Chapitre 2
Classes et Objets

Plan

- Encapsulation :
  - regrouper données et actions
  - cacher les données
- Les données en Java
- this
- static
- Les tableaux
- Les packages
- Contrôle d'accès

Encapsulation

- Le terme Encapsulation possède deux acceptions dans le monde objet.
  1. Il signifie le fait de regrouper dans une entité appelée classe données et actions.
  2. Il signifie le fait de cacher les données (l'implémentation) d'une classe.

Regrouper données et actions

Regrouper données et actions

- Il s'agit du fondement de l'approche objet.
- Avantages :
  - meilleure structuration (organisation)
  - mise en place possible de l'héritage
- L'approche procédurale ne bénéficie pas de ces avantages.

Illustration

- On souhaite écrire une application qui permette de gérer une bibliothèque.
- Dans une version simpliste, on devra pouvoir:
  - ajouter un livre dans une bibliothèque
  - afficher les informations d'un livre
- Comparons l'approche objet à l'approche procédurale.

Approche procédurale

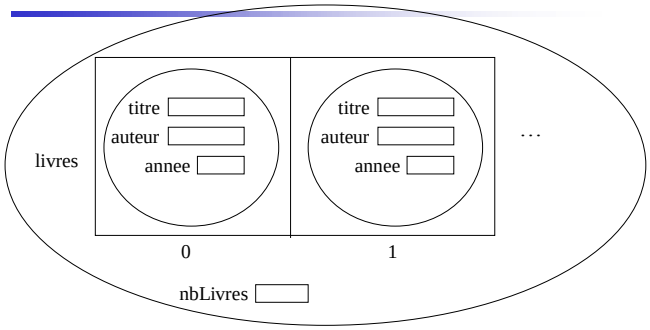
- Cette approche consiste à définir :
  - d'une part les structures de données
  - d'autre part, les actions permettant de manipuler ces structures
- Données et actions sont définies séparément.

Types de données

```
type Livre = structure
    titre: chaîne (de caractères)
    auteur : chaîne (de caractères)
    annee : entier
fin structure
```

```
type Bibliothèque = structure
    livres : tableau de MAX Livre
    nbLivres : entier
fin structure
```

Structures des données



Action de saisie de livre

```
action saisir(out livre : Livre)
début
   ecrire("titre : ")      lire(livre.titre)
   ecrire("auteur : ")     lire(livre.auteur)
   ecrire("annee : ")      lire(livre.annee)
fin
```


Action d'affichage de livre

```
action afficher(in livre : Livre)
début
   ecrire("titre : ",livre.titre)
   ecrire("auteur : ",livre.auteur)
   ecrire("annee : ",livre.annee)
fin
```


Action d'ajout de livre

```
action ajouterLivre(in livre : Livre,
                    in/out bib : Bibliotheque)
début
   bib.livres[bib.nbLivres]=livre;
   bib.nbLivres=bib.nbLivres+1;
fin
```


Action d'affichage de bibliothèque

```
action afficher(in bib : Bibliotheque)
début
   pour i variant de 0 à bib.nbLivres-1 faire
      afficher(bib.livres[i])
   fin pour
Fin
```


Action de recherche de livre

```
action rechercherLivre(in titre : chaîne,
                       in bib : Bibliotheque)
      retourne Livre
début
    pour i variant de 0 à bib.nbLivres-1 faire
      si bib.livres[i].titre = titre alors
            retourner bib.livres[i]
      fin si
   fin pour
   retourner null
fin
```


Surchage

- Nous avons défini deux actions portant le même nom mais de signature différente :
  - action afficher(in livre : Livre)
  - action afficher(in bib : Bibliotheque)
- On désigne cette possibilité sous le terme de surchage.
- Tous les langages n'autorisent pas la surchage.

Approche objet

- Cette approche consiste à définir des classes qui sont des entités regroupant structures de données et actions associées.
- Une classe est décrite par :
  - un ensemble de champs (également appelés attributs ou variables d'instance)
  - un ensemble d'actions (également appelées méthodes ou fonctions)

Classe Livre

```
classe Livre
   titre: chaîne de caractères
   auteur : chaîne de caractères
   annee : entier

   action saisir()
   début
         ecrire("titre : ") lire(titre)
         ecrire("auteur : ") lire(auteur)
         ecrire("annee : ") lire(annee)
   fin
```


Classe Livre

```
   action afficher()
   début
         ecrire("titre : ",titre)
         ecrire("auteur : ",auteur)
         ecrire("annee : ",annee)
   fin
fin classe
```

- En java : Livre.java

Classe Bibliothèque

```
classe Bibliothèque
      livres : tableau de MAX Livre
      nbLivres : entier

      action ajouterLivre(livre : Livre)
      début
         livres[nbLivres]=livre;
         nbLivres=nbLivres+1;
      fin

      action afficher()
      début
         pour i variant de 0 à nbLivres-1 faire
           afficher(livres[i])
         fin pour
      fin
```


Classe Bibliothèque

```
   action rechercherLivre(in titre : chaîne) retourne Livre
   début
         pour i variant de 0 à nbLivres-1 faire
                si livres[i].titre = titre alors
                       retourner livres[i]
                fin si
         fin pour
         retourner null
   fin
fin classe
```

- En Java : Bibliotheque.java

Une petite application

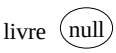
- On désire créer une application qui :
  - construise une bibliothèque
  - construise trois livres
  - permette la saisie de ces trois livres
  - permette d'ajouter les trois livres à la bibliothèque
  - affiche le contenu de la bibliothèque
- En Java : GestionBibliotheque.java

Objets

- Un objet est créé à partir d'une classe. On dit qu'un objet est une instance de la classe à partir de laquelle il est créé.
- Instanciation : new Livre()
- Un objet est ainsi créé mais aucune référence sur celui-ci n'est gardée.

Variable « référence »

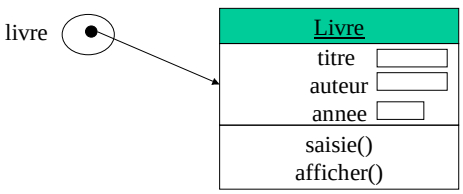
- Déclaration d'une variable « référence »
  Livre livre;



- Une variable « référence » de nom livre est créée. Sa valeur initiale est null indiquant ainsi qu'elle ne référence (pointe) rien.

Instanciation

- Création d'un objet de classe Livre
  livre = new Livre();



- livre référence maintenant ce nouvel objet

Problème (TD/TP)

- On souhaite écrire une application permettant à l'utilisateur (via un menu) d'effectuer des opérations sur les rationnels.
- Plus précisément, cette application (simpliste) doit permettre à l'utilisateur d'effectuer au choix la somme ou le produit de deux nombres rationnels.

Problème (TP)

- On souhaite écrire une application permettant de simuler une partie (simpliste) de 421.
- Après analyse, on distingue les classes suivantes :
  - De
  - Joueur
  - Assistant (de joueur)
  - Arbitre
  - Jeu

Cacher l'implémentation

Cacher l'implémentation

- Cacher l'implémentation consiste à garder privé le fonctionnement interne d'une classe (en se basant sur des champs privés).
- Cela possède deux avantages certains :
  - Permettre un contrôle de cohérence à l'aide de constructeurs et de fonctions d'accès
  - Permettre une modification de l'implémentation sans aucune conséquence pour les utilisateurs de la classe

Date

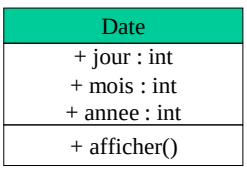
- Considérons un exemple de classe
  - la classe Date.
- Dans un premier temps, une date sera caractérisée par trois champs et une méthode :
  - jour
  - mois
  - annee
  - afficher()

Classe Date

```
class Date
{
   public int jour;
   public int mois;
   public int annee;

   public void afficher()
   {
        System.out.println(jour+"/"+mois+"/"+annee);
   }
}
```

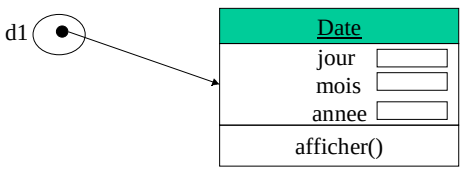

Classe Date



Instanciation

- Création d'un objet de classe Date

  Date d1 = new Date();



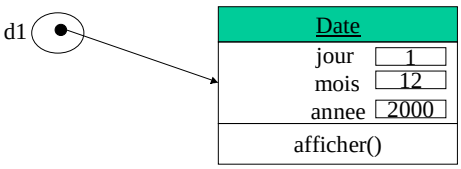
- d1 référence maintenant ce nouvel objet

Initialiser les champs

d1.jour=1;

d1.mois=12;

d1.annee=2000;



Contrôle de cohérence ?

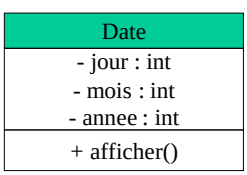
- L'instruction suivante est autorisée :
  d1.jour=-1;
- Aucun contrôle ne peut être assuré sur la validité des valeurs des champs si ceux-ci sont publiques.
- A retenir : déclarer privés autant que possible les champs des classes.

Classe Date

```
public class Date
{
   private int jour;
   private int mois;
   private int annee;

   public void afficher()
   {
        System.out.println(jour+"/"+mois+"/"+annee);
   }
}
```


Classe Date



L'accès aux champs ?

- A retenir : initialiser les champs avec des valeurs cohérentes lors de la création de l'objet (éventuellement avec des valeurs par défaut).
  Ù utiliser des constructeurs.
- A retenir : fournir des fonctions d'accès pour chaque champ.
  Ù utiliser des fonctions get et set

Constructeurs

- Dans chaque classe, peut apparaître 0, 1 ou n constructeurs.
- Chaque constructeur doit posséder une signature différente (une séquence d'arguments se différenciant au niveau du typage).
- Un constructeur a pour objectif d'initialiser de façon cohérente l'état d'un objet.

Constructeurs

- Lors de toute instanciation, un constructeur est appelé automatiquement.
- Si aucun constructeur ne figure dans une classe, un constructeur par défaut est utilisé.
- Un constructeur
  – porte le même nom que la classe
  – n'a pas de type retour et ne comporte donc pas d'instruction return

Classe Date

```
class Date
{
   private int jour, mois, annee;

   public Date(int a)
   {
        jour=1;  mois=1;  annee=a;
   }
   public Date(int m, int a)
   {
        jour=1;
        if (m<1) mois=1;
        else if (m>12) mois=12;
        annee=a;
   }
```

Classe Date

```
    public Date(int j, int m, int a)
    {
        ...
    }
    public static void main(String[] args)
    {
        Date d1 = new Date(1998);
        Date d2 = new Date(1,2000);
        Date d3 = new Date(15,7,1978);
        Date d4 = new Date();  // Error
        ...
    }
}
```


Constructeur sans argument

- Pour pouvoir utiliser un constructeur sans arguments, il faut que celui-ci figure dans la classe sauf si aucun constructeur n'y figure.
- Il faut donc pour notre exemple ajouter :

```
public Date()
{
    jour=1; mois=1; annee=1900;
}
```


Fonctions d'accès de type « set »

- Une fonction d'accès en écriture permet de modifier la valeur d'un champ
- Il s'agit d'une méthode publique dont le nom commence par set.
- Sur notre exemple :

```
public void setJour(int j) { jour=j; }
public void setMois(int m) { mois=m; }
public void setAnnee(int a) { annee=a; }
```


Contrôle de cohérence

- Et le contrôle de cohérence ? Il faut le placer au niveau de chacune de ces fonctions.
- Par exemple :

```
public void setMois(int m)
{
    if (m<1) mois=1;
    else if (m>12) mois=12;
    else mois=m;
}
```


Contrôle de cohérence

- Le codage de la fonction setMois précédente n'est pas parfait car il manque le contrôle par rapport au jour.
- Il semble préférable sur cet exemple par rapport au contrôle de cohérence de disposer des méthodes :
  - setJourMoisAnnee(int j, int m, int a)
  - avancerDe(int nbJours)
  - reculerDe(int nbJours)

Fonctions d'accès de type « get »

- Une fonction d'accès en lecture permet de récupérer la valeur d'un champ
- Il s'agit d'une méthode publique dont le nom commence par get.
- Sur notre exemple :

```
public int getJour() { return jour; }
public int  getMois() { return mois; }
public int getAnnee() { return annee; }
```


Implémentations non cachées

- Considérons une implémentation non cachée de la classe Date et une application utilisant la classe Date.
- Examinons alors les répercussions d'une modification de l'implémentation de la classe Date.

Implémentation non cachée 1

```
class Date
{
    public int jour;
    public int mois;
    public int annee;

    public void afficher()
    {
        System.out.print(jour + "/");
        System.out.print(mois + "/")
        System.out.println(annee);
    }
}
```

Application 1

```
class ApplicationDate
{
   static void main(String[] args)
   {
      Date date = new Date();
      date.jour=31;
      date.mois=2; // problème de cohérence
      date.annee=2000;
      date.afficher();
   }
}
```


Modification de l'implémentation

- On décide pour des raisons d'efficacité (gain de place) d'utiliser le même champ pour coder à la fois le jour et le mois.
- On utilisera alors les opérateurs :
  <<
  >>
- La modification de l'implémentation de la classe Date entraîne une modification de l'implémentation de la classe ApplicationDate.|

Implémentation non cachée 2

```
class Date
{
   public int jourMois;
   public int annee;

   public void afficher()
   {
      System.out.print((jourMois >> 16) + "/");
      System.out.print((jourMois << 16) >> 16) + "/");
      System.out.println(annee);
   }
}
```


Application 2

```
class ApplicationDate
{
   static void main(String[] args)
   {
      Date date = new Date();
      date.jourMois=(date.jourMois | (31 << 16));
      date.jourMois=(date.jourMois | 2);
      date.annee=2000;
      date.afficher();
   }
}
```


Implémentations cachées

- Considérons une implémentation cachée de la classe Date et une application utilisant la classe Date.
- Examinons alors les répercussions d'une modification de l'implémentation de la classe Date.

Implémentation cachée 1

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

   public int getJour() { return jour; }
   public int getMois() { return mois; }
   public int getAnnee() { return annee; }
```


Implémentation cachée 1

```
   public void setJour(int j) { jour=j; }
   public void setMois(int m) { mois=m; }
   public void setAnnee(int a) { annee=a; }

   public void afficher()
   {
      System.out.println(jour + "/");
      System.out.print(mois + "/");
      System.out.println(annee);
   }
}
```


Application 1

```
class ApplicationDate
{
   static void main(String[] args)
   {
      Date date = new Date();

      date.setJour(31);
      date.setMois(2);
      date.setAnnee(2000);
      date.afficher();
   }
}
```

Modification de l'implémentation

- Comme précédemment, on décide pour des raisons d'efficacité (gain de place) d'utiliser le même champ pour coder à la fois le jour et le mois.
- Cependant, la modification de l'implémentation de la classe Date n'entraîne aucune modification de l'implémentation de la classe ApplicationDate.|

Implémentation cachée 2

- Les modifications à apporter à la classe Date sont les suivantes :

```
public int getJour() { return (jourMois>>16); }
public int getMois() { return ((jourMois<<16) >>16); }
public void setJour(int j) { jourMois |= (j<<16); }
public void setMois(int m) { jourMois |= m; }
```

- Dans la méthode afficher, il faut également faire appel aux méthodes getJour et getMois.

Application 2

```
class ApplicationDate
{
   static void main(String[] args)
   {
        Date date = new Date();

        date.setJour(31);
        date.setMois(2);
        date.setAnnee(2000);
        date.afficher();
   }
}
```


Bénéfice de l'encapsulation

- Cacher l'implémentation est primordial car cela garantit qu'il n'y ait pas de répercussions en cas de modification d'implémentation.
- Toutefois l'interface (ensemble des méthodes publiques) doit être « bien pensée » car une modification de l'interface peut entraîner de grosses répercussions.

Les données en Java

Les données en java

- Variables
  - Primitives : contiennent une valeur atomique telle qu'un entier, un réel, ...
  - Références : contiennent l'adresse d'un objet en mémoire...
- Objets
  - Instances de classes représentant les entités « actives » d'une application.

Variables « primitives »

- Variables définies à partir d'un type primitif (élémentaire).
- Exemple de déclarations

```
short s;
s=10;
```

s | 10

```
float f=5.20;
...
```

f | 5.20

Types primitifs

- boolean (1 bit) ⇒ false et true
- char (16 bits) ⇒ unicode 0 à unicode 2^{16}-1
- byte (8 bits) ⇒ -2^8 à 2^8-1
- short (16 bits) ⇒ -2^{15} à 2^{15}-1
- int (32 bits) ⇒ -2^{31} à 2^{31}-1
- long (64 bits) ⇒ -2^{63} à 2^{63}-1
- float (32 bits) ⇒ norme IEEE 754
- long (64 bits) ⇒ norme IEEE 754

Variables « références »

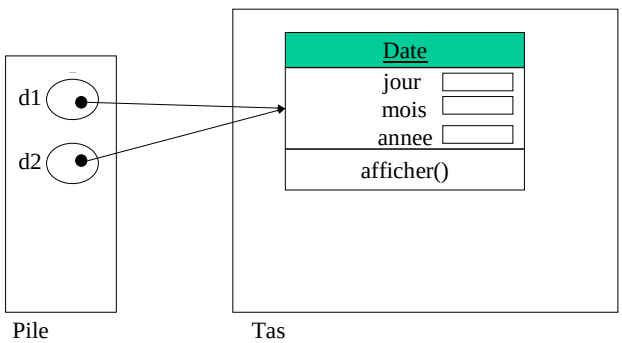
- Variables définies à partir d'une classe.
- Si il n'y a pas initialisation lors de la déclaration, la variable contient la valeur « null ».
- Cette valeur spéciale indique que la variable ne référence (pointe) sur aucun objet.

Objets

- Les objets sont alloués sur le tas. tandis que les variables (primitives et références) sont toujours placées sur la pile.
- Lorsqu'un objet n'est plus référencé, il peut être détruit par le « ramasse-miettes » (« garbage collector »).

Exemple

```
Date d1;
Date d2;
d1 = new Date();
d2=d1;
```



Java versus C++

- Remarque 1

  Date d1; // en Java

  est équivalent à

  Date *d1: // en C++
- Remarque 2

  Il n'y a pas d'équivalent en Java à la déclaration :

  Date d1; // en C++

Java versus C++

- (Gros) Avantage de Java
  - simplicité et clarté
- (Léger) avantage de C++
  - copie automatique via constructeurs de copie et opérateurs d'affectation
  - en Java, il faut cloner pour obtenir une copie complète d'un objet

Statut des variables

- Une variable (que celle-ci soit « primitive » ou « référence ») peut correspondre à :
  - une variable d'instance (un champ)
  - une variable locale
- La différence se situe simplement au niveau de l'endroit où s'effectue la déclaration.

Variable d'instance

- Une variable d'instance (un champ) est une variable définie au niveau d'une classe (en dehors de toute fonction).
- Une variable d'instance :
  - est accessible par toutes les méthodes de la classe où elle est définie,
  - possède toujours une valeur par défaut.
- En quelque sorte, un champ représente une variable globale à la classe.

Variable locale

- Une variable locale est une variable définie dans une méthode.
- Une variable locale :
  - n'est accessible que dans la méthode où elle est définie (à partir de sa déclaration),
  - ne possède jamais de valeur par défaut.
- L'utilisation d'une variable locale non initialisée provoque une erreur.

Variable d'instance ou locale

- Une variable d'instance doit correspondre à une caractéristique logique du type d'objet que l'on définit.
- En pratique, il arrive parfois de définir comme champs des variables de travail utilisés par plusieurs méthodes de la classe.

this

this

- this est employé dans deux contextes distincts :
  - pour référencer l'objet ayant reçu un message
  - pour effectuer l'appel à un constructeur à partir d'un autre

this « référence »

- On peut considérer que de façon interne, chaque méthode (non statique) dispose d'un argument supplémentaire.
  public void afficher(Date this)
- Les appels suivants
  d1.afficher(); d2.afficher();
  correspondent alors à :
  Date.afficher(d1); Date.afficher(d2);

this « référence »

- this est une variable « référence ».
- this désigne l'objet ayant reçu un message à traiter et donc pour lequel on doit exécuter une méthode.
- Le plus souvent, this est implicite.
- Parfois, this doit être explicite.

this implicite

```
class Alpiniste
{
   void grimper() { ... }
   void descendre() { ... }
   void faireAscension()
   {
        grimper();  // équivalent à : this.grimper();
        descendre(); // équivalent à : this.descendre();
   }

}
```


this explicite

```
class Compteur
{
   private int valeur=0;
   public Compteur incrementer()
   {
        valeur++; return this;
   }
   public static void main(String[] args)
   {
        Compteur c = new Compteur();
        c.incrementer().incrementer();
   }
}
```


this explicite

```
class Urne
{
   private int nbBulletins;
   public Urne(int nbBulletins)
   {
        this.nbBulletins=nbBulletins;
   }
}
```

Danger : une erreur de frappe du paramètre

```
   public Urne(int nbBuletins)
```

this « constructeur »

- Parfois, il est intéressant de pouvoir appeler un constructeur à partir d'un autre.
- On utilise pour cela this suivi des arguments nécessaires au constructeur appelé.
- Restrictions
  - doit être la première instruction du constructeur
  - un seul appel possible par constructeur

this « constructeur »

```
class Automobile
{
   private int nbVitesses, vitesseMaximale;
   private boolean airbag;
   private Color couleur=Color.white;
   public Automobile(int nv, int vm)
   {
        nbVitesses=nv; vitesseMaximale=vm;
   }
   public Automobile(int nv, int vm, boolean bag)
   {
        this(nv,vm); airbag=bag;
   }
```


this « constructeur »

```
   public Automobile(int nv, int vm, boolean bag,
   Color c)
   {
        this(nb,vm,bag);
        couleur=c;
   }
   public static void main(String[] args)
   {
        Automobile a1 = new Automobile(4,160);
        Automobile a2;
        a2 = new Automobile(5,200,false,Color.red);
   }
}
```


static

static

- Parfois, il est intéressant de disposer de données et/ou de traitement généraux.
- static versus non static
  - static : point de vue d'une classe
    - Ù variables (champs) et méthodes de classes
  - non static : point de vue d'une instance
    - Ù variables (champs) et méthodes d'instance

Accès aux membres static

- Dans le corps d'une méthode :
  - Pour accéder à un membre (champ ou méthode) static de la même classe, il suffit de préciser son nom.
  - Pour accéder à un membre static d'une autre classe, il faut préfixer le nom du membre par le nom de la classe.

Méthode de classe

- Une méthode de classe (static) ne peut appeler une méthode d'instance (non static).
- Une méthode d'instance (non static) peut appeler une méthode de classe (static).
- Appel d'une méthode de classe :

  class-name.method-name

- static doit être utilisé avec parcimonie.

Emplois de static

- On peut distinguer 4 contextes d'emplois de static :
  1. Programmation procédurale
  2. Bibliothèque de fonctions
  3. Obtention d'instances de classes particulières
  4. Gestion de données globales à une classe
     - données variables
     - données constantes

Programmation procédurale

- Approche tolérable pour de petites applications.
- Tous les membres sont déclarés « static »
  - les champs
  - les méthodes
- Les champs correspondent alors à des variables globales au sein de la classe ou ils sont définis.

Bibliothèque de fonctions

- Il est parfois intéressant de regrouper un ensemble de fonctionnalités générales au sein d'une même entité (une bibliothèque).
- Il suffit alors essentiellement de déclarer une classe avec un ensemble de méthodes static.
- Exemple : les classes Math et Mediator.

Obtention d'instances de classe

- Vis à vis de certaines classes particulières (généralement liées au système), il n'est pas possible de créer directement des objets (avec new).
- Toutefois, on peut récupérer ce type d'objets
  - soit en utilisant des méthodes static de ces classes
  - soit par l'intermédiaire d'autres objets

Exemples

- InetAdress adr = InetAddress.getByName("12.5.3.1");
  permet de récupérer une adresse IP.
- Toolkit toolkit = Toolkit.getDefaultToolkit();
  permet de récupérer la boîte à outil par défaut.
- Image image = toolkit.getImage("duke.jpg");
  permet de récupérer une image via le toolkit

Données globales à une classe

- Il est parfois nécessaire ou intéressant de gérer des données de façon globale à une classe.
- On utilise alors des champs static ainsi qu'éventuellement des méthodes static pour y accéder.
- Par exemple, compter le nombre d'objets chats créés.

Exemple

```
class Chat
{
   private String nom;
   private static int nbChats;
   public Chat(String n)
   {
        nom=n;
        Chat.nbChats++; // ou nbChats++
    }
   public static int getNbChats() { return nbChats; }
   ...
```


Exemple

```
   public static void main(String[] args)
   {
        Chat c1 = new Chat ("ludwig");
        Chat c2 = new Chat ("capucine");
        Chat c3 = new Chat ("toupie");
        int n= Chat.getNbChats();
        System.out.println("il y a  " +  n +  " chats  ");
   }
}
```


Et les constantes ?

- En règle générale, les constantes sont déclarées public et static (et final).
- En effet, il n'est pas utile de posséder un jeu de constantes pour chaque objet créé.
- Exemple de constantes :
  - public static final Color red = …;
  - public static final Color blue = …;
- En règle générale, les noms des constantes sont en majuscule.

Visibilité d'une variable

- En résumé, la visibilité d'une variable peut correspondre à :
  - à une méthode : variable locale
  - un objet : champ d'une classe
  - une classe : champ *static* d'une classe

static et this

- Une méthode de classe exécute une action indépendante d'une instance particulière.
- Elle ne peut utiliser de référence (implicite ou explicite) à une instance courante (this).
- Il serait, par exemple, interdit d'écrire :

```
static String getNom() { return nom; }
```


main

- La méthode main() est nécessairement static. Pourquoi ?
- Il s'agit du point d'entrée pour l'exécution. Aucune instance n'est donc encore créée lorsque la méthode main() commence son exécution.

Tableaux

Tableaux

- En java, un tableau est un objet.
- On peut déclarer des tableaux à :
  - 1D : String t1[];
  - 2D : int t2[][];
  - 3D : Date t3[][][];
  - ....
- t1, t2 et t3 représentent des variables « référence ».

Instanciation

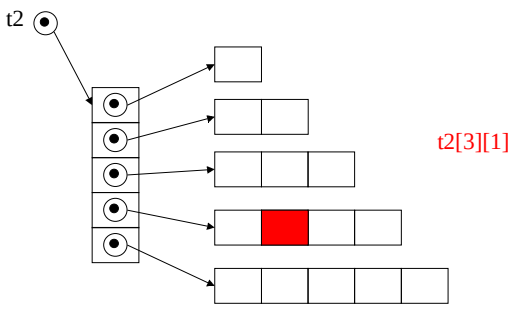
- Une opération d'instanciation est nécessaire pour créer un tableau.
- Le nombre d'éléments est alors donné par length.
  - l'indice du premier élément est 0
  - l'indice du dernier élément est length-1.

Exemple

- Initialiser le tableau t2 « en escalier » avec des nombre aléatoires.

```
Random r = new Random();
t2 = new int[5][];
for (int i=0; i<t2.length; i++)
      t2[i] = new int[i+1];
for (int i=0; i<t2.length; i++)
{
       for (int j=0; i<t2[i].length; i++)
              t2[i][j] = r.nextInt(100);
}
```


Exemple

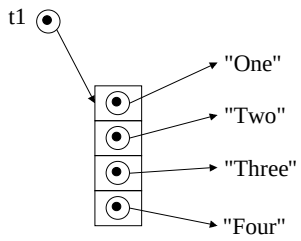


Initialisation

- Il est possible d'initialiser un tableau au moment de sa déclaration.
- C'est le seul cas où new n'apparaît pas.
- Des accolades sont utilisées pour énumérer les éléments du tableau.
- Le nombre de niveaux d 'accolades correspond à la dimension du tableau.

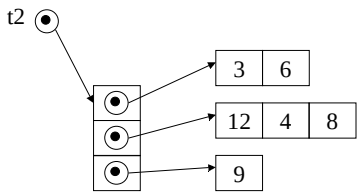
Exemple

- String t1[] = {"one", "two","three","four"};



Exemple

- int t2[][] = {{3,6},{12,4,8},{9}};

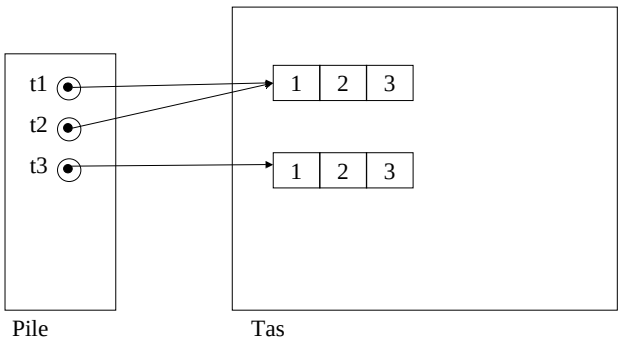


Clonage

- Tout objet peut être cloné. La méthode clone de Object est redéfinie.
- aliasing versus clonage
  - aliasing = copier une référence
  - clonage = dupliquer l'objet


```
int t1[] = {1,2,3};
int t2[] = t1;
int t3[];
t3 = t1.clone();
```

Exemple



Java versus C++

- Les déclarations suivantes en C++:

```
int[10] t1;
int [10] t2 = t1;
int *t3;
```

- correspondent en Java à :

```
int t1[] = new int[10];
int t2[] = t1.clone();
int t3[];
```


Packages

Composants fondamentaux

- Classes
- Fichiers sources (unités de compilation)
  - plusieurs classes possibles dans un fichier
  - mais une seule classe publique par fichier
- Packages (unités de bibliothèque)
  - plusieurs unités de compilation
- API (Application Program Interface)
  - Plusieurs packages

Package

- Un package est un regroupement logique de plusieurs classes.
- Le langage Java est constitué d'un grand nombre de packages.
- Un package correspond à la notion de librairies (ou bibliothèques) de langages tel que le C.

Quelques packages de Java 2

- java.lang : classes de base de Java
- java.util : classes utilitaires
- java.math : fonctions mathématiques
- java.io : entrées-sorties
- java.awt : interfaces graphiques
- javax.swing : interfaces graphiques
- java net : réseau

Importation

- Pour utiliser un package, il est possible d'inclure une directive d'importation en début de fichier (avant toute définition de classe).
- Importer toutes les classes d'un package :
  import java.util.*;
- Importer une seule classe d'un package :
  import java.util.Vector;

v Note : le package java.lang est importé automatiquement.

Utilisation sans importation

- L'importation n'est pas indispensable.
- Il est toujours possible de préfixer le nom de la classe avec le nom du package.
- Par exemple :
  java.util.Vector v = new java.util.Vector();
- Cette utilisation est surtout intéressante pour lever une ambiguïté, à savoir, une interférence entre 2 classes de même nom dans 2 packages différents.

Appartenance à un package

- Format de la directive :
  package nomPackage;
- Une telle directive précède les directives d'importation.
- Un package par défaut regroupe toutes les unités de compilation qui apparaissent dans le même répertoire et qui n'incluent pas d'indication d'appartenance à un package.

Sous-package

- Un package peut contenir des sous-packages.
- Par exemple, javax.swing contient :
  – javax.swing.border
  – javax.swing.event

v Note : l'importation d'un package n'implique pas de façon automatique l'importation des sous-packages.

Nom d'un package

- Le nom d'un package est entièrement déterminé par le chemin indiquant l'emplacement des classes de celui-ci.
- La première partie du chemin peut être omise si celle-ci figure comme l'une des alternatives de la variable CLASSPATH.

Exemple 1/2

- Je dispose de plusieurs classes logiquement rassemblées (sous Windows) dans :
  c:\java\lecoutre\abscon
- La variable d'environnement CLASSPATH est défini (dans autoexec.bat) par :
  SET CLASSPATH=.;c:\jdk1.2.2\lib;c:\java
- Le package que je souhaite définir va alors s'appeler :
  lecoutre.abscon

Exemple 2/2

- La directive suivante doit être placée dans chaque classe de c:\java\lecoutre\abscon :

  package lecoutre.abscon;
- Pour pouvoir utiliser ce package dans une classe par ailleurs, il est alors nécessaire d'inclure la directive suivante :

  import lecoutre.abscon.*;

Contrôle d'accès

Spécificateurs d'accès

- Les membres (champs et méthodes) d'une classe possèdent tous une certaine visibilité.
- Celle-ci est déterminée par un specificateur d'accès. Du plus restrictif au moins restrictif :
  - private
  - "package" (pas de mot-clé)
  - protected
  - public

Visibilité des membres

unité / specifier	classe	package	sous-classe	monde
private	X			
package	X	X		
protected	X	X	X	
public	X	X	X	X

private et package

- Les membres *private* ne sont pas accessibles en dehors de la classe dans laquelle ils sont définis
- Les membres *package* (définis sans spécifier) sont accessibles à partir de toute classe appartenant au même package

protected et public

- Les membres *protected* sont accessibles à partir de toute sous-classe et de toute classe appartenant au même package.
- Les membres *public* sont accessibles à partir de n'importe quelle classe (en général, seules certaines méthodes sont publiques).

Java versus C++

- En java, un mot-clef doit être placé avant chaque champ et méthode.
- En C++, un mot-clef reste valable tant qu'un autre n'apparaît pas.