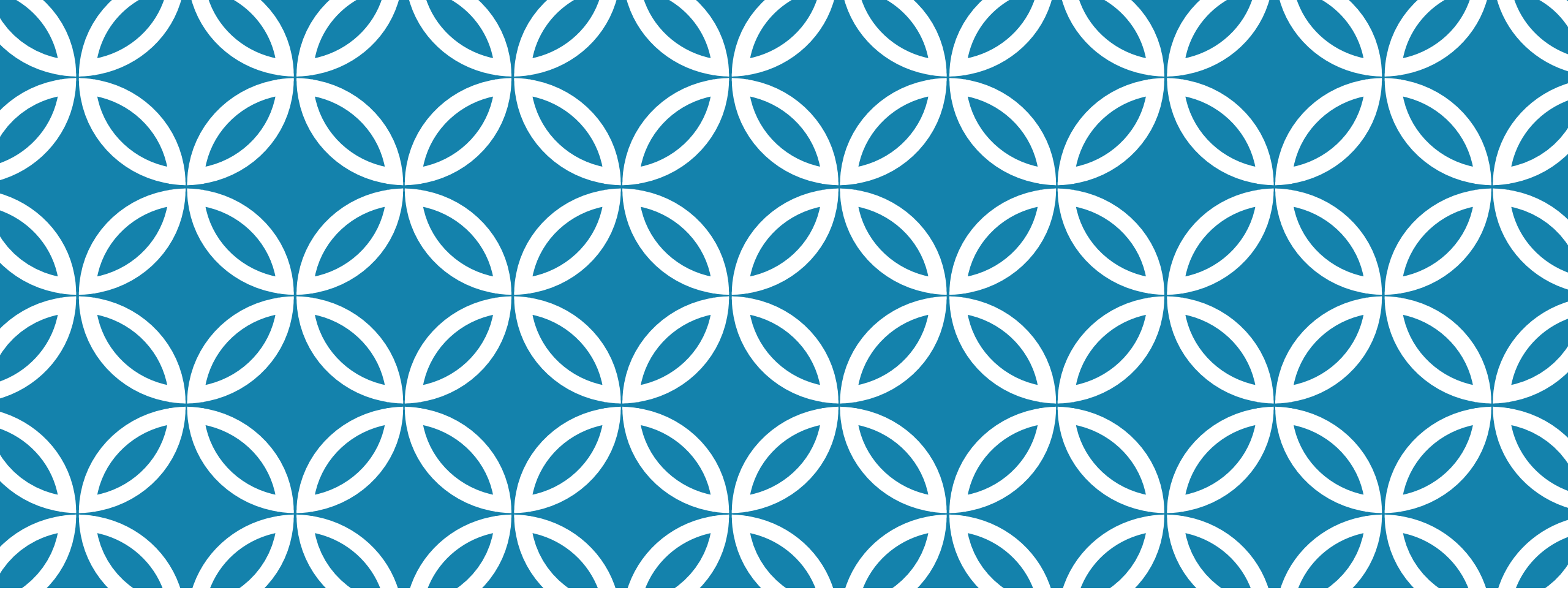
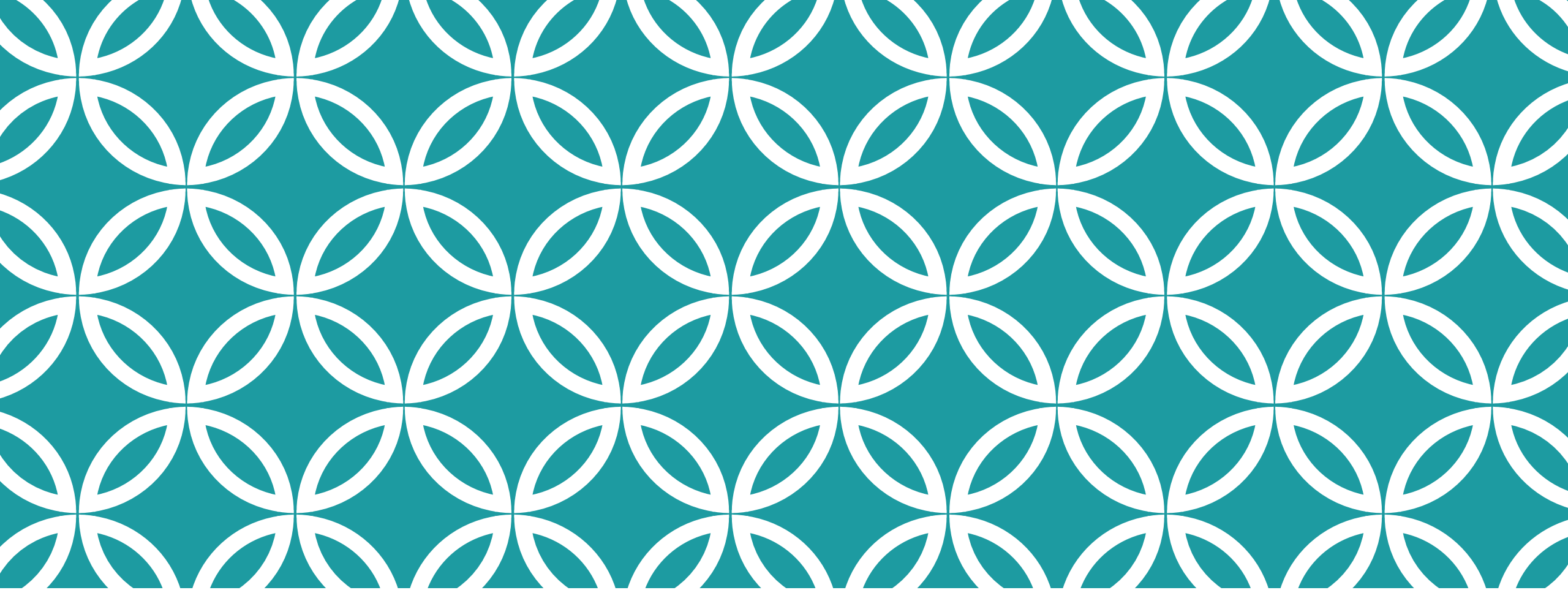


PROGRAMMATION JAVA AVANCÉ

Jérémy PERROUAULT



GÉNÉRICITÉ

Types génériques et templates

TYPE GÉNÉRIQUE

Classe qui permet d'utiliser un type d'objet générique

- Ne connait pas le type d'objet qu'on va utiliser
- Permet de définir les signatures

Définir classe "Classe<T>"

Définir méthode "void utiliser(T param)"

Définir classe "Classe<T, Id>"

Définir méthode "Id utiliser(T param)"

TYPE GÉNÉRIQUE

C’est le cas notamment des listes qu’on utilise de la manière suivante

- ArrayList<Chat> mesChats = new ArrayList<Chat>();

- mesChats.add(new Chat());

TYPE GÉNÉRIQUE

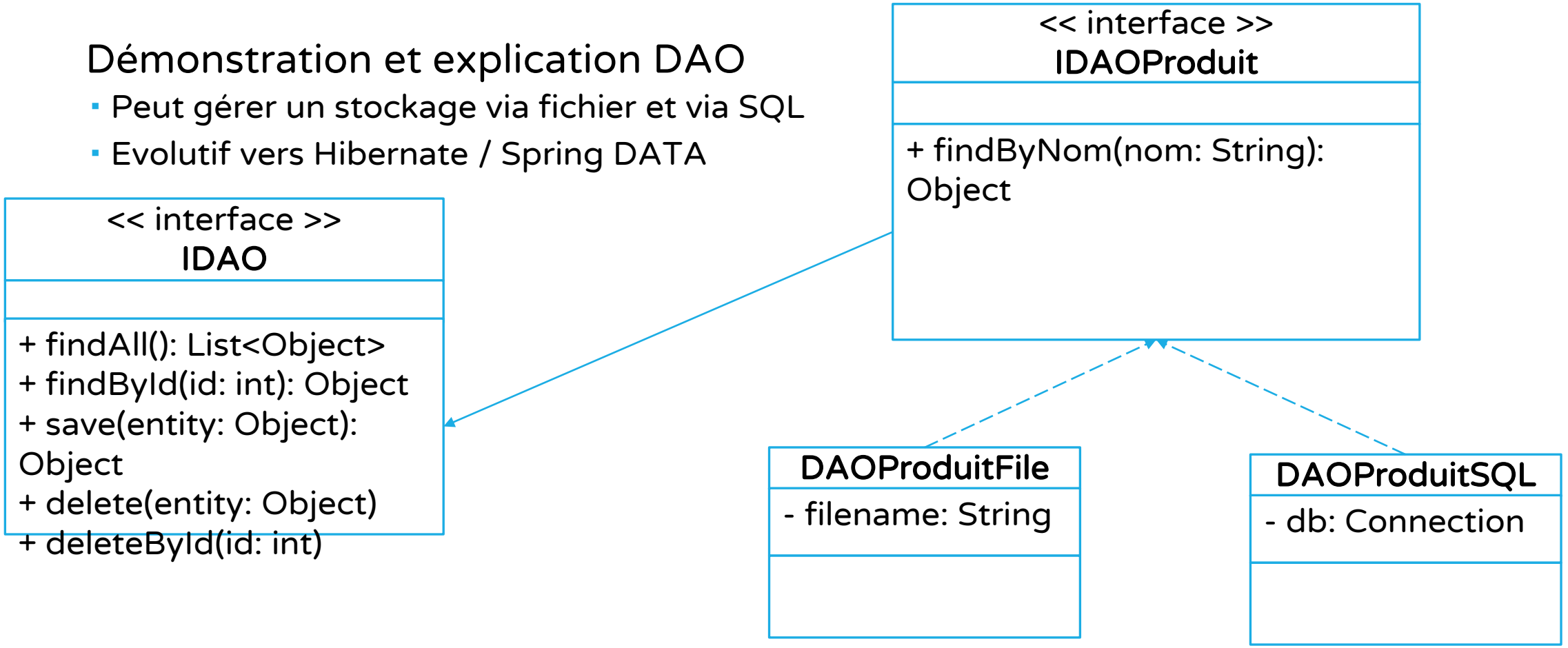
Démonstration classe soigneur générique

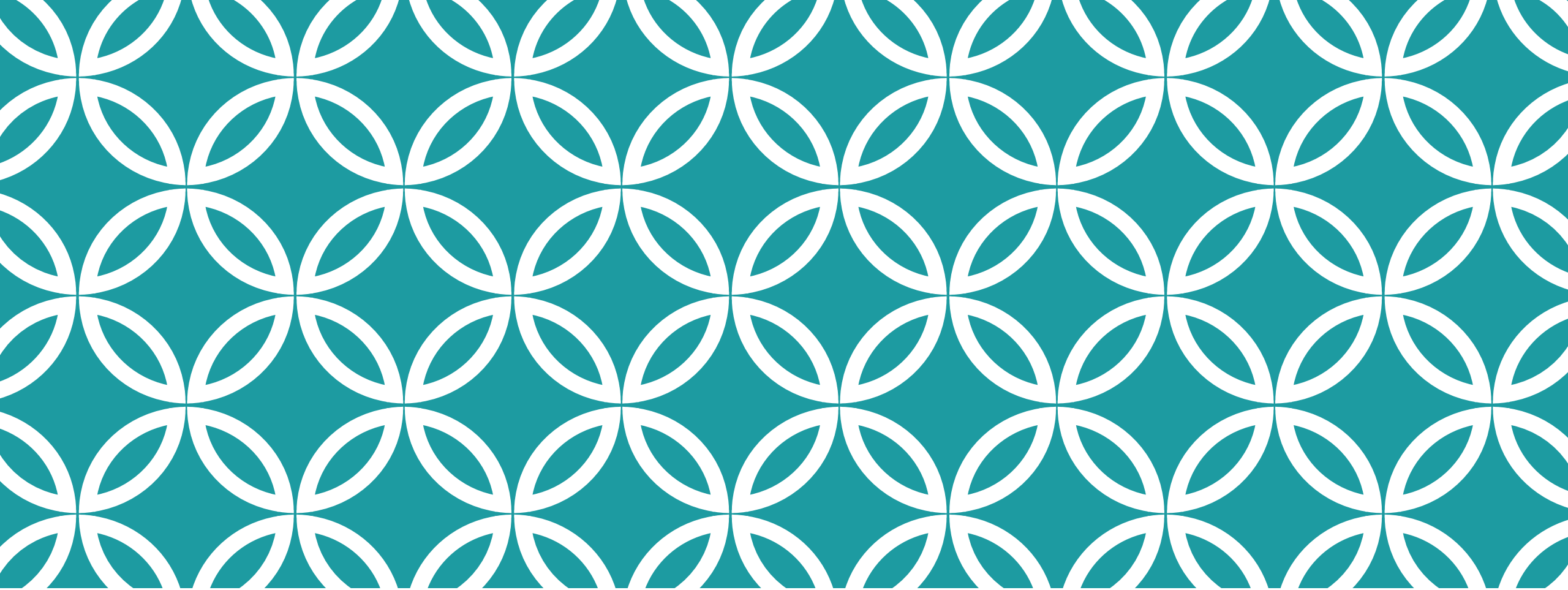
- Avec Object
- Avec généricité
- Avec interfaces

EXERCICE

Démonstration et explication DAO

- Peut gérer un stockage via fichier et via SQL
- Evolutif vers Hibernate / Spring DATA





ELÉMENTS DE BASE

Les types

ELÉMENTS DE BASE

Les classes mises à disposition par Java se trouvent dans des packages particuliers

- Pour pouvoir manipuler ces classes, il faut **importer** le ou les packages concernés
- D'une manière générale, si la classe n'est pas accessible MAIS qu'elle existe belle et bien
  - Eclipse vous propose de l'importer pour la manipuler

```
import java.util.ArrayList;

import java.util.*;
```

ELÉMENTS DE BASE

On peut généraliser et faire des imports statiques

- Permet ainsi de ne pas réécrire l'instruction complète, mais simplement utiliser « out »

```
import static java.lang.System.out;
```

- Fonctionne sur les variables et sur les méthodes statiques

LES ANNOTATIONS

Les annotations sont à positionner selon les cas

- Sur une classe
- Sur un attribut
- Sur une méthode

Elles sont préfixées de « @ »

- @Autowired
- @Override

Elles confèrent un comportement

- A la lecture du code
- A la compilation du code
- A l'exécution du code

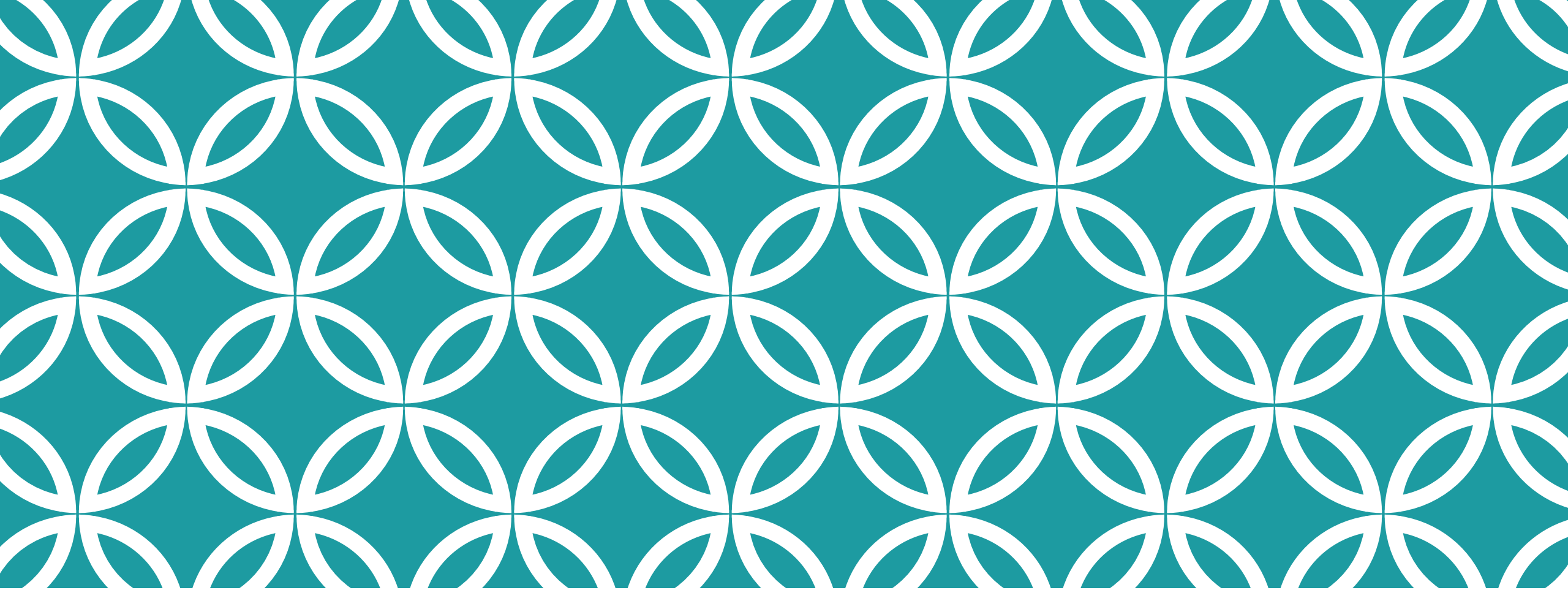
DES ARGUMENTS VARIABLES ?

Arguments variables en dernière position

```
public void multiArgs(String arg1, int... nombres) {
  //nombres est en fait un int[] (Array) !
}
```

```
multiArgs("test", 1, 2, 3);
```

A partir de Java 1.5, c'est le cas de System.out.printf() !



LES ÉNUMÉRATEURS

Enumérations

LES ÉNUMÉRATEURS

Sans les énumérateurs

```
public class Personne {
  private static int TYPE_PERSONNE_CLIENT = 1;
  private static int TYPE_PERSONNE_FOURNISSEUR = 2;
}
```

```
public void testType(int type) {
  if (type == TYPE_PERSONNE_CLIENT) {
    //...
  }
}
```

Avec les énumérateurs

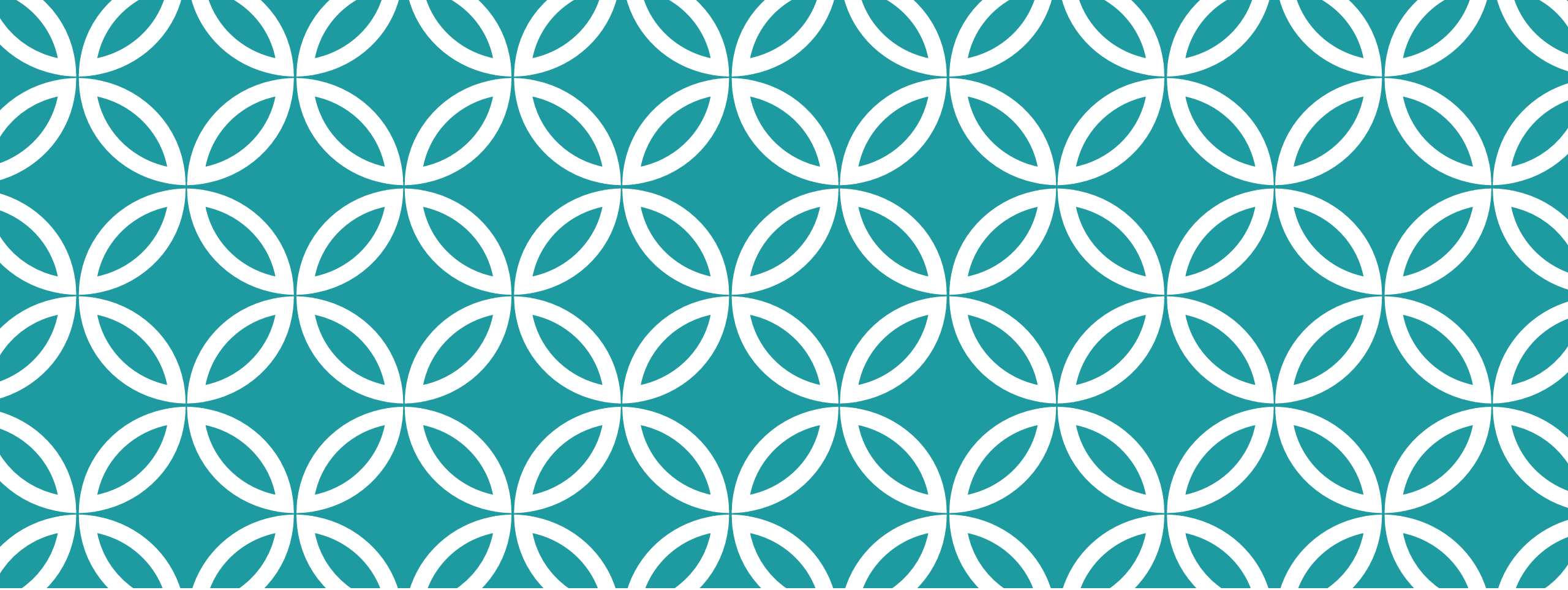
```
public enum TypePersonne {
  Client,
  Fournisseur;
}
```

```
public void testType(typePersonne type) {
  if (type == typePersonne.Client) {
    //...
  }
}
```

LES ÉNUMÉRATEURS

```
public enum TypePersonne {
  Client(1),
  Fournisseur(2);

  private int valeur = 0;

  TypePersonne(int valeur) {
    this.valeur = valeur;
  }

  public int getValeur() {
    return this.valeur;
  }
}
```

```
public void testType(int type) {
  if (type == TypePersonne.valueOf("Fournisseur").getValeur()) {
    //...
  }
}
```



LES EXCEPTIONS

Déclencher et gérer des erreurs

LES EXCEPTIONS

Permet de contrôler les erreurs sans interrompre le programme

- try … catch ! And finally …

Classe **Exception**

Possibilité de lancer une exception

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

LES EXCEPTIONS

Comment créer une Exception spécialisée ?

LES EXCEPTIONS

Possibilité d'attraper plusieurs exceptions et gérer les cas

- try { … }
- catch (MonException mex) { … }
- catch (Exception ex) { … }

```
try {
  //...
}
catch (MonException mex) {
  //...
}
catch (Exception ex) {
  //...
}
```

Possibilité d'exécuter un code que l'exception soit levée ou nc

- finally { … }
- S'exécute même si l'instruction « return » est présente dans le try ou dans

```
try {
  //...
}
catch (Exception ex) {
  //...
}
finally {
  //...
}
```

LES EXCEPTIONS

En JAVA, les méthodes qui – potentiellement – déclenchent une **Exception**

- Doivent signer qu'elles le font avec le mot-clé "throws"

```
public void uneMethode() throws Exception {
  throw new Exception();
}
```

```
public int uneAutreMethode() throws UneException, UneAutreException {
  //...
}
```

EXERCICE

Dans un programme principal

- Afficher « saisir un chiffre »
- Attendre la saisie d'un chiffre avec *Scanner.nextInt()*
- Saisir une lettre
  - L'application va crasher
- Faire en sorte que l'application continue de s'exécuter
  - Demander à l'utilisateur la saisie d'un chiffre, jusqu'à ce que ce soit OK !
  - Utiliser *Scanner.next()* dans le catch pour réinitialiser la mauvaise saisie

EXERCICE

Créer une classe de lecture de chiffre **LireChiffre**

- Avec une méthode *positif()*

Créer une exception **LireChiffreFormatException**

Créer une exception **LireChiffreNegatifException**

Le programme principal fait appel à **LireChiffre**

- Si la saisie est incorrecte, **LireChiffre** lève une exception **LireChiffreFormatException**
- Si le chiffre est négatif, **LireChiffre** lève une exception **LireChiffreNegatifException**
- Le programme principal intercepte les 2 exceptions
  - Il a un comportement différent selon le type d'exception

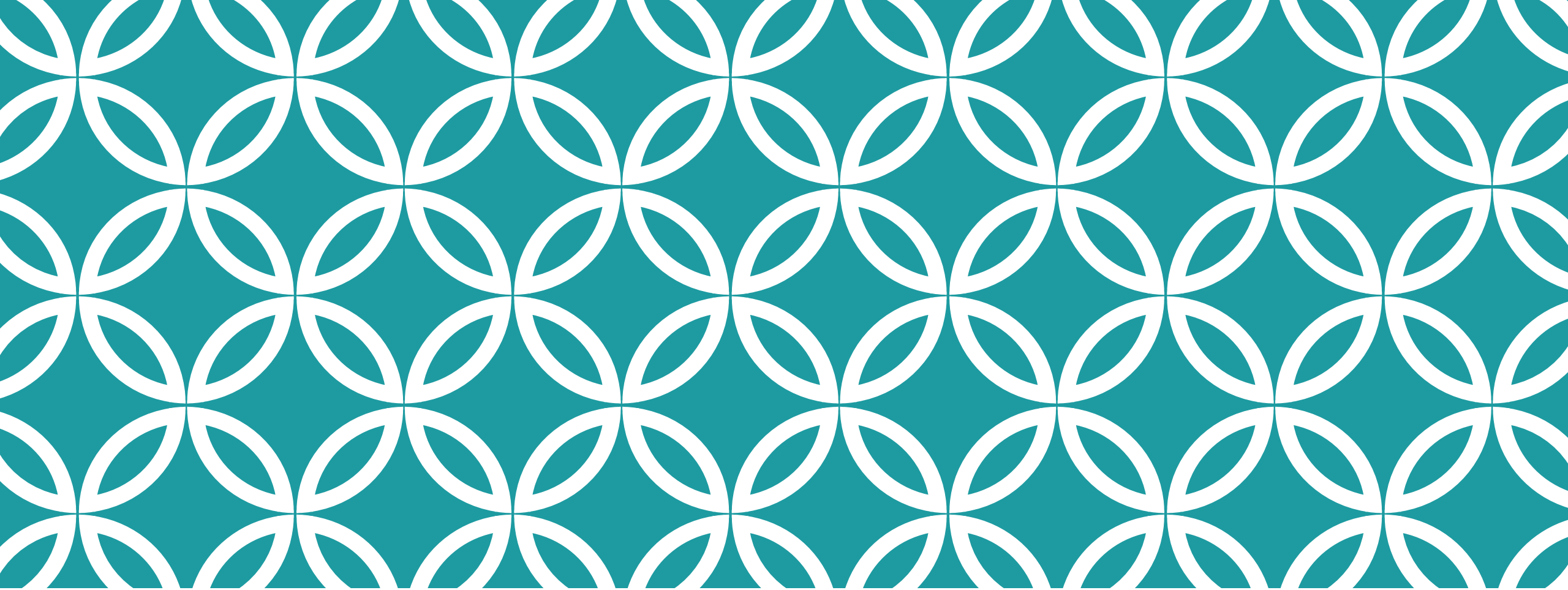
LES EXCEPTIONS – LES FLUX

Un flux ouvert devrait toujours être fermé après son utilisation

- Dans le deuxième exemple, le bloc try catch ferme automatiquement le flux

```
Scanner sc = new Scanner(System.in);

try {
  //...
}

catch (Exception e) {
  //...
}

finally {
  sc.close();
}
```

```
try (Scanner sc = new Scanner(System.in)) {
  //...
}

catch (Exception e) {
  //...
}
```



LES BIBLIOTHÈQUES

Ecrire un code réutilisable

LES BIBLIOTHÈQUES

Tout projet Java peut être exporté en JAR (Java **AR**chive)

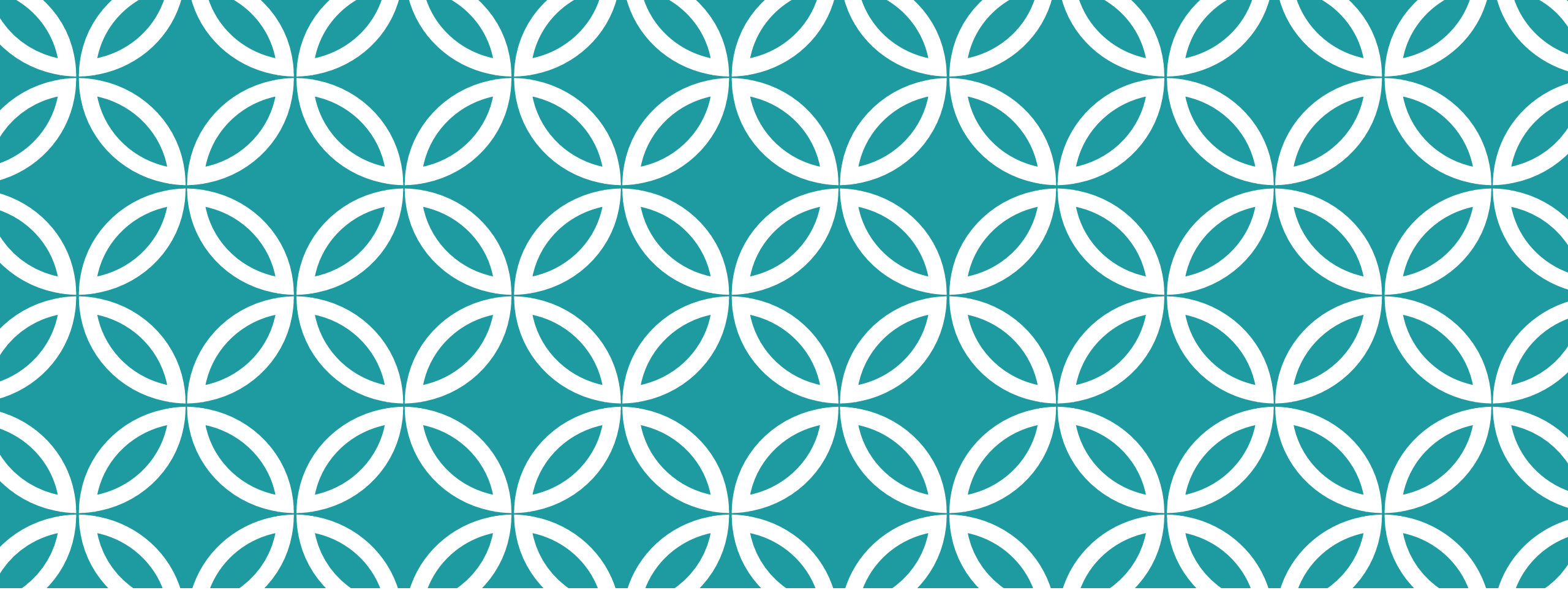
Tout JAR peut être implémenté dans un projet Java

- Ce JAR devient une bibliothèque (une dépendance) vis-à-vis du projet qui l'inclue et qui l'utilise !

LES BIBLIOTHÈQUES

Démonstration

- Création d'un nouveau projet Java par un volontaire
- Création d'une classe **Personne** (nom, prénom, age)
- Exportation du projet en JAR
- Intégration de ce JAR dans un autre projet



FICHIER

Ecrire et lire un fichier

LIRE UN FICHIER

Utilisation des classes

- **FileReader** (java.io)
- **BufferedReader** (java.io)

Principe

- Ouverture du fichier avec **FileReader**
- Lecture du fichier ouvert avec **BufferedReader**
- Parcours des lignes avec la méthode *readLine()*
- Fermeture du lecteur
- Fermeture du fichier

```
FileReader myFileReader = null;
BufferedReader myBufferedReader = null;
String myLigne;

try {
  myFileReader = new FileReader("chemin-du-fichier.txt");
  myBufferedReader = new BufferedReader(myFileReader);

  while ((myLigne = myBufferedReader.readLine()) != null) {
    System.out.println(myLigne);
  }
}
catch (Exception e) {
  e.printStackTrace();
}
finally {
  try {
    myBufferedReader.close();
    myFileReader.close();
  }
  catch (IOException e) {
    e.printStackTrace();
  }
}
```

EXERCICE

Créer un fichier TXT avec quelques lignes

Lire ce fichier via un programme JAVA

ECRIRE DANS UN FICHIER

Utilisation des classes

- **FileWriter** (java.io)
- **BufferedWriter** (java.io)

Principe

- Ouverture du fichier avec **FileWriter**
- Lecture du fichier ouvert avec **BufferedWriter**
- Ecriture d'une information avec la méthode *write()*
- Passage à la ligne avec la méthode *newLigne()*
- Fermeture de l'écriveur
- Fermeture du fichier

```
FileWriter myFileWriter = null;
BufferedWriter myBufferedWriter = null;

try {
  myFileWriter = new FileWriter("chemin-du-fichier.txt");
  myBufferedWriter = new BufferedWriter(myFileWriter);

  myBufferedWriter.write("Information ...");
  myBufferedWriter.newLine();
  myBufferedWriter.write("Autre information !");
}
catch (Exception e) {
  e.printStackTrace();
}
finally {
  try {
    myBufferedWriter.close();
    myFileWriter.close();
  }
  catch (IOException e) {
    e.printStackTrace();
  }
}
```

EXERCICE

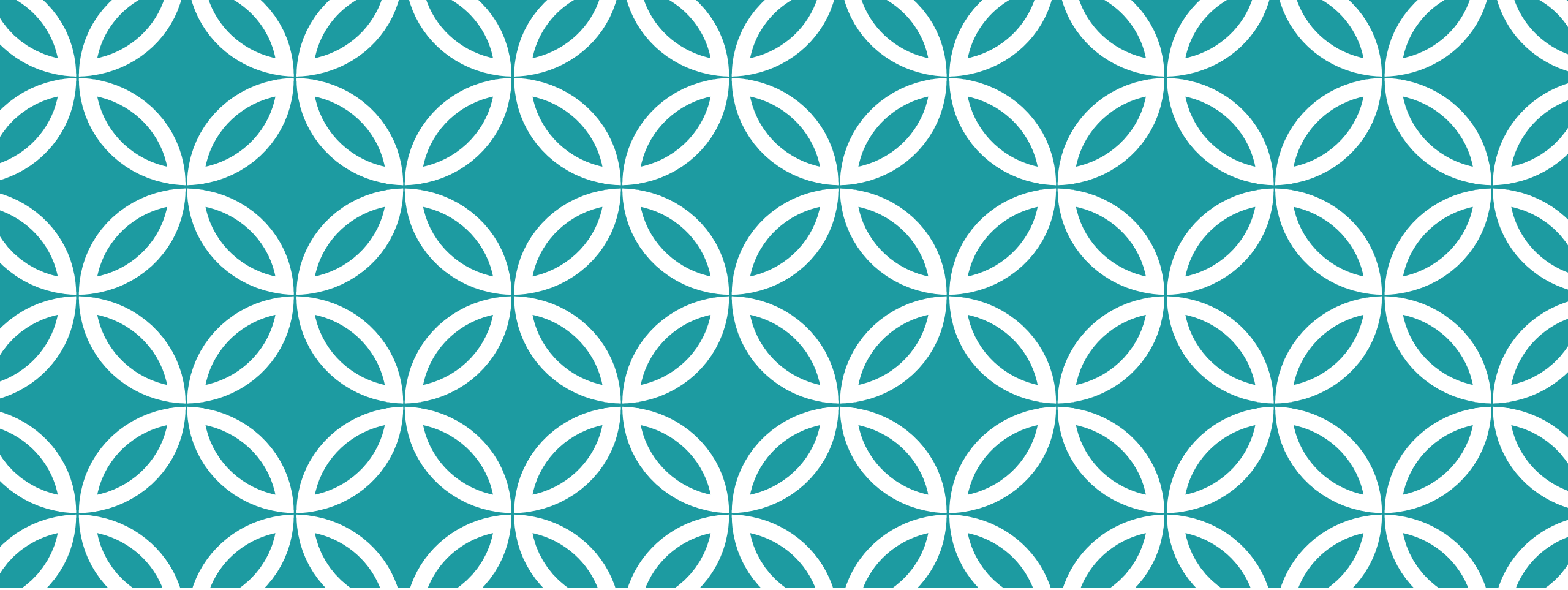
Ecrire un programme JAVA qui insère des lignes dans un fichier

EXERCICE

Reprendre l'exercice du site e-commerce

Créer les méthodes de lecture et d'écriture des données dans un fichier

- Il faut être capable de liste les informations
- Il faut être capable d'ajouter une nouvelle donnée



BASE DE DONNÉES

Ecrire et lire une base de données

BASE DE DONNÉES – ORM

Object Relational Mapping

Classes métier représentent une projection de la base de données

Modèle objet	Modèle relationnel
Graphe d'objets	Base de données relationnelle
Instances de classes	Enregistrements dans une table
Références	Relations (FK → PK)
« Clé primaire » optionnelle	
Héritage	

BASE DE DONNÉES – ORM

Exemple de mapping

Modèle objet (une classe)	Modèle relationnel (une table)
Personne.java	Personne
int id	PER_ID
String nom	PER_NOM
String prenom	PER_PRENOM
String adresse	PER_ADRESSE

BASE DE DONNÉES – ACCÈS

L'accès le plus bas niveau avec Java

- Driver JDBC adapté au serveur SQL manipulé
- Implémentation de la bibliothèque « connecteur MySQL »

```
try {
  Class.forName("com.mysql.jdbc.Driver");
}
catch (ClassNotFoundException e) {
  //...
}
```

On doit charger ce driver adapté

- La classe doit être présente dans le *classpath*

Pour s'y connecter

- Il faut connaître l'URL de connexion

```
try {
  Connection myConnection = DriverManager.getConnection("jdbc:mysql://localhost:3306/nom_base?serverTimezone=UTC", "username", "password");
}

catch (SQLException e) {
  //...
}
```

BASE DE DONNÉES – ACCÈS

Exécuter des requêtes

- Création d'un Statement
- Récupération du résultat dans un ResultSet avec la méthode *executeQuery()*
  - Il existe aussi *execute()* et *executeUpdate()*

```
try {
  Statement myStatement = myConnection.createStatement();
  ResultSet myResult = myStatement.executeQuery("SELECT PER_ID, PER_NOM, PER_PRENOM, PER_ADRESSE FROM personne");

  while (myResult.next()) {
    System.out.println(myResult.getString("PER_NOM"));
    //...
  }
}

catch (SQLException e) {
  //...
}
```

BASE DE DONNÉES – ACCÈS

Exécuter des requêtes

- Création d'un Statement préparé
- Récupération du résultat dans un ResultSet avec la méthode *executeQuery()*
  - Il existe aussi *execute()* et *executeUpdate()*

```
try {
  PreparedStatement myStatement =
    myConnection.prepareStatement("INSERT INTO produit (PRO_NOM, PRO_PRIX, PRO_FOURNISSEUR_ID) VALUES (?, ?, 3)");

  myStatement.setString(1, "GoPRO HERO 6");
  myStatement.setFloat(2, 499.99f);

  myStatement.execute();
}

catch (SQLException e) {
  //...
}
```

BASE DE DONNÉES – ACCÈS

Contrôler les transactions

- *Les instructions de transaction sont accessibles en Java*

```
//On désactive l'auto-commit
myConnection.setAutoCommit(false);

//On joue la transaction
Statement myStatement = myConnection.createStatement();
//...

//On valide la transaction
myConnection.commit();

//On aurait pu l'annuler avec 'myConnection.rollback()'
```

BASE DE DONNÉES – ACCÈS

Pour aller plus loin, vous pouvez consulter la documentation officielle

- *https://docs.oracle.com/javase/tutorial/jdbc/basics/*

BASE DE DONNÉES – EXERCICE

Dans un programme principal

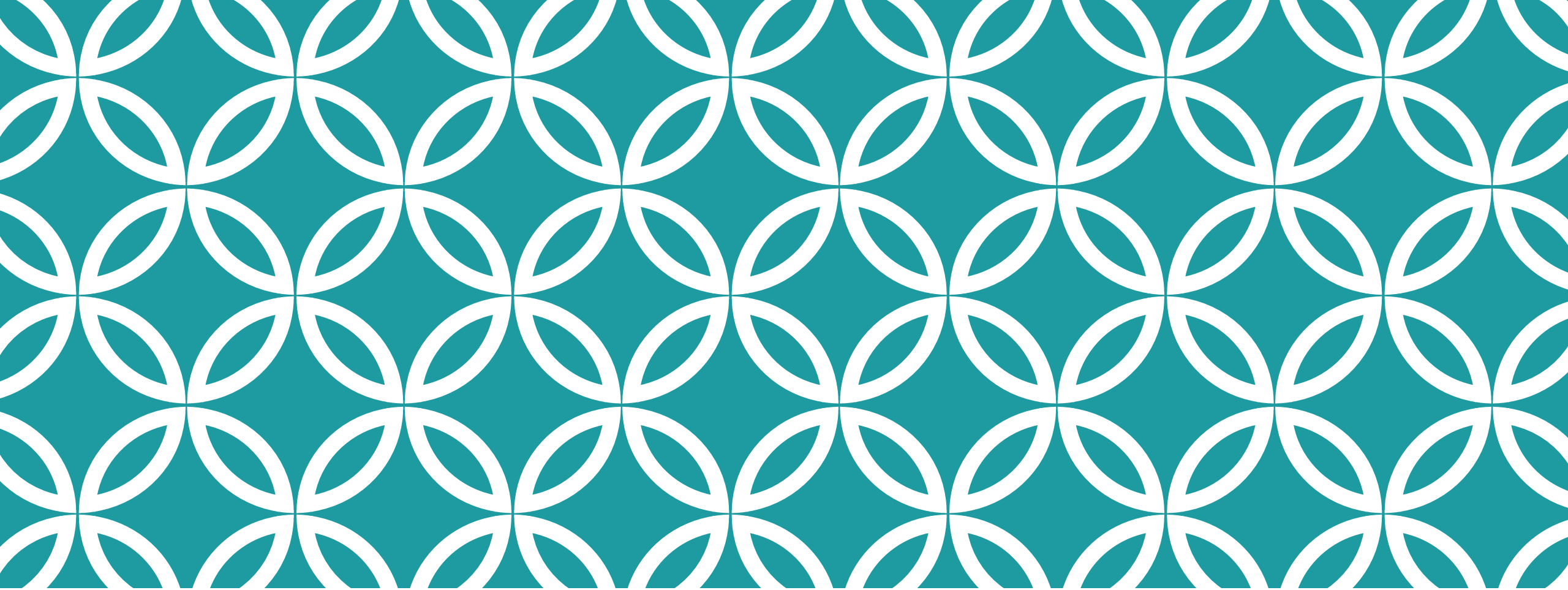
- Lister les produits
- Ajouter un produit

Créer une DAO **DAOProduit**

- La manipuler depuis le programme principal

Implémenter la composition DAO vue précédemment

- Interface **IDAO**
- Interface **IDAOProduit** qui hérite de l'interface **IDAO**
- Classe **DAOProduitSQL** qui implémente l'interface **IDAOProduit**



JAVA 8

Les expressions lambda
Les streams

JAVA 8

Java 8 intègre plusieurs nouveautés

- La Programmation Fonctionnelle
- Les expressions lambdas
- Les streams
- Mot-clé « default » pour définir un comportement à des méthodes d'interface

JAVA 8 – PROGRAMMATION FONCTIONNELLE

Pour trier le tableau ci-dessous

```
String[] myArray = { "Jérémy", "Alissa", "Julie", "Mark" };
Arrays.sort(myArray);
```

- Si on veut le trier sans prendre en compte la casse, il faudra passer par un Comparator<String>

Avec la programmation fonctionnelle

```
Arrays.sort(myArray , String::compareToIgnoreCase);
```

- La notation des 2x deux points « :: » permet de faire appelle à la méthode pour chaque élément

JAVA 8 – PROGRAMMATION FONCTIONNELLE

On peut utiliser

- Des références de méthodes statiques (exemple précédent)
- Des références de méthodes d'instance (il suffit d'avoir l'instance)

```
public class Trieur {
  public int trier(String s1, String s2) {
    return (s1.compareTo(s2));
  }
}
```

```
Trieur myTrieur = new Trieur();
Arrays.sort(myArray , myTrieur::trier);
```

JAVA 8 – EXPRESSIONS LAMBDAS

Une expression lambda est un peu comme une méthode « anonyme »

Reprenons l'exemple précédent

```
String[] myArray = { "Jérémy", "Alissa", "Julie", "Mark" };
```

```
Arrays.sort(myArray, (s1, s2) -> s1.compareTo(s2));
```

```
Arrays.sort(myArray, (s1, s2) -> {
  return s1.compareTo(s2);
});
```

JAVA 8 – EXPRESSIONS LAMBDAS

Interface Math, on choisira son implémentation (addition par exemple)

```
public interface Math {
  public Double compute(Double a, Double b);
}
```

En utilisant les expressions lambdas

```
Math myMath = (a, b) -> a + b;
System.out.println(myMath.compute(42.0, 10.0));
```

```
Math myMath = (a, b) -> {
  return a + b;
};

System.out.println(myMath.compute(42.0, 10.0));
```

JAVA 8 – EXPRESSIONS LAMBDAS

Pour aller plus loin, on peut utiliser des interfaces fonctionnelles

Runnable	0 argument	pas de valeur de retour
Consumer	1 argument	pas de valeur de retour
BiConsumer	2 arguments	pas de valeur de retour
Predicate	1 argument	valeur booléenne en retour
BiPredicate	2 arguments	valeur booléenne en retour
Supplier	0 argument	valeur de retour
Function	1 argument	valeur de retour
BiFunction	2 arguments	valeur de retour

JAVA 8 – EXPRESSIONS LAMBDAS

Exemple

```
public <T> void doSomething(Supplier<T> function, Consumer<T> onSuccess, Consumer<Exception> onError) {
  try {
    T myResult = function.get();
    onSuccess.accept(myResult);
  }

  catch (Exception ex) {
    onError.accept(ex);
  }
}
```

```
doSomething(
  () -> "test",
  System.out::println,
  ex -> System.err.println("Erreur " + ex.getMessage())
);
```

JAVA 8 – EXPRESSIONS LAMBDAS

Tester le code précédent

Décliner l'impression de l'Exception dans une fonction

- L'appeler au format Fonctionnel

JAVA 8 – EXPRESSIONS LAMBDAS

Si on veut plus de 2 arguments, on peut créer une interface fonctionnelle

```
@FunctionalInterface
public interface Math<A, B, C, R> {
  public R compute(A a, B b, C b);
}
```

- A, B et C sont les 3 types d'argument

```
Math<Double, Double, Double, Double> myMath = (a, b, c) -> {
  return a + b + c;
};
```

JAVA 8 – LES STREAMS

Ajout aux Collections, permettent des actions pour chaque élément

```
String[] myArray = { "Jérémy", "Alice", "Julie", "Mark" };
Stream<String> myStream = Arrays.stream(myArray);
```

```
myStream.forEach(str -> {
  System.out.println(str);
});
```

```
myStream.forEach(System.out::println);
```

JAVA 8 – LES STREAMS

Reprendre le tableau de prénoms

- Filtrer les éléments : ne prendre que ceux qui ont la lettre « a »
- Afficher les éléments au format Fonctionnel