7.1 POLIMORFISMO

El polimorfismo se refiere a la capacidad de objetos de diferentes clases para responder de manera diferente al mismo mensaje.

Por ejemplo, un cirujano, un estilista y un actor podría interpretar el comando "¡Cortar!" de formas distintas.

- El cirujano comenzaría a realizar una incisión.
- El estilista comenzaría a cortar el cabello de alguien.
- El actor interrumpiría abruptamente la actuación de la escena actual, esperando orientación del director.

Estos tres profesionales diferentes pueden considerarse como objetos de la clase Persona pertenecientes a diferentes subclases profesionales: Cirujano, Estilista y Actor. A cada uno se le dio el mismo mensaje, "¡Cortar!", pero llevaron a cabo la operación de manera diferente según lo prescrito por la subclase a la que pertenecen

el override (sobreescritura) se refiere a la capacidad de una subclase para proporcionar una implementación específica de un método que está definido en una superclase. Esto se utiliza cuando la subclase necesita cambiar o agregar funcionalidad al método heredado de la superclase.

Ejemplo:

El aspecto más importante es la capacidad de sobrescribir el método print en las subclases para imprimir atributos específicos. Por ejemplo, el método print en la clase Estudiante imprime los atributos básicos, mientras que las subclases (GraduateStudent y UndergraduateStudent) sobrescriben este método para imprimir sus propios atributos adicionales.

```
public class Student {
    private String name;
    private String studentId;
    private String major;
    private double gpa;

    // Public get/set methods would also be provided (details omitted) ...

    public void print() {
        // We can print only the attributes that the Student class
        // knows about.
        System.out.println("Student Name:  " + getName() + "\n" +
            "Student No.:  " +  getStudentId()  + "\n" +
```

```
public class GraduateStudent extends Student {
    // Adding several attributes.
    private String undergraduateDegree;
    private String undergraduateInstitution;

    // Public get/set methods would also be provided (details omitted) ...

    // Overriding the print method.
    public void print() {
        // Reuse code from the Student superclass ...
        super.print();

        // ... and then go on to print this subclass's specific attributes.
        System.out.println("Undergrad. Deg.:  " +  getUndergraduateDegree() +
            "\n" + "Undergrad. Inst.:  " +
            getUndergraduateInstitution() + "\n" +
            "THIS IS A GRADUATE STUDENT ...");
    }
}
 public class UndergraduateStudent extends Student {
     // Adding an attribute.
     private String highSchool;

     // Public get/set methods would also be provided (details omitted) ...

     // Overriding the print method.
     public void print() {
         // Reuse code from the Student superclass ...
         super.print();

         // ... and then go on to print this subclass's specific attributes.
         System.out.println("High School Attended:  " + getHighSchool() +
             "\n" + "THIS IS AN UNDERGRADUATE STUDENT ...");
     }
 }
```

Otro punto importante es cómo podemos almacenar objetos de las subclases en una lista de tipo de la superclase (ArrayList<Student>). Esto se logra gracias al principio de la herencia, donde un objeto de subclase también se considera un objeto de la superclase.

Al iterar sobre esta lista y llamar al método print para cada objeto, se ejecutará la versión correspondiente del método según la subclase del objeto. Esto demuestra el polimorfismo, donde el mismo método puede comportarse de manera diferente según el tipo de objeto en tiempo de ejecución.

```
// Step through the ArrayList (collection) ...
for (Student s : studentBody) {
  // ... invoking the print method of each Student object.
  s.print();
}
```

7.2 POLYMORPHISM SIMPLIFIES CODE MAINTENANCE

El polimorfismo es una característica clave en la programación orientada a objetos que simplifica el mantenimiento del código. En contraste con un enfoque no polimórfico, donde se necesitarían múltiples condicionales para manejar diferentes tipos de objetos, el polimorfismo permite realizar operaciones comunes de manera concisa y robusta.

En un escenario sin polimorfismo, el código se vuelve complicado a medida que se agregan más casos, resultando en una estructura no tan buena. En cambio, con polimorfismo, una sola línea de código puede manejar todos los tipos de objetos, haciendo que el código sea más breve y resistente a cambios.

La fortaleza de el polimorfismo se destaca al introducir nuevas clases derivadas. En un ejemplo con estudiantes, se pueden añadir nuevas clases como PhDStudent y MastersStudent sin cambiar el código existente que itera sobre la colección de estudiantes. La herencia y la capacidad de anular métodos garantizan que el código polimórfico siga siendo efectivo

Ejemplo sin polimorfismo:

```
for (Student s : studentBody) {
    // Process each student.
    // Pseudocode.
    if (s is an undergraduate student)
        s.printAsUndergraduateStudent();
    else if (s is a Masters student)
        s.printAsMastersStudent();
    else if (s is a PhD student)
        s.printAsPhDStudent();
    else if (s is a generic graduate student)
        s.printAsGraduateStudent();
    else if ...
}
```

Con polimorfismo:

```
// Insert them into the ArrayList in random order.
studentBody.add(u1);
studentBody.add(p1);
studentBody.add(g1);
studentBody.add(m1);
// etc.

// Then, later in our application ...

// This is the EXACT SAME CODE that we've seen before!
// Step through the ArrayList (collection) ...
for (Student s : studentBody) {
    // ... and invoke the print method of the next Student object.
    // Because of the polymorphic nature of Java, this next line didn't require
    // any changes!
    s.print();
}

// Declare and instantiate an ArrayList.
ArrayList<Student> studentBody = new ArrayList<Student>;

// Instantiate various types of Student object.  We're now dealing with FOUR
// different derived types!
UndergraduateStudent u1 = new UndergraduateStudent();
PhDStudent p1 = new PhDStudent();
GraduateStudent g1 = new GraduateStudent();
MastersStudent m1 = new MastersStudent();
// etc.
```

En comparación, un enfoque no polimórfico requeriría modificaciones extensas para manejar nuevas subclases, generando un código difícil de mantener. El polimorfismo, al igual que la encapsulación y el ocultamiento de información, emerge como un mecanismo efectivo para minimizar los efectos colaterales y los costos de mantenimiento cuando los requisitos cambian después de la implementación.

7.3 THREE DISTINGUISHING FEATURES OF AN OBJECT-ORIENTED PROGRAMMING LANGUAGE (TRES CARACTERÍSTICAS DISTINGUIDAS DE UN LENGUAJE ORIENTADO OBJETO)

1. **Tipos Definidos por el Usuario:** Permite la creación de tipos personalizados, representando de manera intuitiva objetos del mundo real.
2. **Herencia:** Facilita la extensión de código existente sin cambiarlo, reduciendo significativamente los costos de mantenimiento.
3. **Polimorfismo:** Minimiza los impactos en el código cliente al agregar nuevas subclases a la jerarquía de clases, generando a su vez reducción de costos de mantenimiento.

7.4 THE BENEFITS OF USER-DEFINED TYPES (BENEFICIOS DE TIPOS DEFINIDOS POR EL USUARIO)

- Representación intuitiva de objetos del mundo real.
- Unidades convenientes de código reutilizable.
- Minimización de redundancia de datos a través de la encapsulación.
- Aislamiento contra efectos colaterales mediante el ocultamiento de información.
- Mayor facilidad para aislar errores en la lógica de la aplicación.

7.5 THE BENEFITS OF INHERITANCE (BENEFICIOS DE LA HERENCIA)

- Extensión de código desplegado sin cambios y reevaluación.
- Subclases más concisas, resultando en menos código para escribir y mantener.

7.6 THE BENEFITS OF POLYMORPHISM (BENEFICIOS DE LA POLIMORFISMO)

- Minimiza los “efectos de ondulación” en el código cliente al agregar nuevas subclases, reduciendo significativamente los costos de mantenimiento.

7.7 ABSTRACT CLASSES

Las clases abstractas se definen como clases que no pueden ser directamente instanciadas y pueden contener dentro métodos abstractos (referido a métodos que no tienen una implementación sino solo fueron mencionados).

Estas funcionan especialmente para las clases que heredan de esta clase abstracta inicial; pudiendo obtener una estructura común entre las clases hijas y, compartiendo características pero logrando especificar en mayor medida los métodos abstractos para lo que sea necesario.

Estas clases hijas implementan las clases abstractas de la clase madre, lo que permite una mayor flexibilidad y reutilización de los códigos.

Para llamar a las clases abstractas y sus métodos se usa:

```
abstract class Madre{
        public abstract void metodoAbstracto(); // no se colocan corchetes
        public void metodoNormal(){};
```

}

Class Hija1{

Public void metodoAbstracto(){

System.out.printl("dice a");

};

}

Class Hija2{

Public void metodoAbstracto(){

System.out.printl("dice b");

};

}

7.8 IMPLEMENTING ABSTRACT METHODS - 7.11 AN INTERESTING TWIST ON POLYMORPHISM

El proceso de implementar métodos abstractos cuando se deriva una clase de una superclase abstracta implica simplemente sobrescribir los métodos abstractos heredados con versiones concretas en la subclase. Esto se logra eliminando la palabra clave `abstract` del encabezado del método y proporcionando un cuerpo de método. Esto se ilustra en el ejemplo de derivar la clase `LectureCourse` de la clase abstracta `Course`.

En el ejemplo proporcionado, se muestra cómo se sobrescribe el método `establishCourseSchedule`. Una vez que se proporciona una implementación concreta para este método en la subclase `LectureCourse`, ya no es necesario que la subclase sea declarada como abstracta, ya que no contiene ningún método abstracto. Sin embargo, si una subclase derivada de una clase abstracta no implementa todos los métodos abstractos que hereda, esa subclase también debe ser declarada como abstracta.

Si una subclase olvida implementar un método abstracto, como en el caso de `IndependentStudyCourse`, se debe corregir la omisión implementando el método abstracto faltante o declarando la subclase en su totalidad como abstracta. De lo contrario, la compilación generará un error indicando que la subclase debe ser declarada como abstracta porque no implementa todos los métodos abstractos de la superclase.

En resumen, al derivar una clase de una superclase abstracta, es necesario implementar todos los métodos abstractos heredados en la subclase o declarar la subclase como abstracta si no se implementan todos los métodos abstractos.

Las clases abstractas no pueden ser instanciadas, lo que significa que no se pueden crear objetos de ellas directamente. Esto se debe a que las clases abstractas pueden contener métodos abstractos, los cuales no tienen implementación definida. Si intentamos crear un objeto de una clase abstracta, obtendremos un error de compilación.

La razón de esta restricción es que las clases abstractas pueden contener métodos abstractos que deben ser implementados por las clases concretas que las heredan. Si pudiéramos instanciar una clase abstracta, estaríamos tratando de llamar a métodos que no tienen una implementación definida, lo que conduciría a un comportamiento indefinido en tiempo de ejecución.

Sin embargo, aunque no podemos crear objetos de una clase abstracta, sí podemos declarar variables de referencia de ese tipo. Esto es útil para facilitar el polimorfismo, ya que nos permite tratar objetos de clases concretas que heredan de la clase abstracta de manera uniforme.

Una característica interesante de las clases abstractas es que un método concreto en una clase abstracta puede invocar un método abstracto en la misma clase. Esto puede parecer contradictorio, pero en realidad no lo es, ya que cuando invocamos el método concreto, el método abstracto ya habrá sido implementado por la clase concreta que hereda de la clase abstracta.

En resumen, las clases abstractas proporcionan un mecanismo para definir métodos comunes y requerir que las clases concretas implementen ciertos métodos. Aunque no se pueden instanciar objetos de clases abstractas, son fundamentales para la implementación de la herencia y el polimorfismo en Java.

7.12 INTERFACES

Una clase es una abstracción de un objeto del mundo real del que se han omitido algunos de talles que no son esenciales, por lo que, una clase abstracta es más abstracta que una clase concreta, ya que la clase abstracta omitide los detalles de cómo se van a realizar uno o más servicios particulares.

Con la clase abstracta se pueden evitar programar los cuerpos de los métodos que se declaran abstractos. ¿Qué ocurre con las estructuras de datos de una clase abstracta?

En el ejemplo de clase abstracta curso, prescribimos los atributos que pensamos que serían necesarios genéricamente por todos los tipos de cursos, de modo que una estructura de datos común seria heredada por todas las subclases.

```
private String courseName;
private String courseNumber;
private int creditValue;
private ArrayList enrolledStudents;
private Professor instructor;
```

Supongamos que solo se quiere especificar los comportamientos comunes del curso y no nos molestamos en declarar los atributos, los atributos suelen declararse como privados, puede que no queramos especificar la estructura de datos privada que una subclase debe utilizar para conseguir los comportamientos públicos deseados, pero esta decisión es del diseñador de la subclase.

Ejemplo: queremos definir lo que significa enseñar en una universidad. Tal vez, para enseñar, un objeto debería ser capaz de realizar los siguientes servicios:

1. Aceptar impartir un curso en concreto
2. Designar un libro de texto para el curso
3. Definir un programa de estudios para el curso
4. Aprobar la inscripción de un alumno concreto en el curso

Cada comportamiento puede formalizarse especificando una cabecera de método que represente como un objeto capaz de enseñar se le pediría realizar cada uno de estos comportamientos:

```
public boolean agreeToTeach(Course c)
public void designateTextbook(TextBook b, Course c)
public Syllabus defineSyllabus(Course c)
public boolean approveEnrollment(Student s, Course c)
```

Podríamos declarar una clase abstracta llamada profesor que no prescribe ninguna estructura de datos y solo métodos abstractos:

```
public abstract class Teacher {
    // We omit attribute declarations entirely, allowing subclasses to establish
    // their own class-specific data structures.

    // We declare only abstract methods.
    public abstract boolean agreeToTeach(Course c);
    public abstract void designateTextbook(TextBook b, Course c);
    public abstract Syllabus defineSyllabus(Course c);
    public abstract boolean approveEnrollment(Student s, Course c);
}
```

A continuación, creamos profesor como una derivación concreta de profesor:

```
public Professor extends Teacher {
    // Declare relevant attributes.
    private String name;
    private String employeeID;
    private ArrayList teachingAssignments; // of Section objects
    // etc.

    // Provide concrete implementations of all inherited abstract methods.
    public boolean agreeToTeach(Course c) { ... }
    public void designateTextbook(TextBook b, Course c) { ... }
    public Syllabus defineSyllabus(Course c) { ... }
    public boolean approveEnrollment(Student s, Course c) { ... }

    // Additional methods can also be declared - details omitted.
}
```

Sin embargo, si nuestra intención es declarar un conjunto de cabeceras de métodos abstractos para definir lo que significa asumir determinado rol dentro de una aplicación sin imponer ni una estructura de datos ni comportamiento concreto a las subclases, entonces la forma preferida de hacerlo es con una interfaz, quedaría de esta manera:

```
// Note use of "interface" vs. "class" keyword.
public interface Teacher {
    boolean agreeToTeach(Course c);
    void designateTextbook(TextBook b, Course c);
    Syllabus defineSyllabus(Course c);
    boolean approveEnrollment(Student s, Course c);
}
```

Observaciones sobre la sintaxis de interfaces:

1. Utilizamos la palabra clave interfaz en lugar de clase al declararlas
   ```
   public interface Teacher { ... }
   ```
2. Como todos los métodos de una interfaz son implícitamente públicos y abstractos, no se necesita especificar ninguna de esas dos palabras claves al declararlos (aunque hacerlo no generara un error de compilación). Pero se obtendrá un error si intentamos asignar a un método otra cosa que no sea accesibilidad publica a un método dentro de una interfaz:
   ```
   public interface teacher {
       // This won't compile - interface methods must all be public.
       private void takeSabbatical();
       // etc.
   ```
   Aquí está el error de compilación:
   ```
   modifier private not allowed here
   private void takeSabbatical();
   ^
   ```
3. Dado que todos los métodos prescritos por una interfaz son abstractos, ninguno tiene cuerpo

Al igual que ocurre con las clases, el código fuente de cada interfaz suele ir en su propio archivo .java, cuyo nombre interno debe coincidir con el nombre de la interfaz que contiene (por ejemplo, interfaz profesor iría con un archivo llamado profesor.java).

Las interfaces se compilan en bytecode del mismo modo que se compilan las clases. Ejemplo, el comando:

```
javac Teacher.java
```

Producirá un archivo bytecode llamado Teacher.class

Tener en cuenta, las interfaces no pueden declarar variables y no pueden declarar ningún método implementado. Estas son simplemente. Una colección de cabeceras de métodos abstractos.

7.13 IMPLEMENTANDO INTERFACES

Una vez definida una interfaz como Profesor, podemos designar varias clases de objetos como profesores, por ejemplo, Profesores, Estudiantes u objetos Persona genéricos, simplemente declarando que la clase de interés implementa la interfaz Profesor con la siguiente sintaxis:

```
// Implementing an interface ...
public class Professor implements Teacher { ... }
```

That is, rather than using the `extends` keyword, as we do when one class is derived from another, we use the `implements` keyword.

Si estamos implementando una interfaz de nuestro propio diseño (por ejemplo, Teacher) que pertenece al paquete por defecto, no necesitamos usar la directiva import para hacer que esa interfaz sea visible para el compilador.

Por otro lado, si deseamos implementar un tipo de interfaz Java predefinido (el lenguaje Java proporciona muchos de ellos), debemos utilizar una directiva import para dar a conocer ese tipo de interfaz al compilador, por ejemplo:

```
import packagename.PredefinedInterfaceType;

public class MyClass implements PredefinedInterfaceType { ... }
```

Verás las directivas de importación en uso cuando hablemos de la interfaz java.util.Collection un poco más adelante en este capítulo.

Una vez que una clase declara que implementa una interfaz, la clase que la implementa debe implementar todos los métodos declarados por la interfaz en cuestión para satisfacer al compilador. Por ejemplo, supongamos que codificamos la clase Profesor de la siguiente manera, implementando tres de los cuatro métodos exigidos por la interfaz Profesor, pero olvidando codificar el método aprobarInscripcion:

```
public class Professor implements Teacher {
    private String name;
    private String employeeId;
    // etc.

    // We implement three of the four methods called for by the
    // Teacher interface, to provide method bodies.

    public boolean agreeToTeach(Course c) {
        logic for the method body goes here; details omitted ...
    }

    public void designateTextbook(TextBook b, Course c) {
        logic for the method body goes here; details omitted ...
    }

public Syllabus defineSyllabus(Course c) {
    logic for the method body goes here; details omitted ...
}

// However, we FAIL to provide an implementation of the
// approveEnrollment method.

// Other miscellaneous methods of Professor unrelated to the Teacher
// interface could also be declared ... details omitted.
}
```

Si intentamos compilar la clase Profesor como se acaba de mostrar, obtendríamos el siguiente error:

```
Professor should be declared abstract; it does not define
approveEnrollment(Student, Course) in Teacher
```

Este error es del mismo tipo que se genera si estamos derivando una clase de una clase abstracta y no sobrescribimos uno de los métodos abstractos heredados. Este es el resultado de un ejemplo anterior, que involucra la superclase abstracta curso y la subclase cursoEstudioIndependiente:

```
IndependentStudyCourse should be declared abstract; it does not implement
establishCourseSchedule(String, String) in Course
```

Implementar una interfaz es conceptualmente similar a extender una clase abstracta, en el sentido de que tanto las interfaces como las clases abstractas son construcciones alternativas para prescribir comportamientos abstractos que las implementaciones/subclases deben ser capaces de llevar a cabo.

Cuando usar una o la otra:

1. Si deseamos impartir una estructura de datos concreta que acompañe a estos comportamientos prescritos o si necesitamos proporcionar algunos comportamientos concretos junto con los comportamientos abstractos, crearemos una clase abstracta.

2. Si no, creamos una interfaz

7. 14. OTRA FORMA DE RELACIÓN "ES UNA”

- Si la clase Professor extiende la clase Person, un Professor es un Person.

- Si la clase Professor implementa la interfaz Professor, un Profesor es un Professor.

Cuando una clase A implementa una interfaz X, se puede decir que todas las clases derivadas de A también implementan la misma interfaz X.

Por ejemplo, si derivamos una clase llamada AdjunctProfessor de Professor, entonces como Professor implementa la interfaz Teacher, un AdjunctProfessor también es Professor.

Esto tiene un sentido intuitivo, porque AdjunctProfessor heredará todos los métodos llamados por la interfaz Teacher de Professor tal cual, u opcionalmente anulará uno o más de ellos. De cualquier manera, un AdjunctProfessor estará "equipado" para realizar todos los servicios requeridos para desempeñar el papel de Professor.

7.15. INTERFACES Y CASTING

Por ejemplo, suponiendo que tanto la clase Student como la clase Professor implementen la interfaz Professor, la última línea del siguiente código generará un error de compilación:

```
Professor p = new Professor();
Student s = new Student();
Teacher t;

// Details omitted.

// The compiler won't allow this.
p = t;
```

Ahora, suponiendo que Student y Professor implementan Teacher:

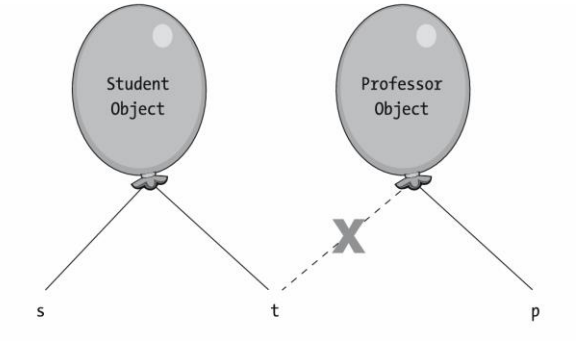
En el siguiente código asignar s a t es válido pero no asignar t como Professor ya que t está refiriendo a un Student.

```
// We instantiate both a Professor and a Student object; recall that
// in this example, both classes implement the Teacher interface.
Professor p = new Professor();
Student s = new Student();
Teacher t;

// We assign a Student reference to t.  This is permitted, because a Student
// is a Teacher.
t = s;

// Details omitted ...

// Later on, we mistakenly try to cast t as a Professor, but t is really
// referring to a Student.
p = (Professor) t;
```



t se refiere a un objeto Student en tiempo de ejecución y, por lo tanto, al intentar emitir t como Professor genera una ClassCastException.

7.16 IMPLEMENTING MULTIPLE INTERFACES (IMPLEMENTACIÓN DE MÚLTIPLES INTERFACES)

Una distinción importante entre extender una clase abstracta e implementar una interfaz es que mientras que una clase dada sólo puede derivar de una superclase directa, una clase puede implementar tantas interfaces como desee, de la siguiente forma

```
public class ClassName implements Interface1, Interface2, ..., InterfaceN { ... }
```

Al hacerlo, la clase implementadora tendría que implementar todos los métodos prescritos por todas estas interfaces colectivamente, por ejemplo, si inventáramos una segunda interfaz llamada Administrador, que a su vez especificara las siguientes cabeceras de método:

```
public interface Administrator {
    boolean approveNewCourse(Course c);
    boolean hireProfessor(Professor p);
    void cancelCourse(Course c);
}
```

Cuando una clase implementa más de una interfaz, sus objetos son capaces de asumir múltiples identidades o roles en una aplicación; por lo tanto, dichos objetos pueden ser "manejados" por varios tipos de variables de referencia, por ejemplo

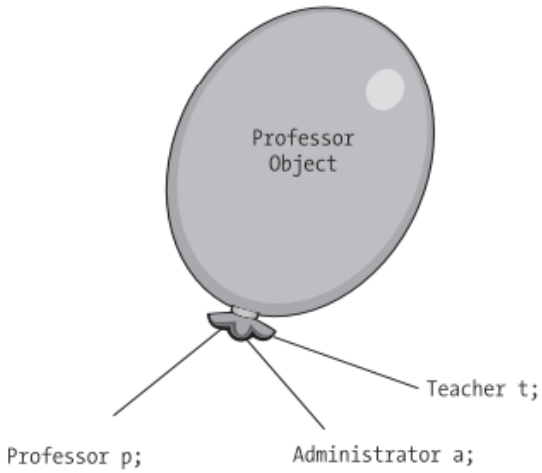
```
// Instantiate a Professor object, and maintain a handle on it via
// a reference variable of type Professor.
Professor p = new Professor();

// We then declare reference variables of the two types of interfaces that the
// Professor class implements.
Teacher t;
Administrator a;

t = p;  // We store a second handle on the same Professor in a reference variable of
        // type Teacher; this is possible because a Professor IS A Teacher!

a = p;  // We store a third handle on the same Professor in a reference variable of
        // type Administrator; this is possible because a Professor IS AN
        // Administrator!
```

Lo anterior mostrado de forma gráfica podría ser lo siguiente:



A Professor object can be “handled” by Professor, Teacher, and Administrator references, because a Professor is a Teacher, and a Professor is an Administrator!

Figure 7-5. *A Professor object has three different identities/roles in our application.*

Una clase puede extender simultáneamente una única superclase e implementar una o más interfaces, como se indica a continuación:

```
public class Professor extends Person implements Teacher, Administrator { ... }
```

7.17 INTERFACES AND CASTING (INTERFACES Y MOLDEADO)

Observemos que, a pesar de que t estaría haciendo referencia a un objeto Profesor, no podemos pedir a t que ejecute un método declarado por la clase Profesor

```
Professor p = new Professor();
Teacher t = p;

// setDepartment is a method defined for the Professor class, but t is
// declared to be of type Teacher ... this won't compile!
t.setDepartment("Computer Science");
```

Arroja el siguiente error

```
cannot find symbol
symbol:  method setDepartment(String)
location:  interface Teacher
```

Si necesitamos que t se refiera a un objeto Profesor en tiempo de ejecución, podemos ordenar a t en tiempo de compilación como Profesor.

```
Professor p = new Professor();
Teacher t = p;

// setDepartment is a method defined for the Professor class; since we know
// that t will refer to a Professor at run time, we use a cast so that
// this will compile.
((Professor) t).setDepartment("Computer Science");
```

7.18 INTERFACES AND INSTANTIATION (INTERFACES E INSTANCIACIÓN)

Al igual que las clases abstractas, las interfaces no pueden instanciarse. Es decir, si definimos que Teacher es una interfaz no podemos instanciarla directamente

```
Teacher t = new Teacher();  // Impossible!
```

Porque las interfaces no tienen constructores, sólo las clases, como plantillas para instanciar objetos.

Aunque se nos impide instanciar una interfaz, se nos permite sin embargo declarar variables de referencia de un tipo de interfaz, como podíamos hacer con las clases abstractas.

```
Teacher t;  // This is OK.
```

Esto es necesario para facilitar el polimorfismo. (siguiente sección)

7.19 INTERFACES AND POLYMORPHISM (INTERFACES Y POLIMORFISMO)

Veamos un ejemplo de polimorfismo aplicado a interfaces. Supondremos que

- Las clases Profesor y Alumno derivan de la clase Persona.

- Profesor y Alumno son clases hermanas, ninguna deriva de la otra.

- Persona implementa la interfaz Profesor y, por tanto, en virtud de la herencia, tanto Profesor como Alumno implementan implícitamente la interfaz Profesor.

Podemos declarar una colección que contenga referencias a Profesor, y luego llenarla con una mezcla de referencias a objetos Alumno y Profesor de la siguiente manera:

```
ArrayList<Teacher> teachers = new ArrayList<Teacher>();
teachers.add(new Student("Becky Elkins"));
teachers.add(new Professor("Bobby Cranston"));
// etc.
```

Podemos iterar a través de la colección de profesores de forma polimórfica, refiriéndonos a todos sus elementos como Profesores:

```
for (Teacher t : teachers) {
    // This line of code is polymorphic.
    t.agreeToTeach(c);
}
```

Porque restringimos la colección para que sólo contuviera referencias a objetos de tipo Maestro cuando la declaramos por primera vez.

7.20 THE IMPORTANCE OF INTERFACES (LA IMPORTANCIA DE LAS INTERFACES)

Las interfaces son una de las características más poco entendidas y, por lo tanto, infrautilizadas, de los lenguajes de programación OO que las admiten. Esto es bastante desafortunado, ya que las interfaces son extremadamente potentes si se usan correctamente.

Siempre que sea posible/viable al diseñar clases, si usamos tipos de interfaz en lugar de tipos de clase específicos al declarar:

• Atributos (privados)

• Parámetros formales a métodos

• Tipos de métodos de retorno

Nuestras clases serán más flexibles en términos de cómo el código del cliente puede usarlas.

El lenguaje Java proporciona muchas interfaces predefinidas. Un ejemplo es la interfaz de colección del paquete java.util. La interfaz de colección impone la implementación de 14 métodos, muchos de los cuales (add, addAll, clear, contains, isEmpty, remove, size, etc.) que se discutieron cuando se habló de varias clases de colección en el Capítulo 6. Estos 14 métodos definen colectivamente los servicios que un objeto tiene que ser capaz de proporcionar para desempeñar el papel de una colección adecuada en una aplicación Java.

La interfaz de la colección está implementada por numerosas clases de Java Collection predefinidas, incluida la clase ArrayList. De hecho, el marco de colecciones, introducido con la versión 1.2 de Java (también conocida como Java 2), se basa en un total de seis interfaces: Map, SortedMap, Collection, Set, List y SortedSet. Las relaciones entre estas interfaces y las diversas clases de colección se ilustran en la Figura 7-6.

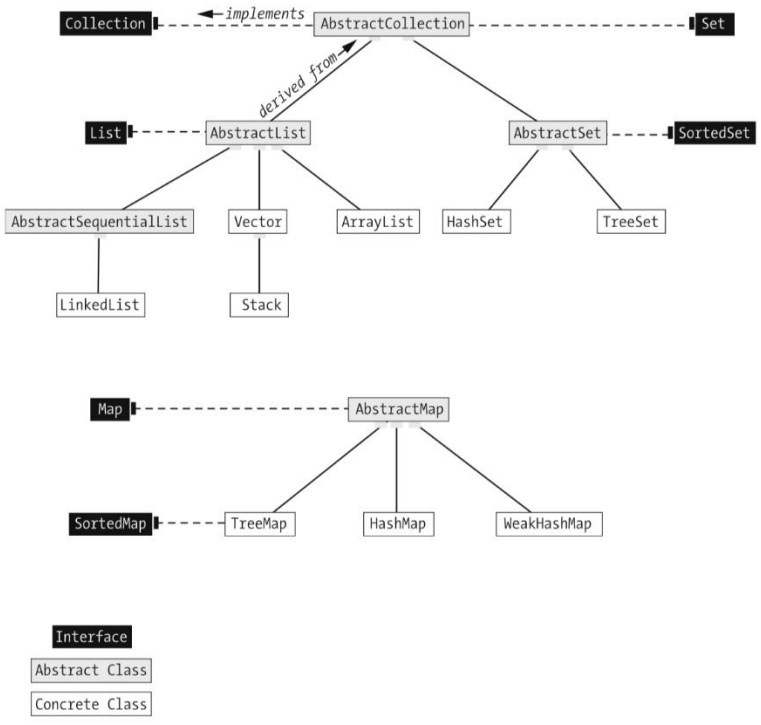


Figure 7-6. *The "family tree" of Java's predefined collection classes*

Si diseñamos métodos que van a operar en colecciones de objetos para aceptar una referencia genérica de colección como argumento (en lugar de exigir que se pase un tipo específico de colección), dichos métodos serán mucho más versátiles; el código del cliente será libre de pasar en cualquier tipo de colección que desee.

Lo mismo ocurre con los métodos que devuelven colecciones de objetos: si los diseñamos para devolver colecciones genéricas en lugar de tipos de colección específicos, entonces somos libres de cambiar los detalles internos de qué tipo de colección estamos elaborando.

En el capítulo 6, nuestra discusión sobre la creación de colecciones personalizadas comenzó mencionando que podríamos, si lo deseábamos, inventar un tipo de colección personalizado desde cero, pero que el lenguaje Java proporciona tantos tipos de colección predefinidos que no suele ser necesario hacerlo. Sin embargo, si alguna vez quiere inventar un nuevo tipo de colección sin extender uno de los tipos de colección predefinidos, asegúrese de que su tipo de colección implementa la interfaz de colección predefinida, como mínimo:

```
import java.util.Collection;

public class MyBrandNewCollectionType implements Collection { ... }
```

Para que su tipo de colección se pueda utilizar en cualquier contexto en el que se requiera una colección genérica.

APENDIX F: How Polymorphism Works Behind the Scenes (Static vs. Dynamic Binding)

Cuando ejecutamos un constructor para crear un objeto, la memoria usada para contener ese objeto se produce cuando se ejecuta el programa o sea en tiempo de ejecución, eso se conoce como instanciación dinámica, esto está relacionado con el overriding que vimos en resúmenes anteriores.

Por otro lado tenemos el enlace estático o en tiempo de compilación que está relacionado con el overloading

Static Binding (**Enlace estático)**

Consiste en que toda variable y método que se use debe si o si estar declarado por lo que si no lo está habrá errores, aquí un ejemplo

```
int x;
int y;

x = y + z;  // Problem!  z was not declared, and so the compiler
            // will object.
```

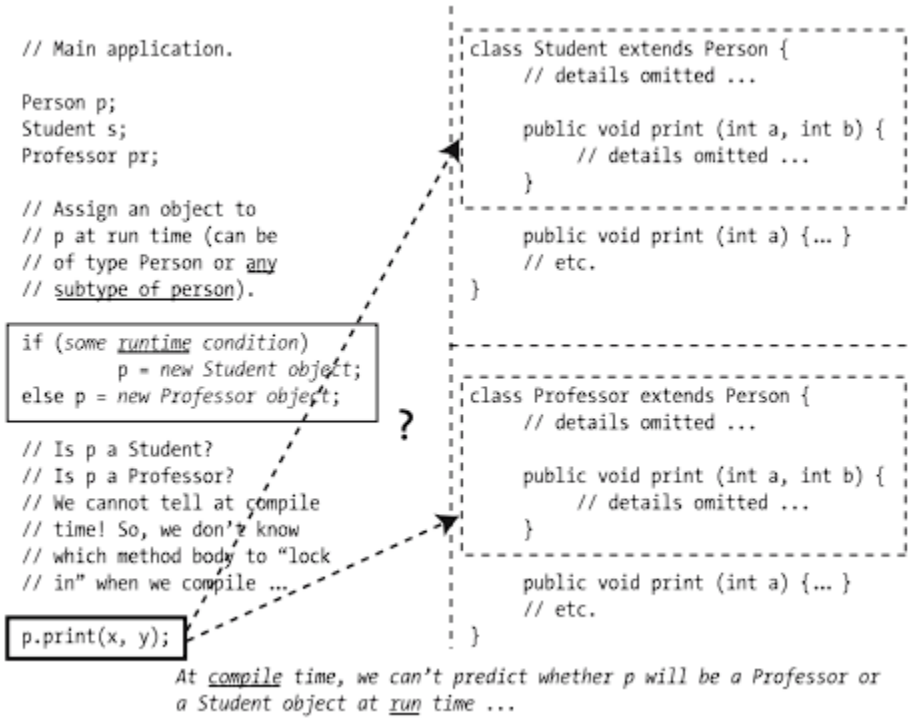
Y con métodos también, si el programa sabe que método está siendo usado y está definido todo saldrá bien

```
int x;
int y;
Student s;

// Details omitted.

s.print(x, y);   // Compiler will look for a method belonging
                    // to class Student whose name is "print" and
                    // whose argument signature calls for two int(egers).
```

Dynamic Binding (**Enlace dinámico)**



Miren que aquí hay dos clases extendidas de personas, una que será profesor y otra estudiante, y en el programa principal hay una condición en la que se creará un objeto estudiante o profesor dependiente de la condición y luego se acudirá a el print de esa clase resultante, entonces todo esto está pasando en tiempo de ejecución. El enlace dinámico permite el polimorfismo permitiendo así la flexibilidad.

Table 7-1. *Syntactical Differences for Declaring Abstract Classes vs. Interfaces*

Example Using an Abstract Class	**Example Using an Interface**
Declaring the `Teacher` **Type As an Abstract Class**	**Declaring the** `Teacher` **Type As an Interface**

`public abstract class Teacher {`
`  // Abstract classes may prescribe`
`  // data structure.`
`  private String name;`
`  private String employeeId;`
`  // etc.`
 	
`  // We declare abstract methods using`
`  // the "abstract" keyword; these`
`  // must also be declared "public".`
`  public abstract void agreeToTeach(`
`  Course c);`
 	
`  // etc.`
 	
`  // Abstract classes may also declare`
`  // concrete methods.`
`  public void print() {`
`  System.out.println(name);`
`  }`
 	
`  // etc.`
`}`
	`public interface Teacher {`
	`  // Interfaces may NOT prescribe`
	`  // data structure.`
	
	`  // We needn't use the "public" or`
	`  // "abstract" keywords - all methods`
	`  // declared by an interface are`
	`  // automatically public and`
	`  // abstract by default.`
	`  void agreeToTeach(Course c);`
	
	`  // etc.`
	
	`  // Interfaces may NOT declare`
	`  // concrete methods.`
	`}`

Table 7-2. *Syntactical Differences for Extending Abstract Classes vs. Implementing Interfaces*

Example Using an Abstract Class	**Example Using an Interface**
`Professor` ***Extends*** `Teacher`	`Professor` ***Implements*** `Teacher`

`public class Professor extends Teacher {`
`  // Professor inherits attributes, if`
`  // any, from the abstract`
`  // superclass, and optionally`
`  // adds additional attributes.`
`  private Department worksFor;`
`  // etc.`
 	
`  // We override abstract methods`
`  // inherited from the Teacher class`
`  // to provide a concrete`
`  // implementation.`
 	
`  public void agreeToTeach(Course c) {`
`  *logic for the method body goes*`
`  *here; details omitted ...*`
`  }`
 	
`  // etc. for other abstract methods.`
 	
`  // Additional methods may be added;`
`  // details omitted.`
`}`
	`public class Professor implements`
	`  Teacher {`
	`  // Professor must provide ALL of`
	`  // its own data structure, as an`
	`  // interface cannot provide this.`
	`  private String name;`
	`  private String employeeId;`
	`  private Department worksFor;`
	`  // etc.`
	
	`  // We implement methods required by`
	`  // the Teacher interface.`
	
	`  public void agreeToTeach(Course c) {`
	`  *logic for the method body goes*`
	`  *here; details omitted ...*`
	`  }`
	
	`  // etc. for other abstract methods.`
	
	`  // Additional methods may be added;`
	`  // details omitted.`
	`}`