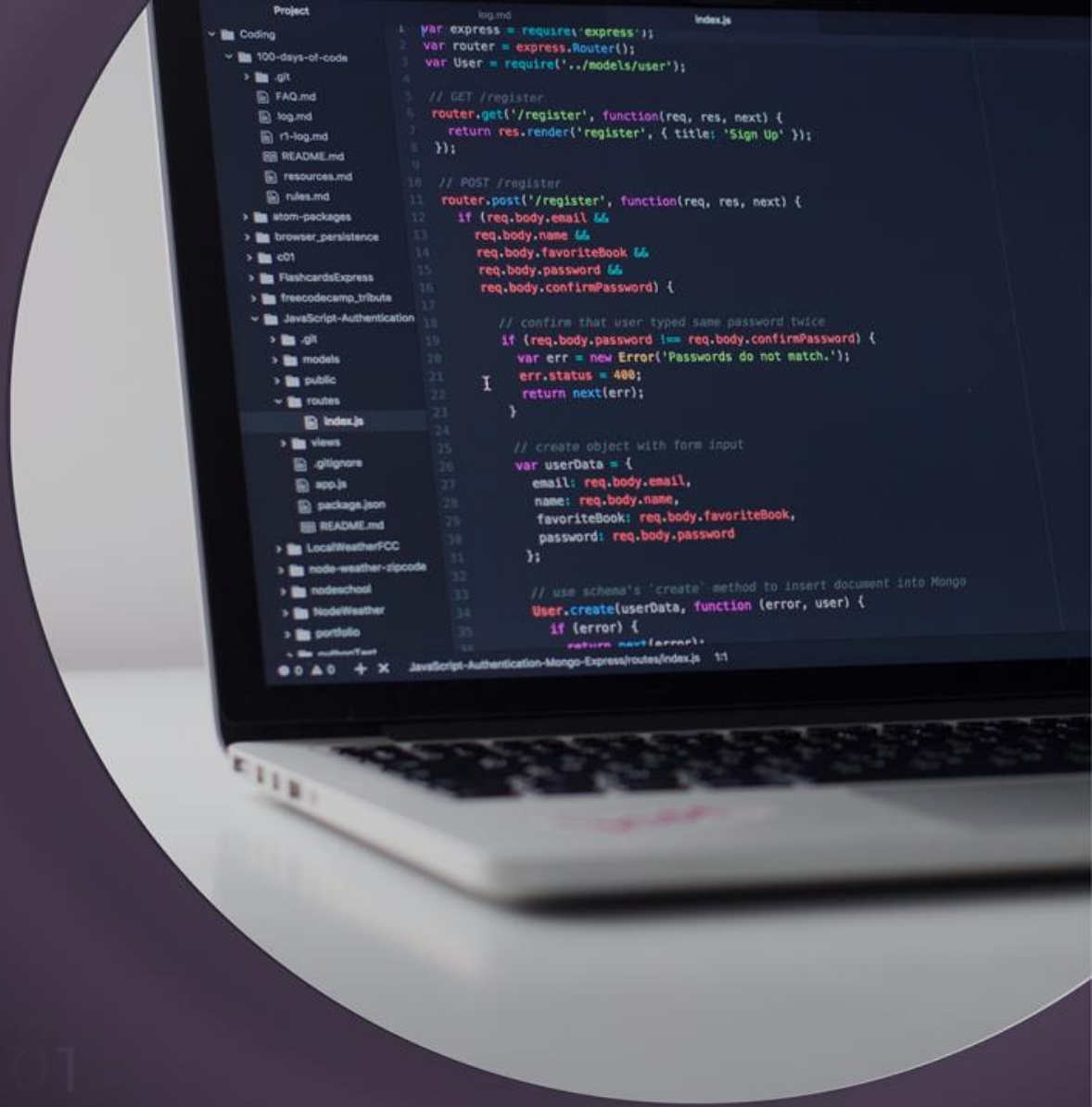
El futuro digital es de todos

MinTIC

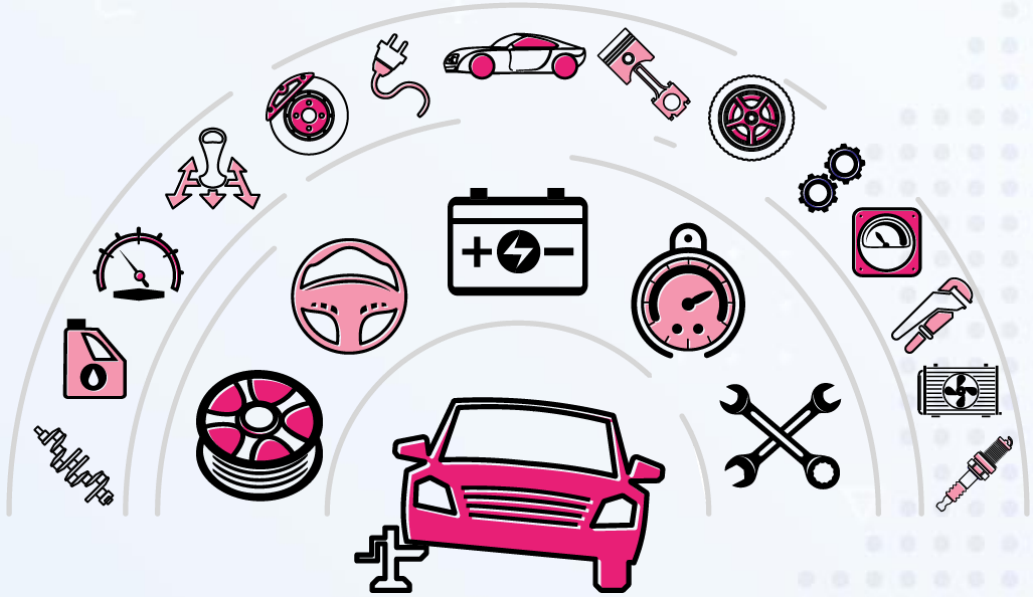


PROGRAMACIÓN
ORIENTADA A OBJETOS Y UML

Universidad Industrial de Santander

MisionTIC 2022

2.1. Introducción

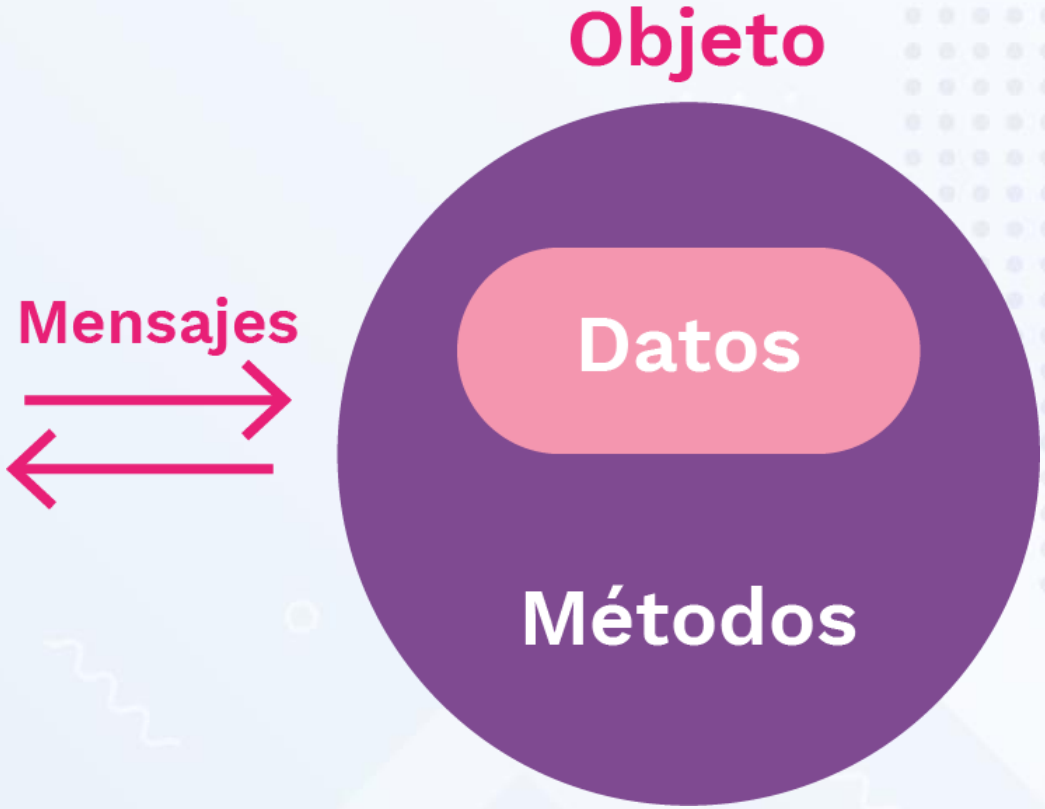


La programación orientada a objetos es un paradigma de programación, en donde se hacen uso de objetos, los cuales son datos abstractos que contienen características inherentes de estos. Se puede decir que un objeto en programación, es una representación de un objeto real, que posee características propias. Este paradigma o modelo permite, mediante cierta analogía, poder representar objetos reales, que podemos encontrar en nuestro diario vivir, por medio de los objetos en determinado lenguaje de programación. Para un mejor entendimiento pensemos en un vehículo, el cual está conformado por un conjunto de partes que se requieren para su buen funcionamiento.

Cada una de estas partes puede pertenecer ya sea al sistema de frenado, al sistema de suspensión, al sistema de eléctrico-electrónico, sistema de refrigeración etc. Estos sistemas a su vez,cumplen una función específica, que gracias a cada parte que lo compone, permite el buen funcionamiento del vehículo. Por ejemplo, el sistema de frenado permite disminuir de manera progresiva la velocidad del vehículo o detenerlo. Como podemos observar, en la vida real tenemos objetos, y estos se encuentran bien organizados por ciertos componentes, los cuales poseen determinadas funciones.

En la programación orientada a objetos, se busca representar objetos con determinadas características, los cuales son creados a partir del uso de clases. Una clase en programación es una plantilla que contiene una estructura de datos que se identifica a través de un nombre, el cual se recomienda que vaya acorde a lo que ella representa. Dentro de esta clase van todas las características y funcionalidades que son propias del objeto que queremos representar con dicha clase.

Por medio de las clases es que se pueden en programación, construir objetos, y un conjunto de estos objetos que se comunican entre sí a través de mensajes, son los que conforman un programa. La programación orientada a objetos es la base de la programación en Java, y dentro de esta, el uso de clases y objetos es de gran importancia para la construcción de programas.



Los mecanismos básicos de la programación orientada a objetos son (ver Fig. 2):

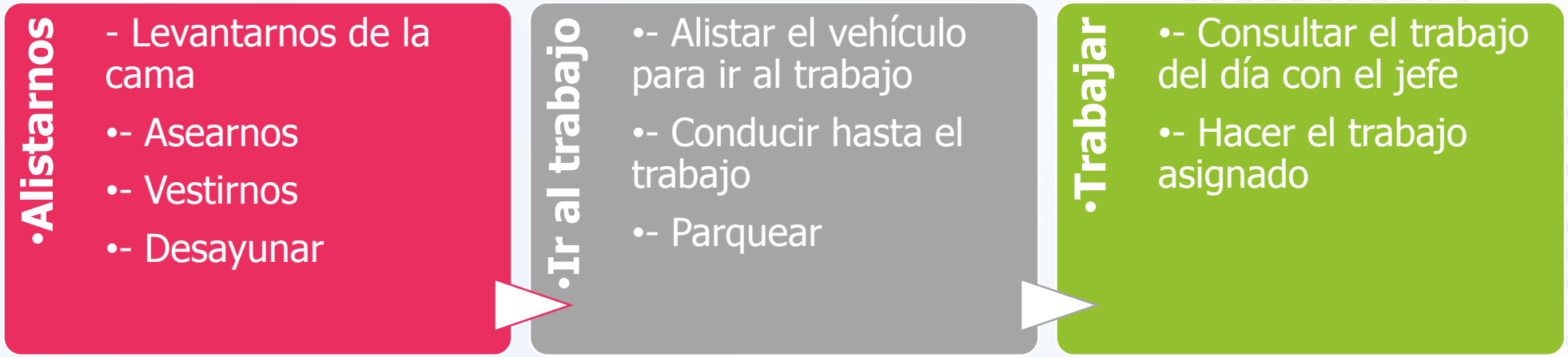
- Objetos: Son los componentes principales de un programa orientado a objetos. Estos se pueden ver como datos abstractos compuestos de datos más simples y de procedimientos o funciones para la manipulación de estos.

- Mensajes: Un programa orientado a objetos al ejecutarse, cada uno de los objetos que lo componen, reciben, interpretan y responden a mensajes que envían otros objetos. Los mensajes son la manera a través de la cual se comunican los objetos en un lenguaje de programación orientado a objetos.

- Métodos: Los métodos son implementaciones o funciones pertenecientes a una clase determinada, los cuales son accesibles mediante los objetos. A través de los métodos, se pueden modificar los atributos de los objetos pertenecientes a determinada clase. Los métodos determinan el comportamiento de los objetos cuando reciben un mensaje, pero estos también permiten enviar mensajes a otros objetos solicitando algún tipo de información o sugiriendo alguna acción.

Como una analogía a una clase podemos pensar en la lista de actividades que realizamos a diario

Esta lista la podemos organizar en grupos



Cada grupo es una Clase con sus distintos métodos, los cuales pueden ser modificados sin alterar las otras actividades. Si por ejemplo, quisiéramos cambiar dentro del grupo "Ir al trabajo" el método o la manera en cómo nos desplazamos al trabajo (ej. taxi, bus, bicicleta, moto), lo podríamos hacer sin inconvenientes. De esta manera, los sistemas de software están estructurados, y a partir de este tipo de estructuras organizadas es que se trabaja, de tal manera que cada una de las partes que lo componen, sea cada vez más entendible.

Como se mencionó anteriormente, los objetos son los actores principales de un programa dentro de la programación orientada a objetos. Al momento de ejecutar un programa orientado a objetos, se crean los objetos que se requieren, se envían mensajes entre los objetos y la información es procesada de manera interna. Por último, cuando los objetos cumplen con su función, y ya no se requiere el uso de ellos, son eliminados, es decir, la memoria asignada a cada uno de ellos es liberada.

Como características de la programación orientada a objeto se tiene:

- ***Abstracción***: La abstracción es una característica que permite enfocarnos en la estructura externa de los objetos y no en sus detalles, y se puede definir como las características que identifican un objeto y lo diferencian de otros.

- ***Encapsulamiento***: El encapsulamiento es una característica que visualiza al objeto como una caja negra, sabemos que hay una información dentro del objeto, pero no sabemos cómo está organizada esa información.

- ***Herencia:*** La herencia es una característica que permite compartir métodos y datos entre clases y subclases del objeto.

Polimorfismo: El polimorfismo es una característica que hace que se puedan hacer implementación de múltiples formas de un método dentro de una clase. Es decir distintos métodos dentro de la clase al cual se accede con un mismo nombre.

Todos los lenguajes que implementan parte o la totalidad de las características y los conceptos anteriormente descritos, son llamados lenguajes orientados a objetos. Este tipo de lenguajes se caracterizan porque permiten la definición de nuevos tipos de datos y de operaciones que se pueden realizar sobre estos. Algunos ejemplos de lenguajes orientados a objetos son: ADA, C++, Objective C, Java, Ruby, Python, Perl, PHP, C#, Kotlin, Visual Basic .NET, entre otros.

Dentro de todo este concepto de programación, existe además el paradigma de programación orientada a eventos. Dentro de este paradigma, la estructura del programa y su ejecución, los determina el usuario y no el programador al momento de desarrollar el programa. Como su nombre lo indica, este tipo de paradigma se basa en la construcción de programas a partir de una serie de eventos, los cuales tienen lugar dentro de la ejecución del programa en el momento en que el usuario decida hacer uso de ellos.

En un programa secuencial, el programador decide que se va a ejecutar y cómo se va a ejecutar para que la tarea que se desea resolver se resuelva y se obtenga el resultado esperado. Este tipo de programación es la base del desarrollo de interfaces de usuario.

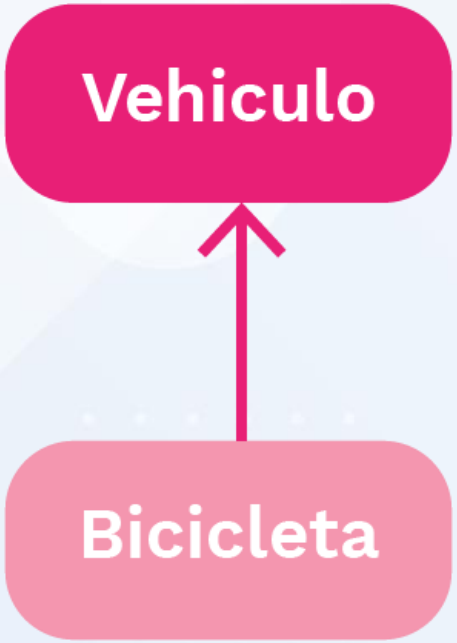
Podemos imaginarnos como ejemplo, al momento de usar un programa de edición de texto, o un programa de edición de imágenes, cada uno de ellos tiene una interfaz de usuario y se encuentra programado de tal manera que cada botón que compone esa interfaz viene con una función específica o evento. La programación orientada a eventos permite interactuar con el usuario a lo largo de todo el tiempo en que se esté ejecutando el programa; por su parte, el comportamiento del programa va a depender del uso que le de el usuario y la manera en que este haga uso de cada uno de los eventos que lo compone.

Quizá en el uso del programa para edición de imágenes, un usuario quiera abrir una imagen, aplicar un filtro, recortar la imagen y guardarla, pero otro usuario simplemente quiera abrir una imagen, modificar un poco los niveles de colores y guardarla. Los programas desarrollados bajo este paradigma están constituidos por un bucle exterior, cuya función es la de recolectar los eventos solicitados por el usuario. Este bucle, que por lo general está oculto al programador, tiene la siguiente forma:

```
while(true) {
    switch(event) {
        case ctrl_s:
        case key_up:
        case mouse_click:
        case default:
    }
}
```

Como se puede observar, hay diferentes "interruptores" dentro de un bucle while, cada uno de estos "interruptores" es activado por el usuario mediante una orden que puede ser pasada usando cualquier periférico de una computadora o dispositivo electrónico. Estos "interruptores" pueden contener uno o varios eventos, y el usuario es quien decide el orden en que son ejecutados y cuándo deben ser ejecutados. Como ejemplo, si tenemos un editor de texto y queremos guardar la información que hemos editado, lo que se puede hacer es llamar al evento guardar mediante la combinación de teclas CTRL + S que inmediatamente envía la orden de guardar al programa. El programa toma el objeto que representa el documento como tal y almacena la información en disco en determinado formato.

2.2. Lenguaje Unificado de Modelado – UML.



El lenguaje unificado de modelado o UML es un lenguaje de modelado de desarrollo que permite estandarizar la manera de visualizar el diseño de un sistema de software o aplicativo. El UML fue desarrollado por Rational Software entre 1994 y 1995, y hoy en día es manejado y adoptado como un estándar por el Object Management Group (OMG). El UML puede ser usado en los siguientes niveles o etapas del desarrollo:

- ***Conceptual***: dentro del diseño en la etapa del problema, donde no existe una conexión fuerte con la codificación. En esta etapa, los diagramas de UML están más relacionados con el lenguaje humano.

- ***Especificación***: dentro de un diseño de software propuesto, donde existe una conexión fuerte con la codificación, ya que se pretende que los diagramas se conviertan en código.

- ***Implementación***: dentro de una implementación de software ya desarrollada, donde existe una conexión fuerte con la codificación. En esta etapa, los diagramas deben seguir ciertas reglas y semántica.

Podemos representar por medio de diagramas cualquier cosa, como por ejemplo, si consideramos la siguiente oración, "Una bicicleta es un vehículo", esta se puede representar como se muestra en la Fig. 4.

En el diagrama de la figura se indica que la bicicleta pertenece al conjunto de los vehículos, lo que quiere decir, que una bicicleta es un caso particular de vehículos.

Las características de un UML son:

- **Visualizando:** Quizá alguno de nosotros haya necesitado antes de resolver un problema hacer una especie de garabato en una hoja de papel para tener una guía de cómo resolver el problema. De la misma manera, los desarrolladores requieren guías visuales para la solución de problemas complejos o para tener una trazabilidad del proceso de desarrollo de una aplicación. Este ejercicio de primero generar una ficha gráfica de la situación a la cual nos enfrentamos, facilita la comprensión y la solución del problema. Un UML es un lenguaje formal y estandarizado que permite llevar a otro nivel estas ayudas gráficas, de tal manera que cualquiera que sepa este lenguaje, pueda entender lo que se quiere transmitir.

- Especificando: El mecanismo de comunicación para transmitir las ideas debe ser precisa y común. Un UML permite de manera fácil y correcta transmitir las especificaciones requeridas.

- Construyendo: Por la simple razón de que el UML es un lenguaje formal compuesto por determinadas reglas y una sintaxis fija, se pueden construir herramientas que interpreten y mapean nuestros modelos a cualquier lenguaje de programación.

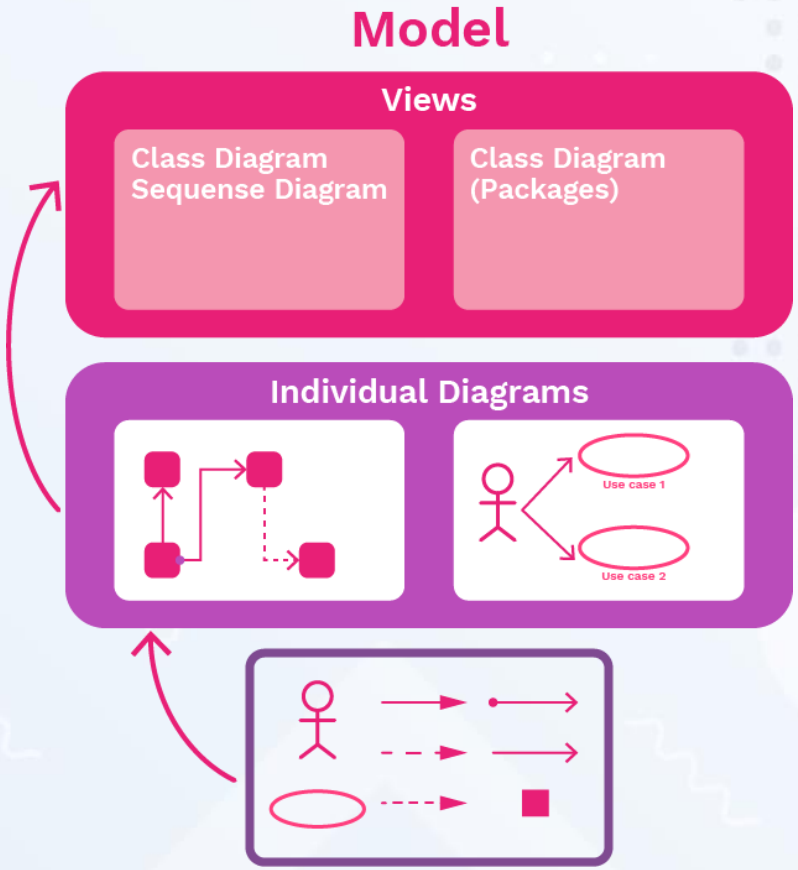
- Documentando: El uso correcto del UML produce como resultado una serie de documentación, la cual puede ser útil como modelo de nuestro sistema de software.

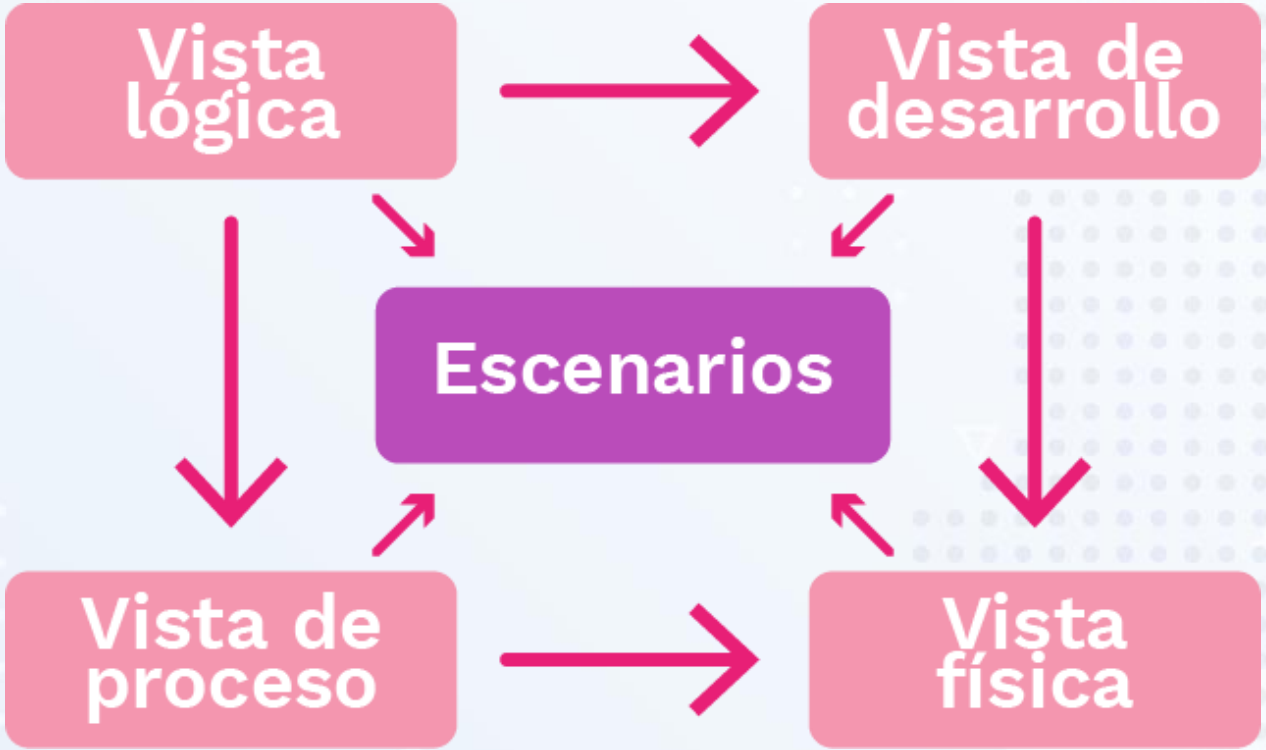
Para poder implementar el UML en un proyecto de software, primeramente se debe saber que existe un esquema jerárquico para la construcción de los modelos. Los modelos se encuentran compuestos de las vistas de nuestro sistema de software, las cuales están conformadas por diagramas individuales, que a su vez están compuestos de ciertos elementos fundamentales (ver Fig. 5).

Los diagramas más comunes son los diagramas de clase, los cuales describen la relación estructural entre las clases que componen nuestro sistema. La combinación de un diagrama de clase y otro diagrama que comunique la dinámica del sistema, es a lo que se conoce como una vista, las cuales describen de manera particular el sistema.

Un modelo para describir la arquitectura del sistema de software es el modelo de vistas 4+1, diseñado por Philippe Kruchten (ver Fig. 6). Se pueden crear modelos completos de nuestro sistema o aplicación, los cuales están compuestos por múltiples vistas, las cuales representan el sistema desde perspectivas distintas, cada una de ellas relacionadas con diferentes individuos asociados con la iniciativa de desarrollo.

La vista lógica tiene que ver con la funcionalidad de la aplicación y está relacionada con los diseñadores y el usuario final o consumidor. La vista de desarrollo está asociada con la gestión de software y está relacionada con los desarrolladores. La vista del proceso está relacionada con el rendimiento, la escalabilidad y el desempeño del sistema , así como con los integradores del sistema. La vista física se relaciona con la topología del sistema, entrega, instalación y telecomunicación y está relacionada con los ingenieros del sistema.




Vista lógica
Vista de desarrollo
Escenarios
Vista de proceso
Vista física

Los diagramas centrales del UML

Los diagramas son estructuras conformadas por elementos fundamentales, y es a partir de una combinación de estos diagramas, que se pueden formar los modelos que son vistos como una combinación de vistas dentro nuestro sistema. Existen diferentes tipos de diagramas en un UML, los cuales son:

- Diagrama de clase
- Diagrama de objeto
- Diagrama de componente
- Diagrama de despliegue
- Diagrama de caso de uso
- Diagrama de estado
- Diagrama de actividad
- Diagrama de secuencia
- Diagrama de colaboración

Estos diagramas pertenecen a dos categorías, se pueden considerar ya sea como diagramas estructurales (aspectos estáticos del sistema) o como diagramas de comportamiento (aspectos dinámicos del sistema).

Diagramas de comportamiento

Son diagramas que se enfocan en aspectos del sistema que contribuyen a satisfacer los requerimientos del mismo. Los diagramas de comportamiento son:

•Diagrama de caso de uso: Permite modelar el núcleo de un sistema. Muestra una serie de actores del sistema y casos de uso.

•Diagrama de actividad: Modela el flujo de una actividad.

•Diagrama de estado: Ilustra el comportamiento interno relacionado con el estado de un objeto.

•Diagrama de secuencia: Es un diagrama de interacción que describe el orden temporal de los mensajes que se envían entre objetos.

•Diagrama de colaboración: Es un diagrama de interacción que describe el diseño organizativo de los objetos que reciben y envían mensajes.

Diagramas estructurales

Este tipo de diagramas se centran en mostrar los aspectos estáticos de nuestro sistema. Dentro de esta categoría encontramos los siguientes diagramas:

•Diagrama de clase: Permite ilustrar las clases y la relación que hay entre ellas dentro del sistema.

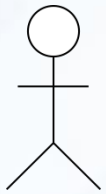
•Diagrama de objeto: Permite mostrar de manera estática una vista del sistema mostrando la relación que existe entre objetos en ese estado.

•Diagrama de componente: Permite mostrar de manera estática la relación que existe entre las componentes del software desplegable.

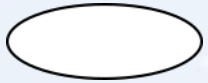
•Diagrama de despliegue: Describe las componentes físicas del sistema.

Elementos fundamentales

Algunos de los elementos fundamentales que se usan para la construcción de los diagramas y las vistas son:



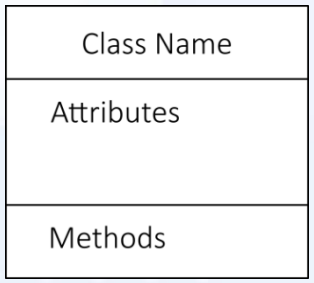
Actor: Este elemento representa un rol que juega el usuario final del sistema de software.



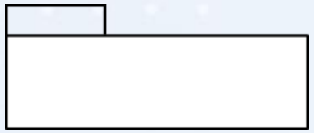
Caso de uso: Elemento que representa las acciones que lleva a cabo el actor sobre el sistema de software.

Colaboración: Permite crear diagramas de clases que trabajan en conjunto unas con otras.

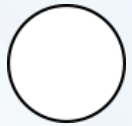
Objeto: Es una instancia de una clase. Este elemento es un rectángulo en donde se puede especificar el nombre de la clase a la cual pertenece el objeto (subrayado y seguido de punto y coma), o se puede simplemente colocar el nombre del objeto (subrayado), o ambos (subrayado).



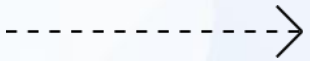
Clase: Es una plantilla para la creación de objetos. Este elemento se compone de tres compartimientos, el primero representa el nombre de la clase, el segundo, los atributos de la clase, y el tercero, los métodos que contiene la clase. Los atributos y los métodos pueden ir adornados con un signo (+), un signo (-) o un signo (#), lo que indica que tales son públicos, privados o protegidos, respectivamente. Si el atributo o el método está subrayado esto indica que es estático.



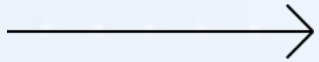
Paquete: Este elemento puede contener cualquier otro tipo de elemento, se puede traducir directamente como un paquete en Java. Dentro de este elemento va el nombre que identifica el paquete



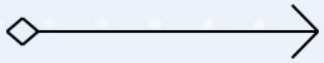
Interfaz: Una interfaz es una lista o colección de acciones que posee una clase. Se traduce directamente a interfaces en Java. Este elemento se representa por un Ícono que es un círculo o por un diagrama de clase con el agregado <<interface>>.



Dependencia: Se usa para representar una relación de uso entre entidades (clases, objetos, paquetes, interfaz).

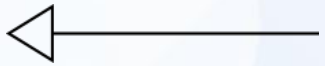


Asociación: Representa la relación estructural entre entidades.

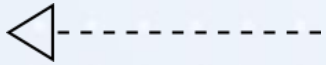


Agregación: Es una forma de asociación que representa parte o toda la relación que hay entre dos clases.

Composición: Es una forma especial de agregación.



Generalización: Permite ilustrar la relación existente entre un elemento más general y un elemento más específico. Esto permite modelar la herencia.

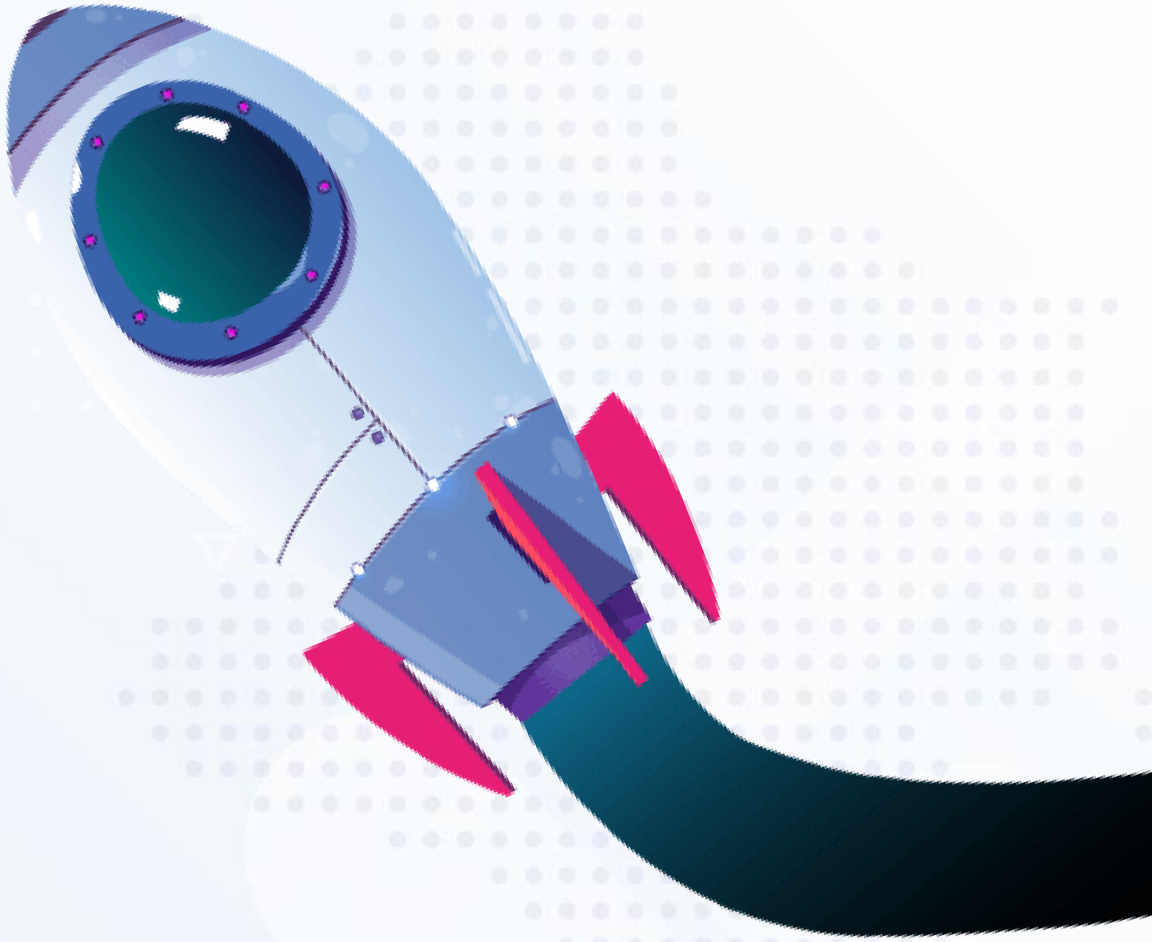


Realización: Muestra la relación que especifica un contrato entre dos entidades. Una de las entidades define un contrato que es garantizado por la otra.

Anotaciones: Permite incluir notas relacionadas con una explicación más amplia de los elementos y diagramas del UML.

Material complementario

https://www.youtube.com/watch?v=Z0yLerU0g-Q



2.3. Objetos y clases: Conceptos, diseño (notaciones), implementación.

Los objetos y las clases son la base de la programación orientada a objetos. Los objetos son instancias de una clase, y las clases son prototipos dentro de los cuales se definen atributos y métodos, por medio de las cuales se construyen los objetos. Los atributos son variables o datos definidos dentro de la clase, los cuales la caracterizan. Los métodos son funciones o subrutinas que representan acciones que operan sobre los atributos o datos de una clase, es a través de estos métodos por medio del cual, los objetos se comunican entre sí. Los métodos les permiten a los objetos enviar y recibir mensajes hacia o desde otros objetos.

Clases

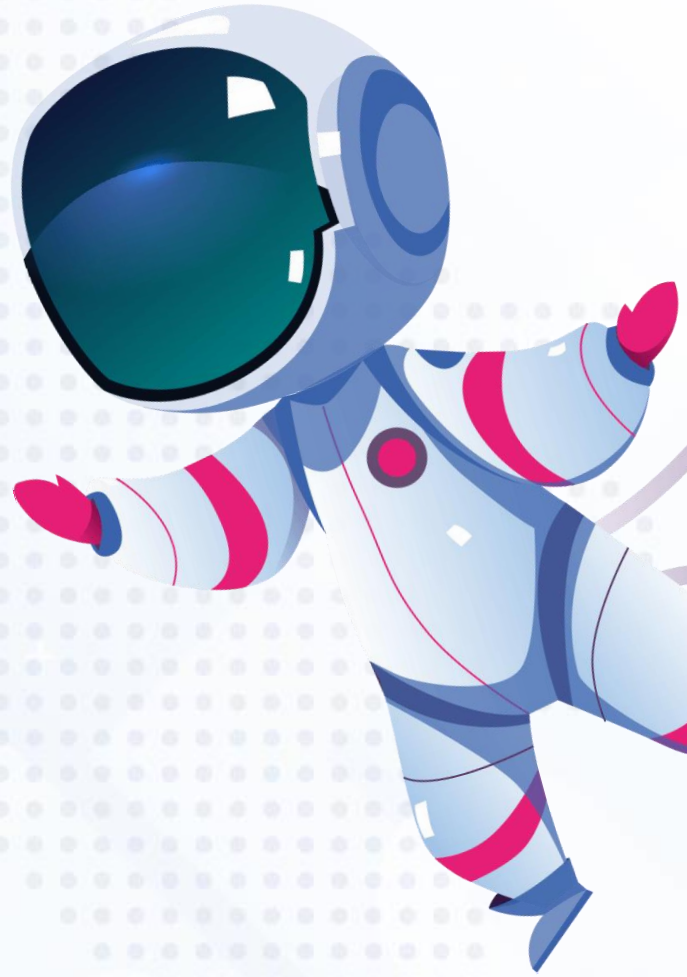
Las clases en Java son almacenadas en ficheros con extensión .java, y tienen una estructura bien definida. Las clases en Java están compuestas por una declaración del paquete al cual pertenece la clase. Seguido de la declaración del paquete, va el importe de paquetes externos, seguido de los comentarios. Luego de los comentarios, viene la definición de la clase que se hace mediante la palabra reservada class. Luego de definir la clase, son ubicadas las variables o atributos de la clase. Y seguido de esto, van los constructores de la clase, donde van ubicadas las características iniciales que se deseen para los objetos que se deriven de la clase

Por último, van los métodos o funciones. A continuación, se muestra un esquema de las componentes de una clase:

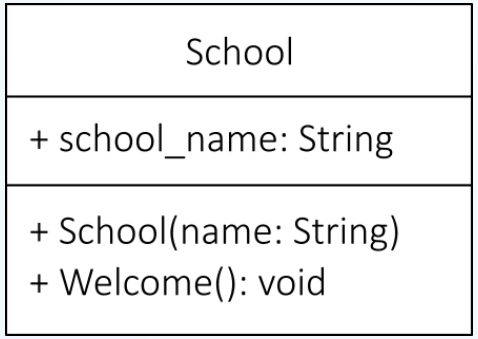
```
paquetes
importe de paquetes externos
comentarios
definición de la clase
    constantes
    atributos
    constructores
    metodos
```

Un ejemplo de una clase en Java, llamada School, es el siguiente

```
package school;

/**
*
* @author
*/
public class School {
    public String school_name;
    public School(String name) {
        this.school_name = name;
    }
    public void Welcome() {
        System.out.print("Welcome: " + this.school_name + "\n");
   }
}
```

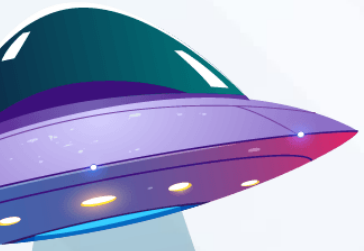


Donde su representación por medio de UML es de la siguiente manera:



A partir del ejemplo anterior, podemos identificar cada una de las componentes. Al inicio de la clase podemos observar el paquete al cual pertenecen, el paquete school. Seguido se puede observar un bloque de líneas de comentarios. Luego se define la clase, llamada School, usando la palabra reservada class.

Dentro de la clase, al inicio, se encuentran sus atributos. Podemos observar un atributo llamado `school_name` de tipo String. Se encuentra también el constructor, que lleva como nombre el mismo nombre de la clase, y que puede o no recibir parámetros de entrada. En este caso, el constructor recibe un parámetro de entrada llamado name de tipo String. Los constructores son usados para la creación de objetos, y su función es la de inicializar los atributos para dichos objetos, en este ejemplo el atributo `school_name`.



Adicionalmente, una clase puede tener métodos, los cuales son funciones que contienen ciertas acciones que operan sobre los objetos que se crean a partir de ella. En este ejemplo hay un solo método, llamado Welcome de tipo `void`, cuya función es la de visualizar un mensaje a la salida. Dentro de una clase, se puede definir un método llamado main, el cual es el punto inicial de todo programa en Java, su sintaxis es siempre `public static void(String[] args),` y recibe como parámetros de entradas argumentos almacenados en la variable `args`, que es de tipo arreglo de cadena de caracteres.

Para usar los atributos y métodos dentro de la misma clase a la cual pertenecen, se hace a través de la palabra reservada this, la cual hace referencia al objeto actual. Si se desea usar los atributos y métodos en otras clases, por lo general, se hace mediante la creación de objetos o instancias de la clase a la cual pertenecen esos atributos y métodos.

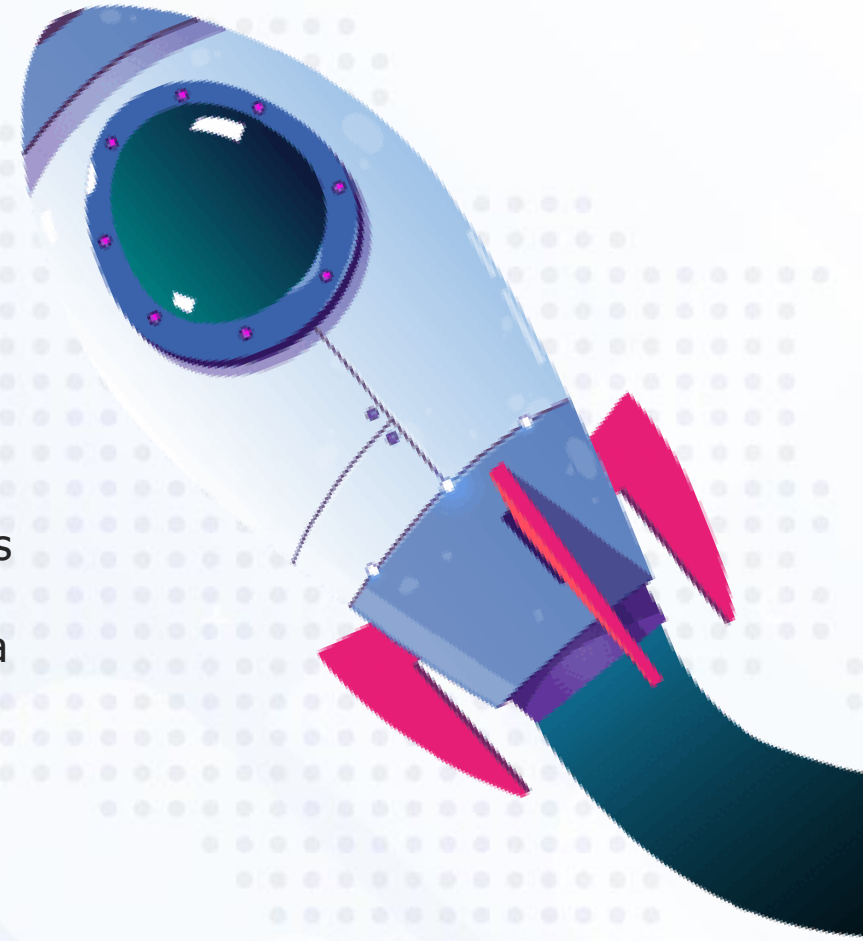
Objetos

Los objetos son instancias de una clase que se pueden ver como una abstracción de datos asociados a una dicha clase. Los objetos son creados usando la siguiente sintaxis:

```
Clase objeto = new Clase;
```

donde `Clase` es el nombre de la clase a partir de la cual se quiere crear el objeto y objeto el nombre que se le quiere dar al objeto. Para definir los objetos es usada la palabra reservada new seguida del nombre de la clase, indicando que se va a crear un nuevo objeto que pertenece a esa clase. Por ejemplo, para definir un objeto de la clase anterior se puede hacer de la siguiente manera:

```
School school = new School("School A");
```



Donde se puede ver que se crea el objeto `school` que pertenece a la clase `School`. En este caso, se indica que se crea un nuevo objeto a partir de la clase `School`, la cual recibe como parámetro de entrada un dato tipo `String`. Si se desea hacer uso de un atributo o un método definido dentro de esa misma clase, se puede hacer mediante el operador "." a través del objeto creado, esto es:

```
objeto.metodo();
```

Como por ejemplo, para el caso de la clase `School`, si se desea implementar el método `Welcome()` se procede de la siguiente manera:

```
school.Welcome();
```

lo que resultaría en la impresión del mensaje "`Welcome: School A`". De esa manera y con esta sintaxis, son creados y usados los objetos en Java.

Paquetes

Los códigos en Java están contenidos en paquetes, los cuales pueden verse como directorios. Y estos paquetes pueden estar organizados de manera jerárquica en nuestro sistema de software. Un ejemplo de la estructura de paquetes en un programa en Java es el siguiente:

School

- **Source Packages**
  - **school**
    - *School.java*
  - **user**
    - *User.java*

Donde se observa que el programa `School` tiene dos paquetes, uno llamado `school` y otro llamado `user`. Cada clase o un conjunto de estas, perteneciente a determinado paquete, son importadas mediante el uso de la palabra reservada `import` de la siguiente manera:

```
import path.java.package.Class;
```

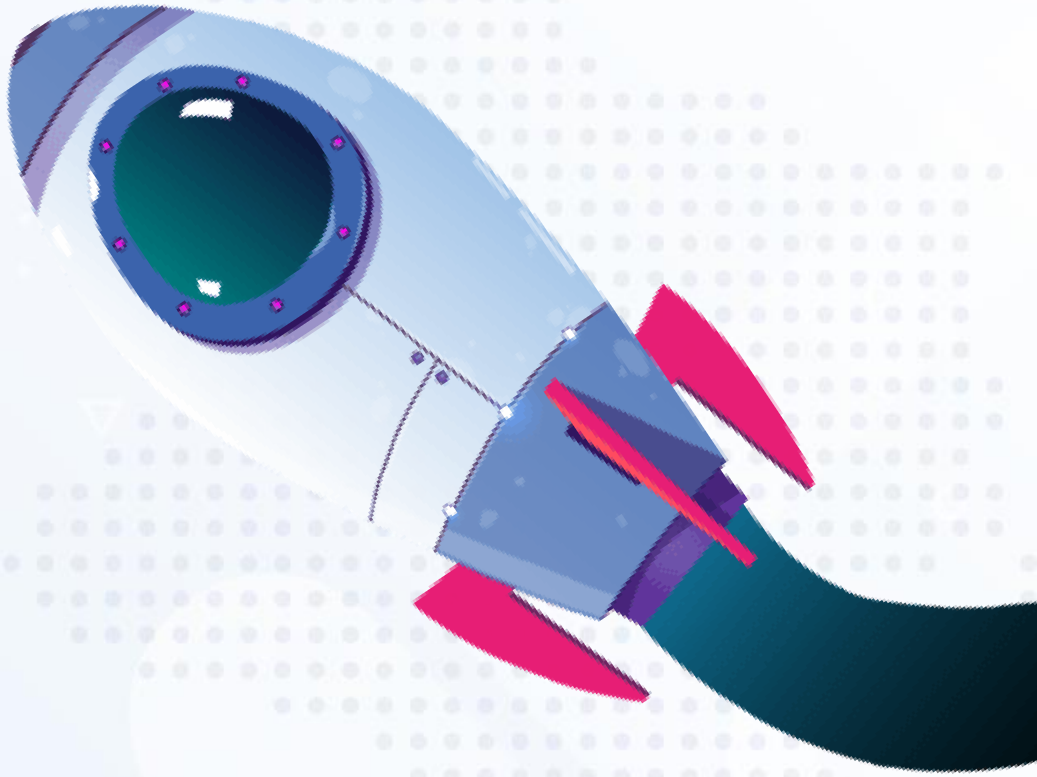
Donde `path.java.package` indica la jerarquía del paquete al cual se desea acceder y `Class` la clase que se desea importar y que pertenece a este paquete. Si lo que se desea es importar todas las clases que pertenecen a ese paquete, se usa de la siguiente manera:

```
import path.java.package.*;
```

Considerando la estructura de la clase `School`, anteriormente mostrada, se realizaron modificaciones en su estructura, de tal manera que quedó como se muestra a continuación:

Primera parte del código

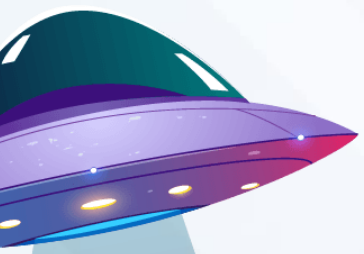
```
package school;
import user.User;
import java.util.List;
import java.util.ArrayList;
/**
*
* @author
*/
public class School {
    public String school_name;
    public int total_students;
    List<User> users = new ArrayList();
    public School(String name) {
        this.school_name = name;
        this.total_students = 0;
    }
```



Segunda parte del código

```
    public void Welcome() {
        System.out.print("Welcome: " + this.school_name + "\n");
    }
    public void save_user(User user) {
        this.users.add(user);
        this.total_students += 1;
        System.out.print("New user saved.\n");
    }
    public void students_list() {
        System.out.println("-----------------------------------");
        System.out.println("---------- USERS LIST ----------");
        System.out.println("-----------------------------------");
        users.forEach(user -> {
            user.info();
        });
        System.out.println("-----------------------------------");
    }
}
```

A partir de lo cual, queremos mostrar que se pueden realizar distintos tipos de importe. Tenemos un importe de un paquete local, como es el caso de la clase `User` del paquete `user`, y un importe de un paquete de java, como es el caso de las clases `List` y `ArrayList`, pertenecientes al paquete `java.util`.



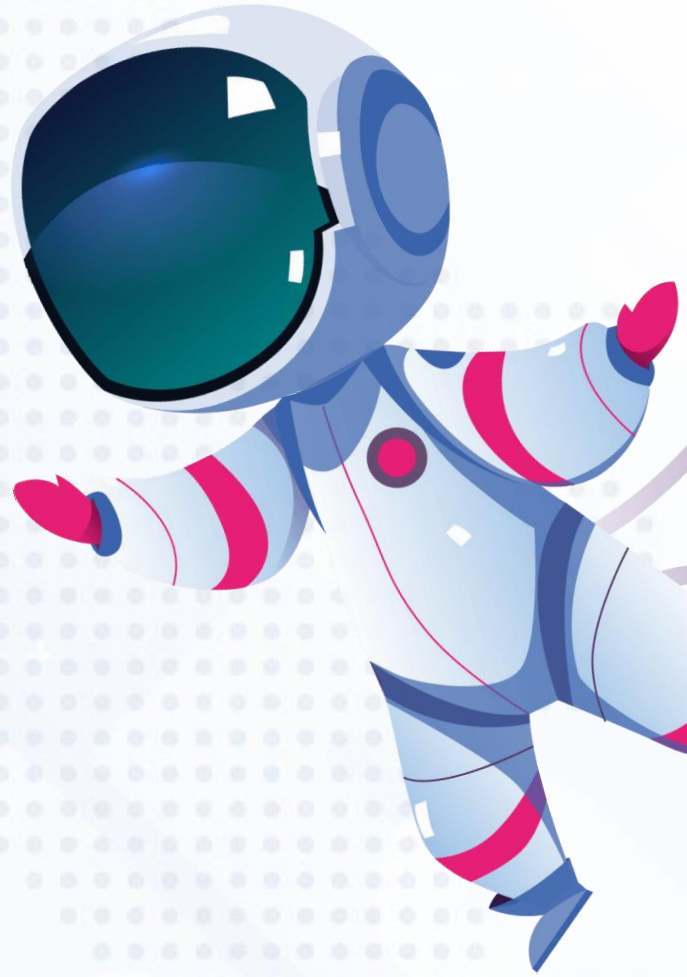
Adicionalmente, se agregan otros elementos que serán explicados más adelante.
La clase `User` del paquete `user` es una clase que tiene la siguiente forma:

Primera parte del código

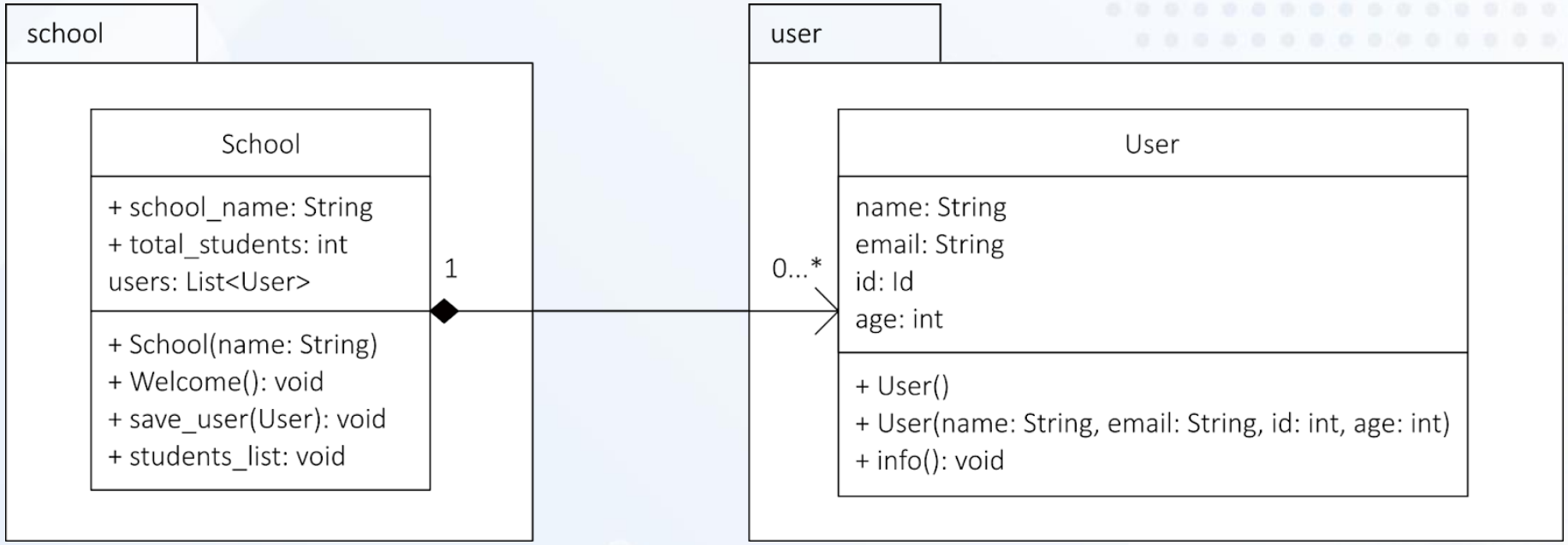
```
package user;

/**
*
* @author
*/
public class User {
    String name;
    String email;
    int id;
    int age;
    public User() {}
    public User(String name, String email, int id, int age) {
        this.name = name;
        this.email = email;
        this.id = id;
        this.age = age;
```

Segunda parte del código

```
    }
    public void info() {
        System.out.println("--------------------------------");
        System.out.println("----------- USUARIO ------------");
        System.out.println("--------------------------------");
        System.out.println("Full name: " + this.name);
        System.out.println("Email: " + this.email);
        System.out.println("ID: " + this.id);
        System.out.println("Age: " + this.age);
      System.out.println("--------------------------------");
    }
}
```



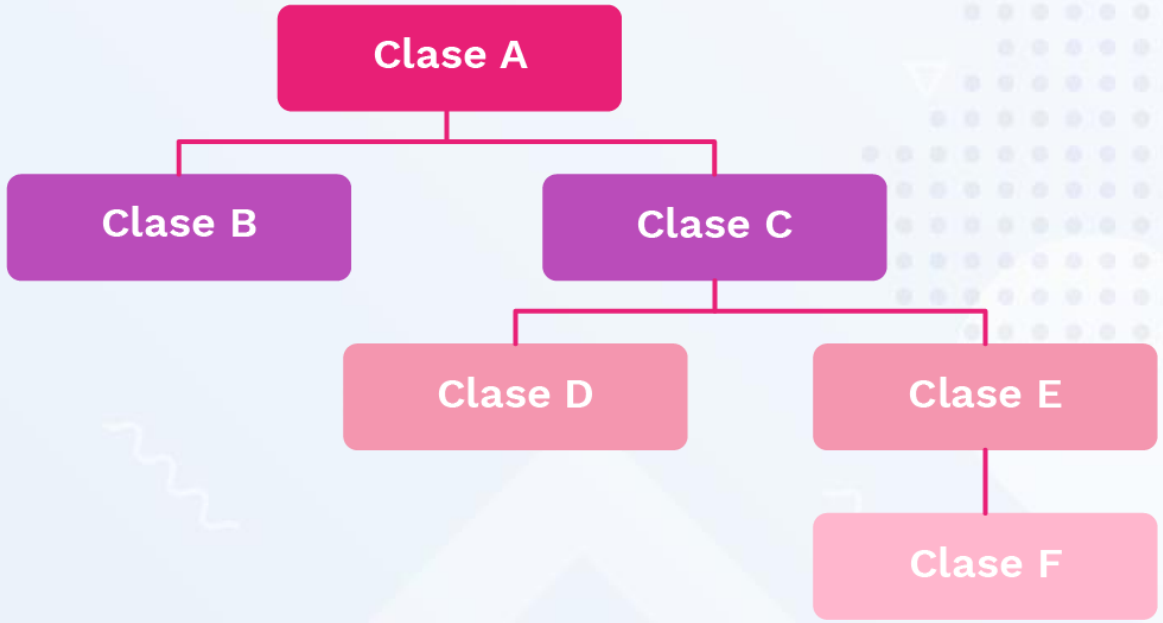
Donde se pueden ver cuatro atributos, name, email, id y age, un constructor que inicializa los atributos, y un método llamado void_info(), que visualiza un mensaje de salida con la información del usuario. En una representación UML, se puede mostrar la relación que existe entre estas dos clases mediante una composición, esto de la siguiente manera:



Donde se tienen la representación de los dos paquetes, school y user, y los diagramas de clase de cada una de las clases pertenecientes a estos paquetes, y la relación que hay entre ellas.

Herencia

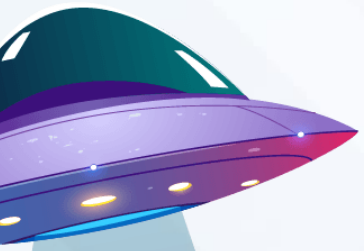
Una clase puede heredar atributos de otra clase y es a lo que se le conoce como herencia en programación orientada a objetos. Por medio de la herencia, se pueden definir nuevas clases a partir de clases que ya existen, y las nuevas clases que se definen, heredan los atributos y métodos de la clase ya existente. Java solamente permite un tipo de herencia, la herencia simple, que indica que solo se pueden definir nuevas clases a partir de únicamente una clase base, a esta clase base, de la cual se deriva una clase, se conoce como superclase y a la clase que se deriva de ella se le conoce como subclase. De la Fig. 7, se observa que la clase Clase A es la superclase de las subclases Clase B y Clase C, y la clase Clase C es la superclase de las clases Clase D y clase E. La clase Clase F es la subclase de la clase Clase E.



Un ejemplo de herencia se puede mostrar a partir de la clase anteriormente definida, la clase User, que tiene la siguiente forma:

Primera parte del código

```
package user;

public class User {
    String name;
    String email;
    int id;
    int age;
    boolean is_teacher;
    public User() {}
    public User(String name, String email, int id, int age) {
       this.name = name;
       this.email = email;
       this.id = id;
      this.age = age;
    }
```



Segunda parte del código

```
    }
    public void info() {
        System.out.println("-----------------------------------");
        if (this.is_teacher) {
            System.out.println("---------- TEACHER ------------");
        }
        else {
            System.out.println("---------- STUDENT ------------");
        }
        System.out.println("-----------------------------------");
        System.out.println("Full name: " + this.name);
        System.out.println("Email: " + this.email);
        System.out.println("ID: " + this.id);
        System.out.println("Age: " + this.age);
        System.out.println("-----------------------------------");
    }
}
```

A partir de esta clase base, se pueden definir dos clases que heredan sus atributos y métodos, estas nuevas clases pueden ser una clase para estudiantes (`Student`) y otra clase para profesores (`Teacher`). La clase `Student` tiene la siguiente estructura:

```
package user;
public class Student extends User {
    String career;
    float score_1;
    float score_2;
      public Student(String name, String email,
String career, int id, int age) {
        super(name, email, id, age);
        this.career = career;
        this.is_teacher = false;
        this.score_1 = 0.0f;
        this.score_2 = 0.0f;
    }
    public float score() {
        float score = (score_1 + score_2)/2.0f;
        return score;
    }
}
```

y la clase `Teacher` la siguiente:

```
package user;

public class Teacher extends User {
    String department;
    public Teacher(String name, String email, String department, int id, int age) {
        super(name, email, id, age);
        this.department = department;
        this.is_teacher = true;
    }
}
```

De estas clases podemos observar que las clases `Student` y `Teacher`, heredan los atributos y métodos de la clase `User`. Se puede observar que la herencia es lograda mediante la palabra reservada `extends`, la cual se muestra al momento de definir las clases Student y Teacher, y seguido de `extends` va la clase base o superclase, en este caso la clase `User`. La sintaxis para la herencia es:

```
class ClassName extends BaseClase {}
```

Donde se define la clase `ClassName` a partir de la clase `BaseClase`. Para estos casos, en los cuales se da la herencia, se debe tener en cuenta que los constructores deben empezar con la función reservada super(), cuya función lo que hace es invocar al constructor de la clase base. Como se puede observar en el ejemplo de la clase `Student`, el constructor empieza con la orden:

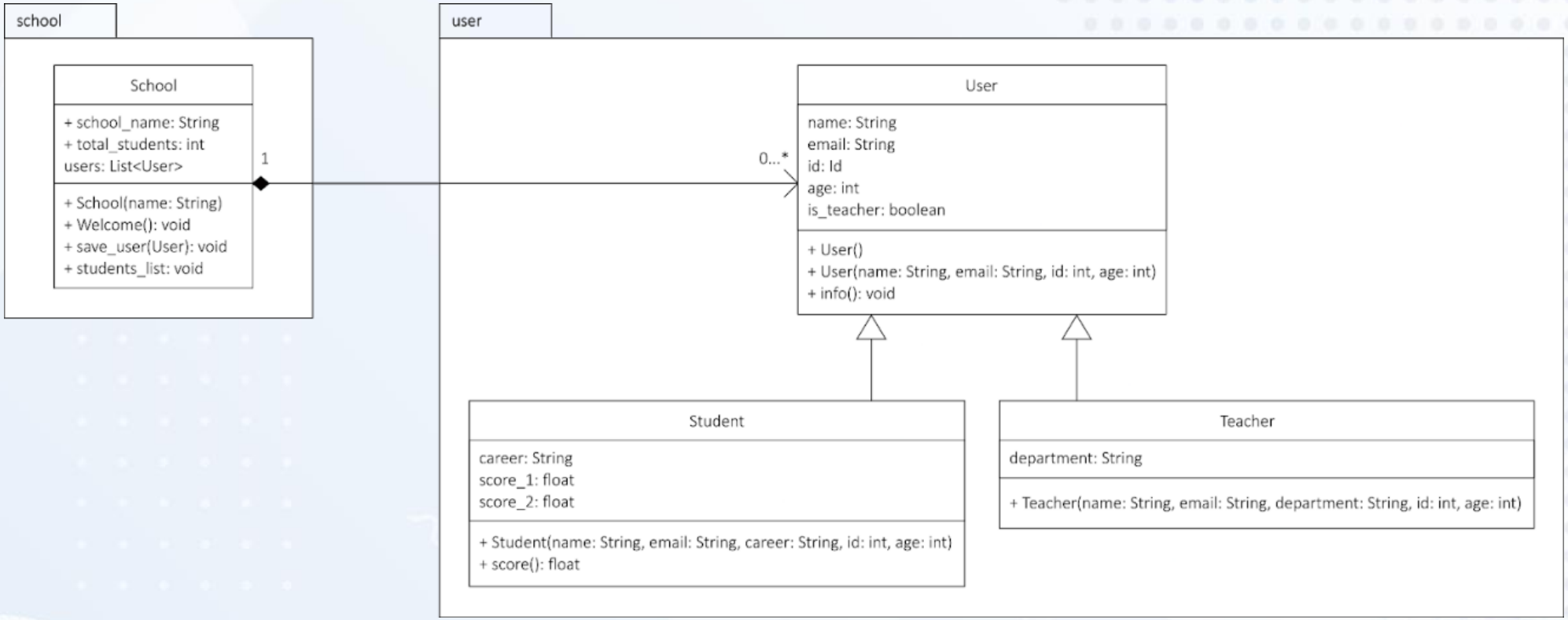
```
super(name, email, id, age);
```

Donde se debe tener en cuenta que los parámetros de entrada de `super()` deben ser los mismos que recibe el constructor de la clase base. Seguido de la implementación de `super()` podemos observar que la clase `Student` le da valores a los atributos propios de ella, y adicional a esto, hace uso del atributo `is teacher` de tipo booleano, que pertenece a la clase base:

```
this.career = career;
this.is_teacher = false;
```

Podemos concluir que, si se desean crear clases que generen objetos cuyas características sean similares, se podría crear una clase base que contenga los atributos que compartan las clases que se desean crear y a partir de esa clase, se generarían las clases derivadas, y se definen dentro de estas, aquellos atributos que no se compartan. En el ejemplo mostrado, se define la clase para los usuarios de una institución educativa y de ella se derivan dos tipos de usuarios, los estudiantes y los profesores.

La manera de representar el sistema de software que contiene dos paquetes, uno llamado school y el otro user. El paquete school contiene una clase, que como vimos se llama School, a la cual se le asocia otra clase, llamada User, que pertenece al paquete user. Adicionalmente, al paquete user fueron agregadas dos clases que se derivan de la clase User, las cuales son Student y Teacher. Este esquema del sistema de software puede ser representado mediante UML de la siguiente manera:



Polimorfismo

En la programación orientada a objetos, se puede definir polimorfismo como la capacidad de los objetos para brindar resultados diferentes de manera independiente de los tipos de objetos que se derivan de una clase.

Por ejemplo, volviendo a las clase `User` y las clases derivadas `Student` y `Teacher`, podemos implementar de manera diferente el hecho de conocer el tipo de usuario, si es un profesor o un estudiante. Anteriormente, lo hicimos mediante un atributo de tipo booleano de la clase `User`, llamado `is_teacher`. En este caso, lo haremos mediante el uso de polimorfismo, que también es conocido como sobrecarga de métodos.

Para esto se creará una nueva función tipo `String` en la clase `User`, llamada `role()`, la cual se sobre escribirá con la definición de la misma función `role()` en las clases `Student` y `Teacher`. Esta función tendrá diferentes formas para cada una de las clases derivadas de `User`, cuyas formas se obtienen mediante la adición de la anotación `@Override` a la función `role()` dentro de las clases `Student` y `Teacher`. Para ello definimos la clase base:

```
package user;

public class User {
    public User(String name, String email, int id, int age) {
        this.name = name;
        this.email = email;
        this.id = id;
    }
    public String role() {
        return “User role”;
    }
}
```

a partir de la cual se crean las clases `Student` y `Teacher`:

```
package user;

public class Student extends User {
    String career;
    public Student(String name, String email, String career, int id, int age) {
        super(name, email, id, age);
        this.career = career;
    }
    @Override
    public String role() {
        return “Student role”;
    }
}
```

y

```
package user;

public class Teacher extends User {
    String department;
    public Teacher(String name, String email, String department, int id, int age) {
        super(name, email, id, age);
        this.department = department;
    }
   @Override
    public String role() {
        return “Teacher role”;
    }
}
```

En cada una de estas clases derivadas se define la función role() que es de tipo String, las cuales retornan valores distintos según la clase derivada. Si se generan dos objetos de la siguiente manera

```
User student = new Student("Pedro Perez", "pedro.perez@colegioa.com", "Maths",
1111111, 25);
User teacher = new Teacher("Julio Gomez", "julio.gomez@colegioa.com", "Sciences",
1111113, 45);
```

por medio de los cuales se accede al método `role()` de la siguiente manera

```
System.out.println(student.is_teacher());
System.out.println(teacher.is_teacher());
```

a partir de lo cual se obtienen los siguientes resultados:

```
Student role
Teacher role
```

La palabra polimorfismo quiere decir “muchas formas”, a partir de lo cual podemos obtener múltiples formas de un atributo o un método, independientemente de la naturaleza de los objetos, siempre y cuando estas sean instancias que pertenecen a clases que se derivan de una misma clase base.

Anotaciones

Java permite la inclusión de información adicional en sus archivos fuentes, a esta información es a la que se le conoce como anotaciones, y esta no altera el comportamiento de un sistema de software. Las anotaciones en Java tienen unas características generales, dentro de las cuales encontramos:

- Empiezan con el carácter '@'.
- No cambian el comportamiento de un programa o sistema de software.
- No son comentarios ya que pueden pueden cambiar la forma en la que el compilador trata al programa.
- Relacionan los componentes del programa con datos.

Para profundizar en el tema relacionado con las anotaciones en Java se recomienda el siguiente enlace:

https://www.tokioschool.com/noticias/anotaciones-en-java/

2.4. Los niveles de visibilidad (modificadores de acceso).

Las clases, los métodos y los atributos, pueden ser declarados de acuerdo a cierto nivel de visibilidad. Existen distintos niveles de visibilidad dentro del lenguaje de programación Java, que restringen el acceso dentro de un sistema de software tanto a las clases, como a los atributos, y a los métodos. Estos niveles de acceso pueden ser modificados mediante un modificador que acompaña a la declaración de cada uno de estos elementos. Cada modificador ofrece un nivel de visibilidad dentro del sistema de software que se esté desarrollando, cuyas características se muestran en la Tab. 1.

Modificador	Clase	Paquete	Subclase	Todo el programa
public	Si	Si	Si	Si
protected	Si	Si	Si	No
Sin modificador	Si	Si	No	No
privado	Si	No	No	No

Tab. 1: Modificadores de acceso dentro del lenguaje de programación Java.

Si deseas conocer un poco más acerca de los modificadores de acceso puedes consultar el siguiente enlace:

https://www.programarya.com/Cursos/Java/Modificadores-de-Acceso.

2.5. Contextos estático y dinámico.

Las clases, los métodos o los atributos, pueden ser declarados estáticos o no estáticos. Si dentro de una clase algún atributo o algún método es declarado estático este podrá ser accedido sin necesidad de recurrir a una instancia de la clase u objeto. Como ejemplo tenemos la clase `java.lang.Math`, cuyas características pueden ser revisadas en https://docs.oracle.com/javase/7/docs/api/java/lang/Math.html, en donde podemos encontrar muchos métodos declarados de manera estática mediante la palabra reservada `static`. Uno de estos métodos es `max(float a, float b)` cuyo uso puede ser el siguiente:

```
import java.lang.Math;
public class Test {
    public static void main(String[] args) {
        float a = 5.0f;
        float b = 8.0f;
        float maximum = Math.max(a, b);
        System.out.println(maximum);
    }
}
```

donde, por medio de este método es calculado el máximo valor entre dos números tipo `float`. Los métodos o atributos declarados como no estáticos, no pueden ser referenciados desde un contexto estático, esto generaría un error al momento de compilar el programa. Una característica de declarar un método tipo estático es evitar su sobreescritura.

Para profundizar en la temática relacionada con contextos estático y dinámico, recomendamos el siguiente enlace:

https://guru99.es/java-static-variable-methods/.