FACULTAD DE INGENIERÍA

FORMATO
PRÁCTICAS DE LABORATORIO

UNIVERSIDAD AUTÓNOMA DE CAMPECHE

NOMBRE DEL PROFESOR: Ing. Héctor Manuel Quej Cosgaya
NOMBRE DE LA PRÁCTICA: Abstracción
PRÁCTICA NÚM. [5]

LABORATORIO:	Centro de Ingeniería Computacional
MATERIA:	Lenguaje de Programación II
UNIDAD:	Subcompetencia II
TIEMPO:	2 horas

OBJETIVO:
Comprender el concepto de Abstracción, sus ventajas e implementación.

MARCO TEÓRICO:
La Abstracción es el proceso mediante el cual, el programador puede dividir un problema complejo en tareas más simples, extrayendo la esencia del mismo. La abstracción permite concentrarnos en una única tarea y darle toda nuestra atención, ignorando el resto, dando por hecho que ya se encuentran resueltas. Concentrándonos en el resultado de una tarea, y no en la manera en cómo fue llevada a cabo, nos permite escribir programas capaces de resolver problemas complejos de una manera muy sencilla. La abstracción por sí misma requiere de un trabajo constante y de mucha práctica para poder adquirirse, sin embargo, esta práctica busca proporcionar a los estudiantes con un sencillo ejemplo del poder que conlleva.

LISTA DE MATERIALES:
• Java SDK • Bloc de notas / Editor SciTE • Archivo con código fuente 'TareaPeriodica.java' • Archivo con código fuente 'Ejecutor.java' • Archivo con código fuente 'Reloj.java'

EQUIPO DE LABORATORIO:
• Computadora Personal

DESARROLLO DE LA PRÁCTICA:
1. **Abre** el archivo '`TareaPeriodica.java`', que se te proporcionó junto con esta práctica. 2. **Examina** la clase.

Observa los dos métodos de `TareaPeriodica` llamados `necesitaEjecucion()` y `ejecutarTarea()` solamente regresan un valor de retorno. Este tipo de métodos son llamados *kludge*. Dado que este tipo de métodos son torpes e inelegantes, vamos a corregirlos.

3. **Borra** los valores de retorno tanto ambos métodos (`necesitaEjecucion()` y `ejecutarTarea()`).
4. **Compila** la clase:

Todo método con un valor de retorno ***necesita*** retornar algo. Pero en nuestra clase, ambos métodos no pueden hacer nada (¿qué tarea se supone que vamos a ejecutar? ¿Cómo saber si dicha tarea necesita ejecutarse?). Como dichas operaciones no se implementarán aquí, sino en las clases que heredarán de `TareaPeriodica`, lo más sensato es que estos métodos se declaren **abstractos**.

5. **Agrega** el modificador `abstract` a ambos métodos, después de su modificador de acceso.
6. **Compila** la clase.

Dado que los métodos **abstractos** deben ser implementados en clases descendientes, éstos no pueden tener cuerpos.

7. **Borra** las llaves de apertura y de cierre de ambos métodos. En su lugar, coloca un punto y coma.
8. **Compila** la clase.

Un método abstracto, por definición, está incompleto. Por tanto, la clase que lo contiene también debe marcarse como incompleta, y por ende, debe ser declarada **abstracta**.

9. **Agrega** el modificador `abstract` a la definición de la clase `TareaPeriodica`, justo después de su modificador de acceso.
10. **Compila** la clase.

La clase `TareaPeriodica` ahora luce bien, pero ¿qué ha pasado con las otras dos?

11. **Abre** la clase '`Ejecutor.java`' e intenta **compilarla**.
12. **Repite** el paso 11 con la clase '`Reloj.java`'

En ambos casos aparece el mismo error: 'Tal clase no es abstracta y no sobrescribe tal método de la clase `TareaPeriodica`'. Esto ocurre porque ambas clases heredan de una clase **abstracta**, es decir, **incompleta**, y todavía no han hecho nada para completarla. Hagámoslo.

13. **Agrega** los siguientes métodos en la clase `Ejecutor`.

```
public boolean necesitaEjecucion() {
   if (!activa) return false;

   // Calcula la hora de la próxima ejecución
   Calendar proximaEjecucion = new GregorianCalendar();
   proximaEjecucion.setTime(ultimaEjecucion);
   proximaEjecucion.add(Calendar.SECOND, periodo);

   Calendar ahora = new GregorianCalendar();

   // Comprobamos si ha pasado a la hora actual
   return (proximaEjecucion.before(ahora));
}

public int ejecutarTarea() {
   try {
      Runtime.getRuntime().exec(comando);
      return 0;
   } catch(IOException e) {
      System.err.println(e.toString());
   }
   return -1;
}
```

14. **Compila** la clase `Ejecutor`. El error ha desaparecido.
15. **Agrega** los siguientes métodos a la clase `Reloj`.

```
public boolean necesitaEjecucion(){
   Calendar cal = new GregorianCalendar();
   cal.setTime(ultimaEjecucion);
   cal.add(Calendar.SECOND, periodo);

   Calendar ahora = new GregorianCalendar();
   return ahora.before(cal);
}

public int ejecutarTarea() {
   Calendar c = new GregorianCalendar();
   System.out.println(String.format("%d%d%d", c.get(Calendar.Hour)
      , c.get(Calendar.MINUTE), c.get(Calendar.SECOND)));
   return 0;
}
```

16. **Compila** la clase `Reloj`. El error también ha desaparecido.
17. **Crea** una clase controladora para probar la clase `Ejecutor`.

```
public static void main(String[] args){
   Ejecutor e = new Ejecutor("calc", 1);
   Scanner leer = new Scanner(System.in);
   System.out.println("Escribe un commando para ejecutar");
   e.comando = leer.next();
   if (e.necesitaEjecucion()) {
      e.ejecutarTarea();
   }
}
```

18. **Ejecuta** la clase controladora recién creada.
19. **Modifica** la clase controladora para probar ahora la clase Reloj.

¿Por qué tomarse tantas molestias? Paciencia, pequeño saltamontes. Con el polimorfismo todo quedará claro. El verdadero poder de la abstracción se manifiesta en conjunto con los demás mecanismos de la Programación Orientada a Objetos.

Tema Extra: **Interfaces**

Las interfaces ahora son tema de Lenguaje de Programación III. Sin embargo, para que los siguientes temas queden claros, te brindaremos una pequeña introducción.

20. **Cambia** la definición de la clase `TareaPeriodica` de la siguiente manera:

```
public abstract class TareaPeriodica implements Runnable {
   …
}
```

21. **Compila** la clase `TareaPeriodica`. No parece haber ningún problema, ¿verdad?
22. **Compila** la clase `Ejecutor`. ¿Qué pasó?
23. **Compila** la clase `Reloj`. ¿Qué es lo que pasa aquí?

Las **interfaces** son clases totalmente abstractas. No son heredadas, sino implementadas. Cualquier método especificado en una interfaz debe ser ***definido*** en cualquier clase que la implemente. Puede verse a una interfaz como un **contrato** que las clases deben cumplir.

24. **Agrega** el siguiente método tanto a la clase `Ejecutor` como a la clase `Reloj`:

```
public void run() {
   if (this.necesitaEjecucion()) {
      this.ejecutarTarea();
   }
}
```

¿El mismo método para ambas clases? ¿Qué acaso el instructor se volvió loco? No es magia, es abstracción.

25. **Compila** ambas clases. El error desaparece.
26. **Modifica** el método `main` de tu clase controladora de la siguiente manera.

```
public static void main(String[] args) {
   new Thread(new Reloj()).start();
   new Thread(new Ejecutor("calc", 0)).start();
}
```

27. **Compila** y **ejecuta** tu clase controladora. ¿Funciona?

Esto es solo una **pequeñísima muestra del poder de la abstracción**.

Fin de la práctica

RETROALIMENTACIÓN:

- Escribe cinco ejemplos diferentes en donde crees jerarquías de objetos, por ejemplo, DispositivoMovil > ConsolaPortatil > GameBoy.

RECOMENDACIONES ADICIONALES:

- Investiga más acerca de las interfaces: qué son y para qué se utilizan. Busca además para qué se usan las siguientes interfaces comunes:
  - Runnable
  - Clonable
  - Serializable

BIBLIOGRAFÍA:

- Dean, J. S., & Dean, R. H. (2009). Introducción a la programación con Java. México: Mc Graw Hill.
- The Java Tutorials: Interfaces and Inheritance: http://docs.oracle.com/javase/tutorial/java/IandI/index.html
- Apuntes del profesor.