This feature **enhances switch to work on any type and to match on more complex patterns**.

These additions are **backwards compatible**, switch with the traditional constants work just as before, for example, with Enum values:

```
var symbol = switch (expression) {
  case ADDITION       -> "+";
  case SUBTRACTION    -> "-";
  case MULTIPLICATION -> "*";
  case DIVISION       -> "/";
};
```

However, now it also **works with type patterns** introduced by JEP 394: Pattern Matching for instanceof:

```
return switch (expression) {
  case Addition expr       -> "+";
  case Subtraction expr    -> "-";
  case Multiplication expr -> "*";
  case Division expr       -> "/";
};
```

A pattern supports **guards**, written as type pattern when guard expression:

```
String formatted = switch (o) {
    case Integer i when i > 10 -> String.format("a large Integer %d", i);
    case Integer i             -> String.format("a small Integer %d", i);
    default                    -> "something else";
};
```

This makes a very nice symmetry with the type patterns used in if statements, because similar patterns can be used as conditionals:

```
if (o instanceof Integer i && i > 10) {
  return String.format("a large Integer %d", i);
} else if (o instanceof Integer i) {
  return String.format("a large Integer %d", i);
} else {
  return "something else";
}
```

Similarly to the type patterns in if conditions, the **scope of the pattern variables are flow sensitive**. For example, in the case below the scope of i is the guard expression and the right hand side expression:

```
case Integer i when i > 10 -> String.format("a large Integer %d", i);
```

Generally it works just as you'd expect, but there are many rules and edge cases involved. If you are interested, I recommend to read the corresponding JEPs or see

the Pattern matching for instanceof chapter.

Switch can now also match null values. Traditionally, when a null value was supplied to a switch, it threw a NullPointerException. For backwards compatibility, this is still the case when there's no explicit null pattern defined. However, now an explicit case for null can be added:

```
switch (s) {
  case null  -> System.out.println("Null");
  case "Foo" -> System.out.println("Foo");
  default    -> System.out.println("Something else");
}
```

Switch expressions always had to be exhaustive, in other words, they have to cover all possible input types. This is still the case with the new patterns, the Java compiler **emits an error when the switch is incomplete**:

```
Object o = 1234;

// OK
String formatted = switch (o) {
    case Integer i             -> String.format("a small Integer %d", i);
    default                    -> "something else";
};

// Compile error - 'switch' expression does not cover all possible input values
// Since o is an Object, the only way to fix it is to add a default case
String formatted = switch (o) {
    case Integer i             -> String.format("a small Integer %d", i);
};
```

This feature **synergizes well with enums, Sealed Classes and generics**. If there's only a fixed set of possible alternatives that exist, the default case can be omitted. Also, this feature helps a lot to **maintain the integrity of the codebase when the domain is extended** — for example, with a new constant in the enum. Due to the exhaustiveness check, all related switch expressions will yield a compiler error where the default case is not handled. For another example, consider the example code from the JEP:

```
sealed interface I<T> permits A, B {}
final class A<X> implements I<String> {}
final class B<Y> implements I<Y> {}

static int testGenericSealedExhaustive(I<Integer> i) {
    return switch (i) {
        case B<Integer> bi -> 42;
    };
}
```

This code compiles because the compiler can detect that only A and B are the valid subtypes of I, and that due to the generic parameter Integer, the parameter can only be an instance of B<Integer>.

Exhaustiveness is **checked at compile time** but if at runtime a new implementation pops up (e.g. from a separate compilation), the compiler also inserts a synthetic default case that throws a MatchException.

The compiler also performs the opposite of the exhaustiveness check: it **emits an error when a case completely dominates another**.

```
Object o = 1234;

// Compile error - the second case is dominated by a preceding case label
String formatted = switch (o) {
   case Integer i             -> String.format("a small Integer %d", i);
   case Integer i when i > 10 -> String.format("a large Integer %d", i);
   default                    -> "something else";
};
```

This makes default cases to emit a compile time error if all possible types are handled otherwise.

For readability reasons, the dominance checking **forces constant case labels to appear before the corresponding type-based pattern**. The goal is to always have the more specific cases first. As an example, the cases in the following snippet are only valid in this exact order. If you'd try to rearrange them, you would get a compilation error.

```
switch(num) {
   case -1, 1 -> "special case";
   case Integer i && i > 0 -> "positive number";
   case Integer i -> "other integer";
}
```

Resources:

- Inside Java Podcast Episode 17 “Pattern Matching for switch” with Gavin Bierman
- Inside Java Podcast Episode 26: “Java 19 is Here!” with Brian Goetz and Ron Pressler
- Inside Java Podcast Episode 28: “Java Language - State of the Union” with Gavin Bierman

Record Patterns

Available since: JDK 21 (Preview in JDK 19 JDK 20)

With Switch and instanceof it's possible match on patterns. The first kind of supported pattern was the *type pattern* that tests if the parameter is of a given type, and if so, it captures it to a new variable:

```
// Pattern matching with a type pattern using instanceof
if (obj instanceof String s) {
  // ... use s ...
}

// Pattern matching with a type pattern using switch
switch (obj) {
    case String s -> // ... use s ...
    // ... other cases ...
};
```

Record Patterns extends the pattern matching capabilities of Java beyond simple type patterns to **match and deconstruct Record values**. It **supports nesting** to enable declarative, data focused programming.

For a demonstration, consider the following ColoredPoint, which is composed of a Point2D and a Color:

```
interface Point { }
record Point2D(int x, int y) implements Point { }
enum Color { RED, GREEN, BLUE }
record ColoredPoint(Point p, Color c) { }

Object r = new ColoredPoint(new Point2D(3, 4), Color.GREEN);
```

Note, that while we use a Point2D to construct r, ColoredPoint accepts other Point implementations as well.

Without pattern matching, we'd need two explicit type checks to detect if object r is a ColoredPoint that holds Point2D, and also quite some work to extract the values x, y, and c:

```
if (r instanceof ColoredPoint) {
  ColoredPoint cp = (ColoredPoint) r;
  if (cp.p() instanceof Point2D) {
    Point2D pt = (Point2D) cp.p();
    int x = pt.x();
    int y = pt.y();
    Color c = cp.c();

    // work with x, y, and c
  }
}
```

Using type patterns for instanceof makes things a bit better as there's no longer a need for the casts, but conceptually the same thing needs to be done as previously:

```
if (r instanceof ColoredPoint cp && cp.p() instanceof Point2D pt) {
    int x = pt.x();
    int y = pt.y();
    Color c = cp.c();

    // work with x, y, and c
}
```

However this can be greatly simplified with Record Patterns that **offer a concise way to test the shape of the data and to deconstruct it**:

```
if (r instanceof ColoredPoint(Point2D(int x, int y), Color c)) {
  // work with x, y, and c
}
```

Patterns can be nested, which allows **easy navigation in the objects, allowing the programmer to focus on the data expressed by those objects**. Also it **centralizes error handling**. A nested pattern will only match if all subpatterns match. There's no need to manually handle each individual branch manually.

It is worth noting the symmetry between the construction and the deconstruction of the record values:

```
// construction
var r = new ColoredPoint(new Point2D(3, 4), Color.GREEN);

// deconstruction
if (r instanceof ColoredPoint(Point2D(int x, int y), Color c)) { }
```

Record patterns also support not just instanceof but the Switch expression as well:

```
var length = switch (r) {
  case ColoredPoint(Point2D(int x, int y), Color c) -> Math.sqrt(x*x + y*y);
  case ColoredPoint(Point p, Color c) -> 0;
}
```

Resources:

- Data Oriented Programming in Java by Brian Goetz

Unnamed Patterns and Variables (Preview 🔍)

Available since: Preview in JDK 21

In Java 8, the compiler emitted a warning when '_' was used as an identifier. Java 9 took this a step further making the underscore character illegal as an identifier, reserving this name to have special semantics in the future. The goal of these actions was to prepare the stage for unnamed variables, which is a preview feature in Java 21.

With **unnamed variables**, now it's possible to use the underscore character wherever a variable name would be used, for example at variable declaration, catch clause, or the parameter list of a lambda. However, it's not a regular variable name: it means "don't care" as **underscore can be redeclared, and can not be referenced**. It does not pollute the scope with an unnecessary variable (potentially causing trouble and leakage), and never shadows any other variable. Simply put, it enforces and communicates the intention of the programmer much better than the old alternatives, of using variable names such as ignore.

```
var _ = mySet.add(x); // ignore the return value

try {
  // ...
} catch (Exception _) { // ignore the exception object
  // ...
}

list.stream()
  .map((_) -> /* ... */) // ignore the parameter
  .toList();
```

A related feature is called **unnamed patterns**, which can be used in pattern matching to ignore subpatterns.

Let's say, we have the following pattern, where we don't need the Color information:

```
if (r instanceof ColoredPoint(Point(int x, int y), Color c)) {
  // do something with x and y, but c is not needed
}
```

With var, it's possible to ignore the type of c, but the match will make c available in the scope:

```
if (r instanceof ColoredPoint(Point(int x, int y), var c)) {
  // ... x ... y ...
```

```
}
```

Using underscore, marking it an unnamed pattern however ignores not just the type, but the name as well:

```
if (r instanceof ColoredPoint(Point(int x, int y), _)) {
  // ... x ... y ...
}
```

Note, that underscore may only be used in a nested position, but not on the top level.

This feature is in **preview** , it has to be explicitly enabled with the --enable-preview flag.

String Templates (Preview 🔍)

Available since: Preview in JDK 21

String Templates are an **extension to the single-line String literals and Text Blocks, allowing String interpolation and much more**.

In previous Java versions String interpolation was a lot of manual work. For example, one could do it with the + operator or the StringBuilder. It's also possible to do with String::format but with that it is very easy to accidentally have the incorrect number of arguments.

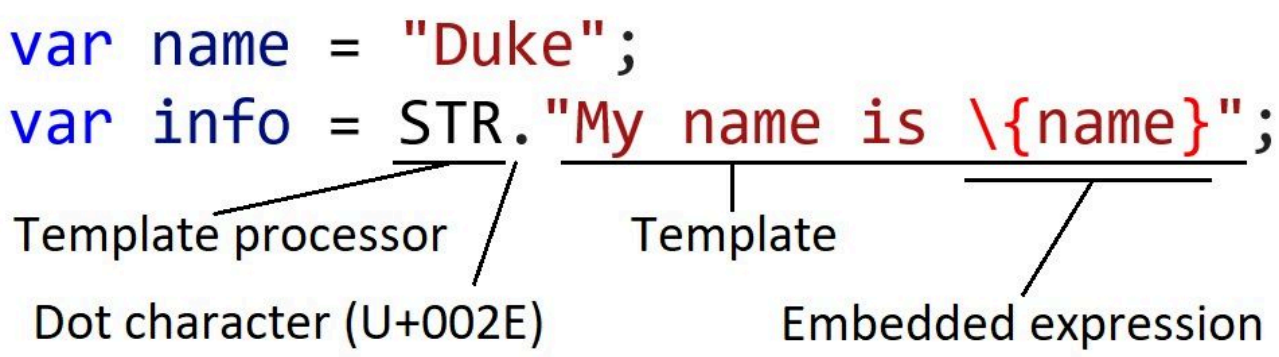
The main use-case for String Templates is to make this use-case easier:

```
// string interpolation in Java
var name = "Duke";
var info = STR."My name is \{name}";
```

Let's compare this with a similar feature, Template literals in JavaScript:

```
// string interpolation in JavaScript
var name = "Duke";
var info = `My name is ${name}`;
```

The most obvious difference is that the new syntax introduced in Java for string interpolation is not a new kind of literal (e.g. a backtick \`` as in Javascript), but the programmer must explicitly use a *template processor* — STR` in this case.



The *template processor* is responsible including the expression results into the *template*, performing any validation and escaping if needed. The evaluation of the embedded expressions happens eagerly when the template is constructed. Templates can't exist on their own, they can only be used in combination with a template processor.

STR is a template processor instance is used for the string interpolation of a supplied template. It is a public static final field which is automatically imported to all java source files for convenience.

In case of simple String interpolation with STR, the processor returns a String, but it's not a requirement. Other processors might emit objects of different types. Also, for STR not much validation and escaping happens, but other processors might choose to do something more complex.

Allowing the programmer to choose and implement template processors as they wish make this feature very flexible and robust.

For example, as mentioned in the JEP, it's possible to easily create a JSON processor, that can be used to turn a templated String into a JSONObject.

```
var JSON = StringTemplate.Processor.of(
  (StringTemplate st) -> new JSONObject(st.interpolate())
);

JSONObject doc = JSON."""
{
   "name":   "\{name}",
   "phone":  "\{phone}",
   "adress": "\{address}"
};
""";
```

Because the template processor has fine-grained access to the evaluation mechanism (like text fragments and embedded expressions), it's also possible to create a processor to produce prepared statements, XMLs, or localization strings, performing all the necessary validation and escaping.

Note, that by convention the **name of template processors are always written in uppercase**, even in case of local variables. A processor may also throw a **checked exception in which case the usual rules apply: the caller has to handle or propagate the exception**.

In addition to STR, there are a few more built-in processors.

One is FMT, which is just like STR but accepts format specifiers that appear left to the embedded expression as specified by java.util.Formatter:

```
double value = 17.8D;
var result = FMT."Value is %7.2f\{value}";
// => "Value is 17.80"
```

Finally, there's the RAW processor which can be used to create StringTemplate objects, to be processed later by an other processor:

```
StringTemplate template = RAW."Hello \{name}";
// ...
String result = STR.process(template);
```

Java uses backslash (\) instead of the usual dollar sign ($) for embedded expressions in templates for **backwards compatibility reasons**. In previous Java versions "${...}" is a valid String, however "{}" result in a compilation error, complaining about an illegal escape character. By choosing a new symbol, it is guaranteed that no existing program will break with the addition of String Templates.

To summarize, Java aims to have a string interpolation that is a bit different from similar features in other languages, trading verboseness for flexibility and robustness, and for backwards compatibility.

This feature is in **preview** , it has to be explicitly enabled with the --enable-preview flag.

Resources:

- Inside Java Podcast Episode 26: “Java 19 is Here!” with Brian Goetz and Ron Pressler
- Inside Java Podcast Episode 28: “Java Language - State of the Union” with Gavin Bierman

Unnamed Classes and Instance Main Methods (Preview 🔍)

Available since: Preview in JDK 21

In old Java versions, one needed write quite some boilerplate code even for the simplest of the applications:

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, World!");
  }
}
```

Usually, this only matters at project setup, but there are cases where it might be problematic:

- Newcomers to the language have to learn quite some extra keywords and concepts before they can try anything.
- It's tedious to write small programs such as shell scripts.

The introduction of **unnamed classes** and **instance main methods** makes life a bit easier in these two cases.

Instance main methods makes the Java launch protocol more flexible, by making some aspects of the main method optional.

From now on, visibility does not matter as long as the main method is **non-private**, the **String[] parameter can also be ommitted**, and it can be an **instance method**.

Because it's possible to define main methods in multiple ways, the new launch protocol defines priorities to choose which one to use:

1. static main with args
2. static main without args
3. instance main with args
4. instance main without args

In case of the non-static main method, an instance of the enclosing class will be automatically created at startup. The introduction of instance main methods brings a slight breaking change: if there's an instance main method present, then static main methods from superclasses won't be considered. In these cases the JVM emits a warning.

```
class HelloWorld {
  void main() {
    System.out.println("Hello, World!");
  }
}
```

This is already quite some improvement, but there's more. **Methods to exist outside of an enclosing class** in which case they are automatically wrapped into a synthetic **unnamed class**.

Unnamed classes work similarly to *unnamed packages* and *unnamed modules*. If a class does not have a package declaration, it will be part of the unnamed package, in which case they can not be referenced by classes from named packages. If a package is not part of a module, it will be part of the unnamed module, so packages from other modules can't refer them.

Unnamed classes are the extension of this idea to one more micro level: when the source file has no explicit class declaration, then the contents will become a member of an unnamed class. The unnamed class is always a member of the unnamed package and the unnamed module. It is final, can not implement any interface or extend any class (other than Object), and can not be referenced. This also means that it's not possible to create instances of an unnamed class, so it is only useful as an entry point to a program. For this reason the compiler even enforces that unnamed classes must have a main method. Otherwise, an unnamed class can be used as a regular class, for example it can have instance and static methods and fields.

So putting it together, the main method can be written as follows:

```
void main() {
  System.out.println("Hello, World!");
}
```

Which makes it very convenient to write short shell scripts:

```
1 #!/usr/bin/env java --source 21 --enable-preview
2
3 void main() {
4     System.out.println("Hello, World!");
5 }
6
```

```
--- ~ » ./hello
Note: ./hello uses preview features of Java SE 21.
Note: Recompile with -Xlint:preview for details.
Hello, World!
```

(The warnings regarding the preview features sadly can not be turned off, but once this feature is production ready, it will go away.)

Resources and potential future work:

- Paving the on-ramp by Brian Goetz
- Script Java Easily in 21 and Beyond - Inside Java Newscast #49
- JEP draft: Launch Multi-File Source-Code Programs

This feature is in **preview** , it has to be explicitly enabled with the --enable-preview flag.

Sealed Classes

Available since: JDK 17 (Preview in JDK 15 JDK 16)

Sealed classes and interfaces can be used to restrict which other classes or interfaces may extend or implement them. It gives a tool to better design public APIs, and provides an alternative to Enums to model fixed number of alternatives.

Older Java versions also provide some mechanisms to achieve a similar goal. Classes marked with the final keyword can not be extended at all, and with access modifiers it's possible to ensure that classes are only extended by members of the same package.

On top of these existing facilities *Sealed classes* add a fine-grained approach, allowing the authors to explicitly list the subclasses.

```
public sealed class Shape
    permits Circle, Quadrilateral {...}
```

In this case, it's only permitted to extend the Shape class with the Circle and the Quadrilateral classes. Actually, the term *permits* might be a bit misleading, since it not only permits, but it is **required that the listed classes directly extend the sealed class**.

Additionally, as one might expect from such a grant, it's a **compile error if any other classes try to extend** the sealed class.

Classes that extend a sealed class have to conform to a few rules.

Authors are forced to always explicitly define the boundaries of a sealed type hierarchy by using exactly one of the following modifiers on the permitted

subclasses:

- final: the subclass can not be extended at all
- sealed: the subclass can only be extended by some permitted classes
- non-sealed: the subclass can be freely extended

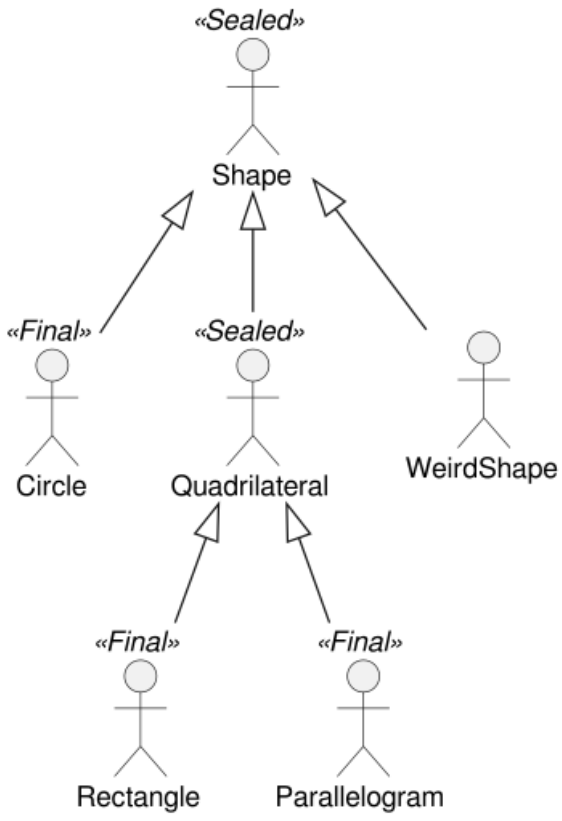
Because the subclasses can also be sealed it means that it's possible to define **whole hierarchies of fixed alternatives**:

```
public sealed class Shape
    permits Circle, Quadrilateral, WeirdShape {...}

public final class Circle extends Shape {...}

public sealed class Quadrilateral extends Shape
    permits Rectangle, Parallelogram {...}
public final class Rectangle extends Quadrilateral {...}
public final class Parallelogram extends Quadrilateral {...}

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



If these classes are short and mostly about data it might make sense to declare all of them in the **same source file in which case the permits clause can be omitted**:

```
public sealed class Shape {
  public final class Circle extends Shape {}

  public sealed class Quadrilateral extends Shape {
```

```
  public final class Rectangle extends Quadrilateral {}
  public final class Parallelogram extends Quadrilateral {}
 }

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

Record classes can also be part of a sealed hierarchy as leafs because they are implicitly final.

Permitted classes must be located in the same package as the superclass — or, in case of using java modules, they have to reside in the same module.

⚠️ Tip: Consider using Sealed classes over Enums

Before *Sealed classes*, it was only possible to model fixed alternatives using *Enum types*. E.g.:

```
enum Expression {
 ADDITION,
 SUBTRACTION,
 MULTIPLICATION,
 DIVISION
}
```

However, all variations need to be in the same source file, and *Enum types* doesn't support modelling cases when an instance is needed instead of a constant, e.g. to represent individual messages of a type.

Sealed classes offer a nice alternative to *Enum types* making it possible to use regular classes to model the fixed alternatives. This will come to full power once *Pattern Matching for switch* becomes production ready, after that *Sealed classes* can be used in switch expressions just like enums, and the compiler can automatically check if all cases are covered.

Enum values can be enumerated with the values method. For Sealed classes and interfaces, the permitted subclasses can be listed with getPermittedSubclasses.

Record Classes

Available since: JDK 16 (Preview in JDK 14 JDK 15)

Record Classes introduce a new type declaration to the language to define immutable data classes. Instead of the usual ceremony with private fields, getters and constructors, it allows us to use a **compact syntax**:

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

The Record Class above is **much like a regular class** that defines the following:

- two private final fields, int x and int y
- a constructor that takes x and y as a parameter
- x() and y() methods that act as getters for the fields
- hashCode, equals and toString, each taking x and y into account

They can be used much like normal classes:

```
var point = new Point(1, 2);
point.x(); // returns 1
point.y(); // returns 2
```

Record Classes intended to be **transparent carriers** of their **shallowly immutable data**. To support this design they come with a set of **restrictions**.

Fields of a Record Class are not only final by default, it's **not even possible to have any non-final fields**.

The header of the definition has to define everything about the possible states. It can't have additional fields in the body of the Record Class. Moreover, while it's possible to define additional constructors to provide default values for some fields, it's not possible to hide the *canonical constructor* that takes all record fields as arguments.

Finally, Record Classes **can't extend other classes**, they **can't declare native methods**, and they are **implicitly final** and **can't be abstract**.

Supplying data to a record is only possible through its constructor. By default a Record Class only has an implicit *canonical constructor*. If the data needs to be validated or normalized, the *canonical constructor* can also be defined explicitly:

```
public record Point(int x, int y) {
  public Point {
    if (x < 0) {
      throw new IllegalArgumentException("x can't be negative");
    }
    if (y < 0) {
      y = 0;
    }
```

```
  }
}
```

The implicit *canonical constructor* has the same visibility as the record class itself. In case it's explicitly declared, its access modifier must be at least as permissive as the access modifier of the Record Class.

It's also possible to define additional constructors, but they must delegate to other constructors. In the end the *canonical constructor* will always be called. These extra constructors might be useful to provide default values:

```
public record Point(int x, int y) {
  public Point(int x) {
    this(x, 0);
  }
}
```

Getting the data from a record is possible via its accessor methods. For each field x the record classes has a generated public getter method in the form of x().

These getters can also be explicitly defined:

```
public record Point(int x, int y) {
  @Override
  public int x() {
    return x;
  }
}
```

Note, that the Override annotation can be used in this case to make sure that the method declaration explicitly defines an accessor and not an extra method by accident.

Similarly to getters, hashCode, equals and toString methods are provided by default considering all fields; these methods can also be explicitly defined.

Finally, Record Classes can also have static and instance methods that can be handy to get derived information or to act as factory methods:

```
public record Point(int x, int y) {
  static Point zero() {
    return new Point(0, 0);
  }

  boolean isZero() {
    return x == 0 && y == 0;
  }
}
```

To sum it up: **Record Classes are only about the data they carry** without providing too much customization options.

Due to this special design, **serialization for records are much easier** and secure than for regular classes. As written in the JEP:

> 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.

Because serialization is based exactly on the field state and deserialization always calls the canonical constructor its impossible to create a Record with invalid state.

From the user point of view, enabling and using serialization can be done as usual:

```
public record Point(int x, int y) implements Serializable { }

public static void recordSerializationExample() throws Exception {
  Point point = new Point(1, 2);

  // Serialize
  var oos = new ObjectOutputStream(new FileOutputStream("tmp"));
  oos.writeObject(point);

  // Deserialize
  var ois = new ObjectInputStream(new FileInputStream("tmp"));
  Point deserialized = (Point) ois.readObject();
}
```

Note that it's no longer required to define a serialVersionUID, as the requirement for matching serialVersionUID values is waived for the Record Classes.

Resources:

- Inside Java Podcast Episode 4: “Record Classes” with Gavin Bierman
- Inside Java Podcast Episode 14: “Records Serialization” with Julia Boes and Chris Hegarty
- Towards Better Serialization - Brian Goetz, June 2019
- Record Serialization

⚠️ Tip: Use Local Records to model intermediate transformations

Complex data transformations require us to model intermediate values. Before Java 16, a typical solution was to rely on Pair or similar holder classes from a library, or to define your own (maybe inner static) class to hold this data.

The problem with this is that the former one quite often proves to be inflexible, and the latter one pollutes the namespace by introducing classes only used in context of a single method. It's also possible to define classes inside a method body, but due to their verbose nature it was rarely a good fit.

Java 16 improves on this, as **now it's also possible to define Local Records** in a method body:

```
public List<Product> findProductsWithMostSaving(List<Product> products) {
  record ProductWithSaving(Product product, double savingInEur) {}

  products.stream()
    .map(p -> new ProductWithSaving(p, p.basePriceInEur * p.discountPercentage))
    .sorted((p1, p2) -> Double.compare(p2.savingInEur, p1.savingInEur))
    .map(ProductWithSaving::product)
    .limit(5)
    .collect(Collectors.toList());
}
```

The compact syntax of the Record Classes are a nice match for the compact syntax of the Streams API.

In addition to Records, this change also **enables the use Local Enums and even Interfaces**.

⚠ Tip: Check your libraries

Record Classes do not adhere to the JavaBeans conventions:

- They have no default constructor.
- They do not have setter methods.
- The accessor methods does not follow the getX() form.

For these reasons, **some tools that expect JavaBeans might not fully work with records**.

One such case is that **records can't be used as JPA (e.g. Hibernate) entities**. There's a discussion about aligning the specification to Java Records on the jpa-dev mailinglist, but so far I did not find news about the state of the development process. It worth to mention however that **Records can be used for projections** without problems.

Most **other tools I've checked** (including Jackson, Apache Commons Lang, JSON-P and Guava) **support records, but since it's pretty new there are also some rough edges**. For example, Jackson, the popular JSON library was an early adopter of records. Most of its features, including serialization and deserialization work equally well for Record Classes and JavaBeans, but some features to manipulate the objects are yet to be adapted.

Related

Working with structured data in Java

Parse, compare and manipulate JSON-based data

Another example I've bumped into is Spring, which also support records out of the box for many cases. The list includes serialization and even dependency injection, but the ModelMapper library — used by many Spring applications — does not support mapping JavaBeans to Record Classes.

My advice is to **upgrade and check your tooling before adopting Record Classes** to avoid surprises, but generally it's a fair to assume that popular tools already have most of their features covered.

Check out my experiments with the tool integration for Record Classes on GitHub.

⚠ Tip: Use pattern matching to easily access members

Instead of doing all the work manually, consider using Pattern Matching for switch or Pattern Matching for instanceof in combination with Record Patterns to

easily access the members of a record.

Pattern Matching for instanceof

Available since: JDK 16 (Preview in **JDK 14** JDK 15)

In most cases, instanceof is typically followed by a cast:

```
if (obj instanceof String) {
    String s = (String) obj;
    // use s
}
```

At least, in the old days, because Java 16 extends instanceof to make this typical scenario less verbose:

```
if (obj instanceof String s) {
    // use s
}
```

The pattern is a combination of a *test* (obj instanceof String) and a *pattern variable* (s).

The *test* **works almost like the test for the old instanceof**, except it results in a compile error if it is guaranteed to pass all the time:

```
// "old" instanceof, without pattern variable:
// compiles with a condition that is always true
Integer i = 1;
if (i instanceof Object) { ... } // works

// "new" instanceof, with the pattern variable:
// yields a compile error in this case
if (i instanceof Object o) { ... } // error
```

Note, that the opposite case, where a pattern match will always fail, is already a compile-time error even with the old instanceof.

The *pattern variable* is extracted from the target only if the test passes. It **works almost like a regular non-final variable**:

- it can be modified
- it can shadow field declarations
- if there's a local variable with the same name, it will result in a compile error

However, special scoping rules are applied to them: a pattern variable is **in scope where it has definitely matched**, decided by flow scoping analysis.

The simplest case is what can be seen in the above example: if the test passes, the variable s can be used inside if block.

But the rule of "definitely matched" also applies for parts of more complicated conditions too:

```
if (obj instanceof String s && s.length() > 5) {
  // use s
}
```

s can be used in the second part of the condition because it's only evaluated when the first one succeeds and the instanceof operator has a match.

To bring an even less trivial example, early returns and exceptions can also guarantee matches:

```
private static int getLength(Object obj) {
  if (!(obj instanceof String s)) {
    throw new IllegalArgumentException();
  }

  // s is in scope - if the instanceof does not match
  //      the execution will not reach this statement
  return s.length();
}
```

The flow scoping analysis works similarly to existing flow analyses such as checking for definite assignment:

```
private static int getDoubleLength(String s) {
  int a; // 'a' declared but unassigned
  if (s == null) {
    return 0; // return early
  } else {
    a = s.length(); // assign 'a'
  }

  // 'a' is definitely assigned
  // so we can use it
  a = a * 2;
  return a;
}
```

I really like this feature as it's likely to reduce the unnecessary bloat caused the explicit casts in a Java program. Contrasting it with more modern languages however, this feature still seems to be a bit verbose.

For example in Kotlin you don't need to define the pattern variable:

```
if (obj is String) {
    print(obj.length)
}
```

In Java's case the pattern variables are added to ensure backwards compatibility as changing the type of obj in obj instanceof String would mean that when obj is used as an argument of an overloaded method, a the call could resolve to a different version of the method.

Text Blocks

Available since: JDK 15 (Preview in JDK 13 JDK 14)

Compared to other modern languages, in Java it was notoriously hard to express text containing multiple lines:

```
String html = "";
html += "<html>\n";
html += "  <body>\n";
html += "    <p>Hello, world</p>\n";
html += "  </body>\n";
html += "</html>\n";

System.out.println(html);
```

To make this situation more programmer-friendly, Java 15 introduced multi-line string literals called Text Blocks:

```
String html = """
        <html>
          <body>
            <p>Hello, world</p>
          </body>
        </html>
        """;

System.out.println(html);
```

They are similar to the old String literals but they can contain **new lines and quotes without escaping**.

Text Blocks start with """ followed by a new line, and end with """. The closing token can be at the end of the last line or in separate line such as is in the example

above.

They can be used anywhere an old String literal can be used and they both produce similar String objects.

For each line-break in the source code, there will be a \n character in the result.

```
String twoLines = """
        Hello
        World
        """;
```

This can be prevented by ending the line with the \ character, which can be useful in case of very long lines that you'd like to split into two for keeping the source code readable.

```
String singleLine = """
        Hello \
        World
        """;
```

Text Blocks can be aligned with neighboring Java code because **incidental indentation is automatically removed**. The compiler checks the whitespace used for indentation in each line to find the least indented line, and shifts each line to the left by this minimal common indentation.

This means that if the closing """ is in a separate line, the indentation can be increased by shifting the closing token to the left.

```
String noIndentation = """
        First line
        Second line
        """;

String indentedByToSpaces = """
        First line
        Second line
      """;
```

The opening """ does not count for the indentation removal so it's not necessary to line up the text block with it. For example, both of the following examples produce the same string with the same indentation:

```
String indentedByToSpaces = """
       First line
       Second line
     """;

String indentedByToSpaces = """
                    First line
```

```
        Second line
    """;
```

The String class also provides some programmatic ways to deal with indentation. The indent method takes an integer and returns a new string with the specified levels of additional indentation, while stripIndent returns the contents of the original string without all the incidental indentation.

Resources:

- Programmer's Guide To Text Blocks
- Definitive Guide To Text Blocks In Java 13
- Java Text Blocks - Bealdung

⚠️ Tip: Preserve trailing spaces

Trailing spaces in Text Blocks are ignored. This is usually not a problem but in some cases they do matter, for example in context of unit test when a method result is compared to a baseline value.

If this is the case be mindful about them and if a line ends with whitespace add \s or \t instead of the last space or tab to the end of the line.

⚠️ Tip: Produce the correct newline characters for Windows

Line endings are represented with different control characters on Unix and Windows. The former one uses a single line feed (\n), while the latter uses carriage return followed by line feed (\r\n).

However, regardless to the operating system you choose to use or how you encode new lines in the source code, Text Blocks will use a single \n for each new line, which can lead to compatibility issues.

```
Files.writeString(Paths.get("<PATH_TO_FILE>"), """
    first line
    second line
    """);
```

If a tool compatible only with the Windows line ending format (e.g. Notepad) is used to open such a file, it will display only a single line. Make sure that you use the correct control characters if you also target Windows, for example by calling String::replace to replace each "\n" with "\r\n".

⚠ Tip: Pay attention to consistent indentation

Text Blocks work well with any kind of indentation: tabs spaces or even the mix of these two. It's important though to use **consistent indentation** for each line in the block, otherwise the incidental indentation can't be removed.

Most editors offer autoformatting and automatically add indentation on each new line when you hit enter. Make sure to use the latest version of these tools to ensure they play well with Text Blocks, and don't try to add wrong indentations.

⚠ Tip: Interpolate Text Blocks with String Templates

Text Blocks do not support interpolation out of the box, but there are a couple of ways to achieve similar results. An alternative that works with older Java versions is to use String::formatted or String::format:

```
var name = "world";
var greeting = """
   hello
   %s
   """.formatted(name);
```

With Java 21 however, string interpolation is possible via the String templates preview feature:

```
var name = "world";
var greeting = STR."""
   hello
   \{name};
   """;
```

Helpful NullPointerExceptions

Available since: JDK 15 (Enabled with -XX:+ShowCodeDetailsInExceptionMessages in JDK 14)

This little gem is not really a language feature, but it's so nice that I wanted to include it in this list.

Traditionally, experiencing a NullPointerException was like this:

```
node.getElementsByTagName("name").item(0).getChildNodes().item(0).getNodeValue();

Exception in thread "main" java.lang.NullPointerException
        at Unlucky.method(Unlucky.java:83)
```

From the exception it's not obvious which method returned null in this case. For this reason many developers used to spread such statements over multiple lines to make sure they'll be able to figure out which step led to the exception.

From Java 15, there's no need to do that because NPE's describe which part was null in the statement. (Also, in in Java 14 you can enable it with the -XX:+ShowCodeDetailsInExceptionMessages flag.)

```
Exception in thread "main" java.lang.NullPointerException:
  Cannot invoke "org.w3c.dom.Node.getChildNodes()" because
  the return value of "org.w3c.dom.NodeList.item(int)" is null
        at Unlucky.method(Unlucky.java:83)
```

(Check the example project on GitHub)

The detailed message contains the action that could not be performed (Cannot invoke getChildNodes) and the reason for the failure (item(int) is null), making it much easier to find the exact source of the problem.

So overall **this feature is good for debugging, and also good for code readability** as there's one less reason to sacrifice it for a technical reason.

The Helpful NullPointerExceptions extension is implemented in the JVM so you get the same benefits for code compiled with older Java versions, and when using other JVM languages, such as Scala or Kotlin.

Note, that **not all NPEs get this extra info, just the ones that are created and thrown by the JVM** upon:

- reading or writing a field on null
- invoking method on null

- accessing or assigning an element of an array (indices are **not** part of the error message)
- unboxing null

Also note that this feature **does not support serialization**. For example, when an NPE is thrown on the remote code executed via RMI, the exception will not include the helpful message.

Currently the **Helpful NullPointerExceptions are disabled by default**, and have to be enabled with the -XX:+ShowCodeDetailsInExceptionMessages flag.

⚠️ Tip: Check your tooling

When upgrading to Java 15, make sure to check your application and infrastructure to ensure:

- sensitive variable names not end up in log files and web server responses
- log parsing tools can handle the new message format
- the additional overhead required to construct the additional details is okay

Switch Expressions

Available since: JDK 14 (Preview in JDK 12 JDK 13)

The good old switch got a facelift in Java 14. While Java keeps supporting the old switch statement, it adds the new **switch expression** to the language:

```
int numLetters = switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY                -> 7;
    default      -> {
        String s = day.toString();
        int result = s.length();
        yield result;
    }
};
```

The most striking difference is that this new form **can be used as an expression**. It can be used to populate variables as demonstrated in the example above, and it can be used wherever an expression is accepted:

```
int k = 3;
System.out.println(
   switch (k) {
      case  1 -> "one";
      case  2 -> "two";
      default -> "many";
   }
);
```

However, there are some other, more subtle differences between switch expressions and switch statements.

First, for switch expressions **cases don't fall-through**. So no more subtle bugs caused by missing breaks. To avoid the need for fall-through, **multiple constants can be specified for each case** in a comma separated list.

Second, each **case has its own scope**.

```
String s = switch (k) {
   case  1 -> {
      String temp = "one";
      yield temp;
   }
   case  2 -> {
      String temp = "two";
      yield temp;
   }
   default -> "many";
}
```

A branch is either a single expression or if it consist of multiple statements it has to be wrapped in a block.

Third, **cases of a switch expression are exhaustive**. This means that for String, primitive types and their wrappers the default case always has to be defined.

```
int k = 3;
String s = switch (k) {
   case  1 -> "one";
   case  2 -> "two";
   default -> "many";
}
```

For enums either a default case has to be present, or all cases have to be explicitly covered. Relying on the latter is quite nice to ensure that all values are considered. Adding an extra value to the enum will result in a compile error for all switch expressions where it's used.

```
enum Day {
   MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
```

```
}

Day day = Day.TUESDAY;
switch (day) {
   case  MONDAY -> ":(";
   case  TUESDAY, WEDNESDAY, THURSDAY -> ":|";
   case  FRIDAY -> ":)";
   case  SATURDAY, SUNDAY -> ":D";
}
```

For all these reasons preferring switch expressions over switch statements can lead to more maintainable code.

⚠️ Tip: Use arrow syntax

Switch expression can not only used with the lambda-like arrow-form cases. **The old switch statement with its colon-form cases can also be used as an expression** using yield:

```
int result = switch (s) {
   case "foo":
   case "bar":
      yield 2;
   default:
      yield 3;
};
```

This version can also be used as an expression, but it's more similar to the old switch statement because

- cases fall through
- cases share the same scope

My advice? Don't use this form, use switch expressions with the arrow syntax instead to get all the benefits.

Local-Variable Type Inference

Available since: JDK 11 (Without lambda support in JDK 10)

Probably the most significant language improvement since Java 8 is the addition of the var keyword. It was initially introduced in Java 10, and was further improved

in Java 11.

This feature allows us to reduce the ceremony of a local variable declaration by omitting the explicit type specification:

```
var greetingMessage = "Hello!";
```

While it looks similar to Javascript's var keyword, this is **not about dynamic typing**.

Take this quote from the JEP:

> We seek to improve the developer experience by reducing the ceremony associated with writing Java code, while maintaining Java's commitment to static type safety.

The type of the declared variables is **inferred at compile time**. In the example above the inferred type is String. Using var instead of an explicit type makes this piece of code less redundant, thus, easier to read.

Here's another good candidate for type inference:

```
MyAwesomeClass awesome = new MyAwesomeClass();
```

It's clear that in many cases this feature can improve code quality. However, sometimes it's better to stick with the explicit type declaration. Let's see a few examples where replacing a type declaration with var can backfire.

⚠️ Tip: Keep readability in mind

The first case is when removing explicit type information from the source code makes it less readable.

Of course, IDEs can help in this regard, but during code-reviews or when you just quickly scanning the code it might hurt readability. For example, consider factories or builders: you have to find the code responsible for object initialization to determine the type.

Here's a little puzzle. The following piece of code is using Java 8's Date/Time API. Guess the types of the variables in the following snippet:

```
var date = LocalDate.parse("2019-08-13");
var dayOfWeek = date.getDayOfWeek();
var dayOfMonth = date.getDayOfMonth();
```

Done? Here's the solution:

The first one is pretty intuitive, the parse method returns a LocalDate object. However, for the next two, you should be a little bit more familiar with the API: dayOfWeek returns a java.time.DayOfWeek, while dayOfMonth simply returns an int.

Another potential problem is that with var the reader has to rely more on the context. Consider the following:

```
private void longerMethod() {
    // ...
    // ...
    // ...

    var dayOfWeek = date.getDayOfWeek();

    // ...
    // ...
    // ...
}
```

Based on the previous example, I bet you'd guess it's a java.time.DayOfWeek. But this time, it's an integer, because the date in this example is from Joda time. It's a different API, behaving slightly differently, but you can't see it because it's a longer method, and you did not read all the lines. (JavaDoc: Joda time / Java 8 Date/Time API)

If the explicit type declaration was present, figuring out what type dayOfWeek has would be trivial. Now, with var, the reader first has to find out the type of the date variable and check what getDayOfWeek does. This is simple with an IDE, not so simple when just scanning the code.

⚠️ Tip: Pay attention to preserve important type information

The second case is when using var removes all available type information, so it can not be even inferred. In most cases, these situations are caught by the Java compiler. For example, var cannot infer type for lambdas or method references, because for these features the compiler relies on the left-hand side expression to figure out the types.

However, there are a few exceptions. For example, var does not play nicely with the Diamond Operator. The Diamond operator is a nice feature to remove some

verbosity from the right-hand side of an expression when creating a generic instance:

```
Map<String, String> myMap = new HashMap<String, String>(); // Pre Java 7
Map<String, String> myMap = new HashMap<>(); // Using Diamond operator
```

Because it only deals with the generic types, there is still redundancy to be removed. Let's try to make it terser with var:

```
var myMap = new HashMap<>();
```

This example is not only valid, but it does not even emit a compiler warning. However, with all these type inference features we ended up not specifying the generic types at all, and the type will be Map<Object, Object>.

Of course, this can be solved easily by removing the Diamond Operator:

```
var myMap = new HashMap<String, String>();
```

Another set of problems can arise when var is used with primitive data types:

```
byte   b = 1;
short  s = 1;
int    i = 1;
long   l = 1;
float  f = 1;
double d = 1;
```

Without explicit type declaration, the type of all these variables would be inferred to int. Use type literals (e.g. 1L) when working with primitive data types, or don't use var in this case at all.

⚠️ Tip: Make sure to read the official style guides

It's ultimately up to you to decide when to use type inference and make sure that it does not hurt readability and correctness. As a rule of thumb, sticking to good programming practices, such as good naming and minimizing the scope of local variables certainly helps a lot. Make sure to read the official style guide and FAQ about var.

Because var has so many gotchas, it's great that it was introduced conservatively and can only be used on local variables, which scope is usually pretty limited.

Also, it has been introduced cautiously, **var is not a new keyword but a reserved type name**. This means that it only has special meaning when it's used as a type name, everywhere else var is continuing to be a valid identifier.

Currently, var does not have an immutable counterpart (such as val or const) to declare a final variable and infer its type with a single keyword. Hopefully, we'll get it in a future release, until then, we can resort to final var.

Resources:

- First Contact With 'var' In Java 10
- 26 Items for Dissecting Java Local Variable Type Inference (Var Type)
- Java 10: Local Variable Type Inference

Allow private methods in interfaces

Available since: JDK 9 (Milling Project Coin)

Since Java 8 it is possible to add default methods to interfaces. With Java 9 these default methods can even call private methods to share code in case you are in a need for reuse, but do not want to expose functionality publicly.

Although it's not a huge deal, it's a logical addition that allows to tidy up code in default methods.

Diamond operator for anonymous inner classes

Available since: JDK 9 (Milling Project Coin)

Java 7 introduced the Diamond Operator (<>) to reduce verbosity by letting the compiler infer the parameter types for constructors:

```
List<Integer> numbers = new ArrayList<>();
```

However, this feature did not work with anonymous inner classes before. According to the discussion on the project's mailing list this was not added as part of the