

Java 15 features

Java 15 features

Java 15, released in **September 2020**, is part of the six-month release cadence introduced by Oracle. It is a **non-LTS (non-Long-Term Support)** release. Here's a summary of the **major features and enhancements** introduced in Java 15:

♦ 1. Text Blocks (Standard) – JEP 378

Purpose: Simplify the writing of multi-line strings in Java.

✅ Promoted from preview (JEP 368).

```
java

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

♦ Benefits:

- No need for escape sequences (`\n` , `\"` , etc.)
 - Easier formatting of SQL, HTML, XML, and JSON in Java code.
-

♦ 2. Sealed Classes (Preview) – JEP 360

Purpose: Restrict which other classes or interfaces may extend or implement a class.

```
java

public sealed class Shape permits Circle, Rectangle {}

final class Circle extends Shape {}
final class Rectangle extends Shape {}
```

♦ Benefits:

- Better control over type hierarchies.
 - Enhances pattern matching and exhaustiveness checks in the future.
-

♦ **3. Hidden Classes – JEP 371**

Purpose: Classes that are **not discoverable** and are meant for use **by frameworks** or **dynamic proxies**, e.g., **JVM-based languages**.

♦ Example use cases:

- Lambda metafactories
 - Dynamic proxies
 - Bytecode generation tools (e.g., ASM, CGLIB)
-

♦ **4. ZGC: Uncommit Unused Memory – JEP 377**

Purpose: Z Garbage Collector (ZGC) can now return unused heap memory to the operating system.

♦ Benefits:

- Reduces memory footprint for idle applications.
 - Especially helpful in containerized environments.
-

♦ **5. Shenandoah: Uncommit Unused Memory – JEP 379**

Purpose: Same goal as JEP 377 but for the **Shenandoah GC**.

♦ Benefits:

- Memory efficiency
 - Improved behavior in memory-constrained environments
-

♦ **6. Foreign-Memory Access API (Second Incubator) – JEP 383**

Purpose: A better way to access memory outside the Java heap (e.g., native memory).

```
java
```

```
try (MemorySegment segment = MemorySegment.allocateNative(100)) {  
    MemoryAccess.setIntAtOffset(segment, 0, 42);  
}
```

♦ Use cases:

- Interfacing with native libraries
- High-performance I/O

♦ **7. Edwards-Curve Digital Signature Algorithm (EdDSA) – JEP 339**

Purpose: Adds support for the **Ed25519** and **Ed448** cryptographic algorithms.

♦ Benefits:

- Modern, secure digital signature algorithms
- Faster than RSA/ECDSA
- Used in many new security protocols (e.g., TLS 1.3, SSH)

♦ **8. Removed:**

- **Nashorn JavaScript Engine – JEP 372**
❌ Removed from JDK 15. Was deprecated in JDK 11.
- **RMI Activation System – JEP 385**
❌ Removed the outdated RMI Activation mechanism.

✅ **Summary Table**

Feature	Type	JEP
Text Blocks	Final	378
Sealed Classes	Preview	360
Hidden Classes	Incubator	371

Feature	Type	JEP
ZGC: Uncommit Unused Memory	Final	377
Shenandoah: Uncommit Unused Memory	Final	379
Foreign-Memory Access API (2nd Incubator)	Incubator	383
EdDSA Algorithm	Final	339
Removed Nashorn	Removal	372
Removed RMI Activation	Removal	385

Would you like code examples for sealed classes or text blocks in a real-world context?

what is Incubator






In Java, "**Incubator**" refers to a mechanism used by the JDK (Java Development Kit) to **introduce new APIs or features that are not yet part of the official Java SE standard**, but are included for **early feedback and experimentation**.

What is an Incubator Module?

An incubator module:

- Is a **JDK-specific module**, not part of the official Java Platform API.
- Lets developers **try out** new APIs before they are finalized.
- May **change or be removed** in future releases.
- Typically used for **low-level or high-impact features**, like native memory access or foreign function interfaces.

Characteristics

Feature	Description
 Experimental	API is still in development/testing phase.
 May Change	The API may undergo changes or be removed in later versions.
 Feedback-Oriented	Designed to gather feedback from the community before standardization.
 Not Enabled by Default	Needs to be explicitly added using <code>--add-modules</code> .
 Located in <code>jdk.incubator</code>	Modules are prefixed with <code>jdk.incubator.*</code> .

Example: Foreign-Memory Access API

```
bash

java --add-modules jdk.incubator.foreign ...
```

This allows you to use APIs for accessing off-heap memory safely without using `sun.misc.Unsafe`.

Where to Find Incubator Modules?

They are usually found in:

```
bash

$JAVA_HOME/jmods/jdk.incubator.*.jmod
```

Incubator vs Preview

Aspect	Incubator	Preview
Scope	API/Module (usually in <code>jdk.incubator</code>)	Language/VM feature (e.g., <code>sealed</code> , <code>record</code>)
Requires Module?	Yes (<code>--add-modules</code>)	No, just enable preview (<code>--enable-preview</code>)
Compatibility	Subject to removal or major change	Might be finalized or removed
Example	Foreign Memory API	Sealed Classes, Switch Expressions

✓ Conclusion

The **Incubator** system helps the Java ecosystem **evolve safely and iteratively** by giving early access to powerful new APIs, while avoiding premature standardization.

Would you like a working example using an incubator module like

```
jdk.incubator.foreign?
```

what about string utility methods formatted and translateescape

Great question!

Java 15 introduced **two helpful utility methods** for working with strings:

◆ 1. `String::formatted` (Java 15)

✓ Purpose:

A **convenient alternative** to `String.format(...)` using the new instance method.

📌 Syntax:

```
java
```

```
String result = "Hello, %s!".formatted("World");
```

This is equivalent to:

```
java
```

```
String result = String.format("Hello, %s!", "World");
```

✓ Example:

```
java
```

```
String name = "Viswanath";
```

```
int age = 30;
```

```
String intro = "My name is %s and I am %d years old.".formatted(name, age);
```

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

Output:

```
pgsql
```

```
My name is Viswanath and I am 30 years old.
```

◆ 2. `String::translateEscapes` (Java 15)

✓ Purpose:

Converts **escape sequences in a string literal** (like `\n`, `\t`, `\\`, etc.) into actual characters.

📌 Syntax:

```
java
```

```
String result = "Line1\\nLine2".translateEscapes();
```

This will return:

```
java
```

```
"Line1\nLine2"
```

✓ Example:

```
java
```

```
String raw = "Hello\nWorld\tJava";  
String escaped = raw.translateEscapes();  
  
System.out.println(escaped);
```

Output:

```
nginx
```

```
Hello  
World Java
```

🧠 Use Cases:

- When reading strings from a file or JSON/XML that include **literal escape sequences**.
- Transforming escaped content into usable output.

✓ Summary Table

Method	Introduced in	Description
<code>String.formatted()</code>	Java 15	Instance-based replacement for <code>String.format()</code>
<code>String.translateEscapes()</code>	Java 15	Converts string escape sequences to actual characters

Would you like a few real-world scenarios where `translateEscapes()` is especially useful (like config file parsing or reading user input)?

these utility methods in java15 standard or any preview

Both `String::formatted()` and `String::translateEscapes()` are **standard features** in Java 15 — not preview features.

◆ Summary:

Method	Introduced In	Status in Java 15	Preview?
<code>String.formatted()</code>	Java 15	✓ Standard	✗ No
<code>String.translateEscapes()</code>	Java 15	✓ Standard	✗ No

📌 You can use both methods directly without any flags (like `--enable-preview`) or module activation.

These methods are part of the **standard Java API** starting from Java 15 and available in `java.lang.String`.

Would you like a version compatibility chart for other new `String` methods added in Java 11-21?

then what are all standard java 15 features

Here's a full list of all **standard (final)** features introduced in **Java 15**, meaning they are **ready for production** and don't require enabling preview or incubator modes:




✓ Standard Features in Java 15

Feature	JEP	Description
Text Blocks	JEP 378	Multi-line string literals with better readability and escaping.
<code>String.formatted()</code> method	—	Instance method to format strings (like <code>String.format()</code>)

Feature	JEP	Description
		but cleaner).
<code>String.translateEscapes()</code> method	—	Converts escape sequences (e.g. <code>\n</code> , <code>\t</code>) into actual characters.
ZGC: Uncommit Unused Memory	JEP 377	Z Garbage Collector can return unused memory back to the OS.
Shenandoah: Uncommit Unused Memory	JEP 379	Same as above, for the Shenandoah GC.
Edwards-Curve Digital Signature Algorithm (EdDSA)	JEP 339	Adds support for Ed25519 and Ed448 cryptographic algorithms.
Removal of the Nashorn JavaScript Engine	JEP 372	Removes the deprecated JavaScript engine (<code>Nashorn</code>).
Removal of RMI Activation System	JEP 385	Removes the obsolete and rarely used RMI Activation mechanism.







Not Standard (Preview/Incubator) in Java 15

These were not standard in Java 15 — they were in **preview** or **incubator** status:

Feature	JEP	Type
Sealed Classes	JEP 360	 Preview
Foreign-Memory Access API (2nd)	JEP 383	 Incubator
Hidden Classes	JEP 371	 Incubator

Summary

If you're looking for **production-ready, stable features**, these are **safe to use** in Java 15:

-  Text Blocks
-  `String.formatted()`
-  `String.translateEscapes()`
-  ZGC & Shenandoah memory improvements
-  EdDSA cryptography
-  Deprecation/clean-up of legacy components (Nashorn, RMI Activation)

Would you like a one-liner or code example for each of these standard Java 15 features?

why Removal of the Nashorn JavaScript Engine is removed

The **Nashorn JavaScript engine** was **removed in Java 15 (JEP 372)** for several well-founded reasons. Here's a complete breakdown:

Why Nashorn Was Removed?

- ◆ **1. Outdated with Respect to ECMAScript (JavaScript) Evolution**
 - **Nashorn was introduced in Java 8 (2014)**, designed to support **ECMAScript 5.1**.
 - However, **modern JavaScript (ES6 and beyond)** has evolved rapidly with features like:
 - Arrow functions
 - Classes
 - Promises
 - Modules
 - Keeping Nashorn up to date with these evolving standards became **infeasible**.
- ◆ **2. Low Usage in Practice**
 - The number of Java applications using Nashorn in production was **relatively small**.

- Many developers moved to **embedding real JavaScript engines** (like Node.js, Graal.js) if needed.
-

◆ 3. High Maintenance Burden

- Nashorn was part of the **JDK core**, meaning:
 - Any JavaScript engine bugs or compatibility issues required fixing within the JDK.
 - This conflicted with the modular and lean direction of the JDK.
-

◆ 4. Better Alternatives Exist

- **GraalVM** offers a modern JavaScript engine (`Graal.js`) that:
 - Supports **new ECMAScript versions**
 - Can run **JavaScript, Python, Ruby, R, and more** in the JVM
 - Offers better **polyglot support**
-

◆ 5. Alignment with Project Jigsaw & JDK Clean-up

- Since **Java 9**, modularity (Project Jigsaw) and trimming unused/legacy APIs has been a goal.
 - Removing Nashorn aligns with removing outdated, rarely used components (like **Java EE modules, CORBA, RMI Activation**, etc.).
-

✅ Timeline of Deprecation and Removal

Java Version	Status
Java 11	! Nashorn deprecated
Java 15	✗ Nashorn removed

🧭 What if I Still Need JavaScript in Java?

Recommended Alternatives:

1. **Graal.js** (via GraalVM): Fully modern JS engine for the JVM
→ <https://www.graalvm.org/javascript/>
 2. **Rhino** (Mozilla): Still maintained for embedded scripting
→ <https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino>
-

Conclusion

Nashorn was removed because it was outdated, underused, and expensive to maintain — and better alternatives exist today. It's part of Java's effort to stay **modern, minimal, and maintainable**.

Would you like a sample of how to run JavaScript with GraalVM instead?

give detailed information with code examples of all standard features in chat, don't skip anything, i need detailed information

Absolutely! Here's a **detailed guide** with **code examples** and full explanations of **all standard (final)** features introduced in **Java 15**. These features are stable, don't require any preview flags, and are ready for production use.

Java 15 – Standard Features (Finalized)

◆ 1. Text Blocks (JEP 378)

Description:

A new way to create **multi-line string literals** using triple double quotes `"""`, improving readability for embedded content like HTML, JSON, SQL, etc.

Before Java 15 (old way):

```
java
```

```
String html = "<html>\n" +  
    " <body>\n" +  
    " <p>Hello, world!</p>\n" +  
    " </body>\n" +  
    "</html>";
```

✅ Java 15 with Text Blocks:

java

```
String html = """  
<html>  
  <body>  
    <p>Hello, world!</p>  
  </body>  
</html>  
""";
```

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

🧠 Key Features:

- Auto line breaks
- No need for `\n` or `\n`
- Preserves indentation using a common baseline

💡 2. `String.formatted()` Method

📌 Description:

A new instance method that acts like `String.format(...)`, but is **cleaner** and **more readable**.

✅ Example:

java

```
String name = "Alice";  
int age = 28;
```

```
// Old way:
```

```
String result1 = String.format("Name: %s, Age: %d", name, age);
```

// New way:

```
String result2 = "Name: %s, Age: %d".formatted(name, age);
```

```
System.out.println(result2); // Output: Name: Alice, Age: 28
```

Why it's useful:

- Improves readability and chaining with other string operations.
- Reduces static method noise.

◆ 3. `String.translateEscapes()` Method

Description:

This method **converts escape sequences** (like `\\n`, `\\t`, `\\uXXXX`) **into actual characters**.

✓ Example:

```
java
```

```
String escaped = "Hello\\nWorld\\tTabbed".translateEscapes();  
System.out.println(escaped);
```

✓ Output:

```
nginx
```

```
Hello
```

```
World Tabbed
```

Use Case:

Useful when escape characters are read from:

- config files
- user input
- external sources (JSON, YAML, etc.)

◆ 4. ZGC: Uncommit Unused Memory (JEP 377)

Description:

The **Z Garbage Collector (ZGC)** can now **return unused heap memory** to the OS **immediately**, reducing memory footprint.

✓ Benefits:

- Better memory efficiency
- Great for **containerized apps**, cloud environments, and microservices

✓ How to enable:

```
bash
```

```
java -XX:+UseZGC -Xmx2g MyApp
```

Notes:

- No code change needed
 - Only JVM option tuning
 - Java process will shrink its memory usage when idle
-

◆ 5. Shenandoah: Uncommit Unused Memory (JEP 379)

Description:

Similar to ZGC, the **Shenandoah GC** can now **release unused heap** memory back to the OS.

✓ Benefits:

- Low-pause GC with memory efficiency
- Useful for apps with **spiky memory usage**

✓ Enable Shenandoah:

```
bash
```

```
java -XX:+UseShenandoahGC MyApp
```

♦ 6. Edwards-Curve Digital Signature Algorithm (EdDSA) – JEP 339

Description:

Adds support for the **Ed25519** and **Ed448** digital signature algorithms, which are:

- **Faster**
- **Smaller keys**
- **More secure**
than traditional RSA or ECDSA.

✅ Example using Ed25519:

```
java

import java.security.*;

public class EdDSAExample {
    public static void main(String[] args) throws Exception {
        KeyPairGenerator keyGen = KeyPairGenerator.getInstance("Ed25519");
        KeyPair pair = keyGen.generateKeyPair();

        byte[] message = "Hello EdDSA!".getBytes();

        Signature signer = Signature.getInstance("Ed25519");
        signer.initSign(pair.getPrivate());
        signer.update(message);
        byte[] signature = signer.sign();

        Signature verifier = Signature.getInstance("Ed25519");
        verifier.initVerify(pair.getPublic());
        verifier.update(message);

        System.out.println("Signature valid? " + verifier.verify(signature));
    }
}
```

✅ Output:

```
nginx

Signature valid? true
```

◆ 7. Removal of Nashorn JavaScript Engine – JEP 372

Description:

- Nashorn was Java's built-in JavaScript engine.
- Removed due to being **outdated**, **rarely used**, and hard to maintain.

Alternative:

Use GraalVM's JavaScript engine (Graal.js) for modern JS support.

Implication:

If your app uses `jdk.nashorn.api.scripting.*`, you'll need to:

- Remove/replace it
- Use external libraries

◆ 8. Removal of RMI Activation System – JEP 385




Description:

- Removes legacy **RMI Activation**, a feature to lazily activate remote objects.
- Was obsolete and very rarely used.

What's Still Available:

 Regular RMI (Remote Method Invocation) is still part of Java SE.

Summary Table of Java 15 Standard Features

Feature	JEP	Type	Description
Text Blocks	378	 Standard	Multi-line strings with better formatting
<code>String.formatted()</code>	—	 Standard	Instance method for string formatting
<code>String.translateE scapes()</code>	—	 Standard	Converts <code>\\n</code> , <code>\\t</code> , etc. into real characters

Feature	JEP	Type	Description
ZGC: Uncommit Unused Memory	377	✓ Standard	ZGC can release unused memory to OS
Shenandoah: Uncommit Unused Memory	379	✓ Standard	Same for Shenandoah GC
EdDSA Signature Algorithm	339	✓ Standard	Adds Ed25519/Ed448 for cryptographic signing
Removal of Nashorn JavaScript Engine	372	✗ Removed	Outdated JS engine removed
Removal of RMI Activation System	385	✗ Removed	Removed obsolete RMI feature

Bonus Tip: Java 15 GC Selection

You can switch between GCs easily using JVM options:

```
bash

# Use ZGC
java -XX:+UseZGC MyApp

# Use Shenandoah
java -XX:+UseShenandoahGC MyApp
```

Would you like a downloadable summary PDF or a cheat sheet of Java 15 features?