot java dot net
Relates to	JEP 360: Sealed Classes (Preview) JEP 397: Sealed Classes (Second Preview)
Reviewed by	Alex Buckley
Endorsed by	Brian Goetz
Created	2021/01/27 13:40
Updated	2022/03/19 20:20
Issue	8260514

Summary

Enhance the Java programming language with sealed classes and interfaces. Sealed classes and interfaces restrict which other classes or interfaces may extend or implement them.

History

Sealed Classes were proposed by JEP 360 and delivered in JDK 15 as a preview feature. They were proposed again, with refinements, by JEP 397 and delivered in JDK 16 as a preview feature. This JEP proposes to finalize Sealed Classes in JDK 17, with no changes from JDK 16.

Goals

- Allow the author of a class or interface to control which code is responsible for implementing it.
- Provide a more declarative way than access modifiers to restrict the use of a superclass.
- Support future directions in pattern matching by providing a foundation for the exhaustive analysis of patterns.

Non-Goals

- It is not a goal to provide new forms of access control such as "friends".
- It is not a goal to change `final` in any way.

Motivation

The object-oriented data model of inheritance hierarchies of classes and interfaces has proven to be highly effective in modeling the real-world data processed by modern applications. This expressiveness is an important aspect of the Java language.

There are, however, cases where such expressiveness can usefully be tamed. For example, Java supports *enum classes* to model the situation where a given class has only a fixed number of instances. In the following code, an enum class lists a fixed set of planets. They are the only values of the class, therefore you can switch over them exhaustively — without having to write a `default` clause:

```
enum Planet { MERCURY, VENUS, EARTH }

Planet p = ...
switch (p) {
  case MERCURY: ...
  case VENUS: ...
  case EARTH: ...
}
```

Using enum classes to model fixed sets of values is often helpful, but sometimes we want to model a fixed set of *kinds* of values. We can do this by using a class hierarchy not as a mechanism for code inheritance and reuse but, rather, as a way to list kinds of values. Building on our planetary example, we might model the kinds of values in the astronomical domain as follows:

```
interface Celestial { ... }
final class Planet implements Celestial { ... }
final class Star   implements Celestial { ... }
final class Comet  implements Celestial { ... }
```

This hierarchy does not, however, reflect the important domain knowledge that there are only three kinds of celestial objects in our model. In these situations, restricting the set of subclasses or subinterfaces can streamline the modeling.

Consider another example: In a graphics library, the author of a class Shape may intend that only particular classes can extend Shape, since much of the library's work involves handling each kind of shape in the appropriate way. The author is interested in the clarity of code that handles known subclasses of Shape, and not interested in writing code to defend against unknown subclasses of Shape. Allowing arbitrary classes to extend Shape, and thus inherit its code for reuse, is not a goal in this case. Unfortunately, Java assumes that code reuse is always a goal: If Shape can be extended at all, then it can be extended by any number of classes. It would be helpful to relax this assumption so that an author can declare a class hierarchy that is not open for extension by arbitrary classes. Code reuse would still be possible within such a closed class hierarchy, but not beyond.

Java developers are familiar with the idea of restricting the set of subclasses because it often crops up in API design. The language provides limited tools in this area: Either make a class `final`, so it has zero subclasses, or make the class or its constructor package-private, so it can only have subclasses in the same package. An example of a package-private superclass appears in the JDK:

```
package java.lang;

abstract class AbstractStringBuilder { ... }
public final class StringBuffer  extends AbstractStringBuilder { ... }
public final class StringBuilder extends AbstractStringBuilder { ... }
```

The package-private approach is useful when the goal is code reuse, such as having the subclasses of `AbstractStringBuilder` share its code for append. However, the approach is useless when the goal is modeling alternatives, since user code cannot access the key abstraction — the superclass — in order to `switch` over it. Allowing users to access the superclass without also allowing them to extend it cannot be specified without resorting to brittle tricks involving non-`public` constructors — which do not work for interfaces. In a graphics library that declares Shape and its subclasses, it would be unfortunate if only one package could access Shape.

In summary, it should be possible for a superclass to be widely *accessible* (since it represents an important abstraction for users) but not widely *extensible* (since its subclasses should be restricted to those known to the author). The author of such a superclass should be able to express that it is co-developed with a given set of subclasses, both to document intent for the reader and to allow enforcement by the Java compiler. At the same time, the superclass should not unduly constrain its subclasses by, e.g., forcing them to be `final` or preventing them from defining their own state.

Description

A *sealed* class or interface can be extended or implemented only by those classes and interfaces permitted to do so.

A class is sealed by applying the `sealed` modifier to its declaration. Then, after any `extends` and `implements` clauses, the `permits` clause specifies the classes that are permitted to extend the sealed class. For example, the following declaration of Shape specifies three permitted subclasses:

```
package com.example.geometry;

public abstract sealed class Shape
    permits Circle, Rectangle, Square { ... }
```

The classes specified by `permits` must be located near the superclass: either in the same module (if the superclass is in a named module) or in the same package (if the superclass is in the unnamed module). For example, in the following declaration of Shape its permitted subclasses are all located in different packages of the same named module:

```
package com.example.geometry;

public abstract sealed class Shape
    permits com.example.polar.Circle,
            com.example.quad.Rectangle,
            com.example.quad.simple.Square { ... }
```

When the permitted subclasses are small in size and number, it may be convenient to declare them in the same source file as the sealed class. When they are declared in this way, the `sealed` class may omit the `permits` clause and the Java compiler will infer the permitted subclasses from the declarations in the source file. (The subclasses may be auxiliary or nested classes.) For example, if the following code is found in `Root.java` then the sealed class `Root` is inferred to have three permitted subclasses:

```
abstract sealed class Root { ...
    final class A extends Root { ... }
    final class B extends Root { ... }
    final class C extends Root { ... }
}
```

Classes specified by `permits` must have a canonical name, otherwise a compile-time error is reported. This means that anonymous classes and local classes cannot be permitted subtypes of a sealed class.

A sealed class imposes three constraints on its permitted subclasses:

1. The sealed class and its permitted subclasses must belong to the same module, and, if declared in an unnamed module, to the same package.
2. Every permitted subclass must directly extend the sealed class.
3. Every permitted subclass must use a modifier to describe how it propagates the sealing initiated by its superclass:

- A permitted subclass may be declared `final` to prevent its part of the class hierarchy from being extended further. (Record classes are implicitly declared `final`.)
- A permitted subclass may be declared `sealed` to allow its part of the hierarchy to be extended further than envisaged by its sealed superclass, but in a restricted fashion.
- A permitted subclass may be declared `non-sealed` so that its part of the hierarchy reverts to being open for extension by unknown subclasses. A sealed class cannot prevent its permitted subclasses from doing this. (The modifier `non-sealed` is the first hyphenated keyword proposed for Java.)

As an example of the third constraint, `Circle` and `Square` may be `final` while `Rectangle` is `sealed` and we add a new subclass, `WeirdShape`, that is `non-sealed`:

```
package com.example.geometry;

public abstract sealed class Shape
    permits Circle, Rectangle, Square, WeirdShape { ... }

public final class Circle extends Shape { ... }

public sealed class Rectangle extends Shape
    permits TransparentRectangle, FilledRectangle { ... }
public final class TransparentRectangle extends Rectangle { ... }
public final class FilledRectangle extends Rectangle { ... }

public final class Square extends Shape { ... }

public non-sealed class WeirdShape extends Shape { ... }
```

Even though the `WeirdShape` is open to extension by unknown classes, all instances of those subclasses are also instances of `WeirdShape`. Therefore code written to test whether an instance of Shape is either a `Circle`, a `Rectangle`, a `Square`, or a `WeirdShape` remains exhaustive.

Exactly one of the modifiers `final`, `sealed`, and `non-sealed` must be used by each permitted subclass. It is not possible for a class to be both `sealed` (implying subclasses) and `final` (implying no subclasses), or both `non-sealed` (implying subclasses) and `final` (implying no subclasses), or both `sealed` (implying restricted subclasses) and `non-sealed` (implying unrestricted subclasses).

(The `final` modifier can be considered a special case of sealing, where extension/implementation is prohibited completely. That is, `final` is conceptually equivalent to `sealed` plus a `permits` clause which specifies nothing, though such a `permits` clause cannot be written.)

A class which is `sealed` or `non-sealed` may be `abstract`, and have `abstract` members. A `sealed` class may permit subclasses which are `abstract`, providing they are then `sealed` or `non-sealed`, rather than `final`.

It is a compile-time error if any class extends a `sealed` class but is not permitted to do so.

Class accessibility

Because `extends` and `permits` clauses make use of class names, a permitted subclass and its sealed superclass must be accessible to each other. However, permitted subclasses need not have the same accessibility as each other, or as the sealed class. In particular, a subclass may be less accessible than the sealed class. This means that, in a future release when pattern matching is supported by switches, some code will not be able to exhaustively `switch` over the subclasses unless a `default` clause (or other total pattern) is used. Java compilers will be encouraged to detect when `switch` is not as exhaustive as its original author imagined it would be, and customize the error message to recommend a `default` clause.

Sealed interfaces

As for classes, an interface can be sealed by applying the `sealed` modifier to the interface. After any `extends` clause to specify superinterfaces, the implementing classes and subinterfaces are specified with a `permits` clause. For example, the planetary example from above can be rewritten as follows:

```
sealed interface Celestial
    permits Planet, Star, Comet { ... }

final class Planet implements Celestial { ... }
final class Star   implements Celestial { ... }
final class Comet  implements Celestial { ... }
```

Here is another classic example of a class hierarchy where there is a known set of subclasses: modeling mathematical expressions.

```
package com.example.expression;

public sealed interface Expr
    permits ConstantExpr, PlusExpr, TimesExpr, NegExpr { ... }

public final class ConstantExpr implements Expr { ... }
public final class PlusExpr     implements Expr { ... }
```

```
public final class TimesExpr    implements Expr { ... }
public final class NegExpr      implements Expr { ... }
```

Sealing and record classes

Sealed classes work well with record classes. Record classes are implicitly `final`, so a sealed hierarchy of record classes is slightly more concise than the example above:

```
package com.example.expression;

public sealed interface Expr
    permits ConstantExpr, PlusExpr, TimesExpr, NegExpr { ... }

public record ConstantExpr(int i)       implements Expr { ... }
public record PlusExpr(Expr a, Expr b)  implements Expr { ... }
public record TimesExpr(Expr a, Expr b) implements Expr { ... }
public record NegExpr(Expr e)           implements Expr { ... }
```

The combination of sealed classes and record classes is sometimes referred to as *algebraic data types*: Record classes allow us to express *product types*, and sealed classes allow us to express *sum types*.

Sealed classes and conversions

A cast expression converts a value to a type. A type `instanceof` expression tests a value against a type. Java is extremely permissive about the types that are allowed in these kinds of expressions. For example:

```
interface I {}
class C {} // does not implement I

void test (C c) {
    if (c instanceof I)
        System.out.println("It's an I");
}
```

This program is legal even though it is currently not possible for a `C` object to implement the interface `I`. Of course, as the program evolves, it might be:

```
...
class B extends C implements I {}

test(new B());
// Prints "It's an I"
```

The type conversion rules capture a notion of *open extensibility*. The Java type system does not assume a closed world. Classes and interfaces can be extended at some future time, and casting conversions compile to runtime tests, so we can safely be flexible.

However, at the other end of the spectrum the conversion rules do address the case where a class can definitely not be extended, i.e., when it is a `final` class.

```
interface I {}
final class C {}

void test (C c) {
    if (c instanceof I)     // Compile-time error!
        System.out.println("It's an I");
}
```

The method `test` fails to compile, since the compiler knows that there can be no subclass of `C`, so since `C` does not implement `I` then it is never possible for a `C` value to implement `I`. This is a compile-time error.

What if `C` is not `final`, but `sealed`? Its direct subclasses are explicitly enumerated, and — by the definition of being `sealed` — in the same module, so we expect the compiler to look to see if it can spot a similar compile-time error. Consider the following code:

```
interface I {}
sealed class C permits D {}
final class D extends C {}

void test (C c) {
    if (c instanceof I)     // Compile-time error!
        System.out.println("It's an I");
}
```

Class `C` does not implement `I`, and is not `final`, so by the existing rules we might conclude that a conversion is possible. `C` is `sealed`, however, and there is one permitted direct subclass of `C`, namely `D`. By the definition of sealed types, `D` must be either `final`, `sealed`, or `non-sealed`. In this example, all the direct subclasses of `C` are `final` and do not implement `I`. This program should therefore be rejected, since there cannot be a subtype of `C` that implements `I`.

In contrast, consider a similar program where one of the direct subclasses of the sealed class is `non-sealed`:

```
interface I {}
sealed class C permits D, E {}
non-sealed class D extends C {}
final class E extends C {}
```

```
void test (C c) {
    if (c instanceof I)
        System.out.println("It's an I");
}
```

This is type-correct, since it is possible for a subtype of the `non-sealed` type `D` to implement `I`.

Consequently, supporting `sealed` classes leads to a change in the definition of narrowing reference conversion to navigate sealed hierarchies to determine at compile time which conversions are not possible.

Sealed classes in the JDK

An example of how sealed classes might be used in the JDK is in the `java.lang.constant` package that models descriptors for JVM entities:

```
package java.lang.constant;

public sealed interface ConstantDesc
    permits String, Integer, Float, Long, Double,
            ClassDesc, MethodTypeDesc, DynamicConstantDesc { ... }

// ClassDesc is designed for subclassing by JDK classes only
public sealed interface ClassDesc extends ConstantDesc
    permits PrimitiveClassDescImpl, ReferenceClassDescImpl { ... }
final class PrimitiveClassDescImpl implements ClassDesc { ... }
final class ReferenceClassDescImpl implements ClassDesc { ... }

// MethodTypeDesc is designed for subclassing by JDK classes only
public sealed interface MethodTypeDesc extends ConstantDesc
    permits MethodTypeDescImpl { ... }
final class MethodTypeDescImpl implements MethodTypeDesc { ... }

// DynamicConstantDesc is designed for subclassing by user code
public non-sealed abstract class DynamicConstantDesc implements ConstantDesc { ... }
```

Sealed classes and pattern matching

A significant benefit of sealed classes will be realized in JEP 406, which proposes to extend `switch` with pattern matching. Instead of inspecting an instance of a sealed class with `if`-`else` chains, user code will be able to use a `switch` enhanced with patterns. The use of sealed classes will allow the Java compiler to check that the patterns are exhaustive.

For example, consider this code using the `sealed` hierarchy declared earlier:

```
Shape rotate(Shape shape, double angle) {
        if (shape instanceof Circle) return shape;
        else if (shape instanceof Rectangle) return shape;
        else if (shape instanceof Square) return shape;
        else throw new IncompatibleClassChangeError();
}
```

The Java compiler cannot ensure that the `instanceof` tests cover all the permitted subclasses of Shape. The final `else` clause is actually unreachable, but this cannot be verified by the compiler. More importantly, no compile-time error message would be issued if the `instanceof Rectangle` test was omitted.

In contrast, with pattern matching for `switch` (JEP 406)the compiler can confirm that every permitted subclass of Shape is covered, so no `default` clause or other total pattern is needed. The compiler will, moreover, issue an error message if any of the three cases is missing:

```
Shape rotate(Shape shape, double angle) {
    return switch (shape) {   // pattern matching switch
        case Circle c    -> c;
        case Rectangle r -> shape.rotate(angle);
        case Square s    -> shape.rotate(angle);
        // no default needed!
    }
}
```

Java grammar

The grammar for class declarations is amended to the following:

```
NormalClassDeclaration:
  {ClassModifier} class TypeIdentifier [TypeParameters]
    [Superclass] [Superinterfaces] [PermittedSubclasses] ClassBody

ClassModifier:
  (one of)
  Annotation public protected private
  abstract static sealed final non-sealed strictfp

PermittedSubclasses:
  permits ClassTypeList

ClassTypeList:
  ClassType {, ClassType}
```

JVM support for sealed classes

The Java Virtual Machine recognizes `sealed` classes and interfaces at runtime, and prevents extension by unauthorized subclasses and subinterfaces.

Although `sealed` is a class modifier, there is no `ACC_SEALED` flag in the `ClassFile` structure. Instead, the `class` file of a sealed class has a `PermittedSubclasses` attribute which implicitly indicates the `sealed` modifier and explicitly specifies the permitted subclasses:

```
PermittedSubclasses_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_classes;
    u2 classes[number_of_classes];
}
```

The list of permitted subclasses is mandatory. Even when the permitted subclasses are inferred by the compiler, those inferred subclasses are explicitly included in the `PermittedSubclasses` attribute.

The `class` file of a permitted subclass carries no new attributes.

When the JVM attempts to define a class whose superclass or superinterface has a `PermittedSubclasses` attribute, the class being defined must be named by the attribute. Otherwise, an `IncompatibleClassChangeError` is thrown.

Reflection API

We add the following `public` methods to `java.lang.Class`:

- `Class<?>[] getPermittedSubclasses()`
- `boolean isSealed()`

The method `getPermittedSubclasses()` returns an array containing `java.lang.Class` objects representing the permitted subclasses of the class, if the class is sealed. It returns an empty array if the class is not sealed.

The method `isSealed` returns true if the given class or interface is sealed. (Compare with `isEnum`.)

Future Work

A common pattern, especially when writing APIs, is to define a public type as an interface and implement it with a single private class. With sealed classes this can be expressed more precisely, as a sealed public interface with a single permitted private implementation. Thus the type is widely accessible but the implementation is not, and cannot be extended in any way.

```
public sealed interface Foo permits MyFooImpl { }
private final class MyFooImpl implements Foo { }
```

A clumsiness of this approach is that implementation methods that accept `Foo` objects require explicit casts, for example:

```
void m(Foo f) {
    MyFooImpl mfi = (MyFooImpl) f;
    ...
}
```

The cast here seems unnecessary, since we know it should always succeed. Yet there is an implicit semantic assumption in the cast, which is that the class `MyFooImpl` is the only implementation of `Foo`. There is no way for the author to capture this intuition so that it can be checked at compile time. Should, in time, `Foo` permit an additional implementation, this cast would remain type-correct but may fail at runtime. In other words, the semantic assumption would be broken but the compiler cannot alert the developer of that fact.

With the precision of sealed hierarchies it may be worth providing developers with the means to express such semantic assumptions, and for the compiler to check them. This could be achieved by adding a new form of reference conversion for assignment contexts which allows the conversion of a sealed supertype to its only subtype, for example:

```
MyFooImpl mfi = f; // Allowed because the compiler sees that MyFooImpl
                   // is the only permitted subtype of Foo.
                   // (A synthetic cast would be added for safety.)
```

Alternatively, we could provide a new form of cast, for example:

```
MyFooImpl mfi = (total MyFooImpl) f;
```

In both cases, should the interface `Foo` be changed to permit another implementation, then both would cause compile-time errors upon recompilation.

Alternatives

Some languages have direct support for algebraic data types (ADTs), such as Haskell's `data` feature. It would be possible to express ADTs more directly and in a manner familiar to Java developers through a variant of the `enum` feature, where a sum of products could be defined in a single declaration. However, this would not support all the desired use cases, such as those where sums range over classes in more than one compilation unit, or sums that range over classes that are not products.

The `permits` clause allows a sealed class, such as the `Shape` class shown earlier, to be accessible-for-invocation by code in any module, but accessible-for-implementation by code in only the same module as the sealed class (or same package if in the unnamed module). This makes the type system more expressive than the access-control system. With access control alone, if Shape is accessible-for-invocation by code in any module (because its package is exported), then Shape is also accessible-for-implementation in any module; and if Shape is not accessible-for-implementation in any other module, then Shape is also not accessible-for-invocation in any other module.

Dependencies

Sealed classes do not depend on any other JEPs. As mentioned earlier, JEP 406 proposes to extend `switch` with pattern matching, and builds on sealed classes to improve the exhaustiveness checking of `switch`.