JEP 395: Records

Owner	Gavin Bierman
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	16
Component	specification / language
Discussion	amber dash dev at openjdk dot java dot net
Relates to	JEP 359: Records (Preview)
	JEP 384: Records (Second Preview)
Reviewed by	Alex Buckley, Brian Goetz
Endorsed by	Brian Goetz
Created	2020/06/08 16:07
Updated	2024/02/03 01:28
Issue	8246771

Summary

Enhance the Java programming language with records, which are classes that act as transparent carriers for immutable data. Records can be thought of as *nominal tuples*.

History

Records were proposed by JEP 359 and delivered in JDK 14 as a preview feature.

In response to feedback, the design was refined by JEP 384 and delivered in JDK 15 as a preview feature for a second time. The refinements for the second preview were as follows:

- In the first preview, canonical constructors were required to be `public`. In the second preview, if the canonical constructor is implicitly declared then its access modifier is the same as the record class; if the canonical constructor is explicitly declared then its access modifier must provide at least as much access as the record class.
- The meaning of the `@Override` annotation was extended to include the case where the annotated method is an explicitly declared accessor method for a record component.
- To enforce the intended use of compact constructors, it became a compile-time error to assign to any of the instance fields in the constructor body.
- The ability to declare local record classes, local enum classes, and local interfaces was introduced.

This JEP proposes to finalize the feature in JDK 16, with the following refinement:

- Relax the longstanding restriction whereby an inner class cannot declare a member that is explicitly or implicitly static. This will become legal and, in particular, will allow an inner class to declare a member that is a record class.

Additional refinements may be incorporated based on further feedback.

Goals

- Devise an object-oriented construct that expresses a simple aggregation of values.
- Help developers to focus on modeling immutable data rather than extensible behavior.
- Automatically implement data-driven methods such as `equals` and accessors.
- Preserve long-standing Java principles such as nominal typing and migration compatibility.

Non-Goals

- While records do offer improved concision when declaring data carrier classes, it is not a goal to declare a "war on boilerplate". In particular, it is not a goal to address the problems of mutable classes which use the JavaBeans naming conventions.
- It is not a goal to add features such as properties or annotation-driven code generation, which are often proposed to streamline the declaration of classes for "Plain Old Java Objects".

Motivation

It is a common complaint that "Java is too verbose" or has "too much ceremony". Some of the worst offenders are classes that are nothing more than immutable *data carriers* for a handful of values. Properly writing such a data-carrier class involves a lot of low-value, repetitive, error-prone code: constructors, accessors, `equals`, `hashCode`, `toString`, etc. For example, a class to carry x and y coordinates inevitably ends up like this:

```
class Point {
    private final int x;
    private final int y;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    int x() { return x; }
    int y() { return y; }

    public boolean equals(Object o) {
        if (!(o instanceof Point)) return false;
        Point other = (Point) o;
        return other.x == x && other.y == y;
    }

    public int hashCode() {
        return Objects.hash(x, y);
    }

    public String toString() {
        return String.format("Point[x=%d, y=%d]", x, y);
    }
}
```

Developers are sometimes tempted to cut corners by omitting methods such as `equals`, leading to surprising behavior or poor debuggability, or by pressing an alternate but not entirely appropriate class into service because it has the "right shape" and they don't want to declare yet another class.

IDEs help us to *write* most of the code in a data-carrier class, but don't do anything to help the *reader* distill the design intent of "I'm a data carrier for x and y" from the dozens of lines of boilerplate. Writing Java code that models a handful of values should be easier to write, to read, and to verify as correct.

While it is superficially tempting to treat records as primarily being about boilerplate reduction, we instead choose a more semantic goal: *modeling data as data*. (If the semantics are right, the boilerplate will take care of itself.) It should be easy and concise to declare data-carrier classes that *by default* make their data immutable and provide idiomatic implementations of methods that produce and consume the data.

Description

Record classes are a new kind of class in the Java language. Record classes help to model plain data aggregates with less ceremony than normal classes.

The declaration of a record class primarily consists of a declaration of its *state*; the record class then commits to an API that matches that state. This means that record classes give up a freedom that classes usually enjoy — the ability to decouple a class's API from its internal representation — but, in return, record class declarations become significantly more concise.

More precisely, a record class declaration consists of a name, optional type parameters, a header, and a body. The header lists the *components* of the record class, which are the variables that make up its state. (This list of components is sometimes referred to as the *state description*.) For example:

```
record Point(int x, int y) { }
```

Because record classes make the semantic claim of being transparent carriers for their data, a record class acquires many standard members automatically:

- For each component in the header, two members: a `public` accessor method with the same name and return type as the component, and a `private final` field with the same type as the component;
- A *canonical constructor* whose signature is the same as the header, and which assigns each private field to the corresponding argument from a `new` expression which instantiates the record;
- `equals` and `hashCode` methods which ensure that two record values are equal if they are of the same type and contain equal component values; and
- A `toString` method that returns a string representation of all the record components, along with their names.

In other words, the header of a record class describes its state, i.e., the types and names of its components, and the API is derived mechanically and completely from that state description. The API includes protocols for construction, member access, equality, and display. (We expect a future version to support deconstruction patterns to allow powerful pattern matching.)

Constructors for record classes

The rules for constructors in a record class are different than in a normal class. A normal class without any constructor declarations is automatically given a *default constructor*. In contrast, a record class without any constructor declarations is automatically given a *canonical constructor* that assigns all the private fields to the corresponding arguments of the `new` expression which instantiated the record. For example, the record declared earlier — `record Point(int x, int y) { }` — is compiled as if it were:

```
record Point(int x, int y) {
    // Implicitly declared fields
    private final int x;
    private final int y;

    // Other implicit declarations elided ...

    // Implicitly declared canonical constructor
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

The canonical constructor may be declared explicitly with a list of formal parameters which match the record header, as shown above. It may also be declared more compactly, by eliding the list of formal parameters. In such a *compact canonical constructor* the parameters are declared implicitly, and the private fields corresponding to record components cannot be assigned in the body but are automatically assigned to the corresponding formal parameter (`this.x = x;`) at the end of the constructor. The compact form helps developers focus on validating and normalizing parameters without the tedious work of assigning parameters to fields.

For example, here is a compact canonical constructor that validates its implicit formal parameters:

```
record Range(int lo, int hi) {
    Range {
        if (lo > hi)  // referring here to the implicit constructor parameters
            throw new IllegalArgumentException(String.format("(%d,%d)", lo, hi));
    }
}
```

Here is a compact canonical constructor that normalizes its formal parameters:

```
record Rational(int num, int denom) {
    Rational {
        int gcd = gcd(num, denom);
        num /= gcd;
        denom /= gcd;
    }
}
```

This declaration is equivalent to the conventional constructor form:

```
record Rational(int num, int denom) {
    Rational(int num, int demon) {
        // Normalization
        int gcd = gcd(num, denom);
        num /= gcd;
        denom /= gcd;
        // Initialization
        this.num = num;
        this.denom = denom;
    }
}
```

Record classes with implicitly declared constructors and methods satisfy important, and intuitive, semantic properties. For example, consider a record class `R` declared as follows:

```
record R(T1 c1, ..., Tn cn){ }
```

If an instance `r1` of `R` is copied in the following way:

```
R r2 = new R(r1.c1(), r1.c2(), ..., r1.cn());
```

then, assuming `r1` is not the null reference, it is *always* the case that the expression `r1.equals(r2)` will evaluate to `true`. Explicitly declared accessor and `equals` methods should respect this invariant. However, it is not generally possible for a compiler to check that explicitly declared methods respect this invariant.

As an example, the following declaration of a record class should be considered bad style because its accessor methods "silently" adjust the state of a record instance, and the invariant above is not satisfied:

```
record SmallPoint(int x, int y) {
  public int x() { return this.x < 100 ? this.x : 100; }
  public int y() { return this.y < 100 ? this.y : 100; }
}
```

In addition, for all record classes the implicitly declared `equals` method is implemented so that it is reflexive and that it behaves consistently with `hashCode` for record classes that have floating point components. Again, explicitly declared `equals` and `hashCode` methods should behave similarly.

Rules for record classes

There are numerous restrictions on the declaration of a record class in comparison to a normal class:

- A record class declaration does not have an `extends` clause. The superclass of a record class is always `java.lang.Record`, similar to how the superclass of an enum class is always `java.lang.Enum`. Even though a normal class can explicitly extend its implicit superclass `Object`, a record cannot explicitly extend any class, even its implicit superclass `Record`.
- A record class is implicitly `final`, and cannot be `abstract`. These restrictions emphasize that the API of a record class is defined solely by its state description, and cannot be enhanced later by another class.
- The fields derived from the record components are `final`. This restriction embodies an *immutable by default* policy that is widely applicable for data-carrier classes.
- A record class cannot explicitly declare instance fields, and cannot contain instance initializers. These restrictions ensure that the record header alone defines the state of a record value.
- Any explicit declarations of a member that would otherwise be automatically derived must match the type of the automatically derived member exactly, disregarding any annotations on the explicit declaration. Any explicit implementation of accessors or the `equals` or `hashCode` methods should be careful to preserve the semantic invariants of the record class.
- A record class cannot declare `native` methods. If a record class could declare a `native` method then the behavior of the record class would by definition depend on external state rather than the record class's explicit state. No class with native methods is likely to be a good candidate for migration to a record.

Beyond the restrictions above, a record class behaves like a normal class:

- Instances of record classes are created using a `new` expression.
- A record class can be declared top level or nested, and can be generic.
- A record class can declare `static` methods, fields, and initializers.
- A record class can declare instance methods.
- A record class can implement interfaces. A record class cannot specify a superclass since that would mean inherited state, beyond the state described in the header. A record class can, however, freely specify superinterfaces and declare instance methods to implement them. Just as for classes, an interface can usefully characterize the behavior of many records. The behavior may be domain-independent (e.g., `Comparable`) or domain-specific, in which case records can be part of a *sealed* hierarchy which captures the domain (see below).
- A record class can declare nested types, including nested record classes. If a record class is itself nested, then it is implicitly static; this avoids an immediately enclosing instance which would silently add state to the record class.
- A record class, and the components in its header, may be decorated with annotations. Any annotations on the record components are propagated to the automatically derived fields, methods, and constructor parameters, according to the set of applicable targets for the annotation. Type annotations on the types of record components are also propagated to the corresponding type uses in the automatically derived members.
- Instances of record classes can be serialized and deserialized. However, the process cannot be customized by providing `writeObject`, `readObject`, `readObjectNoData`, `writeExternal`, or `readExternal` methods. The components of a record class govern serialization, while the canonical constructor of a record class governs deserialization.

Local record classes

A program that produces and consumes instances of a record class is likely to deal with many intermediate values that are themselves simple groups of variables. It will often be convenient to declare record classes to model those intermediate values. One option is to declare "helper" record classes that are static and nested, much as many programs declare helper classes today. A more convenient option would be to declare a record *inside a method*, close to the code which manipulates the variables. Accordingly we define *local record classes*, akin to the existing construct of local classes.

In the following example, the aggregation of a merchant and a monthly sales figure is modeled with a local record class, `MerchantSales`. Using this record class improves the readability of the stream operations which follow:

```
List<Merchant> findTopMerchants(List<Merchant> merchants, int month) {
    // Local record
    record MerchantSales(Merchant merchant, double sales) {}

    return merchants.stream()
        .map(merchant -> new MerchantSales(merchant, computeSales(merchant, month)))
        .sorted((m1, m2) -> Double.compare(m2.sales(), m1.sales()))
        .map(MerchantSales::merchant)
        .collect(toList());
}
```

Local record classes are a particular case of nested record classes. Like nested record classes, local record classes are *implicitly static*. This means that their own methods cannot access any variables of the enclosing method; in turn, this avoids capturing an immediately enclosing instance which would silently add state to the record class. The fact that local record classes are implicitly static is in contrast to local classes, which are not implicitly static. In fact, local classes are never static — implicitly or explicitly — and can always access variables in the enclosing method.

Local enum classes and local interfaces

The addition of local record classes is an opportunity to add other kinds of implicitly-static local declarations.

Nested enum classes and nested interfaces are already implicitly static, so for consistency we define local enum classes and local interfaces, which are also implicitly static.

Static members of inner classes

It is currently specified to be a compile-time error if an inner class declares a member that is explicitly or implicitly static, unless the member is a constant variable. This means that, for example, an inner class cannot declare a record class member, since nested record classes are implicitly static.

We relax this restriction in order to allow an inner class to declare members that are either explicitly or implicitly static. In particular, this allows an inner class to declare a static member that is a record class.

Annotations on record components

Record components have multiple roles in record declarations. A record component is a first-class concept, but each component also corresponds to a field of the same name and type, an accessor method of the same name and return type, and a formal parameter of the canonical constructor of the same name and type.

This raises the question: When a component is annotated, what actually is being annotated? The answer is, "all of the elements to which this particular annotation is applicable." This enables classes that use annotations on their fields, constructor parameters, or accessor methods to be migrated to records without having to redundantly declare these members. For example, a class such as the following

```
public final class Card {
    private final @MyAnno Rank rank;
    private final @MyAnno Suit suit;
    @MyAnno Rank rank() { return this.rank; }
    @MyAnno Suit suit() { return this.suit; }
    ...
}
```

can be migrated to the equivalent, and considerably more readable, record declaration:

```
public record Card(@MyAnno Rank rank, @MyAnno Suit suit) { ... }
```

The applicability of an annotation is declared using a `@Target` meta-annotation. Consider the following:

```
@Target(ElementType.FIELD)
    public @interface I1 {...}
```

This declares the annotation `@I1` that it is applicable to field declarations. We can declare that an annotation is applicable to more than one declaration; for example:

```
@Target({ElementType.FIELD, ElementType.METHOD})
    public @interface I2 {...}
```

This declares an annotation `@I2` that it is applicable to both field declarations and method declarations.

Returning to annotations on a record component, these annotations appear at the corresponding program points where they are applicable. In other words, the propagation is under the control of the developer using the `@Target` meta-annotation. The propagation rules are systematic and intuitive, and all that apply are followed:

- If an annotation on a record component is applicable to a field declaration, then the annotation appears on the corresponding private field.
- If an annotation on a record component is applicable to a method declaration, then the annotation appears on the corresponding accessor method.
- If an annotation on a record component is applicable to a formal parameter, then the annotation appears on the corresponding formal parameter of the canonical constructor if one is not declared explicitly, or else to the corresponding formal parameter of the compact constructor if one is declared explicitly.
- If an annotation on a record component is applicable to a type, the annotation will be propagated to all of the following:
  - the type of the corresponding field
  - the return type of the corresponding accessor method
  - the type of the corresponding formal parameter of the canonical constructor
  - the type of the record component (which is accessible at runtime via reflection)

If a public accessor method or (non-compact) canonical constructor is declared explicitly, then it only has the annotations which appear on it directly; nothing is propagated from the corresponding record component to these members.

A declaration annotation on a record component will not be amongst those associated with the record component at run time via the reflection API unless the annotation is meta-annotated with `@Target(RECORD_COMPONENT)`.

Compatibility and migration

The abstract class `java.lang.Record` is the common superclass of all record classes. Every Java source file implicitly imports the `java.lang.Record` class, as well as all other types in the `java.lang` package, regardless of whether you enable or disable preview features. However, if your application imports another class named `Record` from a different package, you might get a compiler error.

Consider the following class declaration of `com.myapp.Record`:

```
package com.myapp;

public class Record {
    public String greeting;
    public Record(String greeting) {
        this.greeting = greeting;
    }
}
```

The following example, `org.example.MyappPackageExample`, imports `com.myapp.Record` with a wildcard but doesn't compile:

```
package org.example;
import com.myapp.*;

public class MyappPackageExample {
    public static void main(String[] args) {
       Record r = new Record("Hello world!");
    }
}
```

The compiler generates an error message similar to the following:

```
./org/example/MyappPackageExample.java:6: error: reference to Record is ambiguous
       Record r = new Record("Hello world!");
       ^
  both class com.myapp.Record in com.myapp and class java.lang.Record in java.lang match

./org/example/MyappPackageExample.java:6: error: reference to Record is ambiguous
       Record r = new Record("Hello world!");
                      ^
  both class com.myapp.Record in com.myapp and class java.lang.Record in java.lang match
```

Both `Record` in the `com.myapp` package and `Record` in the `java.lang` package are imported with wildcards. Consequently, neither class takes precedence, and the compiler generates an error message when it encounters the use of the simple name `Record`.

To enable this example to compile, the `import` statement can be changed so that it imports the fully qualified name of `Record`:

```
import com.myapp.Record;
```

The introduction of classes in the `java.lang` package is rare but sometimes necessary. Previous examples are `Enum` in Java 5, `Module` in Java 9, and `Record` in Java 14.

Java grammar

```
RecordDeclaration:
  {ClassModifier} `record` TypeIdentifier [TypeParameters]
    RecordHeader [SuperInterfaces] RecordBody

RecordHeader:
 `(` [RecordComponentList] `)`

RecordComponentList:
 RecordComponent { `,` RecordComponent}

RecordComponent:
 {Annotation} UnannType Identifier
 VariableArityRecordComponent

VariableArityRecordComponent:
 {Annotation} UnannType {Annotation} `...` Identifier

RecordBody:
  `{` {RecordBodyDeclaration} `}`

RecordBodyDeclaration:
  ClassBodyDeclaration
  CompactConstructorDeclaration

CompactConstructorDeclaration:
  {ConstructorModifier} SimpleTypeName ConstructorBody
```

Class-file representation

The `class` file of a record uses a `Record` attribute to store information about the record's components:

```
Record_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 components_count;
    record_component_info components[components_count];
}

record_component_info {
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

If the record component has a generic signature that is different from the erased descriptor then there must be a `Signature` attribute in the `record_component_info` structure.

Reflection API

We add two public methods to `java.lang.Class`:

- `RecordComponent[] getRecordComponents()` — Returns an array of `java.lang.reflect.RecordComponent` objects. The elements of this array correspond to the record's components, in the same order as they appear in the record declaration. Additional information can be extracted from each element of the array, including its name, annotations, and accessor method.
- `boolean isRecord()` — Returns true if the given class was declared as a record. (Compare with `isEnum`.)

Alternatives

Record classes can be considered a nominal form of *tuples*. Instead of record classes, we could implement structural tuples. However, while tuples might offer a lightweight means of expressing some aggregates, the result is often inferior aggregates:

- A central aspect of Java's design philosophy is that *names matter*. Classes and their members have meaningful names, while tuples and tuple components do not. That is, a `Person` record class with components `firstName` and `lastName` is clearer and safer than an anonymous tuple of two strings.
- Classes allow for state validation through their constructors; tuples typically do not. Some data aggregates (such as numeric ranges) have invariants that, if enforced by the constructor, can thereafter be relied upon. Tuples do not offer this ability.
- Classes can have behavior that is based on their state; co-locating the state and behavior makes the behavior more discoverable and easier to access. Tuples, being raw data, offer no such facility.

Dependencies

Record classes work well with another feature currently in preview, namely *sealed classes* (JEP 360). For example, a family of record classes can be explicitly declared to implement the same sealed interface:

```
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 record classes and sealed classes is sometimes referred to as *algebraic data types*. Record classes allow us to express *products*, and sealed classes allow us to express *sums*.

In addition to the combination of record classes and sealed classes, record classes lend themselves naturally to pattern matching. Because record classes couple their API to their state description, we will eventually be able to derive deconstruction patterns for record classes as well, and use sealed type information to determine exhaustiveness in `switch` expressions with type patterns or deconstruction patterns.