

Les types de données abstraits


Des types aux interfaces


Le typage

- ◆ Nous avons déjà rencontré les types (primitifs) dans la programmation

- `int i; bool b;`

- `b = (i + 1);` 

- `b = (i == 1);` 


- `b = (b + 1);` 

- ◆ Les types renforcent une notion de validité

- Question : *lien entre les types et les classes ?*

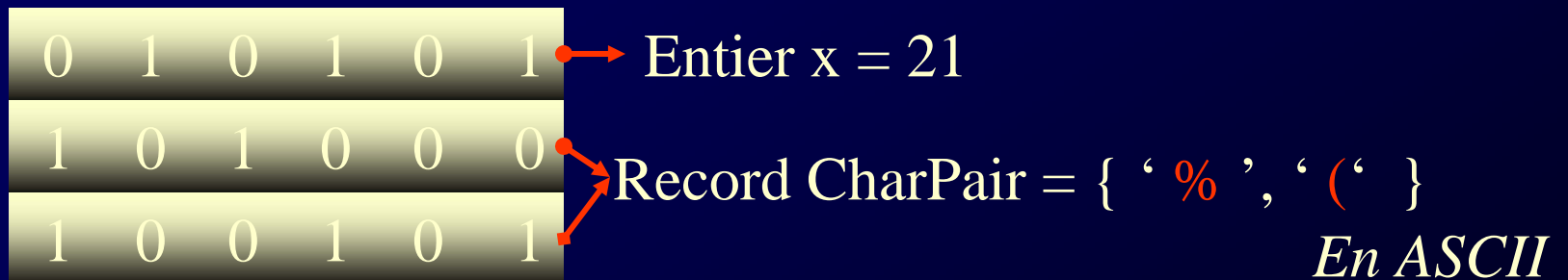
- `Cercle c = new Cercle(2,6,9.0);` 

- `c.dessiner();` 

- `c=7;` 

Représentation des données

- ◆ Toute donnée en mémoire consiste en une séquence de bits (et d'octets)



- ◆ Le programmeur associe une *signification* à toute donnée
 - Entier, tableau de caractères

Type

- ◆ Cette signification définit
 - Les valeurs légales pour une variable
 - Les opérations applicables aux valeurs
 - C-à-d, le type
- ◆ P.ex. le type `int` comprend
 - Les valeurs {`min`, ..., `max`}
 - Les opérations {`+`, `==`, `-`, `*`, `!=` }
 - Et division ?
 - Cela dépend du langage de programmation

Utilité du typage

◆ Sécurité

- Éviter certaines erreurs de programmation
 - `bool b = 3.0; /* erreur de typage */`
 - `int i,j; j = 0; i= (i div j) /*erreur de typage ?*/`

◆ Génie logiciel

- Programmer mieux
- Important pour l'interopérabilité de composants de sources différentes

Un système de types

- ◆ Un langage typé associe un type à chaque variable d'un programme
 - La plupart des langages qu'on connaît: C, Pascal, C++, Eiffel, Java
- ◆ *Un système de types*
 - Détermine le type de toute variable ou expression dans un programme,
 - Et signale les erreurs de type
 - Un système de types est intégré dans le compilateur du langage ou dans son environnement d'exécution

Les règles de typage

- ◆ Tout langage possède ses règles de typage
- ◆ Définissent la validité des expressions d'un langage
 - 1. $\text{int} + \text{int} \rightarrow \text{int}$
 - 2. $0 * \text{int} \rightarrow 0$
 - 3. $(\text{int} = \text{int}) \rightarrow \text{bool}$
 - P.ex: $a=3+6$ est légal si a est du type int par la règle 1
- ◆ La vérification de typage démontre que le programme respecte les règles
- ◆ La vérification est effectuée par le compilateur ou par le run-time

Vérification de typage

- ◆ Statique
 - Vérification s'effectue lors de compilation
 - p. ex. C++, Eiffel
- ◆ Dynamique
 - Vérification à l'exécution
 - P.ex. SmallTalk
- ◆ Java utilise la vérification statique et la vérification dynamique

Typage statique

- ◆ Associe un type à une variable
- ◆ Pas de vérification à l'exécution
 - Toute la vérification est faite à la compilation
- ◆ Restrictif : spécifier les types à la programmation

```
Proc MinMax(int a, b)  
  if a < b then return (a,b);  
  else return (b,a);
```

```
Proc MinMax(Complex a,b)  
  if a < b then return (a,b);  
  else return (b,a);
```

- Un programme qui opère sur des données de types différents est utile pour éviter du code redondant !

Typage dynamique

- ◆ Dynamique

- Type associé à l'objet, pas à la variable
 - Ce programme est correct si le type de a et de b comprend l'opération <

```
Proc MinMax(a,b)
  if type_of(a) != (Complex or Int) then Error;
  if a < b then return (a,b);
  else return (b,a);
```

- ◆ Le typage dynamique s'avère plus coûteux à l'exécution, mais plus souple !
- ◆ Java prend une approche entre les deux extrêmes

Les langages non-typés

- ◆ Tous les langages ne sont pas typés
 - P.ex. Le code machine ne possède pas la notion de type
 - toute donnée n'est que séquence d'octets
- ◆ Avantages des langages non-typés
 - Programmation de « bas niveau »
 - C-à-d, on peut manipuler des données de façon uniforme indépendamment de leurs significations
 - P.ex: pour programmer le *swap* au sein d'un OS
 - Efficacité du langage

Le cadre orienté-objet

♦ Java

- Type Primitifs:
 - byte, int, short, long
 - float, double
 - boolean, char
- Types Composés
 - Classes
 - class Cercle, class Carre
 - Tableaux
 - char[]
 - Cercle[], Cercle[][]

Types de données abstrait

◆ Type Abstrait

- Opérations ou Services fournis sur des structures de données
- Propriétés de ces opérations
 - Surface retourne l'aire
 - X, Y retournent les coordonnées
 - Un objet de la classe Cercle possède le type (abstrait) Cercle
- Permet d'obtenir les qualités attendues d'un logiciel d'aujourd'hui (extensibilité, réutilisabilité, compatibilité)

Cercle

Opérations

Surface

X

Y

Le cadre orienté-objet

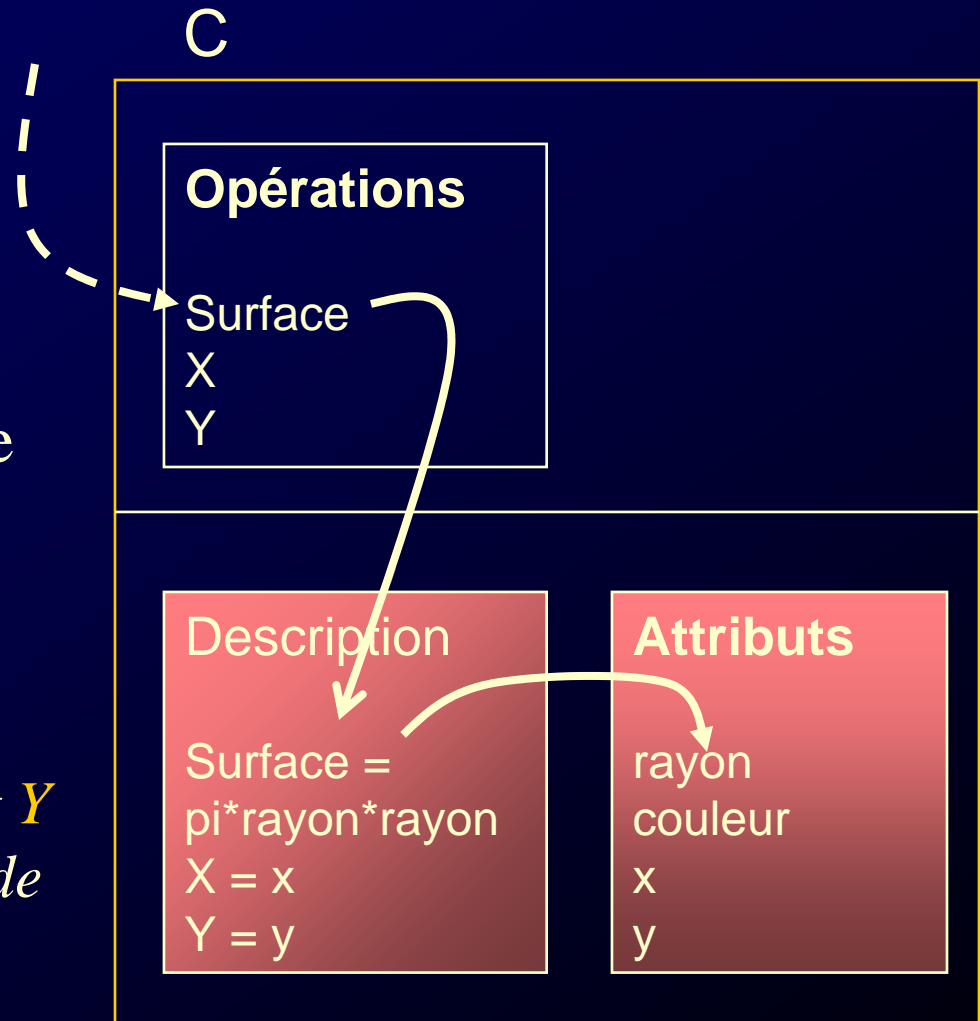
- ♦ Le type abstrait permet une définition précise sans décrire l'implémentation
 - Une classe est une implémentation particulière d'un type abstrait
 - Les opérations du type sont les méthodes définies dans la classe
 - Les valeurs du type sont tous les objets possibles de la classe
 - Il peut y avoir plusieurs implémentations du même type

Types Abstraits et Orienté-objet

« La conception par objets est la construction de systèmes logiciels prenant la forme de collections structurées d'implémentations de types de données abstraits »
(B. Meyer)

Objets

- Un objet fournit un *service* à d'autres objets
- Un objet comprend un ensemble de **messages** (interface)
- Une **méthode** sert chaque message
- Un **état caché** (composé de variables) qui persiste
- Application graphique :
 - Seules les méthodes **Surface**, **X** et **Y** peuvent manipuler les variables de l'état



Exemple : Prog.java

```
/* procédure qui démarre  
l'application */
```

```
main(){
```

```
Ecran e;
```

```
Carre c1, c2;
```

```
Cercle c;
```

```
/* Créer les nouveaux objets */
```

```
c1 = new Carre(1,2,2.0,e)
```

```
c2 = new Carre(4,2,5.0,e)
```

```
c = new Cercle(2,3,2.0,e);
```

```
/* Afficher les formes */
```

```
c1.dessiner(); c2.dessiner();
```

```
c.dessiner();
```

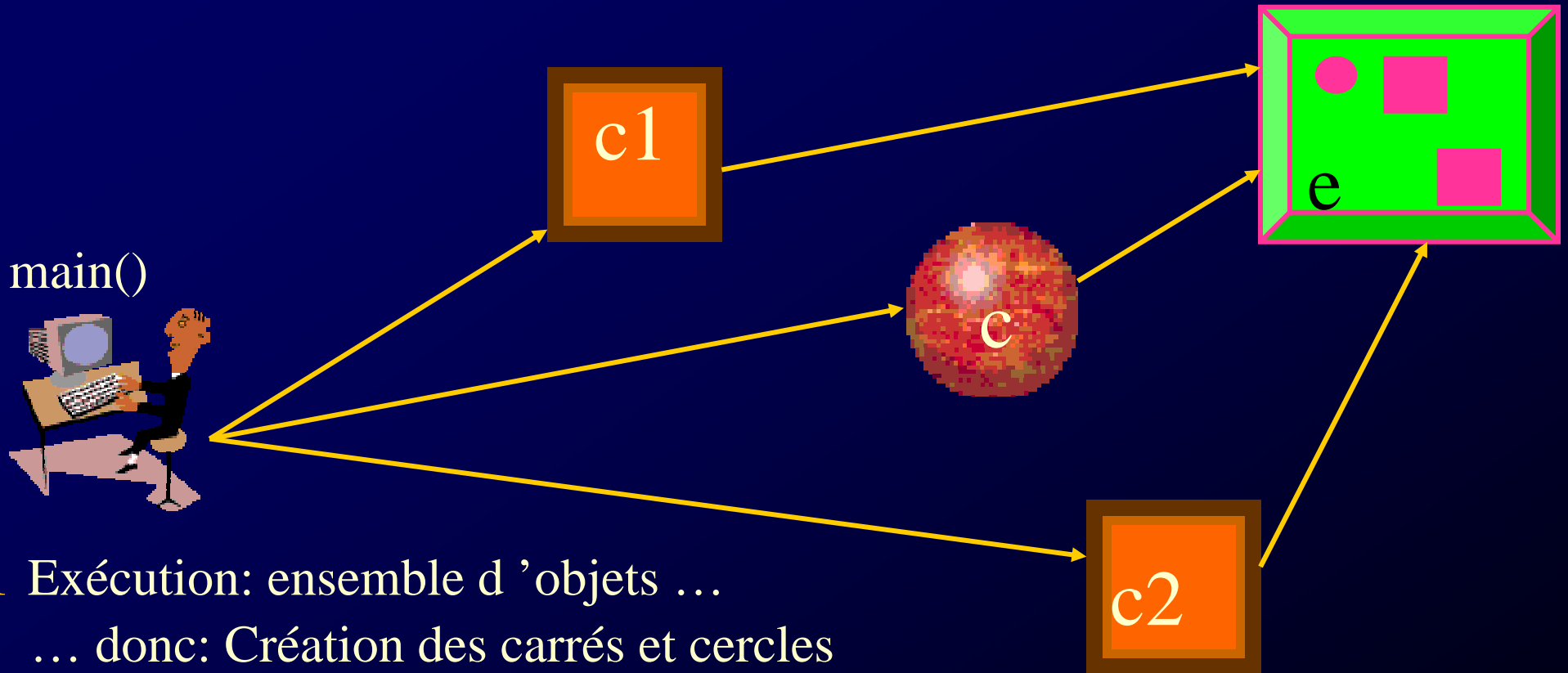
```
/* bouger carre 1 avec les  
paramètres de carre 2 */
```

```
c1.bouger(c2.X(),c2.Y());
```

```
}
```

```
/* Les paramètres du messages  
doivent être connus avant  
l'envoi du message */
```

Objets



1 Exécution: ensemble d'objets ...

... donc: Création des carrés et cercles
(avec transfert d'une référence sur l'écran)

2 Si un objet A possède une référence sur B ...

... alors A peut envoyer un message à B

Messages

- Un message, ou **appel de méthode**, est le moyen de communiquer entre objets
- Un message possède un **format** bien précis:

```
result = obj.meth(arg1, arg2, arg3)
```

- En Java, un argument (arg) et le résultat peuvent être:
 - Le nom d'un autre objet
 - C-à-d : une référence sur un objet
 - Mécanisme de base pour la distribution d'informations dans un système à objets
 - Ou une valeur simple (int, float, char, bool)

Classes

- ◆ Une classe définit un moule d'objet :
 - Les **données** participant à l'état
 - Les **messages** servis (*l'interface*)
 - La **méthode** exécutée pour chaque message
 - Un **constructeur** est une méthode spéciale
 - Elle est exécutée par tout objet lors de sa création
 - Elle initialise les variables d'état
 - Elle rend un nom unique pour l'objet au créateur
 - C-à-d : une référence sur l'objet
 - Créateur est soit le main() du programme, soit un autre objet

Exemple: classe Cercle

```
class Cercle{

    /* Données locales */
    float rayon, int x, y; Ecran ec;

    /* Le constructeur */
    public Cercle(int a, int b, float r,
        Ecran e)
        { rayon = r; x=a; y=b; ec = e;}

    /* Les méthodes des messages */
    public float Surface()
        { return 3.14*rayon * rayon;}
```

```
    public void bouger(int a,int b)
        { x = a; y = b;}

    public int X() { return x;}

    public int Y() { return y;}

    /* Un objet cercle envoie un
       message à l'écran */
    public void dessiner()
        { ec.afficher(this) ;}

}
```

Exemple: classe Carré

```
class Carre{
```

```
/* Données locales */
```

```
int x, y; float cote; Ecran ec;
```

```
/* Le constructeur */
```

```
public Carre(int a, int b, float h,  
            Ecran e)
```

```
{x=a; y=b; cote = h; ec = e;}
```

```
/* Les méthodes ont les mêmes noms  
   et paramètres! */
```

```
public float Surface()  
{ return cote*cote}
```

```
public void bouger(int a,int b)  
{ x = a; y = b;}
```

```
public int X() { return x;}
```

```
public int Y() { return y;}
```

```
public void dessiner()  
{ ec.afficher(this) ;}
```

```
}
```

Classes

- ◆ Une classe est fournie par un programmeur
 - Le code de la classe est recherché dans une librairie lors de la compilation
 - Sinon, la classe peut être recherchée pendant l'exécution
 - Quand un objet crée lui-même un nouvel objet d'une classe Cercle, le support d'exécution (le *run-time*) recherche la classe Cercle dans le système de fichiers ou sur le réseau au moment de l'exécution, lorsqu'un objet invoque:
`new Cercle(...)`

Prog.class à l'exécution

```
main(){  
  Ecran e; - allouer une référence pour e  
  Carre c1, c2; - allouer 2 références pour  
                des Carrés  
  Cercle c - allouer une référence pour un  
             Cercle  
  
  c1 = new Carre(1,2,2.0,e) - demande à  
                           l'environnement de trouver la classe  
                           Carre et de la charger en mémoire  
  c2 = new Carre(4,2,5.0,e)  
  c  = new Cercle(2,3,2.0,e);
```

*A l'aide des classes, les objets
sont créés*

```
c1.dessiner(); c2.dessiner();  
c.dessiner();
```

```
c1.bouger(c2.X(),c2.Y());
```

*- L'envoi des messages entre les
objets*

```
}
```

- ◆ La liaison du code est faite tardivement
 - On est donc sûr d'avoir la version la plus récente

Client - Serveurs - Contrats

- ◆ Un objet demande un service à un autre objet
 - Client: objet qui demande le service
 - Serveur: objet qui satisfait le service
 - Contrat: liste de requêtes qu'un client peut demander à un serveur
 - Satisfaction du contrat:
 - le client n'établit que des requêtes du contrat
 - le serveur répond de manière appropriée aux requêtes
 - Eiffel: attacher des pre- post-conditions aux méthodes
 - *Ex: on ne demande pas une pizza au McDonald*

Interface

- ◆ L'interface sert de contrat entre un objet et ses clients
- ◆ Une interface remplit le rôle d'un type
 - En énumérant les opérations du type
- ◆ Une classe réalise (ou implémente) un type
 - Plusieurs classes peuvent réaliser le même type
 - Cela permet de séparer la spécification d'un composant de sa réalisation (génie logiciel)
- ◆ Pour une interface I
 - Les opérations de I sont celles définies par l'interface
 - Les valeurs de I sont celles de tous les objets dont leurs classes implémentent I

Interface

- ◆ En Java, on peut définir une interface

```
public interface Forme{  
    public float Surface();  
    public void bouger(int a, int b);  
    public void dessiner();  
    public int X();  
    public int Y();  
}
```

*Une interface en Java est
comme une classe mais sans
le code des méthodes*

Une Interface définit un type

Interface

- ◆ Une classe peut réaliser une interface
 - La classe est censée fournir une méthode pour toutes celles de l'interface
 - Les entêtes de Cercle et Carré peuvent être écrites ainsi :
 - « class Cercle implements Forme » ou « class Carre implements Forme »
- ◆ Le client est alors assuré qu'un objet d'une classe qui réalise une interface comprend tous les messages de cette interface

Exemple: classe Carré

```
class Carré implements Forme{
```

```
    int x, y; float cote;
```

```
    Ecran ec;
```

```
    public Carré(int a, int b, float h,  
                Ecran e)
```

```
    { x=a; y=b; cote=h;ec = e;}
```

```
    public float Surface()
```

```
    { return cote*cote}
```

```
    public void bouger(int a,int b)
```

```
    { x = a; y = b;}
```

```
    public int X() { return x;}
```

```
    public int Y() { return y;}
```

```
    public void dessiner()
```

```
    { ec.afficher(this); }
```

```
}
```

Interface

```
main(){
Cercle c1 = new Cercle(2,3,4.0,e);
Carre c2 = new Carre (3,6,7.0,e);

Forme f;

/* 2 affectations légales */
f = c1; f=c2;

/* Exploitation de l'interface */
f.dessiner();
float f1 = f.surface()
```

- ◆ Polymorphisme: utile pour le code qui manipule des objets de diverses sortes (p.ex: dans la classe Ecran)
`void afficher(Forme f)`
- ◆ Extensibilité: on peut facilement ajouter des nouvelles classes pendant la vie de l'application
 - `implements Forme`

Interface vs Implémentation

- ◆ La programmation objet se concentre sur les messages et non sur les données
- ◆ P.ex: Comment faut-il représenter la date ?
 - 17.3.1999 ou 3/17/1999 ou 99.3/17 ?
- ◆ Dans le modèle d'objets
 - Date est une classe avec l'interface: jour(), mois(), année()
 - Pour un objet de la classe date, le mois est donné par d.mois()
 - La représentation interne de la date n'est pas importante

Interface vs Réutilisabilité

- ◆ Une classe peut être réutilisée dans une autre application
 - Un but de la programmation objet !
- ◆ Parfois la classe ne convient pas à 100%
 - P.ex: pour représenter un cercle en couleur, il faut modifier Cercle pour représenter la couleur
 - P.ex: la classe Cercle ressemble beaucoup à la classe Carré, mais on ne peut pas utiliser l'une pour l'autre

Interface vs Réutilisabilité

- ◆ Le principe ouvert - fermé
 - Une classe doit être fermée par rapport à d'autres classes
 - Le principe de la modularité et Information hiding
 - Accès simplement via l'interface ; l'état interne reste caché
 - Une classe doit être suffisamment ouverte afin de pouvoir l'étendre facilement
 - P.ex: définir une classe Cercle à partir d'une classe Carré
 - Le mécanisme d'héritage

Conception - Principes

- ◆ Déterminer les objets
 - Quels sont les buts du système?
 - Quel doit être son comportement vis-à-vis de l'extérieur
 - De quels objets a-t-on besoin pour réaliser les buts?
- ◆ Déterminer les responsabilités
 - Contrats entre les objets
 - Quelles sont les actions à accomplir?
 - Quel objet accomplit quelle action?
- ◆ Déterminer les collaborations
 - Lien entre objets: clients, serveurs (rôles) et contrats

Conception

- ◆ La programmation nécessite une modélisation du système que l'on code
 - Mais comment choisir les classes pour une application ?
 - Expérience !
 - Souvent beaucoup d'essais sont nécessaires
 - Parfois, on est restreint par les classes dont on dispose déjà dans nos librairies
 - Il est souvent préférable de réutiliser une classe que d'en coder une nouvelle