5 Record Classes

Record classes, which are a special kind of class, help to model plain data aggregates with less ceremony than norma

For background information about record classes, see JEP 395.

A record declaration specifies in a header a description of its contents; the appropriate accessors, constructor, `equa` and `toString` methods are created automatically. A record's fields are final because the class is intended to serve a carrier".

For example, here is a record class with two fields:

```
record Rectangle(double length, double width) { }
```

This concise declaration of a rectangle is equivalent to the following normal class:

```
public final class Rectangle {
    private final double length;
    private final double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    double length() { return this.length; }
    double width()  { return this.width; }

    // Implementation of equals() and hashCode(), which specify
    // that two record objects are equal if they
    // are of the same type and contain equal field values.
    public boolean equals...
    public int hashCode...

    // An implementation of toString() that returns a string
    // representation of all the record class's fields,
    // including their names.
    public String toString() {...}
}
```

A record class declaration consists of a name; optional type parameters (generic record declarations are supported); lists the "components" of the record; and a body.

A record class declares the following members automatically:

- For each component in the header, the following two members:
  - A `private final` field with the same name and declared type as the record component. This field is sometimes referred to as a *component field*.
  - A `public` accessor method with the same name and type of the component; in the `Rectangle` record cl example, these methods are `Rectangle::length()` and `Rectangle::width()`.
- A *canonical constructor* whose signature is the same as the header. This constructor assigns each argument fr expression that instantiates the record class to the corresponding component field.

- Implementations of the `equals` and `hashCode` methods, which specify that two record classes are equal if the same type and contain equal component values.
- An implementation of the `toString` method that includes the string representation of all the record class's co with their names.

As record classes are just special kinds of classes, you create a record object (an instance of a record class) with the `r` example:

```
Rectangle r = new Rectangle(4,5);
```

The Canonical Constructor of a Record Class

The following example explicitly declares the canonical constructor for the `Rectangle` record class. It verifies that `l` are greater than zero. If not, it throws an `IllegalArgumentException`:

```
record Rectangle(double length, double width) {
    public Rectangle(double length, double width) {
        if (length <= 0 || width <= 0) {
            throw new java.lang.IllegalArgumentException(
                String.format("Invalid dimensions: %f, %f", length, width));
        }
        this.length = length;
        this.width = width;
    }
}
```

Repeating the record class's components in the signature of the canonical constructor can be tiresome and error-pro you can declare a *compact constructor* whose signature is implicit (derived from the components automatically).

For example, the following compact constructor declaration validates `length` and `width` in the same way as in the p

```
record Rectangle(double length, double width) {
    public Rectangle {
        if (length <= 0 || width <= 0) {
            throw new java.lang.IllegalArgumentException(
                String.format("Invalid dimensions: %f, %f", length, width));
        }
    }
}
```

This succinct form of constructor declaration is only available in a record class. Note that the statements `this.leng` and `this.width = width;` which appear in the canonical constructor do not appear in the compact constructor. compact constructor, its implicit formal parameters are assigned to the record class's private fields corresponding to

Explicit Declaration of Record Class Members

You can explicitly declare any of the members derived from the header, such as the `public` accessor methods that c record class's components, for example:

```
record Rectangle(double length, double width) {

    // Public accessor method
    public double length() {
        System.out.println("Length is " + length);
        return length;
    }
}
```

If you implement your own accessor methods, then ensure that they have the same characteristics as implici iv example, they're declared `public` and have the same return type as the corresponding record class compon ir implement your own versions of the `equals`, `hashCode`, and `toString` methods, then ensure that they have the sa characteristics and behavior as those in the `java.lang.Record` class, which is the common superclass of all recorc

You can declare static fields, static initializers, and static methods in a record class, and they behave as they would in for example:

```
record Rectangle(double length, double width) {

    // Static field
    static double goldenRatio;

    // Static initializer
    static {
        goldenRatio = (1 + Math.sqrt(5)) / 2;
    }

    // Static method
    public static Rectangle createGoldenRectangle(double width) {
        return new Rectangle(width, width * goldenRatio);
    }
}
```

You cannot declare instance variables (non-static fields) or instance initializers in a record class.

For example, the following record class declaration doesn't compile:

```
record Rectangle(double length, double width) {

    // Field declarations must be static:
    BiFunction<Double, Double, Double> diagonal;

    // Instance initializers are not allowed in records:
    {
        diagonal = (x, y) -> Math.sqrt(x*x + y*y);
    }
}
```

You can declare instance methods in a record class, independent of whether you implement your own accessor metl declare nested classes and interfaces in a record class, including nested record classes (which are implicitly static). Fo

```
record Rectangle(double length, double width) {

    // Nested record class
    record RotationAngle(double angle) {
        public RotationAngle {
            angle = Math.toRadians(angle);
        }
    }

    // Public instance method
    public Rectangle getRotatedRectangleBoundingBox(double angle) {
        RotationAngle ra = new RotationAngle(angle);
        double x = Math.abs(length * Math.cos(ra.angle())) +
                   Math.abs(width * Math.sin(ra.angle()));
        double y = Math.abs(length * Math.sin(ra.angle())) +
                   Math.abs(width * Math.cos(ra.angle()));
        return new Rectangle(x, y);
    }
}
```

You cannot declare `native` methods in a record class.

Features of Record Classes

A record class is implicitly `final`, so you cannot explicitly extend a record class. However, beyond these restrictions, behave like normal classes:

- You can create a generic record class, for example:

```
record Triangle<C extends Coordinate> (C top, C left, C right) { }
```

- You can declare a record class that implements one or more interfaces, for example:

```
record Customer(...) implements Billable { }
```

- You can annotate a record class and its individual components, for example:

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface GreaterThanZero { }
```

```
record Rectangle(
    @GreaterThanZero double length,
    @GreaterThanZero double width) { }
```

If you annotate a record component, then the annotation may be propagated to members and constructors of class. This propagation is determined by the contexts in which the annotation interface is applicable. In the pre example, the `@Target(ElementType.FIELD)` meta-annotation means that the `@GreaterThanZero` annota propagated to the field corresponding to the record component. Consequently, this record class declaration wo equivalent to the following normal class declaration:

```
public final class Rectangle {
    private final @GreaterThanZero double length;
    private final @GreaterThanZero double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    double length() { return this.length; }
    double width() { return this.width; }
}
```

Record Classes and Sealed Classes and Interfaces

Record classes work well with sealed classes and interfaces. See Record Classes as Permitted Subclasses for an exam

Local Record Classes

A local record class is similar to a local class; it's a record class defined in the body of a method.

In the following example, a merchant is modeled with a record class, `Merchant`. A sale made by a merchant is also m record class, `Sale`. Both `Merchant` and `Sale` are top-level record classes. The aggregation of a merchant and their t sales is modeled with a *local* record class, `MonthlySales`, which is declared inside the `findTopMerchants` methoc class improves the readability of the stream operations that follow:

```
import java.time.*;
import java.util.*;
import java.util.stream.*;

record Merchant(String name) { }

record Sale(Merchant merchant, LocalDate date, double value) { }

public class MerchantExample {
    List<Merchant> findTopMerchants(
        List<Sale> sales, List<Merchant> merchants, int year, Month month) {

        // Local record class
        record MerchantSales(Merchant merchant, double sales) {}

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

    double computeSales(List<Sale> sales, Merchant mt, int yr, Month mo) {
        return sales.stream()
            .filter(s -> s.merchant().name().equals(mt.name()) &&
                s.date().getYear() == yr &&
                s.date().getMonth() == mo)
            .mapToDouble(s -> s.value())
            .sum();
    }

    public static void main(String[] args) {

        Merchant sneha = new Merchant("Sneha");
        Merchant raj = new Merchant("Raj");
        Merchant florence = new Merchant("Florence");
        Merchant leo = new Merchant("Leo");

        List<Merchant> merchantList = List.of(sneha, raj, florence, leo);

        List<Sale> salesList = List.of(
            new Sale(sneha,    LocalDate.of(2020, Month.NOVEMBER, 13), 11034.20),
            new Sale(raj,      LocalDate.of(2020, Month.NOVEMBER, 20),  8234.23),
            new Sale(florence, LocalDate.of(2020, Month.NOVEMBER, 19), 10003.67),
            // ...
            new Sale(leo,      LocalDate.of(2020, Month.NOVEMBER,  4),  9645.34));

        MerchantExample app = new MerchantExample();

        List<Merchant> topMerchants =
            app.findTopMerchants(salesList, merchantList, 2020, Month.NOVEMBER);
        System.out.println("Top merchants: ");
        topMerchants.stream().forEach(m -> System.out.println(m.name()));
    }
}
```

Like nested record classes, local record classes are implicitly static, which means that their own methods can't access the enclosing method, unlike local classes, which are never static.

Static Members of Inner Classes

Prior to Java SE 16, you could not declare an explicitly or implicitly static member in an inner class unless that memb variable. This means that an inner class cannot declare a record class member because nested record classes are im

In Java SE 16 and later, an inner class may declare members that are either explicitly or implicitly static, which includ members. The following example demonstrates this:

```
public class ContactList {

    record Contact(String name, String number) { }

    public static void main(String[] args) {

        class Task implements Runnable {

            // Record class member, implicitly static,
            // declared in an inner class
            Contact c;

            public Task(Contact contact) {
                c = contact;
            }
            public void run() {
                System.out.println(c.name + ", " + c.number);
            }
        }

        List<Contact> contacts = List.of(
            new Contact("Sneha", "555-1234"),
            new Contact("Raj", "555-2345"));
        contacts.stream()
                .forEach(cont -> new Thread(new Task(cont)).start());
    }
}
```

Record Serialization

You can serialize and deserialize instances of record classes, but you can't customize the process by providing `write` `readObject`, `readObjectNoData`, `writeExternal`, or `readExternal` methods. The components of a record cla serialization, while the canonical constructor of a record class governs deserialization. See Serializable Records for m and an extended example. See also the section Serialization of Records in the *Java Object Serialization Specification*

APIs Related to Record Classes

The `abstract` class `java.lang.Record` is the common superclass of all record classes.

You might get a compiler error if your source file imports a class named `Record` from a package other than `java.l` file automatically imports all the types in the `java.lang` package though an implicit `import java.lang.*;` state includes the `java.lang.Record` class, regardless of whether preview features are enabled or disabled.

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

```
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 ma

./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 ma
```

Both `Record` in the `com.myapp` package and `Record` in the `java.lang` package are imported with a wildcard. Con
class takes precedence, and the compiler generates an error when it encounters the use of the simple name `Record`

To enable this example to compile, change the `import` statement so that it imports the fully qualified name of `Reco`

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

Note: The introduction of classes in the `java.lang` package is rare but necessary from time to time, su
Java SE 5, `Module` in Java SE 9, and `Record` in Java SE 14.

The class `java.lang.Class` has two methods related to record classes:

- `RecordComponent[] getRecordComponents()`: Returns an array of `java.lang.reflect.RecordComp`
  objects, which correspond to the record class's components.
- `boolean isRecord()`: Similar to `isEnum()` except that it returns `true` if the class was declared as a record c

Previous Page Next Page