DÉMO
INF 2120

Démo 1 : pointeurs et héritage

Pointeurs

1. Écrivez le code pour une classe **Couleur**. Cette classe va contenir trois champs privés : **rouge**, **vert** et **bleu**. Utilisez des **int** pour ces champs. Ajoutez un constructeur, des ***get/set***, et une méthode **toString**() qui construit une chaine de caractères à l'aide des champs d'une instance qui seront placés entre parenthèse et séparés par des virgules. Les **set** doivent vérifier que les composantes (rouge, verte et bleu) soient entre 0 et 255.
2. Écrivez une classe principale avec un main. Dans ce main, construisez et utilisez des instances de la classe couleur :

```
public static void main( String [] args )
{
    Couleur c1 = new Couleur( 1, 4, 6 );
    Couleur c2 = c1;

    System.out.println( c2.toString() );

    c2.setRouge( 100 );

    System.out.println( c1.toString() );
    System.out.println( c2.toString() );
}
```

3. Écrivez une méthode du nom de **blanchir**() dans la classe Couleur. Cette méthode augmente chaque composante de la façon suivantes :

   - `rouge = ( rouge + 255 ) / 2`
   - `vert = ( vert + 255 ) / 2`
   - `bleu = ( bleu + 255 ) / 2`

   Ensuite ajouter les lignes suivantes à la méthode main :

```
c1.blanchir();
System.out.println( c1.toString() );
System.out.println( c2.toString() );
```

Héritage

1. Construisez une classe **Bien**, qui contiennent un champs **prixEtalage**.
2. Construisez une sous-classe **NonTaxable** qui hérite de **Bien**.
3. Construisez une sous-classe **TaxeSimple** qui hérite de **Bien**.
4. Construisez une sous-classe **TaxeDouble** qui hérite de **Bien**.
5. Construisez une sous-classe **Legume** qui hérite de **NonTaxable**.
6. Construisez une sous-classe **Livre** qui hérite de **TaxeSimple**.
7. Construisez une sous-classe **Meuble** qui hérite de **TaxeDouble**.
8. Ajouter une méthode **prix** à ces classes (pas à toutes les classes, seulement où nécessaire). Cette méthode retourne le prix d'un bien en y ajoutant la taxe. **TaxeSimple** ajoute 5% et **TaxeDouble** ajoute %5 et 9.975%.
9. Construisez une classe **Principale** contenant une méthode **main**. Ajouter dans cette classe une méthode statique qui reçoit en argument un tableau de **Bien** et calcule le montant de la facture.

Démo 2 : classe abstraite

Vous allez construire un exemple qui manipule des classes abstraites et concrètes. Nous allons réutiliser les **Forme2D** vues en classe.

1. Construisez la classe **Forme2D**. Ajouter un constructeur. Cette classe doit être abstraite et contenir une méthode abstraite pour le calcul de l'**aire**.
2. Construisez une classe pour les **Cercle** qui hérite de **Forme2D** et ajoutez un champ pour le **rayon**. Cette classe doit être concrète (elle doit implémenter la méthode d'**aire**). Ajouter une méthode **toString**.
3. Construisez une classe pour les **Rectangle** qui hérite de **Forme2D** et ajoutez un champ pour la **hauteur** et la **largeur**. Cette classe doit aussi implémenter la méthode d'**aire**. Ajouter une méthode **toString**.

Vous allez maintenant construire une deuxième hiérarchie de classe qui va utiliser les **Forme2D**.

1. Construisez une classe abstraite **Forme3D** qui contiennent une méthode abstraite de calcul pour le **volume**.
2. Construisez une classe **Sphere** qui hérite de **Forme3D** et ajoutez un champ pour le **rayon**. Cette classe doit aussi implémenter la méthode de **volume**. Ajouter une méthode **toString**.
3. Construisez une classe **CylindreDroit** qui hérite de **Forme3D**. Ajoutez un champ pour la forme de **base**, ce champ sera de type **Forme2D**. Aussi une champ **hauteur** doit être ajouté. N'oubliez pas le constructeur. Ajouter une méthode **toString**.
4. La classe **CylindreDroit** doit implémenter la méthode **volume**. Le **volume** du cylindre est l'**aire** de la forme de **base** multipliée par la **hauteur**.
5. Construisez-vous une classe **Principale** avec un **main** pour tester vos classes. Entre autres, faites un tableau de **Forme3D** et appliquez la méthode **volume** sur chaque élément du tableau. Affichez chaque élément du tableau.

Démo 3 : Type générique et ArrayList

Type générique

Écrivez les classes suivantes :

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

```
public abstract class PeutEtre <T> {
	public abstract boolean estQQChose();
	public abstract boolean estRien();
	public abstract T qQChose() throws ARien;
}
```

```
public class Rien <T> extends PeutEtre<T> {
	public Rien(){}
	public boolean estQQChose(){
		return false;
	}
	public boolean estRien(){
		return true;
	}
	public T qQChose() throws ARien{
		throw new ARien();
	}
}
```

```
public class QQChose <T> extends PeutEtre<T> {
	private T _valeur;
	public QQChose( T a_valeur ){
		_valeur = a_valeur;
	}
	public boolean estQQChose(){
		return true;
	}
	public boolean estRien(){
		return false;
	}
	public T qQChose() throws ARien{
		return _valeur;
	}
}
```

1. Étudiez-les et décrivez leurs fonctionnements et utilités.

2. Construisez une classe Principale et placez une méthode statique ayant la signature suivante :

```
public static <T> PeutEtre<Integer> trouverElement( T[] a_tableau, T
a_element )
```

Cette méthode fouille un tableau pour trouver un élément et retourne QQChose contenant l'indice de l'élément dans le tableau ou elle retourne Rien si l'élément n'est pas dans le tableau.

Tester votre méthode.

ArrayList

1. Écrivez une méthode statique ayant la signature suivante :

```
public static ArrayList<Double> tweens( double depart, double fin, int
nbrInterval )
```

Cette méthode construit un `ArrayList<Double>` qui aura `nbrInterval` + 1 éléments. Le premier élément aura la valeur `depart` et le dernier aura la valeur `fin`. Les éléments intermédiaires seront répartis équitablement dans l'intervalle. Par exemple, *`tweens`*`( 1.0, 3.0, 4 )` donnera la liste : 1.0, 1.5, 2.0, 2.5, 3.0. Il y a cinq valeurs, donc quatre intervalles égaux.

2. Testez votre méthode.

Démo 4 : Interface

1. Écrivez l'interface suivant :

```
public interface Nombre< N > {
      N add( N x );
      N sub( N x );
      N mul( N x );
      N div( N x );
}
```

Cette interface décrit les différentes fonctions entre deux nombres, soit l'addition, la soustraction, la multiplication et la division.

2. Écrivez une classe `NDouble` qui représente un `double` et qui implémente l'interface `Nombre`. L'entête de cette classe devrait être :

```
public class NDouble implements Nombre<NDouble>
```

Elle va contenir un champ de type `double` et un constructeur. Elle doit aussi implémenter les fonctions de l'interface et ajoutez un `toString` qui affiche le champ unique de l'instance.

3. Écrivez une classe `Fraction` qui implémente l'interface `Nombre`. Placez une méthode `toString`. Voici les règles simples pour chaque méthode :
   - add($\frac{a}{b}, \frac{c}{d}$) = $\frac{ad+bc}{bd}$
   - sub($\frac{a}{b}, \frac{c}{d}$) = $\frac{ad-bc}{bd}$
   - mul($\frac{a}{b}, \frac{c}{d}$) = $\frac{ac}{bd}$
   - div($\frac{a}{b}, \frac{c}{d}$) = $\frac{ad}{bc}$
4. Écrivez une classe principale qui contient un `main` pour tester vos deux classes.
5. Ajoutez une méthode **`static`** dans votre classe principale qui fera la somme d'un tableau de valeur. Cette méthode retournera **`null`** si le tableau est vide. Voici la signature de cette méthode :

```
public static < N extends Nombre< N > > Nombre< N > somme( ArrayList< N >
tableau )
```

6. Testez votre méthode à l'aide deux `ArrayList`, un premier de `NDouble` et un second de `Fraction`.

Démo 5 : TDA

File

Construisez la classe pour le type File.

```
public class File<T> {
	public File()
	}
	public int taille()
	}
	public boolean estVide()
	}
	public T tete() throws PileVide
	}
	public void enfiler( T a_element )
	}
	public void defiler() throws PileVide
}
```

Utilisez une référence sur le premier et le dernier élément. La direction de votre liste est un choix important pour simplifier le code.

Démo 6 : Préparation à l'intra.

Voir l'examen intra de l'été 2013 disponible sur le site du cours.

Démo 7 : Itérateur et Récursion

Itérateur

Construisez un itérateur pour la structure de File de la démo 5.

Fonction récursive.

Encodez les fonctions suivantes en utilisant la récursion.

1. Additionner(n, 0) = n
   Additionner(n, m) = Additionner(n+1, m-1)
2. Pgcd(n, 0) = n
   Pgcd(n, m) = Pgcd(m, n % m)
3. Cette fonction retourne une chaine de caractères :
   Dec2bin(n) = si n == 0 alors “0” sinon dec2bin_rec(n)
   Où
   Dec2bin_rec(0) = “”
   Dec2bin_rec(n) = Dec2bin_rec(n / 2) + (n % 2)

Transformez chacun des fonctions récursives en fonction itérative. Donc, en utilisant une simple boucle (**`while`** ou **`for`**).

Démo 8 : Stream et abstraction lambda

Stream

Vous trouverez une archive pour cette démo sur le site du cours. Cette archive contient les classes **`Client`**, **`Emprunt`**, **`Livre`**, **`LivreDejaEmprunte`** et **`ClientAtteintLimite`**.

1. Ajoutez une classe Bibliothèque qui contient les champs suivants :

```
private ArrayList< Client >  clients;
private ArrayList< Livre >   livres;
private ArrayList< Emprunt > emprunts;

public static final long LIMITE_PAR_CLIENT = 5;
```

   Ajoutez un constructeur qui initialise les listes.

2. Ajoutez une méthode **`public boolean`** `estEmprunte( Livre livre )`. Cette méthode va vérifier si un livre est emprunté. Vous devez utiliser un `Stream` pour faire cette vérification.
3. Ajoutez une méthode **`public long`** `nombreEmprunt( Client client )`. Cette méthode va compter le nombre de livre qu'un client à empruntés. Utilisez un `Stream`. Remarque : il existe une méthode `count()` qui peut être appliqué à un `Stream`. Cette méthode retourne le nombre d'élément contenu dans le `Stream`.
4. Ajoutez une méthode **`public void`** `emprunter( Client client, Livre livre )` **`throws`** `LivreDejaEmprunte, ClientAtteintLimite`. Cette méthode ajoute un nouvel emprunt dans la liste d'emprunt. Un exception est lancé si le livre est déjà emprunté ou si le client a déjà emprunter le nombre limite de livres permis (***`LIMITE_PAR_CLIENT`***).
5. Ajoutez une méthode **`public void`** `afficherLivresDisponibles()`. Cette méthode affiche le titre des livres non empruntés.
6. Ajoutez une méthode **`public`** `Object[] listeEmprunt( Client client )`. Cette méthode retourne un tableau des livres empruntés par un client. Utilisez un `Stream` et la méthode `toArray()` pour construire le tableau résultant.
7. Ajoutez une méthode **`public double`** `valeurInventaire()`. Cette méthode calcule la somme du prix des livres appartenant à la bibliothèque, peu importe s'ils sont empruntés ou non. Vous devez utiliser un `Stream`, la méthode `mapToDouble` permet de passer d'un `Stream` générique à un s de double natif.
8. Ajoutez une méthode **`public void`** `devaloriser( double perte )`. Cette méthode modifie le prix de chaque livre. Le nouveau prix est l'ancien prix multiplié par la valeur `( 1 - perte )`. Utilisez un `Stream` pour faire les changements.

Démo 9 : Interface utilisateur

GUI java swing.

Pour cette démo, vous allez construire un petit interface java swing. Allez chercher les fichiers du projet '***Synthetiseur***' sur le site du cours. Voici la liste des fichiers que vous y trouverez :

`ADSR, Bruit, Carre, Compose, Filtre, Graphic, Mixe, Onde, Pdemo, PulseGenerique, ScieD, ScieM, Sinusoidale, Triangle, TriangleGenerique.`

Le fichier **`Pdemo`** contient le 'main', c'est aussi le fichier que vous allez modifier. Ce projet fait des 'sons', vous pouvez utiliser des écouteurs pour les entendre.

Pour cette démo, nous allons :

- Construire une fenêtre (`JFrame`).
- Ajouter des composants dans notre fenêtre.
- Écouter des événements sur les composants de notre fenêtre.
- Implémenter des actions en fonction des événements.

Voici les éléments à ajouter, modifier. (Utilisez des `JLabel` pour décorer les différents éléments de l'interface au besoin.) Les éléments de l'interface seront placés dans le constructeur de la classe `Pdemo`.

1. En ce moment, le son est joué à l'ouverture de la fenêtre, ajoutez un `JButton` qui joue le son sur demande. Pour l'action à faire sur le bouton : `actionsJouerNote()`. Ajoutez le bouton dans la section `BorderLayout.`***`NORTH`*** de l'interface.
2. Ajouter un `JList<TypeOnde>` pour choisir le type d'onde pour l'onde 1 et 2. (Champs : `typeOnde`, ce champ est un tableau, l'indice 0 est la première onde et l'indice 1 est la deuxième onde.) La commande `addListSelectionListener` permet d'ajouter une action sur une liste. Consultez la documentation sur internet pour apprendre à placer les éléments dans la liste et vérifier lequel est choisie. (Remarquez, il est possible de construire la composante avec un tableau d'élément, et le type énuméré pour `TypeOnde` contient un tableau de tous les éléments possibles (`TypeOnde.`*`enumeration`*). Placez la liste pour la première onde dans la section `BorderLayout.`***`WEST`*** et celle pour la deuxième onde dans la section `BorderLayout.`***`EAST`***.
3. Ajouter un `JSlider` pour choisir le ratio haut/bas dans le cas d'une onde ***`PULSE_GENERIQUE`*** ou ***`TRIANGLE_GENERIQUE`***, faire cet ajout pour les 2 ondes. (Champs : `ondeRatio`.) Aussi, essayez d'activer/désactiver cette option dépendamment du type de l'onde choisie. Les valeurs permissent pour ce champ sont de 0.0 à 1.0. Par contre, un `JSlider` utilise des valeurs entières. Vous devrez permettre à la composante de prendre des valeurs de 0 à 100 et ensuite diviser la valeur obtenue par 100.0 pour avoir une valeur de type double. Placez la première composante dans la section `BorderLayout.`***`WEST`*** et la deuxième dans la section `BorderLayout.`***`EAST`***. Puisqu'il y a déjà des composantes dans ces sections, vous allez devoir construire des `JPanel` pour placer à la section `BorderLayout.`***`EAST`*** et un autre à la section `BorderLayout.`***`WEST`***. Déplacez vos listes du numéro précédant dans ces `JPanel` avec les `JSlider`. Utilisez un `GridLayout` dans les `JPanel`.

4. Ajouter un `JCheckBox` pour décider si la deuxième onde est utilisée. (Champs : `utilise2Ondes`.) Placez cette composante dans la section `BorderLayout.`***`SOUTH`*** de l'interface.
5. Ajouter un `JSlider` pour choisir le ratio de volume entre les deux ondes. (Champs : `ratioVolume`. Ce champ contient une valeur entre 0.0 et 1.0. Placez cette composante dans la section `BorderLayout.`***`SOUTH`*** de l'interface. Cela va vous demandez d'ajouter un `JPanel` dans la section `BorderLayout.`***`SOUTH`*** et d'y transferer le `JCheckBox` du numéro précédant.
6. Ajouter un `JSlider` pour choisir la durée. (Champs : `duree`.) Cette valeur doit être entre 0.0 et 2.0. Placez les limites de la composante pour choisir une valeur entre 0 et 2000 et ensuite divisez cette valeur par 1000.0. Ajouter cette composante dans la section `BorderLayout.`***`NORTH`*** de l'interface. Cela va vous demandez d'y place un `JPanel`, ne placez pas de `Layout` dans ce contenant.
7. Ajouter un `JTextField` pour la fréquence. (Champs : `frequence`.) Cette valeur est un double comprit entre 20 et 20000. Placez cette composante dans la section `BorderLayout.`***`NORTH`***.
8. Ajouter un `JCheckBox` pour choisir si le filtre est actif. (Champ : `utiliseFiltre`.) Placez cette composante dans la section `BorderLayout.`***`SOUTH`***.
9. Ajouter des `JSlider` pour choisir les valeurs ADSR du filtre. Il vous faudra quatre `JSlider`. (Champ : `filtreA`, `filtreD`, `filtreS` et `filtreR`.) Placez cette composante dans la section `BorderLayout.`***`SOUTH`***. Ils ont tous des valeurs entre 0.0 et 1.0.

Démo 10 : Recherche Binaire

Recherche Binaire

Nous allons utiliser le code de la classe `Fraction`. que vous pouvez trouver sur le site du cours, prenez la version pour la démo 10. Cette version implémente l'interface `Comparable<Fraction>`.

1- Écrivez une classe `AlgoFouille` qui contiendra le code pour la méthode de recherche binaire vu en classe et ayant la signature suivante : `public static < E extends Comparable< E > > int FouilleBinaire( E [] tableau, E element ) throws ElementNonPresent`. Ajoutez-y le code vu en classe. Aussi, il vous faudra construire une classe d'exception `ElementNonPresent`.

2- Utilisez la classe suivante pour tester votre méthode.

```
public class Principal {
   public static void main(String[] args) {
         final int TAILLE = 64;
         Fraction [] tab = new Fraction[ TAILLE ];

         // Utilisation d'un Stream pour remplir le tableau
         // de fraction consécutive (en ordre).
         int i = 0;
         Iterator< Fraction > it =
           new Fraction( 1, 2 ).sequence(
             new Fraction( 1, 16 ), TAILLE ).iterator();
         while( it.hasNext() ) {
               tab[ i ] = it.next();
               ++ i;
         }

         try {
               int indice =
                 AlgoFouille.FouilleBinaire( tab,
                   new Fraction( 21, 16 ) );
               System.out.println(
                 "La position de l'element est : " + indice );
         } catch( ElementNonPresent e ) {
               System.out.println(
                 "L'element n'a pas ete trouve." );
         }

         try {
               int indice =
                 AlgoFouille.FouilleBinaire( tab,
                   new Fraction( 21, 17 ) );
               System.out.println(
                 "La position de l'element est : " + indice );
         } catch( ElementNonPresent e ) {
               System.out.println(
                 "L'element n'a pas ete trouve." );
         }
      }
}
```

3- Ajouter une version récursive de la recherche binaire.
`public static < E extends Comparable< E > > int FouilleBinaireR( E [] tableau, E element, int debut, int fin ) throws ElementNonPresent.`
Cette version utilise deux paramètres supplémentaires pour permettre la recherche dans une sous-section du tableau.

4- Allez consulter la documentation de la classe `Arrays`. Cette classe contient des implémentations de l'algorithme de fouille binaire (`binarySearch`). Comment ont-ils résous le problème de l'élément non présent dans le tableau ?

Arbre Binaire de Recherche

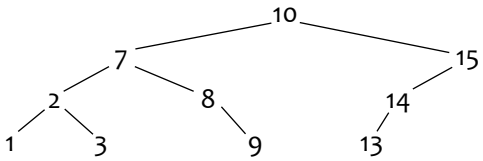
Dans un arbre initialement vide, insérez les éléments suivants :

20, 10, 42, 8, 5, 15, 45, 47, 30, 9.

Démo 11 : ABR

Arbre binaire de recherche

1. Effectuez les suppressions suivantes dans l'arbre : 3, 8, 10. Montrez l'arbre résultant après chaque suppression.



2. Donnez les listes de sommets obtenues par les parcours préfix, infixe et suffixe sur l'arbre précédant avant la suppression des nœuds.
3. Téléchargez le code `ABRNoeud` du site du cours. Ajoutez le type énuméré suivant :

```
public enum Parcours {
    PREFIXE, INFIXE, SUFFIXE
}
```

4. Modifiez la méthode `public List< E > elements( List< E > list )`. Afin qu'elle accepte un paramètre supplémentaire pour indiquer l'ordre de parcours de l'arbre. Testez vos résultats avec un `ArrayList`. Comme indiqué dans les commentaires, la méthode reçoit un `ArrayList` vide en argument pour y placer les éléments parcourus.
5. Ajoutez une méthode permettant de transformer un arbre en un nouvel arbre ayant la même forme, mais avec des éléments différents. Voici la signature :

```
/**
 * Construis un nouvel Arbre Binaire de Recherche en transformant
 * chaque élément à l'aide de la fonction de Transformation.
 * @param f
 *   Une fonction qui transforme les éléments de l'arbre.  Cette
 *   fonction doit conserver l'ordre des elements pour que l'arbre
 *   resultant soit valide.
 *   $a < b → f(a) < f(b)$
 *   $a == b → f(a) == f(b)$
 *   $a > b → f(a) > f(b)$
 * @return
 *   Un arbre ayant la même forme ou les éléments ont été
 *   tranformés par la fonction de transformation.
 */
public <R extends Comparable< R > > ABRNoeud< R > map(
                    Function< ? super E,? extends R > f
)
```

6. Testez votre méthode.

Démo 12 : Tri

Tri

1. Triez la liste suivante en utilisant les tries vues en classe : sélection, insertion, bulles.

   12, 4, 7, 3, 6, 8, 2, 10, 9

   Montrez la liste après chaque échange de valeurs.

2. Le code pour les quatre tries vues en classe est disponible sur le site. Téléchargez-le. Exécutez le code et consultez-le.
3. Ce code contient une deuxième version du `triRapide` (`triPlusRapide`, du moins, nous espérons qu'il est plus rapide). Ce tri utilise un deuxième algorithme de tri quand le tableau devient petit. Du code affiche le temps d'exécution pour cet algorithme sur un tableau de valeurs aléatoires (code de la méthode `testCharge`). Essayez le code pour une `TAILLE` de 10 000, 100 000, 1 000 000 et 10 000 000. Prenez note des temps d'exécutions.
4. Une constante ***`BARRIERE`*** indique à quelle taille l'algorithme de tri utilise `triInsertion` plutôt que `triPlusRapide`. Modifier le programme pour faire des tests avec les valeurs 0, 5, 10, 15, 20 et 25 pour la constante ***`BARRIERE`*** avec un tableau de 10 000, 100 000, 1 000 000 et 10 000 000 éléments (astuce : placez le tout dans une boucle qui modifie la constante, donc modifiez la classe pour que les constantes ne soient plus des constantes). Déterminez la valeur de ***`BARRIERE`*** qui est optimale.
5. Refaites les mêmes testes en remplaçant le `triInsertion` par un `triSelection` à l'intérieur de la méthode `triPlusRapide`.
6. La librairie `Arrays` de Java contient des méthodes de recherche et de tri. Testez la méthode `Arrays.`*`sort`* avec des tableaux de taille 10 000, 100 000, 1 000 000 et 10 000 000. Comparez les résultats avec ceux de nos méthodes.