

Introduction à l'Informatique (INF 311)



F. Morain



Amphi 5: classes

22 mai

- I. Classes et objets.
- II. Organisation mémoire.
- III. Méthodes.
- IV. Compléments.

0. Où en sommes-nous?

- Amphi 1: introduction.
- Amphi 2: programmer en Java.
- Amphi 3: fonctions/fonctions récursives.
- Amphi 4: tableaux/String.
- Amphi 5: classes.
- Amphi 6: tables.
- Amphi 7: algorithmes et complexité.
- Amphi 8: Internet.
- Amphi 9: listes.
- Amphi 10: système/sécurité.

Plongée dans la mémoire

```
public class Prgm{
```


```
... int f(int x){  
    int m;  
    m = x+1;  
    return m;  
}
```

@10	args	@0
@20		
@30		

```
... void main(...){  
    int n, r;  
    n = 1;  
    r = f(n);  
    System.out.println(r);  
}
```

←

```
}
```

Plongée dans la mémoire

```
public class Prgm{
```


```
... int f(int x){  
    int m;  
    m = x+1;  
    return m;  
}
```

@10	args	@0
@20	n	0
@30	r	0

```
... void main(...){  
    int n, r;  
    n = 1;  
    r = f(n);  
    System.out.println(r);  
}
```



```
}
```

Plongée dans la mémoire

```
public class Prgm{
```


```
... int f(int x){  
    int m;  
    m = x+1;  
    return m;  
}
```

@10	args	@0
@20	n	1
@30	r	0

```
... void main(...){  
    int n, r;  
    n = 1;  
    r = f(n);  
    System.out.println(r);  
}
```

```
}
```



Plongée dans la mémoire

```
public class Prgm{
```


```
... int f(int x){  
    int m;  
    m = x+1;  
    return m;  
}
```

@10	args	@0
@20	n	1
@30	r	0

```
... void main(...){  
    int n, r;  
    n = 1;  
    r = f(n);  
    System.out.println(r);  
}
```

```
}
```

⇐

Plongée dans la mémoire

```
public class Prgm{
```

@500	x	1
@510		

```
... int f(int x){  
    int m;  
    m = x+1;  
    return m;  
}
```

←

@10	args	@0
@20	n	1
@30	r	0

```
... void main(...){  
    int n, r;  
    n = 1;  
    r = f(n);  
    System.out.println(r);  
}
```

```
}
```

Plongée dans la mémoire

```
public class Prgm{
```

@500	x	1
@510	m	0

```
... int f(int x){  
    int m;  
    m = x+1;  
    return m;  
}
```



@10	args	@0
@20	n	1
@30	r	0

```
... void main(...){  
    int n, r;  
    n = 1;  
    r = f(n);  
    System.out.println(r);  
}  
}
```

Plongée dans la mémoire

```
public class Prgm{
```

@500	x	1
@510	m	2

```
... int f(int x){  
    int m;  
    m = x+1;  
    return m;  
}
```



@10	args	@0
@20	n	1
@30	r	0

```
... void main(...){  
    int n, r;  
    n = 1;  
    r = f(n);  
    System.out.println(r);  
}  
}
```

Plongée dans la mémoire

```
public class Prgm{
```

@500	x	1
@510	m	2

```
... int f(int x){  
    int m;  
    m = x+1;  
    return m;  
}
```



@10	args	@0
@20	n	1
@30	r	0

```
... void main(...){  
    int n, r;  
    n = 1;  
    r = f(n);  
    System.out.println(r);  
}  
}
```

Plongée dans la mémoire

```
public class Prgm{
```


```
... int f(int x){  
    int m;  
    m = x+1;  
    return m;  
}
```

@10	args	@0
@20	n	1
@30	r	2

```
... void main(...){  
    int n, r;  
    n = 1;  
    r = f(n);  
    System.out.println(r); ←  
}
```

```
}
```

Plongée dans la mémoire

```
public class Prgm{
```


```
... int f(int x){  
    int m;  
    m = x+1;  
    return m;  
}
```

@10	args	@0
@20	n	1
@30	r	2

```
... void main(...){  
    int n, r;  
    n = 1;  
    r = f(n);  
    System.out.println(r);  
}
```

```
}
```

Rem. Les variables locales de `f` ne sont pas accessibles par `main` et réciproquement.

Un pas de plus vers la VÉRITÉ: deux types de mémoire, **locale** (pour chaque fonction), et **globale** (pour le programme).

--	--	--

```
... void f(int[] w){  
    w[0] = -10;  
}
```

@10	args	@0
@20		

```
... void main(... args){  
    int[] t = {1, 2};  
    f(t);  
    System.out.println(t[0]);  
}
```

mémoire globale partagée par le programme

@990	@1000	@1004	@1008	@1012
...				...
...				...

Un pas de plus vers la VÉRITÉ: deux types de mémoire, **locale** (pour chaque fonction), et **globale** (pour le programme).

--	--	--

```
... void f(int[] w){  
    w[0] = -10;  
}
```

@10	args	@0
@20		

```
... void main(... args){  
    int[] t = {1, 2};  
    f(t);  
    System.out.println(t[0]);  
}
```

←

mémoire globale partagée par le programme

@990	@1000	@1004	@1008	@1012
...				...
...				...

Un pas de plus vers la VÉRITÉ: deux types de mémoire, **locale** (pour chaque fonction), et **globale** (pour le programme).

--	--	--

```
... void f(int[] w){
    w[0] = -10;
}
```

@10	args	@0
@20	t	@1000

```
... void main(... args){
    int[] t = {1, 2};
    f(t);
    System.out.println(t[0]);
}
```



mémoire globale partagée par le programme

@990	@1000	@1004	@1008	@1012
...	.length	[0]	[1]	...
...	2	1	2	...

Un pas de plus vers la VÉRITÉ: deux types de mémoire, **locale** (pour chaque fonction), et **globale** (pour le programme).

--	--	--

```
... void f(int[] w){  
    w[0] = -10;  
}
```

@10	args	@0
@20	t	@1000

```
... void main(... args){  
    int[] t = {1, 2};  
    f(t);  
    System.out.println(t[0]);  
}
```



mémoire globale partagée par le programme

@990	@1000	@1004	@1008	@1012
...	.length	[0]	[1]	...
...	2	1	2	...

Un pas de plus vers la VÉRITÉ: deux types de mémoire, **locale** (pour chaque fonction), et **globale** (pour le programme).

@500	w	@1000
------	---	-------

```
... void f(int[] w){  
    w[0] = -10;  
}
```

@10	args	@0
@20	t	@1000

```
... void main(... args){  
    int[] t = {1, 2};  
    f(t);  
    System.out.println(t[0]);  
}
```

mémoire globale partagée par le programme

@990	@1000	@1004	@1008	@1012
...	.length	[0]	[1]	...
...	2	1	2	...

Un pas de plus vers la VÉRITÉ: deux types de mémoire, **locale** (pour chaque fonction), et **globale** (pour le programme).

@500	w	@1000
------	---	-------

```
... void f(int[] w){  
    w[0] = -10;  
}
```

@10	args	@0
@20	t	@1000

```
... void main(... args){  
    int[] t = {1, 2};  
    f(t);  
    System.out.println(t[0]);  
}
```

mémoire globale partagée par le programme

@990	@1000	@1004	@1008	@1012
...	.length	[0]	[1]	...
...	2	-10	2	...

Un pas de plus vers la VÉRITÉ: deux types de mémoire, **locale** (pour chaque fonction), et **globale** (pour le programme).

--	--	--

```
... void f(int[] w){  
    w[0] = -10;  
}
```

@10	args	@0
@20	t	@1000

```
... void main(... args){  
    int[] t = {1, 2};  
    f(t);  
    System.out.println(t[0]); ←  
}
```

mémoire globale partagée par le programme

@990	@1000	@1004	@1008	@1012
...	.length	[0]	[1]	...
...	2	-10	2	...

I. Classes et objets

Plusieurs espèces d'acteurs en **Java**:

- **types primitifs**: `int`, `double`, etc.;
- **tableaux**: collections de valeurs de même type;
- Les **classes** ont un rôle double:
 - ▶ **types** plus complexes (assemblage de types différents);
 - ▶ **rassembler** en un même endroit toutes les fonctions (méthodes) qui s'y rapportent (**bibliothèques**); prédéfinies (comme `string`) ou construites par l'utilisateur.

```
public class Client{
    int badge;
    double solde;
    public static void afficher(Client c){
        System.out.print(c.badge+" "+c.solde);
    }
}
public class TestClient{
    public static void main(String[] args){
        Client fm = new Client();
        fm.badge = 11;
        fm.solde = 10.0;
        Client.afficher(fm);
    }
}
```

`Client.afficher()` nom complet à utiliser de l'extérieur de sa classe de définition.

Rem. on compile `Client.java` et `TestClient.java`, puis on lance `java TestClient` (la classe contenant `main`).

```
public class Client{
    int badge;
    double solde;
    public static void afficher(Client c){
        System.out.print(c.badge+" "+c.solde);
    }
}
public class TestClient{
    public static void main(String[] args){
        Client fm = new Client();
        fm.badge = 11;
        fm.solde = 10.0;
        Client.afficher(fm);
    }
}
```

`Client.afficher()` nom complet à utiliser de l'extérieur de sa classe de définition.

Rem. on compile `Client.java` et `TestClient.java`, puis on lance `java TestClient` (la classe contenant `main`).

```
public class Client{
    int badge;
    double solde;
    public static void afficher(Client c){
        System.out.print(c.badge+" "+c.solde);
    }
}
public class TestClient{
    public static void main(String[] args){
        Client fm = new Client();
        fm.badge = 11;
        fm.solde = 10.0;
        Client.afficher(fm);
    }
}
```

`Client.afficher()` nom complet à utiliser de l'extérieur de sa classe de définition.

Rem. on compile `Client.java` et `TestClient.java`, puis on lance `java TestClient` (la classe contenant **main**).

Propriétés

`badge`, `solde` sont des **champs**; ils se manipulent comme des variables du même type.

`fm` est une **instance** de la classe `Client`, appelée **objet**.

Tout comme `3` est une instance du type `int` ou `int[] t = new int[3];` une instance du type tableau d'`int`.

Déf. `Client()` est le **constructeur** (implicite) de la classe `Client`.

Comme pour un tableau, on doit créer l'espace mémoire nécessaire au stockage de l'objet.

Propriétés

`badge`, `solde` sont des **champs**; ils se manipulent comme des variables du même type.

`fm` est une **instance** de la classe `Client`, appelée **objet**.

Tout comme `3` est une instance du type `int` ou `int[] t = new int[3];` une instance du type tableau d'`int`.

Déf. `Client()` est le **constructeur** (implicite) de la classe `Client`.

Comme pour un tableau, on doit créer l'espace mémoire nécessaire au stockage de l'objet.

Constructeur implicite et explicite

`Client()` est le **constructeur implicite** de la classe `Client`.

On peut écrire un **constructeur explicite**:

```
public Client(int b, double s){ // pas de static
    this.badge = b;
    this.solde = s;
    // pas de return
}
```

`this` fait référence à l'objet qui vient d'être créé et sur lequel on opère.

On utilise le constructeur ainsi:

```
fm = new Client(11, 10.0);
```

Rem. On n'est pas obligé d'en utiliser.

Rem. Si on utilise un constructeur explicite, le constructeur implicite n'est plus accessible.

Constructeur implicite et explicite

`Client()` est le **constructeur implicite** de la classe `Client`.

On peut écrire un **constructeur explicite**:

```
public Client(int b, double s){ // pas de static
    this.badge = b;
    this.solde = s;
    // pas de return
}
```

`this` fait référence à l'objet qui vient d'être créé et sur lequel on opère.

On utilise le constructeur ainsi:

```
fm = new Client(11, 10.0);
```

Rem. On n'est pas obligé d'en utiliser.

Rem. Si on utilise un constructeur explicite, le constructeur implicite n'est plus accessible.

Constructeur implicite et explicite

`Client()` est le **constructeur implicite** de la classe `Client`.

On peut écrire un **constructeur explicite**:

```
public Client(int b, double s){ // pas de static
    this.badge = b;
    this.solde = s;
    // pas de return
}
```

`this` fait référence à l'objet qui vient d'être créé et sur lequel on opère.

On utilise le constructeur ainsi:

```
fm = new Client(11, 10.0);
```

Rem. On n'est pas obligé d'en utiliser.

Rem. Si on utilise un constructeur explicite, le constructeur implicite n'est plus accessible.

II. Organisation mémoire

objet = référence

```
Client fm;           // bon de commande
                    // pour la commode
fm=new Client();    // on construit
                    // la commode
fm.badge = 11;      // on remplit les
fm.solde = 10.0;    // deux tiroirs
```

```
Client fm; // déclaration
```

@10
fm
@0 (null)

```
fm = new Client(); // initialisation
```

@10	@100	@104
fm	fm.badge	fm.solde
@100	0	0.0

```
fm.badge = 11;  
fm.solde = 10.0;
```

@10	@100	@104
fm	fm.badge	fm.solde
@100	11	10.0

```
Client fm; // déclaration
```

@10
fm
@0 (null)

```
fm = new Client(); // initialisation
```

@10	@100	@104
fm	fm.badge	fm.solde
@100	0	0.0

```
fm.badge = 11;  
fm.solde = 10.0;
```

@10	@100	@104
fm	fm.badge	fm.solde
@100	11	10.0

```
Client fm; // déclaration
```

@10
fm
@0 (null)

```
fm = new Client(); // initialisation
```

@10	@100	@104
fm	fm.badge	fm.solde
@100	0	0.0

```
fm.badge = 11;  
fm.solde = 10.0;
```

@10	@100	@104
fm	fm.badge	fm.solde
@100	11	10.0

Comment recopier des objets

```
public class Ref{
    int n;

    public Ref(int n0){
        this.n = n0;
    }
}

public class TestRef{
    public static void main(String[] args){
        Ref r = new Ref(1), s;

        // la mauvaise façon de copier r
        s = r;
        // la bonne façon
        s = new Ref(r.n);
    }
}
```

Comprendre $s = r$ et $s == r$

@10	@100
r	.n
@100	1

Faire $s = r$ fait que $s = @100$, donc c'est la même référence:

@10	@100	@500
r	.n	s
@100	1	@100

Idem pour $s == r$: on compare les références, pas les contenus mémoires.

Passage par valeur

Rappel: Java passe toujours les types par valeur (recopie).

```
public static void f(int n){
    n = -10;
}
public static void main(String[] args){
    int n = 1;

    f(n);
    System.out.println(n);
}
```

Le programme affiche 1.

Pour les objets (comme pour les tableaux): c'est la référence qui est passée. Elle est passée par valeur.

```
public class Ref{
    int n;

    public Ref(int n0){
        this.n = n0;
    }
}

public class TestRef{
    static void f(Ref w){
        w.n = -10;
    }

    public static void main(String[] args){
        Ref r;

        r = new Ref(1);
        f(r);
        System.out.println(r.n); // affiche -10
    }
}
```

En plongée!

--	--	--

```
... void f(Ref w){  
    w.n = -10;  
}
```

@10	args	@0
@20		

```
... void main(...){  
    Ref r;  
    r = new Ref(1);  
    f(r);  
    System.out.println(r.n);  
}
```



mémoire globale partagée par le programme

@990	@1000	@1004	@1008	@1012
...	
...	

En plongée!

--	--	--

```
... void f(Ref w){  
    w.n = -10;  
}
```

@10	args	@0
@20	r	@0

```
... void main(...){  
    Ref r;  
    r = new Ref(1);  
    f(r);  
    System.out.println(r.n);  
}
```



mémoire globale partagée par le programme

@990	@1000	@1004	@1008	@1012
...	
...	

En plongée!

--	--	--

```
... void f(Ref w){  
    w.n = -10;  
}
```

@10	args	@0
@20	r	@1000

```
... void main(...){  
    Ref r;  
    r = new Ref(1);  
    f(r);  
    System.out.println(r.n);  
}
```



mémoire globale partagée par le programme

@990	@1000	@1004	@1008	@1012
...	r.n
...	1

En plongée!

--	--	--

```
... void f(Ref w){  
    w.n = -10;  
}
```

@10	args	@0
@20	r	@1000

```
... void main(...){  
    Ref r;  
    r = new Ref(1);  
    f(r);  
    System.out.println(r.n);  
}
```



mémoire globale partagée par le programme

@990	@1000	@1004	@1008	@1012
...	r.n
...	1

En plongée!

@500	w	@1000
------	---	-------

```
... void f(Ref w){  
    w.n = -10;  
}
```



@10	args	@0
@20	r	@1000

```
... void main(...){  
    Ref r;  
    r = new Ref(1);  
    f(r);  
    System.out.println(r.n);  
}
```

mémoire globale partagée par le programme

@990	@1000	@1004	@1008	@1012
...	r.n
...	1

En plongée!

@500	w	@1000
------	---	-------

```
... void f(Ref w){  
    w.n = -10;  
}
```



@10	args	@0
@20	r	@1000

```
... void main(...){  
    Ref r;  
    r = new Ref(1);  
    f(r);  
    System.out.println(r.n);  
}
```

mémoire globale partagée par le programme

@990	@1000	@1004	@1008	@1012
...	r.n
...	-10

En plongée!

--	--	--

```
... void f(Ref w){  
    w.n = -10;  
}
```

@10	args	@0
@20	r	@1000

```
... void main(...){  
    Ref r;  
    r = new Ref(1);  
    f(r);  
    System.out.println(r.n); ←  
}
```

mémoire globale partagée par le programme

@990	@1000	@1004	@1008	@1012
...	r.n
...	-10

Encore un pour la route:

```
public class Client{
    int badge;
    double solde;

    public static void f(Client c){
        c = new Client(8, 123.0);
    }
}

public class TestClient{
    public static void main(String[] args){
        Client fm = new Client(11, 10.0);

        Client.f(fm);
        System.out.println(fm.solde); // affiche 10.0
    }
}
```

Encore un pour la route:

```
public class Client{
    int badge;
    double solde;

    public static void f(Client c){
        c = new Client(8, 123.0);
    }
}

public class TestClient{
    public static void main(String[] args){
        Client fm = new Client(11, 10.0);

        Client.f(fm);
        System.out.println(fm.solde); // affiche 10.0
    }
}
```

Retour sur la classe `String`

On a déjà manipulé les chaînes de caractères.

```
String s;
```

@10
s
@0 (null)

```
s = new String("Bonjour");
```

@10	@500
s	"Bonjour"
@500	

On peut aussi écrire:

```
String s = "bonjour";
```

on a fait appel implicitement au constructeur de la classe, qui transforme cette expression en objet de la classe `String`.

Retour sur la classe `String`

On a déjà manipulé les chaînes de caractères.

```
String s;
```

@10
s
@0 (null)

```
s = new String("Bonjour");
```

@10	@500
s	"Bonjour"
@500	

On peut aussi écrire:

```
String s = "bonjour";
```

on a fait appel implicitement au constructeur de la classe, qui transforme cette expression en objet de la classe `String`.

Retour sur la classe `String`

On a déjà manipulé les chaînes de caractères.

```
String s;
```

@10
s
@0 (null)

```
s = new String("Bonjour");
```

@10	@500
s	"Bonjour"
@500	

On peut aussi écrire:

```
String s = "bonjour";
```

on a fait appel implicitement au constructeur de la classe, qui transforme cette expression en objet de la classe `String`.

Des tableaux d'objets

Ex. On veut maintenant gérer un stock de produits.

```
public class Produit{
    String nom;
    int nb;
    double prix;
    public Produit(String N, int n, double p){
        this.nom = N; this.nb = n; this.prix = p;
    }
}

public class GestionStock{
    public static void main(String[] args){
        Produit[] s;

        s = new Produit[2]; // obligatoire!
        s[0] = new Produit("ordinateur", 5, 752.50);
            // obligatoire!
    }
}
```

Des tableaux d'objets

Ex. On veut maintenant gérer un stock de produits.

```
public class Produit{
    String nom;
    int nb;
    double prix;
    public Produit(String N, int n, double p){
        this.nom = N; this.nb = n; this.prix = p;
    }
}
public class GestionStock{
    public static void main(String[] args){
        Produit[] s;

        s = new Produit[2]; // obligatoire!
        s[0] = new Produit("ordinateur", 5, 752.50);
        // obligatoire!
    }
}
```

Des tableaux d'objets

Ex. On veut maintenant gérer un stock de produits.

```
public class Produit{
    String nom;
    int nb;
    double prix;
    public Produit(String N, int n, double p){
        this.nom = N; this.nb = n; this.prix = p;
    }
}
public class GestionStock{
    public static void main(String[] args){
        Produit[] s;

        s = new Produit[2]; // obligatoire!
        s[0] = new Produit("ordinateur", 5, 752.50);
            // obligatoire!
    }
}
```

```
s = new Produit[2];
```

@10	@100	@104	@108
s	s.length	s[0]	s[1]
@100	2	null	null

```
s[0] = new Produit("ordinateur", 5, 752.50);
```

@10	@100	@104	@108
s	s.length	s[0]	s[1]
@100	2	@500	null

@500	@520	@540
.nom	.nb	.prix
@700	5	752.50

@700
"ordinateur"

```
s = new Produit[2];
```

@10	@100	@104	@108
s	s.length	s[0]	s[1]
@100	2	null	null

```
s[0] = new Produit("ordinateur", 5, 752.50);
```

@10	@100	@104	@108
s	s.length	s[0]	s[1]
@100	2	@500	null

@500	@520	@540
.nom	.nb	.prix
@700	5	752.50

@700
"ordinateur"

Objets avec des champs tableaux

```
public class Polynome{
    int deg;
    double[] coeff;
    public Polynome(int d){
        this.deg = d;
        // obligatoire!
        this.coeff = new double[d + 1];
    }
}

public class TestPolynome{
    public static void main(String[] args){
        Polynome p = new Polynome(10);

        p.coeff[0] = 7.0;
        p.coeff[11] = -4.322;
    }
}
```

Objets avec des champs tableaux

```
public class Polynome{
    int deg;
    double[] coeff;
    public Polynome(int d){
        this.deg = d;
        // obligatoire!
        this.coeff = new double[d + 1];
    }
}

public class TestPolynome{
    public static void main(String[] args){
        Polynome p = new Polynome(10);

        p.coeff[0] = 7.0;
        p.coeff[11] = -4.322;
    }
}
```

III. Méthodes

```
public class Rationnel{
    long num, den;
    public Rationnel(long n, long d){
        this.num = n; this.den = d;
    }
    public static void afficher(Rationnel r){
        System.out.print(r.num + "/" + r.den);
    }
    public static Rationnel mult(Rationnel r1,
                                Rationnel r2){
        return new Rationnel(r1.num * r2.num,
                              r1.den * r2.den);
    }
}

public class TestRationnel{
    public static void main(String[] args){
        Rationnel r = new Rationnel(1, 2), r2;

        Rationnel.afficher(r);
        r2 = Rationnel.mult(r, r);
        Rationnel.afficher(r2);
        System.out.println();    }
}
```

III. Méthodes

```
public class Rationnel{
    long num, den;
    public Rationnel(long n, long d){
        this.num = n; this.den = d;
    }
    public static void afficher(Rationnel r){
        System.out.print(r.num + "/" + r.den);
    }
    public static Rationnel mult(Rationnel r1,
                                Rationnel r2){
        return new Rationnel(r1.num * r2.num,
                              r1.den * r2.den);
    }
}

public class TestRationnel{
    public static void main(String[] args){
        Rationnel r = new Rationnel(1, 2), r2;

        Rationnel.afficher(r);
        r2 = Rationnel.mult(r, r);
        Rationnel.afficher(r2);
        System.out.println();    }
}
```

III. Méthodes

```
public class Rationnel{
    long num, den;
    public Rationnel(long n, long d){
        this.num = n; this.den = d;
    }
    public static void afficher(Rationnel r){
        System.out.print(r.num + "/" + r.den);
    }
    public static Rationnel mult(Rationnel r1,
                                Rationnel r2){
        return new Rationnel(r1.num * r2.num,
                              r1.den * r2.den);
    }
}

public class TestRationnel{
    public static void main(String[] args){
        Rationnel r = new Rationnel(1, 2), r2;

        Rationnel.afficher(r);
        r2 = Rationnel.mult(r, r);
        Rationnel.afficher(r2);
        System.out.println();    }}}
```

III. Méthodes

```
public class Rationnel{
    long num, den;
    public Rationnel(long n, long d){
        this.num = n; this.den = d;
    }
    public static void afficher(Rationnel r){
        System.out.print(r.num + "/" + r.den);
    }
    public static Rationnel mult(Rationnel r1,
                                Rationnel r2){
        return new Rationnel(r1.num * r2.num,
                              r1.den * r2.den);
    }
}

public class TestRationnel{
    public static void main(String[] args){
        Rationnel r = new Rationnel(1, 2), r2;

        Rationnel.afficher(r);
        r2 = Rationnel.mult(r, r);
        Rationnel.afficher(r2);
        System.out.println();    }}}
```

Faisons le point

On vient de créer la classe `Rationnel` qui s'utilise comme la classe `Math` (cf. `Math.sqrt()`).

On a créé une nouvelle classe qui opère sur des objets nouveaux. La classe contient à la fois la définition d'un nouveau type (structure des objets), ainsi que les fonctions qui opèrent sur des instances de ce type (le comportement de ces objets).

Vocabulaire: ce qu'on a appelé fonctions jusqu'à présent sont appelées également **méthodes de classe**.

Les méthodes peuvent appeler d'autres méthodes de la même classe, ou bien d'autres classes.

→ Un grand pas vers la modularité.

Faisons le point

On vient de créer la classe `Rationnel` qui s'utilise comme la classe `Math` (cf. `Math.sqrt()`).

On a créé une nouvelle classe qui opère sur des objets nouveaux. La classe contient à la fois la définition d'un nouveau type (structure des objets), ainsi que les fonctions qui opèrent sur des instances de ce type (le comportement de ces objets).

Vocabulaire: ce qu'on a appelé fonctions jusqu'à présent sont appelées également **méthodes de classe**.

Les méthodes peuvent appeler d'autres méthodes de la même classe, ou bien d'autres classes.

→ Un grand pas vers la modularité.

Méthodes d'objets (ou méthodes d'instances)

```
// méthode de classe
public static Rationnel inverser(Rationnel r){
    return new Rationnel(r.den, r.num);
}

// méthode d'objet
public Rationnel inverse(){
    return new Rationnel(this.den, this.num);
}
```

`this` fait référence à l'objet sur lequel la méthode d'objet a été appelée.

Méthodes d'objets (ou méthodes d'instances)

```
// méthode de classe
public static Rationnel inverser(Rationnel r){
    return new Rationnel(r.den, r.num);
}

// méthode d'objet
public Rationnel inverse(){
    return new Rationnel(this.den, this.num);
}
```

`this` fait référence à l'objet sur lequel la méthode d'objet a été appelée.

```
public class TestRationnel{
    public static void main(String[] args){
        Rationnel r, r2, r3;

        r = new Rationnel(1, 2);
        r2 = Rationnel.inverser(r);
        r3 = r.inverse();
    }
}
```

Autre exemple:

```
public Rationnel multo(Rationnel r){
    return new Rationnel(this.num*r.num,
                        this.den*r.den);
}
```

appel (asymétrique): `s.multo(t)` ou `t.multo(s)`.

```
public class TestRationnel{
    public static void main(String[] args){
        Rationnel r, r2, r3;

        r = new Rationnel(1, 2);
        r2 = Rationnel.inverser(r);
        r3 = r.inverse();
    }
}
```

Autre exemple:

```
public Rationnel multo(Rationnel r){
    return new Rationnel(this.num*r.num,
                        this.den*r.den);
}
```

appel (asymétrique): `s.multo(t)` ou `t.multo(s)`.

Encore des mystères résolus!

Les méthodes d'objets sont aussi appelées **dynamiques** (cf. INF 431), par opposition aux méthodes de classe, dites **statiques** (d'où le mot-clef **static**).

On comprend mieux:

```
String s = "Bonjour!";
```

```
int l = s.length();
```

```
char c = s.charAt(2);
```

Méthode de classe ou méthode d'objet

- Syntaxe et utilisation surprenantes au début;
- méthodes d'objets plus “puissantes” que les méthodes de classes (cf. INF-421, INF-431: héritage, etc.);
- deux approches **complémentaires**;
- c'est aussi une **question de goût**.

Dans la suite du cours, nous serons consommateurs de méthodes d'objets, mais pas producteurs...!; cf. INF 421, 431.

IV. Compléments

Variable de classe: variable partagée (accessible) par toutes les méthodes de la classe.

```
public class Alea{
    static int valeur;
}
```

Cette variable existe en un exemplaire unique pour toute la classe (qu'il y ait des objets ou non).

Elle est statique, car connue à la compilation (contrairement aux champs d'un objet).

On peut aussi gérer des constantes partagées à l'aide du mot-clef `final`:

```
final static int M = 100, a = 73, b = 1;
```

avec initialisation obligatoire à la déclaration.

IV. Compléments

Variable de classe: variable partagée (accessible) par toutes les méthodes de la classe.

```
public class Alea{
    static int valeur;
}
```

Cette variable existe en un exemplaire unique pour toute la classe (qu'il y ait des objets ou non).

Elle est statique, car connue à la compilation (contrairement aux champs d'un objet).

On peut aussi gérer des **constantes** partagées à l'aide du mot-clef **final**:

```
final static int M = 100, a = 73, b = 1;
```

avec initialisation obligatoire à la déclaration.

Exemple

```
public class Alea{
    final static int M = 100, a = 73, b = 1;
    static int valeur = 0;

    public static int suivante(){
        valeur = (a * valeur + b) % M;
        // on a modifié la variable de classe!
        return valeur;
    }
}

public class TestAlea{
    public static void main(String[] args){
        Alea.valeur = 1;
        System.out.println(Alea.suivante());
        System.out.println(Alea.suivante());
        System.out.println(Alea.valeur);
    }
}}
```

74

3

3

Exemple

```
public class Alea{
    final static int M = 100, a = 73, b = 1;
    static int valeur = 0;

    public static int suivante(){
        valeur = (a * valeur + b) % M;
        // on a modifié la variable de classe!
        return valeur;
    }
}

public class TestAlea{
    public static void main(String[] args){
        Alea.valeur = 1;
        System.out.println(Alea.suivante());
        System.out.println(Alea.suivante());
        System.out.println(Alea.valeur);
    }}
}
```

74

3

3

Exemple

```
public class Alea{
    final static int M = 100, a = 73, b = 1;
    static int valeur = 0;

    public static int suivante(){
        valeur = (a * valeur + b) % M;
        // on a modifié la variable de classe!
        return valeur;
    }
}

public class TestAlea{
    public static void main(String[] args){
        Alea.valeur = 1;
        System.out.println(Alea.suivante());
        System.out.println(Alea.suivante());
        System.out.println(Alea.valeur);
    }}
}
```

74

3

3

Et hop, un mystère de moins

Quand on écrit `System.out.println`:

- on se réfère à la classe `System` qui contient une variable de classe `out` (de type `PrintStream`);
- l'objet correspondant à `System.out` possède une méthode appelée `println`.

Problèmes fréquents

```
Client dg = new Client();  
dg = new Client();  
dg.badge = 1;  
dg.solde = 0.0;
```

est inutile: vous avez réservé deux fois une table dans le même restaurant!

Dans d'autres langages que Java, c'est une **perte irrécupérable**. En Java, on fait travailler automatiquement le **garbage collector** (glaneur de cellules, ramasse-miettes).

Autre problème(?):

```
System.out.println(dg);
```

```
Client@5d0385c1
```

Problèmes fréquents

```
Client dg = new Client();  
dg = new Client();  
dg.badge = 1;  
dg.solde = 0.0;
```

est inutile: vous avez réservé deux fois une table dans le même restaurant!

Dans d'autres langages que Java, c'est une **perte irrécupérable**. En Java, on fait travailler automatiquement le **garbage collector** (glaneur de cellules, ramasse-miettes).

Autre problème(?):

```
System.out.println(dg);
```

```
Client@5d0385c1
```

Problèmes fréquents

```
Client dg = new Client();  
dg = new Client();  
dg.badge = 1;  
dg.solde = 0.0;
```

est inutile: vous avez réservé deux fois une table dans le même restaurant!

Dans d'autres langages que Java, c'est une **perte irrécupérable**. En Java, on fait travailler automatiquement le **garbage collector** (glaneur de cellules, ramasse-miettes).

Autre problème(?):

```
System.out.println(dg);
```

```
Client@5d0385c1
```

Tableau vs. objet

Point commun: référence à un espace mémoire à construire par `new`; \Rightarrow même règle pour les paramètres des fonctions, même phénomène à la copie, etc.

Différence: Les tableaux sont des objets très particuliers, avec une syntaxe spéciale (`τ []`). Pas vraiment une classe `Tableau`, pas de méthodes.

Résumé

```
public class Wagon{
    final static int WMAX = 100; // constante
    static int nw; // variable de classe
    String nom; // champ d'un objet
    // constructeur explicite
    public Wagon(String n){
        nw++;
        this.nom = n;
    }
    // une fonction (méthode de classe)
    public static void print(Wagon w){
        System.out.println(w.nom);
    }
}

public class TestWagon{
    public static void main(String[] args){
        Wagon w = new Wagon("Thalis");

        Wagon.print(w);
    }
}
```

Résumé

```
public class Wagon{
    final static int WMAX = 100; // constante
    static int nw; // variable de classe
    String nom; // champ d'un objet
    // constructeur explicite
    public Wagon(String n){
        nw++;
        this.nom = n;
    }
    // une fonction (méthode de classe)
    public static void print(Wagon w){
        System.out.println(w.nom);
    }
}

public class TestWagon{
    public static void main(String[] args){
        Wagon w = new Wagon("Thalis");

        Wagon.print(w);
    }
}
```

Résumé

```
public class Wagon{
    final static int WMAX = 100; // constante
    static int nw; // variable de classe
    String nom; // champ d'un objet
    // constructeur explicite
    public Wagon(String n){
        nw++;
        this.nom = n;
    }
    // une fonction (méthode de classe)
    public static void print(Wagon w){
        System.out.println(w.nom);
    }
}

public class TestWagon{
    public static void main(String[] args){
        Wagon w = new Wagon("Thalis");

        Wagon.print(w);
    }
}
```

Résumé

```
public class Wagon{
    final static int WMAX = 100; // constante
    static int nw; // variable de classe
    String nom; // champ d'un objet
    // constructeur explicite
    public Wagon(String n){
        nw++;
        this.nom = n;
    }
    // une fonction (méthode de classe)
    public static void print(Wagon w){
        System.out.println(w.nom);
    }
}

public class TestWagon{
    public static void main(String[] args){
        Wagon w = new Wagon("Thalis");

        Wagon.print(w);
    }
}
```

Un plus gros exemple: la classe Promotion

```
public class Eleve{
    String nom, prenom;
    public Eleve(String n, String p){
        this.nom = n; this.prenom = p;
    }
}

public class Compagnie{
    static final int NB_ELEVES = 100;

    String cc;
    int num;
    Eleve[] te;

    public Compagnie(String c, int n){
        this.cc = c;
        this.num = n;
        this.te = new Eleve[NB_ELEVES];
    }
}
```

Un plus gros exemple: la classe Promotion

```
public class Eleve{
    String nom, prenom;
    public Eleve(String n, String p){
        this.nom = n; this.prenom = p;
    }
}

public class Compagnie{
    static final int NB_ELEVES = 100;

    String cc;
    int num;
    Eleve[] te;

    public Compagnie(String c, int n){
        this.cc = c;
        this.num = n;
        this.te = new Eleve[NB_ELEVES];
    }
}
```

```

public class Promotion{

    static final int NB_COMPAGNIES = 5;

    String cdt;
    Compagnie[] tc;

    public Promotion(String c){
        this.cdt = c;
        this.tc = new Compagnie[NB_COMPAGNIES];
    }
}

public class TestPromotion{
    public static void main(String[] args){
        Promotion X2005 = new Promotion("LCL Lorida");
        X2005.tc[0] = new Compagnie("CNE Hoarau", 6);
        X2005.tc[1] = new Compagnie("LT Sorin", 7);
        X2005.tc[2] = new Compagnie("MAJ Salvi", 8);
        X2005.tc[3] = new Compagnie("CNE De Kermenguy",
        X2005.tc[4] = new Compagnie("CNE Escuret", 10);
        X2005.tc[0].te[0] = new Eleve("Bond", "James");
        // etc.
    }
}

```

Derniers mots

On a vu tout ce qu'on devait savoir sur les classes (pour un début)!

Prochains rendez-vous:

Groupes	TD 5
1-6	13h30-15h30
7-12	15h45-17h45

Prochain amphi: **mardi 6 juin à 8h30 en amphi Poincaré; TD le mercredi.**

Tutorat: 1er juin.

Déjeuner avec les délégué(e)s

