

# Java 8 a 21

**BCP**

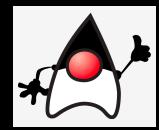
# José Amadeo Díaz Díaz



Principal Engineer



Fundador de JoeDayz.pe



Java Champion



Miembro de @PeruJUG



@jamdiazdiaz



<https://github.com/joedayz>



# Introducción

- Java 1.1 - Inner Classes
- Java 2 - Swing
- Java 5 - Generics
- Java 8
- ...
- Java 21

Nuevas versiones de Java cada 6 meses

JavaFX, JAXB, Pack200 removidos del JDK

# Java 8

# Java 8 - disruptiva

- Liberada en Marzo 2014. El primer release LTS (Long Term Support)
- Lambda Expressions
- Method References
- Default Methods
- Streams
- Annotation changes
- Method parameter reflection
- New date and time API

# LAMBDA EXPRESSIONS

```
//Antes de Java 8
Thread t1 = new Thread(new Runnable() {
    @Override
    public void run() {
        //Tarea que se ejecutará en otro thread
    }
});
```

```
//Post Java 8 usando Lambda
Thread t2 = new Thread(()-> {
    //Tarea que se ejecutará en otro thread
});
```

```
Runnable r1 = new Runnable() {
    @Override
    public void run() {
        //task r1
    }
};
```

```
Runnable r2 = () -> {
    //task r2
};
```

# Method References

```
Task obj = new Task();
Thread thread = new Thread(obj::runThisMethodOnAnotherThread);
```

```
class Task {
    1 usage
    void runThisMethodOnAnotherThread(){
        //task
    }
}
```

# Default Methods

```
public interface Greeting {  
  
    no usages 1 implementation  
    void hello();  
  
    no usages  
    static Greeting createGreeting(){  
        return new GreetingImplementation();  
    }  
  
    no usages  
    default void goodbye(){  
        System.out.println("Goodbye");  
    }  
}
```

# Streams

```
public static void main(String[] args) {  
    List<String> memberNames = new ArrayList<>();  
    memberNames.add("Amitabh");  
    memberNames.add("Shekhar");  
    memberNames.add("Aman");  
    memberNames.add("Rahul");  
    memberNames.add("Shahrukh");  
    memberNames.add("Salman");  
    memberNames.add("Yana");  
    memberNames.add("Lokesh");  
  
    memberNames.stream().filter((s) -> s.startsWith("A"))  
        .forEach(System.out::println);  
}
```

# Annotation Changes

```
@Allow(group="Managers")
@Allow(group="SeniorManagers")
public class AnnotationChanges {  
}
```

```
public @NotNull ResponseObject method(@NotNull String arg){
    return new ResponseObject();
}
```

# Method Parameters Reflection

```
public class Reflection {  
  
    public static void main(String[] args) {  
        Method[] methods = Object.class.getMethods();  
        for(Method m: methods){  
            System.out.print(m.getName() + "(");  
            for(Parameter p: m.getParameters()){  
                System.out.print(p.getName() + " ");  
            }  
            System.out.println(")");  
        }  
    }  
}
```

# New Date and Time APIs

```
public static void main(String[] args) {
    //Previo a Java 8
    Date currentDate = new Date();

    //Java 8
    LocalDate localDate = LocalDate.now();

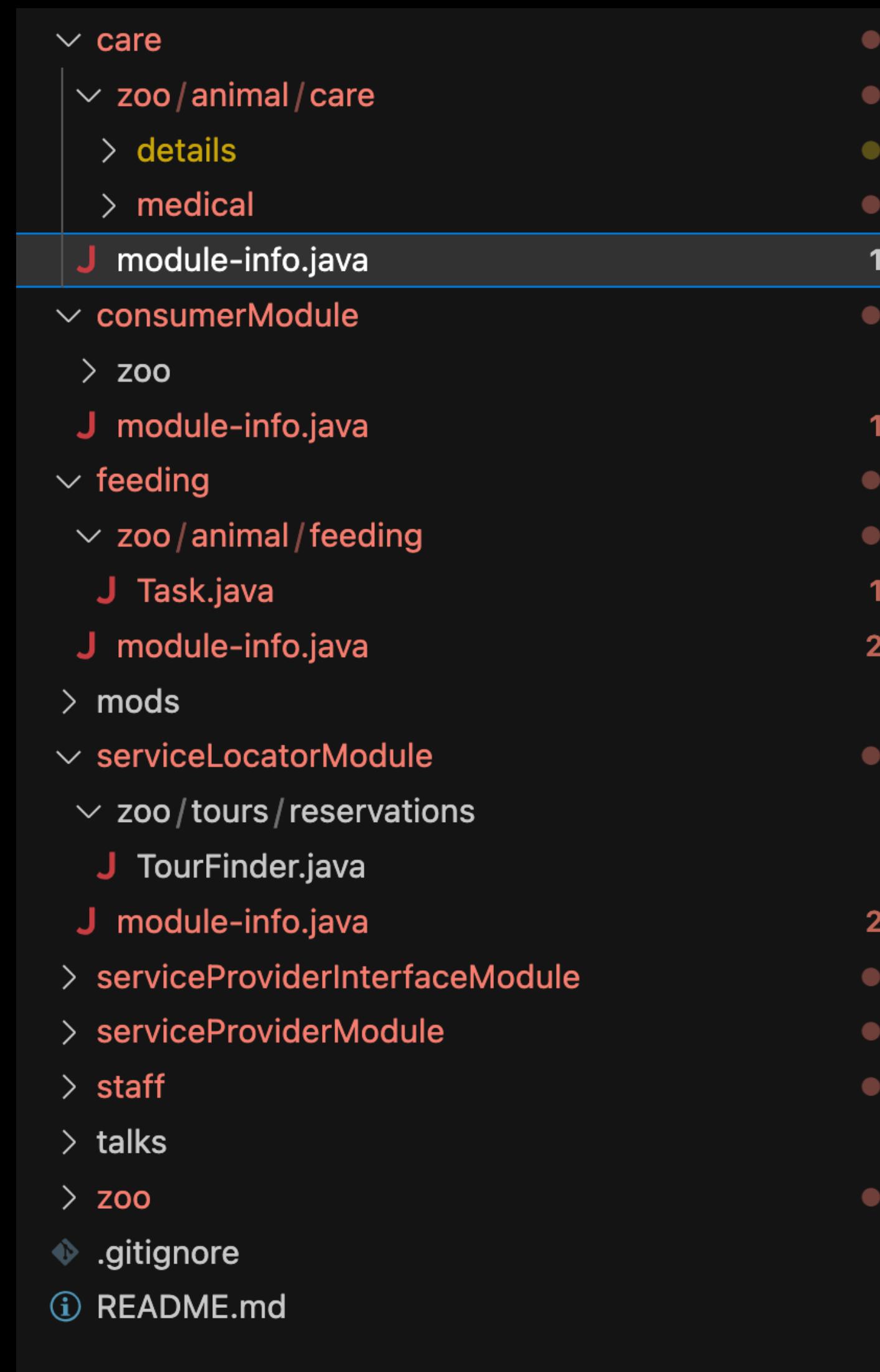
    LocalDate.of(year: 2023, month: 01, dayOfMonth: 01);
    LocalDate.parse(text: "2023-01-01");

    //Previo a Java 8 - 5 semanas en el futuro
    Calendar calendar = Calendar.getInstance();
    calendar.add(Calendar.DAY_OF_YEAR, amount: 35);

    //Java 8
    LocalDate fiveWeeksFromTime = localDate.plus(amountToAdd: 5, ChronoUnit.WEEKS);

}
```

# Módulos AKA Jigsaw (Java 9)



```
J module-info.java 1 ×  
care > J module-info.java  
1 module zoo.animal.care {  
2  
3     exports zoo.animal.care.medical;  
4  
5     requires zoo.animal.feeding;  
6     //requires transitive zoo.animal.feeding;  
7 }  
8
```

# Cambios en la VM

- Memory footprint
- Pauses
- Performance
- Es el último GC.
- Concurrent vs Parallel GC
  - Un Parallel GC tiene múltiples GC threads. Estos GC threads ejecutan su labor de recolección de basura. Son claves para colecciónar a gran escala.
  - Un Concurrent GC le permite a la JVM hacer otras cosas mientras está en la mark phase y opcionalmente durante otros stages.
  - Ambos incurren en unreachable objects, y largos periodos de tiempo donde no reclaman la memoria en el heap.

- Memory footprint
- Pauses
- Performance
- Serial collector
  - Este es un simple-thread GC. Esto significa que es más rápido que muchos GC, pero, tiene más pausas. No util para deployments en el mundo real, pero, facilita el debugging y algunos comportamientos.
  - Un gran caso de uso para este GC es en serverless (lambdas). Solución ideal para memoria física limitada en un simple-core VM.
  - Puedes activar este GC explicitamente usando **-XX:+UseSerialGC**.

# Shenandoah (Java 12)

- Memory footprint
- Pauses
- Performance
- Parallel collector o Throughput collector
  - Es multi-thread e ideal para usar en producción, pero, hay mejores elecciones en otros casos. El serial collector es para benchmarks, y ZGC/G1 usualmente ofrece mejor performance.
  - Activas este GC usando el **-XX:+UseParallelGC**.
  - Este GC es configurable, podemos usar las siguientes JVM Options:
    - **XX:ParallelGCThreads=ThreadCount**: El # de GC threads usados por el collector.
    - **-XX:MaxGCPauseMillis=MaxDurationMilliseconds**: Limite de pausa en milisegundos. El default es sin limite.
    - **-XX:GCTimeRatio=ratio**: Establece el tiempo dedicado a el GC en un  $1/(ratio+1)$ , y entonces, un valor de 9 debería ser  $1/(9+1)$  o 10%, es decir, 10% del tiempo del CPU debería ser gastando en GC. El valor x defecto es 99, el cual significa 1%.

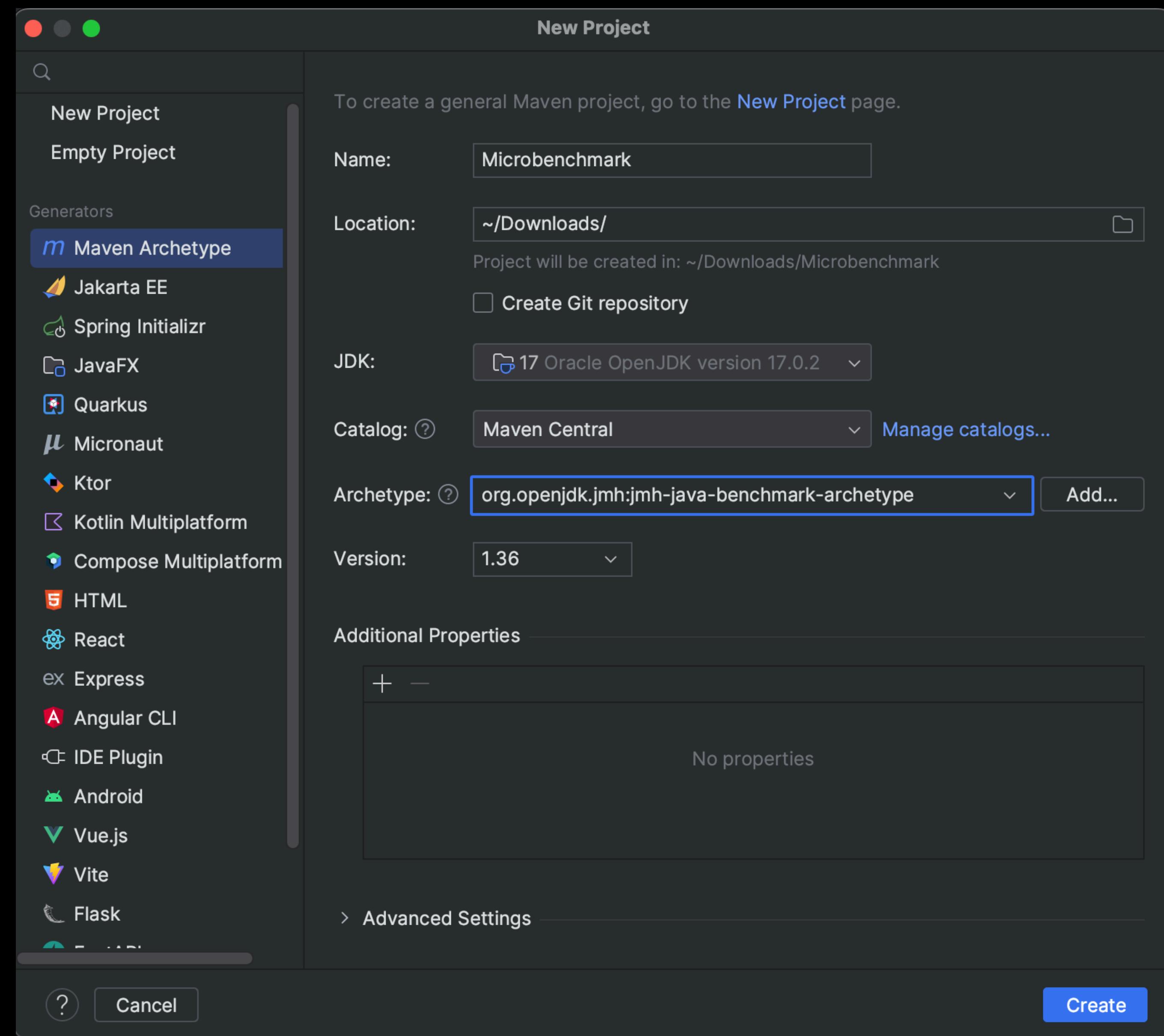
- Memory footprint
- Pauses
- Performance
- G1 garbage collector
  - Es un GC para cargas pesadas en maquinas con gran tamaño de heap (6GB o más). Se adapta a las condiciones de trabajo de la máquina. Se habilita explícitamente usando la JVM option **-XX:+UseG1GC**.
  - G1 es un GC concurrente que trabaja en background y minimiza las pausas. Una de sus principales características es la deduplicación de strings lo que reduce el overhead de strings en RAM. Se activa esta característica usando **-XX:+UseStringDeduplication**.

- Memory footprint
- Pauses
- Performance
- Z Garbage Collector (ZGC)
  - Es diseñado para mayores tamaños de heap que G1 y es también un GC concurrente. Este soporta entornos pequeños y puede ser usado para tamaños de heap tan pequeños desde 8MB a 16 TB como máximo de heap.
  - Una de sus mayores características es que no pausa la ejecución de la aplicación por mas de 10ms. El costo es una reducción en throughput.
  - El ZGC puede ser habilitado usando el **-XX:+UseZGC** JVM option.

## Shenandoah (Java 12)

- Memory footprint
- Pauses
- Performance
- Shenandoah
  - Depende del tamaño del heap. Se ejecuta mejor cuando hay suficiente espacio para todas las asignaciones cuando el GC se ejecuta concurrentemente. Este trabaja bien hasta 128 GB de heap size y puede bajar a 1 GB. Es diseñado para gran escala, similar a G1 y ZGC.
  - Shenandoah tiene pausas mayores que ZGC (10ms vs 1ms), tiene una alta latencia de cola comparada a ZGC.
  - Su performance consistente lo hace mas superior que ZGC. Si la performance y throughput son factores determinantes, entonces, Shenandoah es la mejor opción.
  - No todas las distribuciones de JDK incluyen Shenandoah, pero, si esta disponible lo activas con **-XX:+UseShenandoahGC**.

- Este fue agregando como parte de la JEP (JDK Enhancement Proposal) 230.
- Si queremos ver si un test se ejecuta bien, usualmente escribimos un simple test que se ejecuta en un loop y medimos el tiempo que toma. El problema es que usamos un compilador Just in Time (JIT) que va optimizando el código, y no olvidemos el GC que podría impactar en la performance y afectar los resultados de las pruebas.
- Para evitar esto se puede usar el microbenchmark framework, entonces, necesitamos configurar Maven y crear un arquetipo usando JMH para incluir las anotaciones necesarias.



Usa Java 19

```
44 ► public class MyBenchmark {
45     4 usages
46     private static final int[] ARRAY = new int[1000];
47     2 usages
48     private static final List<Integer> LIST = new ArrayList<>();
49
50     static {
51         Random random = new Random();
52         for(int iter = 0 ; iter < ARRAY.length ; iter++) {
53             ARRAY[iter] = random.nextInt();
54             LIST.add(ARRAY[iter]);
55         }
56     }
57
58     no usages
59     @Benchmark
60     public long primitiveArrayPerformance() {
61         var result = 0L;
62         for(var current : ARRAY) {
63             result += current;
64         }
65         return result;
66     }
67
68     no usages
69     @Benchmark
70     public long listPerformance() {
71         var result = 0L;
72         for(var current : LIST) {
73             result += current;
74         }
75         return result;
76     }
77 }
```

Benchmark	Mode	Cnt	Score	Error	Units
MyBenchmark.listPerformance	thrpt	5	1768538.153 ± 172302.008	ops/s	
MyBenchmark.primitiveArrayPerformance	thrpt	5	4165804.014 ± 819596.316	ops/s	

Process finished with exit code 0

- Este es uno de los mejores cambios después de Java 8.
- Cuando Java se lanza, teníamos green thread. Fueron implementados completamente en Java, y la VM en si mismo parecía un simple thread del OS.
- El problema es que si el thread se colgaba, la JVM se colgaba también. Por eso se hizo el switch a threads nativos que se usa hasta ahora.
- Los threads nativos son magníficos, se ejecutan rápido, y usan los recursos del hardware. Podemos tener múltiples CPU cores y tener diferentes thread en cada core.

# Virtual Threads - Loom (Java 21)

- El Java API abrazo los threads. Cada operación de lectura IO, y cada llamada a un servidor se ejecuta en su propio thread. Esto trabajo bien en la era de los monolitos, pero, con micro servicios, tenemos múltiples aplicaciones ejecutándose en contenedores. Un contenedor es evaluado por el # de conexiones que puede manejar.
- Desafortunadamente, Java tuvo un bajo rendimiento, ya que no puede manejar muchos network requests debido a que cada request requiere un thread.
- Otras plataformas que usan un single-thread se ejecutan mejor. Lo cual ya no es sorpresa cuando se entiende porque.
- En Java, los threads se gastan todo el tiempo bloqueando la lectura de un request desde un cliente o bloqueando la lectura de datos desde una base de datos. Sea cual sea el caso , los thread estuvieron ahí sin hacer nada.

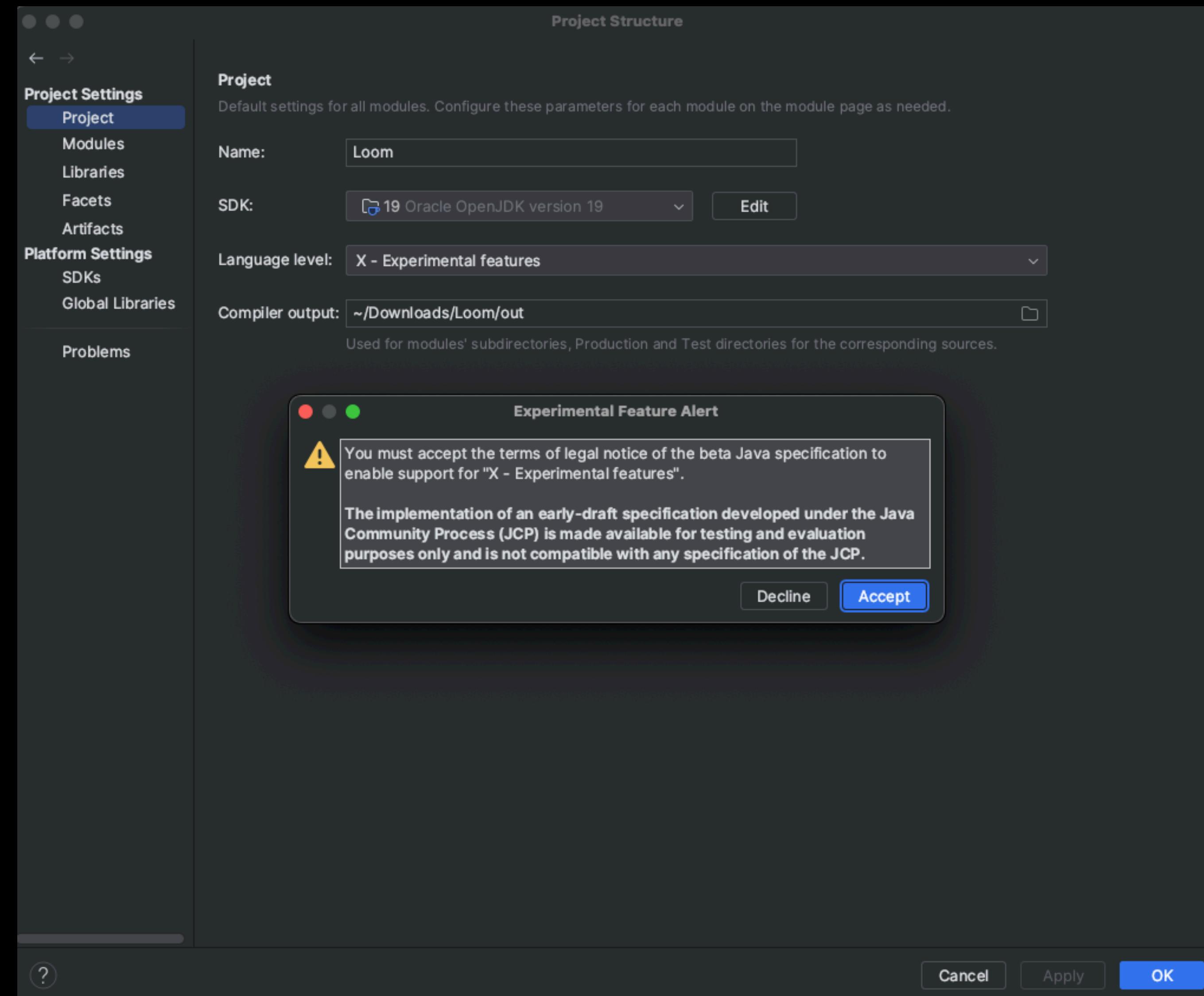
- ¿Entonces, porque necesitamos más threads?
- Si tienes muchos OS Threads estos consumen muchos recursos.
- Hay mucho overhead cuando se da switching entre threads. Y por eso usamos thread pools para reutilizar los OS threads y evitar la creación de nuevos.
- Una solución que apareció fueron las APIs asíncronas con frameworks Java. Pero, su sintaxis es compleja, debido a que Java fue diseñado para codificar de forma sincrona. Algunos lenguajes agregaron keywords como `async`/`await` para hacer el código asíncrono mas legible. Pero, Java escogió otro camino.

- Con un proceso asíncrono, podemos leer la IO de forma diferente. Cuando este hace un bloqueo, el event loop pasa la pelota a otra nueva tarea. El proceso IO enviará un callback cuando mas data este lista. Esto funciona bien, pero, sin embargo, esto significa que los stacks estarán desconectados. Si hay una falla, no tienes un contexto, solo el valor de las variables actuales. No hay encadenamiento.
- Es duro hacer tareas complejas atómicas. La programación sincrona es rápida, simple, fácil de depurar, y mejor en cada ruta. Excepto throughput. No escala.

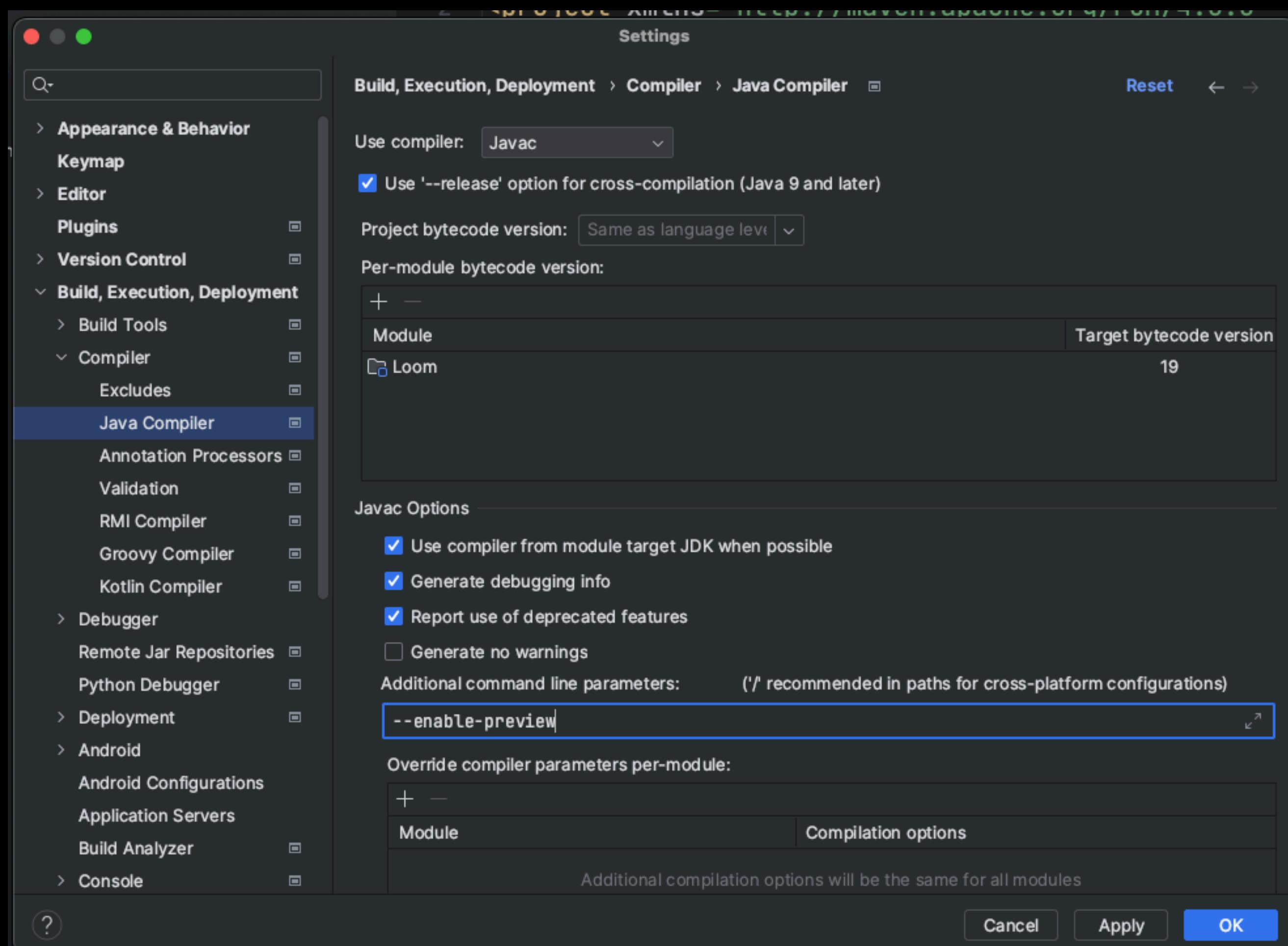
- Project Loom es el nombre para virtual threads. El concepto es simple: en lugar de usar threads nativos, un virtual thread es como un green thread. Pero, a diferencia de un green thread este puede usar multiples OS threads por detrás (como CPUs core tengamos).
- Esto nos da la habilidad de usar el hardware como los threads nativos, pero, también reduce el overhead de crear un thread a nada.
- 1 thread nativo puede tener 2MB RAM de overhead vs 1Kb por virtual thread. Es posible crear un millón de threads y no estoy exagerando.
- El código casi no cambia con Loom. Código existente puede ser convertido a Virtual threads. No necesitas adaptar este como cuando usas un framework asíncrono. Pero, si existen limitaciones.

- Si estamos escribiendo código intensivo en CPU, podría tener mas sentido usar un thread nativo. Caso contrario, algunos virtual threads serían impactados.
- Se puede mezclar ambos: threads nativos vs virtual threads, pero, usando las APIs de creación de threads existentes.
- No tiene sentido tampoco tener pool de virtual threads.
- Para usar virtual threads hay que ejecutar algunos pasos en el intelliJ IDEA.

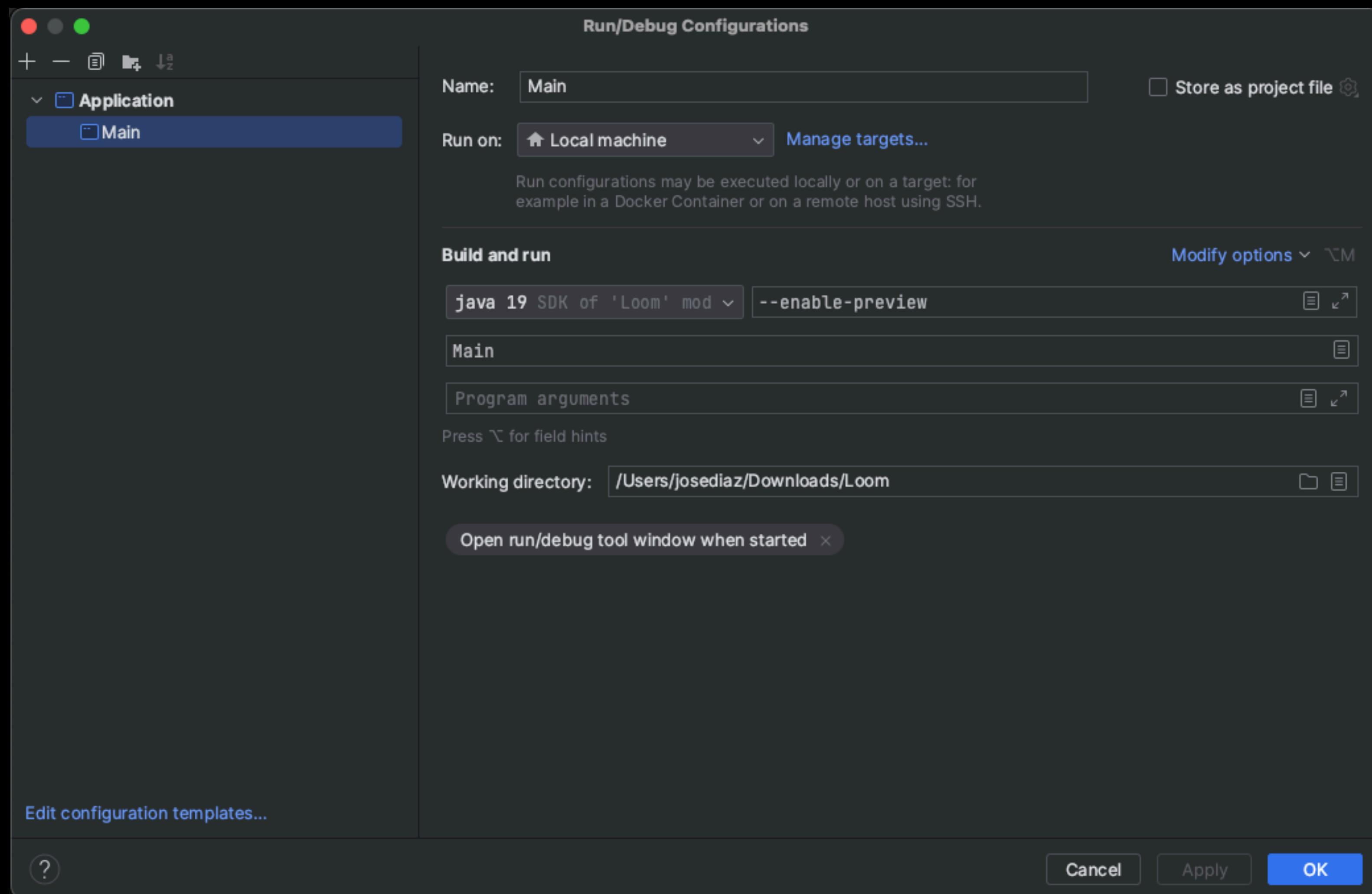
# Experimental Features



# Habilitar --enable-preview en la opción Compiler



# Agregar --enable-preview en VM arguments - Run configuration



- Java 9 depreco el concepto de finalización.
- Java no tenía destructores. No ejecutaba un método cuando el objeto era destruido. Si tiene un método `finalize()`, pero, este es invocado por el GC para indicar que el objeto será destruido. ¿Porque necesitaríamos algo así?
- El finalizer en sí es invocado en un thread interno del GC y puede causar problemas o race problemas. Se convierte también en un security Problem y un technical Problem con fuertes impactos.
- Java 9 agrego el Cleaner API, el cual hace algo similar sin los problemas de la finalización.

- Character encoding es uno de los temas mas dolorosos de entender.
- Java tomo el Unicode como el default.
- Antes de los 90, UTF-8 no era el default. Pero, luego paso a ser el estándar para encoding.

```
public static void main(String[] args) throws UnsupportedEncodingException {
    //Previo a Java 8 este método arrojaba una exception chequeada
    byte[] bytes = { 1, 2, 4, 8, 16, 32, 64, (byte) 128 };
    var str = new String(bytes, charsetName: "UTF-8");

    //Post Java 8 esto aún sería UTF-8
    var str2 = new String(bytes);
}
```

# Cambios en el lenguaje

# Cambios en el lenguaje

- Try-with-resources

```
Scanner scanner = null;
try {
    scanner = new Scanner(new File("test.txt"));
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
} finally {
    if (scanner != null) {
        scanner.close();
    }
}
```

```
try (Scanner scanner = new Scanner(new File("test.txt"))) {
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (FileNotFoundException fnfe) {
    fnfe.printStackTrace();
}
```

# Cambios en el lenguaje

- Métodos privados en interfaces (Java 9)
- var keyword (Java 10)
- Switch expression (Java 14)

```
1  public void daysOfMonth(int month) {  
2      switch (month) {  
3          case 1:  
4          case 3:  
5          case 5:  
6          case 7:  
7          case 8:  
8          case 10:  
9          case 12:  
10         System.out.println("this month has 31 days");  
11         break;  
12     case 4:  
13     case 6:  
14     case 9:  
15         System.out.println("this month has 30 days");  
16         break;  
17     case 2:  
18         System.out.println("February can have 28 or 29 days");  
19         break;  
20     default:  
21         System.out.println("invalid month");  
22     }  
23 }
```

```
1  switch (month) {  
2      case 1, 3, 5, 7, 8, 10, 12 -> System.out.println("this month has 31 days");  
3      case 4, 6, 9 -> System.out.println("this month has 30 days");  
4      case 2 -> System.out.println("February can have 28 or 29 days");  
5      default -> System.out.println("invalid month");  
6  }
```

# Cambios en el lenguaje

1. With arrow labels when a full block is needed:

```
int value = switch (greeting) {
    case "hi" -> {
        System.out.println("I am not just yielding!");
        yield 1;
    }
    case "hello" -> {
        System.out.println("Me too.");
        yield 2;
    }
    default -> {
        System.out.println("OK");
        yield -1;
    }
};
```

2. With traditional blocks:

```
int value = switch (greeting) {
    case "hi":
        System.out.println("I am not just yielding!");
        yield 1;
    case "hello":
        System.out.println("Me too.");
        yield 2;
    default:
        System.out.println("OK");
        yield -1;
};
```

# Pattern Matching (Java 19)

```
1. var height = switch (myShape) {  
2.     case Rectangle r    -> r.height;  
3.     case GeneralPath c -> c.getBounds().height;  
4.     default           ->  
5.         throw new IllegalArgumentException("Unrecognized shape");  
6. };
```

No if ni instanceof test

# Sealed Classes (Java 17)

```
2 usages 2 inheritors
```

```
public sealed class SealedBase permits SealedA, SealedB {  
    |  
}  
}
```

```
1 usage
```

```
final class SealedA extends SealedBase{  
}  
}
```

```
1 usage
```

```
final class SealedB extends SealedBase{  
}  
}
```

```
1. switch (base) {  
2.     case SealedA a -> out.println("It's A");  
3.     case SealedB b -> out.println("It's B");  
4.     case SealedBase theBase -> out.println("It's The Base");  
5. }
```

## Pattern-matching instanceof (Java 16)

```
1. // Before Pattern Matching instanceof  
2. if(number instanceof Float) {  
3.     Float f = (Float) number;  
4.     return Math.round(f);  
5. }  
6.  
7. // After pattern matching  
8. if(number instanceof Float f) {  
9.     return Math.round(f);  
10. }
```

## Text blocks (Java 15)

```
System.out.println(
    ctx.parser()
        .parseResultQuery("""
            SELECT TOP 1 table_schema, count(*)
            FROM information_schema.tables
            GROUP BY table_schema
            ORDER BY count(*) DESC
        """)
        .fetch()
);
```

# Records (Java 16)

```
record Rectangle(double length, double width) { }
```

```
public final class Rectangle {
    private final double length;
    private final double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    double length() { return this.length; }
    double width() { return this.width; }

    // Implementation of equals() and hashCode(), which specify
    // that two record objects are equal if they
    // are of the same type and contain equal field values.
    public boolean equals...
    public int hashCode...

    // An implementation of toString() that returns a string
    // representation of all the record class's fields,
    // including their names.
    public String toString() {...}
}
```

## Record Patterns (Java 19 preview)

```
1. if (o instanceof Point(int x, int y)) {  
2.     System.out.println(x+y);  
3. }
```

# String templates (Java 21 preview)

## Motivation

Developers routinely compose strings from a combination of literal text and expressions. Java provides several mechanisms for string composition, though unfortunately all have drawbacks.

- String concatenation with the `+` operator produces hard-to-read code:

```
String s = x + " plus " + y + " equals " + (x + y);
```

- `StringBuilder` is verbose:

```
String s = new StringBuilder()
    .append(x)
    .append(" plus ")
    .append(y)
    .append(" equals ")
    .append(x + y)
    .toString();
```

- `String::format` and `String::formatted` separate the format string from the parameters, inviting arity and type mismatches:

```
String s = String.format("%2$d plus %1$d equals %3$d", x, y, x + y);
String t = "%2$d plus %1$d equals %3$d".formatted(x, y, x + y);
```

- `java.text.MessageFormat` requires too much ceremony and uses an unfamiliar syntax in the format string:

```
MessageFormat mf = new MessageFormat("{0} plus {1} equals {2}");
String s = mf.format(x, y, x + y);
```

## Description

*Template expressions* are a new kind of expression in the Java programming language. Template expressions can perform string interpolation but are also programmable in a way that helps developers compose strings safely and efficiently. In addition, template expressions are not limited to composing strings — they can turn structured text into any kind of object, according to domain-specific rules.

Syntactically, a template expression resembles a string literal with a prefix. There is a template expression on the second line of this code:

```
String name = "Joan";
String info = STR."My name is \{name}";
assert info.equals("My name is Joan"); // true
```

The template expression `STR."My name is \{name}"` consists of:

1. A *template processor* (`STR`);
2. A dot character (U+002E), as seen in other kinds of expressions; and
3. A *template* ("My name is \{name}") which contains an *embedded expression* (\{name}).

When a template expression is evaluated at run time, its template processor combines the literal text in the template with the values of the embedded expressions in order to produce a result. The result of the template processor, and thus the result of evaluating the template expression, is often a `String` — though not always.

# Unnamed patterns and variables (Java 21 preview)

```
1. try {  
2.     // ....  
3. } catch(Exception ex) {  
4.     genericErrorHandler();  
5. }
```

```
1. try {  
2.     // ....  
3. } catch(Exception _) {
```

```
1. switch (b) {  
2.     case Box(RedBall _), Box(BlueBall _) -> processBox(b);  
3.     case Box(GreenBall _) -> stopProcessing();  
4.     case Box(_) -> pickAnotherBox();  
5. }
```

# APIs

# HttpClient (Java 11)

```
1. HttpClient client = HttpClient.newBuilder()  
2.         .version(Version.HTTP_2)  
3.         .build();  
  
4. HttpRequest request = HttpRequest.newBuilder()  
5.         .uri(URI.create("https://debugagent.com/"))  
6.         .build();  
  
7. var response = client.send(request, HttpResponse.BodyHandlers.  
ofByteArray());  
  
8. System.out.println(new String(response.body()));
```

# Foreign Function and memory API - Panama (Java 19)

## Java Native Interfaces (JNI)

```
1. public class HelloPanama {  
2.     public static void main(String[] args) {  
3.         try (var confined = MemorySession.openConfined()) {  
4.             MemorySegment s = confined.allocateUtf8String("Hello  
Panama");  
5.             printf(s);  
6.         }  
7.     }  
8. }
```

# Structured concurrency (Java 19)

```
1. try(ExecutorServicee=Executors.newVirtualThreadPerTaskExecutor())
   {
2.     e.submit(() -> methodA());
3.     e.submit(() -> methodB());
4. }
```

## Serialization filtering (Java 9)

```
java "-Djdk.serialFilter=!*" -jar my app.jar
```

```
java "-Djdk.serialFilter=!mypackage.*" -jar my app.jar
```

```
java "-Djdk.serialFilter=mypackage.*;!*" -jar my app.jar
```

## Scoped Values (Java 20)

```
1. final static ScopedValue<ClassOfX> X = new ScopedValue<>();  
2.  
3. ScopedValue.where(X, valueOfX, () -> {  
4.     ClassOfX xValue = X.get();  
5.     // ...  
6. });
```

# Sequenced Collection (Java 21)

```
1. interface SequencedCollection<E> extends Collection<E> {  
2.     // new method  
3.     SequencedCollection<E> reversed();  
4.     // methods promoted from Deque  
5.     void addFirst(E);  
6.     void addLast(E);  
7.     E getFirst();  
8.     E getLast();  
9.     E removeFirst();  
10.    E removeLast();  
11. }
```

```
1. interface SequencedMap<K,V> extends Map<K,V> {  
2.     // new methods  
3.     SequencedMap<K,V> reversed();  
4.     SequencedSet<K> sequencedKeySet();  
5.     SequencedCollection<V> sequencedValues();  
6.     SequencedSet<Entry<K,V>> sequencedEntrySet();  
7.     V putFirst(K, V);  
8.     V putLast(K, V);  
9.     // methods promoted from NavigableMap  
10.    Entry<K, V> firstEntry();  
11.    Entry<K, V> lastEntry();  
12.    Entry<K, V> pollFirstEntry();  
13.    Entry<K, V> pollLastEntry();  
14. }
```

# Futuro

- En unos años será la principal JVM.
- 2 JITs: uno para client base workloads y otro para server workloads.
- GraalVM puede ser usado como una JVM regular y un JIT.
  - Polyglot: puede ejecutar otros lenguajes nativos y conectarse a ellos, es decir, código Java puede invocar Python, Ruby, JavaScript y C. Todo hospedado en una simple VM
  - Native Image: GraalVM puede empaquetar una aplicación Java como una imagen nativa. Un JAR puede ser compilado a ejecutable nativo del OS sin dependencias, con un startup time ultra rápido y bajo overhead de RAM.

- Actualmente primitivos y objetos.
- Valhalla desea permitirnos que escribamos objetos como primitivos.

```
1. value class Dimension {  
2.     int width;  
3.     int height;  
4. }
```

- La clase no tendrá pointer. La JVM la tratará como una estructura en el stack que puede ser copiada pero no reverenciada.

- ¿Qué pasaría si queremos comparar valores que no tienen identidad?

```
1. if(dimension1 == dimension2) {  
2. //...  
3. }
```

- El código será tratado como:

```
1. if(dimension1.width == dimension2.width && dimension1.height ==  
dimension2.height) {  
2. //...  
3. }
```

- Valhalla incluye el primitive keyword:

```
1. primitive class Dimension {  
2.     int width;  
3.     int height;  
4. }
```

- El primitivo no acepta null. Esto significa que la asignación de valores trabajará de forma distinta, en algunos casos el objeto será copiado por completo.

```
1. var size = dimension[0];
```

# Que te recomiendo revisar

- <https://openjdk.org/projects/jdk/21/>
- <https://www.youtube.com/@java>
- <https://www.youtube.com/@joedayzperu>
- <https://www.facebook.com/groups/perujug>
- [https://groups.google.com/g/itp\\_java](https://groups.google.com/g/itp_java)
- <https://www.linkedin.com/company/perujug>
- <https://twitter.com/perujug>
- <https://perujug.org/>
- <https://jconf.perujug.org/>
- <https://www.linkedin.com/company/banco-de-credito-bcp/>
- <https://twitter.com/bcpcomunica>



**BCP**

# Gracias

@perujug