1. Clases y objetos

Cada vez que se analiza un problema a resolver utilizando la computadora y en este caso desarrollando una solución con una herramienta orientada a objetos, el análisis se debe hacer sobre los objetos involucrados y la interacción que entre ellos existe.

Tema 1. Metodología para el modelado de clases

El diseño ayuda a construir un modelo de objetos para un sistema a desarrollar. Un modelo de objetos especifica las clases, los atributos, los métodos y sus relaciones con otras clases.

Por ejemplo al hablar de clientes, artículos y proveedores, inmediatamente se debe analizar cada uno por separado, así como la interacción que entre ellos existe y sobre todo la que tiene relación con el problema en cuestión, analizando primero las piezas aisladas y luego entrelazando sus vínculos.

Se debe tomar en cuenta primero las características que competen a los artículos y sobre estas las que conciernen al problema en cuestión o a los posibles problemas futuros que pudieran ser relacionados a los datos ahora analizados.

Viene a la mente un artículo: el número, el nombre, el precio, la cantidad en inventario, el proveedor, entre otros; asimismo hay que tener cuidado con las conductas que reportan estos artículos, sobre todo las que afectan al problema en cuestión, por ejemplo un artículo que se da de alta en el inventario, también se vende, puede ser que se regrese por el cliente o la empresa, si es que está en malas condiciones, entre otras. De manera que se empiezan a definir las variables de instancia que contendrá cada uno de los objetos de todas las clases posibles que se están analizando, comenzándose a determinar los métodos que hacen uso de esas variables de instancia y que reflejan la problemática analizada.

Una vez que se tienen todas las clases definidas con sus características o atributos (variables) y sus conductas (métodos), es cuando se inicia la aplicación que hará uso de estas clases, esto puede ser a través de una aplicación no gráfica, gráfica o un applet.

A continuación se presentan una serie de pasos necesarios para crear un modelo de objetos a partir de la especificación de un problema a resolver:

Pasos para crear un modelo

Identificar clases de objetos a partir de la especificación del problema

Una manera sencilla de hacerlo es empezar por identificar las clases al analizar la descripción textual en la especificación del problema. En este análisis textual, los sustantivos y las frases de sustantivos ayudan a entender con frecuencia los objetos y sus clases. Los sustantivos en singular (cuenta, carro y alumno) y los sustantivos en plural (alumnos, carros y cuentas) indican clases. Los sustantivos propios (la cuenta de Juan) y los sustantivos de referencia directa (el individuo que tenía la cuenta) indican objetos.

Identificar relaciones entre clases

Aquí se sugiere que se cree una tabla n x n, donde n es el número de clases. Se deben nombrar las filas y las columnas con los nombres de las clases y después se debe escribir en cada celda de intersección, una relación de asociación, donde se defina una especialización, si es que la clase del renglón es aún más particular que la clase de la columna, o asimismo se escribe una generalización si la clase de la columna es una especialización aún más específica que la clase del renglón (ver ejemplo de tabla de matriz de relaciones). Si hay alguna relación entre las clases se escribe.

Identificar los atributos de cada clase

Aquí se enumeran los datos propios de cada una de las clases mencionadas.

Identificar los métodos de cada clase

Aquí se deben de reconocer las acciones que deben ser realizadas por los objetos de la clase, ya sean acciones que no tengan relación con otras clases o acciones en donde se involucren otras clases, y así modificar métodos en ambas clases.

Modelar el problema utilizando UML (Unified Modeling Language)

La modelación del problema se hace después de haber hecho estas asociaciones entre clases.

A continuación se presenta una manera en la que se establece una relación específica del enunciado de "un cliente es una persona que tiene una o más cuentas".

Matriz de relaciones

	Persona	Cliente	Cuenta
Persona	No	Generalización	No
Cliente	Especialización	No	Posee alguna cuenta
Cuenta	No	X	No

Tema 2. Definición de una clase

La definición de una clase se entiende mejor al utilizar el diagrama de clase, que es la notación que se usa en UML, así como se muestra a continuación, una clase está representada por un rectángulo con tres secciones; la primera sección es donde se define el nombre de la clase, la segunda es donde se describen los atributos de la clase y la tercera es para los métodos de la clase.

Nombre de la clase
- Atributo1: Tipo - Atributo2: Tipo
+ método1 (parámetro 1: Tipo, parámetro 2: Tipo): Tipo + método2 (parámetro 1: Tipo, parámetro 2: Tipo): Tipo

Esto ayuda a entender visualmente la manera en la que se compone una clase.

El tipo de una variable y el tipo de retorno de un método están especificados por dos puntos (:), seguidos por el nombre del tipo. La especificación de un atributo o método comienza con un símbolo, el cual indica la vista de dicho atributo o método:

- - indica vista privada.
- + indica vista pública.

La vista privada no permite que otras clases tengan acceso a este atributo o método, mientras que la vista pública si.

Normalmente los atributos deben definirse como privados para que de esta manera se utilice el concepto de encapsulamiento en el que el objeto mantiene el control de la información que maneja. Los métodos se definen comúnmente como públicos para que cualquiera pueda hacer uso de ellos, a menos que sean métodos privados definidos para algún cálculo propio de la clase donde se va a definir.

Los atributos privados son entonces utilizados (accedidos o modificados) a través de métodos públicos; el método que sirve para tomar algún valor de un atributo privado es conocido como de acceso (accesor) y el que sirve para cambiar un valor del atributo privado es llamado modificador (mutator).

A continuación se muestra la representación de UML para la clase alumno en la que hay tres atributos privados: matricula, nombre y saldo, así como los métodos públicos: método de acceso para matricula, nombre y saldo, métodos modificadores para matrícula, nombre y saldo. Se define también un método privado para hacer el cálculo del saldo y es privado porque no se requiere usarlo fuera de la clase.

Alumno
- Nombre: String - Matricula: int - Saldo: double
+ getNombre(): String +setNombre(nombre : String) +getMatricula() : int +setMatricula:(matricula : int) +getSaldo() : doublé +setSaldo(saldo : double) -calculaSaldo()

Tema 3. Clases de envoltura (wrapper classes)

Existen las clases de envoltura o wrapper classes, las cuales son utilizadas en Java para manejar los datos primitivos pero a través de objetos, es decir en la mayoría de las clases de Java se tienen métodos que utilizan objetos.

Son pocos los métodos o métodos

constructores que toman como parámetros los datos primitivos (int, float, double, char, entre otros), por esto es que en Java se han definido estas clases de envoltura.

Hay una clase de envoltura para cada diferente tipo de dato primitivo, dentro de estas se tienen una serie de constructores utilizados para crear objetos de clase a partir de diversos valores primitivos.

Hasta ahora se ha utilizado mucho el método parseInt y parseDouble de las clases Integer y Double correspondientemente. Es importante ahora analizar los constructores que se tienen en cada una de estas clases, así como de las demás clases de envoltura para usarlas eficientemente de acuerdo a las necesidades.

Estas clases son:

Tipo de dato primitivo	Clase de envoltura (wrapper class)
Byte	java.lang.Byte
Short	java.lang.Short
Integer	java.lang.Integer
Char	java.lang.Char
Float	java.lang.Float
Double	java.lang.Double
Boolean	java.lang.Boolean

Tema 4. Variables de instancia y de clase

Cuando se define una clase en POO, las variables que se definen dentro de ella van a ser de diferentes valores en cada una de las instancias que se generarán al crear objetos nuevos de la clase.

Una variable de instancia normalmente se determina como privada para que no se pueda modificar desde otra clase, sólo a través de los métodos (como se vio en la clase Punto con el obtenX() y el cambiaX()); pero puede ser definida como pública y en ese caso no se requiere hacer uso de algún método para tomar su valor o modificarla (no es recomendable por el control de la variable de la misma clase).

Una variable de clase es aquella que solamente tiene un valor para toda la clase, ésta debe ser definida como static (estática) para que no se cree un nuevo valor con cada instancia.

La palabra static sirve para definir algo que no tiene que ver con las instancias de la clase, sino con toda la clase.

Cuando alguna variable o algún método se le antepone la palabra static, esto define que será única (variable) o único (método) para toda la clase.

Las variables de instancia se utilizan a través de los métodos de clase y son empleados por los objetos, es decir si para usar la cuenta bancaria con el constructor vacío Cuenta() y el Constructor(double saldo) con parámetro, es necesario definir algunos objetos como los siguientes:

```
Cuenta Juan, Pedro, Luis;
Juan = new Cuenta();
Pedro = new Cuenta(1500.0);
Luis = new Cuenta(3000.0);
```

Al usar la palabra new se crea un nuevo objeto de la clase y con esto se utiliza una nueva plantilla de variables de instancia para el objeto creado, Juan, Pedro y Luis son objetos nuevos de la clase cuenta y por cada variable que se tiene definida cuenta en la clase. Cada uno de estos objetos podrá tener un valor diferente.

Nota: es importante que después de declarar un objeto para una clase (Cuenta objeto;) se haga la creación del objeto, es decir utilizar la instrucción new Cuenta() para ese objeto (objeto = new Cuenta();) de otra manera el objeto no ha sido creado y no se pueden emplear los métodos, pues Java marcará el error NullPointerException.

En el ejemplo anterior si se tiene el método setSaldo(double saldo) para modificar el saldo de la cuenta y el método getSaldo() para acceder al saldo de la cuenta, las siguientes podrían ser instrucciones válidas:

```
luis.setSaldo(2200.0);
juan.setSaldo(350.50);
System.out.println("Saldo de pedro = " + pedro.getSaldo());
```

La manera de usar un método por un objeto es utilizando el formato objeto.método(parámetros);

El siguiente es un ejemplo que muestra el uso de las variables de instancia y de variables estáticas. Suponer que se tiene en la clase punto una variable de clase llamada puntos, la cual servirá para saber el número de objetos que se han creado en la clase. La clase punto a usar sería:

```
import java.awt.*;
public class Punto {
  private int x; // variable para la coordenada en x
  private int y; // variable para la coordenada en y
  private static int puntos=0;

  public Punto() { // método para construir un objeto sin parámetros
    x = 0;
    y = 0;
    puntos++;
  }
      // método para construir un objeto con valores enteros
              public Punto(int x, int y) {
                this.x = x;
          this.y = y;
          puntos++;
        }

        // método para construir un objeto con valores double
              public Punto(double x, double y) {
                this.x = (int) x;
          this.y = (int) y;
          puntos++;
        }

        // método para construir un objeto con otro objeto
              public Punto(Punto obj) {
                this.x = obj.obtenX();
          this.y = obj.obtenY();
          puntos++;
        }
  }
```

```
  // método que da el valor de la coordenada x
  public int obtenX() {
     return x;
  }

  // método que da el valor de la coordenada y
  public int obtenY() {
     return y;
  }
  // método que sirve para cambiar el valor //de la coordenada x
  public void cambiaX(int x) {
     this.x = x;  // this se usa porque se está utilizando (x)
  // como parámetro y como
   // variable de instancia y esto es para que no se confunda
   }
          // método que sirve para cambiar el valor de la coordenada y
public void cambiaY(int y) {
this.y = y; // this se usa porque se está utilizando (y)
  //como parámetro y como
    // variable de instancia y esto es para que no se confunda Java
   }

   // dibujar un punto
   public void dibujaPunto(Graphics g) {
   g.fillOval(x,y, 5, 5);
   }

   // para obtener un objeto Punto en formato String
   public String toString() {
   return "(" + obtenX() + "," + obtenY() + ")";
   }

   // para acceder a la cantidad de objetos creados
   public static int getPuntos() {
  return puntos;
   }

}
```

Note que la variable puntos y el método getPuntos() fueron definidos como estáticos, esto implica que será un solo valor para toda la clase, más no un valor

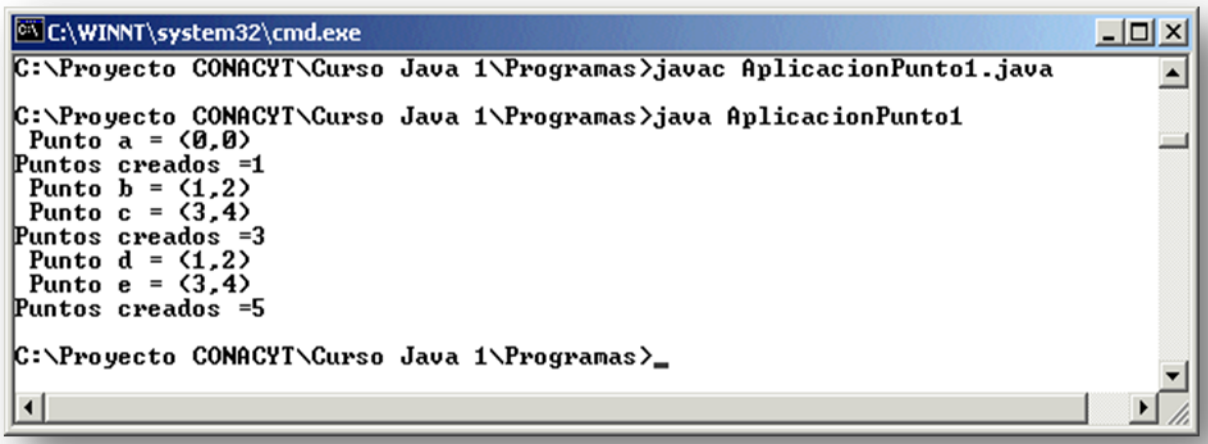
diferente por instancia (objeto) creada de la clase. También observe como la variable puntos es incrementada en uno cada vez que se crea un objeto de la clase (en cualquier constructor).

Esta clase se puede utilizar con la aplicación:

```
public class AplicacionPunto1 {
  private static Punto a, b, c, d, e;

  public static void main(String[] args) {
  a = new Punto();
  System.out.println(" Punto a = " + a.toString());
  System.out.println("Puntos creados =" + Punto.getPuntos());
  b = new Punto(1, 2);
  c = new Punto(3.0, 4.0);
  System.out.println(" Punto b = " + b.toString());
  System.out.println(" Punto c = " + c.toString());
  System.out.println("Puntos creados =" + Punto.getPuntos());
  d = new Punto(b);
  e = new Punto(c);
  System.out.println(" Punto d = " + d.toString());
  System.out.println(" Punto e = " + e.toString());
  System.out.println("Puntos creados =" + Punto.getPuntos());
  }
}
```

El resultado se ve en la siguiente imagen:

Variables de instancia y variables estáticas



Fuente: SENA

Visualice como después de crear el elemento a y desplegar el valor de puntos creados, sólo despliega 1, después al crear b y c, se tienen 3 y al final ya se tienen los 5.

También es importante hacer notar que el método getPuntos() fue utilizado a través de la clase punto, ya que es un método estático y debe ser usado a través de la clase.

Lo anterior no es un requisito, sin embargo es lo más común, ya que se puede tener una llamada a un método estático, pero a través de un objeto de la clase, funcionando exactamente igual; vea ahora una aplicación ligeramente cambiada, pero con el mismo resultado previo:

```
public class AplicacionPunto1 {
  private static Punto a, b, c, d, e;
  public static void main(String[] args) {
  a = new Punto();
  System.out.println(" Punto a = " + a.toString());
  System.out.println("Puntos creados =" + a.getPuntos());
  b = new Punto(1, 2);
  c = new Punto(3.0, 4.0);
  System.out.println(" Punto b = " + b.toString());
```

```
  System.out.println(" Punto c = " + c.toString());
  System.out.println("Puntos creados =" + b.getPuntos());
  d = new Punto(b);
  e = new Punto(c);
  System.out.println(" Punto d = " + d.toString());
  System.out.println(" Punto e = " + e.toString());
  System.out.println("Puntos creados =" + e.getPuntos());
  }
}
```

En esta aplicación se puede observar cómo es que el método estático getPuntos() también puede ser usado a través de un objeto de la clase, igualmente es importante recordar que es una sola variable para toda la clase.

Recolección de basura

En Java cada vez que se utiliza la palabra new se crea una nueva referencia a la memoria y más tarde cuando ya no se use, no se necesita borrarla (como se haría normalmente en C++), ya que Java automáticamente los borra debido a una clase llamada GarbageCollector, la cual se encarga de estar revisando los objetos que ya son obsoletos y que no se vuelven a utilizar.

La recolección de memoria es una facilidad que permite a Java reutilizar la memoria que tenía bloqueada para el uso de algún dato primitivo (int, double, entre otras) o de algún objeto. Gracias a esta facilidad de Java, el programador se olvida de tener que estar liberando la memoria y se enfoca más hacia una mejor utilización de las variables y objetos de la clase.

Herencia

Al definir una clase se precisan las características o variables, los métodos y conductas que pueden poseer todos los objetos que van a ser creados a partir de la clase. Sin embargo existen clases que requieren de una definición más especializada para poder establecer atributos o variables, así como conductas o métodos específicos de los objetos de esas clases más especializadas.

Un ejemplo puede ser la clase persona, esta clase puede tener solamente el nombre de la persona y posiblemente el sexo con sus respectivos constructores y métodos de acceso y modificación de las variables de instancia. La siguiente puede ser una definición de esta clase:

```
public class Persona {
  private String nombre;
  private char sexo;

  public Persona() {
   nombre = new String();
   sexo = ' ';
  }

  public String obtenNombre() {
   return nombre;
  }

  public void cambiaNombre(String nombre) {
   this.nombre = nombre;
  }

  public char obtenSexo() {
   return sexo;
  }

  public void cambiaSexo(char sexo) {
   this.sexo = sexo;
  }
   public String toString() {
  return "" + nombre + " " + sexo;
  }
}
```

Suponer que se quiere definir la clase alumno en la cual obviamente hay que definir el nombre y el sexo del alumno. Cuando se realiza esto no se debe empezar desde cero, se puede decir que la clase alumno sería una clase heredada de la clase persona, solamente que la clase alumno ahora añadirá ciertos atributos que van a ser sólo específicos de la clase alumno y a su vez, se van a anexar los métodos específicos de la clase alumno, ejemplo:

```
public class Alumno extends Persona {
  private int matricula;

  public Alumno() {
   super();
```

```
   matricula = 0;
  }

  public int obtenMatricula() {
   return matricula;
  }

  public void cambiaMatricula(int matricula) {
   this.matricula = matricula;
  }

  public String toString() {
   return "" + super.toString() + " " + matricula;
  }
}
```

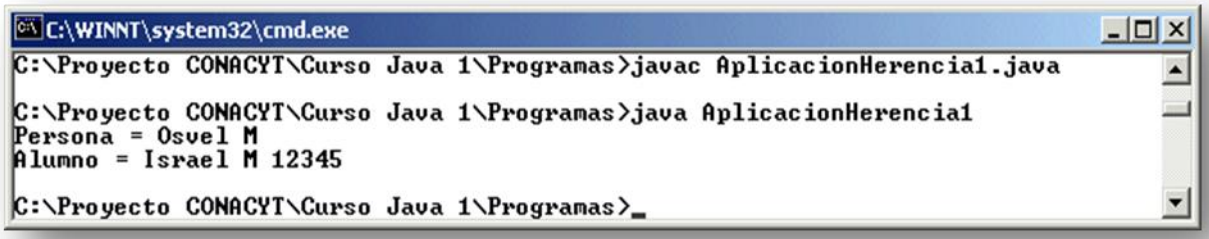
Se observa en esta clase hija (clase que hereda) que en el constructor se usa el método súper(), pero que en ninguna parte está definido este método; la palabra súper() es utilizada para llamar al constructor vacío de la clase padre (clase de la cual se está heredando).

La palabra extends admite dentro de Java definir la herencia, entonces cuando se dice public class Alumno extends Persona, se está diciendo que la clase alumno hereda de la clase persona. Una aplicación que utiliza estas clases sería la siguiente:

```
import java.io.*;

public class AplicacionHerencia1 {

  public static void main(String[] args) {
    Persona osvel = new Persona();
    osvel.cambiaNombre("Osvel");
    osvel.cambiaSexo('M');
    System.out.println("Persona = " + osvel.toString());
    Alumno israel = new Alumno();
    israel.cambiaNombre("Israel");
    israel.cambiaSexo('M');
    israel.cambiaMatricula(12345);
    System.out.println("Alumno = " + israel.toString());
  }
```

```
}
```

Ver los resultados de la ejecución:



Fuente: SENA

De esta manera la clase alumno hereda de la clase persona, y todos los atributos y métodos de la persona están siendo utilizados por el alumno, ya que automáticamente son suyos.

Cuando se emplean clases y la herencia, es entonces que se pone la palabra protected en lugar de private. Protected es utilizada para definir que la variable o método donde es usada será privada para cualquier clase externa, pero que sin embargo será empleada como si fuera pública para cualquier clase hija.

La mejor manera de utilizar las clases anteriores es:

Clase Persona:

```
public class Persona {
  protected String nombre;
  protected char sexo;

  public Persona() {
   nombre = new String();
   sexo = ' ';
  }

  public String obtenNombre() {
   return nombre;
  }
```

```
  public void cambiaNombre(String nombre) {
   this.nombre = nombre;
  }

  public char obtenSexo() {
   return sexo;
  }

  public void cambiaSexo(char sexo) {
   this.sexo = sexo;
  }

  public String toString() {
  return "" + nombre + " " + sexo;
  }
}
```

Clase Alumno:

```
public class Alumno extends Persona {
  protected int matricula;

  public Alumno() {
   super();
   matricula = 0;
  }

  public int obtenMatricula() {
   return matricula;
  }

  public void cambiaMatricula(int matricula) {
   this.matricula = matricula;
  }

  public String toString() {
   return "" + super.toString() + " " + matricula;
  }
}
```

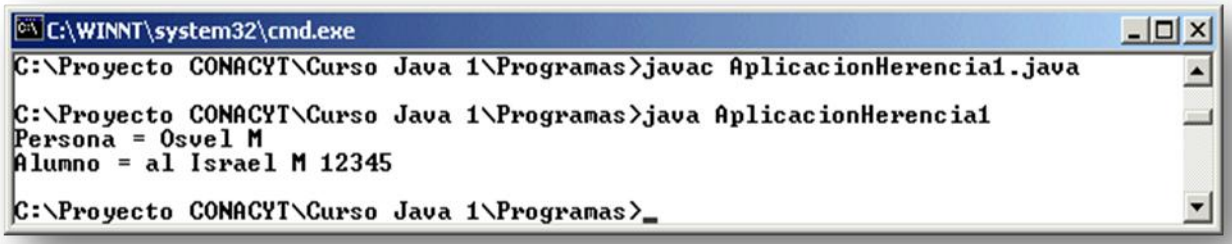
Tema 5. Sobreescritura de métodos

Una clase puede definir la manera en la que se hará un específico comportamiento para todos los objetos de la clase que la componen. Cuando una clase hija quiere cambiar el comportamiento de la manera en la que está definida en la clase padre, lo puede hacer solamente volviendo a definir el método (comportamiento) dentro de la clase que hereda, a este concepto se le denomina sobreescritura de un método.

En el ejemplo anterior suponer que en la clase alumno se desea modificar la manera en la que se cambia el nombre, de forma que al modificar el nombre del alumno se le antepongan las letras "al ", entonces la clase alumno quedaría de la siguiente manera:

```
public class Alumno extends Persona {
  protected int matricula;

  public Alumno() {
   super();
   matricula = 0;
  }

  public int obtenMatricula() {
   return matricula;
  }

  public void cambiaMatricula(int matricula) {
   this.matricula = matricula;
  }

  public void cambiaNombre(String nombre) {
  this.nombre = "al " + nombre;
  }

  public String toString() {
   return "" + super.toString() + " " + matricula;
  }
}
```

Ver el resultado:



Fuente: SENA

Polimorfismo

Se le llama polimorfismo al tener la llamada a un método por diferentes objetos, esto significa que si por ejemplo se tiene el método crecer() para una clase animal, también tendrá el método crecer() para una clase persona, lo mismo para una clase bacteria, esto quiere decir que la misma conducta puede ser desarrollada por diferentes objetos de diferentes clases. Java sabe que método tomar de acuerdo a la declaración del objeto de la clase que está haciendo la llamada al método.

El polimorfismo también actúa por tener un objeto definido como un tipo de clase y creado como un objeto de la clase hija. Por ejemplo, suponer que se tiene la clase animal y algunas clases hijas que heredan de esta clase como: vaca, cerdo y serpiente con la siguiente definición:

Clase animal

```
public class Animal {
  private int peso = 0;

  public void cambiaPeso(int peso) {
   this.peso = peso;
  }

  public int obtenPeso() {
   return peso;
  }

  public String habla() {
   return "Los animales no hablan";
```

```
  }
}
```

Clase vaca

```
public class Vaca extends Animal {
  public String habla() {
   return "MUU";
  }
}
```

Clase cerdo

```
public class Cerdo extends Animal {
  public String habla() {
   return "OINC";
  }
}
```

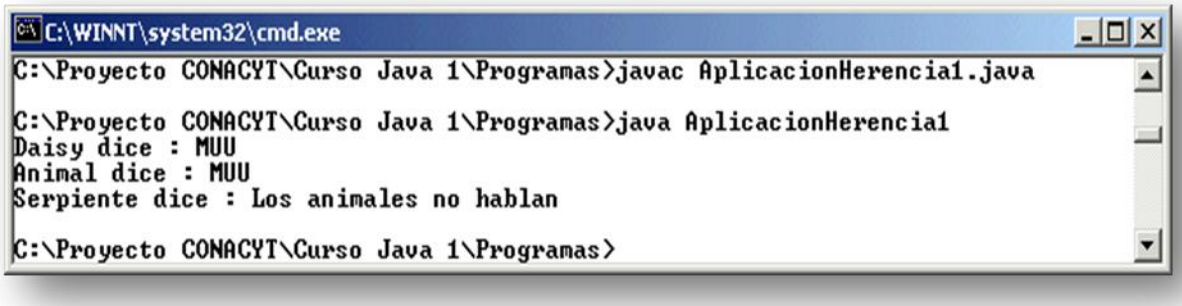
Clase serpiente

```
public class Serpiente extends Animal {
}
```

Aplicación herencia con polimorfismo

```
import java.io.*;

public class AplicacionHerencia1 {
  public static void main(String[] args) {
   Vaca daisy = new Vaca();
   Animal animal = new Vaca();
   Serpiente serpiente = new Serpiente();
   System.out.println("Daisy dice : " + daisy.habla() );
   System.out.println("Animal dice : " + animal.habla() );
   System.out.println("Serpiente dice : " + serpiente.habla() );
  }
}
```

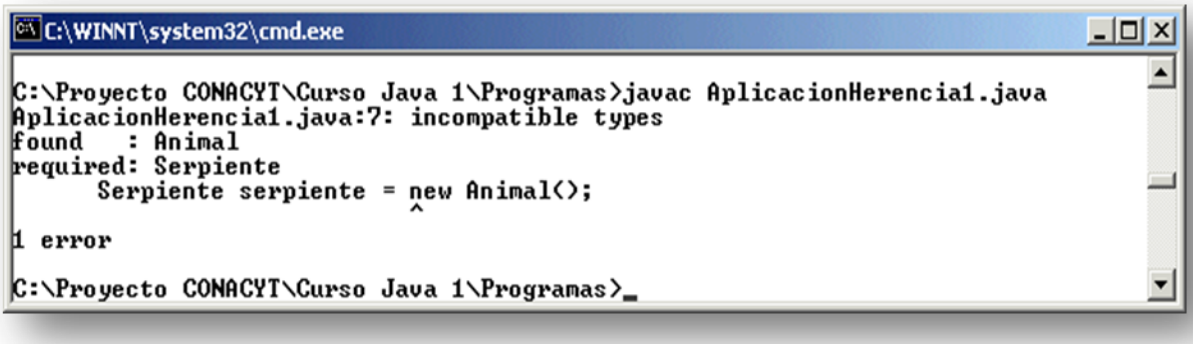
Vea el resultado en la siguiente imagen:



Fuente: SENA

En esta aplicación se observa como vaca pertenece a la clase vaca y el método habla() dice MUU como correspondería a una vaca, luego el animal es un objeto de la clase animal (clase padre) y al ser creado como vaca y llamar al método habla() se ve cómo aunque la clase animal tiene en el método habla() el decir "los animales no hablan", dice "MUU" porque fue creado como un objeto de la clase vaca. Se ve también como el objeto serpiente, el cual es creado como objeto de la clase serpiente al usar el método habla() dice "los animales no hablan", ya que no reescribe el método habla() y así es como está definido en la clase animal.

Al utilizar la herencia se debe tener cuidado cuando se crean los objetos con diferentes clases que definen el objeto, por ejemplo si se define el objeto serpiente como tipo serpiente, pero se crea con el constructor de Animal() dará el siguiente error:



Fuente: SENA

Entonces se debe entender que un objeto padre puede ser creado con un constructor del hijo, pero que no se puede hacer lo contrario.

Clases abstractas

Existen algunas clases que no pueden definir todos los detalles de los comportamientos de los objetos que conforman la clase, pero si pueden saber que existe el comportamiento, entonces estas clases se definen como abstractas, ya que los métodos que definen esas conductas serán definidos como abstractos con la palabra abstract.

Una aplicación no puede crear instancias (objetos) de una clase que está definida como abstracta. Cuando se tiene una clase abstracta se espera que exista al menos una clase que herede de ésta y que defina todos los métodos necesarios para la clase. Por ejemplo la clase figura, la cual determina los métodos perímetro y área como abstractos, entonces puede tener las clases cuadrado y rectángulo que hereden de esta clase figura, pero que definan ambos métodos. Observe las clases implicadas en este ejemplo:

Clase figura

```
public abstract class Figura{
  protected int x, y, ancho, alto;

  public void cambiaX(int x) {
  this.x = x;
  }

  public void cambiaY(int y) {
  this.y = y;
  }

  public void cambiaAncho(int ancho) {
  this.ancho = ancho;
  }

  public void cambiaAlto(int alto) {
  this.alto = alto;
  }
  public abstract float obtenPerimetro();

  public abstract float obtenArea();
}
```

Clase cuadrado

```
public class Cuadrado extends Figura {

  public float obtenPerimetro() {
  return 4 * ancho;
  }

  public float obtenArea() {
  return ancho * ancho;
  }
}
```

Clase rectángulo

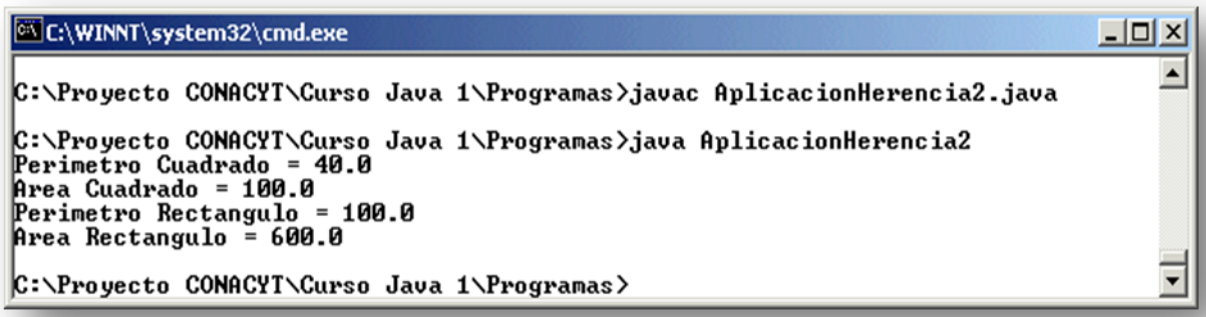
```
public class Rectangulo extends Figura {

  public float obtenPerimetro() {
  return 2 * ancho + 2 * alto;
  }

  public float obtenArea() {
  return ancho * alto;
  }
}
```

Aplicación herencia 2

```
public class AplicacionHerencia2 {
  public static void main(String[] args) {
   Cuadrado c = new Cuadrado();
   c.cambiaAncho(10);
   c.cambiaAlto(10);
   Rectangulo r = new Rectangulo();
   r.cambiaAncho(20);
   r.cambiaAlto(30);
   System.out.println("Perimetro Cuadrado = " + c.obtenPerimetro());
   System.out.println("Area Cuadrado = " + c.obtenArea());
   System.out.println("Perimetro Rectangulo = " + r.obtenPerimetro());
   System.out.println("Area Rectangulo = " + r.obtenArea());
  }
```

```
}
```

La ejecución de esta aplicación muestra lo siguiente:



Fuente: SENA

De esta manera se observa como al definir ambos métodos en la clase particular, ésta quita la abstracción.

Referencias

- Bonaparte, U. (2012). *Proyectos UML, diagrama de clases y aplicaciones Java en NetBeans 6.9.1*. Consultado el 09 de mayo de 2014, en http://www.edutecne.utn.edu.ar/tutoriales/uml_JAVA.pdf

- Sánchez, M. (s.f.). *Gestión de eventos*. Consultado el 24 de abril 2014, en http://odelys2003.files.wordpress.com/2013/01/1-1-javagestioneventos-100310084758-phpapp01.pdf

- Servicio Nacional de Aprendizaje, SENA. (2010). *Desarrollo de aplicaciones con interfaz gráfica, manejo de eventos, clases y objetos: Java.* Colombia: Autor.

Control del documento

	Nombre	Cargo	Dependencia	Fecha
Autor	Jorge Hernán Moreno Tenjo	Instructor	Centro Metalmecánico. Regional Distrito Capital	Mayo de 2013
Adaptación	Ana María Mora Jaramillo	Guionista - Línea de producción	Centro Agroindustrial. Regional Quindío	Mayo de 2014
	Rachman Bustillo Martínez	Guionista - Línea de Producción	Centro Agroindustrial. Regional Quindío	Mayo de 2014