UNIVERSITE DE LAUSANNE

ALGORITHMIQUE

2004 - 2005

Résumé du langage

François Grize

avec la collaboration de Marc Mercier

TABLES DES MATIERES

1 PROGRAMMATION ORIENTEE OBJETS 1

1.1 Définitions de base 1

2 SYNTAXE 3

2.1 Vocabulaire 3
a) Les mots réservés 3
b) Quelques «symboles de ponctuation» 4
2.2 Grammaire 5
a) Schémas syntaxiques 5
b) Diagrammes syntaxiques 5
2.3 Unités syntaxiques 6
a) Identificateurs 6
b) Littéraux numériques 7
c) Littéraux caractères et chaînes 8
d) Commentaires 8

3 TYPES NUMERIQUES ET EXPRESSIONS ARITHMETIQUES 9

3.1 Types numériques 9
a) Opérateurs unaires 9
b) Opérateurs binaires 10

4 CONDITIONS ET EXPRESSIONS BOOLEENNES 11

4.1 Type boolean (booléen) 11
4.2 Tables de vérité 11
a) Opérateurs unaires 11
b) Opérateurs binaires 11
c) Lois de Morgan 12
4.3 Opérateurs booléens 13
4.4 Opérateurs de comparaison 14
4.5 Opérateur conditionnel 14

5 INSTRUCTIONS D'AFFECTATION ET INSTRUCTION COMPOSEE 15

5.1 Instructions d'affectation 15
5.2 Instruction composée et bloc 15
a) Instruction composée 15
b) Bloc 15

6 INSTRUCTIONS CONDITIONNELLES 16

6.1 Instruction conditionnelle simple 16
6.2 Instruction conditionnelle avec alternative 16
6.3 Aiguillage 17

7 INSTRUCTIONS DE REPETITION **18**

7.1 Instruction while (tant que) 18
7.2 Instruction do ... while (faire ... tant que) 18
7.3 Instruction for (pour) 18
7.4 Instruction break (rupture) 19

8 CLASSE **20**

8.1 Variable 20
8.2 Constructeur 21
8.3 Méthode 22
8.4 Signature 23
8.5 Le concept d'héritage 24
8.6 Contrôle d'accès 26
8.7 Variables et méthodes de classe 26

9 PACKAGE **28**

9.1 Définition. Organisation 28
9.2 Modificateur d'accès 28

10 TABLEAUX **30**

10.1 Création 30
10.2 Accès 31
10.3 Parcours 31

11 EXCEPTIONS **33**

11.1 Définition 33
11.2 Exceptions standard non contrôlées (unchecked) 33
11.3 Exceptions contrôlées (checked) 35
11.4 Remarques sur la syntaxe 35
11.5 Exceptions levées par l'utilisateur 36

12 FICHIERS **38**

12.1 Définition 38
12.2 Fichiers à accès séquentiel 38
12.3 Vue partielle de la hiérarchie des classes du paquetage java.io 39
12.4 Flux standard 40
12.5 Fichiers physiques 40
12.6 Fichiers à accès direct 42

ANNEXE 1: Principaux composants graphiques **44**

ANNEXE 2: La classe String **48**

ANNEXE 3: Quelques caractères spéciaux **50**

ANNEXE 4: Hiérarchie partielle des classes dans AWT **51**

ANNEXE 5: Quelques classes en Java **52**

Bibliographie **57**

JAVA
RESUME DU LANGAGE

1. PROGRAMMATION ORIENTEE OBJETS

1.1 Définitions de base

Classe

Concept qui permet d'*encapsuler* des informations et des comportements communs à un ensemble d'objets. Fabrique d'objets.

Objet

A partir d'une classe, on peut fabriquer autant d'objets qu'on veut. On parle aussi d'*instance* de classe.

Attribut

Ce sont les informations de la classe.

Méthode

Ce sont les comportements de la classe.

Message

Pour activer une méthode, on envoie un message à un objet.

Exemple 1-1: déclaration d'une classe

```
class Personne {
    String prenom;
    String nom;
    // méthode
    String presenteToi( ) {
       return prenom + ” ” + nom;
    }
}

Personne moi;
// instanciation de l'objet moi
moi = new Personne();
moi.prenom = ”Jules”;
moi.nom = ”Dupont”;
```

Remarques:

1. une méthode comporte toujours deux parties :
   - une *entête* (`String presenteToi( )`), qui précise son mode d'emploi;
   - un *corps* (`{...}`), qui contient les instructions à exécuter;
2. l'opérateur + permet la concaténation de chaînes;
3. l'opérateur **`new`** permet d'instancier une classe, c'est-à-dire de créer un objet;
4. `moi.presenteToi()` retourne la chaîne `"Jules Dupont"`.

2. SYNTAXE

2.1 Vocabulaire

a) Les mots réservés

Les mots suivants sont réservés à une utilisation bien précise dans le langage Java. Ils ne peuvent pas être utilisés comme noms de classes, de méthodes ou de variables:

abstract	abstrait	voir 9.2
boolean	booléen	voir 4; 6; 7
break	rupture	voir 6.3; 7.4
byte	octet	voir 3.1
case	au cas où	voir 6.3
catch	capturer	voir 11.1
char	caractère	voir 2.3.c
class	classe	voir 1.1; 2.2; 2.3; 3.1; 8; 9.2
const	constante	non utilisé
continue	continuer	
default	défaut	voir 6.3
do	faire	voir 7.2; 7.4
double	double	voir 2.3.b; 3.1
else	sinon	voir 6.2
extends	étendre	voir 8.5
false *	faux	voir 4
final	final	voir 8.7; 9.2
finally	finalement	voir 11.4
float	flottant	voir 2.3.b; 3.1
for	pour	voir 7.3; 7.4
goto	aller à	non utilisé
if	si	voir 6
implements	implémente	
import	importer	voir 9.1
instanceof	instance de	
int	entier	voir 3.1
interface	interface	voir 2.2
long	long	voir 2.3; 3.1
native	natif	
new	nouveau	voir 1.1; 10.1

null *	nul	voir 8.2
package	paquetage	voir 9
private	privé	voir 8.6
protected	protégé	voir 8.6
public	public	voir 8.6; 9.2
return	retourner	voir 1.1; 8.3
short	court	voir 3.1
static	statique	voir 8.7
super	super	voir 8.5
switch	aiguillage	voir 6.3; 7.4
synchronized	synchronisé	
this	celui-ci	voir 8.2
throw	lever une exception	voir 11.5
throws	lève	voir 11.3
transcient		
true *	vrai	voir 6
try	essayer	voir 11.1
void	vide	voir 8.3
volatile	volatile	
while	tant que	voir 7.1; 7.3; 7.4

* il ne s'agit pas vraiment de mots réservés, mais ils se comportent comme tels.

b) Quelques «symboles de ponctuation»

1) séparateurs: `()` `{}` `[]` `;` `,` `.`

2) opérateurs:

```
+     -     *     /     ...
=     <     >           ...
&&    ||    +=    -=    ...
```

2.2 Grammaire

a) Schémas syntaxiques

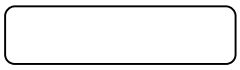
Les éléments de vocabulaire sont entre guillemets.

`{...}`	0 ou plusieurs fois
`[...]`	option
`|`	choix

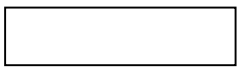
Exemple 2-1: schéma syntaxique de type_declaration

```
type_declaration

  [doc_comment] {class_declaration|interface_declaration} ";"
```

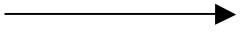
b) Diagrammes syntaxiques



un élément du vocabulaire



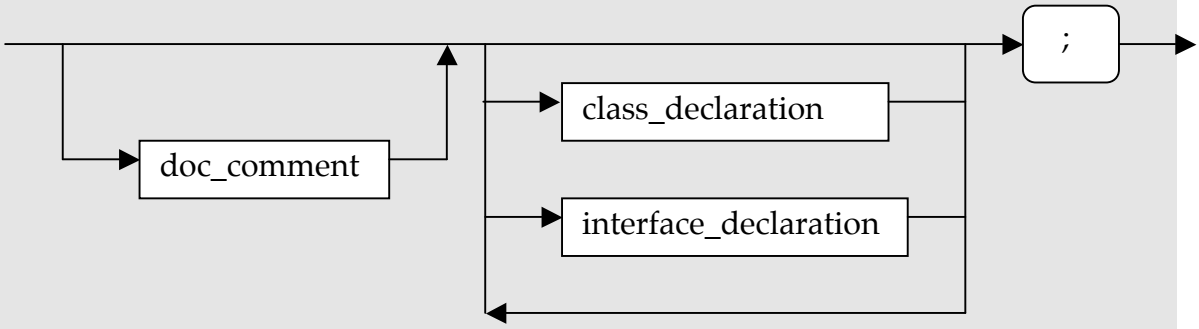
une entrée vers une règle de grammaire



comment suivre la règle

Exemple 2-2: diagramme syntaxique de type_declaration

type_declaration



On trouve tous les schémas syntaxiques et les diagrammes syntaxiques de Java à l'adresse suivante:

http://cui.unige.ch/java/

De là, il faut cliquer sur le lien syntaxe.

2.3 Unités syntaxiques

a) Identificateurs

Une suite de lettres et de chiffres.

Les lettres comprennent les caractères habituels, y compris les lettres accentuées, plus les caractères `$` (dollar) et `_` (souligné).

Le premier caractère doit être une lettre.

Contrairement à certains langages de programmation, on distingue les majuscules des minuscules.

Par convention:

- les identificateurs de classe commencent par une majuscule, les autres par une minuscule;
- pour séparer les mots, on utilise une majuscule.

Exemple 2-3: les identificateurs

```
ceciEstUnIdentificateur
String
boeing707
MAX_VALUE
çàEtLà
àCôté
αγετη
```

b) Littéraux numériques

Les *entiers* sont constitués d'une suite de chiffres, éventuellement précédée par une marque de base (`0` pour octal et `0x` ou `0X` pour héxadécimal), éventuellement suivie du suffixe `L` ou `l` (pour le type **`long`**).

Exemple 2-4: les entiers

```
2514
0xDadaCafe
1998
0777L
214748641
```

Les *flottants* sont constitués des parties suivantes:

- une partie entière;
- un point décimal;
- une partie fractionnaire;
- éventuellement un exposant ;
- éventuellement les suffixes `F` ou `f` (pour le type **`float`**) et `D` ou `d` (pour le type **`double`**).

  Si le suffixe est absent, le nombre est de type **`double`**.

L'exposant est indiqué par la lettre `E` ou `e` suivi d'un entier avec ou sans signe.

Exemple 2-5: les flottants

```
3.14159
2.0f
6.02213e+23
1.0E-9d
1e10
```

c) Littéraux caractères et chaînes

- un littéral caractère. C'est un caractère (éventuellement une séquence *escape*) entouré d'apostrophe: `'a'  'Ω';`
- un littéral chaîne. C'est zéro ou plusieurs caractères entourés de guillemets:

  `" ", "ceci est une chaîne", ""` (chaîne vide).

d) Commentaires

- une ligne de commentaire. Le texte qui commence par `//` est ignoré jusqu'à la fin de la ligne;
- un texte qui commence par `/*` et qui se termine par `*/`.

3. TYPES NUMERIQUES ET EXPRESSIONS ARITHMETIQUES

3.1 Types numériques

1. Les types entiers sont:
   - **`byte`**: -128 à 127
   - **`short:`** -32768 à 32767
   - **`int`**: -2147483648 à 2147483647
   - **`long`**: environ -10^{18} à 10^{18}

2. Les types réels sont:
   - **`float`**: représentés sur 32 bits
   - **`double`**: représentés sur 64 bits

3. Conversion de types:

   `(` `nom de type` `)` `expression`

   convertit `expression` dans le type précisé par `nom de type`.

   Exemple: `(int) 4.7` retourne l'entier `4.`

Les constantes (littéraux numériques) de ces types sont définies en 2.3.b.

a) Opérateurs unaires

`+`	identité
`-`	inversion de signe
`++`	incrémentation préfixée et postfixée
`--`	décrémentation préfixée et postfixée

Exemple 3-1: opérateurs unaires

`++v`	est équivalent à `v = v+1`
`--v`	est équivalent à `v = v-1`
`v++`	retourne `v` puis l'incrémente
`v--`	retourne `v` puis le décrémente

b) Opérateurs binaires

`+`	addition
`-`	soustraction
`*`	multiplication
`/`	division
`%`	reste de la division entière; opérateur modulo.

Remarques:

1. si le membre de gauche et le membre de droite de l'opérateur `/` sont tous deux de type entier, c'est la division entière qui s'applique.

   Exemple: `7 / 2` retourne `3`;

2. lorsqu'il y a mélange d'opérandes entiers et réels, les opérations se font en réel et le résultat est réel;
3. les multiplications et les divisions sont traitées avant les additions et les soustractions. En cas d'égalité de priorité, l'évaluation se fait de gauche à droite. Les parenthèses permettent de changer les priorités;
4. la classe `java.lang.Math` (cf. Annexe 5) contient les principales fonctions arithmétiques standard.

4. CONDITIONS ET EXPRESSIONS BOOLEENNES

Une expression booléenne (en référence au mathématicien britannique George Boole) peut prendre soit la valeur faux (**`false`**) soit la valeur vrai (**`true`**).

4.1 Type boolean (booléen)

Les variables booléennes ne peuvent prendre que deux valeurs: faux (**`false`**) ou vrai (**`true`**).

4.2 Tables de vérité

a) Opérateurs unaires

Soit p une variable de proposition qui prend soit la valeur 0 (en Java: **`false`**), si la proposition est fausse, soit la valeur 1 (en Java: **`true`**), si la proposition est vraie.

On a alors $2^2 = 4$ opérateurs unaires:

- identité (laisse tout comme c'est)
- transforme tout en 1
- transforme tout en 0
- *négation* notée ¬ ou ~ (non; en Java `!`)

p	¬p
1	0
0	1

b) Opérateurs binaires

Soit p et q deux variables de proposition qui prennent soit la valeur 0 (en Java: **`false`**), si la proposition est fausse, soit la valeur 1 (en Java: **`true`**), si la proposition est vraie.

On a alors $2^4 = 16$ opérateurs binaires:

- *conjonction* notée Λ (et; en Java `&`).

p	Λ	q
1	1	1
1	0	0
0	0	1
0	0	0

- *disjonction* notée V (ou; en Java `|`).

p	V	q
1	1	1
1	1	0
0	1	1
0	0	0

- *disjonction exclusive* notée W (ou exclusif; en Java Λ).

p	W	q
1	0	1
1	1	0
0	1	1
0	0	0

Exemple 4-1: évaluation d'expression booléenne

Si `p = false`, `q = true`, `r = true`

`(p V q) Λ r` vaut **`true`**

c) Lois de Morgan

1. `¬ (p Λ q)` est équivalent à `¬ p V ¬ q`
2. `¬ (p V q)` est équivalent à `¬ p Λ ¬ q`

Démonstration 1:

¬	(p	Λ	q)
0	1	1	1
1	1	0	0
1	0	0	1
1	0	0	0

¬ p	V	¬q
0	**0**	0
0	**1**	1
1	**1**	0
1	**1**	1

Démonstration 2: exercice !

4.3 Opérateurs booléens

Opération	**Opérateur Java**
négation (**¬**)	!
conjonction (Λ)	`&`
disjonction (V)	`|`
disjonction exclusive (W)	**Λ**
conjonction conditionnelle	`&&`
disjonction conditionnelle	`||`

Remarques:

- `p && q`: `q` n'est pas évalué si `p` est faux;
- `p || q`: `q` n'est pas évalué si `p` est vrai.

4.4 Opérateurs de comparaison

Opération	Signification	Opérateur Java
égalité	=	==
différence	≠	!=
inférieur à	<	<
inférieur ou égal à	≤	<=
supérieur à	>	>
supérieur ou égal à	≥	>=

4.5 Opérateur conditionnel

`condition` ? `expression-1` : `expression-2`

retourne `expression-1` si `condition` est vrai

sinon retourne `expression-2`

Exemple 4-2: opérateur conditionnel

```
int n;
String s;
...
s = ”soit ” + n + ” franc” + (n > 1 ? ”s” : ””);
```

5. INSTRUCTIONS D'AFFECTATION ET INSTRUCTION COMPOSEE

5.1 Instructions d'affectation

Deux formes possibles :

1. `variable` = `expression` ;

   affecte `expression` à `variable` .

2. `variable` `opérateur binaire` = `expression` ;

   est équivalent à:

   `variable` = `variable` `opérateur binaire` `expression` ;

Remarque:

`opérateur binaire` vaut:

- un des cinq opérateurs binaires arithmétiques (cf. 3.1.b);
- un des opérateurs binaires booléens: `&` `|` Λ ;
- certains opérateurs de décalage.

5.2 Instruction composée et bloc

a) Instruction composée

Permet de regrouper plusieurs instructions entre `{` et `}`, là où la syntaxe impose une seule instruction.

b) Bloc

Une instruction composée qui peut comprendre des déclarations de variables locales.

6. INSTRUCTIONS CONDITIONNELLES

6.1 Instruction conditionnelle simple

if (condition) instruction ;

condition est une expression booléenne (cf. 4).

Si condition est vraie alors instruction est exécutée.

6.2 Instruction conditionnelle avec alternative

if (condition) instructionSiVrai ; **else** instructionSiFaux ;

condition est une expression booléenne (cf. 4).

Si condition est vraie alors instructionSiVrai est exécutée

sinon instructionSiFaux est exécutée.

Remarque:

Lorsqu'on a des instructions conditionnelles imbriquées, le **else** se rapporte toujours au dernier **if**.

Exemple 6-1: if - else

```
i=0;  j=3;
if (i==0)
    if (j==2) i=4;
else i=j;
```

- i vaut 3;
- afin d'éviter la confusion, le **else** devrait être aligné sous le dernier **if**!

6.3 Aiguillage

switch

```
( expression ) {
    case constante-1 : suiteInstructions-1 ;

    case constante-2 : suiteInstructions-2 ;
    ...
    default: suiteInstructions ;
}
```

Remarques:

1. `expression` doit être de type caractère ou entier;
2. `constante-n` est une expression constante; elle peut contenir des opérateurs, mais doit pouvoir être évaluée à la compilation;
3. les constantes doivent être mutuellement exclusives;
4. si `expression` ne correspond à aucun cas prévu, l'instruction **`switch`** est ignorée, sauf s'il y a une partie **`default`**;
5. la partie **`default`** est évidemment optionnelle.

Attention:

Après l'exécution d'un branchement, on exécute le branchement suivant, sauf si on rencontre l'instruction **`break`**.

Exemple 6-2: aiguillage

```
int i, m;
...
switch (i) {
  case 1 : m=1;
  case 2 : m=2;
  break;
  default : m=3;
}
```

si `i` vaut `0` alors `m` vaut `3`
si `i` vaut `1` alors `m` vaut `2`
si `i` vaut `2` alors `m` vaut `2`

7. INSTRUCTIONS DE REPETITION

7.1 Instruction while (tant que)

```
while ( condition ) instruction ;
```

`condition` est une expression booléenne.

Tant que `condition` est vraie, `instruction` est répétée.

7.2 Instruction do ... while (faire ... tant que)

```
do instruction while ( condition );
```

`condition` est une expression booléenne.

Exécute `instruction` tant que `condition` est vraie.

7.3 Instruction for (pour)

```
for ( initialisation ; condition ; miseAjour ) instruction ;
```

`initialisation` est une instruction d'affectation.

`condition` est une expression booléenne.

`miseAjour` est une expression.

Cette instruction est équivalente à:

```
initialisation ;
while ( condition ) {
 instruction ;
 miseAjour ;
}
```

Remarque:

dans la partie `initialisation`, il est possible de déclarer la variable de contrôle.

Exemple 7-1: instruction for

```
int somme = 0;
...
for (int i = 1;  i <= 10;  i++)
  somme  += i;
```

calcule la somme des 10 premiers entiers.

7.4 Instruction break (rupture)

L'instruction **`break`** peut apparaître dans une instruction **`switch`** (cf. 6.3) ou dans n'importe quelle instruction de répétition (**`while`**, **`do ... while`**, **`for`**). Elle permet de sortir de la boucle dans laquelle elle se situe.

Exemple 7-2: instruction break

```
/* recherche le premier nombre premier
dans l'intervalle min .. max  */

int  min,  max;
int  i; // pour parcourir l'intervalle
...
for (i = min;  i <= max;  i++)
  if (estPremier(i)) break;
```

8. CLASSE

```
class  SonNom {

     variables

     constructeur

     méthodes
}
```

Remarques:

1. chacune des trois parties du corps de la classe peut être absente;
2. on fait la distinction entre variable ou méthode d'instance et variable ou méthode de classe (cf. 8.7).

8.1 Variable

Une variable (ou attribut) permet de préciser un trait commun à tous les objets de la classe.

Exemple 8-1: variable

```
class Personne {
  String prenom, nom;
  int age;
  ...
}
```

Remarques:

1. toute personne a un prénom, un nom et un âge;
2. dans ce cas, on éviterait, par contre, de mettre un attribut salaire, car toute personne n'est pas nécessairement salariée.

8.2 Constructeur

Un constructeur permet d'initialiser les variables d'un objet lors de sa création (**`new`**).
Un constructeur a le même nom que la classe qui le contient.
Une classe peut contenir plusieurs constructeurs pourvu qu'ils se distinguent par le nombre ou le type de leurs paramètres (cf. 8.4).
Quand une classe ne prévoit aucun constructeur, Java fournit un constructeur par défaut, sans paramètre, qui initialise les variables d'instances avec des valeurs par défaut conformément à leur type. A savoir:

- pour les entiers: 0
- pour les réels: 0.0
- pour les booléens: **`false`**
- pour les caractères: *nul code* (le caractère dont la valeur vaut 0)
- pour les références à des objets: **`null`**

Exemple 8-2: constructeur

```
class Personne {
  String prenom;
  String nom;
  int age

  // constructeurs
  Personne (String p, String n) {
    prenom = p;
    nom = n;
    age = -1;   //-1 si l'âge est inconnu
  }
  Personne (String p, String n, int a) {
    prenom = p;
    nom = n;
    age = a;
  }
 }
 ...

Personne elle = new Personne ("Julie","Durand");
Personne lui = new Personne ("Alain","Térieur",24);
```

Remarque:

dans le deuxième constructeur (celui qui a l'âge en paramètre), on aimerait pouvoir utiliser le premier; on peut le faire grâce au mot réservé **`this`**.

Exemple 8.3: utilisation de this

```
Personne (String p, String n, int a){
  this (p,n);
  age = a;
}
```

8.3 Méthode

Une méthode permet d'opérer sur un objet.

```
type unNom ( paramètres ) {

  instructions

}
```

Remarques:

1. le `type` retourné par la méthode. Si elle ne retourne rien, on utilise le mot réservé **`void`**;
2. la valeur de la méthode est retournée par le mot réservé **`return`**, qui a aussi pour effet de rendre le contrôle à l'objet appelant;
3. `paramètres` peut être absent.

Exemple 8.4: méthode

```
// méthode de la classe Personne
// cf. Exemples 8-1 et 8-2

String presenteToi( ) {
  String s;
  s = "Je m'appelle " + prenom + " " + nom;
  if (age >= 0) return s + " et j'ai " + age + " ans";
  else return s;
}

...
```

`elle.presenteToi( )` retourne:

```
Je m'appelle Julie Durand
```

`lui.presenteToi( )` retourne:

```
Je m'appelle Alain Térieur et j'ai 24 ans
```

8.4 Signature

Il s'agit du nom (d'un constructeur ou d'une méthode) ainsi que du nombre et du type de ses paramètres formels.

Une classe ne peut pas déclarer deux constructeurs ou deux méthodes ayant la même signature.

Exemple 8-5: deux méthodes de même signature

```
final void move(int dx, int dy)
      ...

void move(int x, int y)
      ...
```

Par contre, il est possible, dans une classe, de donner le même nom à deux constructeurs différents ou à deux méthodes différentes, pourvu que leur signature soit différente.

Exemple 8-6: méthodes de même nom, signatures différentes

```
System.out.println (512);
System.out.println ("toto");
```

8.5 Le concept d'héritage

Une classe est toujours définie à partir d'une autre classe: sa *superclasse*. On parle aussi de *classe mère* et de *classe fille* (ou *sous-classe*). Une classe qui est définie sans préciser sa superclasse est implicitement une classe fille de `Object`. L'ensemble de toutes les classes Java (définies de manière standard ou par le programmeur) forme une *arborescence* dont la racine unique est `Object`.

```
class Fille extends SuperClasse {...}
```

La sous-classe `Fille` *hérite* automatiquement toutes les propriétés de `SuperClasse` à laquelle on n'ajoutera que les propriétés qui lui sont particulières.

Une sous-classe peut donc utiliser toutes les variables, constructeurs ou méthodes de sa superclasse et ceci transitivement jusqu'à la racine.

Une sous-classe peut déclarer un identificateur qui a le même nom qu'un identificateur déjà déclaré dans sa superclasse. Il n'y a pas d'ambiguïté possible, car on se réfère toujours au dernier identificateur déclaré.

Pour désigner explicitement un identificateur de la superclasse, en particulier un de ses constructeurs, on utilise le mot réservé **`super`**.

Exemple 8.7: identificateurs synonymes

```
class M extends ... {
    int a;
    ...
}

class F extends M {
    String a;
    ...
    a = "Toto";
    super.a = 12;
    ...
}
```

Exemple 8-8: héritage

```
class Salarie extends Personne {
    double salaire;

    // constructeur
    Salarie (String p, String n, double s) {
        super (p,n);
        salaire = s;
    }
}
```

Remarques:

1. le mot réservé **`super`**, s'il n'est pas suivi d'un identificateur, désigne un constructeur qui doit être recherché dans la classe mère;
2. si une classe fille ne prévoit aucun constructeur ou si elle ne fait pas explicitement appel à un constructeur de la classe mère, Java appelle le constructeur par défaut de la classe mère, c'est-à-dire le constructeur sans paramètre (ici: `Personne( )`, qui n'a pas été prévu dans l'exemple 8-2).

8.6 Contrôle d'accès

Les entités (variables, constructeurs, méthodes) déclarées dans une classe peuvent être:

- **`public`**
  dans ce cas, elles sont accessibles autant à l'intérieur qu'à l'extérieur de la classe;
- **`private`**
  dans ce cas, elles ne sont accessibles qu'à l'intérieur de la classe;
- **`protected`**
  dans ce cas, elles ne sont accessibles qu'à l'intérieur de la classe et de l'ensemble de ses sous-classes.

Remarques:

1. par défaut, les entités sont déclarées comme **`public`**;
2. par soucis de lisibilité, on utilisera quand même explicitement le mot réservé **`public`**;
3. afin de ne pas devoir entrer dans les détails de l'implémentation, l'utilisateur d'une classe ne devrait pas pouvoir manipuler directement ses attributs si ce n'est à l'aide de méthodes prévues à cet effet (penser aux méthodes `get...( )` et `set...( )`).

8.7 Variables et méthodes de classe

Dans l'exemple 8-2, on a utilisé `-1` pour préciser qu'un âge était inconnu. Si on change cette convention, on devra modifier toutes les occurrences éventuelles de `-1`. Il vaut donc mieux donner un nom à la valeur conventionnelle (par exemple: `ageInconnu`).

Exemple 8-9: convention

```
class Personne {
    private int ageInconnu = -1;
    private String prenom;
    private String nom;
    private int age;
    ...
```

Toute méthode de la classe peut modifier `ageInconnu`, ce qui n'a pas de sens. A l'évidence, il s'agit d'une constante. On peut le préciser à l'aide du mot-clé **`final`**. D'où la déclaration:

```
private final int ageInconnu = -1;
```

ce qui empêche toute modification de `ageInconnu`.

Toute instance de la classe `Personne` comprendra la constante `ageInconnu`, ce qui est parfaitement inutile. En effet, la constante n'est pas liée à un objet particulier, mais elle est spécifique à la classe elle-même. On peut le préciser à l'aide du mot-clé **`static`**. D'où la déclaration:

```
private static final int ageInconnu = -1;
```

Par analogie, une classe doit pouvoir définir des méthodes (fonctions) qui n'agissent pas sur une de ses instances. Là encore, on peut le préciser à l'aide du même mot-clé **`static`**.

Définitions:

On fera la distinction entre *variables de classe* et *variables d'instance* et, respectivement, *méthodes de classe* et *méthodes d'instance*. Les variables et les méthodes de classes sont introduites à l'aide du mot-clé **`static`**.

Remarque:

Evocation d'une méthode:

- d'instance: `nomObject.methode(...)`
- de classe: `nomClasse.methode(...)`

Exemple (méthode de classe): `Math.cos(...)`

Exemple 8-10: méthode de classe vs méthode d'instance

```
// Méthode d'instance
public void add(Fraction f){
  // Ajoute la valeur de f à la Fraction sur laquelle est
  // appelée la méthode
}

// Méthode de classe
public static Fraction sum(Fraction f, Fraction g){
  // Retourne un nouvel objet Fraction dont la valeur est la
  // somme des valeurs de f et g.
}
```

Si `f1` et `f2` sont deux objets de type `Fraction`, on pourra utiliser ces deux méthodes de la manière suivante:

```
f1.add(f2);
Fraction f3 = Fraction.sum(f1, f2);
```

9. PACKAGE

9.1 Définition. Organisation

Le concept de *package* permet d'organiser les classes en les regroupant dans des ensembles fonctionnels.

La librairie standard se compose de plusieurs packages. Par exemple, le package `lang` qui contient, entre autres, les classes `Object, String, Integer, System, Math,` ...

Chaque package est contenu dans un répertoire. Les packages de la librairie Java sont situés dans le répertoire `java`. Attention de ne pas confondre la hiérarchie des classes (cf. 8.5) et l'arborescence des packages.

Chemin d'accès:

```
java.repertoire.sousRepertoire.packages
```

Seules les classes publiques de `java.lang` sont directement utilisables. Dans tous les autres cas, il faut employer le mot réservé **`import`** suivi du chemin d'accès à la classe particulière qu'on veut utiliser.

Exemple 9-1: package util

```
import java.util.Date;
// pour utiliser la classe Date du package util
import java.util.*;
// pour utiliser les classes publiques du package util
```

9.2 Modificateur d'accès

Une classe peut être précédée ou non du modificateur d'accès **`public`**.

- **`public class`** `X ...` `X` est accessible en dehors du package qui la contient;
- **`class`** `Y ...` `Y` n'est accessible qu'aux autres classes du package qui la contient.

Il existe encore deux autres modificateurs de classe:

- **`final`**: la classe précédée de ce mot-clé ne peut pas être sous-classée;
- **`abstract`**: la classe est incomplète (sera étudié plus tard).

Exemple 9-2: modificateurs de classe

```
public final class String extends ... {
        ...
}


public final class Integer extends ... {
        ...
}
```

10. TABLEAUX

10.1 Création

Un tableau est une collection de composants tous de même type qu'on accède grâce à un indice de type entier.

Exemple 10-1: tableaux avec spécification des bornes

```
int vecteur [ ] = new int [12];
  // un vecteur de 12 entiers
double matrice [ ] [ ] = new double [5] [5];
  // une matrice carrée de 25 réels
String page [ ] = new String [42];
  // une page de 42 lignes
char livre [ ] [ ] [ ] = new char [260] [35] [60];
  /* 260 pages contenant chacune 35 lignes de
     au plus 60 caractères */
```

Une autre manière de créer un tableau est de lui donner une valeur initiale.

Exemple 10-2: tableaux avec spécification de valeurs initiales

```
int v [ ] = {2,4,6}; // un vecteur de 3 entiers
double m [ ] [ ] = {{2.0,4.5},{-6.3,2.1}};
String texte [ ] = {"première ligne",
                    "deuxième ligne",
                    "troisième ligne"};
```

Remarque:
il existe plusieurs syntaxes pour définir un tableau. Les crochets peuvent suivre l'identificateur de la variable ou l'identificateur du type.

Exemple 10-3: différentes syntaxes pour déclarer un tableau

```
int v [ ] = ... ;
```

est équivalent à :

```
int [ ] v = ... ;
```

10.2 Accès

identificateur [ind 1] [ind 2] ... [ind n]

ind i expressions de type entier.

Attention:
les éléments sont numérotés à partir de 0 !

Exemple 10-4: accès aux éléments d'un tableau

```
// dans l'exemple 10-1
vecteur [4]     // 5ème élément de vecteur
matrice [i][j]  // élément situé à l'intersection de la
                // (i + 1)ème ligne et (j + 1)ème
                // colonne de matrice

// dans l'exemple 10-2
v [1]     // vaut 4
m [1][0]  // vaut -6.3
texte [3] // est indéfini !
```

10.3 Parcours

Pour parcourir les éléments d'un tableau, on utilise, en général, une ou plusieurs "instructions pour" (cf. 7.3).

Exemple 10-5: parcours de tous les éléments d'un tableau

```
// impression de tous les éléments
// d'un tableau à deux dimensions
for (int i=0; i<2; i++)
   for (int j=0; j<2; j++)
      System.out.println (m[i][j]);
```

Remarque:

L'attribut `length` permet de connaître la taille d'un tableau.

Exemple 10-6: parcours de tous les éléments d'un tableau

```
// parcours du tableau unTableau
for (int i=0; i<unTableau.length; i++)
   ...
```

11. EXCEPTIONS

11.1 Définition

Ce sont des objets (c'est-à-dire des instances de classe) spécialisés qui permettent de gérer les erreurs qui surviennent à l'exécution.

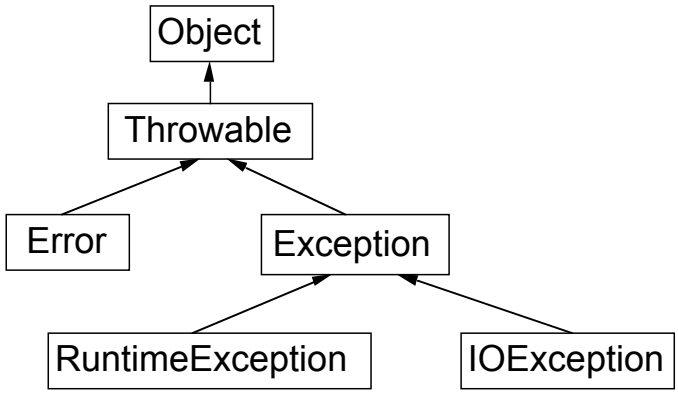
Certaines instructions peuvent *lever* une exception, c'est-à-dire fabriquer un objet (l'exception) qui est "exporté" et que l'on peut *capturer*.

Pour *traiter* des exceptions:

```
try { instructions qui peuvent lever des exceptions }

catch (UneException e) {

  /* faire ce qui convient avec l'exception capturée e
  */
  /* qui est une instance de la classe UneException
  */
  /* (ou d'une de ses sous-classes)
  */

}
```

Si une exception est levée dans le bloc **`try`**, l'exécution du programme est interrompue pour être transférée au bloc **`catch`**.

11.2 Exceptions standard non contrôlée (unchecked)



Les exceptions standard non contrôlées héritent d'une des deux classes:

1. `RuntimeException`
    - arithmétiques: `ArithmeticException`
    - argument illégal: `IllegalArgumentException`
    - indice illégal: `ArrayIndexOutOfBoundsException`
    - pointeur nul: `NullPointerException`
    - ...
2. `Error`
    - plus de mémoire: `OutOfMemoryError`
    - erreur de "hardware": `VirtualMachineError`
    - ...

Exemple 11-1: exception standard non contrôlée

```
int tab [ ] = new int [5];
int i, j;
/* si on tape autre chose qu'un nombre */
/* l'exception n'est pas capturée ==> DUMP */
i = Integer.parseInt(textField1.getText());

try {
  j = 12 / i;
  j = tab[i];
}

catch (ArrayIndexOutOfBoundsException e) {
  System.out.println("Indice hors limite");
}
catch (ArithmeticException e) {
  System.out.println("Division par 0");
}
```

- si `i = 0` alors division par `0;`
- si `i ≥ 5` alors indice hors limite;
- si on tape autre chose qu'un nombre alors DUMP et arrêt du programme.

11.3 Exceptions contrôlées (checked)

a) En général

Elles sont déclarées comme suit:

```
constructeur | méthode  ( paramètres ) throws UneException {
  ...
}
```

ce constructeur ou cette méthode:

1. soit se termine normalement;
2. soit lève une exception de la classe `UneException`.

Si on utilise un constructeur ou une méthode qui peut lever une exception contrôlée, on doit nécessairement en prévoir le traitement

- soit en la propageant au niveau supérieur (clause **`throws`**);
- soit en prévoyant un bloc **`try`** et un bloc **`catch`**.

b) Exceptions contrôlées standard

Ce sont des sous-classes de la classe `Exception` autres que `RuntimeException`. Par exemple, la sous-classe `IOException` qui gère toutes les exceptions d'entrée (*input*) et sorties (*output*).

11.4 Remarques sur la syntaxe

```
instructions

try {
   ...
}
catch (...) {
   ...                    } peut être répété n fois
}
finally {
   ...                    } optionnel
}

instructions
```

Remarques:

1. on ne peut rien insérer entre un bloc **`try`** et un bloc **`catch;`**
2. lorsqu'il y a un bloc **`finally`**, il est exécuté qu'il y ait ou non une exception;
3. les { et } sont obligatoires dans les trois blocs (**`try`**, **`catch`** et **`finally`**);
4. un bloc **`try`** ne peut pas exister seul, il est toujours suivi de 0 ou plusieurs blocs **`catch`**, eux-mêmes suivis optionellement d'un bloc **`finally;`**
5. s'il n'y a pas de bloc **`catch`**, il y a nécessairement un bloc **`finally`**.

11.5 Exceptions levées par l'utilisateur

Comme ce sont des objets, les exceptions levées par l'utilisateur sont instanciées avec **`new`**, puis elles sont levées par **`throw`** (qu'il ne faut pas confondre avec **`throws`**!).

Exemple 11-2: lecture d'un entier compris entre deux bornes

```
public class EntierException extends Exception {
  public EntierException (String message) {
    super(message);
  }
}

public static int getInteger(TextField t, int min, int max)
  throws EntierException
{
  /* le nombre doit être >= min et <= max */
  int n; // valeur du nombre décodé

  try{
    n = Integer.parseInt(t.getText());
    // ici on est sûr qu'on a un entier
    if (n < min) throw
     new EntierException("nombre inférieur à " + min);
    else if (n > max) throw
     new EntierException("nombre supérieur à " + max);
    // ici on est sûr que le nombre est correct
    return n;
  }

  // exception qui peut être levée par parseInt
  catch (NumberFormatException nfe){
    throw new EntierException("nombre incorrect");
  }
}
```

```
...
// lecture d'un entier syntaxiquement correct
// et compris entre les deux bornes 1 et 100
int entierSur;
try {
  entierSur = getInteger(textField1,1,100);
  // ... suite du programme qui utilise
  // ... la valeur retournée
}
catch (EntierException ee) {
  System.out.println(ee.getMessage());
}
...
```

12. FICHIERS

12.1 Définition

Jusqu'à maintenant, toute l'information traitée était *volatile*.

Les fichiers permettent de garder de l'information de façon *permanente*.

- mémoire de masse
- éléments de base des systèmes d'exploitation.

12.2 Fichiers à accès séquentiel

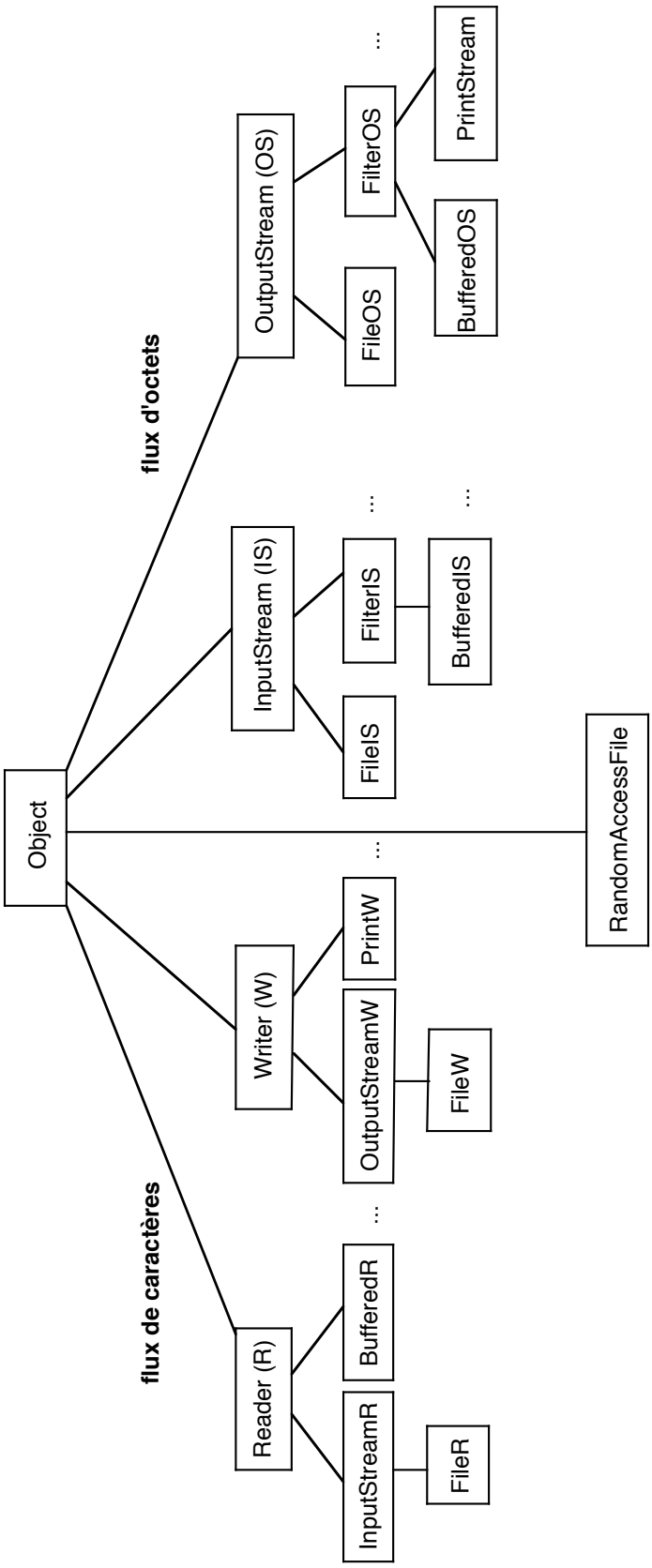
- soit on lit (*read*)
- soit on écrit (*write*)

Pour accéder au n^{ème} élément d'un fichier à accès séquentiel, il faut, au préalable, accéder aux (n-1) qui précèdent.

Idée: un enregistreur à cassette.

On parle aussi de *flux* (*stream*).

12.3 Vue partielle de la hiérarchie des classes du paquetage java.io



12.4 Flux standard

Il y a trois flux standard (d'octets) définis dans la classe `System`.

Ce sont:

- `in` (de type `InputStream`) => clavier;
- `out` (de type `OutputStream`) => écran;
- `err` (de type `OutputStream`) => écran.

Exemple 12-1: utilisation des flux standard

```
import java.io.*;

...

 int i = 0;
 int nbCarTapes = 0;
 System.out.print("Taper qqch: ");

 // les méthodes read() et available() peuvent
 // lever une exception
 try {
  i = System.in.read();
  nbCarTapes = System.in.available();
 }
 catch (IOException e) {
  System.err.println(e.getMessage());
 }

 System.out.println("Lu: " + (char)i);
 System.out.print("Il reste " + nbCarTapes);
 System.out.println(" caractères à lire");
```

12.5 Fichiers physiques

Ce sont les fichiers qui résident sur les mémoires de masse, par exemple le disque dur interne.

La manière de les désigner dépend du système d'exploitation. Par exemple, sur Mac:

```
/disque/répertoire/sous-répertoire/.../fichier
```

Pour pouvoir lire ou écrire un fichier physique, il faut faire un lien entre ce fichier et le fichier logique, i.e. celui qu'on manipule dans le programme.

Ouvrir en lecture:
le fichier doit exister, sinon l'exception `FileNotFoundException` est levée.

Ouvrir en écriture:

- si le fichier n'existe pas, on le crée;
- si le fichier existe déjà, l'ancien est détruit !

Exemple 12-2: écrire des chaînes de caractères dans un fichier

```
import java.io.*;

...

 try {
  // créer un fichier physique en écriture
  FileWriter fichierOut =
     new FileWriter("toto.txt");

  // associer le fichier créé à un flux de sortie
  PrintWriter p = new PrintWriter (fichierOut);

  // écrire dans le fichier
  p.println("salut");
  p.println("bonjour");
  p.println("hello");
  p.println("world");
  p.close();

  // ouvrir le fichier en lecture
  FileReader fichierIn =
     new FileReader("toto.txt");
  // associer le fichier créé à un flux d'entrée
  BufferedReader b = new BufferedReader(fichierIn);
  // tant qu'il y a des lignes ....
  while (b.ready()) System.out.println(b.readLine());
  b.close();
 }
 // on capture toutes les exceptions d'entrées - sorties
 catch (IOException ee) {System.out.println("erreur");}
```

12.6 Fichiers à accès direct

Un fichier à *accès direct* se comporte comme un immense tableau. En principe, ses éléments doivent tous être de même taille. On peut accéder directement à n'importe quel élément du fichier. Le temps d'accès ne dépend pas de la position de l'élément. C'est pourquoi on parle aussi de fichier à *accès aléatoire*.

Un fichier à accès direct dispose d'une sorte de curseur, le *pointeur de fichier*, qui joue un rôle analogue aux indices d'un tableau. Le pointeur de fichier s'exprime en nombre d'octets.

Il y a deux méthodes pour agir sur le pointeur de fichier:

- `long getFilePointer()`
- `void seek(long pos)`

Il y a deux modes d'accès:

- `r (read):`
  on ne peut que lire les éléments, qui bien entendu doivent exister;

- `rw (read/write):`
  on peut lire et écrire.

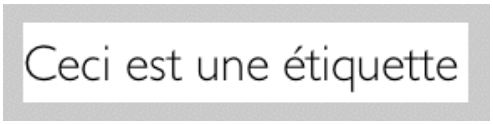
ANNEXES
ET
BIBLIOGRAPHIE

ANNEXE 1

Principaux composants graphiques

1. Etiquette

Une étiquette *(Label)* permet d'afficher un texte:



Méthodes:

```
public String getText()
```

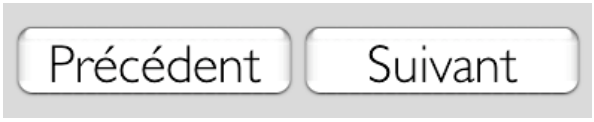
cette méthode retourne, sous forme de chaîne, le texte contenu dans l'étiquette.

```
public void setText(String texte)
```

cette méthode permet de modifier le texte de l'étiquette.

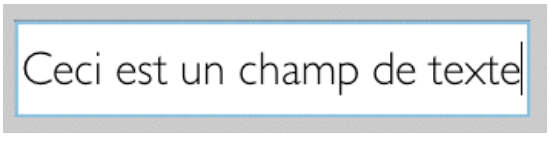
2. Bouton

Un bouton *(Button)* permet de déclencher une action.



3. Champ de texte

Un champ de texte *(TextField)* permet de saisir un texte.



Méthodes:

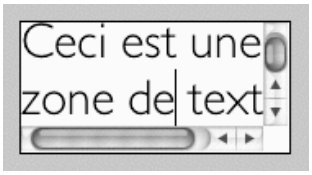
```
public String getText()
```

retourne le texte qui se trouve dans le champ de texte.

public void `setText(String t)`
permet de modifier le contenu d'un champ de texte.

4. Zone de texte

Une zone de texte *(TextArea)* permet de saisir un texte sur plusieurs lignes.



Méthodes:

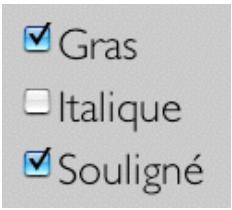
public `String getText()`
retourne le texte qui se trouve dans la zone de texte.

public void `setText(String t)`
permet de modifier tout le contenu d'une zone de texte.

public void `append(String t)`
ajoute le texte `t` à la fin de la zone de texte.

5. Case à cocher

Une case à cocher *(Checkbox)* permet de fixer des choix.



Méthodes:

public boolean `getState()`
retourne **`true`** si la case est cochée et **`false`** dans le cas contraire.

public void `setState(`**boolean** `b)`
coche ou décoche la case selon la valeur de `b`.

6. Bouton radio

Un bouton radio *(RadioButton)* permet d'effectuer un choix unique dans un ensemble d'options.

7. Liste

Une liste *(List)* présente plusieurs champs qui peuvent être sélectionnés. Une liste est munie ou non d'une barre de défilement verticale.

Méthodes:

```
public int [] getSelectedIndexes()
```

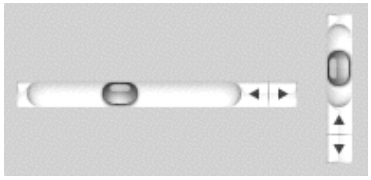
retourne les indices des éléments sélectionnés sous forme d'un tableau d'entiers.

```
public String[] getSelectedItems()
```

retourne les éléments sélectionnés sous forme d'un tableau de chaînes.

8. Barre de défilement

Une barre de défilement ou glissière *(Scrollbar)* permet de commander le déplacement du contenu d'un composant. Elle permet aussi de choisir une valeur entre un minimum et un maximum. On en trouve de deux types: les horizontales et les verticales.



9. Canevas

Un canevas *(Canvas)* permet de dessiner.

ANNEXE 2

La classe String

La classe `String` permet de représenter des chaînes de caractères. Tout littéral de chaîne, comme "Hello", est implémenté par une instance de cette classe.
Un objet de la classe `String` est toujours constant, sa valeur ne peut pas être changée après sa création.

Exemple A2-1: la classe String

```
String s = "Hello";
```

est équivalent à :

```
String s;
...
s = new String("Hello");
```

Opérateur:
L'opérateur + permet de concaténer des chaînes de caractères.

Exemple A2-2: concaténation de chaînes de caractères

```
String s ="Hello";
String str;
...
str = s + " World";
```

`str` vaut : `"Hello World"`

Remarque:
lors d'une opération de concaténation, les opérandes de type primitif (entier, réel, caractère et booléen) sont automatiquement convertis en chaînes.

Exemple A2-3: conversion de types primitifs

```
int i = 124;
double pi = 3.14159;
String s;
...
s = "Résultat: " + i + '*' + pi;
```

`s` vaut: `Résultat: 124*3.14159`

Quelques méthodes:
le premier caractère d'une chaîne porte l'indice 0 !

- **`int`** `length()`

  retourne la longueur de la chaîne.

- **`char`** `charAt(`**`int`** `index)`

  retourne le caractère qui occupe la position spécifiée par `index`.

- **`boolean`** `equals(String str)`

  vrai si la chaîne est égal à `str`.

- **`int`** `compareTo(String str)`

  nul: les chaînes sont égales ;
  négatif: la chaîne précède lexicographiquement son argument ;
  positif: la chaîne suit lexicographiquement son argument.

- **`int`** `indexOf(String str)`

  retourne l'indice du premier caractère de `str` dans la chaîne. Si `str` n'est pas une sous-chaîne de la chaîne, la méthode retourne -1.

- `String substring(`**`int`** `beginIndex,` **`int`** `endIndex)`

  retourne la sous-chaîne qui commence par `beginIndex` et se termine par `endIndex-1`. On a donc: `s` est identique à `s.substring (0, s.length()).`

- `String toLowerCase()`

  transforme la chaîne en minuscules.

- `String toUpperCase()`

  transforme la chaîne en majuscules.

ANNEXE 3

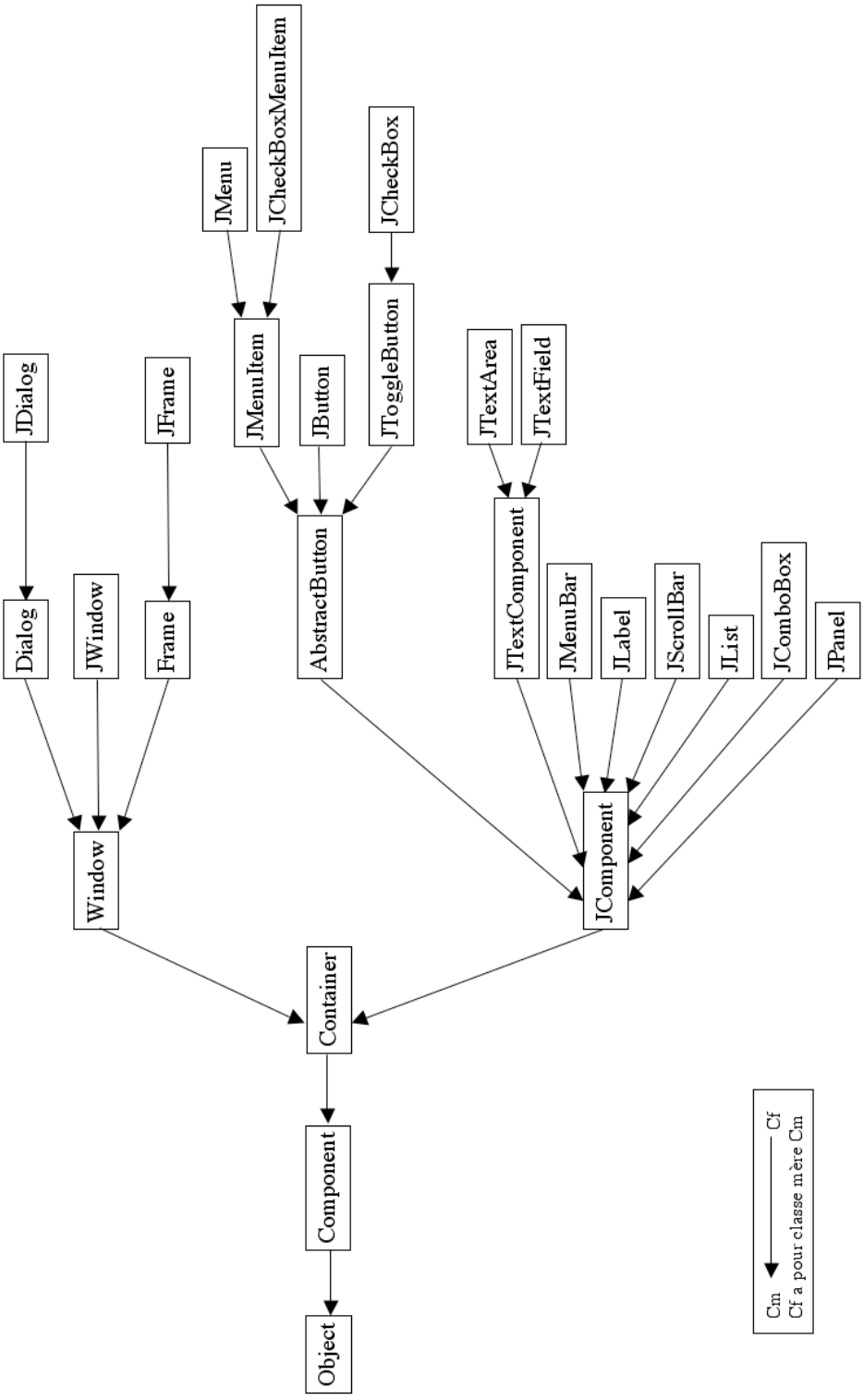
Quelques caractères spéciaux

Caractères graphiques	
°	degré
§	paragraphe
"	guillemet
%	pourcentage
&	et (commercial)
/	slash
\	backslash [<maj><alt>/]
@	at sign, a commercial, "queue de singe" [<alt>g]
#	dièse [<alt>3]
$	dollar
£	pound, "livre sterling"
~	tilde [<alt>n]
[ ]	crochets [<alt>5 et <alt>6]
{ }	accolades [<alt>8 et <alt>9]

Caractères non graphiques	
↵	retour de chariot
→|	tabulateur
←	backspace, touche effacer
⌘	touche pomme
<esc>	escape - sortir, s'échapper
<ctrl>	control - contrôler
<alt>	touche alt (alternative)

ANNEXE 4

Hiérarchie partielle des classes dans Swing



ANNEXE 5

Quelques classes en Java

StringTokenizer

La classe `StringTokenizer` permet de découper en morceaux une chaîne de caractères en fonction d'un certain nombre de caractères de séparation.

Constructeur:

```
StringTokenizer(String chaineADécouper,String
chaineDeSeparation)
```

ce constructeur a deux arguments: la chaîne que l'on désire découper et la chaîne que l'on emploie pour le découpage.

Méthodes:

```
int countTokens()
```

cette méthode retourne le nombre de morceaux détecté dans la chaîne.

```
String nextToken()
```

cette méthode retourne le prochain morceau.

```
boolean hasMoreTokens()
```

cette méthode retourne vrai s'il reste des morceaux à retourner, faux sinon.

Exemple:

on dispose d'une chaîne contenant les jours de la semaine séparés par des barres obliques:

```
String chaine="lundi/mardi/mercredi/jeudi/vendredi";
```

On désire afficher dans la console le nombre de jours ainsi que les jours. On commence par découper la chaîne à l'aide d'un `StringTokenizer`:

```
StringTokenizer stTo = new StringTokenizer(chaine,"/");
```

On affiche le nombre de jours à l'aide de la méthode `countTokens()`:

```
System.out.println("il y a "+stTo.countTokens()+" jours");
```

Puis à l'aide d'une boucle `while` qui s'exécute tant qu'il reste des éléments, on affiche les jours:

```
while(stTo.hasMoreTokens()){
       System.out.println(stTo.nextToken());
}
```

Ce programme affiche dans la console le texte suivant:

```
il y a 5 jours
lundi
mardi
mercredi
jeudi
vendredi
```

Math

Cette classe permet de manipuler des nombres. Elle contient de nombreuses méthodes très utiles pour effectuer des calculs avec des nombres entiers et réels. Cette classe se trouve dans le paquetage java.lang.

Constantes:

`public final static double` `E`

`E` représente une approximation de la base des logarithmes naturels (2.71828183).

`public final static double` `PI`

`PI` représente une approximation du nombre irrationnel pi (3.14159265).

Méthodes:

`public static` `nombre abs(nombre n)`

retourne la valeur absolue du nombre n qui peut être de type **`double`**, **`float`**, **`int`** ou **`long`**.

`public static double` `exp(`**`double`** `a)`

retourne le nombre exponentiel `e` élevé à la puissance du **`double`** a.

`public static double` `log(`**`double`** `a)`

retourne le logarithme naturel (en base `e`) du **`double`** a.

public static nombre max(nombre m, nombre n)

retourne le maximum entre les deux nombres m et n qui peuvent être de type **double**, **float**, **int** ou **long**.

public static nombre min(nombre m, nombre n)

retourne le minimum entre les deux nombres m et n qui peuvent être de type **double**, **float**, **int** ou **long**.

public static double pow(**double** a, **double** b)

retourne sous la forme d'un **double** le premier argument élevé à la puissance du second.

public static int round(réel a)

cette méthode arrondit le nombre réel a qui peut être de type **float** ou **double**.

public static double random()

retourne un **double** aléatoire compris entre 0.0 et 1.0.

public static double sin(**double** a)

retourne un **double** qui est le sinus de a.

public static double cos(**double** a)

retourne un **double** qui est le cosinus de a.

public static double tan(**double** a)

retourne un **double** qui est la tangente de a.

public static double sqrt(**double** a)

retourne la racine carrée du **double** a.

Exemples:

- Math.pow(5,2)=25
- Si l'on désire obtenir un entier entre 0 et 10, il faut prendre un nombre aléatoire, le multiplier par 10 et l'arrondir:

```
int nombre_entre_0_et_10 =
  (int) Math.round(Math.random()*10);
```

- Math.cos(Math.PI)=-1.0

Integer

Cette classe se trouve dans le paquetage java.lang. Elle permet de traiter des nombres entiers comme des objets et non comme des types simples. Dans la pratique, on l'utilise surtout pour convertir des chaînes de caractères en nombres entiers.

Constantes:

`public static final int MAX_VALUE`

la plus grande valeur qui peut être stockée dans une variable entière (2'147'483'647).

`public static final int MIN_VALUE`

la plus petite valeur qui peut être stockée dans une variable entière (-2'147'483'648).

Méthode:

`public static int parseInt(String s)`

convertit la chaîne `s` en un entier **`int`**. Convertit par exemple la chaîne `"347"` en le nombre `347` qui sont des choses différentes. On l'emploie par exemple lorsque l'on veut récupérer un nombre entré par un utilisateur dans un champ de texte (cf. Annexe 1).

Graphics

La classe Graphics permet de dessiner sur l'applet ou sur un des divers composants graphiques.

Constructeur:

On n'emploie pas de constructeur pour déclarer un objet de type Graphics. A la place, on utilse la méthode `getGraphics()` qui permet de récupérer, à partir d'un composant donné, un objet graphique sur lequel on peut dessiner.

```
Graphics g = canvas1.getGraphics();
```

Méthodes:

`void drawLine(int x1, int y1, int x2, int y2)`

cette méthode permet de dessiner une ligne du point (x1,y1) au point (x2,y2).

`void drawRect(int x, int y, int largeur, int hauteur)`

cette méthode permet de dessiner un rectangle défini par les coordonnées de son sommet supérieur gauche et par sa largeur et sa hauteur.

`void drawOval(int x, int y, int largeur, int hauteur)`

cette méthode permet de dessiner un ovale. Cet ovale est défini par le rectangle qui le contient. Les paramètres servent donc à définir le rectangle, comme dans la méthode `drawRect()`.

`void setColor(Color c)`

cette méthode permet de changer la couleur courante. Cette couleur sera employée pour tous les dessins faits dans l'objet graphique. Par défaut la couleur est noire.

`void clearRect(int x, int y, int largeur, int hauteur)`

cette méthode permet d'effacer ce qui se trouve dans le rectangle défini par les paramètres.

Exemple:

```
g.setColor(Color.pink)
```

sélectionne la couleur rose pour dessiner dans l'objet graphique `g`.

BIBLIOGRAPHIE

- CAMPIONE M., WALRATH K.
  The Java Tutorial
  3rd Edition, Reading, MA: Addison-Wesley, 2001

- DEITEL H.M., DEITEL P.J.
  Comment Programmer en Java
  4ème Edition, Reynald Goulet Inc., 2002

- CLAVEL G., MIROUZE N., MUNEROT S., PICHON E., SOUKAL M., TITTANNEAU S.
  Java: La Synthèse
  3ème Edition, Paris: Masson, 2000

- FLANAGAN D.
  Java in a Nutshell
  4th Edition, Sebastopol, CA: O'Reilly, 2002

- LEMAY L., CADENHEAD R.
  Java 2
  2ème édition, Campuspress, 2004

- ARNOLD K.
  Le Langage Java
  Magnard, 1998

- GOSLING J.
  The Java Language Specification
  Reading, MA, Addison Wesley, 1998

- ACREMANN D., DUPIN S., MOUJEARD G.
  JBuilder 3
  Campuspress, 1999