0-language-and-features

Contents

- Release
  - Language
    - Switch
    - Text-blocks
    - Instanceof
    - Records
    - Sealed
    - Floating
    - SIMD
    - Memory
  - Library
    - String
    - Files
  - Platform
    - Packaging
    - Messaging
    - Encapsulation

Release

Generally there are several improvements in the algorithms for garbage collection, as well as new types of garbage collection algorithms which were introduced in the newer versions between java 12 and 17.

Language

Switch

With java 17, the new improved switch statement was standardized. The new switch expression syntax allows one to define a switch statement like a lambda expression. This allows the switch statement to be used to obtain a result, in other words give a switch statement a return value.

```
int day = 4;
String dayType = switch (day) {
    case 1, 2, 3, 4, 5 -> "Weekday";
    case 6, 7 -> "Weekend";
    default -> throw new IllegalArgumentException("Invalid day: " + day);
};
```

Text-blocks

Before it was not possible to define multi-line text strings or blocks, unless they were concatenated from multiple lines using the `+` sign, with java 17 the new syntax using `"""` was added which allows one to define continuous text or string blocks

```
String json = """
    {
        "name": "John Doe",
        "age": 30
    }
    """;
```

Instanceof

Since java 16, the `instanceof` keyword has been improved to support pattern matching. It is very common in code to express a conditional statement which checks if a given object is an instance of another, and in the body of the condition a cast is performed, since java 16, that cast is no longer needed, the compiler will correctly interpret the `instanceof` check in the enclosing block and auto cast the object to the type being checked against

```
Object obj = "Hello, World!";
if (obj instanceof String s) {
    // a cast for obj to String is not required to use
    System.out.println("String length: " + s.length());
}
```

Records

The record is a way to improve the verbose creation of simple `immutable` data transfer objects, which are very common in the java language, these types usually provide transparent/pass-through setter and `getter` methods, and constructors, along with generic equals and `hashCode` and `toString`, that is why the record keyword allows one to create very simple `DTOs`.

```
public record Person(String name, int age) {}

Person person = new Person("Alice", 25);
System.out.println(person); // Person[name=Alice, age=25]
System.out.println(person.name); // this member access is valid, prints the
   name
System.out.println(person.name()); // this is also valid, prints the name as
   well
```

All fields or members in the Record are effectively final, meaning that they can not be modified, once declared, or assigned in the constructor, each member generates a method in the form `member()` there is not `get` prefix for the methods, the member `accessor` can also be used instead of the `getter method`. Records provide an implicit canonical constructor (a constructor that initializes all fields in the header in order). But it is possible for one to manually fill up the constructor, it is also important to note that the Record can have only one constructor

Additional methods may be added to the record, those however must not, they can not modify the state of the record. It is also possible to define a custom constructor, which must however delegate to the default constructor

```
public record Circle(double radius) {
    public double area() {
        return Math.PI * radius * radius;
    }
}
```

It is important to note that the constructor can be re-defined, but no members can be assigned manually, these will be delegated to the default constructor generated by the run-time, the custom constructor can possibly only do validation on the input arguments or parameters. Attempting to assign to the members of the record will produce a `compiler error`

```
public record Person(String name, int age) {}
    // This is not valid syntax, members can not be assigned externally, only
        the implicit constructor
    public Person {
        this.name = name; // this will produce a compiler error
        this.age = age * 2; // this will produce a compiler error
    }

    // This however is allowed, the members assignment will be delegated to
       the default constructor
    public Person {
        if (age > 18) {
            throw new IllegalStateException("Age must not be below 18")
        }
    }
}
```

A final example which demonstrates an extended constructor and custom methods for a record can be seen below.

```
public record Product(String name, double price, int quantity) {
    // Custom constructor with validation, members assigned automatically
       from the default constructor provided by the run-time
```

```
    public Product {
        if (price < 0) {
            throw new IllegalArgumentException("Price cannot be negative");
        }
        if (quantity < 0) {
            throw new IllegalArgumentException("Quantity cannot be negative")
               ;
        }
    }

    // Additional method for computing total cost, can not and does not
       modify, the state of the record instance
    public double totalCost() {
        return price * quantity;
    }
}

Product apple = new Product("Apple", 0.50, 10); // Create a product record,
   with given parameters
System.out.println(apple);                      // Product[name=Apple, price
   =0.5, quantity=10]
System.out.println(apple.totalCost());          // Output: 5.0
```

Sealed

Sealed classes in java are those classes which specialize which sub-classes can extend or sub-class from it, the new keyword `permits` is used to implement this behavior

```
public sealed class Shape permits Circle, Square {} // Specify which classes
   can extend Shape

public final class Circle extends Shape {} // The Circle is allowed to extend
    from Shape
public final class Square extends Shape {} // The Square is allowed to extend
    from Shape
public final class Sphere extends Shape {} // This will produce a compile
   time error
```

The main purpose of sealed classes is to enforce a well-defined hierarchy. By requiring that all permitted `subclasses` are known to the `superclass`, the class hierarchy becomes fully defined and cannot be extended unpredictably.

Floating

New hexadecimal support for floating numbers was added, allowing one two specify a floating hex number as a literal like so, shown below

```
double hexValue = 0x1.0p-3; // Equivalent to 1 / 8 = 0.125
```

SIMD

New `vectorization` API provides access to intrinsic and `SIMD` vector instructions for supported architectures. The new elements in the standard java libraries allow one to access system `intrinsics` directly from the

language instead of having to do some sort of RMI or FFI access to underlying C libraries.

```
// allocate some input and output structures to store the input and the
   result of the operations to be executed.
int[] arrayA = {1, 2, 3, 4, 5, 6, 7, 8};
int[] arrayB = {8, 7, 6, 5, 4, 3, 2, 1};
int[] result = new int[arrayA.length];

// Define the species (shape) for a 256-bit IntVector
VectorSpecies<Integer> SPECIES = IntVector.SPECIES_256;

int i = 0;
// Process in chunks of 256 bits (8 lanes for 32-bit integers)
// The loop below will effectively loop over 8 elements at a time, in this
   example just a single loop iteration will be
// executed, the offset per iteration is +8, The data from the input arrayA
   and arrayB will be loaded into an int vector
// which are then going to be added together into the result vector, and the
   result copied back to the result array.
// This is a common pattern which is usually performed and used when dealing
   with SIMD intrinsics
for (; i < SPECIES.loopBound(arrayA.length); i += SPECIES.length()) {
    // Load vectors from the arrays, from java land
    IntVector va = IntVector.fromArray(SPECIES, arrayA, i);
    IntVector vb = IntVector.fromArray(SPECIES, arrayB, i);

    // Perform a vectorized addition
    IntVector vc = va.add(vb);

    // Copy the result back into java land
    vc.intoArray(result, i);
}

// Print the result stored in the array
for (int value : result) {
    System.out.print(value + " ");
}
```

Memory

Allows for access / linking against foreign native API for example from the C run-time. Allows for manipulating the native memory layout

MemoryLayout can be used to describe the contents of a memory segment in a language neutral fashion. There are two leaves in the layout hierarchy a value layout which are used to represent value of a given size and kind, and padding layout which are used as the name suggests to represent a portion of a memory segment whose contents should be ignored and which are primarily present or alignment reasons.

In the example below it is shown how one can create a simple data structure layout, which is the equivalent of the same in C

```
struct {
    uint8_t  kind;
```

```
    uint32_t padding1;
    uint32_t padding2;
};
```

```
// The memory layout below constructs a native struct which contains as the
   first member, with a size in bits - 8, and
// the bytes order which is either going to be BIG_ENDIAN or LITTLE_ENDIAN.
   Then additional padding is defined in this
// case 2x32 bits padding entries
MemoryLayout taggedValuesWithHole = MemoryLayout.sequenceLayout(5,
    MemoryLayout.structLayout(
        MemoryLayout.valueLayout(8, ByteOrder.nativeOrder()).withName("kind")
           ,
        MemoryLayout.paddingLayout(32),
        MemoryLayout.paddingLayout(32)
));
```

There is a simple example below, which demonstrates how a simple `struct` can be created with the usage of the `MemoryLayout` and the `ValueLayout` - which is a helper type which provides common value layouts such as integer, float, double, and so on. The `struct` below contains two integer fields - x, y which represent some sort of 2 dimensional point in space. Then the memory layout is allocated in a try-with-resources block since, the actual memory that is produced is not within the scope or control of the JVM, which means that the caller has the responsibility to free up this memory, this is done by putting it in a try with resource block, which will make sure to free up the memory after use, the structure is then initialized with values, and the values are read back

```
// Define a struct layout for two 32-bit integers
MemoryLayout pointLayout = MemoryLayout.structLayout(
    ValueLayout.JAVA_INT.withName("x"),
    ValueLayout.JAVA_INT.withName("y")
);

try (MemorySegment segment = MemorySegment.allocateNative(pointLayout)) {
    // Write values to the fields
    MemoryAccess.setIntAtOffset(segment, pointLayout.byteOffset(PathElement.
       groupElement("x")), 10);
    MemoryAccess.setIntAtOffset(segment, pointLayout.byteOffset(PathElement.
       groupElement("y")), 20);

    // Execute work with the contents of the point structure, writing or
       updating them

    // Read values from the fields
    int x = MemoryAccess.getIntAtOffset(segment, pointLayout.byteOffset(
       PathElement.groupElement("x")));
    int y = MemoryAccess.getIntAtOffset(segment, pointLayout.byteOffset(
       PathElement.groupElement("y")));

    System.out.println("Point coordinates: (" + x + ", " + y + ")");
}
```

Library

Several small improvements to the base java package library API were done, demonstrated below, mostly for convenience the String and other commonly used classes were extended with new static utility methods to facilitate better interop with the language, avoiding reliance on external 3rd party libraries and utility packages.

String

There are several new methods added to the String type, which improve the readability and usability of String instances, without the need of third party libraries and utilities. New methods such as String::lines, String::repeat. The new lines method returns a `Stream<String>`. The `repeat` method on the other hand repeats the current contents of the String instance, N number/count time

```
String text = "Hello\nWorld";
text.lines().forEach(System.out::println); // Prints each line separately
```

```
String text = "abc";
System.out::println(text.repeat(3)); // Prints abc 3 times, - "abcabcabc"
```

Files

A new method to the files class, which returns the first byte position where the two files have a mismatch, effectively the new method compares two files, and finds where the two files have a mismatch, the signature of the method is like follows, shown below

```
/* Finds and returns the position of the first mismatched byte in the content
* of two files, or {@code -1L} if there is no mismatch. The position will be
* in the inclusive range of {@code 0L} up to the size (in bytes) of the
* smaller file. */
public static long mismatch(Path path, Path path2) throws IOException
```

Platform

Packaging

A new `jpackage` CLI tool to simplify the packaging of native applications for different platforms, it outputs an executable installer. It can produce `exe` `dmg` or `deb/rpm` packages for the different operating systems. The installer can also be specified with an icon, signing authority (important for macos) and other properties.

Messaging

Several improvements to the native run-time exceptions like `NullPointerException`, to provide more fine grained details and context information for the stack traces. The new messages include more descriptive error messages for when a `null` value is attempted to be read or accessed.

Encapsulation

Access to internal libraries and `APIs` is greatly restricted, to improve security and stability. Forces libraries and applications to rely on public, stable `API` improving compatibility.