Strings & Regular Expressions in Java

1️⃣ String Class in Java

What is a String?

A String represents a **sequence of characters**.

```
String s = "Java";
```

Key Characteristics

- String is **immutable**
- Stored as **UTF-16 characters**
- String is a **final class**
- Part of java.lang (no import required)

1.1 String Immutability (VERY HIGH EXAM WEIGHT)

Once a String object is created, **its content cannot be changed**.

```
String s = "Java";
s.concat(" World");
System.out.println(s); // Java
```

✔ A **new object** is created
✔ Original object remains unchanged

Why Strings Are Immutable

- **Security** (used in class loaders, DB URLs, file paths)

- **Thread safety**
- **Performance via caching**
- **String Pool optimization**

1.2 String Constant Pool (SCP)

The **String Constant Pool** is a special memory area inside the **heap**.

```
String s1 = "Java";
String s2 = "Java";
```

✔ Only **one object** is created
✔ `s1 == s2` → `true`

Using `new` Keyword

```
String s3 = new String("Java");
```

✔ Creates **two objects**:

1. One in SCP (if not already present)
2. One in heap

```
s1 == s3 // false
```

`intern()` Method

```
String s4 = s3.intern();
```

✔ Returns SCP reference
✔ Used to optimize memory

1.3 String Creation Summary

Syntax	Objects Created
`"Java"`	1 (SCP)
`new String("Java")`	2
`"A"+"B"`	1 (compile-time)
`"A"+x`	Runtime object

1.4 `==` vs `equals()` (EXAM FAVORITE)

Operator	Meaning
`==`	Reference comparison
`equals()`	Content comparison

```
String a = new String("Java");
String b = new String("Java");

a == b         // false
a.equals(b)   // true
```

1.5 Important String Methods

Method	Purpose
`length()`	Character count
`charAt(int)`	Character at index
`substring()`	Extract substring
`indexOf()`	Find position
`toUpperCase()`	Case conversion

`trim()`	Remove spaces
`replace()`	Replace characters
`split()`	Tokenization

`substring()` (Classic Trap)

```
String s = "Java";
s.substring(1,3); // "av"
```

✔ Start index inclusive
✔ End index exclusive

2️⃣ `StringBuilder`

Why `StringBuilder`?

To handle **mutable strings** efficiently.

```
StringBuilder sb = new StringBuilder("Java");
sb.append(" World");
System.out.println(sb); // Java World
```

Key Characteristics

- **Mutable**
- **Not thread-safe**
- Faster than `StringBuffer`
- Introduced in **Java 1.5**

Internal Working

- Uses a **resizable char array**
- Default capacity = `16`
- Capacity grows as:

```
newCapacity = (oldCapacity * 2) + 2
```

Important Methods

Method	Purpose
`append()`	Add text
`insert()`	Insert at index
`delete()`	Remove characters
`reverse()`	Reverse sequence
`capacity()`	Current capacity
`ensureCapacity()`	Manual resize

Equality Behavior (Very Important)

```
StringBuilder sb1 = new StringBuilder("Java");
StringBuilder sb2 = new StringBuilder("Java");

sb1.equals(sb2); // false
```

✔ `equals()` is **not overridden**
✔ Reference comparison only

3️⃣ `StringBuffer`

Purpose

Same as `StringBuilder`, but **thread-safe**.

```
StringBuffer sb = new StringBuffer("Java");
sb.append(" World");
```

Key Characteristics

- **Mutable**
- **Thread-safe**
- Slower than `StringBuilder`
- Introduced in **Java 1.0**

Thread Safety

- Methods are **synchronized**
- Suitable for **multi-threaded environments**

4️⃣ `String` vs `StringBuilder` vs `StringBuffer`

Feature	String	StringBuilder	StringBuffer
Mutability	❌ Immutable	✔ Mutable	✔ Mutable
Thread-safe	✔	❌	✔
Performance	Slow	Fastest	Slower
SCP	✔	❌	❌
Introduced	1.0	1.5	1.0

5️⃣ Conversion Between String Types

```
String s = "Java";
StringBuilder sb = new StringBuilder(s);
String s2 = sb.toString();
```

✔ Commonly tested

6️⃣ Why `StringBuilder` Doesn’t Override `equals()`

- Mutable objects should not be used as map keys
- Equality based on content would break hashing contracts
- Hence `equals()` remains reference-based

7️⃣ Regular Expressions (Regex) in Java

What is Regex?

A **pattern-matching mechanism** for text processing.

```
import java.util.regex.*;
```

7.1 Core Regex Classes

Class	Role
`Pattern`	Compiled regex
`Matcher`	Performs matching
`PatternSyntaxException`	Invalid regex

7.2 Basic Regex Flow

```
Pattern p = Pattern.compile("ab");
```

```
Matcher m = p.matcher("ababbaba");

while(m.find()) {
    System.out.println(m.start());
}
```

7.3 Predefined Character Classes

Regex	Meaning
`\d`	Digit
`\D`	Non-digit
`\w`	Word character
`\W`	Non-word
`\s`	Whitespace
`\S`	Non-whitespace

7.4 Quantifiers (HIGH EXAM VALUE)

Symbol	Meaning
`+`	One or more
`*`	Zero or more
`?`	Zero or one
`{n}`	Exactly n
`{n,}`	At least n
`{n,m}`	Between n and m

7.5 Character Classes

```
[a-z]   → lowercase
[A-Z]   → uppercase
[0-9]   → digits
[^a-z]  → negation
```

7.6 Anchors

Anchor	Meaning
`^`	Start of line
`$`	End of line
`\b`	Word boundary

7.7 `String` vs `Pattern` Regex Methods

`String` Methods

```
s.matches("regex");
s.split("regex");
s.replaceAll("regex","x");
```

✔ Entire string must match

`Pattern / Matcher`

✔ Used for **multiple matches**
✔ Better performance

7.8 Common Regex Examples (Exam Useful)

Requirement	Regex
Mobile number	`[6-9][0-9]{9}`

Email	`[a-zA-Z0-9._]+@[a-zA-Z]+\\.[a-z]{2,}`
Password	`(?=.*\\d)(?=.*[A-Z]).{8,}`

8️⃣ Regex Compilation Flags

```
Pattern.compile("abc", Pattern.CASE_INSENSITIVE);
```

Flag	Purpose
`CASE_INSENSITIVE`	Ignore case
`MULTILINE`	Line-based matching
`DOTALL`	`.` matches newline

Certification Takeaways (VERY IMPORTANT)

- `String` is immutable & final
- SCP exists only for `String`
- `StringBuilder` is fastest but not thread-safe
- `StringBuffer` is synchronized
- `equals()` behavior differs
- Regex matching rules are strict
- `matches()` requires full match
- Quantifiers & character classes are heavily tested

Modern Relevance

- Prefer `StringBuilder` in loops
- Use `Pattern` for heavy regex usage
- Avoid `StringBuffer` unless thread safety is required
- Regex is core to validation, parsing, logs, APIs

1) Validate: 10-digit Indian mobile number

Rule: starts with 6–9, then 9 digits.

```
String mobile = "9876543210";
boolean ok = mobile.matches("[6-9][0-9]{9}");
System.out.println(ok); // true
```

- `[6-9]` → first digit 6/7/8/9
- `[0-9]{9}` → exactly 9 digits after that
- `matches()` must match the **entire string**

---

2) Validate: Simple email (exam-level, not perfect RFC)

```
String email = "name.surname_12@gmail.com";
boolean ok =
email.matches("[a-zA-Z0-9._]+@[a-zA-Z]+\\.[a-zA-Z]{2,}");
System.out.println(ok); // true
```

- `+` → one or more
- `\\.` → literal dot (`.` is special in regex, so we escape it)

---

3) Extract all numbers from a sentence (`Pattern` + `Matcher`)

Use this when you want **multiple matches** (not just true/false).

```
import java.util.regex.*;

String text = "Order 512 delivered in 3 days, cost 1499.";
Pattern p = Pattern.compile("\\d+"); // one or more digits
Matcher m = p.matcher(text);
```

```
while (m.find()) {
    System.out.println(m.group());  // 512, 3, 1499
}
```

- `find()` scans the string for the next match
- `group()` returns the matched substring

---

4) Find positions (start/end index) of matches

```
import java.util.regex.*;

String text = "abc123xyz45";
Matcher m = Pattern.compile("\\d+").matcher(text);

while (m.find()) {
    System.out.println(m.group() + " at " + m.start() + "-" +
(m.end()-1));
}
// 123 at 3-5
// 45 at 9-10
```

- `start()` is inclusive
- `end()` is exclusive

---

5) Replace multiple spaces with a single space

```
String s = "Java    is   fun";
String cleaned = s.replaceAll("\\s+", " ");
System.out.println(cleaned); // "Java is fun"
```

- `\\s+` → one or more whitespace characters

---

6) Split by comma with optional surrounding spaces

```
String line = "apple,  banana ,orange,   mango";
String[] parts = line.split("\\s*,\\s*");

for (String p : parts) System.out.println(p);
```

- `\\s*` → zero or more spaces
- Good for CSV-like input cleaning

---

7) Validate password with lookaheads (common interview + advanced)

Rule: min 8 chars, at least 1 digit, 1 uppercase, 1 lowercase.

```
String pwd = "A1bcdefg";
boolean ok = pwd.matches("(?=.*\\d)(?=.*[A-Z])(?=.*[a-z]).{8,}");
System.out.println(ok); // true
```

- `(?=.*\\d)` → must contain a digit somewhere
- `.{8,}` → at least 8 characters

---

8) Anchors: match only at start/end

Validate exactly “Java” (not “Java11”)

```
System.out.println("Java".matches("^Java$"));     // true
System.out.println("Java11".matches("^Java$"));   // false
```

- `^` start of string, `$` end of string

`Pattern` and `Matcher` in Java (java.util.regex)

Java regex is built around two core classes:

- **`Pattern`** = the *compiled* regular expression (the regex “engine object”)
- **`Matcher`** = the *stateful worker* that runs the pattern against a given input string

Think:

- `Pattern` is **what** to search for
- `Matcher` is **where/how** you search in a specific text

---

1) `Pattern` class

What it does

- Compiles a regex into an efficient internal form.
- Reusable across many inputs (good for performance).

How you create it

```
Pattern p = Pattern.compile("\\d+");
```

Key methods (cert + practical)

- `compile(regex)` / `compile(regex, flags)`
- `matcher(input)` → gives a `Matcher`
- `pattern()` → returns the regex as String
- `split(input)` → splits using the pattern

---

2) `Matcher` class

What it does

- Applies a `Pattern` to a specific input.
- Maintains **search state** (current position, last match, groups).
- Lets you:
  - **find** occurrences
  - check **full match** / **prefix match**
  - extract **groups**
  - get match **start/end indices**
  - do **replace** operations

Key methods (cert + practical)

- `find()` → find next occurrence anywhere in the input
- `matches()` → entire input must match (same idea as `String.matches()`)
- `lookingAt()` → matches from the beginning only (prefix match)
- `group()` / `group(n)` → matched text (whole or group)
- `start()` / `end()` → match boundaries
- `replaceAll(repl)` / `replaceFirst(repl)`
- `reset(newInput)` → reuse same matcher on new input

Example A: Extract all numbers with positions

```
import java.util.regex.*;

public class Demo {
    public static void main(String[] args) {
        String text = "Order 512 delivered in 3 days, cost 1499.";

        Pattern p = Pattern.compile("\\d+");  // one or more digits
        Matcher m = p.matcher(text);          // matcher tied to
this input

        while (m.find()) {                    // find next match
            System.out.println(
                "Match: " + m.group() +
                ", start=" + m.start() +
                ", end=" + (m.end() - 1)
            );
        }
    }
}
```

What happens internally

- `Pattern.compile("\\d+")` builds a compiled regex.
- `p.matcher(text)` creates a matcher with a cursor at the start.
- Each `m.find()` moves the cursor forward to the next match.
- `group()` returns the current matched substring.
- `start()` / `end()` return indices (end is exclusive).

Example B: `find()` vs `matches()` vs `lookingAt()`

```
import java.util.regex.*;
```

```
public class Compare {
    public static void main(String[] args) {
        Pattern p = Pattern.compile("\\d+");

        System.out.println(p.matcher("abc123xyz").find());      //
true  (123 exists)
        System.out.println(p.matcher("abc123xyz").matches());   //
false (whole string not digits)
        System.out.println(p.matcher("123xyz").lookingAt());    //
true  (starts with digits)
        System.out.println(p.matcher("xyz123").lookingAt());    //
false (doesn't start with digits)
    }
}
```

Summary

- `find()` → "Is there a match anywhere?"
- `matches()` → "Does the whole string match?"
- `lookingAt()` → "Does it match starting at index 0?"

Example C: Capturing groups (extract username + domain)

```
import java.util.regex.*;

public class GroupsDemo {
    public static void main(String[] args) {
        String email = "name.surname_12@gmail.com";

        Pattern p =
Pattern.compile("([a-zA-Z0-9._]+)@([a-zA-Z0-9.-]+)");
        Matcher m = p.matcher(email);

        if (m.matches()) {
            System.out.println("Full: " + m.group(0)); // whole
match
            System.out.println("User: " + m.group(1)); // group 1
            System.out.println("Host: " + m.group(2)); // group 2
        }
    }
}
```

Notes

- Parentheses `(...)` create **capturing groups**
- `group(0)` is always the whole match
- `group(1)`, `group(2)` are the captured parts

Example D: Replacement using `Matcher`

```
import java.util.regex.*;

public class ReplaceDemo {
    public static void main(String[] args) {
        String text = "User: Vishal, Phone: 9876543210";

        Pattern p = Pattern.compile("\\d{10}");
        Matcher m = p.matcher(text);

        String masked = m.replaceAll("XXXXXXXXXX");
        System.out.println(masked);
    }
}
```

When to use `Pattern/Matcher` instead of `String.matches()`?

Use `String.matches()` when:

- you need only **true/false**
- you want to validate the **entire string**

Use `Pattern/Matcher` when:

- you need **multiple matches**
- you need **groups**
- you need **positions**
- you need **performance** (compile once, apply many times)

Mini performance note (practical + exam-safe)

If you are checking many inputs with the same regex, compile once:

```
static final Pattern MOBILE = Pattern.compile("[6-9][0-9]{9}");

boolean ok = MOBILE.matcher("9876543210").matches();
```

This avoids recompiling the regex repeatedly.