# Exercise 1: Regex Testing

**40** Points

**1/27/2025**

Attempt 1

Review Feedback
1/27/2025

Attempt 1 Score:

View Feedback

Anonymous Grading: **no**

**Unlimited Attempts Allowed**
1/17/2025 to 1/31/2025

**Details**

In this assignment, you will test and implement scenarios for regular expressions to practice testing, working with specifications, and regular expressions (which are an introduction to grammars for later on).

## Submission

You will submit a Zip file of the entire `src` folder (containing just the single `src` folder), which should contain both your implementation (`src/main/java`) and tests (`src/test/java`). On Windows, right-click on the src folder and "Compress to ZIP file" should have the expected behavior here.

- The test submission is **Thursday, January 23**: **Exercise 1: Regex Testing (TEST) (https://ufl.instructure.com/courses/523541/assignments/6419669)**
- The final submission is **Monday, January 27**.

## Project Setup

See the following links for provided code and setup instructions:

- **RegexTesting.zip (https://ufl.instructure.com/courses/523541/files/94092097?wrap=1)** ↓ **(https://ufl.instructure.com/courses/523541/files/94092097/download?download_frd=1)**
- **Project Setup (IntelliJ & Gradle) (https://ufl.instructure.com/courses/523541/pages/project-setup-intellij-and-gradle)**

## Regex Overview

A Regex (REGular EXpression) defines a grammar that can be used to pattern match strings. There are many flavors of regex (some that break the "regular" piece), but for this course we'll focus on the main features of regex which are standard across nearly all implementations. Regexes are an incredibly useful too in their own right, but are also a great stepping stone into the

concept of grammars/parsing that will kickoff our language project this semester. Some useful links for working with regex regex:

- **RegExr (https://regexr.com/)** : Realtime regex matching and reference sheet
- **regexper (https://regexper.com/#(%3F%3ARailroad%7CRegex))** : Railroad diagram visualization of regexes
- **Java Regex Cheat Sheet (https://www.jrebel.com/blog/java-regular-expressions-cheat-sheet)** : Java-specific reference sheet

# JUnit Overview

JUnit is a testing framework for Java. For this course, we'll be using JUnit 5 (Jupiter) which includes a couple new features that will be helpful to us - in particular, parameterized tests. Here are some helpful links on JUnit - while you can probably get by for now with just copying the examples, you should make sure you have an understanding of what's happening as we'll be using JUnit throughout the course.

- **JUnit5 User Guide (https://junit.org/junit5/docs/current/user-guide/)**
- **(https://junit.org/junit5/docs/current/user-guide/) JUnit5 Parameterized Tests Tutorial (https://www.baeldung.com/parameterized-tests-junit-5)**

# Part 1. Testing Regex

First, you will write test cases for a regex has been implemented for you. This part will have component of the grade based on test coverage, i.e. how well do your tests verify the intended behavior of the regex. You must include at least 5 matching and 5 non-matching tests, however you will likely need more for full credit on coverage.

| Regex | Description | Examples |
|---|---|---|
| `EMAIL` | Matches an email, or something close enough. This does not cover all valid emails (and may include some invalid ones!) - the point is to use this as the definition of correctness and build tests around what it does/ doesn't match.<br>• `[A-Za-z0-9._\-]+@[A-Za-z0-9-]*\.[a-z]{2,3}` | • `email@example.com`<br>• `thelegend27@gmail.com`<br>• `email@missingdot`: invalid; missing dot<br>• `#$%@example.com`: invalid; illegal chars |

## Example

**Note:** This example was reviewed and discussed in M0L3: Regex & Testing.

| Regex | Description | Examples |
|---|---|---|

| | | |
|---|---|---|
| `URL` | Matches a URL, or something close enough. Note that the domain piece here is very similar to th<br>• `https?:\/\/[A-Za-z]*\.[a-z]{2,3}`<br>◦ `https?:\/\/`: matches the protocol, e.g `http://`.<br>◦ `[A-Za-z]*`: matches the website name, e.g. `example`.<br>◦ `\.`: matches the literal dot before the top-level domain.<br>◦ `[a-z]{2, 3}`: matches the top-level domain, e.g. `com`. | • `http://example.com`<br>• `example.com`: invalid; missing protocol |

Our general strategy is the following:

1. Start with a *baseline test*, which is the simplest working example.
2. Next, methodically create test *variations* that test specific functionality and edge cases. Each variation should make a *minimal* number of changes from the baseline to better isolate the specific feature being tested.
   - The better our variations isolate and test different functionality, the easier it will be to identify the causes of test failures.

A good baseline test is `http://example.com` - it's simple, and `example.com` is **reserved for testing (https://www.iana.org/help/example-domains)** as well (which makes it's purpose clear and also reduces any unintentional overlaps with real services. Another reasonable option might be `https://example.com` (http secure), if you think the baseline should be the "secure" version under the idea of secure-by-default. In the context of regex, using the "ignore" case for an optional `s?` generally makes the most sense.

Next, let's start with the protocol `https?:\/\/`. The easy first candidate is `https://example.com`, testing the optional `s?` being present in the protocol. It's important that the rest of the example remains identical to the baseline to ensure our test is actually testing what we want (`s?`) and not other functionality. For example; `https://ufl.edu` is effectively the same, but it's less clear what's being testing as both the website and domain change at the same time.

Continuing, the website name is represented by `[A-Za-z0-9-]*`. Notice that this uses `*`, which means *zero or more*. It's always a good idea to test zero, one, and multiple cases, so let's add `http://.com` and `http://e.com` to our tests (multiple is covered by our baseline already, though in some cases it may not hurt to add it for clarity in complex scenarios). Next, we should consider the characters allowed - we already have tests covering lowercase letters, so one with uppercase letter like `http://EXAMPLE.com` is a good choice. Finally, we should also consider the characters *not* allowed (digits, symbols, whitespace, emojis...) and add non-matching tests for those as we want to ensure

it's not just accepting anything. This isn't completely foolproof, but does a decent enough job of covering common test scenarios.

The same process continues for the rest of the regex, which is part of the email example in your assignment.

# Part 2. Writing (and Testing) Regex

Second, you will write your own regexes for the scenarios given and corresponding test cases. This part is primarily graded based on how accurate your interpretation of the specification is - in a certain sense, the regex itself is the easy part! You must still provide 5 matching and 5 non-matching test cases (and should probably have more for your own testing), however we will not grade coverage like in Part 1.

**Important:** As tempting as it may be to start by writing regex, you should start with the tests (continuing the ideas from above) for practice with testing and to help you think through the edge cases involved. There will be tasks that are harder than regex!

| Regex | Description | Examples |
|---|---|---|
| `DISCOUNT_CSV` | Matches a bare-minimum version of CSV that can be used to frustrate developers trying to parse actual CSV.<br>• A CSV consists of one or more values.<br>• Values consists of one or more non-comma, non-whitespace (as defined by the `\s` character class) characters.<br>• Values are separated by a single comma, which may itself be optionally surrounded by zero or more whitespace characters. | • `single`<br>• `one,two,three`<br>• `first , second`<br>• `first,,third`: invalid; missing value |
| `DICE_NOTATION` | Matches a simplified version of dice notation. All dice notations begin with `<count>d<faces>`, e.g. `1d6`, encoding the count of dice to roll and how many faces those dice should have. Additionally, an optional "bonus" value (or penalty) can be included at the end, e.g. `1d6+4`.<br>• `count`, `faces`, and `bonus` are named capturing groups.<br>◦ Hint: See test cases for a clearer idea of how this should work.<br>• `count` and `faces` are both positive integers, | • `1d6`<br>◦ `count = "1"`<br>◦ `faces = "6"`<br>• `1d6+4`<br>◦ `count = "1"`<br>◦ `faces = "6"`<br>◦ `bonus = "+4"`<br>• `1d`: invalid; missing faces<br>• `-1d6`: invalid; negative count |

| | | |
|---|---|---|
| | without leading zeros.<br>• `bonus`, if present, is any signed integer (zero included), without leading zeros. Hence, the sign should be included in the capturing group. | |
| `NUMBER` | Matches an integer or decimal number, including exponents.<br>• A number contains a required integer part, an optional decimal part (starting with `.`), and an optional exponent part (starting with `e`).<br>• All numeric parts consist of one or more decimal digits (0-9) and, for simplicity, allow leading/trailing zeros.<br>• The integer and exponents parts may have an optional `+`/`-` sign.<br>Note: This is the same format for numbers that will be used in the lexer. | • `1`<br>• `123.456`<br>• `1e5`<br>• `.5`: invalid; missing integer part<br>• `10e`: invalid; missing exponent digits |
| `STRING` | Matches a string literal, as defined below.<br>• A string starts and ends with a double quote (`"`).<br>• A string contains zero or more characters, which may not be a double quote (`"`) or a line ending (the actual `\n`/`\r`, written in Java strings as `"\n"`/`"\r"`), since these will end the string prematurely.<br>• Escape characters start with `\` and must be followed by one of `bfnrt'"\`.<br>◦ **Important:** Remember that Java and Regex have escape characters! If you want a string with the value `"1\\2"`, you need to escape these with `"\"1\\\\2\""`. If you want a regex matching `"1\\2"`, the backslash must be escaped *again* with `"\"1\\\\\\\\2\""` (which | • `""`<br>• `"string"`<br>• `"1\\2"`<br>• `"unterminated`: invalid; missing end quote<br>• `"invalid\escape"`: invalid; `\e` is not a valid escape character |

| | | |
|---|---|---|
| | looks ridiculous, but is correct) because it's also a special character in regex! This will occur frequently in this course, so make sure to understand this idea!<br>Note: This is the same format for strings that will be used in the lexer. | |

# Example

**Note:** This example was reviewed and discussed in M0L3: Regex & Testing.

| | | |
|---|---|---|
| `HEX_COLOR` | Matches a hexadecimal color, as defined below.<br>• A color always starts with `#`.<br>• A color contains either `3` or `6` case-insensitive hexadecimal digits.<br>  ◦ `3` digits is a common shorthand; `#123` means `#112233`. | • `#FFA500`<br>• `#abc`<br>• `FFA500`: invalid; missing `#`<br>• `#ghi`: invalid; illegal digits<br>• `#123456789`: invalid; too many digits |

We can start with a regex for any case-insensitive hexadecimal digit, `[0-9A-Fa-f]`. There is a clever trick worth showing to handle either 3 or 6 digits by matching 3 digits either once (3 total) or twice (6 total), which looks like `([0-9A-Fa-f]{3}){1,2}` (this probably isn't good practice, but... it's cool). Following that, don't forget the leading hashtag for the final solution `#([0-9A-Fa-f]{3}){1,2}`.

Next, test cases. The examples ones provide a good mix of behavior, but it may be better to split them up for better separation (e.g. `#123456`, `#abcdef`, and `#ABCDEF` for each kind of digit). Other useful tests might be `#aBcDeF` (mixed case) and `#123` (short numbers) for matching cases, and then `#$&*` (other characters) and `#` (hashtag only) for non-matching.

# Changelog

| | |
|---|---|
| Jan. 23 | • Fix minor syntax error in URL example regex (`{2, 3}` instead of `{2,3}`)<br>• Clarify line ending characters in string specification refer to the "real" characters and not escapes using `NL`/`CR` - these are visual representations for the otherwise "invisible" characters. |
| Jan. 25 | • Clarified zero is included for the dice notation bonus (as in test submission) and removed confusing wording. |