Unit-5

Utility classes and Regular expressions

Unit-5 content

- **Utility Classe**s: Date, Calendar, Gregorian Calendar, TimeZone, SimpleTimeZone, Locale, Random.
- **Regular Expressions:** Regular Expression Processing: Pattern, Matcher, Regular Expression Syntax, Demonstrating Pattern Matching, Two Pattern - Matching Options, Exploring Regular Expressions, Reflection.

Regular Expression

- **Regular expressions** are a great tool to process strings .Using them, you can set a pattern that a string or substring should correspond to.
- A **regular expression** is written using alphabetic and numeric characters, also **metacharacters** are used that are characters that have a special meaning (are only used in the syntax of regular expressions).

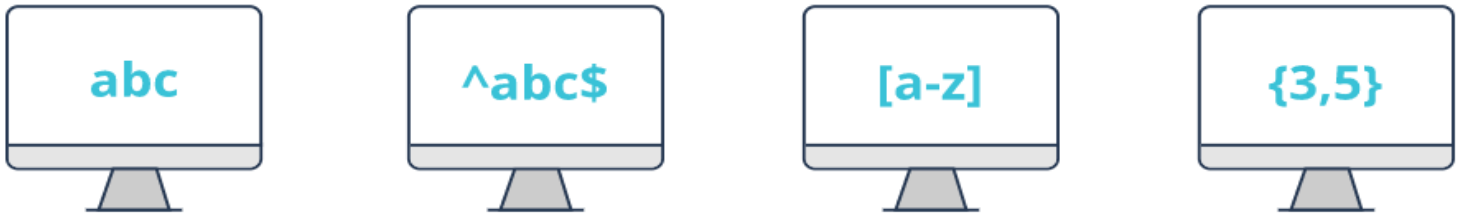
Searching for Information

The following options to search for information exist:

- Searching for a word
- Searching for words that start with certain characters
- Searching for words that end with certain characters

Checking for Match : Checking for a match to a certain pattern. For example, validating a phone number, email address, password, etc.

Components of regular expressions



Character | Metacharacter | Character class | Quantifiers

Metacharacters to search for a match of the strings or text boundaries

- ^ — string beginning.
- $ — string end.
- \b — word boundary.
- \B — not a word boundary.
- \A — input start.
- \G — end of the previous match.
- \Z — input end, except for the end terminator, if applicable.
- \z — input end.

Metacharacters to search for character classes

- \d — numeric character.
- \D — non-numeric character.
- \s — whitespace character.
- \S — non-whitespace character.
- \w — alphanumeric character or an underscore.
- \W — any character, except for an alphabetic, numeric character or the underscore character.
- . — (full stop) any character, except for the new string character.

Metacharacters to search for text delimiter characters

- \t — tabulation character.
- \n — new line character.
- \r — carriage return character.
- \f — switching to a new page.
- \u0085 — next line unicode character.
- \u2028 — line separator unicode character.
- \u2029 — paragraph separator unicode character.

Metacharacters to group characters

- [abc] — any of the listed (a,b, or c).
- [^abc] — any, except for the listed (neither a, nor b, nor c).
- [a-zA-Z] — merging ranges (Roman characters from a to z without considering case).
- [a-d[m-p]] — combining characters (from a to d and from m to p).
- [a-z&&[def]] — overlapping characters (characters d,e,f).
- [a-z&&[^bc]] — subtracting characters (characters a, d-z).

Quantifiers

These are metacharacters that are used to indicate the number of characters. They always come after a character or a group of characters.

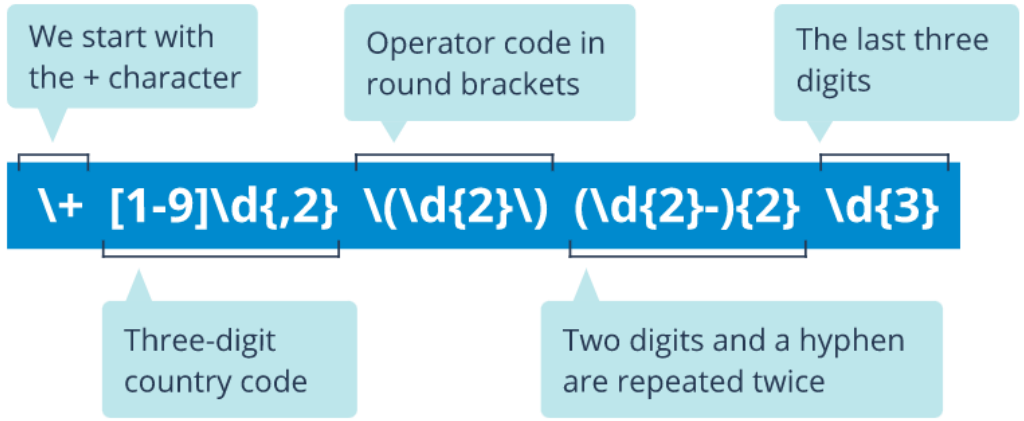
- ? — one or absent.
- * — zero or more times.
- + — one or more times.
- {n} — n times.
- {n,} — n times and more.
- {n,m} — at least n times but no more than m times.

Escaping

If you need to use the designation of a metacharacter or quantifier as a regular character, then escaping is applied:

- \<metacharacter> (example: *, \+, \., \?)
- [<metacharacter>] (example: [+], [?], [*], if then follows a quantifier)

As an example, look at the form for specifying the cell phone number:

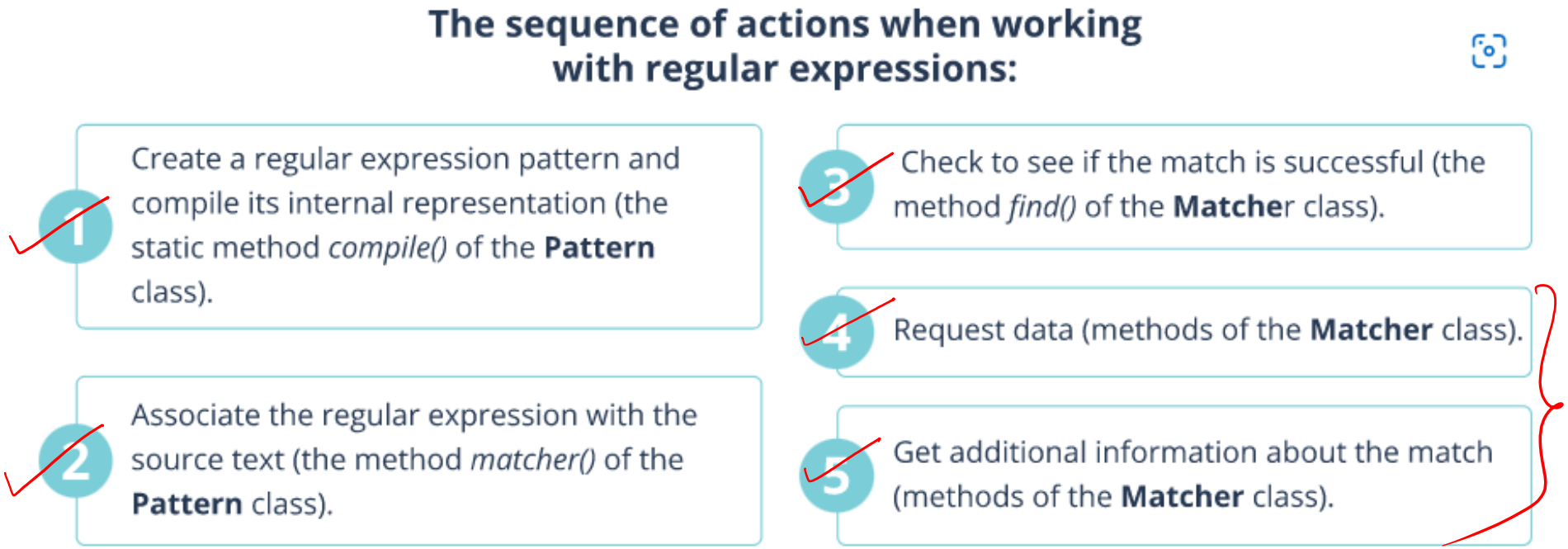


Example of strings matching the template:

```
+380(99)22-44-888
+380(67)98-54-321
```

Regular Expression processing(in java)

- **java.util.regex** is a package of the standard Java library containing major classes to work with regular expressions.
- For processing Regular expression we need two classes
  - ✓ Pattern – It is used to define a regular expression
  - ✓Matcher – match the pattern against sequence.



Pattern class

- Create a Pattern class object using compile method.
  static Pattern compile(String pattern)
  - It is used to transform the string in pattern into a pattern that can be used for pattern matching by Matcher .

- Create a Matcher class object using matcher() method provided by pattern.
  Matcher matcher(CharSequence str)
  - str is the character sequence that the pattern will be matched against

```
Pattern p   = Pattern.compile("a*b");

Matcher m = p.matcher("aaaaab");

boolean b  = m.matches();
```

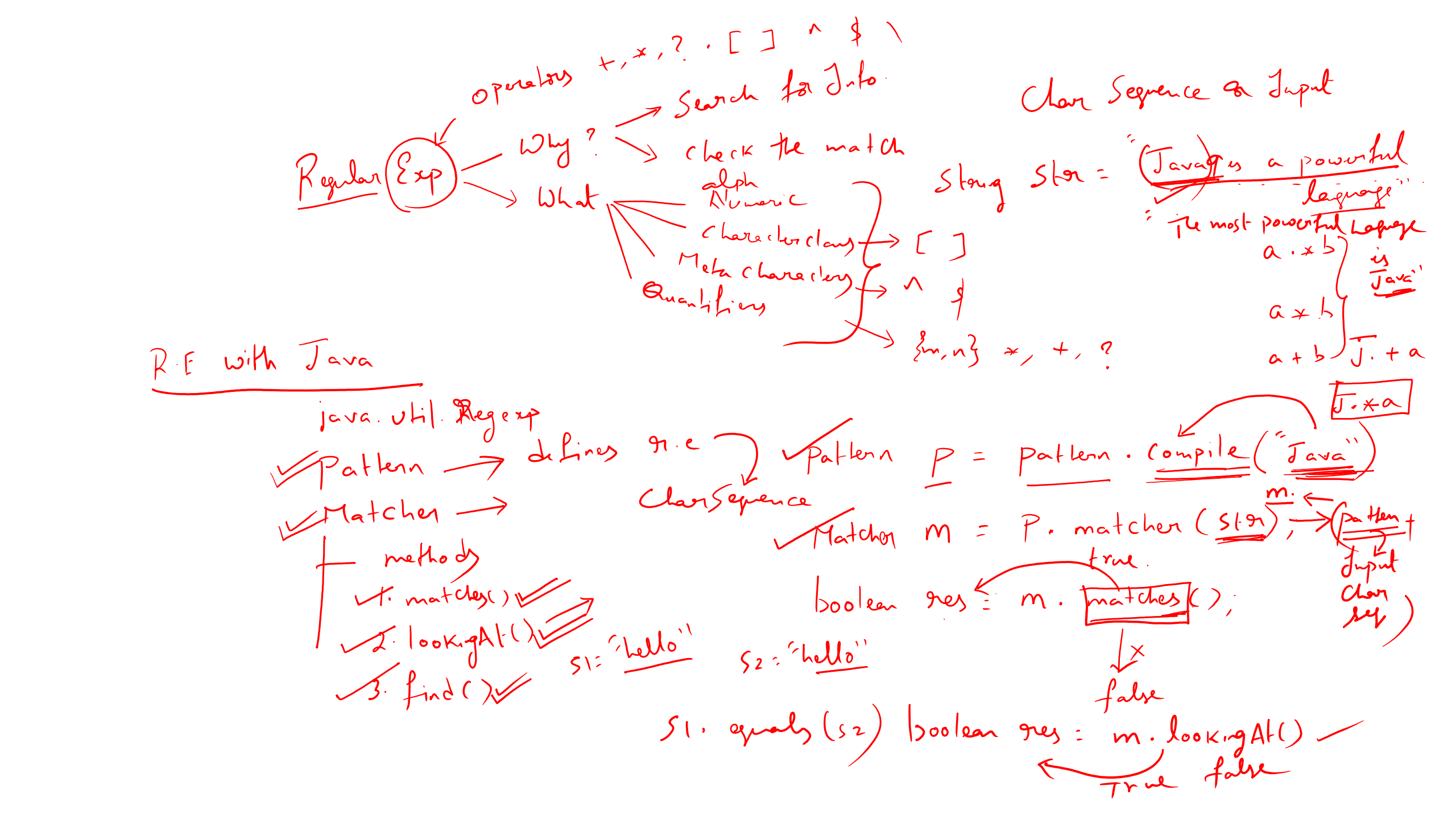
Pattern (regular expression)

Input string to match the pattern

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class RegExpDemo1 {
    public static void main(String[] args) {
        Pattern p;
        Matcher m;
        boolean found;
        p=Pattern.compile("Java");
        m=p.matcher("Java");
        found=m.matches();
        System.out.println("checking Java against Java");
        if(found)
            System.out.println("matches");
        else
            System.out.println("No match");
        m=p.matcher("Java 9");
        found=m.matches();
        System.out.println("checking Java 9 against Java");
        if(found)
            System.out.println("matches");
        else
            System.out.println("No match");
    }
}
```

Compiling the regular expression

To match the regular expression against the sequence of characters


operators +, *, ?, ., [ ], ^, $, \
Regular Exp
Why?
Search for Info.
check the match
What
alphaNumeric
Character class
Meta characters
Quantifiers
[ ]
^ $
{m,n} *, +, ?
Char Sequence as Input
String str = "Java is a powerful language"
The most powerful Language is Java
a.*b
a*b
a+b
J.+a
J.*a
R.E with Java
java.util.Regexp
Pattern → defines r.e
CharSequence
Matcher →
methods
1. matches()
2. lookingAt()
3. find()
Pattern P = Pattern.compile("Java")
Matcher m = P.matcher(str);
m:
pattern
Input Char Seq
boolean res = m.matches();
true
false
s1: "hello"
s2: "hello"
s1.equals(s2)
boolean res = m.lookingAt()
True
false

Using find() to match subsequences

The method *find()* is designed to search for the next subsequence of characters in the input sequence that matches the pattern.

There are two ways of how this method works:

- The search starts at the beginning of the given text.
- The search starts from the first character after the preceding match. This is possible only if the result of the previous invocation of this method is ***true*** and the matcher has not been reset.

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegularExpDemo {
    public static void main(String[] args) {
        Pattern p= Pattern.compile("test");
        Matcher m=p.matcher("test 1 2 3 test");
        while(m.find()) {
            System.out.println("test found at "+m.start());
        }
    }
}
```

test found at 0
test found at 11

Matcher class

An engine that performs match operations on a character sequence by interpreting a Pattern.

A matcher is created from a pattern by invoking the pattern's matcher method. Once created, a matcher can be used to perform three different kinds of match operations:

- The matches() method attempts to match the entire input sequence against the pattern.
- The lookingAt() method attempts to match the input sequence, starting at the beginning, against the pattern.
- The find() method scans the input sequence looking for the next subsequence that matches the pattern.

boolean	**matches**()Attempts to match the entire region against the pattern.
boolean	**lookingAt**()Attempts to match the input sequence, starting at the beginning of the region, against the pattern.
boolean	**find**()Attempts to find the next subsequence of the input sequence that matches the pattern.
boolean	**find**(int start)Resets this matcher and then attempts to find the next subsequence of the input sequence that matches the pattern, starting at the specified index.

Matcher class- methods (contd..)

String	**replaceAll**(**String** replacement) Replaces every subsequence of the input sequence that matches the pattern with the given replacement string.
String	**replaceFirst**(**String** replacement) Replaces the first subsequence of the input sequence that matches the pattern with the given replacement string.
String	**group**() Returns the input subsequence matched by the previous match.
int	**start**()Returns the start index of the previous match.
int	**end**()Returns the offset after the last character matched.

Pattern class- methods

static **Pattern**	**compile**(**String** regex) Compiles the given regular expression into a pattern.
static **Pattern**	**compile**(**String** regex, int flags) Compiles the given regular expression into a pattern with the given flags.
Matcher	**matcher**(**CharSequence** input) Creates a matcher that will match the given input against this pattern.
static boolean	**matches**(**String** regex, **CharSequence** input) Compiles the given regular expression and attempts to match the given input against it.
String	**pattern**() Returns the regular expression from which this pattern was compiled.
int	**flags**() Returns this pattern's match flags.

Using wildcards and quantifiers

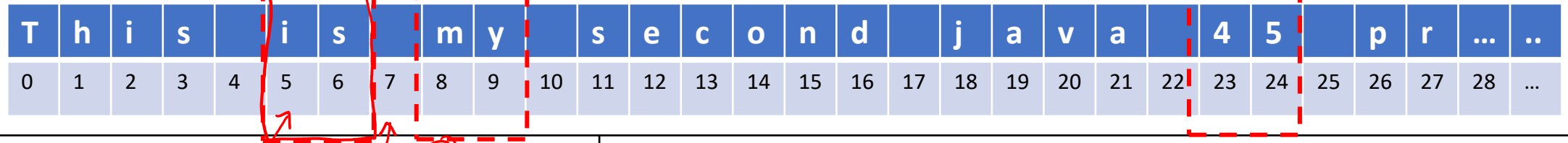
```
public class RegExpDemo2 {
    public static void main(String[] args) {
        String text = "This is my second java 45 project.\n" +
        "It is wonderful to learn polysemantics and arrays.\n" +
        "The weather is cold like it should be in winter, but we are all looking
        forward to spring.";
        Pattern p1 = Pattern.compile("\\b[\\w]{2}\\b");
        Matcher m1 = p1.matcher(text);
        while (m1.find()) {
            int start = m1.start();
            int end = m1.end();
            System.out.println("Found matches " + text.substring(start,end) + " from
            "+ start + " to " + (end-1) + " positions");
        }
    }
}
```

Note: In string literals describing a regular expression pattern, you can often see "\" (for example, for metacharacters) . In Java, it has to be doubled for the compiler to interpret it correctly:

The regular expression \\b[\\w]{2}\\b matches two length words surrounded by non-characters

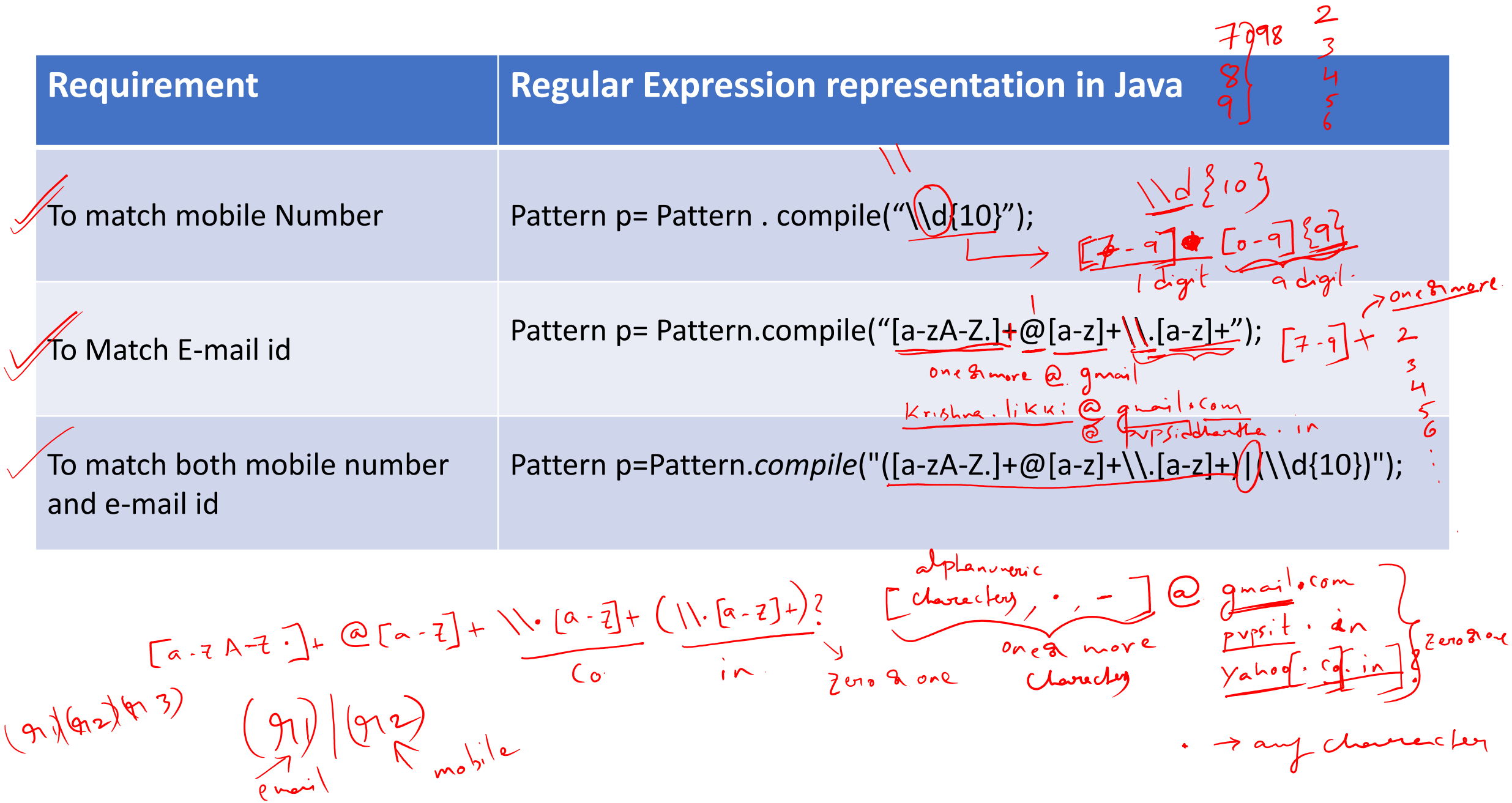
\b : to match word boundary i.e., non-character(all characters except letter , number and _)
[\\w] : to match word contains alphanumeric and underscore
[\\w]{2} : matches words of length 2 only.

```
Pattern p1 = Pattern.compile("\\b[\\w]{2}\\b");
Matcher m1 = p1.matcher("This is my second java 45 project");
```



m1.start()	Returns the start index of previous match i.e., start index of is,my and 45 which is 5,8 and 23
m1.end()	Returns the offset after the last character matched i.e., 7, 10, and 25
m1.group()	Returns the input subsequence matched by the previous match.

Requirement	Regular Expression representation in Java
To match mobile Number	Pattern p= Pattern . compile("\\d{10}");
To Match E-mail id	Pattern p= Pattern.compile("[a-zA-Z.]+@[a-z]+\\.[a-z]+");
To match both mobile number and e-mail id	Pattern p=Pattern.*compile*("([a-zA-Z.]+@[a-z]+\\.[a-z]+)|(\\d{10})");



replaceFirst()

```
public class RegExpDemoReplace {
    public static void main(String[] args) {
        String text = "This is my second java 45 project.\n" +
        "It is wonderful to learn polysemantics and arrays.\n" +
        "The weather is cold like it should be in winter, but we are all looking
        forward to spring.";
        System.out.println("Before replace:\n"+text);
        Pattern p1 = Pattern.compile("\\b[\\w]{2}\\b");
        Matcher m1 = p1.matcher(text);
        text=m1.replaceFirst("lab2");
        System.out.println("After replacement:\n"+text);
    }
}
```

Before replacement:

```
This is my second java 45 project.
It is wonderful to learn polysemantics and arrays.
The weather is cold like it should be in winter, but we are all looking forward to
spring.
```

After replacement:

```
This lab2 my second java 45 project.
It is wonderful to learn polysemantics and arrays.
The weather is cold like it should be in winter, but we are all looking forward to
spring.
```

replaceAll()

```
public class RegExpDemoReplace {
    public static void main(String[] args) {
        String text = "This is my second java 45 project.\n" +
        "It is wonderful to learn polysemantics and arrays.\n" +
        "The weather is cold like it should be in winter, but we are all looking
        forward to spring.";
        System.out.println("Before replace:\n"+text);
        Pattern p1 = Pattern.compile("\\b[\\w]{2}\\b");
        Matcher m1 = p1.matcher(text);
        text=m1.replaceAll("lab2");
        System.out.println("After replacement:\n"+text);
    }
}
```

Before replacement:

This is my second java 45 project.
It is wonderful to learn polysemantics and arrays.
The weather is cold like it should be in winter, but we are all looking forward to spring.

After replacement:

This lab2 lab2 second java lab2 project.
lab2 lab2 wonderful lab2 learn polysemantics and arrays.
The weather lab2 cold like lab2 should lab2 lab2 winter, but lab2 are all looking forward lab2 spring.

Two Pattern-matching options

1. Using compile and matcher method

```
Pattern p=Pattern.compile("J.+a");
Matcher m=p.matcher("Java");
System.out.println(m.matches());
```

2. Using matches method

```
System.out.println(Pattern.matches("J.+a","Java"));
System.out.println(Pattern.matches("J.+a","Java JavaScript"));
```

***Note**: If the same pattern is using repeatedly ,then it is less efficient than method 1 (compile and use pattern-matching methods of matcher class)*

Reflection

The *core reflection facility*, java.lang.reflect, offers programmatic access to arbitrary classes. Given a Class object, you can obtain Constructor, Method, and Field instances representing the constructors, methods, and fields of the class represented by the Class instance. These objects provide programmatic access to the class's member names, field types, method signatures, and so on

Reflection allows one class to use another, even if the latter class did not exist when the former was compiled.

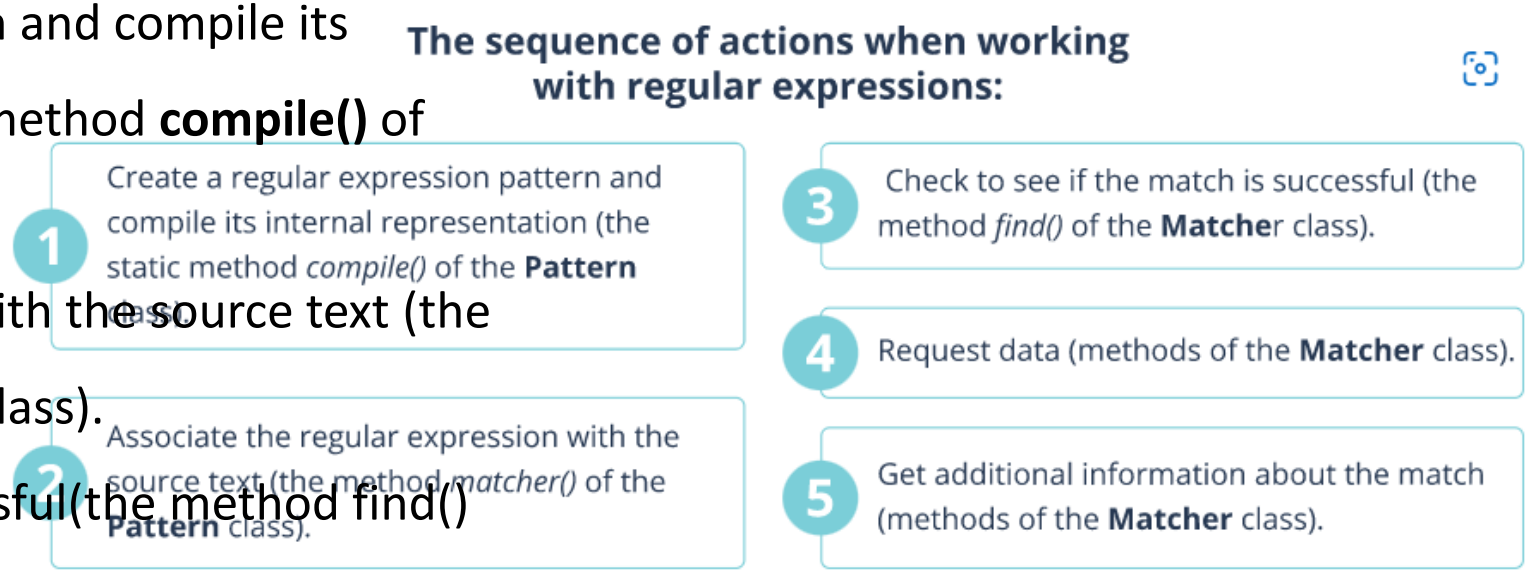
You lose all the benefits of compile-time type checking, including exception checking. If a program attempts to invoke a nonexistent or inaccessible method reflectively, it will fail at runtime unless you've taken special precautions.

- **The code required to perform reflective access is clumsy and verbose.** It is tedious to write and difficult to read.
- **Performance suffers.** Reflective method invocation is much slower than normal method invocation. Exactly how much slower is hard to say, as there are many factors at work.

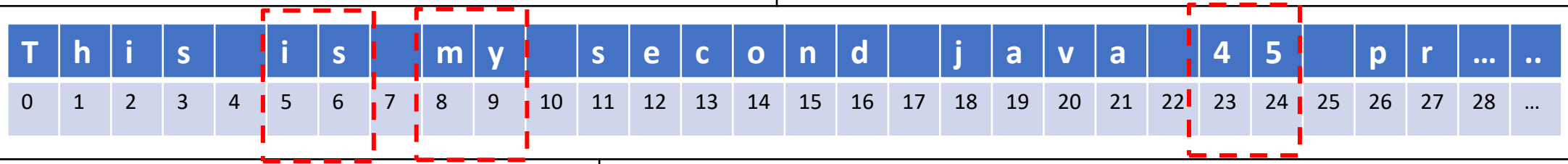
There are a few sophisticated applications that require reflection. Examples include code analysis tools and dependency injection frameworks

The sequence of actions for working with regular expressions

1. Create a regular expression pattern and compile its internal representation(the static method **compile()** of the **Pattern** class)
2. Associate the regular expression with the source text (the method **matcher**() of the **Pattern** class).
3. Check to see if the match is successful(the method find() of the Matcher class)
4. Request data(methods of the Matcher class).
5. Get additional information about the match(methods of the Matcher class).



Pattern p1 = Pattern.compile("\\b[\\w]{2}\\b"); Matcher m1 = p1.matcher(text); \b : to match word boundary i.e., non-character(all characters except letter , number and _) [\\w] : to match word contains alphanumeric and underscore [\\w]{2} : matches words of length 2 only. Matches two character words in the text	text="This is my second java 45 project". is ,my and 45 – which is surrounded by spaces(non-character)



m1.start()	Returns the start index of previous match i.e., start index of is,my and 45 which is 5,8 and 23
m1.end()	Returns the offset after the last character matched i.e., 7, 10, and 25
m1.group()	Returns the input subsequence matched by the previous match. i.e., is , my, and 45