Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Base de la Programmation Orientée Objet

IUT de Clermont-Ferrand
Université d'Auvergne

1 février 2016

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Quiz

- En POO, qu'est ce qui caractérise un objet ?

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Quiz

- Quel lien existe-t-il entre un objet et sa classe ?

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Quiz

- Quelles sont les trois points qu'un développeur doit avoir à l'esprit le plus tôt possible lorsqu'il conçoit et développe une bibliothèque pour d'autres développeurs ?

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Quiz

- Rappelez les différentes phases principales du cycle de vie d'un objet.

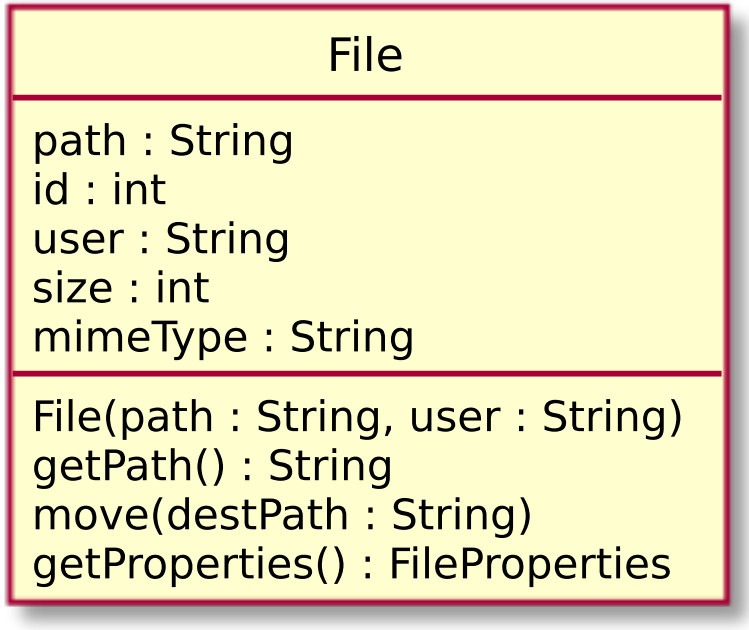
Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Quiz

- Grâce à quoi peut-on représenter graphiquement une modélisation (dans un cadre *Objet*).

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

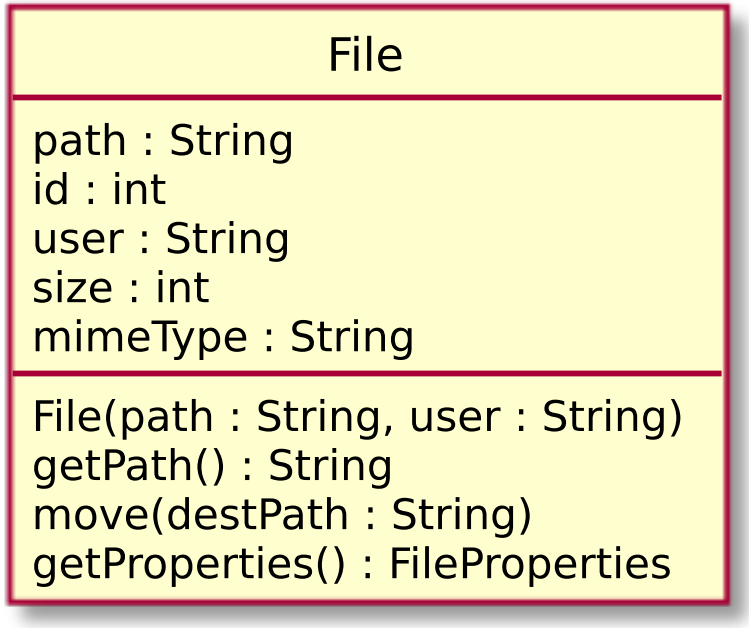
Quiz



- Qu'est-ce ?

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Quiz



- Comment se nomment les différents éléments de ce diagramme ?

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Quiz

- À quoi sert le mot clé this en Java ?

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Un peu de conception basique

1. Mon problème : j'ai besoin de modéliser un point en 2D
   - Mon « cahier des charges »
     - Un point sera représenté par ses coordonnées cartésiennes x et y
     - Je veux pouvoir calculer la distance d'un point à l'origine

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Un peu de conception basique

1. Mon problème : j'ai besoin de modéliser un point en 2D
   - Mon « cahier des charges »
     - Un point sera représenté par ses coordonnées cartésiennes x et y
     - Je veux pouvoir calculer la distance d'un point à l'origine

2. Très impressionné par ma classe Point2D mon collègue décide de l'utiliser pour modéliser un nouveau concept : le segment
   - Son cahier des charges :
     - Un segment est représenté par ses deux extrémités qui sont... des points bien évidemment
     - Il veut pouvoir calculer la longueur d'un segment

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Un peu de conception basique

3. Je découvre les coordonnées polaires. Émerveillé, je décide de représenter mon point 2D avec ces dernières
   - Pratique ! Ça me simplifie beaucoup ma fonction distanceOrigine()

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Un peu de conception basique

3. Je découvre les coordonnées polaires. Émerveillé, je décide de représenter mon point 2D avec ces dernières
- Pratique ! Ça me simplifie beaucoup ma fonction distanceOrigine()

Problème ???

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Un peu de conception basique

3. Je découvre les coordonnées polaires. Émerveillé, je décide de représenter mon point 2D avec ces dernières
- Pratique ! Ça me simplifie beaucoup ma fonction distanceOrigine()

Problème ???

- J'ai rendu un collègue malheureux... sa classe Segment2D ne fonctionne plus :(
- Pas bien !

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Autre situation

- Je veux créer une classe qui modélise le concept de rectangle
- Un rectangle est représenté par un point, qui représente son coin supérieur gauche, ainsi que sa longueur et sa largeur

Rectangle
coinHautGauche : Point2D largeur : double longeur : double
translater(dx : double, dy : double) aire() : double

- Rien ne m'empêche de faire : `rect.largeur = -3.45;`
- C'est gênant une largeur négative !

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Contrôle d'accès

- Lors de la définition d'une classe il est possible de *restreindre* la visibilité des attributs ou des méthodes des instances de cette classe

3 niveaux de visibilité

public	accessible par tout le monde (tous ceux qui ont une référence sur l'objet)
protected	accessible depuis les instances de la classe et ses classes dérivées
private	accessible uniquement depuis des instances de la classe

- +1 par défaut (si on ne met rien) : accès package

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Intérêt

- Masquer l'implémentation
  ↪ le programmeur « client » n'a pas besoin de connaître la composition interne de l'objet
- Évolutivité
  ↪ il est possible de modifier les données `private` sans impact direct pour l'utilisateur de la classe
- Protection
  ↪ ne pas permettre l'accès à tout dès que l'on possède une référence sur l'objet

C'est l'encapsulation

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

- Mais comment je fais pour accéder aux coordonnées de mon point alors ?

Règle générale

Les attributs de l'objet sont déclarés `private`, mais des méthodes `public` permettant de modifier/d'accéder à ces attributs sont fournies si nécessaire.

Il peut y avoir des exceptions à cette règle quand c'est justifié

- Méthodes appelées des accesseurs (getter en anglais) et mutateurs (setter en anglais)

Usage

Si attribut `auteur` $\Longrightarrow$ `Auteur getAuteur()` : accesseur
(`boolean isAuteur()` si predicat)
`setAuteur(Auteur a)` : mutateur

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Illustration

```
public class Point2D {
  // données membres
  private double x;
  private double y;

  // constructeur
  public Point2D(double x, double y) {
    this.x = x;
    this.y = y;
  }
  // accesseurs
  public double getX() { return x; }
  public void setX(double x) { this.x = x; }
  public double getY() { return y; }
  public void setY(double y) { this.y = y; }


   // méthode
   public double distanceOrigine() {
     return Math.sqrt(getX() * getX() + getY() * getY());
     // ou bien : return Math.sqrt(x * x + y * y);
   }
 }
```

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Illustration

```
public class Point2D {
  // données membres
  private double x;
  private double y;

  // constructeur
  public Point2D(double x, double y) {
    this.x = x;
    this.y = y;
  }
  // accesseurs
  public double getX() { return x; }
  public void setX(double x) { this.x = x; }
  public double getY() { return y; }
  public void setY(double y) { this.y = y; }


   // méthode
   public double distanceOrigine() {
     return Math.sqrt(getX() * getX() + getY() * getY());
     // ou bien : return Math.sqrt(x * x + y * y);
   }

 }
```

Attention, ces notions d'accessibilité ont un sens pour les méthodes et attributs, mais pas pour les variables « classiques »

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Squelette d'une « bonne » classe

```
public class MonObjet {
  private Type attribut;                          // attributs
  ...
  public MonObjet() {...}
  public MonObjet(...) {...}
  public MonObjet(MonObjet o) {...}
  ...
  public Type getAttribut() {...}
  public void setAttribut(Type attr) {...}
  ...
  public Type interfaceMethod(...) {...}
  ...
  private Type maMethod(...) {...}
  ...
  public String toString() {...}
  public boolean equals(Object obj) {...}
}
```

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Squelette d'une « bonne » classe

```
public class MonObjet {
  private Type attribut;
  ...
  public MonObjet() {...}
  public MonObjet(...) {...}                    // constructeurs
  public MonObjet(MonObjet o) {...}
  ...
  public Type getAttribut() {...}
  public void setAttribut(Type attr) {...}
  ...
  public Type interfaceMethod(...) {...}
  ...
  private Type maMethod(...) {...}
  ...
  public String toString() {...}
  public boolean equals(Object obj) {...}
}
```

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Squelette d'une « bonne » classe

```
public class MonObjet {
  private Type attribut;
  ...
  public MonObjet() {...}
  public MonObjet(...) {...}
  public MonObjet(MonObjet o) {...}
  ...
  public Type getAttribut() {...}                    // accesseurs
  public void setAttribut(Type attr) {...}
  ...
  public Type interfaceMethod(...) {...}
  ...
  private Type maMethod(...) {...}
  ...
  public String toString() {...}
  public boolean equals(Object obj) {...}
}
```

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Squelette d'une « bonne » classe

```
public class MonObjet {
  private Type attribut;
  ...
  public MonObjet() {...}
  public MonObjet(...) {...}
  public MonObjet(MonObjet o) {...}
  ...
  public Type getAttribut() {...}
  public void setAttribut(Type attr) {...}
  ...
  public Type interfaceMethod(...) {...}     // API publique
  ...
  private Type maMethod(...) {...}
  ...
  public String toString() {...}
  public boolean equals(Object obj) {...}
}
```

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Squelette d'une « bonne » classe

```
public class MonObjet {
  private Type attribut;
  ...
  public MonObjet() {...}
  public MonObjet(...) {...}
  public MonObjet(MonObjet o) {...}
  ...
  public Type getAttribut() {...}
  public void setAttribut(Type attr) {...}
  ...
  public Type interfaceMethod(...) {...}
  ...
  private Type maMethod(...) {...}    // bidouilles internes
  ...
  public String toString() {...}
  public boolean equals(Object obj) {...}
}
```

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Squelette d'une « bonne » classe

```
public class MonObjet {
  private Type attribut;
  ...
  public MonObjet() {...}
  public MonObjet(...) {...}
  public MonObjet(MonObjet o) {...}
  ...
  public Type getAttribut() {...}
  public void setAttribut(Type attr) {...}
  ...
  public Type interfaceMethod(...) {...}
  ...
  private Type maMethod(...) {...}
  ...
  public String toString() {...}          // méthodes utiles
  public boolean equals(Object obj) {...}
}
```

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Réutilisation

Problème

Comment utiliser une classe comme brique de base pour concevoir d'autres classes ?

- Dans une conception objet on définit des associations (relations) pour exprimer la réutilisation entre classes

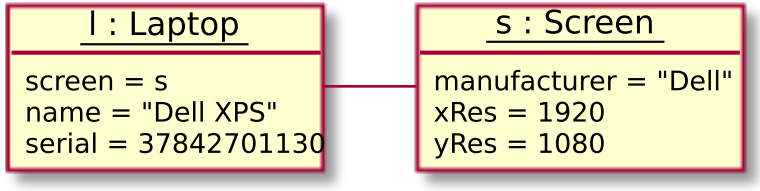
Solutions possibles

- Un objet peut embarquer et faire appel à un autre objet : c'est la composition
- Un objet peut être crée à partir du « moule » d'un autre objet : c'est l'héritage

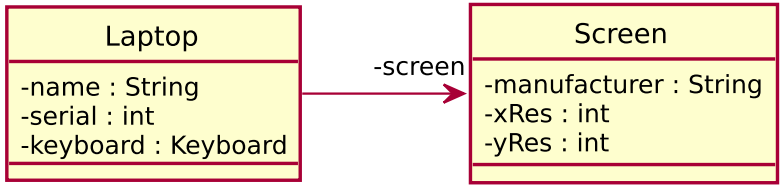
Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

La composition

- Un objet `o1` instance de la classe `C1` embarque un objet `o2` instance de la classe `C2`



- La classe `C1` comporte des attributs de type `C2`
- `C1` est la classe composée
- `C2` est la classe composante



- La classe composée (`C1`) possède une référence du type de la classe composante (`C2`)

```
public classe C1 {
  private C2 attr;
  ...
}
```

```
public classe C2 {
  ...
}
```

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

La composition

Exemple de la classe `Cercle`

- Un rayon : `double`
- Un centre : deux `double` `x` et `y` ou mieux, un `Point`

- L'association entre les classes `Cercle` et `Point` exprime le fait que qu'un cercle *possède* un centre
- On parle de relation a-un (has-a)

```
public class Cercle {
  // Centre du cercle
  private Point centre;
  // Rayon du cercle
  private double rayon;
  ...
  public void translation(double dx,
                          double dy) {
    centre.translation(dx, dy);
  }
  ...
}
```

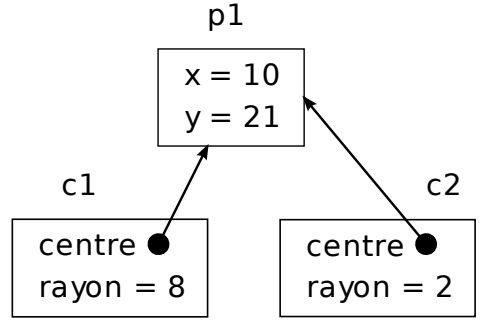
Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

La composition : 2 cas

```
public class Cercle {
  // Centre du cercle
  private Point centre;
  // Rayon du cercle
  private double rayon;
  ...
  public Cercle (Point c, double r){
    this.centre = c;
    this.rayon = r;
  }
  ...
}
```

```
Point p1 = new Point(10, 21);
Cercle c1 = new Cercle(p1, 8);   ⟶
Cercle c2 = new Cercle(p1, 2);
```

- Le point représentant le centre du cercle a une existence autonome (cycle de vie indépendant)
- Il peut être partagé (à un même moment il peut être lié à plusieurs instances d'autres classes)
- Il peut être utilisé en dehors du cercle (attention aux effets de bord)



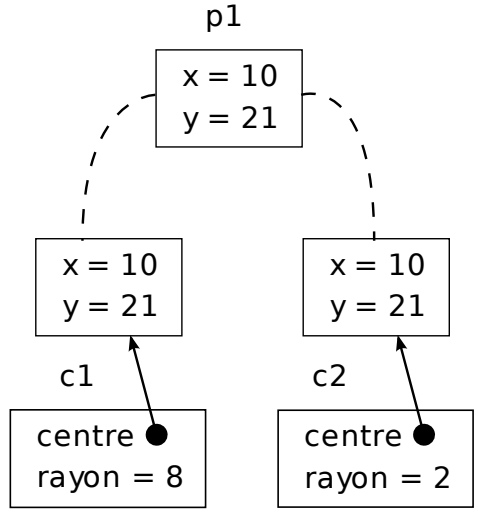
Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

La composition : 2 cas

```
public class Cercle {
  // Centre du cercle
  private Point centre;
  // Rayon du cercle
  private double rayon;
  ...
  public Cercle (Point c, double r){
    this.centre = new Point(c);
    this.rayon = r;
  }
  ...
}
```

```
Point p1 = new Point(10, 21);
Cercle c1 = new Cercle(p1, 8);   ⟶
Cercle c2 = new Cercle(p1, 2);
```

- Le point représentant le centre du cercle n'est pas partagé (chaque cercle possède sa propre instance de `Point`)
- Les cycles de vie du point et du cercle sont liés : si le cercle est détruit (ou copié), le point l'est aussi

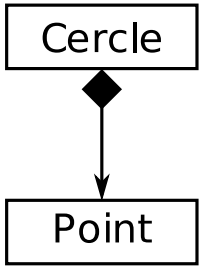


Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Agrégation / Composition

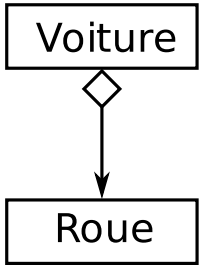
- Les deux exemples précédents traduisent deux nuances (sémantiques) de l'association a-un entre la classe `Cercle` et la classe `Point`
- UML distingue ces deux sémantiques en définissant deux type de relations :

Composition



A un même moment, une instance du composant (`Point`) ne peut être liée qu'à un seul agrégat (`Cercle`), et le composant a un cycle de vie dépendant de l'agrégat

Agrégation



L'élément agrégé (`Roue`) a une existence autonome en dehors de l'agrégat (`Voiture`)

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Les tableaux

Comme en C... ou presque

Tableau

Agrégat de composants, primitifs ou objets, de même type, dont l'accès se fait par un indice calculé

- Déclaration d'un tableau d'éléments de type `Type` : `Type[] tab;`
- La déclaration d'un tableau n'alloue pas d'espace pour le tableau !
- C'est simplement une référence à un objet de type tableau

Allocation

```
float[] premierEssai = new float[42];
Random[] genAleatTab = new Random[3];
```

- L'allocation n'alloue que des références pour les composants de type objets. Aucun instance de la classe n'est créée !

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Les tableaux

- Par défaut les composants d'un tableau sont initialisés à 0, 0.0, `false`, '\u0000' ou `null` selon le type

Définition

```
int[] t1 = {4, 11, -2};
Date[] t2 = {null, new Date()};
Date[] t3;
t3 = new Date [] {null, new Date(1843)};
```

- Le nombre d'éléments d'un tableau et ainsi fixé une fois pour toute à sa création et stocké dans un champ `length`
- Un tableau `t` est donc indicé de 0 à `t.length - 1`

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Tableaux multidimensionnels

Déclaration

Déclaration d'un tableau `tab` à n dimensions

`Type` $\underbrace{\texttt{[][]...[]}}_{n}$ `tab;`

Allocation

Pour chacune des dimensions on indique le nombre de composants

```
tab = new Type[N1][N2]...[Nn];
```

Exemple

```
int[][] tabNbr = new int[2][3];
double[][] matrice = new double[][]{{3.4, .3},
                                    {5, 1.3},
                                    {2.3, 0}};
```

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Parcours

Exemple : boucle for classique

```
double k = 1;
for (int i = 0; i < matrice.length; ++i)
  for (int j = 0; j < matrice[i].length; ++j)
    matrice[i][j] += k++;
```

Exemple : boucle de type *for each*

```
String s = "";
for (double[] ligne : matrice) {
  s += "| ";
  for (double x : ligne)
    s += x + " ";
  s += "|\n";
}
```

Résultat avec la matrice de l'exemple précédent :

```
| 3.4 0.3 |
| 5.0 1.3 |
| 2.3 0.0 |
```

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Délégation

- Composition/agrégation : vision structurelle des choses
- En terme de comportement on parle de délégation

Délégation

En POO, un objet peut confier la résolution d'une de ses opérations à un autre objet. C'est la délégation. L'objet endossant cette responsabilité est nommé le délégué (delegate en anglais).

- Composition/agrégation : une concrétisation de la délégation
- Version plus légère : l'utilisation

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Illustration

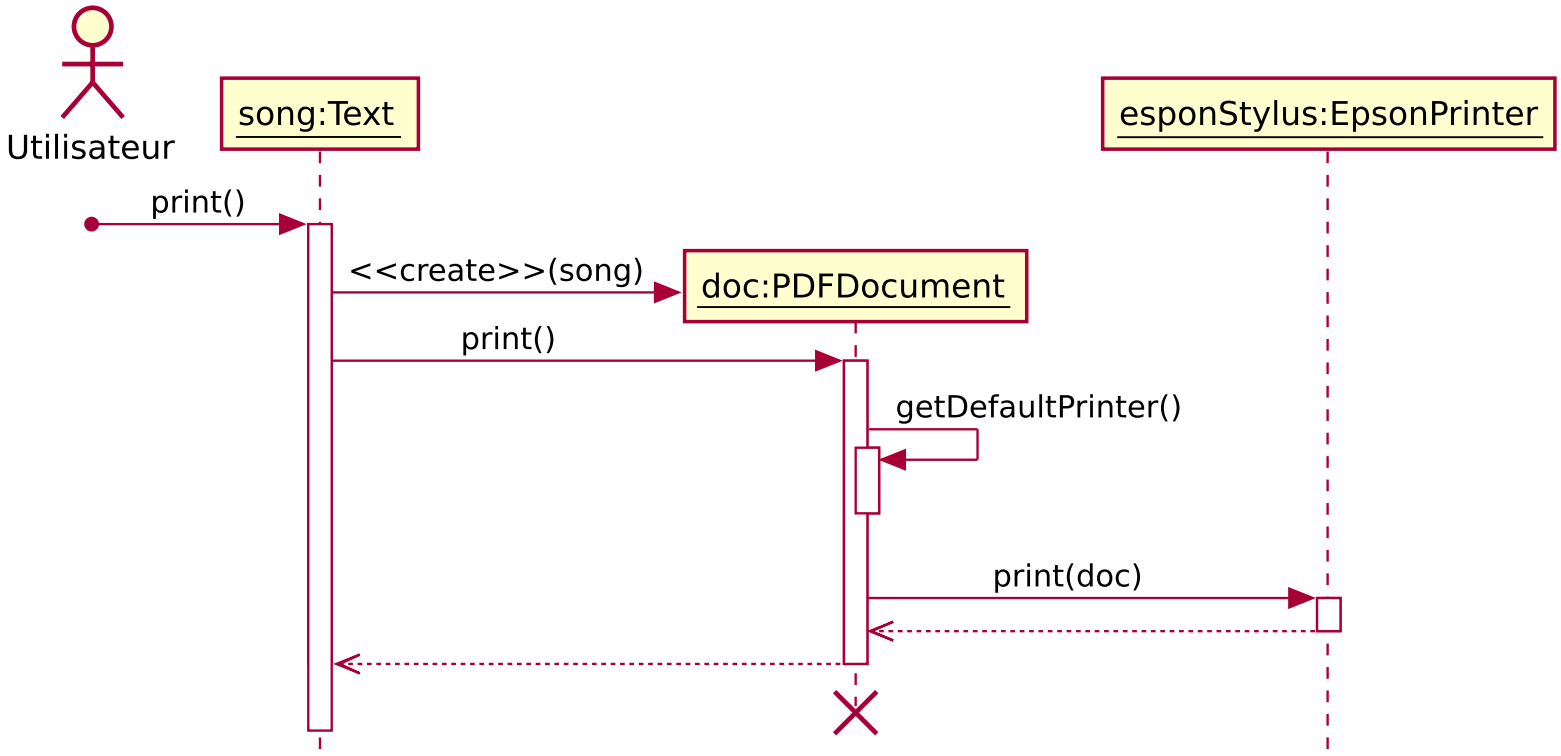
Activité

Je dispose d'un objet `song`, instance d'une classe `Text` représentant le texte mis en page d'une chanson. J'aimerais imprimer ce texte sur mon imprimante `epsonStylus` (instance de la classe `EpsonPrinter`). Malheureusement cette imprimante ne permet que d'imprimer des objets de type `PDFDocument`, à travers sa méthode publique `print(doc:PDFDocument)`. Mais par chance je dispose d'une classe `PDFDocument` avec un constructeur permettant de créer un document PDF à partir d'un `Text`. Cette classe dispose d'une méthode privée `getDefaultPrinter()` permettant d'obtenir l'instance de l'imprimante par défaut du système (ici `epsonStylus`) ainsi qu'une méthode publique `print()` permettant d'imprimer le document PDF courant sur l'imprimante par défaut.

Illustrez le principe de la délégation dans ce problème.

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Illustration



Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Traitement d'erreurs

```
#include ...
#include <string.h> /* strerror() */
#include <errno.h>  /* errno */

static void teste(const char *nombre)
{
    unsigned long res;

    /* On reinitialise errno */
    errno = 0;

    /* Appel de strtoul : conversion d'une chaine en un entier */
    res = strtoul(nombre, NULL, 10);

    /* Erreur strtoul si retour = ULONG_MAX et errno non nul. */
    if (res == ULONG_MAX && errno != 0)
    {
        /* Il y a eu une erreur ! */
        fprintf(stderr, "Erreur conversion (%s)", strerror(errno));
    }
    else
    {
        printf("Conversion OK. Valeur = %lu", res);
    }
}
```

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Exceptions

Question : Comment gérer les erreurs dans un programme Java ?

Statiquement — À la compilation

- On essaye de détecter un maximum d'erreurs à la compilation du code
  ↪ merci au typage
  ↪ malheureusement, pas toujours possible

Dynamiquement — À l'exécution

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Exceptions

Question : Comment gérer les erreurs dans un programme Java ?

Statiquement — À la compilation

Dynamiquement — À l'exécution

1. On met au point des codes d'erreurs
   - Les fonctions renvoient les codes d'erreurs pour signifier leur succès ou leur échec
     ↪ ⊖ lourd à gérer
     ↪ ⊖ difficile à traiter
     ↪ ⊖ mélange entre code « utile » et code de « contrôle »
     ↪ ⊖ pas naturel
2. On met en place une gestion « d'exceptions »

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Exceptions

Question : Comment gérer les erreurs dans un programme Java ?

Statiquement — À la compilation

Dynamiquement — À l'exécution

1. On met au point des codes d'erreurs
2. On met en place une gestion « d'exceptions »
   - Lors d'une erreur un objet est généré pour décrire l'erreur et le programmeur peut utiliser cet objet pour traiter l'erreur
     ↪ ⊕ ciblage de l'erreur dans le code
     ↪ ⊕ séparation « cas normaux » / « cas exceptionnels »
     ↪ ⊕ une entité spéciale traite l'exception : le *gestionnaire d'exception* (exception handler)

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Exceptions

- Qu'est-ce qu'un « cas exceptionnel »
  ↪ une situation qui ne correspond pas au « fonctionnement normal » du programme tel qu'on l'a envisagé

Diviser un nombre par 0 :	`ArithmeticException`
Envoyer un message en passant par une référence `null`	`NullPointerException`
Accéder à des cases d'un tableau en dehors des indices valables	`ArrayIndexOutOfBoundsException`
Downcaster un objet vers une classe dont il n'est pas une instance	`ClassCastException`
Tenter lire un fichier qui n'existe pas	`FileNotFoundException`

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Exceptions

Exemple

```
public class ExampleCasExeptionnel {
  public static void main (String[] args) {
      int monTab[] = {1, 2, 3, 4};
      for (int i = 0; i <= monTab.length; ++i)
        System.out.println(monTab[i]);
    }
  }
```

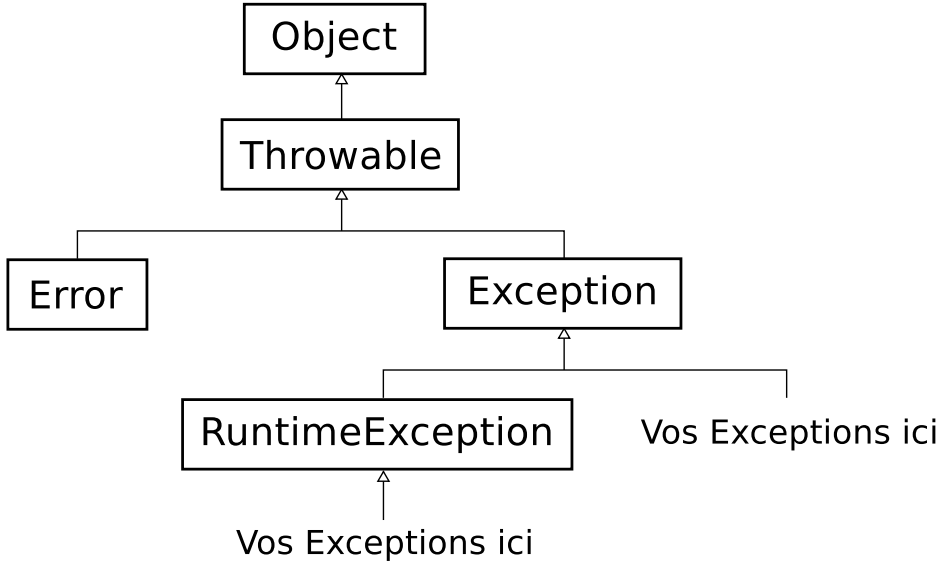
Résultat

```
1
2
3
4
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 4 at
ExampleCasExeptionnel.main(ExampleCasExeptionnel.java:5)
```

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Exceptions

- En Java les exceptions sont des objets



- Elles sont toutes du type `Exception` et leur type « précis » est un « sous-type » de `Exception`
- Par convention ces classes se nomment `QuelqueChoseException`

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Exceptions

- Comment créer une nouvelle classe d'exceptions ?

Trivial : on utilise la solution par défaut

```
class SimpleException extends Exception {}
```

- Des portions de code peuvent *générer* ou *lever* (raise) des exceptions, signe d'un problème
- Le programmeur dispose d'un moyen de *gérer* ou *capturer* (catch) des exceptions, et ainsi proposer une solution ou une alternative à l'erreur rencontrée

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Exceptions

Lever une exception

Instruction **throw**

```
String attention = "Je vais lancer un exception simple !";
throw new SimpleException();
String pasLa = "On arrive jamais ici";
```

- Les exceptions sont des objets !
- Nouvelle instance crée à la levée de l'exception
- Utilisation des constructeurs de `SimpleException`

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Exceptions

Capturer une exception

Instructions **try**... **catch**

```
try {
  // Code susceptible de lancer une exception
}
catch (TypeExceptionACapturer e) {
  // Traitement de l'exception (infos à travers e)
}
```

- Lorsqu'une exception est levée cela n'arrête pas le progamme
- On quitte le bloc `try` dès qu'une exception est levée dans ce bloc
- Si l'exception est capturée, le traitement associé à cette capture est exécuté

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Exceptions

Capturer une exception

- Un même bloc peut être susceptible de lever plusieurs exceptions
- Il est possible des les traiter séparément ou globalement

Exemple

```
try {
  double x = 1 / obj.getValue();
}
catch (NullPointerException e) {
  System.out.println("obj ne référence aucun objet valide!");
}
catch (ArithmeticException e) {
  System.out.println("La valeur de obj est zéro");
}
```

```
try {
  double x = 1 / obj.getValue();
}
catch (Exception e) {
  System.out.println("Ooops... voici l'erreur : " + e);
}
```

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Exceptions

Capturer une exception

Clause **finally**

```
try {
  // Code susceptible de lancer une exception
}
catch (TypeException1 e1) {
  // Traitement de l'exception de type 1
}
...
catch (TypeExceptionN eN) {
  // Traitement de l'exception de type N
}
finally {
  // Bloc toujours exécuté
}
```

- Seul le premier bloc `catch` compatible avec l'exception levée est exécuté
- Les instructions du bloc `finally` sont toujours exécutées

Rafraichissement
Encapsulation
Réutilisation
Composition
Délégation
Exceptions

Exceptions

Tranférer une exception (exception specification)

- Si on ne désire pas traiter l'exception dans une fonction qu'on écrit, on le spécifie (on est poli) aux clients

Instruction **throws**

```
public void uneMethode(...) throws SimpleException {
    ...
}
```

- On peut transférer plusieurs exceptions
- Il faut spécifier toutes les exceptions susceptibles d'être levées par la fonction
- C'est vérifié par le compilateur Java (sauf pour les `RuntimeException`)