UML

Conceptos de Objetos

Prof. Daniel Riesco

®

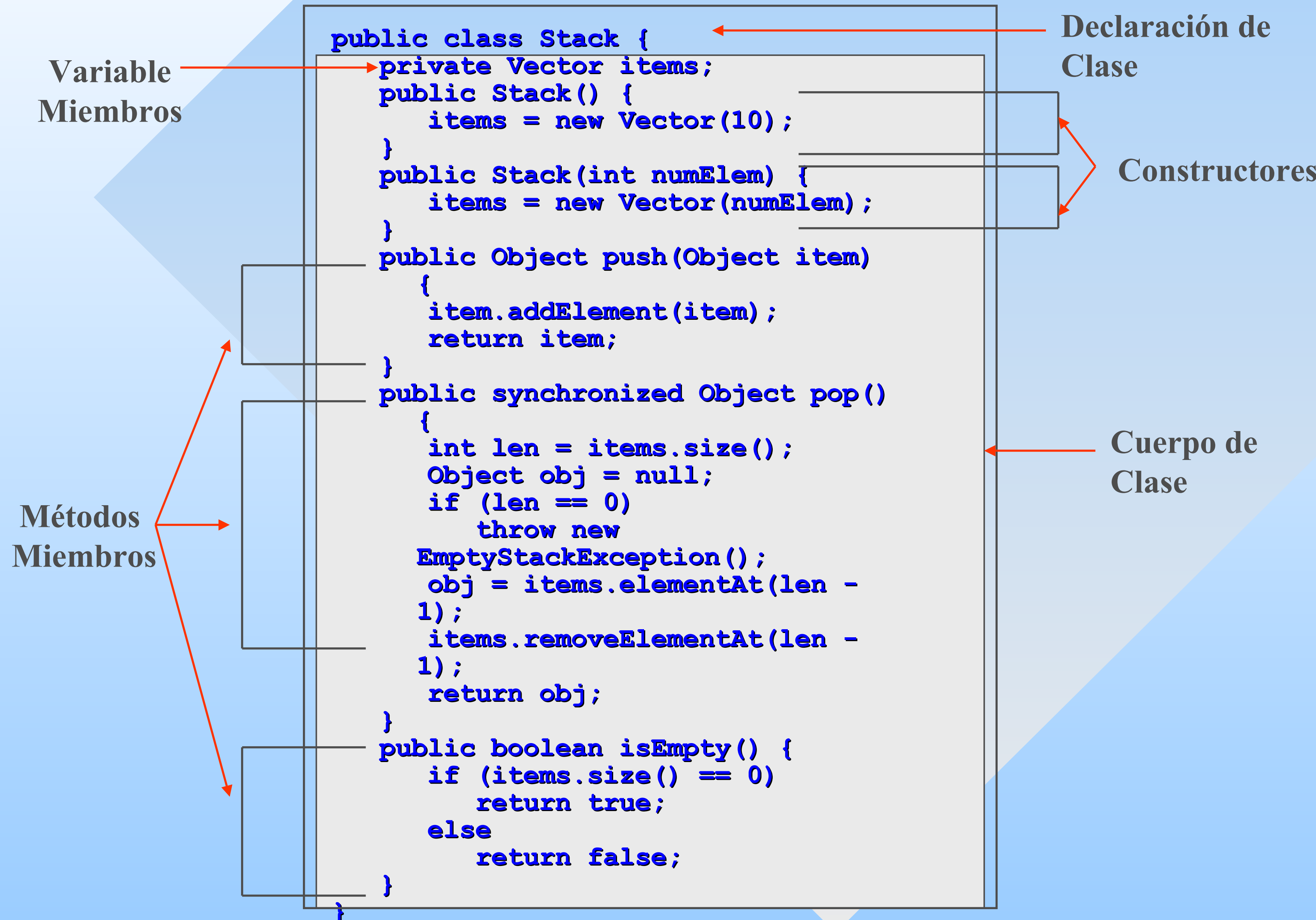
Abstracción

- Surge del reconocimiento de similitudes (concentración de las similitudes y olvidarse de las diferencias).
- Barrera de abstracción: concentrarse en la separación del comportamiento de su implantación (Visión externa de un objeto).
- Se debe aplicar el principio de mínima sorpresa (Evitar los efectos colaterales).
- Abstracción de:
  - Entidades (vocabulario del Dominio)
  - Acciones
  - Máquinas Virtuales
  - Coincidencia: Conjunto de operaciones que no tienen relación entre si.
- Concepto de Invariancia: establece las condiciones del contrato mediante pre y pos condiciones.
- Se deben centrar en las responsabilidades de la clase, no en su representación (obviar miembros privados).
- Protocolo: visión externa completa, estática y dinámica

Encapsulamiento

- Permite independencia de implementación.
- La implementación será indiferente en relación con el contrato con el cliente.
- Abstracción se centra en el comportamiento observable y el encapsulamiento se centra en la implementación de ese comportamiento (complementarios).
- También se llama ocultamiento de la información.
- Sirve para separar la interfaz de una abstracción de su implementación.

Definición de Clase



Modularidad

- Fragmentar un programa en componentes individuales para reducir su complejidad.
- En algunos lenguajes, las clases y objetos forman una estructura lógica, y éstas se situan en módulos para producir la estructura física del sistema.
- Cuando hay cientos de clases, los módulos son necesarios para manejar la complejidad.
- Los módulos pueden compilarse de forma separada. Independencia de implementación.
- Permiten diseñarse y revisarse independientemente
- Módulos cohesivos: agrupación de abstracciones que guardan cierta lógica.
- Módulos débilmente acoplados: Minimización de dependencias entre ellos.

Paquetes (Java)

- Un paquete es una colección de clases e interfaces relacionadas que proveen gestión del espacio de nombre y protección de acceso
- Identico nombre de archivo y Nombre de paquete
- Paquetes son importados vía `import`

```
package graphics;
class Circle extend Graphic implements Draggable
    {       ...    }
class Rectangle extends Graphic implements Draggable
    {       ...    }
interface Draggable {
        ...
    }
```

Objeto

- Objeto = Identidad + Estado + Comportamiento
- El estado está representado por los valores de los atributos
- Un atributo toma un valor en un dominio concreto
- Identidad (Object Identifier):
  - Constituye un identificador único y global para cada objeto dentro del sistema.
  - Es determinado en el momento de la creación del obj.
  - Es independiente de la localización física del objeto.
  - Es independiente de las propiedades del objeto, lo cual implica independencia de valor y de estructura
  - No cambia durante toda la vida del objeto.
  - No se tiene ningún control sobre los oids y su manipulación resulta transparente

Estado

- El estado de un objeto abarca todas las propiedades (normalmente estáticas) del mismo más los valores actuales (normalmente dinámicos) de cada una de esas propiedades
- El estado evoluciona con el tiempo
- Algunos atributos pueden ser constantes

Comportamiento

- El comportamiento agrupa las competencias de un objeto y describe las acciones y reacciones de ese objeto
- Denota un servicio que una clase ofrece a sus clientes.
- Una operación o mensaje es una acción que un objeto efectúa sobre otro con el fin de provocar una reacción.
- Las operaciones de un objeto son consecuencia de un estímulo externo representado como mensaje enviado desde otro objeto
- Los mensajes navegan por los enlaces, a priori en ambas direcciones
- Estado y comportamiento están relacionados

Mensaje

- La unidad de comunicación entre objetos se llama mensaje
- El mensaje es el soporte de una comunicación que vincula dinámicamente los objetos que fueron separados previamente en el proceso de descomposición
- Protocolo: los métodos asociados con un objeto forman su protocolo.
- Para un objeto no trivial se pueden formar grupos lógicos de comportamiento según el rol a desempeñar. Un rol define un contrato entre la abstracción y sus clientes. Una persona puede representar el rol de padre, docente, político, etc. y es la misma persona que cambia su rol.

Jerarquía

- Clasificación u ordenación de abstracciones.
- Especialización / Generalización. Simple.
- Múltiple introduce ciertas complejidades:
  - Colisión de nombres.
    - Class A {int a; m1()}; Class B {real a, m1()} Class C: A, B {...}; Soluciones:
    - No permitirlo por compilación – Smalltalk – Eiffel
    - Si es atributo y se refiere al mismo, toma uno. CLOS
    - Se hace referencia con calificación completa al atributo o método. C++.
    - Lista de Preferencia de Clases. CLOS.
      E → B → A, E → D → C → A.
  - Herencia repetida.
- Agregación – parte de.

Tipos

- Beneficios de los lenguajes con tipeo estricto
  - No se puede enviar un mensaje sobre un objeto si el mismo no está definido en la clase o superclase
  - Concordancia de tipos.
  - Reduce el tiempo de depuración
  - Ayuda a documentar
  - Codigo más eficiente generado por los compiladores
  - Sin tipos, un programa puede estallar misteriosamente
- Ligadura estática: Se fijan los tipos de todas las vbles en tpo de compilación.
  Cuadrado c1(10); ...; c1.dibujar()
- Ligadura dinámica: los tipos de las vbles no se conocen hasta la ejecución.
  Figura *figs[2];
  fig[0] = new cuad(10); fig[1] = new rect(10,20);

Herencia – Términos y Conceptos

- Toda clase en Java es derifada de la Clase Object
- Las clases en java pueden ser organizadas en jerarquías usando la palabra clave **extends**
- Establecer la relación **Superclase/Subclase**
- Una **Superclase** contiene miembros comunes a sus **Subclases** - Generalización
- **Subclases** contiene diferentes miembros de la **Superclase** compartida - Especialización

Subclase & Superclase

- Subclase
  - La subclase es una clase que extiende otra clase
  - Hereda el estado y comportamiento de sus ancestros
  - La subclase puede usar las variables y funciones miembros **heredadas** y sobreescribir las funciones miembros **heredadas**
- Superclase
  - Superclase es el ancestro directo de la clase

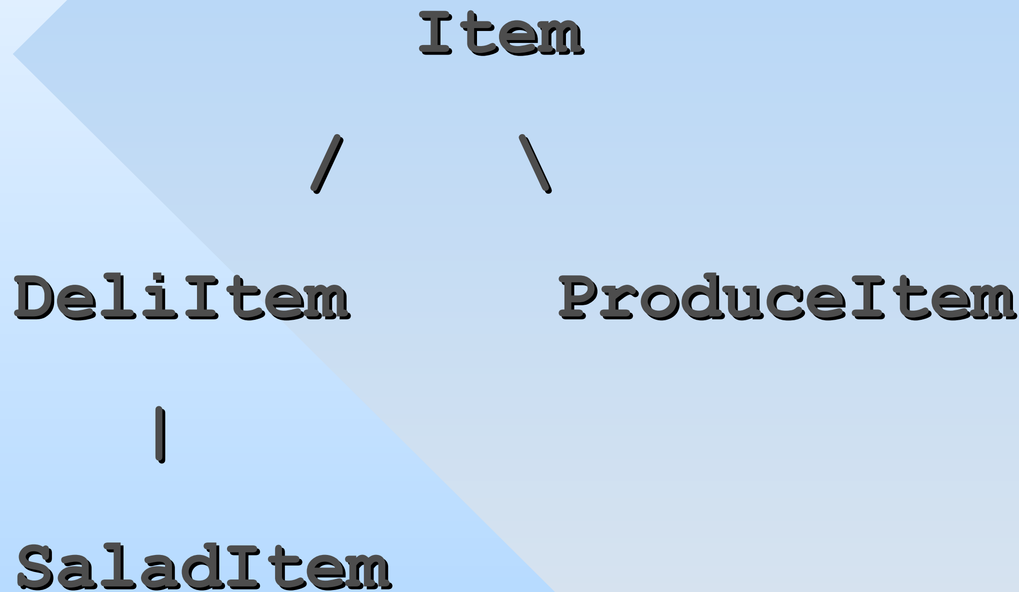
Una Superclase en Java

- Un Item (producto)

```
class Item {
  private   String   UPC, Name;
  private   int      Quantity;
  private   double   RetailCost;
  protected double   WholeCost;

  public           Item() { ... };
  public void      finalize() { ... };
  public boolean Modify_Inventory(int Delta){...};
  public int       Get_InStock_Amt()
                         {return Quantity;};
};
```

Herencia – Definiendo subclases




```
class DeliItem extends Item { ... };
class SaladItem extends DeliItem { ... };
class ProduceItem extends Item { ... };
```

Miembros heredados por una subclase

- La subclase hereda todos los miembros públicos/protegidos de una superclase
  - DeliItem hereda **Public/Protected** de Item
- La subclase no hereda miembros privados de una superclase
  - DeliItem no hereda **Private** de Item
- La subclase no hereda de un miembro de la superclase si la subclase declara un miembro con el mismo nombre
  - Métodos miembros – la subclase sobreescribe el de la superclase

Sobreescritura de métodos

```
class parentClass {
  boolean state;
  void setState() {
      state = true;
  }
}
class childClass extends parentClass {
  void setState() {
      state = false;
      super.setState();
      System.out.println(state);
      System.out.println(super.state)
  }
}
```

Clases y Métodos Abstractos

- Clases Abstractas
  - No pueden ser instanciadas
  - Solamente pueden crearse subclases
  - Ejemplo de una clase Abstracta es Number en el paquete java.lang
- Métodos Abstractos
  - Clases Abstractas pueden contener métodos abstractos
  - Esto permite a una clase abstracta provea a todas sus subclases con la declaración de métodos para todos los métodos

Ejemplo de Clase Abstracta

```
abstract class Item {
  protected String  UPC, Name;
  protected int     Quantity;
  protected double  RetailCost, WholeCost;

  public            Item() { ... };
  abstract  public  void     finalize();
  abstract  public  boolean
                       Modify_Inventory(int Delta);
  public    int     Get_InStock_Amt() {...};
  public    double  Compute_Item_Profit() {...};
  protected boolean
           Modify_WholeSale(double NewPrice);{...};
 };
```

Métodos y Clase Abstracta

```
abstract class GraphicsObject{
  int x, y;
  void moveTo(int x1,y1) { . . . . . }
  abstract void draw();
}
class Circle extends GraphicsObject{
  void draw() { . . . . . }
}
class Rectangle extends GraphicsObject{
  void draw() { . . . . . }
```

Polimorfismo

- Existe cuando hay herencia y ligadura dinámica
- Un solo nbre puede denotar objetos de distintas clases que se relacionan con alguna superclase, y reaccionan distinto con el mismo mensaje

```
...
for (i = 0...
    area = figuras[i].computar-area() ...
```

donde figura es un array de figura, siendo la superclase de cuadrado, rectángulo, triángulo, etc

- Agregar una nueva clase círculo.
- Pascal / Ada son monomórfico porque todo valor y vble puede interpretarse que tiene un tipo y sólo uno. El polimórfico una vble puede tener más un tipo.

Polimorfismo

- El polimorfismo vía Dispatching permite la elección dinámica o en tiempo de ejecución del método a ser llamado basado sobre la clase TIPO de la instancia invocada.
- **Promueve el reuso y la evolución del código**
- Incurre en costo/overhead en tiempo de compilación y de ejecución

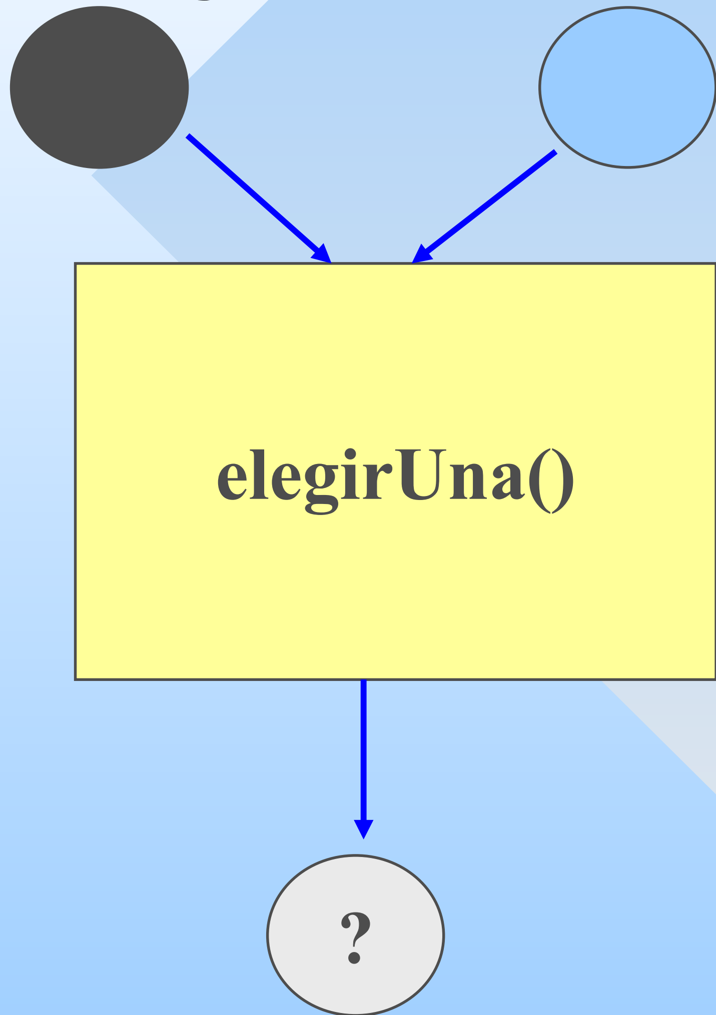
Polimorfismo

- Sustitución
  - Siempre que el valor de un cierto tipo es esperado, un subtipo puede ser provisto
  - En el contexto de herencia, todas las subclases pueden ser tratadas como una clase raíz común
  - Simplifica el código y facilita el reuso
- Tipo Estático
  - Tipo definido en la declaración de variable
- Tipo Dinámico
  - Tipo de valor real contenido por la variable en tiempo de ejecución

Polimorfismo

- Una variable es polimórfica siempre que pueda tener diferentes tipos estáticos y dinámicos.
  - Variable estática `I1` definida del tipo Item
  - Acceso a Variable dinámica permite `I1.Print()` sea invocada sobre las instancias de ItemEntregado, ItemProducido, etc.,
- Problemas:
  - **Polimorfismo inverso**: Puede la variable subtipo retornar su valor luego de la asignación de su valor a un supertipo?
  - **Ligadura de Métodos**: Cuando invocamos a un método sobre una variable, ¿sería seleccionado según su tipo estático o dinámico?

Un ejemplo



- Tenemos la clase Bola y dos subclases BolaBlanca y BolaNegra
- El método elegirUna() toma una BolaBlanca y una BolaNegra como argumentos y returna una de ellas elegidas aleatoriamente
- Preguntas:
  - ¿Que tipo retorna elegirUna()?
  - ¿Cómo conocemos que bola retornará?

Ligadura Dinámico y Casting

```
public class Ball {
   public String id = new String("I'm a
     ball");
   public String GetId() {
      return id;    }
}
public class WhiteBall extends Ball {
   public String id = new String("I'm
     white");
   public String GetId() {
      return id;
   }
   public void OnlyWhite() {
      System.out.println("Yes, I'm
     white");
   }
}
public class BlackBall extends Ball {
   public String id = new String("I'm
     black");
   public String GetId() {
      return id;
   }
   public void OnlyBlack() {
      System.out.println("Yes, I'm
     black");
   }
}
```

El tipo estático de b es Ball, pero su tipo dinámico puede ser *WhiteBall* o *BlackBall*.

```
class balls {
   public static void main(String[]
     args) {
      WhiteBall wb = new WhiteBall();
      BlackBall bb = new BlackBall();
      Ball b = SelectOne(wb, bb);
      System.out.println(b.GetId());
      System.out.println(b.id);
      if (b instanceof WhiteBall) {
          wb = (WhiteBall)b;
          wb.OnlyWhite();
      } else {
          bb = (BlackBall)b;
          bb.OnlyBlack();
      }
   }
   public static Ball
     SelectOne(WhiteBall w, BlackBall
     b) {
      if (Math.random() > 0.5)
         return w;
      else
         return b;
   }
}
```

Cual se imprime?

Clase Student

```
public class Student
{ // protected used to facilitate
   inheritance
  protected String name, SSN;
  protected float gpa;

  // constructor used to initialize
   object
  public Student(String n, String
   ssn,float g)
    {
      name = n;
      SSN = ssn;
      gpa = g;
    }

  public String getName()
    { return name; }

  public float getGpa()
    { return gpa; }

public String getSSN()
    { return SSN; }

  // diplay student
   information
  public void print()
    {

   System.out.println("Student
   name: " + name);
        System.out.println("
   SSN: " + SSN);
        System.out.println("
   gpa: " + gpa);
    }
}
```

Clase Grad

```
public class Grad extends Student
{
  private String thesis;

  public Grad(String name, String ssn, float gpa, String t)
    {
      // call parent's constructor
      super(name, ssn, gpa);
      thesis = t;
    }

  public String getThesis()
    { return thesis; }

  public void print()
    {
      super.print();
      System.out.println("      thesis: " + thesis);
    }
}
```

Clase Undergrad

```
public class Undergrad extends Student {
  private String advisor;
  private String major;

  public Undergrad(String name, String ssn, float gpa,
   String adv, String maj){
      super(name, ssn, gpa);
      advisor = adv;
      major = maj;
    }

  public String getAdvisor()
    { return advisor; }

  public String getMajor()
    { return major; }

  public void print() {
      super.print();
      System.out.println("     advisor: " + advisor);
      System.out.println("       major: " + major);
    }
}
```

Clase StudentDB

```
import java.util.Vector;

public class StudentDB{
  private Vector stuList;

  // constructor
  public StudentDB(int size){
      // allocate memory for
   vector
      stuList = new
   Vector(size);
    }

  // returns number of
   students stored
  public int numOfStudents()
  { return (stuList.size()); }

  public void insert(Student
   s)
  { stuList.addElement(s); }
```

```
// search for student by
   name
public Student
   findByName(String
   name){
   Student stu = null;
   // temp student
   boolean found =
   false;
   int count = 0;
   int DBSize =
   stuList.size();
   while ((count <
   DBSize)||(found ==
   false)){
     stu = (Student)
   stuList.elementAt(cou
   nt);
     if
   (stu.getName().equals
   (name))
             found =
   true;
           count++;
   }
   return stu;
 }
```

Clase StudentDB (II)

```
public Student
    remove(String name)
  {
    Student stu = null; //
    temp student
    boolean found = false;
    int count = 0;
    int DBSize =
    stuList.size();
    while ((count < DBSize)
    || (found == false)){
       stu = (Student)
    stuList.elementAt(count
    );
       if
    (stu.getName().equals(n
    ame))
        found = true;
       count++;
    }
    if (found == true)

    stuList.removeElementAt
    (count - 1);
   return stu;
  }
```

```
public void displayAll()
   {
     int DBSize =
    stuList.size();
       for (int i = 0; i <
    DBSize; i++)
     {
       Student stu = (Student)
    stuList.elementAt(i);
       stu.print();
       System.out.println();
     }
   }
}
```

Clase MainInterface

```
public class MainInterface
{
  private StudentDB db;

  public static void
   main(String[] args) {
    MainInterface studentInt =
   new MainInterface();
    studentInt.displayMenu();
  }

  // constructor
  public MainInterface() {
    db = new StudentDB(5);
  }
```

```
public void displayMenu(){
    int option = 0;
   System.out.println("\n  1. Insert
      ");
    System.out.println("  2. Delete ");
    System.out.println("  3. Search ");
    System.out.println("  4. Display
      ");
    System.out.println("  5. Exit \n");
    option = Console.readInt("Enter
      your choice: ");

    while (option > 0 && option < 5)
      {
      processOption(option);
      System.out.println("\n  1. Insert
      ");
      System.out.println("  2. Delete");
      System.out.println("  3. Search
      ");
      System.out.println("  4. Display
      ");
      System.out.println("  5. Exit
      \n");
      option = Console.readInt("Enter
      your choice: "); } }
```

Clase MainInterface (II)

```
public void processOption(int option)
{
String name, SSN;
float gpa;
switch (option){
  case 1:
    int type = Console.readInt("1. Grad or 2. Undergrad? ");
    name = Console.readString(“Name: ");
    SSN = Console.readString(“SSN: ");
    gpa = (float) Console.readDouble("gpa: ");
    if (type == 1){
      String thesis = Console.readString("Enter thesis
 title:");
        Student g = new Grad(name, SSN, gpa, thesis);
        db.insert(g);
      }
```

Clase MainInterface (III)

```
 else{
     String advisor = Console.readString("Enter advisor:");
     String major = Console.readString("Enter major: ");
     Student u = new Undergrad(name, SSN, gpa, advisor,
                                                major);
     db.insert(u);
  }
   break;

case 2:
     name = Console.readString(”Name");
     Student s = db.remove(name);
     s.print();
     break;
```

Clase MainInterface (IV)

```
case 3:
    name = Console.readString("Enter name: ");
    Student stu = db.findByName(name);
    System.out.println();
    stu.print();
    break;

 case 4:
    System.out.println();
    db.displayAll();
  }
 }
}
```

Visibilidad

- Public / Protected / Private.
- Clase A : B
  - Todos los miembros public y protected de B son también public y protected de A
- Clase A : private B
  - Los miembros public y protected de B se conviernte en privados de A.

Class A {protected:...a1; private:...a2; public:... a3}

Class B: private A {protected:...b1; private:...b2; public:... b3}

Class C: public B.

Válido: Desde C → b1, b3, Desde B → a1, a3

No válido: Desde C → b2, a3, a1, Desde B → a2

Medida de Calidad de una Abstracción

- Acoplamiento
  - La medida de la fuerza de la asociación establecida por una conexión entre una clase y otra.
- Cohesión
  - La medida del grado de conectividad entre los elementos de una sóla clase.
  - La menos deseable es la cohesión por coincidencia, inclusión de dos abstracciones sin ninguna relación.
  - La más deseable es la funcional, elementos de la misma clase trabajan todos juntos para proporcionar algún comportamiento bien delimitado.
- Suficiente: Captura suficientes características (interfaz mínima) para permitir interacción significativa y eficiente.
- Completo: Captura todas las características significativas de la abstracción (interfaz completa).

Beneficios de la Tecnología de Objetos

- Reutilización de componentes
  - Desarrollo más rápido
  - Mejor calidad
  - Estructura descompuesta → mantenimiento más fácil
  - Más fáciles de adaptar y escalar

Beneficios de la Tecnología de Objetos

- Proximidad de los conceptos de modelado respecto de las entidades del mundo real
  - Mejora captura y validación de requisitos
  - Acerca el “espacio del problema” y el “espacio de la solución”
- Modelado integrado de propiedades estáticas y dinámicas del ámbito del problema
  - Facilita construcción, mantenimiento y reutilización

Redefinición / Slicing

Class CA {public : virtual m1(); virtual m2()}

Class CB : CA {public: virtual m3(); virtual m1() }

Class CC : CA {public:virtual m4(); virtual m1()}

CA a1; CB b1, b2; CC c1;

- Son válidos:

a1 = b1 (Slicing – pérdida de información)

a1.m1(); b1.m1(); b1.m3(); b1.m2()

- No son válidos

b1 = a1; b1 = c1

a1.m3(); b1.m4()