Inside the Language: Sealed Types

How Java is moving toward pattern matching, improved enums, and better switch statements

by Ben Evans

October 16, 2019

In this article, I present a programming language concept that is new to Java: sealed types. This feature is under active development at present and is expected to appear in a future version of Java. Some essential platform-level machinery for sealed types was introduced in Java 11—in the feature called nestmates. Java 12 also introduced switch expressions as a preview feature, which gives us a glimpse into how sealed types might be used in future versions of Java.

To follow along fully you’ll need to be fairly knowledgeable (or at least curious) about how programming languages are designed and implemented. To start our discussion of what sealed types are, let’s take a close look at Java’s enums and how they work under the hood.

Enums are a very well-known Java language feature. They enable you to model a finite set of alternatives that represent all possible values of a type, making enums effectively typesafe constants.

For example, consider a Pet enum:

```java
enum Pet {
    CAT,
    DOG
}
```

The platform implements this by having the Java compiler automatically generate a special form of class type. (Full disclosure: The runtime actually treats the library type `java.lang.Enum`—which all enum classes directly extend—in a slightly special way compared to other classes, but the details of this need not concern us here.)

Let’s decompile this enum and see what the compiler generates:

```bash
eris:src ben$ javap -c Pet.class
Compiled from "Pet.java"
final class Pet extends java.lang.Enum<Pet> {
    public static final Pet CAT;
    public static final Pet DOG;

    // Private constructor
}
```

Within the class file, all of the possible values of the enum are defined as `public static final` variables and the constructor is private, so additional instances cannot be constructed.
In effect, an enum is like a generalization of the singleton pattern, except that instead of being only one instance of the class, there are a finite number of instances. This is an extremely useful pattern, because it gives us the notion of *exhaustiveness*—given a not-null `Pet` object, you know for sure that it is either the `CAT` or the `DOG` instance.

However, suppose you want to model many different dogs and cats in Java as it exists today. A single instance will not suffice for each type of pet, so you're left to choose between two unpalatable alternatives.

First, you can choose to have a single implementing class, `Pet`, with a state field holding the actual type. This pattern works because the state field is of enum type and provides the bits that indicate which type is really meant for this specific object. This is obviously suboptimal, as it requires you to keep track of bits that are really the proper concern of the type system.

Alternatively, you can declare an abstract base type, `Pet`, and have concrete types `Cat` and `Dog` that subclass it. The issue here is that Java has always been designed as an open language that is extensible by default. Classes are compiled at one time, and subclasses can be compiled years (or even decades) later.

As of Java 11, the only class inheritance constructs permitted in the Java language are open inheritance (default) and no inheritance (`final`). This exposes a major weakness in the pattern of using an abstract base with concrete subtypes.

The issue is that although classes can declare a package private constructor, which effectively means “can be extended only by package mates,” nothing in the runtime prevents users from creating new classes in that package, so this is incomplete protection at best.

This means that if you define a `Pet` class, nothing will prevent a third party from creating a `Skunk` class that inherits from `Pet`. Worse still, this unwanted extension can happen years (or decades) after the `Pet` type was compiled, which is hugely undesirable.

The conclusion is that within current versions of Java, the abstract base approach is unsafe. This means developers are constrained and must use a field to hold the actual type of `Pet`.

Recent work on Project Amber within the overall OpenJDK project (which is where the standard version of Java is developed) aims to change this state of affairs, by allowing a new way to control inheritability in a more fine-grained way: the sealed type.

This capability is present in several other programming languages in various forms and has become fashionable in recent years (although it is actually quite an old idea).

In its Java incarnation, the concept that *sealing* expresses is that a class can be extended, but *only by a known list of subtypes* and no others.

Other languages may see the feature differently, but in Java it should be thought of as the ability to represent “almost final” classes.

Let's look at the current version of the new syntax for a simple example:

```java
class Pet {
    protected final String name;

    public void speak() {
    }
}

public final class Cat extends SealedPet {
    public Cat(String name) {
        super(name);
    }
}
```

In its Concrete Syntactic Form:}

```java
public abstract sealed class SealedPet permits Cat,
    protected final String name;
    public abstract void speak();
    public SealedPet(String name) {
        this.name = name;
    }
}

public final class Cat extends SealedPet {
    public Cat(String name) {
        super(name);
    }
    public void speak() {
    }
```
There are several things to notice here: First, `SealedPet` is now an abstract sealed class, which is not a keyword that has been permitted in Java until now. Second, the use of a second new keyword, permits, enables the developer to list the permissible subclasses of this sealed type. (If no list of permitted subtypes is provided, it is inferred from the subtypes in the same compilation unit.)

Finally, you have the bonus that because `Cat` and `Dog` are proper classes, they can have behaviors specific to their types, such as catching mice or chasing cars. This would not be straightforward if we were using a field to indicate the “real type” of the object, as all methods for all subtypes would have to be present in the base type or force us to use ugly downcasts. (Note that in this example, I have made the subclasses both `static` and `final`, but other combinations of modifiers are possible such as static final inner classes.)

If I now program with these types, I know that any `Pet` instance I encounter must be either a `Cat` or a `Dog`. What’s more, the compiler can use this information too. This means that library code can now safely assume that these are the only possibilities and this assumption cannot be violated by client code.

In terms of object-oriented programming theory, this represents a new kind of formal relationship: You can say that an object `o` is a `Cat` or a `Dog`. That is, the set of possible types for `o` is the union of `Cat` and `Dog`.

These types are therefore referred to as disjoint union types—also called tagged unions or sum types in various languages—but note that they are slightly different from C’s union.

For example, Scala programmers can implement a very similar idea with case classes and their own version of the sealed keyword.

Beyond the JVM, the Rust language also provides a notion of disjoint union types, although it refers to them with the `enum` keyword, which is potentially extremely confusing for Java programmers.

On the face of it, these types seem like a completely new concept in Java, but their deep similarity to enums should provide a good starting point for many Java programmers. In fact, there is one place where something similar to sum types already exists: the type of the exception parameter in a multicatch clause.

From the Java 11 language specification: “The declared type of an exception parameter that denotes its type as a union with alternatives \( D_1 | D_2 | \ldots | D_n \) is \( \text{lub}(D_1, D_2, \ldots, D_n) \)” (from JLS 11, Section 14.20). The lub in this quotation refers to for Least Upper Bound, that is, the closest common supertype of \( D_1, D_2, \ldots, D_n \)—which may be `java.lang.Object` or something more specific. For exceptions in multicatch the lub will be `Throwable` or possibly something more specific.
Sealed types and the new switches

The usefulness of sealed types does not end with the better modeling capabilities they provide, however. In fact, an interesting feature that arrived with JDK 12 (and was refined in JDK 13) is switch expressions, which provide a glimpse into how Java developers will be able to leverage sealed types in the future. Let’s look at this a bit more deeply.

Switch expressions are a major upgrade of Java’s ancient (and very C-like) `switch` keyword (which is a statement). This new feature allows a switch to act as an expression. This design aims to close a linguistic gap with more functionally oriented languages (such as Haskell, Scala, and Kotlin) that treat `switch` (or similar language structures) as expressions that return a value.

The final form is not expected to be very different from the version that arrived with Java 12, but because I’m writing before the Java 13 form is finalized, I’ll use the Java 12 form for illustration. Here is an example:

```java
import java.time.DayOfWeek;

public static boolean isWorkDay(DayOfWeek day){
    var today = switch(day) {
        case SATURDAY, SUNDAY -> false;
        case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> true;
    };

    // Do any further processing to e.g. account for
    // other factors like bank holidays
    return today;
}
```

By itself, this feature is useful, because it enables you to simplify a very common case of the use of `switch`, where it can now behave somewhat like a function, yielding an output value based on the input value. In fact, the rule for switch expressions is that every input value must be guaranteed to produce an output value.

For switch expressions that take an `int` (for example) then, you must include a `default` clause because it is not possible to list all possible cases. However, the sharp-eyed developer will notice that in the case of enums, the compiler can use the exhaustiveness of the constants. Predictably, if all the possible enum constants are present in a switch expression, it is not necessary to include a default statement, as in the example.

Switch expressions are also a stepping stone toward a major feature in a possible future version of Java.

The intent is to introduce a more-advanced construction called pattern matching in a later release. This proposed language-level feature should not be confused with regular expression pattern matching (that is, regex). Pattern matching, as meant here, enables the match expression (a generalization of the switch case labels) to extend beyond the simple constants we are accustomed to in Java.

For example, you could match not on the value but on the type of the object you are examining (known as a type pattern). This would help programmers deal with a situation where they have an object whose type is unknown at compile time and where disparate choices are possible but still valid. For example, when parsing JSON, you might encounter values that are strings, numbers, booleans, or other JSON objects. After parsing, a type pattern would allow you to create specialized handling for the case of `String`, `Double`, or `Boolean` by matching on the type of the object representing the value.

This type of dynamic flexibility is something Java frameworks often bring to the language—and to have it supported directly at the language level would be a huge step forward.
In the case of sealed types, the object-oriented idea that we can now say that this object is of type X or Y or Z but no other types is enormously powerful. The exhaustiveness we saw with enums in switch expressions can now occur at the type level too. Sealed types provide a parallel to Java’s existing enum language feature—but whereas enums are about a “finite number of possible instances,” sealed types are about “a finite number of possible types.”

The combination of match expressions, sealed types, and some other JVM technologies under development provides a further extension of Java’s support for both functional and object-oriented styles of programming.

Developers who want to experiment with sealed types right away will need to build their own OpenJDK binary from source, because early-access binaries are not available yet. Please remember that until a finalized version of Java is delivered that contains a specific language feature, programmers should not rely on it. When talking about future and possible features, as I have in this article, it should always be understood that the possible future discussed is for exploration purposes only. It does not represent a commitment on behalf of anyone (including Oracle) that a final version of Java supporting these features will actually ship.

---

**Ben Evans**

Ben Evans (@kittylyst) is a Java Champion, a tech fellow and founder at jClarity, an organizer for the London Java Community (LJC), and a member of the Java SE/EE Executive Committee. He has written four books on programming, including the recent *Optimizing Java* (O’Reilly).