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 : Récursion

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 récursive en boucle simple (**`while`** ou **`for`**)

Démo 7 : Interface utilisateur

GUI java swing.

Pour cette prochaine démo, vous allez construire un petit interface GUI, java swing. Allez chercher les fichiers du projet '***Synthetiseur***' sur ***Moodle***.

`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.
- Apprendre à modifier du code existant en y rajoutant de nouvelles fonctionalités.

Voici les éléments à ajouter, modifier. (Utilisez des **`JLabel`** pour décorer les différents éléments de l'interface au besoin.)

1. En ce moment, le son est joue à l'ouverture de la fenêtre, ajouter un bouton qui joue le son sur demande. Pour l'action à faire sur le bouton : prendre les 4 dernières lignes du fichier **`Pdemo`** :

```
Onde onde = ecran.construireOnde();
dessin.setFonction( onde );
dessin.repaint();
jouerNote( onde );
```

2. Ajouter un **`JPopupMenu`** pour choisir le type d'onde pour l'onde 1 et 2. (Champs : **`typeOnde`**.)

3. Ajouter un **`JTextField`** pour choisir le ratio d'une dans le cas d'une onde **`PULSE_GENERIQUE`** ou **`TRIANGLE_GENERIQUE`**, faire cet ajout pour les 2 ondes. (Champs : **`ondeRatio`**.) Aussi, essaye d'activer/désactiver cette option dépendamment de l'onde choisie.

4. Ajouter un **`JCheckBox`** pour décider si la deuxième onde est utilisée. (Champs : **`utilise2Ondes`**.)

5. Ajouter un **`JSlider`** pour choisir le ratio de volume entre les deux ondes. (Champs : **`ratioVolume`**.)

6. Ajouter un **`JSlider`** pour choisir la durée. (Champs : **`duree`**.)

7. Ajouter un **`JTextField`** pour la fréquence. (Champs : **`frequence`**.)

8. Ajouter un **`JCheckBox`** pour choisir si le filtre est actif. (Champ : **`utiliseFiltre`**.)

9. Ajouter des **`JTextField`** pour choisir les valeurs ADSR du filtre. (Champ : **`filtreA`**, **`filtreD`**, **`filtreS`**, **`filtreR`**.)

Enfin pour terminer, vous devez implémenter les actions sur tous composants que nous avons ajoutés. Obtenir les valeurs de chaque champ se trouvant dans l'interface et écrire le comportement approprié. Tous les comportements (les actions) doivent être implémentés dans la fonction **`public void actionPerformed`**(ActionEvent e).

Démo 8 : Recherche Binaire

GUI, suite

Terminer la séance sur les interfaces graphiques.

Recherche Binaire

Écrire le code d'une recherche binaire dans un tableau de chaînes de caractères de façon itérative et récursive. Implémentez l'interface ci-dessous :

```
public interface IRechercheBinaire {
	/**
	* Effectue une recherche binaire (Dichotomique) sous la
	* forme iterative
	* @param liste : La liste dans laquelle la recherche est
	* faite
	* @param valeurRecherchee : La valeure que l'on souhaire
	* trouvee dans la liste
	* @return la position de valeurRecherchee ou -1 si la
	* valeur introuvable dans la liste
	*/
	public int rechercheBinaireIterative(String [] liste,
	                                     String valeurRecherchee);
	/**
	* Effectue une recherche binaire (Dichotomique) sous la
	* forme recursive
	* @param liste : La liste dans laquelle la recherche est
	* faite
	* @param valeurRecherchee : La valeure que l'on souhaire
	* trouvee dans la liste
	* @param depart : Position de depart
	* @param fin: Position de fin
	* @return la position de valeurRecherchee
	* @throws RechercheBinaireException: Lance une exception
	* si la valeure est introuvable
	*/
	public int rechercheBinaireRecursive(String [] liste,
	                String valeurRecherchee, int depart,
	                int fin) throws RechercheBinaireException;
}
```

Arbre Binaire de Recherche

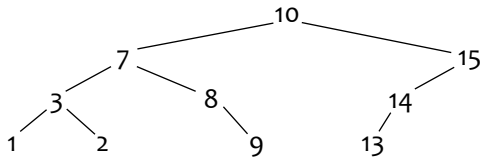
Exemples au tableau d'insertion dans un arbre binaire de recherche. Dans un arbre initialement vide, insérez les éléments suivants :

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

Démo 9 : Tri

Arbre binaire de recherche

Effectuez les suppressions suivantes dans l'arbre : 2, 8, 10. Montrez l'arbre résultant après chaque suppression.



Donnez les listes de sommets obtenues par les parcours préfixe, infixe et suffixe sur l'arbre précédant avant la suppression des nœuds.

Tri

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.

Programmez et testez le tri insertion.