INF 431

F. Morain

Génie logiciel avec Java

7 février 2007

Délégué(e)s

Groupe	délégué(e)
1 6	HUANG Changyin AGOGUE Marine
2 7	Aijadallah Belabess Daniel Voinéa
3 8	Mouton Sebastien Chu Christophe
4 9	Chardin Rostagno
5 10	? ?

Où en est-on ?

Amphi 1: Introduction.

Amphi 2: génie logiciel avec Java.

Amphi 3: analyse lexicale.

Amphi 4: analyse syntaxique.

Amphi 5: graphes (I).

Amphi 6: graphes (II).

Amphi 7: graphes (III).

Amphi 8: graphes (IV).

Plan

I. Généralités sur le génie logiciel

Sources: cours de P. Dagail au CNAM; le livre de P. A. Darnell & P. E. Margolis, *C: a software engineering approach*.

Navette spatiale: 1200 hommes années, 2200 Kisl (Kilo Instructions Source Livrées), 6 ans.

B777: 4000 Kisl, 7 ans.

Étude US 1997: sur les logiciels développés par l'industrie

- 40 % sont des échecs complets;
- 33 % ont des dépassements de délais;
- 33 % ont des dépassements de coûts;
- 27 % finissent à temps et dans le budget.

Coût des 80,000 projets arrêtés : 81 milliards de dollars (en 1995).

Pourtant: besoin majeur, compliqué par la gestion moderne (outsourcing).

Sofware Almanach

Données rassemblées par *Quantitative Software Management* (QSM) pour l'année 2006 (source: *Communications ACM*, septembre 2006).

564 projets récents, 31 entreprises dans 16 branches dans 16 pays.

Projet moyen:

- < 7 personnes;
- < 8 mois;
- < 58 homme-mois;
- COBOL majoritairement, en passe d'être rattrapé par Java.
- $< 9,200$ lignes.

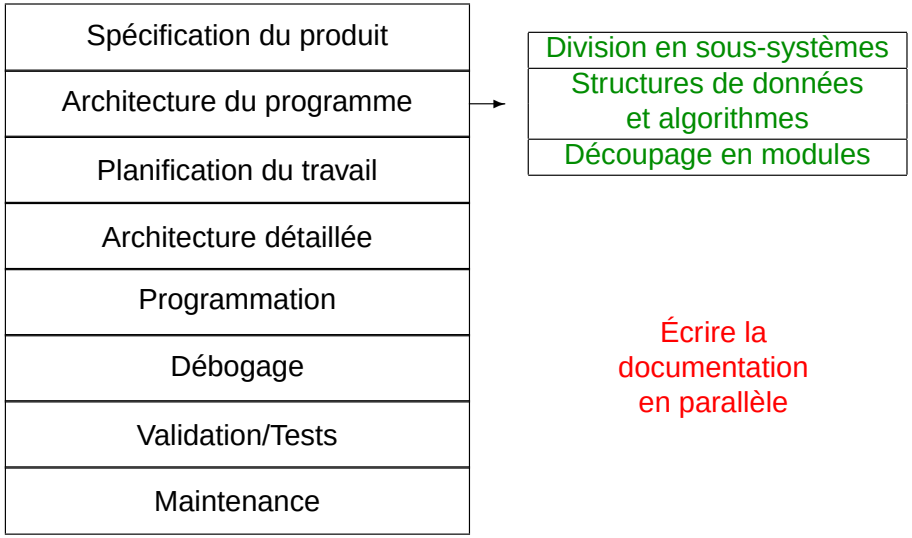
Un programme ressemble à un pont

- Plus le projet est grand, plus il faut soigner l'architecture et le planning. Les problèmes humains ne peuvent être négligés.
- Découvrir les erreurs très vite est essentiel (ou *la découverte tardive est catastrophique*).
- Les erreurs peuvent être désastreuses (Ariane 5 – 1 milliard de dollars).
- Utiliser des préfabriqués permet de gagner du temps.

Un programme n'est pas un pont

- Le logiciel est purement abstrait; il est invisible, car il n'est vu que par son action sur un matériel physique.
- Le logiciel est écrit pour être changé, amélioré.
- Le logiciel est en partie réutilisable.
- Le logiciel peut être testé à tout moment de sa création.

La chaîne de production logicielle



Un logiciel ne se résume pas à la programmation.

Mais: la programmation reste l'endroit où on a le plus de prise sur le produit.

Une architecture excellente peut être gâtée par une programmation médiocre; une excellente programmation ne peut pas rattraper complètement une architecture désastreuse.

Spécification: faite ensemble (attention au chameau).

- Que doit faire le programme? Qui doit l'utiliser?
- Quelles sont les opérations spécifiques ?
- Quel doit être le temps de réponse ?

Commencer à écrire la documentation dès le début, avec mise à jour à chaque fois qu'on écrit une fonction.

Architecture

1. Division en sous-systèmes: interface (entrée des données, formatage des sorties)/moteur (fabricant les données de sortie); bases de données; communications; graphique; etc.
Pourquoi?
   - simplicité:
   - protection contre les changements: surtout dans l'interface;
   - modularité:
   - division du travail: minimiser les interactions.

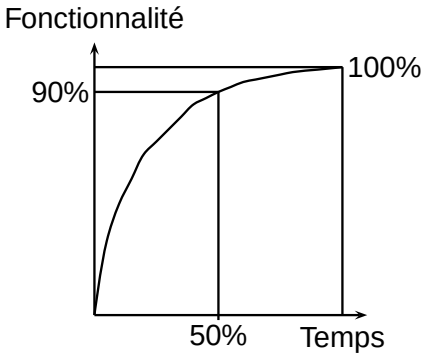
2. Structures de données et algorithmes: cf. cours.

3. Découpage en modules: cf. cours.

Planifi cation du travail

Cf. *The mythical Man-Month* de F. P. Brooks (OS de l'IBM 360, au début des années 1970).
Répartition du temps :

- $1/3$ de spécification;
- $1/6$ de programmation;
- $1/4$ de test (alpha);
- $1/4$ d'intégration et test (beta).



Architecture détaillée

Analyse descendante et raffinage

Ex. Compter le nombre de mots dans un fichier.

- ouvrir le fichier;
- aller au début du fichier;
- tant que la fin du fichier n'est pas atteinte
  - lire un mot;
  - incrémenter le nombre de mots;
- afficher le nombre de mots;
- fermer le fichier.

On peut encore raffiner : comment lire un mot, etc.

Étudier l'architecture soigneusement et vérifier qu'elle marche avant de continuer.

Règles supplémentaires :

- Toujours commencer par le commencement (!);
- oublier Java (temporairement);
- repousser les détails de programmation au plus tard possible;
- ne pas descendre de niveau tant que vous n'êtes pas convaincu(e)s que le niveau actuel est satisfaisant;
- si un problème apparaît, c'est sans doute qu'il trouve sa source au niveau immédiatement supérieur. Remonter et régler le problème.

Le débogage

Déboguer est un art qui demande patience, ingéniosité, expérience, un temps non borné et. . . du sommeil!

Les trois lois du débogage:

- tout logiciel complexe contient des bogues;
- le bogue est probablement causé par la dernière chose que vous venez de modifier;
- si le bogue n'est pas là où vous pensez, c'est qu'il est ailleurs;
- Un bogue algorithmique est beaucoup plus difficile à trouver qu'un bogue de programmation pure.
- On ne débogue pas un programme qui marche!

Méthode scientifique: isoler le bogue et être capable de le reproduire. Sans cela, rien à faire. Très difficile dans des programmes non déterministes (attention aux générateurs aléatoires – mieux vaut les débrancher au départ; parallélisme).

Tester son programme

Alpha test: tests par l'équipe de développement; le code est gelé, seuls les bogues corrigés.

Tests de fonctionnalité du programme: impératifs, font partie du projet.

Beta test: tests par des testeurs sélectionnés et extérieurs à l'équipe de développement.

Si vous avez de vrais ami(e)s prêt(e)s à servir de beta testeu(r|se)s, c'est le rêve. On ne beta teste que quand l'alpha test est terminé.

Écrire des tests n'est pas toujours facile. Ils doivent couvrir tous les cas normaux ou anormaux (boîte de verre, boîte noire, etc.).

Analyse de performance (benchmarks)

Mesurer la vitesse de son programme (chrono de TC.java) ou bien **`time`** d'Unix, quitte à répéter de nombreuses fois les mesures.

On peut écrire un programme de test qui affiche les paramètres pertinents.

On peut tester 2 fonctions et produire deux courbes de temps, qu'il reste à afficher et commenter (**`xgraphic`** ou **`gnuplot`** en Unix).

Si l'algorithme théorique est en $O(n^2)$, on teste avec n, $2n$, $3n$ et on regarde si le temps varie par un facteur 4, 9, . . . Si non, on regarde fonction par fonction.

Il ne reste plus qu'à commenter, déduire, etc.
L'informatique, c'est aussi une science expérimentale!

Rappels sur l'écriture des programmes

- On écrit d'abord pour soi, mais aussi pour les autres (son binôme, un collègue, le prof). On doit pouvoir se relire soi-même (au moins!).
- Choisir un style de programmation et s'y tenir.
- Choisir des noms pertinents pour les types, les variables, etc., ni trop courts, ni trop longs; surtout pour les variables non locales (pensez à votre binôme).
- Une instruction par ligne.
- Indenter le code, c'est-à-dire aligner les instructions de même niveau et les décaler.
- Documenter les méthodes cruciales (entrée, sortie, effets de bord); commenter les passages délicats du code.
- Le programme doit être facilement utilisable (messages).

LES TROPHÉES DES BINÔMES DE L'ANNÉE

PDG/DRH

2006

Et vous dans tout ça: un projet en binôme (!!)

Comment Java peut vous aider

On cache de l'information (autre nom de la programmation objet). Pas de clash entre noms, si les deux membres du binôme écrivent leurs propres classes.

Un schéma possible:

```
class B1a{...}
class B1b{...}
...
class TestB1{
... main(){
...
}}
```

↔

```
class Com1{
...}
class Com2{
...}
...
class Test{
... main(){
...
}
class Projet{
... main(){
}}
```

↔

```
class B2a{...}
class B2b{...}
...
class TestB2{
... main(){
... }
}
```

II. Modularité

Pourquoi?

- structure de bloc dans les langages de programmation
  ⇒ variables locales ou globales;
- insuffisant, car les variables communes à plusieurs fonctions deviennent globales
  ⇒ accessibles depuis l'extérieur.

⇒ constructions spéciales dans les langages de programmation:

- modules ou classes;
- données et fonctions privées ou publiques.

⇒ un programme sera découpé en modules.

Modules

- un module a deux parties:
  - l'interface accessible depuis l'extérieur du module
  - l'implémentation accessible de l'intérieur du module
- principe d'encapsulation
  - Une fi le a 3 opérations: construction, ajouter, supprimer
  - Deux implémentations possibles: tableau circulaire, liste
  - L'extérieur n'a pas à connaitre l'implémentation
  - On peut changer l'implémentation sans que l'extérieur ne s'en rende compte
- on peut composer les modules pour en faire de plus gros

Modularité en Java

Considérons l'exemple des files d'attente.
Interface publique: **`FIFO`**, **`ajouter`** et **`supprimer`**.
Implémentation privée

```
public class FIFO{
  private int      debut, fin;
  private boolean  pleine, vide;
  private int[]    contenu;

  public FIFO(int n){
    debut = 0; fin = 0;
    pleine = n == 0; vide = true;
    contenu = new int[n];
  }
```

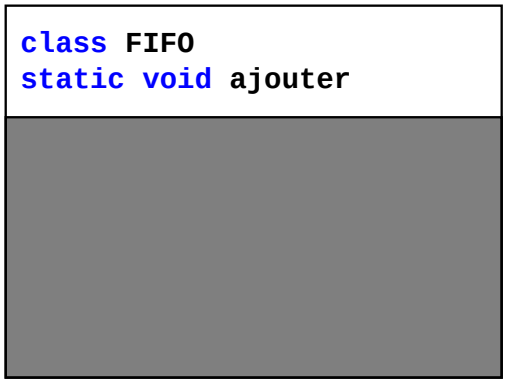


```
public static void ajouter(int x, FIFO f){
  if(f.pleine)
      throw new Error ("File Pleine.");
  f.contenu[f.fin] = x;
  f.fin = (f.fin + 1) % f.contenu.length;
  f.vide = false; f.pleine = f.fin == f.debut;
}

public static int supprimer(FIFO f){
  if (f.vide)
      throw new Error ("File Vide.");
  int res = f.contenu[f.debut];
  f.debut =  (f.debut + 1) % f.contenu.length;
  f.vide = f.fin == f.debut; f.pleine = false;
  return res;
}
```

« interface »

implementation



L'« interface » public est visible, l'implémentation est opaque.

En Java, le mot-clé `interface` a une autre signifi cation. Dans certains langages de programmation (Modula, ML), il y a une distinction entre interfaces et implémentations, permettant la compilation séparée.


```
public class FIFO {
  private Liste debut, fin;
  public FIFO(int n){ debut = null; fin = null; }
  public static void ajouter(int x, FIFO f){
    if (f.fin == null) f.debut = f.fin = new Liste (x);
    else {
      f.fin.suivant = new Liste (x);
      f.fin = f.fin.suivant;
    }
  }
  public static int supprimer(FIFO f){
    if (f.debut == null) throw new Error ("File Vide.")
    else {
      int res = f.debut.val;
      if (f.debut == f.fin)  f.debut = f.fin = null;
      else f.debut = f.debut.suivant;
      return res;
    }
  }
```

⇒ Mêmes signatures pour les variables et fonctions publiques dans les deux implémentations (constructeur compris).

Paquetages et espace des noms

- En Java, structures de données = objets (ou tableaux).
  ⇒ Beaucoup de classes, de fichiers `.class`,
  ⇒ risques de collisions dans l'espace des noms.
- les classes sont regroupées en paquetages.

```
package ma_lib1;          package ma_lib2;
public class FIFO{...}    public class FIFO{...}
```

- On importe un paquetage pour l'utiliser:

```
import ma_lib1.FIFO;
class Test{
  public static void main(String[] args){
    int n = Integer.parseInt(args[0]);
    FIFO f = new FIFO(n);
    FIFO.ajouter(1, f);
  }
}
```


- L'instruction

  package $\mathrm{id}_1.\mathrm{id}_2.\ldots.\mathrm{id}_k$**;**

  qualifie les noms des champs publics d'une unité de compilation
- exemples de noms qualifiés:
  - **ma_lib1.FIFO**, **ma_lib1.FIFO.ajouter**, **ma_lib1.FIFO.supprimer** (premier paquetage)
  - **ma_lib2.FIFO**, **ma_lib2.FIFO.ajouter**, **ma_lib2.FIFO.supprimer** (deuxième paquetage)
- on référence ces champs publics avec leurs noms qualifiés (classe **Test**)
- ou avec une forme courte si on importe la classe avant son utilisation

  import $\mathrm{id}_1.\mathrm{id}_2.\ldots.\mathrm{id}_k.C$**;**

  où C est un nom de classe ou le symbole ***** pour importer toutes les classes d'un paquetage.

- L'emplacement des paquetages dépend de deux paramètres:
  - le nom $id_1.id_2.\ldots.id_k$ qui désigne l'emplacement $id_1/id_2/\ldots/id_k$ dans l'arborescence des répertoires du système de fi chiers.
  - la variable d'environnement **`CLASSPATH`** qui donne une suite de racines à l'arborescence des paquetages.

- forme compressée des répertoires: fichiers **`.jar`** (*java archives*). Ils contiennent une arborescence de paquetages en un seul fichier (cf. la commande Unix **`jar`**). Les fichiers **`.zip`** sont aussi utilisables.

- La valeur de **`CLASSPATH`** est fixée par une commande Unix:

  `setenv CLASSPATH ".:$HOME/if431/DM1:/users/profs/info/chas`

  (pour **`csh`**, **`tcsh`**) ou

  `export CLASSPATH=".:$HOME/if431/DM1:/users/profs/info/chas`

  (pour **`sh`**, **`bash`**).

- Un fichier **`.java`** démarre souvent comme suit:

```
package ma_lib1;
import java.util.*;
import java.io.*;
import java.awt.*;
```

  Cette unité de compilation fera partie du paquetage **`ma_lib1`** et importe les paquetages:
  - **`java.util`** qui contient des classes standards pour les tables, les piles, les tableaux de taille variable, etc.
  - **`java.io`** qui contient les classes d'entrées-sorties.
  - **`java.awt`** qui contient les classes graphiques.
  - **`java.math`** qui contient les classes mathématiques.
- Cf. la liste des paquetages *Les classes de Java* dans la page web du cours.
- Si aucun paquetage précisé, l'unité de compilation fait partie du paquetage anonyme localisé dans le répertoire courant.

Contrôle d'accès

Pour les membres d'une classe, il y a 3 types d'accès:

- **`public`** pour permettre l'accès depuis toutes les classes.
- **`private`** pour restreindre l'accès aux seules expressions ou fonctions de la classe courante.
- par défaut pour autoriser l'accès depuis toutes les classes du même paquetage.

Pour les membres d'un paquetage, il y a 2 types d'accès:

- une classe publique peut être accédée de l'extérieur de son paquetage
- une classe sans qualificatif n'est accessible que depuis son paquetage;
- de l'extérieur du paquetage, double-protection: classe $+$ champ doivent être publics;
- Remarque: le *class loader* ne vérifie pas cette discipline, car il accède à la méthode **`main`** sans que sa classe ne soit publique!
- une seule classe publique C par unité de compilation
  $\Rightarrow$ une seule classe publique par fichier C`.java`
  (Certains compilateurs sont laxistes. Mais mieux vaut privilégier la portabilité)

III. Programmation par objets

- programmation procédurale $\simeq$ Fortran, Algol, Pascal, C, Ada, ML, Caml, Haskell, programmation fonctionnelle.
- programmation dirigée par les données $=$ programmation par objets, Simula, Smalltalk, C++, Eiffel, Java, Ocaml.

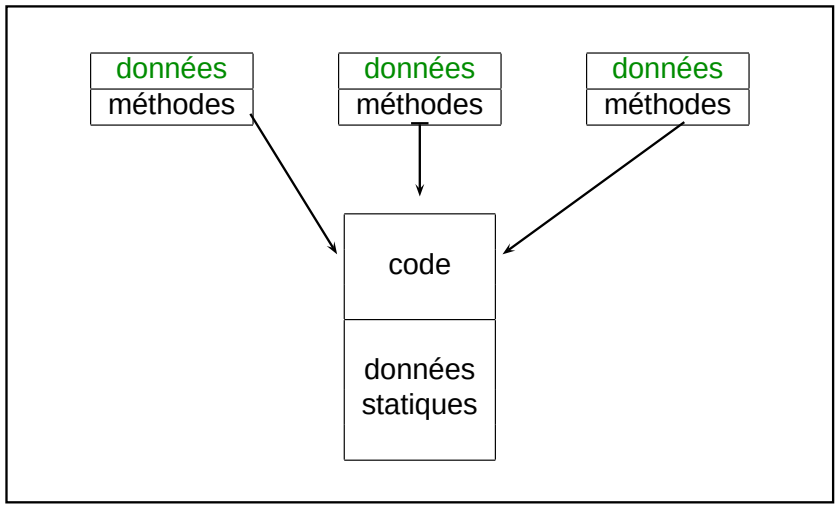
	modification des données	modification du programme
programmation procédurale	changement global	changement local
programmation par objets	changement local	changement global

Pourquoi la programmation par objets?

- Modularité: les objets favorisent le regroupement des fonctions et des données $\Rightarrow$ structuration.
  Rem. Modularité $\not\Rightarrow$ programmation par objets (cf. Modula, ML).
- Programmation incrémentale: on définit des classes qu'on peut étendre par héritage *a posteriori* (e.g. AWT).

Objets en Java

Les objets sont les instances d'une classe. Ils ont un état (la valeur de leurs champs de données) et un ensemble de méthodes attachées.



IV. Héritage

```
class Point{
  double x, y;
}
class PointC extends Point{
  final static int JAUNE = 0, ROUGE = 1;
  int c;
  PointC(double x0, double y0, int col){
    x = x0; y = y0; c = col;
  }
  public static void main(String[] args){
    PointC pc = new PointC(3, 4, JAUNE);
  }
}
```

La classe **PointC** est une sous-classe de **Point**. Elle hérite des champs définis dans sa classe père.

Propriétés de l'héritage

- une classe peut être sous-classe d'une autre classe et d'une seule (héritage simple); une classe peut avoir plusieurs sous-classes;
- une classe hérite des méthodes (de classe, d'objets), des variables de classes de la classe père;
- dans la sous-classe, on ajoute ou on redéfinit des champs ou méthodes: on dit qu'on a spécialisé ces champs ou méthodes;
- quand une méthode est appelée, on choisit statiquement la méthode la plus spécialisée, c'est-à-dire appartenant à la plus petite sous-classe connue contenant l'objet; il n'y a pas d'ambiguité grâce à l'héritage simple.

Rem. En C++, Smalltalk ou Ocaml, l'héritage est multiple.


```
class Point{
  double x, y;
  static int NP = 1;
  static void afficherAbscisse(Point p){
    System.out.println("P: "+p.x);
  }
}
class PointC extends Point{
  final static int JAUNE = 0, ROUGE = 1;
  int c;
  PointC(double x0, double y0, int col){
    x = x0; y = y0; c = col;
  }
  public static void main(String[] args){
    PointC pc = new PointC(3, 4, JAUNE);
    afficherAbscisse(pc);
    System.out.println("NP="+NP);
  }
}
```

Conversion implicite de **`PointC`** en **`Point`**.


```
class Point{
  ...
  static void afficherAbscisse(Point p){
    System.out.println("Point: "+p.x);
  }
}
class PointC extends Point{
  ...
  static void afficherAbscisse(Point p){
    System.out.println("PointC: "+p.x);
  }
  public static void main(String[] args){
    PointC pc = new PointC(3, 4, JAUNE);
    afficherAbscisse(pc);
    Point.afficherAbscisse(pc);
  }
}
```


Exemple de spécialisation

```
class Point{
  ...
  void afficher(){
    System.out.println("p: "+x);
  }
}
class PointC extends Point{
  ...
  void afficher(){
    System.out.println("pc: "+x);
  }
  public static void main(String[] args){
    PointC pc = new PointC(3, 4, JAUNE);
    Point p = new Point();
    p.x = 1;

    pc.afficher();
    p.afficher();
  } }
```

Utilisation de **super**

- Héritage simple, chaque classe a une seule classe parente, accessible par le mot clef **super** (un peu comme **(ClassePere)this**); on ne peut l'utiliser que dans des méthodes d'objets;
- **super.f** est la méthode **f** de la classe parente;
- **super()** (avec d'éventuels arguments) est le constructeur (appelé par défaut) de la classe parente;

```
class PointC extends Point{
  final static int JAUNE = 0, ROUGE = 1;
  int c;
  PointC(double x0, double y0, int col){
    super(); // facultatif
    c = col;
  }
  public static void main(String[] args){
    PointC pc = new PointC(3, 4, JAUNE);
  } }
```



```
class Point{
  double x, y;
  Point(double x0, double y0){
    x = x0; y = y0;
  }
}
class PointC extends Point{
  final static int JAUNE = 0, ROUGE = 1;
  int c;
  PointC(double x0, double y0, int col){
    super(x0, y0); // nécessaire
    c = col;
  }
  public static void main(String[] args){
    PointC pc = new PointC(3, 4, JAUNE);
  }
}
```


Contrôle d'accès et sous-classes

Les champs ou méthodes d'une classe peuvent être:

- **public** pour permettre l'accès depuis toutes les classes,
- **private** pour restreindre l'accès aux seules expressions ou fonctions de la classe courante,
- par défaut pour autoriser l'accès depuis toutes les classes du même paquetage,
- **protected** pour restreindre l'accès aux seules expressions ou fonctions de sous-classes de la classe courante.
- un champ **final** ne peut être redéfini (on peut donc optimiser sa compilation).
- une classe **final** ne peut être spécialisée.
- une méthode peut être déclarée **final** (pas besoin de déclarer toute la classe **final**).

Résumé

- une classe contient un ensemble de champs (données ou fonctions)
- une sous-classe contient au moins tous les champs de sa classe parente, et de nouveaux champs.
- une sous-classe peut redéfinir (spécialiser) un certain nombre de champs de sa classe parente (*overriding*).
- les méthodes redéfinies doivent avoir la même signature (arguments et résultat) que dans la sur-classe.
- les méthodes redéfinies doivent fournir au moins les mêmes droits d'accès.

V. Sous-typage et héritage

Quelques notions de typage:

- Un type est un ensemble de valeurs partageant une même propriété.
- Types primitifs: **`int`**, **`char`**, etc.
- En Java, une classe est un type.
- Classes prédéfinies: **`String`**, etc.

Vérification du typage: statique (à la compilation) ou dynamique (à l'exécution).

Intérêt du typage statique: détection d'erreurs, optimisation de code, sécurité (pas d'atteinte à la mémoire).

Intérêt du typage dynamique: plus précis (**`if(a) return 1; else return false;`**).
La théorie des types est introduite dans le cours Principes des Langages de Programmation de Majeure 1. Cf. livre de Benjamin Pierce ou de Abadi-Cardelli.

Le typage en Java

- En Java, chaque objet possède un certain type lors de sa création, qu'il conserve pendant toute sa durée de vie (contrairement à Pascal, ML, C, C++).
- L'opérateur **`instanceof`** teste l'appartenance d'un objet à une classe. Ainsi:

```
if(p instanceof PointC)
  System.out.println ("couleur = " + p.c);
```

- Donc l'expression **`(PointC)p`** équivaut à:

```
if(p instanceof PointC)
  p
else
  throw new ClassCastException();
```


Sous-classe et sous-typage

Conversion implicite: $t <: t'$ signifie $x : t \Rightarrow x : t'$ pour tout x. On dit que t est un sous-type de t'.

En Java:
`byte` $<:$ **`short`** $<:$ **`int`** $<:$ **`long`**
`float` $<:$ **`double`**
`char` $<:$ **`int`**

Une classe est un type; une sous-classe un sous-type. On propage la notation: $C <: C'$ si C est une sous-classe de C'.

Ex. `PointC` $<:$ **`Point`**.

Racine de la hiérarchie des classes

`Object` est la classe la plus générale:

$$\forall C \quad C <: \texttt{Object}$$

(C est une classe ou un tableau quelconque).

- La hiérarchie des classes est une simple arborescence.
- Les méthodes de **`Object`** sont **`clone`**, **`finalize`**, **`getClass`**, **`hashCode`**, **`notify`**, **`notifyAll`**, **`wait`**, **`equals`**, **`toString`**.
- Seules **`equals`**, **`toString`**, **`finalize`** peuvent être redéfinies.
- Tous les objets contiennent ces méthodes.

- On convertit les scalaires **`int`**, **`float`**, **`char`**, ... en objets avec un « conteneur »:

```
int x = 3;
Objet xobj = new Integer(x);
int y = xobj.intValue();
```

Type réel et type apparent

```
class A{
  int x;

  A(){ x = 1; }
}
class B extends A{
  B(){ x = 2; }
  public static void main(String[] args){
    B b = new B();
    A a = new B();
    System.out.println(b.x);
    System.out.println(a.x);
  }
}
```

Le type réel de **a** est **B**, mais son type apparent est **A**; **a** n'a pas tout oublié de ses origines.

Retour aux points

```
class Point{
  double x, y;
  static void translation(Point p,
                          double dx, double dy){
    p.x += dx; p.y += dy;
  } }
class PointC extends Point{
  final static int JAUNE = 0, ROUGE = 1;
  int c;
  PointC(double x0, double y0, int col){
    x = x0; y = y0; c = col;
  }
  public static void main(String[] args){
    PointC pc = new PointC(3, 4, JAUNE);
    translation(pc, 1, 2);
  } }
```

Conversion implicite + type apparent.

Ce qui ne peut marcher

```
class Point{
  ...
  static Point translation(Point p,
                           double dx, double dy){
    Point pt = new Point();
    pt.x = p.x + dx; pt.y = p.y + dy;
    return pt;
  } }
class PointC extends Point{
  ...
  public static void main(String[] args){
    PointC pc = new PointC(3, 4, JAUNE);
    pc = (PointC)translation(pc, 1, 2);
  } }
```

Pas de conversion père → fils. La compilation marche (avec le cast), mais pas l'exécution (**ClassCastException**).

Héritage et surcharge

- la surcharge est déterminée à la compilation;
- Liaison retardée: l'héritage permet d'appeler une méthode de la même classe sans savoir exactement quelle sous-classe va l'instancier.

Ex. Quelle est la valeur affichée par le programme suivant?

```
class C{
  void f() { g(); }
  void g() { System.out.println(1); } } }
class D extends C{
  void g() { System.out.println(2); }
  public static void main(String[] args){
    D x = new D();
    x.f();
} }
```

VI. Classes abstraites – Interfaces

```
abstract class Aliment{
  String nom;
  abstract String modeDeConsommation();
  void info(){
    System.out.println(this.modeDeConsommation());
  } }
class Pain extends Aliment{
  String modeDeConsommation(){
    return "cru ou cuit";
  } }
class FoieGras extends Aliment{
  String modeDeConsommation(){
    return "cru ou cuit, avec un verre de Sauternes";
  } }
class Menu{
  public static void main(String[] args){
    (new Pain()).info();
    (new FoieGras()).info();
  } }
```


Propriétés

- Une classe abstraite contient des champs indéfinis.
- On ne peut pas créer d'objets de classes abstraites.
- Permet d'utiliser des types disjonctifs (comparer avec les types somme de Caml, les variantes de Pascal; les **union** de C).

Interfaces

- Une interface est une classe abstraite dont tous les champs sont indéfinis.
  Ses champs de données sont constants;
  ses méthodes sont abstraites et publiques.
- Une interface peut spécialiser une autre interface.
- Une classe peut implémenter une ou plusieurs interfaces.
- Notion différente des interfaces de Modula, ML, Ada, Mesa. En Java, la classe qui les implémente porte un nom différent.
- Pas de fonctions statiques, ou champs de données modifiables dans un interface.
- Les interfaces sont une bonne manière d'exiger la présence de certains champs dans une classe.


```
public interface Couleur{
  final int ROUGE = 0, JAUNE = 1;
}
interface PointTournant{
  void static rotation(Point p, double theta);
}
interface PointMobile{
  void static translation(Point p,
                          double dx, double dy);
}
class Point implements PointTournant, PointMobile{
  ...
}
```

Polymorphisme (depuis Java 1.5)

GJ ([Bracha, Odersky, Stoutamire, Wadler, 1997]).

- on peut avoir des variables de types

```
public class LinkedList<E> {
  void add(int i, E x) { ... }
  E get(int i) { ... }
  E getFirst() { ... }
  ListIterator<E> listIterator(int i) { ... }
}
public interface ListIterator<E>
       extends Iterator<E>
  boolean hasNext();
  E next();
}
```

- génériques de Java $\neq$ génériques de C++ (templates) pas de duplication du code;
- on ne peut pas écrire `E x = new E();` ou `E[] t = new E[10];`
- polymorphisme total $\Rightarrow$ ML, Caml, Haskell.

Un exemple pour consolider

Contexte: écrire une base de données des élèves de l'X.

```
class Date{
  int jour, mois, annee;
  Date(int j, int m, int a){
    this.jour = j;
    this.mois = m;
    this.annee = a;
  }
  int compareTo(Date d){
    if(annee > d.annee) return 1;
    if(annee < d.annee) return -1;
    if(mois > d.mois) return 1;
    if(mois < d.mois) return -1;
    if(jour > d.jour) return 1;
    if(jour < d.jour) return -1;
    return 0;
  }
}
```

Rem. On aurait pu utilisé la classe `Date`.


```
class Humain{
  char genre;
  Date date_de_naissance;
  String nom;
  String prenoms;
  String adn;

  public String toString(){
    return nom + ", " + prenoms;
  }
}
class EleveDeLX extends Humain{
  String promotion_d_entree;
  int rang_d_entree;
  int promotion_de_sortie, rang_de_sortie;
  String[] sports;
  String[] binets;
  String[] notes;
```



```
  EleveDeLX(String n, String p, Date d,
            String promo, int r){
    this.nom = n;
    this.prenoms = p;
    this.date_de_naissance = d;
    this.promotion_d_entree = promo;
    this.rang_d_entree = r;
  }

  public String toString(){
    return super.toString()
              + ", " + promotion_d_entree;
  }
```

```
  // si date > 100 ans : état civil
  // si date > 120 ans : scolarité
  String infos(Date d){
    d.annee -= 100;
    if(d.compareTo(date_de_naissance) <= 0)
      return null;
    String s = this.toString();
    d.annee -= 20;
    if(d.compareTo(date_de_naissance) > 0)
      return s + notes[0];
    else
      return s;
  }
}
```


La classe de test peut contenir quelques informations sur Pierre Faurre.

```
class TestX{
  public static void main(String[] args){
    EleveDeLX PFaurre =
            new EleveDeLX("Faurre", "Pierre",
                          new Date(15, 1, 1942),
                          "X1960", 1);
    System.out.println(PFaurre);
    System.out.println(
           PFaurre.infos(new Date(7, 2, 2007)));
  }
}
```


Procédural ou Objets?

- choix du contrôle: par les fonctions ou par les données?
- le style de programmation diffère entre petits programmes et gros programmes ($> 10^4$ lignes).
- dépend de la stratégie de modification.
- en dernière analyse, c'est affaire de goût.
- objets $\neq$ modularité.
- programmation par objets utilisée dans les boîtes à outils graphiques (*look* commun à toutes les fenêtres) et réseau (données et méthodes se déplacent simultanément).
- programmation incrémentale très populaire dans l'industrie.

Héritage ou pas?

Principe de base: on n'hérite pas si on ne spécialise rien ou si on n'ajoute rien à la classe père.

$\Rightarrow$ pas de **`extends`** pour le plaisir.

Cas le plus utile: on définit une classe abstraite et on construit des classes concrètes (cf. `grapheX`).

Résumé du cours

- Quelques données sur le génie logiciel.
- Modularité; héritage, typage, etc.
- Classes abstraites.
- Attention à la programmation objet, elle recèle de nombreux pièges.
- Dans la suite du cours: on utilise surtout les classes prédéfinies, et on peut décoder leurs définitions.
- Vous pouvez programmer à l'ancienne, ou bien vous lancer dans la modernité échevelée. Nous resterons en général conservateur dans le domaine.

Prochains rendez-vous: TD cet après-midi; amphi 03 mercredi prochain 14/02.