HowToDoInJava

Java 21 Features (LTS): Practical Examples and Insights

Lokesh Gupta

January 21, 2024

Java

Java 21, Java Features

Java 21 is finally released on *19-Sep-2023* as the next long-term support (LTS) release of Oracle's standard Java implementation. In this comprehensive guide, we will delve into the new and preview features that Java has offered in this latest version. Check out this guide for other versions of Java and included features.

1. Main Highlights of Java 21

Among other changes, the following are the main changes that we should be aware of as a Java developer.

- Virtual Threads have been finalized.
- The Record Patterns (Project Amber) have been finalized.
- Pattern Matching for *switch* statements has been finalized.
- A new collection, *SequencedCollection*, has been added that provides direct access to an ordered collection's first and last elements.
- String templates and unnamed classes & instance main() methods are available as preview features.

x

- Unnamed Classes and Instance Main Methods are introduced as a preview feature.
- Structured Concurrency continues to be a preview feature.

The other included changes are:

- Generational ZGC [JEP-439]
- Pattern Matching for switch [JEP-441]
- Foreign Function & Memory API (*Third Preview*) [JEP-442]
- Vector API (*Sixth Incubator*) [JEP-448]
- Deprecate the Windows 32-bit x86 Port for Removal [JEP-449]
- Prepare to Disallow the Dynamic Loading of Agents [JEP-451]
- Key Encapsulation Mechanism API [JEP-452]

Let us discuss the important changes in more detail.

2. Virtual Threads (Project Loom)

The virtual threads are JVM-managed lightweight threads that will help in writing high-throughput concurrent applications (throughput means how many units of information a system can process in a given amount of time). [JEP-425, JEP-436 and JEP-444] In Java 21, virtual threads are ready for production use.

With the introduction of virtual threads, it becomes possible to execute millions of virtual threads using only a few operating system threads. The most advantageous aspect is that there is no need to modify existing Java code. All that is required is instructing our application framework to utilize virtual threads in place of platform threads.

To create a virtual thread, using the Java APIs, we can use the *Thread* or *Executors*.

```
//2
Thread virtualThread = Thread.ofVirtual().start(runnable);

//3
var executor = Executors.newVirtualThreadPerTaskExecutor();
executor.submit(runnable);
```

Please note that virtual threads are not faster than platform threads. They should be used to scale the number of concurrent tasks that spend much of their time waiting. For example, server applications that handle many client requests and perform blocking I/O operations. For resource/processing-intensive tasks, continue using the traditional platform threads, as virtual threads will not provide any advantage.

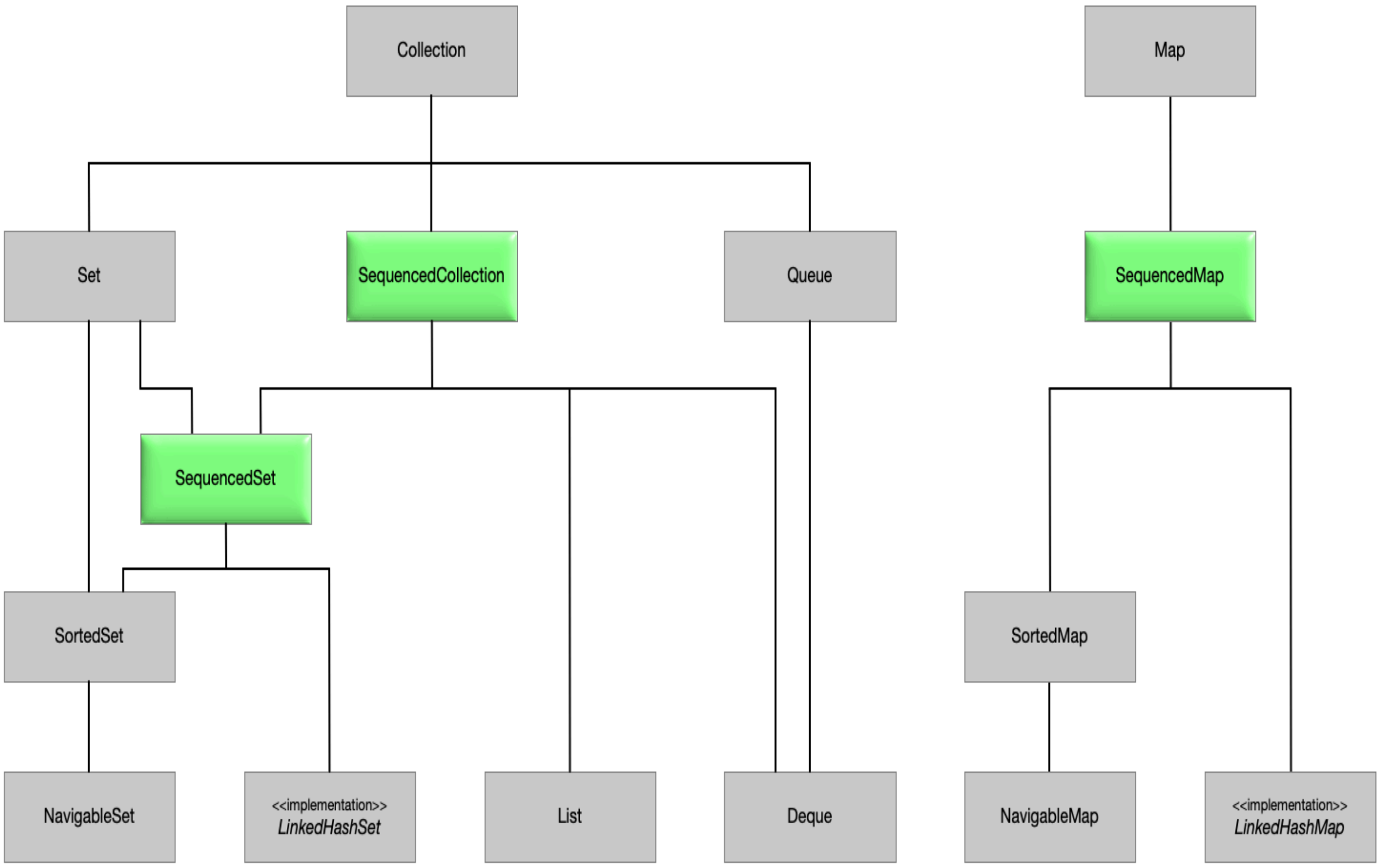
Also, beware that more threads mean more dependent system resources, and these resources may not scale in proportion. To prevent resource exhaustion and ensure optimal system utilization, we must run tests by limiting the number of concurrent threads using mechanisms such as *Semaphore*.

3. Sequenced Collections

The new interfaces created under the sequenced collections initiative represent **collections with a defined encounter order**. The order will have a well-defined first element, second element, and so forth, up to the last element. The newly added interfaces provide a uniform API to access these elements in a sequence, or in the reverse order.

All popular and commonly used collection classes now implement the *java.util.SequencedCollection*, *java.util.SequencedSet* or *java.util.SequencedMap* interfaces as well, according to the corresponding collection type.

X



The new interfaces have additional methods to support sequential access to the elements. For example, *SequencedCollection* has the following methods:

```java
interface SequencedCollection<E> extends Collection<E> {

  // new method
  SequencedCollection<E> reversed();

  // methods promoted from Deque
  void addFirst(E);
  void addLast(E);
  E getFirst();
  E getLast();
  E removeFirst();
  E removeLast();
}
```

Note that any modifications to the original collection are visible in the reversed collection view. Let us understand with a Java program how we can use the new sequenced *ArrayList*:

x

```
ArrayList<Integer> arrayList = new ArrayList<>();

arrayList.add(1);               // [1]

arrayList.addFirst(0);  // [0, 1]
arrayList.addLast(2);       // [0, 1, 2]

arrayList.getFirst();   // 0
arrayList.getLast();    // 2
```

To understand the benefits, see how these simple operations were too much verbose in Java 17.

```
arrayList.get( arrayList.iterator().next() ); // first element
arrayList.get( arrayList.size() - 1 ); // last element
```

4. Record Patterns

Records in Java are transparent and immutable carriers for data (similar to POJO). We create a *record* as follows:

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

Previously, if we need to access the components of a *record*, we should destructure it as follows:

```
Point obj = new Point(1,2);

if (obj instanceof Point p) {

  int x = p.x();
  int y = p.y();
```

With Java 21, we can rewrite it in a less verbose manner using `Point(int x, int y)` syntax that is called record pattern.

The record patterns eliminate the declaration of local variables for extracted components and initialize the components by invoking the accessor methods when a value is matched against the pattern.

```
if (obj instanceof Point(int x, int y)) {

    System.out.println(x+y);
}
```

5. Pattern Matching for *switch*

Since Java 21, we can use the record patterns with the `switch` statements. Note that the *switch* block must have clauses that deal with all possible values of the selector expression.

For example, in Java 16, we could have done something like this:

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

public void print(Object o) {

  switch (o) {

    case Point p        -> System.out.printf("o is a position: %d/%d%n", p.x(),
    case String s   -> System.out.printf("o is a string: %s%n", s);
    default         -> System.out.printf("o is something else: %s%n", o);
  }
}
```

In Java 21, we can write the similar expression with *record* pattern as follows:

```
public void print(Object o) {
```

x

```
      case String s                -> System.out.printf("o is a string: %s%n", s);
      default                      -> System.out.printf("o is something else: %s%n"
    }
  }
```

6. String Templates (Preview)

Using the string templates, we can create string templates containing embedded expressions (evaluated at runtime). The template strings can contain variables, methods or fields, computed at run time, to produce a formatted string as output.

Syntactically, a template expression resembles a string literal with a prefix.

```
let message = `Greetings ${ name }!`;           //TypeScript

String message = STR."Greetings \{ name }!";  //Java
```

In the above template expression:

- STR is the template processor.
- There is a dot operator `(.)` between the processor and the expression.
- Template string with embedded expression. The expression is in the form of `(\{name})`.

Note that the result of the template processor, and thus the result of evaluating the template expression, is often a String — though not necessarily always.

7. Unnamed Patterns and Variables (Preview)

It is common in some other programming languages (such as Scala and Python) that we can skip naming a variable that we will not use in the future. Now, since Java 21, we can use

x

```
String s = ...;

try {
    int i = Integer.parseInt(s);
    //use i
} catch (NumberFormatException ex) {
    System.out.println("Invalid number: " + s);
}
```

Notice we have created the variable ex but we used it nowhere. In the above example, the variable is unused and its name is irrelevant. The unnamed variables feature allows us to skip naming the variable and simply use an underscore (_) in place of it. Here **underscore signifies the absence of a name**.

```
String s = ...;

try {
    int i = Integer.parseInt(s);
    //use i
} catch (NumberFormatException _) {
    System.out.println("Invalid number: " + s);
}
```

Similarly, we can use the unnamed variables in the `switch` expressions also:

```
Object obj = 1; // Use Object type to accommodate different types

String result = switch (obj) {

  case Byte _, Short _, Integer _, Long _ -> "Input is a Number";
  case Float _, Double _ -> "Input is a floating-point number";
  case String _ -> "Input is a string";
  default -> "Object type not expected";
};

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

X

the unnamed variables, thus resulting in **unnamed patterns**.

In the following example, we are not using the value of y so we could simply declare it as an unnamed variable.

```
public void print(Object o) {

  switch (o) {

    case Point(int x, int _)        -> System.out.printf("The x position is : %d
    //...
  }
}
```

8. Unnamed Classes and Instance Main Methods (Preview)

> This is preview language feature, disabled by default. To use it, we must enable preview features with `--enable-preview` flag.

In Java, unnamed modules and packages are a familiar concept. When we do not create a *module-info.java* class then the module is automatically assumed by the compiler. Similarly, if we do not add the package statement in the class in the root directory, the class just compiles and runs fine.

Same way, we can now create unnamed classes. Quite obviously, **an unnamed class is a class without a name**.

Consider the following class declaration that we generally create to test code snippets or a simple concept.

```
public class TestAConcept {

  public static void main(String[] args) {
```

x

```
  static String method() {

    //...
  }
}
```

From Java 21, we can write the above class without the class declaration as follows. It removes the `class` declaration, `public` and `static` access modifiers etc. to have a cleaner class.

```
void main(String[] args) {

  System.out.println(method());
}

String method() {

    //...
}
```

In Java 21, we can the above class using the command. Note that we have saved the class in a file *TestAConcept.java*.

```
$ java --enable-preview --source 21 TestAConcept.java
```

It is important to note that similar to the code in named packages within a named module cannot directly access code in unnamed packages or the unnamed module, **code from named classes cannot access unnamed classes**.

9. Scoped Values (Preview)

If you are familiar with *ThreadLocal* variables, the **scoped values are a modern way of**

x

Scoped values are usually created as `public static` fields so we can access them directly without passing them as a parameter to any method. However, it is important to understand that if the value is checked in multiple methods, the current value will depend on the execution time and state of the thread. The value may change over time when accessed over time in different methods.

To create scoped values, use the *ScopedValue.newInstance()* factory method.

```
public final static ScopedValue<USER> LOGGED_IN_USER = ScopedValue.newInstance()
```

With `ScopedValue.where()`, we bind the scoped value to the object instance; and then we run a method, for whose call duration the scoped value should be valid. Note that a scoped value is written once and is then immutable, thus nobody can change the *loggedInUser* in the invoked method.

```
class LoginUtil {

    public final static ScopedValue<USER> LOGGED_IN_USER = ScopedValue.newInstan

    //Inside some method
    User loggedInUser = authenticateUser(request);
    ScopedValue.where(LOGGED_IN_USER, loggedInUser).run(() -> service.getData())
}
```

Inside the invoked thread, we can directly access the scoped value:

```
public void getData() {

    User loggedInUser = LoginUtil.LOGGED_IN_USER.get();
    //use loggedInUser
}
```

X

The structured concurrency feature aims to simplify Java concurrent programs by treating multiple tasks running in different threads (forked from the same parent thread) as a single unit of work. Treating all such child threads as a single unit will help in managing all threads as a unit; thus, canceling and error handling can be done more reliably.

In structured multi-threaded code, if a task splits into concurrent subtasks, they all return to the same place i.e., the task's code block. This way, the lifetime of a concurrent subtask is confined to that syntactic block.

In this approach, subtasks work on behalf of a task that awaits their results and monitors them for failures. At run time, structured concurrency builds a tree-shaped hierarchy of tasks, with sibling subtasks being owned by the same parent task. This tree can be viewed as the concurrent counterpart to the call stack of a single thread with multiple method calls.

```
try (var scope = new StructuredTaskScope.ShutdownOnFailure()()) {

    Future<AccountDetails> accountDetailsFuture = scope.fork(() -> getAccountDet
    Future<LinkedAccounts> linkedAccountsFuture = scope.fork(() -> fetchLinkedAc
    Future<DemographicData> userDetailsFuture = scope.fork(() -> fetchUserDetail

    scope.join();   // Join all subtasks
    scope.throwIfFailed(e -> new WebApplicationException(e));

    //The subtasks have completed by now so process the result
    return new Response(accountDetailsFuture.resultNow(),
            linkedAccountsFuture.resultNow(),
            userDetailsFuture.resultNow());
}
```

11. Conclusion

In article discussed the main **developer features in Java 21** that we should be aware of and learn over time. Java 21 has finalized a few features, such as record patterns and virtual threads, while other features are still in preview mode.

x

Happy Learning !!

Source Code on Github

Further Reading:

- **Java Concurrency Interview Questions**
- **Java Virtual Threads: Project Loom**
- **Java 21 Scoped Values: A Deep Dive with Examples**
- **Capture and Analyze a Thread Dump in Java**
- **Java ArrayList: A Comprehensive Guide for Beginners**
- **Java 21 Unnamed Patterns and Variables (with Examples)**

Comments

Subscribe

Join the discussion

8 COMMENTS Most Voted

MrHIDEn
3 months ago

This:

```
let message = "Greetings {{ name }}!";  //TypeScript
```

Last edited 3 months ago by MrHIDEn

0 Reply

Lokesh Gupta Author
Reply to MrHIDEn 3 months ago

You are right. Updated.

0 Reply

Bert Roex
7 months ago

case Byte, Short, Integer, Long _ -> System.out.println("Input is a number");
results in error: expected case Byte, Short, Integer, Long

0 Reply

Lokesh Gupta Author
Reply to Bert Roex 7 months ago

You are right. I updated the snippet.

0 Reply

Seshadri Ramaswami
7 months ago

It woud have been nice if the presented examples were tested for error-free compilation before presenting this blog. For example, the line:
void addFirst(E);
threw an error of missing identifier after E. Such errors appeared dime a dozen times in the examples that followed.

0 Reply

Lokesh Gupta Author
Reply to Seshadri Ramaswami 7 months ago

Hi Rama, The snippets have been run and tested. Can you please share the code where you face any error?

X

Seshadri Ramaswami
Reply to Lokesh Gupta 7 months ago

Thanks for the prompt. Please address me as Sesh, as Ramaswami is my father's name. I have already given you the error statement (void addFirst(E)) and the error message too.

0 Reply

Lokesh Gupta Author
Reply to Seshadri Ramaswami 7 months ago

Hi Sesh, I am still not sure what you are trying to run. The method `void addFirst(E)` is mentioned as the newly added method in `SequencedCollection` interface. Why would someone like to execute an interface?

Did you tried the next code snippet where this method is actually used?

```
arrayList.addFirst(0);
```

0 Reply

Search ...

Weekly Newsletter

Stay Up-to-Date with Our Weekly Updates. Right into Your Inbox.

Email Address

x

X

x

X