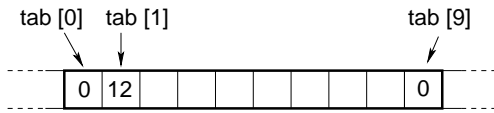


Les objets

Des tableaux aux objets

Un tableau regroupe plusieurs données de même type sous un même nom de variable.

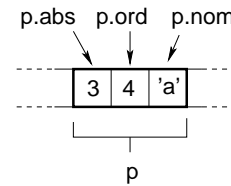


```
int[] tab = new int[10];  
tab[1] = 12;
```

Un *objet* permet de regrouper plusieurs données de types éventuellement différents sous un même nom de variable.

En Java, tout objet doit être construit suivant un certain *modèle*, appelé sa *classe*.

1

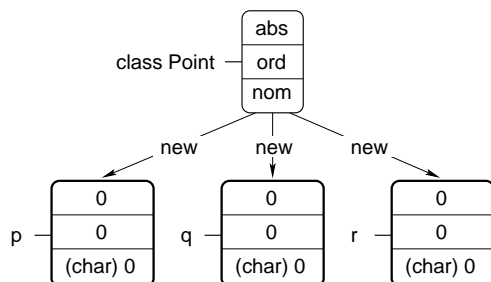


```
/* modèle abstrait de tout point */  
class Point {  
    int abs;  
    int ord;  
    char nom;  
}  
public class MonProgramme {  
    public static void main(String[] args) {  
        /* création d'un nouveau point */  
        Point p = new Point();  
        /* accès au contenu du point p */  
        p.abs = 3;  
        p.ord = 4;  
        p.nom = 'a';  
    }  
}
```

2

- La déclaration d'une classe ne crée aucun objet, mais simplement un *modèle* d'objet.
- Tout objet doit être créé à l'aide de l'opérateur **new**.
- Les composantes d'un objet sont appelées ses *champs*. Chaque champ d'un objet a une valeur *nulle* après création de cet objet.

```
Point p = new Point(),  
      q = new Point(),  
      r = new Point();
```



3

Objets et Références

Les variables **p**, **q**, **r**... ne sont pas en elles-mêmes des objets, mais servent à *désigner des objets stockés en mémoire*. Une variable d'objet s'appelle aussi une *référence*.

```
Point p = new Point();
```

est décomposable en

```
Point p;  
p = new Point();
```

- "Point p;" soit **p** une nouvelle référence, ne désignant pour l'instant aucun objet particulier.
- "new Point()" soit un nouvel objet de la classe Point.
- "p =" désigner par **p** l'objet mentionné après le symbole d'affectation.

4

- Chaque référence désigne au plus un objet à un instant donné.
- Deux références différentes peuvent, à un instant donné, désigner le même objet.

```

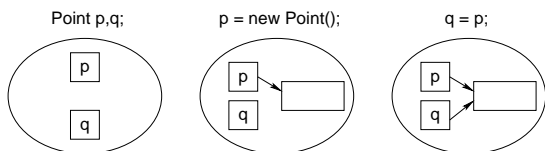
/* soient deux nouvelles références */
Point p, q;

/* créer un nouveau point, et le désigner par p */
p = new Point();

/* désigner également par q l'objet désigné par p */
q = p;

/* p.abs et q.abs sont maintenant synonymes,
 * de même que p.ord et q.ord, etc...
 */

```



Note plus technique

Une référence contient en fait l'adresse de l'objet qu'elle désigne - concrètement, sous la forme d'un numéro de case mémoire.

On dit aussi qu'elle est une référence vers cet objet, ou simplement qu'elle référence cet objet.

`q = p;` copie dans `q` l'adresse stockée dans `p`. Après cette instruction, `p` et `q` contiennent la même adresse, donc référencent le même objet.

Dans `p = new Point();` l'opérateur `new` crée un nouvel objet et renvoie l'adresse de l'objet créé : cette adresse est stockée dans `p`.

- Une même référence peut désigner des objets différents au cours du temps.
- Lorsqu'un objet n'est plus désigné par aucune référence, il devient inaccessible : il est automatiquement effacé de la mémoire.

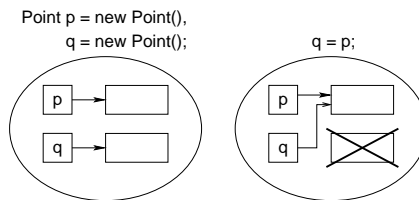
```

/* créer deux nouveaux objets, désignés par p et q */
Point p = new Point(), q = new Point();

/* désigner à présent par q l'objet désigné par p */
q = p;

/* l'objet désigné au départ par q est maintenant */
/* inaccessible : son contenu est définitivement */
/* perdu, et sera effacé de la mémoire.
 */

```



Abus de langage usuel

Lorsqu'aucune ambiguïté n'est possible, on peut se permettre de confondre :

- la référence `p` vers un certain objet
- cet objet lui-même

et parler ainsi du *point* `p` au lieu, formellement :

- de l'objet de la classe `Point` désigné par la référence `p`, ou bien,
- de l'objet de la classe `Point` référencé par `p`, ou encore,
- de l'objet de la classe `Point` vers lequel `p` est une référence.

Passage ou retour de références

Une fonction peut prendre en argument une référence et/ou renvoyer une référence.

Par abus de langage, on peut dire qu'elle prend en argument un objet et/ou renvoie un objet.

```
static Point dupliquer(Point a) {
    Point b = new Point();
    b.abs = a.abs;
    b.ord = a.ord;
    b.nom = a.nom;
    return b;
}
```

Formellement : **dupliquer** crée une copie de l'objet désigné par **a**, et renvoie la valeur d'une référence vers cette copie.

Informellement : **dupliquer** crée une copie de **a**, et renvoie cette copie.

9

Exemple d'appel

```
Point p = new Point();
p.abs = 10; p.ord = 20; p.nom = 'a';
Point q = dupliquer(p);

/* q désigne à présent l'objet créé par
 * dupliquer, distinct en mémoire mais de
 * même contenu que l'objet désigné par p.
 */
```

Exercices

1. Dans le code précédent, que se passerait-il en mémoire si l'on remplaçait l'instruction

```
Point q = dupliquer(p);
par : p = dupliquer(p); ?
```

2. Quel serait l'effet des deux instructions précédentes si l'on avait défini **dupliquer** par :

```
static Point dupliquer(Point a) {
    Point b = a;
    return b;
}
```

10

3. Ecrire une fonction

```
static Point nouveau(int x, int y, char c)
renvoyant un nouveau point dont les champs seront initialisés à x, y et c.
```

4. A l'aide de **nouveau**, écrire une fonction

```
static Point translate(Point a, int dx, int dy)
renvoyant un nouveau point identique au point a, mais translaté de dx horizontalement et de dy verticalement.
```

5. Ecrire une fonction

```
static Point milieu(Point a, Point b, char c)
renvoyant un nouveau point de nom c, situé au milieu du segment ayant pour extrémités les points a et b.
```

11

Action sur un objet dans une fonction

Comme pour les tableaux, le passage d'une référence à une fonction permet à cette fonction de modifier le contenu de l'objet désigné.

```
class Point {
    int abs, ord;
    char nom;
}

public class MonProgramme {
    static void reset(Point a) {
        a.abs = 0;
        a.ord = 0;
        a.nom = (char) 0;
    }

    public static void main(String[] args) {
        Point p = new Point();

        p.abs = 10; p.ord = 20; p.nom = 'a';
        /* les champs de l'objet désigné par p
         * valent 10, 20 et 'a'
         */
        reset(p);
        /* ces champs sont tous revenus à 0
         */
    }
}
```

12

Note technique

Rappelons que si **f** est définie par

```
static void f(int a) { ... }
```

alors exécuter **f(e)** revient à exécuter le code de **f** après avoir déclaré une nouvelle variable **int a** initialisée par **a = e**.

De la même façon, exécuter **reset(p)** revient à exécuter le code de **reset** en déclarant une nouvelle référence **Point a** initialisée par **a = p**.

Durant toute l'exécution de ce code, **a** et **p** désignent donc le *même objet*, **a.abs** est synonyme de **p.abs**, etc... Les champs de l'objet désigné par **p** seront donc bien remis à zéro.

13

Exercices

1. Ecrire une fonction

```
static void deplacer(Point a, int x, int y)
```

remplaçant par **x** l'abscisse du point **a**, et par **y** son ordonnée.

2. A l'aide de **deplacer**, écrire une fonction

```
static void translater(Point a, int dx, int dy)
```

ajoutant **dx** à l'abscisse du point **a**, et **dy** à son ordonnée.

3. A l'aide de **translater**, écrire une fonction

```
static void ajouter(Point a, Point b)
```

ajoutant à l'abscisse et à l'ordonnée du point **a**, respectivement, l'abscisse et l'ordonnée du point **b**.

14

Initialisation des objets

On peut spécifier, pour un ou plusieurs champs, une autre valeur que la valeur nulle comme valeur d'initialisation :

```
class Point {
    int abs, ord;
    char nom = 'a';

    /* Tout point sera créé avec un champ nom
    * initialisé automatiquement à 'a'.
    *
    * Les champs abs et ord garderont leur
    * valeur par défaut, c'est-à-dire 0.
    */
}
```

15

Initialisation par constructeurs

On peut également ajouter à une classe du code spécialement dédié à l'initialisation des objets de cette classe.

Ce code peut recevoir des arguments.

```
class Point {
    int abs;
    int ord;
    char nom;

    Point() {
        nom = 'a';
    }

    Point(int x, int y) {
        abs = x;
        ord = y;
        nom = 'a';
    }

    Point(int x, int y, char c) {
        abs = x;
        ord = y;
        nom = c;
    }
}
```

16

Chaque fragment de code de la forme `Point(...){...}` est appelé un *constructeur*.

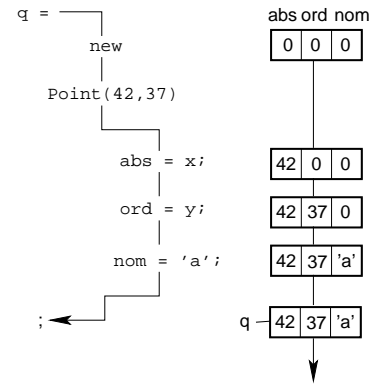
Pour appeler un constructeur particulier, il suffit de lui fournir une suite d'arguments du bon type, immédiatement après un `new` :

```
Point p = new Point();
/* crée un point, puis appelle le 1er
 * constructeur, qui initialise le champ
 * nom de ce point à la valeur 'a'.
 */

Point q = new Point(42,37);
/* crée un point, puis appelle le 2nd
 * constructeur, qui initialise ses
 * champs abs, ord, nom à : 42, 37, 'a'.
 */

Point r = new Point(12,23,'b');
/* crée un point, puis appelle le 3ème
 * constructeur, qui initialise ses
 * champs abs, ord, nom à : 12, 23, 'b'.
 */
```

- Les noms de champs mentionnés dans un constructeur désigneront, à l'exécution, les *champs de l'objet créé à l'aide du new qui précède l'appel de ce constructeur*.



Une classe peut contenir un nombre arbitraire de constructeurs, mais :

- tous doivent porter le nom de la classe `Point()`, `Point(int x, int y) ...`
- tous doivent avoir des suites d'arguments distinctes, au nom des paramètres près :

```
Point(int x, int y) { ... }
Point(int x, int y, char c) { ... }

sont considérés comme des constructeurs
distincts, mais pas

Point(int x, int y) { ... }
Point(int z, int t) { ... } // erreur !
```

Si l'on ne déclare aucun constructeur, c'est comme si l'on en déclarait qu'un seul, sans arguments, et ne faisant rien :

```
Point(){ } // constructeur par défaut
```

Remarque. Un constructeur n'est **pas** une fonction :

- il doit porter le nom de sa classe, et être placé dans celle-ci,
- il n'a pas de type de retour (pas même `void`) et ne peut renvoyer aucune valeur,
- il ne peut être appelé explicitement. Il est appelé implicitement par l'opérateur `new`, afin d'initialiser l'objet créé.

Exercice

Ecrire un constructeur `Point(Point a)`, initialisant les champs d'un point aux mêmes valeurs que celles des champs du point `a`.

Encapsulation d'objets

Un champ d'un objet peut être une référence vers un autre objet :

```
class Point {
    int abs, ord;
    char nom;
}
class Cercle {
    Point centre;
    int rayon;
}
public class MonProgramme {
    public static void main(String[] args) {

        /* création d'un cercle sans centre */
        Cercle c = new Cercle();

        /* création du centre */
        Point p = new Point();
        p.abs = 10; p.ord = 20; p.nom = 'a';

        /* initialisation des champs du cercle */
        c.rayon = 1;
        c.centre = p;
    }
}
```

21

La création complète d'un cercle nécessite de créer deux objets : un cercle sans centre, et un point servant de centre.

1. `Cercle c = new Cercle();`

ne crée qu'un seul objet de la classe `Cercle`. Le `new` ne crée aucun objet de la classe `Point`.

Après cette instruction `c.centre` ne désigne encore aucun point : le cercle n'a pas encore de centre.

2. `Point p = new Point();`

crée un nouveau point.

3. `c.centre = p`

force `c.centre` et `p` à désigner le même objet.

`c.centre.abs` et `p.abs` sont maintenant synonymes, de même que `c.centre.ord` et `p.ord`, etc.

22

Le programme suivant délègue la création du centre d'un cercle à un constructeur.

```
class Point {
    int abs, ord;
    char nom;
}
class Cercle {
    Point centre;
    int rayon;
    Cercle() {
        centre = new Point();
    }
}
public class MonProgramme {
    public static void main(String[] args) {

        /* création d'un cercle, donc, création
         * du centre par le constructeur appelé.
         */
        Cercle c = new Cercle();

        /* initialisation du rayon */
        c.rayon = 1;

        /* initialisation du centre */
        c.centre.abs = 10;
        c.centre.ord = 20;
        c.centre.nom = 'a';
    }
}
```

23

Exercices

1. Ecrire dans la classe `Cercle` un constructeur `Cercle(Point p)`. Ce constructeur devra créer un nouveau point comme centre du cercle, et initialiser ses champs aux mêmes valeurs que `p`.

2. A partir de la classe `Point`, définir une nouvelle classe `Segment` contenant deux champs `debut` et `fin` de type `Point`, représentant les extrémités d'un segment. Ecrire ensuite dans cette classe un constructeur

```
Segment(int x0, int y0, char c0,
        int x1, int y1, char c1)
```

Cette constructeur devra :

- créer chaque extrémité du segment,
- initialiser le début du segment à `x0, y0, c0`,
- initialiser la fin du segment à `x1, y1, c1`,

24

Tableaux d'objets

La création d'un tableau d'objets mime celle d'un tableau usuel. `Point[]` est le type des tableaux de références vers les points.

Elle se fait en deux étapes :

```
class Point {
    int abs, ord;
    char nom;
}

public class MonProgramme {
    public static void main(String[] args) {
        /* création d'un tableau d'objets */

        /* 1. création du contenant */
        Point[] tab = new Point[3];

        /* 2. création du contenu */
        tab[0] = new Point();
        tab[1] = new Point();
        tab[2] = new Point();
    }
}
```

25

Un tableau est en fait un objet comme un autre : il doit être créé, et se manipule à l'aide d'une référence. Mais il est impossible d'écrire des constructeurs pour sa classe, celle-ci étant déduite de la classe de ses éléments.

Comme n'importe quel autre type de référence, une fonction peut recevoir une référence vers un tableau, ou renvoyer une référence vers un tableau créé.

Exercices

1. A l'aide de

```
Deug.drawLine(int x0, int y0, int x1, int y1)
```

écrire

```
static void dessinerCourbe(Point[] t)
```

Cette fonction suppose que `t` désigne un tableau entièrement construit. Elle doit tracer chaque segment reliant deux points successifs de ce tableau.

2. A l'aide de `Deug.readInt()` et `Deug.readChar()`, écrire d'abord une fonction

27

Etape 1

- `Point[] tab = new Point[3]`
crée un tableau de références à trois éléments.

Le tableau est créé, mais il n'est pour le moment qu'une coquille vide : les références `tab[0]`, `tab[1]` et `tab[2]` ne désignent encore aucun point. Il reste encore trois points à créer pour compléter la construction.

Etape 2

Pour `i` valant 0, 1, 2 :

- `tab[i] = new Point();`
crée un nouveau point, et le désigne par `tab[i]`.

Le tableau est à présent entièrement construit : chaque `tab[i]` désigne un point. On peut maintenant accéder à `tab[1].abs`, `tab[2].nom`, etc.

26

```
static Point lirePoint()
```

Cette fonction devra créer un nouveau point, lire au clavier son abscisse, son ordonnée et son nom, et initialiser ses champs aux valeurs lues. Elle renverra une référence vers le point créé.

Ecrire ensuite

```
static Point[] lireCourbe(int n)
```

Cette fonction devra :

- construire un nouveau tableau de références vers des points, de longueur `n`, désigné par `tab`,
- pour chaque `i`, appeler `lirePoint` et désigner par `tab[i]` le point créé par `lirePoint`.
- renvoyer `tab`.