Java 21 New Features Overview (Important)

Guide Java New Java Features About 2724 words About 9 minutes

JDK 21 was released on September 19, 2023. This is a very important version and a milestone.

JDK21 is the LTS (Long Term Support) version. So far, there are four long-term support versions: JDK8, JDK11, JDK17 and JDK21.

JDK 21 has 15 new features. This article will introduce some of the more important ones in detail:

- JEP 430: String Templates (Preview)
- JEP 431: Sequenced Collections
- JEP 439: Generational ZGC (generational ZGC)
- JEP 440: Record Patterns
- JEP 441: Pattern Matching for switch
- JEP 442: Foreign Function & Memory API (Preview 3)
- JEP 443: Unnamed Patterns and Variables (Preview)
- JEP 444: Virtual Threads
- JEP 445: Unnamed Classes and Instance Main Methods (Preview)

JEP 430: String Templates (Preview)

String Templates are currently still a preview feature in JDK 21.

String Templates provide a simpler and more intuitive way to dynamically construct strings. By using placeholders `${}` , we can embed variable values directly into strings without manual processing. At runtime, the Java compiler replaces these placeholder the actual variable values. Furthermore, expressions support local variables, static and non-static fields, even methods and calculation results.

In fact, String Templates exist in most programming languages:

```typescript
"Greetings {{ name }}!";  //Angular
`Greetings ${ name }!`;    //Typescript
$"Greetings { name }!"    //Visual basic
f"Greetings { name }!"    //Python
```

Before Java had String Templates, we usually used string concatenation or formatting methods to build strings:

```java
//concatenation
message = "Greetings " + name + "!";

//String.format()
message = String.format("Greetings %s!", name);  //concatenation

//MessageFormat
message = new MessageFormat("Greetings {0}!").format(name);

//StringBuilder
message = new StringBuilder().append("Greetings
").append(name).append("!").toString();
```

These methods all have some disadvantages to a greater or lesser extent, such as being difficult to read, lengthy, and complex.

Java uses String Templates for string concatenation, which allows you to embed expressions directly in strings without any additional processing:

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

In the template expression above:

- STR is the template processor.
- `\{name}` These expressions will be replaced by the corresponding variable values when running.

Java currently supports three template processors:

- STR: Automatically performs string interpolation, i.e. replaces each embedded expression in the template with its value (converted to a string).

- FMT: Similar to STR, but it also accepts format specifiers, which appear to the left of the embedded expression to control the style of the output.
- RAW: Does not automatically process string templates like the STR and FMT template processors do, but returns an `StringTemplate` object that contains information about the text and expressions in the template.

```java
String name = "Lokesh";

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

//FMT
String message = FMT."Greetings %-12s\{name}.";

//RAW
StringTemplate st = RAW."Greetings \{name}.";
String message = STR.process(st);
```

In addition to the three template processors that come with JDK, you can also implement `StringTemplate.Processor` the interface to create your own template processor. You only need to inherit `StringTemplate.Processor` the interface and then implement `process` the method.

We can use local variables, static/non-static fields and even methods as embedded expressions:

```java
//variable
message = STR."Greetings \{name}!";

//method
message = STR."Greetings \{getName()}!";

//field
message = STR."Greetings \{this.name}!";
```

You can also perform calculations within expressions and print the results:

```java
int x = 10, y = 20;
String s = STR."\{x} + \{y} = \{x + y}";  //"10 + 20 = 30"
```

To improve readability, we can split the embedded expression into multiple lines:

```java
String time = STR."The current time is \{
    //sample comment - current time in HH:mm:ss
    DateTimeFormatter
      .ofPattern("HH:mm:ss")
      .format(LocalTime.now())
  }.";
```

JEP 431: Serializing Collections

JDK 21 introduces a new collection type: **Sequenced Collections** , which are collections with a defined encounter order (no matter how many times we iterate over such a collection, the order of elements remains fixed). Sequenced Collections provide simple ways to manipulate the first and last elements of a collection, as well as reverse views (the reverse order of the original collection).

Sequenced Collections includes the following three interfaces:

- `SequencedCollection`
- `SequencedSet`
- `SequencedMap`

`SequencedCollection` The interface inherits `Collection` the interface and provides methods to access, add or remove elements from both ends of the collection and obtain the reverse view of the collection.

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

  // New Method

  SequencedCollection<E> reversed();

  // Promoted methods from Deque<E>

  void addFirst(E);
  void addLast(E);

  E getFirst();
```

```java
    E getLast();

    E removeFirst();
    E removeLast();
}
```

`List` The and `Deque` interface implements `SequencedCollection` the interface.

Here we take as `ArrayList` an example to demonstrate the actual usage effect:

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

arrayList.add(1);   // List contains: [1]

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

Integer firstElement = arrayList.getFirst();  // 0
Integer lastElement = arrayList.getLast();  // 2

List<Integer> reversed = arrayList.reversed();
System.out.println(reversed); // Prints [2, 1, 0]
```

`SequencedSet` The interface directly inherits `SequencedCollection` the interface and overrides `reversed()` the method.

```java
interface SequencedSet<E> extends SequencedCollection<E>, Set<E> {

    SequencedSet<E> reversed();
}
```

`SortedSet` and `LinkedHashSet` implement `SequencedSet` the interface.

Here we take as `LinkedHashSet` an example to demonstrate the actual usage effect:

```java
LinkedHashSet<Integer> linkedHashSet = new LinkedHashSet<>
(List.of(1, 2, 3));

Integer firstElement = linkedHashSet.getFirst();   // 1
Integer lastElement = linkedHashSet.getLast();    // 3
```

```
linkedHashSet.addFirst(0);  //List contains: [0, 1, 2, 3]
linkedHashSet.addLast(4);   //List contains: [0, 1, 2, 3, 4]

System.out.println(linkedHashSet.reversed());   //Prints [4, 3, 2, 1, 0]
```

`SequencedMap` The interface inherits `Map` the interface and provides methods for accessing, adding or removing key-value pairs at both ends of the collection, obtaining a collection containing a key `SequencedSet` , a value `SequencedCollection` , an entry (key-value pair) `SequencedSet` , and obtaining a reverse view of the collection.

```java
interface SequencedMap<K,V> extends Map<K,V> {

  // New Methods

  SequencedMap<K,V> reversed();

  SequencedSet<K> sequencedKeySet();
  SequencedCollection<V> sequencedValues();
  SequencedSet<Entry<K,V>> sequencedEntrySet();

  V putFirst(K, V);
  V putLast(K, V);


  // Promoted Methods from NavigableMap<K, V>

  Entry<K, V> firstEntry();
  Entry<K, V> lastEntry();

  Entry<K, V> pollFirstEntry();
  Entry<K, V> pollLastEntry();
}
```

`SortedMap` and `LinkedHashMap` implements `SequencedMap` the interface.

Here we take as `LinkedHashMap` an example to demonstrate the actual usage effect:

```java
LinkedHashMap<Integer, String> map = new LinkedHashMap<>();

map.put(1, "One");
map.put(2, "Two");
map.put(3, "Three");

map.firstEntry();   //1=One
map.lastEntry();    //3=Three

System.out.println(map);  //{1=One, 2=Two, 3=Three}

Map.Entry<Integer, String> first = map.pollFirstEntry();   //1=One
Map.Entry<Integer, String> last = map.pollLastEntry();    //3=Three

System.out.println(map);  //{2=Two}

map.putFirst(1, "One");     //{1=One, 2=Two}
map.putLast(3, "Three");    //{1=One, 2=Two, 3=Three}

System.out.println(map);  //{1=One, 2=Two, 3=Three}
System.out.println(map.reversed());   //{3=Three, 2=Two, 1=One}
```

JEP 439: Generational ZGC

JDK21 extends ZGC's functionality by adding generational GC. However, it is disabled by default and needs to be enabled through configuration:

```bash
// 启用分代ZGC
java -XX:+UseZGC -XX:+ZGenerational ...
```

In future versions, we will officially set ZGenerational as the default value, which means that ZGC's generational GC will be enabled by default. In later versions, non-generational ZGC will be removed.

> In a future release we intend to make Generational ZGC the default, at which point -XX:-ZGenerational will select non-generational ZGC. In an even later release we intend to remove non-generational ZGC, at which point the ZGenerational option will becc obsolete.

> In a future release, we plan to make Generational ZGC the default, at which point -XX:-ZGenerational will select non-generational ZGC. In an even later release, we plan to remove non-generational ZGC, at which point the ZGenerational option will become obsolete.

Generational ZGC can significantly reduce pause times during garbage collection and improve application responsiveness. This is very valuable for performance optimization of large Java applications and high-concurrency scenarios.

JEP 440: Recording Mode

Record mode was first previewed in Java 19, proposed by JEP 405. It was previewed again in JDK 20, proposed by JEP 432. Finally, Record mode was officially adopted in JDK 21.

The Java 20 New Features Overview has already introduced the record mode in detail, so I will not repeat it here.

JEP 441: Pattern Matching in Switch

Enhances the switch expression and statement in Java to allow patterns to be used in case labels. When the pattern matches, the code corresponding to the case label is executed.

In the following code, the switch expression uses type patterns for matching.

```java
static String formatterPatternSwitch(Object obj) {
    return switch (obj) {
        case Integer i -> String.format("int %d", i);
        case Long l    -> String.format("long %d", l);
        case Double d  -> String.format("double %f", d);
        case String s  -> String.format("String %s", s);
        default        -> obj.toString();
    };
}
```

JEP 442: Foreign Function and Memory API (Third Preview)

This API enables Java programs to interoperate with code and data outside the Java runtime. By efficiently calling external functions (that is, code outside the JVM) and safely accessing external memory (that is, memory not managed by the JVM), this API enables Java programs to call native libraries and process native data without the risks and brittleness of JNI.

The foreign function and memory APIs went through their first incubation phase in Java 17, as proposed by JEP 412. They went through their second incubation phase in Java 18, as proposed by JEP 419. They received their first preview in Java 19, as proposed by JEP 424. They received their second preview in JDK 20, as proposed by JEP 434. And they received their third preview in JDK 21, as proposed by JEP 442 .

In the Java 19 New Features Overview , I introduced the external function and memory API in detail, so I will not give additional introductions here.

JEP 443: Unnamed Patterns and Variables (Preview)

Unnamed patterns and variables allow us to use underscores `_` to represent unnamed variables and components not used in pattern matching, aiming to improve code readability and maintainability.

Typical scenarios for unnamed variables are exception variables in `try-with-resources` statements, clauses, and loops. When a variable is not needed , you can use an underscore to clearly identify it as unused. `catch` `for` `_`

```java
try (var _ = ScopedContext.acquire()) {
  // No use of acquired resource
}
try { ... }
catch (Exception _) { ... }
catch (Throwable _) { ... }

for (int i = 0, _ = runOnce(); i < arr.length; i++) {
  ...
}
```

An unnamed pattern is an unconditional pattern that is not bound to any value. An unnamed pattern variable appears within a type pattern.

```java
if (r instanceof ColoredPoint(_, Color c)) { ... c ... }

switch (b) {
    case Box(RedBall _), Box(BlueBall _) -> processBox(b);
    case Box(GreenBall _)                -> stopProcessing();
    case Box(_)                          -> pickAnotherBox();
}
```

JEP 444: Virtual Threads

Virtual threads are a major update and must be taken seriously!

Virtual threads were first previewed in Java 19, proposed by JEP 425. They were previewed again in JDK 20. Finally, virtual threads were officially implemented in JDK 21.

Virtual threads have been introduced in detail in the Java 20 New Features Overview , so I will not repeat it here.

JEP 445: Unnamed Class and Instance Main Methods (Preview)

This feature mainly simplifies `main` the method declaration. For Java beginners, this `main` method declaration introduces too many Java syntax concepts, which is not conducive to beginners getting started quickly.

Without using this feature before defining a `main` method:

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

To use this new feature, define a `main` method:

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

Even more concise (unnamed classes allow us to not define a class name):

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

refer to

- Java 21 String Templates: https://howtodoinjava.com/java/java-string-templates/
- Java 21 Sequenced Collections: https://howtodoinjava.com/java/sequenced-collections/

Recently Updated2025/3/18 18:23
Contributors: Guide , Mr.Hope