FACULTAD DE INGENIERÍA

FORMATO
PRÁCTICAS DE LABORATORIO

UNIVERSIDAD AUTÓNOMA DE CAMPECHE

NOMBRE DEL PROFESOR: Ing. Héctor Manuel Quej Cosgaya
NOMBRE DE LA PRÁCTICA: Herencia
PRÁCTICA NÚM. [4]

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

OBJETIVO:

Comprender el principio de la Herencia: la manera en como se implementa, sus mecanismos de funcionamiento y las palabras reservadas relacionadas con ella.

MARCO TEÓRICO:

La Herencia de la Programación Orientada a Objetos sigue el mismo principio de la herencia del mundo real: la transmisión de características de una entidad a otra. De este modo, una clase puede heredar los miembros de otras clases: atributos y métodos pueden ser utilizados sin tener que escribirlos de nuevo. La Herencia nos permite, entre otras cosas, un modelado más natural de la realidad, ahorrar código y extender la funcionalidad de nuestras clases de una manera rápida y sencilla. En la práctica, se procede a ejemplificar los fundamentos de la Herencia, así como los diferentes mecanismos que posee para ampliar su funcionalidad.

LISTA DE MATERIALES:

- Java SDK
- Bloc de notas / Editor SciTE
- Archivo de código fuente 'DispositivoMovil.java'
- Archivo de código fuente 'TestDispositivo.java'

EQUIPO DE LABORATORIO:

- Computadora Personal

DESARROLLO DE LA PRÁCTICA:

Primera parte: **Implementación de la Herencia**

1. **Abre** el archivo '`DispositivoMovil.java`', que se te proporcionó junto con esta práctica.
2. **Examina** la clase.

Observa que salvo el modificador de acceso `protected`, `DispositivoMovil` es una clase bastante normal.

3. **Compila** la clase.
4. **Crea** una nueva clase llamada `Laptop`, y haz que *herede* los atributos y métodos de `DispositivoMovil` utilizando la siguiente declaración:

```
public class Laptop extends DispositivoMovil {

}
```

5. **Guarda** la nueva clase en un archivo llamado '`Laptop.java`'.
6. **Compila** la clase `Laptop`.

A pesar de que la clase `Laptop` está aparentemente vacía, en realidad contiene todos los miembros no-privados de `DispositivoMovil`. Vamos a comprobarlo.

7. **Abre** el archivo '`TestDispositivo.java`', que se te proporcionó junto con esta práctica.
8. **Compila** y **ejecuta** la clase `TestDispositivo`.

Observa que `TestDispositivo` crea un objeto de tipo `Laptop`, y llama a sus métodos `encender()`, `verCargaBateria()` y `apagar()`. `Laptop` es capaz de llamar a esos métodos ya que los heredó de `DispositivoMovil`. Sin embargo, dichos valores no son inicializados en ningún lado.

9. **Agrega** el siguiente constructor en la clase `Laptop`:

```
public Laptop() {
   velocidadProcesador = 2.47f;
   cargaBateria = 100;
   memoria = 4096;
   sistemaOperativo = "Microsoft Windows 7";
   marca = "HP";
   modelo = "Pavilion dv5-2247la";
}
```

10. **Compila** la clase `Laptop`. El compilador lanzará un error.

Dado que los atributos `memoria` y `sistemaOperativo` fueron declarados como **privados**, la clase `Laptop` no los heredó: por tanto no puede utilizarlos. Una clase hereda únicamente los miembros **públicos** y **protegidos** de su superclase.

11. **Cambia** el modificador de acceso de todos los atributos de DispositivoMovil para hacerlos

protegidos.

12. **Compila** la clase `DispositivoMovil`.
13. **Compila** la clase `Laptop`. El error ha desaparecido.

Segunda parte: **Encadenamiento de constructores y palabra reservada super**

Ahora viene algo interesante. ¡Presta MUCHA atención, que es importante!

14. **Agrega** el siguiente constructor en la clase `DispositivoMovil`:

```
public DispositivoMovil (int x) {
   System.out.println("Haré que Laptop deje de funcionar! :D");
}
```

¿Será cierta esa amenaza? ¡Comprobémoslo!

15. **Compila** la clase `DispositivoMovil`.
16. **Compila** la clase `Laptop`. ¿¡*Qué pasó aquí*!?

Java inserta en cada constructor de una **subclase**, una llamada *implícita* a un constructor de la **superclase**. Anteriormente, como no teníamos ningún constructor definido en `DispositivoMovil`, Java le agregaba el constructor por defecto (que como ya sabes, no recibe parámetros). En este caso, no había problema: `Laptop` llamaba implícitamente al constructor por defecto de `DispositivoMovil`.

Sin embargo, una vez que agregamos *al menos un constructor* a `DispositivoMovil`, Java no agrega el constructor por defecto a `DispositivoMovil`. De este modo, cuando `Laptop` quiere llamar al constructor sin parámetros de `DispositivoMovil` (que ya no existe, pues nunca fue agregado) es que el compilador lanza el error. ¡Vamos a solucionar este problema!

17. **Agrega** la siguiente sentencia como la *primera* instrucción del constructor de `Laptop`:

```
super(0);
```

18. **Compila** la clase `Laptop`. ¡Bye bye, error!

Haciendo una llamada **explícita** al constructor de la **superclase**, Java ya no agrega la llamada *implícita*. La llamada a un constructor de la superclase se hace con la palabra reservada **`super`**, seguida de los argumentos definidos en algún constructor de la superclase (de manera similar a como se hace con la palabra reservada **`this`**)

Tercera parte: **Mecanismos de la herencia**

19. **Crea** una nueva clase llamada `Celular`, y haz que herede de `DispositivoMovil`

```
public class Celular extends DispositivoMovil {

}
```

20. **Guarda** la clase en un archivo llamado '`Celular.java`'.

Un celular *es un* dispositivo móvil, pero que tiene la capacidad de realizar llamadas. `DispositivoMovil` no tiene dicha capacidad, pero no hay problema, ¡implementémosla!

21. **Agrega** el siguiente fragmento de código a la clase `Celular`.

```
private String numero;

public Celular(String numero) {
   super(0);
   this.numero = numero;
}

public void llamar(String numero) {
   System.out.println("Llamando… " + numero);
}
```

22. **Compila** la clase `Celular`.
23. En la clase `TestDispositivo`, añade el siguiente fragmento de código al final del método `main`:

```
Celular cel = new Celular("123-4567890");
cel.encender();
cel.verCargaBateria();
cel.llamar("098-7654321");
```

24. **Compila** y **ejecuta** a `TestDispositivo`.
25. **Añade** la siguiente sentencia al *final* del método `main` en la clase `TestDispositivo`:

```
lap.llamar("123-4567890");
```

26. **Compila** la clase de nuevo.

Llamar es un método añadido por `Celular`, por lo que la clase `Laptop` no conoce nada sobre él. Los métodos añadidos por una subclase no afectan a las demás subclases.

27. **Borra** la sentencia que añadiste en el paso 25.
28. En la clase `Celular`, **redefine** el método `apagar()` de la siguiente manera:

```
public void apagar() {
   System.out.println("No quiero apagarme! :D");
}
```

29. **Compila** la clase `Celular`.
30. De regreso a la clase `TestDispositivo`, **añade** las siguientes sentencias al final del método `main`:

```
lap.apagar();
cel.apagar();
```

31. **Compila** y **ejecuta** la clase `TestDispositivo`.

De manera análoga a la *adición*, una **redefinición** en la subclase no afecta a las demás subclases.

32. En la definición de `apagar()` en la clase `Celular`, **realiza** una llamada al método original `apagar()` de la superclase, utilizando la siguiente sintaxis:

```
public void encender() {
   super.encender();
   System.out.println("No quiero apagarme! :D");
}
```

33. **Compila** la clase `Celular`.
34. **Compila** y **ejecuta** la clase `TestDispositivo`.

Observa el cambio en el comportamiento del método `apagar()` en `Celular`: se conserva el comportamiento original de la **superclase**, más el comportamiento añadido en la **subclase**. La clase `Laptop` no resulta afectada en lo más mínimo.

35. **Modifica** la declaración del método `apagar()` en la clase `DispositivoMovil` para hacerlo ***final***:

```
public final void apagar() {
   […]
}
```

36. **Compila** la clase `DispositivoMovil`.
37. **Compila** la clase `Celular`. El compilador te marcará un error.
38. **Borra** la redefinición del método `apagar()` en `Celular`.
39. **Compila** la clase `Celular` de nuevo.
40. **Modifica** la declaración de la clase `DispositivoMovil` para hacerla ***final***.

```
public final class DispositivoMovil {
   […]
}
```

41. **Compila** la clase `DispositivoMovil`.
42. Intenta **compilar** cualquiera de las otras dos clases. ¡Tanto trabajo ahora se ha perdido!
43. **Borra** la cláusula '***final***' de la declaración de la clase `DispositivoMovil`.
44. **Compila** todas las clases de nuevo.

Cuarta parte: La Clase Object

45. **Añade** el siguiente código al final del método `main` en la clase `TestDispositivo`:

```
Laptop lap2 = new Laptop();
System.out.println("Lap es igual a Lap2? : " + lap.equals(lap2));
System.out.println("El objeto de " + lap2.getClass().toString() +
   " llamado lap2 vive en " + lap2.toString());
```

46. **Compila** la clase `TestDispositivo`.

Hemos utilizado un montón de métodos raros. ¿De donde salieron? ¡Pues de la superclase de todas las clases, la clase ***Object***!

47. **Ejecuta** la clase `TestDispositivo`.

Fin de la práctica

RETROALIMENTACIÓN:

- Investiga para qué sirven los siguientes métodos de la clase Object
  - equals
  - hashCode
  - toString
  - clone
- Escribe otras dos clases que hereden de DispositivoMovil (por ejemplo, Tablet o GameBoy). Implementa nuevos métodos en cada una de ellas y pruébalos con una clase controladora.

RECOMENDACIONES ADICIONALES:

- Lee el capítulo 12 del Dean (Agregación, Composición y Herencia)

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.