### Conceptos previos

**Casteos**

**Definición**:  
      El casteo (casting) es un procedimiento para transformar una variable primitiva de un tipo a otro, o transformar un objeto de una clase a otra clase siempre y cuando haya una relación de herencia entre ambas.  
      Existen distintos tipos de casteo (casting) de acuerdo a si se utilizan tipos de datos o clases.  
  
**Casteo Implícito (Widening Casting)**  
      No se necesita escribir código para que se lleve a cabo. Ocurre cuando se realiza una conversión ancha – widening casting – es decir, cuando se coloca un valor pequeño en un contenedor grande.  
      Ejemplo:  
                 
                //Define una variable de tipo int  con el valor 100  
                int numeroEntero = 100;  
                //Define una variable de tipo long a partir de un int  
                long numeroLargo = numeroEntero;  
  
**Casteo Explicito (Narrowing Casting)**  
      Se produce cuando se realiza una conversión estrecha – narrowing casting – es decir, cuando se coloca un valor grande en un contenedor pequeño. Son susceptibles de perdida de datos y deben realizarse a través de código fuente, de forma explicita.

Ejemplo:  
                 
                 //Define una variable del tipo int con el valor 250  
              int numeroEntero = 250;  
              //Define una variable del tipo short y castea la variable numero  
              short s = (short) numeroEntero;

System.out.println(s);

**CLASE 3**

**ASPECTOS AVANZADOS DE OOP**

**Encapsulamiento:**

Se refiere al modo como ha sido implementado el estado, los atributos de un objeto. Este aislamiento brinda una función de seguridad para que no se generen modificaciones indeseadas de los datos por quien no tiene acceso a esos valores. Se accede a ese estado mediante los métodos públicos.

Tambien se lo llama “information hiding”.

**Ej conceptual:**

Objeto empleado => atributo sueldo

empleado.sueldo = -1000. No tiene sentido. Es decir se protege a los objetos de quien no conoce verdaderamente el negocio o de posibles errores. Se previene la incorrecta modificación de los atributos, mediante los **Getter y Setter**.

Con ALT+ INSERTAR insertamos automáticamente GETTERS y SETTERS EN Netbeans.

Botón derecho => source => generar getters y setters (ECLIPSE)

*ORDEN DEL CODIGO DENTRO DE UNA CLASE:*

1. ATRIBUTOS.
2. CONSTRUCTORES.
3. METODOS.
4. GETTERS y SETTERS.

**Metodos de Acceso:**

Son el medio de acceder a los atributos privados del objeto. Son métodos públicos del objeto.

**Getter**

Accede a los atributos de un objeto en forma de solo lectura. Retornan el valor de los atributos.

**Setter**

Accede a los atributos de un objeto en forma de escritura. Son los métodos que establecen el valor de los atributos.

**Ej 1: Crear una clase Persona con los siguientes atributos (nombre, apellido, edad y domicilio). Crear los getters y los setters. Ejercicio para hacer entre todos**

//ALT + INSERT para insertar

Clase Domicilio: Atributos calle, nro y localidad. **Ejercicio para hacer entre todos**

Crear un constructor que tome los 3 argumentos correspondientes a los atributos.

Main **Ejercicio para hacer entre todos**

1. Instanciar un objeto de tipo domicilio usando el constructor creado (3 parametros)
2. Crear un objeto de tipo persona y agregarle el nombre, la edad y asignarle el domicilio creado previamente.
3. Imprimir el nombre de la persona.
4. Imprimir la localidad de la persona.

**Atributos y Metodos estaticos:**

**Atributo de Clase:**

Es un *atributo común* a todos los objetos instanciados de la clase. Al estar definido en la clase *no hace falta instanciar la clase* para utilizarlo. Las **constantes** se suelen definir como atributos de clase.

Ej:

Clase Persona

package clase1;

public class Persona {

public static int cantOrejas = 2;

}

Main

package clase1;

public class Clase1 {

public static void main(String[] args) {

Persona juan = new Persona();

Persona pedro = new Persona();

System.out.println(juan.cantOrejas);

System.out.println(pedro.cantOrejas);

System.out.println(Persona.cantOrejas); //hago referencia al atributo directamente

//sin instanciar un objeto.

//cambio el valor de la variable estatica

Persona.cantOrejas=3;

System.out.println("Nuevo Valor: "+Persona.cantOrejas);

System.out.println("Nuevo Valor Juan: "+juan.cantOrejas);

}

}

**Ej en ejercicio de Persona, Alumno**

Creo la clase EducacionIt

public class EducacionIt {

public static String nombre = "EducacionIT";

}

**Main**

//atributo estatico

System.out.println(EducacionIT.nombre);

Lo accedo directamente desde la clase. No necesito instanciar el objeto.

**Metodos de Clase o Estaticos:**

Son un comportamiento común a todas las intancias que pertenecen a la misma clase. Al ser un método de clase no hace falta instanciar un objeto para utilizarlo. Estos métodos no hablan del estado actual de una clase, sino de un **comportamiento genérico** de la misma, de un procedimiento que solo utiliza los parámetros de entrada o las variables estáticas. Son para un uso particular.

Los métodos de clase o estáticos pueden ser invocados con la notación de punto de estas dos maneras:

NombreDeLaClase.nombreDelMétodo (parámetros si los hay);

NombreDelObjeto.nombreDelMétodo (parámetros si los hay);

La diferencia es que no necesitamos instanciar un objeto, directamente podemos invocar un método estatico a través de una clase.

Los métodos estáticos tienen las siguiente Limitaciones:

* No pueden acceder a campos de instancia.
* No pueden invocar a un método de instancia.

Ej de interacción de una variable estatica con un constructor y un método de clase.

public class Estudiante {

int a; // inicializado a cero

static int b; // inicializado a cero solo cuando la clase no está cargada para cada objeto creado.

Estudiante() {

// Constructor incrementa la variable estática b

b++;

}

public void showData() {

System.out.println("Valor de a =" + a);

System.out.println("Valor de b =" + b);

}

// public void increment () {

// a ++;

//} }

}

Main

public class PracticaMain {

public static void main(String[] args) {

Estudiante s1 = new Estudiante();

s1.showData();

Estudiante s2 = new Estudiante();

s2.showData();

}

}

**Ej con el ejercicio de PERSONA ALUMNO o agregando este método en cualquier clase**

//METODO DE CLASE

public static int cantidadDeOjos(){

return 2;

}

Main

System.out.println(Persona.cantidadDeOjos());

**Herencia**

La herencia se produce a partir de dos clases relacionadas, es decir una subclase que hereda los atributos y los métodos de la superclase. Esta relación se da cuando se dice que una clase es “un tipo de” otra clase. Por ej: la clase alumno es un tipo de persona.

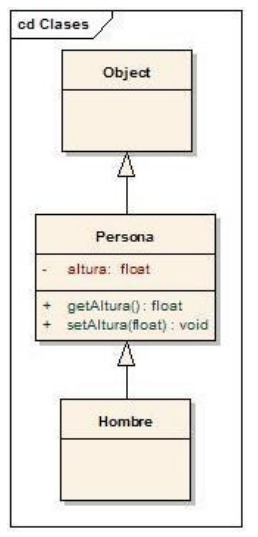
Ej:

public class Hombre extends Persona () {

}

//Esto significa que la clase padre es Persona y la clase hija la cual hereda los atributos y métodos del padre, es Hombre

Todas las clases en java heredan de la clase **Object.**

****

**Una de las ventajas de la herencia es la reutilización del codigo.**

Ej de Herencia:

**Main**

public class PracticaMain {

public static void main(String[] args) {

Estudiante s1 = new Estudiante();

s1.showData();

Estudiante s2 = new Estudiante();

s2.showData();

AspiranteEstudiante aspirante1 = new AspiranteEstudiante();

System.out.println("Aspirante 1 valor A: "+aspirante1.a);

System.out.println("Aspirante 1 valor B: "+aspirante1.b); //porque vale 3 b. Vale 3 porque el constructor fue invocado 3 veces.

}

}

**Clase Estudiante**

public class Estudiante {

int a; // inicializado a cero

static int b; // inicializado a cero solo cuando la clase no está cargada para cada objeto creado.

Estudiante() {

// Constructor incrementa la variable estática b

b++;

}

public void showData() {

System.out.println("Valor de a =" + a);

System.out.println("Valor de b =" + b);

}

// public void increment () {

// a ++;

//} }

}

**Clase AspiranteEstudiante**

public class AspiranteEstudiante extends Estudiante{

}

**Ej2:** MAIN, DOMICILIO, **ALUMNO**, PERSONA

Creo una clase Alumno subclase de Persona

public class Alumno extends Persona {

private int numeroAlumno;

public int getNumeroAlumno() {

return numeroAlumno;

}

public void setNumeroAlumno(int numeroAlumno) {

this.numeroAlumno = numeroAlumno;

}

Main

System.out.println();

//creo un domicilio para Alumno 2.

Domicilio domi2 = new Domicilio("Las Heras", 110, "RAMOS MEJIA");

//Creo un alumno y le cargo un apellido, nombre, nro alumno, domicilio

Alumno alum1 = new Alumno();

alum1.setNumeroAlumno(1000);

alum1.setNombre("Pedro");

alum1.setDomicilio(domi2);

//Muestro por consola la localidad, el nro de alumno y el nombre.

System.out.println("Nro Alumno: "+alum1.getNumeroAlumno());

System.out.println("Nombre Alumno: "+alum1.getNombre());

System.out.println("Localidad Alumno: "+alum1.getDomicilio().getLocalidad());

**Polimorfismo**

El polimorfismo se produce cuando un método adopta más de una forma. Un método puede modificar su comportamiento de acuerdo a su necesidad.

Ej del método toString(): Para mostrar como se hace un override del método toString() de la clase object.

En este caso estamos instanciando un objeto de la misma clase que contiene al main.

package polimorfismo;

public class Robot {

public static void main(String[] args) {

Robot robot = new Robot();

System.out.println(robot);

}

@Override //@Override no se debe utilizar cuando no esta sobreescribiendo

// un método de la clase padre. No es obligatorio que este.

//aunque si es recomendable.

public String toString(){

return "Hola, soy un robot";

}

}

Ej 2: continuar por aqui

Creamos una clase Numero con un override al método toString() de la clase object.

public class Numero {

protected int numero;

public void proximo(){}

public String toString(){

return String.valueOf(numero);

}

}

Creamos las clases NumeroPositivo y NumeroNegativo que hereden de Numero su atributo y que sobreescriban el método próximo(), con un constructor que setee 0 para numero positivo y -1 para nro negativo.

public class NumeroPositivo extends Numero {

//constructor

public NumeroPositivo(){

numero=0;

}

public void proximo(){

numero++;

}

}

public class NumeroNegativo extends Numero {

public NumeroNegativo(){

numero = -1;

}

public void proximo(){

numero--;

}

}

**Main**

Instanciamos un objeto positivo y otro negativo. Mostramos sus valores iniciales y luego aplicamos el método próximo();

Numero positivo = new NumeroPositivo();

Numero negativo = new NumeroNegativo();

System.out.println("numero positivo inicial:"+positivo);

System.out.println("numero negativo inicial:"+negativo);

positivo.proximo();

negativo.proximo();

System.out.println("numero positivo proximo:"+positivo);

System.out.println("numero negativo proximo:"+negativo);

positivo.proximo();

negativo.proximo();

System.out.println("numero positivo proximo:"+positivo);

System.out.println("numero negativo proximo:"+negativo);

Conclusiones del ejercicio:

* El **polimorfismo ­**permite que el método **próximo()** se comporte de manera distinta en cada caso (en el main) ya que llama al método de cada clase.
* Otro punto es que al imprimir los objetos por pantalla se deben convertir a String. Y para ello se hizo un **override** al método **toString**() de la clase **Object**. Este método lo hereda cada clase (NumeroPositivo y NumeroNegativo) de la clase padre.

**Ej2** PERSONA, ALUMNO, DOMICILIO, MAIN (Practicamos herencia)

Clase persona: Le agrego un método

//POLIMORFISMO , creo un METODO

public String saluda(){

return "Hola, soy una persona";

}

Y desde el main pruebo saludar con una persona y con un alumno, veremos que el resultado es el mismo ya que alumno hereda el método de persona.

//POLIMORFISMO

Persona p2 = new Persona();

System.out.println("Persona Saluda: "+p2.saluda());

Alumno a2 = new Alumno();

System.out.println("Alumno Saluda: "+a2.saluda());

**Polimorfismo Con Redefinicion**

Se da cuando en una clase sobreescribimos un método definido en la superclase o clase padre.

**Ej conceptual**

public class Persona {

public String cantar(){

return “Una persona canta normalmente”;

}

}

Public class Cantante extends Persona{

public String cantar(){

return “Un cantante canta con mas alcance y mayor calidad”;

}

}

Lo que podemos deducir es que los métodos se pueden sobreescribir. Y lo que devuelve el método de la clase hija puede ser diferente a lo que devuelve el de la clase padre.

La **especialización**  que se logra con este método permite cambiar el comportamiento de una familia de clases.

Ej 2: Podemos sobreescribir el método saluda() en alumno. Pisamos la inteligencia de la clase Padre. Creamos nuestra propia implementación.

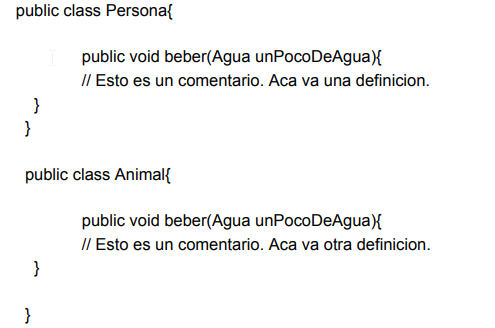
public String saluda(){

return "Hola, soy un alumno";

}

**Polimorfismo sin Redefinicion**

Se llama polimorfismo sin redefinición cuando un método tiene distinto comportamiento de acuerdo a la clase a la que pertenece.



Si bien el nombre del método es igual beber(), la diferencia es que la clase Perona lo hace como persona y la clase Animal lo hace como animal. El comportamiento es bien distinto aunque el nombre sea el mismo.

Ej 2: Ejercicio donde muestra la independencia de clases

Creo una clase “Mesa” con con el método saluda(){}.

**Clase Mesa (la creo)**

public String saluda(){

return "Hola, soy una mesa";

}

**Main**

Mesa mesa = new Mesa();

System.out.println(mesa.saluda());

Aunque no este relacionada con las clases Alumno y Persona, también puede tener el método saluda().

**Otros tipos de Polimorfismo: Sobrecarga de Metodos**

Es cuando sobreescribo un método pero con diferentes atributos.

Ej: En la clase Persona quiero que el método saluda() reciba un string como parámetro

Persona

public String saluda(String s){

return "Hola "+ s;

}

Main

System.out.println(a2.saluda("mundo."));

**Ejemplo de toString con el ejemplo de Alumno - Persona**

El método toString esta definido en Object la cual es la super clase de todas las clases en Java. Convierte a toString cualquier objeto JAVA.

El override de un método representa una descripción del comportamiento de mi objeto.

Ej de PERSONA, ALUMNO

**Persona**

**@Override**

**public String toString() {**

**return nombre + ", " + apellido + ". Edad: "+ edad;**

**}**

**Main**

**Persona p3 = new Persona();**

**p3.setApellido("Juarez");**

**p3.setEdad(40);**

**p3.setNombre("Lucas");**

**System.out.println(p3);**

**HACEMOS EL EJERCICIO DEL LABORATORIO**

**CONSIGNAS**



**Concepto de @Override:** Tambien conocido como polimorfismo dinamico nos permite hacer varias implementaciones de un método padre en una o varias clases hijas.

Se le agregan diferencias en el return del metodo.

Ventajas de su uso:

1. Indica que un método esta sobreescrito y por lo tanto proviene de una SuperClase o Interfaz.
2. Si en una superclase se elimina un método que se hereda en clases concretas y en esas clases esta el @Override, el compilador dará un aviso de que ya no se esta sobreescribiendo el método.
3. Avisa a los programadores si se han equivocado sobreescribiendo o implementando un método. Por ejemplo en un error de escritura en la subclase.

En los casos 2 y 3 se da la siguiente excepción.

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

The method metodoPrincipal(Integer) of type MainClass must override or implement a supertype method

Notas:

Una opción para subir un proyecto a Netbeans o Eclipse es descargar la carpeta del proyecto en una locación. E ir a FILE=> OPEN PROJECT=> seleccionar el proyecto.