1. **Objetivo General**

Practicar los conceptos de herencia y polimorfismo.

1. **Objetivos Específicos:**

* Implementar relaciones de herencia utilizando Java.
* Comprender el comportamiento polimórfico que se obtiene gracias a la herencia.

1. **Contexto:**

Situándonos nuevamente en el contexto de las entidades financiaras mencionadas en laboratorios anteriores, considere estos nuevos requerimientos:

1. Ofrecer a los clientes la apertura de cuentas de diferentes tipos, es decir, debe ser posible abrir cuentas de ahorro y cuentas corrientes.
2. Cobrar comisiones en función del tipo de cuenta, esto es, la lógica de negocios para cobrar la comisión de las cuentao de herencia que ofrecen los lenguajes de programación orientados a objetos.

En este laboratorio se utilizará la clase Cuenta como clase base o superclase donde se incluirán las características (atributos) y comportamientos (métodos) comunes a los diferentes tipos de cuentas. Será necesario crear dos clases hijas o subclases para satisfacer los requerimientos antes mencionados: CuentaAhorro y CuentaCorriente. Estas deberán heredar de la superclase Cuenta. Las subclases podrán incluir los atributos que son particulares para cada tipo de cuenta. Además las subclases podrán sobreescribir métodos de la superclase con el objetivo de agregar comportamiento especializado. Finalmente, las subclases podrían agregar métodos adicionales diferentes a los heredados por la superclase. Si ello ocurre estos métodos deberán ser privados.

1. **Desarrollo:**

La relación de herencia permite organizar las diferentes abstracciones que forman parte del sistema. La herencia favorece el diseño de las abstracciones que se encuentran relacionadas por características y comportamiento compartidos.

A partir de la clase Cuenta desarrollada en el laboratorio anterior crearán dos clases adicionales: la clase CuentaAhorro y CuentaCorriente. La clase Cuenta servirá de clase padre para la clase CuentaAhorro y CuentaCorriente y mantendrá aquellas características que comparten estas clases. La clase CuentaAhorro y CuentaCorriente quedarán, entonces, vinculadas como hijas de la clase Cuenta y por tanto, heredarán las características y comportamiento de su padre.

Note que en la superclase se incluyen los atributos y métodos que son comunes para las subclases. En este contexto tenemos un caso especial: se debe incorporar en el diseño de la superclase Cuenta un método que permita cobrar las comisiones. Debido a que este comportamiento es compartido por ambas subclases. Pero, no se debe perder de vista que la implementación del método cobrar comisiones debe ser diferente en cada subclase, así lo establece el requerimiento del negocio.

Si tenemos el contexto anterior, debemos declarar el método para cobrar las comisiones como un método abstracto en la superclase. Lo anterior significa que el método aplica para todas las subclases, pero no debe programarse en la superclase porque la lógica es particular para cada caso.

En UML, los métodos abstractos se escriben en *itálica* y la clases abstractas también.

Importante: si una clase tiene al menos un método abstracto, la clase deberá ser declarada como abstracta. Las implicaciones de esto son: solo se podrán declarar objetos del tipo de la clase abstracta y la clase abstracta no podrá ser instanciada.

**Investigue usted la diferencia entre declarar y crear un objeto.**

En la clase CuentaAhorro, las comisiones se cobrarán el primero de cada mes sobre la base de un valor fijo establecido por la entidad financiera.

En la clase CuentaCorriente, las comisiones se cobrarán el primero de cada mes sobre la base del número de operaciones realizadas por el cliente, y que hayan excedido el número de operaciones exentas establecidas por la entidad financiera.

**Tenga presente que un método solo debe implementar UNA y solo UNA responsabilidad.   
¿Recuerda el método retirar? ¿Recuerda el método validarRetiro?  
Cada UNO de ellos implementaba UNA y solo UNA responsabilidad.**

**Si la programación sugerida en este laboratorio se llega a irrespetar dicha regla, usted deberá proveer la programación necesaria para cumplirla.**

**Primera Parte**

En este apartado el estudiante deberá copiar las clases del laboratorio anterior en un nuevo proyecto, además deberá modificar la clase Cuenta para poder heredar a otras clases.

Pasos a seguir:

1. Crear el proyecto

**Cree un nuevo proyecto y asígnele el nombre de la siguiente forma:   
PRYLAB2APELLIDO\_NOMBREDELESTUDIANTE\_Contexto**, en mi caso el proyecto se llamaría: **PRYLAB4CHAVARRIA\_LUIS\_Cuentas**

Use su apellido y nombre para nombrar el proyecto según corresponda.

1. **Agregue un nuevo paquete y llámelo logicadenegocios**
2. **Copie las clases que se utilizaron en el laboratorio 4 dentro del paquete logicadenegocios (menos el Apl).**
3. **Ubique y modifique el encabezado de la clase Cuenta. Declare esta clase como una clase abstracta.**

**public abstract class Cuenta { ...**

Se utiliza la palabra reservada **abstract**, para indicar que la clase contiene métodos que no poseen implementación.

1. Agregar en la clase “Cuenta” un atributo de tipo String nombrado tipoCuenta, debe declararse protegido para poder ser accedido desde las clases hijas.

protected String tipoCuenta;

La palabra reservada **protected** es utilizada por los atributos de las clases padres, con el objetivo de brindar acceso a estos atributos desde las clases hijas.

1. Debido que los modificadores de acceso de la clase cuenta son private, no podrán ser utilizados por las clases hijas, por lo cual, es necesario cambiarlos a protected para poder ser accedidos desde sus hijas. ¿Será conveniente usar package en vez de protected?

**protected** **int** numCuenta = 0;

**protected** Cliente duenio = **null**;

**protected** **double** saldo = 0;

**protected** Date fechaCreacion;

**protected** ArrayList<Operacion> operaciones;

**protected** **int** numOperaciones = 0;

protected String tipoCuenta;

1. Elimine el atributo cantCuentas de la clase Cuenta.

Esta decisión es tomada, ya que, es conveniente llevar el número de cuentas de ahorro y el número de cuentas corrientes por separado. Cada clase hija deberá llevar el control de los objetos creados. Al no hacerse de esta forma los números de cuentas de ahorro y de cuentas corrientes irán mezclados.

1. Cree los métodos accesores para el atributo tipoCuenta.

**public** String getTipoCuenta() {

**return** tipoCuenta;

}

**public** **void** setTipoCuenta(String pTipoCuenta) {

**this**.tipoCuenta = pTipoCuenta;

}

1. Modifique en la clase Cuenta el constructor para recibir el valor que corresponde a tipoCuenta y al número de cuenta.

**public** Cuenta( String pTipo, **int** pNumero, Cliente pDuenio, **double** pMonto ) {

1. Modifique el cuerpo del constructor de la clase Cuenta para inicializar los cada uno de los atributos de forma correcta. El cuerpo del constructor quedará de la siguiente manera:

setTipoCuenta( pTipo );

setNumCuenta( pNumero );

operaciones = **new** ArrayList<Operacion>();

depositar( pMonto );

setDuenio( pDuenio );

setFechaCreacion( );

1. Agregue en el método toString() de la clase Cuenta lo siguiente:

msg += "Tipo: "+ getTipoCuenta( ) + "\n";

Lo anterior se deberá agregar después de la línea de código:

msg = "Cuenta Número: " + getNumCuenta() + "\n";

1. **Agregue, en la clase Cuenta, un método sin implementación nombrado cobrarComisiones.**

**public** **abstract** String cobrarComisiones();

Observe que al declarar este método abstracto, las clases hijas están obligadas a implementarlo (de otra forma, serán consideradas abstractas también).

Guarde los cambios.

**ANALISIS DE LA PRIMERA PARTE:**

En esta primera parte, se copiaron las clases del laboratorio 4. Además, la clase Cuenta sufrió varias modificaciones para poder heredar a otras clases.

Los modificadores de acceso de la clase Cuenta cambiaron a protected para ser accedidos desde las clases hijas.

Se eliminó el atributo cantCuentas (estático) y se agregó el atributo tipoCuenta.

Por supuesto, no se puede olvidar el uso de la palabra reservada abstract para el encabezado de la clase Cuenta y el método para cobrar comisiones.

**Segunda Parte**

En esta parte, se creará la clase CuentaAhorro, se declará esta clase como hija de la clase Cuenta utilizando la palabra reservada extends, además se implementará el método cobrarComisiones.

1. Cree una nueva clase dentro del paquete logicaDeNegocios y asígnele como nombre CuentaAhorro.
2. Agregue en la clase CuentaAhorro los siguientes atributos:

**private** **static** **int** *scantCuentas* = 0;

private double cuotaMantenimiento;

1. Implemente los métodos accesores para los atributos anteriores.
2. Modifique el encabezado de la clase CuentaAhorro, para indicar que será la clase hija de la clase Cuenta, de la siguiente forma.

public class CuentaAhorro extends Cuenta {

Tenga en mente el uso de la palabra reservada extends que representa que la clase CuentaAhorro heredara de la clase Cuenta.

1. Mostrará un error que indicándonos la existencia de un método abstracto, por lo cual hay que implementar el método, como posteriormente se indicará.
2. Agregue en la clase CuentaAhorro un constructor.

**public** CuentaAhorro( Cliente pDuenio, **double** pMonto) {

**super** ("Cuenta de Ahorro", ++*scantCuentas*, pDuenio, pMonto);

setCuotaMantenimiento (300);//cuota de mantenimiento por omisión

}

Observe que debe de llamar al constructor de la clase padre, utilizando la palabra reservada: super. Debe suministrar los parámetros que permitan inicializar correctamente los atributos heredados. Dichos parámetros deben suministrarse en el orden correspondiente.

1. Agregue en la clase CuentaAhorro otro constructor, después del anterior, el cual contendrá además de los parámetros anteriores, otro parámetro que tomará el valor de la cuota de mantenimiento.

**public** CuentaAhorro( Cliente pDuenio, **double** pMonto, **double** pCuota ) {

**super**("Cuenta de Ahorro", ++*scantCuentas*, pDuenio, pMonto);

setCuotaMantenimiento(pCuota);

}

1. Cree el método cobrarComisiones de la con el siguiente contenido:

**public** String cobrarComisiones() {

String msg = " ";

Calendar calendario;

calendario = Calendar.*getInstance*();

**int** dia = calendario.get(Calendar.*DAY\_OF\_MONTH*);

**if** (dia == 1) {

retirar(cuotaMantenimiento);

numOperaciones = 0; // Se reinicia para el mes siguiente

// el número de operaciones

msg = " Se cobró por comisión : " + cuotaMantenimiento;

}

**return** msg;

}

El código anterior producirá marcadores de error por cuanto hay que importar el paquete java.util

**ANALISIS DE LA SEGUNDA PARTE:**

En esta segunda parte creamos la clase CuentaAhorro, la cual se declaraba como hija de la clase Cuenta.

Se implementó el método abstracto de la clase padre, y además se usó la palabra reservada super, para poder acceder al constructor de la clase padre y así inicializar todos los atributos que necesitaban inicialización.

**TERCERA PARTE:**

Se creará la clase CuentaCorriente, se declarará como hija de la clase Cuenta e implementará el método abstracto de su padre.

1. Cree la clase CuentaCorriente dentro del paquete logicaDeNegocios.
2. Agregue los siguientes atributos en la clase CuentaCorriente:

**private** **static** **int** *scantCuentas* = 0;

**private** **int** operacionesExentas;

**private** **double** cuotaOperaciones;

1. Implemente los métodos accesores para los atributos anteriores.
2. Modifique el encabezado de la clase CuentaAhorro, para indicar que será la clase hija de la clase Cuenta, de la siguiente forma.

public class CuentaCorriente extends Cuenta {

1. Agregue después de los atributos un constructor para la clase CuentaCorriente.

**public** CuentaCorriente(Cliente pDuenio, **double** pMonto) {

**super** ( "Cuenta Corriente", ++*scantCuentas*, pDuenio, pMonto );

setOperacionesExentas(30); //operaciones exentas por omisión

setCuotaOperaciones(30); //Cuota por operaciones por omisión

}

1. Después del constructor anterior agregue un nuevo constructor de la clase:

**public** CuentaCorriente(Cliente pDuenio,**double** pMonto,

**int** pOperExentas, **double** pCuotaOper) {

**super** ( "Cuenta Corriente", ++*scantCuentas*, pDuenio, pMonto );

setOperacionesExentas( pOperExentas );

setCuotaOperaciones ( pCuotaOper );

}

**Tenga presente que si UNA clase tiene dos o más métodos con el mismo nombre, esto se llama sobrecarga de métodos. Los métodos tienen el mismo nombre, pero tienen listas de parámetros distintas. ¿Existirá alguna otra consideración de los métodos sobre cargados? Investigue!**

1. Cree el método cobrarComisiones de la con el siguiente contenido:

**public** String cobrarComisiones() {

String msg = " ";

**int** operacNoExentas;

**double** monto;

Calendar calendario = Calendar.*getInstance*();

**int** dia = calendario.get(Calendar.*DAY\_OF\_MONTH*);

**if** ( dia == 1 ) {

operacNoExentas = numOperaciones - getOperacionesExentas();

**if** (operacNoExentas > 0) {

monto = getCuotaOperaciones() \* operacNoExentas;

retirar(monto);

msg = " Se cobró por comisión : " +monto;

}

numOperaciones = 0; // Se reinicia para el mes siguiente

}

**return** msg;

}

El código anterior producirá marcadores de error por cuanto hay que importar el paquete java.util

**ANALISIS DE LA TERCERA PARTE:**

En esta tercera parte se siguieron pasos similares a la parte anterior, a diferencia que el segundo constructor recibiá un parámetro de más, y el método implementado de la clase abstracta era un poco más complejo debido a su funcionalidad.

**CUARTA PARTE:**

Se creará el paquete aplicación donde se copiará el AplAplicacion del laboratorio anterior y se modificará para obtener los resultados deseados.

1. **Cree un paquete asignándole como nombre aplicacion.**
2. **Copie dentro de éste paquete la clase AplCuenta del laboratorio 3.**
3. **Sustituya el contenido del método main por el siguiente:**

Cliente cliente1 = **new** Cliente( "234", "Clark", "Kent" );

Cliente cliente2 = **new** Cliente( "568", "Bruce", "Wayne" );

Cuenta cuenta1 = **new** CuentaAhorro(cliente1,3500);

Cuenta cuenta2 = **new** CuentaCorriente(cliente2,1700);

cuenta1.depositar(1800);

cuenta1.depositar(2200);

cuenta1.retirar(350);

cuenta2.retirar(500);

System.*out*.println( cuenta1.cobrarComisiones() );

System.*out*.println( cuenta2.cobrarComisiones() );

System.*out*.println( );

System.*out*.println( cuenta1.toString() );

System.*out*.println( cuenta2.toString() );

El código anterior generará indicadores de error. Haga las importaciones necesarias para arreglar los códigos de error.

1. Modifique la fecha del sistema para que sea el primero de Agosto de 2017.
2. **Ejecute la aplicación.**
3. **El resultado de la ejecución se muestra a continuación:**

Se cobró por comisión: 300.0

Cuenta Número: 1

Tipo: Cuenta Ahorro

Fecha Creación: 01/08/17

Cédula: 234

Nombre: Clark Kent

Saldo: 6850.0

Registro de Operaciones

Número  Fecha  Operación  Monto

1 01/08/17 depósito 3500.0

2 01/08/17 depósito 1800.0

3 01/08/17 depósito 2200.0

4 01/08/17 retiro 350.0

5 01/08/17 retiro 300.0

Cuenta Número: 1

Tipo: Cuenta Corriente

Fecha Creación: 01/08/17

Cédula: 568

Nombre: Bruce Wayne

Saldo: 1200.0

Registro de Operaciones

Número  Fecha  Operación  Monto

1 01/08/17 depósito 1700.0

2 01/08/17 retiro 500.0

1. Modifique la programación para cumplir las reglas establecidas. Si usted considera que la programación sugerida cumple las reglas entonces no deberá realizar modificaciones.
2. **Usted debe realizar el diagrama de clases con detalles de implementación, esto es: cada atributo debe indicar: el modificador de acceso, el nombre del atributo, el tipo del atributo y el valor por omisión (si aplica). Cada método debe indicar: el modificador de acceso, el nombre del método, la lista de parámetros (nombre y tipo del parámetro) y el valor de retorno del método (si aplica).**
   1. **El diagrama debe ser un documento PDF que incorpore TODAS las clases del paquete lógica de negocios con sus respectivas relaciones (rol y multiplicidad).**
   2. **El diagrama debe ser totalmente consistente con la programación de este laboratorio.**

**Aspectos a Considerar**

* El laboratorio debe realizarse de forma individual.
* **Cada clase deberá contener un encabezado de JavaDocs.**
* **Cada método de cada clase deberá tener su respectivo JavaDocs.**
* El proyecto debe ser enviado a través del TEC-DIGITAL en un archivo comprimido .ZIP con el nombre: **LAB6APELLIDONOMBRE\_DEL\_ESTUDIANTE.ZIP**, por ejemplo: **LAB6CHAVARRIA\_LUIS.ZIP**

**El ZIP debe contener:**

1. **El proyecto de este laboratorio “PRYLAB6CHAVARRIA\_LUIS\_Cuentas”.**
2. **El PDF que corresponde al diagrama de clase con detalles de implementación de este proyecto (solo las clases del paquete de lógica de negocios).**
3. **Adjunte un PDF con evidencia de la ejecución del punto 32 (cuarta parte). La evidencia debe corresponder con un screenshot del resultado obtenido en consola.**

Para entregar la tarea en el TEC-DIGITAL debe:

1. Ingresar a su cuenta.
2. Seleccionar el curso correspondiente.
3. Acceder a la pestaña de Evaluaciones.
4. Adjuntar archivo comprimido en la opción Enviar Respuesta del Laboratorio 5

**Luego de estos cuatro laboratorios y las lecturas, Usted debería tener claridad (al menos) sobre los siguientes aspectos:**

1. **¿En qué casos es requerido incorporar un método abstracto dentro de una superclase?**
2. **¿Qué implicaciones tiene que una clase sea abstracta?**
3. **¿Qué significa sobrecargar métodos?**
4. **¿Qué significa sobreescribir métodos?**
5. **¿Qué es una superclase?**
6. **¿Qué es una subclase?**
7. **¿Qué características tiene un diagrama de clases con detalles de implementación?**
8. **¿Cuál es la notación UML para representar una clase abstracta?**
9. **¿Cuál es la notación UML para representar un método abstracto?**
10. **¿Cuál es la notación UML para representar un atributo de clase?**
11. **¿Cuál es la notación UML para representar un método de clase?**
12. **¿Cuál es la notación UML de los diferentes modificadores de acceso?**
13. **¿Cuál es la notación UML para representar la asociación, agregación, composición y herencia?**
14. **¿Para qué se necesitan los atributos de clase?**
15. **¿Para qué se necesitan los métodos de clase?**
16. **¿Por qué es importante que cada método implemente UNA y SOLO UNA responsabilidad?**
17. **¿Por qué se usan los paquetes?**
18. **¿Cuáles son los diferentes modificados de acceso y cuando usar cada uno de ellos?**
19. **¿Por qué es importante MANTENER la consistencia entre el diseño y la programación?**
20. **¿Existe en este laboratorio el comportamiento polimórfico? Si fuera así, en qué casos se presenta el comportamiento polimórfico?**
21. **¿Por qué las clases contenidas en el paquete de lógica de negocios no tienen instrucciones como** System.*out*.println ?
22. **Haga un listado sobre otros aspectos que haya aprendido hasta hoy.**

**Para reflexionar:**

