

*Mise en Oeuvre*

---

*Run-time*

# *Pourquoi « mise en œuvre » ?*

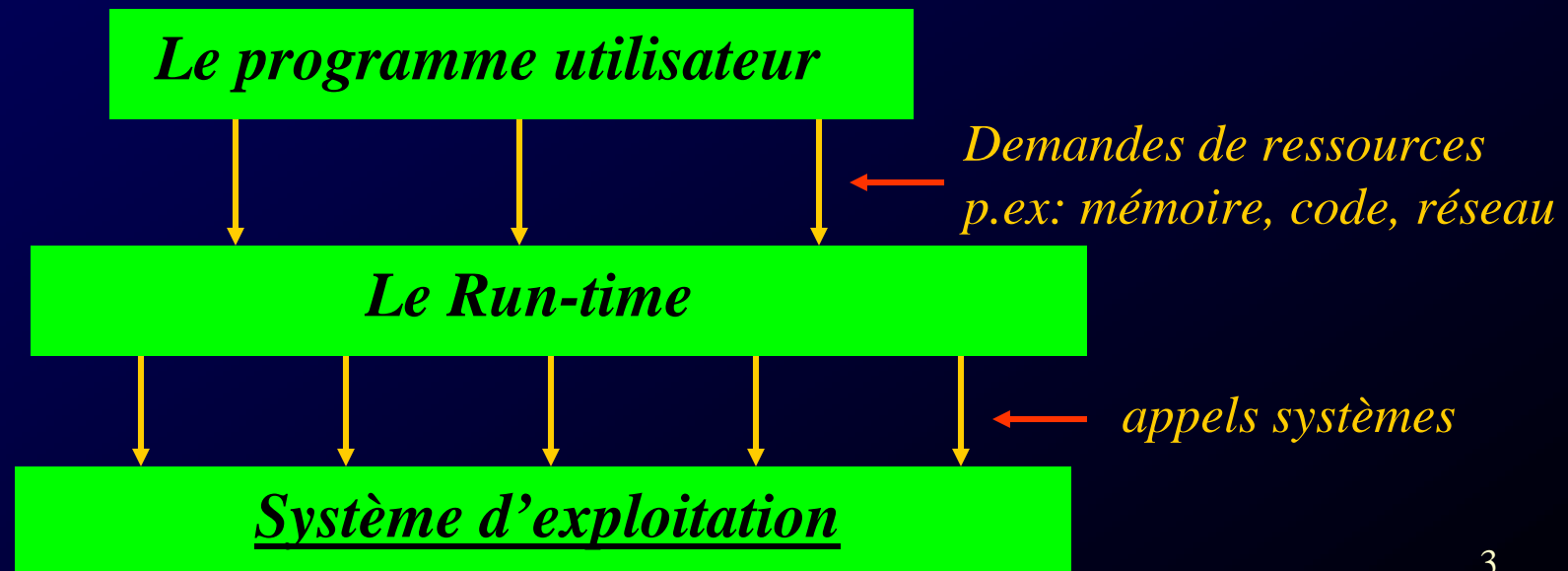
---

- ◆ A quoi ressemble un programme Java pendant son exécution ?
  - Comment exploite-t-il la mémoire de la machine ?
    - Le CPU ?
    - Ou encore le réseau ?
- ◆ Par exemple
  - Java n'offre pas un moyen de détruire un objet
    - Façon d'éviter des erreurs de programmation
    - Mais la mémoire d'un système n'est pas sans borne !
      - Il faut que « quelqu'un » enlève les objets qui ne sont plus utilisés

# *Le run-time*

---

- ♦ Le run-time est l'environnement d'exécution d'un programme
  - Une couche de logiciel (ensemble de procédure) qui sert d'interface entre le programme et le système d'exploitation (OS)



# *Le run-time*

---

- ◆ Le run-time d'un programme C (sur Unix) comprend :
  - L'environnement d'un processus Unix
    - Les variables STDOUT, PATH, PID, UID, .....
  - L'environnement minimal
    - Un programme qui désire accéder à une ressource (p.ex: un fichier) la demande directement à l'OS à travers un appel système
  - Dans le monde de la programmation objet (et de l'Internet)
    - Un run-time est plus complexe !
    - Le run-time est un programme qui est chargé avant le chargement du programme utilisateur
    - Le run-time sert d'interface entre le programme et l'OS
    - Il met en œuvre les concepts O-O!
    - Java: il est formé de l'interpréteur et de la machine virtuelle Java

# *Les buts du run-time*

---

## Gestion de mémoire

- Un programme Java crée un objet avec l'opérateur *new()*
- Allocation d'un espace mémoire:  
le *new()* est une fonction du run-time, qui utilise la fonction *malloc()*
- Mise à jour de pointeurs: il faut pouvoir accéder à l'espace mémoire où se trouve l'objet
- Un processus **ramasse-miettes** tourne occasionnellement pour libérer des cellules mémoires occupées par des objets qui ne sont plus utilisés

# *Les buts du run-time*

---

## ◆ Gestion du CPU

- Un programme (Java) peut contenir plusieurs threads (processus légers)
- Le run-time doit assurer l'exécution et l'ordonnancement des threads

## ◆ Gestion du réseau

- Un objet peut envoyer un message à un objet qui se trouve sur une autre machine
  - Le run-time écrit le message sur un socket
  - Le run-time sur la machine à distance lit la requête et exécute l'appel
    - L'utilisation des sockets est cachée aux programmeurs

# *Les buts du run-time*

---

- ◆ La liaison dynamique du code
  - Dans un programme objet, le code d'une classe est recherché la première fois qu'un objet de la classe est créé
    - Le run-time est chargé de trouver le code de la classe
      - Sur disque ou bien sur un site à distance
- ◆ Hétérogénéité des OSs et des machines
  - Un programme doit tourner sur n'importe quelle plate-forme
    - Le run-time doit fournir une interface standard qui cache l'OS du code du programme
    - Et qui transforme le code du programme en code machine compréhensible pour le processeur de la plate-forme

# *Les buts du run-time*

---

- ◆ Le respect de la sémantique du langage
  - Un programme objet comprend des notions telles que l'encapsulation des objets
    - Un programme ne peut pas accéder directement aux variables d'un objet
      - Il ne doit passer que par les méthodes !
  - Mais le code machine ne comprend que des opérations de lecture et écriture des cellules
    - Un programme écrit en code machine peut donc violer la sémantique du langage !



# *La représentation de programmes*

---

*Le modèle simple*

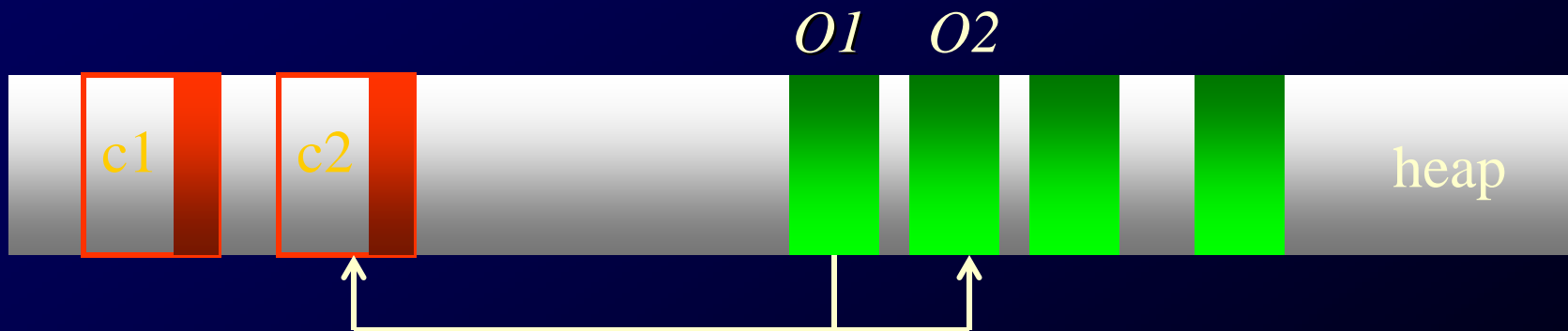
# *La représentation de programmes*

---

- ◆ Notion d'objet et de classe à l'exécution
  - La représentation d'un programme en mémoire primaire
  - Désignation, partage et protection des objets
- ◆ Objets de durée de vie non-déterminée
  - Archivage et ramasse-miettes
- ◆ Héritage et sous-typage
  - Sélection de méthode

# Représentation à l'exécution

- ◆ Modèle (simple) de mémoire partagée
  - Application emploie 1'espace d'adressage
  - P.ex. processus Unix



Objet *O1* possède une référence pour *O2*

Variables statiques

*Pointeur d'objet* = (*addr\_classe*, *addr\_objet*)

**Rappel : une référence n'est pas un simple pointeur**

# Désignation

---

## ◆ Désignation et partage des objets

- Un objet est nommé par une paire d'adresses
  - Classe et Instance
- Utilisation des adresses facilite le partage des objets
  - Car l'interprétation des références est la même pour chaque objet
- L'accès à l'objet depuis sa référence est rapide

## ◆ Messages

- L'envoi d'un message revient à un appel de procédure
  - Encore efficacité
  - (mais ceci deviendra plus complexe à cause du sous-typage)

# Protection

---

- ◆ Aucune protection n'existe au sein d'un espace d'adressage

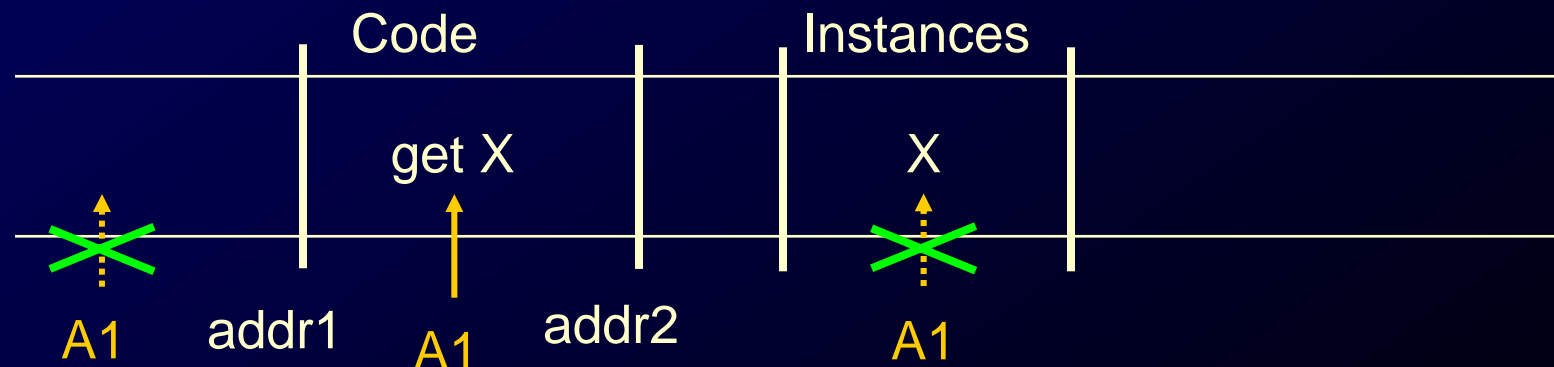
`int *I; while(1) { *I=0; I++; } /* Détruit les données */`

- Ce programme viole l'encapsulation des objets dans le même espace
- ◆ Préservation de l'encapsulation (1): typage
  - Détection d'erreurs via le typage (Eiffel, Java)
    - Si le programme source (le .java) ne contient pas d'erreurs, alors
      - transformation directe en code machine ne contient pas d'erreurs non plus
    - Mais quelques objets des bibliothèques peuvent être écrits en code machine (donc on ne peut jamais être sûr de la sécurité)

# Protection

## ◆ Préservation de l'encapsulation (2)

- Insertion de code ou Sand-boxing
  - Pour éviter l'accès à des zones interdites
  - A tout objet est alloué un espace fixe [addr1..addr2]
  - Le compilateur insère du code qui vérifie dynamiquement que les adresses utilisées sont dans [addr1..addr2]
    - **MOVE A1, A2** dans la source est remplacé par
      - **CMPV addr1,A1; BLT erreur; CMPV addr2,A1; BGT erreur; MOVE A1,A2;**
    - **erreur** est l'adresse de la procédure EncapsulationException



# *Protection*

---

- ◆ Préservation de l'encapsulation (3)
  - Sand-boxing utile pour code faiblement typé ou pour des librairies qui contiennent du code natif (code machine ou code C)
    - Beaucoup de librairie Java contiennent du code natif,
      - P.ex: System, swing, gestion de réseau

# Persistence

---

- ◆ La durée de vie d'un objet peut dépasser celle de l'application qui l'a créé
  - Run-time utilise le système de fichiers du système d'exploitation
- ◆ Mais Objet  $\neq$  Fichier
  - Il faut stocker la classe et l'objet
  - Il faut un nom d'objet en plus de l'adresse !
    - Une adresse a seulement de la signification dans un espace d'adressage
    - P.ex: un nom de fichier ou un URL



# La destruction d'objets

---

- ◆ L'opérateur *new()* exécuté sur une classe crée un objet de cette classe
  - Il n'y a pas d'opérateur pour détruire un objet !
    - Et pour optimiser l'utilisation de la mémoire primaire de la machine, il faut enlever les objets qui ne sont pas utilisés
- ◆ En C, l'opérateur *malloc()* alloue les espaces mémoires et *free()* les détruit (les libère)
  - Mais leur utilisation s'avère trop susceptible aux erreurs de programmation
    - Dans la programmation objet, un objet doit rester dans l'application jusqu'au moment où le run-time le supprime

# *La destruction d'objets*

---

- ◆ Un objet consomme du CPU & de la mémoire
  - Et doit être détruit lorsqu'il n'est plus utilisé par l'application
- ◆ Un objet est une **miette** s'il n'est plus utilisable ...
  - ... Si aucun autre objet ne possède une référence sur cet objet, alors: on ne peut pas envoyer un message à cet objet et donc, cet objet ne peut pas contribuer à l'application
- ◆ Un processus de ramasse-miettes (*garbage collection*) tourne périodiquement pour enlever les miettes de la mémoire

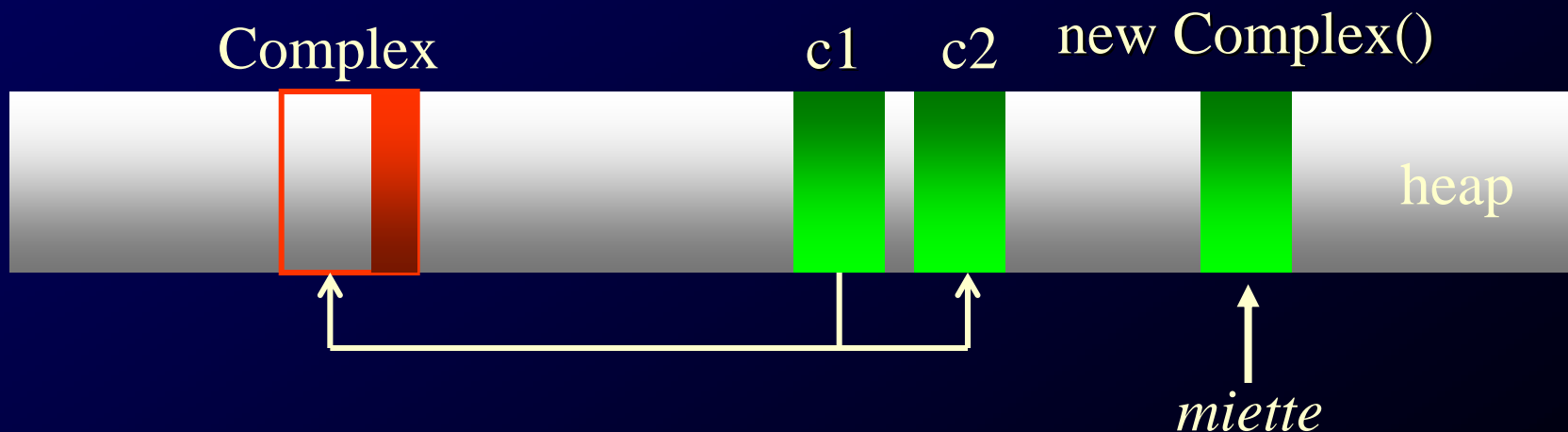
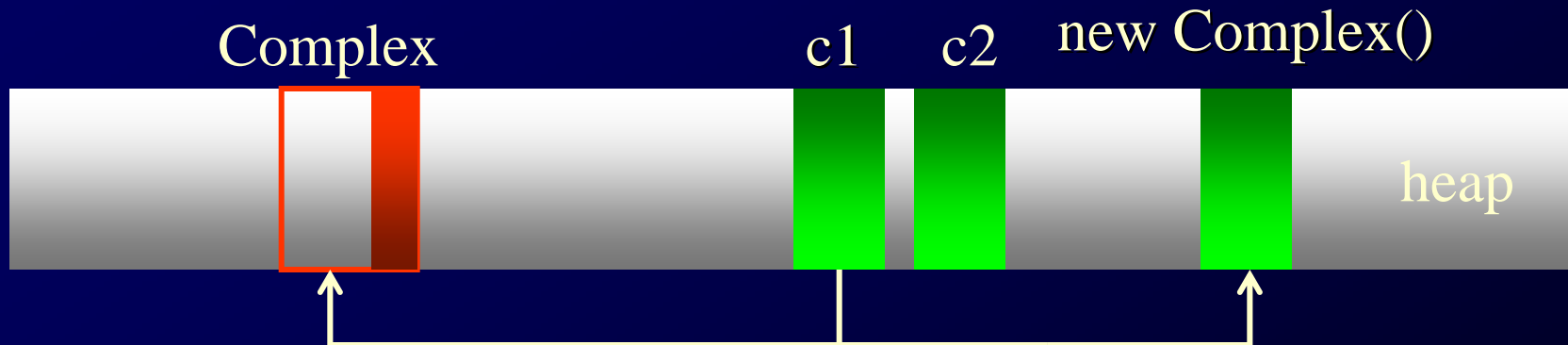
# *Le ramasse-miettes*

---

- ◆ Un objet est non-utilisé (miette) s'il n'est plus accessible depuis un programme
  - p..ex. aucune référence n'existe pour l'objet suite à des affectations
    - `Complex c1, c2; .... c1 = new Complex(); c1=c2;`
- ◆ 2 approches
  - 1. Compteur de références
  - 2. Traçage

# *Le ramasse-miettes*

Complex c1, c2; .... c1 = new Complex(); c1=c2;



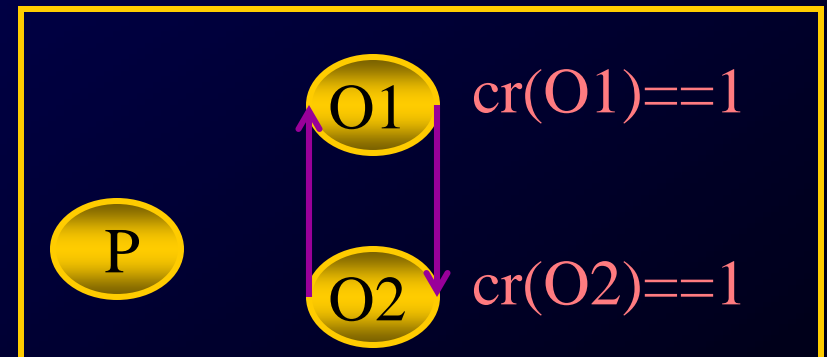
# Compteurs de références

- ◆ Tout objet possède un compteur  $cr$

- Création d'objet :  $cr = 1$ ;
- Affectation d'une référence :  $cr++$
- Destruction d'une référence :  $cr--$
- Si  $cr == 0$  alors miette

- ◆ Inefficace

- Plusieurs instructions
- Cycles: les miettes restent
  - O1 et O2 sont miettes



# *Traçage*

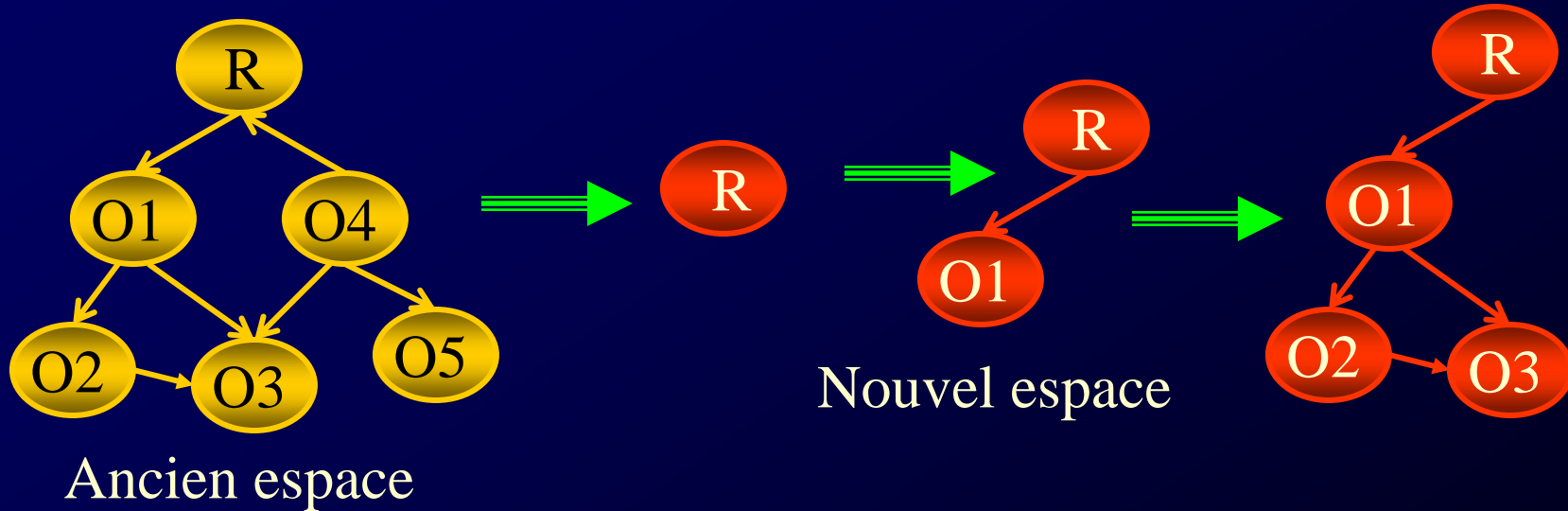
---

## ◆ Solution de Java et Eiffel

- Racine: objet qui existe toujours, p. ex. E/S
  - Miette: objet qui n'est pas accessible de la racine
- Ramasse-miette : processus système qui
  - Crée un nouvel espace mémoire d'objets
  - Copie la racine dans le nouvel espace
  - Copie tout objet référencé par la racine
  - Copie tout objet référencé par ces objets
  - L'ancien espace ne contient que des miettes
  - Ramasse-miettes (garbage collector) exécuté périodiquement ou lorsque le système est inactif

# Exemple

- Objets O4 et O5 sont miettes



# Héritage

---

- ◆ Le partage de méthodes entre les classes suggère le partage du code à l'exécution
  - P.ex: les classes C1 et C2 partagent le code de la méthode m1

```
Class C1 is
  int x;
  method m1 is
    x := 0;
end.
```

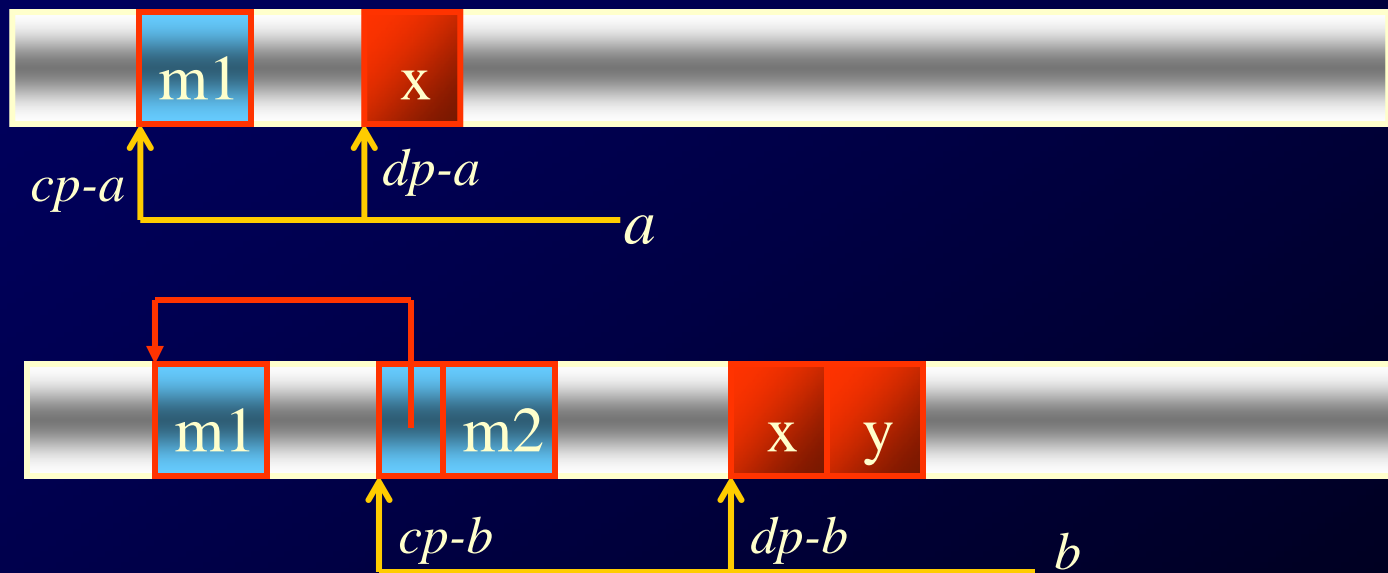
```
Class C2 is
  subclass of C1
  /* variables comprennent x et y */
  int y;
  /* méthodes comprennent m1 et m2 */
  method m2 is
    y := 1;
end.
```



# Partage de code

- soit  $a:C1$ ,  $b:C2$

$$a = \{cp-a, dp-a\} \quad b = \{cp-b, dp-b\}$$



$a.m1() \Rightarrow \text{MOVEV } cp\_a, A1; \text{ADDV } m1\_offset, A1; \text{JMP } A1;$

$b.m2() \Rightarrow \text{MOVEV } cp\_b, A1; \text{ADDV } m2\_offset, A1; \text{JMP } A1;$

# *Liaison Dynamique*

---

```
p1,p2:Point;p3:ColPoint;  
.....  
if (f()) p1=p2; else p1=p3;  
bool b = p1.null();
```

## Rappel :

- ◆ Choix de méthode  
null() dépend du type  
dynamique de la  
variable p1

### Class Point


```
int x,y;  
equal(Point p){  
    return(p.x ==x && p.y ==y);}  
null() return(x==y==0);
```

### Class ColPoint is Point

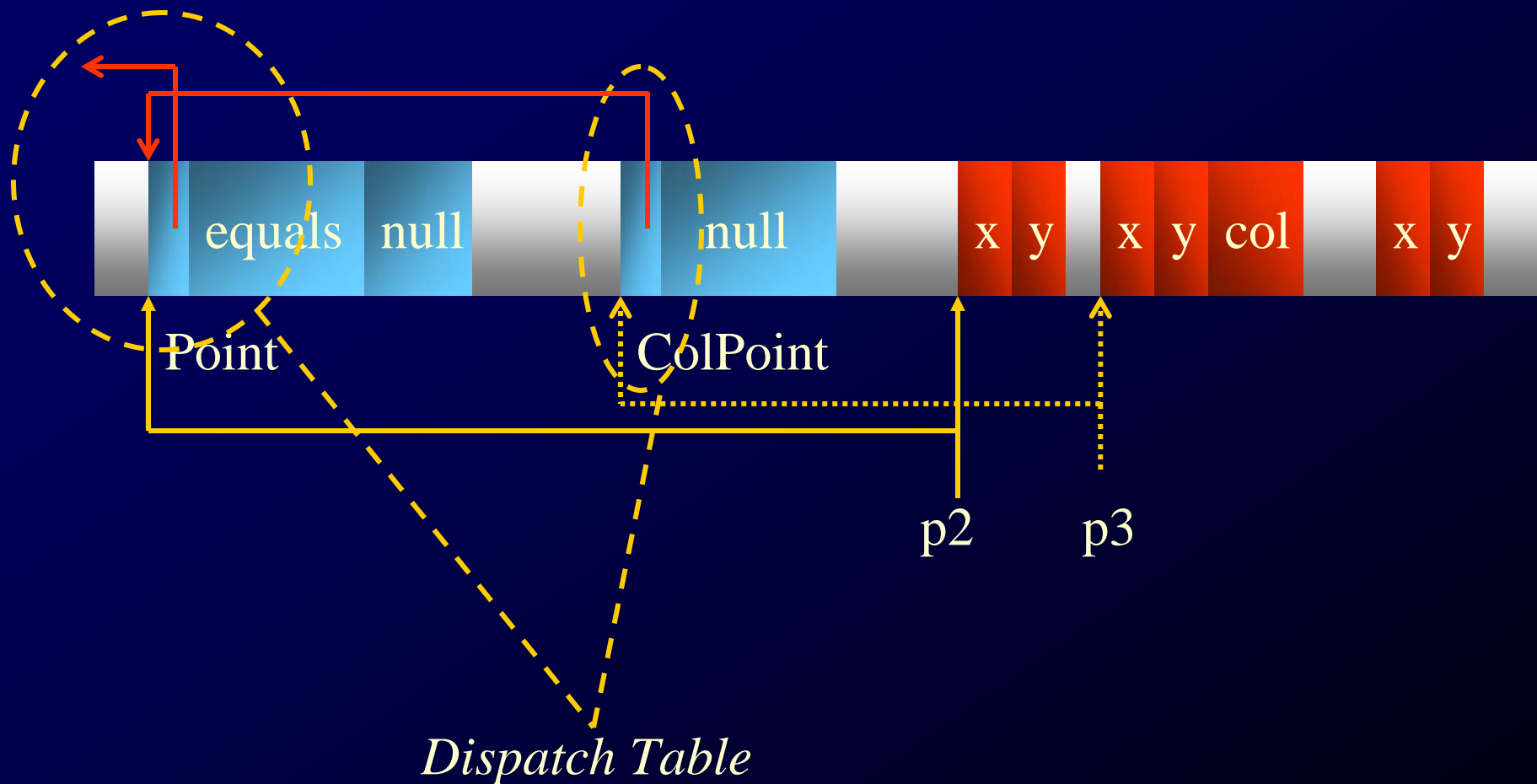
```
int col;  
null(){  
    return (x ==0 && y ==0 &&  
            col==0);}
```

# Sous-typage

---

- ◆ La classe d'un objet doit être discernable de la référence
  - pointeur = (adresse de données et de la classe) 
- ◆ Chaque classe contient une dispatch table
  - But: lier un message à une méthode
  - Table contient liste des méthodes quelle réalise
    - Et un pointeur vers sa super-classe
  - Une recherche *dynamique* est nécessaire

# *Plan de mémoire*



# *La représentation de programmes*

---

*Le cas de l'Internet*

# *Version Internet*

---

- ◆ La projection d'une application dans un espace possède des limitations pour ..
  - Applications de grande taille
  - Objets persistants
  - Haute sécurité
    - Code de sources différentes ne doivent pas être mis dans le même espace d'adressage
  - Applications distribuées
    - Comme sur l'Internet

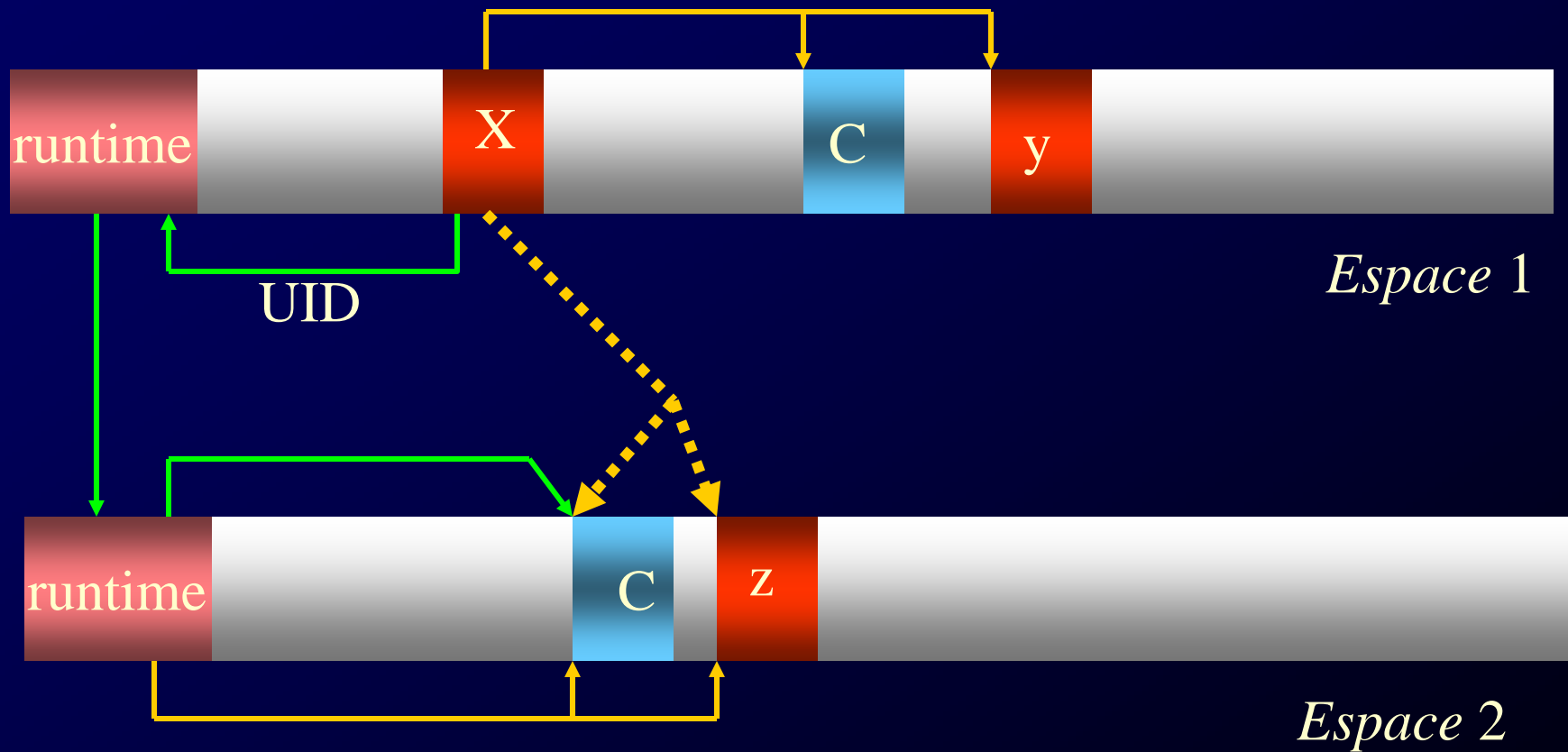
# Version Internet

---

- ◆ L'approche générale
  - Utiliser des UUIDs (Unique Identifier) comme références d'objets
    - Identificateur unique par rapport à la machine sur laquelle il est généré
    - unicité est obtenue: UUID + Adresse Internet de la machine
    - similaire aux URLs
- ◆ Pour utiliser un objet local
  - Charger l'objet dans son espace d'adressage
    - Puis exploiter le modèle de mémoire partagée !
  - Appel de méthode sur un UUID implique l'envoi d'un message entre sites
    - RPC (*Remote Procedure Call*)
    - Mécanisme employé par CORBA, DCOM, etc...

# Version Internet

---





# *Les classes exécutables*

---

*Comment exécuter le code qu'on écrit ?*

## ◆ **Compilation**

- Transforme les classes en code machine
- Le run-time fait tourner ce code
- P.ex: C++, Eiffel

## ◆ **Interprétation**

- L'interpréteur lit chaque commande du programme source, signale des erreurs ou la donne au run-time
- P.ex: Tcl, SmallTalk

# *Avantages de la compilation*

---

## ◆ Efficacité temporelle

- Pas de parsing, de liaison dynamique d'une instruction au code de run-time qui la réalise, ..
- Code exécuté directement par la machine

## ◆ Efficacité spatiale

- Sans interpréter, le système est plus compact
- Format des données basé sur ceux de la machine

# *Avantages de l'interprétation*

---

## ◆ 1. Portabilité

- Classes tournent sur plusieurs plates-formes,
- Et migrent entre machines à l'exécution.
- Qualité dans un système réparti, e.g., Java

## ◆ 2. Environnements dynamiques

- Changement de code ne nécessite pas un redémarrage de l'application
  - Utile pour construire des prototypes

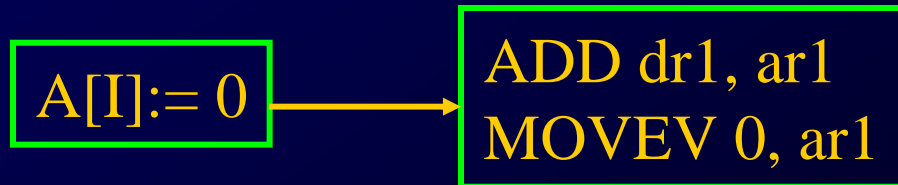
# Avantages de l'interprétation

---

## ♦ 3. Sécurité

- mise en œuvre du langage *est plus proche de sa sémantique*

- p. ex. scope, typage



- ici, l'idée que I doit être borné est perdue à la compilation
- Détecte des erreurs plus facilement (important pour code d'une source différente !)

# *Approche « hybride »*

---

- ◆ Plus d'efficacité avec interprétation
  - **Compiler** la classe en un assembleur de haut niveau (byte-code) pour une machine virtuelle
    - P.ex. machine virtuelle de Java
    - **fload, fstore, aload, invokevirtual, dup,**
  - **Interpréter** le bytecode
- ◆ Plus de portabilité avec la compilation
  - Run-time généralement écrit en C
  - Compilateur génère du code C, p.ex. Eiffel

# *La représentation de programmes*

---

*... en Java*

# *Run-Time Java*

---

## ◆ Machine Virtuelle

- machine abstraite
- jeu d'instructions
- manipule des espaces mémoire

## ◆ Interpréteur

- traduit et exécute le bytecode (code intermédiaire pour la machine virtuelle) ligne après ligne

*Le programme utilisateur*



*Le Run-time: Java VM*



*Système d'exploitation*

Windows ou Unix ou Linux

# *Espaces de données*

---

- ◆ Chaque thread a son espace de données qui contient:
  - son propre registre **compteur ordinal (co)**
    - contient l'adresse mémoire de la prochaine instruction à exécuter (de la méthode en cours d'exécution)
  - sa propre pile d'exécution
    - stocke des « frames », contient les variables locales, les résultats partiels et sert à l'invocation des méthodes
    - un frame par méthode invoquée (variables locales, pile d'opérandes, référence symbolique de la méthode)



# *Espaces de données*

---

## ◆ Heap:

- espace de données partagé parmi tous les threads
- espace de données à partir duquel toutes les instances de classes sont allouées
- soumis au ramasse-miettes

## ◆ Espace de méthodes

- espace partagé parmi tous les threads
- stocke les structures de classes (champs, méthodes, code, constructeurs)
- fait partie du heap, pas forcément soumis au ramasse-miettes

# *Espaces de données*

---

- ◆ Run-Time Constant Pool
  - table des symboles (une par type: classe ou interface)
  - contient des références symboliques qui deviennent concrètes au fur et à mesure de l'exécution

# *A l'exécution*

---

## ◆ Liaison dynamique

- Transforme les références symboliques sur les méthodes en références concrètes.
- Chargement de classes si nécessaire
- Transforme l'accès aux variables en l'offset nécessaire pour y accéder

## ◆ Exceptions

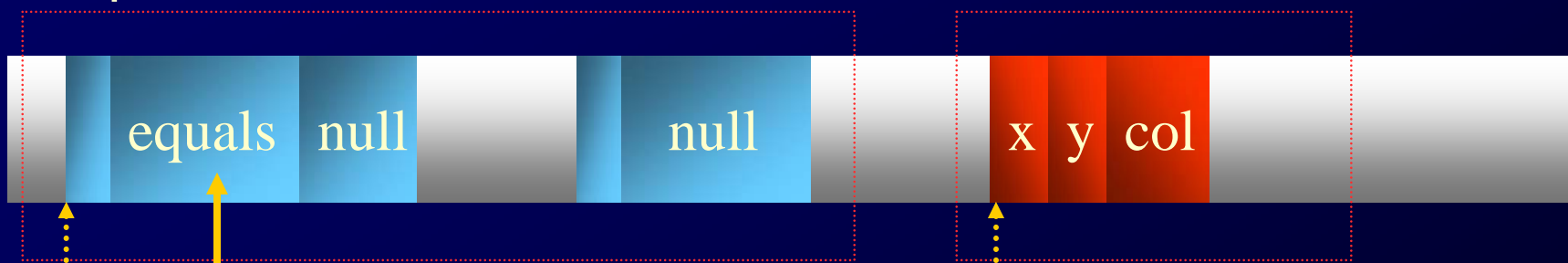
- Transfert immédiat du flot de contrôle du point où l'exception est levée au point où une clause catch gère l'exception

# Espaces de données

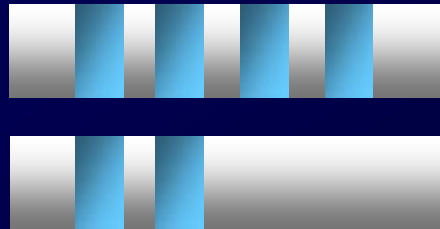
Espace de méthodes

Instances

Heap



Run-Time Constant Pools



Type1

Type2

co

Thread1

frame

frame

Pile

Thread2

Pile

# Démarrage

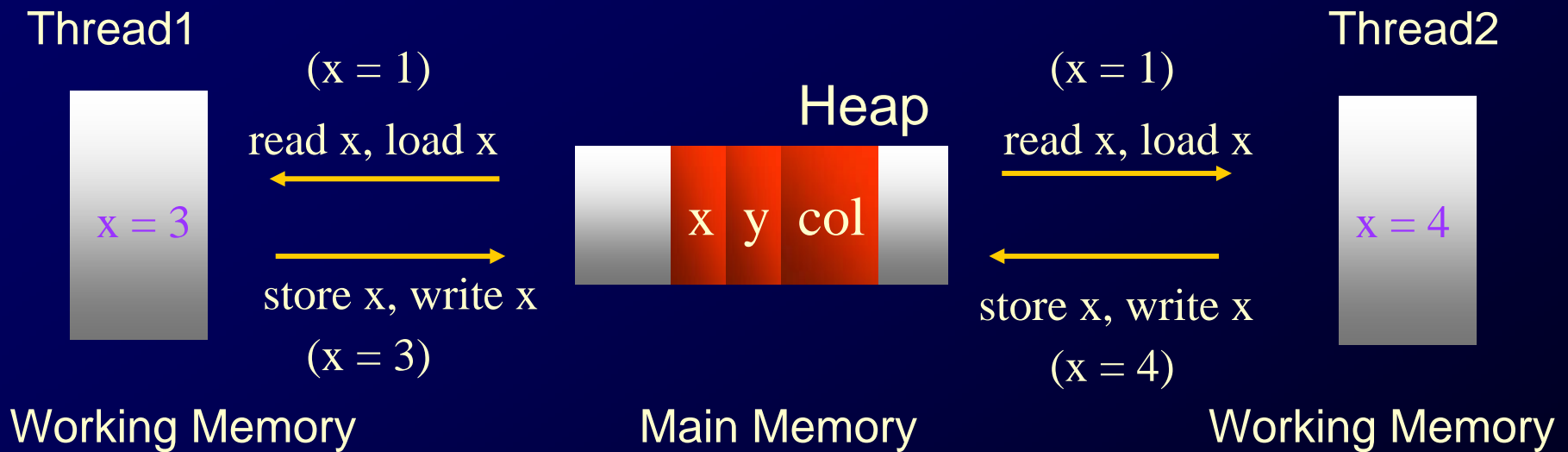
---

## ♦ Comment ça marche ....

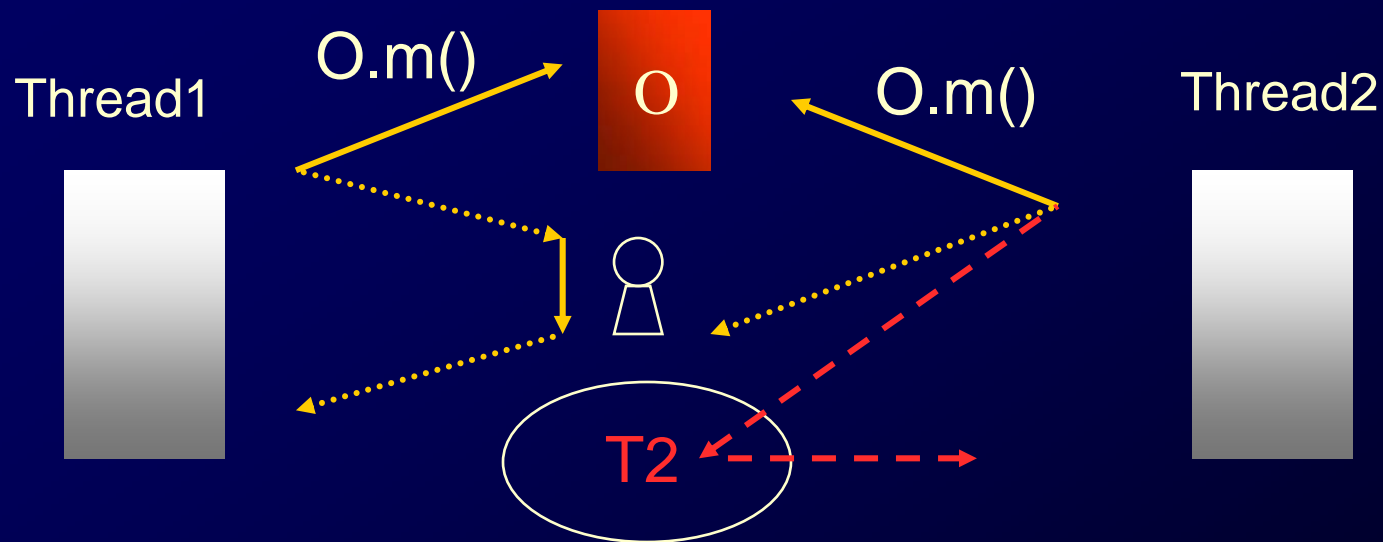
- **chargement** de la classe initiale (.class qui contient le main())
- **liaison**:
  - vérification (bien formé, table des symboles, sémantique est respectée)
  - préparation (allocation mémoire)
  - résolution (déterminer des valeurs concrètes à partir des références symboliques, charger éventuellement de nouvelles classes mentionnées dans les références)
- **initialisation**:
  - invoquer les initialisations de variables et de méthodes statiques
  - initialiser la superclasse directe (etc, récursivement)
- **invoquer le main()**

# Threads

---



# Threads et Verrous



- 1 verrou par object
  - thread prend le verrou lorsqu'il entre dans un méthode synchronized
  - thread relâche le verrou lorsqu'il quitte la méthode
- 1 wait set par objet (ensemble de référence de thread en attente sur l'objet)