5.1 ASSOCIATIONS AND LINKS

El Nombre de una relación estructural que existe entre clases es una asociación. Ejemplo con respecto al Sistema de registro de estudiantes

- Un Estudiante **esta matriculada a** un curso
- Un profesor **imparte** un curso
- Un programa de estudio r**equiere** un curso

Una asociación se refiere a una relación entre clases, el termino enlace se utiliza para referirse a una relación estructural que existe entre dos objetos específicos(instancias)

Dada la asociación "un alumno esta matriculada a un curso", se puede tener el siguiente enlace:

- Chloe Shylow (un objeto Estudiante en particular) está matriculada en Matemáticas 101 (un objeto Curso en particular).
- Fred Schnurd (un objeto Estudiante en particular) está matriculado en Basketweaving 972 (un objeto Curso en particular).
- Mary Smith (un objeto Estudiante concreto) está matriculada en Cestería 972 (un objeto Curso concreto; resulta que es el mismo objeto Curso al que está vinculado Fred Schnurd).

De la misma manera que un objeto es una instancia especifica de una clase con sus valores de atributo, un enlace se puede considerar como una instancia especifica que tiene objetos involucrados. (Figura 5.1)

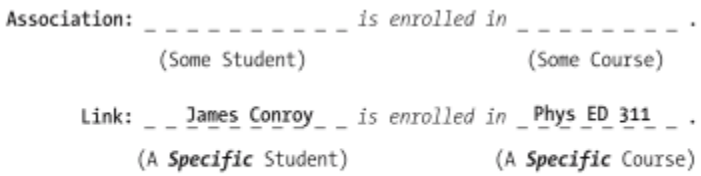


Figure 5-1. *An association is a template for creating links.*

Diferencia entre una asociación y un enlace:

- Una asociación es una relación potencial entre objetos de un determinado tipo/clase
- Un enlace es una relación real entre objetos de esos tipos concretos.

Por ejemplo

Dado cualquier objeto estudiante X y cualquier objeto curso Y, existe la posibilidad de que exista un enlace de tipo esta matriculada entre esos dos objetos precisamente porque hay una asociación este matriculado definida entre las dos clases a las que pertenecen los objetos, es decir, las asociaciones permiten establecer vínculos.

La mayoría de asociaciones surgen entre dos clases diferentes, son las asociaciones binarias y unarias.

Por ejemplo, la asociacion “esta matriculado” es una asociacion binaria ya que relaciona dos clases diferentes: “alumno” y “curso”. En cambio una asociacion unaria o reflexiva se produce entre dos instancias de la misma clase

Hay casos poco frecuentes donde un mismo objeto desempeñe los dos papeles en una relación reflexiva, ejemplo, en una relación con la asociación “un profesor es el jefe de departamento que representa a otros profesores”, el profesor que preside un departamento seria su propio representante.

Tambien son posibles asociaciones de orden superior. Una asociacion ternaria implica tres clases

Ejemplo

“un Estudiante toma un Curso de un Profesor en particular" se ilustra de esta manera

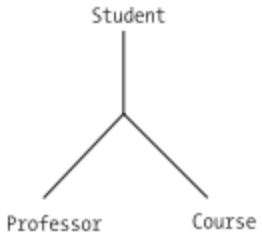


Figure 5-2. *A ternary association*

Normalmente se descomponen las asociaciones de orden superior en un numero apropiado de asociaciones binarias, ejemplo: podemos representar la asociacion tripartita anterior como tres asociaciones binarias:

- Un alumno asiste a un curso

- Un profesor imparte un curso
- Un profesor instruye un alumno

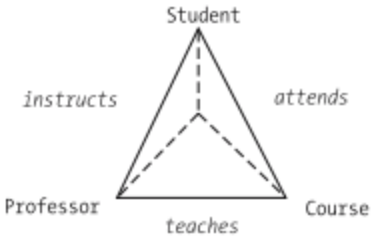


Figure 5-3. *An equivalent representation using three binary associations*

Dentro de una asociación determinada, cada clase participante tiene un papel. En la asociación (“un profesor asesora a un alumno”) el papel del profesor es ser “asesor” y el papel de estudiante seria “asesorado”.

Solo asignamos nombres a los roles de los objetos que participan en una asociación si ayuda a aclarar la abstracción. Ejemplo, en la asociación “un Alumno está matriculado en un curso” no hay necesidad de inventar nombres de rol para los extremos estudiante y curso de la asociación, porque tales nombres de rol no añadirían nada significativo a la claridad de la abstracción.

5.2 MULTIPLICITY

Para un determinado tipo de asociacion X entre las clases A y B, el termino multiplicidad se refiere al numero de objetos de tipo A que pueden asociarse a una instancia dada del tipo B. Por ejemplo un Alumno asiste a varios cursos, pero un alumno solo tiene un profesor en el papel de asesor.

Con una asociación uno a uno, exactamente una instancia de la clase A esta relacionada exactamente una instancia de la clase B, ni mas ni menos viceversa. Ejemplo:
- Un alumno tiene exactamente un expediente académico, y un expediente académico pertenece exactamente a un alumno.

- Un profesor preside exactamente un departamento, y un departamento tiene exactamente un profesor en el papel de presidente.

Podemos restringir aún mas una asociación indicando si la participación de la clase en es opcional u obligatoria. Por ejemplo, podemos cambiar la asociación anterior con lo siguiente:

- Un catedrático puede presidir exactamente un departamento, pero es obligatorio que un departamento tenga exactamente un catedrático como presidente.

En una **asociación uno a muchos** puede haber muchas instancias de la clase B relacionadas con una única instancia de la clase A de una manera particular, desde la perspectiva de una instancia de la clase B, solo puede haber una instancia de la clase A que este relacionada. Ejemplo:

- Un departamento emplea a muchos profesores, pero un profesor trabaja exactamente para un departamento.
- Un profesor asesora a muchos estudiantes, pero un estudiante determinado tiene exactamente un profesor como asesor.

Muchos puede ser de “cero o mas(opcional)” o como “uno o mas(obligatorio)”. Para ser específicos se pueden refinar las asociaciones anteriores de uno a muchos:

- Un Departamento emplea a uno o más ("muchos"; obligatorio) Profesores, pero un Profesor trabaja exactamente para un Departamento.

- Un Profesor asesora a cero o más ("muchos"; opcional) Estudiantes, pero un Estudiante dado tiene exactamente un Profesor como asesor.

En una **asociacion de muchos a muchos,** una sola instancia de la clase A puede tener muchas instancias de la clase B relacionadas con ella, y viceversa. Ejemplo:

- Un Estudiante se matricula en muchos Cursos, y un Curso tiene muchos Estudiantes matriculados en él.
- Un curso determinado puede tener muchos cursos como prerrequisito, y un curso determinado puede ser a su vez prerrequisito de muchos otros cursos. (Éste es un ejemplo de asociación reflexiva de muchos a muchos).

Muchos puede ser cero o mas o uno o mas en cualquiera de los extremos de una asocacion (m:m) ejemplo:

- Un Estudiante se matricula en cero o más ("muchos"; opcional) Cursos, y un Curso tiene uno o más ("muchos"; obligatorio) Estudiantes matriculados en él. más ("muchos"; obligatorio) Estudiantes matriculados en él.

5.3 MULTIPLICITY AND LINKS

Este capítulo comienza mostrando las diferencias entre las multiplicidades y los links, pero dando a entender que estos 2 son entre los objetos:

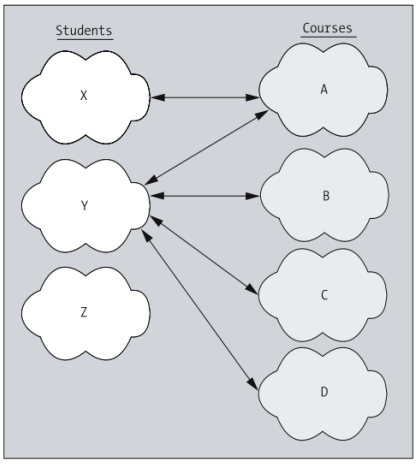
Multiplicidad: como aprendimos en diagrama de conceptos, las multiplicidades pueden ser: 1, *, 0..*, 1..*

Estas nos indican cuanto es la cantidad de relaciones que debería haber entre los 2 objetos que tienen su multiplicidad

Links: los enlaces son respecto a los objetos que fueron ya creados, como se relacionan a partir de la multiplicidad con los demás objetos.

El libro pone 2 ejemplos uno con multiplicidad 0..* a 1..* entre estudiantes y cursos: “un estudiante puede tener 0 o más cursos y un curso puede tener 1 o más estudiantes”.

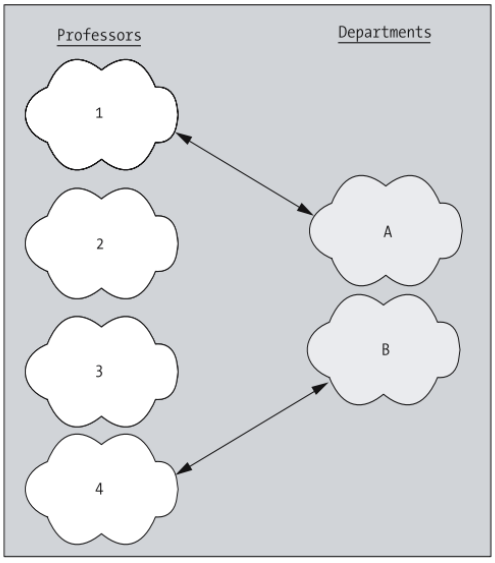
- Student X has one link (to Course A).
- Student Y has four links (to Courses A, B, C, and D).
- Student Z has no links to any Course objects whatsoever. (Z is taking the semester off!)



El otro ejemplo es con una multiplicidad de 0..1 a 1:

“un catedrático puede o no tener un departamento y un departamento debe tener exactamente un catedrático con el rol de ‘Presidente’ ”.

- `Professor` objects 1 and 4 each have one link, to `Department` objects A and B, respectively.
- `Professor` objects 2 and 3 have no such links.



5.4 AGGREGATION AND COMPOSITION

La agregación, se puede considerar como las clases que usan otras clases como complemento para desarrollarse de forma completa, además que la composición de puede considerar como el conjunto de todas estas clases y subclases que conforman la clase principal; el libro muestra 2 ejemplos donde se entiende la composición de estas clases:

For example, a car is composed of an engine, a transmission, four wheels, etc., so if `Car`, `Engine`, `Transmission`, and `Wheel` were all classes, then we could form the following aggregations:

- A `Car` ***contains*** an `Engine`.
- A `Car` ***contains*** a `Transmission`.
- A `Car` ***is composed of*** many (in this case, four) `Wheels`.

Or, as an example related to the SRS, we can say that

- A `University` ***is composed of*** many `Schools` (the School of Engineering, the School of Law, etc.).
- A `School` ***is composed of*** many `Departments`.

También se comenta que es necesario tener clara la diferencia de asociación con agregación, aunque en papel no es notoria la diferencia y asta el cap 10 no lo veamos con

mayor precisión, el libro sugiere que, si uno va a tomar el camino de modelaje UML, debe tener en cuenta esta sutil diferencia.

Además, la composición tiene mas fuerza que la agregación, la idea es que este sea que las clases dependientes no “funcionen o no sirvan” sin la clase principal, mientras que la agregación si permite que las clases funcionen para otras cosas aunque sea dependientes.

5.5 INHERITANCE

Mientras que muchas de las técnicas OO que ha aprendido para lograr un alto grado de flexibilidad y mantenibilidad del código (por ejemplo, encapsulación y ocultación de información) se pueden lograr con lenguajes que no son OO de una forma u otra, el mecanismo de herencia es lo que realmente distingue los lenguajes OO de sus contrapartes no OO. Antes de profundizar en una discusión en profundidad sobre cómo funciona la herencia, establezcamos un caso convincente a favor de la herencia analizando los problemas que surgen en su ausencia.

5.6 RESPONDING TO SHIFTING REQUIREMENTS WITH A NEW ABSTRACTION

Se plantea un caso para proponer las soluciones presentadas en las siguientes secciones:

Todas las demás características necesarias para describir a un estudiante de posgrado (los atributos nombre, ID de estudiante, etc., junto con los métodos de acceso correspondientes) son las mismas que ya hemos programado para la clase Estudiante, porque un estudiante de posgrado es un estudiante, después de todo. ¿Cómo podríamos abordar este nuevo requisito para una clase de GraduateStudent? Si no conociéramos bien los conceptos orientados a objetos, podríamos probar uno de los siguientes enfoques.

5.7 (INAPPROPRIATE) APPROACH #1: MODIFY THE STUDENT CLASS
((INADECUADO) ENFOQUE Nº 1: MODIFICAR LA CLASE STUDENT)

Tener que resolver si un estudiante es o no un estudiante graduado en todos y cada uno de los métodos de Student (el método de visualización es sólo uno) resulta en un código complicado que es difícil de depurar y mantener. Sin embargo, esto se complica si tenemos que añadir un tercer, o un cuarto, o un quinto tipo de estudiante "especializado" a la mezcla.

5.8(INAPPROPRIATE) APPROACH #2: “CLONE” THE STUDENT CLASS TO CREATE A GRADUATESTUDENT CLASS
((INADECUADO) ENFOQUE Nº 2: "CLONAR" LA CLASE STUDENT PARA CREAR UNA CLASE GRADUATESTUDENT)

Si quisiéramos cambiar el funcionamiento de un método concreto o cómo se define un atributo más adelante, por ejemplo, cambiar el tipo del atributo birthDate de String a Date,

con el correspondiente cambio en los métodos de acceso a ese atributo, entonces tendríamos que hacer los mismos cambios en ambas clases. El problema se complica.

Afortunadamente, disponemos de otro enfoque muy potente que podemos adoptar específico de los lenguajes de programación OO: podemos aprovechar el mecanismo de la herencia.

5.8 THE PROPER APPROACH (#3): TAKING ADVANTAGE OF INHERITANCE (EL ENFOQUE ADECUADO (#3): APROVECHANDO LA HERENCIA)

Herencia:

permite crear una nueva clase basada en una existente, especificando solo lo que es diferente en la nueva clase. en Java, la herencia se activa en una declaración de clase usando la palabra clave "extends"

ejemplo de clase GraduuateStudent que hereda características de Studen

la clase original de la que partimos, en este caso "Student", se denomina (direct) superclase. La nueva clase, "GraduateStudent", se llama (direct) subclase. Se dice que una subclase extiende a su superclase directa.

5.9 THE “IS A”NATURE OF INHERITANCE (LA NATURALEZA "ES UN" DE LA HERENCIA)

La herencia se suele referir como la relación "es un" entre dos clases, porque si una clase B (GraduateStudent) se deriva de una clase A (Student), entonces B realmente es un caso especial de A. A continuación se muestra un ejemplo:

- Un Student asiste a clases, entonces un GraduateStudent también asiste a clases.

si hay algo que se pueda decir sobre una clase A que no se pueda decir sobre una subclase propuesta B, entonces B realmente no es una subclase válida de A.

Como las subclases son casos especiales de sus superclases, se utiliza el término **especialización** para referirse al proceso de derivar una clase de otra

La generalización es un término que se utiliza para referirse al proceso opuesto: es decir, reconocer las características comunes de varias clases existentes y crear una nueva superclase común para todas ellas.

Ejemplo:

Supongamos que ahora deseamos declarar una clase Professor para complementar nuestra clase Student. Los estudiantes y los profesores tienen algunas características en común: atributos como nombre, fecha de nacimiento, etc., y los métodos que manipulan estos atributos. Sin embargo, también tienen características únicas, por ejemplo:

La clase Professor podría requerir los atributos título (un String) y worksFor (una referencia a un Departamento).

Debido a que cada clase tiene atributos que la otra encontraría inútiles, ninguna clase puede derivarse de la otra. No obstante, duplicar sus declaraciones de atributos y código de método común en dos lugares sería muy ineficiente. En tales circunstancias, querríamos inventar una nueva superclase llamada Person que tendrá todas las características que las dos clases comparten

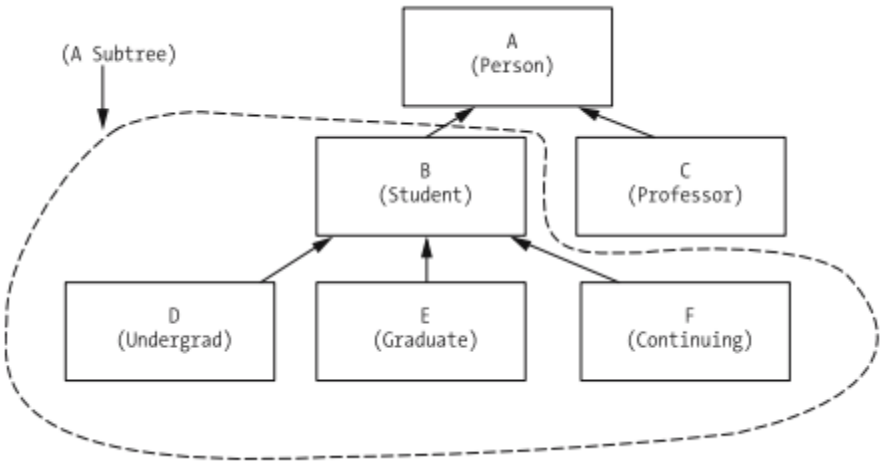
5.10 THE BENEFITS OF INHERITANCE (LOS BENEFICIOS DE LA HERENCIA)

La herencia es uno de los aspectos más poderosos y únicos de un lenguaje de programación orientado a objetos por las siguientes razones:

- **Reducción de redundancia de código:** Al heredar atributos y métodos de clases padre, se reduce la duplicación de código, facilitando el mantenimiento cuando cambian los requisitos.
- **Concisión de las subclases:** Las subclases son más concisas al contener solo lo que las diferencias de sus superclases directas.
- **Reutilización y extensión de código:** La herencia permite reutilizar y extender el código existente sin necesidad de modificarlo, lo que ahorra tiempo y evita la necesidad de volver a probar grandes partes de la aplicación.
- **Derivación de nuevas clases:** Se puede derivar una nueva clase de una existente incluso sin poseer el código fuente original, lo que permite una mayor productividad al construir sobre el trabajo existente sin necesidad de reescribir desde cero.

5.11 CLASS HIERARCHIES (JERARQUIA DE CLASES)

Es un árbol invertido de clases que se interrelacionan a través de la herencia. Se utilizan flechas para apuntar hacia arriba desde cada subclase a su superclase directa.



A continuación, un poco de su nomenclatura:

- Podemos referirnos a cada clase como un nodo de la jerarquía.
- Se dice que cualquier nodo de la jerarquía deriva (directa o indirectamente) de todos los nodos que se encuentran por encima de él en la jerarquía.
- Todos los nodos situados por encima de él en la jerarquía, conocidos colectivamente como sus antepasados.
- El antepasado inmediatamente superior a un nodo de la jerarquía se considera
- la superclase directa de ese nodo.
- A la inversa, todos los nodos situados por debajo de un nodo determinado en la jerarquía se consideran sus descendientes.
- El nodo situado en la parte superior de la jerarquía se denomina nodo raíz.
- Un nodo terminal, u hoja, es aquel que no tiene descendientes.
- Dos nodos derivados de la misma superclase directa se denominan hermanos.

Como cualquier jerarquía, ésta puede evolucionar con el tiempo:

- Puede ampliarse con la adición de nuevos hermanos/ramas en el árbol.
- Puede ampliarse hacia abajo como resultado de una futura especialización.
- Puede ampliarse hacia arriba como resultado de una futura generalización.

Estos cambios en la jerarquía se realizan a medida que surgen nuevos requisitos o que mejora nuestra comprensión de los requisitos existentes.

5.12 THE OBJECT CLASS (LA CLASE OBJECT)

En el lenguaje Java, la clase incorporada Object sirve como superclase definitiva para todos los demás tipos de referencia, tanto los definidos por el usuario como los incorporados al lenguaje. Incluso cuando no se declara explícitamente que una clase extiende a Object, dicha extensión está implícita. Es decir

```
public class Person { ... }
```

es como si hubiéramos escrito

```
public class Person extends Object { ... }
```

Así, la verdadera raíz de la jerarquía ilustrada y de todas las jerarquías de clases es la clase Object.

5.13 IS INHERITANCE REALLY A RELATIONSHIP? (¿ES LA HERENCIA REALMENTE UNA RELACIÓN?)

Se dice que la asociación, la agregación y la herencia son relaciones entre clases. Donde la herencia difiere de la asociación y la agregación es a nivel de objeto.

Las jerarquías de clases se expanden inevitablemente con el tiempo. entre sí en virtud de la existencia de una asociación entre sus respectivas clases. La herencia, en cambio, no implica la vinculación de objetos distintos, sino que es una forma de describir las características colectivas de un único objeto.

Por lo que la herencia es, en efecto, una relación entre clases, pero no entre objetos distintos.

5.14 AVOIDING “RIPPLE EFFECTS” IN A CLASS HIERARCHY (EVITAR EL "EFECTO DOMINÓ" EN UNA JERARQUÍA DE CLASES)

Los cambios en las clases que no son hojas (es decir, aquellas clases que tienen descendientes) tienen el potencial de introducir efectos dominó no deseados más abajo en la jerarquía.

Siempre que sea posible, evite añadir características a las clases que no sean hojas una vez que se hayan establecido en forma de código en una aplicación, para evitar efectos dominó en toda una jerarquía de herencia.

Sin embargo, refuerza la importancia de dedicar todo el tiempo posible a las fases de análisis de requisitos y modelado de objetos de un proyecto de desarrollo de aplicaciones OO antes de sumergirse en la fase de codificación.

5.15 RULES FOR DERIVING CLASSES: THE “DO’S” (REGLAS PARA DERIVAR CLASES: LO QUE HAY QUE HACER)

Al derivar una nueva clase, podemos hacer varias cosas para especializar la superclase de la que partimos.

- Podemos ampliar la superclase añadiendo características.
- También podemos especializar la forma en que una subclase realiza uno o más de los servicios heredados de su superclase.
- Especializar la forma en que una subclase realiza un servicio, es decir, cómo responde a un mensaje determinado en comparación con la forma en que su superclase habría respondido al mismo mensaje, se consigue mediante una técnica conocida como overriding.

5.16 OVERRIDING

overriding es lo que nos permite reescribir una clase sin tener que cambiar esa clase como tal, asi podemos modificar metodos instanciados en otra clase de el “mismo” tipo sin ser ella misma, aqui un ejemplo

Este ejemplo es sobre estudiantes, entonces tenemos la clase estudiantes y sus atributos, y un metodo para imprimir dichos atributos.

```
public class Student {
  // Attributes.
  private String name;
  private String studentId;
  private String majorField;
  private double gpa;
  // etc.

  // Accessor methods for each attribute would also be provided; details omitted.

  public void print() {
    // Print the values of all the attributes that the Student class
    // knows about; again, note the use of accessor methods.
    System.out.println("Student Name:  " + this.getName() + "\n" +
                       "Student No.:  " + this.getStudentId() + "\n" +
                       "Major Field:  " + this.getMajorField() + "\n" +
                       "GPA:  " + this.getGpa());
  }
}
```

Ahora podemos extender esta clase usandola en si misma, y ademas aumentar atributos o modificar metodos asi.

```
public class GraduateStudent extends Student {
  private String undergraduateDegree;
  private String undergraduateInstitution;

public void print() {
```



```
    // We print the values of all of the attributes that the
    // GraduateStudent class knows about:  namely, those that it
    // inherited from Student plus those that it explicitly declares above.
    System.out.println("Student Name:  " + this.getName() + "\n" +
                       "Student No.:  " + this.getStudentId() + "\n" +
                       "Major Field:  " + this.getMajorField() + "\n" +
                       "GPA:  " + this.getGpa() + "\n" +
                       "Undergrad. Deg.:  " + this.getUndergraduateDegree() +
                        "\n" + "Undergrad. Inst.:  " +
                        this.getUndergraduateInstitution());
  }
}
```

veamos que usamos extends para modificar la clase que sigue la cual es Student, y a esta extension la estamos llamando GraduateStudent, para modificar la funcion print y que esta ademas de imprimir todos los atributos de student, tambien imprima los nuevos atributos undergraduateDegree y undergraduateInstitution.

5.17 REUSING SUPERCLASS BEHAVIORS: THE “SUPER” KEYWORD

Ahora en lo anterior se repetia codigo y habia redundancia, entonces para evitar esto esta la palabra clave super, que hace referencia a la superclase a la que esta referenciada dicha extension, entonces usando esta opcion tenemos.

```
public class GraduateStudent extends Student {
  // Details omitted.

  public void print() {
    // Reuse code by calling the print method as defined by the Student
    // superclass ...
    super.print();

    // ... and then go on to do something extra - namely, print this derived
    // class's specific attributes.
    System.out.println("Undergrad. Deg.:  " + this.getUndergraduateDegree() + "\n" +
                       "Undergrad. Inst.:  " + this.getUndergraduateInstitution());
  }
}
```

vea que ahora hacemos super.print() haciendo refencia a que haga el print de la super clase, osea el print de student, y luego hace el system.out.println() de los nuevos atributos de GraduatedStudent.

De esta manera evitamos hacer muchas versiones de algo muy parecido solo que varie en pequeñas cosas.

5.18 RULES FOR DERIVING CLASSES: THE “DON’TS” Y 5.19 PRIVATE FEATURES AND INHERITANCE

Al derivar una nueva clase, hay algunas cosas que no debemos intentar hacer. Las reglas para derivar clases, conocidas como "Don'ts", incluyen:

1. No cambiar la semántica de una característica. Por ejemplo, si un método de impresión de una superclase como Estudiante está destinado a mostrar los valores de todos los atributos de un objeto en la ventana de comandos, entonces el método

de impresión de una subclase como EstudianteGraduado no debería anularse para dirigir toda su salida a un archivo en su lugar.

2. No se pueden eliminar físicamente características, ni deberíamos eliminarlas efectivamente ignorándolas. Intentar hacerlo rompería el espíritu de la jerarquía "es un". Por definición, la herencia requiere que todas las características de todos los ancestros de una clase A también se apliquen a la clase A misma para que A sea realmente una subclase adecuada. Si un EstudianteGraduado pudiera eliminar el atributo gradoBuscado que hereda de Estudiante, ¿realmente sería un EstudianteGraduado un Estudiante después de todo? Estrictamente hablando, la respuesta es no.
3. No deshabilitar un método prácticamente anulándolo con una versión de "no hacer nada". Esto rompe el contrato implícito de la herencia y puede conducir a comportamientos inesperados en el código.

Además, no debemos intentar cambiar la firma de un método cuando lo anulamos. Por ejemplo, si el método print heredado por la clase Estudiante de la clase Persona tiene la firma print(), entonces la clase Estudiante no puede cambiar el encabezado de este método para aceptar un argumento, digamos, print(int cantidadCopias). Hacerlo sería crear un método completamente diferente, debido a otro aspecto del lenguaje conocido como sobrecarga, un concepto que discutimos en el Capítulo 4. En resumen, al hacer esto, estaríamos violando las reglas de la sobrecarga de métodos y creando confusión en el código.

5.20 INHERITANCE AND CONSTRUCTORS (HERENCIA Y CONSTRUCTORES)

En el capítulo 4 se aprendió acerca de los constructores como procedimientos especiales utilizados para crear instancias de objetos. Ahora que se sabe acerca de la herencia, existen varias complejidades con respecto a constructores en el contexto de jerarquías de herencia. ***Los constructores no se heredan.***

super(...) for Constructor Reuse (*super(...) para la reutilización del constructor)*

Logramos la reutilización del código de un constructor de superclase mediante la misma superpalabra clave que se analizó anteriormente para la reutilización de métodos estándar de una superclase. Sin embargo, la sintaxis para reutilizar el código del constructor es un poco diferente. Si deseamos reutilizar explícitamente el constructor de una clase principal en particular, nos referimos a él de la siguiente manera en el cuerpo del constructor de la subclase:

```
super(arguments);  // note that there is no "dot" involved
                   // when reusing CONSTRUCTOR code
```

Usando super(argumentos); invocar un constructor de superclase es similar a usar this(arguments); para invocar un constructor desde otro en la misma clase.

Seleccionamos cuál de los constructores de una superclase deseamos reutilizar, si existe más de uno, en virtud de los argumentos que pasamos a super(...); Debido a que los constructores, si se sobrecargan para una clase determinada, tienen todos firmas de argumentos únicas, el compilador no tiene dificultad para seleccionar qué constructor de superclase estamos invocando.

Una cosa importante a tener en cuenta es que si llamamos explícitamente a un constructor de superclase desde un constructor de subclase usando la sintaxis super(...), la llamada debe ser la primera declaración en el constructor de la subclase; es decir, el siguiente constructor no podrá compilar:

```
public Student(String n, String s, String m) {
  this.setMajor(m);

  // This won't compile, because the call to the superclass's
  // constructor must come first in the subclass's constructor.
  super(n, s);
}
```

El requisito de realizar una llamada a super(...) como primera línea de código en un constructor surge en virtud de la naturaleza "es un" de la herencia. Cuando creamos un objeto Estudiante, en realidad estamos creando simultáneamente un Objeto, una Persona y un Estudiante, todo en uno. Entonces, ya sea que llamar explícitamente a un constructor de superclase desde un constructor de subclase usando super(...) o no, el hecho es que Java siempre intentará ejecutar constructores para todas las clases ancestrales de una clase determinada, desde la más general hasta la más específica de la jerarquía de la clase, antes de lanzarse al código constructor de esa clase dada.

Replacing the Default Parameterless Constructor *(Reemplazo del constructor sin parámetros predeterminado)*

Si no nos molestamos en definir ningún constructor explícito para una clase en particular, entonces Java intentará proporcionarnos un constructor sin parámetros predeterminado para esa clase. Cuando invocamos el constructor predeterminado para una clase derivada el compilador intentará automáticamente invocar un constructor sin parámetros para cada una de las clases antecesoras en la jerarquía de herencia de forma descendente.

La implicación es que si derivamos una clase B de la clase A y no escribimos constructores explícitos para B, entonces el constructor sin parámetros (predeterminado) de B buscará automáticamente un constructor sin parámetros de A. Esto generará un error al compilar. Esto se debe a que el compilador de Java intenta crear un constructor predeterminado sin parámetros y sin argumentos para la clase. Para hacerlo, el compilador sabe que necesitará poder invocar un constructor sin parámetros para la super clase desde el constructor predeterminado de Estudiante; sin embargo puede no existir tal constructor en la super clase.
La mejor manera de evitar tal dilema es recordar programar siempre explícitamente un constructor sin parámetros para una clase X cada vez que programe cualquier constructor explícito para la clase X, para reemplazar el constructor predeterminado "perdido".

5.21 A FEW WORDS ABOUT MULTIPLE INHERITANCE – 5.22 THREE DISTINGUISHING FEATURES OF AN POOL, REVISITED.

En este capitulo se trata el concepto de herencia múltiple en la programación orientada a objetos, donde una clase puede tener más de un ancestro inmediato, permitiendo la fusión de características de diversas clases. Para ejemplificar esto, consideremos las clases Professor y Student, ambas heredando de la clase base Person.

```
Public class Person {
   Private String name;

   Public String printDescription() {
      Return getName();
   }
}

Public class Student extends Person {
   Private String major;
   Private String studentId;

   Public String printDescription() {
      Return getName() + " [" + getMajor() + "; " + getStudentId() + "]";
```

```
    }
}

Public class Professor extends Person {
    Private String title;
    Private String employeeId;

    Public String printDescription() {
        Return getName() + " [" + getTitle() + "; " + getEmployeeId() + "]";
    }
}
```

Ambas subclases, Student y Professor, sobrescriben el método printDescription de la clase base para reflejar sus atributos específicos. Sin embargo, surge una complicación al considerar la creación de una clase híbrida, StudentProfessor, que herede de ambas Student y Professor.

El problema radica en que StudentProfessor no puede heredar ambas versiones del método printDescription, ya que generarían métodos sobrecargados con firmas idénticas, lo cual no es permitido por el compilador de Java. Además, puede que no sea deseable heredar ninguno de los métodos si ninguno aprovecha completamente los atributos de la otra clase.

Ante estas complicaciones, Java opta por no admitir herencia múltiple directa. En su lugar, ofrece interfaces como una alternativa para combinar características de múltiples clases sin los conflictos asociados a la herencia múltiple. Este enfoque se explora más a fondo en el capítulo 7 del texto. La decisión de Java refleja la preocupación por evitar ambigüedades y conflictos inherentes a la herencia múltiple, priorizando la claridad y simplicidad en el diseño de clases.