

LES BASES DE LA PROGRAMMATION ORIENTEE OBJET

Romuald GRIGNON

Coding Factory by ITESCIA
CCI Paris Ile-de-France
Cergy-Pontoise

Année scolaire 2018-2019

PRESENTATION DU COURS

Le langage JAVA

- Langage compilé (OU interprété suivant le point de vue)
- ByteCode (analogie avec le code Assembleur)
- Machine Virtuelle Java (JVM) - Analogie avec le processeur, interprète le Bytecode
- Portabilité accrue, indépendant de la machine utilisée (seule la JVM est requise)
- Fortement typé, Rigoureux
- Robuste (une majorité de problèmes est vue à la compilation et non à l'exécution)
- Programmation Orientée Objet
- Créé par James Gosling et Patrick Naughton (Société SUN)
- Date officielle : 23 mai 1995
- Nombre d'équipements utilisant Java : plusieurs milliards

Les outils de développement

- Java Runtime Environment (JRE) ou Java Development Kit (JDK)
- Exécutables : java
- Environnement de développement (IDE) : NetBeans
- Chaîne de compilation : Maven
- Création de projet
- Exécution et Debug

P.O.O. : LES CONCEPTS

(1^{ère} partie)

Classes d'objets

- Un objet est considéré comme une variable (complexe) et sa définition est une classe
- Une classe est le type de l'objet (au même titre qu'une variable peut être de type entier, flottant, booléen, ...)
- Une classe regroupe des données à stocker (variables, objets) et du code exécutable
- Les éléments contenus dans une classe sont appelés **membres**
- Les membres servant au stockage sont des **attributs** ou **propriétés**
- Les membres servant à l'exécution sont des **méthodes** (analogie avec les fonctions)

Propriétés / Attributs

- Analogie avec les variables
- Les types primitifs en Java
 - `boolean` : information logique (prend 2 valeurs : `true` ou `false`)
 - `char` : codage ASCII d'un caractère (sur 1 octet)
 - `byte` : valeur entière signée sur 1 octet (de -128 à +127)
 - `short` : valeur entière signée sur 2 octets (de -32768 à +32767)
 - `int` : valeur entière signée sur 4 octets (de -2147483648 à +2147483647)
 - `long` : valeur entière signée sur 8 octets (O_o)
 - `float` : valeur décimale signée sur 4 octets
 - `double` : valeur décimale signée sur 8 octets
 - La précision des valeurs décimales dépend de la valeur entière

Propriétés / Attributs

- Les types d'objet :
 - Toute classe définit un type d'objet. Par conséquent, un objet peut avoir comme attribut un autre objet
 - Contrairement à un attribut de type primitif, qui contient directement la valeur à stocker, un attribut de type objet contient seulement la référence de l'objet
 - Un objet, comme tout autre attribut de type primitif, lors de sa création va être alloué en mémoire. Après cette allocation, c'est l'adresse mémoire de l'objet, sa référence, qui va être stockée et utilisée pour y accéder
 - Cette référence servira pour accéder aux attributs et méthodes de l'objet. On peut faire l'analogie avec les pointeurs mémoire utilisés dans d'autres langages

Propriétés / Attributs

- Les tableaux :
 - Il est possible de déclarer des tableaux (de type primitif ou objet) à 1 ou plusieurs dimensions
 - En Java, la déclaration des tableaux ou leur accès se fait d'une manière similaire à plusieurs langages de programmation en utilisant les symboles crochets
- Les chaînes de caractères :
 - En langage Java, la chaîne de caractères n'est pas gérée comme un type primitif mais comme un objet de la classe `String`. Cette classe propose donc tous les services nécessaires aux traitements de ce type d'objet.
- Les constantes :
 - Il est possible de définir des constantes, autant pour un type primitif qu'un objet.
 - Le mot-clé utilisé est « `final` »

Encapsulation / Portée

- Les membres d'une classe peuvent être limités en portée. C'est à dire qu'ils peuvent ne pas être accessible de n'importe quel endroit du code. Cette limitation est explicitement décrite lors de la définition du membre.
- En Java, les mots-clés définissant ces limitations sont les suivantes :
 - **public** : le membre est accessible depuis n'importe quel endroit du code
 - Si le membre est un attribut, on peut y accéder librement à partir du moment où l'on possède la référence de l'objet
 - Si le membre est une méthode, on peut l'exécuter à partir du moment où l'on possède la référence de l'objet
 - **private** : le membre n'est accessible que depuis sa classe
 - Si le membre est un attribut, il n'est accessible qu'à partir des méthodes de la classe
 - Si le membre est une méthode, elle n'est exécutable que depuis une autre méthode de la classe

Encapsulation / Portée

- **protected** : le membre est accessible depuis toute classe (ou depuis toute sous-classe, même en dehors du package si la classe mère est visible en dehors de son package d'origine, voir **héritage** et **package**)
 - C'est un mode à mi-chemin entre les modes **public** et **private**, permettant une souplesse d'accès au membre, tout en garantissant un minimum de sécurité
- si il n'y a aucun mot clé, le membre est, par défaut, accessible depuis toute classe du package (mais pas les sous-classes)
 - Il est préférable d'indiquer explicitement la portée des membres afin d'indiquer les besoins. Dans le cas de l'encapsulation par défaut, comme il n'y a pas de mot-clé, un commentaire dans le code est plus qu'approprié, ceci afin d'indiquer aux autres développeurs que ce n'est pas un oubli mais une décision intentionnée

Objet

- Création
 - Une fois que les classes ont été créées, il faut instancier une classe afin de créer un objet en mémoire. Cette opération passe inévitablement par l'appel d'une méthode de la classe que l'on appelle le **constructeur**. Cette méthode peut prendre des paramètres en entrée afin de pouvoir configurer l'objet en fonction des besoins, et va retourner une référence sur l'objet nouvellement créé.
- Destruction
 - Pour retirer un objet de la mémoire, il suffit que plus aucun autre objet n'y fasse référence. On ne détruit pas explicitement un objet, mais on l'oublie.
 - Si personne ne fait plus référence à un objet cela veut dire que l'on peut l'enlever de la mémoire sans risque : cette opération est effectuée de manière automatique et asynchrone par le **garbage collector** (voir paragraphe associé).

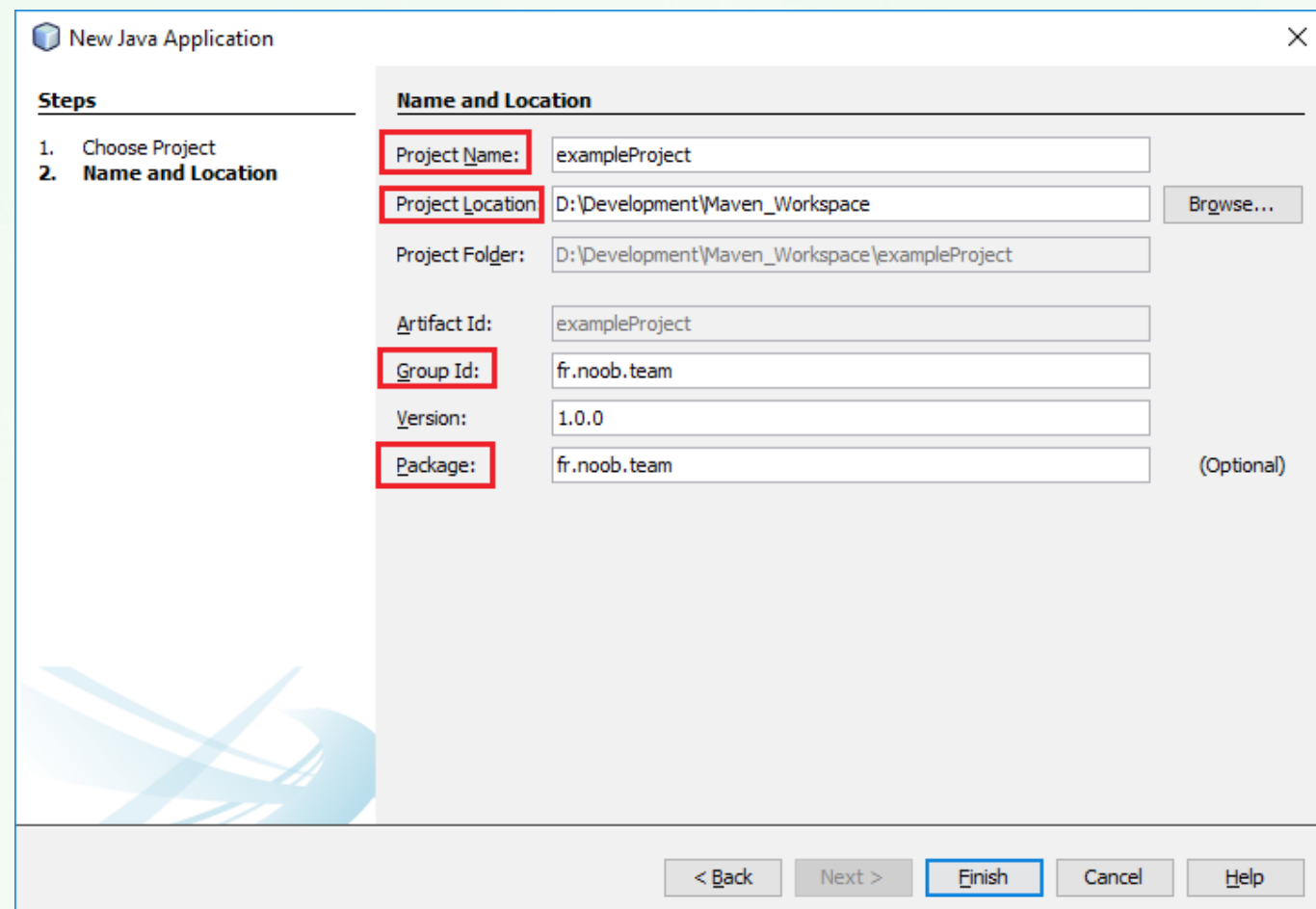
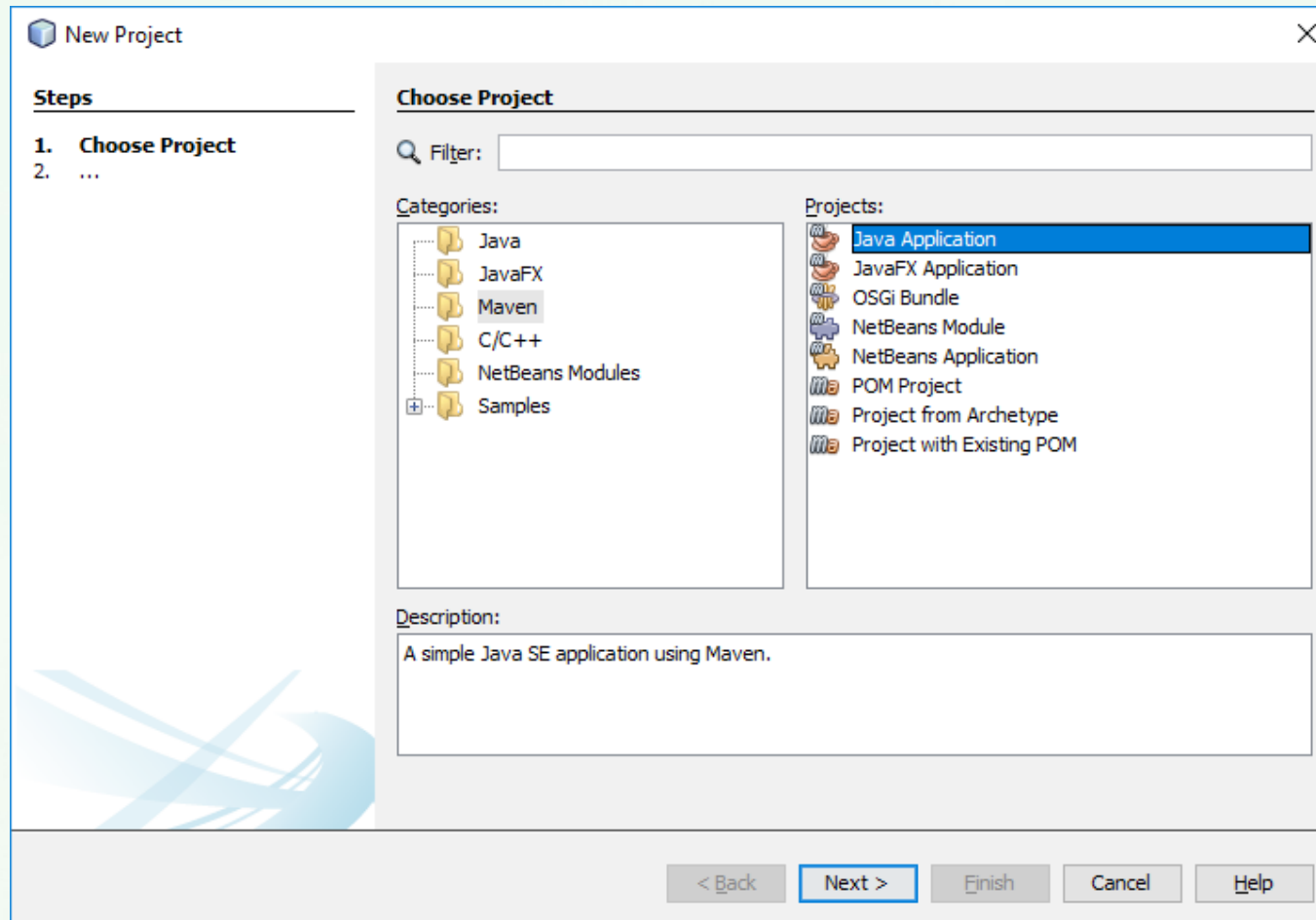
Objet

- instance VS static
 - Par défaut les membres d'une instance d'un objet n'appartiennent qu'à cet objet
 - Il existe un mécanisme permettant de partager des membres (attribut ou méthode) entre plusieurs instances d'objet : en Java c'est le mot-clé `static`
 - Le membre sera donc unique pour toutes les instances
 - Si le membre est un attribut, chaque objet verra la même valeur (analogie avec les variables globales)
 - Pour ce membre, l'accès ne se fait pas depuis la référence d'un objet mais par le nom de la classe directement

Environnement de développement : NetBeans

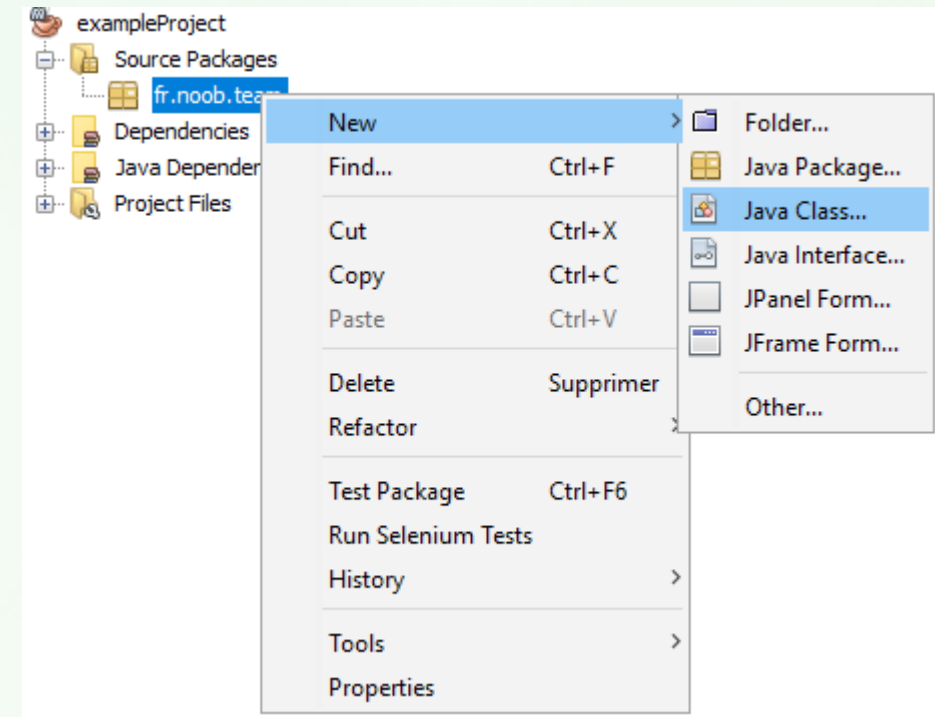
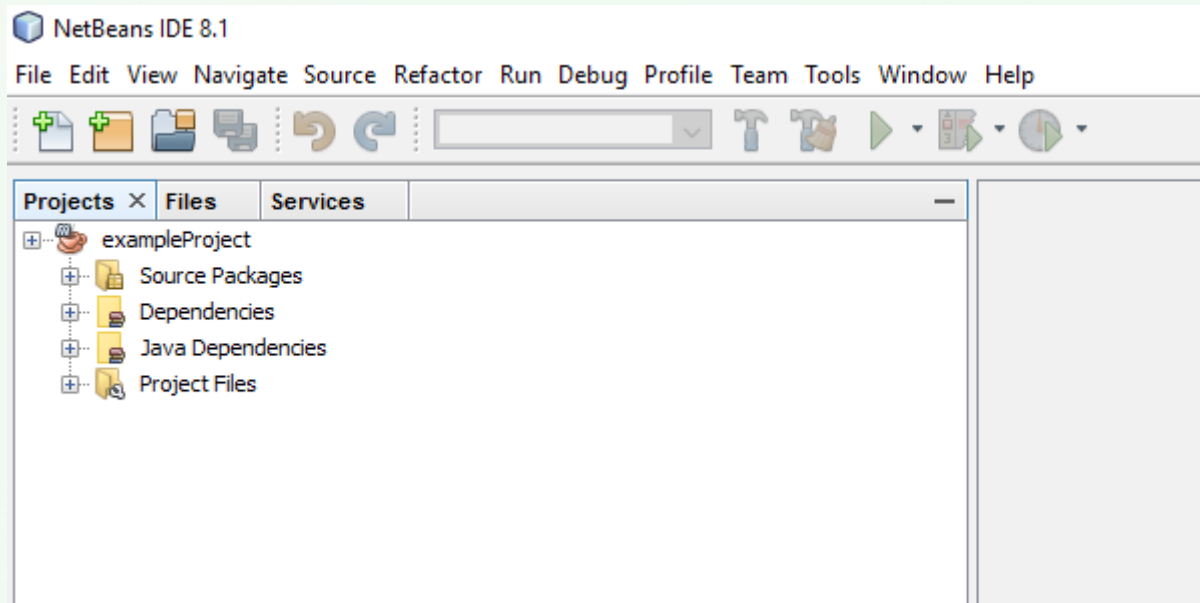
IDE : NetBeans

- Création d'un projet de type Maven



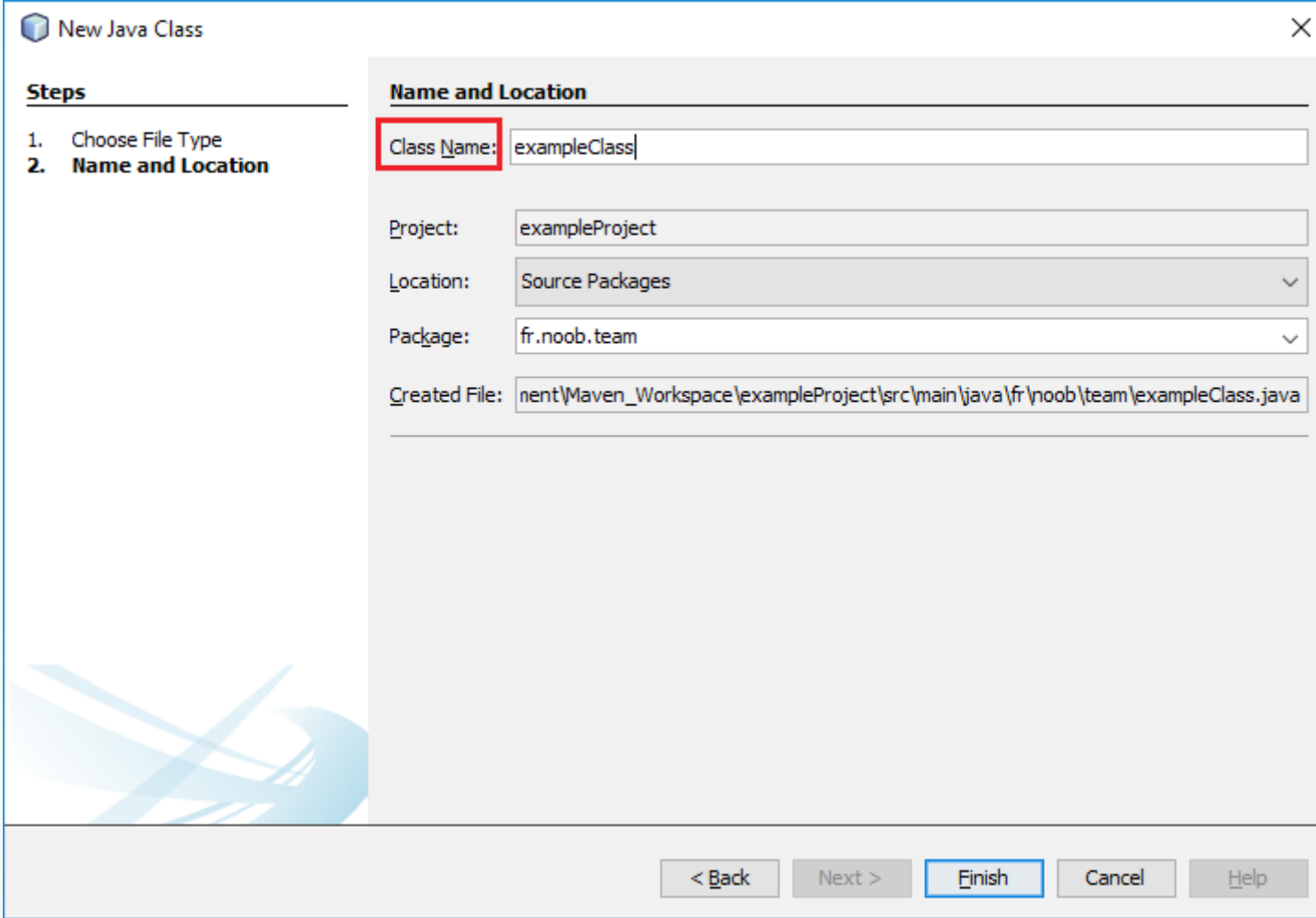
IDE : NetBeans

- Création d'une classe



IDE : NetBeans

- Création d'une classe



The image shows the 'New Java Class' dialog box in NetBeans. The dialog is titled 'New Java Class' and has a close button (X) in the top right corner. On the left, there is a 'Steps' section with two steps: '1. Choose File Type' and '2. Name and Location'. The 'Name and Location' section is currently active. It contains several input fields: 'Class Name:' (with a red box around it) containing 'exampleClass', 'Project:' containing 'exampleProject', 'Location:' (a dropdown menu) set to 'Source Packages', 'Package:' (a dropdown menu) set to 'fr.noob.team', and 'Created File:' containing the full path 'nent\Maven_Workspace\exampleProject\src\main\java\fr\noob\team\exampleClass.java'. At the bottom of the dialog, there are five buttons: '< Back', 'Next >', 'Finish' (highlighted with a blue border), 'Cancel', and 'Help'.

New Java Class

Steps

1. Choose File Type
2. **Name and Location**

Name and Location

Class Name: exampleClass

Project: exampleProject

Location: Source Packages

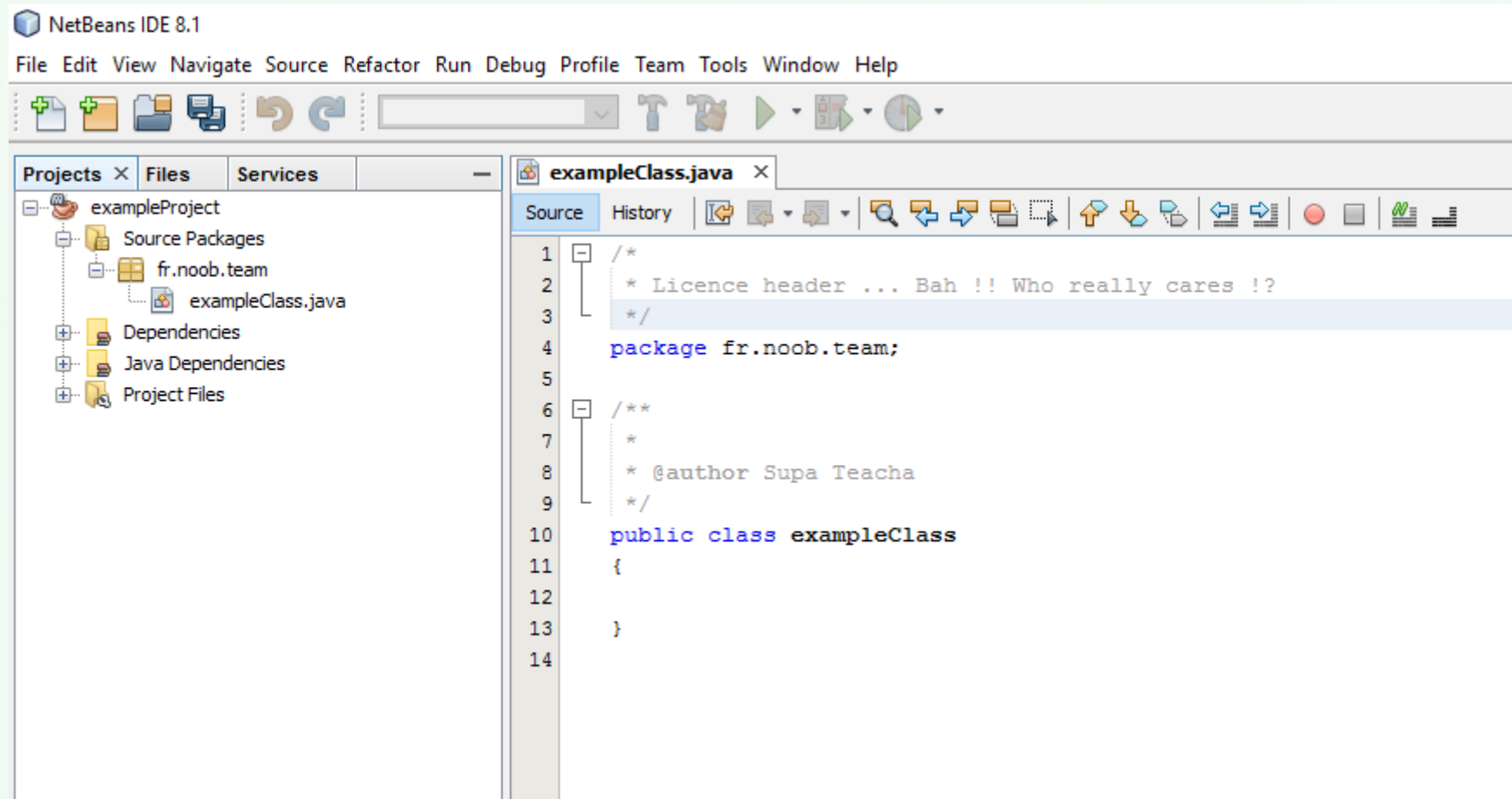
Package: fr.noob.team

Created File: nent\Maven_Workspace\exampleProject\src\main\java\fr\noob\team\exampleClass.java

< Back Next > Finish Cancel Help

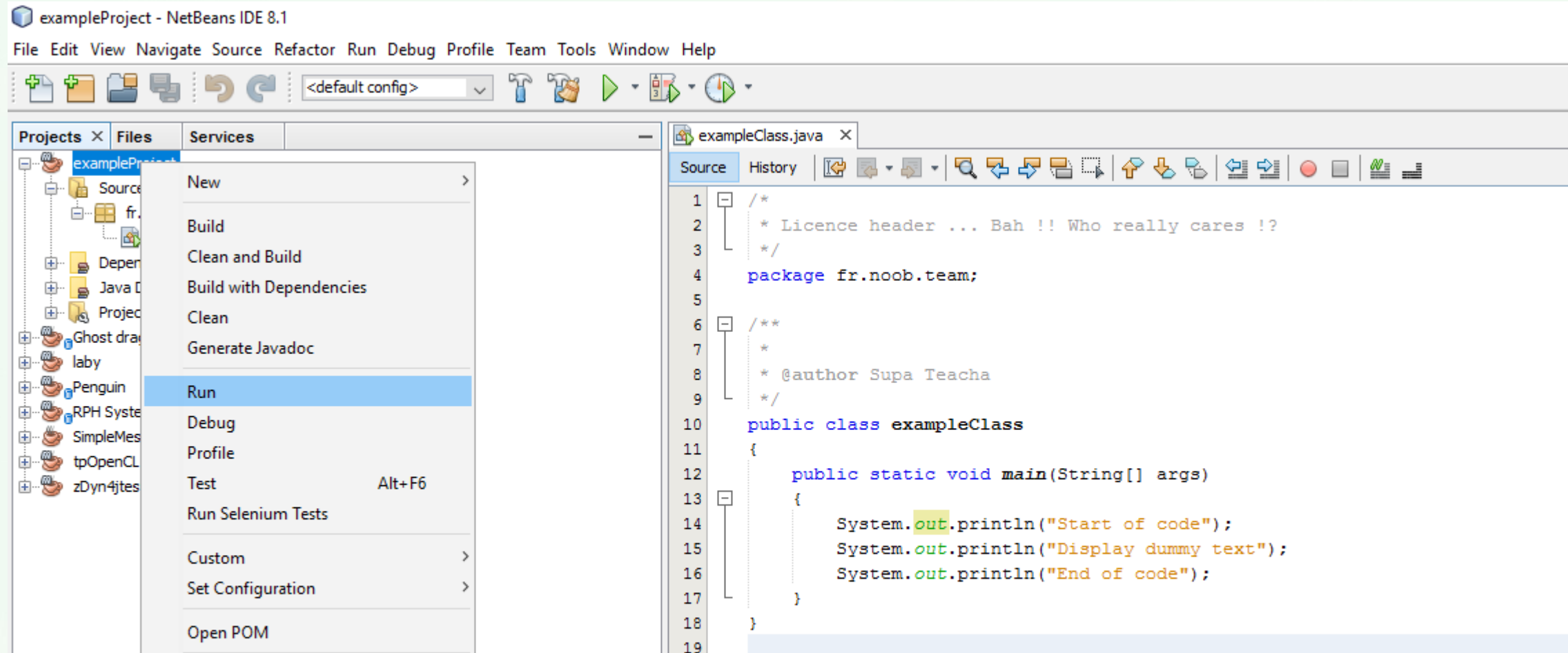
IDE : NetBeans

- Création d'une classe



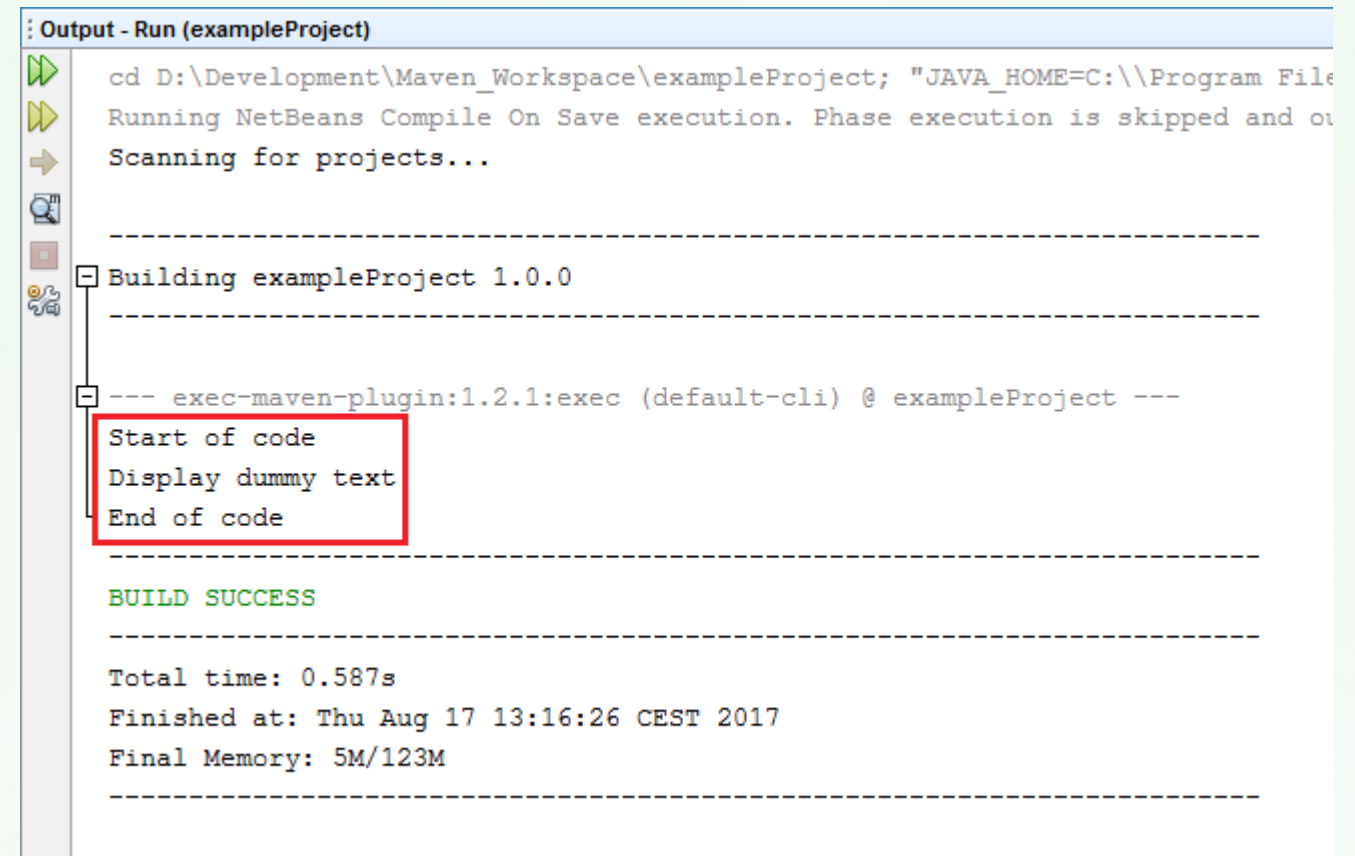
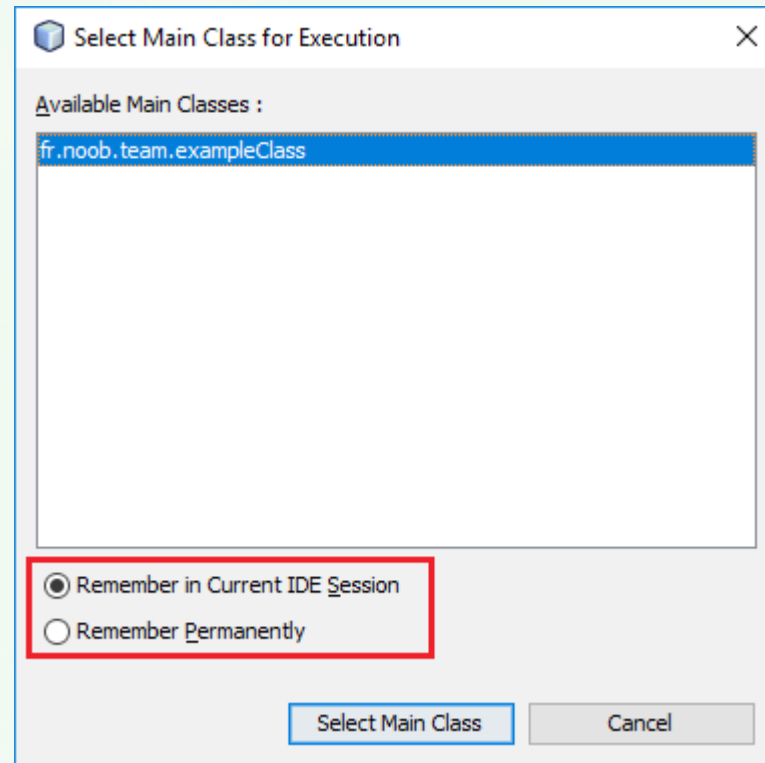
IDE : NetBeans

- Exécution du programme
- Il est nécessaire d'avoir défini une méthode principale (voir chapitre correspondant)



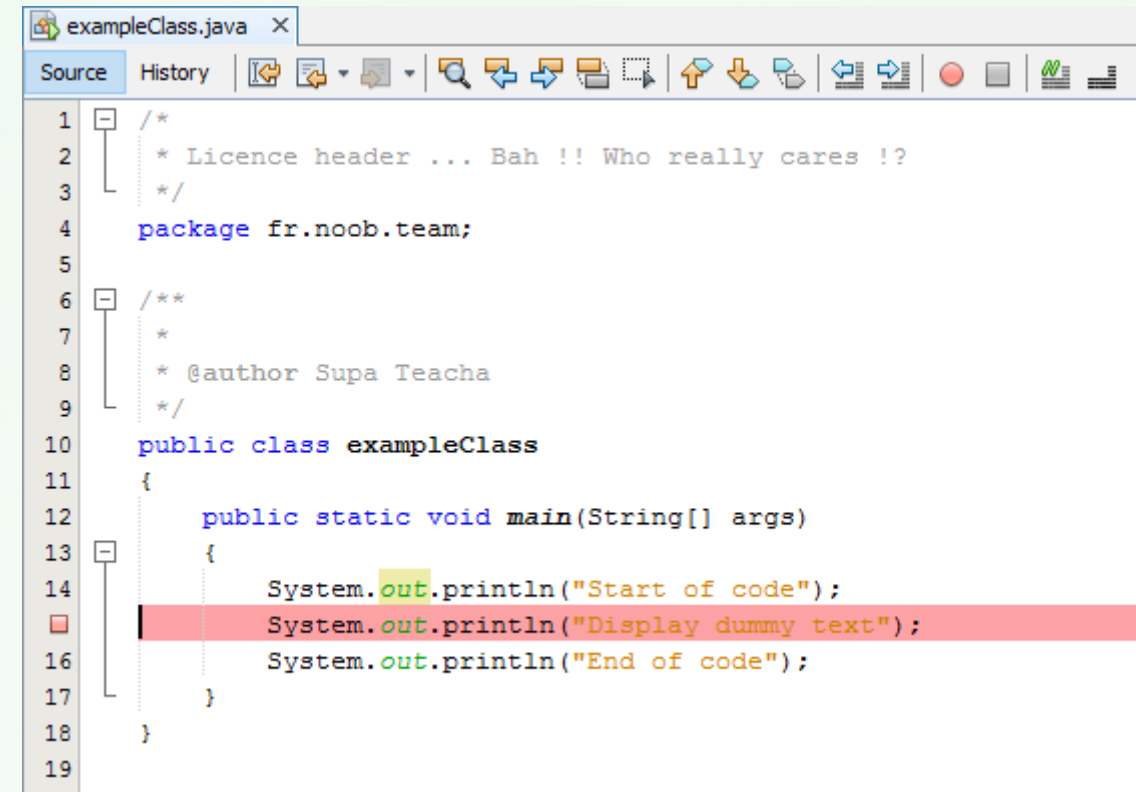
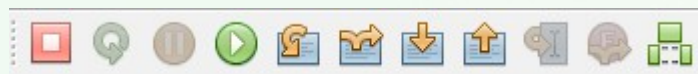
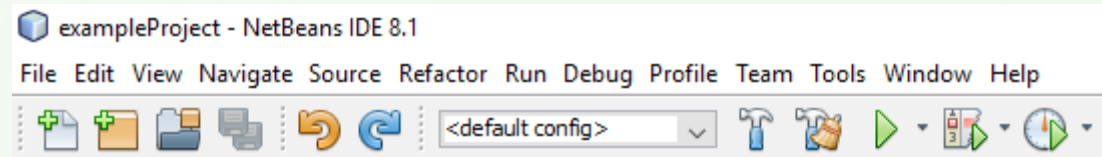
IDE : NetBeans

- Exécution du programme : la classe principale



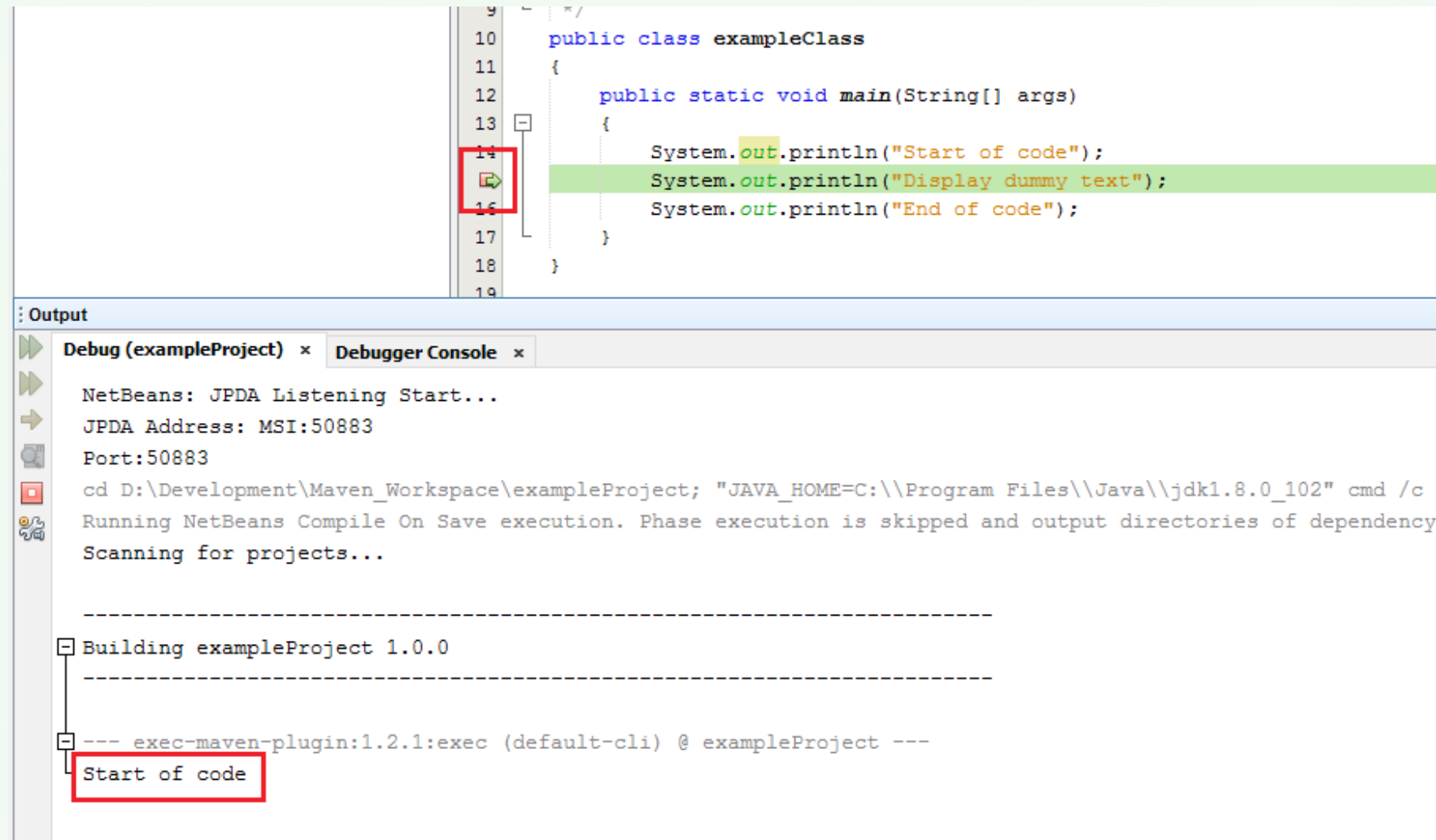
IDE : NetBeans

- Exécution du programme : Debug / points d'arrêt



IDE : NetBeans

- Exécution du programme : Debug / points d'arrêt



IDE : NetBeans

- Exécution du programme : Debug / Variables
- TODO

Exécution d'un Programme JAVA (1^{ère} partie)

Syntaxe générale

- Les instructions se terminent toujours par « ; »
- Les lignes de commentaires (non compilées) débutent par « // »
- Les blocs de plusieurs lignes de commentaires sont entourés par « /* » et « */ »
- Les caractères « { » et « } » sont utilisés pour encadrer des blocs de code
- Les blocs de code permettent de définir des variables locales accessibles uniquement dans ce bloc
- Le caractère « . » est utilisé pour accéder aux membres d'un objet

Classes

- Chaque classe doit être stockée dans un fichier portant le nom de la classe suivi de « .java »
- La casse du nom de la classe doit être identique au nom du fichier
- Un nom de classe commence toujours par une lettre majuscule et chaque début de mot commence lui aussi par une majuscule

```
class PseudoRandomGenerator
{
    ...
}

class Launcher
{
    ...
}
```

- Une erreur s'est glissée à plusieurs endroits dans les slides précédents : l'avez-vous remarquée ?

Méthode principale

- Pour exécuter un programme Java, il nous faut un point d'entrée, C'est une méthode spécifique qui servira de point de départ : la méthode principale

```
...  
public static void main(String[] args)  
{  
    ...  
}  
...
```

- Notez le mot clé « static » qui indique que cette méthode est unique dans le code (surtout le fait qu'elle n'est liée à aucune instance de la classe en particulier)
- Notez le tableau de chaînes de caractères qui est passé en paramètres, comme la plupart des programmes exécutables

Constructeur

- Dans chaque classe existe un constructeur, une méthode qui est appelée à chaque instantiation de la classe

```
...  
public PseudoRandomGenerator()  
{  
    ...  
}  
...
```

- On peut écrire plusieurs constructeurs différents, qui prennent différents paramètres, en fonction des besoins. C'est lors de l'écriture du programme que le compilateur saura quelle méthode appeler (voir [surcharge](#))

```
...  
public PseudoRandomGenerator(int initialSeed)  
{  
    ...  
}  
...
```

Méthodes

- Les nom des méthodes, comme tous les membres d'une classe, commencent par une lettre minuscule et chaque nouveau mot du nom commence par une Majuscule

```
...  
private int getNextValue()  
{  
    ...  
}  
...  
public double random()  
{  
    ...  
}  
...
```

Attributs

- Les nom des attributs, comme tous les membres d'une classe, commencent par une lettre minuscule et chaque nouveau mot du nom commence par une Majuscule. Les constantes sont souvent écrites en majuscules uniquement

```
final static private int  INIT_SEED = 123456789 ;
final static private long COEF      = 22696477;
final static private long OFFSET   = 1;
final static private long MODULO    = (long) (Math.pow(2, 61))-1;
Final static private long MASK      = 0x7FFFFFFF;
private int internalValue;
```

- Notez l'association des mots-clés **final** et **static** qui définissent une constante unique pour la classe (non-instanciée)
- Notez l'utilisation d'une méthode **static** de la classe **Math** pour définir la valeur de la constante **MODULO**
- Notez que la valeur initiale de notre variable privée **internalValue** n'est pas définie. Ici seuls les type, nom et portée de la variable sont définis

Attributs

- En Java, il existe un mécanisme qui transforme automatiquement une valeur de type primitif en référence de la classe associée : l'**autoboxing**
- Par exemple, une valeur entière déclarée avec le mot-clé **int** pourra être utilisée comme référence d'objet de type **Integer**. Cette dernière classe est appelée **wrapper**
- La transformation implicite inverse existe elle aussi, affecter directement la référence d'un objet de type **Float** dans une variable de type **float** : c'est l'**unboxing**
- Liste de types primitifs et classes wrappers associées

```
boolean → Boolean  
byte    → Byte  
short   → Short  
int      → Integer  
long     → Long  
float    → Float  
double   → Double  
char     → Char
```


Finalisation de la classe d'exemple

- Maintenant que nous avons défini notre classe `PseudoRandomGenerator`, ses attributs, ses méthodes, et ses constructeurs, il nous reste à remplir le code des méthodes
- Le constructeur : On initialise les variables internes dont on a besoin (ici la variable `internalValue`)

```
Public PseudoRandomGenerator()  
{  
    this.internalValue = PseudoRandomGenerator.INIT_SEED;  
}  
Public PseudoRandomGenerator(int initialSeed)  
{  
    this.internalValue = initialSeed;  
}
```

- Notez l'utilisation du mot-clé `this` pour indiquer que l'on accède à l'instance courante
- Notez l'utilisation du nom de la classe elle-même pour accéder aux membres `static`
- D'une manière générale, privilégiez l'initialisation des attributs dans le constructeur

Finalisation de la classe d'exemple

- La méthode privée : on va calculer une valeur pseudo-aléatoire à partir de la valeur courante (l'attribut privé de la classe) et des constantes privées

```
private int getNextValue()
{
    // On initialise une variable locale 64 bits avec la valeur courante
    long value = (long)this.internalValue;
    // on multiplie par un coefficient
    value *= PseudoRandomGenerator.COEF;
    // on ajoute un offset
    value += PseudoRandomGenerator.OFFSET;
    // on applique un modulo
    Value %= PseudoRandomGenerator.MODULO;
    // On garde les 31 bits du milieu de notre variable locale 64 bits
    // 31 bits pour avoir une valeur strictement positive
    this.internalValue = (int)((value>>16) & PseudoRandomGenerator.MASK);
    // On retourne la nouvelle valeur
    return this.internalValue;
}
```

- Notez les différents opérateurs de somme, multiplication, modulo, décalage de bits, masque binaire, qui sont détaillés dans la section [opérateurs](#)

Finalisation de la classe d'exemple

- La méthode publique : ici on va faire appel au calcul de la méthode privée, formater ce résultat avant de le renvoyer à la couche appelante

```
public double random()
{
    // on récupère la prochaine valeur pseudo-aléatoire
    long value = this.getNextValue();
    // on la converti en valeur décimale normalisée entre 0.0 et 1.0
    // et on retourne ce résultat
    return ((double)v)/PseudoRandomGenerator.MASK;
}
```

- Dans ce petit exemple, nous venons de créer une classe dont chaque instance est capable de générer une suite de nombres décimaux pseudo-aléatoires (entre 0.0 et 1.0).
- Le service proposé par une bibliothèque logicielle pour générer des nombres aléatoires est un classique. Nous pourrions comparer les performances, en terme de vitesse d'exécution, de notre méthode `random` avec `Math.random`

Finalisation de la classe d'exemple

- Il nous reste maintenant à utiliser notre classe. Pour cela, nous allons créer une classe **Launcher** qui va seulement contenir une méthode principale (comme vu précédemment).
- Cette classe va nous servir à instancier la classe PseudoRandomGenerator et appeler la méthode publique random()

```
Class Launcher
{
    public static void main(String[] args)
    {
        PseudoRandomGenerator prng;
        double value = 0.0;
        prng = PseudoRandomGenerator();
        for(int i=0;i<10;i++)
        {
            value = prng.random();
            System.out.println( value )
        }
    }
}
```

Exécution et performances

- Pour que notre programme puisse compiler, il nous reste tout de même à lui indiquer où trouver les définitions de notre classe. Il faut donc « l'importer » :

```
import PseudoRandomGenerator
```

```
Class Launcher  
{  
    ...  
}
```

- Nous pouvons maintenant mesurer le temps d'exécution de notre programme :

```
...  
long myTime = System.nanoTime() ;  
for(int i=0;i<100000000;i++)  
{  
    value = prng.random();  
}  
myTime = System.nanoTime()-myTime;  
System.out.println( "execution time = " + Long.toString(myTime) + " nsec" );  
...
```

- Modifiez l'appel `prng.random` par `Math.random` et comparez les performances

Opérateurs en JAVA

- Liste des opérateurs utilisés dans le langage JAVA :

pri	opérateur	syntaxe	pri	opérateur	syntaxe
1	++	++<ari>	5	<	<ari> < <ari>
	++	<ari>++		<=	<ari> <= <ari>
	--	--<ari>		>	<ari> > <ari>
	--	<ari>--		>=	<ari> >= <ari>
	+	+<ari>	6	instanceof	<val> instanceof <cla>
	-	-<ari>		==	<val>==<val>
	!	!<boo>		!=	<val>!=<val>
2	(type)	(type)<val>	7	&	<ent>&<ent>
	*	<ari>*<ari>	8	&	<boo>&<boo>
	/	<ari>/<ari>		^	<ent>^<ent>
3	%	<ari>%<ari>	9	^	<boo>^<boo>
	+	<ari>+<ari>			<ent> <ent>
	-	<ari>-<ari>	10		<boo> <boo>
4	+	<str>+<str>		&&	<boo>&&<boo>
	<<	<ent> << <ent>	11		<boo> <boo>
	>>	<ent> >> <ent>	12	?:	<boo>?<ins>:<ins>
			13	=	<var>=<val>

- Notez la priorité de ces opérateurs. Si vous hésitez sur les priorités inter-opérateurs, utilisez plutôt des parenthèses afin de séquencer manuellement les calculs à faire

Contrôles d'exécution en JAVA

- Les branchements conditionnels `if...else` :

```
if (...)
{
    ...
}
else
{
    ...
}
```

- Si la condition testée est vraie, seul le premier bloc de code est exécuté
- Si la condition testée est fausse, seul le deuxième bloc de code est exécuté (else)
- Notez que le deuxième bloc de code est optionnel

Contrôles d'exécution en JAVA

- La boucle `while` :

```
while (...)
{
    ...
}
```

- Ici on exécute le bloc de code plusieurs fois, ceci tant que la condition testée est vraie

- La boucle `do...while` :

```
do
{
    ...
}
while (...)
```

- Ici le principe est le même que précédemment mais le bloc de code est exécuté avant de tester la condition (même si la condition est fausse)
- Notez que si les conditions sont mal conçues, le programme peut partir dans un cycle que l'on appelle « boucle infinie » et ne jamais s'arrêter

Contrôles d'exécution en JAVA

- La boucle **for** :

```
for( <init> ; <conditions> ; <opérations> )  
{  
    ...  
}
```

- Ici le principe reste le même que pour la boucle **while**. La différence tient dans le fait que l'on peut exécuter des instructions **<init>** avant de démarrer le premier test de condition **<conditions>**. Si la condition est vraie, le bloc de code est exécuté puis avant de tester à nouveau la condition, on exécute du code **<opérations>**
- Ici aussi on s'appliquera à vérifier que les instructions **<init>**, **<opérations>** et les **<conditions>** ne créent pas un état de boucle infinie

Contrôles d'exécution en JAVA

- Les interruptions de boucle :
- On peut explicitement demander à passer à la boucle suivante (même si le bloc de code actuel n'est pas terminé). On utilise pour cela le mot-clé `continue`
- On peut également demander à sortir de la boucle, même si les conditions testées sont encore vraies. On utilise pour cela le mot-clé `break`

Garbage Collector et effacement de référence

- Le garbage collector, est une fonctionnalité de la machine virtuelle Java, qui permet d'effacer de la mémoire certains objets alloués précédemment
- D'une manière générale, les objets ne sont pas explicitement effacés en Java, mais collectés en tâche de fond de manière asynchrone lorsqu'ils ne sont plus utilisés
- Pour déterminer si un objet est encore utilisé par le programme, c'est qu'un autre objet y fait référence. A l'inverse, si personne ne fait plus référence à notre objet alors nous pouvons l'effacer sans risque
- Pour "oublier" une référence d'objet, il suffit d'affecter dans notre variable la valeur `null` qui est un mot-clé pour dire « pas de référence » :

```
...  
PseudoRandomGenerator rng = new PseudoRandomGenerator();  
...  
rng = null;    ← ici on oublie la référence retournée par le constructeur  
...
```

Les entrées /sorties : le clavier

- En langage Java il existe un ensemble de services pour lire ou écrire des informations qui transitent dans le système (clavier, fichier, écran, ...)
- Nous avons déjà vu un exemple avec la méthode `println` de l'objet `System.out` qui permet d'envoyer des flux de caractères sur la sortie standard
- Il est également possible de lire des appuis sur le clavier :

```
System.out.println("Entrez du texte et appuyez sur ENTREE : ");
Scanner sc = new Scanner(System.in);
String text = sc.nextLine();
System.out.println("Vous avez entré : " + text);
```

- Pour ne récupérer qu'un type particulier (ex : un entier) une méthode existe :

```
int number = sc.nextInt();
```

- Dans cet exemple si vous entrez autre chose qu'un entier nous obtiendrons une erreur `InputMismatchException` (voir [exceptions](#))
- Notez que la lecture du clavier est bloquante dans l'exécution de votre programme

Méthodes : surcharge

- Une méthode d'une classe peut être utilisée pour retourner différents types d'objets et/ou prendre en paramètres différents types d'objets
- Cette méthode est dupliquée (nom identique) : seuls les types des paramètres d'entrée et de retour, ainsi que la portée, sont modifiés
- Voici un exemple en langage Java de surcharge de méthode de classe :

```
public int multiply(int a, int b);  
public float multiply(float a, float b);  
private float multiply(short a, short b);
```

- C'est lors de la compilation que le programme sait de quels types sont les paramètres d'entrée, et si l'affectation du paramètre de sortie correspond à la déclaration de la variable qui doit le stocker.

P.O.O. : LES CONCEPTS

(2^{ème} partie)

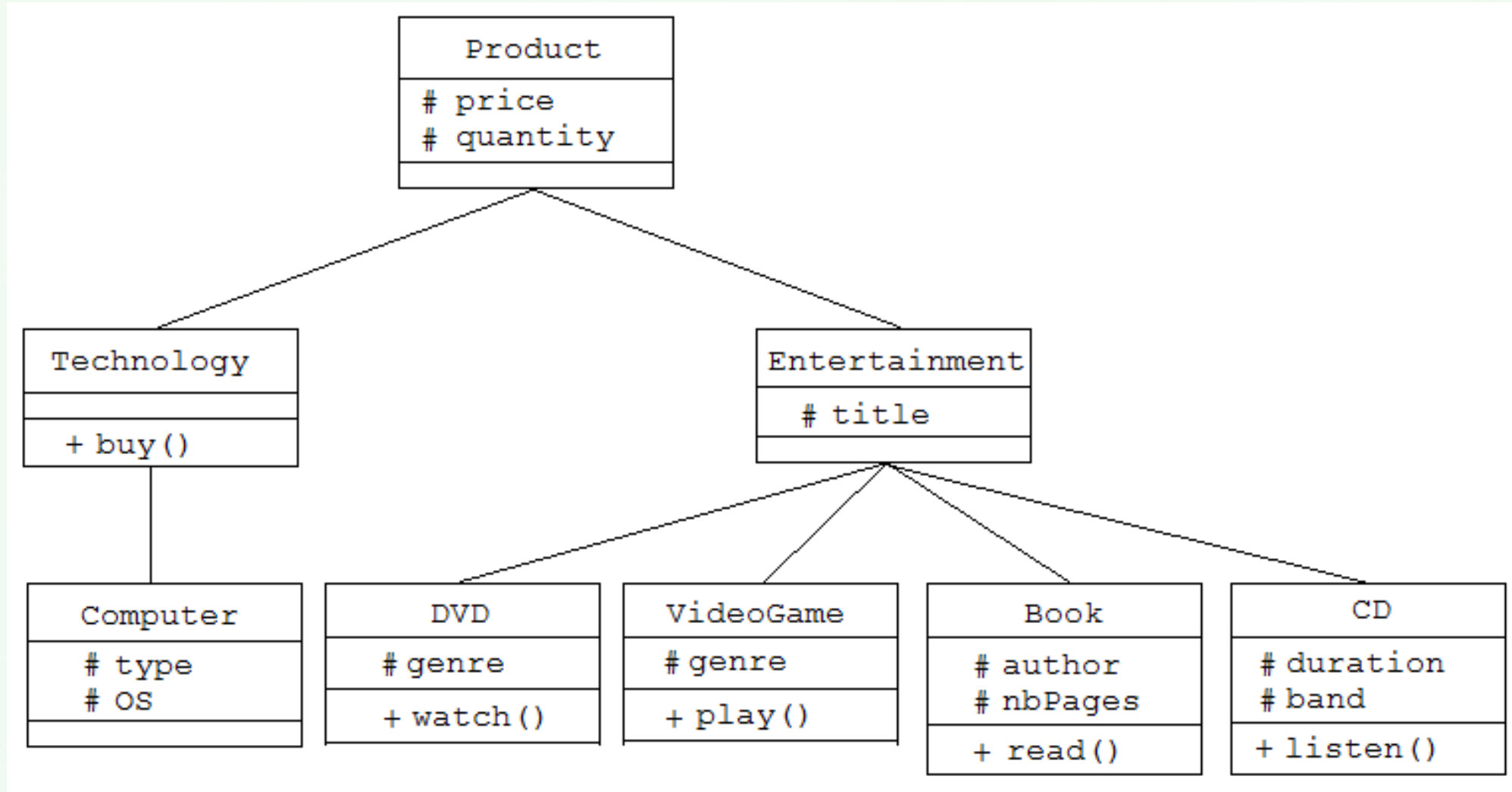
Héritage

- En programmation objet, on peut mutualiser les services d'une classe (méthodes et propriétés) pour les offrir à plusieurs autres classes, ceci afin d'optimiser le code, mais également de simplifier la conception (et le debug) → c'est l'héritage (ou l'extension)
- Une classe-fille qui hérite d'une classe-mère peut donc utiliser les services déjà existants mais peut aussi ajouter ses propres services. La classe fille devient donc, par extension, plus complète, ou plus riche en termes de services
- Dans le langage Java on utilise le mot-clé `extends`
- Une classe-fille ne peut hériter que d'**une et une seule** classe-mère

Héritage : affectation des références

- Upcasting :
 - On peut affecter une référence de classe-fille dans une variable de type classe-mère. Par contre, si l'on souhaite utiliser un service de la classe-fille (appel à une méthode par ex.) nous obtiendrons une erreur à la compilation
- Downcasting :
 - On peut affecter une référence de classe-mère dans une variable de type classe-fille mais ce cast déclenchera une erreur à l'exécution cette fois
- D'une manière générale, évitez le casting autant que possible et optez pour une affectation claire de vos références d'objets

Héritage : polymorphisme



Héritage : polymorphisme

- Un objet dont la classe réelle est une sous-classe d'une hiérarchie, a la particularité d'être de tout type de classe supérieure :
 - Prenons un objet de type **Book**
 - La classe **Book** hérite de la classe **Entertainment**
 - La classe **Entertainment** hérite de la classe **Product**
 - Nous obtenons un objet qui est une instance de **toutes** ces classes à la fois
- Cette particularité est pratique lorsque l'on souhaite effectuer des tests sur l'origine des objets dans notre application
- En langage Java, on utilise le mot-clé **instanceof** pour tester la(les) classe(s) d'appartenance

Héritage : redéfinition des membres

- Si une classe mère définit des membres `public` ou `protected`, alors ils sont accessibles par les instances des classes-filles
- Si une classe-fille redéfinit des membres avec le même nom que ceux de la classe-mère, ce sont ces nouveaux membres qui sont alors utilisés
- Prenons la classe `Computer`, on pourrait lui rajouter un attribut `price` (déjà défini dans la classe grand-mère `Product`), afin de définir un prix temporaire utilisé pendant une période de soldes, afin de pouvoir afficher les deux prix et le pourcentage de réduction associé
- Il est toujours possible d'accéder aux membres de la classe mère (si ils ne sont pas défini comme `private` ou `par défaut`), en castant la référence de notre objet par le type de la classe supérieure (rappelez-vous le polymorphisme)
- Si une classe mère possède un membre avec le mot-clé `final`, cela signifie que ce membre ne peut pas être redéfini dans les classes filles. On peut également empêcher l'héritage en définissant la classe elle-même avec `final`

Héritage : l'évolution des applications

- L'héritage crée un lien fort entre les classes, et peut rendre difficile les évolutions
- L'illustration précédente montre une hiérarchie de classes d'objets servant à fournir des services en ligne d'achat d'ordinateur, d'essai de jeu vidéo en ligne, ou de visualisation et d'écoute de contenus de divertissement
- Suite aux résultats d'une étude marketing, il s'est avéré que de nombreux clients voudraient pouvoir commander une version physique des jeux vidéo
- Prenons la classe `VideoGame` : elle ne contient pas de méthode `buy()` comme la classe `Technology`. Il va donc falloir, soit dupliquer le code, soit déplacer ce code dans une classe supérieure commune : ici la classe `Product`
- Si on déplace le code, il va falloir patcher le fait que l'on ne peut pas acheter les autres produits de la classe `Entertainment`
- Si on duplique le code, il faut dans les couches supérieures qui gèrent l'achat, tester les différentes classes qui peuvent fournir le service `buy()`

Héritage : l'évolution des applications

- Une autre étude marketing nous indique que vendre des livres-audio serait profitable pour la société. On va donc créer une nouvelle classe pour ce type de produit : **AudioBook** qui peut contenir à la fois le service **listen()** et le service **read()**
- Ces services sont déjà présents dans les deux classes **CD** et **Book**
- Ici on pourrait mutualiser le code : créer une classe supérieure aux trois pré-citées : **MusicAndBooks**, qui contiendrait les deux méthodes. Mais en faisant cela, on propose un service superflus à **CD** et à **Book**, il va donc falloir brider la fonctionnalité
- On pourrait également dupliquer les méthodes avec les problématiques déjà abordées précédemment : tester le type de la classe pour pouvoir appeler la méthode **listen()** ou **read()** → nécessite de modifier les couches supérieures de l'application
- Cette modification peut avoir un impact sur tout le reste du code qui utilise les classes **CD** et **Book**. Dans une très grosse application, le nombre de lignes de code à modifier suite à un changement d'architecture peut être colossal, sans parler des régressions fonctionnelles qui peuvent apparaître.

Interfaces

- L'interface est un type, au même titre qu'une classe
- Type abstrait, ne peut donc pas être instanciée
- Ne contient pas de variables mais peut contenir des constantes
- Contient des **signatures** de méthodes mais pas leur contenu
- En langage Java, on utilise le mot clé **interface** en lieu et place de **class**

```
public interface IRead
{
    public int read();
}
```

- Une interface peut hériter d'une autre interface comme vu précédemment avec les classes
- A l'inverse de l'héritage, une classe peut implémenter **plusieurs** interfaces et peut répondre, en partie, aux problématiques de l'héritage vues précédemment

Interfaces

- Dans notre exemple avec les classes `Book` et `AudioBook`, il nous faut maintenant indiquer que l'on implémente l'interface `IRead` dans chacune d'elles
- en Java on utilise le mot-clé `implements`
- C'est à la charge des classes d'implémenter les interfaces (le code des méthodes)

```
public class Book implements IRead
{
    ...
    public int read()
    {
        ...
    }
    ...
}
```

- D'une manière générale, privilégiez autant que possible les interfaces aux héritages
- Ici les interfaces ne nous empêchent pas de dupliquer le code mais elles n'impactent pas la hiérarchie, **seulement** le code des classes que l'on souhaite faire évoluer. Les couches supérieures testeront le type `IRead` pour pouvoir appeler `read()`

Les classes abstraites

- La classe abstraite est un concept à mi-chemin entre l'interface et la classe-mère
- Elle ne peut pas être instanciée : pour créer un objet avec les membres de cette classe, il faut instancier une classe qui en hérite
- Elle peut contenir des attributs et des méthodes comme n'importe quelle classe
- En Java on utilise le mot-clé `abstract`

```
...
abstract class Product
{
    ...
}
...
```

- Une classe abstraite peut contenir une méthode abstraite (une signature de méthode comme dans l'interface) qui doit être implémentée dans la classe fille

```
abstract public int getInformation();
```

- La classe abstraite s'applique bien à `Product`, `Entertainment` et `Technology`

Les Packages

- En langage Java il est possible de regrouper plusieurs classes dans un même ensemble afin de pouvoir distribuer un ensemble de fonctionnalités intégrables par des tiers. On indique l'appartenance d'une classe par le mot clé `package`

```
package fr.noob.team;
```

- L'encapsulation des membres de classe permet un partage d'informations au sein d'un même package (protected ou default) tout en garantissant qu'il n'est pas accessible de l'extérieur
- D'une manière générale, le nom d'un package fait référence à l'entité qui le détient. Il est construit à la manière d'une URL de site web, mais à l'envers.
- Ex : une bibliothèque qui fournit des méthodes de calculs mathématiques, et distribuée par une société basée au royaume uni pourrait s'ecrire :

```
uk.co.mycompany.software.lib.maths
```

Exécution d'un Programme JAVA (2^{ème} partie)

Création d'une hiérarchie de classes

- Implémentez la hiérarchie de classes telle que présentée sur la figure précédente
- Dans chaque classe, ajoutez deux méthodes par attribut pour le récupérer et le mettre à jour (exemple dans `Book`, pour l'attribut `author`, ajoutez `getAuthor` et `setAuthor`)
- Définissez les classes abstraites comme vu précédemment
- Instanciez une fois chaque classe finale (`DVD`, `Book`, `Computer`, ...) en définissant des constructeurs adéquats
- Faîtes des essais pour récupérer/modifier les attributs de chaque objet
- Ajoutez deux méthodes `setPrice` et `getPrice` dans la classe `Computer`, et branchez-les sur celles de la classe mère (nous utiliserons le mot-clé `super` pour accéder à la classe supérieure)
- Ajoutez un attribut privé `price` dans la classe `Computer` et branchez les méthodes `set` et `get` associées

Création d'une hiérarchie d'interfaces

- Modifiez votre code pour utiliser des interfaces pour chaque service
- Instanciez chaque objet comme précédemment et testez l'accès aux membres
- Créez une nouvelle classe `AudioBook` et implémentez les services associés
- Castez vos différents objets `Book`, `CD` et `AudioBook` avec les interfaces afin de tester le polymorphisme
- Notez que vous pouvez utiliser les méthodes `getClass()` de vos objets ainsi que l'instruction `instanceof`

Exécution d'un Programme JAVA (3^{ème} partie)

Les énumérations

- Il est possible de créer des valeurs de type constantes pour rendre le code plus lisible et maintenable, comme dans la plupart des langages de programmation
- En Java on utilise le mot-clé `enum`
- En Java l'enum est un type à part entière. Il est conçu sur le modèle des classes. Chaque valeur de l'enum est un objet à part entière. Par défaut il n'a pas de valeur bien spécifique, mais plutôt une référence qui sera unique dans le code

```
public enum EFruit
{
    APPLE,
    ORANGE,
    BANANA,
    STRAWBERRY,
}
```

- Un peu comme avec les attributs déclarés en `static`, on accèdera aux valeurs d'enum par :

```
EFruit.APPLE ou EFruit.BANANA
```

Les énumérations

- En Java, l'énumération est comme une classe qui hérite de la classe `Enum`
- Nous pouvons forcer la valeur de chaque entrée de l'enum en créant un constructeur privé pour définir la valeur, et une méthode publique pour la récupérer :

```
public enum EFruit
{
    APPLE("green"),
    ORANGE("orange"),
    BANANA("yellow")
    STRAWBERRY("red");
    private String value;
    private Toto(String param)
    {
        this.value = param;
    }
    public String getValue()
    {
        return this.value;
    }
}
```


Les énumérations

- La classe `Enum` possède des méthodes utiles pour la manipulation des énumérations telles que :

- `toString()` ← affiche le nom de l'enum sous forme de `String`

```
EFruit.APPLE.toString() ← retourne "APPLE"
```

- `valueOf()` ← méthode statique, retourne un enum à partir d'un objet `String`

```
EFruit.valueOf("BANANA") ← retourne EFruit.BANANA
```

- `values()` ← méthode statique, retourne un tableau des valeurs possibles

- `ordinal()` ← renvoi le numéro d'ordre de l'enum dans la liste

```
EFruit.STRAWBERRY.ordinal() ← retourne 3
```

```
EFruit.valueOf("STRAWBERRY").ordinal() ← retourne 3
```

- `CompareTo()` ← compare les numéros d'ordre de deux valeurs d'enum

```
EFruit.STRAWBERRY.compareTo( EFruit.ORANGE ) ← retourne +1
```

```
EFruit.APPLE.compareTo( EFruit.BANANA ) ← retourne -1
```

```
EFruit.APPLE.compareTo( EFruit.APPLE ) ← retourne 0
```

Les exceptions

- En langage Java, il est possible que le programme rencontre un problème d'exécution et renvoie un type d'erreur correspondant : c'est l'exception
- Il existe une classe Java `Exception`, qui peut-être héritée par d'autres classes qui précisent le type d'erreur. Voici quelques classes filles :

```
Exception           → classe mère des exception : type par défaut
NullPointerException → référence d'objet inexistante
ArithmeticException → problème arithmétique (ex: division par zéro)
FileNotFoundException → fichier non trouvé
```

- Il est possible de définir sa propre classe d'exception :

```
class myProcessException extends Exception
```

- Toutes les méthodes qui renvoient des exceptions particulières doivent l'indiquer dans la définition :

```
public int myMethod(double value) throws myProcessException
```

- Notez l'utilisation du mot-clé `throws`

Les exceptions

- Il est possible de forcer le lancement d'une exception :

```
...  
throw new Exception();  
...
```

- Il est possible de « capturer » les exceptions et donc de les traiter :

```
...  
try  
{  
    ...                ← exécution de notre portion de code  
}  
catch (NullPointerException npe)  
{  
    ...                ← traitement d'une exception de type pointeur nul  
}  
catch (ArithmeticException ae)  
{  
    ...                ← traitement d'une erreur arithmétique  
}  
...
```

Les exceptions

- Il est possible de « capturer » une exception pour en relancer une autre :

```
...  
try {  
    ...  
}  
catch (Exception e) {  
    ...  
    throw new MyOwnExceptionClass();  
}
```

← exécution de notre portion de code

← traitement de l'exception

← envoi d'une exception spécifique

- Pour des raisons de fonctionnement, il peut s'avérer utile d'effectuer un post-traitement, que l'exception soit levée ou non :

```
...  
catch {  
    ...  
}  
finally {  
    ...  
}
```

← exécution de notre portion de code

- Notez les mot-clés de gestion des exception **try...catch...finally**

Les tableaux

- Ils peuvent s'appliquer sur n'importe quel type d'attribut (ici un tableau de 5 entiers)

```
int[] myList1D = new int[5];
```

- On accède aux éléments du tableau par leur indice (de 0 à taille-1) :

```
myList1D[0]++;
```

```
myList1D[4]--;
```

```
myList1D[5]++; ← retourne une exception de type ArrayIndexOutOfBoundsException
```

- Les tableaux peuvent être à une ou plusieurs dimensions :

```
int[][] myList2D = new int[3][4];
```

- Tout comme les objets, le nom du tableau est une variable qui contient la référence vers les valeurs :

```
myList = myList2D[0]; ← affectation de référence valide
```

- L'accès aux éléments des tableaux est plus protégé que dans des langages comme le langage C par exemple
- Une fois déclarés, les tableaux ont une taille fixe, ne peuvent pas être réduits ou agrandis sans devoir faire une copie de tableau à tableau
- La taille d'un tableau se récupère grâce à son attribut `length`

Les listes

- Il est possible de créer des listes variables d'objets en utilisant la classe `ArrayList` et l'interface `List` :

```
List myList = new ArrayList() ;
```

- On peut utiliser les services d'ajout et de suppression pour modifier notre liste :

```
myList.add( 5 );  
myList.add( "Coucou" );  
myList.add( bookObject );  
myList.remove( 5 );
```

- La taille d'une liste est accessible par la méthode `size()`

```
int listLength = myList.size();
```

- Récupérer un objet de la liste se fait grâce à la méthode `get()`

```
String str = (String) (myList.get(2));
```

← on récupère l'objet à l'index 2

- Toutes les entrées de la liste sont de type `Object`, qui est la classe mère pour tous les objets en Java

Les listes

- Il est possible de définir une liste contenant seulement un certain type d'objet :

```
List<Book> myList = new ArrayList<Book>() ;
```

- Cette fois-ci, seuls les objets de type **Book** peuvent être ajoutés :

```
myList.add( 5 );           ← erreur à la compilation  
myList.add( "Coucou" );    ← erreur à la compilation  
myList.add( bookObject );
```

- Pour récupérer un objet, le type est déjà connu : pas besoin de cast

```
Book myBook = myList.get(1); ← on récupère l'objet à l'index 1
```

- Notez les chevrons **<** et **>** qui servent à indiquer le type d'objet contenu dans la liste

Les classes génériques

- Utile quand nous avons besoin d'utiliser des services sur n'importe quel type d'objet
- Le type d'objet n'est défini qu'au moment de l'instanciation

```
public class UniversalProcess<T>
{
    private T internalObjectRef;
    public UniversalProcess(T objectRef)
    {
        this.internalObjectRef = objectRef;
    }
    public T getObjectRef()
    {
        return this.internalObjectRef;
    }
    public void setObjectRef(T newRef)
    {
        this.internalObjectRef = newRef;
    }
}
```

- Notez le caractère T qui représente le type générique de notre classe

Les classes génériques

- Une fois la classe générique instanciée, le type d'objet qu'elle gère ne peut plus être modifié
- Après instanciation, si on passe le mauvais type d'objet dans la méthode `setObjectRef`, nous obtiendrons une exception

```
Book myBook1 = new Book();  
Book myBook2 = new Book();  
DVD myDVD1 = new DVD();  
UniversalProcess univProc = new UniversalProcess(myBook1);  
univProc.setObjectRef(myBook2);  
univProc.setObjectRef(myDVD1);           ← exception
```