Key Java 11 Features

1. New `String` Methods

Java 11 added several handy methods to the `String` class:

Method	Description	Example
isBlank()	Returns true if the string is empty or contains only white spaces.	" ".isBlank() → true
lines()	Returns a stream of lines from the string, separated by line terminators.	"A\nB\nC".lines().count() → 3
strip(), stripLeading(), stripTrailing()	Similar to trim(), but Unicode-aware.	" Java ".strip() → "Java"
repeat(int count)	Returns a new string repeated count times.	"Hi".repeat(3) → "HiHiHi"

Difference Between `isBlank()` and `isEmpty()`

Feature	isBlank()	isEmpty()
Introduced In	Java 11	Java 6
Checks For	Whether a string is *empty* or contains *only whitespace characters*	Whether a string has *zero length*
Whitespace Handling	Returns true if string contains only spaces, tabs, or line breaks	Returns false if string contains spaces or tabs
Unicode Support	Unicode-aware (recognizes all kinds of whitespace)	Not Unicode-aware
Example 1	" ".isBlank() → true	" ".isEmpty() → false
Example 2	"".isBlank() → true	"".isEmpty() → true
Example 3	"Java".isBlank() → false	"Java".isEmpty() → false

Summary:

- Use `isEmpty()` when you only want to check if a string has *no characters*.
- Use `isBlank()` when you want to check if a string is *empty or only whitespace*.

Example Code:

```
public class StringCheck {
    public static void main(String[] args) {
        String str1 = "";
        String str2 = "   ";
        String str3 = "Java";

        System.out.println(str1.isEmpty()); // true
        System.out.println(str1.isBlank()); // true

        System.out.println(str2.isEmpty()); // false
        System.out.println(str2.isBlank()); // true

        System.out.println(str3.isEmpty()); // false
        System.out.println(str3.isBlank()); // false
    }
}
```

In **Java 11**, the `lines()` method was introduced in the `String` class.
It provides an easy way to **split a string into a stream of lines**.

Definition

```
public Stream<String> lines()
```

Description

The `lines()` method: Returns a **Stream of strings**, where each element is a line from the original string.

Lines are **split by line terminators**, such as

`\n` (newline – Unix/Linux)

`\r` (carriage return – old Mac)

`\r\n` (Windows)

The **line terminators are not included** in the resulting strings.

Example

```
public class LinesExample {
   public static void main(String[] args) {
      String text = "Java 11\nintroduced\nlines() method";

      text.lines().forEach(System.out::println);
   }
}
```

Output:

```
Java 11
introducedlines() method
```

Another Example — Counting Lines

```
String paragraph = "Hello\nWorld\nJava 11";long count = paragraph.lines().count();
System.out.println("Number of lines: " + count);
```

Output:

```
Number of lines: 3
```

Key Points

It's **memory-efficient**, because it uses **streams**, not arrays.

You can combine it with other stream operations like `filter()`, `map()`, or `collect()`.

Useful for reading and processing multiline strings or file contents.

Example: Filter Non-empty Lines

```
String data = "Line 1\n\nLine 3\nLine 4";
data.lines()
    .filter(line -> !line.isBlank())
    .forEach(System.out::println);
```

Output:

```
Line 1
Line 3
Line 4
```

In short:

`String.lines()` in Java 11 =
A simple and efficient way to get a Stream of lines from a multiline string.

Excellent question `strip()`, `stripLeading()`, `stripTrailing()`) were **introduced in Java 11** to handle **whitespace trimming** more accurately than the old `trim()` method.

Let’s go through them one by one

Problem with `trim()` (Before Java 11)

`String.trim()` removes **leading and trailing whitespace**, **but only ASCII spaces** (code point ≤ `\u0020`).

It **does not handle Unicode whitespace** (like non-breaking space `\u00A0`, or other language-specific spaces).

That's why Java 11 introduced `strip()` family of methods that use **Unicode-aware whitespace handling**.

---

1. `strip()`

Definition

```
public String strip()
```

Use

Removes **both leading and trailing whitespace** (Unicode-aware).

Example

```
String text = "   Java 11   ";
System.out.println(text.strip());
```

Output:

```
Java 11
```

---

2. `stripLeading()`

Definition

```
public String stripLeading()
```

Use

Removes **only leading (left-side)** whitespace.

Example

```
String text = "   Hello World   ";
System.out.println(">" + text.stripLeading() + "<");
```

Output:

```
>Hello World   <
```

3. `stripTrailing()`

Definition

```
public String stripTrailing()
```

Use

Removes **only trailing (right-side)** whitespace.

Example

```
String text = "   Hello World   ";
System.out.println(">" + text.stripTrailing() + "<");
```

Output:

```
>   Hello World<
```

Example Showing Unicode Whitespace Handling

```
String unicodeSpace = "\u2002Hello\u2002"; // 'EN SPACE' (Unicode)
System.out.println("trim(): >" + unicodeSpace.trim() + "<");
System.out.println("strip(): >" + unicodeSpace.strip() + "<");
```

Output:

```
trim(): >  Hello  <   // trim() fails (does not remove Unicode
spaces)strip(): >Hello<      // strip() works correctly
```

Summary Table

Method	Removes	Handles Unicode?	Example Output
trim()	Leading & Trailing ASCII spaces	No	" Hello " → " Hello "
strip()	Leading & Trailing Unicode spaces	Yes	" Hello " → "Hello"
stripLeading()	Leading Unicode spaces	Yes	" Hello " → "Hello "
stripTrailing()	Trailing Unicode spaces	Yes	" Hello " → " Hello"

In short:

`strip()` → Removes all leading & trailing Unicode whitespaces.

`stripLeading()` → Removes only leading spaces.

`stripTrailing()` → Removes only trailing spaces.

`trim()` (old) → Works only for ASCII spaces.

What is `repeat()` in Java 11?

The `repeat()` method allows you to **repeat a string multiple times** without writing loops or manual concatenation.

Definition

```
public String repeat(int count)
```

Introduced in

Java 11

Purpose

To return a **new string** consisting of the original string **repeated `count` times**.

Example

```
public class RepeatExample {
    public static void main(String[] args) {
        String word = "Java ";
        String repeated = word.repeat(3);
        System.out.println(repeated);
    }
}
```

Output:

```
Java Java Java
```

Key Points

- If `count = 0` → returns an **empty string** `""`
- If `count = 1` → returns the **same string**

Throws `IllegalArgumentException` if `count` is **negative**

Examples

1 ☐ Basic Usage

```
"Hi ".repeat(4);
```

Output: `"Hi Hi Hi Hi "`

2 ☐ Repeating Symbols

```
"*".repeat(10);
```

Output: `"**********"`

3 ☐ Combine with Other Methods

```
String line = "-".repeat(20);
System.out.println(line);
System.out.println("Java 11 Features");
System.out.println(line);
```

Output:

```
--------------------Java 11 Features
--------------------
```

Error Example

```
"Hi".repeat(-2);
```

Throws: `java.lang.IllegalArgumentException: count is negative`

Interview Notes Summary

Property	Description
Method	String repeat(int count)
Since	Java 11

Property	Description
Returns	New string repeated count times
If count = 0	Returns "" (empty string)
If count < 0	Throws IllegalArgumentException
Common Uses	Printing patterns, separators, or repeated text

Shortcut Memory Tip

repeat() = easy string loop — no manual concatenation.”

What is LVTI (Local Variable Type Inference)?

LVTI stands for **Local Variable Type Inference** —
it allows Java to **infer the type of a local variable automatically** from its initializer expression, using the keyword `var`.

Introduced in:

Java 10 (and available in **Java 11** too)

Syntax

```
var variableName = value;
```

The compiler **infers** the data type of `variableName` from the type of the value assigned to it.

Example

```
public class LVTIExample {
    public static void main(String[] args) {
        var name = "Java";       // inferred as String
        var version = 11;        // inferred as int
        var price = 99.99;       // inferred as double

        System.out.println(name + " " + version + " : " + price);
    }
}
```

Output:

```
Java 11 : 99.99
```

- `name → String`
- `version → int`
- `price → double`

Key Rules of LVTI

Rule	Description
1	☐ var **can only be used for local variables** (inside methods, constructors, or blocks).
2	☐ The variable **must be initialized** at the time of declaration.
3	☐ The **type is inferred at compile-time**, not runtime.
4	☐ You **cannot** use var for **fields, parameters, or return types**.
5	☐ You **cannot** assign null because type can’t be inferred.

Invalid Examples

```
var x;          //      Error  –  no initializer, type unknownvar y = null;   //      Error  –  type cannot be inferred
```

Valid Use Cases

1 ☐ Inside Loops

```
for (var i = 0; i < 5; i++) {
    System.out.println(i);
}
```

2 ☐ With Collections

```
var list = List.of("Java", "Python", "Go");for (var item : list) {
    System.out.println(item);
}
```

3 ☐ With Streams

```
var stream = list.stream().filter(s -> s.length() > 3);
stream.forEach(System.out::println);
```

What Happens Internally

The `var` keyword **does not make Java dynamically typed** —
it's still **statically typed**.
The compiler just infers the type during compilation and replaces `var` with the actual type.

For example:

```
var message = "Hello";
```

Compiler converts it internally to:

```
String message = "Hello";
```

Benefits

- Reduces code clutter (especially with generics).
- Improves readability when type is obvious.
- Encourages concise code.

When NOT to Use `var`

When it makes code **less readable or ambiguous**.

Example:

```
var data = getData();  //      unclear what type data is
```

Better:

```
String data = getData();
```

Interview Notes Summary

Feature	LVTI (var)
Full Form	Local Variable Type Inference
Introduced In	Java 10
Keyword	var
Purpose	Let compiler infer local variable type
Scope	Only for local variables (not fields, params, returns)

Feature	LVTI (var)
Initialization	Mandatory
Type Checking	At compile-time (static typing)
Not Allowed With	null, uninitialized vars

Example for Notes

```
var msg = "Java";   // Stringvar num = 10;       // intvar list = List.of(1, 2, 3); // List<Integer>
```

Let’s understand `Files.readString()` and `Files.writeString()` clearly with examples and key points.

Introduced in:

Java 11 (in `java.nio.file.Files` class)

These methods make **file reading and writing much simpler** compared to older approaches like using `BufferedReader` or `FileWriter`.

---

1 ☐ Files.readString()

Definition

```
public static String readString(Path path) throws IOException
```

Purpose

Reads the **entire content of a file** into a single **String**.

Example

```
import java.nio.file.*;
public class ReadFileExample {
    public static void main(String[] args) throws Exception {
        Path filePath = Path.of("example.txt");
        String content = Files.readString(filePath);
        System.out.println(content);
    }
}
```

If `example.txt` contains:

```
Hello Java 11!File readString() is easy.
```

Output:

```
Hello Java 11!File readString() is easy.
```

Overloaded Version

You can also specify a **character encoding**:

```
Files.readString(filePath, StandardCharsets.UTF_8);
```

2 ☐ Files.writeString()

Definition

```
public static Path writeString(Path path, CharSequence content, OpenOption...
options) throws IOException
```

Purpose

Writes text content (a `String` or `CharSequence`) **directly to a file**.

Example

```
import java.nio.file.*;
public class WriteFileExample {
    public static void main(String[] args) throws Exception {
        Path filePath = Path.of("output.txt");
        Files.writeString(filePath, "Hello from Java 11!");
    }
}
```

create the file `output.txt` (if it doesn’t exist) and **write** the text `"Hello from Java 11!"`.

Append Example

To **append** text instead of overwriting:

```
Files.writeString(filePath, "\nAppended line", StandardOpenOption.APPEND);
```

Key Differences from Older Methods

Old Way (Before Java 11)	Java 11 Way
Files.readAllLines(path) returns List<String>	Files.readString(path) returns a single String
BufferedWriter writer = Files.newBufferedWriter(path)	Files.writeString(path, content) (one line!)
Needed loops to combine lines	No need — direct String read/write

---

Key Points for Interview

Method	Description	Returns
Files.readString(Path)	Reads all text from a file into one string	String
Files.writeString(Path, CharSequence)	Writes text to file (creates/overwrites by default)	Path
Charset Supported?	Yes, StandardCharsets.UTF_8 by default	
Options	Use StandardOpenOption.APPEND, CREATE, TRUNCATE_EXISTING, etc.	
Exceptions	Throws IOException on error	

Notes

Both methods are **static** methods of `java.nio.file.Files`.

Use `Path.of("filename")` to create a `Path` object (Java 11 shortcut).

Both use **UTF-8 encoding by default**.

Ideal for **small to medium files** (not huge ones).

Quick Comparison

Operation	Old Code	Java 11 Code
Read file	new String(Files.readAllBytes(Paths.get("file.txt")))	Files.readString(Path.of("file.txt"))
Write file	Files.write(Paths.get("file.txt"), content.getBytes())	Files.writeString(Path.of("file.txt"), content)

Interview Notes Summary

Feature	Files.readString()	Files.writeString()
Introduced	Java 11	Java 11
Package	java.nio.file	java.nio.file
Default Charset	UTF-8	UTF-8
Returns	String	Path
Use Case	Read file as text	Write text to file
Overwrites File?	N/A	Yes (unless APPEND used)
Throws	IOException	IOException

One-Line Summary

**Files.readString() reads entire file as a String;
Files.writeString() writes text directly to a file — simple, clean, and UTF-8 by default.”**

4. New HTTP Client (Standardized)

The new HTTP Client API (introduced as incubator in Java 9, finalized in Java 11) replaces the old `HttpURLConnection`.

Supports:

HTTP/1.1 and HTTP/2

Asynchronous (non-blocking) requests

WebSocket support

Example:

```
HttpClient client = HttpClient.newHttpClient();HttpRequest request =
HttpRequest.newBuilder()
        .uri(URI.create("https://example.com"))
        .build();
HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
System.out.println(response.body());
```

5. Running Java Files Without Compilation

You can now run `.java` files directly using `java` command — no need to compile first.

```
java HelloWorld.java
```

This is great for scripting or quick tests.

6. New `Optional.isEmpty()` Method

Simplifies checking for absence of value:

```
Optional<String> name = Optional.empty();if (name.isEmpty()) {
    System.out.println("No value present");
}
```

7. Deprecated and Removed Features

Removed Java EE and CORBA modules: (`java.xml.ws`, `java.xml.bind`, etc.)

Nashorn JavaScript Engine removed

Deprecated: `Applets` and `Pack200` API

---

8. Flight Recorder

A low-overhead data collection framework built into the JVM to record diagnostic and profiling data — useful for production monitoring.

---

9. Z Garbage Collector (Experimental)

A **low-latency garbage collector** introduced as experimental.
Goal: Pause times not exceeding 10ms, even with large heaps.

Enable with:

```
java -XX:+UnlockExperimentalVMOptions -XX:+UseZGC
```

---

10. TLS 1.3 Support

Improved security performance and encryption with TLS 1.3 added to Java 11.

Summary Table

Category	Feature
Strings	isBlank(), lines(), strip(), repeat()
Lambda	var in parameters
Files	readString(), writeString()
HTTP	New HTTP Client API
Optional	isEmpty()

Category	Feature
Runtime	Run .java without compilation
JVM	Flight Recorder, ZGC
Security	TLS 1.3
Removed	CORBA, Java EE modules, Nashorn

======= The End JAVA 11 Feature==========

1 ☐ What are the main new features introduced in Java 11?

Key Java 11 features:

- `var` for lambda parameters (extends Java 10 LVTI)
- New `String` methods: `isBlank()`, `lines()`, `strip()`, `stripLeading()`, `stripTrailing()`, `repeat()`
- New file methods: `Files.readString()` and `Files.writeString()`
- `Path.of()` method (simpler than `Paths.get()`)
- HTTP Client API (standardized from incubator in Java 9/10)
- Running single-file Java programs without `javac` (e.g., `java Hello.java`)
- Removed Java EE and CORBA modules
- Flight Recorder & Mission Control (for performance monitoring)

---

2 ☐ Difference between `trim()` and `strip()` methods in Java 11?

Method	Removes	Unicode Aware	Remarks
trim()	ASCII whitespace only	No	Older method, not Unicode-aware

Method	Removes	Unicode Aware	Remarks
strip()	Unicode whitespace	Yes	Recommended replacement for trim()
stripLeading()	Leading Unicode whitespace only		—
stripTrailing()	Trailing Unicode whitespace only		—

3 ☐ What is the `lines()` method in `String`?

Splits a string into a **Stream of lines** separated by `\n`, `\r`, or `\r\n`.

Terminators are **not included** in the result.

```
"Java\n11\nFeatures".lines().forEach(System.out::println);
```

Output:

```
Java11
Features
```

4 ☐ What does the `isBlank()` method do?

Returns `true` if the string is **empty** or contains **only whitespace (including Unicode)**.

```
"   ".isBlank();  // true"Java".isBlank(); // false
```

5 ☐ Explain `repeat()` method in Java 11.

Repeats the string **n times**:

```
"Hi ".repeat(3); // "Hi Hi Hi "
```

If `count=0` → returns empty string

If negative → throws `IllegalArgumentException`

6 ☐ Explain `Files.readString()` and `Files.writeString()`.

Introduced in Java 11 (java.nio.file.Files)

```
Path path = Path.of("test.txt");String data = Files.readString(path);
Files.writeString(path, "Hello Java 11!");
```

Simplifies file read/write operations

Uses UTF-8 by default

Suitable for small/medium files

---

7 ☐ What is LVTI (Local Variable Type Inference)?

Introduced in Java 10, available in Java 11

Allows compiler to **infer variable type** using `var`.

```
var message = "Hello"; // inferred as String
```

Still **statically typed** (type known at compile-time).

Only for **local variables**, not fields, method params, or returns.

Must be initialized at declaration.

---

8 ☐ Can we use `var` with null or without initialization?

No, both are invalid:

```
var x;        // Error – type not inferredvar y = null;  // Error – type unknown
```

---

9 ☐ What is the new HTTP Client API in Java 11?

Introduced in `java.net.http` package.

Replaces the old `HttpURLConnection`.

Supports **synchronous**, **asynchronous**, and **WebSocket** communication.

```
HttpClient client = HttpClient.newHttpClient();HttpRequest request =
HttpRequest.newBuilder()
        .uri(URI.create("https://api.github.com"))
        .build();

HttpResponse<String> response =
        client.send(request, HttpResponse.BodyHandlers.ofString());
System.out.println(response.body());
```

Supports: HTTP/1.1, HTTP/2, Async (via `CompletableFuture`).

`Path.of()`?

Simpler alternative to `Paths.get()` introduced in Java 11.

```
Path path = Path.of("test.txt");
```

Equivalent to:

```
Path path = Paths.get("test.txt");
```

---

11 ☐ How to run a Java source file without compilation in Java 11?

You can directly run a `.java` file:

```
java Hello.java
```

> The JVM **compiles and runs** it internally (good for scripting & small programs).

---

12 ☐ What modules were removed in Java 11?

Removed Java EE and CORBA modules:

- `java.xml.ws`, `java.xml.bind`, `java.activation`
- `java.corba`, `java.transaction`, `java.se.ee`, etc.

These are now available as **separate external dependencies** (Jakarta EE).

---

13 ☐ What is the difference between Java 8 and Java 11?

Feature	Java 8	Java 11
LTS	Yes	Yes
var keyword		(local vars)
New String methods		
New File methods		
HTTP Client		
Single-file launch		
Removed modules	N/A	(EE, CORBA)
Performance	Good	Improved (GC, startup, memory)

---

14 ☐ What is the default character encoding used by `Files.readString()` and `writeString()`?

UTF-8 by default.

---

15 ☐ What are Flight Recorder and Mission Control in Java 11?

Java Flight Recorder (JFR): a built-in profiler that records JVM performance data.

Java Mission Control (JMC): GUI tool to analyze JFR recordings for performance tuning.

Very useful in production monitoring and debugging.

---

16 ☐ Can we use `var` in lambda expressions in Java 11?

Yes — Java 11 allows using `var` for lambda parameters:

```
(var x, var y) -> x + y
```

¸ Either all or none of the lambda parameters must use `var`.

---

17 ☐ What happens if `Files.writeString()` is called on an existing file without options?

By default, it **overwrites** the file.
Use `StandardOpenOption.APPEND` to append content.

---

18 ☐ What is ZGC (Z Garbage Collector)?

ZGC is a **low-latency garbage collector** introduced experimentally in Java 11.

Pause times typically under **10 ms**, even for large heaps.

Suitable for **large-memory, low-latency** systems.

Enable with:

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

---

19 ☐ What is the `Optional.isEmpty()` method?

Introduced in Java 11:

```
Optional<String> opt = Optional.empty();if (opt.isEmpty()) {
    System.out.println("No value present");
}
```

Cleaner alternative to `!opt.isPresent()`.

---

20 ☐ Can Java 11 code run on Java 8 JVM?

No.
Bytecode compiled with Java 11 (`--release 11`) **won't run** on Java 8 JVM due to version mismatch.
However, you can **compile for Java 8 target**:

```
javac --release 8 MyClass.java
```

Summary — Quick Interview Notes

Category	New Java 11 Feature
Language	var in lambdas, type inference
String	isBlank(), lines(), strip(), repeat()

Category	**New Java 11 Feature**
Files API	readString(), writeString()
Path	Path.of()
HTTP	New HttpClient (HTTP/2, async)
JVM	ZGC, Flight Recorder
Optional	isEmpty()
Others	Removed Java EE modules, single-file source execution

Would you like me to make this into a **formatted "Java 11 Interview Questions & Answers" PDF (2-page)** so you can revise it quickly before interviews?

You said:

please find most asked interview question from java 11 verwsion for 9 years experience developer provide list with answer

ChatGPT said:

Here are some *senior / 9-year-experience level* Java 11 interview questions + sample answers. These are deeper, focusing both on language/API changes in Java 11 *and* architecture/JVM/performance tradeoffs that someone at that level is expected to know. You can pick and prepare those most relevant to your domain (web, backend, performance, etc.).

---

1. What are the JVM / GC improvements in Java 11? Explain ZGC & Epsilon GC.

Answer:

> Java 11 introduced **Z Garbage Collector (ZGC)** as an experimental, low-latency, scalable GC. It aims to keep pause times very small (on the order of <10ms) even for very large heaps (hundreds of gigabytes to terabytes). DEV Community+3MangoHost+3GeeksforGeeks+3
>
> Also **Epsilon GC**, a "no-op" / "dead" GC (i.e. allocation works but reclamation does not) for testing / benchmarking scenarios where you want to measure memory allocation behavior without interference of GC. Not for production generally. examlabs.com+3Baeldung on Kotlin+3MangoHost+3
>
> Other changes: performance optimizations on AArch64, improved intrinsics for string and array operations, math functions (sin, cos, log) on those architectures. Baeldung on Kotlin

-

Why this matters for senior devs:

Choosing the right GC (tradeoffs between throughput, latency, footprint)

Understanding whether ZGC is production-ready for your vendor / version, fallback / tuning strategies

Being able to diagnose GC pauses, do GC logging, heap sizing etc.

---

2. What are "dynamic class-file constants" (JEP 309) in Java 11?

Answer:

Java 11 added support for a new constant-pool form called `CONSTANT_Dynamic` (part of JEP 309). Baeldung on Kotlin+1

Purpose: delayed resolution of constant pool entries via bootstrap methods. This improves class file size, reduces runtime overhead for some constants (i.e. some constants that are expensive / rarely used don't have to be eagerly resolved). Also helps for code generators / frameworks that generate lots of constant pool entries. Baeldung on Kotlin

-

---

3. Explain Nest-Based Access Control (Nestmates) in Java 11. Why was it added?

Answer:

In Java 11, new class attributes `NestHost` and `NestMembers` were added, along with methods `Class.getNestHost()`, `Class.getNestMembers()`, `Class.isNestmateOf(...)`. iqcode.com+1

Purpose: better handling of Java's inner/nested classes: allow classes that are part of the same "nest" (outer class + inner classes) to have private access to each other without synthetic bridge/accessor methods. This improves encapsulation, reduces bytecode clutter, reduces overhead from synthetic members. iqcode.com

-

---

4. Which modules were deprecated or removed in Java 11, and what implications for large enterprise systems?

Answer:

Java 11 removed the Java EE / CORBA modules: e.g. `java.xml.ws`, `java.xml.bind`, `java.activation`, `java.corba`, etc. These were deprecated in earlier versions and removed in Java 11. MangoHost+1

Implications: enterprise applications that used JAXB, JAX-WS, CORBA etc will have to add external dependencies (e.g. via Maven), adjust build and deployment. In migration: ensure tests, integrations don't fail due to missing modules, licensing, differences in versions. Also affects security / patching (external libraries may not be updated as automatically).

---

5. What are some of the new / enhanced String API methods in Java 11? Why do they matter in practical code?

Answer:

Some of the newer String API methods:

Method	What it does
isBlank()	checks if string is empty or contains only whitespace (including Unicode whitespace)
lines()	returns a stream of lines from a string (splits on line terminators)
strip(), stripLeading(), stripTrailing()	Unicode-aware trimming vs the older trim()
repeat(int count)	repeat the string multiple times
Also String::indent introduced in a later version but related in this space	

Why practical:

Improved code readability and less boilerplate.

Unicode correctness (important in global apps).

Stream-friendly APIs make processing multi-line data easier.

Less error prone when dealing with whitespace, formatting, etc.

---

6. What is Java Flight Recorder (JFR) and Java Mission Control (JMC)? How are they used?

Answer:

- **Java Flight Recorder** is a profiling / monitoring feature integrated into the JDK. It allows low-overhead recording of JVM events, performance metrics, GC events, method profiling, thread behavior etc. Available in Java 11 (with some vendor licensing conditions but more open with OpenJDK builds) DEV Community+1.

- **Java Mission Control** is a tool to load, analyze, and visualize the data captured by JFR. Useful for diagnosing performance issues, memory leaks, thread contention, etc.

Use by senior devs:

- In production or staging / performance testing, to gather flight recordings to find bottlenecks.

- Often used in conjunction with GC logs, heap dumps, to get holistic view.

---

7. How would you handle migration from Java 8 to Java 11 for a large codebase? What are key things to watch out for?

Answer:

Some migration concerns:

- Removed modules (as above). Dependencies on deprecated / removed Java EE / CORBA modules need replacement.

- Compatibility: ensure third-party libraries support Java 11. Build tools (Maven / Gradle) updated for Java 11 support.

- Tools: continuous integration, code coverage, static analysis must be updated.

- JVM flags: deprecated flags removed; default GC changed? G1 is default since Java 9. Some flags may have changed meaning or been removed.

- New behavior in String API, unicode etc may affect existing logic (especially whitespace handling)

- Performance / memory: benchmarking required under load, GC tuning

- Testing: regression / integration / serialization / reflection / classloading / module path issues.

---

8. Explain how to use HTTP Client API in Java 11. What's new compared to older ways (HttpURLConnection)?

Answer:

Java 11 standardizes the `java.net.http.HttpClient` API (from earlier incubator modules) to provide a modern HTTP client: synchronous & asynchronous requests, HTTP/2 support, better API design. Baeldung on Kotlin.

Advantages: simpler, cleaner API; non-blocking I/O; built in support for WebSocket; better handling of redirects etc; easier to set headers, timeouts etc. Less boilerplate than `HttpURLConnection`.

---

9. What are Compact Strings, and how do they improve memory usage/ performance?

Answer:

`Compact Strings` was introduced earlier (Java 9), but in Java 11 there are continued improvements to string implementation. The idea is strings that can be encoded in ISO-8859-1 or Latin-1 are stored more efficiently (one byte per character) instead of using two bytes per character for all strings. This reduces memory footprint for typical strings (ASCII / Latin1). MangoHost+1

---

10. What is toArray(IntFunction<T[]>) in Collections, and why is it useful?

Answer

Java 11 adds a new overload for `Collection.toArray(IntFunction<T[]>)` so you can do something like `List of elements .toArray(String[]::new)` which is more type-safe and avoids needing casting. examlabs.com+2MangoHost+2.

It simplifies code where you need to convert collection to array, especially with generics.