Programmation Orientée Objet

IUT Béthune
DUT2 Réseaux & Télécommunications

Daniel Porumbel

Premier exemple d'objet en Ruby

```
1 monTelPerso = Telephone.new ("06 76 93 33 33")
2 monContact = Contact.new ("06 45 32 34 12")
3 monTelPerso.appeler(monContact)
```

- Le mot `monTelPerso` indique un objet
- Le mot `Telephone` indique une classe
- Le mot `new` indique un constructeur :
  - construit un objet (une instance de classe `Telephone`) avec un attribut représentatif : le numéro de téléphone
- Le mot `appeler` indique une action que l'objet exécute

A retenir : Un objet représente une instance d'une classe : `monTelPerso` est un objet de classe `Telephone`

Premier exemple d'objet en Ruby

```
1 monTelPerso = Telephone.new ("06 76 93 33 33")
2 monContact = Contact.new ("06 45 32 34 12")
3 monTelPerso.appeler(monContact)
```

Ce code est facile mais peut-il vraiment tourner sur un smartphone ?
Oui, mais il faut une des deux conditions :

1. On définit nous même la classe `Telephone`
2. On réutilise une classe `Telephone` déjà écrite dans un paquetage d'une bibliothèque
   - En principe, il suffirait d'ajouter au début du programme : `import "Telephone"`

La réutilisation du code

Si on imposait une classe `Telephone` standard

- tout smarphone `Android` devrait définir une classe `Téléphone`
- on pourrait utiliser cette classe sans connaître le modèle de smartphone
- le code de la classe `Telephone` est réutilisé à chaque utilisation d'un objet
  - ce code reste caché pour l'utilisateur

`Telephone`

A retenir : La réutilisation du code est un important avantage de la programmation objet

Un exemple du projet Ruboto (Android Ruby)

```
1 import "android.content.Intent"
2 import "android.net.Uri"
3 intent = Intent.new(Intent::ACTION_VIEW)
4 intent.setData(Uri.parse("tel:4444"))
5 $activity.startActivity(intent)
```

- classes importées : `Uri` (paquetage `android.net`) et `Intent` (paquetage `android.content`)
  - les paquetages sont distribués dans une bibliothèque (“library” en anglais)
- `$activity` : variable globale qui fait référence à un objet “activité”,

  activité : concept de base `Android` qui représente une action précise (un “écran”) que l'utilisateur exécute à un moment donné (ici, un appel)

Classes Ruby et classes Java

- documentation en-ligne pour toutes les classes `Android` : `developer.android.com/reference/classes.html`
- on observe que notre programme `Ruby` utilise en fait des classes `Java`
- Challenge `Java` : trouver dans la documentation la meilleure classe pour chaque besoin et l'utiliser correctement

A retenir : savoir bien utiliser ces classes `Java` en `Ruby` = pouvoir assez facilement les utiliser en `Java`

Ruby, Java et Android

`Ruby` sert uniquement à écrire des résultats dans un terminal ?

`jruby`

- `Ruby` est capable d'utiliser toute classe `Java` sous Linux (machine classique)
  - Nous verrons la bibliothèque graphique `Swing`

`ruboto`

- `Ruby` est capable d'utiliser toute classe `Java` pour Android (smartphone)
- `ruboto` est une application Android disponible dans `Google Play` (nouveau nom de `Google Market`)
- l'application `ruboto` fournit des exemples de programmes pour : faire des mini-jeux, utiliser des connexions TCP/IP, passer des appels (déjà présenté)

1. Écrire ses Propres Classes Ruby
   - Conseils et Bonnes Pratiques de Programmation
   - Classe `Complexe`
   - Classes `Agenda` et `Telephone`
   - Héritage
2. Utiliser des Classes `Java` en `Ruby`
   - Paquetage `java.util` : classes `Java` standard
   - La bibliothèque graphique `Swing`
3. Notions de Java

Rappels Ruby : documentation

- D. Carrera, L. Sansonetti. *Apprenez Ruby (en français)*. `ruby-doc.org/docs/beginner-fr/xhtml/`
- Yukihiro Matsumoto et Eric Jacoboni. *Ruby In A Nutshell (en français)*. Editeur : O'Reilly. ISBN : 978-2841772100
- David Thomas. *Programming Ruby – The Pragmatic Programmer's Guide*.
  - `ruby-doc.org/docs/ProgrammingRuby/`
  - Ce livre est une référence complète du langage `Ruby`.
- `Swing` : `JRuby` et `Java`
  - `http://www.jruby.org/`
  - Les bases Java : Cyrille Herby. *Apprenez à programmer en Java*. Éditeur : SimpleIT. ISBN 978-2953527834

- Pour le contrôle TP : `http://iut-rt/~porumbel/i5/`

Bonnes pratiques de programmation

- Éditeur conseillé : `geany` – facile à installer et utiliser
- L'affichage d'une chaîne
  - `puts "Salut"`
  - `puts 'Salut'`

  Appel conseillé :
  - `puts("Salut");`
    - Utiliser des guillemets pour des chaînes et des apostrophes pour une seule lettre
- L'affichage d'une valeur
  - `puts("La valeur de a est: #{a}")` style `Bash`
  - `printf("La valeur de a est: %d",a)` style `C/C++`
  - `puts("La valeur de a est:"+a.to_s())`

    →→→ style `Java`, méthode conseillée

Bonnes pratiques de programmation 2

Ne pas utiliser : des accents, caractères spéciaux ou espaces dans les noms de variables et fichiers

- Exemple : `telephone` au lieu de `téléphone`
- Vous évitez des problèmes inutiles de codage de caractères

- la première lettre d'un nom de classe est une majuscule
- la première lettre d'un nom de méthode/variable est une minuscule

Objectif : manipuler les numéros complexes

On souhaite permettre à ce code de fonctionner.

```
1 c1 = Complexe.new(2,3);
2 puts c1.reel();          #affiche 2
3 puts c1.imag();          #affiche 3
4 c2 = Complexe.new(4,5);
5 puts c2                  #affiche 4 + i5
6 c3 = c2.produit(c1);
7 puts c3                  #affiche -7 +i22
8 c4 = c1*c2
9 puts c4                  #affiche -7 +i22
```

Solution → définir une classe `Complexe`

Étape 1 : Constructeur et attributs

Pour faire fonctionner

```
1 c1 = Complexe.new(2,3);
```

Il faut définir la classe et un constructeur qui construit (initialise) les les attributs

attributs données qui décrivent la structure interne des objets

```
1 class Complex
2     @r;
3     @i;
4     def initialize(arg1, arg2)
5         @r = arg1;
6         @i = arg2;
7     end
8 end
```


Étape 2 : Des méthodes

Pour faire fonctionner

```
1 puts c1.reel();          #affiche 2
2 puts c1.imag();          #affiche 3
```

Il faut définir deux méthodes (`reel()` and `imag()`) qui peuvent être appelées par un objet de classe `Complexe`.

```
1 def reel()
2     return @r
3 end
4 def imag()
5     return @i
6 end
```


Étape 3 : Méthode `to_s`

Pour faire fonctionner

```
1 puts c2
```

Il faut définir une méthode to_s, appelé automatiquement par `puts`
Cette méthode renvoie (via `return`) une chaîne de caractères.

```
1 def to_s()
2     return @r.to_s()+" + i"+@i.to_s()
3 end
```

Le même principe existe en `Java` : tout objet définit une méthode `toString()`

Étape 4 : Méthode `produit(autreCompl)`

Pour faire fonctionner

```
1 c3 = c2.produit(c1)
```

Il faut définir une méthode produit(autreCompl) qui renvoie le produit avec un autre complexe `autreCompl`

```
1 def produit(autreCompl)
2     r2 = autreCompl.reel()
3     i2 = autreCompl.imag()
4     res_r = @r*r2 - @i*i2;
5     res_i = @r*i2 + @i*r2;
6     res = Complexe.new(res_r, res_i);
7     return res;
8 end
```

Étape 5 : Opérateur de multiplication

Pour faire fonctionner

```
1 c3 = c2*c1;
```

Il faut définir un `opérateur` de multiplication qui appelle la méthode produit(autreCompl)

```
1     def *(autreCompl)
2         return produit(autreCompl)
3     end
```


Implémentation et Utilisation

Séparation entre Implémentation et Utilisation

- l'utilisateur doit pouvoir appeler les méthodes publiques
  - méthodes publiques dans notre classe : `reel()`, `imag()`, `to_s`, `produit(autreCompl)`
- l'utilisateur n'est pas censé connaître les attributs ou méthodes privés
  - nos attributs privés : `@r`, `@i`, pas de méthode privés

Les attributs privés sont considérés comme des détails d'implémentation par l'utilisateur d'une classe.

Modifier l'implémentation

Cette séparation est un principe de base de la programmation objet :

- Ruby : instruction impossible `puts c1.@r`
- `Java` : chaque classe doit être définie dans un autre fichier

---

Pas obligatoire d'utiliser la forme algébrique ($x + iy$)

- On peut utiliser la représentation polaire : module m et angle α
- Multiplication plus facile :

compl 1 m_1, α_1
compl 2 m_2, α_2
produit $m_1 \cdot m_2, \alpha_1 + \alpha_2$

Objectif : nouvelle implémentation

```
1 c1 = Complexe.new(2,3);
2 puts c1.reel();              #affiche 2
3 puts c1.imag();              #affiche 3
4 c2 = Complexe.new(4,5);
5 puts c2                      #affiche 4 + i5
6 c3 = c2.produit(c1);
7 puts c3                      #affiche -7 +i22
```

La nouvelle implémentation `Complexe` devrait :

1. déclarer :
   - deux variables `module` et `angle`
   - méthodes d'accès à ces variables (appelées"getteur" et "setteur")
2. changer le constructeur publique (arguments r et i)
3. changer les méthodes `reel()`, `imag()`, `to_s()`, `produit(autreCompl)`

Étape 1 : variables, getteurs et setteurs

```
1 class Complex
2     @module;
3     @alpha;
4     def getModule()                  //Un "getteur"
5         return @module;
6     end;
7     def getAlpha()
8         return @alpha
9     end
10    def setModule(newModule)     //Un "setteur"
11        @module = newModule;
12    end;
13    def setAlpha(newAlpha)
14        @alpha = newAlpha;
15    end
16 end
```


Étape 2 : Le constructeur

Pour faire fonctionner

```
1 c1 = Complexe.new(2,3);
```

Le constructeur doit calculer les paramètres polaires (module, angle) à partir des paramètres algébriques (réel, imaginaire).

```
1     def initialize(reel, imag)
2         @module = Math.sqrt(reel**2+imag**2);
3         @alpha = Math.atan2(imag,reel);
4     end
```


Étape 2 : Les autres méthodes

Pour faire fonctionner

```
1 puts c1.reel();          #affiche 2
2 puts c1.imag();          #affiche 3
3 puts c2                  #affiche 4 + i5
```

Il faut définir : `reel()`, `imag()`, and `to_s()`.

```
1     def reel()
2         return Math.cos(@alpha) * @module
3     end
4     def imag()
5         return Math.sin(@alpha) * @module
6     end
7     def to_s()
8         return reel().to_s()+" + i"+imag().to_s
              ()
9     end
```


Étape 3 : Méthode `produit(autreCompl)`

La multiplication est plus rapide en coordonnées polaires.

```
1     def produit(autreCompl)
2         module2 = autreCompl.getModule()
3         alpha2  = autreCompl.getAlpha()
4         produit = Complexe.new(0,0);
5         produit.setModule(@module * module2);
6         produit.setAlpha(@alpha + alpha2);
7         return produit;
8     end
```

Objectif : agenda téléphonique

Comment faire fonctionner ce code ?

```
1 monAgenda = Agenda.new();
2 contact1  = Contact.new("Daniel", "01 23");
3 monAgenda.ajouter(contact1);
4 contact2  = Contact.new("Janie", "05 07");
5 monAgenda.ajouter(contact2);
6 contact3  = Contact.new("Daniel", "06 ...");
7 monAgenda.ajouter(contact3);#impossible !!!
8 monTel= Telephone.new (06 76 93 33 33);
9 monTel.appeler(monAgenda.trouver("Daniel"));
    #appelle Daniel (01 23)
```

Il faut définir 3 classes : `Contact`, `Agenda` et `Telephone`

- seule méthode spécifique à la machine :`appeler(num)`, classe `Telephone`

Classe Telephone

Voici un exemple pour Android. L'implémentation est différente sur d'autres systèmes d'exploitation.

```
1 import "android.content.Intent"   #Uniquement
2 import "android.net.Uri"          #Android
3 class Telephone
4   @numeroPerso
5   def initialize(numero)
6       @numeroPerso = numero;
7   end
8   def appeler(numero)
9       puts "Appeler"+autreNumero;
10      intent = Intent.new(Intent::ACTION_VIEW)
11      intent.setData(Uri.parse("tel:"+numero))
12      $activity.startActivity(intent)
13  end
14 end
```


Classe `Contact`

Une classe assez basique :

- deux attributs : un nom et un numéro
- méthodes d'accès aux attributs : 2 seteurs et 2 getteurs

```
1 class Contact
2     @nom;
3     @numero;
4     def initialize(unNom, unNumero)
5         @nom = unNom;
6         @numero = unNumero;
7     end
```


Classe `Contact` : méthodes d'accès

Une bonne pratique de programmation : commencer la définition par les seteurs et geteurs.

```
1     def getNom()
2         return @nom;
3     end
4     def getNumero()
5         return @numero
6     end
7     def setNom(unNom)
8         @nom = unNom;
9     end
10    def setNumero(unNumero)
11        @numero = unNumero;
12    end
```

Classe Agenda

Rappel Objectif : pouvoir exécuter un code

```
monAgenda = Agenda.new();
monAgenda.ajouter(contact1);
monAgenda.ajouter(contact2);
...
monAgenda.ajouter(contact999);
monTel.appeler(monAgenda.trouver("Daniel"))
```

Solution : utiliser deux tableaux `lesNoms` `lesNumeros` : `lesNoms[i]` correspond à `lesNumeros[i]`

```
class Agenda
    @lesNoms;
    @lesNumeros;
    def initialize()
        @lesNoms = Array.new()
        @lesNumeros = Array.new()
    end
```


Classe Agenda 2

La méthode `trouver(unNom)` devrait renvoyer :

- un numéro si le nom `unNom` existe
- chaine vide sinon

```
    def trouver(unNom)
        i = 0
        while i< @lesNoms.size() do
            if(@lesNoms[i]==unNom)
                return @lesNumeros[i]
            end
            i = i+1
        end
        return ""
    end
```


Classe Agenda 3

La méthode `ajouter(cont)` ne devrait rien ajouter si le `Contact cont` existe déjà.

```
    def ajouter(cont)
        unNom = cont.getNom();
        unNumero = cont.getNumero();
        if (trouver(unNom)=="")
            @lesNoms.insert(0,unNom)
            @lesNumeros.insert(0,unNumero)
        else
            puts "Le contact existe. Impossible
                d'ajouter "+unNom
        end
    end
```

Android : Ce code pourrait effectivement tourner sur un portable sous `Android`

L'héritage : classe de base

→ Quel est le résultat du code ci-dessous ?

```
class Automobile
  def vitesse_max_autoroute()
    return 130
  end
  def temps_trajet (distance)
    return distance/vitesse_max_autoroute()
  end
end
ma_voiture = Automobile.new
temps = ma_voiture.temps_trajet(130.0)
puts("Temps de trajet pour 130km en voiture"
    + temps.to_s()+ "heures.")
```

L'héritage : classe dérivée

→ Quel est le résultat du code ci-dessous ?

```
1 class Bus
2     def vitesse_max_autoroute()
3         return 90
4     end
5 end
6 mon_bus = Bus.new
7 temps = mon_bus.temps_trajet(130.0)
8 puts("Temps de trajet pour 130 km en bus: "+
      temps.to_s()+ "heures.")
```

- Comment peut-on appeler la méthode `temps_trajet` ?

L'héritage : classe dérivée

→ Quel est le résultat du code ci-dessous ?

```
1 class Bus<Automobile
2     def vitesse_max_autoroute
3         return 90
4     end
5 end
6 mon_bus = Bus.new
7 temps = mon_bus.temps_trajet(130.0)
8 puts("Temps de trajet pour 130 km en bus: "+
      temps.to_s()+ "heures.")
```

- Comment peut-on appeler la méthode `temps_trajet` ?
  Réponse : Par héritage → la classe `Bus` hérite toutes les méthodes et tous les `attributs` de la classe `Automobile`

Héritage classes Java

- Nous avons déjà vu la classe `android.content.Intent`
- La documentation est disponible si on cherche le nom de la classe sur Internet →

Analyse héritage

- Comme toute classe `Java`, cette classe hérite la classe `Object`, du paquetage `java.lang` (paquetage par défaut)

1 Écrire ses Propres Classes Ruby

2 Utiliser des Classes `Java` en `Ruby`
- Paquetage `java.util` : classes `Java` standard
- La bibliothèque graphique `Swing`

3 Notions de Java

JRuby : la puissance de Java en Ruby

- JRuby : une implémentation de Ruby qui permet d'utiliser des classes Java
  - Facile à installer/lancer : jruby ou jirb (interpréteur)
    - Java doit être installé aussi
- L'utilisation de classes Java **en principe** identique en Java ou Ruby
  - La syntaxe est assez différente : { = begin, } = end
  - Objectif d'écrire du code Ruby similaire avec un code Java
    - écrire puts("a="+a.to_s()) au lieu de puts "a=#{a}"
    - to_s en Ruby est remplacé toString en Java

Hashtable : une première classe java

Hashtable : une table de hachage

- un tableau indexé par des clés (pas forcement des entiers)
- chaque clé est associé à une valeur

Implémentation : l'en-tête du ficher doit indiquer l'utilisation de classes java ainsi que l'importation de la classe Hashtable

```
1 require "java";
2 import java.util.Hashtable;
```


Exemple Hashtable

1. Trouver la documentation de la classe : chercher les mots clés "Hashtable javadoc"
2. Observer quelques méthodes : put(clé,valeur), get(clé), containsKey(clé)
3. Construire un objet Hashtable et utiliser ses méthodes

```
1 agenda = Hashtable.new();
2 agenda.put("Daniel","1234");
3 agenda.put("Claire","1234");
4 if (not agenda.containsKey("Daniel"))
5     agenda.put("Daniel","9999");
6 end
7
8 puts(agenda.get("Daniel"));
```


Exemple StringTokenizer

Objectif : Couper une chaîne de caractères en plusieurs mots séparés par un délimiteur

- Une tache récurrente en programmation
- Le code s'explique tout seul avec un peu d'anglais
  - token=mot, next=suivant, hasMore=ilExisteEncore

```
1 require "java";
2 import java.util.StringTokenizer;
3
4 strTok = StringTokenizer.new("a b:x z:13",":");
5 while(strTok.hasMoreTokens())
6     puts strTok.nextToken()
7 end
```

Comment afficher let mots a, b, x, z, et 13 ?

Le paquetage `java.util`

Paquetage essentiel de `Java` qui mets à notre disposition :

- des classes utilitaires pour gérer les chaînes (`StringTokenizer`), mais aussi pour gérer les dates et le temps, les séquences aléatoires, etc.
- des grandes structures d'organisation de données (collections) : tableau de hachage (`Hashtable`), des tableaux, des listes triés, arbres, queues, piles, etc.

Documentation → `docs.oracle.com/javase/7/docs/api/java/util/package-summary.html`

1 Écrire ses Propres Classes Ruby

2 Utiliser des Classes `Java` en `Ruby`
- Paquetage `java.util` : classes `Java` standard
- La bibliothèque graphique `Swing`

3 Notions de Java

Pour quoi `Swing` ?

Swing : la principale bibliothèque graphique pour `Java`

- Une évolution importante apportée par `Java 2` en 1996
- Il y a des produits (`ajaxswing`) qui permettent de transformer une application `Swing` en une application Web
- Relativement plus facile quela bibliothèque Android
  - Les événements (I6) sont très similaires sur Android ou bien sur d'autres plate-formes

Objectifs `Swing` : I5, I6

Réaliser des applications Swing en `Ruby` et `Java`.

I5 La trace d'images 2D et l'animation
- pixels, lignes, polygones, etc.
- mini-jeux (échecs, poker)

I6 La programmation événementielle et l'interface utilisateur (ou GUI : `Graphical User Interface`)
- des buttons, clics, etc.

Exemple Swing en Jruby et Java

Jruby

```
1 require "java" ;
2 import javax.swing.JFrame;
3 import javax.swing.JLabel;
4 fenetre = JFrame.new("Titre␣Frame");
5 fenetre.getContentPane().add((JLabel.new("Bonjour")));
6 #fenetre.getContentPane().add((JLabel.new("Bonjour")));
7 #impossible, car le panel de base affiche un seul composant
8 fenetre.setLocationRelativeTo(NIL);
9 fenetre.setSize(100,500);
10 fenetre.setVisible(true);
11 fenetre.setDefaultCloseOperation(JFrame::DISPOSE_ON_CLOSE);
```

Java

```
1 import javax.swing.JFrame;
2 import javax.swing.JLabel;
3 public class Bonjour {
4  public static void main(String[] args) {
5   JFrame fenetre = new JFrame("Titre␣Frame");
6   fenetre.getContentPane().add((new JLabel("Bonjour")));
7   fenetre.setSize(500,500);
8   fenetre.setVisible(true);
9   fenetre.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
10  }
11 }
```


le cadre `JFrame` : un conteneur de base

- `JFrame` représente un conteneur de base (*top-level*)
- Tous les autres objets graphiques sont placés sur ce conteneur

Classes Swing

composant : des boutons (`JButton`), des labels (`JLabel`), des zones de texte (`JTextField`), des objets graphiques individuels

conteneur : un objet qui regroupe plusieurs composants : `JFrame` (top-level), des *panels* (`JPanel`)

- les composants sont ajoutés via `add(...)`

Objet `JFrame` : la fenêtre principale

Note programme Swing manipule une fenêtre `JFrame`

```
import javax.swing.JFrame;
...
fenetre = JFrame.new("Bonjour");
fenetre.setSize(500,500);
fenetre.setVisible(true)
...
```

Remplacer `JFramme` par `JApplet` ⇒ des applets WEB [1]

1. `mainline.brynmawr.edu/Courses/cs110/spring2002/Applets/Examples.html`

Les panels `Jpanel`

- Un objet `JFrame` dispose d'un panel ("content pane") accessible via la méthode `getContentPane()`
- Ce panel permet de afficher un composant unique (`JLabel`)
- Ce panel peut être remplacé avec un objet de classe `JPanel` (`setContentPanel(...)`)
  - l'objet `JPanel` permet d'afficher par défaut plusieurs composants ou encore d'autres panels

⇒

Hiérarchie des conteneurs et composants

Approche classique : utiliser `setContentPanel(...)` et ajouter plusieurs composants sur l'objet `JPanel`

Un exemple de dessin

```
 1 require "java"
 2 import javax.swing.JFrame
 3 class Toile<java.awt.Canvas
 4     def paint(g)
 5         g.drawRect(20,0,100,100)
 6         g.fillRect(120,100,100,100)
 7     end
 8 end
 9 maToile = Toile.new()
10 maToile.setPreferredSize(java.awt.Dimension.new
       (500,300))
11 frame = JFrame.new("Titre␣Frame");
12 frame.getContentPane().add(maToile)
13 frame.setSize(500,500);
14 frame.setVisible(true);
15 frame.setDefaultCloseOperation(JFrame::
       DISPOSE_ON_CLOSE);
```

Quel est le résultat du code ? Comment faire une animation ?

Développement application Swing : 3 étapes

1. écrire une nouvelle classe par héritage : `class Toile<java.awt.Canvas`
2. la classe `Canvas` possède une méthode `paint(g)` à redéfinir dans la classe dérivée (`Toile`)
   - Le paramètre g est un objet de classe `Graphics`
3. Construire l'objet `Toile` à placer sur le cadre `JFrame`
   - `maToile = Toile.new()`
   - Utiliser `setPreferredSize` pour une dimension précise : `maToile.setPreferredSize(java.awt.Dimension.new(500,300));`

Deux méthodes d'animation

`paint(g)` : l'objet est tracé en mode **XOR**($\oplus$)

- tracer couleur c_2 sur couleur $c_1 \Rightarrow$ couleur résultante $c_1 \oplus c_2$
- tracer encore une fois c_2 sur $c_1 \oplus c_2 \Rightarrow c_1 \oplus c_2 \oplus c_2 = c_1$

```
1 def paint(g)
2   g.setXORMode(java.awt.Color.white)
3   for i in (1..20) do
4       g.fillRect(120+i*10,100,100,100)
5       sleep(0.3)
6       g.fillRect(120+i*10,100,100,100)
7    end
8 end
```

- Autre méthode : appeler `repaint()` sur l'objet toile
  - résultat : la toile est effacée et `paint(g)` est re-appelée
  - modifier une variable globale `$i` qui indique la position d'un objet avant d'appeler `repaint()`

1. Écrire ses Propres Classes Ruby
2. Utiliser des Classes `Java` en `Ruby`
3. Notions de Java

Principes de base `java`

- Langage orienté objet présenté officiellement en 1995, 25 ans après `C` et 10 ans avant `Ruby`
- Objectif principal code facilement portable sur (presque) tout système d'exploitation et tout type de machine

---

- Java s'est inspiré de `C/C++` et a beaucoup influencé d'autres langages (`C#`)
  - Règles plus strictes que `Ruby`
- Un livre : Cyrille Herby. *Apprenez à programmer en Java*. Éditeur : Simple IT.

Étapes de programmation

Étapes :

1. Chaque classe `Java` doit être écrite dans un fichier séparé
2. Donner au fichier le même nom que la classe +`".java"`
   - Ex.: classe `Exemple` dans le fichier `Exemple.java`
   - Attention : ce nom doit commencer par une majuscule
3. La compilation : en ligne de commande avec `javac`
   - Ex. `javac Exemple.java` →`Exemple.class`
4. Exécution : `java`
   - Ex.`java Exemple`

La Compilation et l'exécution

1 Le fichier `Exemple.class` un exécutable virtuel indépendant de plate-forme
2 Une machine virtuelle `Java` exécute `Exemple.class`
   - La méthode `main()` est exécutée

Différences par rapport à `Ruby` et `C`

Ruby	pas d'exécutable ⇒ le programme réellement exécuté est assez différent du code
C/C++	exécutable machine généré : les instructions exécutées sont (presque) celles du code source

D'autres Nouveautés Java

- Il faut toujours définir une classe principale :
  - pour nous : classe `Exemple` dans le *fichier Exemple.java*
- La classe principale doit définir une méthode `main`
  - `public static void main(String[] args)`
  - cette déclaration n'est pas analysée pour l'instant
- utiliser { et } au lieu de begin et end

Le reste n'est pas si différent

- Construisons un premier exemple...

Premier programme Java

```ruby
require "java";
import java.util.StringTokenizer;
strTok= StringTokenizer.new("a b:x z:13",": ");
while(strTok.hasMoreTokens())
    puts strTok.nextToken()
end
```

Traduction Java →

```java
import java.util.StringTokenizer;
public class Exemple {
  public static void main(String[] args) {
    StringTokenizer strTok = new
       StringTokenizer("a b:x z:13",": ");
    while (strTok.hasMoreElements())
      System.out.println(strTok.nextToken());
}}
```


Paquetages java.swing et java.util

+ les mêmes classes sont utilisées : `StringTokenizer`, `JFrame`, `JPanel`, etc.
- Syntaxe `Java` plus stricte et un peu plus complexe
  - `System.out.println(...);` au lieu de `puts()`
  - `JFrame cadre = new JFrame("titre cadre")` au lieu de `cadre=JFrame.new("titre")`
  - Toute instruction est suivie par `;` : Le saut de ligne est uniquement utilisé pour l'indentation/lisibilité
  - `x=7; x="salut"` possible ? **NON** : chaque variable doit avoir un type (i.e. `Integer`, `String`, etc.), car `Java` n'est pas un langage interprété comme `Ruby`

Première fenêtre Java

```ruby
require "java"
import javax.swing.JFrame
import javax.swing.JLabel
frame = JFrame.new("Bonjour");
frame.getContentPane().add((JLabel.new("Bonjour")));
frame.setLocationRelativeTo(NIL)
frame.setSize(100,500);
frame.setVisible(true);
frame.setDefaultCloseOperation(JFrame::DISPOSE_ON_CLOSE);
```

Translation Ruby → Java:

```java
import javax.swing.JFrame;
import javax.swing.JLabel;
public class Bonjour {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Bonjour");
        frame.getContentPane().add((new JLabel("Bonjour")));
        frame.setLocationRelativeTo(null);
        frame.setSize(500,500);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    }
}
```


Tracer une figure : étape 1

1. Soit le code `Ruby`

```ruby
require "java"
class Toile<java.awt.Canvas
    def paint(g)
        g.drawRect(20,0,100,100)
        g.fillRect(120,100,100,100)
    end
end
```

2. Traduction `Java` → fichier Toile.java

```java
public class Toile extends java.awt.Canvas{
    public void paint(java.awt.Graphics g){
        g.drawRect(20,0,100,100);
        g.fillRect(120,100,100,100);
    }
}
```

Tracer une figure : étape 2

- Le programme principal est similaire au programme `Bonjour` précédent
- On va construire une classe principale `Cadre` dans un fichier Cadre.java :

```
1 import javax.swing.*;
2 public class Cadre {
3   public static void main(String[] args){
4     JFrame cdr = new JFrame("Cadre");
5     cdr.getContentPane().add(new Toile());
6     cdr.setSize(500,500);
7     cdr.setVisible(true);
8     cdr.setDefaultCloseOperation(JFrame.
        DISPOSE_ON_CLOSE);
9     }
10 }
```


Animations : méthode `sleep()`

- Un seul problème : l'appel `sleep(0.3)` ne fonctionne pas, car la méthode `sleep` n'existe pas en java.
- Il faut définir cette méthode dans la classe `Toile`

```
1 public class Toile extends java.awt.Canvas{
2  public void sleep(double seconds){
3    try{
4         Thread.currentThread().sleep(300);
5    }catch (Exception e){}
6  }
7   public void paint(java.awt.Graphics g){
8    //voir la diapo d'avant
9   }
10 }
```


Animations : modifier la classe `Toile`

- Traduction ligne par ligne : `Ruby` → `Java`

```
1 def paint(g)
2    g.setXORMode(java.awt.Color.white)
3    for i in (1..20) do
4        g.fillRect(120+i*10,100,100,100)
5        sleep(0.3)
6        g.fillRect(120+i*10,100,100,100)
7     end
8 end
```

```
1 public void paint(java.awt.Graphics g){
2   g.setXORMode(java.awt.Color.white);
3   for(int i=0;i<20;i++) {
4     g.fillRect(120+i*10,100,100,100);
5     sleep(0.3);
6     g.fillRect(120+i*10,100,100,100);
7    }
8 }
```