**4. Encapsulamiento**

**4.1 Introducción**

El encapsulamiento es uno de los fundamentos de POO. Se refiere a la agrupación de datos con los métodos que operan sobre esos datos. El encapsulamiento se usa para ocultar las variables o el estado de un objeto de datos estructurados dentro de una clase, evitando el acceso directo de las partes no autorizadas a ellos.

Los métodos públicamente accesibles generalmente se proporcionan en la clase (los denominados getters y setters) para acceder a las variables, y otras clases de clientes llaman a estos métodos para recuperar y modificar los valores dentro del objeto.

Se dice que el objeto "publica sus interfaces". Otros objetos se adhieren a estas interfaces para usar el objeto sin tener que preocuparse por la forma en que el objeto lo logra. La idea es "no me digas cómo lo haces, solo hazlo".

**4.2 Ventajas**

La principal ventaja del encapsulamiento es la seguridad de los datos.

Los beneficios del encapsulamiento incluyen:

* Proteger un objeto del acceso no deseado por parte de los clientes.
* Permitir el acceso a un nivel sin revelar los detalles complejos por debajo de ese nivel.
* Reducir los errores humanos.
* Simplificar el mantenimiento de la aplicación.
* Hacer que la aplicación sea más fácil de entender.

​

Para un mejor encapsulamiento, los datos de los objetos casi siempre deben estar restringidos a privados o protegidos. Si elige establecer el nivel de acceso en público, asegúrese de comprender las ramificaciones de la elección.

**4.3 Modificadores de nivel de acceso**

Los modificadores de nivel de acceso determinan si otras clases pueden usar un objeto particular o invocar un método particular. Hay dos niveles de control de acceso:

* En el nivel superior: public (público) o *package-private* (sin modificador explícito).
* En el nivel de miembro: public (público), private (privado), protected (protegido) o *package-private* (sin modificador explícito).

**4.4 Visibilidad**

* public (público): accesible por cualquier clase.
* protected (protegido): accesible por la clase, todas las clases heredadas y las clases del paquete actual (editado).
* private (privado): accesible solo por la clase.
* Sin modificador: accesible para todas las clases del paquete actual.

**4.5 Implementación**

public class Encapsulamiento {

​

   public char variablePublica;

   protected int variableProtegida;

   private String variablePrivada;

​

​

   public char getVariablePublica() {

       return variablePublica;

  }

   public void setVariablePublica(char variablePublica) {

       this.variablePublica = variablePublica;

  }

​

   public char getVariablePublica() {

       return variablePublica;

  }

   public void setVariablePublica(char variablePublica) {

       this.variablePublica = variablePublica;

  }

   public char getVariablePublica() {

       return variablePublica;

  }

   public void setVariablePublica(char variablePublica) {

       this.variablePublica = variablePublica;

  }

​

   public void metodoPublico() {

  }

​

   protected void metodoProtegido() {

  }

​

   private void metodoPrivado() {

  }

​

}

**5. Abstracción**

**5.1 Introducción**

La abstracción es uno de los conceptos clave en POO. Su objetivo principal es manejar la complejidad al ocultar detalles innecesarios, separando la intefaz e implementación. Eso permite implementar una lógica más compleja sobre la abstracción proporcionada sin comprender o incluso pensar en toda la complejidad oculta.

La abstracción intenta minimizar los detalles para que el programador pueda enfocarse en unos pocos conceptos a la vez. La abstracción es la base del desarrollo de software.

**5.2 Ventajas**

* Se evita la duplicación de código y, por lo tanto, el programador no tiene que repetir tareas bastante comunes cada vez que se realiza una operación similar.
* Permite modificar los detalles de implementación interna sin afectar a los usuarios de la abstracción.
* Ayuda a definir las características de subclases de manera eficiente.

**6. Módularidad y reutilización**

En la presente sección aprenderemos los conceptos y técnicas de desarrollo relacionados con la modularidad y la reutilización de código. Los actuales proyectos de desarrollo, dependiendo de su objetivo final se vuelven tediosos de manejar, pero con la correcta utilización de las características del paradigma orientado a objetos, específicamente la modularidad, nos ayudará a que proyectos complejos se vuelvan fáciles de interpretar y manejar, que nuestro código esté ordenado y que podemos reutilizar bloques para hacer eficiente los trabajos y los tiempos de entrega.

**6.1 Modularidad:**

En el anterior contenido del parcial 1 del módulo No. 2 del presente curso hemos hecho una introducción acerca del término, discutiremos entonces algunas de las características y beneficios principales de modular el código de nuestras aplicaciones software, algunas generalidades y las características que giran alrededor del término.

Booch nos deja dos objetivos primordiales, primero que los módulos deben ser cohesivos, con estructuras de datos compartidos, algunas de las situaciones que pueden motivar la utilización de esta característica pueden ser las siguientes:

1. Modificaciones repetitivas en diferentes secciones del código
2. Escritura de código similar en diferentes partes para resolver una tarea común
3. Hacer archivos que sean de fácil lenctura e interpretación.
4. Separar las diferenctes lógicas que puedan actuar al mismo tiempo como piezas independientes usables en otras partes de nuestra aplicación.

https://youtu.be/EIUCew7iMBE

<https://youtu.be/O2A-12_iqcI>

# Herencia

La herencia es otra de las caracterísiticas de la Programación Orientada a Objetos, nos da la facilidad de crear funcionalidades en clases derivados de otras. En los módulos anteriores hemos hecho una pequeña introducción para el concepto, ahora, conoceremos más a fondo sus posibilidades.

Java, como lenguaje de programación orientado a objetos nos dá esa facilidad, nos permite crear estructuras de clase utilizando todas las cualidades de la herencia.

#### Definición:

Es la capacidad de pasar las características de campos y métodos de una clase a otra, heredarlas para generar una jerarquía que permita la generación de código más eficiente. Las clases que derivan de otras son llamadas subclases, y la clase de la cual derivan otras es llamada superclase (Clase base / Clase padre).

#### Algunas generalidades en herencia

* Las clases ubicadas arriba son generales o abstractas.
* Las clases ubicadas abajo son más específicas o concretas.
* La herencia nos dá la posibilidad de tener un código más organizado y fácil de entender.
* La evolución y mantenimiento del software se simplifican y reducen los costos de operación.
* Las clases comparten características comunes, lo que elimina la repetición de código.
* En java la herencia permite a una clase tener un número ilimitado de subclases y una superclase.
* Los constructores no son miembros, por lo que no son heredados, pero si puede ser invocado por una subclase.

La idea de la herencia es simple pero poderosa, cuando queremos definir una nueva clase en nuestro proyecto y ya existen algunas características que necesitamos en otra clase, podemos derivarla de esa existente sin necesidad de escribir ni depurarla nosotros mismos.

#### Representación básica:

En este caso el objeto Bicycle representa la superclase, de la cual surgen tres clases diferentes, llamadas subclases, en la representación Mountain Bike, Road Bike y Tandem Bike (Tres diferenctes tipos de bicicletas), comparten las características y comportamientos de la superclase Bicycle.

En Java, todas las clases descienden de la clase java.lang.object. A exceptción de Object, todas las clases tienen una superclase.

La clase **Object**, está arriba de toda la jerarquía de clases en Java, definida en el paquete java.lang, se encarga de definir el comportamiento común para todas las clases incluyendo las que escribimos.

### public class Automovil {

### // Campos de la clase Automovil

### public String placa;

### public int año;

### 

### // Un Constructor de la clase bicicleta

### public Automovil(String placa, int año) {

### placa = placaInicial;

### año = añoInicial;

### 

### }

### 

### // métodos de la clase bicicleta

### public void establecerplaca(String nuevoValor) { //Set

### placa = nuevoValor;

### }

### 

### public void estableceraño(int nuevoValor) {

### año = nuevoValor;

### }

### 

### public class Motocicleta extends Automovil

### // Nuevo campo para la clase Motocicleta

### public String Modelo;

### ​

### // Constructor para la clase Motocicleta

### public Motocicleta(String Modelo,

### int palcaInicial,

### int añoInicial) {

### super(placaInicial, añoInicial);

### Modelo = modeloInicial;

### }

### 

### // Agregamos un nuevo método a la subclase

### public void establecerModelo(String nuevoValor) {

### Modelo = nuevoValor;

### }

### public class Camioneta extends Automovil

### // Nuevo campo para la clase Camioneta

### public String Color;

### ​

### // Constructor para la clase Camioneta

### public Camioneta(String Color,

### int palcaInicial,

### int añoInicial) {

### super(placaInicial, añoInicial);

### Color = colorInicial;

### }

### 

### // Agregamos un nuevo método a la subclase

### public void establecerColoro(String nuevoValor) {

### Color = nuevoValor;

### }

### Ejemplo básico de aplicación de herencia en Java

public class Bicicleta {

   // Campos de la clase Bicicleta

   public int cadencia;     //Revolciones por mínuto o RPM

   public int engranaje;

   public int velocidad;

   // Un Constructor de la clase bicicleta

   public Bicicleta(int cadenciaInicial, int velocidadInicial, int engranajeInicial) {

       engranaje = engranajeInicial;

       cadencia = engranajeInicial;

       velocidad = velocidadInicial;

  }

   // 4 métodos de la clase bicicleta

   public void establecerCadencia(int nuevoValor) { //Set

       cadencia = nuevoValor;

  }

   public void establecerEngranaje(int nuevoValor) {

       engranje = nuevoValor;

  }

   public void Frenar(int decrementar) {

       velocidad -= decrementar;

  }

   public void Acelerar(int incrementar) {

       velocidad += incrementar;

  }

}

Ahora, definiremos una subclase que extienda de la clase Bicicleta:

Usamos extends para hereder las características y comportamiento de una clase padre.

public class bicicletaMontanesa extends Bicicleta {

   // Nuevo campo para la clase bicicletaMontanesa

   public int alturaAsiento;

​

   // Constructor para la clase bicicletaMontanesa

   public bicicletaMontanesa(int alturaInicial,

                       int cadenciaInicial,

                       int velocidadInicial,

                       int engranajeInicial) {

       super(cadenciaInicial, velocidadInicial, engranajeInicial);

       alturaAsiento = alturaInicial;

  }

   // Agregamos un nuevo método a la subclase

   public void establecerAltura(int nuevoValor) {

       alturaAsiento = nuevoValor;

  }

}

* Los atributos definidos en el primer bloque de código corresponden a la clase padre o superclase Bicicleta
* Los atributos definidos en el segundo bloque de código corresponden a la subclase bicicletaMontanesa, en donde se han definido algunas propiedades que son agregadas con las que ya se han heredado de la clase padre.
* Sin hacer más tenemos una clase bicicletaMontnesa que tiene atributos y métodos heredados lo que simplfica la codificación en aplicaciones complejas y que mejora la eficiencia en el desarrollo.

## Subclases

Heredan todos los miembros públic y protected de la superclase, sin importar en que paquete está.

#### Consideraciones:

Si la subclase está en el mismo paquete que su clase padre, también puede heredar los miembros privados de esa clase.

#### ¿Qué se hace con los miembros que se heredan?

Los campos heredados pueden usarse como cualquier otro.

Se puede definir un campo en la subclase con el mismo nombre, lo que permitirá su ocultación.

Declarar nuevos campos en la subclase que no estén definidos en la clase padre.

Los métodos heredados pueden ser usados tal y como son.

Podemos definir una nueva instancia en la subclase con el mismo nombre de la clase padre, lo que permitirá la sobreescritura.

Podemos declarar nuevos métodos que no estén definidos en la superclase

### Miembros privados en Superclase

Para acceder y usar los campos privados de una superclase, se hace por medio de sus métodos públic o protected.

Las clases anidadas en el mismo paquete tienen acceso a los miembros privados. Por lo tanto, una clase anidada protected o public heredada por una subclase tiene acceso indirecto a todos los miembros privados de la superclase.

#### Clases anidadas

Las clases anidadas se usan cuando queremos definir una lógica de aplicación agrupada en un solo lugar, es cuando esa clase solo será usada sólo por otra clase, ayuda a incrmentar la encapsulación y promueve una modificación y mantenimiento de código rápida. La estructura de las clases anidadas es la siguiente:

class OuterClass {

  ...

   class NestedClass {

      ...

  }

}

Las clases anidadas pueden ser: Estáticas, declaradas como static. Las que no son estáticas están definidas como clases internas o Inner Classes

class OuterClass {

  ...

   static class StaticNestedClass {

      ...

  }

   class InnerClass {

      ...

  }

}

\*\*El contenido se complementa con las prácticas para herencia del módulo de Desarrollo

<https://youtu.be/xJuFt7J7d9Q>

**Polimorfismo**

Si buscamos el término polimorfismo en nuestro diccionario, encontramos la siguiente definición:

"Calidad de Poliformo. Propiedad de las especies de seres vivos cuyos individuos pueden presentar diferentes formas o aspectos, bien por diferenciarse en castas, como las termitas, bien por tratarse de distintas etapas del ciclo vital, como la oruga y la mariposa."

La idea se aplica también en la programación Orientada a objetos, la función que pueden cumplir las subclases con diferentes comportamientos y al mismo tiempo compartir alguna funcionalidad de las clases padre.

**Algunas consideraciones importantes en cuanto al polimorfismo**

* El polimorfismo está directamente relacionado con la herencia
* Cuando implementamos un método abstracto en la clase padre, las subclases deben implementarlo de forma obligatoria.
* Las subclases pueden tomar comportamientos diferentes, y ese es el potencial del polimorfismo. El funcionamiento básico es el siguiente:

Consideremos a **Empresa** como Superclase que tiene los siguientes atributos y métodos que representan a los integrantes de una empresa común:

* numeroCedula
* nombre
* edad
* recibirCapacitaciones()
* diaTrabajo()

Hemos declarado algunas características que son comunes para las subclases, todos tienen un nombre, una edad y un numero de cédula. También declaramos los métodos recibirCapacitaciones() y diaTrabajo, son acciones que un contador, técnico o ingeniero de software puede ejecutar.

La definición de esos métodos en la clase Padre sería:

public void recibirCapacitaciones() {

    System.out.println("Recibiendo capacitacion de ética");

}

​

public abstract void diaTrabajo();

* La declaración del método abstracto diaTrabajo obliga a las subclases a implementarse en cada una de ellas.

Ahora, consideremos a **Contador, Técnico e Ingeniero de Software** como subclases que serán heredados de la clase padre **Empresa**, que pueden tener los siguientes atributos:

**Contador**

nombreColegiatura

registrarContabilidad()

**Técnico**

especializacion

repararComputadoras()

**Ingeniero de Software**

tecnologia

escribirProgramas()

Se han declarado entonces nuevos atributos y métodos que se sumarán a los que ya hemos heredado de la clase empresa, generando un concepto de especialización y diferenciación de cada una de sus clases.

Todas las subclases reciben capacitaciones, nada cambia, entonces no es necesario sobreescribir métodos, por defecto ya lo hemos heredado de la clase Empresa.

Con respecto al método diaTrabajo()las tres subclases comparten la acción pero cada uno hace diferentes cosas.

Para un contador agregaríamos lo siguiente en la clase Contador

 @Override

  public void diaTrabajo() {

     System.out.println("Registra los asientos contables y estados financieros");

  }

Para la clase Tecnico, el código para diaTrabajo sería el siguiente:

 @Override

  public void diaTrabajo() {

     System.out.println("Dar soporte a los equipos de computo");

  }

​

Para la clase Ingeniero de Software, el código para el método diaTrabajo() es el siguiente:

 @Override

  public void diaTrabajo() {

     System.out.println("Escribir código de programas para la empresa");

  }

​

* **Todas las subclases implementan el método diaTrabajo() con la etiqueta @Override lo que nos permite sobreescribir los métodos de la clase padre, lo que redefine el comportamiento de la misma. Si vemos cada empleado ejecuta una acción diferente en el método.**
* **Al tener una sobreescritura del método diaTrabajo(), estamos redefiniendo a cada uno de los objetos de la empresa a tener comportamientos totalmente diferentes, compartiendo siempre las características de la clase Padre.**

El contenido se complementa con prácticas en el módulo de Desarrollo.

<https://youtu.be/HKkk_ixtfLU>

Modulo 6

## Sistema de Control de Versiones

#### Programación colaborativa

En la programación de aplicaciones, nos encontramos con algunas dificultadas con relación al manejo de las versiones de nuestros programas. Algunos de los escenarios que se repiten el los proyectos de software son los siguientes:

1. Mal entendimiento con el equipo de trabajo.
2. Versiones de código repetidas en carpetas en nuestra computadora.
3. Cuando surge un error, no hay un histórico de los cambios que se han realizado.
4. Cambios directos en versiones de producción.

#### ¿Que escenarios se evitan con el uso de estos sistemas de control de versiones?

1. Creación de nombres de archivos repetidos como: trabajo\_final, trabajo\_final2, trabajo\_final\_final, trabajo\_definitivo.
2. Cometer errores en la modificación de código y no tener un hostorico que permita restablecer cambios.
3. Los miembros de un equipo trabajan de forma separada, sin retroalimentación.

Un escenario típico para el manejo de los programas de software sin el uso de Git es el siguiente:

### Tipos de SCV

* Git
* CVS
* Subversion
* Bazaar

### Acerca de Git

Es un sistema de control de versiones distribuído, creado en y para Linux, y usado en todas las demás plataformas.

#### Instalación de Git en diferentes plataformas.

Debian: # aptitude install git

Archlinux: # pacman -S git

Windows: [http://windows.github.com](http://windows.github.com/)

Mac: [http://mac.github.com](http://mac.github.com/)

## Github

Utiliza git para el manejo de versiones, permite a un equipo trabajar de forma colaborativa sin importar desde donde se realice. algunos conceptos esenciales para trabajar en su entorno son repositorios, ramas, commits y pull request.

**Repositorios:** Organiza un proyecto de forma singular, los repositorios pueden contener folders, archivos, imágenes, videos, etc.

**Ramas:** Es la forma de trabajar en diferentes versiones de un repositorio a la vez. La rama por defecto es la master, que es considerada como la rama definitiva. El funcionamiento básico cuando se crea una rama apartir de otra es el siguiente:

Existiendo una rama llamada "master", se ha creado una copia en la cual se harán algunas modificaciones del código fuente, dichas modificaciones son discutidas y aprobadas por los usuarios que tengan el acceso determinado. Quien hace las modificaciones puede ejecutar un pull request, que es una solicitud al propietario de la rama master para que su código sea incorporado a la versión de produción del código.

**Pull Request:** Son el elemento esencial del trabajo colaborativo en Github, cuando se abre una solicitud de Pull Request para que alguien pueda revisarlo y aprobarlo para que puedan unirlos en sus ramas, mostrando de manera intuituva los cambios realizados, tanto las líneas de código agregadas como las que se han quitado.

**Commits:** Son los cambios guardados que se hacen sobre determinada rama, reflejan también un histórico de esos cambios, permitiendo regresar a versiones funcionales cuando tengamos problema con los proyectos. Cada uno tiene un identificador que indica los cambios que hemos realizado, lo que ayuda a un mayor control sobre los programas.

# Bitbucket

Permite una serie de ventajas ya que ofrece repositorios privados gratuitos, basado tambien en Git, permite un control de versiones de código fuente muy eficiente y con un manejo coordinado de los equipos de trabajo.

1. Tiene los siguientes elementos, básicamente con un funcionamiento similar a GitHub:
2. Permite asignar diferentes permisos para los colaboradores, evitando cambios no autorizados a las versiones del código fuente:

1. La estructura de un pull request en donde se muestran los cambios en las lineas agregadas y quitadas de los archivos de código fuente:

## Algunos comandos generales

##### Crear un repositorio



git init

##### Agregar archivos y commit



git add ARCHIVOS

git commit -m MENSAJE

##### Agregar una etiqueta a un commit



git tag -a ETIQUETA -m "MENSAJE"

##### Información del repositorio

git status

​

git log

​

git show

##### Trabajando con ramas

Mostrar las ramas



git branch

##### Crear una Rama



git branch NombreRama

##### Moverse a una rama



git checkout NombreRama

##### Crear una rama - posicionarse en ella



git checkout -b RAMA

##### Recuperar los cambios de una rama



git merge Rama

##### Eliminación de una rama



git branch -d RAMA

git branch -D RAMA

##### Clonando Repositorio



git clone

##### Descargar cambios del repositorio remoto



git pull

##### Subir los cambios al repositorio remoto



git push

##### Mostrar ramas remotas



git branch -r

##### Mostrar todas las ramas



git branch -a

##### Clonar un repositorio existente



git clone URL

git clone git@github.com:usuario/OrientadaObjetos.git

## Formas de trabajar con Git

#### Un repositorio y varios colaboradores:

Todos pueden enviar sus cambios al las ramas del repositorio, teniendo una autoridad compartida para modificaciones.

#### Varios colaboradores - Varios repositorios

Cada uno de los colaboradores tiene en su poder un repositorio con el cual pueden trabajar y manejar los cambios correspondientes.

#### Varios colaboradores - Un lider

Se maneja un lider del equipo, quien es el único autorizado para enviar los cambios a las ramas de producción.