Classes

Une classe est composée de deux parties. Les champs (les futurs données des objets de cette classe) et les méthodes (ce que pourront faire les objets de cette classe)

Champs :
private Type nomDuChamp

Méthodes :
public TypeRetour nomFct(Type1 entrée1, …) {
  //code
}

Les types peuvent être soit de base (int, boolean, ..) soit complexe (d'autre Classes).

```
private  String nom;
private  String prenom;
private  String email;
private int age;

public  boolean  estPlusVieuxQue(int  autreAge) {
   return this.age > autreAge;
}
```

Principe d'encapsulation des champs

Les champs (données d'un objets) ne sont pas directement accessible depuis « le monde extérieur » à l'objet. On doit utiliser des « accesseurs » pour les récupérer ou des « modificateurs » pour les modifier. Les deux sont des méthodes.

Accesseurs :
public TypeChamp getChamp() {
  return this.champ ;
 }

```
public String getNom() {
   return this.nom ;
 }
```

Modificateurs :
public void setChamp(TypeChamp nouvelleVal) {
  this.champ = nouvelleVal ;
}

```
public void setNom(String  nom) {
   this.nom = nom;
}
```

Constructeur

Les classes doivent avoir une méthode appelée « constructeur » qui permet la création d'objets. Ceux qui n'en n'ont pas sont des classes à part comme des bibliothèques de fonctions, des interfaces ou autres. Le rôle standard d'un constructeur est de renseigner tous les champs qui composent l'objet en création (c.à.d qu'il affecte des données aux différent champs de l'objet). Sa syntaxe est particulière pour bien le différencier des autres méthodes. Il est possible d'en avoir plusieurs si besoin. Le mot clé « this » représente l'objet en cours « de création ».

```
public Personne (String nom, String prenom, int age, String email) {
   this.nom = nom ;
   this.prenom = prenom ;
   this.age = age ;
   this.email = email ;
}

public Personne (String nom, String prenom, String email) {
   this.nom = nom ;
   this.prenom = prenom ;
   this.age = 18; // l'âge est fixé par défaut à 18 ans
   this.email = email ;
}
```

Objets : déclaration, affectation, création et utilisation (appelle) des méthodes

On peut désormais simplement créer des objets en appelant le constructeur de la classe :

Type nomObjet = new NomConstructeur(…) ;

```
Personne p1 = new Personne("Dupond", "Jean", 30);
```

On utilise simplement une méthode grâce au nom de l'objet un "." et le nom de la méthode :

nomObjet.nomMethode(Type entree1, …) ;

```
p1.setNom ("Dupont") ;
```

Si la méthode nous retourne une information, on peut par exemple la stockée :

TypeRetour nomVar = nomObjet.nomMethode(…) ;

```
String pernomP1 = p1.getPrenom() ;
```

Si le type de retour n'est pas un type de base (int, char, …) on peut directement enchaîner un appel de méthode sur l'objet retourné (« chaining » en angl.)

```
char premLettrePrenom = p1.getPrenom().charAt(0) ;
```

Agrégation

Une classe peut être composée de champs qui ont comme type une autre Classe. Le lien entre ces classes est nommé agrégation. On peut aussi le formuler avec les termes « contient » où « possède ».

```
public class Forme {
   // agrégation
   private Color couleurFond;
   // …
}
```

Méthodes particulières (convention de nommage et appelle implicite)

Comme la méthode « constructeur » d'autres méthodes apparaissent souvent dans les Classes (des comportement «classiques », nous y reviendront dans la partie « Interface » de cet aide mémoire). Pour s'y retrouver facilement, les signatures (nom, type de retour et paramètres de ces méthodes) ont été fixées. Outre les « accesseurs » et les « modificateurs » on peut citer les deux méthodes suivantes :

toString

Cette méthode effectue la génération d'une chaine de caractère représentant l'objet de manière « textuelle ». Elle est très utile lors des tests (on affiche alors simplement le retour de la méthode) ou pour des fichiers de log (des fichiers « texte » regroupant des informations sur l'exécution d'un programme) . la méthode toString est implicitement appelée par java lors d'une transformation d'un objet en String.

```
public String toString() {
   // le « caractère » retour à la ligne est représenté par \n
   // le « caractère » de tabulation est représenté par \t
   return "Nom: " + this.nom + "\n"
            + "Prénom : " + this.prenom + "\n"
            + "\tAge : " + this.age ;
}
```

equals

Cette méthode permet de comparer un objet à un autre. Elle prend donc en paramètre un objet à comparer avec l'objet actuel (this). La réponse est toujours soit « true » si les objets sont considérés comme égaux, « false » sinon.

```
// deux personnes sont « égales » si elles ont le même email
public boolean equals(Object autre) {
   // Si l'objet à comparer est bien une personne
   if (autre instanceof Personne) {
      return this.email.equals(autre.getEmail()) ;
   }
   return false ;
}
```

Héritages de Classe ou de Interface (comportement)

Si l'on remarque des similitudes entre deux (ou plus) classes on peut simplifier et structurer ces classes afin d'éviter entre autre la redondance de code (et ses défauts : mise à jour du code complexe, programmation plus fastidieuse, …).

Si les deux classes on des similitudes de champs ou de méthodes et que l'on peut leurs trouver une Classe commune plus générique, on utilise un héritage standards entre les classes Filles (celles qui héritent) et la classe Mère. L'héritage de multiple classe Mère n'est pas possible en Java. On peut aussi mettre en œuvre ce mécanisme si on n'a qu'une seul classe Fille pour des questions structurelles. La classe Fille peut réécrire certaine méthode, ou en ajouter, ainsi qu'ajouter des champs si nécessaire. Une classe Fille ne peut par contre pas « supprimer » une méthode ou un champ hérité.

```
public class Cercle extends Forme {
   // …
}
public class Rectangle extends Forme {
   // …
}
public class Employe extends Personne {
   // …
}
```

Si l'on remarque des similitudes de méthodes entre classes, mais qu'aucune classe générique ne les relie (par exemple Avion et Oiseau), on utilise l'héritage de comportement (interface). L'héritage de multiples comportements est possible en Java. La classe qui implémente une Interface DOIT « réécrire » les méthodes héritées. Certaines Interfaces supportent les « Generic » (Comparable par exemple).

```
public class Oiseau implements Flyable {
   //…
}

public class Avion implements Flyable, Comparable<Avion> {
   // …
}
```

Collections simples d'objets (ArrayList)

Les problèmes composés de multiples objets du même type (Classe) sont courant. La structure de données « tableau » semble donc idéal pour les stocker. Comme on remarque une similitude des méthodes couramment utilisées sur les tableaux (ajout d'élément, modification, …) on peut utiliser le concept d'héritage. On utilise alors la classe ArrayList (déjà codée en Java) qui

```
public class ListePersonnes extends ArrayList<Personne> {
   // …
}
```

possède déjà une structure interne de tableau et les méthodes standards adéquate (add, get, set, remove, …). On utilise fréquemment l'héritage d'ArrayList conjointement à l'utilisation d'un Generic qui permets de spécifier le type (Classe) des objets contenus dans la collection.

```
public class ListeFormes extends ArrayList<Formes> {
   // …
}
```

Algorithme de recherche dans une collection d'objets

L'algorithme de recherche dans une collection d'objets basé sur un tableau (ArrayList) est similaire à l'algorithme de recherche dans un tableau. L'algorithme retourne l'objet recherché ou la valeur « null » pour indiquer que l'objet n'a pas été trouvé. Lorsqu'on adapte l'algorithme en fonction du problème de recherche, c'est souvent uniquement la condition de recherche qui change (en gras dans les exemples).

```
//version « while » de recherche d'un livre grâce à son titre
public Book find(String title) {
      int i = 0;
      while (i < this.size() && !(this.get(i).equalsToTitle(title))) {
         i++;
      }
      if (i >= this.size()) {
         return null;
      } else {
         return this.get(i);
      }
}

// Version « for » de recherche d'un livre grâce à son titre et auteur
public Book find(String title, String author) {
      for (int i = 0; i < this.size(); i++) {
         if (this.get(i).equalsToTitleAndAuthor(title, author)) {
            return this.get(i);
         }
      }
      return null;
}
```

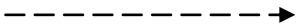
Représentation des relations dans un diagramme de Classes

L'héritage standard (de Classe) se représente par une flèche partant de la classe Fille vers la classe Mère.

extends

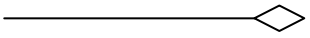
L'héritage de comportement (d'Interface) se représente par une flèche pointillée partant de la classe qui possède se comportement vers la classe Interface.

implements



L'agrégation se représente par une « flèche » dont la pointe est un losange partant de la classe qui « contient » vers celle « contenue ». On peut y ajouter une cardinalité (juste en dessus du losange) pour indiquer que la classe « contient » plusieurs liens vers la classe « contenue ».

un champ private interne à la classe qui contient et de type de la classe contenue



Si la cardinalité est 0..N (ou *) le champ private est alors une collection (ArrayList par ex.) avec un Generic de type de la classe contenue, ou encore mieux une nouvelle classe héritant d'ArrayList (avec un Generic de type de la classe contenue)