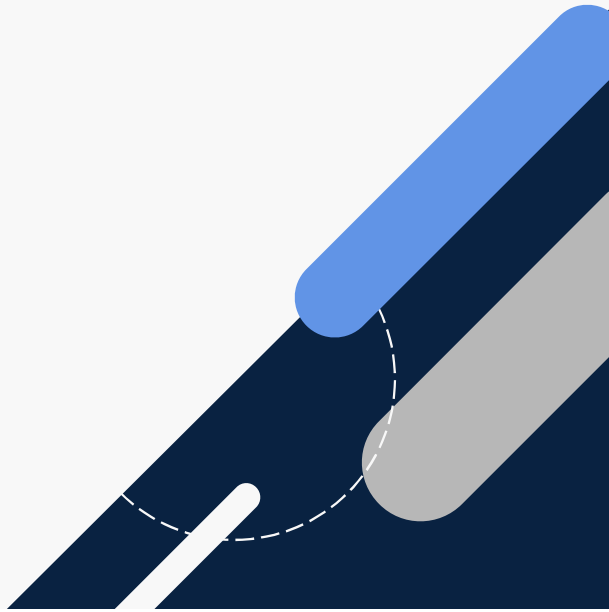
Pattern Singleton



Définition

Le Singleton est un patron de création qui garantit qu'une classe ne possède qu'une seule instance dans tout le système, et fournit un point d'accès global à cette instance.

Singleton = une seule instance + accès global.

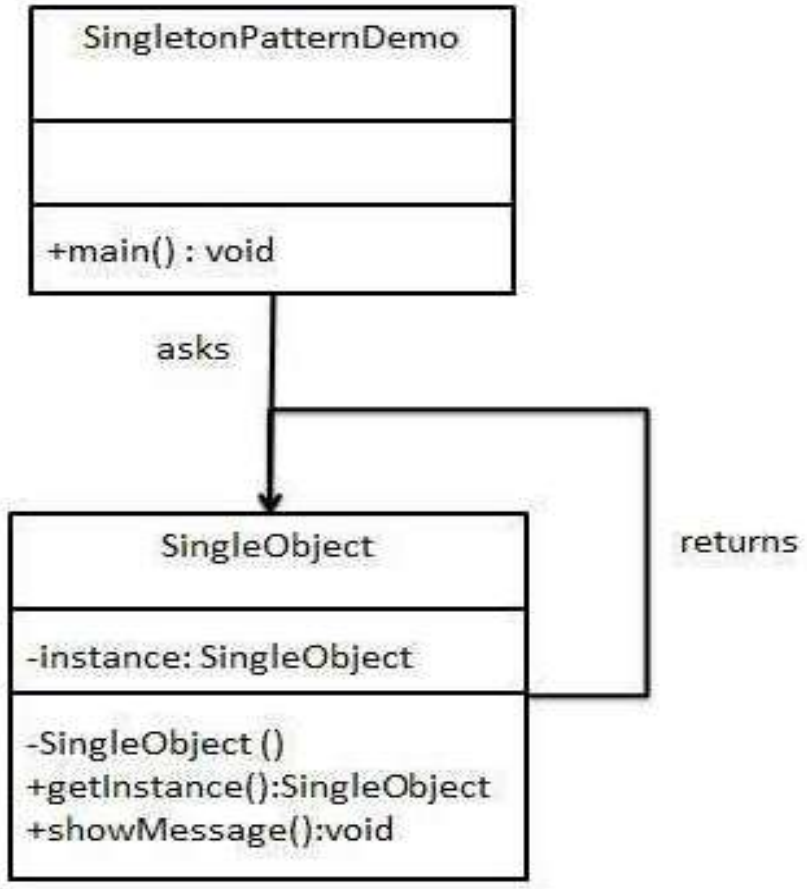
Très utile pour les ressources partagées.

Objectifs

1. Contrôler l'unicité : empêcher la création de plusieurs instances d'une même classe.
2. Fournir un point d'accès global : rendre l'instance accessible de manière centralisée.
3. Réduire la consommation mémoire : éviter la duplication inutile d'objets coûteux.
4. Assurer la cohérence : partager le même état dans tout le système.

Structure UML

- Etapes :
  - Rendre privé le constructeur,
  - Construire une instance privée de la classe comme attribut statique de la classe,
  - Fournir une méthode publique d'accès à cette instance.



Implémentation d’un Singleton

```
public class Singleton {
    // Instance unique (stockée en privé)
    private static Singleton instance;

    // Constructeur privé (empêche new Singleton())
    private Singleton() {}

    // Méthode d'accès global
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
```

```
public class Main {
    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();

    // Vérification : s1 et s2 pointent vers la même instance
        System.out.println(s1 == s2); // true
    }
}
```

Cas typiques d’utilisation

- Gestionnaire de configuration (fichier de configuration chargé une seule fois).
- Gestionnaire de connexion à une base de données (une seule connexion partagée).
- Logger (un seul journal d’événements utilisé dans toute l’application).
- Gestionnaire de threads ou de cache.

Exercice

Dans une application de gestion, plusieurs modules (facturation, clients, rapports) doivent accéder à la même base de données.

Pour des raisons de performance et de cohérence, il est interdit d'ouvrir plusieurs connexions. On souhaite donc créer une classe *DatabaseConnection* qui :

- garantit qu'une seule instance de connexion est utilisée dans tout le programme,
- permet d'ouvrir (connect()) et de fermer (disconnect()) la connexion,
- affiche des messages pour indiquer l'état de la connexion.

Exercice

Travail demandé

1. Implémentez la classe DatabaseConnection en appliquant le patron Singleton
2. Ajoutez deux méthodes :

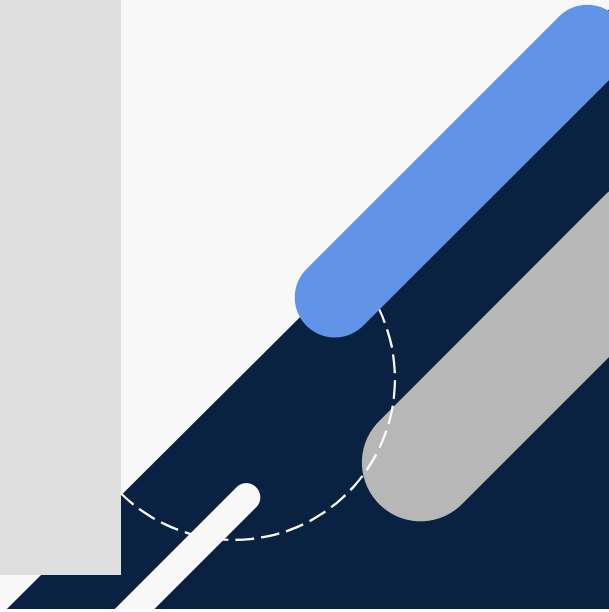
- connect() : affiche un message "Connexion à la base de données..." si aucune connexion n'est ouverte, sinon "Connexion déjà ouverte.«
- disconnect() : affiche un message "Déconnexion de la base de données..." si une connexion est ouverte, sinon "Aucune connexion active.«

3. Dans une classe Main, testez votre implémentation en :

- récupérant deux objets db1 et db2 via getInstance(),
- vérifiant qu'ils pointent vers la même instance,
- appelant successivement connect() et disconnect() sur db1 et db2.

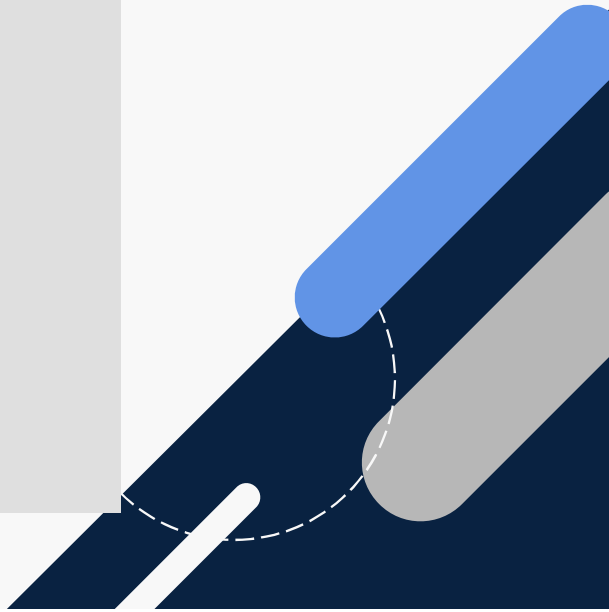
Solution

```
public class DatabaseConnection {
    // Instance unique
    private static DatabaseConnection instance;

    // Simule l'état de la connexion
    private boolean connected;

    // Constructeur privé
    private DatabaseConnection() {
        connected = false;
    }

    // Accès global à l'instance
    public static DatabaseConnection getInstance() {
        if (instance == null) {
            instance = new DatabaseConnection();
        }
        return instance;
    }
```



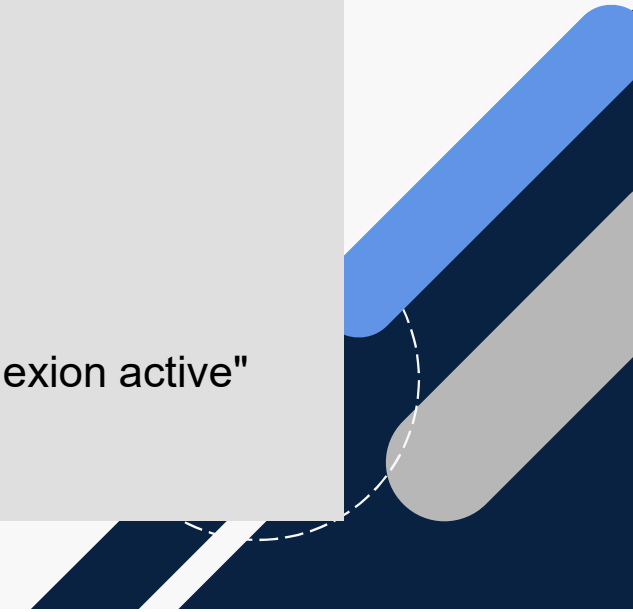
Solution

```
public void connect() {
        if (!connected) {
            System.out.println("Connexion à la base de données...");
            connected = true;
        } else {
            System.out.println("Connexion déjà ouverte.");
        }
    }

public void disconnect() {
        if (connected) {
            System.out.println("Déconnexion de la base de
données...");
            connected = false;
        } else {
            System.out.println("Aucune connexion active.");
        }
    }
```



Solution

```
public class Main {
    public static void main(String[] args) {
        DatabaseConnection db1 = DatabaseConnection.getInstance();
        DatabaseConnection db2 = DatabaseConnection.getInstance();

        // Vérification
        System.out.println(db1 == db2); // true

        // Utilisation
        db1.connect();
        db2.connect(); // même instance => message "déjà ouverte"
        db1.disconnect();
        db2.disconnect(); // même instance => message "aucune connexion active"
    }
}
```

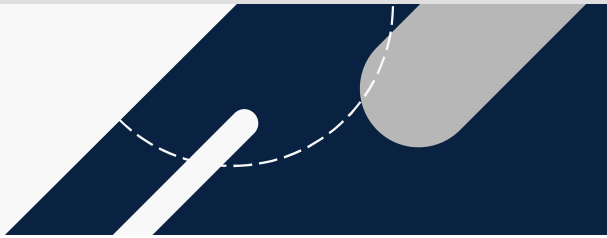


Exercice

Questions de réflexion:

1. Que se passe-t-il si deux threads différents appellent getInstance() en même temps avant que l'instance ne soit créée ?

Si plusieurs threads entrent simultanément dans getInstance() alors que instance == null, il est possible que deux instances différentes de DatabaseConnection soient créées.
Conséquence : cela viole le principe du Singleton (unicité), et peut causer des incohérences (accès concurrents non contrôlés à la base).



Exercice

Questions de réflexion:

2. Proposez une solution pour rendre votre Singleton **thread-safe**.

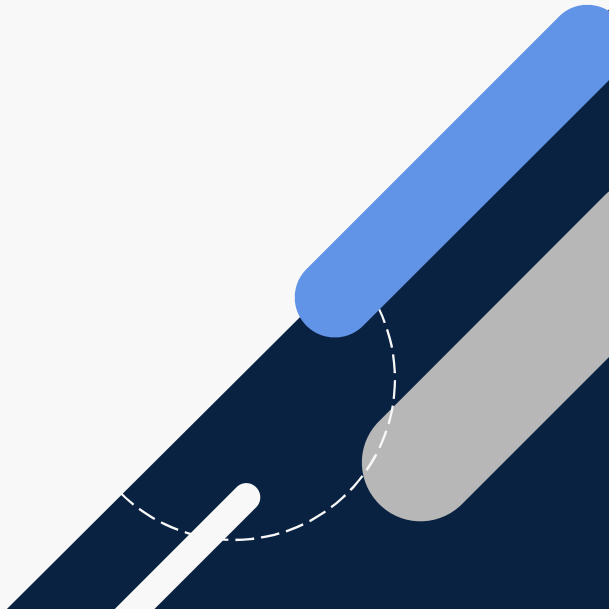
Synchroniser la méthode getInstance()

```
public static synchronized DatabaseConnection getInstance() {
 if (instance == null) {
 instance = new DatabaseConnection();
}   return instance;
}
```

Initialisation statique

```
private static final DatabaseConnection instance = new DatabaseConnection();
public static DatabaseConnection getInstance() {
return instance;}
```

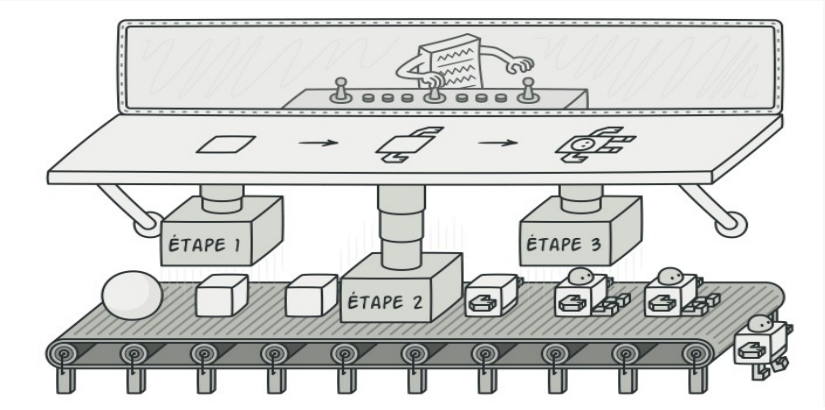
Pattern Builder



Définition

Le Builder est un patron de conception de création qui permet de construire des objets complexes étape par étape.

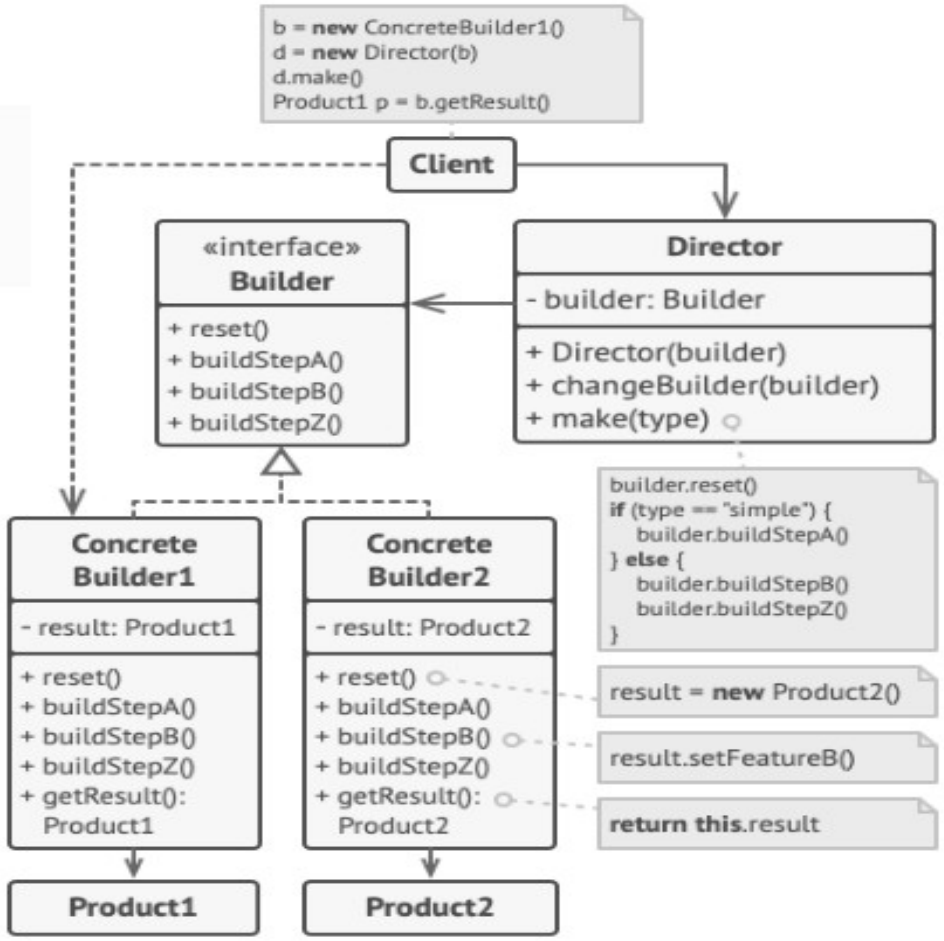
Il sépare la construction d'un objet de sa représentation finale, de sorte qu'un même processus de construction puisse créer différentes représentations.



Objectifs

1. Isoler la construction d'un objet complexe (ex. un objet avec de nombreux paramètres optionnels).
2. Éviter les constructeurs trop longs (avec trop d'arguments → problème du telescoping constructor).
3. Améliorer la lisibilité du code (approche fluide/chaînée).
4. Permettre de créer des variantes d'un même objet en réutilisant le même processus de construction.

Structure UML



1 L'interface du **Monteur** déclare les étapes communes de la construction du produit entre tous les monteurs.

2 Les **Monteurs Concrets** fournissent différentes implémentations des étapes de la construction. Ils peuvent créer des produits qui ne reprennent pas l'interface commune.

3 Les **Produits** sont les résultats retournés. Les produits construits par les différents monteurs ne sont pas obligés d'appartenir à la même hiérarchie de classes ni d'avoir la même interface.

4 Le **Directeur** indique l'ordonnancement des étapes de construction et offre la possibilité de créer et de réutiliser des configurations spécifiques pour les produits.

5 Le **Client** doit associer l'un des monteurs au directeur. En général cette association n'est réalisée qu'une seule fois, grâce aux paramètres du constructeur du directeur. Pour toute construction ultérieure, le directeur utilise l'objet monteur. En guise d'alternative, le client peut passer l'objet monteur à la méthode de production du directeur. Dans ce cas, vous pouvez utiliser un monteur différent chaque fois que vous lancez une production avec le directeur.

Conception sans Builder

Imaginons qu'on veuille créer un Burger sans Builder.

On utilise un constructeur avec beaucoup de paramètres.

```
public class Burger {
    private String bread;
    private String meat;
    private boolean cheese;
    private boolean salad;
    private boolean tomato;

    // Constructeur avec tous les paramètres
    public Burger(String bread, String meat, boolean
cheese, boolean salad, boolean tomato) {
        this.bread = bread;
        this.meat = meat;
        this.cheese = cheese;
        this.salad = salad;
        this.tomato = tomato;
    }

    @Override
    public String toString() {
        return "Burger [bread=" + bread + ", meat=" + meat +
                ", cheese=" + cheese + ", salad=" + salad + ",
tomato=" + tomato + "]";
    }
}
```

Conception sans Builder

Problème

Créons deux burgers.

```
public class Main {
    public static void main(String[] args) {
        Burger burger1 = new Burger("Pain complet", "Poulet", true, true, false);
        Burger burger2 = new Burger("Pain brioché", "Boeuf", false, true, true);

        System.out.println(burger1);
        System.out.println(burger2);
    } }
```

Inconvénients

1. Lisibilité faible: Quand on lit new Burger("Pain complet", "Poulet", true, true, false), on ne sait pas directement à quoi correspondent les true et false.
2. Risque d'erreurs: Si on se trompe dans l'ordre des paramètres, la compilation ne détectera rien (ex. inverser salad et tomato).
3. Scalabilité limitée: Si on ajoute un nouvel ingrédient (ex. sauce), il faut modifier le constructeur et donc casser du code existant.

Conception avec Builder

Ancien code :

```
new Burger("Pain complet", "Poulet", true, true, false);
```

Nouveau code : Solution avec builder

```
new Burger.Builder()
   .bread("Pain complet")
   .meat("Poulet")
   .cheese(true)
   .salad(true)
   .tomato(false)
   .build();
```

C’est beaucoup plus lisible, flexible et sûr.

Conception avec Builder

Code complet:

```
public class Burger {
   private String bread;
   private String meat;
   private boolean cheese;
   private boolean salad;
   private boolean tomato;

   // Constructeur privé : seul le Builder peut créer un Burger
   private Burger(Builder builder) {
      this.bread = builder.bread;
      this.meat = builder.meat;
      this.cheese = builder.cheese;
      this.salad = builder.salad;
      this.tomato = builder.tomato;
   }

   @Override
   public String toString() {
      return "Burger [bread=" + bread + ", meat=" + meat +
            ", cheese=" + cheese + ", salad=" + salad + ", tomato=" + tomato +
"]";
   }
```

```
// Classe interne statique : le Builder
  public static class Builder {
    private String bread;
    private String meat;
    private boolean cheese;
    private boolean salad;
    private boolean tomato;

    public Builder bread(String bread) {
      this.bread = bread;
      return this;
    }

    public Builder meat(String meat) {
      this.meat = meat;
      return this;
    }

    public Builder cheese(boolean cheese) {
      this.cheese = cheese;
      return this;
    }

    public Builder salad(boolean salad) {
      this.salad = salad;
      return this;
    }

    public Builder tomato(boolean tomato) {
      this.tomato = tomato;
      return this;
    }

    // Méthode de construction finale
    public Burger build() {
      return new Burger(this);
    }
  }
}
```

Conception avec Builder

Test Builder:

```
public class Main {
  public static void main(String[] args) {
    // Création fluide et lisible
    Burger burger1 = new Burger.Builder()
                .bread("Pain complet")
                .meat("Poulet")
                .cheese(true)
                .salad(true)
                .tomato(false)
                .build();

    Burger burger2 = new Burger.Builder()
                .bread("Pain brioché")
                .meat("Boeuf")
                .cheese(false)
                .salad(true)
                .tomato(true)
                .build();

    System.out.println(burger1);
    System.out.println(burger2);
  }
}
```

Application de Builder avec Director

- Builder (interface/abstraite) : définit les étapes de construction.
- ConcreteBuilder : implémente les étapes et construit le produit final.
- Product : l'objet complexe à construire.
- Director : orchestre l'ordre de construction en appelant les méthodes du Builder.
- Client : demande la construction au Director et récupère le produit.

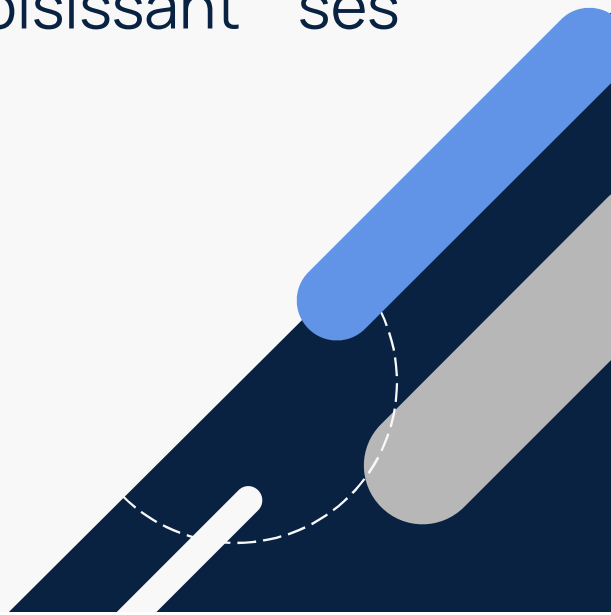
Exercice 1 : Application de Builder avec Director

Enoncé:

Vous devez modéliser un système de commande de pizzas personnalisées. Une Pizza doit être composée de certains éléments obligatoires et d'autres optionnels. Chaque client doit pouvoir créer facilement une pizza en choisissant ses ingrédients.

Une Pizza peut contenir les éléments suivants :

- Pâte (dough) → obligatoire
- Sauce → obligatoire
- Fromage (cheese) → optionne
- Garniture (topping) → optionnel



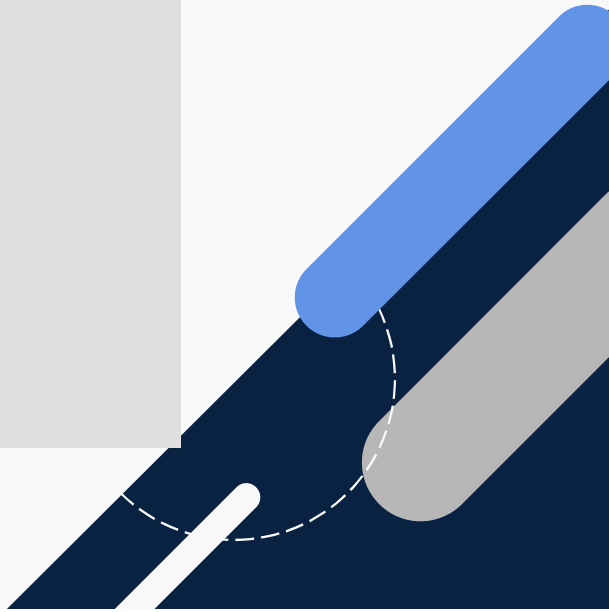
Exercice 1 : Application de Builder avec Director

Travail demandé :

Ecrire un programme qui permet de construire trois builders correspondants aux trois types de pizzas suivants:

- Margherita : pâte fine, sauce tomate, fromage mozzarella.
- Pepperoni : pâte épaisse, sauce tomate, fromage cheddar, garniture pepperoni.
- Végétarienne : pâte complète, sauce pesto, fromage de chèvre, garniture légumes.

```
public class Pizza {
    private String dough;
    private String sauce;
    private String cheese;
    private String topping;

    public void setDough(String dough) { this.dough = dough; }
    public void setSauce(String sauce) { this.sauce = sauce; }
    public void setCheese(String cheese) { this.cheese = cheese; }
    public void setTopping(String topping) { this.topping = topping; }

    @Override
    public String toString() {
        return "Pizza:\n" +
                "Dough: " + dough + "\n" +
                "Sauce: " + sauce + "\n" +
                "Cheese: " + (cheese != null ? cheese : "None") + "\n" +
                "Topping: " + (topping != null ? topping : "None");
    } }
```

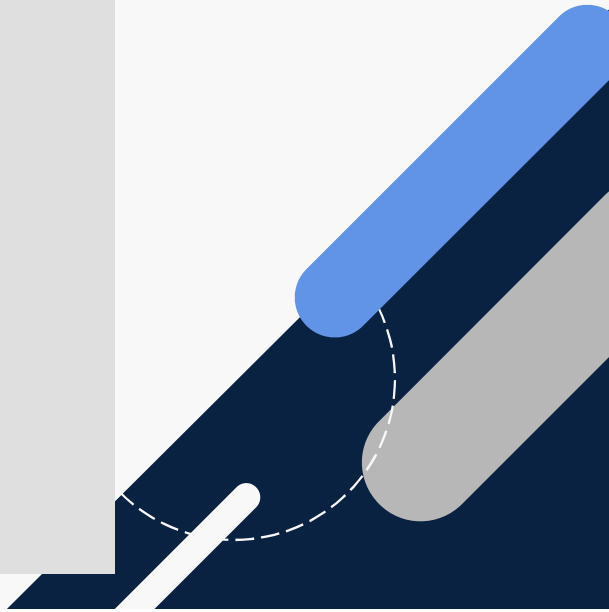


```
public interface PizzaBuilder {
    void buildDough();
    void buildSauce();
    void buildCheese();
    void buildTopping();
    Pizza getPizza();
}
```

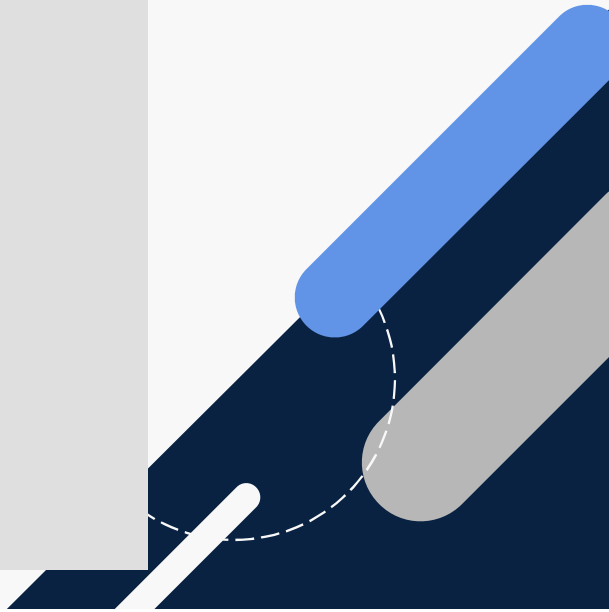
```
// Margherita Builder
public class MargheritaBuilder implements PizzaBuilder {
    private Pizza pizza = new Pizza();

    public void buildDough() { pizza.setDough("Thin Crust"); }
    public void buildSauce() { pizza.setSauce("Tomato"); }
    public void buildCheese() { pizza.setCheese("Mozzarella"); }
    public void buildTopping() { /* pas de topping */ }
    public Pizza getPizza() { return pizza; }
}
```

```
// Pepperoni Builder
public class PepperoniBuilder implements PizzaBuilder {
    private Pizza pizza = new Pizza();
    public void buildDough() { pizza.setDough("Thick Crust"); }
    public void buildSauce() { pizza.setSauce("Tomato"); }
    public void buildCheese() { pizza.setCheese("Cheddar"); }
    public void buildTopping() { pizza.setTopping("Pepperoni"); }
    public Pizza getPizza() {
// processus de création
this.buildDough(); this.buildSauce(); this.buildCheese();
        this.buildTopping();
return pizza; } }
```

```
public class PizzaDirector {
    private PizzaBuilder builder;

    public PizzaDirector(PizzaBuilder builder) {
        this.builder = builder;
    }
   /* on peut définir ce processus de création dans la classe Builder*/
   /* Méthode optionnelle */
    public void constructPizza() {
        builder.buildDough();
        builder.buildSauce();
        builder.buildCheese();
        builder.buildTopping();
    }

    public Pizza getPizza() {
        return builder.getPizza();
    }
}
```



```
public class Main {
    public static void main(String[] args) {
        // Pizza Margherita
        PizzaBuilder margheritaBuilder = new MargheritaBuilder();
        PizzaDirector director1 = new PizzaDirector(margheritaBuilder);
        director1.constructPizza();
        Pizza margherita = director1.getPizza();

        // Pizza Pepperoni
        PizzaBuilder pepperoniBuilder = new PepperoniBuilder();
        PizzaDirector director2 = new PizzaDirector(pepperoniBuilder);
        director2.constructPizza();
        Pizza pepperoni = director2.getPizza();

        // Affichage
        System.out.println("=== Margherita ===");
        System.out.println(margherita);

        System.out.println("\n=== Pepperoni ===");
        System.out.println(pepperoni);
    } }
```



Exercice 2 : Application de Builder sans Director

Vous devez modéliser un système qui permet de configurer un ordinateur sur mesure (PC).

Un ordinateur peut contenir plusieurs éléments :

- Processeur (obligatoire)
- Mémoire RAM (obligatoire)
- Disque dur (optionnel)
- Carte graphique (optionnel)
- Carte mère (obligatoire)
- Alimentation (optionnel)

Exercice 2 : Application de Builder sans Director

Travail demandé:

1. Créez une classe Computer avec les attributs ci-dessus.

- Les champs doivent être privés.
- Le constructeur de Computer doit être privé, afin de forcer l'utilisation du Builder.

2. Ajoutez une classe interne statique Builder qui contient:

- Des attributs identiques à ceux de Computer.
- Des méthodes chaînées (processor(String p), ram(int r), disk(String d), etc.) pour initialiser les champs.
- Une méthode build() qui retourne l'objet final Computer.

Exercice 2 : Application de Builder sans Director

Travail demandé:

3. Dans une classe Main, créez deux configurations d'ordinateurs :

- Un PC Gamer (processeur puissant, 16 Go RAM, carte graphique, disque SSD).
- Un PC Bureau (processeur moyen, 8 Go RAM, sans carte graphique, disque dur classique).