HAUTE ÉCOLE DE NAMUR-LIÈGE-LUXEMBOURG

Chapitre 1

Introduction aux principes de la programmation orientée objet

Plan

- Les langages de programmation
- Approche procédurale
- Approche orientée objet
- Comparaison : procédural >< OO
- Présentation du cours

Langage de programmation

*On appelle **langage informatique** un langage formel […] utilisé lors de la conception, la mise en œuvre ou l'exploitation d'un système d'information. Le terme est toutefois utilisé dans certains contextes dans le sens plus restrictif de langage de programmation.*

(Source : Wikipedia)

Langage de programmation

Les langages de programmation peuvent se répartir en plusieurs catégories correspondant à divers **paradigmes** de programmation.

Les principaux paradigmes se distinguent par les éléments dont les programmes sont constitués.

```
…
x += 7;
printf("Ici, x = %d", x);
y = x*x + 2*x – 1;
scanf("%d", &z);
y *= z;
…
```

En programmation **impérative**, les programmes sont composés d'instructions (= ordres, d'où le terme "impératif") qui s'enchaînent les unes après les autres.
"Fais ceci, puis ceci, puis cela…"

Langage de programmation

Et l'orienté objet dans tout ça ?

Correspond à une autre organisation du code.

Programmation procédurale :

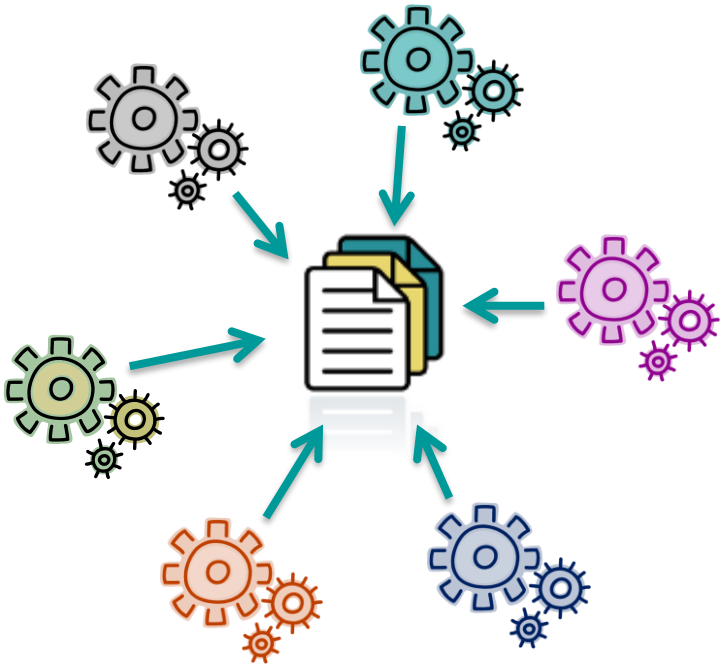
- *on organise le code en fonction des **actions** à accomplir*

Programmation OO :

- *on organise le code en fonction des (types de) **données***

Approche procédurale

En programmation procédurale…

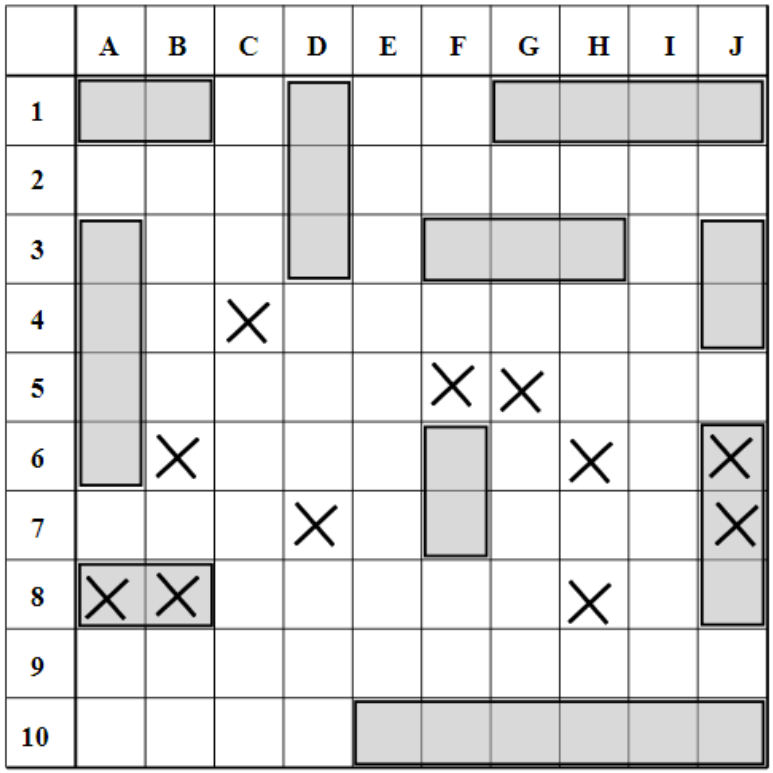


- On définit des données "centrales" utilisées par tout le programme
- On définit des procédures/fonctions qui agissent sur ces données
- Chaque procédure/fonction accomplit une tâche particulière :
  le code est découpé selon les actions/les fonctions désirées (**découpe fonctionnelle**)

Approche procédurale

Un exemple, le touché-coulé :

- deux joueurs
- chacun ayant 10 navires
- placés sur une grille de 10 × 10
- dont certaines cases peuvent avoir été détruites par un missile



Approche procédurale

On définit des variables pour les données :

- fiche/structure pour chaque navire
- tableaux `flottille1`, `flottille2` reprenant les navires de chaque joueur

On définit des fonctions qui s'occupent des tâches :

- `afficheGrille` : affiche l'état actuel d'une flottille
- `attaque` : pour gérer une attaque sur une case
- `estVictoire` : vérifie si tous les navires ont été coulés
- …

Approche procédurale

1) Données globales

Types

```
#define TAILLE 6
#define NB_NAVIRES 10

typedef enum direction {
  HAUT, DROITE, BAS, GAUCHE
} Direction;

typedef struct navire {
  int coordonnéeXTête, coordonnéeYTête;
  Direction direction;
  int taille;
  bool casesDétruites [TAILLE];
} Navire;
```

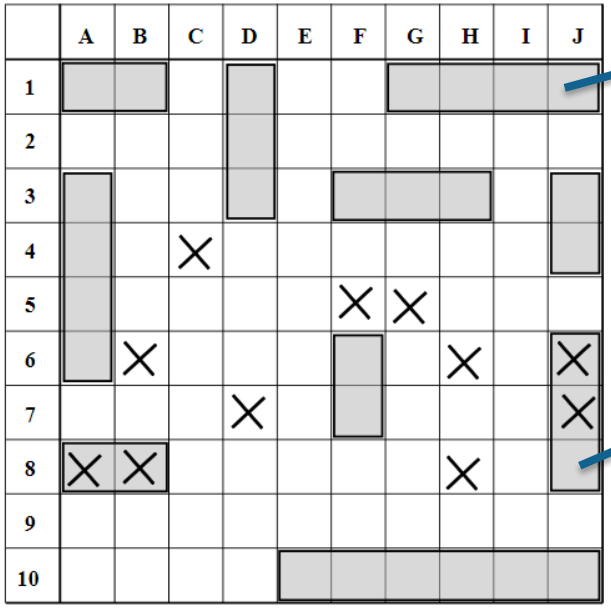
Données

```
Navire flottille1 [NB_NAVIRES];
Navire flottille2 [NB_NAVIRES];

char nomJoueur1 [30];
char nomJoueur2 [30];
```

Approche procédurale

1) Données globales



Exemple

```
Navire navire1 = {
  7,                  // coordonnéeXTête
  1,                  // coordonnéeYTête
  DROITE,             // direction
  4,                  // taille
  {false, … }         // casesDétruites
};

Navire navire2 = {
  10,                 // coordonnéeXTête
  6,                  // coordonnéeYTête
  BAS,                // direction
  3,                  // taille
  {true, true, false, … }
};

Navire flottille1 [] = { navire1,
          navire2, …, navire5, … };
```

Approche procédurale

1) Données globales

2) Modules

```
Navire flotille1 [NB_NAVIRES];
Navire flottile2 [NB_NAVIRES];

char nomJoueur1 [30];
char nomJoueur2 [30];
```

Exemple :

attaque

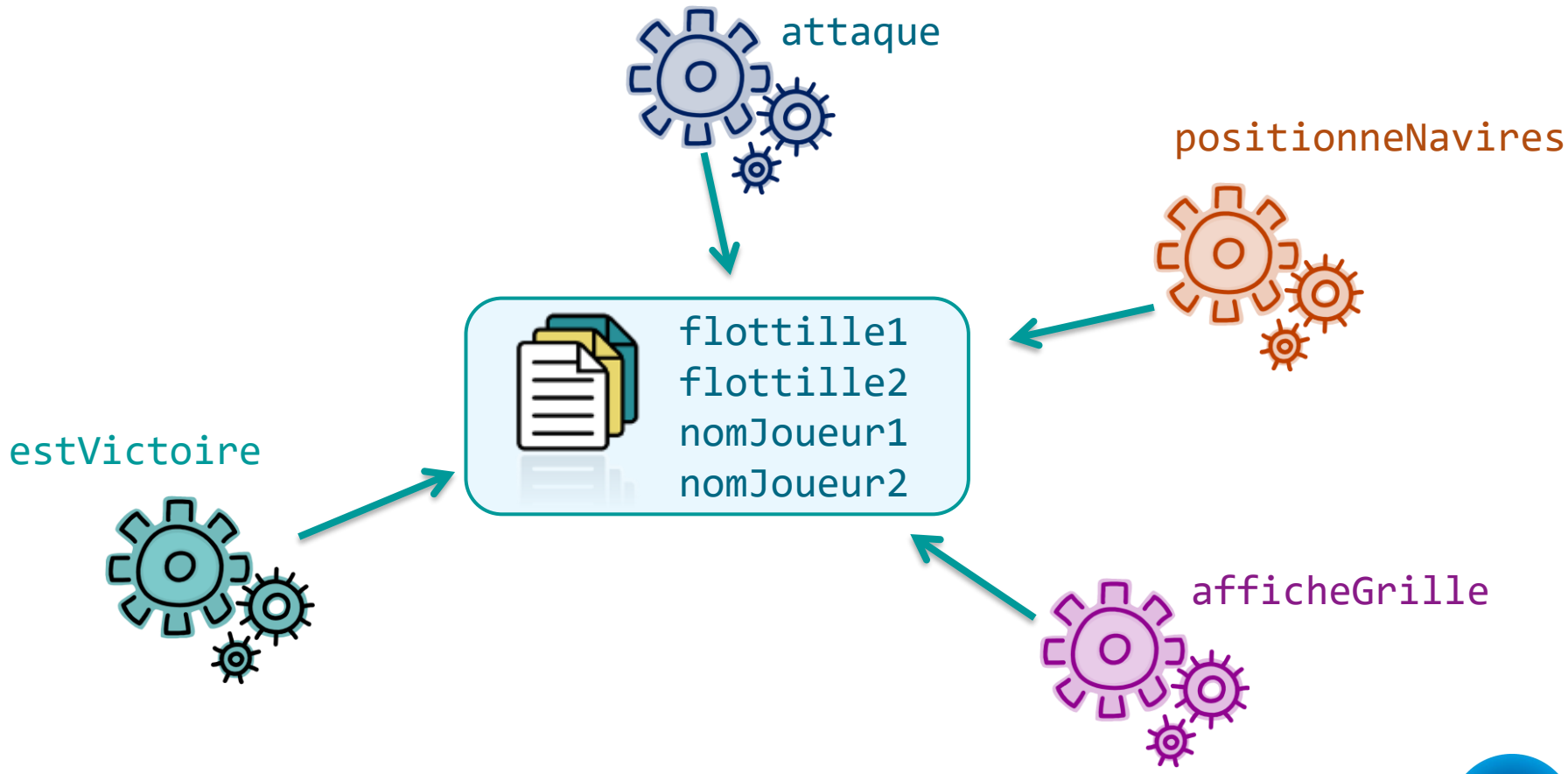
↓ flottilles, cible

↓ touché, coulé, flottilles

```
touché ← false
coulé ← false
Examiner les navires un à un à la recherche de la case cible
if on trouve iNavire tel que flottille[iNavire] touché
| marquer la case cible comme détruite dans le navire
| touché ← true
| Vérifier si toutes les cases du navire sont détruites
| if c'est le cas
| | coulé ← true
| endif
endif
```

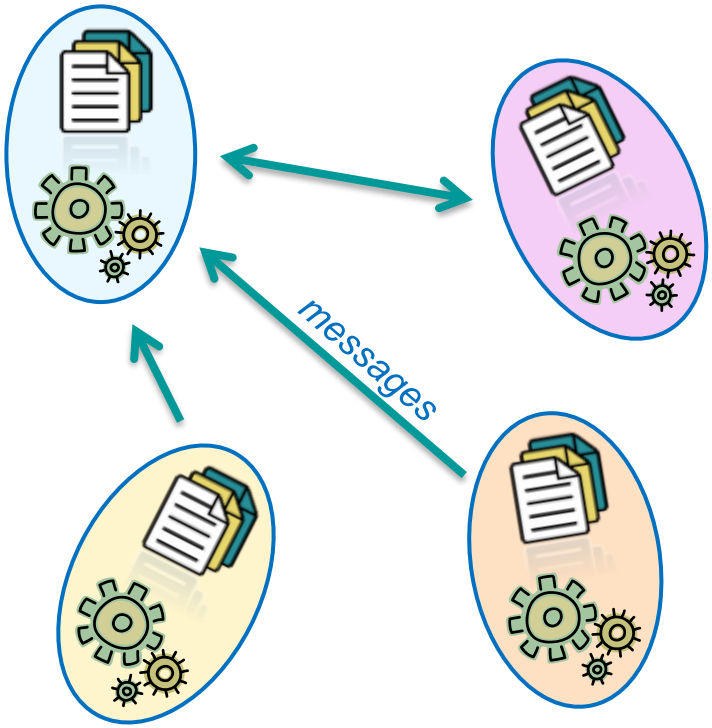
Approche procédurale

Les modules travaillent directement sur les données



Approche orientée objet

En programmation OO…



- On définit des "capsules" qui renferment une partie des données **et** le code qui travaille sur ces données
- Un objet contient donc à la fois des données et du code
- Ces "capsules" (= objets) peuvent s'envoyer des messages pour demander une action ou de l'information
- Le programme est découpé selon les objets / acteurs / types de données

Approche orientée objet

On identifie les (types d') "acteurs"

- navire, flottille, joueur…

On leur associe des tâches (les acteurs peuvent faire appel les uns aux autres pour les réaliser)

- flottille :
  - indiquer quel navire (s'il existe) est touché par une attaque
  - indiquer si tous les navires ont été coulés
- navire :
  - gérer une attaque sur une de ses cases
  - indiquer s'il est coulé

Approche orientée objet

1) Types d'objets



- Navire

  = position X et Y de la tête, direction, taille, tableau de cases détruites

- Flottille

  = tableau d'éléments de type Navire

- Joueur

  = nom, un élément de type Flottille

En Java :

```
class Navire {
  int coordonneeXTete;
  int coordonneeYTete;
  char direction;
  int taille;
  boolean casesDetruites [];
  …
}
```

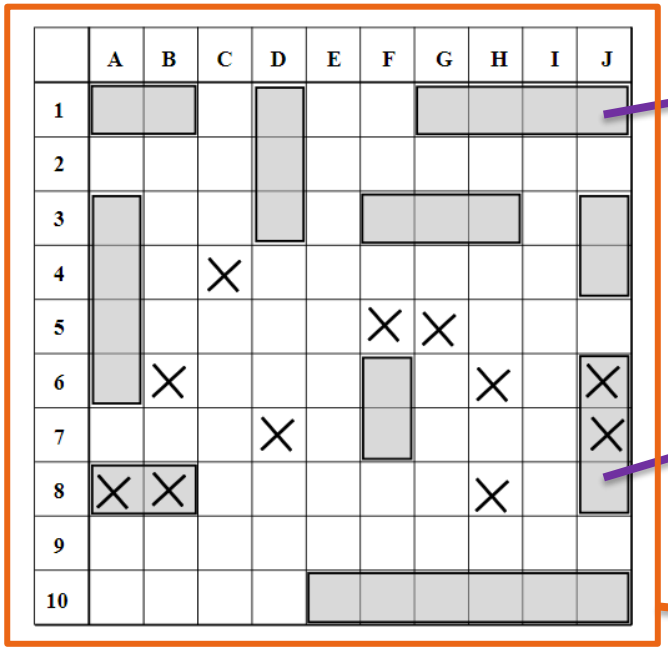
Approche orientée objet



2) Actions associées

- Navire
  - afficher le navire sur la grille
  - indiquer si une position donnée correspond à une case du navire
  - marquer une position comme détruite / gérer une attaque
  - indiquer si le navire a été coulé / entièrement détruit
- Flottille
  - afficher la grille et la position des navires
  - résoudre une attaque (dans l'eau, touché ou touché-coulé)
  - indiquer si une flottille a été entièrement détruite
- Joueur
  - effectuer un tour de jeu (demander la cible et gérer l'attaque)

Approche orientée objet



navire3

```
coordonneeXTete = 7
coordonneeYTete = 1
taille = 4
direction = 'D'
casesDetruites = …
```

```
estDetruit()
contientCase(x,y)
marqueDetruite(x,y)
affiche()
```

navire8

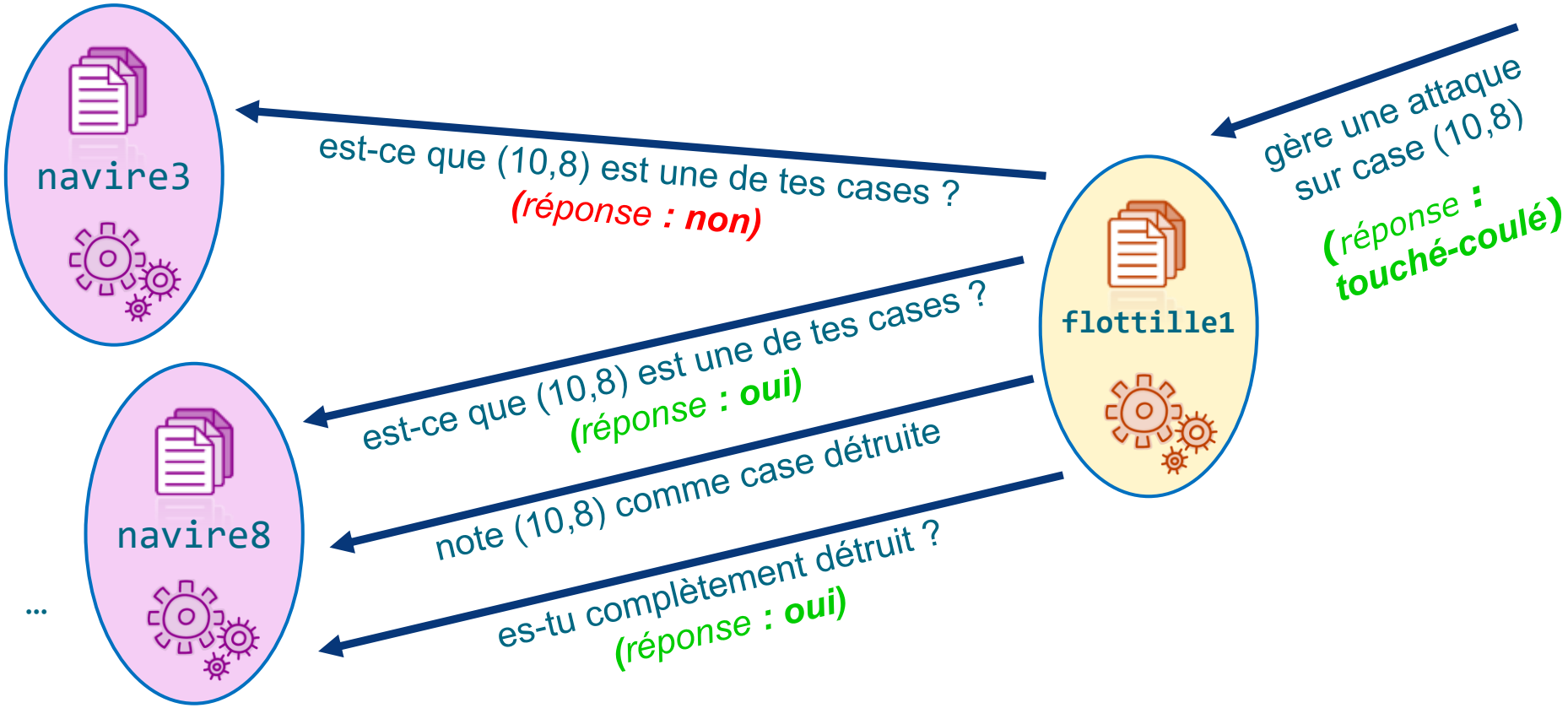
flottille1

```
navires = {…,…,…,…}
```

```
affiche()
resoudreAttaque(x,y)
estDetruite()
```

Approche orientée objet

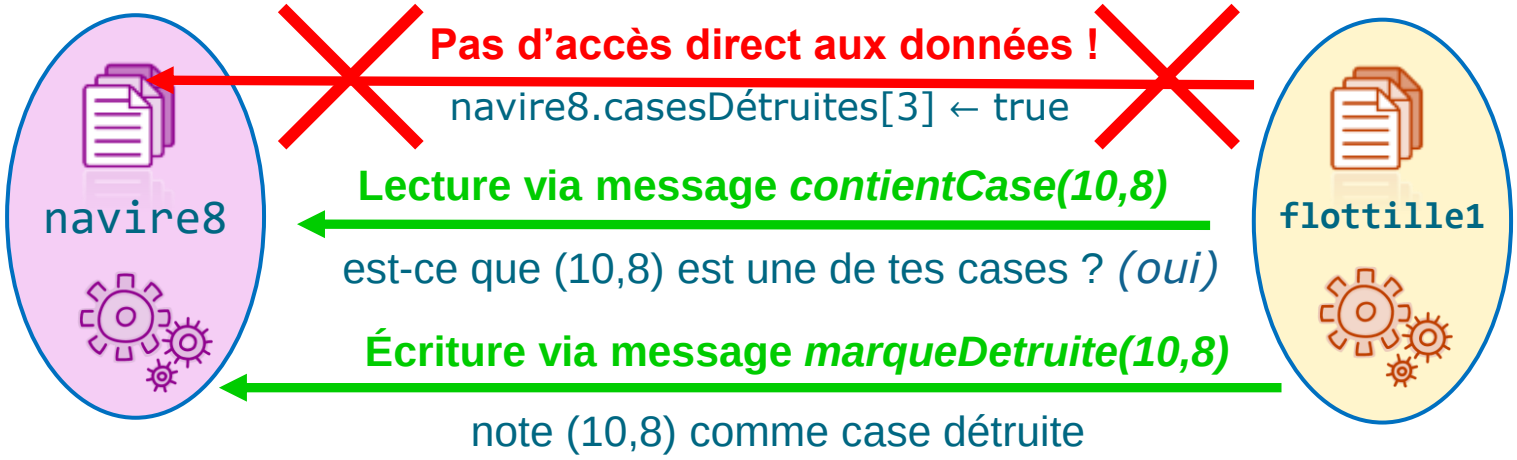
Les objets font appel les uns aux autres pour accomplir les tâches



C'est ce qu'on appelle des "messages" entre les objets

Approche orientée objet

On n'accède pas directement aux données des objets ;
tout passe par des messages/délégations (= appels de fonction sur les objets) !



⇨ Indépendance des implémentations
(on peut changer la structure d'un navire sans devoir refaire tout le code)

⇨ Empêche des modifications invalides des données

Approche orientée objet

```
touché ← false
coulé ← false
Examiner les navires un à un à la recherche la case cible
if on trouve iNavire tel que flottille[iNavire] touché
| marquer la case cible comme détruite dans le navire
| touché ← true
| Vérifier si toutes les cases du navire sont détruites
| if c'est le cas
| | coulé ← true
| endif
endif
```

Code procédural
On va lire et **modifier** les données des navires

```
touché ← false ; coulé ← false
Demander à chaque navire s'il contient la case cible
if on trouve un navire répondant positivement,
| Demander au navire de marquer la case comme détruite
| touché ← true
| Demander au navire s'il est entièrement détruit
| if c'est le cas
| | coulé ← true
| endif
endif
```

Code OO
On ne travaille pas directement sur les données des navires ; on **demande** aux navires de le faire !

Approche orientée objet

Cette structuration est concrétisée par l'organisation du code (exemple en Java)

Un fichier par type d'acteur (classe)

```
class Navire {

  int coordonneeXTete;
  int coordonneeYTete;
  boolean casesDetruites [];
  …

  void marqueDetruite (int x, int y) {
    …
  }

  boolean estCoule () {
    …
  }

  …

}
```

Partie "données"

Partie "actions"

Appel correspondant à un message

```
class Flottille {

  Navire [] navires;
  …

  void afficheGrille () {
    …
  }

  boolean estDetruite () {
    …
    … navires[i].estCoule()…
    …
  }

  …

}
```

Comparaison : procédural >< OO

1. *En procédural :*
   Le programme se découpe selon les fonctionnalités

   En OO :
   Le programme se découpe selon les acteurs / types de données

2. *En procédural,* si on travaille à plusieurs,
   - on se met d'accord sur la structure des données puis
   - on peut se répartir les modules à construire (découpe fonctionnelle)

   En OO, si on travaille à plusieurs,
   - on se met d'accord sur les messages que les objets peuvent s'échanger puis
   - on peut se répartir les classes (= types d'objets) à construire

Comparaison : procédural >< OO

3. *En procédural :*

Chaque module peut accéder à / modifier la plupart des données (sans autorisation).

En OO :

Les objets n'accèdent (généralement) pas directement aux données d'autres objets : toute modification se fait sous la forme d'un message (libre à l'objet concerné de traiter la demande ou pas).

Exemple : marquer une case de navire comme détruite	
Procédural	`navire.casesDetruites[i]` peut être modifié par n'importe qui
OO	Si on rend le tableau des cases détruites inaccessible de l'extérieur, on ne peut le modifier que via un message au navire : `navire.marqueDetruite(x,y)` Il est donc possible d'interdire certaines modifications indésirables

Comparaison : procédural >< OO

4. *En procédural :*
Si on change la structure des données, il faut revisiter tout le code :
le code est "fortement couplé"

En OO :
- Comme toute modification doit passer par un message, on peut changer le code interne d'une classe sans impact sur le code externe.

- Le code interne à une classe et le code qui utilise cette classe sont indépendants : on peut en modifier un sans devoir changer l'autre.
C'est ce qu'on appelle un **couplage faible**

Exemple : si on change la manière dont les navires sont encodés (*utiliser un tableau `casesOccupées[]` au lieu des coordonnées de la tête, direction et taille*)	
Procédural	Quasiment tout le code doit être réécrit
OO	Il suffit de réécrire le code des actions liées aux navires. Le reste du code (appel via messages) reste valable.

Comparaison : procédural >< OO

EN PROCÉDURAL	EN ORIENTÉ OBJET
Données = structures Données et code séparés	Objet = entité "encapsulée" = données + traitement
Le code manipule directement les données	Normalement, pas d'accès direct aux données : toute action passe par un message (possibilité de refuser d'exécuter la demande)
Le code est divisé selon les fonctionnalités (modularité)	Le code est divisé selon les acteurs (autre modularité).
Code et données fortement liés (couplage fort)	Couplage faible entre les objets (si on fait bien les choses)
Concepts plus simples ?	Concepts plus naturels ? (on peut utiliser un objet sans savoir comment il est codé)
On retrouve la même quantité de code et le "même" code des deux côtés.	
	Clairement "tendance"

Présentation du cours

	1er quadrimestre	2e quadrimestre
Cours	PP	PPOO
Langage associé	C	Java
Notations utilisées	Diagrammes d'actions	Diagrammes de classes UML
"Briques" utilisées	Instructions	Objets
"Ciment" utilisé	Modules, séquences d'instructions	Organisation en classes, messages entre objets
Éléments avancés	Conditionnelles Boucles Tableaux Structures	Héritage Polymorphisme

Présentation du cours

Lien avec le cours de Langage de programmation orienté objet (Java)

Évaluation (théorie + exercices) :

- interrogation formative
- examen