Entrées-sorties 2

Université de Nice - Sophia Antipolis
Version 2.0.2 – 13/1/12
Richard Grin

Plan de cette partie 2

- Expressions régulières
- Mise en forme des entrées-sorties
- Imprimer
- Séparation des entrées-sorties

Expressions régulières

Généralités

- JDK 1.4 a ajouté des classes pour traiter les expressions régulières « à la Perl 5.0 »
- La classe **`String`** a aussi été complétée par des nouvelles méthodes de traitement des expressions régulières
- On peut ainsi facilement rechercher des chaînes de caractères correspondant à un certain modèle, décomposer une chaîne en sous-chaînes, en extraire des sous-chaînes ou en modifier une partie

Syntaxe des expressions régulières

- La syntaxe est complexe mais on peut se restreindre à n'utiliser que les formes les plus simples, bien connues si on travaille avec Perl ou Unix
- Consulter la documentation de la classe **`java.util.regex.Pattern`**

Un caractère

- **`.`** : n'importe quel caractère, sauf une fin de ligne si on n'est pas en mode **`DOTALL`** (voir méthode **`compile`** de la classe **`Pattern`**
- **`\t`**, **`\n`**, **`\r`**, **`\f`** (form feed), **`\e`** (escape) ont leur signification habituelle
- **`\c`*`x`*** est le caractère Ctrl ***`x`***
- **`\u`*`hhhh`*** : caractère unicode dont le code hexadécimal est ***`hhhh`***

Un caractère

- **`[abc]`** : a, b ou c
- **`[^abc]`** : ni a, ni b, ni c
- **`[a-l]`** : de a à l ; **`[^a-l]`** : pas de a à l
- **`[a-z&&[def]]`** : d, e ou f (intersection)
- **`[a-z&&[^def]]`** : a à z, sauf d, e ou f
- **`[a-zA-Z]`** : union
- **`[a-z[A-Z]]`** : union

« Échapper » un caractère

- **`\`** : enlève la signification spéciale du caractère suivant pour une expression régulière
- Attention de ne pas oublier de doubler les « \ » dans le code Java :
  - par exemple, pour rechercher $ on doit donner la chaîne « **`\\$`** »
  - pour rechercher « \ », on doit entrer la chaîne « **`\\\\`** » ; \ est un caractère spécial !
  - mais pour rechercher un guillemet, il faut entrer « **`\"`** »

Types de caractères prédéfinis

- **`\d`** : un chiffre ; **`\D`** : pas un chiffre
- **`\s`** : un espace « blanc » (espace, **`\t`**, **`\n`**, **`\f`**, **`\r`**)
- **`\S`** : pas un espace blanc
- **`\w`** : une lettre ou un chiffre
- **`\W`** : ni une lettre, ni un chiffre

« Bords »

- **`^`** : début (et début de ligne si mode **`MULTILINE`**, option de méthode **`compile`** étudiée plus loin)
- **`$`** : fin (et fin de ligne si mode **`MULTILINE`**)
- **`\b`** : début ou fin de mot
- **`\B`** : pas un début ou fin de mot
- **`\A`** : le début du texte
- **`\z`** : la fin du texte
- **`\G`** : la fin de la précédente chaîne correspondant à l'expression régulière

Quantifieurs

- Soit X un caractère ou un groupe
- **`X?`** : 0 ou 1 fois X (X est une expression régulière quelconque)
- **`X*`** : 0 ou plusieurs fois X
- **`X+`** : 1 ou plusieurs fois X
- **`X{n}`** : n fois X ; **`X{n,}`** : au moins n fois X
- **`X{n,m}`** : n à m fois X

Glouton ou pas ?

- Par défaut, la recherche d'une chaîne de caractères qui correspond à une expression régulière englobe le plus de caractères possible, tant qu'une chaîne correspond à l'expression régulière
- Si on veut que le moins de caractères possible soient englobés, il faut ajouter **`?`** derrière un quantifieur

Exemple

- Si on cherche dans la chaîne **`Abcdec`**,
- Les expressions règulières **`"A.*c"`** et **`"A.+c"`** correspondent à **`Abcdec`**
- Les expressions règulières **`"A.*?c"`** et **`"A.+?c"`** correspondent à **`Abc`**

Super-glouton

- Un 3ème mode (**`+`** ajouté derrière le quantifieur) mange le plus possible de caractères, même si ça fait ignorer une chaîne qui pourrait correspondre à l'expression régulière
- Exemple : avec la chaîne **`Abdec`** les expressions règulières **`"A.*+c"`** et **`"A.++c"`** ne correspondent à rien car le caractère **`c`** final est « mangé » par **`.*+`** ou **`.++`**

Opérateurs

- Si **`X`** et **`Y`** sont des expressions régulières,
- **`XY`** : X suivi de Y
- **`X|Y`** : X ou Y

Groupes

- **`(X)`** : X, groupe « capturé »
- **`\`*n*** : le *n*ième groupe capturé
- Les groupes sont numérotés selon l'occurrence de la parenthèse ouvrante, en commençant par 1 pour la parenthèse la plus à gauche
- Le groupe spécial de numéro 0 correspond à toute la chaîne qui correspond à l'expression régulière
- Attention, si l'expression régulière est de la forme **`X | Y`**, on ne peut travailler avec les groupes de Y

Autres possibilités (1)

- Si vous ne trouvez pas votre bonheur dans la syntaxe qui vient d'être exposée, allez voir d'autres possibilités dans la documentation de la classe **`Pattern`**
- Quelques exemples :
  `[a-z&&[^bc]]` : de a à z, sauf b et c (**`&&`** indique une intersection)
  `\p{Alpha}` une lettre
  `\p{Blank}` un espace ou une tabulation

Autres possibilités (2)

- D'autres exemples :
  `\d+(?!€)` : repère des chiffres qui ne sont pas suivis par le caractère « **`€`** » (*negative lookahead*)
  `(?<!\d+)€` : repère le caractère « **`€`** » qui n'est pas précédé par des chiffres (*negative lookbehind*)
  `(?:X)` : pour regrouper, par exemple pour changer la priorité des opérateurs, sans créer un groupe

Nouvelles méthodes de **`String`** (1)

- **`boolean matches(String`** ***`expreg`*****`)`**
  renvoie vrai si la chaîne (en entier) correspond à l'expression régulière
- **`String replaceAll(String`** ***`expreg`*****`, String`** ***`remplacement`*****`)`**
  renvoie une chaîne dans laquelle les sous-chaînes qui correspondent à ***`expreg`*** sont remplacées par la chaîne ***`remplacement`*** ; ***`remplacement`*** peut comporter des **`$`*****`n`*** pour désigner des portions parenthésées de ***`expreg`***
- Variante : **`replaceFirst`**

Exemple

```
String phrase =
  "Bonjour bonhomme veux-tu un bon bonbon ?";
String phraseModifiee =
  phrase.replaceAll("\\bbon(\\w+)", "mal$1");
System.out.println(phraseModifiee);
```

- affichera
  `Bonjour malhomme veux-tu un bon malbon ?`
- Si on veut ne pas tenir compte des majuscules-minuscules, il faut utiliser les classes dédiées **`Pattern`** et **`Matcher`**

Nouvelle méthode **`split`** de **`String`**

- **`String[] split(String`** ***`expreg`*****`)`**
  éclate la chaîne en un tableau de chaînes séparées par un délimiteur qui correspond à l'expression régulière
- 2 délimiteurs accolés délimitent une chaîne vide (pas la valeur **`null`**)
- Les valeurs vides finales sont ignorées

Compléments sur **`split`**

- Un 2ème paramètre n de type **`int`** peut aussi servir à limiter à n - 1 le nombre de recherches des délimiteurs (le dernier élément contient la fin de la chaîne), si n > 0
- Il peut faire prendre en compte les valeurs vides finales, si n est ≠ 0
- n = 0 correspond au cas où il n'y a pas de 2ème paramètre
- Cette méthode rend obsolète l'utilisation de la classe **`StringTokenizer`**

Exemples (1)

- Décomposer en « mots » séparés par un seul caractère blanc (4 mots « c'est », « » (chaîne vide), « un », « test. ») :

```
String[] resultat =
  "c'est  un test.".split("\\s");
for (int i = 0; i < resultat.length; x++)
  System.out.println(resultat[i]);
```

2 espaces

- Décomposer en « mots » séparés par un ou plusieurs caractères blancs (3 mots « c'est », « un », « test. ») :

```
String[] resultat =
  "c'est  un test.".split("\\s+");
. . .
```


Exemples (2)

- Décomposer en « mots » séparés par un ou plusieurs caractères non lettre ou chiffre (4 mots « c », « est », « un », « test ») :

```
String[] resultat =
  "c'est  un test.".split("\\W+");
. . .
```

- Décomposer en « mots » séparés par un ou plusieurs caractères non lettre, non chiffre, et pas « **`'`** » (3 mots « c'est », « un », « test ») :

```
String[] resultat =
  "c'est  un test.".split("[^a-zA-Z0-9']+");
. . .
```

Exemples (3)

- Attention, avec un seul paramètre, **`split`** ignore une ou plusieurs valeurs vides finales : **`"toto,,machin,".split(",");`** renvoie un tableau de 3 valeurs (une chaîne vide entre toto et machin, mais pas à la fin)
- **`"toto,,machin,".split(",", -1);`** renvoie un tableau de 4 valeurs (une chaîne vide entre toto et machin, et à la fin)

Classes dédiées

- Ces méthodes de **`String`** sont des raccourcis qui utilisent les classes **`Pattern`** et **`Matcher`** (du paquetage **`java.util.regex`**) dédiées aux expressions régulières

Classes dédiées

- Si on veut effectuer plusieurs recherches ou remplacement avec une même expression régulière, il est plus performant d'utiliser directement ces classes
- Elles permettent aussi d'effectuer des traitements complexes, autres que des simples remplacements, à chaque occurrence d'une chaîne correspondant à une expression régulière

Classe **`Pattern`**

- Correspond à une expression régulière « compilée », c'est-à-dire préparée pour que les futures recherches soient accélérées
- Si une expression régulière est utilisée plusieurs fois, on a intérêt à la compiler (donc à utiliser **`Pattern`** et pas directement les méthodes de **`String`**)
- La javadoc de cette classe décrit la syntaxe des expressions régulières

Méthode **`compile`**

- **`static Pattern compile(String exprReg)`** compile une expression régulière
- **`compile`** est surchargée ; un 2ème paramètre de type **`int`** joue sur les recherches futures de l'expression régulière (voir transparent suivant)

Options de la méthode **`compile`**

- Ce 2ème paramètre est un drapeau qui peut correspondre à plusieurs options (utiliser « **`|`** » entre 2 options pour « ajouter » les bits)
- Exemple : si on veut ne pas tenir compte des majuscules et considérer que « . » peut correspondre à une fin de ligne, on écrit :

```
Pattern patternTR =
  Pattern.compile("truc(.*?)machin",
    Pattern.CASE_INSENSITIVE |
    Pattern.DOTALL);
```

Options de la méthode **`compile`**

- **`CASE_INSENSITIVE`** : ne pas tenir compte des majuscules/minuscules (par défaut, seules les lettres du code ASCII sont prises en compte, donc pas les lettres accentuées)
- **`UNICODE_CASE`** : toutes les lettres du code unicode sont prises en compte pour l'option précédente
- **`DOTALL`** : « **`.`** » inclut les caractères de fin de ligne (utile quand une expression régulière peut correspondre à une chaîne de caractères sur plusieurs lignes)
- **`MULTILINE`** : **`^`** et **`$`** correspondent à un début ou fin de ligne (ou un début ou fin d'entrée)

Interface **`CharSequence`**

- L'interface **`java.lang.CharSequence`**, ajoutée depuis le JDK 1.4, est implémentée par les classes **`String`**, **`StringBuilder`** et **`StringBuffer`** (et **`java.nio.CharBuffer`**)
- Elle représente une suite de caractères lisibles (**`char`**) dont on peut extraire une sous-suite
- Les chaînes de caractères dans lesquelles on recherche une expression régulière sont du type **`CharSequence`**

Méthodes de **`Pattern`**

- **`String[] split(CharSequence texte)`** éclate le texte en prenant le *pattern* comme délimiteur ; un 2ème paramètre permet de limiter le nombre de morceaux (voir javadoc)
- **`static boolean matches(String exprReg, CharSequence texte)`** permet de faire directement une recherche sans passer explicitement par la classe **`Matcher`** (idem **`Pattern.compile(exprReg).matcher(texte).matches()`**)

Méthode **`matcher`**

- **`Matcher matcher(CharSequence texte)`** renvoie une instance de **`Matcher`** pour rechercher l'expression régulière dans **`texte`** (et éventuellement effectuer des remplacements)

Classe **`Matcher`**

- Permet de rechercher des chaînes qui correspondent à une expression régulière
- On obtient une instance avec la méthode **`matcher`** de la classe **`Pattern`** en donnant le texte dans lequel se fera la recherche
- Les méthodes **`reset(CharSequence)`** et **`reset()`** réinitialisent un matcher pour effectuer une nouvelle recherche

Méthodes de la classe **`Matcher`**

- **`replaceAll`** , **`replaceFirst`**, **`appendReplacement`** et **`appendTail`** remplacent l'expression régulière par une chaîne de caractères
- **`lookingAt`**, **`find`** et **`matches`** renvoient **`true`** si une expression régulière a été trouvée
- Si l'expression a été trouvée, on peut en savoir plus avec les méthodes **`start`**, **`end`** et **`group`**

Méthodes de la classe **`Matcher`**

- **`String replaceAll(String remplacement)`** renvoie une nouvelle chaîne dans laquelle toutes les sous-chaînes qui correspondent au pattern sont remplacées par **`remplacement`** ; remplacement peut contenir des références à des sous-groupes du pattern (**`$`***n*)
- La variante **`replaceFirst`** ne remplace que la 1ère occurrence du pattern

Méthodes de la classe **`Matcher`**

- **`boolean matches()`** renvoie vrai si le texte en entier correspond à l'expression régulière
- **`boolean lookingAt()`** renvoie vrai si le début du texte correspond à l'expression régulière
- **`boolean find()`** recherche la prochaine occurrence de l'expression régulière (utilisé le plus souvent dans une boucle) ; renvoie **`false`** s'il ne reste plus d'occurrence ; **`find(int debut)`** ne recherche qu'à partir du caractère numéro **`debut`**

Informations sur la sous-chaîne

- Quand une sous-chaîne qui correspond à l'expression régulière vient d'être trouvée (par **`lookingAt`**, **`find`** ou autre), on peut avoir des informations sur cette sous-chaîne
- **`String group()`** renvoie la sous-chaîne
- **`int start()`** donne le numéro du 1er caractère
- **`int end()`** donne le numéro du dernier caractère + 1

Groupes

- Une expression régulière peut contenir des portions entre parenthèses
- Par exemple, cette expression correspond aux sous-chaînes formées d'un mot qui commence par « bon » ; la portion entre parenthèses désigne le reste du mot : **`\bbon(\w*)`**
- Toutes les portions entre parenthèses sont appelées des groupes et sont numérotées par ordre d'apparition de leur parenthèse ouvrante dans l'expression régulière (le groupe 0 désigne toute la sous-chaîne)

Méthodes pour les groupes

- **`String group(int n)`** renvoie la sous-chaîne qui correspond au **`n`**ième groupe de l'occurrence qui vient d'être trouvée (**`group(0)`** est équivalent à **`group()`**)
- On obtient le numéro du caractère qui débute et termine (+ 1) chaque groupe avec les méthodes **`int start(int n)`** et **`int end(int n)`**

Exemple de code

```
import java.util.regex.*;
. . .
String phrase = "Bonjour bonhomme veux-tu un
  bon bonbon ?";
String expReg = "\\bbon(\\w+)";
Pattern p = Pattern.compile(
  expReg, Pattern.CASE_INSENSITIVE);
Matcher m = p.matcher(phrase);
System.out.println(m.replaceAll("mal$1"));
System.out.println(m.matches());
System.out.println(m.lookingAt());
System.out.println(m.find());
```

Fin du code

```
// Affiche détails de la recherche
while (m.find()) {
  System.out.print("Trouvé " + m.group()
                   + " en position "
                   + m.start());
  System.out.println(" suffixe : "
                     + m.group(1)
                     + " en position "
                     + m.start(1));
}
```


Affichage de l'exemple

```
maljour malhomme veux-tu un bon malbon ?
false
true
true
Trouvé Bonjour en position 0 ; suffixe :
  jour
Trouvé bonhomme en position 8 ; suffixe :
  homme
Trouvé bonbon en position 32 ; suffixe :
  bon
```


Remplacement

- La méthode **`appendReplacement`** offre plus de souplesse que **`replaceAll`** : elle ajoute dans un **`StringBuffer`**, en effectuant les remplacements un à un
- On peut choisir ce que l'on fait entre chaque remplacement

Remplacement

- **`Matcher appendReplacement(StringBuffer nouvTexte, String remplace)`** : ajoute les caractères jusqu'à la fin de la précédente correspondance (par **`find`** ou autre), son remplacement compris
- **`StringBuffer appendTail(StringBuffer nouvTexte)`** : ajoute les caractères qui restent après la dernière correspondance

Exemple de remplacement

```
// Même phrase que l'exemple précédent,
// avec la même expression régulière
StringBuffer sb = new StringBuffer();
while (m.find()) {
  m.appendReplacement(sb, "mal$1");
  System.out.println(sb);
}
// Ajoute ce qui suit la dernière occurrence
m.appendTail(sb);
System.out.println(sb);
```


Affichage de l'exemple

Phrase du départ :

Bonjour bonhomme veux-tu un bon bonbon ?

Pattern : \bbon(\w+)

```
maljour
maljour malhomme
maljour malhomme veux-tu un bon malbon
maljour malhomme veux-tu un bon malbon ?
```

Réinitialisation

- Ces 2 méthodes réinitialisent le matcher en mettant la position courante au début (cf. **`find`** et **`appendReplacement`**)
- **`Matcher reset()`** : remet la position courante au début du texte actuel
- **`Matcher reset(CharSequence texte)`** : change de texte et met la position courante au début

Interface `MatchResult`

- Paquetage **`java.util.regex`**
- Représente une des occurrences trouvées lors d'une recherche effectuée par un matcher ; elle a été introduite par le JDK 5
- La méthode **`toMatchResult()`** de la classe **`Matcher`** renvoie un **`MatchResult`** que l'on peut ensuite interroger par les méthodes **`end`**, **`group`**, **`start`**
- Cet objet renvoyé conserve les informations, même après le déplacement du matcher à la prochaine occurrence

Exemple

```
List<MatchResult> resultats =
  new ArrayList<MatchResult>();
Matcher m = pattern.matcher(texte);
while(m.find())
  resultats.add(m.toMatchResult());
// On peut ensuite traiter les occurrences
// ou les passer à une autre méthode
. . .
```


Recherche dans un fichier

- Si le fichier n'est pas trop grand, on peut lire le fichier dans une **`String`** ou **`StringBuffer`** dans laquelle on effectue alors les recherches
- Si on peut travailler ligne par ligne, on peut aussi lire le fichier ligne par ligne et ensuite de rechercher dans chaque ligne
- Sinon, on peut d'abord transformer le fichier en un **`CharBuffer`** (du paquetage **`java.nio`**, qui implémente **`CharSequence`**) pour le passer à un pattern

Exemple de recherche dans un fichier

```
FileInputStream fis = . . .;
FileChannel fc = fis.getChannel();
ByteBuffer bbuf =
  fc.map(FileChannel.MapMode.READ_ONLY,
         0, (int)fc.size());
CharBuffer cbuf =
  Charset.forName("8859_1").newDecoder().
    decode(bbuf);
Pattern pattern = Pattern.compile("pat");
Matcher matcher = pattern.matcher(cbuf);
while (matcher.find())
  String match = matcher.group();
```


Utilisation de type envoi de mailing

- Avec les expressions régulières on peut créer un document qui contient des modèles particuliers, par exemple **`<#nom#>`** qui seront remplacés par des valeurs correspondants au modèle (par exemple, la valeur de la variable « nom »)
- *StringTemplate* est un projet open source qui a le même type de fonctionnalité (publipostage)

StringTemplate

- Projet issu du projet open source ANTLR, qui permet de générer des textes à partir d'un modèle qui contient des « trous » à combler avec des données
- Les trous à combler peuvent comporter des « if », ce qui permet, par exemple, d'insérer dans le document « Monsieur » ou « Madame » suivant le sexe d'une personne

Mise en forme des sorties

- En Java, pas de méthode *printf* (comme dans le langage C) avant le JDK 5
- Sans entrer dans les détails, en particulier en ce qui concerne l'internationalisation, voici quelques exemples qui présentent des classes et méthodes utiles pour mettre en forme des sorties

Paquetage **`java.text`**

- Pour mettre en forme les nombres et dates
- La classe **`NumberFormat`** permet de mettre en forme des nombres, des valeurs monétaires et des pourcentages
- Pour les dates, **`DateFormat`**
- Ces 2 classes suffisent pour les formats les plus courants ; sinon, on peut utiliser leur classe fille **`DecimalFormat`** ou **`SimpleDateFormat`**

Mise en forme des nombres

```
import java.text.NumberFormat;
import java.util.Locale;
. . .
NumberFormat nf = NumberFormat.getInstance();
System.out.println(nf.format(1234567.256));
System.out.println(nf.format(1234567.2));
System.out.println(nf.format(1234567));
nf = NumberFormat.getInstance(Locale.US);
System.out.println(nf.format(1234567.2));
```

Locale en cours

affichera (*locale* courante **`French`**)

```
1 234 567,256
1 234 567,2
1 234 567
1,234,567.2
```


Autres méthodes de **`NumberFormat`**

- Donner le nombre minimum ou maximum de la portion du nombre située avant le séparateur décimal :
  `setMinimumIntegerDigits(int n)`
  `setMaximumIntegerDigits(int n)`
- Idem pour la partie décimale :
  `setMinimumFractionDigits(int n)`
  `setMaximumFractionDigits(int n)`

Mise en forme spéciale de nombres

```
import java.text.*;
. . .
NumberFormat nf = new DecimalFormat("00,000,000.00");
System.out.println(nf.format(1234567.256));
System.out.println(nf.format(1234567.2));
nf = new DecimalFormat("##,000,000.##");
System.out.println(nf.format(1234567.2));
System.out.println(nf.format(1234567));
```

affichera (*locale* courante **`French`**)

```
01 234 567,26
01 234 567,20
1 234 567,2
1 234 567
```

Les séparateurs de milliers et décimales sont indépendants de la locale en cours

Mise en forme des dates

```
Date maintenant = new Date();
DateFormat df = DateFormat.getDateInstance();
System.out.println(df.format(maintenant));
df = DateFormat.getDateInstance(DateFormat.MEDIUM,
                                Locale.US);
System.out.println(df.format(maintenant));
```

affichera

```
16 oct. 01
Oct 16, 2001
```


Mise en forme spéciale de dates

```
SimpleDateFormat sdf
  = new SimpleDateFormat("yyyy.MM.dd G 'à' "
                         + "hh:mm:ss a zzz");
Date maintenant = new Date();
System.out.println(sdf.format(maintenant));
sdf = new SimpleDateFormat("dd/MM/yyyy 'à' "
                           + "HH:mm:ss");
System.out.println(sdf.format(maintenant));
```

Attention aux majuscules/ minuscules

affichera

```
2000.08.07 ap. J.-C. à 02:15:20 PM CEST
07/08/2000 à 14:15:20
```


Autres types de mise en forme

- **`NumberFormat`** permet aussi de mettre en forme des valeurs monétaires et des pourcentages avec les méthodes **`getCurrencyInstance`** et **`getPercentInstance`**

Encore plus pour formater…

- Les classes suivantes (du paquetage **`java.text`**) permettent d'aller plus loin pour formater des messages :
  - **`ChoiceFormat`** pour formater différemment un message selon la valeur d'un nombre
  - **`MessageFormat`** pour définir des messages avec du texte fixe et des parties variables
- Le plus souvent ces classes sont utilisées avec des fichiers de ressources pour internationaliser les applications

Exemple avec **`Choice`**

```
double[] limites = {0,1,2,10};
String[] chaines =
  {"Pas de fichier", "Un fichier",
   "des fichiers", "beaucoup de fichiers"};
ChoiceFormat choice =
  new ChoiceFormat(limites, chaines);
for (int i = -1; i < 14; i++) {
  System.out.println(i + " : "
                     + choice.format(i));
}
```

Exemple avec **MessageFormat**

```
double[] limites = {0,2};
String[] chaines = {"fichier", "fichiers"};
ChoiceFormat choice = new
  ChoiceFormat(limites, chaines);
MessageFormat mf = new MessageFormat(
    "On a trouvé {0,number,integer} {1}");
for (int i = 0; i < 5; i++) {
  String choix = choice.format(i);
  Object[] arguments =
    new Object[] {new Integer(i), choix};
  System.out.println(mf.format(arguments));
}
```


Nouveautés du JDK 5

- Des nouvelles méthodes **format** ont été ajoutées aux classes d'écriture dans un flot et à la classe **String** pour faciliter la mise en forme « à la printf de C »

Méthodes **format**

- Les classes **PrintStream** et **PrintWriter** contiennent des méthodes **format** qui permettent de mettre en forme ce que l'on envoie dans le flot de sortie
- Types des paramètres :
  Locale *l*, String *format*, Object... *args*
- Une variante prend la locale par défaut :
  String *format*, Object... *args*
- Les méthodes renvoient l'instance de la classe (**this** ; on peut ainsi enchaîner)

String.format

- La classe **String** contient elle aussi 2 méthodes **static format** qui renvoient une **String** (elles ont les mêmes types de paramètres que les méthodes de **PrintStream** et **PrintWriter**)
- Elles permettent de récupérer une chaîne de caractères mise en forme

Méthodes **printf**

- Pour ne pas décontenancer les programmeurs C, les classes **PrintStream** et **PrintWriter** ont 2 méthodes **printf** qui font exactement la même chose que les méthodes **format**

Exemples simples

```
System.err.printf(
  "Unable to open file '%1$s': %2$s",
  fileName,
  exception.getMessage());
```

```
System.err.printf(
  "Unable to open file '%s': %s",
  fileName,
  exception.getMessage());
```

Syntaxe des formats

- **`%[n$][option][larg][.précision]type`**
- **`n`** : numéro ordre ; **`3$`** se réfère au 3ème paramètre
- **`option`** : par exemple, « - » pour justifier à gauche, « , » le nombre comportera des séparateurs de milliers, « 0 » pour imposer 0 au début (**`larg`** doit être donnée)
- **`larg`** : largeur minimale de la zone réservée
- **`précision`** : nombre de chiffres après la virgule pour les nombres décimaux, largeur maximale pour les autres types
- **`type`** : le type du paramètre ; quelques types : **`s`** pour **`String`**, **`d`** pour un entier décimal, **`f`** pour un réel

Passage à la ligne

- **`%n`** (la lettre n) fait passer à la ligne
- Tient compte de la plateforme sur laquelle on travaille

Mise en forme de dates

- On peut aussi mettre en forme des dates et des heures ou des **`BigDecimal`**
- Les dates et les heures sont adaptées à la locale choisie (ou à la locale par défaut)
- L'exemple suivant montre aussi l'utilité du numéro d'ordre (**`$1`**)
- ```
  Calendar c = ...;
  String s = String.format(
  "Anniversaire : %1$te %1$tB %1$tY", c);
  ```


Syntaxe complète des formats

- Seule une petite partie des possibilités a été donnée dans ce cours
- On peut trouver la syntaxe complète dans la javadoc de la classe **`java.util.Formatter`**

Classe `java.util.Formatter`

- En fait les méthodes **`format`** que l'on a vues utilisent la classe **`Formatter`**
- On passe au constructeur un paramètre de type **`Appendable`** ou **`String`** (pour un nom de fichier) qui représente la destination de ce qui sera mis en forme
- La méthode **`format`** envoie une représentation formatée des paramètres vers la destination donnée dans le constructeur

Scanner des entrées
  ```

Classe `java.util.Scanner`

- Depuis le JDK 5 la classe **`Scanner`** permet de récupérer des données au milieu d'un flot d'entrée, à l'image de la fonction *scanf* du langage C
- Elle implémente **`iterator<String>`**

Utilisation

- On crée une instance de **`Scanner`** avec un de ses nombreux constructeurs
- On lui passe en paramètre le flot d'entrée dans lequel on va extraire les données ; par exemple, pour lire l'entrée standard :
```
Scanner sc = new Scanner(System.in);
```
- On extrait ensuite les données une à une avec une des méthodes nextXXX :
```
int i = sc.nextInt();
```


Constructeurs

- Ils peuvent prendre en paramètre une instance de **`java.lang.Readable`** (source de données de type caractère), **`java.nio.channels.ReadableByteChannel`** (source de données de type octet) ou **`InputStream`**, ou **`File`**, ou **`String`**
- Pour **`InputStream`** et **`File`**, on peut indiquer le nom d'un codage pour les caractères (voir classe **`java.nio.Charset`** ; en France c'est par défaut ISO-8859-1)

Délimiteurs

- Par défaut les données sont séparées par des espaces, tabulations, passages à la ligne,… (ce qui renvoie **`true`** avec la méthode **`Character.isWhitespace`**)
- On peut en indiquer d'autres avec la méthode **`useDelimiter`** qui prend en paramètre une expression régulière :
```
sc.useDelimiter("\\s*truc\\s*");
```


Localisation

- Une instance de **`Scanner`** utilise la locale par défaut
- On peut modifier la locale avec la méthode **`useLocale`**

Méthodes nextXXX

- On trouve une méthode next par type primitif : **`nextInt`**, **`nextDouble`**,…
- et aussi **`nextBigDecimal`**, **`nextBigInteger`**
- Pour les entiers, on peut passer en paramètre la base pour les interpréter (10 par défaut)
- Ces méthodes renvoient une exception **`java.util.InputMismatchException`** (non contrôlée) si le type de l'élément suivant ne correspond pas au type attendu

Méthodes nextXXX

- **`next()`** retourne une **`String`** qui contient la prochaine donnée placée entre 2 délimiteurs
- **`nextLine()`** avance jusqu'à la prochaine ligne et retourne ce qui a été sauté

Méthodes hasNextXXX

- Une méthode hasNextXXX par méthode nextXXX : **`hasNextInt`**, **`hasNextDouble`** (sauf pour **`nextLine`**)
- Retourne **`true`** si la prochaine donnée correspond au type que l'on cherche
- Ces méthodes nextXXX et hasNextXXX peuvent bloquer en attente du flot d'entrée

Autres méthodes

- **`findInline`** prend une expression régulière en paramètre et renvoie la prochaine occurrence dans la ligne de cette expression régulière (ne tient pas compte des délimiteurs)
- **`findWithinHorizon`** fait de même mais dans les n prochains caractères (n passé en paramètre) ; 0 correspond à un horizon à l'infini

Exemple

```
Scanner s = new Scanner(ligne);
s.useDelimiter("\\s*;\\s*");
nom = s.next();
age = s.nextInt();
if (s.hasNextInt()) {
  option = s.nextInt();
}
```


`IOException`

- Il n'est pas obligé de gérer les **`IOException`** car les méthodes de **`Scanner`** ne lèvent pas cette exception
- Si une **`IOException`** survient dans le flot sous-jacent, le scanner suppose qu'il a atteint la fin du flot
- On peut retrouver l'exception avec la méthode **`ioException()`** de la classe **`Scanner`**

Imprimer

Interfaces

- Le paquetage **`java.awt.print`** contient 2 interfaces pour représenter ce que l'on veut imprimer :
  - **`Printable`** correspond à un objet qui peut s'imprimer d'une façon simple
  - **`Pageable`** qui correspond à des impressions plus complexes, avec plusieurs formats d'impression (qui correspondent à plusieurs **`Printable`**)

Tâche d'impression

- On commence par obtenir une tâche d'impression de la classe **`PrinterJob`** (du paquetage **`java.awt.print`**) avec la méthode **`public static PrinterJob getPrinterJob()`**
- On passe ensuite à cette tâche un objet qui représente ce que l'on veut imprimer :
  - soit une instance de **`Printable`** (avec **`setPrintable`**)
  - soit une instance de **`Pageable`** (avec **`setPageable`**)

Interface **`Printable`**

- Il comporte une seule méthode

```
public int print(
    Graphics graphics,
    PageFormat pageFormat,
    int pageIndex)
throws PrinterException
```

- Et les 2 constantes (de type **`int`**) **`PAGE_EXISTS`** et **`NO_SUCH_PAGE`** qui peuvent être renvoyées par la méthode **`print`**

Méthode **`print`**

- Paramètres :
  - contexte graphique pour rendre la page à imprimer (peut être casté en **`Graphics2D`**)
  - format de la page d'impression qui décrit la taille et l'orientation de la page
  - numéro de la page à imprimer (0 pour la 1ère page)
- On remarque que, s'il y a plusieurs pages, on doit pouvoir les écrire dans n'importe quel ordre en donnant leur numéro, ce qui n'est pas toujours facile

Exemple d'objet imprimable

```
class HelloImprimable implements Printable {
  public int print(Graphics g,
                   PageFormat pf,
                   int pageIndex)
      throws PrinterException {
    if (pageIndex != 0) return NO_SUCH_PAGE;
    Graphics2D g2 = (Graphics2 D)g;
    g2.setFont(new Font("serif",
                        Font.PLAIN, 12));
    g2.drawString("Hello World", 100, 100);
    return PAGE_EXISTS;
  }
}
```


Zone imprimable

- Quand on écrit la méthode **`print`**, il faut souvent translater le **`Graphics`** pour tenir compte de la zone imprimable de la page :

```
Graphics2D g2d = (Graphics2D)g;
g2d.translate(pf.getImageableX(),
              pf.getImageableY());
```

Exemple de code pour imprimer

```
PrinterJob tache = PrinterJob.getPrinterJob();
tache.setPrintable(objetAImprimer);
try {
  tache.print();
}
catch(PrinterException e) { . . . }
```

doit implémenter l'interface **Printable**

Choix de l'imprimante

- La tâche d'impression imprime sur l'imprimante par défaut
- On peut faire afficher une fenêtre de dialogue qui permet à l'utilisateur d'indiquer une autre imprimante et d'autres informations comme les pages à imprimer et le nombre de copies :

```
if (tache.printDialog())
  tache.print();
```

renvoie false si l'utilisateur a annulé

Choix du format de page

- La tâche d'impression peut donner le format de page par défaut :
  `PageFormat pf = tache.defaultPage();`
- On peut modifier le format de page par défaut
  - par programmation :
    `pf.setOrientation(PageFormat.LANDSCAPE);`
  - en affichant une fenêtre de dialogue :
    `pf = tache.pageDialog(pf);`
- Pour utiliser un format de page, il suffit de le passer à la méthode **`setPrintable`** :
  `tache.setPrintable(ObjetAImprimer, pf);`

Exemple de code

```
PrinterJob tache =
  PrinterJob.getPrinterJob();
tache.setPrintable(objetAImprimer);
PageFormat pf =
  tache.pageDialog(printJob.defaultPage());
if (tache.printDialog()) {
  try {
    tache.print();
  }
  catch(PrinterException e) { . . . }
}
```

Seulement si on veut changer les dimensions de la page

Interface **`Pageable`**

- Elle possède 3 méthodes :
  - **`int getNumberOfPages()`**
  - **`Printable getPrintable(int numeroPage)`**
  - **`PageFormat getPageFormat(int numeroPage)`**

Classe **`Book`**

- Elle implémente l'interface **`Pageable`**
- Outre les méthodes de **`Pageable`**, elle contient 3 méthodes pour ajouter ou remplacer des pages :
  - **`void append(Printable p, PageFormat pf)`** : ajoute une page (à la fin)
  - **`void append(Printable p , PageFormat pf, int n)`** : ajoute n pages (à la fin)
  - **`void setPage(int n, Printable p, PageFormat pf)`** : remplace la page numéro n

Exemple schématique avec **Book**

```
Book livre = new Book();
// Ajoute 1 page en paysage et 10 en portrait
PageFormat paysage = tache.defaultPage();
paysage.setOrientation(PageFormat.LANDSCAPE);
livre.append(new Page1(), paysage);
PageFormat portrait = tache.defaultPage();
paysage.setOrientation(PageFormat.PORTRAIT);
livre.append(new Page2(), portrait, 10);
//
tache.setPageable(livre);
```


Conclusion

- L'API standard n'offre que des fonctionnalités de bas niveau
- Le programmeur doit donc ajouter beaucoup de code s'il veut imprimer un document de plusieurs pages, avec des formats de pages non simplistes

APIs non standard

- Des APIs non standard sont disponibles sur le Web pour faciliter l'impression de rapports complexes, directement ou en passant par l'exportation en formats divers (le plus souvent en PDF)
- Ces APIs sont fournies par des projets open source ou par des produits commerciaux
- Quelques exemples sont donnés dans les transparents suivants

APIs open source et gratuites

- *JasperReports* permet de générer des rapports sophistiqués de formats divers (PDF, HTML, XML, RTF,…) en prenant les données dans divers sources (base de données relationnelle, XML,…) ; *iReport* est un outil wysiwyg pour générer des fichiers de description de rapports utilisés ensuite par *JasperReports*

APIs open source et gratuites

- *iText* est une API qui permet de générer des documents PDF ou RTF (Word)
- *Apache POI* permet de manipuler en lecture et écriture les fichiers associés à la suite Office de Microsoft (fichiers Word ou Excel en particulier)

Autres APIs

- Il est aussi possible d'utiliser XSL-FO (du monde XML)
- *Crystal Reports* est un produit commercial payant (et cher !)
- *JPedal* est un produit commercial qui offre une version gratuite pour les organisations à but non lucratif
- Différentes librairies open source à l'adresse http://java-source.net/open-source/pdf-libraries

Isoler les entrées-sorties

Pourquoi ?

- Les entrées-sorties ont souvent des API complexes
- Elles compliquent les tests
- De plus, il n'est pas rare de changer de façon d'accéder à des ressources ou données externes (fichiers, BD relationnelle, BD objet, XML,…)
- Il vaut donc mieux isoler les entrées-sorties du reste de l'application

Implémentation

- Il est bon d'encapsuler les accès aux données dans des classes à part de telle sorte que
  - le reste de l'application puisse accéder aux ressources plus facilement : interfaces simples et adaptées à l'application
  - les interfaces ne dépendent pas du type de persistance choisi
  - on puisse simuler des entrées-sorties pendant les tests (sans nécessairement disposer du support de persistance)