Escuela de Sistemas y Tecnologías

Transparencias del Curso **ANALISTA DE SISTEMAS**

Edición 2017 – Materia: Java Web

TEMA: REPASO DE POO CON JAVA

Consideraciones

- Estas transparencias **no** tienen el objetivo de suplir las clases.
- Por tanto, serán **complementadas** con ejemplos, códigos, profundizaciones y comentarios por parte del docente.
- El material aquí expuesto es (en su mayoría) un resumen del repartido teórico denominado "**OOP en Java**" por lo que se exhorta su lectura como complemento (indispensable) de este material.
- El **orden** de dictado de estos temas está sujeto a la consideración del docente.
- Las **discusiones** aquí propuestas están sujetas al tiempo disponible para las mismas a consideración del docente (la mayoría de ellas se encuentran también en "**OOP en Java**").

Agenda

- Introducción a la Orientación a Objetos
- Definiciones y Conceptos Básicos
- Creación de Objetos
- Atributos y Métodos de Clase
- Herencia
- Operaciones y Clases Abstractas
- Polimorfismo
- Casteos
- Interfaces
- Asociaciones entre Clases
- Inner Classes e Inner Classes Anónimas

Agenda

- **Introducción a la Orientación a Objetos**
- Definiciones y Conceptos Básicos
- Creación de Objetos
- Atributos y Métodos de Clase
- Herencia
- Operaciones y Clases Abstractas
- Polimorfismo
- Casteos
- Interfaces
- Asociaciones entre Clases
- Inner Classes e Inner Classes Anónimas

Intro. a OOP (1)

- **Orientación a Objetos**:
  - *"Es una técnica de programación, un paradigma."* [Bjarne Stroustrup, 1991, creador de C++]
  - La principal idea de este paradigma es diseñar una estructura de objetos junto con su comportamiento que permita a los objetos interactuar (a través de mensajes) para lograr un objetivo.
- **Lenguaje OO**:
  - *"Es un lenguaje que provee mecanismos que soportan el paradigma de la programación orientada a objetos."* [Ídem]

Intro. a OOP (2)

- **La OO como Metáfora**:
  - Una de las características más importantes de la OO es su poder para servir como metáfora, tanto de un problema del mundo real, como de su solución lógica, y finalmente de su implementación en código.
  - **Análisis**: los objetos se utilizan para representar el problema.
  - **Diseño**: los objetos se utilizan para representar la solución (lógica) de software.
  - **Implementación**: los objetos se utilizan para implementar la solución en un lenguaje de programación.

Intro. a OOP (3)

- **La OO como Metáfora (cont.)**:
  - Esto significa que se utiliza la OO no sólo para programar (codificar) clases y objetos, lo cual se denomina "*Implementación Orientada a Objetos*".
  - También se la puede utilizar para entender el problema de la realidad. Esto se denomina "*Análisis Orientado a Objetos*".
  - También se la puede utilizar para representar una solución lógica (en papel) al problema.
    Esto se denomina "*Diseño Orientado a Objetos*".

Intro. a OOP (4)

- **La OO como Metáfora (cont.)**:
  - Todo esto contribuye a reducir la diferencia entre el problema de la vida real y la solución de software y hardware de ese problema.
  - En inglés: "*bridge the gap*"
  - Este concepto (OO como Metáfora para Análisis, Diseño e Implementación) es absolutamente fundamental y básico, pero su verdadera comprensión ocurre luego de comenzar a estudiar/trabajar con la OOP.

Intro. a OOP (5)

- **La OO como Abstracción**:
  - Uno de los motivos por los que es válido utilizar la metáfora de la OO es su gran poder de abstracción.
  - Un solo objeto encapsula muchos datos y muchas funciones en una unidad indivisible.
  - **Antes**: los datos por un lado, las funciones por otro.
  - **Ahora**: todo junto (al menos lo relativo a "algo") en un solo concepto: el objeto.
  - Esto permite **elevar el nivel de abstracción** con el cual pensamos, elevando así el nivel de abstracción con el cual desarrollamos software.

Intro. a OOP (6)

- Estos conceptos, al igual que muchos de OOP, no son de fácil comprensión.
- Aún luego de comprendidos, debe incorporarlos como nuevo conocimiento para **comenzar a pensar en términos de objetos** ¡De eso se trata aprender OOP!
- Se recomienda volver a éstos luego de haber comprendido los elementos básicos de la OO.
- Sólo así se logrará tener un verdadero panorama general, sin perderse en los detalles de la Teoría de Objetos.

Intro. a OOP (7)

- **Evolución de OOP (<u>Resumen</u>)**:
  - **Orígenes**:
    - 1965-67: Simula I y Simula 67 (Alan Key introduce el término "*object-oriented programming*")
    - 1970: Smalltalk
  - **Popularización**:
    - Mediados 1980: C++
    - Mediados 1990: Java de Sun® Microsystems
    - Principios 2000: Plataforma .NET de Microsoft® con los lenguajes C# y Visual Basic .NET (entre otros).

Intro. a OOP (8)

- **Etapas y Herramientas (<u>Resumen</u>):**
  - **Análisis OO:**
    - Modelo Conceptual / Dominio (Diag. Clases UML)
    - Comportamiento del Sistema (Diag. Secuencia UML)
  - **Diseño OO:**
    - Diseño de Alto Nivel / Arquitectura (varios diag. UML)
    - Diseño de Interacciones (Diag. Comunic. UML)
    - Diseño de Clases (Diag. Clases UML)
  - **Implementación OO**
- **Este curso se centra en la Implementación OO con una breve presentación de UML**

Agenda

- Introducción a la Orientación a Objetos
- **Definiciones y Conceptos Básicos**
- Creación de Objetos
- Atributos y Métodos de Clase
- Herencia
- Operaciones y Clases Abstractas
- Polimorfismo
- Casteos
- Interfaces
- Asociaciones entre Clases
- Inner Classes e Inner Classes Anónimas

Definiciones y Conceptos Básicos (1)

- **Objeto**:
  - Un objeto es una entidad en memoria con identidad, estado y comportamiento.
  - Un objeto es una instancia de una clase.
- **Clase**:
  - Una clase describe un conjunto de objetos.
  - Sirve de "template" para la creación de objetos.
  - La clase es lo que se codifica en Java, los objetos son creados a partir de una clase.

Definiciones y Conceptos Básicos (2)

- **Ejemplo**:
  - Para permitir que un programa manipule objetos que representan alumnos, se codifica la clase Alumno.
  - Dicha clase Alumno definirá qué caracteriza a un alumno en términos de sus datos (ej: nombre, edad) y de lo que puede hacer (ej: rendirExamen(), cumplirAños()).
  - Gracias a que, en tiempo de diseño, el programador codificó la clase Alumno, en tiempo de ejecución existirán muchos objetos de tipo Alumno (instancias de la clase).

Definiciones y Conceptos Básicos (3)

- **Ejemplo** (cont.):

```
// Código de la clase Alumno
public class Alumno {
    public String nombre;
    public int edad;

    public void rendirExamen(int cod_Examen) {
        ...
    }

    public void cumplirAños() {
        edad = edad + 1;
    }
}
```

Definiciones y Conceptos Básicos (4)

- **Ejemplo** (cont.):

```
// Creando un objeto Alumno
Alumno a = new Alumno();

// Asignándole el nombre "Juan"
a.Nombre = "Juan";

// Juan rinde el examen 4011
a.rendirExamen(4011);

...
```

Definiciones y Conceptos Básicos (5)

- **Identidad**:
  - La identidad es una característica de todos los objetos.
  - Un objeto es capaz de diferenciarse de otro gracias a la identidad.
  - La identidad de un objeto está dada automáticamente.
  - Aunque dos objetos luzcan iguales (ej: contienen los mismos datos) la identidad ayudará a diferenciarlos.
  - Se volverá a este punto en breve…

Definiciones y Conceptos Básicos (6)

- **Atributo**:
  - Un atributo es una variable definida dentro de una clase.
  - Instancias de esa clase le asignarán un valor a cada atributo definido en la clase.
- **Ejemplo**:
  - En el ejemplo anterior, las variables `nombre` y `edad` son atributos de la clase Alumno.
  - El nombre de la instancia `a` es “Juan”, mientras que el nombre de otra instancia `b` podría ser “María”.

Definiciones y Conceptos Básicos (7)

- **Operación**:
  - Una operación es una especificación de una función que se le puede solicitar a un objeto que ejecute.
  - Una operación dice QUE se debe hacer al ejecutarse, pero no dice COMO hacerlo.
- **Ejemplo**:
  - En el ejemplo anterior,
    `public void rendirExamen(int cod_Examen)`
    y `public void cumplirAños()`
    son operaciones (sólo el encabezado / la firma).

Definiciones y Conceptos Básicos (8)

- **Método**:
  - Un método es la implementación de una operación.
  - Un método dice COMO hay que hacer para llevar adelante la ejecución de una operación.
- **Ejemplo**:
  - En el ejemplo anterior, el método para la operación `public void cumplirAños()` en la clase Alumno es:

```
{
    edad = edad + 1;
}
```

Definiciones y Conceptos Básicos (9)

- **Estado**:
  - El estado de un objeto es el conjunto de los valores de los atributos en un instante del tiempo.
  - El estado de un objeto puede cambiar a lo largo del tiempo, pero sigue siendo el mismo objeto (pues su identidad no cambia).
- **Ejemplo**:
  - En el ejemplo anterior, el estado del objeto `a` al recién crearlo es indefinido (pues no tiene nombre ni edad).
  - En la segunda sentencia, el estado del objeto `a` pasa a ser "Juan" (con la edad aún indefinida).

Definiciones y Conceptos Básicos (10)

- **Comportamiento**:
  - El comportamiento de un objeto es el conjunto de operaciones que el objeto puede ejecutar.
- **Ejemplo**:
  - En el ejemplo anterior, las operaciones `public void rendirExamen(int cod_Examen)` y `public void cumplirAños()` pueden ser ejecutadas sobre objetos de la clase Alumno.

Definiciones y Conceptos Básicos (11)

- **Estado y Comportamiento:**
  - El estado de un objeto dependerá entonces de los atributos definidos dentro de su clase (los que determinan qué cosas el objeto puede guardar) así como del valor de esos atributos en cada instante de tiempo (que determina qué dato se está guardando en cada atributo).
  - El comportamiento de un objeto dependerá de las operaciones definidas dentro de su clase, no de su implementación (método).

Definiciones y Conceptos Básicos (12)

- **Referencia:**
  - Una referencia es una variable cuyo tipo de datos es una clase.
  - Se utiliza para manipular objetos, pues una referencia apunta a un objeto (o a ninguno, en cuyo caso se dice "referencia nula").
- **Ejemplo**:
  - En el ejemplo anterior, `a` en realidad no es el objeto, sino una referencia al objeto (el cual está alojado en memoria).

Definiciones y Conceptos Básicos (13)

- **Referencia (cont.):**
  - Una referencia es similar al concepto de "puntero" en otros lenguajes de programación.
  - En Java (y en la mayoría de los lenguajes OO modernos) no se permite la llamada "aritmética de punteros".
  - Como se hizo en el ejemplo, es habitual abusar de la notación y decir "el objeto `a`" en vez de "el objeto referenciado por `a`".
  - **Cuidado**: conceptualmente es muy importante notar la diferencia entre objeto y referencia.

Definiciones y Conceptos Básicos (14)

- **Referencia (cont.):**
  - Volviendo sobre el concepto de identidad, dos objetos son idénticos si la comparación de sus referencias es `true`.
- **Ejemplo**:
  - Si `a==b` retorna `true`, entonces las referencias `a` y `b` apuntan al mismo objeto.
  - Por lo tanto, formalmente es un error el planteo inicial de "dos objetos son idénticos si…" pues en realidad se trata del mismo objeto apuntado por dos referencias.

Definiciones y Conceptos Básicos (15)

- **Acceso a Atributos y Operaciones:**
  - Para acceder a los atributos (públicos) y las operaciones (públicas) de un objeto, basta con utilizar el punto luego de una referencia al objeto seguido del nombre del atributo u operación que se desea acceder.
  - Esto muestra que se parte del objeto (en realidad de la referencia al objeto) para solicitarle algo (atributo u operación) lo cual es esencial en la orientación a objetos.
- **Ejemplo**:
  - En el ejemplo anterior: `a.rendirExamen(4011);`

Definiciones y Conceptos Básicos (16)

- **Modificadores de Acceso:**
  - Los modificadores de acceso permiten modificar desde donde se puede acceder a un atributo u operación.
  - Los más utilizados son:
    - `public` permite que el atributo/operación sea accedido desde cualquier porción de código (cualquier clase, incluso fuera del paquete).
    - `private` permite que el atributo/operación sea accesible solamente desde dentro de la propia clase.
    - `protected` permite que el atributo/operación sea accesible solamente desde dentro del paquete en el que se encuentra la clase.
- **Ejemplo**:
  - En el ejemplo anterior tanto los atributos como las operaciones fueron definidos como públicos.

Definiciones y Conceptos Básicos (17)

- **Modificadores de Acceso (cont.):**
  - Generalmente, se sugiere que los atributos sean privados, brindando formas de obtener el valor del atributo (*getter*) y/o de modificarlo (*setter*).
  - Las operaciones pueden ser públicas o privadas dependiendo de su objetivo:
    - Una operación interna a la propia clase que, por ejemplo, ayuda en algún cálculo complejo o es utilizada por muchas otras operaciones (factoriza cierto código) entonces será privada (utilizando el modificador de acceso `private`).
    - Las operaciones *getter* y *setter* deben ser públicas (`public`).

Definiciones y Conceptos Básicos (18)

- **Discusión** ¡¿?!
  - Dado que una referencia no es más que una variable, ¿Qué valor cree que guarda?
  - ¿Qué significa entonces la asignación de referencias?
  - ¿Qué efecto tiene? ¿Se copian los objetos apuntados?
  - Discutir cómo se asigna el valor nulo a una referencia y qué efecto tiene hacerlo.
  - ¿Un programador siempre querrá considerar como distintos a dos objetos que lucen iguales?

Agenda

- Introducción a la Orientación a Objetos
- Definiciones y Conceptos Básicos
- **Creación de Objetos**
- Atributos y Métodos de Clase
- Herencia
- Operaciones y Clases Abstractas
- Polimorfismo
- Casteos
- Interfaces
- Asociaciones entre Clases
- Inner Classes e Inner Classes Anónimas

Creación de Objetos (1)

- **Constructor:**
  - Un constructor es un método que es invocado al momento de crear una nueva instancia de una clase.
  - Su objetivo es inicializar el objeto que se está creando.
  - Siempre que se crea una instancia se llama a algún constructor (pueden haber varios constructores en la misma clase).
  - Por tanto toda clase siempre tendrá al menos un constructor (aún cuando el programador no lo defina).

Creación de Objetos (2)

- **Constructor (cont.):**
  - En el preciso momento que se crea una instancia se está invocando algún constructor.
  - El nombre del constructor debe ser el mismo que el nombre de la clase.
- **Ejemplo**:
  - En el ejemplo: `Alumno a = new Alumno();`
  - En este caso se está invocando al constructor sin pasarle ningún parámetro (denominado constructor por defecto).

Creación de Objetos (3)

- **Constructor (cont.):**
  - Una clase puede definir cualquier cantidad de constructores, todos con diferente firma.
  - Los constructores se diferencian del resto de los métodos por no definir un tipo de retorno (ni siquiera `void`).
  - Si el programador no define ningún constructor para una clase, entonces el lenguaje asumirá la existencia de un constructor por defecto (como en el caso del ejemplo).

Creación de Objetos (4)

- **Constructor (cont.):**
  - Los constructores se pueden clasificar como:
    - **Por defecto**: puede haber solo uno (si existe) que no recibe parámetros.
    - **Común**: pueden existir muchos con diferente firma.
    - **Completo**: puede haber sólo uno (si existe) que recibe todos los parámetros necesarios para una completa inicialización de la instancia que se está creando.
  - Un constructor puede invocar a otro mediante el uso de la palabra reservada `this` seguida de una lista de parámetros que son pasados (debe ser la primera línea de código dentro del constructor).

Creación de Objetos (5)

- **Declaración e Instanciación:**
  - La declaración (definición) de una referencia no es lo mismo que su posterior instanciación (creación de un objeto que es apuntado por la referencia).
  - Puede realizarse en una sola línea de código o en dos.
- **Ejemplo**:
  - ```
    Alumno a;                // Declaración
    a = new Alumno();  // Instanciación
    ```
  - ```
    Alumno a = new Alumno();  // Ambas
    ```

Creación de Objetos (6)

- **Discusión** ¡¿?!
  - Puede verse como que los constructores sí retornan algo (indispensable, por cierto): ¿Qué sería?
  - ¿Por qué puede ser útil definir muchos constructores?
  - ¿Por qué es útil que un constructor invoque a otro constructor dentro de la propia clase?
  - ¿Por qué si dentro de un constructor se invoca a otro constructor debe ser la primera línea de código?
  - ¿Qué significa que un constructor reciba como parámetro otra instancia? ¿Podría ser útil esto?

Agenda

- Introducción a la Orientación a Objetos
- Definiciones y Conceptos Básicos
- Creación de Objetos
- **Atributos y Métodos de Clase**
- Herencia
- Operaciones y Clases Abstractas
- Polimorfismo
- Casteos
- Interfaces
- Asociaciones entre Clases
- Inner Classes e Inner Classes Anónimas

Atributos y Métodos de Clase (1)

- **Atributo de Instancia:**
  - La definición dada de atributo se denomina formalmente "Atributo de Instancia" pues cada instancia guardará su propio valor del atributo.
  - Por tanto, en caso de haber 100 instancias, habrán 100 valores de ese atributo (el "dueño" del atributo es la instancia).
- **Atributo de Clase**:
  - Un atributo de clase es aquel cuyo valor es único y compartido entre todas las instancias de la clase (el "dueño" del atributo es la clase).

Atributos y Métodos de Clase (2)

- **Atributo de Clase (cont.):**
  - También son llamados "Atributos Estáticos" pues la mayoría de los lenguajes OO utiliza la palabra reservada *static* para definirlos.
- **Ejemplo**:
  - Siguiendo el ejemplo anterior, si todos los alumnos tienen el mismo color de pelo, entonces el atributo `colorPelo` puede ser definido como estático:

```
private static colorPelo;
```

Atributos y Métodos de Clase (3)

- **Cuidado:**
  - ¡El hecho que sean llamados estáticos no significa que su valor no pueda cambiar!
  - Las que no cambian su valor son las constantes, que no son lo mismo que los atributos estáticos.
  - Las constantes suelen definirse como estáticas pues todos los objetos compartirán el valor constante.
  - Por tanto:
    - Generalmente las constantes son atributos estáticos, pero
    - No todo atributo estático es una constante.

Atributos y Métodos de Clase (4)

- **Atributo de Clase (cont.):**
  - Una consecuencia de que los atributos estáticos pertenezcan a la clase y no a las instancias, es que su valor es independiente del hecho de que existan instancias o no de esa clase.
  - Por tanto un atributo estático puede alojar un valor aún cuando no exista ningún objeto de esa clase (cosa que no puede hacerse con los atributos de instancia).
  - Para acceder a un atributo estático (público) se utiliza también el punto pero sobre el nombre de la clase en lugar del nombre de la referencia.

Atributos y Métodos de Clase (5)

- **Método de Clase:**
  - Un método de clase es un método que pertenece a la clase y por tanto puede ser invocado sin necesidad de contar con una instancia.
  - Sólo permite manipular los atributos estáticos de la clase a la cual pertenece independientemente de la existencia de objetos.
  - Se utiliza la misma forma de acceso que para los atributos estáticos: `Clase.metodoEstatico()`
  - Para su definición también se utiliza la palabra reservada `static`

Atributos y Métodos de Clase (6)

- **Discusión** ¡¿?!
  - ¿Por qué desde un método estático (de clase) no se puede acceder a los atributos de instancia?
  - ¿Desde un método no estático (de instancia) se puede acceder a atributos y métodos estáticos? ¿Por qué?
  - ¿Para que puede ser útil definir y utilizar atributos y métodos estáticos?
  - ¿Cómo haría para contar la cantidad de veces que un método (de instancia) es invocado sobre un objeto?
  - ¿Y para para contar la cantidad de veces que ese método se invocó sobre todos los objetos de esa clase?

Agenda

- Introducción a la Orientación a Objetos
- Definiciones y Conceptos Básicos
- Creación de Objetos
- Atributos y Métodos de Clase
- **Herencia**
- Operaciones y Clases Abstractas
- Polimorfismo
- Casteos
- Interfaces
- Asociaciones entre Clases
- Inner Classes e Inner Classes Anónimas

Herencia (1)

- **Herencia:**
  - Es el mecanismo por el cual se permite a una clase heredar los atributos y métodos de otra clase.
  - Una clase solo puede heredar de una clase (herencia simple), no de varias (esto último, en los lenguajes que lo permiten, se denomina herencia múltiple).
  - La clase de la cual se hereda se denomina clase base o super clase; la clase que hereda se denomina subclase o clase derivada.

Herencia (2)

- **Herencia (cont.):**
  - Se heredan todos los atributos y métodos de la clase base (aunque lo marcado con **`private`** en la clase base no sea accesible desde la clase derivada).
- **Ejemplo**:
  - Para que la clase Alumno herede de la clase Persona (asumiendo que ya existe) la sintaxis es:

```
public class Alumno extends Persona {
   …
}
```

Herencia (3)

- **Subclase Directa e Indirecta:**
  - Si la clase `B` es subclase de `A` entonces "`B` es subclase directa de `A`".
  - Si además `C` es subclase de `B` entonces "`C` es subclase directa de `B` e indirecta de `A`".
- **Instancia Directa e Indirecta**:
  - Si una instancia `a` es el resultado de `new A()` se dice que "`a` es instancia directa de `A`".
  - Si la clase `B` es subclase de `A` y `b` es una instancia resultado de `new B()` se dice que "`b` es instancia directa de `B` e indirecta de `A`"

Herencia (4)

- **La clase `Object`:**
  - La clase `Object` es una clase ya predefinida en el lenguaje y es la clase base de todas las demás clases.
  - Esto significa que cualquier clase es subclase (directa o indirecta) de `Object`.
- **Reemplazabilidad**:
  - Si `B` es subclase de `A` y `b` es resultado de `new B()` entonces `b` es de tipo `B` pero también de tipo `A`.
  - Ej.: si `Auto` es subclase de `Vehiculo`, entonces un auto es un vehículo (aunque no todo vehículo será un auto).

Herencia (5)

- **¡Todo es un objeto!:**
  - Como consecuencia de la definición de `Object` y del Principio de Reemplazabilidad: cualquier objeto es de tipo `Object`.
  - Esto es una consecuencia muy importante que permite, por ejemplo, la manipulación de cualquier objeto mediante una referencia declarada como de tipo `Object`
- **`Object` no tiene clase base**:
  - La clase `Object` no hereda de nadie, por definición.

Herencia (6)

- **Constructores bajo Herencia:**
  - Dado que las clases que el programador codifica siempre heredarán de otra clase (ya que por defecto cualquier clase hereda directamente de `Object`) entonces se hace necesario rever los constructores.
  - Cuando se crea un objeto de una clase (por ejemplo `B`) se invoca primero al constructor de esa clase (al de `B`) pero antes de ejecutar su código se invoca al constructor **por defecto** de la clase base de B (A) y así sucesivamente hasta invocar al constructor de `Object`. Luego se van ejecutando en sentido descendente. Esto se conoce como "encadenamiento de constructores"

Herencia (7)

- **Encadenamiento de Constructores (cont.):**
  - Por lo tanto, para crear una instancia de una clase "abajo" en la jerarquía de clases, se debe invocar a todos los constructores desde esa clase hasta Object y luego ejecutarlos "hacia abajo" en la jerarquía.
  - Esto puede entenderse debido a que para crear un Auto, dado que es un tipo de Vehiculo, haya que inicializar los datos del vehículo y además los datos propios del auto (y ante todo los datos de Object pues todo es un objeto).
  - Cada constructor debería encargarse de inicializar los datos que le corresponden, es decir aquellos de la clase a la que pertenece.

Herencia (8)

- **Invocando Constructores:**
  - Ya se dijo que mediante `this(…)` se invocan constructores de la propia clase.
  - Mediante `super(…)` se puede alterar el encadenamiento de constructores invocando al constructor deseado de la super clase (pasando los parámetros correspondientes) también como la primera línea de código.
  - Cuando el programador no define ningún constructor y el lenguaje provee un constructor por defecto, ese constructor por defecto lo único que hace es invocar al constructor por defecto de su super clase.

Herencia (9)

- **Invocando Constructores (cont.):**
  - Es importante notar que cada subclase agrega sus atributos específicos.
  - Por tanto sólo los constructores de esa subclase podrán y deberán inicializar esos atributos específicos, y dejar que los atributos heredados los inicialice el constructor de la superclase (pasando sus valores con super(…)).
  - Esto justifica el buen uso e invocación de constructores a lo largo de toda la jerarquía de clases.

Herencia (10)

- **Redefinición:**
  - Dado que la subclase hereda las operaciones de la clase base, ¿Qué sucede si la subclase necesita proveer la misma operación que la clase base pero implementada de otra forma (con otro método)?
  - Una operación redefine a otra operación de su clase base cuado le asigna otro método en la subclase.
  - La redefinición es útil para brindar el mismo servicio (la misma operación) pero con las especificidades de la nueva subclase (otro método).

Herencia (11)

- **Redefinición (cont.):**
  - En muchos casos la subclase desea realizar más acciones de las que están definidas en el método de la super clase.
  - En estos casos no sería deseable tener que repetir todo el método de la clase base para luego agregar el código específico de la subclase.
  - Por tanto se permite, desde la subclase, invocar al método de la clase base mediante `super.metodo()`
  - Notar el uso del punto en contraposición a la invocación de constructores de la clase base (`super(...)`)

Herencia (12)

- **Sobrecarga:**
  - La sobrecarga es la capacidad del lenguaje de permitir que varias operaciones en la misma clase reciban el mismo nombre pero con diferente firma (es decir, diferente cantidad/tipo/orden de parámetros).
  - Los constructores son un buen ejemplo de sobrecarga.
  - No es un concepto nuevo de la OO.
  - La diferencia con la redefinición es que ésta última trata de la misma operación con diferentes métodos (en superclase y subclase), mientras que la sobrecarga son diferentes operaciones (en la misma clase).

Herencia (13)

- **Discusión** ¡¿?!
  - La primera línea de código dentro de un constructor puede ser o bien una invocación a otro constructor de la propia clase (`this(…)`) o a otro constructor de la clase base (`super(…)`) pero no se puede invocar a ambos. ¿Por qué cree que no se puede? ¿Qué ocurriría si se pudiera invocar a `this(…)` y además a `super(…)` en el mismo constructor?
  - ¿Cuál sería el código del constructor por defecto que se brinda automáticamente cuando el programador no define ningún constructor?

Herencia (14)

- **Discusión (cont.)** ¡¿?!
  - Operaciones que muestran (o devuelven) el estado actual de un objeto son comunmente redefinidas a lo largo de una jerarquía de clases, ¿Por qué?
  - En esos casos, ¿Suele convenir invocar a la operación redefinida? ¿Por qué?
  - ¿En qué casos no invocaría a la operación redefinida?
  - ¿Qué puede contener la clase `Object`?

Agenda

- Introducción a la Orientación a Objetos
- Definiciones y Conceptos Básicos
- Creación de Objetos
- Atributos y Métodos de Clase
- Herencia
- **Operaciones y Clases Abstractas**
- Polimorfismo
- Casteos
- Interfaces
- Asociaciones entre Clases
- Inner Classes e Inner Classes Anónimas

Operaciones y Clases Abstractas (1)

- **Operación Abstracta:**
  - Una operación abstracta es una operación sin método.
  - **Importante**: ya que gracias a la herencia se puede tener una operación definida en la clase base y redefinida en la subclase (siendo la misma operación con diferentes métodos) es importante notar que una operación será abstracta en aquella clase que no le provee un método.
  - Es decir, una operación puede ser abstracta en una clase, pero concreta en otra.

Operaciones y Clases Abstractas (2)

- **Clase Abstracta:**
  - Una clase abstracta es una clase que no puede instanciarse.
  - **Pero**: un objeto es instancia directa de la clase sobre la que se creó el objeto, e instancia indirecta de todas las super clases de ésa.
  - **Por lo tanto**: instancias de las subclases de una clase abstracta, serán instancias indirectas de la clase abstracta.
  - **Entonces**: una clase abstracta es una clase de la cual no se pueden obtener instancias directas.

Operaciones y Clases Abstractas (3)

- **Operación Abstracta y Clase Abstracta:**
  - ¿Qué relación existe entre ambos conceptos?
  - ¿Por qué comparten lo "abstracto"?
  - Notar que, en principio, son conceptos muy diferentes.
- **Caso**:
  - Suponga que Ud. tiene una clase concreta con una operación abstracta, y que crea una instancia directa de esa clase y llama a esa operación.
  - ¿Qué ocurrirá?

Operaciones y Clases Abstractas (4)

- **Operación Abstracta y Clase Abstracta (cont.):**
  - Este es un razonamiento por absurdo que muestra que existe algún error.
  - El único error es haber supuesto que una clase concreta puede tener una operación abstracta.
  - **Regla**: si una clase contiene (al menos) una operación abstracta, entonces la clase debe ser abstracta.
  - **Cuidado**: el recíproco no es cierto (una clase puede ser abstracta sin necesidad de tener alguna operación abstracta).

Operaciones y Clases Abstractas (5)

- **Operación Abstracta y Clase Abstracta (cont.):**
  - Como consecuencia de esa regla, si una clase B hereda una operación abstracta de su clase base A, entonces B tiene dos opciones:
    - Definirle un método en B
    - No definirle un método en B
  - En el segundo caso tendrá una operación abstracta (pues la hereda y no la redefine) por lo que debe ser ella también una clase abstracta (tanto A como B serán clases abstractas).

Operaciones y Clases Abstractas (6)

- **Discusión** ¡¿?!
  - ¿Cuándo definiría una clase abstracta?
  - ¿Tiene sentido que a una clase abstracta se le definan constructores?
  - ¿Cuándo definiría una operación abstracta?

Agenda

- Introducción a la Orientación a Objetos
- Definiciones y Conceptos Básicos
- Creación de Objetos
- Atributos y Métodos de Clase
- Herencia
- Operaciones y Clases Abstractas
- **Polimorfismo**
- Casteos
- Interfaces
- Asociaciones entre Clases
- Inner Classes e Inner Classes Anónimas

Polimorfismo (1)

- **Polimorfismo:**
  - Es un comportamiento en tiempo de ejecución que permite, al invocar una operación sobre una referencia, despachar el método definido en la clase del objeto actual apuntado por dicha referencia, aún cuando ésta sea declarada como de tipo clase base.
  - Se basa en el despacho dinámico de métodos, es decir el despacho basado en la información dinámica de una referencia (la información del objeto apuntado).
  - Existen varias formas de lograr un comportamiento polimórfico.

Polimorfismo (2)

- **Formas de obtener Polimorfismo:**
  - **Mediante redefinición de operaciones concretas**: por ejemplo, se redefine una operación con método heredado de la clase base.
  - **Mediante redefinición de operaciones abstractas**: se define (por 1ª vez) el método para una operación abstracta heredada de la clase base.
  - **Mediante redefinición de operaciones de una interfaz que se está implementando**: se verá más adelante.
- En los 3 casos se tiene la misma operación con diferentes métodos (¡cada uno en una clase diferente!)

Polimorfismo (3)

- **Uso del Polimorfismo:**
  - ¿Cómo utilizan las demás clases (ej: el método `main()` de la clase Program) una operación polimórfica?
  - **Idea**: utilizar una referencia declarada como del tipo de la clase base pero que apunte a un objeto de alguna de las subclases. Eso significa que el `main()` no sabe (ni le importa) verdaderamente a quién está apuntando.
  - **Entonces**: por reemplazabilidad + polimorfismo, basta con llamar a la operación polimórfica para que se ejecute el método correcto (es decir el método de la subclase cuyo objeto se está apuntando).

Polimorfismo (4)

- **Ventajas del Polimorfismo:**
  - El código "cliente" (ej: `main()`) no tiene porqué saber el tipo real del objeto apuntado por la referencia (sólo basta con que la referencia se declare como de tipo clase base).
  - ¡La invocación automáticamente hará despachar el método correcto! (*ver despacho estático y dinámico*)
  - **En caso de agregar una nueva subclase con su propio método para esa operación, todo el código cliente ya escrito permanece sin cambios, ¡pero con la posibilidad de llamar al nuevo método de la nueva clase!**

Agenda

- Introducción a la Orientación a Objetos
- Definiciones y Conceptos Básicos
- Creación de Objetos
- Atributos y Métodos de Clase
- Herencia
- Operaciones y Clases Abstractas
- Polimorfismo
- **Casteos**
- Interfaces
- Asociaciones entre Clases
- Inner Classes e Inner Classes Anónimas

Casteos (1)

- **Casteo:**
  - El casteo permite temporalmente a partir de una referencia, obtener una nueva referencia de otro tipo, apuntando al mismo objeto que la referencia original.
  - El casteo se hace necesario debido a la remplazabilidad.
- **Ejemplo**:
  - `A a = new B();  // B es subclase de A`
  - Sabemos que el tipo "real" de `a` es `B` (pues apunta a un objeto de tipo `B`) pero `a` fue declarada como de tipo `A`.
  - Entonces a la referencia `a` sólo se le podrán solicitar atributos y operaciones definidas en `A` (aún cuando sabemos que además de ser de tipo `A` también es de tipo `B`).

Casteos (2)

- **Ejemplo (cont.)**:
  - Por tanto, para poder acceder a los atributos y operaciones de B desde la referencia a, debemos castearla: `(B)a`
  - Este casteo es conocido como "casteo hacia abajo" pues partimos de una referencia como de tipo clase base, y terminamos con una referencia como de tipo subclase (asumiendo siempre que las jerarquías de clases se dibujan con la clase base arriba y la/s subclase/s abajo).
  - Si `oper()` fuese una operación que sólo existe en B, entonces para llamarla se escribe: `((B)a).oper()`
  - Los paréntesis extra son necesarios debido al orden de precedencia (el operador punto se ejecuta antes que el casteo).

Casteos (3)

- **Tipos de Casteo:**
  - **Casteo hacia abajo**: como en el ejemplo, es necesario explicitar "a qué clase se desea bajar", es decir cuál es la subclase de destino (pues pueden haber varias subclases diferentes).
  - **Casteo hacia arriba**: es implícito, no hay que escribir nada, pues por remplazabilidad, un objeto de tipo subclase también es de tipo clase base. Además, siempre que no exista herencia múltiple, habrá sólo una clase "a la cual subir".

Casteos (4)

- **Cuidado:**
  - El casteo (en especial "hacia abajo") es responsabilidad del programador.
  - Esto significa que aunque compile, si al momento de ejecutar la sentencia de código que contiene el casteo, la referencia **no** apunta "en realidad" a un objeto de la subclase a la cual se desea castear, se lanzará una **excepción**.
  - Por tanto muchas veces es conveniente consultar el tipo "real" mediante el operador `instanceof`
  - **Ejemplo**: `if (a instanceof B) …`

Polimorfismo y Casteos

- **Discusión** ¡¿?!
  - ¿Cómo lograría un resultado similar al del polimorfismo sin utilizar polimorfismo?
  - En base a lo anterior: ¿Qué consecuencias tiene el no utilizar polimorfismo?
  - ¿Qué sucede con el código cliente (ya programado y funcionando) si se agrega una nueva subclase en la jerarquía y se desea incluir el comportamiento de los nuevos objetos de esa subclase (analice el caso con y sin polimorfismo)?

Agenda

- Introducción a la Orientación a Objetos
- Definiciones y Conceptos Básicos
- Creación de Objetos
- Atributos y Métodos de Clase
- Herencia
- Operaciones y Clases Abstractas
- Polimorfismo
- Casteos
- **Interfaces**
- Asociaciones entre Clases
- Inner Classes e Inner Classes Anónimas

Interfaces (1)

- **Interfaz:**
  - Una interfaz es un conjunto de operaciones (no tiene métodos; tampoco constructores) con un nombre.
  - Sirve para especificar comportamiento.
  - Se codifica de forma muy similar a una clase (interface en lugar de class): luce como una clase asbtracta, sin atributos y cuyas operaciones son todas públicas y abstractas (aunque esto no hay que indicarlo).
  - Define un tipo de datos con el cuál se pueden declarar referencias, parámetros, retornos, etc.
  - No se puede obtener una instancia directa de una interfaz.

Interfaces (2)

- **Interfaz (cont.):**
  - El análogo a la herencia en el caso de interfaces se denomina **implementar** (o **realizar**): "la clase `A` implementa la interfaz `I`"
  - ¿Cuál es entonces la diferencia entre una interfaz y una clase abstracta con todas sus operaciones abstractas?
  - Que una clase puede heredar de solo una clase pero puede implementar cualquier cantidad de interfaces.
  - Una clase puede a la vez heredar de otra e implementar varias interfaces.

Interfaces (3)

- **Polimorfismo con Interfaces:**
  - De forma muy similar al polimorfismo mediante redefinición de operaciones concretas y abstractas, se puede obtener polimorfismo mediante interfaces.
  - La diferencia es que si se utiliza polimorfismo mediante interfaces, no es necesaria la herencia.
  - Dado que una clase sólo puede heredar de una sola otra clase, hace este tipo de polimorfismo más flexible.
  - ¿Cuál es el costo de esta flexibilidad? No poder heredar nada (¡salvo responsabilidades!)

Interfaces (4)

- **Discusión** ¡¿?!
  - Algunos lenguajes de programación no tienen interfaces (ej: C++): ¿Qué relación existe entonces entre las interfaces y la herencia múltiple?
  - Discuta las ventajas y desventajas del polimorfismo con y sin interfaces.
  - ¿El operador `instanceof` servirá también para interfaces? ¿Por qué?

Agenda

- Introducción a la Orientación a Objetos
- Definiciones y Conceptos Básicos
- Creación de Objetos
- Atributos y Métodos de Clase
- Herencia
- Operaciones y Clases Abstractas
- Polimorfismo
- Casteos
- Interfaces
- **Asociaciones entre Clases**
- Inner Classes e Inner Classes Anónimas

Asociaciones entre Clases (1)

- **Asociación:**
  - Una asociación entre dos clases es una relación entre ambas que permitirá a sus respectivos objetos estar vinculados durante un cierto tiempo.
- **Link**:
  - Mientras las clases se asocian, sus respectivos objetos se vinculan o linkean.
  - Un link es entonces una instancia de una asociación.
- La asociación es sólo una forma de relacionar dos clases (la herencia es otra, la implementación también – aunque involucra una interfaz).

Asociaciones entre Clases (2)

- **Multiplicidad:**
  - Las multiplicidades indican con cuántos otros se relacionan (linkean) los objetos de cada clase asociada.
  - Existen dos multiplicidades por cada asociación.
- **Ejemplo**:
  - Si cada cliente tiene muchas cuentas pero cada cuenta es de un cliente, entonces existirá una multiplicidad de N (de cliente a cuenta) y otra multiplicidad de 1 (de cuenta a cliente).
  - La multiplicidad N (muchos) suele denominarse *

Asociaciones entre Clases (3)

- **Navegabilidad:**
  - La navegabilidad indica el sentido en el cual se "conocen" los objetos linkeados.
- **Casos Posibles**:
  - El cliente conoce sus cuentas pero una cuenta no conoce de qué cliente es: navegabilidad de Cliente hacia Cuenta.
  - El cliente no conoce sus cuentas pero una cuenta sabe de qué cliente es: navegabilidad de Cuenta a Cliente.
  - Ambos se conocen: doble navegabilidad.

Asociaciones entre Clases (4)

- **Como Implementar una Asociación:**
  - Los lenguajes orientados a objetos no proveen una construcción específica (una palabra reservada).
  - No existe nada como `public association ...`
  - Por tanto se utilizan referencias.
- **Cuidado**:
  - La implementación de una asociación depende de las navegabilidades así como de las multiplicidades.
  - Éstas determinan en qué clase colocar la referencia y el tipo de referencia (si es una referencia a un objeto o si es una referencia a una colección).

Asociaciones entre Clases (5)

- **Discusión** ¡¿?!
  - ¿Por qué cree Ud. que se dice que la asociación es la forma de componer clases?
  - ¿Por qué no suele ser una buena idea implementar navegabilidades dobles en todas las asociaciones?
  - ¿Por qué cree que es útil en muchas ocasiones que una de las multiplicidades sea cero o uno?
  - ¿Cómo cree que se implementa (en código) que un objeto referencie a cero otros objetos?

Agenda

- Introducción a la Orientación a Objetos
- Definiciones y Conceptos Básicos
- Creación de Objetos
- Atributos y Métodos de Clase
- Herencia
- Operaciones y Clases Abstractas
- Polimorfismo
- Casteos
- Interfaces
- Asociaciones entre Clases
- **Inner Classes e Inner Classes Anónimas**

Inner Classes (1)

- Una Inner Class (o clase anidada) es una clase definida dentro del cuerpo de otra clase.
- Puede ser privada, protegida o pública.
- Tiene acceso a todos los miembros definidos en su clase huésped, incluso aquellos marcados como privados.
- La definición de una inner class dentro de una clase se justifica cuando su existencia depende de la existencia de la clase huésped.
- Generalmente se define para utilizarla exclusivamente dentro de la clase huésped.

Inner Classes (2)

- Si se piensa instanciar la clase anidada fuera de la clase huésped, además de definirle visibilidad public o protected, debe marcarse como estática.
- En tal caso los miembros de la clase anidada no podrán acceder a miembros de instancia de la clase huésped aunque sí obviamente a los marcados como static.

Inner Classes (3)

- **Ejemplo**:

```
public class ClaseHuesped {
    public String unAtributo;
    public int otroAtributo;

    public void unMetodo(int unParametro) {
        ClaseAnidada ca = new ClaseAnidada();
        ca.metodoEnAnidada();
    }

    private class ClaseAnidada {
        public void metodoEnAnidada() {
            unAtributo = "un valor";
            otroAtributo = 1;
        }
    }
}
```

Inner Classes Anónimas (1)

- Una inner class anónima es una clase que se define justo en el momento en que se necesita una instancia de dicha clase, y se sabe que no se necesitará en otro lugar.
- No tiene nombre (anónima) y se define a partir de una clase base existente o más comúnmente de una interfaz.
- En el mismo lugar donde se define, se utiliza el operador new para crear la instancia.
- En el cuerpo de la inner class anónima, generalmente se redefinen operaciones de su clase baso o interfaz. Es decir, son útiles para proporcionar nuevas implementaciones a las operaciones de la clase base o interfaz que serán utilizadas en un único lugar.

Inner Classes Anónimas (2)

- Por ejemplo, si un método necesita que le pasen por parámetro una instancia de A, se le puede pasar una nueva instancia de una inner class anónima que extienda o implemente A (según A sea una clase o interfaz), definida dentro de la propia invocación al método.
- Dentro de la inner class anónima se podrán redefinir operaciones de A, por lo que la instancia pasada al método, será un A (por reemplazabilidad), pero con una implementación particular.

Inner Classes Anónimas (3)

- **Ejemplo**:

```
...
public void unMetodo(A a) { ... }
...


...
unMetodo(new A() {
    // Aquí generalmente se redefinen operaciones
  de A…
});
...
```

Fin de Repaso de POO con Java

(lea por su cuenta nuevamente la Introducción)