Sesión 4

Programación orientada a objetos (II)

Repasando...


```
public class Amigo {
    Amigo miAmigo;
    String nombre;

    Amigo(String nombre) {
        this.nombre = nombre;
        this.miAmigo = null;
    }

    boolean quieresSerMiAmigo() {
        return (miAmigo == null);
    }

    void decir(String mensaje) {
        System.out.println(nombre + " dice: " + mensaje);
    }

    void saludar(String saludo) {
       System.out.println(saludo);
    }
```

Repasando...


```
 void relacionar(Amigo miNuevoAmigo) {
      if (miAmigo == null) {
         //Preguntar si quiere ser mi amigo
         if (miNuevoAmigo.quieresSerMiAmigo()) {
            miAmigo = miNuevoAmigo;
            miAmigo.saludar(nombre + " dice: Hola, mucho
gusto," + miAmigo.nombre);
            decir("Estoy contento porque mi nuevo amigo es " +
miAmigo.nombre);
            miAmigo.tomaMiAmistad(this);
         } else {
            decir(miNuevoAmigo.nombre + " no quiso ser mi
amigo :(");
         }
      } else {
         decir("No puedo tener otro amigo a menos que termine
mi amistad con " + miAmigo.nombre);
      }
   }
```

Repasando...


```
    void romperAmistad() {
        if (miAmigo != null) {
            miAmigo.yaNoQuieroTuAmistad();
            miAmigo = null;
        } else {
            decir("No puedo terminar una amistad cuando no la
tengo");
        }
    }

    void tomaMiAmistad(Amigo miNuevoAmigo) {
        this.miAmigo = miNuevoAmigo;
        miNuevoAmigo.saludar(nombre + " dice: Igualmente, " +
miAmigo.nombre + ", mucho gusto");
    }
```

Repasando...

```
    void yaNoQuieroTuAmistad() {
        if (miAmigo != null) {
            decir(miAmigo.nombre + " ya no quiere nada
conmigo :~(");
            miAmigo = null;
        }
    }
} //Fin clase Amigo
```

Amigo.java

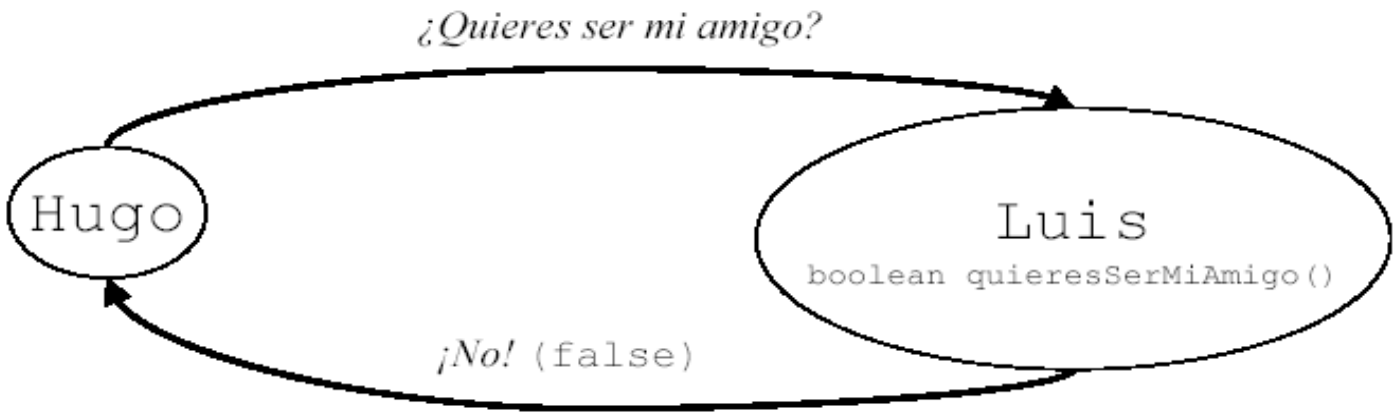
Repasando...


```
public class Amistad {

    public static void main(String[] args) {
        Amigo hugo = new Amigo("Hugo");
        Amigo paco = new Amigo("Paco");
        Amigo luis = new Amigo("Luis");

        hugo.relacionar(paco);
        luis.relacionar(hugo);

        paco.romperAmistad();

        luis.relacionar(hugo);
    }
}
```

Amistad.java

Mensajes

Del ejemplo anterior, se encuentra que entre los diferentes objetos existe interacción en forma de mensajes. Un *mensaje* entre dos objetos a y b es una llamada a un método de b invocada por a, o viceversa.



Más sobre constantes

Cuando un parámetro de una función se declara como constante, entonces su valor es inmodificable en el contexto de la función.

```
int funcionX(final double x) {
    x = x*x; //Error
    return (int) x;
}
```

La condición ?:

Una forma más simple de escribir una condición es mediante la expresión siguiente:

(condición)? sentencia1: sentencia2;

Si *condición* se cumple entonces se ejecuta *sentencia1*; en otro caso, *sentencia2*.

¿Cómo se reescribiría la función factorial?

```
long factorial(int n) {
    if (n <= 1)
        return 1;
    else
        return n*factorial(n-1);
    }
```

La condición ?:(...)


```
long factorial(int n) {

    return (n <= 1)? 1: n*factorial(n-1);
}
```

Ejercicio en equipos

Defina una clase `Factorial` con un miembro de clase `Factorial` y un método `longfactorial(n)`. Además, añada un método que modifique dicho miembro. En una segunda clase ejecutable `Factorial Distribuido`, en el método main construya 3 objetos de clase `Factorial` y enlázelos, uno tras otro formando un anillo. La implementación del método `factorial` tal que, si n<=1 devuelve 1; en otro caso, devuelve n*lo que devuelva el método `factorial`(n-1) del siguiente objeto enlazado.

Paquetes

Se llama **paquete** a una asociación de clases relacionadas por un nombre común. El propósito es lograr una mejor estructura y organización de las clases.

El API de Java posee una gran variedad de paquetes como `java.lang`, `java.awt`, `java.applet`, etc.

Para estos casos se dice que java es el *paquete raíz*, y los paquetes `lang`, `awt` y `applet` son *subpaquetes* del paquete Java.

Paquetes(...)

Ejemplo:

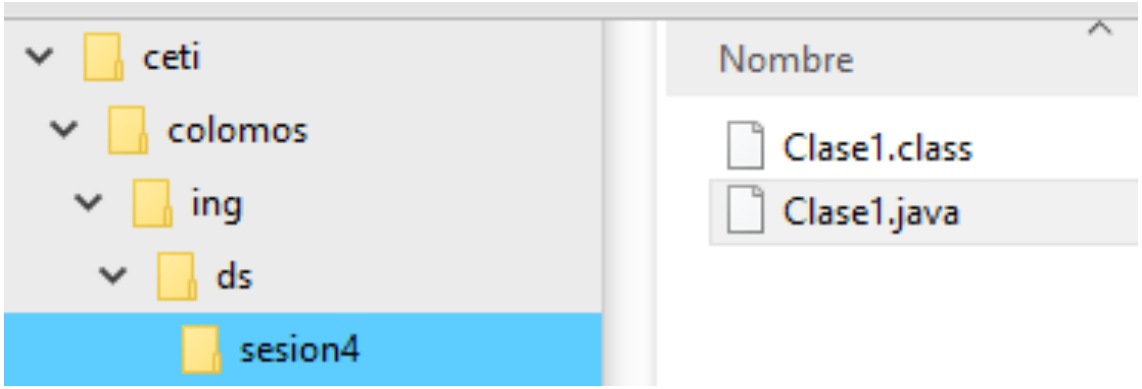
```
package ceti.colomos.ing.ds.session4;

public class Clase1 {
    …
}
```

El nombre correcto de Clase1 entonces es

ceti.colomos.ing.ds.session4.Clase1

Paquetes(...)

Al compilarse el archivo Clase1.java se creará una estructura de directorio de tal forma que se crea un directorio para el paquete raíz, y subdirectorios subsecuentes para cada subpaquete del paquete raíz recursivamente.



Paquetes(...)

Ahora suponga que `Clase1` cuenta con el método `main` definido de la siguiente forma:

```
public static void main(String[] args) {
  System.out.println("Esta es la clase 1");
}
```

¿Cómo ejecutar la clase?

1. Cambiar un directorio arriba del directorio raíz (el directorio correspondiente al paquete)
2. Ejecutar el intérprete de la siguiente manera:

```
java ceti.colomos.ing.ds.sesion4.Clase1
```

Importación de paquetes

Una de las potencialidades del paradigma orientado a objetos es el reuso de clases dentro de otras clases.

Esto se logra mediante la importación. Si una clase no está contenida en un paquete, por defacto, estará contenida en el paquete por default. Todas las clases se encuentran en el directorio ".", esto es, en el directorio actual. Para que una clase utilice objetos de esa clase, basta con nombrarla por su nombre, ya que ambos estarán contenidos bajo el mismo paquete por default.

Importación de paquetes(...)

Sin embargo, cuando una clase que está en un paquete diferente a otra clase que se desea utilizar, es necesario importar la clase por su nombre completo (*nombre paquete* + *nombre clase*) o importar todo el paquete. Esto se logra mediante la expresión:

```
import nombre_paquete.NombreClase;
```

O el paquete completo por medio de:

```
import nombre_paquete.*;
```

Importación de paquetes(...)

Suponga la clase `A` en el paquete `paquete1`, y la clase `B` en el paquete `paquete2`, y que `A` declara un campo que corresponde a un objeto de clase `B`

```
import paquete2.B;
public class A {
        B b;
...
}
```

Paquetes estándar

Existen algunos paquetes que son estándar al lenguaje Java y que vienen incluidos en el JDK. Estos son:

- `java.lang`, es el esqueleto del lenguaje y contiene básicamente funciones hacia la máquina virtual. Este paquete se importa implícitamente en toda clase.
- `java.io`, contiene clases para efectuar operaciones de escritura y lectura sobre archivos y flujos
- `java.awt`, contiene un conjunto mínimo para creación de aplicaciones GUI
- `javax.swing`, contiene clases más sofisticadas para creación de aplicaciones GUI con mayor sencillez

Paquetes estándar(...)

- `java.applet`, contiene las clases necesarias

  para la generación de aplicaciones empotradas en páginas HTML
- `java.util`, contiene utilerías para el cálculo de

  operaciones gráficas, manejo de estructuras de datos, números aleatorios, etc.
- `java.sql`, contiene las clases que interactúan con

  un controlador de base de datos.

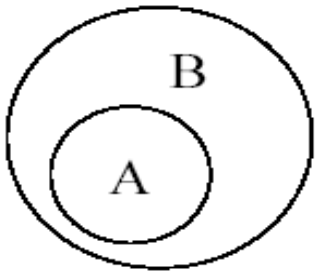
Convención de nomenclatura

La especificación de JDK sugiere la siguiente nomenclatura para los identificadores:

Identificador	Muestra
Paquetes	`udg.cucea.mti.pa`
Clases	`MiClase`
Métodos	`miMetodo`
Constantes	`CONSTANTE`

Herencia

La *herencia* es una relación jerárquica entre clases que permite la transmisión de propiedades de una clase a otra clase.




```
class A {
    int x;
    A(int x) {
      this.x = x;
    }
}
```

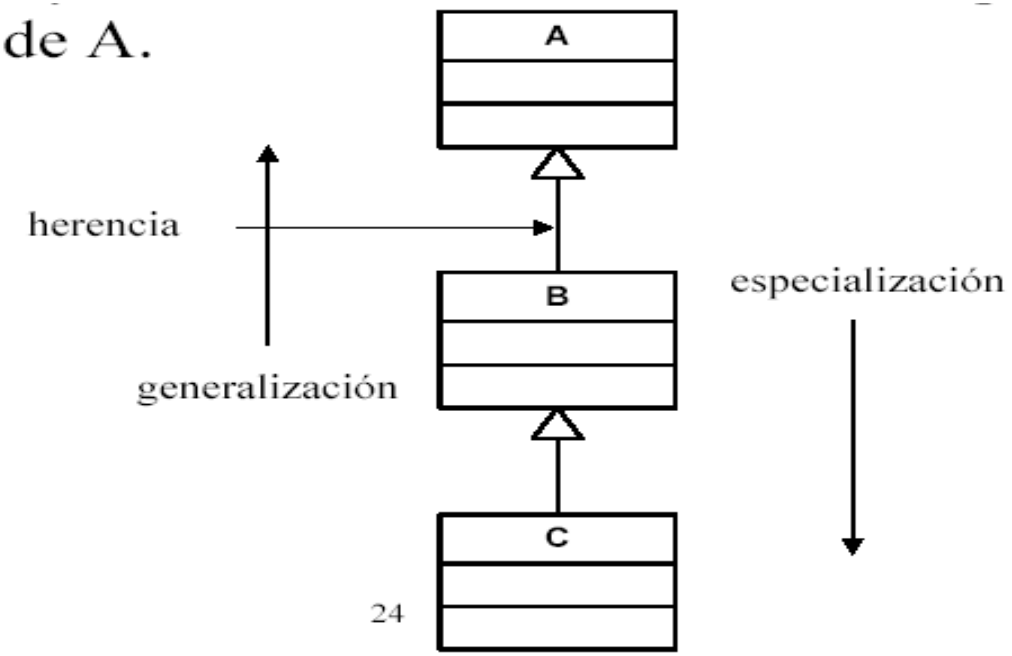
```
class B extends A {
    int y;
    B(int x, int y) {
      super(x);
      this.y = y;
    }
}
```

Herencia(...)

- En el ejemplo anterior, se dice que A es *superclase* de B, y que B es *subclase* de A.
- La única clase que no tiene superclase, es decir, que no hereda de ninguna otra es la clase `java.lang.Object`
- Todas las clases heredan de `java.lang.Object`.
- No existe en Java la herencia múltiple.

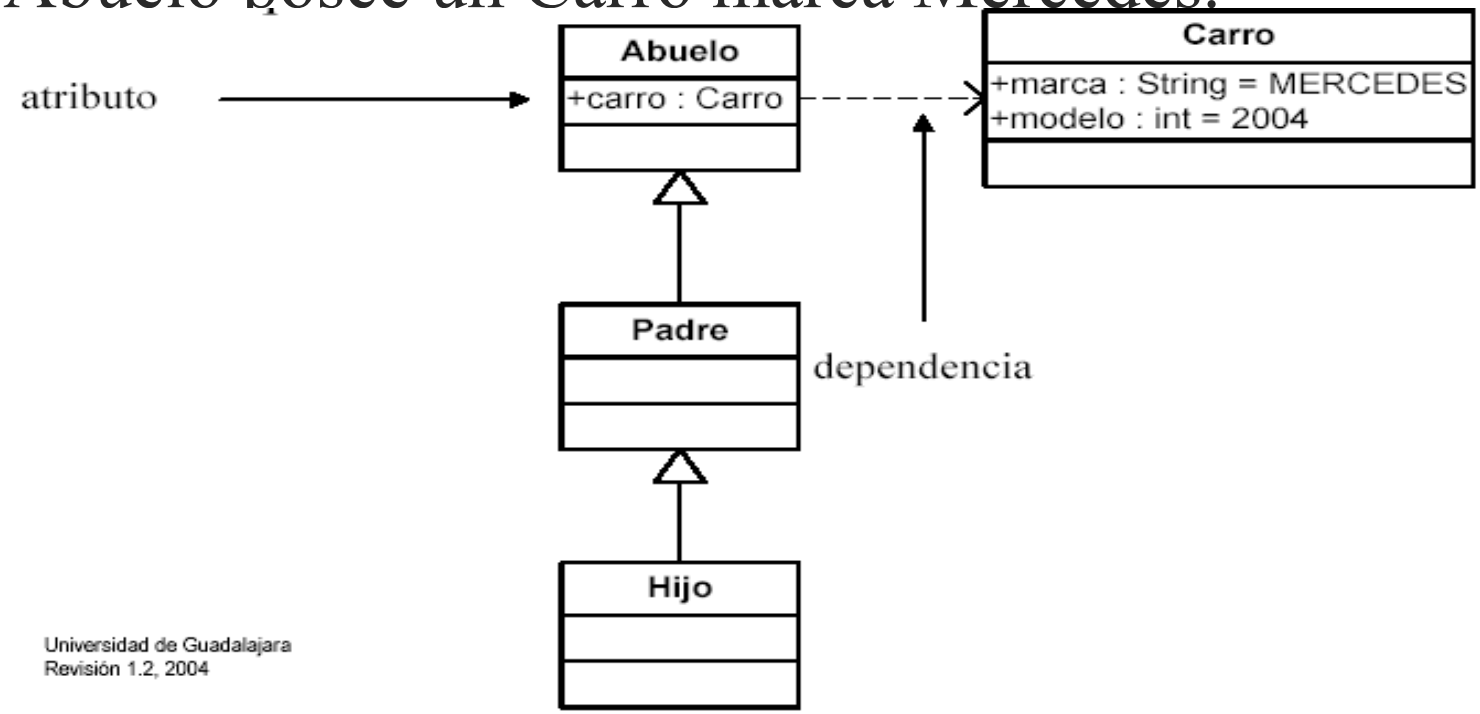
Herencia(...)

La relación de herencia es transitiva, esto es, si B es subclase de A y C es subclase de B entonces C posee los atributos de A.



Herencia(...)

Suponga la relación Abuelo, Padre e Hijo, donde el Abuelo posee un Carro marca Mercedes.



Universidad de Guadalajara
Revisión 1.2, 2004

Herencia(...)


```
public class Carro {
   String marca;
   int modelo;

   public Carro(String marca, int modelo) {
      this.marca = marca;
      this.modelo = modelo;
   }
}
```

Carro.java

Herencia(...)


```
public class Abuelo {
  Carro carro = new Carro("Mercedes", 2004);
}
```

Abuelo.java

```
public class Padre extends Abuelo {
}
```

Padre.java

```
public class Hijo extends Padre {
  public Hijo() {
    System.out.println("Wow! mi carro es un " +
carro.marca);
  }
}
```

Hijo.java

Acceso a miembros

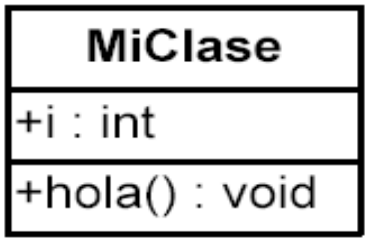
En la realidad, los objetos poseen atributos que pueden ser visibles y manipulables, y también atributos que son totalmente ocultos y de uso exclusivo por el objeto dueño.

De la misma manera, el paradigma orientado a objetos provee diferentes tipos de acceso a los atributos y métodos de una clase. Tales son:

- Público,
- Privado, y
- Protegido

Acceso a miembros(...)

Una variable, un método o una clase definida como `public` indica que puede ser accedida desde cualquier ámbito (ej. por otras clases, por otras máquinas virtuales).



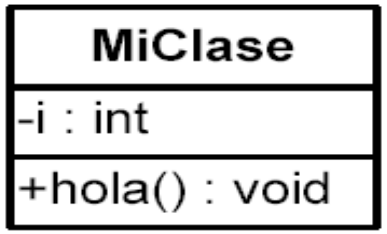

```
public class MiClase {
        public int i;
        public void hola() { ... }
}
```

Acceso a miembros(...)


```
class A {
   public int i;
}

public class B extends A {

  public static void main(String[] args) {
     B b = new B();
     b.i = 5;
     System.out.println(b.i);
  }
}
```

Acceso a miembros(...)

Con el modificador `private`, los miembros de una clase son solamente visibles dentro de la clase y nadie más.




```
public class MiClase {
      private int i;
      public void hola() { ... }
}
```

Acceso a miembros(...)

Ejercicio:

De la clase Abuelo, modifique el campo carro por privado. Compile nuevamente, ¿qué sucede?

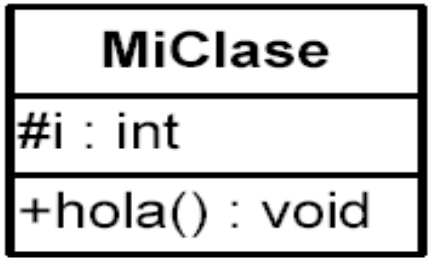
¿Quiénes pueden "utilizar" el carro?

```
public class Abuelo {

  private Carro carro = new Carro("Mercedes",
2004);

}
```

Acceso a miembros(...)

Otro tipo de acceso es la que se logra en la misma línea genealógica, es decir, entre superclases y subclases,conocido como acceso protegido (protected). Un atributo protegido solo es visible en las subclases, esto es, “todo queda entre familia”. Sin embargo, si una clase pertenece al mismo paquete pero no tiene ninguna relación genealógica, también puede acceder a los miembros protegidos. (¿Algún pariente lejano?)

```
public class MiClase {
        protected int i;
        public void hola() { ... }
}
```



Acceso a miembros(...)

Ejercicio:

Defina una cuarta clase `AmigoFamiliar` con un método `main`. Dentro del método declare y construya un objeto de tipo `Hijo`.

Trate de imprimir el campo `modelo` del campo `carro`.

¿Qué sucede?

Acceso a miembros(...)

Ejercicio:

Ahora defina una quinta clase `Colado` dentro de un paquete `otros` con un método `main`. Dentro del método declare y construya un objeto de tipo `Hijo`. Trate de imprimir el campo `modelo` del campo `carro`.

¿Qué sucede?

Acceso a miembros(...)

En resumen, el acceso a miembros se podría resumir así:

Tipo de acceso	*Visibilidad*
`public`	• Una clase pública es visible para toda clase desde cualquier paquete • Un miembro público es visible desde cualquier código interno o externo a la clase propietaria
`private`	• Una clase privada no es visible para ninguna clase. • Un miembro privado solo es visible en la clase propietaria. No son heredables.
`protected`	• Una clase protegida es visible por cualquier clase pero NO es instanciable. • Un miembro protegido es visible dentro de la clase propietaria y subclases de la misma, pero no es visible en código externo.

Sobreescritura

Hay situaciones donde las subclases necesitan especializar un código heredado. Por ejemplo, suponga un señor que tiene un puesto de tortas ahogadas y ha desarrollado un estilo de elaboración de las mismas. Suponga que este negocio se lo hereda a su hijo, y quizá desee modificar la receta para imprimir su propio estilo.

Este relación se le conoce como *sobreescritura*.

Sobreescritura(...)


```
public class TortasElGuero {
    protected int torta = 5;

    public int demeUnaTorta() {
        return torta;
    }
}

public class TortasElGueroJr extends TortasElGuero {
    protected int salsa = 2;

    public int demeUnaTorta() {
        return torta + salsa;
    }
}
```

Sobreescritura(...)


```
public class Profeco {

  public static void main(String[] args) {
      TortasElGuero elDon;
      TortasElGueroJr elHijo;
      ...
      1 System.out.println(elDon.demeUnaTorta());
      2 System.out.println(elHijo.demeUnaTorta());
  }
}
```

Tarea

Defina el diagrama, haga una clase que reciba como parámetro un polinomio de cualquier grado y construya una ecuación la cual se constituye de un vector de términos.