Basics of Regular Expressions

For String manipulation

By the end of this video you will be able to…

- Write regular expressions to match String patterns
- Use regular expressions to split strings

String method split(String pattern)

```
String text = "Can you hear me?  Hello, hello?"
String[] words = text.split(" ");
```

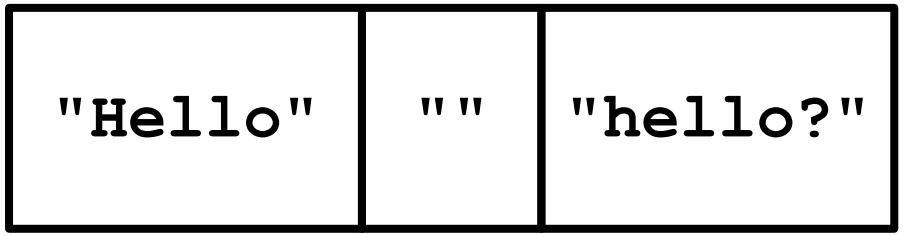
split(String regex)

Splits this string around matches of the given **regular expression**.

Regular expression: Characters are basic units

```
String text = "Hello_ hello?"
String[] words = text.split(" ");
```

This single space is a regular expression.
It matches single spaces



Regular expressions: 3 ways to combine

Repetition

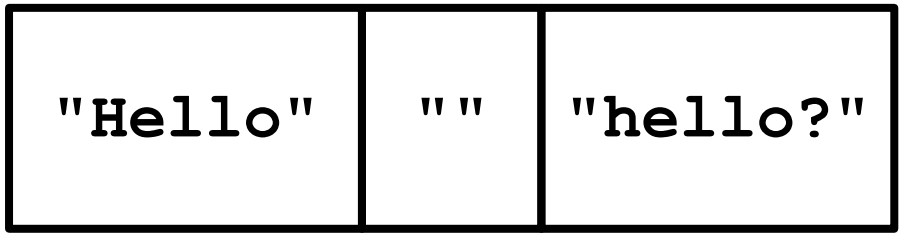
Concatenation

Alternation

Regular expression: Characters are basic units

```
String text = "Hello  hello?"
String[] words = text.split(" ");
```

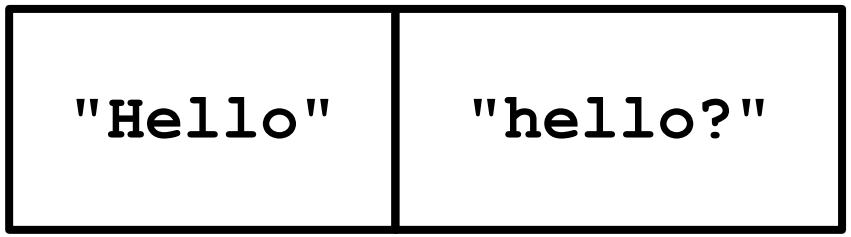
This single space is a regular expression.
It matches single spaces



Repetition: + means 1 or more

```
String text = "Hello  hello?"
String[] words = text.split(" +");
```

Matches 1 or more spaces in a row



Relating regex's to the project

```
public abstract class Document
{
    // The text of the whole document
    private String text;
```

Relating regex's to the project

```
public abstract class Document
{
    // The                    document
    private

    protected List<String> getTokens(String pattern)
    {
        …
```

returns a List of "tokens"

regular expression defining the "tokens"

```
public abstract class Document
{    …
     protected List<String> getTokens(String pattern)
     {
          …
```

Repetition

Assume you have a Document object, d, whose text is
"Hello hello?"

d.getTokens(" +"); → **[" "]**

Matches 1 or more spaces

```
public abstract class Document
{    …
     protected List<String> getTokens(String pattern)
     {
          …
```

Concatenation

Assume you have a Document object, d, whose text is

```
"Splitting a string, it's as easy as 1 2 33!  Right?"
```

```
d.getTokens("it");
```

Two regular expressions side by side
Matches when both appear one after the other

```
public abstract class Document
{    …
     protected List<String> getTokens(String pattern)
     {
          …
```

Concatenation

Assume you have a Document object, d, whose text is
"Splitting a string, it's as easy as 1 2 33! Right?"

d.getTokens("it"); → **["it", "it"]**

Two regular expressions side by side
Matches when both appear one after the other

```
public abstract class Document
{    …
     protected List<String> getTokens(String pattern)
     {
          …
```

Concatenation and Repetition

Assume you have a Document object, d, whose text is
"Splitting a string, it's as easy as 1 2 33! Right?"

```
d.getTokens("it+");
```

+ means "one or more"

```
public abstract class Document
{    …
     protected List<String> getTokens(String pattern)
     {
          …
```

Concatenation and Repetition

Assume you have a Document object, d, whose text is
"Splitting a string, it's as easy as 1 2 33! Right?"

d.getTokens("it+"); → **["itt", "it"]**

+ means "one or more"

```
public abstract class Document
{    …
     protected List<String> getTokens(String pattern)
     {
          …
```

Concatenation and Repetition

Assume you have a Document object, d, whose text is
"Splitting a string, it's as easy as 1 2 33! Right?"

d.getTokens("i(t+)")→ ["itt", "it"]

Use parens to group r.e.'s if you are not sure of grouping

```
public abstract class Document
{    …
     protected List<String> getTokens(String pattern)
     {
          …
```

Concatenation and Repetition

Assume you have a Document object, d, whose text is
"Splitting a string, it's as easy as 1 2 33! Right?"

```
d.getTokens("it*");
```

*** means "zero or more"**

```
public abstract class Document
{    …
     protected List<String> getTokens(String pattern)
     {
          …
```

Concatenation and Repetition

Assume you have a Document object, d, whose text is
"Splitting a string, it's as easy as 1 2 33! Right?"

d.getTokens("it*"); → **["itt", "i", "i", "it", "i"]**

*** means "zero or more"**

Alternation

```
public abstract class Document
{    …
     protected List<String> getTokens(String pattern)
     {
          …
```

Assume you have a Document object, d, whose text is
`"Splitting a string, it's as easy as 1 2 33!  Right?"`

```
d.getTokens("it|st");
```

| means OR

Alternation

```
public abstract class Document
{    …
     protected List<String> getTokens(String pattern)
     {
          …
```

Assume you have a Document object, d, whose text is
"Splitting a string, it's as easy as 1 2 33! Right?"

d.getTokens("it|st"); → **["it", "st", "it"]**

| means OR

```
public abstract class Document
{    …
     protected List<String> getTokens(String pattern)
     {
          …
```

Character classes

Assume you have a Document object, d, whose text is
"Splitting a string, it's as easy as 1 2 33! Right?"

d.getTokens("[123]"); → **["1", "2", "3", "3"]**

[ ] mean match "anything in the set"

```
public abstract class Document
{    …
     protected List<String> getTokens(String pattern)
     {
          …
```

Character classes

Assume you have a Document object, d, whose text is
"Splitting a string, it's as easy as 1 2 33! Right?"

d.getTokens("[1-3]"); → **["1", "2", "3", "3"]**

- **indicates a range (any character between 1 and 3)**

```
public abstract class Document
{    …
     protected List<String> getTokens(String pattern)
     {
          …
```

Character classes

Assume you have a Document object, d, whose text is
"Splitting a string, it's as easy as 1 2 33! Right?"

```
d.getTokens("[a-f]");
```

- **indicates a range (any character between a and f)**

```
public abstract class Document
{    …
     protected List<String> getTokens(String pattern)
     {
          …
```

Character classes

Assume you have a Document object, d, whose text is
`"Splitting a string, it's as easy as 1 2 33!  Right?"`

`d.getTokens("[a-f]");` → `["a", "a", "e", "a", "a"]`

- **indicates a range (any character between a and f)**

```
public abstract class Document
{    …
     protected List<String> getTokens(String pattern)
     {
          …
```

Excluding a character

Assume you have a Document object, d, whose text is
"Splitting a string, it's as easy as 1 2 33! Right?"

```
d.getTokens("[^a-z123 ]");
```

^ indicates NOT any characters in this set

Negation

```
public abstract class Document
{    …
     protected List<String> getTokens(String pattern)
     {
          …
```

Assume you have a Document object, d, whose text is
"Splitting a string, it's as easy as 1 2 33! Right?"

```
d.getTokens("[^a-z123 ]"); → ["S", ",", "'", "!", "R", "?"]
```

^ indicates NOT any characters in this set