Java after 11

What's new?

- Better NullPointerExceptions
- Garbage Collection Improvements
- Text Blocks
- Pattern matching for instanceof
- Switch Expressions
- Records
- Sealed Classes
- And more!

Better NullPointerExceptions

```
a.b.c.i = 99; // Throws a NullPointerException
```

Before...

```
Exception in thread "main" java.lang.NullPointerException at Prog.main(Prog.java:5)
```

After...

```
Exception in thread "main" java.lang.NullPointerException: Cannot read field "c" because "a.b" is null ...
```

Garbage Collection (GC)

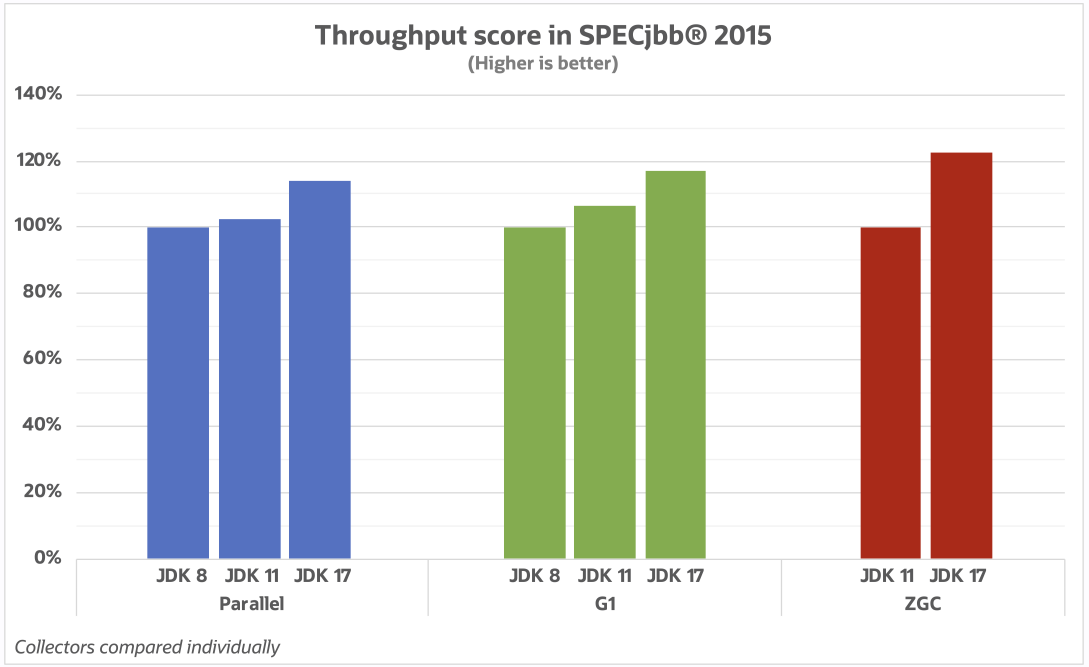
GC Trade-offs

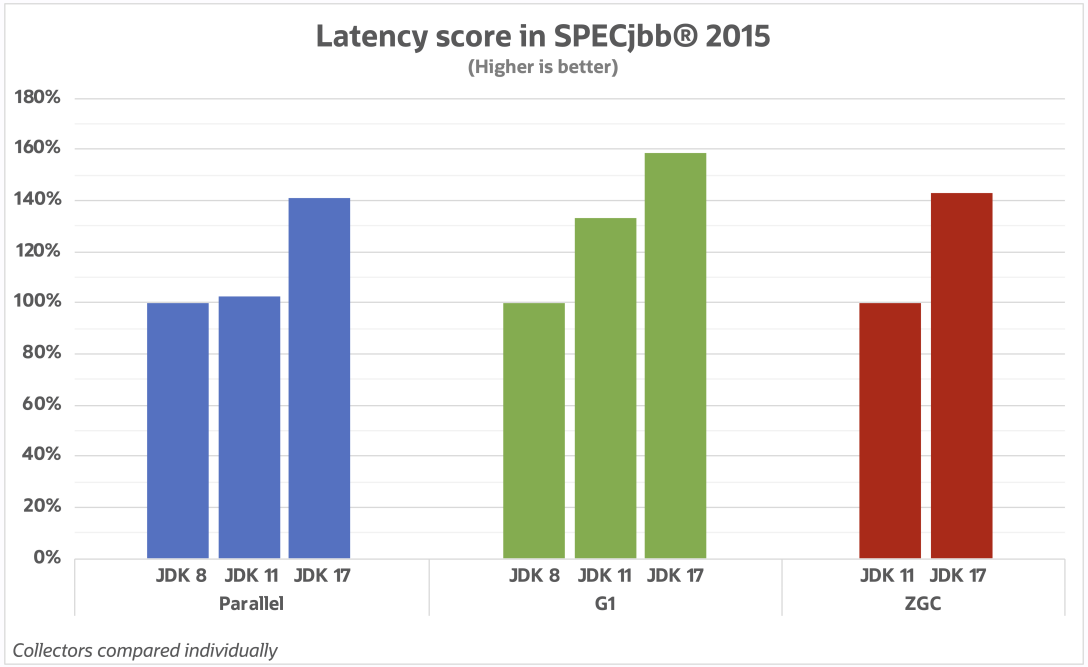
- Throughput
  - How much time is spent doing actual application work vs GC work?
- Latency
  - How does gc affect individual app operations?
- Footprint
  - What additional resources does GC require?

Java GCs

- Serial: simple, single threaded
- Parallel: throughput
- G1 (default): balance of throughput and latency
- Shenandoah: latency
- ZGC: latency
- Epsilon: no-op collector

GC Benchmarks





- Stefan Johansson - GC progress from JDK 8 to JDK 17

Text Blocks

Before...

```
String grossJson = "{\n\"id\": 1,\n\"qty\": 5,\n\"price\": 100.00}";
```

After...

```
String prettyJson = """
                {
                    "id": 1,
                    "qty": 5,
                    "price": 100
                }
                """;

String singleLine = """
                  Lorem ipsum dolor sit amet, consectetur adipiscing \
                  elit, sed do eiusmod tempor incididunt ut labore \
                  et dolore magna aliqua.
                  """;
```

Pattern Matching for instanceof

Before...

```
Object o = someRandomObject();
if (o instanceof String) {
    String s = (String)o;
    // do something with String s...
} else if (o instanceof Number) {
    Number n = (Number)o;
    // do something with Number n...
}
```

After...

```
Object o = someRandomObject();
if (o instanceof String s) {
    // do something with String s...
} else if (o instanceof Number n) {
    // do something with Number n...
}
```

Before...

```
public final boolean equals(Object o) {
    if (!(o instanceof Point)) return false;
    Point other = (Point) o;
    return x == other.x && y == other.y;
}
```

After...

```
public final boolean equals(Object o) {
    return (o instanceof Point other)
        && x == other.x && y == other.y;
}
```

Switch Expressions

Before...

```
int numLetters; // gross
switch (day) {
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        numLetters = 6;
        break;
    case TUESDAY:
        numLetters = 7;
        break;
    // Thursday, Saturday, Wednesday...
}
```

After...

```
int numLetters = switch (day) {
    // Arrows means no breaks needed, they don't "fall through"
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY                -> 7;
    case THURSDAY, SATURDAY     -> 8;
    case WEDNESDAY              -> 9;
}
```

- Expression returns a value
- Must be exhaustive, but 'default' is not required

Records

Before...

```
final class Range {
    private final int start;
    private final int end;

    Range(int start, int end) {
        this.start = start;
        this.end = end;
    }

    public int start() { return start; }
    public int end() { return end; }
    public boolean equals(Object o) { /*...*/ }
    public int hashCode() { /*...*/ }
    public String toString() { /*...*/ }
}
```

After...

```
record Range(int start, int end) { }
```

Usage:

```
var range = new Range(2, 3);
System.out.println(range.start());
System.out.println(range.end);
```

Record Properties

- Immutable
- Transparent
- Can't extend any class (implicitly extends record)
- Can't be extended
- Can implement interfaces

Record Constructors

- Automatically given `canonical constructors`
- **All** constructors must ultimately call the canonical constructor

```
record Range(int start, int end) {
    // Canonical constructor that uses the compact syntax
    Range {
        if (end < start) { throw new IllegalArgumentException("start must be less than end"); }
    }

    // Has to use the canonical constructor
    Range(int end) { this(0, end); }
}
```

Sealed Classes

```
class Shape { } // No limits to extension
```

```
final class Shape { } // Nothing can extend
```

- A sealed class can only be extended by classes **permitted** to do so

```
sealed class Shape {
    permits Circle, Rectangle, Triangle {
}
class Circle extends Shape { }
class Rectangle extends Shape { }
class Triangle extends Shape { }
```

What happens when we combine these?

- Pattern Matching
- Switch Expressions
- Records
- Sealed Classes

Data Oriented Programming

```
sealed interface AsyncResult<V> {
    record Success<V>(V result) implements AsyncResult<V> { }
    record Failure<V>(Throwable cause) implements AsyncResult<V> { }
    record Timeout<V>() implements AsyncResult<V> { }
    record Interrupted<V>() implements AsyncResult<V> { }
}
```

```
AsyncResult<V> r = future.get();
switch (r) {
    case Success<V>(var result): ...
    case Failure<V>(Throwable cause): ...
    case Timeout<V>(): ...
    case Interrupted<V>(): ...
}
```

Fun Stuff

Stream::toList

Before...

```
var nums = IntStream.range(0, 10)
                    .boxed()
                    .collect(Collectors.toList());
```

After...

```
var nums = IntStream.range(0, 10)
                    .boxed()
                    .toList();
```

Do these compile?

```
int x = 1;

int class = 1;

int goto = 1;

int static = 1;

int var = 1;

int void = 1;

int const = 1;
```

Solution

```
int x = 1;           // Yes...

int class = 1;       // No, java keyword

int goto = 1;        // No, java keyword that is not actually used (reserved)

int static = 1;      // No, java keyword

int var = 1;         // Yes! Reserved type name, not a keyword!

int void = 1;        // No, java keyword

int const = 1;       // No, another reserved java keyword

var var = "var";     // Yes!
```

Comparison Method Violates its General Contract!

- Based on talk by Stuart Marks

Does this work?

```
List<Integer> numbers = ...
Comparator<Integer> comparator = (a,b) -> a - b;
numbers.sort(comparator);
```

No!

- Example
  - a: large positive
  - b: large negative
- (a - b) overflows, creating a negative number
- Since (a - b) is negative, comparator thinks a < b

Does this work??

```
List<Integer> numbers = ...
Comparator<Integer> comparator = (a,b) -> a < b
    ? -1
    : a == b ? 0 : 1;
numbers.sort(comparator);
```

No!

- Auto-unboxing is the problem!
- the a == b is performing reference equality
- so a == b is always false
  - Unless a and b are the same object

Does this work???

```
List<Double> numbers = ...
Comparator<Double> comparator = (a,b) -> a < b
    ? -1
    : a > b ? 1 : 0
```

No!

- Example:
  - a: NaN
  - b: any number
- ANY comparison with NaN evaluates to false

```
NaN < 1000 -> false
NaN > 1000 -> false
NaN == 1000 -> false
```

...does this work????

```
List<Integer> numbers = ...
numbers.sort(Integer::compare);
```

Yes!

- Lesson: Just use `Integer::compare`

Conclusion

- Java 17 improves...
  - System Performance
    - Enhanced garbage collectors
  - Developer Velocity
    - Better null pointer exceptions
    - Text blocks, Stream::toList
    - Pattern matching, switch expressions, and records
  - Developer Flexibility
    - Sealed classes
    - Data Oriented Programming (Brian Goetz Article)