0-language-and-features

Contents

Release

In the following document are reflected the most recent java releases, Java 19 - Released on September 15, 2022 Java 20 - Released on March 21, 2023 Java 21 - Released on September 12, 2023. These dates reflect the typical six-month release cadence that Oracle has established for Java, with feature releases occurring in March and September of each year.

Language

Patterns

There are several improvements in the pattern matching feature of the `instanceOf` keyword, for records. It is now possible to deconstruct and reference the fields of a record in a logical/conditional block which uses instanceOf operation. This gives the ability to access record's fields without needing to first reference the instance variable itself, collisions are not handled (meaning that two records which have the same fields can not be deconstructed, it will yield compiler error)

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

public static void main(String[] args) {
        Point point = new Point(10, 20);
        if (point instanceof Point(int x, int y)) {
            // The fields of the record are accessed without having to use
               the instance variable itself, note that if
            // two Records have the same field names, this might produce a
               compiler error
            System.out.println("X: " + x + ", Y: " + y);
        }
    }
```

Switch

The switch statement match is also extended to match based on type patterns and conditional expressions. This however is not only applicable for `record` types, but it is very useful, since it avoids having different types

```
public static String formatShape(Object shape) {
    return switch (shape) {
        case Circle(double radius) -> "Circle with radius " + radius;
        case Rectangle(double length, double width) -> "Rectangle with length
            " + length + " and width " + width;
        case Triangle(double base, double height) -> "Triangle with base " +
           base + " and height " + height;
        default -> "Unknown shape";
    };
}

public record Circle(double radius) {}
public record Rectangle(double length, double width) {}
public record Triangle(double base, double height) {}
```

The extended form of the switch expression matching also shown below, allows one to not only match on the type pattern, but also on the value of a property, in this case the radius of the Circle, when positive, the returned `string` message is different than the default `circle` type pattern match. Obviously `switch` expressions are still evaluated top-down, meaning that the specialized cases must come before the general ones

```
public static String describeShape(Object shape) {
    return switch (shape) {
        case Circle(double radius) when radius > 0 -> "Circle with positive
           radius: " + radius;
        case Circle(double radius) -> "Circle with radius: " + radius;
        case Rectangle(double length, double width) -> "Rectangle of " +
           length + " x " + width;
        default -> "Unknown shape";
    };
}
```

The example below shows how the type pattern matching can be applied to a regular class hierarchy structure, instead of a record, however the class hierarchy has to have a well defined structure, and no ambiguity, otherwise compile time error will occur. The when keyword is also applicable for type pattern class matching as well, allowing for a more refined checks and bounds

```java
public abstract class Shape {}

public class Circle extends Shape {
    public final double radius;
    public Circle(double radius) { this.radius = radius; }
}

public class Rectangle extends Shape {
    public final double length;
    public final double width;
    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
}

public static String describeShape(Shape shape) {
    return switch (shape) {
        case Circle c when c.radius > 0 -> "Circle with positive radius: " +
           c.radius;
        case Rectangle r when r.length == r.width -> "Square with side: " + r
           .length;
        case Circle c -> "Circle with radius: " + c.radius;
        case Rectangle r -> "Rectangle with dimensions: " + r.length + " x "
           + r.width;
        default -> "Unknown shape";
    };
}
```

If you think about that, the example above is really two nested switch statements, which are flattened within a single one, the first top level switch matches on the type, the old java 8 traditional way would be to match on a property called `type` then in the nested switch statement one can match on the `properties` of the instance (either the circle's radius or the rectangle's side length etc), the new syntax simply flattens this multi level switch representation into a single level statement

The multi-level equivalent of the switch expression block above will look something like that.

```java
public static String describeShape(Shape shape) {
    switch (shape.getClass().getSimpleName()) {  // Outer switch on type
        case "Circle" -> {
            Circle c = (Circle) shape;  // Manual casting required
            switch (c.radius > 0 ? "positive" : "non-positive") {  // Inner
               switch on radius condition
                case "positive" -> {
                    return "Circle with positive radius: " + c.radius;
                }
                default -> {
                    return "Circle with radius: " + c.radius;
                }
            }
        }
        case "Rectangle" -> {
            Rectangle r = (Rectangle) shape;  // Manual casting required
            switch (r.length == r.width ? "square" : "rectangle") {  // Inner
                switch for square vs rectangle
                case "square" -> {
                    return "Square with side: " + r.length;
                }
                default -> {
                    return "Rectangle with dimensions: " + r.length + " x " +
                        r.width;
                }
            }
        }
        default -> {
            return "Unknown shape";
        }
    }
}
```

The new pattern matching for switch in Java effectively flattens what would traditionally require a multi-level switch or nested if-else structure. This flattening makes it much easier to write, read, and maintain branching logic that involves both type-checking and value-based conditions.

Templates

While still preview feature in Java 21, this is a general implementation in the `jdk`, which provides alternative of `String.format` method to support common template syntax such as `${varable}` that is done to avoid the usual concatenation which can occur if one needs to embed variables within a string. The example below demonstrates how the Traditional way is replaced by the new approach which uses the variables which are correctly parsed and bound into the string template after calling. Any variable, method or field accessible to the scope of the string template at the time of definition is allowed to be used within a templated string.

Note that any runtime value can be evaluated within the expression, but the returned value will not, never be treated as a template, meaning that there is no way for external variables to randomly inject dangerous template within another template, every value that is within the template will be evaluated as literal string, even if it contains template keywords/symbols, this prevents the misuse of templates for by bad actors.

```
int age = 30;
String name = "Alice";

// Traditional way
String message = "Hello, " + name + "! You are " + age + " years old.";

// Using string templates (Java 21)
String messageTemplate = STR."Hello, ${name}! You are ${age} years old.";

System.out.println(messageTemplate);  // Output: Hello, Alice! You are 30
   years old.
```

The new STR. syntax allows the string to reference variables from the local context, in the example above the String variables name and age are referenced within the template,

```
this is a syntax sugar, which gets compiled to a String.format type of expression but it
provides an easy way to construct strings with embeded varialbes.
```

Platform

There are several small improvements on the platform level, listed below, the biggest one of which is the introduction of lightweight platform thread alternative - virtual threads. Which provide seamless integration with existing code, with very few small adjustments. Mostly using different implementations of the `ExecutorService` which spawns and works with virtual threads instead of the native platform ones

Virtual Threads

One of the most major additions to the language in general is the new virtual thread model, which provides means of creating very lightweight fast threads, which are much less resource intensive than the regular platform OS level threads, which have been the de-facto standard in java thus far. The new virtual threads, work and are manged by the virtual machine (`JVM`) itself. How does that work ? Instead of creating dedicated platform threads, the `JVM` instead creates internal threads which it manages, the `JVM` acts in a way like an operating system scheduler for each virtual thread. The virtual threads themselves are executed on actual platform OS threads, however the `JVM` has much more control over the virtual threads, allowing it to run multiple virtual threads on a single platform thread, which was thus far not possible. Each virtual thread is run for a specific amount of time and it can yield control over to another virtual thread running on the same platform thread in case the first virtual thread blocks for I/O input or takes too much time, the `JVM` distributes execution time to all virtual threads running on the same platform thread, scheduling them (just as the operating system would schedule platform threads). This allows the `JVM` to create much less platform threads, and many virtual threads, to run on these platform threads. Effectively introducing a 3rd level of threading.

- `CPU multi core threading` - true multi threading
- `platform/os system threading` - operating system schedules tasks on per CPU core level
- `java virtual threads` - the java virtual machine schedules tasks on platform/system threads

The example below demonstrates how the existing API, can be used to create the familiar `ExecutorService` implementations already mentioned, with a new parameter providing a different factory for the internal threads, since the new virtual threading model is, at least on a user level, indistinguishable from the platform threads, that allows for no friction in the use of virtual threads

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class FixedThreadPoolWithVirtualThreads {
    public static void main(String[] args) {
        // The new executor factory methods are extended with a factory
           parameter allowing one to specify the type of
        // thread to create, in this case by default the executors would
           default to platform threads, unless specified
        // otherwise, below virtual thread factory is provided instead
        ExecutorService executor = Executors.newFixedThreadPool(4, Thread.
           ofVirtual().factory());

        for (int i = 0; i < 10; i++) {
            int taskId = i;
            // Submit task to the pool, the submit would immediately trigger
               the task, and start its execution
            executor.submit(() -> {
                System.out.println("Running task " + taskId + " in virtual
                   thread: " + Thread.currentThread().getName());
                try {
                    // Simulating some blocking work, the java virtual
                       machine would yield control to over virtual
                    // threads when a specific thread starts blocking, to
                       evenly distribute and schedule all currently
                    // requested concurrent tasks
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });
        }

        // Shutdown will make sure that all the pending tasks are finished,
           however no new tasks are allowed to be
        // accepted/submitted
        executor.shutdown();
    }
}
```

FFM

With the latest releases Java is aiming at deprecating the old `JNI` and `RMI` interfaces, in favor of more modern approach which is the Foreign Function & Memory interface. This is also known as `FFI` - foreign function interface in other languages, allowing one to interact with functions implemented in other languages - primarily C and C++.

Having the following example below, which demonstrates how one can call and manage memory from Java into an external static or dynamic library written in c

```
// defined in the .h file
typedef struct {
    int age;
    char name[20];
} Person;

void print_person(Person* p);

// defined in the .c file
#include <stdio.h>
#include "person.h"

void print_person(Person* p) {
    printf("Name: %s, Age: %d\n", p->name, p->age);
}
```

Compile the code above into a static library, that has to be located on the classpath when the java application is invoked

```
gcc -shared -o libperson.so -fPIC person.c
```

The java application which interacts with the static library built above. The implementation manages the memory, creating the data required for the `struct` to be populated. Note that the `struct` memory layout matches exactly the one defined in the C API, it is also allocated on the heap. The address of it obtained and passed to the method to invoke it. It is important to note that Java itself does not manage dynamic structures when interacting with FFM - meaning that the dynamic structures (such as char arrays or strings) have to be appropriately sized, otherwise all kinds of overflow issues will occur, similarly to how they would if using plain C, there are some guardrails put in place, but in general the FFM itself does not really protect against such scenarios, it just provides means of interacting with external libraries written in other languages.

```java
import jdk.incubator.foreign.*;
import static jdk.incubator.foreign.CLinker.*;
import static jdk.incubator.foreign.MemoryLayout.*;
import static jdk.incubator.foreign.MemorySegment.*;

public class StructExample {
    public static void main(String[] args) {
        // Define the layout of the Person struct
        MemoryLayout personLayout = MemoryLayout.structLayout(
            ValueLayout.JAVA_INT.withName("age"),
            MemoryLayout.ofArray(ValueLayout.JAVA_BYTE.withName("name").
               withSize(20))
        );

        try (MemorySession session = MemorySession.openConfined()) {
            // Allocate memory for the struct
            MemorySegment personSegment = session.allocate(personLayout);

            // Set values
            personSegment.set(ValueLayout.JAVA_INT, 0, 30);
            personSegment.set(MemoryLayout.ofArray(ValueLayout.JAVA_BYTE.
               withName("name").withSize(20)), 4, "Alice".getBytes());

            // Call the native function to print the person
            LibraryLookup library = LibraryLookup.ofLibrary("person");
            SymbolLookup lookup = library.lookup("print_person").orElseThrow
               ();
            VarHandle printPersonHandle = lookup.get("print_person").get();
            printPersonHandle.invoke(personSegment.address());
        }
    }
}
```