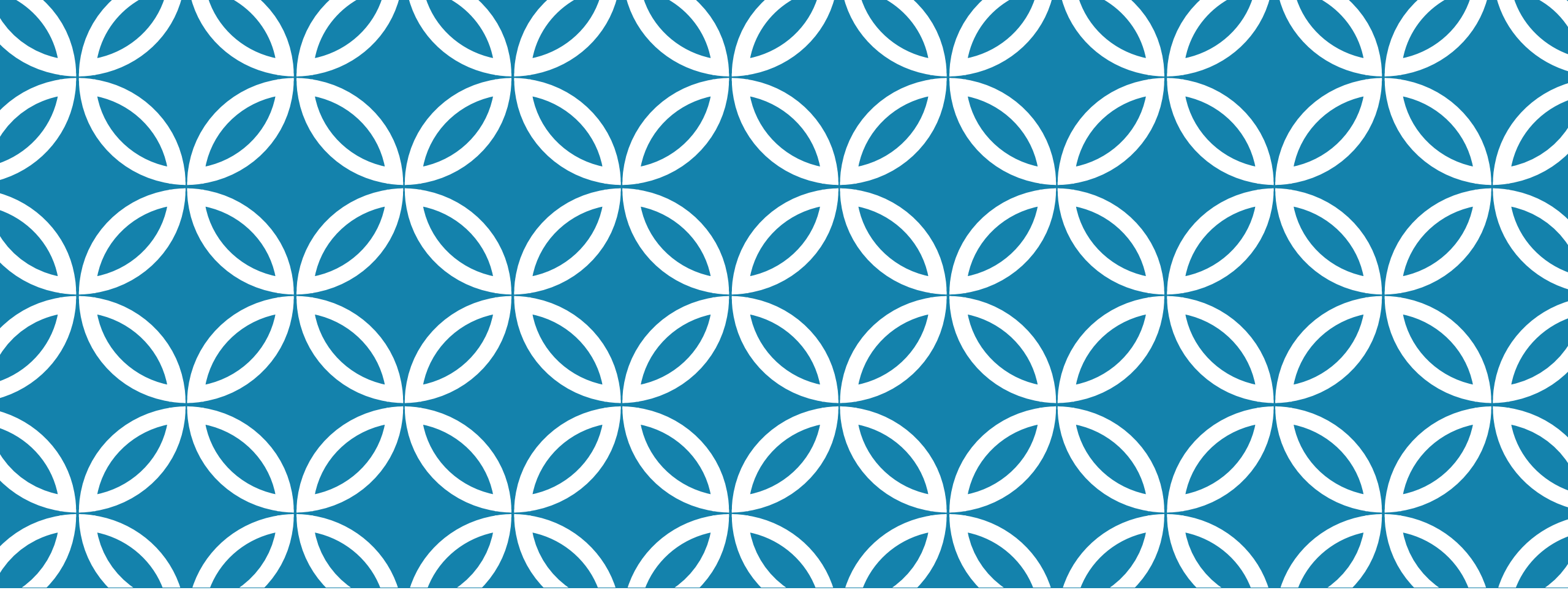
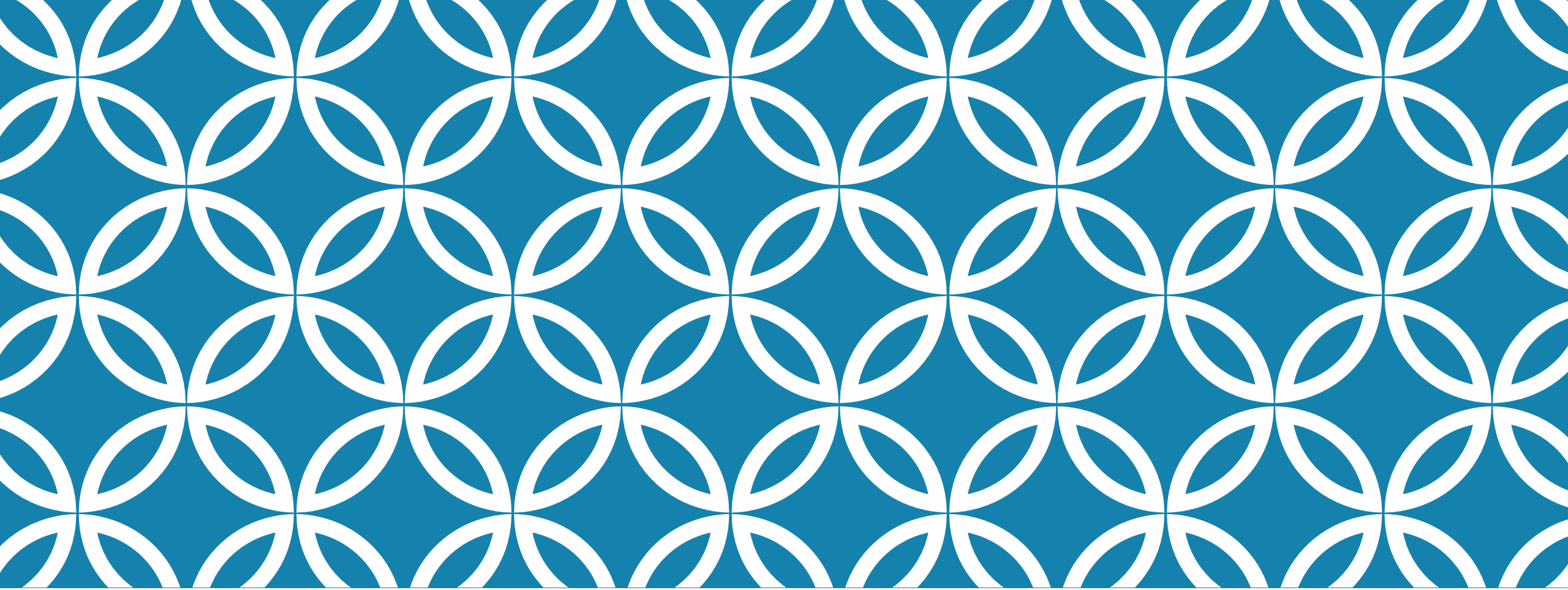


FORMATION JAVA

Jordan ABID



DÉVELOPPEMENT JAVA

QU'EST CE QUE JAVA?

Java est utilisé pour créer :

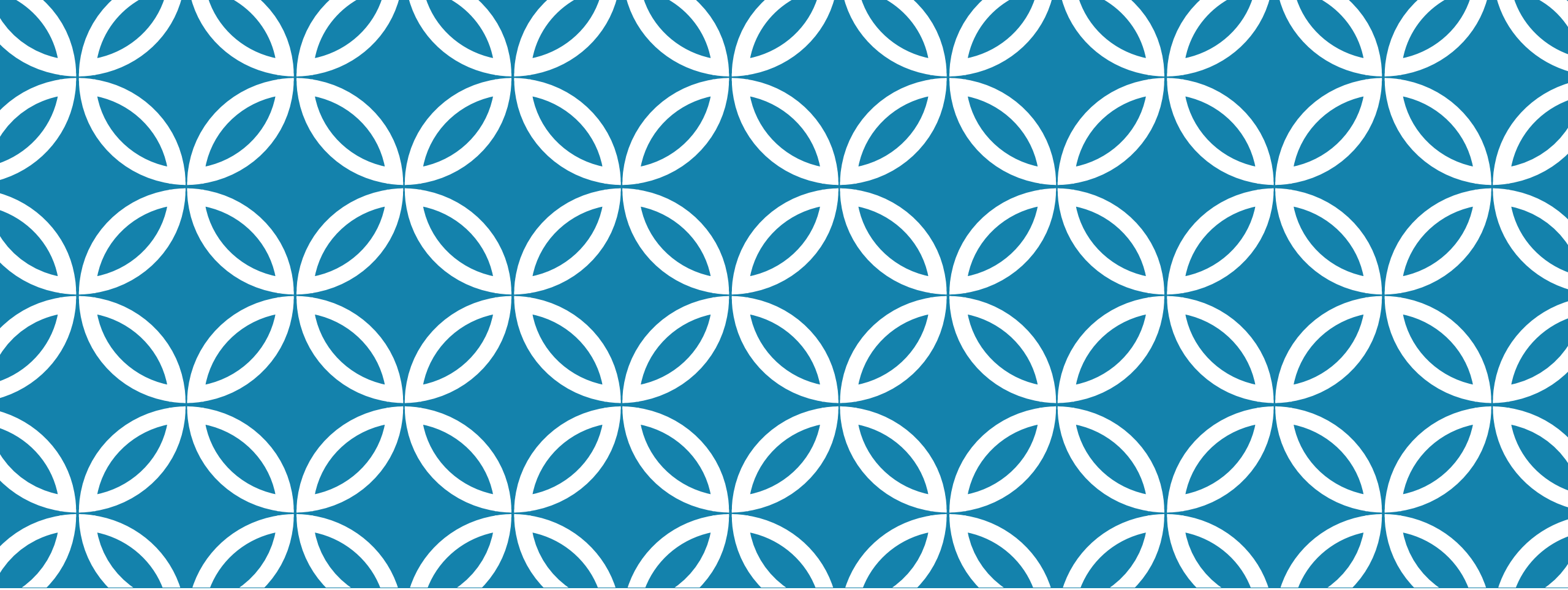
- Des applications Desktop
- Des applets java (applications java destinées à s'exécuter dans une page web)
- Des applications pour les smart phones
- Des applications embarquées dans des cartes à puces
- Des application JEE (Java Entreprise Edition)

Pour créer une application java, il faut installer un kit de développement java

- JSDK : Java Standard Developpement Kit, pour développer les applications Desktop
- JME : Java Mobile Edition, pour développer les applications pour les téléphones potables
- JEE : Java Entreprise Edition, pour développer les applications qui vont s'exécuter dans un serveur d'application JEE (Web Sphere Web Logic, JBoss).
- JCA : Java Card Editon, pour développer les applications qui vont s'exécuter dans des cartes à puces.

Au niveau syntaxe, Java est un langage de programmation qui ressemble beaucoup au langage C++

Toute fois quelques simplifications ont été apportées à java pour des raisons de sécurité et d'optimisation.



PROGRAMMATION ORIENTÉE OBJET AVEC JAVA

MÉTHODE ORIENTÉE OBJET

La méthode orientée objet permet de concevoir une application sous la forme d'un ensemble d'objets reliés entre eux par des relations

L’une des caractéristiques de cette méthode permet de concevoir de nouveaux objets à partir d'objets existants.

On peut donc réutiliser les objets dans plusieurs applications.

La réutilisation du code fut un argument déterminant pour venter les avantages des langages à objets.

Pour faire de la programmation orientée objet il faut en maitriser les fondamentaux, à savoir:

- **Objet et classe**
- **Encapsulation (Accessibilité)**
- **Héritage**
- **Polymorphisme**

OBJET

Un objet est une structure informatique définie par un état et un comportement

Objet = identité + état + comportement

L'état regroupe les valeurs instantanées de tous les attributs de l'objet.

Le comportement regroupe toutes les compétences et décrit les actions et les réactions de l'objet. Autrement dit le comportement est défini par les opérations que l'objet peut effectuer.

L'état d'un objet peut changer dans le temps.

Généralement, c'est le comportement qui modifie l'état de l'objet

IDENTITÉ D'UN OBJET

En plus de son état, un objet possède une identité qui caractérise son existence propre.

Cette identité s'appelle également référence ou handle de l'objet

En terme informatique de bas niveau, l'identité d'un objet représente son adresse mémoire.

Deux objets ne peuvent pas avoir la même identité : c'est-à-dire que deux objet ne peuvent pas avoir le même emplacement mémoire

CLASSE

Les objets qui ont des caractéristiques communes sont regroupés dans une entité appelé classe.

La classe décrit le domaine de définition d'un ensemble d'objets.
Chaque objet appartient à une classe

Les généralités sont contenues dans les classe et les particularités dans les objets.

Les objets informatique sont construits à partir de leur classe par un processus qui s'appelle l'instanciation.

Tout objet est une instance d'une classe.

CARACTÉRISTIQUES D'UNE CLASSE

Une classe est définie par :

- Ses attributs
- Ses constructeurs
- Ses méthodes

Les attributs permettent de décrire l'état des objets de cette classe, chaque attribut est défini par :

- Son nom
- Son type
- Eventuellement sa valeur initiale

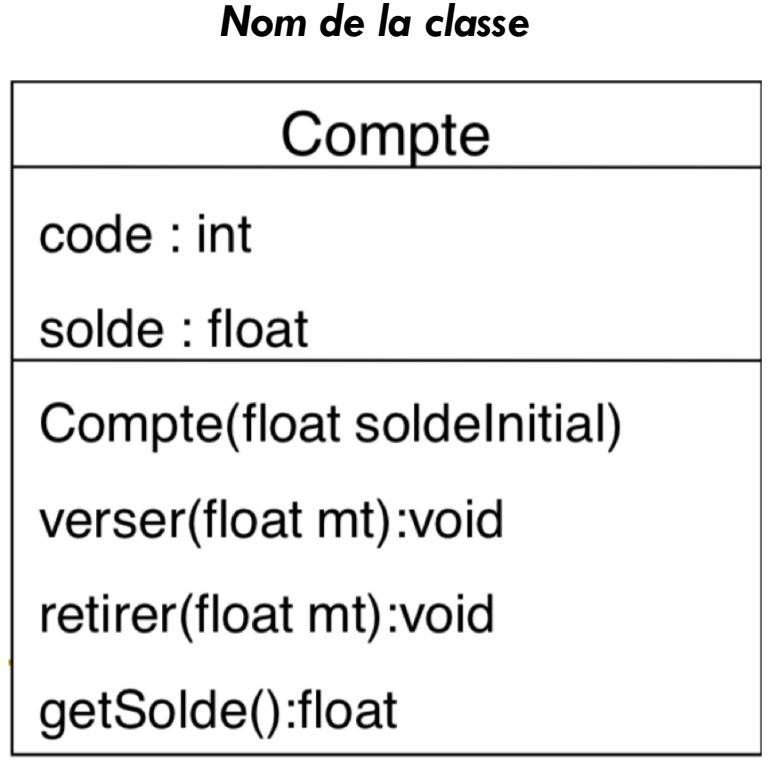
Les méthodes permettent de décrire le comportement des objets de cette classe.
Une méthode représente une procédure ou une fonction qui permet d'exécuter un certain nombre d'instructions.

Des méthodes qui sont appelées au moment de la création d'un objet de cette classe.
Ces méthodes sont appelées **CONSTRUCTEURS**

REPRÉSENTATION UML D'UNE CLASSE

Une classe est représenté par un rectangle à 3 compartiments:

- Un compartiment qui contient le nom de la classe
- Un compartiment qui contient la déclaration des attributs
- Un compartiment qui contient les méthodes



ACCESSIBILITÉ AU MEMBRES D'UNE CLASSE

Dans java, il existe 4 **niveaux** de **protection** :

- **private** (-) : Un membre privé d'une classe n'est accessible qu'à l'intérieur de cette classe.

- protected (#) : un membre protégé d'une classe est accessible à :

- L'intérieur de cette classe
- Aux classes dérivées de cette classe.
- Aux classes du même package.

- public (+) : accès à partir de toute entité interne ou externe à la classe

- Autorisation par défaut : dans java, en l'absence des trois autorisations précédentes, l'autorisation par défaut est **package.** Cette autorisation indique que uniquement les classes du même package ont l'autorisation d'accès.

EXEMPLE D'IMPLÉMENTATION D'UNE CLASSE

```
public class Compte {
// Attributs
private int code;
protected float solde;

// Constructeur
public Compte(int c, float s)
{
  code=c;
  solde=s;
}

// Une méthode qui retourne l'état du compte
public String toString()
{
     return(" Code="+code+" Solde="+solde);
}
```

CRÉATION DES OBJETS DANS JAVA

Dans java, pour créer un objet d'une classe , On utilise la commande new suivie du constructeur de la classe.
La commande new Crée un objet dans l'espace mémoire et retourne l'adresse mémoire de celui-ci.

```
public class Application {
    public static void main(String[] args)
    {
        Compte c1=new Compte(1,5000);
        Compte c2=new Compte(2,6000);
        System.out.println(c1.toString());
     }
}
```

CONSTRUCTEUR PAR DÉFAUT

Quand on ne définit aucun constructeur pour une classe, le compilateur crée le constructeur par défaut.

Le constructeur par défaut n'a aucun paramètre et ne fait aucune initialisation

```
public class Personne {
 private int code;
 private String nom;
}
```

```
Personne p=new Personne();
```

GETTERS ET SETTERS

Les attributs privés d'une classe ne sont accessibles qu'à l'intérieur de la classe.

Pour donner la possibilité à d'autres classes d'accéder aux membres privés, il
faut définir dans la classe des méthodes publiques qui permettent de :

- lire les variables privés. Ce genre de méthodes s'appellent les accesseurs ou Getters
- modifier les variables privés. Ce genre de méthodes s'appellent les mutateurs ou Setters

Les getters retourne toujours le même type que l'attribut correspondant.

Les setters sont toujours de type void et reçoivent un paramètre qui est de même type que la variable

EXEMPLE D'IMPLÉMENTATION D'UNE CLASSE

```
public class Compte {
// Attributs
private int code;
protected float solde;
// Constructeur
public Compte(int c, float s)
{
  code=c;
  solde=s;
}
// Getters des attributs code et solde
public int getCode(){return code;}
public float getSolde(){return solde;}
// Setters des attributs code et solde
public void setCode(int code){this.code=code;}
public void setSolde(float solde){this.solde=solde;}
// Une méthode qui retourne l'état du compte
public String toString()
{
    return(" Code="+code+" Solde="+solde);
}
```

SURCHARGE

Dans une classe, on peut définir plusieurs constructeurs. Chacun ayant une signature différente (paramètres différents)

On dit que le constructeur est surchargé

On peut également surcharger une méthode. Cela veut dire qu'on peut définir, dans la même classe plusieurs méthodes qui ont le même nom et des signatures différentes;

La signature d'une méthode désigne la liste des arguments avec leurs types.

Dans la classe Compte, par exemple, on peut ajouter un autre constructeur sans paramètre

Un constructeur peut appeler un autre constructeur de la même classe en utilisant le mot `this()` avec des paramètres éventuels

EXEMPLE D'IMPLÉMENTATION D'UNE CLASSE

```
public class Compte {
  private float decouvert;
  //Premier constructeur
  public Compte (float decouvert)
 {
    this.decouvert=decouvert;
 }
 //Deuxième constructeur
 public Compte()
 {
    this(0);
 }
}
```

```
Compte c1= new Compte (5000);
Compte c2= new Compte ();
```

MEMBRES STATIQUES D'UNE CLASSE

Les membres statiques d'une classes sont des membres qui appartiennent à la classe et sont partagés par toutes les instances de cette classe.

- Les membres statiques ne sont pas instanciés lors de l'instanciation de la classe
- Les membres statiques sont accessible en utilisant directement le nom de la classe qui les contient.

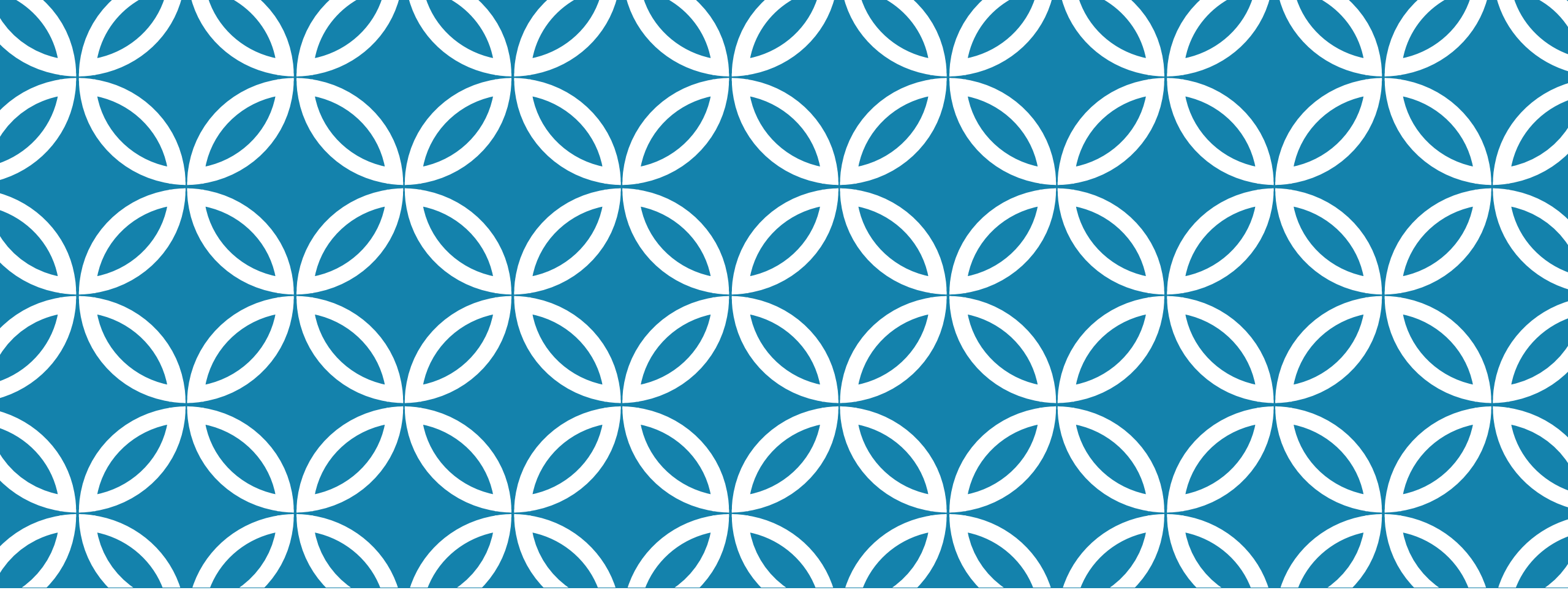
Exemple d'utilisation :

```
public class Application {

public static void main(String[] args)
    {
        Compte c1=new Compte(1,5000);
        Compte c2=new Compte(2,6000);
        System.out.println(c1.getCode);
        System.out.println(Compte.getCpt());
     }
```

```
public class Compte {
  private int code;
  private static int cpt=1;

  public Compte()
  {
   code= cpt;
  cpt ++;
  }
  public int getCode()
  {return code;}
  public static int getCpt()
 {return cpt;}
}
```



ENUMÉRATIONS ET COLLECTIONS

LES ENUMÉRATIONS

Ce n'est pas une classe, c'est une liste d'options.
Type qui permet de manipuler des catégories de manière pratique (plus facile à gérer qu'un tableau par ex.)
Constitué d'une liste de possibilité

Chaque option peut être associée à une ou plusieurs valeurs
Instances crées au chargement de l'enum, on ne peut pas en créer d'autres, appelle le constructeur avec les valeurs définis dans l'enum

```
enum Langage {
java(1000), csharp(2000), cplusplus;

private int prix;
private Langage(){}
private Langage (int prix){ this.prix=prix;}

public String toString(){
return this.name()" coute "+ prix;} }
```

LES ENUMÉRATIONS - MÉTHODES

- **name()** : renvoie la chaine de caractère du nom de l'option
- ***values() : renvoie le tableau des options possibles***
- ***valueOf : converti une chaine de caractère en l'option correspondante***

On peut coder ses méthodes comme pour une classe, (getPrix(), toString,...)

COLLECTIONS - DÉFINITION

Une collection est un tableau dynamique d'objets de type Object.

Une collection fournit un ensemble de méthodes qui permettent:

- D'ajouter un nouvel objet dans le tableau
- Supprimer un objet du tableau
- Rechercher des objets selon des critères
- Trier le tableau d'objets
- Contrôler les objets du tableau…

Dans un programme, les tableaux peuvent être utilisés quand la dimension du tableau est fixe.

Dans le cas contraire, il faut utiliser les collections
Java fournit plusieurs types de collections:

- ArrayList
- Vector
- Iterator
- HashMap…

COLLECTIONS – ARRAYLIST

ArrayList est une classe du package java.util, qui implémente l'interface List.

Déclaration d'une collection de type List qui devrait stocker des objets de type Eleve :

```
//Création de la liste:
List<Eleve> lst=new ArrayList<Eleve>();
//Ajouter deux objets de type Eleve à la liste:
lst.add(new Eleve());
lst.add(new Eleve());
```

Faire appel à la méthode affiche() de tous les objets de la liste:

En utilisant la boucle classique for

```
for(int i=0;i<lst.size();i++)
{lst.get(i).affiche(); }
```

En utilisant la boucle for each

```
for(Eleve e: lst)
{e.affiche(); }
```

Supprimer le deuxième Objet de la liste

```
lst.remove(1);
```

COLLECTIONS – FILE D'ATTENTE

Pour manipuler des files d'attentes
FIFO : First In First Out : premier arrivé premier sorti
add() ajoute des éléments en fin de liste.
size() : le nombre d' éléments dans la liste,...
poll() : supprime le premier élément de la liste et le renvoie
peek() : renvoie le premier élément sans le supprimer de la liste

```
LinkedList<String> queue = new LinkedList<String>();
queue.add("1er patient");
queue.add("2eme patient");
queue.add("dernier patient");
String a = queue.poll(); // a vaut "1er patient"
String b = queue.peek(); // b vaut  "2eme patient"
String c = queue.peek(); // c vaut  "2eme patient"
```

COLLECTIONS – HASHMAP

La collection HashMap est une classe qui implémente l'interface Map. Cette collection permet de créer un tableau dynamique d'objet de type Object qui sont identifies par une clé.

Déclaration et création d'une collection de type HashMap qui contient des élèves identifiés par une clé de type String :

```
Map<String, Eleve> lst=new
HashMap<String, Eleve>();
```

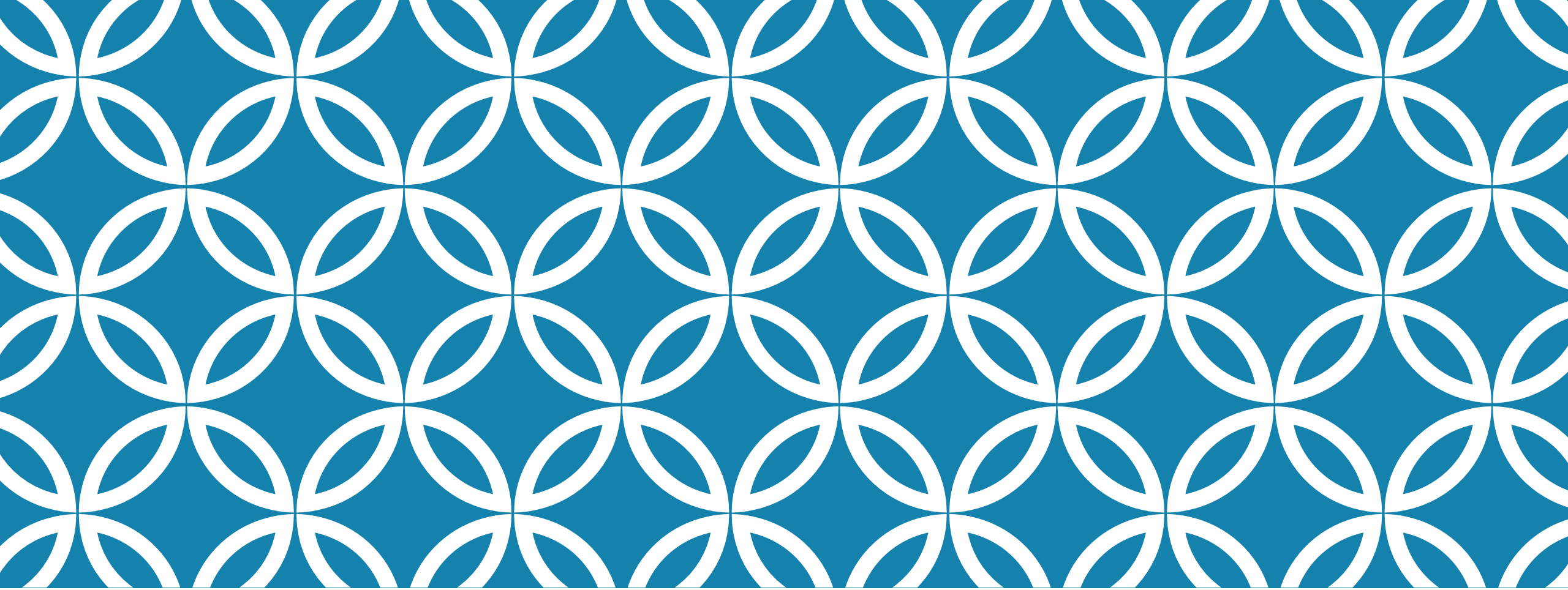
Ajouter deux objets de type Eleve à la collection

```
lst.put("p1", new Eleve());
lst.put("p2", new Eleve());
```

Récupérer un objet ayant pour clé "p1"

```
Eleve e=lst.get("p1");
e.affiche();
```

```
Iterator<String>
it=lst.keySet().iterator();
while(it.hasNext()){
  String key=it.next();
  Eleve ee=lst.get(key);
  System.out.println(key);
  ee.affiche();
}
```



HÉRITAGE ET INTERFACES

HÉRITAGE

Dans la programmation orientée objet, l'héritage offre un moyen très efficace qui permet la réutilisation du code.

En effet une classe peut hériter d'une autre classe ses attributs et ses méthodes.

L'héritage, quand il peut être exploité, fait gagner beaucoup de temps en terme de développement et en terme de maintenance des applications.

La réutilisation du code fut un argument déterminant pour venter les méthodes orientées objets.

EXEMPLE DE PROBLÈME

Supposons que nous souhaitions créer une application qui permet de manipuler différents types de comptes bancaires: les ***compte simple***, les ***comptes épargnes*** et les ***comptes payants***.

Tous les types de comptes sont caractérisés par:
Un **code** et un **solde**

Lors de la création d'un compte, son code qui est défini automatiquement en fonction du nombre de comptes crées

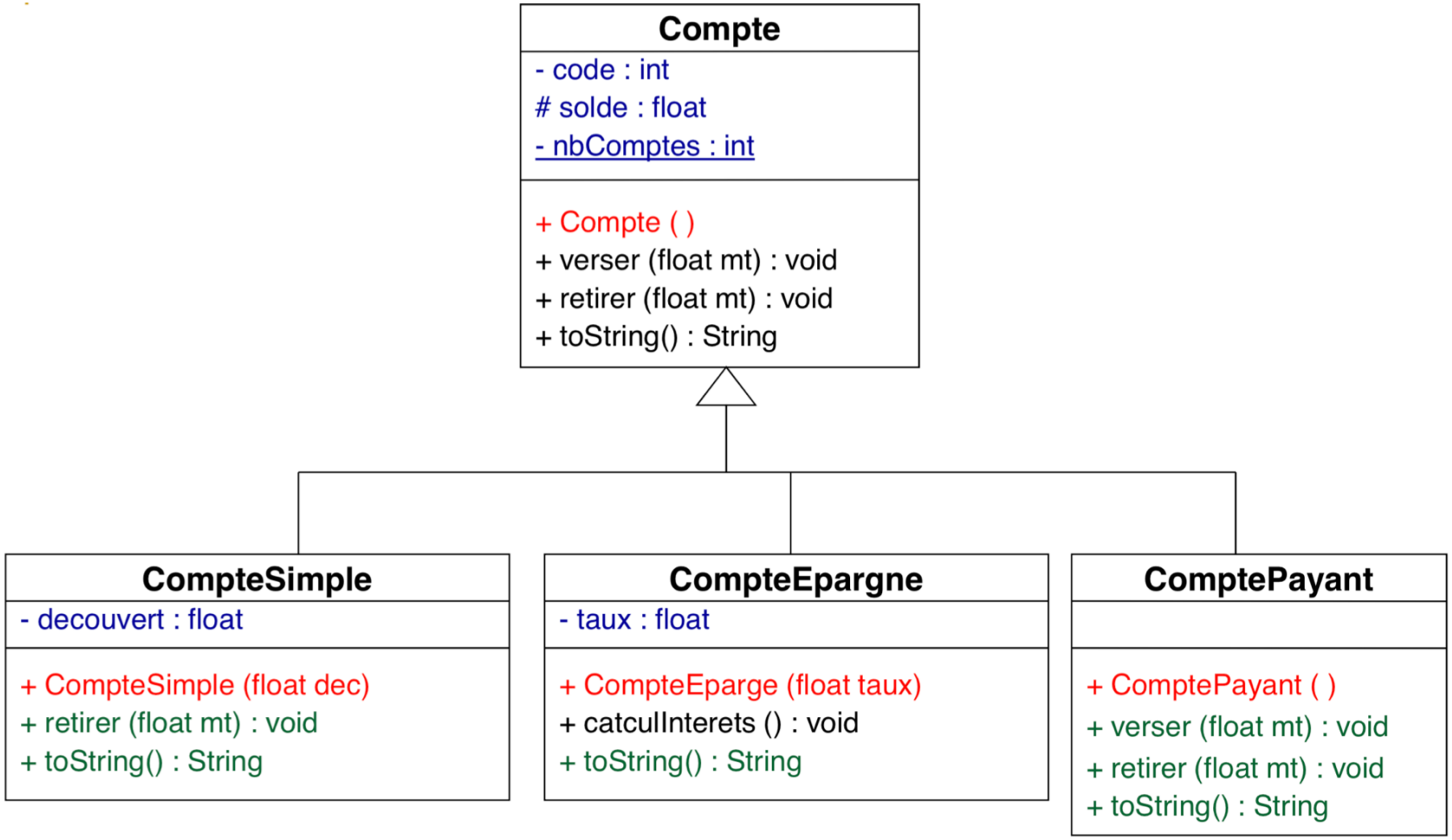
Un compte peut subir les opérations de versement et de retrait. Pour ces deux opérations, il faut connaitre le montant de l'opération.

Un compte simple est un compte qui possède un découvert. Ce qui signifie que ce compte peut être débiteur jusqu'à la valeur du découvert.

Un compte Epargne est un compte bancaire qui possède en plus un champ «tauxInteret» et une méthode calculInteret() qui permet de mettre à jour le solde en tenant compte des intérêts.

Un ComptePayant est un compte bancaire pour lequel chaque opération de retrait et de versement est payante et vaut 5 % du montant de l'opération.

DIAGRAMME DE CLASSES



IMPLÉMENTATION JAVA DE LA CLASSE COMPTE

```
public class Compte {
  private int code;
  protected float solde;
  private static int nbComptes;

  public Compte( ){
  nbComptes++;
  code=nbComptes;
  this.solde=0; }

  public void verser(float mt){solde+=mt;}
  public void retirer(float mt)
  {
    if(mt<solde) {solde-=mt;}
  }
  public String toString()
  {
    return("Code="+code+" Solde="+solde);
  }
}
```

HÉRITAGE - EXTENDS

La classe CompteSimple est une classe qui hérite de la classe Compte.

Pour designer l'héritage dans java, on utilise le mot extends

```
public class CompteSimple extends Compte {}
```

La classe CompteSimple hérite de la classe Compte tous ses membres sauf le constructeur.

Dans java une classe hérite toujours d'une seule classe.

Si une classe n'hérite pas explicitement d'une autre classe, elle hérite implicitement de la classe Object.

La classe Compte hérite de la classe Object.

La classe CompteSimple hérite directement de la classe Compte et indirectement de la classe Object.

DÉFINIR LES CONSTRUCTEURS DE LA CLASSE DÉRIVÉE

Le constructeur de la classe dérivée peut faire appel au constructeur de la classe parente en utilisant le mot super() suivi de ses paramètres.

```
public class CompteSimple extends Compte {
 private float decouvert;
 //constructeur
 public CompteSimple(float decouvert)
 {
  super();
  this.decouvert=decouvert;
 }
}
```

REDÉFINITION DES MÉTHODES

Quand une classe hérite d'une autre, elle peut redéfinir les méthodes héritées.

Dans notre cas la classe CompteSimple hérite de la classe Compte la méthode retirer()

Nous avons besoin de redéfinir cette méthode pour prendre en considération la valeur du découvert.

```
public class CompteSimple extends Compte {
 private float decouvert;
 // constructeur
 public CompteSimple(float decouvert)
 {
  super();
  this.decouvert=decouvert;
 }
 // Redéfinition de la méthode retirer
 public void retirer(float mt)
 {
  if(mt-decouvert<=solde) solde-=mt;
 }
}
```

CLASSE ABSTRAITE

Une classe abstraite est une classe qui ne peut pas être instanciée.

La classe Compte de notre modèle peut être déclarée abstract pour indiquer au compilateur que cette classe ne doit pas être instancié.

Une classe abstraite est généralement créée pour en faire dériver de nouvelles classes par héritage.

```
public abstract class Compte {
 private int code;
 protected float solde;
 private static int nbComptes;
 // Constructeurs
 // Méthodes
}
```

LES MÉTHODES ABSTRAITES

Une méthode abstraite peut être déclarée à l'intérieur d'une classe abstraite.

Une méthode abstraite est une méthode qui n'a pas de définition.
Une méthode abstraite est une méthode qui doit être redéfinie dans les classes dérivées.

Exemple :
On peut ajouter à la classe Compte une méthode abstraite nommée afficher() pour indiquer que tous les comptes doivent redéfinir cette méthode.

```
public abstract class Compte {
 ...
 // Méthode abstraite
 public abstract void afficher();
}
```

```
public class CompteSimple extends Compte {
  ...
 public void afficher(){ System.out.println("Solde="+solde+"
 Découvert="+decouvert); }
}
```

CLASSE DE TYPE FINAL

Une classe de type final est une classes qui ne peut pas être dérivée.
Autrement dit, on ne peut pas hériter d'une classe final.

La classe CompteSimple peut être déclarée final en écrivant :

```
public final class CompteSimple extends Compte {
 private float decouvert;
 public void afficher()
 {
  System.out.println("Solde="+solde+"Découvert="+decouvert);
 }
}
```

VARIABLES ET MÉTHODES FINAL

Une variable final est une variable dont la valeur ne peut pas changer. Autrement dit, c'est une constante :
Exemple : final double PI=3.14;

Une méthode final est une méthode qui ne peut pas être redéfinie dans les classes dérivées.
Exemple : La méthode verser de la classe suivante ne peut pas être redéfinie dans les classes dérivées car elle est déclarée final

```
public class Compte {
 private int code;
 protected float solde;
 private static int nbComptes;

 public final void verser(float mt){ solde+=mt;}
 public void retirer(float mt){
 if(mt<solde) {solde-=mt; }
 }
}
```

INTERFACES

Une interface est une sorte de classe abstraite qui ne contient que des méthodes abstraites.

Dans java une classe hérite d'une seule classe mais peut hériter en même temps de plusieurs interfaces.

On dit qu'une classe implémente une ou plusieurs interfaces.

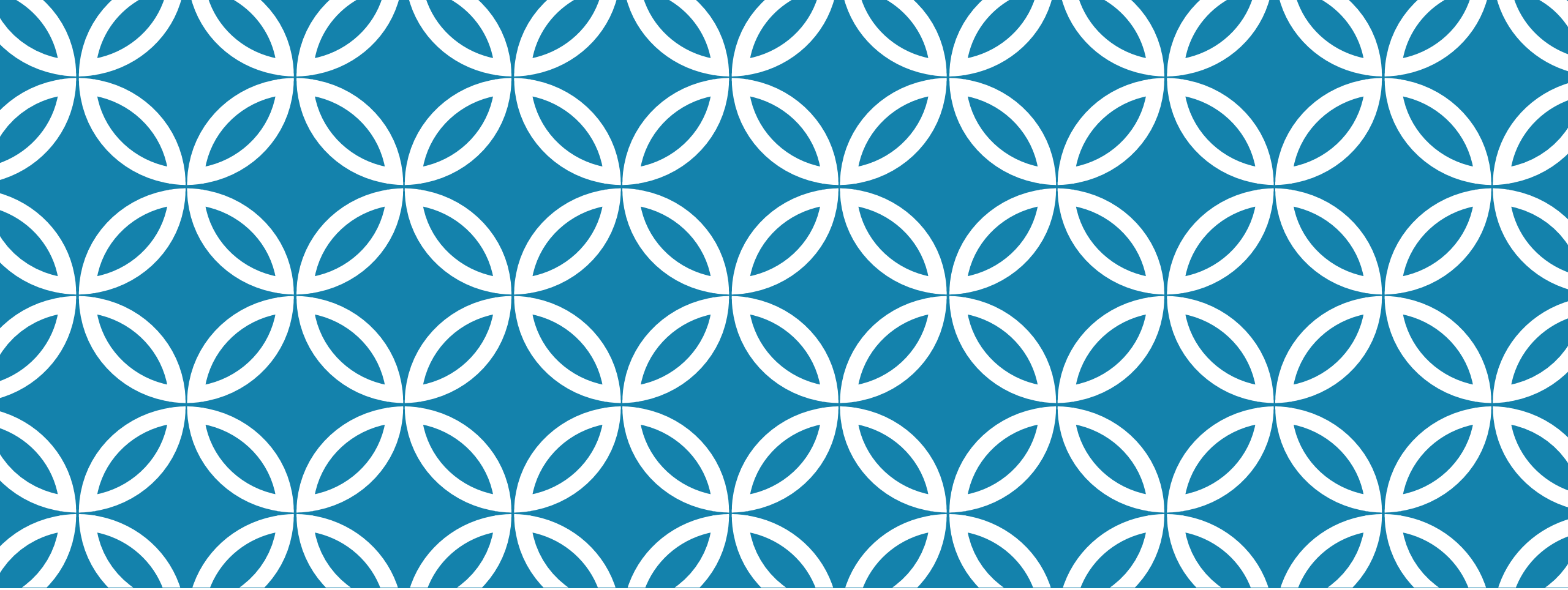
Une interface peut hériter de plusieurs interfaces. Exemple d'interface:

Pour indiquer que la classe CompteSimple implémente cette interface on peut écrire :

```
public interface Solvable {
public void solver();
public double getSolde(); }
```

```
public class CompteSimple extends Compte implements Solvable {
 private float decouvert;
 public void afficher()
 {System.out.println("Solde="+solde+" Découvert="+decouvert); }

 public double getSolde() { return solde;}
 public void solver() { this.solde=0; }
}
```



TRY/CATCH ET EXCEPTIONS

LES EXCEPTIONS

En Java, on peut classer les exceptions en deux catégories :

- Les exceptions **surveillées**
- Les exceptions **non surveillées**

Java oblige le programmeur à traiter les erreurs surveillées. Elles sont signalées par le compilateur

Les erreurs non surveillées peuvent être traitées ou non. Et ne sont pas signalées par le compilateur

En java, pour traiter les exceptions, on doit utiliser le bloc try catch de la manière suivante:

```
try
{
   resultat=divison(a, b);
}
catch (ArithmeticException e) { System.out.println("Division par zero");}
```

PRINCIPALE MÉTHODES D'UNE EXCEPTION

Tous les types d'exceptions possèdent les méthodes suivantes

- getMessage() : retourne le message de l'exception

```
System.out.println(e.getMessage());
Résultat affiché : / by zero
```

- toString() : retourne une chaine qui contient le type de l'exception et le message de l'exception.

```
System.out.println(e.toString());
Résultat affiché : java.lang.ArithmeticException: / by zero
```

- printStackTrace: affiche la trace de l'exception

```
e.printStackTrace();
Résultat affiché :
    java.lang.ArithmeticException: / by zero at App1.calcul(App1.java:4) at App1.main(App1.java:13)
```

GÉNÉRER ET JETER UNE EXCEPTION SURVEILLÉE DE TYPE EXCEPTION

Considérons le cas d'un compte qui est défini par un code et un solde et sur lequel, on peut verser un montant, retirer un montant et consulter le solde.

```
public class Compte {
 private int code;
 private float solde;

 public void verser(float mt)
 {
    solde=solde+mt;
 }
 public void retirer(float mt)throws Exception
 {
  if(mt>solde) {throw new Exception("Solde Insuffisant");}
  solde=solde-mt;
 }
```

TRAITER L'EXCEPTION

Deux solutions pour traiter cette exception :

Soit utiliser le bloc try catch

```
try
{
   cp.retirer(mt2);
}
catch (Exception e){ System.out.println(e.getMessage());}
```

Ou déclarer que cette exception est ignorée dans la méthode main et dans ce cas là, elle remonte vers le niveau supérieur. Dans notre cas la JVM.

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

En générant une Exception de type RunTimeException, le bloc Try catch devient optionnel

PERSONNALISER LES EXCEPTIONS MÉTIER

L'exception générée dans la méthode retirer, dans le cas ou le solde est insuffisant est une exception métier.
Il plus professionnel de créer une nouvelle Exception nommée SoldeInsuffisantException de la manière suivante :

```
public class SoldeInsuffisantException extends Exception {
  public SoldeInsuffisantException(String message)
  {super(message);}
}
```

En héritant de la classe Exception, nous créons une exception surveillée.

Pour créer une exception non surveillée, vous pouvez hériter de la classe RuntimeException

GÉNÉRER ET JETER NOS PROPRES EXCEPTIONS

Utilisation de cette nouvelle exception métier

```
public class Compte {
 private int code;
 private float solde;

 public void verser(float mt)
 {
    solde=solde+mt;
 }
 public void retirer(float mt)throws SoldeInsuffisantException
 {
  if(mt>solde) {throw new SoldeInsuffisantException("Solde Insuffisant");}
  solde=solde-mt;
 }
```

LE BLOC FINALY

La syntaxe complète du bloc try est la suivante :

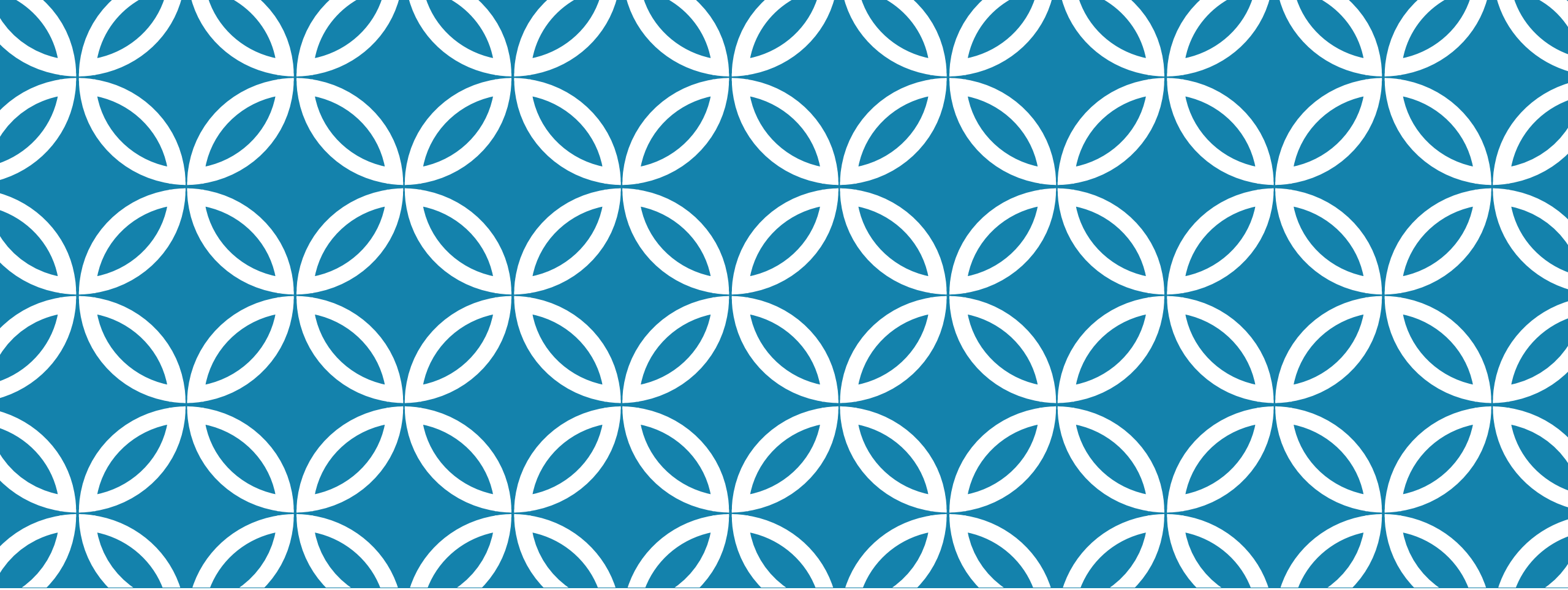
```
try {System.out.println("Traitement Normale"); }

catch (SoldeInsuffisantException e) {
   System.out.println( "Premier cas Exceptionnel" );
 }

catch (NegativeMontantException e) {
  System.out.println( "deuxième cas Exceptionnel" );
 }

finally{System.out.println( "Traitement par défaut!" ); }


System.out.println( "Suite du programme!"   );
```



ENTRÉES/SORTIES

PRINCIPE DES ENTRÉES / SORTIES

Pour effectuer une entrée ou une sortie de données en Java, le principe est simple et se résume aux opérations suivantes :

- Ouverture d'un moyen de communication.
- Ecriture ou lecture des données.
- Fermeture du moyen de communication.

La classe File permet de donner des informations sur un fichier ou un répertoire

La création d'un objet de la classe File peut se faire de différentes manières :

```
File f1=new File("c:/projet/fichier.ext");

File f2=new File("c:/projet", " fichier.ext");

File f3=new File("c:/projet");
```

PRINCIPALES MÉTHODES DE LA CLASSE FILE

String getName();	Retourne le nom du fichier.
String getPath();	Retourne la localisation du fichier en relatif.
String getAbsolutePath();	Idem mais en absolu.
String getParent();	Retourne le nom du répertoire parent.
boolean renameTo(File newFile);	Permet de renommer un fichier.
boolean exists() ;	Est-ce que le fichier existe ?
boolean canRead();	Le fichier est t-il lisible ?
boolean canWrite();	Le fichier est t-il modifiable ?
boolean isDirectory();	Permet de savoir si c'est un répertoire.
boolean isFile();	Permet de savoir si c'est un fichier.
long length();	Quelle est sa longueur (en octets) ?
boolean delete();	Permet d'effacer le fichier.
boolean mkdir();	Permet de créer un répertoire.

LIRE ET ÉCRIRE SUR UN FICHIER TEXTE

```
File f1=new File("c:/Fichier1.txt");
FileReader fr=new FileReader(f1);                    //Créer un flux de lecture vers f1

File f2=new File("c:/Fichier2.txt");                 //Créer le fichier f2 si il n'existe pas
FileWriter fw=new FileWriter(f2);                    //Créer un flux de lecture vers f2

int c;
while((c=fr.read())!=-1)                             /*La lecture d'un fichier se fait en parcourant
{                                                    chaque caractère.  read() retourne -1 lorsque
   fw.write(c);                                      le fichier est terminé*/
}

fr.close();
fw.close();
```

LA SÉRIALISATION

La sérialisation est une opération qui permet d'envoyer un objet sous forme d'une tableau d'octets dans une sortie quelconque (Fichier, réseau, port série etc..)

Les applications distribuées utilisent beaucoup ce concept pour échanger les objets java entre les applications via le réseau.

Pour sérialiser un objet, on utiliser la méthode **writeObject()** de la classe **ObjectOutputStream**

La désérialisation est une opération qui permet de reconstruire l'objet à partir d'une série d'octets récupérés d'une entrée quelconque.

Pour dé sérialiser un objet, on utilise la méthode **readObject()** de la classe **ObjectInputStream**.

Pour pouvoir sérialiser un objet, sa classe doit implémenter l'interface **Serializable**

Pour designer les attributs d'un objet qui ne doivent pas être sérialisés, on doit les déclarer **transient**

SÉRIALISATION / DÉSÉRIALISATION

```
Eleve e1=new Eleve("Jordan");
Eleve e2=new Eleve("Jeremy");

File f=new File( classe.txt");
FileOutputStream fos=new FileOutputStream(f);
ObjectOutputStream oos=new ObjectOutputStream(fos);

oos.writeObject(e1);
oos.writeObject(e2);

oos.close();
```

```
File f=new File("classe.txt");
FileInputStream fis=new FileInputStream(f);
ObjectInputStream ois=new ObjectInputStream(fis);
Eleve e1=(Eleve) ois.readObject();
Eleve e2=(Eleve) ois.readObject();
```