Programmation avec Java

Pourquoi Java ?

- langage moderne (moins de détails étranges que le C++)
- utilise l'approche orientée objet (OO)
- syntaxe de base similaire au C et C++
- librairie standard très riche
- très utilisé; voir les **statistiques**

Mais le langage a beaucoup moins d'importance que la qualité du programme.

Qu'est-ce qu'un programme ?

- *une séquence d'instructions indiquant à un ordinateur ce qu'il doit faire*
- l'ordre d'exécution de cette séquence est important

Qu'est-ce qu'un langage de programmation ?

Un langage de programmation est un ensemble de règles qui déterminent quand une séquence de symboles constitue un programme dans ce langage.

Deux types de règles

- règles syntaxiques
- règles sémantiques

Exemple :

Règles syntaxiques :
- le numéro de téléphone est écrit comme

  (chiffre chiffre chiffre) chiffre chiffre chiffre - chiffre chiffre chiffre chiffre

- les espaces ne sont pas permis

Règles sémantiques :
- les premiers trois chiffres constituent le code régional
- les autres chiffres constituent le numéro local

Avantages et inconvénients d'un langage de haut niveau

Avantages	Inconvénients
programmation plus rapide	incompréhensible pour l'unité centrale (CPU)

instructions plus riches et plus compréhensibles	doit être traduit (ou interprété) en langage machine (processus de compilation)
concepts plus abstraits (variables, structures, ...)	
les chances de se tromper beaucoup moins élevées	

Qu'est-ce qu'un algorithme ?

L'ordinateur est stupide !

Il comprend des instructions très primitives. Si l'on veut qu'il fasse des choses plus avancées, il faut lui dire comment le faire, il faut lui fournir un algorithme.

Un algorithme est (cf. Wikipédia)

- *un processus systématique de résolution, par le calcul, d'un problème permettant de présenter les étapes vers le résultat à une autre personne physique (un autre humain) ou virtuelle (un calculateur)*
- un énoncé d'une suite d'opérations permettant de donner la réponse à un problème.

Algorithme

- Faire chauffer le four à 375f.
- Mélanger l'oeuf avec la cassonade, l'huile, la mélasse et les bananes.
- Ajouter le gruau, le son de blé, la farine, la graine de lin, le soda, le sel, la poudre à pâte, la cannelle, les raisins et les pépites de chocolat.
- Ajouter le lait en dernier.
- Bien mélanger et mettre dans les moules à muffin.
- Cuire environ 25 min à 375f.

Exécution séquentielle, parallèle, répartie

Si les opérations s'exécutent en séquence, on parle d'*algorithme séquentiel*. Si les opérations s'exécutent sur plusieurs processeurs en parallèle, on parle d'*algorithme parallèle*. Si les tâches s'exécutent sur un réseau de processeurs on parle d'algorithme *réparti* ou *distribué*.

Deux types de programmes Java

Les Applets

- s'exécutent dans des navigateurs Web
- ne peuvent pas accéder aux fichiers locaux

Les applications

- programmes autonomes qui résident sur la machine qui les exécute
- ne peuvent pas être intégrés dans une page Web
- toute application doit contenir une méthode principale appelée "main", exécutée quand le programme démarre

Les composants d'un programme Java

Première vue

- données
- actions

Exemples des actions

- cercle : afficher, calculer le périmètre, modifier le rayon, modifier la couleur, ...
- souris : déplacer, cliquer, double-cliquer, glisser-déplacer, ...
- dictionnaire : trouver la traduction, ajouter un nouveau terme, effacer un terme existant, modifier un terme existant, ...

Exemples des attributs (propriétés)

- cercle : rayon, centre, couleur, ...
- souris : position, état, type, ...
- dictionnaire : date de création, nombre de termes, langage source, langage cible, ...
- colis postal : largeur, hauteur, profondeur, poids, distance, ...
- étudiant : nombre de crédits à faire, nombre de crédits réussis, code permanent, ...
- variable : nom, type, emplacement, taille, ...

Synonymes :

- donnée, objet, variable, attribut
- action, méthode, instructions

Différences entre une déclaration et un usage

- un programme manipule des données caractérisées par un nom et un type

- les données sont stockées en mémoire; au moment de la traduction du programme, le compilateur affecte à chaque donnée un emplacement en mémoire caractérisé par une adresse et une taille. Il le fait en utilisant les informations trouvées dans sa déclaration
- les déclarations permettent au compilateur de détecter des erreurs de programmation

Exemple :

```
Si x est déclarée comme une chaîne de caractères,
l'expression x * x sera considérée
comme une erreur de programmation.
```

Java est un langage fortement typé : chaque entité (variable, type, méthode, classe, package, exception) doit être déclarée avant sa première utilisation.

Bases de la programmation orientée objet

Les objets

- système logiciel = collection d'objets qui coopèrent pour résoudre un problème
- les objets correspondent à des entités du monde réel
- la solution d'un problème est équivalente à l'exécution de plusieurs tâches
- les tâches sont exécutées par les objets
- un objet possède des propriétés et un comportement - la manière dont il réagit aux messages lui envoyés;
  le comportement est défini par l'ensemble des messages que l'objet est capable de reconnaitre et de réaliser;
  ces messages sont appelées des *méthodes* (ou des *fonctions membres*)
- une méthode = une action exécutée par un objet suite à la réception d'un message
- un objet possède une identité qui permet de le distinguer des autres objets;
  cette identité est appelée *le nom de l'objet*
- la liste de valeurs des propriétés constitue *l'état de l'objet*;

  Exemple :

  objet : cours5043
  prof : Marc Ennuyeux
  titre : PHI3033 Philosophie des mathématiques
  université : UQO

Les classes et les objets

- il y a une multitude des objets; ils sont donc regroupés

- une *classe* regroupe des objets qui partagent les mêmes propriétés et les mêmes comportements (répondent aux mêmes requêtes)
- on dit qu'un objet est *une instance d'une classe* (une instanciation d'une classe) (il est issu de cette classe)
- une classe comprend deux parties :
  - les *attributs* (appelés souvent *données membres*) : il s'agit des données représentant l'état de l'objet
  - les *méthodes* (appelées souvent *fonctions membres*) : il s'agit des opérations applicables aux objets
- notation pour accéder à un attribut :
  - *nom_de_l_objet . nom_de_l_attribut*
    exemple : etud1.cours
- notation pour invoquer (appeler) une méthode :
  - *nom_de_l_objet . nom_de_la_methode* (arg_1, arg_2, ..., arg_n)
    exemple : etud1.abandonner("inf1563")

Structure de programme Java

- le programme est une classe qui peut utiliser d'autres classes
- les classes utilisées sont des classes de la **librarie Java** ou des classes possiblement écrites par d'autres personnes; principe de la POO : réutiliser le code !
- les classes peuvent être regroupées en paquetages ("packages")

Variables et types

Qu'est-ce qu'une variable ?

- les variables permettent de stocker de l'information
- une variable contient une valeur à la fois
- chaque variable possède trois attributs : nom, type et valeur
- la valeur peut être modifiée à l'aide de l'instruction d'affectation
- le nom et le type ne peuvent pas être modifiés

Chaque donnée (variable ou constante) a son type en Java.

Le type sert à déterminer la taille de l'espace mémoire et la façon dont est interprété le code binaire de la valeur qui y est stockée.

Java utilise les types de données suivants :

- les nombres entiers
- les nombres réels
- les caractères et chaînes de caractères
- les booléens
- les objets

Types de données prédéfinis :

Type	Mot-clé Java	Codage	Valeurs
caractère	char	2 octets	'a', 'A', '"', ' '
entier	int	4 octets	2009, 0xFF, -13
entier "long"	long	8 octets	2147418112
octet	byte	1 octet	-128, 0, 127
entier "court"	short	2 octets	-32768, 0, 32767
nombre réel	float	4 octets	134.456F, 23E7F
nombre réel "long"	double	8 octets	134.456, -45E-16
booléen	boolean	1 octet	false, true
chaîne de caractères	String	???	"INF1563", "a", "aujourd'hui", "", " "
...	...	...	...

Représentation de données en Java

objet	attribut	type
colis postal	largeur	float
étudiant	nombre de crédits à faire	short
	sexe	char
	détenteur d'un DEC	boolean
	code permanent	String
navette spatiale	vitesse	double

compte de banque	solde en $	int

Déclaration de variables

Deux syntaxes possibles :

- <type> <nomDeVariable>;
- <type> <nomDeVariable1>,...,<nomDeVariableN>;

Exemple :

```
int i;
float largeur, longueur, hauteur;
```

Constantes

Une constante est une variable dont la valeur est inchangeable lors de l'exécution d'un programme.

En Java, le mot clé `final` permet de définir une variable dont la valeur ne peut pas être modifiée après son initialisation.

Exemple

```
public static final double PI = 3.1415926535897793;
```

Java ne supporte pas de vraies constantes. À chaque exécution du programme, la valeur utilisée pour initialiser la variable peut être différente.

Exemple

```
final String ARG1 = args[0];
```

Bonne pratique de la programmation

- l'utilisation de variables constantes améliore la lisibilité du programme
- au cas où on doit changer la valeur de la constante, il n'y a qu'une seule place pour apporter la modification
- les noms de constantes sont écrits en majuscules avec le caractère "_" comme séparateur

Syntaxe

La grammaire du langage

- sensible à la casse (aux majuscules/minuscules)
- indépendante de la mise en page

Éléments syntaxiques

- mots réservés :

```
abstract continue for          new       switch
assert   default  goto         package
synchronized
boolean  do       if           private   this
break    double   implements   protected throw
byte     else     import       public    throws
case     enum     instanceof   return    transient
catch    extends  int          short     try
char     final    interface    static    void
class    finally  long         strictfp  volatile
         const    float    native       super
         while
```

- identificateurs (noms) : args cercle main String System out println
- littéraux : "Salut Jean"
- ponctuation : , {} ; [] ()
- commentaires : // Dis Bonjour!

Les entités suivantes sont identifiées par un nom :

- variables
- classes
- packages
- types (prédéfinis)
- méthodes
- exceptions

Syntaxe de noms

- tous les caractères alphanumériques et _ sont permis dans les noms
- un nom commence par une lettre, $ ou _
- mots-clés ne sont pas permis

Exemples de noms corrects :

i
somme
Taxe_de_vente
INF1563

Exemples de noms incorrects :
1563INF
facture*2
mon-chien

Bonnes pratiques de la programmation : choix de noms

- dans le cycle de vie d'un produit logiciel, la phase de maintenance représente la majeur partie du temps (environ 80%)
- un logiciel est rarement développé par une seule personne
- plusieurs personnes vont lire le code et seront obligées de le comprendre
- en règle générale, ce ne sont pas ceux qui ont procédé à sa création; leur temps d'adaptation avant une pleine productivité dépend de leur capacité à comprendre le code source et à assimiler la documentation relative au projet
- la réussite d'un projet logiciel dépend des moyens mis en oeuvre pour assurer une consistance dans le codage
- des conventions strictes respectées par tous sont nécessaires
- des outils de développement proposent certaines fonctionnalités qui facilitent la consistance du code source; ces fonctionnalités ne sont pas suffisantes
- nous allons au cours du cours présenter des directives ou des stratégies appelées "les bonnes pratiques de la programmation"

Directives pour choisir un bon nom

- Le nom doit être à la fois explicit (dénoter le contenu) et court
  - nombreEtuds au lieu de compteur
  - périmètre au lieu de lePerimetreDuCercle
- Être descriptif pour distinguer des variables reliées
  - ancienSolde et nouveauSolde au lieu de solde1 et solde2
- Ne pas inclure le type de la variable dans son nom
  - poids au lieu de poidsFloat
- Les noms de variable doivent être en minuscule hormis les initiales des mots le composant (sauf le premier)
  - nouveauSolde au lieu de NouveauSolde

POO : Conception de classes

Méthodes

Définition des méthodes

un algorithme paramétré = une méthode (ou une fonction)

Structure d'une méthode

- nom : preparerMuffins
- paramètres : fruits, température du four, le temps de cuisson
- séquence d'actions : faire chauffer le four, mélanger l'oeuf avec la cassonade, l'huile, ...
- valeur de retour : plusieurs muffins

Terminologie Java

- paramètre = entrée
- valeur de retour = sortie
- séquence d'actions = séquence d'instructions = l'implémentation de l'algorithme
- le nom et les types de paramètres = la signature de la méthode
- la méthode/fonction est invoquée ou appelée

Exemples :

```
static double somme(double p1, double p2){
  return p1 + p2;
}

static void somme(double p1, double p2){
  double s = p1 + p2;
  System.out.println("Somme = " + s);
}
```

Pourquoi les méthodes ?

- au lieu d'écrire le code entier du programme, nous le divisons en morceaux réutilisables
- le code est plus facile à comprendre
- facile de corriger les erreurs et d'apporter des modifications
- plus facile de faire abstraction de certains détails sans importance pour la méthode
- plus facile de diviser le travail parmi les membres d'une équipe

Utilisation des méthodes

- les méthodes correspondent à des opérateurs sur les objets
- des centaines de méthodes prédéfinies :
  - `Integer.parseInt ( valeur_de_type_String );`
  - `Math.sin ( valeur_de_type_double );`
  - *objet_de_type_String* `. replace ( valeur_de_type_String , valeur_de_type_String );`
  - `System.out . print ( valeur_de_type_String );`

Remarques

- l'ordre des paramètres est important : uneChaineDeCaracteres.replace("ab", "cd") est très différent de uneChaineDeCaracteres.replace("cd", "ab")
- *paramètre effectif (actuel)* = une expression dont le type est convertible au type du paramètre formel
  paramètre formel = une variable locale initialisée avec la valeur du paramètre effectif lorsque la méthode est appelée

  Exemple :

```
static double somme(double p1, double p2){
  return p1 + p2;
}
double x = somme(1, 2.9);
```

  Remarque : la valeur 1 de type `int` sera convertie en une valeur 1.0 de type `double` et cette dernière sera utilisée pour initialiser p1

  Exemple :

```
static double somme(float p1, float p2){
  return p1 + p2;
}
double x = somme(1, 2.9);
```

  Remarque : le littéral 2.9 est par défaut de type `double`; il ne peut pas être converti en une valeur de type `float` sans perte de précision

- le nombre et le type de paramètres dans la déclaration et dans l'appel doivent correspondre; sinon, une erreur de compilation sera détectée
- même si une méthode change ses paramètres formels, les valeurs des paramètres effectifs ne seront pas changées

  Exemple :

```
public static void afficherSomme(int i, int j) {
  i = i + j;
```

```
  System.out.println(i);
}

public static void main(String[] args) {
  int k = 1, m = 2;
  afficherSomme(k, m);
  System.out.println(k); // k == 1
}
```

Résultats :

```
3
1
```

- une méthode peut retourner une valeur
  - si une méthode ne retourne pas de valeur, le type de retour est `void`

    l'exécution de la méthode est terminée si la dernière instruction a été exécutée ou l'instruction `retour;` est exécutée

  - si une méthode retourne une valeur, le type de retour doit être spécifié

    la valeur retournée peut être utilisée comme une expression de son type

    Exemple :

    ```
    public int max(Vector v){
      ...
      return v.elementAt(3);
    }
    Vector v = ...;
    int i = max(V) + 5;
    ```

    L'exécution de la méthode est terminée si l'instruction `retour` une_expression`;` est exécutée

- si une méthode est appelée dans la même classe, l'objet est connu; on peut donc utiliser une syntaxe plus simple :

  nomDeMethode (listeDesParamètresActuels);

Conception des méthodes

Métodes accesseurs vs. métodes mutateurs

- méthodes *accesseurs* : des méthodes dont l'appel permet d'obtenir la valeur d'une propriété particulière de l'objet mais la valeur de l'objet ne sera pas modifiée;

  le type de retour n'est pas `void`

  Exemple :

```
double getPerimetre(){
  return 2*a + 2*b;  // a et b représentent les
deux côtés du rectangle
}
...
Rectangle r;
...
double perimetre = r.getPerimetre();
```

- méthodes *mutateurs* : des méthodes dont l'appel permet de définir la valeur d'une propriété particulière de l'objet;

  en général, le type de retour est `void`

  Exemple :

```
void setCouleurFond(col){
  couleur = col;  // couleur représente la couleur
de fond du rectangle
}
...
Rectangle r;
...
r.setCouleurFond("#FFAA80");
```

Il existe aussi des méthodes appelées *constructeurs* dont le but est de créer des objets.

Comment construire une méthode ?

1. identifiez des opérations dans l'algorithme de solution de notre problème
   - des opérations qui sont utilisées plusieurs fois ou qui constituent des unités naturelles de conception
2. choisir un nom qui reflète la tâche effectuée par la méthode
   - un verbe si on modifie un objet
   - un nom ou un verbe si on accède à un attribut d'un objet

   Exemples : estEgal, setPerimetre, getPerimetre (perimetre), colorer, inscrire

3. identifier les paramètres
4. choisir les noms des paramètres
5. identifier le type de retour

6. écrire le pseudocode de l'algorithme
7. traduire le pseudocode en Java
8. tester le code

Exemple

Probleme : trouver les solutions d'une équation du second degré

Une équation du second degré se présente sous la forme suivante :

$ax^2 + bx + c$

Les étapes :

1. nom de la méthode : racine
2. coefficients de l'équation (de type `double`)
3. noms de paramètres : a, b, c
4. pas de type de retour (la méthode affichera directement les solutions trouvées)
5. pseudocode :

```
racine(a, b, c) {
  delta = "discriminant"  // b² - 4ac
  si discriminant == 0 alors
    x1 = -b/2a
  si discriminant > 0 alors
    x1 = (-b - "racine carrée"(discriminant))/2a
    x2 = (-b + "racine carrée"(discriminant))/2a
  si discriminant < 0 alors
    System.out.println("Il n'y a pas de solution");
  si discriminant == 0 alors
    System.out.println("Il y a une solution : x = " + x1);
  si discriminant > 0 alors
    System.out.println("Il y a deux solutions : x1 = " +
x1 + ", x2 = " + x2);
}
```

6. code :

```
static void racine (double a, double b, double c){
    double x1 = 0;
    double x2 = 0;

    double delta = b * b - 4 * a * c;
    if (delta == 0)
      x1 = -b / 2.0 / a;
    else if (delta > 0){
      x1 = (-b - Math.sqrt(delta))/(2 * a);
      x2 = (-b + Math.sqrt(delta))/(2 * a);
```

```
    }
    if (delta < 0)
      System.out.println("Il n'y a pas de solution");
    else if (delta == 0)
      System.out.println("Il y a une solution : x = " +
x1);
    else
      System.out.println("Il y a deux solutions : x1 = " +
x1 + ", x2 = " + x2);
  }
```

7. tests :

```
  racine(1, 0, 1);
  racine(1, 2, 1);
  racine(1, 0, -1);
```

Introduction aux classes

Classe

Une classe est un modèle qui définit les objets d'un type. Une classe comprend des variables et des méthodes. Chaque objet de ce type possède toutes les variables d'instance et peut utiliser toutes les méthodes déclarées dans la classe.

Les valeurs des variables d'instance d'un objet représentent son état. La manière dont l'objet réagit aux applications des méthodes (messages) représente son comportement.

Héritage

L'héritage (en anglais inheritance) est un principe permettant de créer une nouvelle classe à partir d'une classe existante.

- un principe propre à la POO
- la nouvelle classe est appelée une classe dérivée; elle contient les attributs et les méthodes de sa superclasse (la classe dont elle dérive); autrement dit, elle hérite des caractéristiques de la classe "mère"
- la modification de la superclasse implique la modification automatique de toutes les sous-classes
- toutes les méthodes de la classe héritée peuvent être redéfinies
- une classe peut possèder un nombre illimité de sous-classes; par contre, elle ne peut possèder qu'une seule superclasse.
- par défaut, une classe hérite de la "super-superclasse" nommée **Object**.

Avantages

- la possibilité de pouvoir définir de nouveaux attributs et de nouvelles méthodes pour la classe dérivée, qui viennent s'ajouter à ceux et celles héritées
- les programmeurs Java conçoivent une hiérarchie de classe de telle façon que les propriétés et méthodes communes à plusieurs classes soient placées dans une superclasse. Cette approche permet de réutiliser des composants existants et de leur ajouter un comportement

Syntaxe

class classe_derivee extends classe_mere { // Corps de la classe }

- le mot-clé `super` permet de désigner la classe mère
  - pour manipuler une proriété de la classe mère, nous écrivons : super.nom_de_la_propriete
  - pour manipuler une méthode de la classe mère, nous écrivons : super.nom_de_la_methode()

Variables de classe

Les variables statiques, aussi appelées variables de classe, n'appartiennent pas à une instance particulière, elles appartiennent à la classe.

- les variables statiques sont partagées par toutes les instances de la classe
- si une instance modifie la valeur d'une variable statique, la modification affecte toutes les instances
- une variable statique existe dès que sa classe est chargée, indépendamment de toute instanciation.

Exemple :

```
public class Compteur {
  static int nb = 0;
  public Compteur(){
    nb++;
  }
  ...
}

public class TestCompteur {
  public static void main(String[] args) {
    Compteur cmpt1 = new Compteur();
    Compteur cmpt2 = new Compteur();
    Compteur cmpt3 = new Compteur();
    Compteur cmpt4 = new Compteur();
    ...
```

```
    System.out.println ("Il existe " + Compteur.nb + "
instances de la classe Compteur.");
  }
}
```

Méthodes de classe

Une méthode de classe ou une méthode statique est une méthode qui n'agit pas sur des variables d'instance mais uniquement sur des variables de classe.

- elle peut être appelée même sans avoir instancié la classe
- elle peut accéder uniquement à des variables et méthodes statiques
- dans la déclaration, le nom de la méthode est précédé du mot clé `static`
- elle peut être appelée avec la notation classe.methode() au lieu de objet.methode()

Exemple : la méthode `main`

Exemple :

```
i = Math.abs(j);  // valeur absolue
```

Droits d'accès

- Java fournit des spécificateurs d'accès pour permettre au créateur de classe de dire au programmeur client ce qui est disponible et ce qui ne l'est pas
- on sépare l'interface de la classe de son implémentation
  - on sépare les choses qui changent des choses qui ne changent pas
  - les gens qui utilisent une bibliothèque existante, n'auront pas à réécrire du code si une nouvelle version de la bibliothèque sort
- quatre niveaux de contrôles d'accès
  - public : les champs et méthodes sont accessibles par toutes les classes
  - private : les champs et méthodes ne sont accessibles que par les méthodes de cette classe
  - pas de spécificateur : (l'accès amical) toutes les autres classes du package courant ont accès au membre; pour les classes hors du package, le membre devient `private`
  - protected : la ressource est accessible à toutes les classes qui héritent de la classe courante
- les spécificateurs sont placés devant la définition de chaque membre (un champ ou une méthode)

Exemple :

```
class Date {
  private int mois;
  private int jour;
  private int annee;
  ...
}
```

Les données mois, jour et année ont été déclarées privées. Elles ne seront accessibles que par des méthodes définies de la classe Date.

Comment utiliser les spécificateurs d'accès ?

Une bonne pratique de la programmation

- restreindre l'accès direct aux données avec le spécificateur `private`
- fournir des méthodes accesseurs / mutateurs (méthodes "get/set") qui lisent et changent la donnée

Visibilité (portée) des variables

Variables locales

- elles existent à partir de leur déclaration jusqu'à la fin du bloc
- une variable ne peut pas être redéfinie dans la portée de la même variable
- exemple :

```
{
  int i = 1;
  if (true) {
    int i = 0; // illégal
    int j = 1;
  }
  if (true) {
    int j = 1; // légal
  }
  int j = 0; // légal
}
```

- exception : variable de contrôle de la boucle `for`

```
for (int i = 0; i < 10; i++) {
}
int i; // légal
```

Variables statiques de classe

- elles existent pendant toute l'exécution du programme
- déclarées dans le bloc de la classe
- elles peuvent être redéclarées dans les méthodes (éviter si possible)
- exemple :

```
public class Exemple {
  public static final int MAX = 38; // légal
  public static int a = 1;          // légal mais
déconseillé
  public static void f(float f){
    System.out.println(a);          // affiche 1
    double a = 2.0;                 // légal mais
déconseillé
    System.out.println(a);          // affiche 2.0
    System.out.println(Exemple.a);  // affiche 1
  }
}
```

Variables non-statiques de classe

- elles existent pendant l'existence d'un objet de la classe
- déclarées dans le bloc de la classe
- peuvent être redéclarées dans les méthodes (éviter si possible)
- en général, elles déclarées comme `private` (privées) c.-à-d. connues seulement dans les méthodes de la classe
- exemple :

```
public class Exemple {
  public final int MAX = 38;
  private int a = 1;

  public void f(float f){
    System.out.println(a);       // affiche 1
    double a = 2.0;              // légal mais
déconseillé
    System.out.println(a);       // affiche 2.0
    System.out.println(this.a); // affiche 1
    System.out.println(MAX);     // affiche 38
  }
}
```

Expressions

Les expressions les plus simples et affectations

Une expression est une construction qui décrit comment calculer une valeur

L'évaluation d'une expression produit une valeur.

Les expressions les plus simples

- valeurs littérales (constantes)
  - 0 3 -5 'x' 0.1 3.14159 3.4E-5
- variables
  - Considérons les déclarations suivantes :
    int i, j; char c;
    i, j et c constituent des expressions

Affectations

Une affectation est une opération qui permet d'attribuer une valeur à une variable.

Exemple :

```
int i, j;
j = 3;
i = j;
```

- "j = 3;" signifie "3 est copié dans j"
- "j = 3;" ne correspond pas à "est-ce que j est égal à 3?"

Exemple :

```
int i;
...
i = i + 1;
```

L'affectation est considérée comme une expression à cause du fait qu'elle renvoie une valeur, à savoir la valeur affectée.

Exemple :

```
int i, j, k;
i = j = k = 1;
int somme = i + j + k;          // Calcule 3
System.out.println( "Somme == " + somme );
```

Opérateurs

Les opérateurs servent à former des expressions plus complexes.

Pour chaque opérateur, Java définit

- le nombre d'arguments (opérandes)
  les opérateurs unaires ont un argument et les opérateurs binaires ont deux arguments
- les types des arguments
- le type du résultat

Les opérateurs arithmétiques

Opérateur	Opération
+	addition
-	soustraction
*	multiplication
/	division
%	modulo

Exemple :

```
public class TestOperateurs  {

  public static void main( String [] args ) {

    int i = 10;
    int j = 3;
    int resultat;
    float f = 10;
    float resultatFloat;
    double d = 10;
    double resultatDouble;

    resultat = i + j;     // Calcule 13
    System.out.println( i + "+" + j + " == " + resultat
);
```

```
    resultat = i - j;     // Calcule 7
    System.out.println( i + "-" + j + " == " + resultat
);

    resultat = i * j;     // Calcule 30
    System.out.println( i + "*" + j + " == " + resultat
);

    resultat = i / j;     // Calcule 3
    System.out.println( i + "/" + j + " == " + resultat
);

    resultat = i % j;     // Calcule 1
    System.out.println( i + "%" + j + " == " + resultat
);

    resultatFloat = f / j;     // Calcule 3.3333333
    System.out.println( f + "/" + j + " == " +
resultatFloat );

    resultatDouble = d / j;     // Calcule
3.3333333333333335
    System.out.println( d + "/" + j + " == " +
resultatDouble );

  }

}
```

Exemple :

```
int total = 763;           // cents
int dollars = total / 100; // = 7
int cents = total % 100;   // = 63
System.out.println("total = " + dollars + "." + cents +
"$"); // "total = 7.63$"
```

Bonne pratique de la programmation

Le caractère *espace* autour des opérateurs (sauf + et - unaires).

Exemple :

```
j = -7 + 2 * (-i)
```

Priorité (précédence) des opérateurs

1. unaires +, - (par exemple, -7)
2. *, /, %
3. binaires +, -

Les opérateurs numériques ayant la même priorité sont évalués de gauche à droite.

Les parenthèses peuvent être utilisées pour changer l'ordre de l'évaluation.

Exemple :

expression	**valeur**
2*6+4/2	14
8-4*(3-5)	16
8/4/2*1	1

Les expressions mixtes

- le résultat est double si n'importe quel opérande est double
- le résultat est float si un opérande est float et l'autre n'est pas double
- le résultat est int si chaque opérande est int, short ou byte
- le résultat est long si un opérande est long et l'autre est entier

Les classes Float et Double contiennent deux constantes Float.POSITIVE_INFINITY et Float.NEGATIVE_INFINITY qui représentent le résultat de la division par zéro en virgule flottante.

Exemple :

Soit les déclarations suivantes :
int i = 10;
float f = 1/3;
double d = 1/3.0;
byte b = 100;
short sh = 1000;
long ll = 100;

expression	**valeur**	**type**
i * i	100	int
i + f	10.0	float
i + d	10.333333333333334	double

i * d	3.333333333333333	double
i / f	Infinity	float
b * b	10000	int
sh - b	900	int
ll - b	0L	long

Opérateurs arithmétiques et affectation combinés

affectation	notation équivalente
i = i + 3;	i += 3;
i = i * 4;	i *= 4;
i = i + 1;	i += 1; ou i++;
i = i - 1;	i -= 1; ou i--;

- n'est pas plus efficace à exécuter
- plus concise

Les opérateurs de transtypage ("cast operators")

Le transtypage (ou cast) est la conversion d'une expression d'un certain type en une expression d'un autre type.

- une perte d'information possible; exemple : float -> int

Transtypage de données de types primitifs

- une conversion implicite possible s'il n'y a pas de perte d'information; on passe d'un type "plus petit" vers un type "plus grand"
  Exemples :

```
int i;
short s;
long l;
byte b;
float f;
double d;
 ...
```

```
// les instructions suivantes sont valides
l = i = s = b;
i = b;
d = i;
d = f = s;
```

- une conversion explicite nécessaire s'il y a une perte d'information;
  variable_de_nouveau_type = (nouveau_type)expression_a_convertir;

```
int i;
short s;
...
s = (short)i;
```

Transtypage de référence d'objet

- le transtypage implicite vers un type ancêtre

  Exemples :

```
class Forme { }
class Quadrilatere extends Forme {  }
class Losange extends Quadrilatere {  }
class Carre extends Losange {  }
class Cercle extends Forme {  }
class Triangle extends Forme {  }
  ...
// les instructions suivantes sont valides
Quadrilatere q = new Carre();
Forme f1 = new Losange();
Forme f2 = new Triangle();
```

- le transtypage explicite vers un type dérivé

  Exemple :

```
Forme f1 = new Quadrilatere();
...
q = (Quadrilatere)f1;
```

  Attention : f1 doit être un Quadrilatere !

- certains transtypages ne sont pas possibles

```
Triangle tr = new Triangle();
Losange los = new Losange();
los = (Losange)tr;
```

Opérateurs relationnels

Les opérateurs relationnels sont des opérateurs qui retournent un résultat de type boolean. Ils évaluent les rapports entre les valeurs des opérandes. Une expression relationnelle renvoie true si le rapport est vrai, false dans le cas opposé.

opérateur	signification	domaine d'application
<	plus petit que	types numériques, caractères
>	plus grand que	types numériques, caractères
<=	plus petit que ou égal à	types numériques, caractères
>=	plus grand que ou égal à	types numériques, caractères
==	égalité de valeurs (types primitifs) égalité de références (types objets)	tous les types de données disponibles dans le langage
!=	inégalité de valeurs (types primitifs) inégalité de références (types objets)	tous les types de données disponibles dans le langage

Les opérateurs sur les chaînes de caractères

- opérateur de concaténation : +
- conversion à String est forcée si n'importe quel opérande est String
- la classe String contient plusieurs méthodes : length, substring, toLowerCase, ...
- les éléments de chaînes de caractères sont indexés
- le premier caractère a l'indice 0

Exemples :

Soit la déclaration suivante :
String sigle = "INF1563";

expression	valeur

sigle.**length**()	7
sigle.**substring**(3, 7)	"1563"
sigle.**charAt**(0)	'I'
sigle.**charAt**(1)	'N'
sigle.**toLowerCase**()	"inf1563"
sigle.**isEmpty**()	false
sigle.**compareTo**("inf1563")	-32
(int)'I'	73
(int)'i'	105
sigle.**substring**(3, 7) + sigle.**substring**(3, 7)	15631563
sigle.**compareTo**("inf1563") < 0	true
"INF1563".**compareTo**("INF1593) < 0	true
"INF".**compareTo**(sigle) < 0	true

Méthode equals

La méthode `equals` appartient à la classe `Object` et toutes les autres classes en héritent.

Voici sa définition initiale qui compare les références :

```
public boolean equals(Object obj) {
  return (this == obj) ;
}
```

`equals` se comporte exactement comme `==`. Dans la plus part des classes la méthode est redéfinie pour qu'elle compare les contenus des objets plutôt que leurs références.

Exemple :

```
String s1 = "INF1563";
String s2 = "INF1563";
System.out.println(s1 == s2);          // true
System.out.println(s1.equals(s2));     // true

String s3 = "INF";
String s4 = s1.substring(0, 3);        // "INF"
System.out.println(s3 == s4);          // false
System.out.println("INF".equals(s4)); // true
```

Les opérateurs logiques

Les opérateurs logiques acceptent obligatoirement des opérandes booléennes et calculent un résultat booléen.

opérateur	signification	exemple	vrai si
&&	et	a && b	a et b sont vrais
||	ou	a || b	soit a soit b est vrai
!	non	!a	a est faux

Une évaluation partielle possible

- l'évaluation des expressions logiques est arrêtée dès lors que le résultat est déterminé;
  exemple : expression_vraie || expression_quelconque
- l'évaluation partielle optimise le code mais peut avoir des effets indésirables

Les structures de contrôle de flux

Introduction

L'ordre de flux est l'ordre dans lequel les instructions d'un programme sont exécutées.

les instructions qui ne changent pas d'ordre d'exécution

- déclaration de variable
- instruction d'affectation
- expression

- toute expression comme i++ ou une invocation de méthode devient une instruction si elle est suivie du séparateur d'instructions
- exemple :

```
i++ ;
imprimer(a,b);
```

- bloc d'instructions;
  - regroupe une suite d'instructions placées entre accolades { et }
  - exemple :

```
{
  int i, j;
  i = Math.min(x, y);
  j = Math.max(x, y);
  System.out.println("la valeur de " + i + "
<= que la valeur de " + j);
}
```

- instruction vide
  - permet au programmeur d'omettre une instruction là où la syntaxe en exige une
  - utilisée au cas où aucune action n'est prévue ou cette action sera définie plus tard
  - exemple :

```
if (pile.vide()) ;
else pile.depiler();
```

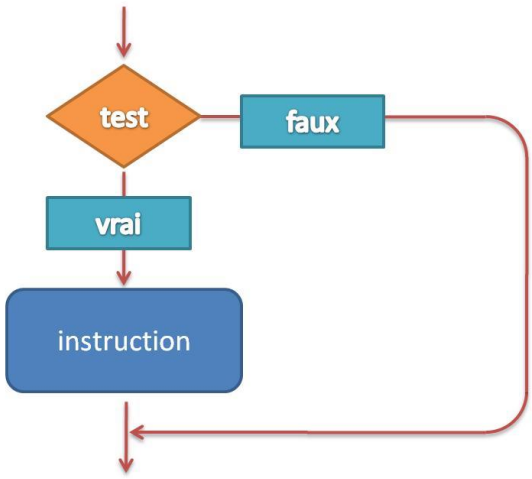
- les instructions qui changent l'ordre d'exécution
  - itérations
    moyens de répéter une séquence d'instructions
    - la boucle `for`
      utiliser si l'on connait à l'avance le nombre d'itérations
    - la boucle `while` avec 'test avant'
      utiliser si la condition est déterminable avant le traitement, ou si l'instruction itérée peut ne pas être exécutée du tout
    - la boucle `while` avec 'test après'
      utiliser si la condition n'est déterminable qu'après une itération, ou si l'itération doit être exécutée au moins une fois
  - instructions conditionnelles
    - permettent de choisir quelles instructions seront exécutées ou pas en fonction de certaines conditions
    - `if else` : le choix est basé sur une expression booléenne
    - `switch` : le choix est basé sur une expression de type entier ou caractère
  - les ruptures de séquence
    - appel de méthode

- `return;`
  elle termine l'exécution d'une méthode ; l'exécution reprend dans la méthode appelante, après le point d'appel
- `return expression;`
  si la méthode est typée, return doit être suivi d'une expression du type de la méthode; la valeur calculée sera transmise à l'appelant
- `break`
  elle a pour effet de quitter directement le switch ou la boucle et de directement poursuivre le programme à l'instruction suivante
- `continue`
  elle permet d'arrêter l'exécution du corps de la boucle et de passer directement à l'itération suivante

Instructions conditionnelles (alternatives)

Les instructions conditionnelles sont des instructions qui permettent d'exécuter un ensemble d'instructions si et seulement si une certaine condition est vérifiée.

L'instruction if



Si l'évaluation de l'expression booléenne *test* donne *true* (la valeur 'vrai'), alors l'instruction est exécutée; sinon on exécute l'instruction suivante.

Syntaxe

if (<expression booléenne>) <instruction>

Exemple :

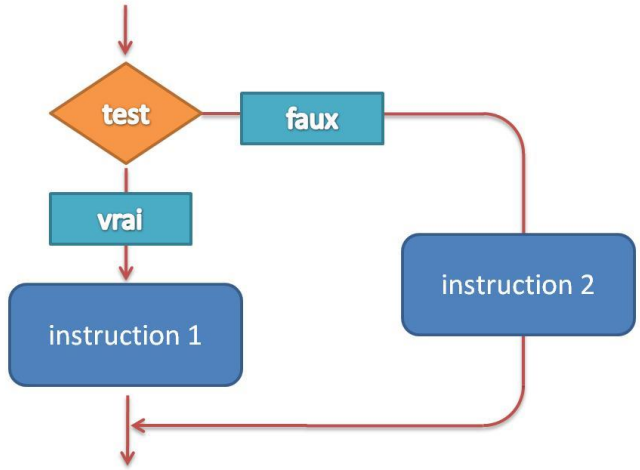
```
if (solde >= montantPreleve)
  solde = solde - montantPreleve;
```

S'il y a plus qu'une instruction à exécuter, utilisez un bloc.

Exemple :

```
if (resultat >= 93.5) {
  etudiant.setNote("A+");
  feliciterEtudiant(etudiant);
}
```

L'instruction if-else



Si l'évaluation de l'expression booléenne *test* donne *true* (la valeur 'vrai'), alors l'instruction 1 est exécutée; sinon l'expression booléenne a donné *false* (la valeur 'faux') et l'instruction 2 est exécutée.

Syntaxe

if (<expression booléenne>) <instruction> else <instruction>

Exemple :

```
if (solde >= montantPreleve)
  solde = solde - montantPreleve;
else
  System.out.println("Solde insuffisant.");
```

Instructions if-else imbriquées

Exemple :

```
if (personne.getAge() >= 18)
  if (personne.consommeAlcool())
    personne.offrir("bière");
  else
    personne.offrir("thé");
else
  personne.offrir("jus");
```

Attention

Le code suivant :

```
if (cond1)
  if (cond2)
    instr1;
else
  instr2;
```

est interprété comme suit :

```
if (cond1) {
  if (cond2) {
    instr1;
  }
  else {
    instr2;
  }
}
```

L'instruction switch

L'instruction `switch`, appelée aussi une instruction de sélection multiple, permet de faire plusieurs tests sur la valeur d'une même expression.

Syntaxe

```
switch (<expression de type int ou String>) {
  case <constante> : <instruction>; break;
  case <constante> : <instruction>; break;
  case <constante> : <instruction>; break;
       ...
  case <constante> : <instruction>; break;
  default : <instruction>;
}
```

Sémantique

- les parenthèses qui suivent le mot clé `switch` indiquent une expression dont la valeur est testée successivement par chacun des `case`
- lorsque l'expression testée est égale à une des valeurs suivant un `case`, la liste d'instructions qui suit celui-ci est exécutée
- le mot clé `break` indique la sortie de la structure conditionnelle
- le mot clé `default` précède la liste d'instructions qui sera exécutée si l'expression n'est jamais égale à une des valeurs
- l'expression et les constantes doivent de type : `byte`, `short`, `int`, `char` ou `String`

Exemple

```
public static void imprimerMois(int mois){
  switch (mois) {
    case 1:  System.out.println("Janvier"); break;
    case 2:  System.out.println("Février"); break;
    case 3:  System.out.println("Mars"); break;
    case 4:  System.out.println("Avril"); break;
    case 5:  System.out.println("Mai"); break;
    case 6:  System.out.println("Juin"); break;
    case 7:  System.out.println("Juillet"); break;
    case 8:  System.out.println("Août"); break;
    case 9:  System.out.println("Septembre"); break;
    case 10: System.out.println("Octobre"); break;
    case 11: System.out.println("Novembre"); break;
    case 12: System.out.println("Décembre"); break;
    default: System.out.println("Mois invalide.");
break;
  }
}
```

Exemple :

```
public String eliminerAccents(String s){
  char tab[] = s.toCharArray();
  for (int i = 0; i<tab.length; i++){
    switch (tab[i]){
      case 'à' :
      case 'â' :
        tab[i] = 'a';
        break;
      case 'ç' :
        tab[i] = 'c';
        break;
      case 'é' :
      case 'ê' :
      case 'è' :
      case 'ë' :
        tab[i] = 'e';
        break;
      case 'î' :
      case 'ï' :
        tab[i] = 'i';
        break;
      case 'ô' :
        tab[i] = 'o';
        break;
      case 'ù' :
      case 'û' :
      case 'ü' :
        tab[i] = 'u';
```

```
        break;
      default :
    }
  }
  return new String(tab);
}
```

Remarques

- l'instruction `switch` n'ajoute pas de nouveau pouvoir d'expression par rapport à l'instruction `if else` imbriquée
- facile à lire

Comparons le dernier code avec le code suivant :

```
if (tab[i] == 'à' || tab[i] == 'â')
  tab[i] = 'a';
else if (tab[i] == 'ç')
  tab[i] = 'c';
else if (tab[i] == 'é' || tab[i] == 'è' || tab[i] == 'ê'
|| tab[i] == 'ë')
  tab[i] = 'e';
else
  ... ;
```

Instructions répétitives

Introduction

Une itération est un processus dans lequel une opération peut être effectuée plus d'une fois.

Exemples :

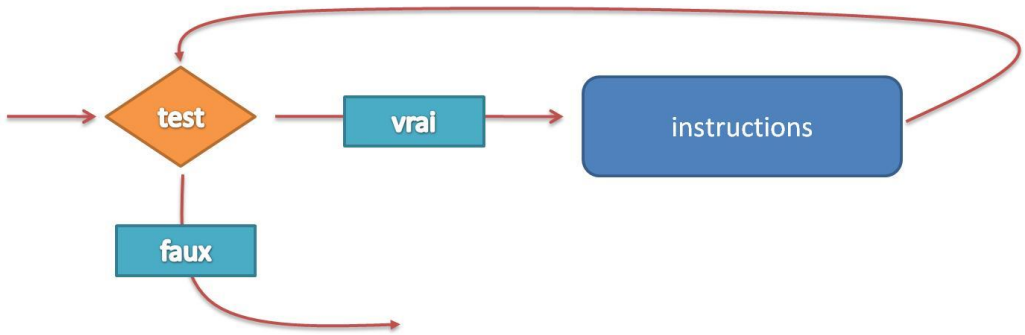
- l'envoi de plusieurs lettres; l'opération à répéter : "apposer un timbre"
- le calcul de la moyenne de plusieurs valeurs; nous devons d'abord calculer leur somme; l'opération à répéter : "additioner"

Combien de fois répéter une action ?

Pour arrêter l'itération :

- nous pouvons spécifier le nombre de répétitions
  exemple : "apposer le timbre" **5 fois**
- nous pouvons spécifier une condition d'arrêt
  exemple : tant que "**il fait froid**" "ajouter du bois de chauffage"
  la condition d'arrêt : il ne fait plus de froid

La boucle "while"



- si la condition *test* est vraie, on entre dans la boucle (une première fois ou à nouveau)
- si la condition *test* est fausse, l'exécution de la boucle est terminée
- dans la boucle, après avoir exécuté les instructions, on retourne à l'évaluation de la condition

Exemple :

```
"mesurer la température"
tant que "la température < 20 degrés"
  "ajouter du bois de chauffage"
  "attendre 3 minutes"
  "mesurer la température"
```

En Java :

```
float temp = Chauffage.getTemperature();
while (temp < 20) {
  Chauffage.ajouterBois();
  Thread.sleep(180000);
  temp = getTemperature()
}
```

Attention

- si l'évaluation de la condition *test* donne la valeur `true`, on bouclera infiniment
- une erreur souvent rencontrée
- pour éviter cette erreur, la valeur de la condition doit dépendre des actions exécutées dans la boucle

Exemple : calcul de la somme 1 + 2 + … + 100

Algorithme (pseudocode) :

```
somme = 0;
"choisir la première valeur"
```

```
while "il reste des valeurs à additionner"
  somme += valeur
  "choisir la valeur suivante"
```

Code Java

```
int somme = 0;
int valeur = 1;
final int derniere = 100;

while (valeur <= derniere) {
  somme += valeur;
  valeur++ ;
}
```

Exemple : calcul de la somme des entiers lus au clavier

Algorithme (pseudocode) :

```
somme = 0;
"lire la première valeur"
while "il reste des valeurs à additionner"
  somme += valeur
  "lire la valeur suivante"
"afficher la somme"
```

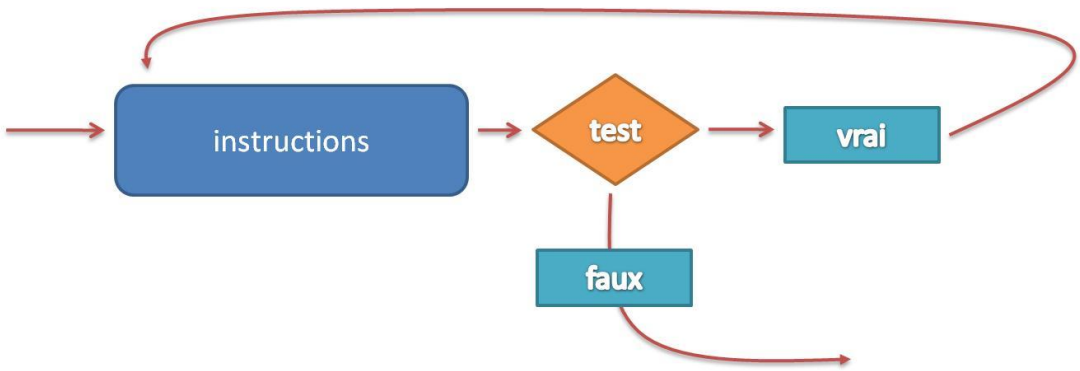
Code Java

```
// condition d'arrêt : la dernière valeur est suivie de
-1
int valeur, somme = 0;
BufferedReader b = new BufferedReader(new
InputStreamReader(System.in));

try {
  String s = b.readLine();        // lecture de la
  valeur = Integer.parseInt(s);  // prochaine valeur
  while (valeur != -1) {
    somme += valeur;
    s = b.readLine();               // lecture de la
    valeur = Integer.parseInt(s);  // prochaine valeur
  }
}
catch (IOException e){
}
System.out.println("somme = " + somme);
```

La boucle "do while"

On répète au moins une fois un groupe d'instructions; on arrête la boucle lorsque la condition est fausse,

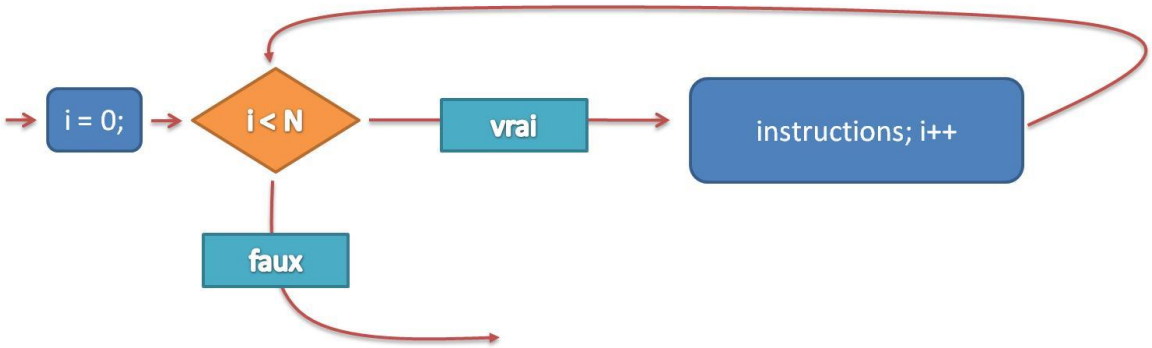


Exemple :

```
do {
  "faire un traitement";
  "évaluer les résultats";
} while "les résultats ne sont pas satisfaisants";
```

La boucle "for"

- un schéma fréquent de la boucle while



- la boucle est exécutée N fois
- la variable *i* prend les valeurs 0, 1, ..., N-1

Syntaxe

for (<instruction d'initialisation>; <condition de continuation>; <instruction de mise à jour>) <instruction à répéter>;

Problème : afficher les valeurs paires entre 0 et 100 (inclusivement)

Solution

```
for (int compteur = 0; compteur <= 100; compteur++)
```

```
  if (compteur % 2 == 0)
    System.out.println(compteur);
```

Une autre solution

```
for (int compteur = 0; compteur <= 100; compteur+=2)
  System.out.println(compteur);
```

Problème : afficher les valeurs paires entre 1 et 100 (inclusivement), 10 valeurs par ligne

Solution incluant du pseudo-code

```
for (int compteur = 1; compteur <= 100; compteur++){
  if (compteur % 2 == 0) {
    System.out.print(compteur + " ");
    if ("10 valeurs affichées sur cette ligne")
      System.out.println();
  }
}
```

Solution

```
for (int compteur = 1; compteur <= 100; compteur++){
  if (compteur % 2 == 0) {
    System.out.print(compteur + " ");
    if (compteur % 20 == 0)
      System.out.println();
  }
}
```

Bonne pratique de la programmation

Les deux programmes suivants bouclent infiniment :

```
int n = 0;
while (n < 10);
 System.out.println(n);
 n = n + 1;


int n = 0;
while (n < 10)
 System.out.println(n);
 n = n + 1;
```

La version correcte :

```
int n = 0;
```

```
while (n < 10) {
  System.out.println(n);
  n = n + 1;
}
```

Conseil :

- utilisez toujours des blocs !
- l'ajout de nouvelles lignes sera aussi plus facile
- faites attention à l'indentation

Ruptures de séquence

L'instruction "break"

`break` force la sortie d'un bloc, d'un choix (switch) ou d'une boucle (for, while, do).

`break` est surtout utilisé

- pour ne pas exécuter le `case` suivant dans une instruction `switch`
- pour arrêter un traitement itératif lorsqu'une erreur d'entrée ou une autre condition d'arrêt est rencontrée

Exemple :

```
while (true) {
  travailler(55); // 55 min.
  if (ilFaitBeau())
    break;
  prendrePause(5); // 5 min.
}
jouerAuGolf(240); // 4 heures
```

Exemple :

```
public static boolean contient(String s, char c){
  boolean trouve = false;
  for (int i = 0; i < s.length(); i++)
    if (s.charAt(i) == c){
      trouve = true;
      break;
    }
  if (trouve)
    System.out.println("Le caractère '" + c + "' a été
trouvé.");
  return trouve;
```

```
}
```

Exemple :

```
  // calcul de la valeur décimale d'une constante octale
  String cst = "17777777777";
  int val = 0;
  int i = 0;
  final int max = Integer.MAX_VALUE;
  do {
    int chiffre = Character.digit(cst.charAt(i), 8);
    if (chiffre > 7 || chiffre < 0)
      break; // seulement les chiffres octaux permis
    if (max / 8 < val)
      break; // valeur trop grande
    val = val * 8;
    if (max - chiffre < val)
      break; // valeur trop grande
    val = val + chiffre;
    i++;
  } while (i < cst.length());
  System.out.println("Les caractères acceptés
représentent la valeur " + val + ".");
```

Une manière plus intelligente d'effectuer la même tâche :

```
System.out.println(Integer.parseInt(cst, 8));
```

L'instruction "continue"

Dans une itération, l'instruction `continue` saute les instructions situées jusqu'à la fin de la boucle et l'exécution itérative reprend.

- l'instruction `continue` n'a de signification que dans une boucle : `for`, `while` ou `do`
- dans les boucles `while` et `do`, le contrôle saute à la vérification de la condition et pour les boucles `for`, le contrôle saute à la partie de mise à jour

Exemple :

```
String aChercher = "Un petit pâté péripatéticien paré
pour partir pâtir petit péripâté";
int max = aChercher.length();
int nbDeP = 0;

for (int i = 0; i < max; i++) {
  // intéressé seulement par la lettre p
  if (aChercher.charAt(i) != 'p')
```

```
    continue;
  // traiter le p
  nbDeP++;
}
System.out.println("Il y a " + nbDeP + " lettres p dans
cette chaîne.");
```

Blocs d'instructions

- les variables déclarées dans un bloc ne sont visibles que dans ce bloc

Exemple :

```
int x, y;
// ...
if (x > 0) {
  int min, max;
  min = Math.min(x, y);
  max = Math.max(x, y);
}
System.out.println("la valeur de " + min +
    " <= que la valeur de " + max);   // Erreur de
compilation
```

Portée des variables

- *la portée d'une variable indique sa durée de vie en terme d'endroit dans le programme où elle existe*
- une variable commence à exister à partir de sa déclaration et elle finit d'exister à la fin du bloc de code contenant sa déclaration
- on ne peut pas déclarer une variable locale si une variable ayant le même nom existe déjà dans la portée courante

```
  public static void test(int i){
    // variables existantes : i
    float x = 5;
    // variables existantes : i, x
    if (x >= 0.0) {
      // variables existantes : i, x
      int j = i * 10;
      // variables existantes : i, j, x
    }
    else {
      // variables existantes : i, x
      int j = i + 20;
      // variables existantes : i, j, x
```

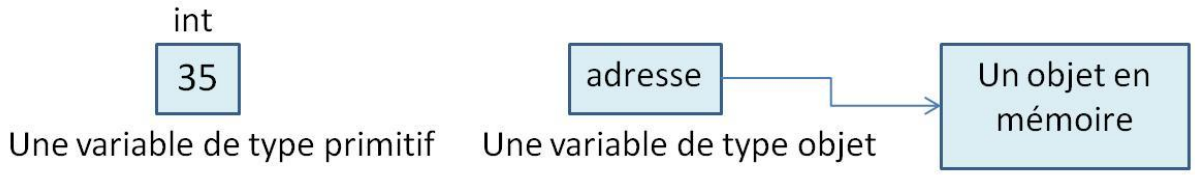
```
      float x;     // erreur de compilation : Duplicate
local variable x
    }
    // variables existantes : i, x
  }
```

POO : un peu plus

Les constructeurs

L'introduction

- deux types de données en Java: les données de type primitif et les données de type objet
- deux types de variables : les variables de type primitif et les variables de type objet
- les variables de type primitif contiennent des données primitives : des caractères, des entiers, des réels ou des booléens
- les variables de type objet contiennent des références vers des objets



- une variable de type primitif contient toujours une certaine valeur;
  une variable de type objet contient la valeur `null` après sa déclaration;
  l'objet est vide, il n'existe pas, il doit être créé

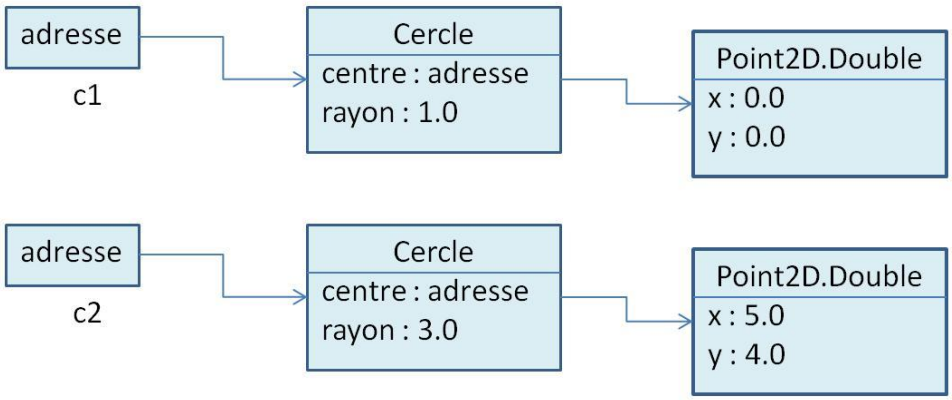
Caractéristiques de constructeurs

Un constructeur est une méthode d'une classe donnée, servant à créer des objets.

- il a le même nom que la classe
- un constructeur n'a pas de type de retour (mais on ne met pas de void!)
- son rôle principal : l'allocation de la mémoire et l'initialisation des attributs de l'objet créé
- un constructeur crée (instancie) un objet en appliquant l'opérateur `new`

Exemple :

```
Cercle c1 = new Cercle();
Cercle c2 = new Cercle(new Point2D.Double(5,4), 3);
```



- deux objets cercle sont créés et les références vers ces objets sont stockées dans c1 et c2
- les constructeurs acceptent la surcharge
- si aucun constructeur n'est spécifié dans la définition de la classe, un constructeur sans paramètre est fourni par Java
- si vous définissez au moins un constructeur, le constructeur par défaut n'est plus fourni

Les destructeurs

La machine virtuelle Java se charge de repérer les objets inutiles et de libérer la mémoire inaccessible (faire le ramasse-miettes).

Il est aussi possible de détruire un objet explicitement avec la méthode `finalize()`.

Exemple :

```
class Cercle {
    //...

    public void finalize() {
        System.out.println("Objet Cercle détruit");
    }
    //...
}
```

Les mots-clés this et super

Les mots-clés `this` et `super` désignent respectivement des références sur l'instance courante et sur la classe mère.

La surcharge

Redéfinition des champs

- les champs déclarés dans la classe dérivée sont toujours des champs supplémentaires
- si l'on définit un champ ayant le même nom qu'un champ de la classe de base, il existera deux champs de même nom et le nom de champ désignera celui déclaré dans la classe dérivée
- pour avoir accès au champ de la classe de base, il faut
  - changer le type de la référence pointant sur l'objet, ou
  - utiliser `super`

Exemple :

```
class X {
  public int i;
}
class Y extends X {
  public int i;
  public void methode(){
    i = 0;              // champ défini dans Y
    this.i = 0;         // champ défini dans Y
    super.i = 1;        // champ défini dans X
    ((X)this).i = 1;    // champ défini dans X
  }
}
```

Redéfinition des méthodes (polymorphisme d'héritage)

- une classe dérivée peut fournir un nouveau comportement d'une méthode
- la méthode redéfinie a la même signature (le même nombre d'arguments et les mêmes types des arguments) que la classe mère

Exemple :

```
public class Rectangle {
  double a, b;
  double perimetre(){
    return 2*a + 2*b;
  }
  public String toString(){
    return "Rectangle(" + a + "," + b + ")";
  }
}

class Carre extends Rectangle {
  // a = b
  double perimetre(){
    return 4*a;
```

```
  }

  public String toString(){
    return "Carre(" + a + ")";
  }

}
```

Généricité (polymorphisme paramétrique)

- permet de définir plusieurs fois une même méthode avec des arguments différents
- le compilateur choisit la méthode qui doit être appelée en fonction du nombre et du type des arguments

Exemple :

```
// calculer la somme des entiers entre premier et
dernier
int somme(int premier, int dernier){
  ...
}
// calculer la somme des réels entre premier et dernier
avec delta étant 0.1
double somme(double premier, double dernier){
  ...
}
//calculer la somme des réels entre premier et dernier
avec delta donné
double somme(double premier, double delta, double
dernier){
  ...
}
```

Polymorphisme ad hoc

- permet d'avoir des fonctions de même nom, avec des fonctionnalités similaires, dans des classes sans aucun rapport entre elles
- exemple : *afficher* dans les classes *Beaute*, *Message* et *Amour*

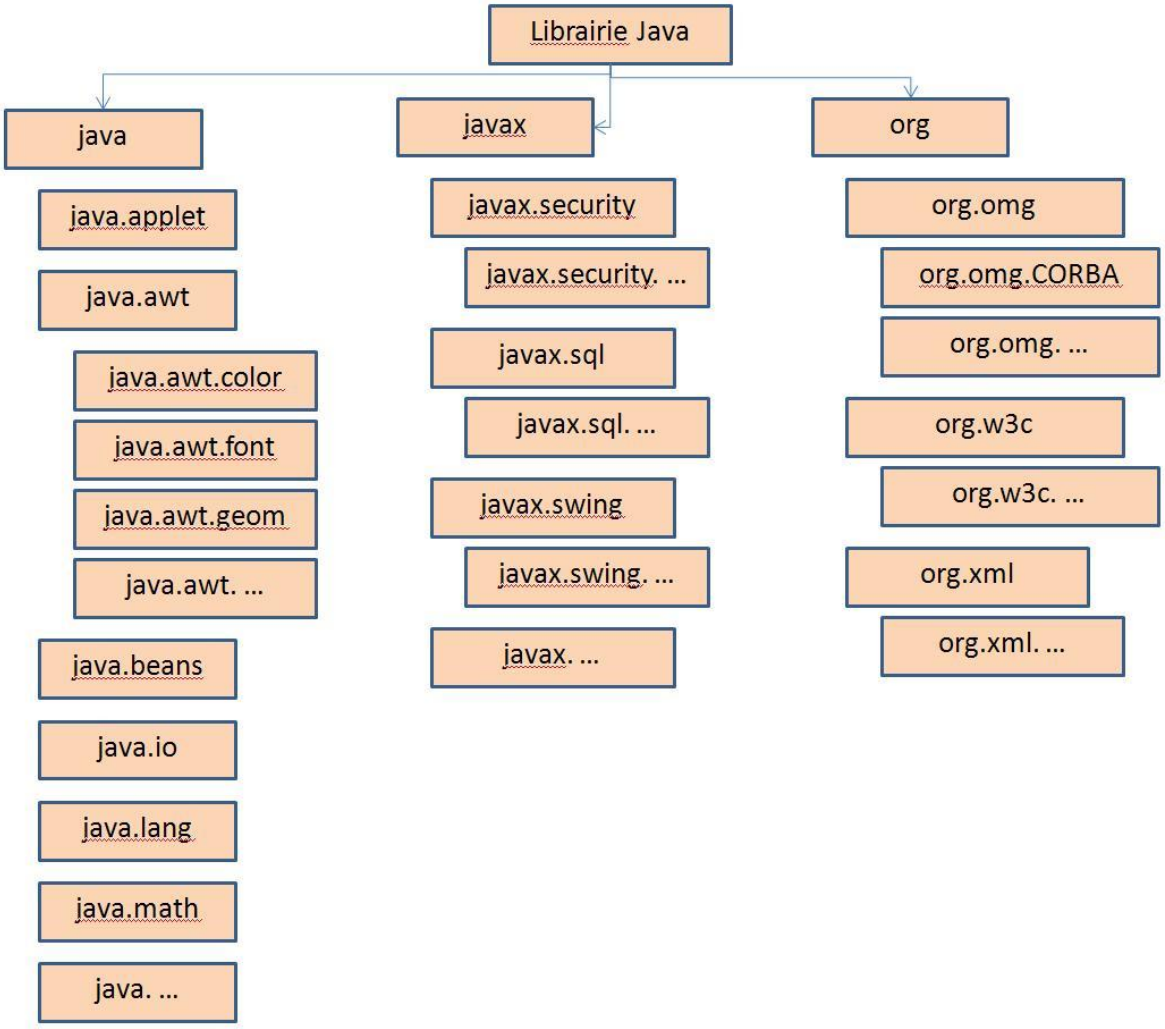
Note

Le nom de polymorphisme vient du grec et signifie *qui peut prendre plusieurs formes*.

Les packages

Un package est regroupement de classes autour d'une fonctionnalité précise et commune.

- une multitude de classes en Java
- les packages servent à structurer l'ensemble des classes
- ils facilitent la recherche de l'emplacement physique des classes
- ils rendent très improbable qu'il y ait confusion entre des classes de même nom
- ils présentent l'ensemble des classes selon une structure arborescente
- si vous organisez vos classes sous la forme de plusieurs packages, vous devez placer les classes compilées dans des répertoires dont la structure reflète la structure arborescente des packages;
  un package portant le nom MonPackage doit être stocké dans un répertoire du même nom
- Pour pouvoir accèder aux classes d'un package à partir d'une classe qui ne fait pas partie du package, on importe les classes du package.
- la **libraire standard Java** venant avec le compilateur a aussi une structure arborescente



- le compilateur connait l'endroit où se trouve cette librairie
- pour que le compilateur puisse trouver vos packages, il doit connaître l'emplacement des packages; il utilise une variable d'environnement appelée *classpath* donnant la liste des chemins d'accès aux classes

Tableaux et vecteurs

Introduction

Nous voulons avoir des objets correspondant aux mois.
Devons-nous les créer de la manière suivante ?

```
Mois jan; Mois fevrier; Mois mars; Mois avril; ... Mois
decembre;
```

Le problème serait encore plus difficile pour traiter les objets dont la quantité est plus élevée.

Qu'est-ce qu'un tableau ?

- un tableau est une structure composée d'une séquence contiguë de variables, **toutes de même type**
- chacune des variables est stockée dans le tableau à une position bien précise appelée *indice* (ou index)
- la première position dans le tableau est celle d'indice 0
- la taille du tableau est déterminée lors de sa création
- en Java, les tableaux sont des objets; pour créer un nouveau tableau on utilise l'opérateur `new`
- chaque objet tableau a une variable d'instance appelé `length` qui contient la taille du tableau;
  il est impossible de modifir la valeur de cette variable, elle fixée lors de l'instanciation du tableau
- la dernière position dans le tableau est `length - 1`

Exemple :

```
int tab[];
tab = new int[6]; // un tableau de 6 entiers (un entier
prend 4 octets en mémoire)
```

Indice :	0	1	2	3	4	5	
Éléments :	élément 1	élément 2	élément 3	élément 4	élément 5	élément 6	
Octets :	0	4	8	12	16	20	24

Déclaration d'un tableau

```
type[] nomDuTableau;
nomDuTableau = new type[combien];
```

où

- *type* est le type de chaque élément
- *combien* est une expression de type *entier* qui détermine le nombre de cellules dans le tableau
- nomDuTableau est une *référence* sur l'espace mémoire occupé par le tableau

Initialisation

- la valeur initiale de chaque cellule est `0` pour les nombres, le caractère `(char)0` pour les caractères et `null` pour les objets
- on peut initialiser un tableau avec des valeurs initiales :

```
type[] nomDuTableau = { valeur_1, valeur_2, ...,
valeur_N };
```

Utilisation d'un tableau

- l'opérateur d'indexation de tableau ([]) permet d'accéder à un élément du tableau;
  il est utilisé pour accéder à une valeur d'un élément ou pour y en affecter une autre

  Exemple :

  le code

```
int a[] = new int[3];
a[1] = 11;
a[0] = 3;
a[2] = a[1] + 4;
System.out.println(a[2]);
```

va afficher la valeur 15

- l'opérateur d'indexation de tableau vérifie automatiquement si l'indice est correct;
  si l'indice n'est pas correct (< 0 ou >= `length`), une erreur est signalée;
  plus exactement, l'exception `ArrayIndexOutOfBoundsException` est lancée
- attention : pour accéder au *i-ème* élément du tableau, nous utilisons l'indice *i - 1*

Exemple :

```
// On crée et on affiche un tableau des
// 100 premières valeurs entières élevées au carré
int [] carres = new int[100];

for (int valeur = 0; valeur < 100; valeur++) {
  int carre = valeur * valeur;
  carres[valeur] = carre;
}

for (int valeur = 0; valeur < 100; valeur++) {
  System.out.println(valeur + "² = " + carres[valeur]);
}
```

Exemple :

La fonction `main` affiche la liste des arguments du programme fournis sur la ligne de commande.

```
public class TesterArguments  {
  public static void main(String [] args) {

    // On affiche les arguments de la ligne de commande
(s'il y en a).
    int longueur = args.length;

    if (longueur > 0) {
      System.out.println("Voici la liste des arguments
de la commande :");
      for (int i = 0; i < longueur; i++) {
        System.out.println(args[i]);
      }
    }
  }
}
```

L'exécution du programme à partir de la console avec la commande

```
java TesterArguments INF1563 INF4023 INF4123 MAT1153
GEN1623
```

produira les résultats suivants :

```
Voici la liste des arguments de la commande :
INF1563
INF4023
INF4123
MAT1153
GEN1623
```

Passage d'un tableau en paramètre

- un tableau est un objet
- si l'on passe un tableau en paramètre, on passe une copie de la référence vers ce tableau

Exemple :

```
int tab[] = new int[3];
// ...
imprimer(tab);
// ...
void imprimer(int[] tableau) {
  System.out.print (tableau[0] + " ");
  System.out.print (tableau[1] + " ");
  System.out.print (tableau[2]);
}
```

Avantages

- le temps d'accès à un élément par son indice est constant

Limites

- un tableau est représenté en mémoire sous la forme de cellules contiguës; les opérations d'insertion et de suppression d'élément sont impossibles, sauf si on crée un nouveau tableau et cela implique plusieurs opérations

Tableaux d'objets

- le type de chaque élément d'un tableau peut être n'importe quoi, incluant un objet
- un tableau d'objets est en fait un tableau de références sur des objets

- lors de l'instanciation du tableau, toutes les références sont initialisées à null

Exemple :

```
Etudiant[] etuds = new Etudiant[ N ];
for (int i = 0; i < etuds.length; i++)
  etuds[i].enregistrer();
```

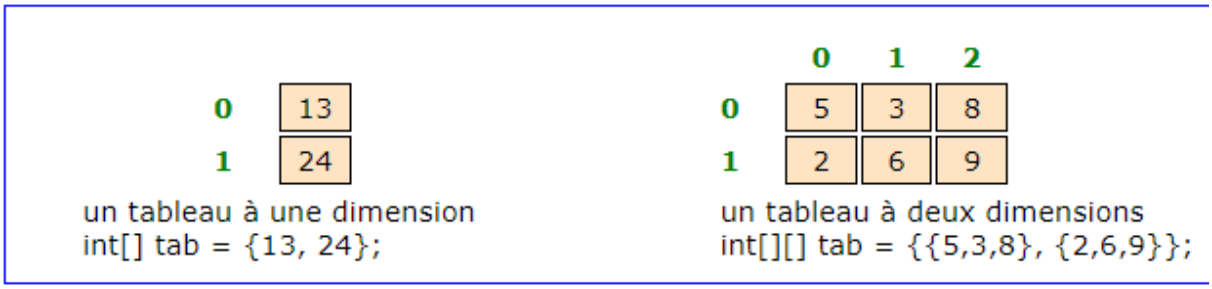
etuds[i] == null pour tout i; **erreur d'exécution !**

Une solution correcte serait :

```
Etudiant[] etuds = new Etudiant[ N ];
for (int i = 0; i < etuds.length; i++)
  etuds[i] = new Etudiant();
for (int i = 0; i < etuds.length; i++)
  etuds[i].enregistrer();
```

Tableaux multi-dimensionnels

- le type de chaque élément d'un tableau peut être n'importe quoi incluant un type tableau
- un tableau de tableaux est appelé un *tableau à deux dimensions*



- un tableau de tableaux de tableaux est appelé un *tableau à trois dimensions*
- pour déclarer un tableau à deux dimensions, on utilise la notation suivante:
  type_des_elements[][] nom;

Exemple :

```
int[][] mat = new int[10][5];
for (int i=0; i<mat.length; i++)
  for (int j=0; j<mat[0].length; j++){
    mat[i][j] = i+j;  // une valeur quelconque
    System.out.println(mat[i][j]);
  }
```

Tableaux inégaux

Il est aussi possible de créer un tableau à deux dimensions dont chaque ligne ne contient pas le même nombre de colonnes :

```
int[][] tab = {{1, 2}, {3, 4, 5}, {6}};
```

Faites attention au traitement d'un tel tableau !

Traitement de chaînes de caractères

Particularités de la classe String

- les chaines de caractères sont, en Java, des objets de type String
- le seul type non primitif pour lequel il existe des littéraux
  exemples : "abc", "Abcd", "", "a\nb\tc", "ab\"c"\"d"
- une méthode spéciale de création d'un objet; l'instruction

  ```
  String str = "abc";
  ```

  crée un nouvel objet de type String avec les caractères 'a', 'b', 'c'

- opérateur de concaténation "+"

  ```
  String str = "abc";
  str = "di" + "manche"; // str == "dimanche"
  ```

Séquences d'échappement, Unicode

Séquence	Signification
\t	tabulation
\n	nouvelle ligne
\r	retour de chariot
\"	guillement

\'	apostrophe
\\	backslash
\u????	code Unicode

Tous les caractères en Java sont codés en **Unicode** et ils ont une valeur comprise entre 0 et 65535. Pour insérer un caractère quelconque dont le numéro Unicode est connu, on utilise la séquence d'échappement \u suivi du numéro du caractère en hexadécimal sur quatre chiffres.

Exemple :

```
System.out.println("Micha\u0322");
```

Résultats :

```
Michał
```

Création d'un objet *String*

- création d'une chaîne de caractères à partir d'un tableau de caractères :

```
char[] s = { 'I', 'N', 'F', '1', '5', '6', '3' };
String sigle = new String(s);
```

- création d'une chaîne de caractères vide :

```
String vide = new String();  // équivalent à String vide = "";
```

- création d'une copie d'une chaîne de caractères :

```
String sigle2 = new String(sigle);
```

Méthode *toString*

- la méthode toString est applicable à tous les objets
- ceci permet d'appeler `System.out.print` avec n'importe quel objet
- la méthode peut être redéfinie

Exemple :

```
import java.util.Date;
```

```
import java.awt.*;
public class Exemple {

  public static void main(String[] args) {
    Date d = new Date();
    System.out.println(d);  // toString redéfinie dans
la classe Date
    Frame f = new Frame();
    System.out.println(f);  // toString redéfinie dans
une classe parent de Frame
    Exemple e = new Exemple();
    System.out.println(e);  // on utilise
Object.toString()
  }
}
```

Résultats :

```
Fri Aug 07 14:37:03 EDT 2009
java.awt.Frame[frame0,0,0,0x0,invalid,hidden,layout=java
.awt.BorderLayout,title=,resizable,normal]
Exemple@173a10f
```

Méthode *valueOf*

- la méthode `valueOf` transforme l'argument étant une valeur de type primitif en une chaîne de caractères.
- observez que la méthode toString s'applique aux objets

Exemple :

```
String annee = String.valueOf(2009);
System.out.println("L'année " + annee + " s'écrit MMIX
en chiffres romains.");
System.out.println("L'année " + 2009 + " s'écrit MMIX en
chiffres romains.");
```

Résultats :

```
L'année 2009 s'écrit MMIX en chiffres romains.
L'année 2009 s'écrit MMIX en chiffres romains.
```

Les String sont immuables

Attention : la classe String est immuable, la chaine originale n'est jamais modifiée.

Exemple :

```
String s1 = "ABc";
String s2 = s1.toLowerCase();
System.out.println(s1);
System.out.println(s2);
```

Résultats :

```
ABc
abc
```

Quelques méthodes de la classe String

Informations sur l'état d'un String	
length()	retourne la longueur de la chaîne
charAt(i)	retourne le i^e caractère de la chaîne; i doit être entre 0 et length() - 1
equals(String)	indique si l'objet est égal à l'argument
compareTo(String)	retourne 0 si si l'objet est égal à l'argument, un nombre négatif s'il est plus petit dans l'ordre lexicographique et un nombre positif s'il est plus grand
Opérations qui créent une nouvelle chaîne	
toLowerCase()	change toutes les lettres en minuscules
trim()	supprime tous les caractères blancs (espace, tabulation, saut de ligne) se trouvant au début et à la fin de la chaine
replace(char c1, char c2)	remplace tous les caractères c1 par c2
replace(String s1, String s2)	remplace toutes les chaînes s1 par s2
substring(int debut, int fin)	extrait une sous-chaîne qui commence à l'indice *debut* et qui se termine à l'indice *fin* - 1

Nombres à virgule flottante

- les nombres à virgule flottante sont des approximations de nombres réels
- ils possèdent un signe *s* (-1 lub 1}), une mantisse *m* et un exposant *e* le triplet (*s,m,e*) représente la valeur réelle $s*m*2^e$
- en faisant varier *e*, on fait « flotter » la virgule décimale

Erreurs d'arrondi

- l'utilisation des types `float` et `double` cause souvent une perte de précision
- la taille de la mantisse est limitée (23 bits pour le type float et 52 bits pour le type double); il arrive fréquemment que la perte de chiffres significatifs cause des imprécisions et même des erreurs dans les programmes

Exemple :

```
double racineCarre = Math.sqrt(2);
double zero = racineCarre * racineCarre - 2;
if (zero == 0)
  System.out.println("sqrt(2) * sqrt(2) moins 2 est 0");
else
  System.out.println("sqrt(2) * sqrt(2) moins 2 n'est
pas 0 mais " + zero);
```

Le résultat

```
sqrt(2) * sqrt(2) moins 2 n'est pas 0 mais
4.4408920985006262E-16
```

- n'utilisez jamais l'opérateur '==' pour les nombres en point flottant
- vérifiez plutôt si les deux valeurs à comparer sont proches

Exemple :

```
final double EPS = 1E-10;
if (Math.abs(zero) <= EPS)
  System.out.println("sqrt(2) * sqrt(2) moins 2 est
très proche du 0");
else
  System.out.println("sqrt(2) * sqrt(2) moins 2 est
loin du 0 (" + zero + ")");
```

Le résultat

```
sqrt(2) * sqrt(2) moins 2 est très proche du 0
```

NaN

- `NaN` (« not a number ») est le résultat de la tentative de division flottante par zéro, ou de la racine carrée d'un nombre strictement négatif
- les NaN se propagent : les opérations qui reçoivent un NaN comme argument donnent un NaN comme résultat

Exemple :

```
// exemple de "NaN":
System.out.print("1.0 / 0.0 * 2 est \"Not-a-Number\" :
");
double d = 1.0 / 0.0 * 2;
System.out.println(d);
```

Résultat :

```
1.0 / 0.0 * 2 est "Not-a-Number" : NaN
```

Traitement des exceptions

Introduction

- plusieurs types d'erreurs !
- mais c'est rare que suite à une détection d'une erreur, l'exécution doit être terminée
- des *erreurs fatales* (non récupérables) : le programme s'arrête à la suite de ce type de situation
- des *erreurs récupérables* : le programme exécute une action corrective et poursuit son exécution

  une erreur corrigible = un signal indiquant qu'une situation anormale a eu lieu

  exemples : division par zéro, une erreur syntaxique dans un programme, une mauvaise commande dans un système d'exploitation, l'absence d'un fichier, mémoire insuffisante

- **attention** : il n'est pas nécessaire que la gestion d'une exception se trouve dans la méthode qui est susceptible de déclencher cette exception; une méthode peut ignorer la gestion d'une exception à condition qu'elle transmette l'exception à la méthode appelante

Traitement d'erreurs en Java

- Java supporte les deux types d'erreurs :
  - les erreurs fatales sont représentées par des objets de la classe `java.lang.Error`

    - les erreurs récupérables sont appelées des exceptions et sont représentées par des objets de la classe `java.lang.Exception`
- hiérarchie des classes
    - Object
        - Throwable
            - Error
                - AssertionError
                - ...
            - Exception
                - RuntimeException
                    - NullPointerException
                    - IndexOutOfBoundsException
                        - ArrayIndexOutOfBoundsException
                        - StringIndexOutOfBoundsException
                    - ArithmeticException
                    - MonException1
                - IOException
                    - EOFException
                    - FileNotFoundException
                - ...
                - MonException2
        - ...

Traitement d'exceptions

- lorsqu'une situation anormale est détectée, une exception est lancée
- si cette exception n'est pas attrapée dans le bloc de code où elle a été lancée, elle est propagée au niveau du bloc englobant
- si ce dernier ne l'attrape pas, elle est transmise au bloc de niveau supérieur et ainsi de suite
- si l'exception n'est pas attrapée dans la méthode qui l'a lancée, elle est propagée dans la méthode qui a invoqué cette dernière
- si la structure de bloc de la méthode d'invocation ne contient aucune instruction attrapant l'exception, celle-ci est à nouveau propagée vers la méthode de niveau supérieur
- si une exception n'est jamais attrapée (incluant la méthode main()), un message d'erreur et le contenu de la pile des appels sont affichés et le programme s'arrête

Techniquement, les exceptions sont traitées à l'aide de la construction

```
try {
  instructions
}
catch (class d'exception e) {
  instructions
}
...
catch (class d'exception e) {
```

```
  instructions
}
```

- `try { .. }` détermine une séquence d'instructions susceptibles de déclencher une exception
- `catch (class d'exception e) { ... }` attrappe les exceptions de type indiqué et exécute les actions nécessaires pour traiter la situation qui a déclenché l'exception attrapée
- si une instruction du bloc try déclenche une exception
  - les instructions du bloc `try` qui la suivent ne sont pas exécutées
  - les instructions du bloc `catch` son exécutées
  - le programme reprend son cours normalement avec l'instruction suivant le bloc `catch`

Exemple (version 1) :

But : calculer le double des valeurs entières passées sur la ligne de commande.

```
public class Exemple {

  public static void main(String args[]) {
    for (int i=0; i<args.length; i++){
      System.out.println(Integer.parseInt(args[i]) * 2);
    }
  }
}
```

java Exemple 12 45 3a 78

```
Exception in thread "main"
java.lang.NumberFormatException: For input string: "3a"
  at
java.lang.NumberFormatException.forInputString(NumberFor
matException.java:48)
  at java.lang.Integer.parseInt(Integer.java:458)
  at java.lang.Integer.parseInt(Integer.java:499)
  at Exemple.main(Exemple.java:7)
24
90
```

Exemple (version 2) :

```
public class Exemple {

  public static void main(String args[]) {
    try {
      for (int i=0; i<args.length; i++){
```

```
        System.out.println(Integer.parseInt(args[i]) *
2);
      }
    }
    catch (NumberFormatException e){
      System.out.println(1);
    }
  }
}
```

java Exemple 12 45 3a 78

```
24
90
1
```

Les exceptions de la classe RuntimeException

- la classe mère des exceptions qui peuvent être déclenchées au cours de l'exécution d'un programme
- une méthode ne doit pas nécessairement déclarer dans sa clause `throws` toute sous-classe de RuntimeException qui pourrait être déclenchée durant l'exécution de la méthode mais ne pas être attrapée
- l'exception NumberFormatException est une sous-classe de RuntimeException

Définition de nouveaux types d'exceptions

- si l'on veut signaler une situation exceptionnelle, non prévue par l'API de Java, il faut dériver la classe Exception et définir une nouvelle classe
- cette classe contient en général une redéfinition de la méthode toString
- lors du lancement d'une telle exception avec l'instruction `throw`, on crée une instance de cette nouvelle classe
- la méthode susceptible de lance l'exception x qui n'est pas une sous-classe de RuntimeException, doit avoir la clause `throws x` à la fin de sa signature

Exemple :

```
class NoteException extends Exception {
  public String toString() {
    return("Une note trop basse !\n");
  }
}
```

Exemple :

```
public class NoteException extends Exception {
  int noChapitre;
  public String toString() {
```

```
    return("Une note trop basse pour le quiz " +
noChapitre + " !\n");
  }
}

public class Quiz {
  int note;
  ...
}

public class Cours {

  static Quiz q[] = new Quiz[10];

  void verifierQuiz() throws NoteException {
    for (int i=0; i<q.length; i++){
      if (q[i].note < 1)
        throw new NoteException();
      else
        System.out.println("la note pour le quiz " + i +
" est " + q[i].note);
    }
  }

  void reprendreChapitre(int no){
    ...
  }

  public static void main(String args[]) {

    Cours inf1563 = new Cours();
    ...
    try {
      inf1563.verifierQuiz();
    }
    catch (NoteException e){
      System.out.println(e.toString());
      inf1563.reprendreChapitre(e.noChapitre);
    }
  }
}
```

Remarque : Toute méthode susceptible de déclencher une exception n'étant pas une sous-classe de RuntimeException doit

- soit l'attraper
- soit la déclarer explicitement avec la clause `throws`

Une mauvaise pratique

```
try {
  ...
}
catch (Exception e) {
}
```

le bloc "finally"

- les clauses `catch` sont suivies optionnellement par un bloc `finally`

```
...
try {
  // ouvrir un fichier et effectuer certains traitements
  // susceptibles de déclencher l'exception IOException
}
catch (IOException e) {
  // traiter l'exception
}
catch (Exception e){
  // traiter d'autres exceptions si c'est le cas
}
finally {
  //fermer le fichier
}
```

- le bloc `finally` contient du code qui sera exécuté quelle que soit la manière dont le bloc try a été quitté
- son exécution est garantie peu importe si
  - le bloc `try` s'est exécuté normalement sans aucune exception déclenchée
  - le bloc `try` déclenche une exception attrapée par l'un des blocs `catch`
  - le bloc `try` déclenche une exception qui n'est attrapée par aucun des blocs `catch` qui le suivent

Documentation de programmes

Pourquoi documenter le code ?

- si vous documentez votre code, vous comprendrez mieux ce que vous avez écrit
- votre code sera sans doute lu ou modifié par d'autres
- la doc réduit le coût de la maintenance du code
- le code et sa documentation ne peuvent pas être séparés (autrement, il serait difficile d'assumer leur concordance)

Documentation de l'interface

- des commentaires utilisés pour donner information aux usagers du programme; ceux derniers ont seulement besoin de l'information concernant l'usage de la classe - son interface.
- Javadoc est un outil développé par Sun Microsystems qui permet, en inspectant le code Java des classes, de produire une documentation d'API de votre code
- inclus dans tous les JDK (ou SDK) de Java
- la documentation générée par Javadoc est appelée souvent javadoc
- l'outil génère des pages HTML contenant au minimum les listes des classes, des méthodes et des variables publiques
- les **doclets** permettent d'exporter la doc JavaDoc en différents formats, tels que PDF, XML, DocBook, LaTeX, etc

Les commentaires Javadoc

- chaque commentaire java commençant par /** sera intégré dans la documentation du code source
- ces commentaires utilisent des tags Javadoc qui commencent par un @
- le tableau suivant résume les tags utilisés :

TagDescription	
@author	nom du développeur
@deprecated	marque la méthode comme dépréciée
@exception	documente une exception lancée par une méthode
@param	définit un paramètre de méthode
@return	documente la valeur de retour; ce tag ne devrait pas être employé pour des méthodes définies avec un type de retour void
@see	documente une association à une autre méthode ou classe
@since	précise à partir de quelle version une méthode existe dans la classe

@throws	documente une exception lancée par une méthode; un synonyme pour @exception
@version	définit la version d'une classe ou d'une méthode

Documentation de l'implémentation

Cette doc est destinée aux programmeurs qui vont modifier le fichier dans l'avenir et qui ont besoin d'une connaissance des détails de l'implémentation.

- expliquer le choix technique effectué
- pourquoi tel algorithme et pas un autre
- problèmes à corriger
- ...

Exemple :

```
private int nbr; // nombre de lignes dans le fichier
...
// la méthode de tri ci-dessous doit être améliorée
...
/* le code suivant est basé sur l'algorithme de Knuth;
   pour plus de détails consulter le livre The Art of
   Computer Programming, vol. 4A */
...
```

Méthodes abstraites et interfaces

Qu'est ce qu'une interface ?

- un ensemble des méthodes publiques à travers lesquelles on peut interagir avec un objet
  (un ensemble de services visibles depuis l'extérieur)
- exemple : une interface télécommande de télévision
  diverses méthodes publiques : augmenter ou diminuer le son, monter ou descendre de chaîne
- une interface est définie dans un fichier séparé avec le mot réservé `interface`
- une interface contient des déclarations de constantes et de méthodes *abstraites*
- une méthode abstraite est une méthode sans corps (sans implémentation)

Exemple :

```
abstract float perimetre();
abstract float surface();
```

- une classe est abstraite dès qu'elle contient une méthode abstraite et elle doit être déclarée `abstract`
- une classe abstraite ne peut pas être instanciée
  une classe abstraite est une classe qui contient des méthodes qui n'ont pas été implémentées; elle sert avant tout à factoriser du code

```
import java.awt.Color;

abstract class Forme {
  Color fond = Color.black;

  abstract float perimetre(); // méthode abstraite

  abstract float surface();   // méthode abstraite

  void colorer(Color c) {
    fond = c;
  }
}
```

- une sous-classe d'une classe abstraite sera encore abstraite si elle ne définit pas toutes les méthodes abstraites dont elle hérite

```
abstract class Polygone extends Forme {

}
```

- les différences entre une interface et une classe abstraite :
  - les classes abstraites servent à factoriser du code, tandis que les interfaces servent à définir des contrats de service
  - toutes les méthodes d'une interface sont abstraites alors qu'une classe abstraite peut avoir à la fois des méthodes non abstraites et des méthodes abstraites
  - dans une interface
    - toutes les méthodes d'une interface sont publiques; les mots clés `public` et `abstract` sont implicites et n'apparaissent pas
    - aucune méthode n'est `static`
    - tous les champs sont `public`, `static` et `final`; les mots clés `static` et `final` sont implicites
    - il n'y a pas de constructeur
    - le seul qualificatif est `public`; si l'on l'utilise, l'interface peut être utilisé par n'importe quelle classe; sinon, elle ne peut être utilisée que par les classes du même package

- contrairement aux classes, une interface peut dériver plusieurs autres interfaces

Comment implémenter des interfaces ?

```
interface Service {
  ...
}
class X implements Service {
  ...
}
```

- par l'utilisation du mot `implements`, la classe promet d'implémenter toutes les méthodes déclarées dans l'interface
- la signature d'une méthode implémentée doit être identique à celle qui apparait dans l'interface;
  dans le cas contraire, la méthode est considérée comme une méthode de la classe et non de l'interface