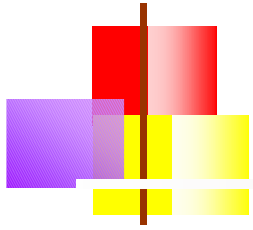


CNAM-NFP121

an 8

Traitement des Chaînes en JAVA



La classe String revisitée



java.lang.String

Les problèmes de traitement de chaînes de caractères sont multiples et très importants dans la programmation "moderne" :
Lecture de fichiers texte, correcteurs d'orthographe, formatages divers, scanning, etc, etc ...

La classe **String** fournit déjà un certain nombre de méthodes...

java.lang Interface CharSequence

char **charAt**(int index)

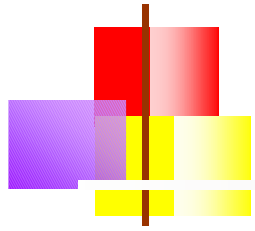
int **length**()

CharSequence **subSequence**(int start, int end)

String **toString**()

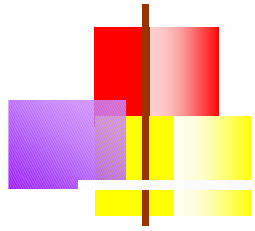
All Known Implementing Classes:

CharBuffer, String, StringBuffer, StringBuilder



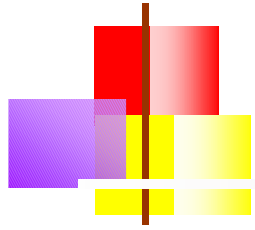
Classe String : quelques méthodes

- Comparaisons :
 - int **compareTo**(String anotherString)
 - boolean **contains**(CharSequence s)
 - boolean **contentEquals**(CharSequence cs)
 - boolean **equalsIgnoreCase**(String anotherString)
 - int **indexOf**(String str)
 - boolean **startsWith**(String prefix)
 - boolean **regionMatches**(int toffset, String other, int ooffset, int len)
 - Etc...



Classe String : equals, intern(), ...

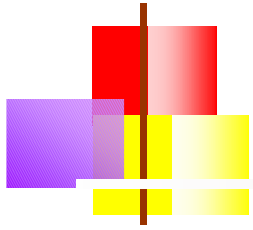
- boolean **equals**(Object anObject)
- String **intern**() : pour 2 String s et t,
s.intern() == t.intern() est vraie si et seulement si
s.equals(t) est vraie.
- Extractions de sous chaînes, modifications
 - String **substring**(int beginIndex)
 - StringReplace(CharSequence target,
CharSequence replacement)



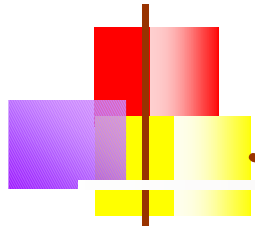
String : construction, découpage, ...

- String **concat**(String str)
- String **toLowerCase**() , String **toUpperCase**()
- String **trim**()

- Apparition d'un paramètre nommé regex :
 - String **replaceAll**(String regex, String replmnt)
 - String **replaceFirst**(String regex, String replmnt)
 - String[] **split**(String regex)



java.util.StringTokenizer

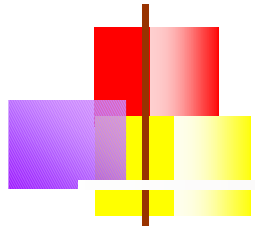


java.util.StringTokenizer

```
StringTokenizer st = new StringTokenizer("this is a test");  
while (st.hasMoreTokens()) {  
    System.out.println(st.nextToken());  
}
```

Sortie obtenue:

- This
- is
- a
- test



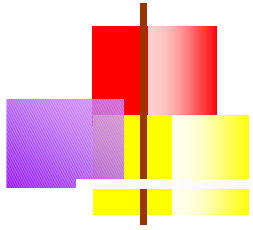
StringTokenizer : pour mémoire...

La classe `StringTokenizer` permet de découper des chaînes en tokens/lexèmes.

La méthode de "tokenization" est beaucoup plus simple que celle utilisée dans la classe `StreamTokenizer`...

`StringTokenizer(String str, String delim)`

Cette classe est maintenue pour des raisons de compatibilité mais maintenant on utilisera plutôt les méthodes `split()` de la classe `String` ou encore le package `java.util.regex`.



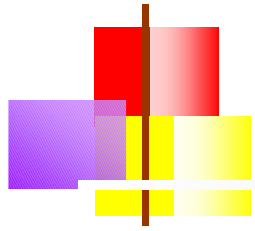
java.io.StreamTokenizer



java.io.StreamTokenizer ...

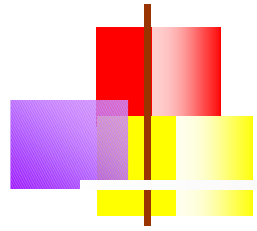
La classe `StreamTokenizer` permet de découper un flot d'entrée en tokens/lexèmes et de lire ces tokens un par un. Elle peut reconnaître des identificateurs, des nombres, des chaînes et différents types de commentaires.

`StreamTokenizer(Reader r)`



StreamTokenizer : nextToken, ...

- **public int nextToken()** throws IOException
 - Parse le token suivant dans l'entrée et retourne le type de ce token : TT_EOF , TT_EOL , TT_NUMBER , TT_WORD. Ce type est aussi la valeur de l'attribut ttype.
 - D'autres informations sur le token dans les attributs nval field ou sval.
- **public int ttype**
 - Après chaque appel à nextToken() contient le type du token lu (cf. ci dessus). Pour les token d'un seul caractère la valeur est ce caractère.
 - La valeur initiale est -4.

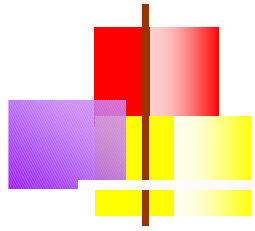


StreamTokenizer , utilisation

Utilisation typique de cette classe :

1/ configuration du tokenizer : **ordinaryChar**(int ch) ,
whitespacesChar(...) , **wordChars**(...) , etc...

2/ boucle avec appels de **nextToken** pour analyse des
tokens successifs ("rateau à 4 ou 5 branches") jusqu'à
TT_EOF.

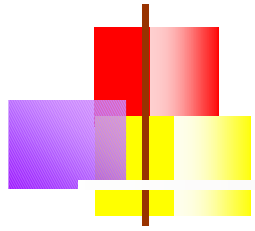


StreamTokenizer ...exemple ⁽¹⁾

Au moment de l'intropection/reflexion on obtenait les liste des méthodes déclarées et héritées d'une classe par :

```
try {  
    Class c = Class.forName(args[0]);  
    Method[] m = c.getMethods();  
    for (int i = 0; i < m.length; i++)  
        System.out.println(m[i]);  
} catch(ClassNotFoundException e) {...
```

Mais les noms de classes sont alors qualifiés ...

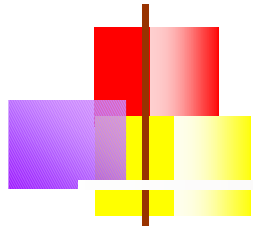


StreamTokenizer ...exemple ⁽²⁾

Mais les noms de classes sont alors qualifiés :

- public `java.lang.String[]`
`java.lang.String.split(java.lang.String)`
- public boolean
`java.lang.String.startsWith(java.lang.String,int)`
- On préférerait :
- public `String[] split(String)`
- public boolean `startsWith(String,int)`

- Utilisation d'un `StreamTokenizer`!



StreamTokenizer ...exemple ⁽³⁾

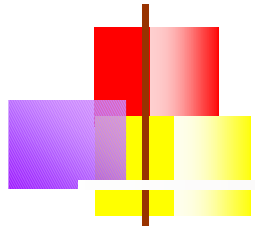
```
public static String strip(String qualified) {
    st = new StreamTokenizer(new StringReader(qualified));
    st.ordinaryChar(' '); // garde les espaces qui deviennent des tokens
    String s = "", si;
    while((si = getNext()) != null) {
        int lastDot = si.lastIndexOf('.');
        if(lastDot != -1)
            si = si.substring(lastDot + 1);
        s += si;
    }
    return s;
}
```

- Remarque : public class **StringReader** extends Reader
A character stream whose source is a string !!!



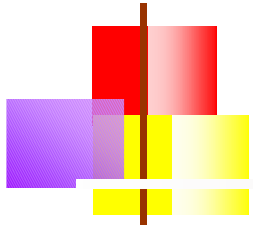
StreamTokenizer ...exemple ⁽⁴⁾

```
■ public String getNext() {
    String s = null;
    try {
        int token = st.nextToken();
        if(token != StreamTokenizer.TT_EOF) {
            switch(st.ttype) {
                case StreamTokenizer.TT_EOL: s = null; break;
                case StreamTokenizer.TT_NUMBER:s = Double.toString(st.nval);
                break;
                case StreamTokenizer.TT_WORD:s = new String(st.sval);break;
                default: s = String.valueOf((char)st.ttype);
            } } } catch(IOException e) {}
    return s;
}
```



StreamTokenizer ...exercice

- Compter les différents tokens d'un fichier selon les catégories de `StreamTokenizer` (`TT_EOL`, `TT_NUMBER`, `TT_WORD` et *token d'un caractère*)
- Les commentaires "à la Java" sont ignorés
- Le caractère ! indique aussi un commentaire "fin de ligne"

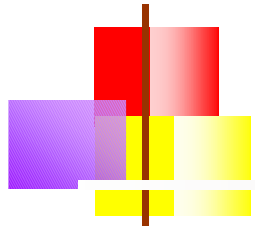


java.util.Scanner



java.util.Scanner ...

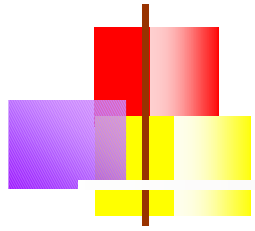
- "depuis java 1.5, Java dit" : Un simple scanner de texte qui peut reconnaître les types primitifs et les strings en utilisant les expressions régulières.
- Le Scanner découpe l'entrée en tokens en utilisant des delimitateurs sous forme de pattern, (par défaut les espaces). Les tokens résultants peuvent être convertis dans des valeurs de différents types par les différentes méthodes 'next'.
- *Scanner est 'orienté caractères' :*
- Scanner(Readable source)
- Scanner(String source)
- void **close()**



Scanner : exemple 1

- lire un nombre sur l'entrée courante, sinon *java.util.InputMismatchException* :

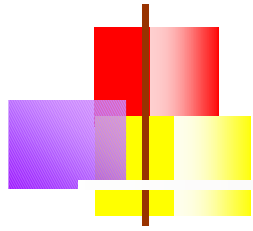
```
Scanner sc = new Scanner(System.in);  
int i = sc.nextInt();
```



Scanner : itérateur

- Scanner est d'abord un itérateur :
- boolean **hasNext()**
- boolean **hasNext**(Pattern pattern)
- boolean **hasNext**(String pattern)
- ...
- MatchResult **match()**

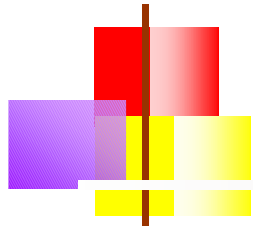
- String **next()**
- String **next**(Pattern pattern)
- String **next**(String pattern)



Scanner ...

- Plus puissant que StreamTokenizer
 - Parse de nombreux types : 18 "hasNextXXX()"
 - 2/4 modes de recherche : "findXXX()"
 - Séparateurs de token sous forme d'expressions régulières

```
String input = "1 fish 2 fish red fish blue fish";  
Scanner s = new Scanner(input);  
s.findInLine("(\\d+) fish (\\d+) fish (\\w+) fish (\\w+)");  
MatchResult result = s.match();  
for (int i=1; i<=result.groupCount(); i++)  
    System.out.println(result.group(i));  
s.close();
```



Scanner VS StreamTokenizer

String input = "1 fish 2 fish red fish blue fish";

```
Scanner s = new Scanner(input).useDelimiter("\\s*fish\\s*");  
    System.out.println(s.nextInt());  
    System.out.println(s.nextInt());  
    System.out.println(s.next());  
    System.out.println(s.next());  
    s.close();
```

Résultats :

1

2

red

blue



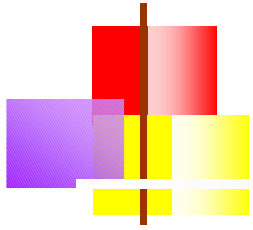
Scanner exercice

Reprendre l'exercice de comptage des tokens d'un fichier de `StreaTokenizer`



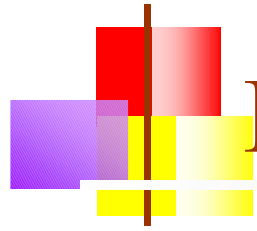
java.util.Scanner ...

Apparitions d'argument de type `Pattern` ou de type de retour `MatchResult`, classes qui font partie du paquetage `java.util.regex`



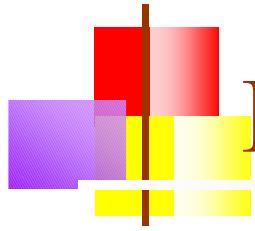
Expressions Régulières (ER)

`java.util.regex`



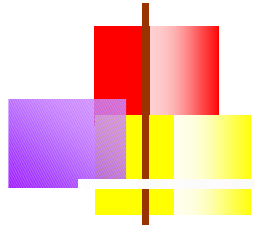
Expressions Régulières (ER)

- Une **ER** est une sorte de pattern (ou motif) qui peut être appliqué à un texte (**Strings**, en Java)
- Ou bien le texte (ou une partie du texte) est une instance de l'ER
- Ou bien la reconnaissance échoue.



Expressions Régulières (ER)

- SI une ER correspond à une partie du texte, on peut trouver facilement quelle partie.
 - SI une ER est complexe, on peut trouver facilement quelle partie de l' ER correspond à quelle partie du texte.
 - Avec cette information, on peut extraire des parties de textes en lisant, ou faire des substitutions dans le texte
-
- Les ER sont un outil essentiel pour la manipulation de texte.
 - Les ER sont très utilisées dans la génération automatique de pages Web.



regex(Perl) et java.util.regex

- Depuis Java 1.4, Java propose le paquetage :
java.util.regex
 - Les ER de Java sont pratiquement celles de Perl.
 - Elles augmentent énormément les capacités de Java en traitement de texte.
 - **java.util.regex** est un paquetage normal i.e. sans nouvelle syntaxe



java.util.regex.*

une interface

MatchResult implantée par la classe **Matcher**

2 classes :

Pattern : Représentation compilée d'un motif.

Matcher : Moteur de recherche d'un motif dans une chaîne

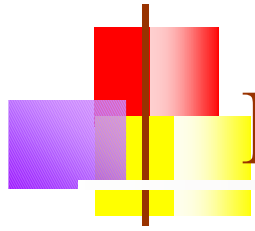
et une classe exception :

PatternSyntaxException : Exception levée lorsqu'une erreur apparaît dans la syntaxe des motifs employés.



java.util.regex : exemple

- L'ER "[a-z]+" matche/reconnaît toutes les séquences de lettres minuscules d'au moins une lettre.
 - [a-z] signifie *"tous les caractères de a à z inclus"*,
 - + signifie *"1 ou plus"*



Regex : exemple ⁽²⁾

- 3 manières d'appliquer ce motif "[a-z]+" sur la chaîne : "Now is the time"
 - *Sur toute la chaîne* : alors échec car la chaîne contient des caractères autres que des lettres et contient aussi des majuscules
 - *Sur le début de la chaîne* : Echec dès le premier caractère.
 - *Recherche du motif dans la chaîne* : 4 succès :
 - Tout d'abord **ow**
 - Puis en répétant la recherche **is**, et **the**, et **time**,
 - puis échec



java.util.regex, ⁽ⁱ⁾

- D'abord, il faut COMPILER le pattern dans un objet de la classe **Pattern**

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

```
Pattern p = Pattern.compile("[a-z]+");
```

- PUIS, Créer un **Matcher** pour un texte particulier en envoyant ce texte au pattern créé précédemment

```
Matcher m = p.matcher("Now is the time");
```



java.util.regex.Pattern

- **static Pattern compile(String regex)** : pas de constructeur mais compilation d'une ER pour obtenir un objet de la classe **Pattern**

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

```
Pattern p = Pattern.compile("[a-z]+");
```

- **Matcher matcher(CharSequence input)** : pour l'utilisation d'un pattern par un **Matcher** (en envoyant un texte au pattern)

```
Matcher m = p.matcher("Now is the time");
```



Pattern

- static boolean **matches**(String regex, CharSequence input): essai "local" de reconnaissance d'une chaîne **input** par l'ER **regex**

```
import java.util.regex.*;  
boolean b = Pattern.matches("[a-z]+", "azert);
```
- public static String **quote**(String s) : Returns a literal pattern String for the specified String.
- ...



java.util.regex, ⁽ⁱⁱ⁾

Remarques :

- **Pattern** et **Matcher** sont les classes de `java.util.regex`
- Ni **Pattern** ni **Matcher** n'ont de constructeur publique;
- Création d'instances par `compile()` et `matcher()`
- Un **Matcher** a l'information sur le **Pattern** à utiliser *ET sur* le texte sur lequel l'appliquer.



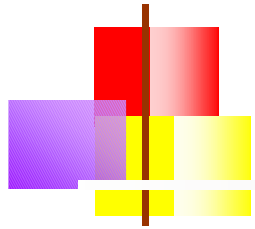
java.util.regex.Matcher... (iii)

- `Matcher m=p.matcher("...");` :
 - `m.matches()` retourne **true** si le pattern matche TOUT le texte, et **false** sinon
 - `m.lookingAt()` retourne **true** si le pattern matche le but du texte, et **false** sinon
 - `m.find()` retourne **true** si le pattern matche une partie du texte, et **false** sinon
 - Un nouvel appel de `m.find()` redémarre la recherche depuis le dernier succès
 - `m.find()` retournera **true** pour autant de succès trouvé dans le texte puis il retourne **false**
 - Quand `m.find()` retourne **false**, le matcher **m** se replace au début du texte (reset) (et peut être à nouveau utilisé)



Matcher : Trouver ce qui matche...

- *Après un succès*, `m.start()` retourne l'index du premier caractère de la sous-chaîne que le motif matche
- *Après un succès*, `m.end()` retourne l'index du dernier caractère de la sous-chaîne que le motif matche dans le texte, *plus un (+1)*
- Si pas de succès attendu ou si échec, alors `m.start()` et `m.end()` lève une exception `IllegalStateException`
 - C'est une `RuntimeException`, (donc pas de catch nécessaire)
- `m.end()` retourne l'index du dernier caractère du motif dans le texte, *plus un (+1)*
 - ainsi, `"Now is the time".substring(m.start(), m.end())` retourne exactement la sous-chaine matchée

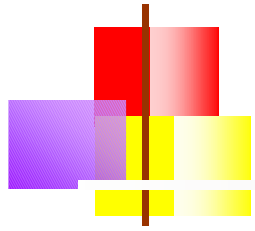


Matcher : un exemple complet

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

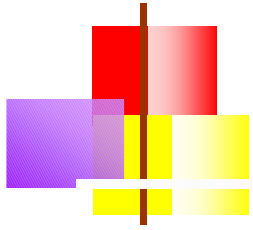
```
public class RegexTest {  
    public static void main(String args[]) {  
        String pattern = "[a-z]+";  
        String text = "Now is the time";  
        Pattern p = Pattern.compile(pattern);  
        Matcher m = p.matcher(text);  
        while (m.find()) {  
            System.out.print(text.substring(m.start(), m.end()) + "*");  
        }  
    }  
}
```

Output: `ow*is*the*time*`

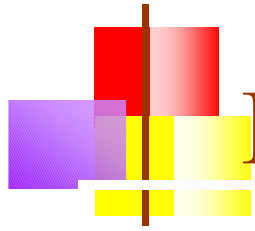


Matcher autres méthodes...

- Si `m` est un `Matcher`, alors
 - `m.replaceFirst(replacement)` retourne un nouveau texte où le première sous-chaine reconnue est remplacée par *replacement*
 - `m.replaceAll(replacement)` retourne un nouveau texte où toutes les sous-chainnes reconnues sont remplacées par *replacement*
 - `m.find(startIndex)` démarre la recherche à partir de l'index spécifié *startIndex*
 - `m.reset()` ré-initialise le matcher
 - `m.reset(newText)` ré-initialise matcher avec un nouveau texte à examiner (qui peut être de type `String`, `StringBuffer`, ou `CharBuffer`)

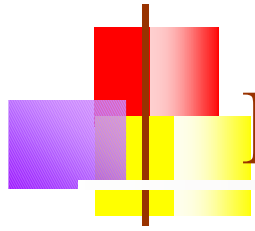


Pattern : syntaxe des ER



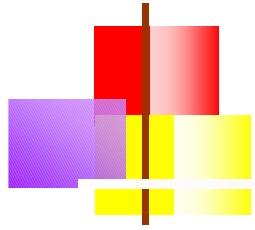
ER : motifs simples

- abc** exactement cette séquence de 3 lettres
- [abc]** une lettre parmi **a**, **b** et **c**
- [^abc]** tous les caractères *sauf* **a**, **b**, ou **c**
(immédiatement derrière **[**, **^** signifie “not,”
ailleurs il signifie simplement le caractère **^**)
- [a-z]** un caractère entre **a** et **z**, compris
- [a-zA-Z0-9]** une minuscule, une majuscule ou un chiffre



ER : Séquences and alternatives

- Si un motif suit un autre motif, les deux motifs doivent matcher "consécutivement"
 - exemple, `[A-Za-z]+[0-9]` matche "une ou plusieurs lettres immédiatement suivies par un chiffre"
- La barre, `|`, est utilisée pour l'alternative
 - exemple, le motif `abc|xyz` matche soit `abc` soit `xyz`



ER : Classes de caractères prédéfinies

- . Tout caractère sauf "fin de ligne"
- \d chiffre: [0-9]
- \D NON chiffre: [^0-9]
- \s espaces: [\t\n\x0B\f\r]
- \S NON espace: [^\s]
- \w caractères alphanumériques : [a-zA-Z_0-9]
- \W NON alphanumériques : [^\w]



ER : frontières

- Ces motifs(1 caractère) contraignent la position des ER:

^ début de ligne

\$ fin de ligne

\b aux frontières d'un mots

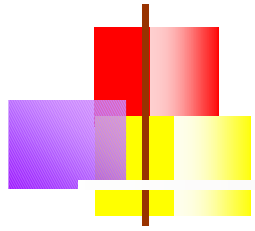
\B NON aux frontières d'un mots

\A début de l'entrée (peut être sur plusieurs lignes)

\Z à la fin de l'entrée sauf si caractère de fin particulier

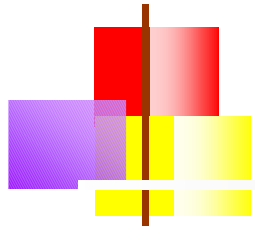
\z à la fin de l'entrée

\G à la fin du match précédent



ER : quantifieurs

- les **avides** (*greedy*) : lecture de toute la chaîne d'entrée avant de rechercher des occurrences en partant de la fin dans le but de trouver le maximum d'occurrences,
 - $X?$ X^* X^+ $X\{n\}$ $X\{n,\}$ $X\{n,m\}$
- - les **réticents** (*reluctant*) : lecture de la chaîne d'entrée caractère par caractère à la recherche des occurrences,
 - On obtient un quantifieur réticent en ajoutant $?$ par exemple
 $X??$ $X*?$ $X+?$ $X\{n\}?$ $X\{n,\}?$ $X\{n,m\}?$
- - les **possessifs** (*possessive*) : lecture de toute la chaîne d'entrée avant de chercher une occurrence.
 - On obtient un quantifieur réticent en ajoutant $+$ par exemple
 - $X?+$ $X*+$ X^{++} $X\{n\}+$ $X\{n,\}+$ $X\{n,m\}+$



ER : Quantifieurs "avides" (*Greedy*)

Soit X représentant un motif quelconque

$X?$ optionel, 1 ou 0 occurrence de X

X^* 0 ou plus occurrence(s) de X

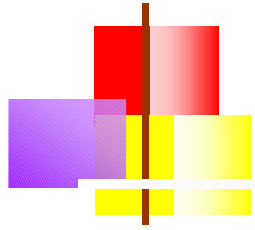
X^+ 1 ou plus occurrence(s) de X

$X\{n\}$ exactement n occurrence(s) de X

$X\{n,\}$ au moins n occurrence(s) de X

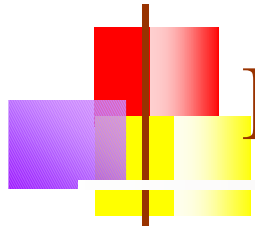
$X\{n,m\}$ entre n et m occurrences de X

Remarque : opérateurs *postfixes*



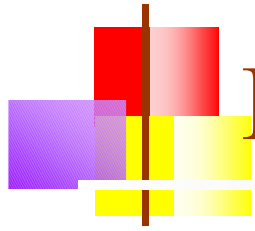
ER, quantifieurs : exemples

- Soit le texte **aardvark**
 - motif **a*ardvark** (**a*** est avide):
 - **a*** matche d'abord **aa**, mais pas **ardvark**
 - puis **a*** “backs off” et matche **a**, seul permettant au reste du motif (**ardvark**) de réussir
 - motif **a*?ardvark** (**a*?** est réticent):
 - **a*?** Matche d'abord 0 caractère (la chaîne vide), et ensuite **ardvark** ne matche pas
 - **a*?** Alors est étendue et matche le premier **a**, ensuite le reste du motif matche (**ardvark**) donc succès
 - motif **a*+ardvark** (**a*+** est possessif):
 - **a*+** matche **aa**, mais pas de “back off”, ainsi **ardvark** ne matche jamais et on obtient un échec



ER : Groupes de capture

- Dans les ER, les parenthèses sont utilisées pour le groupage, mais aussi elles **capturent** (pour un usage ultérieur) ce qui est matché par cette partie du motif
 - Exemple: `([a-zA-Z]*)([0-9]*)` matche n'importe quelle suite de lettres suivie de n'importe quelle suite de chiffres
 - Si succès, `\1` repère la suite de lettres et `\2` repère la suite de chiffres
 - De plus, `\0` repère ce qui matche avec le motif complet
- Les groupes de capture sont numérotés en comptant les parenthèses ouvrantes de gauche à droite :
 - $((A)(B(C)))$
 $\backslash 0 = \backslash 1 = ((A)(B(C))), \quad \backslash 2 = (A), \quad \backslash 3 = (B(C)), \quad \backslash 4 = (C)$
- Exemple: `([a-zA-Z])\1` va matcher une lettre double comme dans, letter. **ATTENTION** : `Pattern.compile("([a-zA-Z])\1");`



ER : Groupes de capture et Java

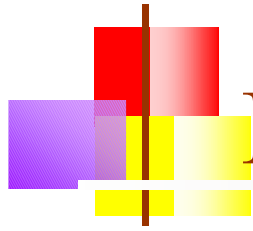
- Si **m** est un Matcher qui vient de "matcher" avec succès, alors
 - **m.group(*n*)** retourne la chaîne matchée par le groupe de capture *n*
 - Ce peut être la chaîne vide
 - retourne **null** si le motif a un match global mais ce groupe particulier ne matche
 - **m.group()** retourne la chaîne matchée par le motif complet (idem **m.group(0)**) Ce peut être la chaîne vide

si **m** échoue (ou n'a pas été utilisé), exception **IllegalStateException** levée.



ER : Exemple

- Soit **word** un mot anglais
- supposons que nous voulions transférer toutes les consonnes au début de **word** (s'il y en a) à la fin de **word** (exemple **string** devient **ingstr**)
 - ```
Pattern p = Pattern.compile("([aeiou]*)(.*)");
Matcher m = p.matcher("word");
if (m.matches()) {
 System.out.println(m.group(2) + m.group(1));
}
```
- remarque **(.\*)** utilisé pour "tous les caractères restants"



## ER : Double backslashes

---

- Les Backslashes ont une signification particulière dans les ER. Exemple , `\b` signifie "frontière de mot"
- Les Backslashes ont aussi une signification particulière en java. Exemple, `\b` signifie "caractère backspace"
- La syntaxe Java s'applique d'abord !
  - Si on a écrit `"\b[a-z]+\b"` on obtient une chaîne avec des caractères backspace : ce n'est pas ce que l'on veut !
  - `"\\b[a-z]+\\b"` est la chaîne pour le motif correct



# ER : Escaping metacharacters

---

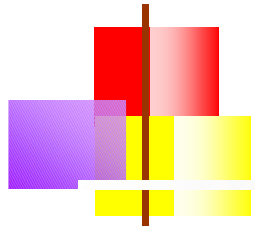
- A lot of special characters--parentheses, brackets, braces, stars, plus signs, etc.--are used in defining ER. these are called **metacharacters**
- Suppose you want to search for the character sequence **a\*** (an **a** followed by a star)
  - "a\*"; doesn't work; that means "zero or more **a**s"
  - "a\\*"; doesn't work; since a star doesn't *need* to be escaped (in Java String constants), Java just ignores the \
  - "a\\\*" *does* work; it's the three-character string **a**, **\**, **\***
- Just to make things even more difficult, it's *illegal* to escape a *non*-metacharacter in a ER



# ER : Espaces

---

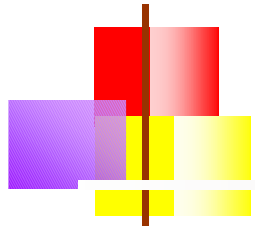
- There is only one thing to be said about spaces (blanks) in ER, but it's important :
  - *Spaces are significant!*
- A space stands for a *space*--when you put a space in a pattern, that means to match a space in the text string
- It's a *really bad idea* to put spaces in a ER just to make it look better



# Java 1.4 : Les ajouts dans String

---

- `public boolean matches(String regex)`
- `public String replaceFirst(String regex, String replacement)`
- `public String replaceAll(String regex, String replacement)`
- `public String[ ] split(String regex)`
- `public String[ ] split(String regex, int limit)`
  - *If the limit  $n$  is greater than zero then the pattern will be applied at most  $n - 1$  times, the array's length will be no greater than  $n$ , and the array's last entry will contain all input beyond the last matched delimiter.*
  - *If  $n$  is non-positive then the pattern will be applied as many times as possible*



# The End : Thinking in ER

---

- Les ER are *not* easy to use at first
  - It's a bunch of punctuation, not words
  - The individual pieces are not hard, but it takes practice to learn to put them together correctly
  - ER form a miniature programming language
    - It's a different kind of programming language than Java, and requires you to learn new thought patterns
  - In Java you can't just *use* a ER. you have to first create Patterns and Matchers
  - Java's syntax for String constants doesn't help, either
- Despite all this, ER bring so much power and convenience to String manipulation that they are well worth the effort of learning