Séquence 3 : Attributs et méthodes d'un objet

Objectifs : Appropriation du langage. Notion d'état interne de l'objet. Différence entre l'état interne et les services mis à disposition par l'objet.

Exercice 1

Réaliser en Java les actions suivants:

- On crée un tableau de 100 entiers natifs (c'est à dire *int*)
- On met la valeur 0 à l'index 0, 1 à l'index 1,... 99 à l'index 99.
- On calcule et affiche la somme des valeurs présentées alors par ces 100 entiers.
- On affiche cette valeur.

Correction

Fichier Launcher.java

```
public class Launcher {
	public static void main(String[] args) {
		int [] tab=new int[100];
		for(int i=0;i<100;i++) tab[i]=i;
		int sum=0;
		for(int i:tab) sum+=i;
		System.out.println(sum);
	}
}
```

Exercice 2

On reprend le compteur simple de la séquence 2. Réaliser en Java les actions suivants:

- On crée un tableau de 100 compteurs
- On incrémente 0 fois le compteur à l'index 0, 1 fois celui à l'index 1,... 99 fois celui à l'index 99.
- On calcule et affiche la somme des valeurs présentées alors par ces 100 compteurs
- On affiche cette valeur.

Correction

Fichier Launcher.java

```
public class Launcher {
	public static void main(String[] args) {
		Compteur [] tab=new Compteur[100];
		for(int i=0;i<100;i++) tab[i]=new Compteur();
		for(int i=0;i<100;i++)
			for(int k=0;k<i;k++)
				tab[i].up();
		int sum=0;
		for(Compteur c:tab) sum+=c.getValue();
		System.out.println(sum);
	}
}
```

Exercice 3

Réaliser en Java les actions suivants en sollicitant l'*autoboxing*:

- On crée un tableau de 100 entiers objet (c'est à dire *Integer*)
- On met la valeur 0 à l'index 0, 1 à l'index 1,... 99 à l'index 99.
- On calcule et affiche la somme des valeurs présentées alors par ces 100 entiers.
- On affiche cette valeur.

Correction

Fichier Launcher.java

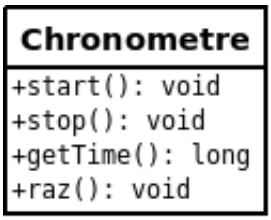
```
public class Launcher {
	public static void main(String[] args) {
		Integer [] tab=new Integer[100];
		for(int i=0;i<100;i++) tab[i]=i;
		int sum=0;
		for(Integer i:tab) sum+=i;
		System.out.println(sum);
	}
}
```

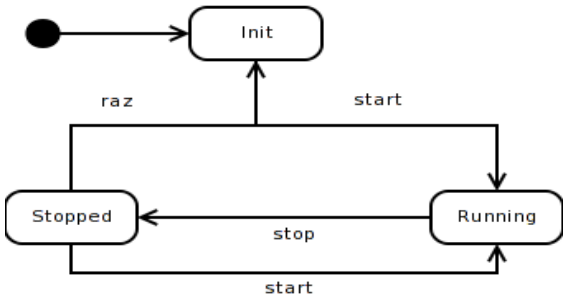
Exercice 4

On cherche à créer une classe reproduisant les fonctionnalités typiques d'un chronomètre simple.
Un chronomètre est vue comme une entité présentant les services suivants :

- *start* : démarre le comptage des secondes
- *stop* : stoppe le comptage des secondes
- *raz* : remise à zéro
- *getTime* : permet d'accéder (en millisecondes) à l'indication de durée.

On considère que ce chronomètre peut présenter 3 états internes différents : état *init* (juste après l'instanciation), état *running* et état *stopped*.





Ce schéma indique les transitions d'état importantes. Quand une action ne change pas l'état courant (*start* ou *raz* sur l'état *Running* par exemple) elle n'est pas indiquée. La logique de fonctionnement du chronomètre est simulée sans faire appel à la notion de *thread*. La classe *java.util.Date* sera utilisée. La création d'un objet par *new Date* permet de saisir l'instant courant. L'invocation de *getTime* sur cet objet permet d'obtenir un entier long représentant le nombre de millisecondes écoulées depuis le 01/01/1970.

- Q1 : établir l'interface de la classe *Chronometre*

Correction

```
public interface IChronometre {
	public  void start();
	public  void raz();
	public  void stop();
	public  long getTime();
}
```

Rem : le mot réservé *interface* utilisé ici sera justifié plus tard

- Q2 : Imaginer une implémentation interne de ses trois états

Correction

Il y a plusieurs solutions dont certaines anticipent sur le cours.
Le plus simple est de doter l'objet d'un attribut interne de type entier (*int*) nommé *state* ci-dessous et d'associer arbitrairement chaque état à une valeur entière (0 pour *init*, 1 pour *running* et 2 pour *stopped* par exemple)

```
class Chronometre {
	protected int state;
	public Chronometre() {
		state=0;
	}
	etc...
}
```

On peut expliciter cette association entre un entier et un état en ajoutant 3 identifiants dans le but de rendre la lecture du code plus facile dans les méthodes.

```
class Chronometre {
	protected int state;
	protected int INIT=0, RUNNING=1, STOPPED=2;
	public Chronometre() {
		state=INIT;
	}
	etc...
}
```

Ces trois identifiants sont en fait des constantes car leur valeur ne doit changer au cours du temps. C'est la valeur de *state* qui change en fonction de l'état du chronomètre et non la valeur de *INIT*, *STOPPED* ou

RUNNING. On peut expliciter ça en en faisant des constantes au moyen du mot-clef *final*. Ce type d'identifiants constants est conventionnellement écrit en majuscules et appelés constantes..

```
class Chronometre {
	protected int state;
	protected final int INIT=0, RUNNING=1, STOPPED=2;
	public Chronometre() {
		state=INIT;
	}
	etc...
}
```

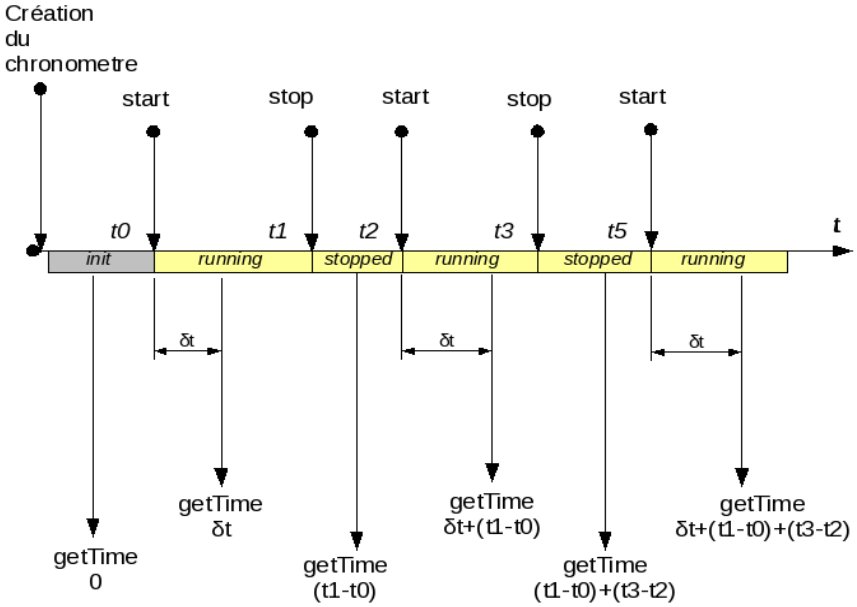
Enfin cette formulation conduit à ce que chaque instance de chronomètre embarque avec elle un attribut d'état (*state*) ce qui est bien normal, mais aussi 3 attributs qui n'ont que vocation à rappeler l'association faite dans le code entre un entier (par exemple 0) et un état (par exemple INIT). S'il y a 500 chronomètres dans l'application il y aura donc 500 variables *state* embarquées (ce qui est normal puisque chaque chronomètre peut être dans un état distinct de l'autre). Mais il y aura également dans chaque instance 500 exemplaires de la variable INIT possédant la même valeur 0, 500 variables STOPPED possédant la valeur 2 et 500 variables RUNNING possédant la valeur 1. C'est évidemment absurde et l'utilisation de variables dites *statiques* permettent d'indiquer qu'on souhaite une occurrence unique dans une classe déterminée de la variable marquée par le mot-clef *static*. D'où la formulation finale proposée ici :

```
class Chronometre {
	protected int state;
	protected final static int INIT=0, RUNNING=1, STOPPED=2;
	public Chronometre() {
		state=INIT;
	}
	etc...
}
```

Remarque : depuis la version 1.5 java propose une formulation élégante de constantes au moyen du mot réservé *enum*, voir le manuel technique sur cette question.

```
public class Chronometre {
	enum STATE {INIT, RUNNING, STOPPED };
	STATE state;
	public Chronometre() {
		state=STATE.INIT;
	}
	etc...
}
```

- Q3 : Établir l'algorithme de *getTime* (en fonction de l'état du chronomètre). En déduire les variables membres nécessaires pour exprimer complètement l'état interne d'un chronomètre. Le schéma suivant explique le principe de calcul de la valeur rendue par *getTime*.



La valeur rendue par getTime est donc la suivante :

- dans l'état *stopped* : $\sum t_{2i+1} - t_{2i}$
- dans l'état *running* : $\delta t + \sum t_{2i+1} - t_{2i}$

– dans l'état *init* : 0

A chaque appui sur *start* il faut donc mémoriser l'instant t_{2i} correspondant (par exemple dans *lastStartDate*) de façon à être en mesure de calculer *δt* (par *newDate().getTime()-lastStartDate*). A chaque appui sur *stop* il faut donc mémoriser la valeur du cumul $\sum t_{2i+1} - t_{2i}$ dans la variable *sum* (par exemple) en effectuant *sum= sum+ newDate().getTime()-lastStartDate.* Chaque instance de chronomètre devra donc embarquer ces deux variables (*lastStartDate* et *sum*) de façon à pouvoir calculer la valeur rendue par *getTime* quelque soit l'état interne (*init, running* ou *stopped*) de cette instance.

Correction

```
public long getTime() {
      switch (state) {
      case INIT:return 0L;
      case RUNNING:return new Date().getTime()-lastStartDate.getTime()+sum;
      case STOPPED:return sum;
      default: return 0L;
      }
}
```

- Q4 : Implémenter

Correction

```
import java.util.Date;

public class Chronometre {
      Date lastStartDate;
      final static int INIT=0, RUNNING=1, STOPPED=2;
      long sum=0;
      int state;
      public Chronometre() {  state=INIT; }
      public void start() {
            switch (state) {
            case INIT:lastStartDate=new Date();state=RUNNING;break;
            case RUNNING:break;
            case STOPPED:lastStartDate=new Date();state=RUNNING;break;
            }
      }
      public void raz() {
            switch (state) {
            case INIT:break;
            case RUNNING:break;
            case STOPPED:sum=0;state=INIT;break;
            }
      }
      public void stop() {
            switch (state) {
            case INIT:break;
            case RUNNING:state=STOPPED;;sum+=
                  new Date().getTime()-lastStartDate.getTime();break;
            case STOPPED:break;
            }
      }

      public long getTime() {
            switch (state) {
            case INIT:return 0L;
            case RUNNING:return new Date().getTime()-lastStartDate.getTime()+sum;
            case STOPPED:return sum;
            default: return 0L;
            }
      }
}
```

- Q5 : Tester. Exemple de code de test:

`public static void delay(int ms) {`	**Résultat :**

```
        try { Thread.sleep(ms);
        } catch (InterruptedException e) { }
    }

    public static void main(String[] args) {
        Chronometre chrono=new Chronometre();
        System.out.println(chrono.getTime());
        chrono.start();
        delay(100);
        System.out.println(chrono.getTime ());
        delay(100);
        delay(100);
        chrono.stop();
        System.out.println(chrono.getTime ());
        delay(100);
        chrono.start();
        delay(100);
        chrono.stop();
        System.out.println(chrono.getTime ());
        chrono.raz();
        System.out.println(chrono.getTime ());

    }
```

```
0
103
314
418
0
```