Université Paris Cité

L1 informatique - Année 2022–2023

TD de Concepts Informatique n° 7 (Correction)

Dans l'Exercice 2 ci-dessous, nous utiliserons les classes `Bloc`, `BlocR`, `BlocF` et `BlocG`.

- La classe `Bloc` est utilisée pour représenter un bloc d'activation minimal pour une fonction `void foo()` avec type de retour `void`, sans paramètres et sans variables locales.
- La classe `BlocR` est utilisée pour modéliser un bloc d'activation pour une fonction de la forme `int foo()` sans variables locales. Rappelons que `BlocR` peut être vu comme un sous-type de `Bloc` dont il hérite l'adresse de retour.
- Remarquez que la classe `BlocF` est utilisé pour modéliser un bloc d'activation pour les appels de la fonction `f` de l'Exercice 2. En plus de l'adresse de retour (héritée de la classe `Bloc`), cette classe possède également des attributs pour un argument et une variable locale, tous deux de type `int`.
- De même, la classe `BlocG` est utilisée pour modéliser un bloc d'activation pour les appels de la fonction `g` de l'Exercice 2. En plus de la valeur de retour de type `int` et de l'adresse de retour (qui sont héritées de `BlocR`), cette classe possède également des attributs pour deux arguments de type `int`.

```
public class Bloc {
  private int adresseRetour;

  public Bloc(int adr) {
    this.adresseRetour = adr;
  }

  public int getAdresse() {
    return this.adresseRetour;
  }
}
```

```
public class BlocR extends Bloc{
  private int valeurDeRetour;

  public BlocR(int adresseDeRetour) {
    super(adresseDeRetour);
  }

  public int getVal() {
    return this.valeurDeRetour;
  }

  public void setVal(int valeurDeRetour) {
    this.valeurDeRetour = valeurDeRetour;
  }
}
```

```
public class BlocF extends Bloc{
  private int argument1;
  private int variable1;

  public BlocF(int adr, int arg1) {
    super(adr);
    this.argument1 = arg1;
  }

  public int getArg1() {
    return this.argument1;
  }

  public int getVar1() {
    return this.variable1;
  }

  public void setVar1(int variable1) {
    this.variable1 = variable1;
  }
}
```

```
public class BlocG extends BlocR{
  private int argument1;
  private int argument2;

  public BlocG(int adr, int arg1, int arg2) {
    super(adr);
    this.argument1 = arg1;
    this.argument2 = arg2;
  }

  public int getArg1() {
    return this.argument1;
  }

  public int getArg2() {
    return this.argument2;
  }
}
```

Exercice 1. *Blocs d'activation.*

1. Donner un diagramme pour représenter la relation de sous-typage entre ces classes.
2. Lesquelles des lignes suivantes compilent ?
   `Bloc b1 = new BlocF(2, 7);`
   `BlocF b2 = new Bloc(2);`
   `BlocF b3 = (BlocF)b1;`
   `BlocR b4 = (BlocR)b1;`
   Correction : `Bloc b1 = new BlocF(2, 7);` et `BlocF b3 = (BlocF)b1;`
3. Nous allons modéliser la pile d'exécution avec un objet de type `Stack<Bloc>`. Justifier ce choix.
   Correction : `Bloc` représente le bloc d'activation minimal, toutes les autres classes que nous pourrions utiliser en sont des sous-types. Ainsi à une pile de type `Stack<Bloc>` on peut ajouter des objets de toutes les classes BlocF, BlocG, etc. qui étendent Bloc. Nous pourrons alors utiliser les fonctionnalités supplémentaires de ces classes en utilisant un opérateur de conversion ou *downcast*.
4. Pour chaque méthode dans les classes ci-dessus, précisez si elle est utilisée par le code appelant ou le code appelé et justifiez.
   Correction :
5. les constructeurs sont utilisés par le code appelant
6. `getAdresse` – code appelé
7. `getVal` – code appelant
8. `setVal` – code appelé
9. `getArg`, `setVar`, `getVar` – code appelé

Exercice 2. *Appel général.*
On considére le programme suivant :

```
public class AppelFG {
  static int a = 2;
  static int b = 1;

  public static void f(int i) {
    int tmp = 0;
    if (i % 2 == 0) {
      a = 3 * a;
      b = 2 * b;
    } else {
      tmp = a;
      a = b;
      b = tmp;
    }
  }

  public static int g(int x, int y) {
    return x * a + y * b;
  }

  public static void main(String[] args) {
    int [] t = {2,3,3,4};
    for (int i = 0; i < 4; i++) {
      f(t[i]);
    }
    System.out.println(g(1,6));
  }
}
```

1. Que affiche le programme ?
   Correction : 42

2. Annoter le code.
   Correction :

```
public class AppelFG {
  static int a = 2;     // mem[0]
  static int b = 1;     // mem[1]

  public static void f(int i) {
    int tmp = 0;                // 100
    if (i % 2 == 0) {           // 101 saut cond.
      a = 3 * a;                // 102
      b = 2 * b;                // 103
    }                           // 104 saut incond.
      else {
      tmp = a;                  // 105
      a = b;                    // 106
      b = tmp;                  // 107
    }
  }                             // 108 sortie de f

  public static int g(int x, int y) {
    return x * a + y * b;       // 200 m.a.j. de la valuer de retour de g
 }                               // 201 sortie de g

  public static void main(String[] args) {
    int [] t = {2,3,3,4};               // 0 allouer la memoire pour t
                                        // 1, 2, 3, 4  initialiser t[0],..., t[3]
    for (int i = 0; i < 4; i++) {       // 5 ( i = 0), 6 saut cond, 11 i++
      f(t[i]);                          // 7 appel
                                        // 8 retour
    }                                   // 9 saut incond.
    System.out.println(g(1,6));         // 10 appel
                                        // 11 retour
  }                                     // 12 sortie du main
  }
```

3. Traduire le programme.
   Correction :

```
public class AppelFGTraduit {
  static int ic = 0;
  static int mem[] = new int[1000];
  static Stack<Bloc> p = new Stack<Bloc>();
  public static void main(String[] args) {
    mem[0] = 2;
    mem[1] = 1;
    while(true) {
      switch(ic) {
        case 0:  mem[2] = 200; ic++; break;
        case 1:  mem[mem[2]] = 2; ic++; break;
        case 2:  mem[mem[2]+1] = 3; ic++; break;
        case 3:  mem[mem[2]+2] = 3; ic++; break;
        case 4:  mem[mem[2]+3] = 4; ic++; break;
        case 5:  mem[3] = 0;  ic++; break;
        case 6:  if (mem[3] >= 4) {
                   ic += 5;
                 } else {
                   ic++;
                 } break;
        case 7:   p.push(new BlocF(ic+1, mem[mem[2]+mem[3]])); ic = 100; break;
        case 8:   p.pop(); ic++; break;
        case 9:   mem[3]++; ic++; break;
        case 10:  ic-=4; break;
        case 11:  p.push(new BlocG(ic+1,1,6)); ic = 200; break;
        case 12:  System.out.println(((BlocR)p.pop()).getVal()); ic++; break;
        case 13:  System.exit(0);

        case 100: ((BlocF)p.peek()).setVar1(0); ic++; break;
        case 101: if (((BlocF)p.peek()).getArg1() % 2 != 0) {
                    ic+=4;
                  } else {
                    ic++;
                  } break;
        case 102:  mem [0] = 3 * mem[0]; ic++; break;
        case 103:  mem[1] = 2 * mem[1]; ic++; break;
        case 104:  ic += 4; break;
        case 105:  ((BlocF)p.peek()).setVar1(mem[0]); ic++; break;
        case 106:  mem[0] = mem[1]; ic++; break;
        case 107:  mem[1] = ((BlocF)p.peek()).getVar1(); ic++; break;
        case 108:  ic = p.peek().getAdr(); break;

        case 200: ((BlocR)p.peek()).setVal(
                    ((BlocG)p.peek()).getArg1() * mem[0] +
                    ((BlocG)p.peek()).getArg2() * mem[1]
                  ); ic++; break;
        case 201:  ic = p.peek().getAdr(); break;
      }
    }
  }
}
```

4. Décrire l'évolution de la pile d'appel (chaque `push`, chaque `pop`).