CAFEBABE

Gestion des objets et des appels de méthodes

Laffineur Gérôme

21 Novembre 2016

Université de Namur

Dans les épisodes précédents..

Features déjà prises en compte

Pour tous les types primitifs (int,long,double,short,..) :

Chargement de constantes

Opérations arithmétiques

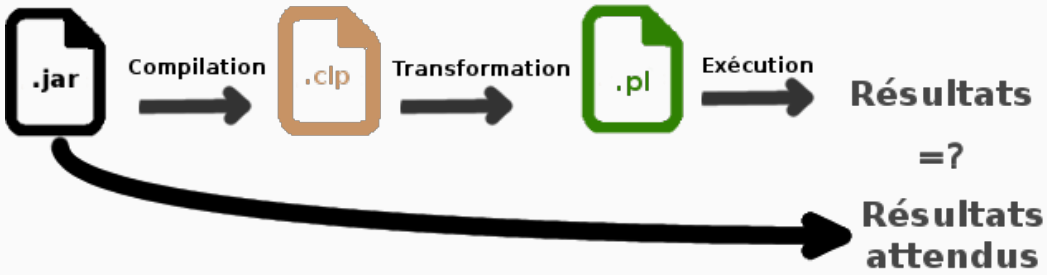
Structures conditionnelles

Gestion des boucles

Instructions liées à la gestion des variables

Nouvelle version du module de test

La nouvelle version du module de test



PRINCIPE : Exécution des clauses produites et comparaison des valeurs obtenues avec les valeurs attendues.

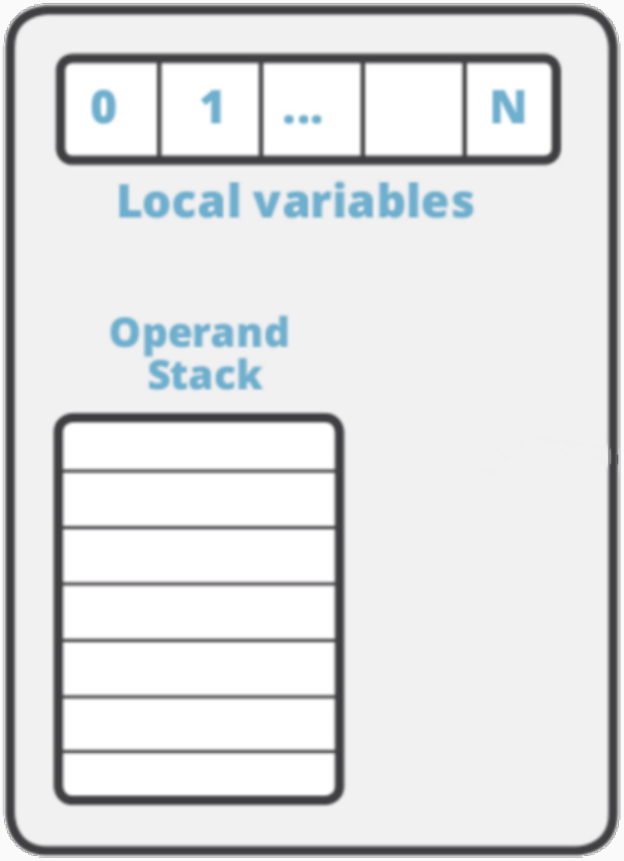
- Interfaçage Java-Prolog avec JPL
- Module de transformation en Prolog fourni par Gonzague :-)
- Plus robuste par rapport aux changements de représentations
- Actuellement plus d'une quarantaine de tests unitaires (Voir Démo)
- Permet d'avoir l'état final des les variables **R**
- Exécution = Requête sur point d'entrée (**pStart**)

La représentation de la mémoire

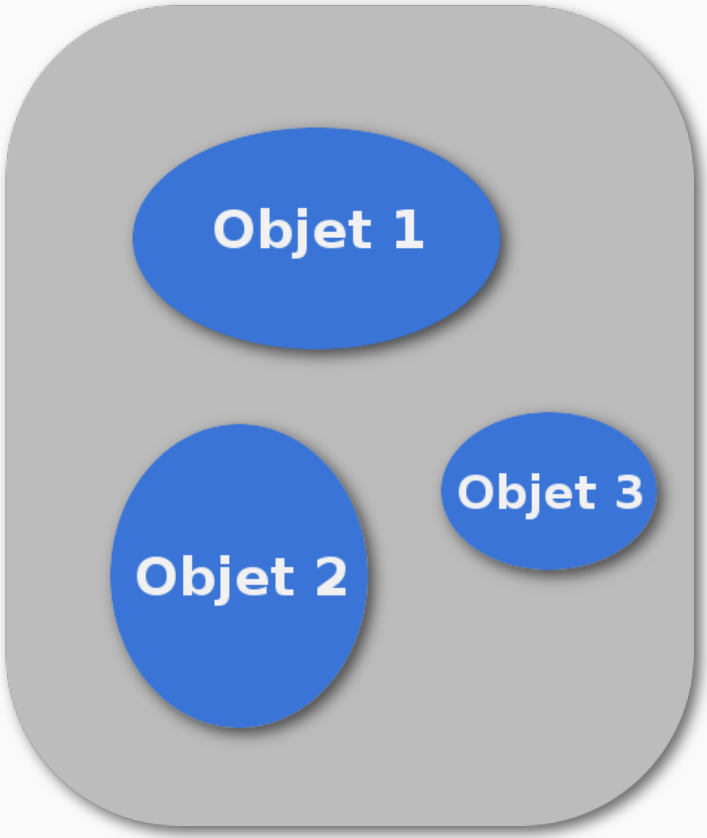
Les composantes mémoires de la JVM

2 types de composants pour stocker les données :

Le stack frame



Le heap



Les composantes mémoires de la JVM

2 types de composants pour stocker les données :

Le stack frame (Registres **IV**&**NV**)

- Un stack frame par méthode
- Environnement de travail
- Travail avec les types primitifs
- $\forall$ Programme P : Possible de calculer sa taille de manière statique
- On peut donc le dérouler

Le heap (**assert**-**retract**)

- Contient les objets
- $\exists$ Programme P : Impossible de calculer sa taille de manière statique
- Il faut donc un mécanisme d'allocation mémoire **dynamique** pour le représenter (assert/retract, tableaux dynamiques, ..)

La représentation du heap

Approche de développement :

D'abord essayer d'avoir la représentation la plus générale possible, pour élargir un maximum l'ensemble de programmes d'input.
Puis développer des cas d'optimisation pour les sous ensembles de programmes plus simples qui n'ont pas besoin de représentations dynamiques. Permet de faciliter le travail des analyseurs

La représentation du heap

Write Memory (IN,IN,IN)

```
writeMemory(OBJECTREF,FIELD,VALUE) :-
    { },
    retract(object(OBJECTREF,FIELD,X)),
    assert(object(OBJECTREF,FIELD,VALUE)).
writeMemory(OBJECTREF,FIELD,VALUE) :-
    { },
    assert(object(OBJECTREF,FIELD,VALUE)).
```

Read Memory (IN,IN,OUT)

```
readMemory(OBJECTREF,FIELD,VALUE) :-
    { },
    object(OBJECTREF,FIELD,VALUE).
```

La représentation du heap

Write Memory (IN,IN,IN)

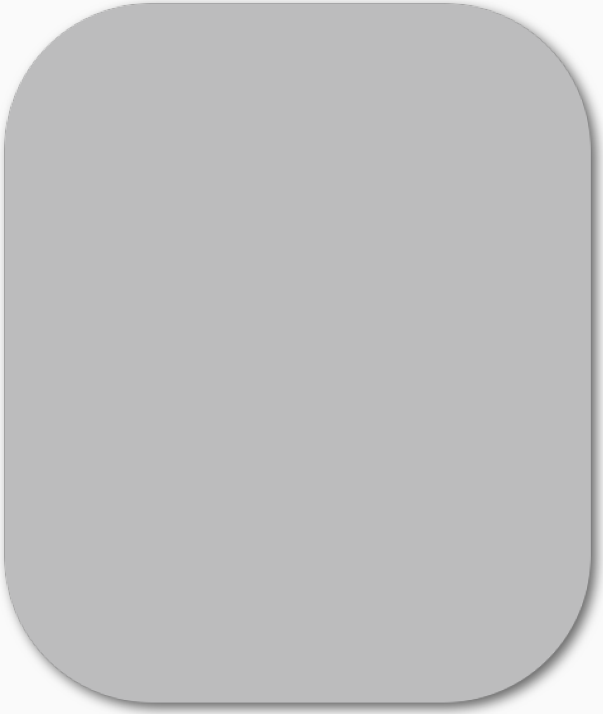
writeMemory(OBJECTREF,FIELD,VALUE) :-
{ },
retract(object(OBJECTREF,FIELD,X)),
assert(object(OBJECTREF,FIELD,VALUE)).
writeMemory(OBJECTREF,FIELD,VALUE) :-
{ },
assert(object(OBJECTREF,FIELD,VALUE)).

Read Memory (IN,IN,OUT)

readMemory(OBJECTREF,FIELD,VALUE) :-
{ },
object(OBJECTREF,FIELD,VALUE).

writeMemory(1,0,54)
writeMemory(1,1,22)
writeMemory(1,2,12)

writeMemory(2,0,14)
writeMemory(2,1,18)



La représentation du heap

Write Memory (IN,IN,IN)

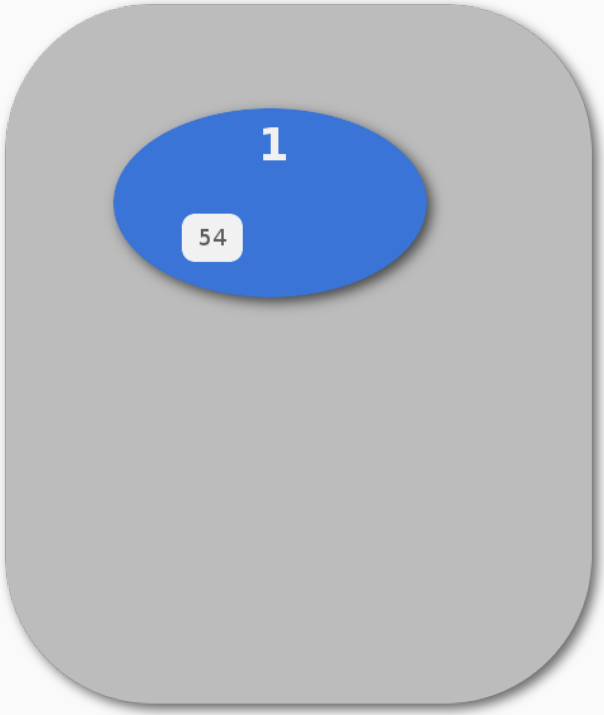
writeMemory(OBJECTREF,FIELD,VALUE) :-
{ },
retract(object(OBJECTREF,FIELD,X)),
assert(object(OBJECTREF,FIELD,VALUE)).
writeMemory(OBJECTREF,FIELD,VALUE) :-
{ },
assert(object(OBJECTREF,FIELD,VALUE)).

Read Memory (IN,IN,OUT)

readMemory(OBJECTREF,FIELD,VALUE) :-
{ },
object(OBJECTREF,FIELD,VALUE).

writeMemory(1,0,54)
writeMemory(1,1,22)
writeMemory(1,2,12)

writeMemory(2,0,14)
writeMemory(2,1,18)



La représentation du heap

Write Memory (IN,IN,IN)

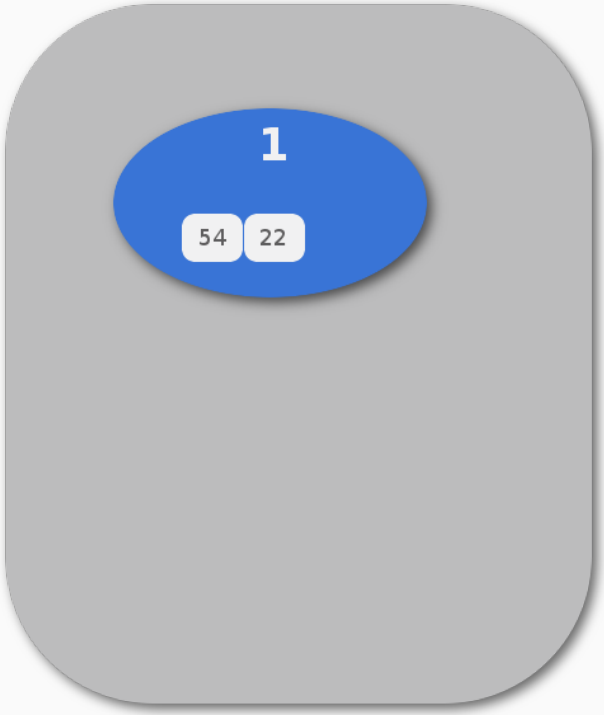
```
writeMemory(OBJECTREF,FIELD,VALUE) :-
    { },
    retract(object(OBJECTREF,FIELD,X)),
    assert(object(OBJECTREF,FIELD,VALUE)).
writeMemory(OBJECTREF,FIELD,VALUE) :-
    { },
    assert(object(OBJECTREF,FIELD,VALUE)).
```

Read Memory (IN,IN,OUT)

```
readMemory(OBJECTREF,FIELD,VALUE) :-
    { },
    object(OBJECTREF,FIELD,VALUE).
```

writeMemory(1,0,54)
writeMemory(1,1,22)
writeMemory(1,2,12)

writeMemory(2,0,14)
writeMemory(2,1,18)



La représentation du heap

Write Memory (IN,IN,IN)

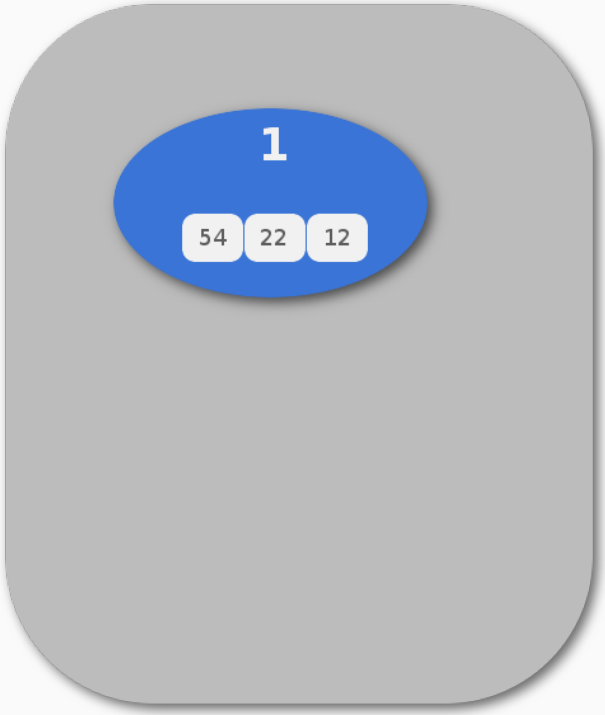
writeMemory(OBJECTREF,FIELD,VALUE) :-
{ },
retract(object(OBJECTREF,FIELD,X)),
assert(object(OBJECTREF,FIELD,VALUE)).
writeMemory(OBJECTREF,FIELD,VALUE) :-
{ },
assert(object(OBJECTREF,FIELD,VALUE)).

Read Memory (IN,IN,OUT)

readMemory(OBJECTREF,FIELD,VALUE) :-
{ },
object(OBJECTREF,FIELD,VALUE).

writeMemory(1,0,54)
writeMemory(1,1,22)
writeMemory(1,2,12)

writeMemory(2,0,14)
writeMemory(2,1,18)



La représentation du heap

Write Memory (IN,IN,IN)

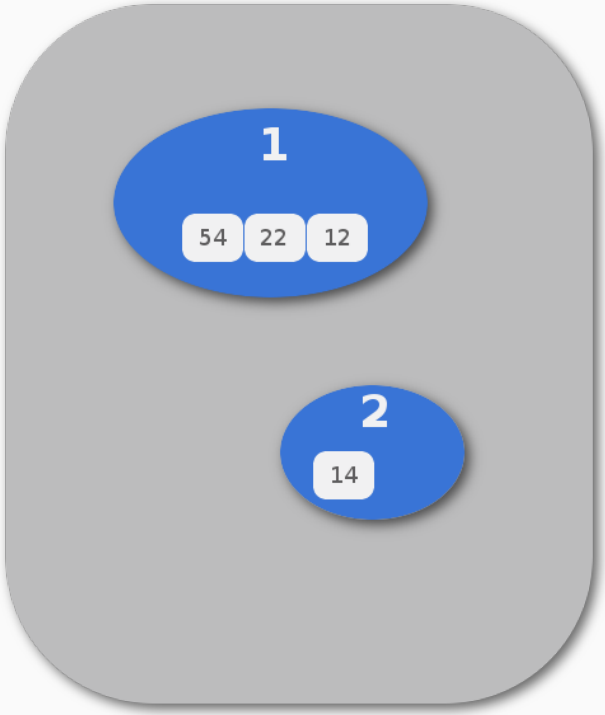
```
writeMemory(OBJECTREF,FIELD,VALUE) :-
    { },
    retract(object(OBJECTREF,FIELD,X)),
    assert(object(OBJECTREF,FIELD,VALUE)).
writeMemory(OBJECTREF,FIELD,VALUE) :-
    { },
    assert(object(OBJECTREF,FIELD,VALUE)).
```

Read Memory (IN,IN,OUT)

```
readMemory(OBJECTREF,FIELD,VALUE) :-
    { },
    object(OBJECTREF,FIELD,VALUE).
```

writeMemory(1,0,54)
writeMemory(1,1,22)
writeMemory(1,2,12)

writeMemory(2,0,14)
writeMemory(2,1,18)



La représentation du heap

Write Memory (IN,IN,IN)

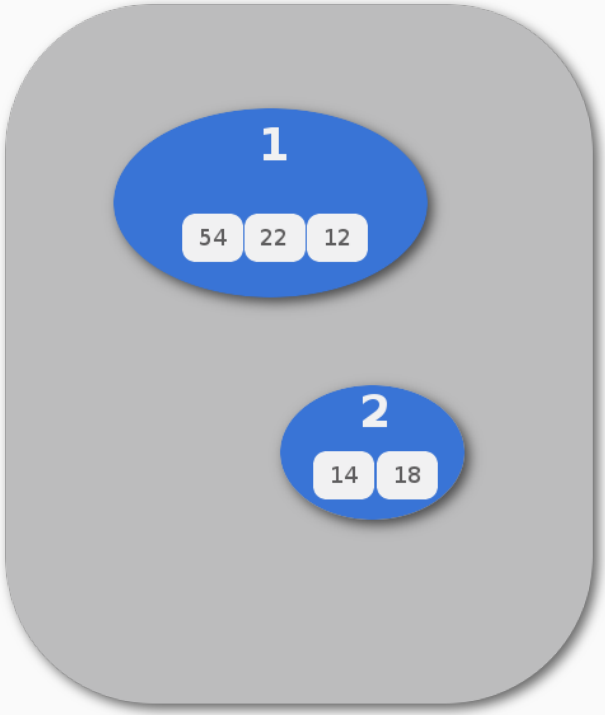
writeMemory(OBJECTREF,FIELD,VALUE) :-
{ },
retract(object(OBJECTREF,FIELD,X)),
assert(object(OBJECTREF,FIELD,VALUE)).
writeMemory(OBJECTREF,FIELD,VALUE) :-
{ },
assert(object(OBJECTREF,FIELD,VALUE)).

Read Memory (IN,IN,OUT)

readMemory(OBJECTREF,FIELD,VALUE) :-
{ },
object(OBJECTREF,FIELD,VALUE).

writeMemory(1,0,54)
writeMemory(1,1,22)
writeMemory(1,2,12)

writeMemory(2,0,14)
writeMemory(2,1,18)



La gestion des tableaux

- Réutilisation du mécanisme de gestion des objets pour gérer les tableaux à une ou plusieurs dimensions
- Field -1 utilisé pour stocker la longueur du tableau (arraylength)
- Tableau multidimensionnel :
  Contient des références vers des tableaux

Les attributs 'static' : Initialisation des classes

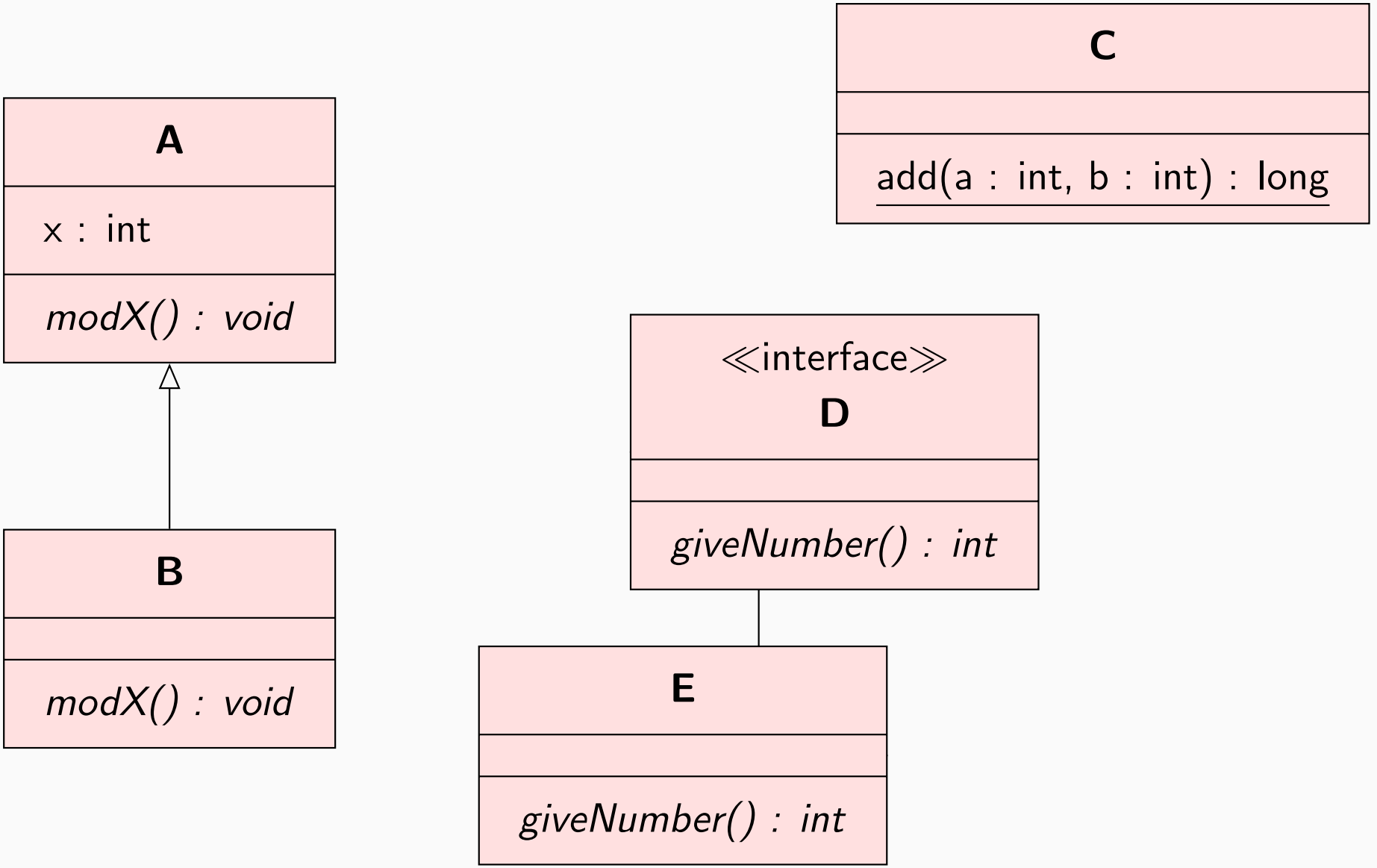
- $\exists$ attributs *static* commun à tous les objets d'une classe
- Espace d'adressage différent des objets traditionnels
- Initialisation exprimée dans la méthode $\langle$clinit$\rangle$
- Appel à $\langle$clinit$\rangle$ jamais exprimé dans le bytecode
- Gestion implicite par le classLoader de la JVM

$\rightarrow$ Simulé par des appels aux différentes méthodes $\langle$clinit$\rangle$ avant l'évaluation du point d'entrée du programme clp

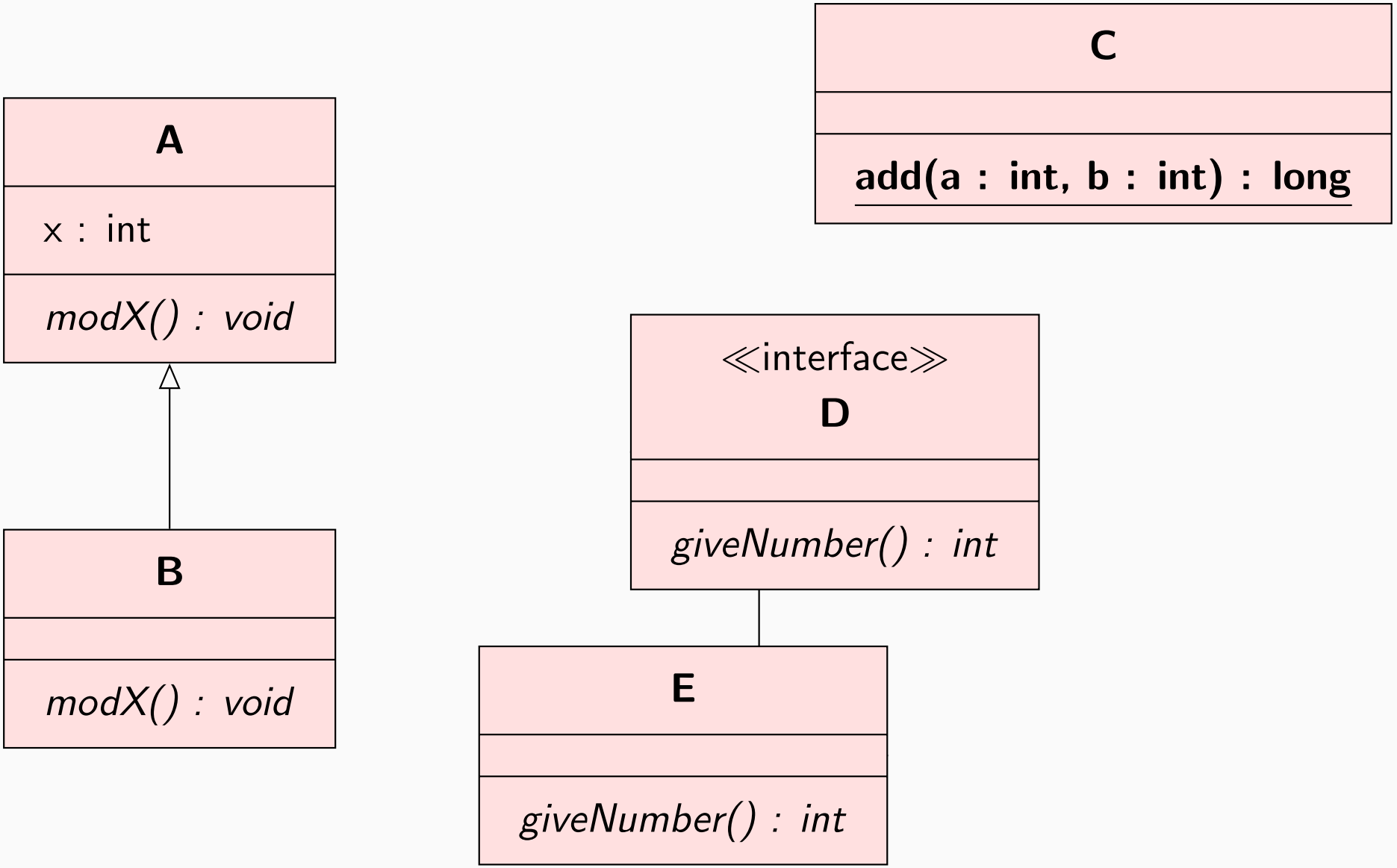
Pstart -: $\langle$clinit1$\rangle$, $\langle$clinit2$\rangle$, .., p0.

Les appels de méthodes

Piqûre de (r)appel : Quelques exemples

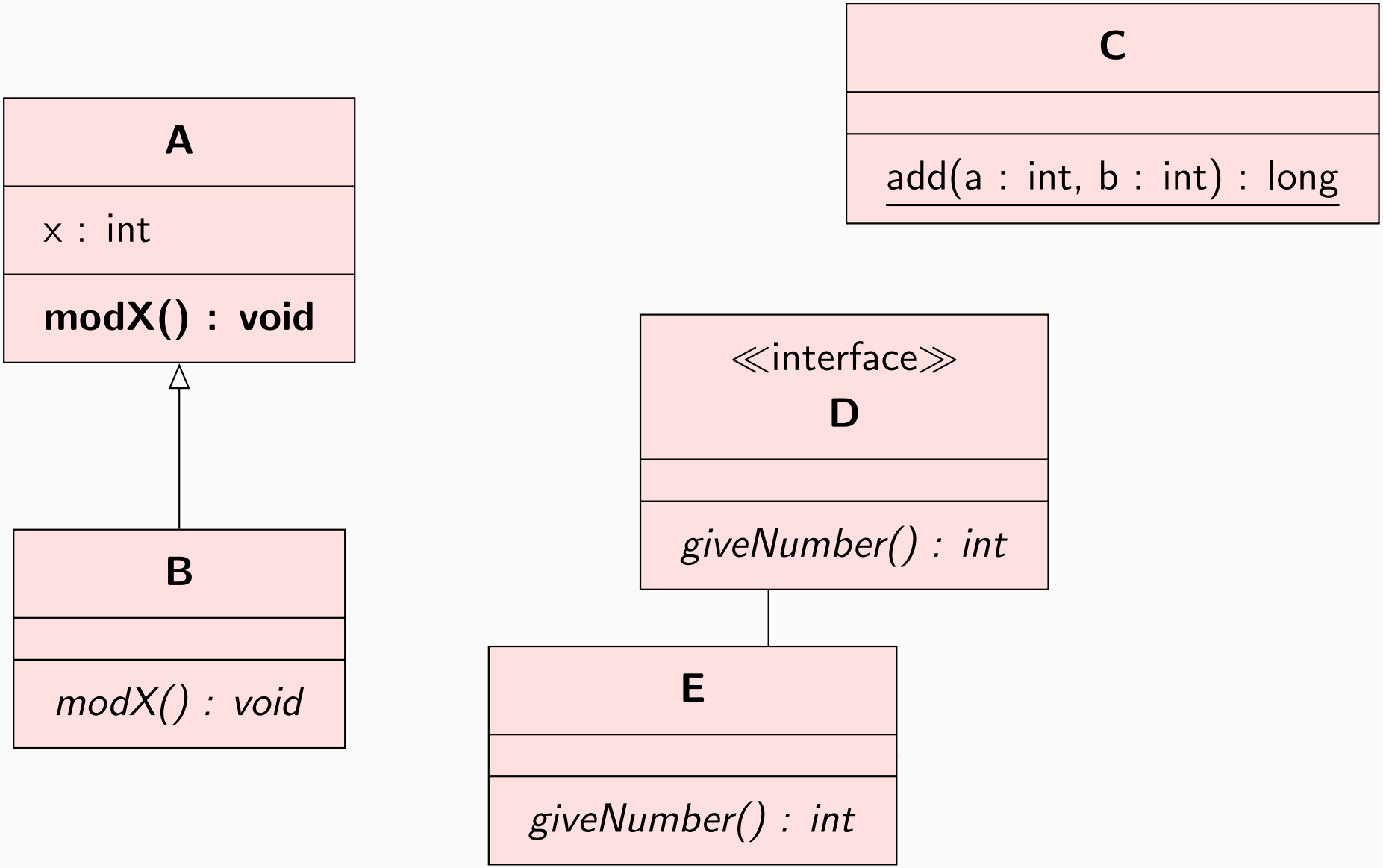


Piqûre de (r)appel : Quelques exemples



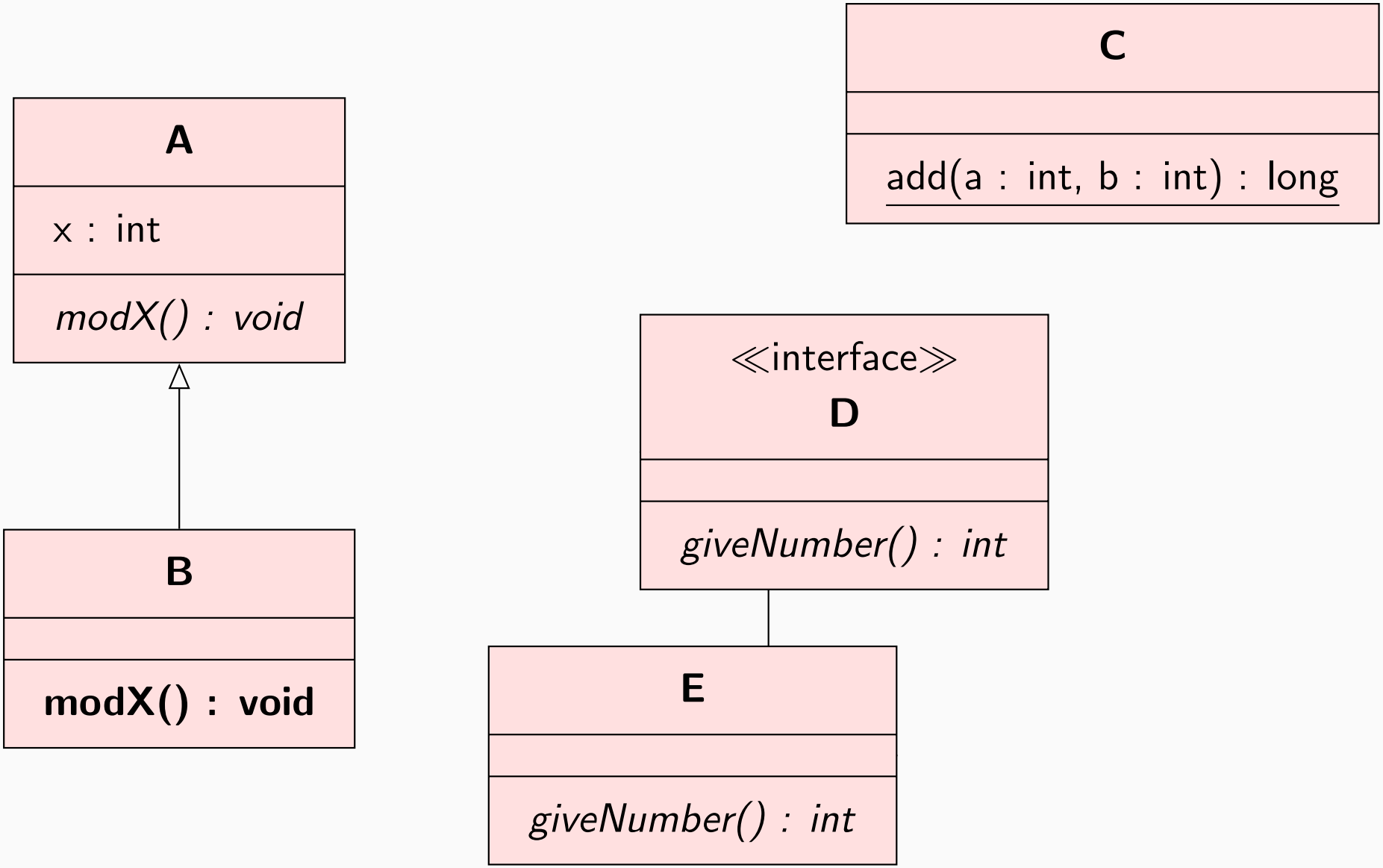
int a = C.add(5,3);

Piqûre de (r)appel : Quelques exemples



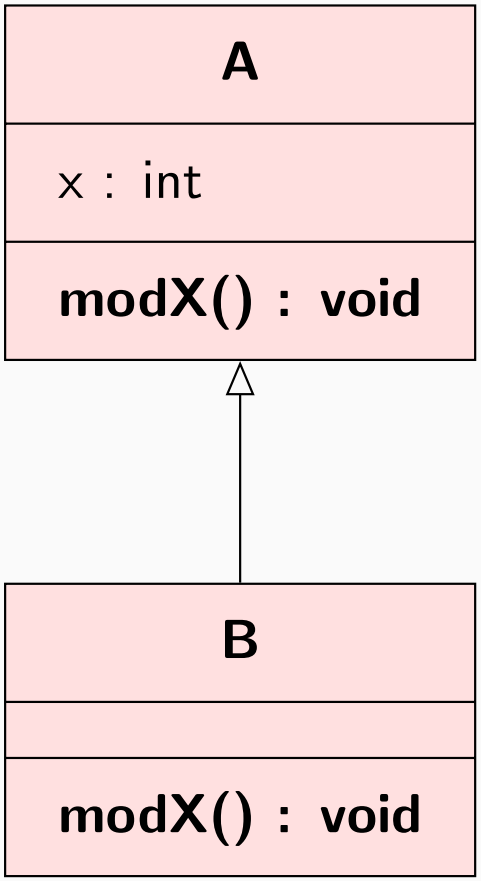
A oa = new A(); oa.modX();

Piqûre de (r)appel : Quelques exemples



A ob = new B(); ob.modX();

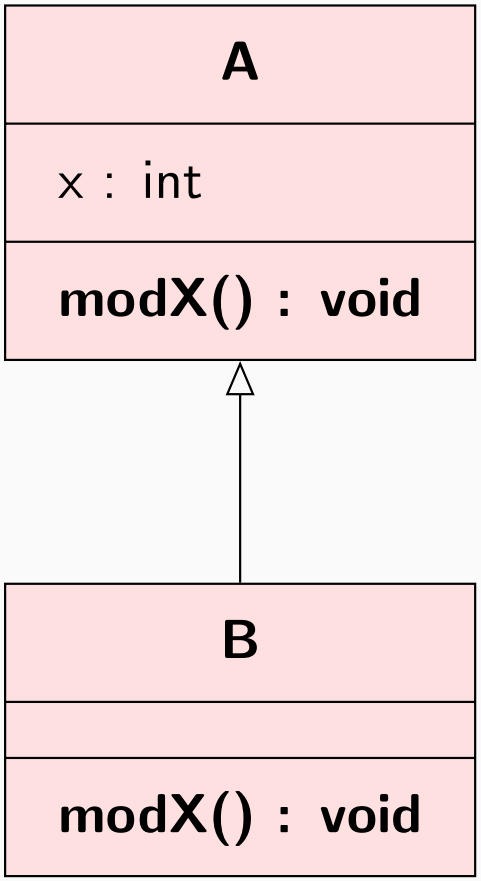
Piqûre de (r)appel : Quelques exemples



Une dernière pour les champions !

A o = (input ⟩ 0)? new A(); : new B(); o.modX();

Piqûre de (r)appel : Quelques exemples



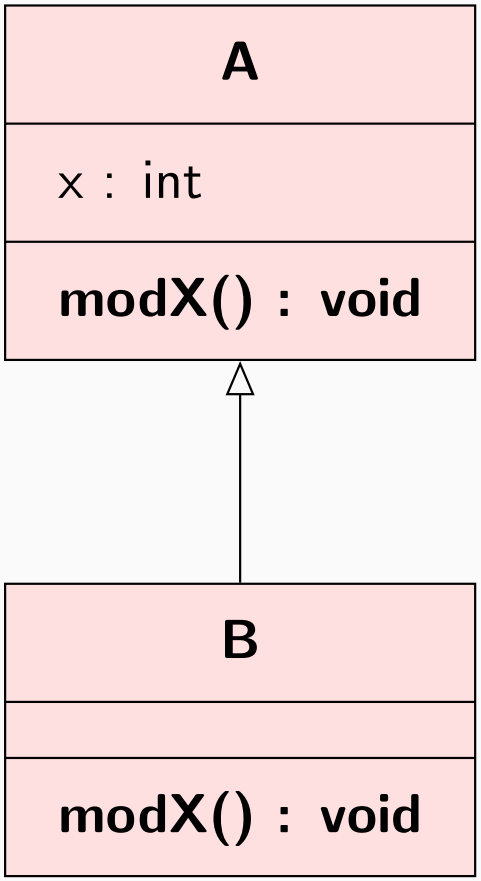
Une dernière pour les champions !

A o = (input ⟩ 0)? new A(); : new B(); o.modX();

Dépend de l'exécution...

→ **Mécanisme de lookup :**

Piqûre de (r)appel : Quelques exemples



Une dernière pour les champions !

A o = (input ⟩ 0)? new A(); : new B(); o.modX();

Dépend de l'exécution...

→ **Mécanisme de lookup :**

Sous forme de clauses, on défini de manière statique un mécanisme de lookup qui va choisir dynamiquement la bonne méthode à appeler en fonction du type réel de l'objet à l'exécution.

Les invocations de méthodes en bytecode

Les différentes instructions **invoke** référencent un lien symbolique qu'il faut résoudre différemment en fonction du type d'invocation.

invokestatic **(static binding)**

L'argument de l'instruction pointe vers une entrée de la *constant pool* qui contient une référence symbolique vers le nom de la classe où se trouve la méthode, le nom de la méthode à invoquer ainsi que les types des arguments

Pour résoudre cette référence (Savoir quelle clause appeler) :

- Parcours de l'arbre pour obtenir l'index de la classe, l'index de la méthode et le nombre d'arguments de la méthode.

Les invocations de méthodes en bytecode

Les différentes instructions **invoke** référencent un lien symbolique qu'il faut résoudre différemment en fonction du type d'invocation.

invokevirtual&invokeinterface **(dynamic binding)**

L'argument de l'instruction pointe vers une entrée de la *constant pool* qui contient une référence symbolique qui contient le nom de classe du type déclaré de l'instance, le nom de la méthode à invoquer et les types des arguments.

Pour résoudre cette référence (Savoir quelle clause appeler) :

- Mécanisme de lookup
- On génère une clause pour le type déclaré et chacune de ses sous classes. (Couvre tous les types effectifs possibles de l'instance)
- Pour chaque sous classe, on détermine la clause de l'appel en prenant la première méthode qui match en remontant la hiérarchie.

Les invocations de méthodes en bytecode

Les différentes instructions **invoke** référencent un lien symbolique qu'il faut résoudre différemment en fonction du type d'invocation.

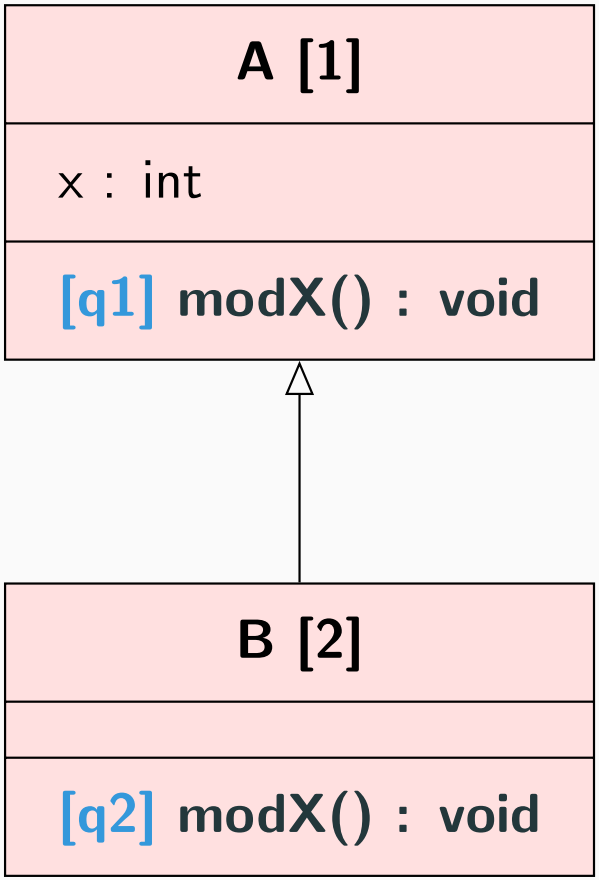
invokespecial **(static binding)**

L'argument de l'instruction pointe vers une entrée de la *constant pool* qui contient une référence symbolique qui contient le nom de classe du type déclaré de l'instance, le nom de la méthode à invoquer et les types des arguments.

Pour résoudre cette référence (Savoir quelle clause appeler) :

- Cas particulier de **invokevirtual** pour les constructeurs, les méthodes *private* et les méthodes invoquées via *super*
- Ces méthodes nécessitent un *static binding*, le *dynamic binding* mènerait à des résultats erronés

Exemple du mécanisme de lookup (invokevirtual)



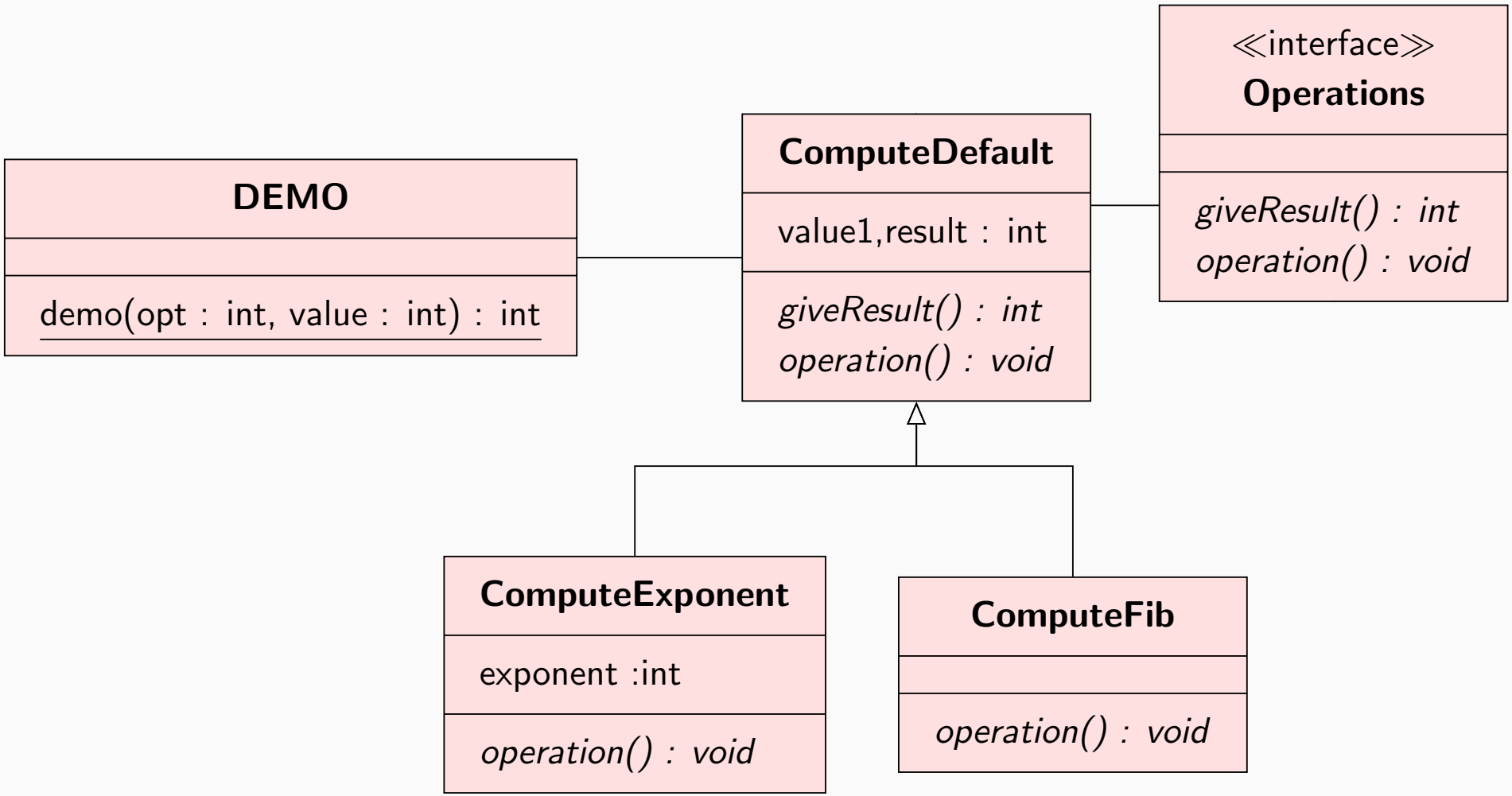
La réponse des champions.. (Version Prolog)

A o = (input 〉 0)? new A(); : new B(); **o.modX();**

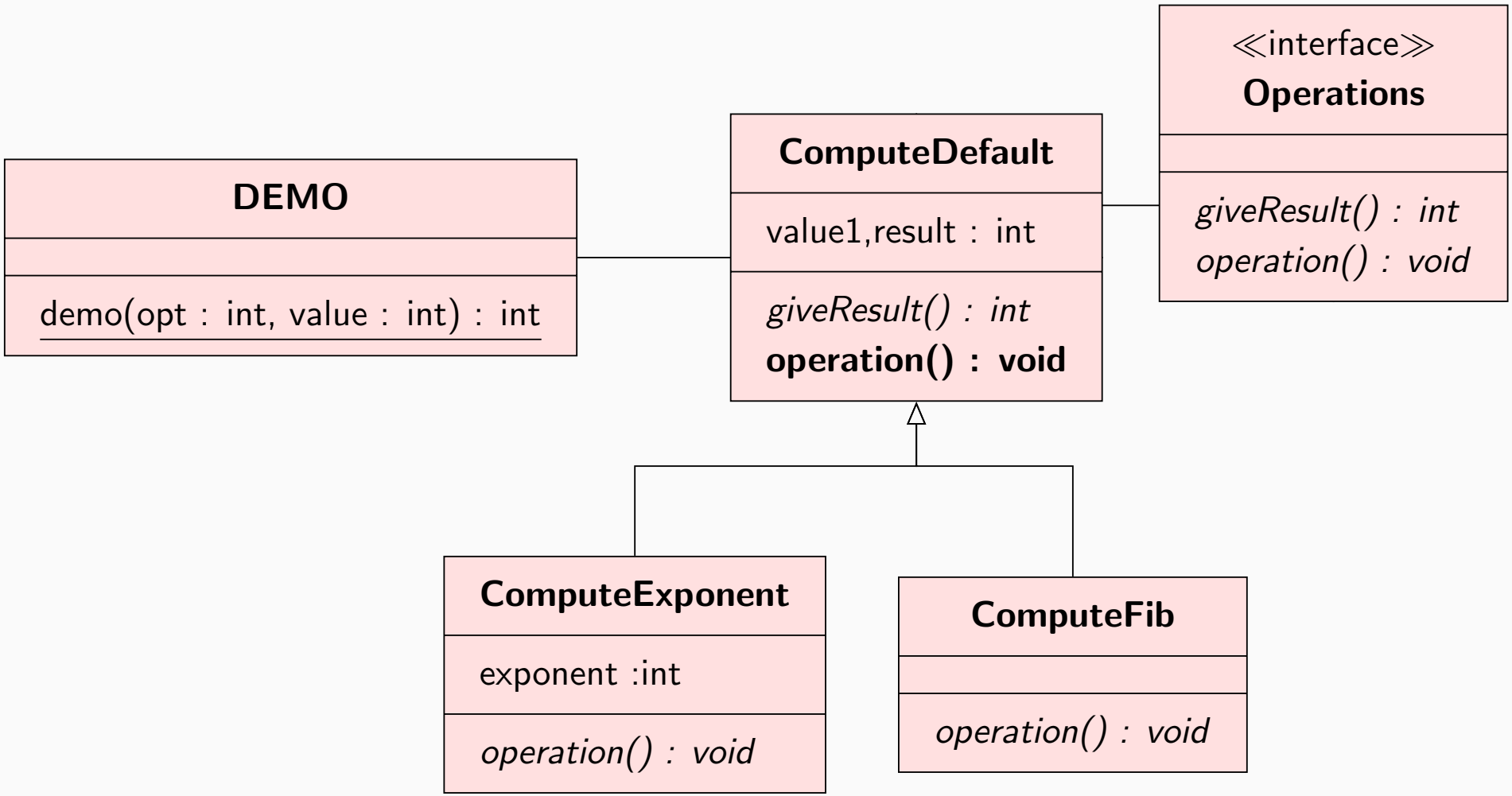
```
p1(IV0,IV1,IV2,SP_INDEX,IHS,OHS,R) :-
{ SP_INDEX_NEW = SP_INDEX -1,SP_INDEX_CALL = 0,
ClassID = 1},
read(SP_INDEX,IV0,IV1,IV2,READ_VALUE),
readMemory(READ_VALUE,0,ClassID),
q1(READ_VALUE,0,_,_,SP_INDEX_CALL,IHS,OHS_CALL,R_NEW),
p2(IV0,IV1,IV2,SP_INDEX_NEW,OHS_CALL,OHS,R).

p1(IV0,IV1,IV2,SP_INDEX,IHS,OHS,R) :-
{ SP_INDEX_NEW = SP_INDEX -1,SP_INDEX_CALL = 0,
ClassID = 2},
read(SP_INDEX,IV0,IV1,IV2,READ_VALUE),
readMemory(READ_VALUE,0,ClassID),
q2(READ_VALUE,0,_,_,SP_INDEX_CALL,IHS,OHS_CALL,R_NEW),
p2(IV0,IV1,IV2,SP_INDEX_NEW,OHS_CALL,OHS,R).
```

Démonstration

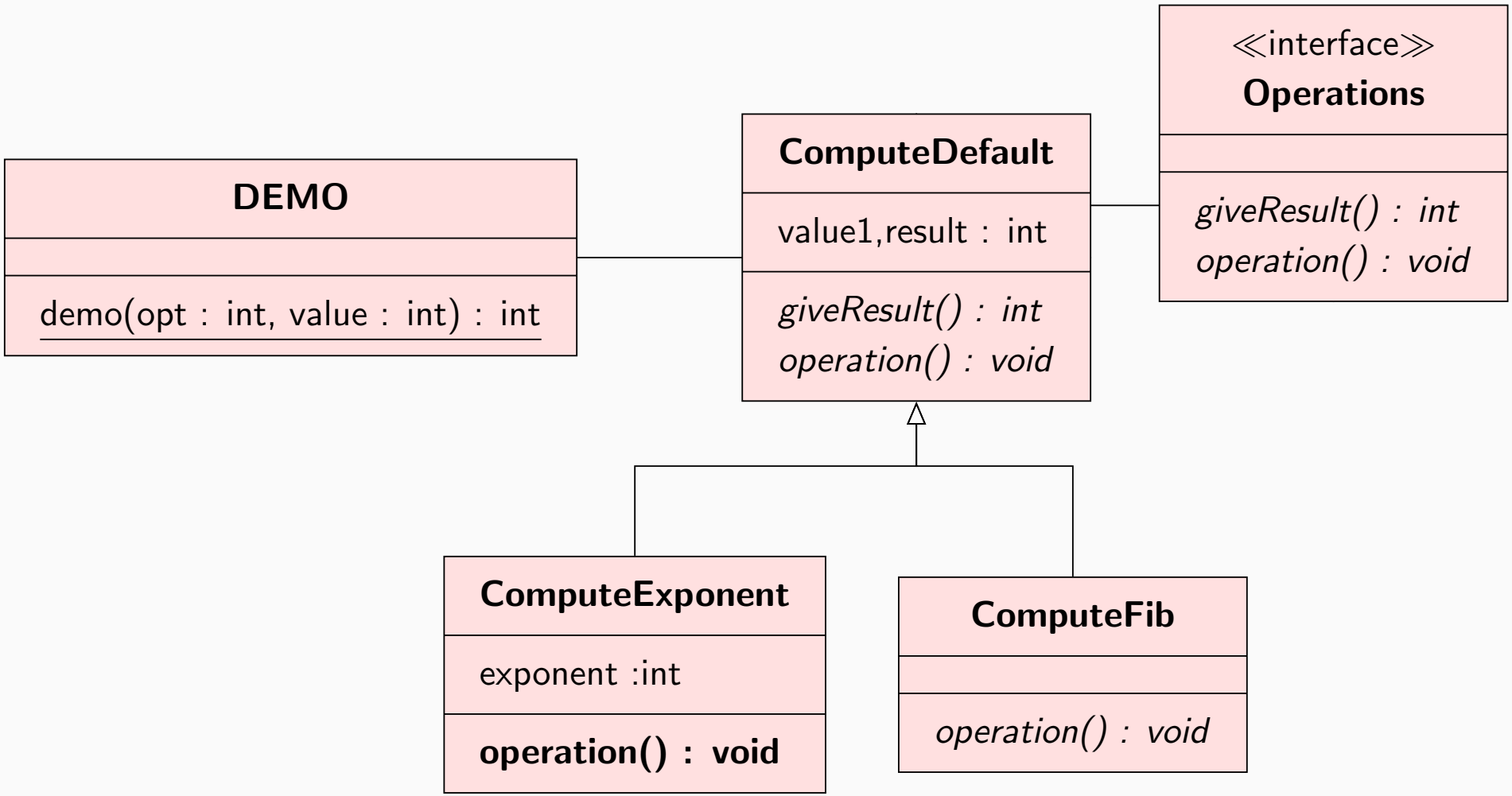
Un exemple concret..



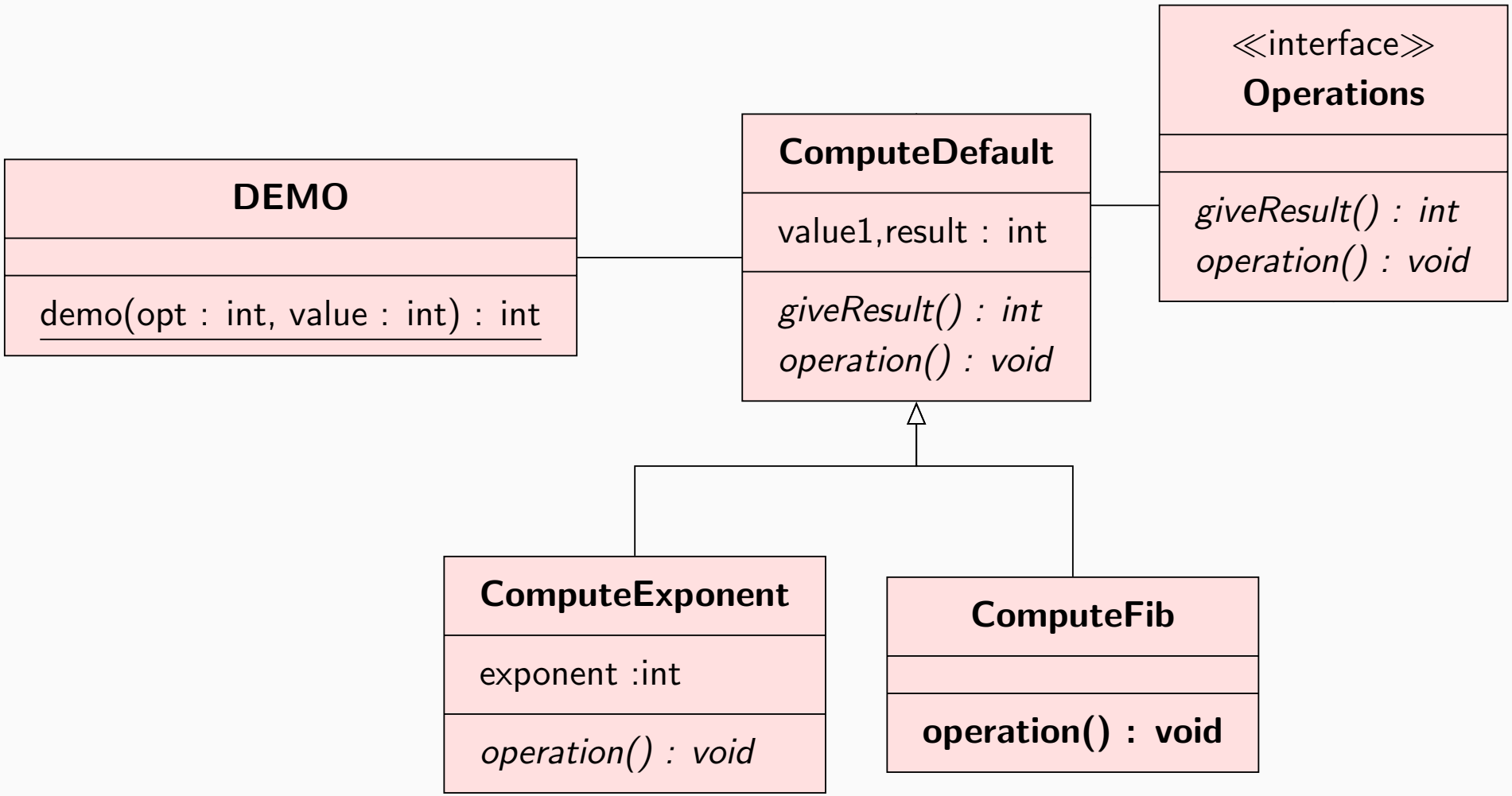
Un exemple concret..



Un exemple concret..



Un exemple concret..



Un exemple concret.. (DEMO.java)

```
public class DEMO {
   public static int demo(int opt, int value){
      int r = 0;
      int[] a = new int[1];
      a[0] = 3;

      ComputeDefault o;

      switch(opt) {
         case 1 :
            //Compute exponent (a[0])
            o = new ComputeExponent(value,a[0]);
            break;

case 2 :
   //Fibonacci [0, 1, 1, 2, 3, 5, 8, 13, 21]
   o = new ComputeFib(value);
   break;

default:
   //Compute factorial
   o = new ComputeDefault(value);
   break;
}

o.operation();
r = o.giveResult();
return r;
   }
}
```

Un exemple concret.. (ComputeDefault.java)

```
public class ComputeDefault implements Operations {
   protected int value1;
   protected int result;

   public ComputeDefault(int x) {
      this.value1 = x;
   }

   @Override
   public int giveResult() {
      return result;
   }

   @Override
   public void operation() { //Compute Factorial
      int fact = 1;
      int i = value1;
      while(i >= 1){
         fact = fact*i;
         i-  -;
      }
      result=fact;
   }
}
```

Un exemple concret.. (ComputeExponent.java)

```
public class ComputeExponent extends ComputeDefault{
   int exponent;
   public ComputeExponent(int base, int exponent) {
      super(base);
      this.exponent = exponent;
   }

   @Override
   public void operation() {
      int r = 1;
      int i = 0;
      while(i⟩exponent){
         i++;
         r = r * value1;
      }
      result = r;
   }

}
```

Un exemple concret.. (ComputeFib.java)

```
public class ComputeFib extends ComputeDefault {
  public ComputeFib(int x) {
    super(x);
  }

  @Override
  public void operation() {
    result = fib(value1);
  }

  private static int fib(int n) {
    if (n < 2) return(n);
    return( fib(n-2) + fib(n-1) );
  }

}
```

A venir..

Ce qu'il reste à faire

Optimisations

Gestion du code non inclus dans le jar

Ce qu'il reste à faire

Optimisations

Gestion du code non inclus dans le jar

refactoring

MERCI DE VOTRE ATTENTION