Plan du cours Java

- Java de base (ce document et le suivant)
- Héritage, polymorphisme, classes abstraites et interfaces
- Exceptions
- Compléments sur le langage
- Collections
- Généricité
- Interface graphique
- Entrées-sorties

Plan de cette partie

- Présentation du langage
- Notions sur la programmation objet
- Classes
- Structure lexicale du langage
- Quelques principes de programmation

Principales propriétés de Java

- Langage orienté objet, à classes (les objets sont décrits/regroupés dans des classes)
- de syntaxe proche du langage C
- fourni avec le JDK (*Java Development Kit*) :
  - outils de développement
  - ensemble de paquetages très riches et très variés
- portable grâce à l'exécution par une machine virtuelle : « *Write once, run everywhere* »

Un but difficile à atteindre

Autres propriétés importantes

- multi-tâches (*thread*)
- sûr
  - fortement typé
  - nombreuses vérifications au chargement des classes et durant leur exécution
- adapté à Internet
  - chargement de classes en cours d'exécution (le plus souvent par le réseau : *applet* ou RMI)
  - facilités pour distribuer les traitements entre plusieurs machines (sockets, RMI, Corba, EJB)

Premier programme Java

Le code source du premier programme

```
public class HelloWorld {
  public static void main(String[] args){
    System.out.println("Hello world");
  }
}
```

point d'entrée d'exécution

- La classe Helloworld est **`public`**, donc le fichier qui la contient doit s'appeler (en tenant compte des majuscules et minuscules)

`HelloWorld.java`

Compilation d'un code source

- Un code source ne peut être exécuté directement par un ordinateur
- Il faut traduire ce code source dans un langage que l'ordinateur (le processeur de l'ordinateur) peut comprendre (langage *natif*)
- Un compilateur est un programme qui effectue cette traduction

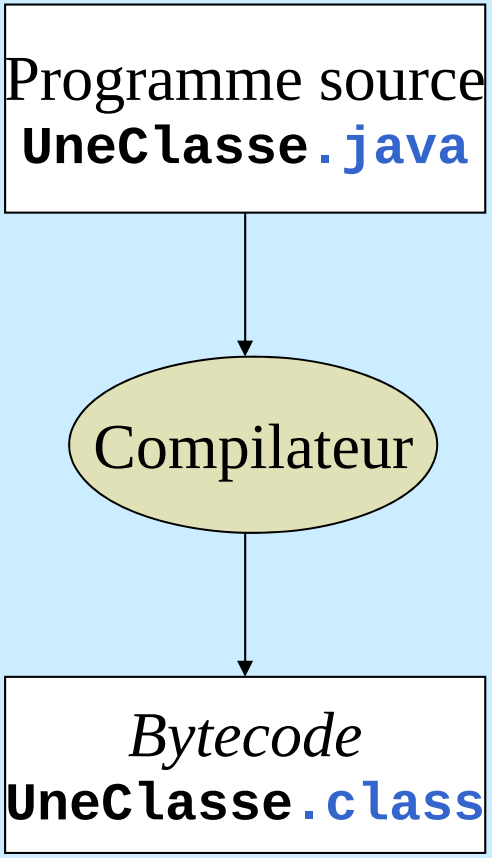
Compilation en Java → *bytecode*

- En Java, le code source n'est pas traduit directement dans le langage de l'ordinateur
- Il est d'abord traduit dans un langage appelé « *bytecode* », langage d'une machine virtuelle (JVM ; *Java Virtual Machine*) définie par *Oracle*
- Ce langage est indépendant de l'ordinateur qui va exécuter le programme

La compilation fournit du *bytecode*

Programme écrit en Java

Programme en *bytecode*, indépendant de l'ordinateur



Compilation avec *javac*

- *Oracle* fournit le compilateur *javac* avec le JDK
- `javac HelloWorld.java`
  crée un fichier « **`HelloWorld.class`** » qui contient le *bytecode*, situé dans le même répertoire que le fichier « **`.java`** »
- Le fichier à compiler peut être désigné par un chemin absolu ou relatif :
  `javac util/Liste.java`

Exécution du *bytecode*

- Le *bytecode* doit être exécuté par une JVM
- Cette JVM n'existe pas ; elle est simulée par un programme qui interprète le *bytecode* :
  - lit les instructions (en *bytecode*) du programme **`.class`**,
  - les traduit dans le langage natif du processeur de l'ordinateur
  - lance leur exécution

Exécution avec *java*

- *Oracle* fournit le programme *java* qui simule une JVM
- **`java HelloWorld`** exécute le *bytecode* de la méthode **`main`** de la classe **`HelloWorld`**
- **`HelloWorld`** est un nom de classe et pas un nom de fichier. Donc
  - on ne peut pas donner un chemin
  - pas de suffixe **`.class`**

Nom d'une classe
(pas d'un fichier) ;
pas de suffixe `.class` !

Où doit se trouver le fichier .class ?

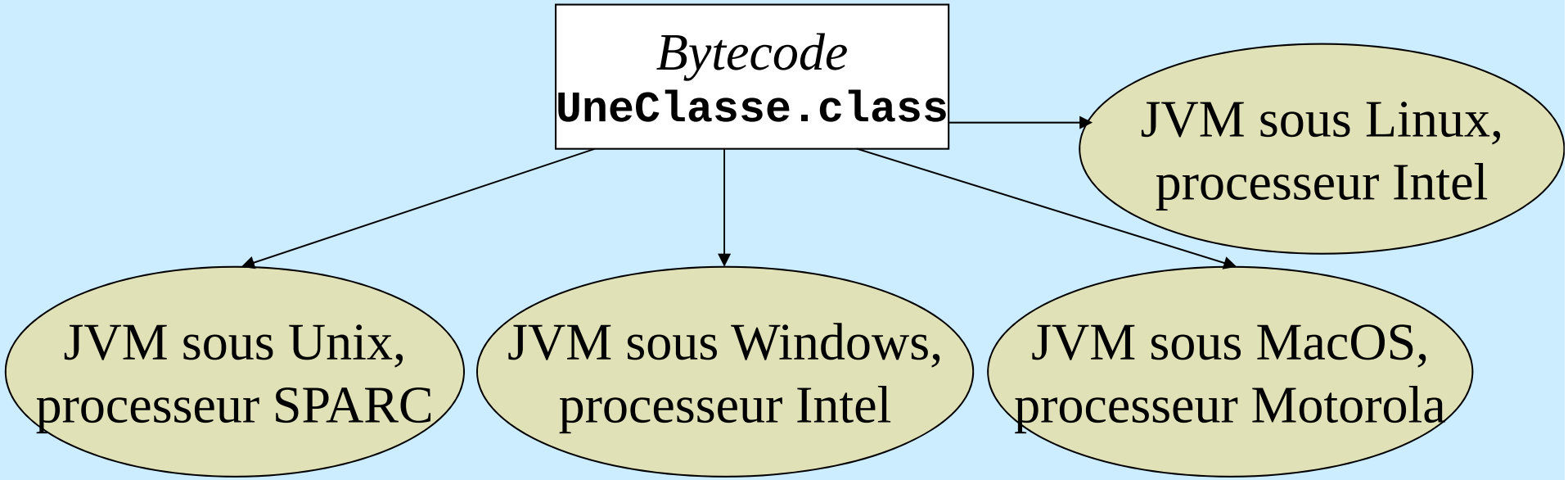
- `java HelloWorld`

  `HelloWorld.class` doit se trouver dans le *classpath*
- Le *classpath* peut recevoir une valeur
  - avec l'option **`-classpath`** de la commande java : **`java -classpath rep1/rep2 HelloWorld`**
  - avec la variable d'environnement **`CLASSPATH`** (pas recommandé)
- Par défaut le *classpath* est le répertoire courant

Les JVM

- Les systèmes qui veulent pouvoir exécuter un programme Java doivent fournir une JVM
- Tous les systèmes ont une JVM (Linux, Windows, MacOs,…)
- Il existe aussi quelques JVM « en dur », sous forme de processeurs dont le langage natif est le *bytecode* ; elles sont rarement utilisées

Le *bytecode* peut être exécuté par n'importe quelle JVM



Si un système possède une JVM, il peut exécuter tous les fichiers **`.class`** compilés sur n'importe quel autre système

Avantages de la JVM pour Internet

- Grâce à sa portabilité, le *bytecode* d'une classe peut être chargé depuis une machine distante du réseau, et exécutée par une JVM locale
- La JVM fait de nombreuses vérifications sur le *bytecode* avant son exécution pour s'assurer qu'il ne va effectuer aucune action dangereuse
- La JVM apporte donc
  - de la souplesse pour le chargement du code à exécuter
  - mais aussi de la sécurité pour l'exécution de ce code

Une certaine lenteur...

- Les vérifications effectuées sur le *bytecode* et l'étape d'interprétation de ce *bytecode* (dans le langage natif du processeur) ralentissent l'exécution des classes Java
- Mais les techniques « *Just In Time* (JIT) » ou « *Hotspot* » réduisent ce problème : elles permettent de ne traduire qu'une seule fois en code natif les instructions qui sont (souvent pour Hotspot) exécutées

Java et les autres langages

- Java est devenu en quelques années un des langages de développement les plus utilisés, surtout pour les applications qui ont besoin d'une grande portabilité ou d'une grande souplesse sur Internet
- Pour les applications qui nécessitent une très grande rapidité d'exécution, on peut préférer encore les langages C, C++, ou le bon vieux Fortran (qui a des bibliothèques très utilisées pour le calcul scientifique)

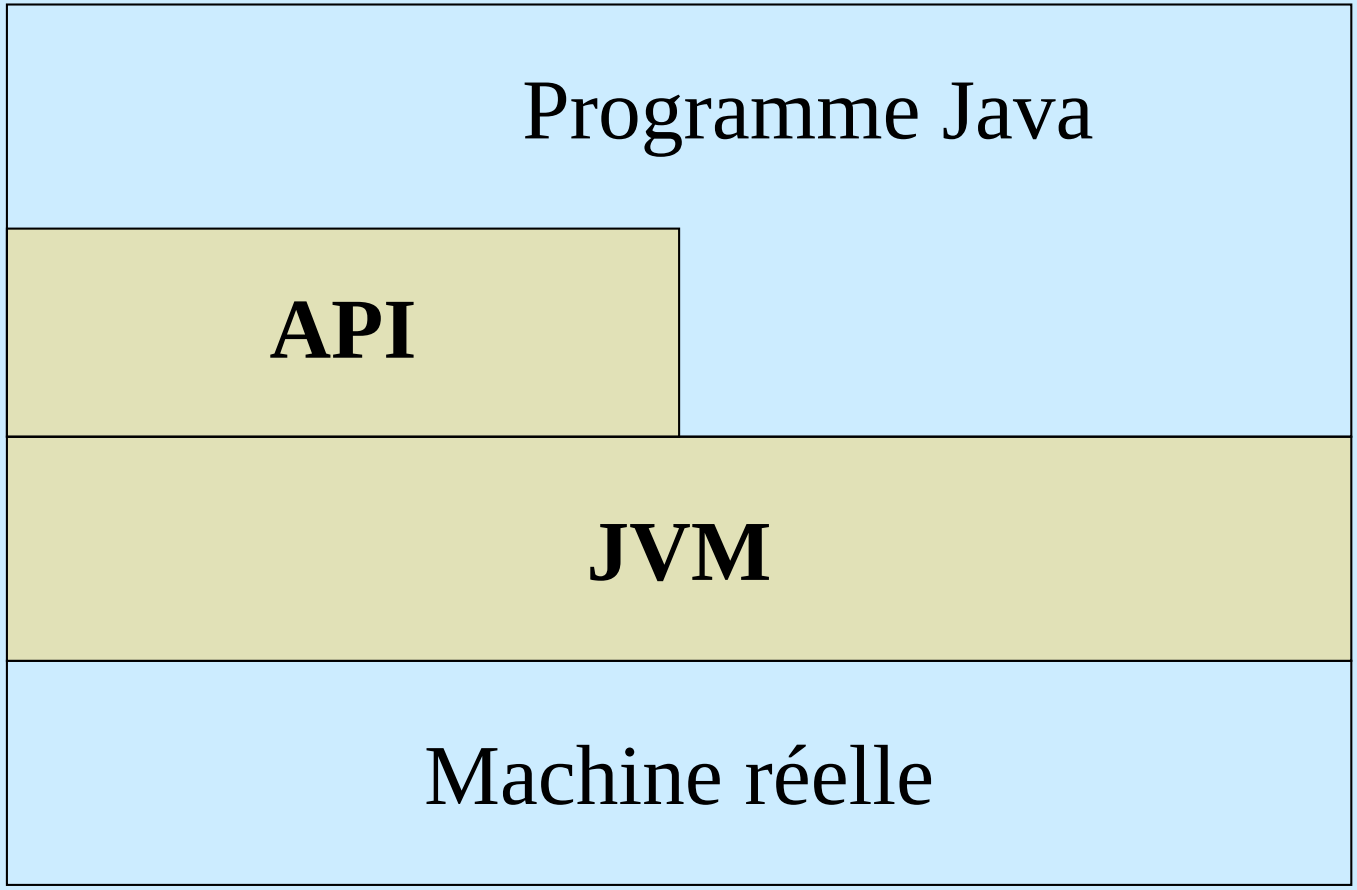
Spécifications de Java

- Java, c'est en fait
  - le langage Java : http://java.sun.com/docs/books/jls/
  - une JVM : http://java.sun.com/docs/books/vmspec/
  - les API : selon la documentation *javadoc* fournie avec les différents paquetages
- Java n'est pas normalisé ; son évolution est gérée par le JCP (Java Community Process ; http://www.jcp.org/) dans lequel Oracle tient une place prépondérante

Implémentation de référence

- Oracle accompagne les spécifications Java
  - d'une implémentation de référence
  - de nombreux tutoriels

Plate-forme Java



API (*Application Programming Interface*) : bibliothèques de classes standard

3 éditions de Java

- Java SE (J2SE) : Java Standard Edition ; JDK = *Java SE Development Kit*
- Java EE (J2EE) : Enterprise Edition qui ajoute les API pour écrire des applications installées sur les serveurs dans des applications distribuées : servlet, JSP, JSF, EJB,…
- Java ME (J2ME) : Micro Edition, version pour écrire des programmes embarqués (carte à puce/*Java card*, téléphone portable,…)

Version couverte par le cours

- Java SE 7
- Attention, Java est passé directement de la version 1.4 à la version 5.0
- En effet, l'ancienne numérotation des différentes versions (1.0, 1.1, 1.2,…) ne reflétaient pas les importantes modifications effectuées ; elles auraient plutôt dû s'appeler 1, 2,…

Votre environnement de développement

- Éditeur de texte (*emacs*, avec JDE)
- Compilateur (*javac*)
- Interpréteur de *bytecode* (*java*)
- Aide en ligne sur le JDK (sous navigateur Web)
- Générateur automatique de documentation (*javadoc*)
- Testeur pour applet (*appletviewer*)
- Débogueur (*jdb*)
- . . .
- Après l'étude des paquetages, éventuellement un IDE tel que NetBeans ou Eclipse

Integrated Development Environment

Compléments sur la compilation et l'exécution

Variables d'environnement

- PATH : doit inclure le répertoire qui contient les utilitaires Java (**`javac`**, **`java`**, **`javadoc`**,…)
- CLASSPATH : indique le chemin de recherche des classes de l'utilisateur
- **Évitez la variable `CLASSPATH`**

Une classe **Point**

```
/** Modélise un point de coordonnées x, y */
public class Point {
  private int x, y;
  public Point(int x1, int y1) {     // un constructeur
    x = x1;
    y = y1;
  }
  public double distance(Point p) {  // une méthode
    return Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)*(y-p.y));
  }
  public static void main(String[] args) {
    Point p1 = new Point(1, 2);
    Point p2 = new Point(5, 1);
    System.out.println("Distance : " + p1.distance(p2));
  }
}
```

2 classes dans 1 fichier

```
/** Modélise un point de coordonnées x, y */
public class Point {
  private int x, y;
  public Point(int x1, int y1) {
    x = x1; y = y1;
  }
  public double distance(Point p) {
    return Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)*(y-p.y));
  }
}
```

Fichier **Point.java**

```
/** Teste la classe Point */
class TestPoint {
  public static void main(String[] args) {
    Point p1 = new Point(1, 2);
    Point p2 = new Point(5, 1);
    System.out.println("Distance : " + p1.distance(p2));
  }
}
```

Compilation et exécution de la classe Point

- La compilation du fichier **`Point.java`**

  `javac Point.java`

  fournit 2 fichiers classes : **`Point.class`** et **`TestPoint.class`**
- On lance l'exécution de la classe **`TestPoint`** qui a une méthode **`main()`**

  `java TestPoint`

2 classes dans 2 fichiers

```
/** Modélise un point de coordonnées x, y */
public class Point {
  private int x, y;
  public Point(int x1, int y1) {
    x = x1; y = y1;
  }
  public double distance(Point p) {
    return Math.sqrt((x-p.x)*(x-p.x) + (y-p.y)*(y-p.y));
  }
}
```

Fichier **Point.java**

```
/** Pour tester la classe Point */
class TestPoint {
  public static void main(String[] args) {
    Point p1 = new Point(1, 2);
    Point p2 = new Point(5, 1);
    System.out.println("Distance : " + p1.distance(p2));
  }
}
```

Fichier **TestPoint.java**

Architecture d'un programme source Java

- Programme source Java = ensemble de fichiers « **`.java`** »
- Chaque fichier « **`.java`** » contient une ou *plusieurs* définitions de classes
- Au plus une définition de classe **`public`** par fichier « **`.java`** » (avec nom du fichier = nom de la classe publique)

Chargement dynamique des classes

- Durant l'exécution d'un code Java, les classes (leur *bytecode*) sont chargées dans la JVM au fur et à mesure des besoins
- Une classe peut être chargée
  - depuis la machine locale (le cas le plus fréquent)
  - depuis une autre machine, par le réseau
  - par tout autre moyen (base de données,…)

Applications indépendantes et *applets*

Deux types de programmes

- Applications indépendantes
- Applets référencée par une page HTML et exécutée dans la JVM d'un navigateur Web

Application indépendante

- Lancement de l'exécution de la classe de lancement de l'application (dite classe principale ; *main* en anglais) ; par exemple :

```
java TestPoint
```

- *java* lance l'interprétation du code de la méthode `main()` de la classe **`TestPoint`**

Méthode **`main`**

- Le « *profil* » d'une méthode est donné par son en-tête de définition ; celui de **`main()`** doit être :

```
public static void main(String[] args)
```

  - Signature d'une méthode : nom de la méthode et ensemble des types de ses paramètres
  - Signature de la méthode **`main()`** : **`main(String[])`**
  - En Java, le type de la valeur de retour de la méthode ne fait pas partie de sa signature (au contraire de la définition habituelle d'une signature)

Applet

- Objet Java, référencé dans une page Web (écrite dans le langage HTML)
- En fait cet objet doit appartenir à une classe Java qui hérite de la classe **`Applet`** (voir cours sur l'héritage)
- Le lancement d'une (un ?) applet(te ?) se fait quand la partie de la page Web qui référence l'applet est affichée par le client Web

Exemple de page Web qui contient une applet

```
<HTML>
<HEAD>
  <TITLE> Une applet </TITLE>
</HEAD>
<BODY>
  <H2> Exécuter une applet </H2>
  <APPLET code="HelloApplet.class"
          width=500
          height=300>
     Votre navigateur ne peut exécuter une applet
  </APPLET>
</BODY>
</HTML>
```

Dimensions de l'emplacement réservé à l'affichage de l'applet

Exemple d'applet

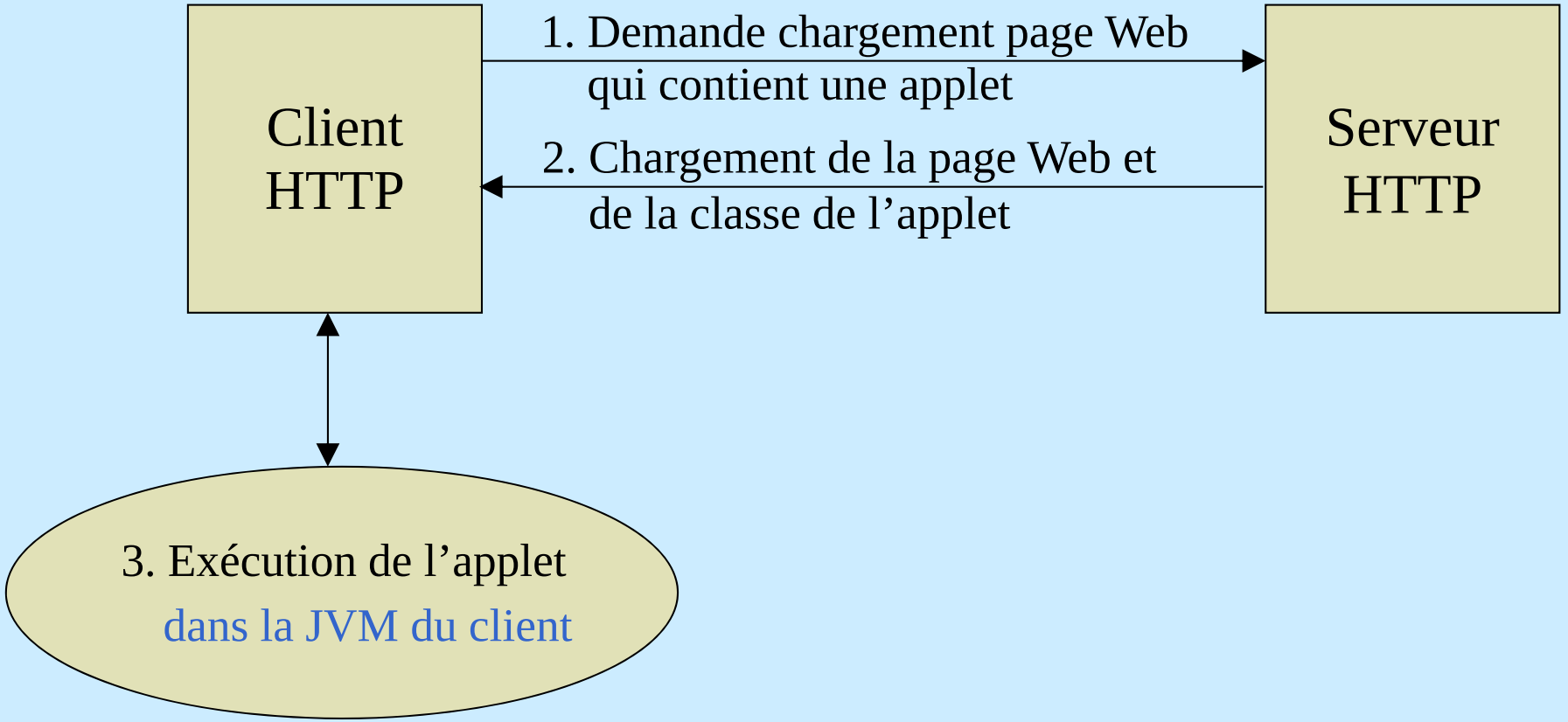
```
import java.awt.Graphics;
import java.applet.Applet;

public class HelloWorldApplet extends Applet {
  public void paint(Graphics g) {
    g.drawString("Hello world", 50, 25);
  }
}
```

Héritage

Pixel où commencera l'affichage : x = 50 pixels, y = 25 pixels

Représente l'emplacement de la page Web où l'applet s'affichera

Étapes pour l’exécution d’une applet



Exécution de l'applet

- Le navigateur a sa propre machine virtuelle
- Un programme Java spécial démarré par le navigateur va lancer certaines méthodes héritées de la classe **`Applet`** : **`init()`**, **`start()`**, **`stop()`**, **`destroy()`**, **`paint()`**
- **`init()`** est exécuté seulement quand l'applet est lancée pour la première fois
- **`paint()`** dessine l'applet dans la page Web

Utilité des *applets*

- Les applets permettent de faire des pages Web plus riches (grâce aux possibilités offertes par Java)
- La page Web peut contenir
  - des animations ou des mises en forme complexes pour mettre en valeur certaines informations
  - des résultats de calculs complexes
  - des informations « dynamiques » (pas connues au moment où la page Web statique est créée) trouvées en interrogeant une base de données
  - ...

Notions de base sur la programmation objet

Langage orienté objet

- Manipule des objets
- Les programmes sont découpés suivant les types des objets manipulés
- Les données sont regroupées avec les traitements qui les utilisent
- Une classe **`Facture`** regroupe, par exemple, tout ce que l'on peut faire avec une facture, avec toutes les données nécessaires à ces traitements

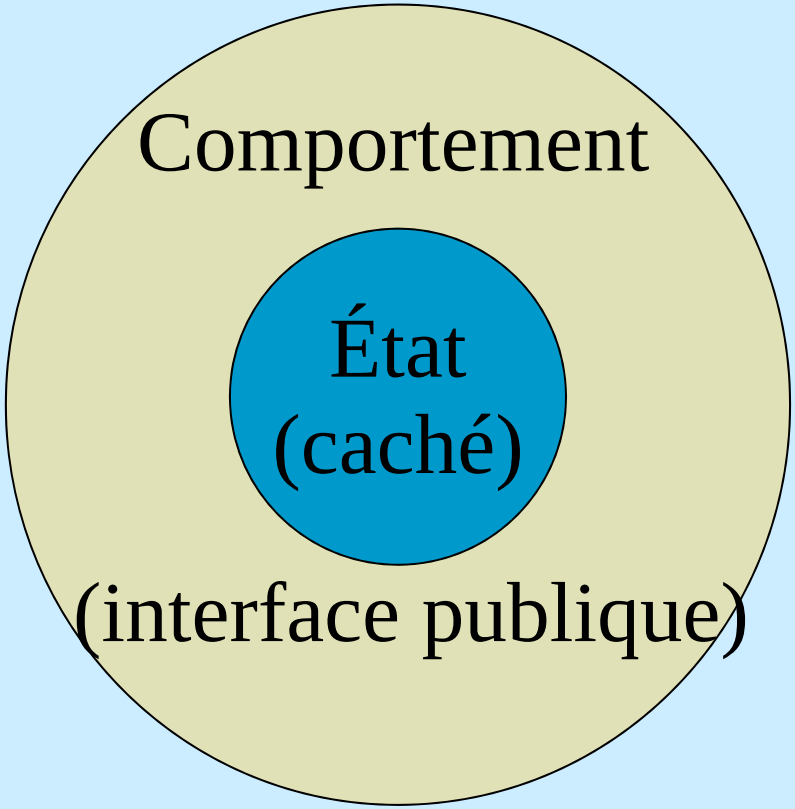
Qu'est-ce qu'un objet ?

- Toute entité identifiable, concrète ou abstraite, peut être considérée comme un objet
- Un objet réagit à certains messages qu'on lui envoie de l'extérieur ; la façon dont il réagit détermine le comportement de l'objet
- Il ne réagit pas toujours de la même façon à un même message ; sa réaction dépend de l'état dans lequel il est

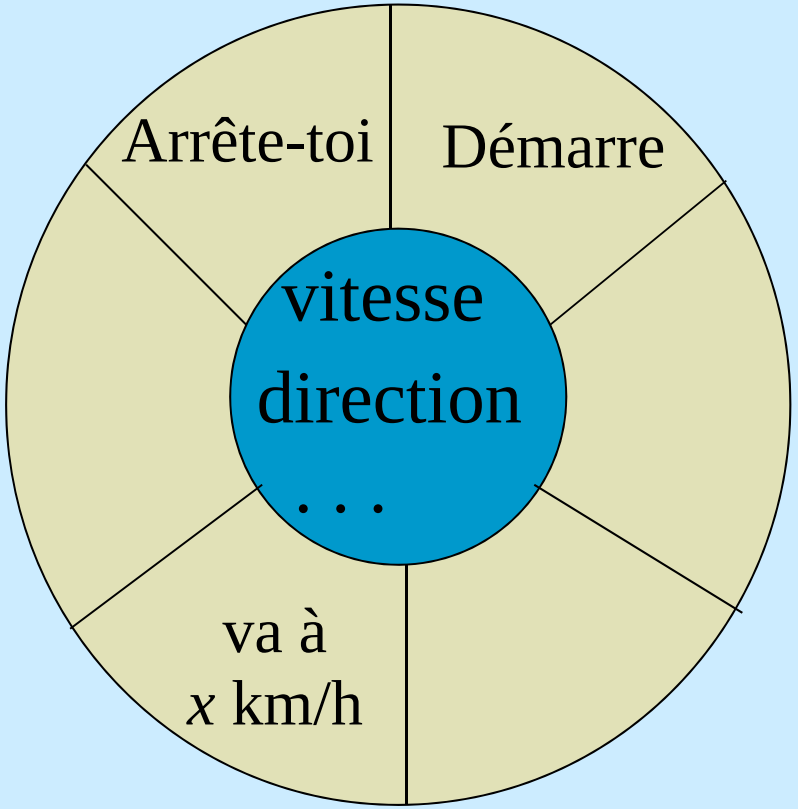
Notion d'objet en Java

- Un objet a
  - une adresse en mémoire (identifie l'objet)
  - un comportement (ou interface)
  - un état interne
- L'état interne est donné par des valeurs de variables
- Le comportement est donné par des fonctions ou procédures, appelées méthodes

Un objet



Un objet



Une voiture

Interactions entre objets

- Les objets interagissent en s'envoyant des messages synchrones
- Les méthodes de la classe d'un objet correspondent aux messages qu'on peut lui envoyer : quand un objet reçoit un message, il exécute la méthode correspondante
- Exemples :

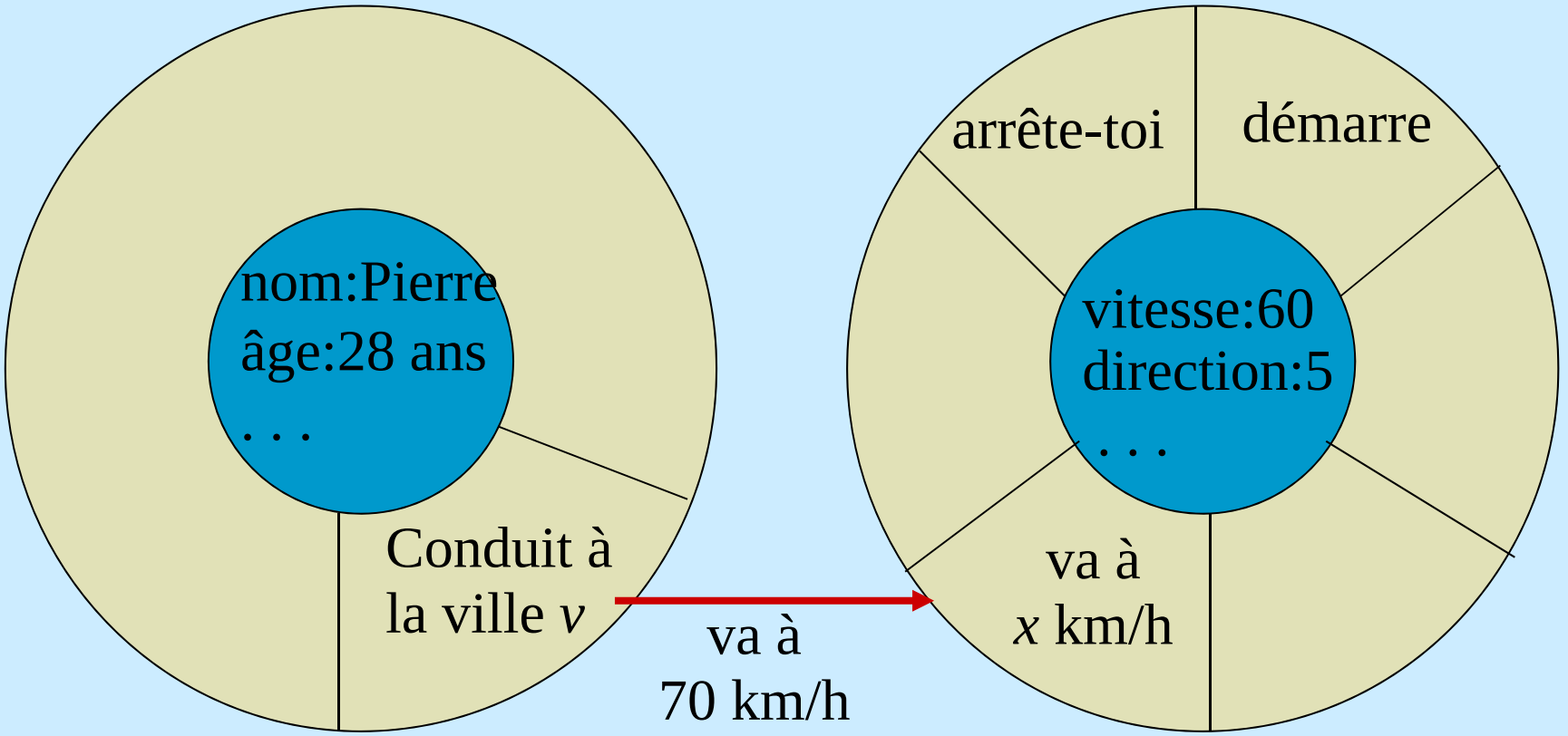
```
objet1.decrisToi();
employe.setSalaire(20000);
voiture.demarre();
voiture.vaAVitesse(50);
```

Objet qui reçoit le message

Message envoyé

Paramètre du message

Messages entre objets



Pierre

La voiture de Pierre

Pierre envoie un message à sa voiture :

```
maVoiture.vaAVitesse(70);
```

Paradigme objet

- La programmation objet est un paradigme, une manière de « modéliser le monde » :
  - des objets ayant un état interne et un comportement
  - collaborent en s'échangeant des messages (pour fournir les fonctionnalités que l'on demande à l'application)
- D'autres paradigmes :
  - programmation impérative (Pascal, C)
  - programmation fonctionnelle (Scheme, Lisp)

Les classes en Java

Regrouper les objets

- Les objets qui collaborent dans une application sont souvent très nombreux
- Mais on peut le plus souvent dégager des types d'objets : des objets ont une structure et un comportement très proches, sinon identiques
- Par exemple, tous les livres dans une application de gestion d'une bibliothèque
- La notion de classe correspond à cette notion de types d'objets

Eléments d'une classe

- Les constructeurs (il peut y en avoir plusieurs) servent à créer les instances (les objets) de la classe
- Quand une instance est créée, son état est conservé dans les variables d'instance
- Les méthodes déterminent le comportement des instances de la classe quand elles reçoivent un message
- Les variables et les méthodes s'appellent les membres de la classe

Exemple : classe Livre

```
public class Livre {
  private String titre, auteur;
  private int nbPages;
  // Constructeur
  public Livre(String unTitre, String unAuteur) {
    titre = unTitre;
    auteur = unAuteur;
  }
  public String getAuteur() {       // accesseur
    return auteur;
  }
  public void setNbPages(int nb) { // modificateur
    nbPages = nb;
  }
}
```

Variables d'instance

Constructeurs

Méthodes

Rôles d'une classe

- Une classe est
  - un type qui décrit une structure (variables d'instances) et un comportement (méthodes)
  - un module pour décomposer une application en entités plus petites
  - un générateur d'objets (par ses constructeurs)
- Une classe permet d'encapsuler les objets : les membres **`public`** sont vus de l'extérieur mais les membres **`private`** sont cachés

Conventions pour les identificateurs

- Les noms de classes commencent par une majuscule (ce sont les seuls avec les constantes) : **`Cercle`**, **`Object`**
- Les mots contenus dans un identificateur commencent par une majuscule : **`UneClasse`**, **`uneMethode`**, **`uneAutreVariable`**
- Les constantes sont en majuscules avec les mots séparés par le caractère souligné « _ » : **`UNE_CONSTANTE`**
- Si possible, des noms pour les classes et des verbes pour les méthodes

Les constructeurs

Classes et instances

- Une instance d'une classe est créée par un des constructeurs de la classe
- Une fois qu'elle est créée, l'instance
  - a son propre état interne (les valeurs des variables d'instance)
  - partage le code qui détermine son comportement (les méthodes) avec les autres instances de la classe

Constructeurs d'une classe

- Chaque classe a un ou plusieurs constructeurs qui servent à
  - créer les instances
  - initialiser l'état de ces instances
- Un constructeur
  - a le même nom que la classe
  - n'a pas de type retour

Création d'une instance

```
public class Employe {
   private String nom, prenom;
   private double salaire;
   // Constructeur
   public Employe(String n, String p) {
     nom = n;
     prenom = p;
   }
   . . .
   public static void main(String[] args) {
     Employe e1;
     e1 = new Employe("Dupond", "Pierre");
     e1.setSalaire(1200);
     . . .
   }
}
```

variables d'instance

création d'une instance de **Employe**

Plusieurs constructeurs (surcharge)

```
public class Employe {
   private String nom, prenom;
   private double salaire;
   // 2 Constructeurs
   public Employe(String n, String p) {
     nom = n;
     prenom = p;
   }
   public Employe(String n, String p, double s) {
     nom = n;
     prenom = p;
     salaire = s;
   }
   . . .
   e1 = new Employe("Dupond", "Pierre");
   e2 = new Employe("Durand", "Jacques", 1500);
```

Désigner un constructeur par `this()`

```
public class Employe {
  private String nom, prenom;
  private double salaire;
  // Ce constructeur appelle l'autre constructeur
  public Employe(String n, String p) {
    this(n, p, 0);
  }
  public Employe(String n, String p, double s) {
    nom = n;
    prenom = p;
    salaire = s;
  }
  . . .
  e1 = new Employe("Dupond", "Pierre");
  e2 = new Employe("Durand", "Jacques", 1500);
```

Constructeur par défaut

- Lorsque le code d'une classe ne comporte pas de constructeur, un constructeur sera automatiquement ajouté par Java
- Pour une classe **`Classe`**, ce constructeur par défaut sera :

```
[public] Classe() { }
```

Même accessibilité que la classe (**`public`** ou non)

Les méthodes

Accesseurs

- Deux types de méthodes servent à donner accès aux variables depuis l'extérieur de la classe :
  - les accesseurs en lecture pour lire les valeurs des variables ; « accesseur en lecture » est souvent abrégé en « accesseur » ; *getter* en anglais
  - les accesseurs en écriture, ou modificateurs, ou mutateurs, pour modifier leur valeur ; *setter* en anglais

Autres types de méthode

- La plupart des méthodes permettent aux instances de la classe d'offrir des services plus complexes aux autres instances
- Enfin, des méthodes (**`private`**) servent de « sous-programmes » utilitaires aux autres méthodes de la classe

Paramètres d'une méthode

- Souvent les méthodes ou les constructeurs ont besoin qu'on leur passe des données initiales sous la forme de paramètres
- On doit indiquer le type des paramètres dans la déclaration de la méthode :

```
setSalaire(double unSalaire)
calculerSalaire(int indice, double prime)
```

- Quand la méthode ou le constructeur n'a pas de paramètre, on ne met rien entre les parenthèses :

```
getSalaire()
```

Type retour d'une méthode

- Quand la méthode renvoie une valeur, on doit indiquer le type de la valeur renvoyée dans la déclaration de la méthode :

```
double calculSalaire(int indice, double prime)
```

- Le pseudo-type **`void`** indique qu'aucune valeur n'est renvoyée :

```
void setSalaire(double unSalaire)
```

Exemples de méthodes

```
public class Employe {
   . . .
   public void setSalaire(double unSalaire) {
     if (unSalaire >= 0.0)
       salaire = unSalaire;
   }
   public double getSalaire() {
     return salaire;
   }
   public boolean accomplir(Tache t) {
     ...
   }
}
```

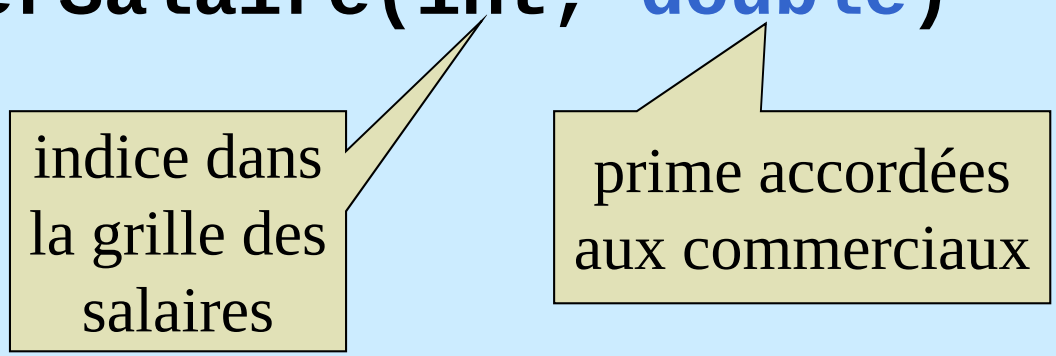
Modificateur

Accesseur

Surcharge d'une méthode

- En Java, on peut surcharger une méthode, c'est-à-dire, ajouter une méthode qui a le même nom mais pas la même signature qu'une autre méthode :

```
calculerSalaire(int)
calculerSalaire(int, double)
```



Surcharge d'une méthode (2)

- En Java, il est interdit de surcharger une méthode en changeant seulement le type de retour
- Autrement dit, on ne peut différencier 2 méthodes par leur type retour
- Par exemple, il est interdit d'avoir ces 2 méthodes dans une classe :

```
int calculerSalaire(int)
double calculerSalaire(int)
```

`toString()`

- Il est conseillé d'inclure une méthode **`toString`** dans toutes les classes que l'on écrit
- Cette méthode renvoie une chaîne de caractères qui décrit l'instance
- Une description compacte et précise peut être très utile lors de la mise au point des programmes
- **`System.out.println(`*`objet`*`)`** affiche la valeur retournée par ***`objet`*`.toString()`**

Exemple

```
public class Livre {
  ...
  public String toString() {
    return "Livre [titre=" + titre
      + ",auteur=" + auteur
      + ",nbPages=" + nbPages
      + "]";
  }
}
```

Les variables

Types de variables

- Les variables d'instances
  - sont déclarées en dehors de toute méthode
  - conservent l'état d'un objet, instance de la classe
  - sont accessibles et partagées par toutes les méthodes de la classe
- Les variables locales
  - sont déclarées à l'intérieur d'une méthode
  - conservent une valeur utilisée pendant l'exécution de la méthode
  - ne sont accessibles que dans le bloc dans lequel elles ont été déclarées

Variable locale ou variable d'instance ?

- Il arrive d'hésiter entre référencer un objet
  - par une variable locale d'une méthode
  - ou par une variable d'instance de la classe
- Si l'objet est utilisé par plusieurs méthodes de la classe, l'objet devra être référencé par une variable d'instance

Déclaration des variables

- Toute variable doit être déclarée avant d'être utilisée
- Déclaration d'une variable : on indique au compilateur que le programme va utiliser une variable de ce nom et de ce type

```
double prime;
Employe e1;
Point centre;
```

Affectation

- L'affectation d'une valeur à une variable est effectuée par l'instruction

  variable = *expression*;

- L'expression est calculée et ensuite la valeur calculée est affectée à la variable
- Exemple :

```
x = 3;
x = x + 1;
```

Initialisation d'une variable

- Une variable doit être initialisée (recevoir une valeur) avant d'être utilisée dans une expression
- Si elles ne sont pas initialisées par le programmeur, les variables d'instance (et les variables de classe étudiées plus loin) reçoivent les valeurs par défaut de leur type (0 pour les types numériques, par exemple)
- L'utilisation d'une variable locale non initialisée par le programmeur provoque une erreur (pas d'initialisation par défaut) à la compilation

Initialisation d'une variable (2)

- On peut initialiser une variable en la déclarant
- La formule d'initialisation peut être une expression complexe :

```
double prime = 200.0;
Employe e1 = new Employe("Dupond", "Jean");
double salaire = prime + 500.0;
```

Déclaration / création

```
public static void main(String[] args) {
  Employe e1;
  e1.setSalaire(1200);
  …
}
```

OK ?

provoque une erreur **`NullPointerException`**

- Il ne faut pas confondre
  - déclaration d'une variable
  - création d'un objet référencé par cette variable
- « **`Employe e1;`** »
  - déclare que l'on va utiliser une variable **`e1`** qui référencera un objet de la classe **`Employe`**,
  - mais aucun objet n'est créé

Déclaration / création (2)

- Il aurait fallu écrire :

```
public static void main(String[] args) {
   Employe e1;
   e1 = new Employe("Dupond", "Pierre");
   e1.setSalaire(1200);
   . . .
}
```

Désigner les variables d'une instance

- Soit un objet **o1** ; la valeur d'une variable **v** de **o1** est désignée par **o1.v**
- Par exemple,

```
Cercle c1 = new Cercle(p1, 10);
System.out.println(c1.rayon); // affiche 10
```

- Remarque : le plus souvent les variables sont **private** et on ne peut pas y accéder directement en dehors de leur classe

Accès aux membres d’une classe

Degrés d’encapsulation

- Java permet plusieurs degrés d’encapsulation pour les membres (variables et méthodes) et les constructeurs d’une classe

Types d'autorisation d'accès

- **`private`** : seule la classe dans laquelle il est déclaré a accès (à ce membre ou constructeur)
- **`public`** : toutes les classes sans exception y ont accès
- Sinon, par défaut, seules les classes du même paquetage que la classe dans lequel il est déclaré y ont accès (un paquetage est un regroupement de classes ; notion étudiée plus loin dans le cours)
- **`protected`** sera étudié dans le cours sur l'héritage

Granularité de la protection des attributs d'une classe

- En Java, la protection des attributs se fait classe par classe, et pas objet par objet
- Un objet a accès à tous les attributs d'un objet de la même classe, même les attributs privés

Protection de l'état interne d'un objet

- Autant que possible l'état d'un objet (les variables d'instance) doit être **`private`**
- Si on veut autoriser la lecture d'une variable depuis l'extérieur de la classe, on lui associe un accesseur, avec le niveau d'accessibilité que l'on veut
- Si on veut autoriser la modification d'une variable, on lui associe un modificateur, qui permet la modification tout en contrôlant la validité de la modification

Exceptions pour les protections des variables (à éviter)

Encore plus rare !

- Dans de **rares cas**, on peut autoriser l'accès au paquetage, ou à tous (**`public`**)
  - si la variable ne risque pas de recevoir des valeurs aberrantes
  - si l'on veut un accès rapide pour améliorer les performances
  - pour simplifier l'écriture du code qui accède à la variable

Désigner l’instance qui reçoit le message, « **`this`** »

`this`

- Le code d'une méthode d'instance désigne
  - l'instance qui a reçu le message (l'instance courante), par le mot-clé **`this`**
  - donc, les membres de l'instance courante en les préfixant par « **`this.`** »
- Lorsqu'il n'y a pas d'ambiguïté, **`this`** est optionnel pour désigner un membre de l'instance courante

Exemple de **`this`** implicite

```
public class Employe {
   private double salaire;
   . . .
   public void setSalaire(double unSalaire) {
      salaire = unSalaire;
   }
   public double getSalaire() {
      return salaire;
   }
   . . .
}
```

Implicitement **`this.salaire`**

Implicitement **`this.salaire`**

`this` explicite

- `this` est utilisé surtout dans 2 occasions :
  - pour distinguer une variable d'instance et un paramètre qui ont le même nom :

```
public void setSalaire(double salaire)
  this.salaire = salaire;
}
```

  - un objet passe une référence de lui-même à un autre objet :

```
salaire = comptable.calculeSalaire(this);
```

Dans quelle classe peut-on trouver ce code ?

Comptable, calcule le salaire de **moi**

Autre exemple de **`this`** explicite

```
public class Document {
  ...
  public void imprimer(Imprimante imprimante) {
    imprimante.ajouterRequete(this);
  }
  ...
```

```
public class Imprimante {
  ...
  public void ajouterRequete(Document doc) {
    // Ajoute le fichier associé au document
    // dans la file d'attente d'impression
    fileAttente.ajouter(doc.getFichier());
    ...
```

Interdit de modifier `this`

- **`this`** se comporte comme une variable **`final`** (mot-clé étudié plus loin), c'est-à-dire qu'on ne peut le modifier ; le code suivant est interdit :
  `this = valeur;`

Méthodes et variables de classe

Variables de classe

- Certaines variables sont partagées par toutes les instances d'une classe. Ce sont les variables de classe (modificateur **`static`**)
- Si une variable de classe est initialisée dans sa déclaration, cette initialisation est exécutée une seule fois quand la classe est chargée en mémoire

Exemple de variable de classe

```
public class Employe {
  private String nom, prenom;
  private double salaire;
  private static int nbEmployes = 0;
  // Constructeur
  public Employe(String n, String p) {
    nom = n;
    prenom = p;
    nbEmployes++;
  }
  . . .
}
```

Méthodes de classe

- Une méthode de classe (modificateur **`static`** en Java) exécute une action indépendante d'une instance particulière de la classe
- Une méthode de classe peut être considérée comme un message envoyé à une classe
- Exemple :

```
public static int getNbEmployes() {
   return nbEmployes;
}
```

Désigner une méthode de classe

- Depuis une autre classe, on la préfixe par le nom de la classe :

```
int n = Employe.getNbEmploye();
```

- Depuis sa classe, le nom de la méthode suffit
- On peut aussi la préfixer par une instance quelconque de la classe (à éviter car cela nuit à la lisibilité : on ne voit pas que la méthode est **`static`**) :

```
int n = e1.getNbEmploye();
```

Méthodes de classe

- Comme une méthode de classe exécute une action indépendante d'une instance particulière de la classe, elle ne peut utiliser de référence à une instance courante (**`this`**)
- Il serait, par exemple, interdit d'écrire

```
static double tripleSalaire() {
   return this.salaire * 3;
}
```

Méthodes de classe

- Une méthode de classe ne peut avoir la même signature qu'une méthode d'instance

Une colle

- La méthode **`main()`** est nécessairement **`static`.** Pourquoi ?
- La méthode **`main()`** est exécutée au début du programme. Aucune instance n'est donc déjà créée lorsque la méthode **`main()`** commence son exécution. Ça ne peut donc pas être une méthode d'instance.

Blocs d'initialisation `static`

- Ils permettent d'initialiser les variables **`static`** trop complexes à initialiser dans leur déclaration :

```
class UneClasse {
  private static int[] tab = new int[25];
  static {
    for (int i = 0; i < 25; i++) {
      tab[i] = -1;
    }
  }
  . . .
```

- Ils sont exécutés une seule fois, quand la classe est chargée en mémoire

Représentation graphique d'une classe en notation UML (*Unified Modeling Language*)

Cercle
private Point centre private int rayon
public Cercle(Point, int) public void setRayon(int) public int getRayon() public double surface()

Cercle
- Point centre **-** int rayon
+ Cercle(Point, int) **+** void setRayon(int) **+** int getRayon() **+** double surface()

(**-** : private, **#** : protected, **+** : public, **$** (ou souligné) : static)

Structure lexicale du langage Java

Codage Unicode pour les programmes

- Pour les identificateurs, les commentaires, les valeurs de type caractère ou chaîne de caractères, Java utilise les caractères du code Unicode
- Le reste d'un programme Java est formé de caractères ASCII (qui sont les 128 premiers caractères du code Unicode)
- Dans un programme, le caractère Unicode dont le code est la valeur hexadécimale **xxxx** peut être représenté par **\uxxxx**

Identificateurs

- Un identificateur Java
  - est de longueur quelconque
  - commence par une lettre Unicode (caractères ASCII recommandés)
  - peut ensuite contenir des lettres ou des chiffres ou le caractère souligné « _ »
  - ne doit pas être un mot-clé ou les constantes **`true`**, **`false`** ou **`null`**

Mots-clés Java

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

*: pas encore utilisé

** : depuis Java SE 5

Commentaires

- Sur une seule ligne :

```
// Voici un commentaire
int prime = 1500; // prime fin de mois
```

- Sur plusieurs lignes :

```
/* Première ligne du commentaire
   suite du commentaire */
```

- Documentation automatique par ***javadoc***

```
/**
 * Cette méthode calcule ...
 * Elle utilise ...
 */
```

Rappels : Quelques principes de programmation

À ne pas oublier !

- Un programme est écrit une fois
- S'il est utile,
  - il sera modifié
  - corrigé
  - on lui ajoutera des fonctionnalités
  - des dizaines (ou centaines) de fois
  - sans doute plusieurs années après son écriture

Ce qu'il faut rechercher

- Une plus grande facilité de programmation
- Mais surtout
  - une maintenance plus aisée
  - et une extensibilité accrue

Comment ?

- Modularité : décomposer en éléments plus simples
- Encapsulation : cacher ce qu'il n'est pas indispensable de voir
- Lisibilité : faciliter la compréhension des programmes
- Réutilisabilité : écrire des modules réutilisables dans les futurs développements (difficile)

Modularité

- Un programme est modulaire s'il est découpé en modules (plus ou moins) indépendants
- Un bon découpage doit satisfaire les 2 critères :
  - forte cohésion des éléments d'un module
  - faible couplage entre deux modules différents
- Ces 2 principes favorisent l'utilisation, la réutilisation et la maintenance des modules :
  - plus de souplesse : un module - une fonctionnalité
  - les modifications d'un module ont le moins d'impacts possible sur les autres modules

Encapsulation

- L'encapsulation est le fait de ne montrer et de ne permettre de modifier que ce qui est nécessaire à une bonne utilisation
  - on montre l'interface (services offerts) d'un module
  - on cache l'implémentation (comment sont rendus les services)
- Les avantages en sont :
  - simplification de l'utilisation (la complexité d'utilisation ne dépend que de l'interface publique)
  - meilleure robustesse du programme
  - simplification de la maintenance de l'application

Attribution des fonctionnalités

- Il peut être difficile de choisir l'objet qui doit être le responsable de l'exécution d'une fonctionnalité
- On peut faire la liste des informations nécessaires à l'exécution
- L'objet qui possède le plus d'informations est souvent le meilleur choix
- Localisation => modularité et encapsulation facilitées

Vers une programmation par composants

- Sur le modèle des circuits électroniques :
  - chaque composant remplit des fonctionnalités bien déterminées et offre des possibilités de connexion avec d'autres composants
  - pas besoin d'aller lire le code pour les utiliser (on ne sait rien de l'implémentation)
- En mieux…
  - on peut configurer le composant
  - et sauvegarder cette configuration, ou un assemblage de composants, pour la réutiliser

Bibliographie - Gratuit

- On peut apprendre Java sans débourser un sou
- Tutoriel d'Oracle à consulter d'abord lorsque l'on étudie une nouvelle API (en anglais) : http://download.oracle.com/javase/tutorial/
- Un bon livre gratuit en ligne (un peu bavard) : « Thinking in Java » de Bruce Eckel ; site Web http://mindview.net/Books/TIJ/DownloadSites ; en français : http://penserenjava.free.fr/

Bibliographie – 2 classiques

- Un livre qui couvre de nombreux paquetages : « Core Java », en 2 volumes mais le premier suffit pour commencer, éditeur Prentice-Hall ; http://www.horstmann.com/corejava.html ; en français : « Au cœur de Java », éditions « Campus Press »
- « Java in a Nutshell » de David Flanagan, éditeur O'Reilly ; en français : « Java en concentré »

Bibliographie – utilisateur avancé

- En anglais : « Effective Java », 2ème édition adaptée à Java SE 6 ; très bon livre pour apprendre à bien programmer en Java ; auteur Joshua Bloch ; « The Java Series », éditeur Addison-Wesley ; seule la 1ère édition a été traduite en français (« Java efficace »)