Static data types

Using regular expressions

A regular expression (or *regex* in its abbreviated form) is a sequence of special characters that lets us detect some patterns in texts. For instance, an *id* card made of 8 digits and an uppercase letter, or an e-mail containing a `@`. Using regular expressions, we can easily detect these patterns in a given text and, also, force a text to meet a given pattern when it's being entered by the user.

1. Basic regex syntax

In order to work with regular expressions in Java, we need to use some classes from `java.util.regex` package. To be more precise, we are going to rely on:

- `Pattern` class, that lets us define a given pattern for a regular expression
- `Matcher` class, that lets us check if a givent text meets a given *Pattern*.

1.1. Pattern class

Regarding `Pattern` class, there are a couple of interesting methods inside it that we need to manage:

- `compile`: it creates a *Pattern* object for a given regular expression
- `matcher`: it returns a *Matcher* object to match a pattern with a text

1.2. Matcher class

Inside `Matcher` class, we can use these methods:

- `find`: checks if a given pattern is found in a text
- `matches`: checks if the whole text meets a pattern (not only a part of it). There is also a static method in *Pattern* class, `Pattern.matches` that produces the same result.

1.3. Example

Let's check in the following example if a text contains a digit between 0 and 9. This digit can be represented by the symbol `\d`, so we use it inside the pattern, this way:

```
String text = "Hi, my name is Nacho and I'm 44 years old";
Pattern p = Pattern.compile("\\d");
Matcher m = p.matcher(text);

if (m.find())
{
    System.out.println("The text contains digit(s)");
}
else
{
    System.out.println("The text does not contain any digit");
}
```

Note that we need to escape the `\` when we place it inside the string, as `\\d`

1.4. Some basic symbols

In this table you can find some basic symbols that we can use inside patterns.

Symbol	Meaning
`x`	'x' character
`\t`	Tabulation
`\n`	New line
`[abc]`	Character 'a', 'b' or 'c'
`[^abc]`	Anything but characters 'a', 'b' or 'c'
`[a-zA-Z]`	Range from 'a' to 'z' or from 'A' to 'Z'
`.`	Any character
`^`	Beginning of line (there's nothing before)
`$`	End of line (there's nothing after)
`\d`	Digit from 0 to 9
`\D`	Anything but a digit
`\s`	Spacing char (white space, tab, new line...)
`\S`	Anything but a spacing char
`\w`	Alphanumeric char (letters, digits and underscore)
`\W`	Anything but an alphanumeric char
`(one|two)`	Text 'one' or text 'two'

With these symbols, we can made expressions such as:

- Text finished with a dot (we need to *escape* the dot with `\`):

```
Pattern p = Pattern.compile("\\.$");
```

- Text made of 4 digits

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

- The seasons of the year:

```
Pattern p = Pattern.compile("(winter|spring|summer|autumn)");
```

> **Exercise 1:**
>
> Create a project called **CarIDCheck** that asks the user to enter a cad id, and checks if it's made of 4 digits followed by 3 uppercase letters. We are not going to check if these letters are vowels or not, we just check if they are uppercase.

2. Some complex expressions

In order to make some more complex expressions, we need to make use of some additional symbols in the pattern. To be more precise, we need to specify the cardinality of some parts of the expressions, in orde to shorten them. This is a list of cardinality symbols that you can use in your patterns:

Symbol	Meaning
`x?`	x symbol appears 0 or 1 times
`x+`	x symbol appears 1 or more times
`x*`	x symbol appears 0 or more times
`x{n}`	x symbol appears n times
`x{n,}`	x symbol appears at least n times
`x{n, m}`	x symbol appears between n and m times (both included)

This way, we can easily check:

- If a text is made of 4 digits with this:

```
Pattern p = Pattern.copile("^\\d{4}$");
```

- An *id* card made of 8 digits and an uppercase letter:

```
Pattern p = Pattern.copile("^\\d{8}[A-Z]$");
```

> **Exercise 2:**
>
> Repeat previous exercise using cardinality symbols

> **Exercise 3:**
>
> Create a program called **EmailChecker** that asks the user to enter an e-mail and checks if it's valid. We will consider that a valid e-mail will be made of alphanumeric characters (at least one), followed by a `@`, one or more alphanumeric characters, a dot and one or more alphanumeric characters. So `myEmail@one.com` is a valid e-mail, but `myOtherMail@aaa` is not.

3. Using groups

Groups lets us isolate some parts of a text that meet a given pattern, so we can treat them later in the code. We can use `group` method inside `Matcher` class to detect groups, and each group must be defined between parentheses `(...)` in the pattern. Groups are explored from left to right.

Let's see how groups work with the following example: we are going to get every sequence of 4 digits in a text:

```
String text = "Einstein was born in 1879 and Edison in 1847";
Pattern p = Pattern.compile("(\\d{4})");
Matcher m = p.Matcher(text);

if (!m.find())
{
    System.out.println("The text has no sequence of 4 digits");
}
else
{
    do
    {
        String data = m.group();
        System.out.println("Found " + data);
    }
    while(m.find());
}
```

As you can see, every time we call `group` method we *move* to next group identified in the text, until there are no more matches pending.

3.1. Multiple groups

We can define more than one group in a single expression. In this case, `group` method admits an additional parameter indicating which of the groups we are choosing (starting from 1).

The following example identifies names and surnames in a text:

```
String text = "Albert Einstein was born in 1879" +
    " and Thomas Edison in 1847";
Pattern p = Pattern.compile("([A-Z][a-z]+) ([A-Z][a-z]+)");
Matcher m = p.Matcher(text);

if (!m.find())
{
    System.out.println("The text has no names");
}
else
{
    do
    {
        String name = m.group(1);
        String surname = m.group(2);
        System.out.println("Found " + name + " " + surname);
    }
    while(m.find());
}
```

It would output *Found Albert Einstein* in the first iteration and *Found Thomas Edison* in the second one.

> **Exercise 4:**
>
> Create a program called **HourIdentifier** that looks for hours in a text. An hour is made of two digits, followed by `:`, and two digits. For instance, `08:45`. We are not going to check if the hour is valid or not, we just need to identify them. Then, store these hours in a list and show them in ascending order.