Java 18 features

java18 features

Java 18 (released in **March 2022**) introduced a mix of **preview features**, **incubating APIs**, and a few **standard enhancements**. Here is a **detailed list of Java 18 features**, including code examples where applicable:

✅ Standard Features

1. UTF-8 by Default (JEP 400)

- Java now uses **UTF-8 as the default charset** for all platforms (Windows, Linux, macOS).

Before:
Default charset was platform-dependent (e.g., `Windows-1252` on Windows).

After Java 18:

```java
System.out.println(Charset.defaultCharset()); // Always UTF-8
```

---

2. Simple Web Server (JEP 408)

- Lightweight, command-line-based web server for prototyping and static file hosting.
- No external dependencies.

Usage:

```sh
# Start server on port 8000
$ jwebserver
```

```
# With specific directory and port
$ jwebserver --directory /path/to/site --port 9000
```

Programmatic API:

```java
HttpServer server = SimpleFileServer.createFileServer(
    new InetSocketAddress(8080),
    Path.of("/path/to/files"),
    OutputLevel.INFO
);
server.start();
```

🧪 Preview & Incubator Features

3. Code Snippets in Java API Documentation (JEP 413)

- Enables JavaDoc to use `<snippet>` tags for better code examples.

Example in JavaDoc:

```java
/**
 * <snippet>
 * var list = List.of("a", "b", "c");
 * </snippet>
 */
```

---

4. Pattern Matching for `switch` (Second Preview, JEP 420)

- Enhances the `switch` statement to support patterns and guarded conditions.

Example:

```java
static String formatter(Object o) {
    return switch (o) {
        case Integer i -> "int: " + i;
        case Long l    -> "long: " + l;
        case String s  -> "string: " + s;
        default        -> "unknown";
    };
}
```

- Supports `yield` and `when` guards:

```java
switch (obj) {
    case String s when s.length() > 5 -> System.out.println("Long string");
    default -> System.out.println("Other");
}
```

> 💡 Requires `--enable-preview` and `--source 18`.

---

5. Vector API (Third Incubator, JEP 417)

- Improves performance of vector computations on supported hardware (SIMD).
- Part of the `jdk.incubator.vector` module.

Example:

```java
 var a = FloatVector.fromArray(FloatVector.SPECIES_PREFERRED, new float[]{1f, 2f, 3f, 4f}, 0);
var b = FloatVector.broadcast(FloatVector.SPECIES_PREFERRED, 2f);
var c = a.mul(b);
```

---

6. Internet-Address Resolution SPI (JEP 418)

- Allows plugging in custom DNS resolvers.
- Enhances flexibility and testing of network-related features.

🧪 Foreign Function & Memory API (Second Incubator, JEP 419)

- Facilitates interop with native code using memory segments and function descriptors.
- Replaces JNI for many use cases.

Example:

```java
try (Arena arena = Arena.ofConfined()) {
    MemorySegment segment = arena.allocate(100);
    segment.set(ValueLayout.JAVA_INT, 0, 42);
    int value = segment.get(ValueLayout.JAVA_INT, 0);
}
```

> ✅ Use with `--add-modules jdk.incubator.foreign`

✅ Summary Table

Feature	Type	JEP
UTF-8 by Default	Standard	400
Simple Web Server	Standard	408
Code Snippets in JavaDoc	Standard	413
Pattern Matching for switch (2nd Preview)	Preview	420
Vector API (3rd Incubator)	Incubator	417
Foreign Function & Memory API (2nd Incubator)	Incubator	419
Internet-Address Resolution SPI	Standard	418

Would you like a **sample Java 18 project using any of these features**?

what is internet-address resolution SPI

🌐 Java 18: Internet-Address Resolution SPI (JEP 418)

🔍 What is it?

The **Internet-Address Resolution SPI (Service Provider Interface)** introduced in **Java 18 (JEP 418)** allows developers to **plug in custom implementations** for resolving hostnames and IP addresses — instead of always relying on the JVM's built-in resolution logic (usually OS DNS calls).

✅ Why was this needed?

Previously, `InetAddress.getByName()` and similar methods used the **system default resolver**, making it:

- **Hard to test** DNS logic
- Impossible to use **alternative name services** (like DNS over HTTPS, custom resolvers, or mock DNS)
- Not suitable for **containerized/cloud environments** with specialized DNS needs

---

🧩 What does the SPI provide?

You can now write your own resolver by implementing:

```java
java.net.spi.InetAddressResolverProvider
```

Which supplies:

```java
java.net.spi.InetAddressResolver
```

⚙️ How does it work?

1. **Create a class implementing** `InetAddressResolverProvider` and `InetAddressResolver`
2. **Place it in** `META-INF/services` for SPI discovery
3. **JVM uses your resolver** instead of the default

🧑‍💻 Code Example

✅ Custom DNS Resolver (Mock Example)

```java
public class MyDnsProvider extends InetAddressResolverProvider {
    @Override
    public String name() {
        return "mydns";
    }

    @Override
    public InetAddressResolver get(Configuration config) {
        return new MyCustomResolver();
    }
}
```

```java
public class MyCustomResolver implements InetAddressResolver {
    @Override
    public LookupResult[] lookupByName(String host, LookupPolicy policy) {
        System.out.println("Intercepted DNS Lookup: " + host);
        return new LookupResult[] {
            LookupResult.of("127.0.0.1") // Return localhost for everything
        };
    }

    @Override
    public String lookupByAddress(byte[] addr) {
        return "mock.local";
    }
}
```

🏁 Running it

- Put `MyDnsProvider` into `META-INF/services/java.net.spi.InetAddressResolverProvider`
- Run your app with:

```bash
```

```
java -Djava.net.spi.InetAddressResolverProvider=mydns -cp your.jar YourApp
```

📌 When to Use It

- For mocking DNS in **tests**
- To use **custom or encrypted DNS** (DoH, DoT)
- In cloud systems or containers with **non-standard resolution logic**
- For **fine-grained control** over name resolution performance or security

---

🧠 Notes

- If no custom provider is specified, JVM uses the default one.
- Part of standard Java 18 — **no preview/experimental flags** needed.

Would you like a **complete working Maven/Gradle project** using this SPI with a real example?