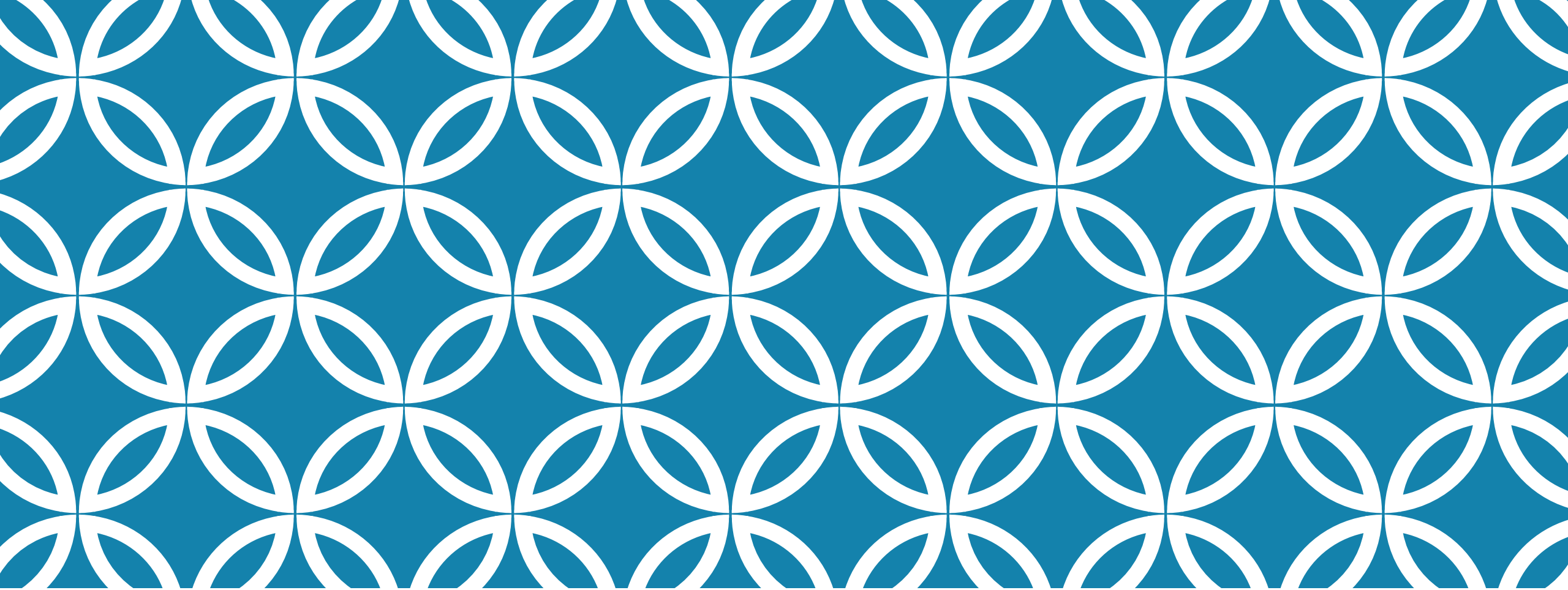
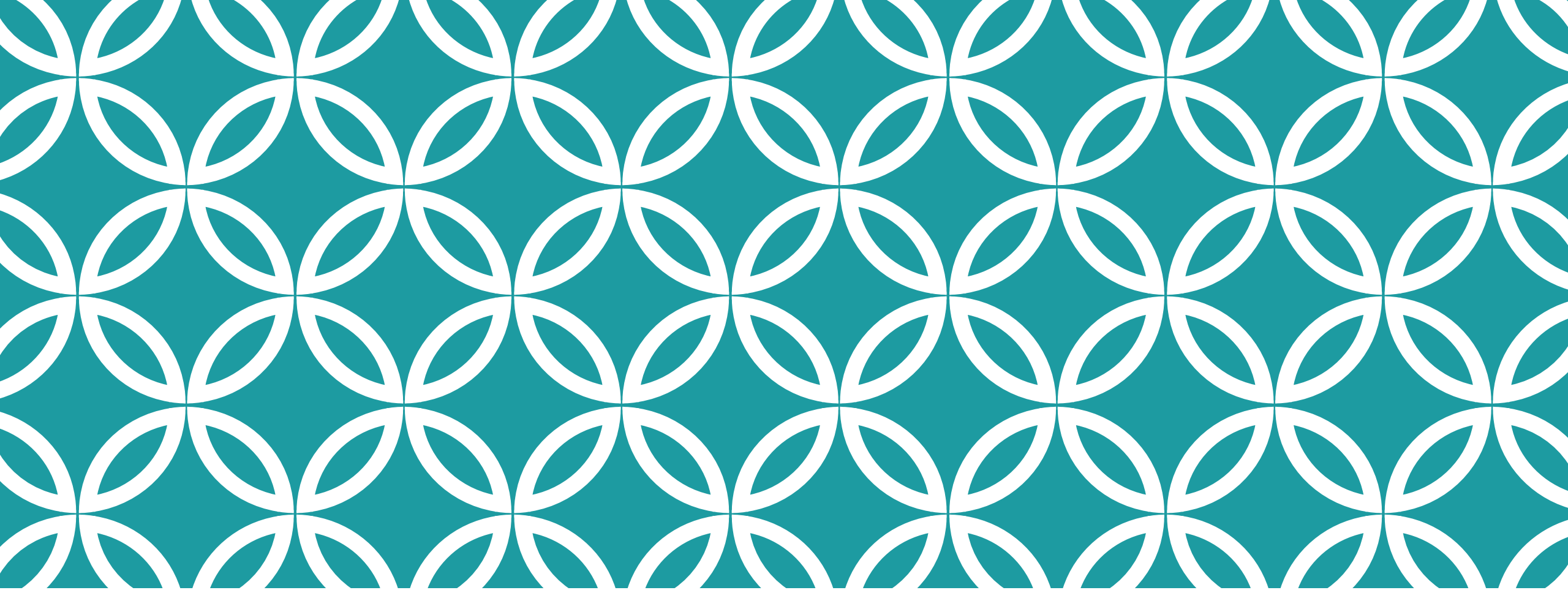


CERTIFICATION JAVA SE 8

Jérémy PERROUAULT



LES FONDAMENTAUX

Les bases

LES FONDAMENTAUX

Scope de variables

Structure des classes Java

Créer et exécuter un programme Java

Importer d'autres packages Java

Comparer les fonctionnalités et composants de Java

SCOPES DES VARIABLES

Class-Level Scope

- Déclaration dans une classe, en dehors de toute méthode
- Accessible depuis n'importe où dans la classe, éventuellement en dehors selon son modificateur

Modificateur	Classe	Package	Classes filles	World
public	X	X	X	X
protected	X	X	X	
private	X			
-	X	X		

SCOPES DES VARIABLES

Method-Level Scope

- Déclaration dans une méthode (variable locale, ou paramètres)
- Accessible dans la méthode jusqu'à la fin de celle-ci

Block-Level Scope

- Déclaration dans un bloc d'instructions (accolades)
- Accessible uniquement dans ce bloc, et les sous-blocs

- Une variable ne peut être déclarée qu'une seule fois.

SCOPES DES VARIABLES

```
public class ScopeApplication {
    private int number1 = 0;

    public static void main(String[] args) {
        int number2 = 0;

        //TODO
    }

    int number3 = 0;
}
```

Quelle variable est accessible de l'extérieur de la classe ?

- number1
- number2
- number3
- Aucune

SCOPES DES VARIABLES

```
public class ScopeApplication {
    public static void main(String[] args) {
        int number = 0;
        {
            int number = 1;
        }

        System.out.println(number);
    }
}
```

Que se passe-t-il quand la méthode *main* est compilée et exécutée ?

- Erreur à la compilation
- Erreur à l'exécution
- « 0 » est imprimé dans la console
- « 1 » est imprimé dans la console

SCOPES DES VARIABLES

```
public class ScopeApplication {
    static int number = 0;

    public static void main(String[] args) {
        number++;

        {
            number++;
        }

        System.out.println(number);
    }
}
```

Que se passe-t-il quand la méthode *main* est compilée et exécutée ?

- Erreur à la compilation
- Erreur à l'exécution
- « 1 » est imprimé dans la console
- « 2 » est imprimé dans la console

LES FONDAMENTAUX

Scope de variables

Structure des classes Java

Créer et exécuter un programme Java

Importer d'autres packages Java

Comparer les fonctionnalités et composants de Java

DÉCLARATION DE CLASSES

- Modificateurs public ou private
- Le mot-clé « class »
- Le nom de la classe, avec une majuscule par convention (CamelCase)
- Le nom de la classe dérivée (si disponible) précédé du mot-clé « extends »
- Le nom des interfaces (si disponibles) précédés du mot-clé « implements » et séparées par des virgules (si plusieurs)
- Le corps de la classe (attributs, constructeurs, méthodes) encadré par des accolades

```
public class ClassDeclaration {
    //Attributs

    //Constructeurs

    //Méthodes
}
```

DÉCLARATION D'ATTRIBUTS

- Modificateurs (ou aucun)
- Le type de l'attribut
- Le nom de l'attribut, avec une minuscule par convention (lowerCamelCase)

DÉCLARATION DE MÉTHODES

- Modificateurs (ou aucun)
- Le type de retour de la méthode, ou *void* si pas de valeur de retour
- Le nom de la méthode, avec une minuscule par convention (lowerCamelCase)
- La liste des paramètres, séparés par des virgules, avec pour chacun son type et son nom
- Une liste d'exceptions, séparées par des virgules
- Le corps de la méthode, entouré par des accolades
- La première ligne constitue la « signature » de la méthode

DÉCLARATION DE CONSTRUCTEURS

- Modificateurs (ou aucun)
- Le nom de la classe
- La liste des paramètres, séparés par des virgules, avec pour chacun son type et son nom
- Une liste d'exceptions, séparées par des virgules
- Le corps du constructeur, entouré par des accolades

STRUCTURES

```java
public class ClassDeclaration {
    //Attributs
    int number1 = 0;

    //Constructeurs
    public ClassDeclaration(int number1) {
        //TODO
    }

    //Méthodes
    public int add(int a, int b) {
        return a + b;
    }
}
```

STRUCTURES

```
public class ClassDeclaration extends SuperClass, implements InterfaceA { }

public class ClassDeclaration implements InterfaceA extends SuperClass  { }

public class ClassDeclaration extend SuperClass implement InterfaceA { }

class ClassDeclaration extends SuperClass, SuperClassB implements InterfaceA { }
```

Quelle est la bonne déclaration ?

- A
- B
- C
- D
- Aucune

STRUCTURES

```
void private add(Integer integer) throws Exception { }

public add(Integer integer) throws Exception { }

protected void add(Integer integer) throws Exception { }
```

Quelle est la bonne déclaration ?

- A
- B
- C
- Toutes
- Aucune

STRUCTURES

Un attribut et une méthode peuvent partager le même nom dans une même classe.

- Vrai
- Faux

LES FONDAMENTAUX

Scope de variables

Structure des classes Java

Créer et exécuter un programme Java

Importer d'autres packages Java

Comparer les fonctionnalités et composants de Java

APPLICATION EXÉCUTABLE

Une classe publique

Une méthode publique, statique, qui s'appelle *main* et qui attend un tableau de chaines

```
public class ExecutableApplication {
    public static void main(String[] args) {
        System.out.println("Allo le monde !");
    }
}
```

APPLICATION EXÉCUTABLE

La compilation est possible grâce à un compilateur Java

- Transforme le fichier *java* en fichier *class*

```
> javac .\ExecutableApplication.java
```

L'exécution est possible grâce à un exécuteur Java

```
> java ExecutableApplication
```

APPLICATION EXÉCUTABLE

Que fait *javac* ?

- Utilise le code source et produit un code natif
- Utilise le code source et produit un bytecode
- Utilise un bytecode et produit un code natif
- Utilise un code natif et produit un bytecode

APPLICATION EXÉCUTABLE

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

Quelle est la méthode *main* ?

- A
- B
- C
- D

APPLICATION EXÉCUTABLE

Que se passe-t-il si le nom du fichier compilé et exécuté est différent de celui de la classe qui contient la méthode *main* ?

- Erreur de compilation
- Erreur d'exécution
- Le programme s'exécute normalement

APPLICATION EXÉCUTABLE

Que doit-on passer comme premier argument à la commande *java* ?

- Le nom de la classe principale, en ajoutant .java
- Le nom de la classe principale, en ajoutant .class
- Le nom de la classe principale
- Le premier argument du programme

APPLICATION EXÉCUTABLE

Que doit-on passer comme premier argument à la commande *java* ?

- Le nom de la classe principale, en ajoutant .java
- Le nom de la classe principale, en ajoutant .class
- Le nom de la classe principale
- Le premier argument du programme

LES FONDAMENTAUX

Scope de variables

Structure des classes Java

Créer et exécuter un programme Java

Importer d'autres packages Java

Comparer les fonctionnalités et composants de Java

IMPORTER DES PACKAGES

Sans import, le nom complet de la classe (ou interface) doit être utilisé

- java.util.List
- java.util.Map

```
java.util.List myList = null;
```

La liste des imports doit être fait au début de fichier, après l'instruction *package*

```
import java.util.List;
import java.util.Map;
```

```
List myList = null;
```

IMPORTER DES PACKAGES

On peut importer un package complet en utilisant le caractère wildcard

```
import java.util.*;
```

IMPORTER DES PACKAGES

```
import fr.formation.*;
```

Quels types peuvent-être utilisés par son nom simple ?

- frformation.ClasseA
- frformation.souspackage.ClasseA
- fr.formation.sous.package.ClasseA
- Aucun

IMPORTER DES PACKAGES

```
import fr.formation.Cl*;
```

Quels types peuvent-être utilisés par son nom simple ?

- Tous les types de fr.formation commençant par « Cl »
- Tous les types dans le package fr.formation.Cl
- A et B
- Aucun

IMPORTER DES PACKAGES

Importer un type depuis le même package produira une erreur de compilation.

- Vrai
- Faux

IMPORTER DES PACKAGES

```
import java.util.List;
import fr.formation.List;
```

Qu'est-ce qui est correct par rapport à la compilation ?

- La compilation échoue à la deuxième instruction d'import
- La compilation échoue lorsqu'on fait appel à la classe *List*
- La compilation échoue si on fait référence au nom simple *List*, mais pas si on utilise son nom complet

LES FONDAMENTAUX

Scope de variables

Structure des classes Java

Créer et exécuter un programme Java

Importer d'autres packages Java

Comparer les fonctionnalités et composants de Java

JAVA BUZZWORDS

Simple

Orienté objet

Distribué

Interprété

Robuste

Securisé

Architecture neutre

Indépendance des plateformes

Haute performance

Multithreaded

Dynamique

CONCEPTS D'ORIENTATION OBJET

Héritage

Polymorphism

- overloading
- overwriting

Abstraction

- Cacher certains détails et ne montrer que les fonctionnalités essentielles, simplifier

Encapsulation

- Envelopper des données et du code en une seule entité

JAVA BUZZWORDS

Qu’est-ce qui n’est pas une fonctionnalité de Java ?

- Distribué
- Flexible
- Interprété
- Robuste

JAVA BUZZWORDS

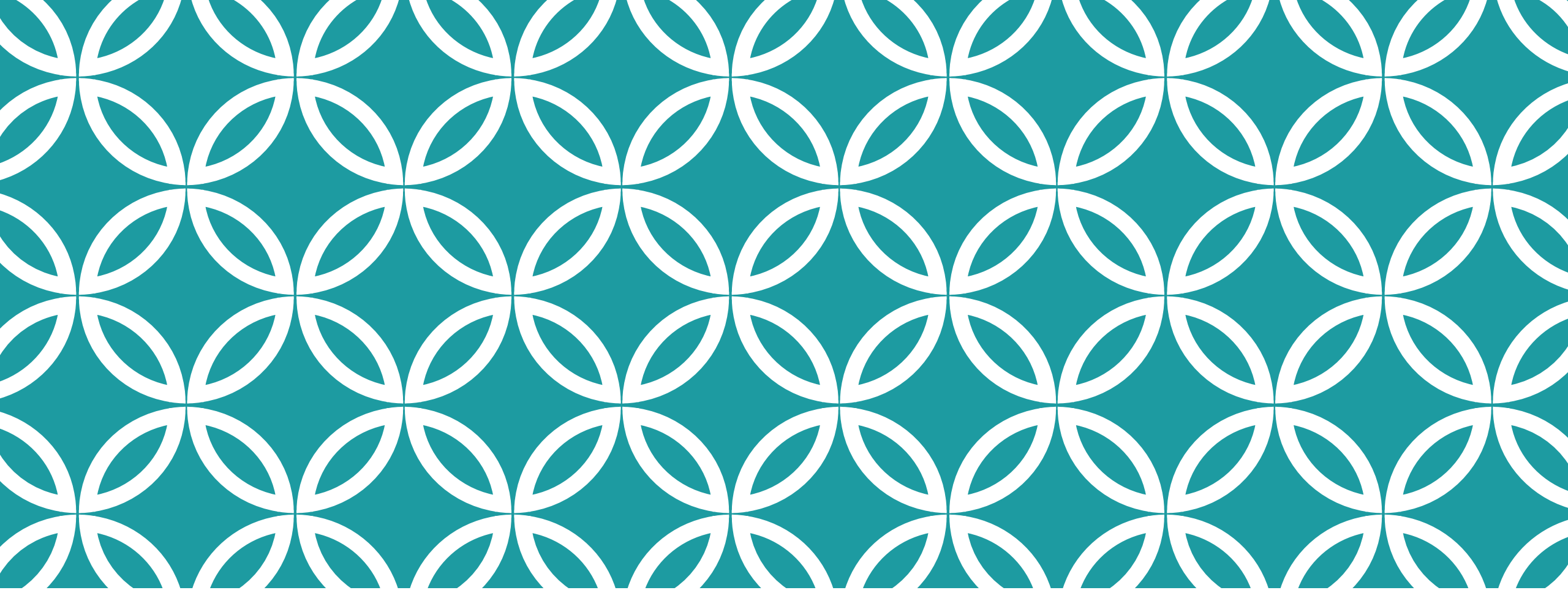
L’indépendance des plateformes signifie qu’un programme Java aura toujours le même comportement, peu importe la plateforme.

- Vrai
- Faux

JAVA BUZZWORDS

Quel est le concept lorsqu'une variable d'une classe est cachée de l'extérieur, et peut être accessible depuis une méthode publique de sa classe ?

- Héritage
- Polymorphisme
- Abstraction
- Encapsulation



LES TYPES DE DONNÉES

Les types de données

CYCLE DE VIE D'UN OBJET

Création d'un objet

- Instanciation (new)
- Initialisation (constructeur)

Destruction d'un objet

- Lorsqu'il n'est plus utilisé*
- Le garbage collector de Java va s'en charger
  - Dès lors que l'objet n'est plus référencé du tout dans l'application
- Impossible d'explicitement demandé sa destruction
  - Possible d'utiliser la technique de « déréférencement par réaffectation », l'objet devient alors « éligible » au garbage collector

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

CYCLE DE VIE D'UN OBJET

```java
public class DataLifecycle {
    private int number;

    public DataLifecycle() {
        this.number = 0;
    }

    public DataLifecycle(int number) {
        this.number = number;
    }
}
```

```java
DataLifecycle data1 = new DataLifecycle();
DataLifecycle data2 = new DataLifecycle(0);
DataLifecycle data3 = new DataLifecycle(0);
```

Combien d'objets sont créés ?

- 1
- 2
- 3

CYCLE DE VIE D'UN OBJET

```java
public class DataLifecycle {
    private int number;

    public DataLifecycle() {
        this.number = 0;
    }

    public DataLifecycle(int number) {
        this.number = number;
    }
}
```

```java
DataLifecycle data1 = new DataLifecycle();
DataLifecycle data2 = data1;
DataLifecycle data3 = data2;
```

Combien d'objets sont créés ?

- 1
- 2
- 3

CYCLE DE VIE D'UN OBJET

```java
public class DataLifecycle {
    private int number;

    public DataLifecycle() {
        this.number = 0;
    }

    public DataLifecycle(int number) {
        this.number = number;
    }
}
```

```java
DataLifecycle data1 = new DataLifecycle();
DataLifecycle data2 = data1;
data2 = null;
```

Aucun objet n'est éligible pour le garbage colletor.

- Vrai
- Faux

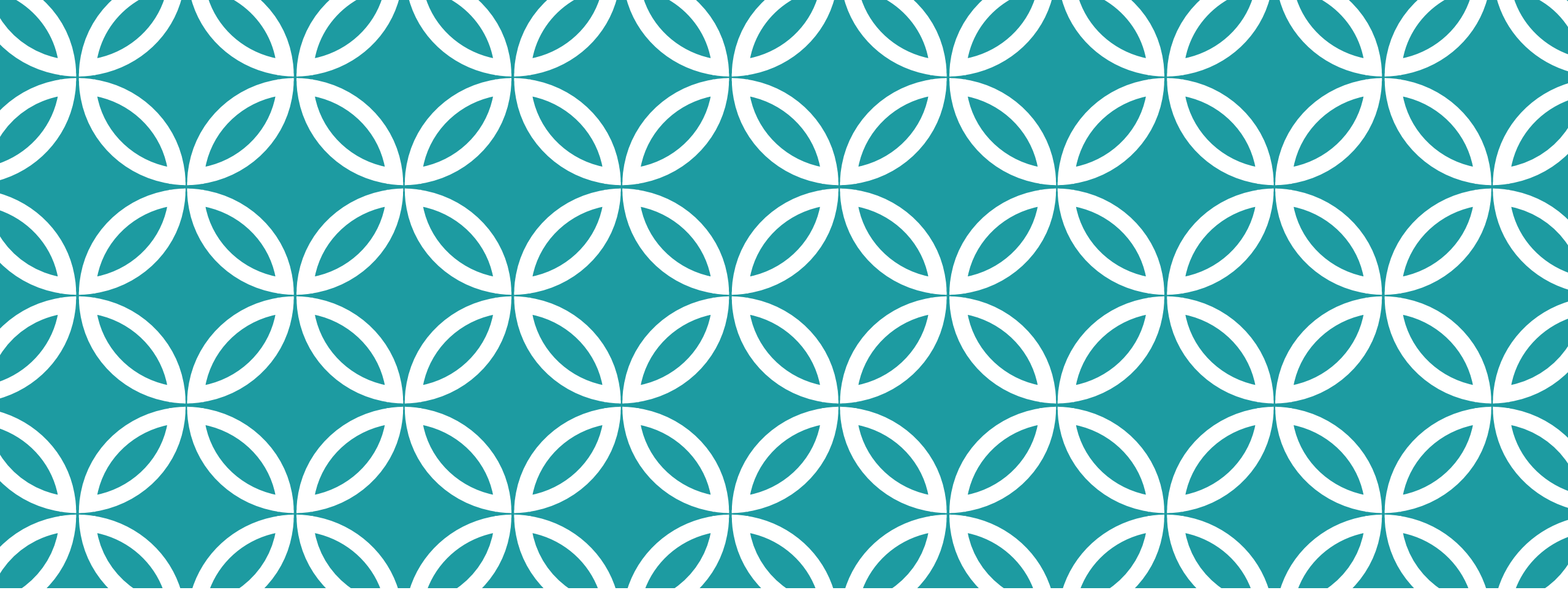
CYCLE DE VIE D'UN OBJET

```java
public class DataLifecycle {
    public DataLifecycle garbage() {
        DataLifecycle data1 = new DataLifecycle();
        DataLifecycle data2 = new DataLifecycle();

        return data1;
    }
}
```

```java
DataLifecycle data3 =
  new DataLifecycle().garbage(); //#1
```

Combien d'objets sont éligibles au garbage collector ?

- 0
- 1
- 2
- 3



OPÉRATEURS

Les opérateurs

LES OPÉRATEURS

Opérateurs Java et priorités

Comparaisons entre objets

Opérateur ternaire

Bloc *switch*

OPÉRATEURS

Opérateurs	Priorité élevée
Suffixe	expr++ expr--
Unaire	++expr -expr +expr -expr ~ !
Multiplicatif	* / %
Additif	+ -
Décalage	<< >> >>>
Relationnel	< > <= >= instanceof
Egalité	== !=

Opérateurs	Priorité faible
AND bit à bit	&
OR bit à bit exclusif	^
OR bit à bit inclusif	|
AND logique	&&
OR logique	||
Ternaire	? :
Affectation	= += -= *= /= %= &= ^= |= <<= >>= >>>=

OPÉRATEURS

```java
int integer1 = 1 + 2 * 3;
int integer2 = (1 + 2) * 3;
System.out.println(integer1); // 7
System.out.println(integer2); // 9


boolean boolean1 = true || true && false;
boolean boolean2 = (true || true) && false;
System.out.println(boolean1); // true
System.out.println(boolean2); // false
```

Les parenthèses permettent de changer la priorité par défaut

- Multiplication est évaluée avant l'addition
- AND est évalué avant OR

OPÉRATEURS

```
int i1 = 1, i2 = 2;
int i = i2 = i1;

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

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- Imprime « 1 »
- Imprime « 2 »
- La compilation échoue

OPÉRATEURS

```
int i1 = 1, i2 = 2;
int i = i2 * ++i1 * i1;

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

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- Imprime « 2 »
- Imprime « 4 »
- Imprime « 8 »
- La compilation échoue

OPÉRATEURS

```
boolean b1 = false, b2 = true;
boolean b = (!b1 || !b2) && b1;

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

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- Imprime « true »
- Imprime « false »

OPÉRATEURS

```
int i = 4 << 4 / 2;
System.out.println(i);
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- Imprime « 16 »
- Imprime « 32 »
- Imprime « 64 »
- Imprime « 128 »

LES OPÉRATEURS

Opérateurs Java et priorités

Comparaisons entre objets

Opérateur ternaire

Bloc *switch*

COMPARAISONS

L'opérateur « == » compare les références d'objets

- Ce qui implique que le résultat sera vrai seulement si les références sont identiques

```
String string1 = new String("Toto");
String string2 = "Toto";
String string3 = "Toto";

System.out.println(string1 == string2); // false
System.out.println(string2 == string3); // true
```

COMPARAISONS

L’utilisation de la méthode permet de comparer par valeur

```
String string1 = new String("Toto");
String string2 = "Toto";
String string3 = "Toto";

System.out.println(string1.equals(string2)); // true
System.out.println(string2.equals(string3)); // true
```

COMPARAISONS

```
String string1 = new String("Albert");
String string2 = new String(string1);

System.out.println(string1 == string2);
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- Imprime « true »
- Imprime « false »

COMPARAISONS

```
String string1 = new String("Albert");
String string2 = new String("albert");

System.out.println(string1.equals(string2));
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- Imprime « true »
- Imprime « false »

LES OPÉRATEURS

Opérateurs Java et priorités

Comparaisons entre objets

Opérateur ternaire

Bloc *switch*

OPÉRATEUR TERNAIRE

Raccourcis aux instructions if-then-else avec la syntaxe suivante :

- condition ? vrai : false

```
int i = 1;
String output = i > 0 ? "Positif" : "Négatif";
System.out.println(output);
```

OPÉRATEUR TERNAIRE

```java
int i = 1;
short s = 2;
int number = i > s ? i : s;
System.out.println(number);
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- Imprime « 1 »
- Imprime « 2 »
- La compilation échoue

OPÉRATEUR TERNAIRE

```
int i = 1;
int j = ++i + 1 > 1 ? 1 + 1 : 3;
System.out.println(j);
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- Imprime « 1 »
- Imprime « 2 »
- Imprime « 3 »
- La compilation échoue

LES OPÉRATEURS

Opérateurs Java et priorités

Comparaisons entre objets

Opérateur ternaire

Bloc *switch*

SWITCH

Switch fonctionne avec *byte*, *short*, *char*, *int*, *enum* et *String*

Pour chaque cas, le corps doit être arrêté par *break*

Un cas par défaut *default* peut être ajouté

```
nder gender = Gender.UNISEX;

switch (gender) {
    case MALE:
        System.out.println("MALE"); break;

    case FEMALE:
        System.out.println("FEMALE"); break;

    default:
        System.out.println("UNISEX");
}
```

SWITCH

```
int i = 2;
switch (i % 2 == 0) {
    case true:
        System.out.println("Paire");
    case false:
        System.out.println("Impaire");
}
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- Imprime « Paire »
- Imprime « Impaire »
- Imprime « Paire » et « Impaire »
- La compilation échoue

SWITCH

```
String s = "Toto";
switch (s) {
    default:
        System.out.println("Java");
    case "Toto":
        System.out.println("OCA");
}
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

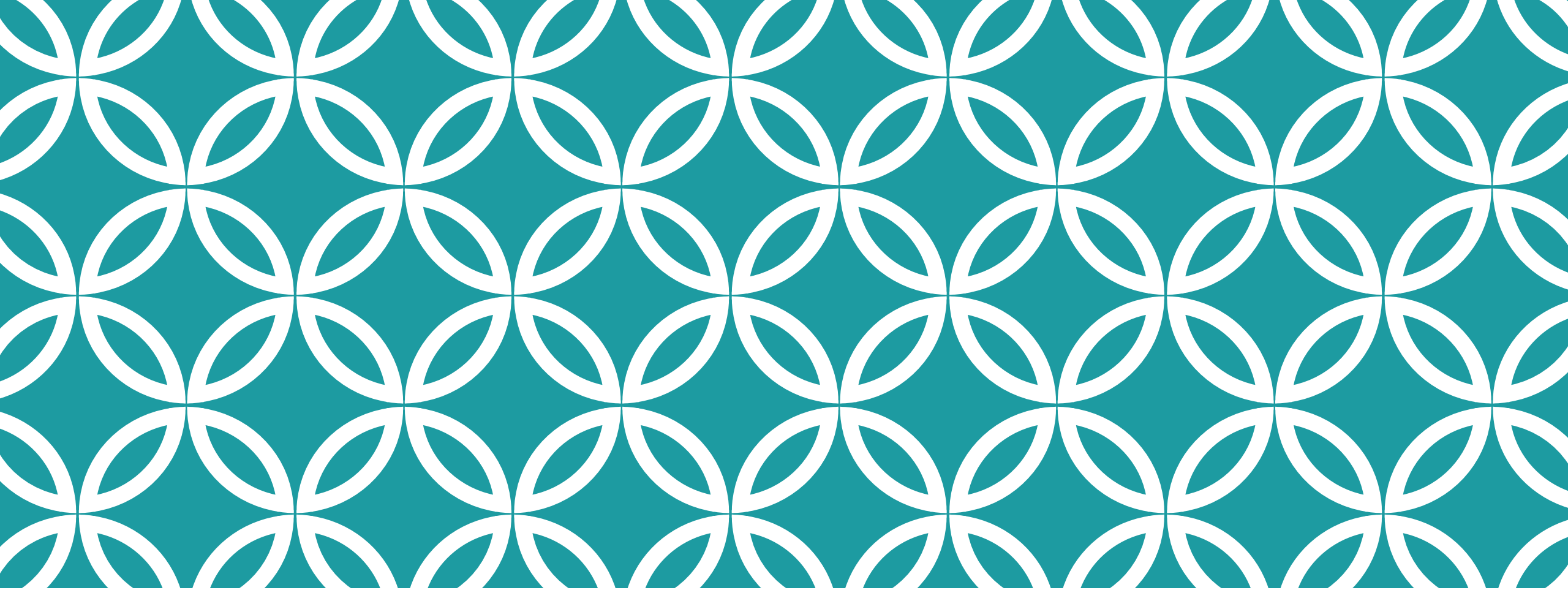
- Imprime « Java »
- Imprime « OCA»
- Imprime « Java » et « OCA »
- La compilation échoue

SWITCH

```
String s = "Toto";
switch (s) {
    case "Toto":
        System.out.println("OCA");
    default:
        System.out.println("Java");
}
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- Imprime « Java »
- Imprime « OCA»
- Imprime « Java » et « OCA »
- La compilation échoue



DÉCLARATION DE TABLEAUX

Déclarer un tabeau

TABLEAU 1-D

Déclarer un tableau de types, on déclare le tableau avec le type, les crochets et son nom, ou son type, son nom suivi des crochets

```
int[] intArray;
String stringArray[];
```

Pour instancier un tableau, on utilise le mot-clé *new* et on précise la taille du tableau

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

Pour instancier et initialiser un tableau, on précise les valeurs entre accolades

```
int[] intArray = new int[] { 5, 4, 3, 2, 1 };
```

```
int[] intArray = { 5, 4, 3, 2, 1 };
```

TABLEAU N-D

Utiliser un crochet par dimension

```
int[][] intArray;
String[][][] stringArray;
```

Pour instancier un tableau, on utilise le mot-clé *new* et on précise la taille du tableau

```
int[][] intArray = new int[5][2];
```

Pour instancier et initialiser un tableau, on précise les valeurs entre accolades

```
int[][] intArray = new int[][] { { 5, 4 }, { 3, 2 } };
```

```
int[][] intArray = { { 5, 4 }, { 3, 2, 1 } };
```

TABLEAUX

Quelles déclarations sont valides ? Sélectionnez-en 2.

- int[] intArray;
- []int intArray;
- int intArray[];
- int []intArray;

TABLEAUX

```
long[] myArray = new int[] { 1, 2 };
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- L'éxécution échoue
- La variable myArray sera un tableau d'int avec 2 éléments
- La variable myArray sera un tableau de long avec 2 éléments
- La compilation échoue

TABLEAUX

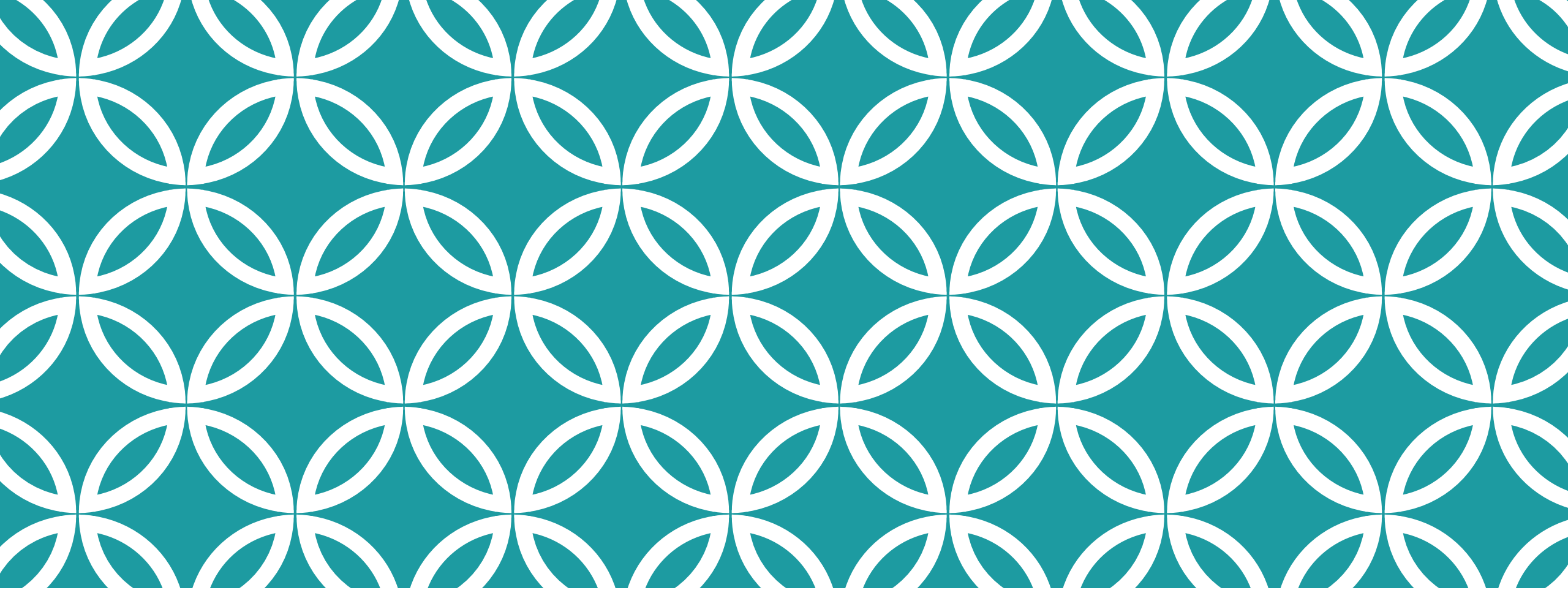
Quelle déclaration est valide ?

- intArray = new int[2][3][];
- intArray = new int[][][4];
- intArray = new int[][][];
- Aucune

TABLEAUX

Tous les éléments dans un même niveau d'un tableau multi-dimensionnel doivent avoir la même taille.

- Vrai
- Faux



BREAK & CONTINUE

S’orienter dans une boucle

BREAK & CONTINUE

Break et *continue* existent tous deux sous la forme non-labellisé et labellisé

Break interrompt le bloc d'itération

Continue esquive l'itération en cours et passe à la suivante

BREAK & CONTINUE

```java
public static void main(String... args) {
    int j = 1;

    for (int i = 1; i < 4; i++) {
        while (j < 4) {
            j++;
            System.out.println("inner");
        }

        System.out.println("outer");
    }
} // inner inner inner outer outer outer
```

```java
public static void main(String... args) {
    int j = 1;

    for (int i = 1; i < 4; i++) {
        while (j < 4) {
            if (j % 2 == 0) {
                break;
            }

            j++;
            System.out.println("inner");
        }

        System.out.println("outer");
    }
} // inner outer outer outer
```

BREAK & CONTINUE

```
public static void main(String... args) {
    int j = 1;

    oca: for (int i = 1; i < 4; i++) {
        while (j < 4) {
            if (j % 2 == 0) {
                break oca;
            }

            j++;
            System.out.println("inner");
        }

        System.out.println("outer");
    }
} // inner
```

BREAK & CONTINUE

```
public static void main(String... args) {
    int j = 1;

    for (int i = 1; i < 4; i++) {
        while (j < 4) {
            j++;

            if (j % 2 == 0) {
                continue;
            }

            System.out.println("inner");
        }

        System.out.println("outer");
    }
} // inner outer outer outer
```

```
public static void main(String... args) {
    int j = 1;

    oca: for (int i = 1; i < 4; i++) {
        while (j < 4) {
            j++;

            if (j % 2 == 0) {
                continue oca;
            }

            System.out.println("inner");
        }

        System.out.println("outer");
    }
} // inner outer
```

BREAK & CONTINUE

```
int i = 0, j = 0;
while (i < 2) {
    while (j < 2) {
        if (i + j == 2) break;
        j++;
    }

    i++;
}

System.out.println(i + " " + j);
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- Imprime « 0 0 »
- Imprime « 1 1 »
- Imprime « 2 2 »
- Imprime « 0 2 »

BREAK & CONTINUE

```
int i = 0, j = 0;
while (i < 2) {
    oca: while (j < 2) {
        j++;
    }

    if (i + j == 2) break oca;
    i++;
}

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

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- La compilation échoue
- L'exécution échoue
- Imprime « 1 »
- Imprime « 2 »

BREAK & CONTINUE

```
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
        System.out.println("OCA");
        continue 1;
    }
}
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

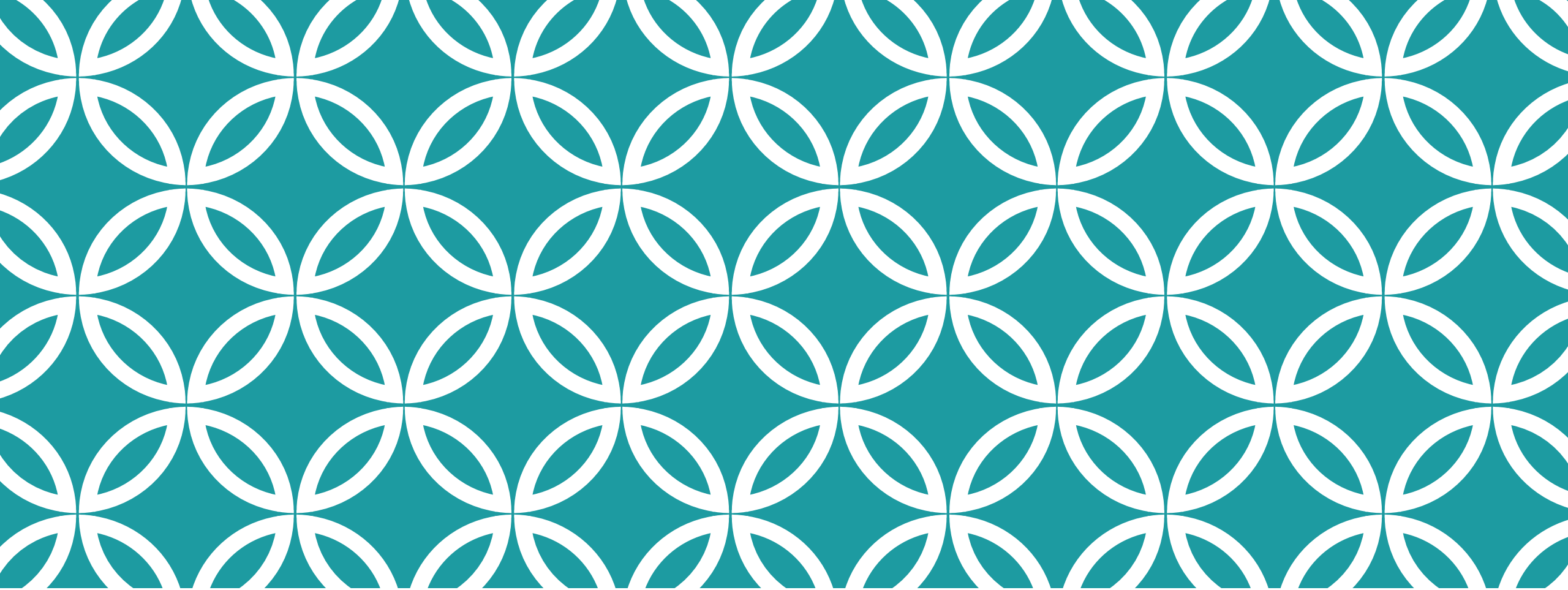
- La compilation échoue
- Imprime « OCA » 4 fois
- Imprime « OCA » 2 fois
- Rien n'est imprimé mais il n'y a pas d'erreur

BREAK & CONTINUE

```
for (int i = 0; i < 1; i++) {
    while (true) {
        continue;
    }
}
```

Ce code exécute une boucle infinie.

- Vrai
- Faux



POO

Orienté objet

PROGRAMMATION ORIENTÉ OBJET

Héritage

Polymorphisme

- Overloading
- Overwriting

Static, final

Casting

This & super

Abstract & interfaces

PROGRAMMATION ORIENTÉ OBJET

```
public class Person {
    private String name;

    public String upperCase() {
        this.name = this.name.toUpperCase();
        return this.name;
    }

    public String lowerCase() {
        this.name = this.name.toLowerCase();
        return this.name;
    }
}
```

Ce code applique l'encapsulation.

- Vrai
- Faux

PROGRAMMATION ORIENTÉ OBJET

```java
public class Super {
    public int field1;
}

public class Me extends Super {
    private int field2;
}

public class Sub extends Me {
}
```

La classe Sub hérite de quels attributs ?

- field1 seulement
- field2 seulement
- Les deux
- Aucun des deux

PROGRAMMATION ORIENTÉ OBJET

```
public void swap(int value1, int value2) {
    int tmp = value1;
    value1 = value2;
    value2 = tmp;
}
```

```
int value1 = -1, value2 = 1;
swap(value1, value2);
System.out.println(value1 + " " + value2);
```

Que produit cet extrait de code lorsqu'il est compilé et exécuté ?

- -1 -1
- -1 1
- 1 -1
- 1 1

PROGRAMMATION ORIENTÉ OBJET

```
public class Data {
    int value;

    Data(int value) {
        this.value = value;
    }
}
```

```
public void swap(Data data1, Data data2) {
    Data tmp = data1;
    data1 = data2;
    data2 = tmp;
}
```

```
Data data1 = new Data(-1), data2 = new Data(1);
swap(data1, data2);
System.out.println(data1.value + " " + data2.value);
```

Que produit cet extrait de code lorsqu'il est compilé et exécuté ?

- -1 -1
- -1 1
- 1 -1
- 1 1

PROGRAMMATION ORIENTÉ OBJET

```java
public class Data {
    int value;

    Data(int value) {
        this.value = value;
    }
}
```

```java
public void increase(int value, Data data) {
    value++;
    data.value++;
}
```

```java
int value = 0;
Data data = new Data(0);
increase(value, data);
System.out.println(value + " " + data.value);
```

Que produit cet extrait de code lorsqu'il est compilé et exécuté ?

- 0 0
- 0 1
- 1 0
- 1 1

PROGRAMMATION ORIENTÉ OBJET

```
public interface InterfaceA {
    public static int field;
    public static void methodA() {
    }
}
```

Ce code est valide.

- Vrai
- Faux

PROGRAMMATION ORIENTÉ OBJET

```
public interface InterfaceA {
    public void methodA();
}

public abstract class ClassA implements InterfaceA {
    private String field;
}
```

Ce code est valide.

- Vrai
- Faux

PROGRAMMATION ORIENTÉ OBJET

```
Object myObject = new Object();
Integer myInteger = (Integer)myObject;
Long myLong = (Long)myObject;
Number myNumber = (Number)myObject;
```

A quelle ligne le casting produit une erreur de compilation ?

- Ligne 1
- Ligne 2
- Ligne 3
- Ligne 4
- Aucune

PROGRAMMATION ORIENTÉ OBJET

```java
public class Super {
    protected int value = 0;
    protected int increment() {
        return value + 1;
    }
}

public class Sub extends Super {
    public int value = 2;
    public int increment() {
        return value + 2;
    }
}
```

```java
Super mySuper = new Sub();
System.out.println(mySuper.increment());
```

Que produit cet extrait de code lorsqu'il est compilé et exécuté ?

- Imprime « 1 »
- Imprime « 2 »
- Imprime « 3 »
- Imprime « 4 »

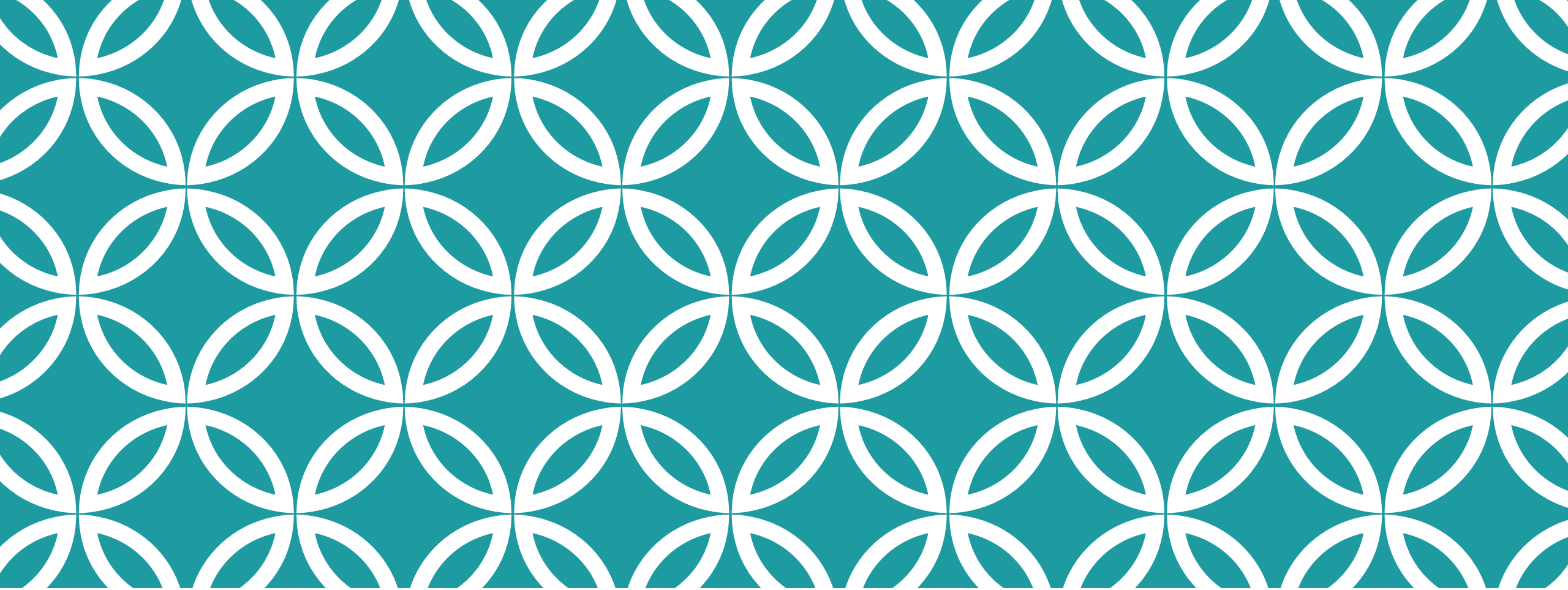
PROGRAMMATION ORIENTÉ OBJET

```java
public class Wind {
    public int number;

    public Wind() {
        this.number++;
        this(0);
    }

    public Wind(int number) {
        this.number = number;
    }
}
```

```java
System.out.println(new Wind().number);
```

Que produit cet extrait de code lorsqu'il est compilé et exécuté ?

- Une erreur à la compilation
- Imprime « 0 »
- Imprime « 1 »



EXCEPTIONS

Gérer les exceptions

EXCEPTIONS

Checked Exception

- Exception interne à l'application qui peut être anticipée
- Exception qui doit être gérée dans le code
  - Par un bloc try .. catch
  - Par l'utilisation du mot-clé throws sur la signature d'une méthode
- Dérive de **Exception**
- Toute les exceptions sont « checked », sauf celles indiquées comme Error ou RuntimeException

Unchecked Exception

- Exception interne à l'application qui ne peut pas être anticipée
- Dérive de **RuntimeException**

Error

- Exception externe à l'application ne pouvant pas être anticipée
  - Exemple : OutOfMemoryError, StackOverflowError
- Dérive de **Error**

EXCEPTIONS

Pour gérer une Exception

- bloc try autour de ce qui peut lever une Exception
- bloc(s) catch pour traiter une (ou plusieurs) Exception
- bloc finally (optionnel)
  - S'exécutera quoi qu'il arrive, avant un *return* si celui-ci est dans le bloc try et/ou catch

EXCEPTIONS

```
Connection c = null;

try {
    c = DriverManager.getConnection("");
    //...
}

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

finally {
    if (c != null) {
        try {
            c.close();
        }

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

```
try (Connection c = DriverManager.getConnection("")) {
    //...
}

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

EXCEPTIONS

Une méthode lève une « unchecked » Exception. Quelle est la solution ?

- Englober la méthode par un bloc try
- Spécifier l'exception avec la clause *throws*
- Aucune de ces propositions

EXCEPTIONS

```
int[] myArray = new int[0];

if (myArray.length > 0) {
    myArray[1] = 1;
}
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- NullPointerException
- ArithmeticException
- ArrayIndexOutOfBoundsException
- Pas d'exception

EXCEPTIONS

```
int[] array1 = new int[0];
int[] array2 = null;

array1 = array2;
array1[0] = array1.length;
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

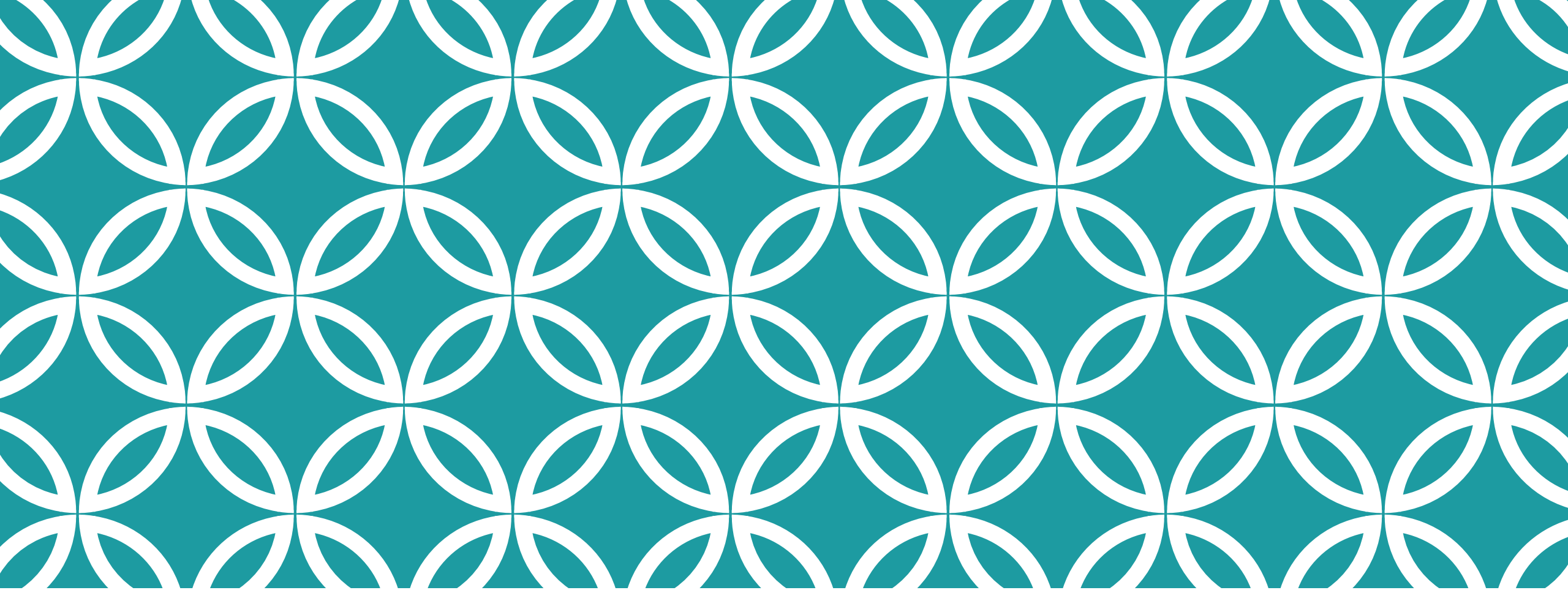
- NullPointerException
- ArithmeticException
- ArrayIndexOutOfBoundsException
- RuntimeException

EXCEPTIONS

```
int[] myArray = new int[0];

if (myArray.length == 0) {
    myArray = null;
}

myArray[0] = (Integer)(Number)0d;
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- NullPointerException
- ArrayIndexOutOfBoundsException
- ClassCastException
- La compilation échoue



JAVA API

Quelques API JAVA

API JAVA

StringBuilder

String

Date

Lambda

Concurrency

STRINGBUILDER

Permet de fabriquer une chaine de caractères

- Ajouter
- Insérer
- Remplacer
- Supprimer

```
StringBuilder builder = new StringBuilder("W");

builder.append("d2");
builder.insert(1, "e");
builder.replace(1, 2, "in");
// builder.delete(4, 5);
builder.deleteCharAt(4);

System.out.println(builder); // Wind
```

STRINGBUILDER

Autres fonctionnalités

- reverse

```
System.out.println(builder.reverse()); // dniW
```

STRINGBUILDER

```
StringBuilder builder1 = new StringBuilder("W");
StringBuilder builder2 = builder1.append(builder1);

System.out.println(builder1);
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- L'éxécution échoue
- Imprime « W »
- Imprime « WW »

STRINGBUILDER

```
StringBuilder builder = new StringBuilder("Wind");
builder = builder.insert(2, 4);
System.out.println(builder);
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- L'éxécution échoue
- Imprime « Wind2 »
- Imprime « Wind4 »
- Imprime « Wi4nd »

STRINGBUILDER

```
StringBuilder builder = new StringBuilder("Wind");
builder.delete(1, 10);
System.out.println(builder);
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- L'éxécution échoue
- La compilation échoue
- Imprime « »
- Imprime « Wind »
- Imprime « W »

API JAVA

StringBuilder

String

Date

Lambda

Concurrency

STRING

Création d'un String

```
String string = "Wind";
String string = new String("Wind");
```

Remplacement

- replace Remplace toutes les occurrences
- replaceAll Remplace toutes les parties correspondantes à une expression régulière
- replaceFirst Remplace la première partie correspondante à une expression régulière

```
String string = "Winn";
System.out.println(string.replace("n", "d"));  // Widd
```

```
String string = "Wind is here";
System.out.println(string.replaceAll("\\s", "-")); // Wind-is-here
```

```
String string = "Wind is here";
System.out.println(string.replaceFirst("\\s", "-")); // Wind-is here
```

STRING

Couper

- split Découpe toutes les parties autour d'une expression régulière

```
String string = "Wind is here";
String[] parts = string.split("\\s");
```

Concaténer

- concat Ajoute une chaine à la fin de la chaine

```
String string1 = "Wind ";
String string2 = string1.concat("is here");
System.out.println(string2); // Wind is here
```

STRING

Jointure

- join Regroupe un tableau de chaines avec un élément « glue »

```
String string = String.join(".", "Wind", "is", "here");
System.out.println(string); // Wind.is.here
```

Formater

- format Formatte une chaine avec des arguments

```
String string = String.format("%s is %s", "Wind", "here");
System.out.println(string); // Wind is here
```

STRING

```
String oldText = "Wind is here";
String newText = oldText.replace("\\s", "A");
System.out.println(newText);
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- L'éxécution échoue
- Imprime « Wind is here »
- Imprime « WindAisAhere »
- Imprime « WindAis here »

STRING

```
String string = "Wind";
String[] parts = string.split("\\S");
System.out.println(parts.length);
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- L'éxécution échoue
- Imprime « 0 »
- Imprime « 1 »
- Imprime « 4 »

STRING

```
String string = String.join("+", "-", "A", "B", "C");
System.out.println(string);
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- La compilation échoue
- L'exécution échoue
- Imprime « A+B+C »
- Imprime « -+A+B+C »

STRING

```
String string = String.format("%1$d<%d", 1, 2, 3);
System.out.println(string);
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- L'exécution échoue
- Imprime « 1<1 »
- Imprime « 1<2 »
- Imprime « 1<3 »

API JAVA

StringBuilder

String

Date

Lambda

Concurrency

DATE

Manipulation de java.time.LocalDateTime, java.time.LocalDate, java.time.LocalTime, java.time.format.DateTimeFormatter, java.time.Period

DATE

LocalDateTime

- Objet immuable qui représente une date-heure, année-mois-jour-heure-minute-seconde
- L'heure est représentée avec une précision en nanoseconde

LocalDate

- Objet immuable qui représente une date, année-mois-jour

LocalTime

- Objet immuable qui représente une heure, heure-minute-seconde
- L'heure est représentée avec une précision en nanoseconde

DATE

Pour créer une nouvelle instance : now, parse, of*

```
LocalDateTime.now();
LocalDate.parse(text);
LocalTime.of(hour, minute, second);
```

Pour récupérer des informations : get*

```
localDate.getYear();
```

Pour effectuer des opérations : plus*, minus*

```
localDate.plusDays(5);
localDateTime.plusHours(2);
```

Pour changer une information : with*

```
localDate.withDays(6);
```

DATE

DateTimeFormatter aide à formatter et à parser une date-heure, avec :

- Des constantes prédéfinies, comme ISO_LOCAL_DATE
- Des patterns, comme uuuu-MMM-dd
- Des styles locaux, comme « date longue » ou « date courte »

```
DateTimeFormatter formatter = DateTimeFormatter.ISO_DATE;
LocalDate date = LocalDate.now();

String text = date.format(formatter);
System.out.println(text); // 2021-03-16

LocalDate parsedDate = LocalDate.parse(text, formatter);
```

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");
LocalDate date = LocalDate.now();

String text = date.format(formatter);
System.out.println(text); // 16/03/2021
```

DATE

Period modélise une quantité de temps en années, mois et jours

Pour créer une nouvelle instance : prase, of*

```
Period.parse(text);
Period.ofDays(days);
```

Pour obtenir des informations : get*

```
period.getDays();
```

Pour effectuer des opérations : plus*, minus*

```
period.plusDays(2);
period.minusYears(1);
```

Pour changer une information : with*

```
period.withDays(5);
```

DATE

Une période vide (0 jour) sera représenté « P0D »

Une période d'un an sera représenté « P1Y »

Une période de deux ans, 4 mois, 6 jours sera représenté « P2Y4M6D »

Une période de 4 ans et 50 jours sera représenté « P4Y50D »

DATE

```
LocalDate date = LocalDate.of(2021, 3, 16);
System.out.println(date);
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- L'éxécution échoue
- La compilation échoue
- Imprime « 2021-03-16 »
- Imprime « 16 Mar 2021 »

DATE

```
LocalTime time = LocalTime.of(0, 1, 2);
time.withHour(3).withMinute(4).withSecond(5);

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

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- L'éxécution échoue
- La compilation échoue
- Imprime « 00:01:02 »
- Imprime « 03:04:05 »

DATE

```
LocalDate date = LocalDate.of(2021, 3, 16);
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yy mm dd");

String formattedDate = date.format(formatter);
System.out.println(formattedDate);
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- L'éxécution échoue
- La compilation échoue
- Imprime « 21 00 16 »
- Imprime « 21 03 16 »

DATE

```
Period period = Period.parse("p01y");
System.out.println(period);
```

Que se passe-t-il lorsque ce code est compilé et exécuté ?

- L'éxécution échoue
- Imprime « p01y »
- Imprime « P1Y »
- Imprime « P01Y »

API JAVA

StringBuilder

String

Date

Lambda

Concurrency

LAMBDA

Les expressions lambda permettent de traiter une fonctionnalité en argument de méthode, ou du code en tant que donnée

Similaires aux classes anonymes, sauf qu'elles peuvent seulement être déclarées tel que l'implémentation des interfaces fonctionnelles

Trois éléments :

- Paramètres formels séparés par une virgule, entourés de paranthèses
- Une flèche (->)
- Un corps d'une instruction, ou d'un bloc d'instructions

```
(Integer i, Integer j) -> { return i == j; }
```

LAMBDA

Interfaces fonctionnelles pré-existantes

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

LAMBDA

```
public interface NumberComparator {
    boolean compare(Integer a, Integer b);
}
```

```
public static void compare(Map<Integer, Integer> map, NumberComparator comparator) {
    for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
        if (comparator.compare(entry.getKey(), entry.getValue())) {
            System.out.println(entry);
        }
    }
}
```

```
public static void main(String... args) {
    Map<Integer, Integer> map = new HashMap<>();
    map.put(0, 1);
    map.put(2, 2);

    compare(map, (Integer i, Integer j) -> {
        return i == j;
    });
}
```

LAMBDA

```
compare(map, (Integer i, Integer j) -> { return i == j; });
```

```
compare(map, (i, j) -> { return i == j; });
```

```
compare(map, (i, j) -> i == j);
```

LAMBDA

```
public static void doSomething(Boolean check) { }
```

Quelle instruction est valide pour invoquer doSomething ?

- `doSomething(() -> true);`
- `doSomething(() -> return true);`
- `doSomething(() -> { return new Boolean(true); });`
- Aucune

LAMBDA

```java
public interface ValueComparator {
    boolean compare(Integer a, Integer b);
}
```

Quelle instruction est valide pour déclarer ValueComparator ?

- `ValueComparator comparator = (i, j) -> i > j;`
- `ValueComparator comparator = (int i, int j) -> i > j;`
- `ValueComparator comparator = (Integer i, Integer j) -> return i > j;`
- Aucune

API JAVA

StringBuilder

String

Date

Lambda

Concurrency

CONCURRENCY

Threads et Runnables

```
Runnable task = () -> {
    String tName = Thread.currentThread().getName();
    System.out.println(tName);
};

task.run();

Thread thread = new Thread(task);
thread.start();
```

CONCURRENCY

Executors

- Peuvent gérer un pool de Threads
- Peuvent exécuter des tâches asynchrones

```
Runnable task = () -> {
    String tName = Thread.currentThread().getName();
    System.out.println(tName);
};

task.run();

ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(task);
```

CONCURRENCY

Callables et Futures

```
Callable<Integer> task = () -> {
    try {
        TimeUnit.SECONDS.sleep(1);
        return 123;
    }

    catch (InterruptedException e) {
        return 0;
    }
};

ExecutorService executor = Executors.newFixedThreadPool(1);
Future<Integer> future = executor.submit(task);

while (!future.isDone()) {
    System.out.println(future.get());
}
```

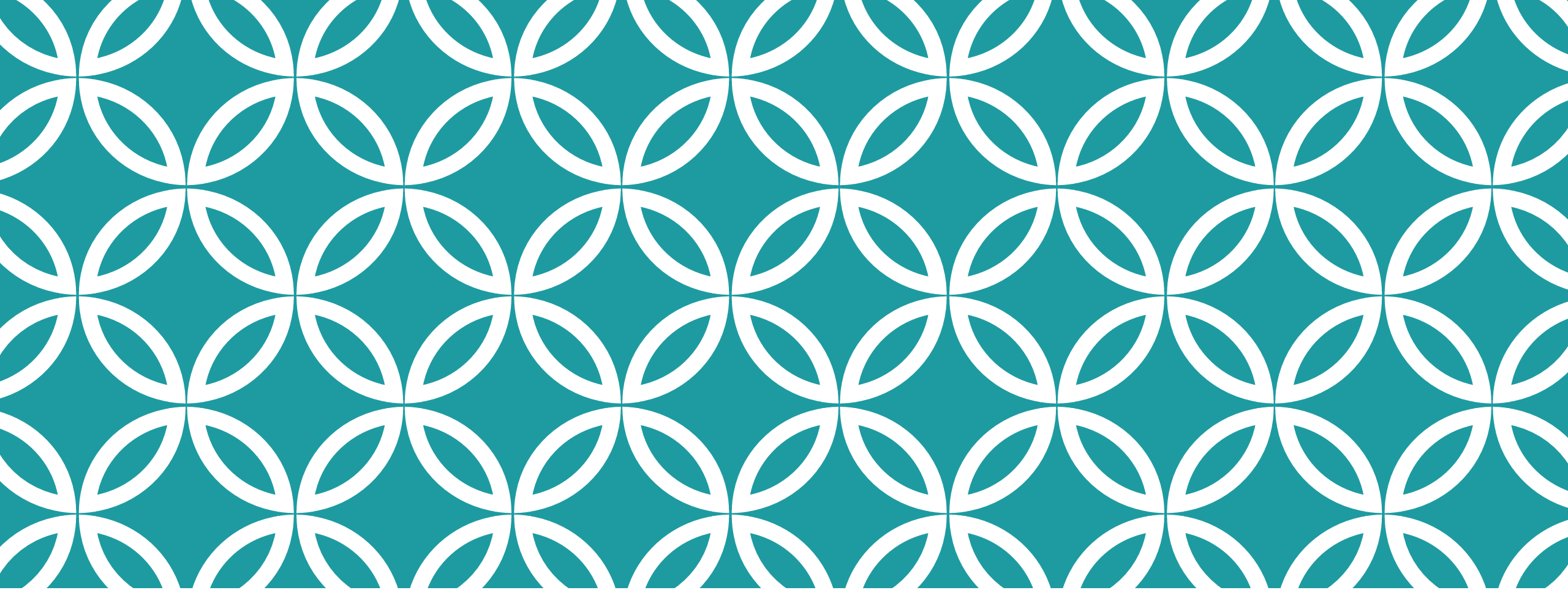
```
while (!future.isDone()) {
    System.out.println(future.get(500, TimeUnit.MILLISECONDS));
}
```

CONCURRENCY

ScheduledExecutors

- Ce sont des Executors, qui vont se jouer à une intervalle définie

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

Runnable task = () -> System.out.println(System.currentTimeMillis());
executor.scheduleAtFixedRate(task, 0, 5, TimeUnit.SECONDS);
```



ASTUCES

Quelques astuces

ASTUCES

Ne pas hésiter à refaire de petits exemples sur les nouveaux concepts, et à tester par soi-même

L'objectif de cette certification est de tester vos compétences dans les fondamentaux de Java, des concepts de programmation orientée objet, et des fonctionnalités propres à Java 8 (Lambda, Date, etc.)

Il n'y a pas beaucoup de questions basées sur le par cœur. Pratiquez et évitez de mémoriser chaque détail d'une API en lisant seulement

Une question peut ne pas concerner un seul objectif, mais tester votre expertise sur des sujets variés

Soyez vigilant sur les déclarations des variables (scopes), les boucles while et do-while

ASTUCES

Souvenez-vous de l'ordre des instructions dans un fichier (package, imports, déclarations)

Attention à l'ordre des paramètres lorsqu'il y a du varargs, il doit être en dernier

Lors qu'il y a implémentation d'interface, vérifier la portée des méthodes et si les méthodes sont toutes bien implémentées, sinon vérifier si la classe est abstraite

Soyez vigilant sur les overriding, overloading, surcharges de constructeurs avec paramètres

Bien connaitre les différences entre ++expr, expr++, &, &&, |, ||