SANTA ANA
Y SAN RAFAEL
Madrid

COLEGIOS
MARIANISTAS

POO: PROGRAMACIÓN AVANZADA DE CLASES

UD 6 Wrappers, Fechas, Clases Abstractas, Polimorfismo

PROGRAMACIÓN – 1ero DAM

WRAPPERS

- ✓ Son **clases ”envoltorio”** de datos primitivo JAVA.
- ✓ Se utilizan para tratar los **datos primitivos** de JAVA como **objetos**.
- ✓ Las **clases wrapper** implementan **objetos “inmutables”** ya que una vez convertidos no podrán ser cambiados de nuevo a datos primitivos.
- ✓ Ventaja: el uso de **métodos existentes para estas clases**.

  Por ejemplo, para convertir cadenas de caracteres (texto) en números. Esto es útil cuando se leen valores desde el teclado, desde un archivo de texto, etc.

WRAPPERS

- Los **constructores** crean envoltorios, **wrappers**, a partir de los datos numéricos o de cadenas de caracteres

Ejemplo:
```
Integer x=new Integer(34);
Double y= new Double("3.58");
int z=61;
Integer w=new Integer(z);
```

- **Método de instancia** para extraer el dato numérico del envoltorio: ***xxxValue()***

Ejemplo:
```
int a=x.intValue();
double b=y.doubleValue();
```

Primitive Data Type	Wrapper Class
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

WRAPPERS DESDE CADENA DE CARACTERES

- ✓ **Métodos de clase** para crear números a partir de cadenas de caracteres ***Xxx.parseXxx(String);***

Ejemplo:

```
int i=Integer.parseInt(“123”);
double d= Double.parseDouble(“34.89”);
```

- ✓ **Método de clase** para crear envoltorios de números a partir de cadenas de caracteres: ***xxxValueOf()***

Ejemplo:

```
Integer x=Integer.valueOf(“123”);
Double y= Double.valueOf(“34.89”);
int x=Integer.valueOf(str).intValue();
int x=Integer.parseInt(str);
```

WRAPPERS LÓGICOS

- ✓ Los **constructores** crean envoltorios o wrappers a partir de valores lógicos o de cadenas de caracteres.

  Ejemplo:
  Boolean bo=new Boolean(“false”);
  // Si la cadena no es *true* o *false* se toma el valor *false*, no hay error de compilación

- ✓ **Método de instancia** para extraer el valor lógico del envoltorio *boolean*: ***booleanValue();***

  Ejemplo:
  boolean bo=(new Boolean(“false”).booleanValue());

- ✓ **Método de clase** para crear un envoltorio lógico a partir de cadenas de caracteres: ***Boolean.valueOf(String);***

  Ejemplo:
  Boolean bo=Boolena.valueOf(str);

WRAPPERS DE CARACTERES

- ✓ **Constructor** único que crea un envoltorio a partir de un carácter:

```
Character co=new Character('a');
```

- ✓ **Método de instancia** para extraer el dato carácter del envoltorio *char*: ***charValue();***

```
char c = co.charValue();
```

- ✓ **Métodos de clase** para comprobar el tipo de carácter:

```
boolean Character.isDigit (char);          boolean Character.isLetter(char);
boolean Character.isLowerCase (char);      boolean Character.isUpperCase(char);
```

- ✓ **Métodos de clase** para convertir caracteres:

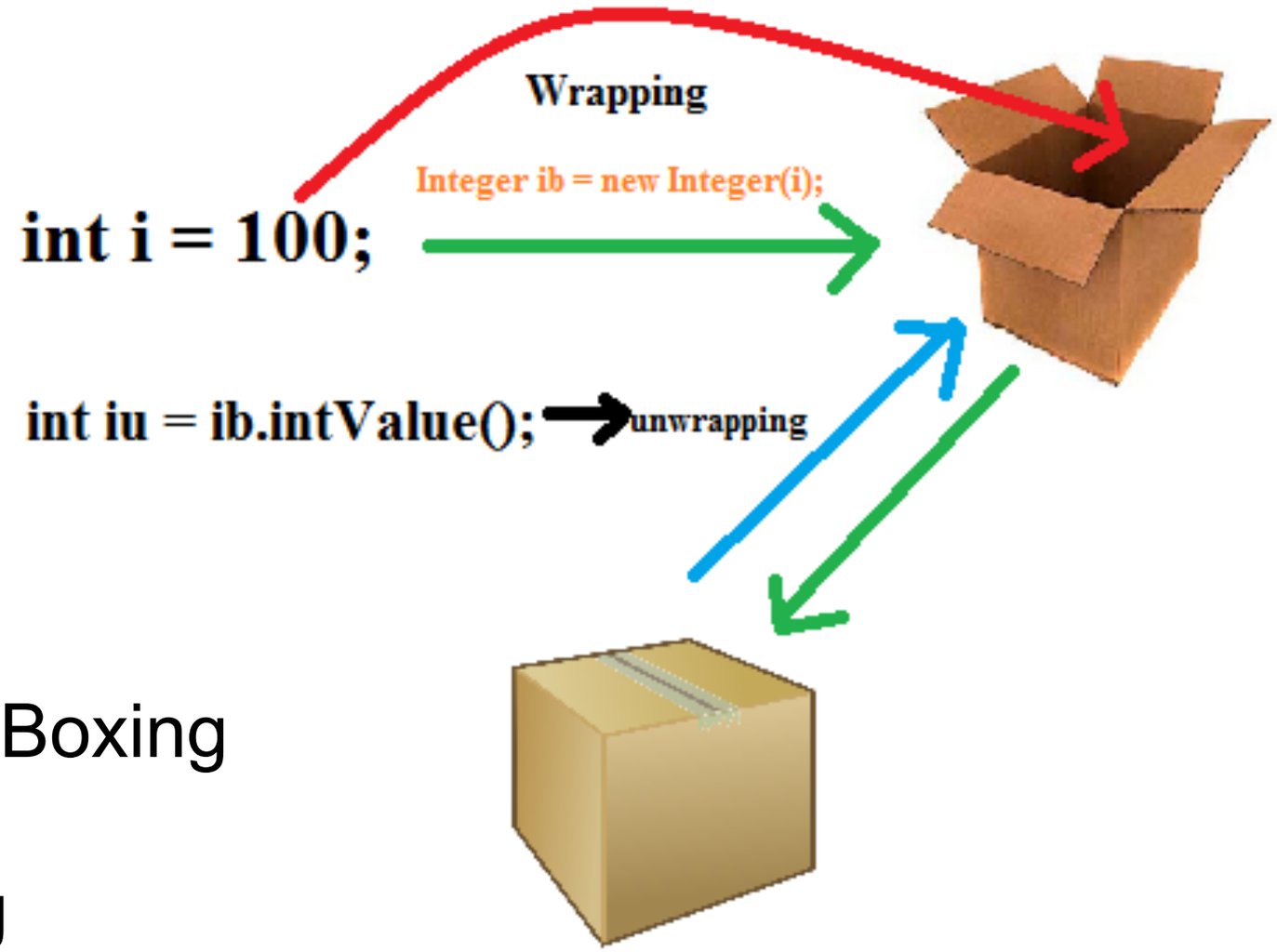
```
char Character.toLowerCase(char);       char Character.toUpperCase(char);
```

Ejemplos:

```
boolean b = Character.isLowerCase('g');
char c = Character.toUpperCase('g');
```

WRAPPERS

✓ Operaciones: ***Boxing*** o ***Wrapping*** y ***Unboxing*** o ***Unwrapping***.



Otro ejemplo:
Integer x=new Integer(34); //Boxing

int a=x.intValue(); //unboxing

WRAPPERS

- ✓ Operaciones: ***Autoboxing*** y ***Autounboxing***.
- ✓ En el **JDK 1.5** las operaciones de inserción y extracción de tipos primitivos incluyen automáticamente las operaciones de **boxing** y **unboxing**.

```
int pX= 56;
Integer wX = pX;     //autoboxing
int pY= wX;          //autounboxing
```

```
public static Integer danoReferencia = new Integer("7");
// Obsoleto por compatibilidad con otros JDK's
public static Integer danoReferencia = 7;
// Mejor usar valueOf()
public static Integer danoReferencia = Integer.valueOf("7");
```

CLASES ABSTRACTAS

- ✓ Concepto de ***Abstracción:*** es el proceso de mostrar solo la información requerida al usuario ocultando los detalles.
- ✓ En Java y en sus clases: se extrae la **esencia y el comportamiento básico** y se **generaliza**.
- ✓ En JAVA, **abstracción es sinónimo de genérico**.
- ✓ Las **clases abstractas** son clases pensadas para ser **genéricas** ya que no tiene sentido instanciar objetos de dichas clases.

EJEMPLO:
Vehículo es una clase genérica, porque en cualquier programa implementaremos objetos de tipo moto, coche, etc. pero nunca de vehículo.
Igual con clase Personaje: es una clase genérica.

CLASES ABSTRACTAS

- ✓ Un **método abstracto** no tiene implementación o definición: sólo tiene declaración.
- ✓ Un **método abstracto** no puede ser ***static***.
- ✓ Las subclases que implementan esa clase abstracta están obligadas a **redefinir ese método**, o bien, a declararlo como ***abstract***.

```
public abstract class Personaje {
  private int vida;

  Personaje() { vida=100; }

  public int getVida(){
    return vida;
  }
  public void setVida(int vida){
    this.vida = vida;
  }
  public abstract int movimientoLucha();
}
```

- ✓ Una **clase abstracta** puede tener **métodos abstractos** y no abstractos.
- ✓ De las **clases abstractas** no pueden crearse objetos.
- ✓ Si una clase tiene un **método abstracto**, por fuerza la clase pasa a ser abstracta.

FINALES

- ✓ La palabra clave **final** se usa en distintos contextos en JAVA pero siempre con un significado de finitud, de constancia o de no variabilidad.

VARIABLES FINALES

- En la declaración de variables: permite **declarar constantes**, no se puede asignar un nuevo valor a la variable una vez inicializada.

```
public final int MAX_SPEED = 50;
public int speed = 0;
```

CONSTANTES DE CLASE

- Variable "**static final**" es una **constante de clase:** un atributo común a todos los objetos de esa clase.

```
public class Cohete {
  private   static   final   int
numcohetes = 0;

  Cohete() { numcohetes++; }
  ...
}
```

UD06 POO: PROGRAMACIÓN AVANZADA DE CLASES

OBJETOS FINALES

- Cuando un objeto se declara como **objeto final**, no se pueden crear nuevos objetos con la misma referencia.

```
final cuadrado c1 = new cuadrado(5);
cuadrado c2 = new cuadrado(15);
c1=c2 // ERROR
```

MÉTODOS FINALES

- Métodos finales son àquellos que se indican para que no cambien y no se puedan sobreescribir en las subclases.

```
public class Personaje {
   ...
   public final void setVida(int vida) {
      this.vida = vida;
   }
   ...
}
```

CLASES FINALES

- **Clases finales** son aquellas clases que no pueden tener subclases que deriven de ella o descendencia.

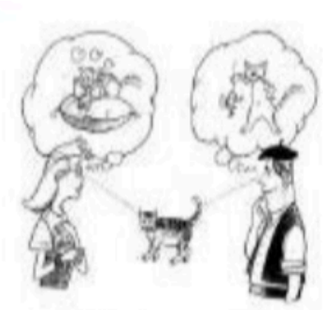
```
public final class Mago {
   ...
   ...
}
```

UD06 POO: PROGRAMACIÓN AVANZADA DE CLASES

POLIMORFISMO

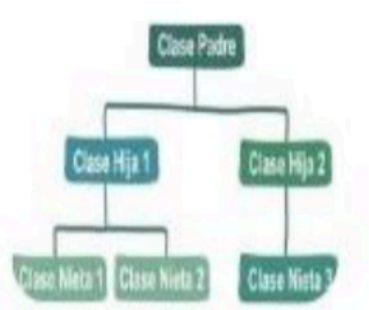
- ✓ Concepto de **polimorfismo** (RAE): cualidad de lo que tiene o puede tener varias formas.
- ✓ El **polimorfismo** se define como la característica que tienen los objetos de una clase base referenciada para apuntar a objetos de una clase derivada o subclase.

- **Abstracción**
- **Encapsulamiento**
- **Herencia**
- **Modularidad**
- **Polimorfismo**



Abstracción

Encapsulación



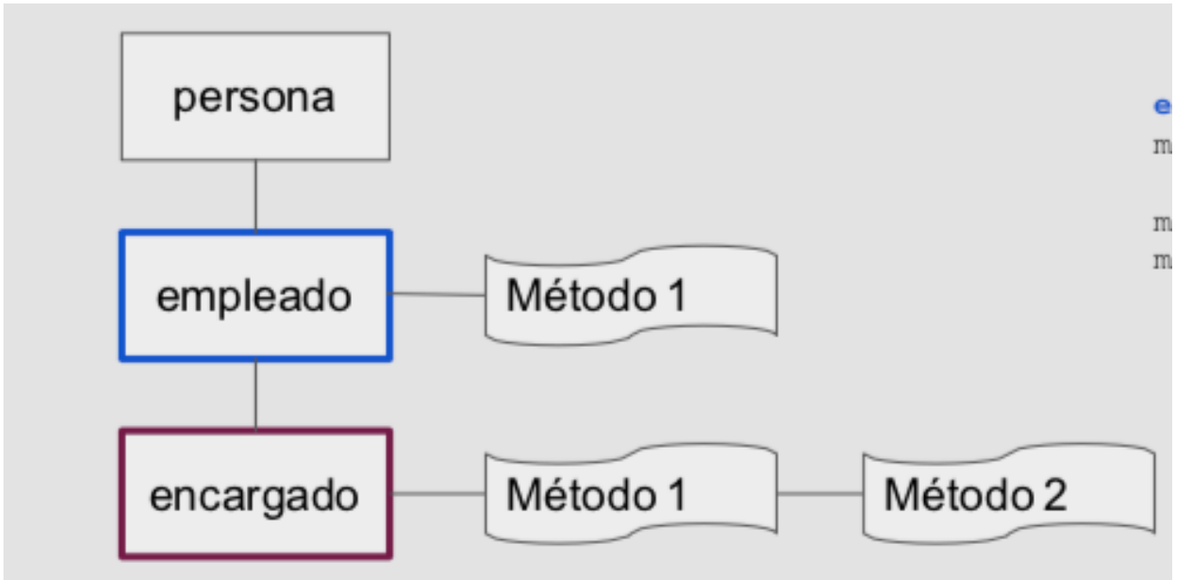
Herencia

Polimorfismo

Modularidad

POLIMORFISMO (PASO 1)

- ✓ El **polimorfismo** se define como la característica que tienen los objetos de una clase base referenciada para apuntar a objetos de una clase derivada o subclase.



Objeto miEmpleado se crea de una **clase base referenciada Empleado**

```
Empleado miEmpleado;
```

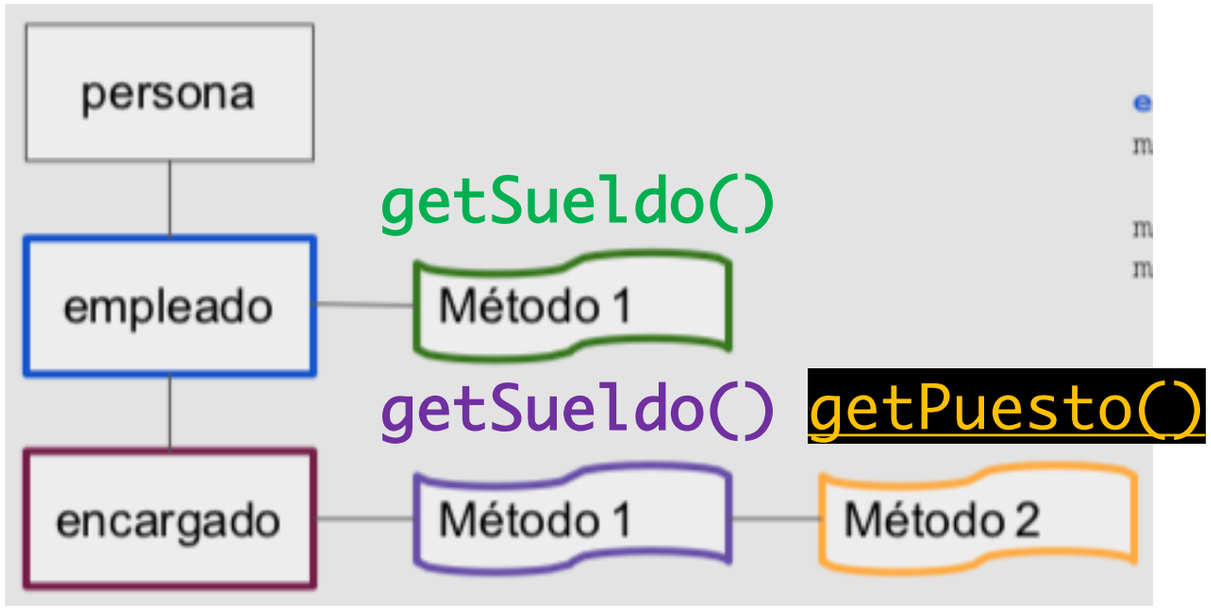
Objeto miEmpleado apunta a una clase derivada o subclase **Encargado**

```
miEmpleado = new Encargado();
//apunta a Encargado

miEmpleado.metodo2();
miEmpleado.metodo1();
```

POLIMORFISMO (PASO 2)

- ✓ Podemos **sobre escribir los métodos.**
- ✓ Para **sobreescribir un método** debe:
  - Tener el mismo nombre.
  - El retorno debe ser el mismo en el padre que en el hijo.
  - La lista de argumentos debe ser la misma y con los mismos tipos.



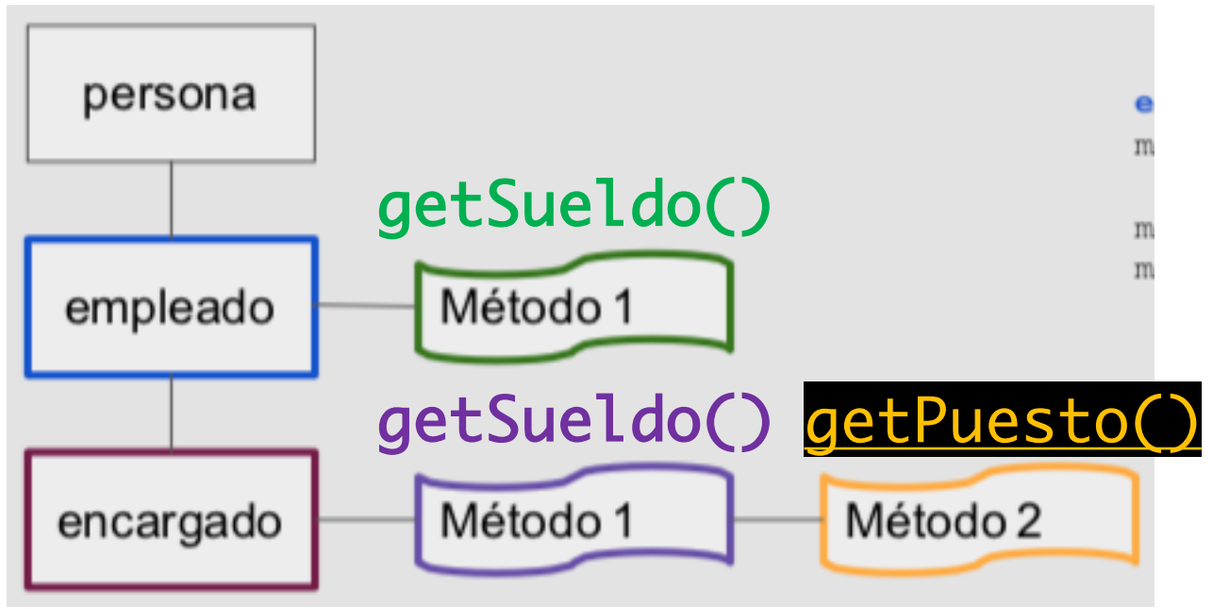

```java
public class Empleado extends Persona{
  private int sueldoBase;
  public int getSueldo() {
    return this.sueldoBase;
   }
  public void setSueldoBase(int suBa){
    this.sueldoBase = suBa;
   }
}
```

```java
public class Encargado extends Empleado {
 final private String puesto = "Encargado";
 // El encargado tiene 10% más de sueldo base
 public int getSueldo() {
  Double d = Double.valueOf(sueldoBase * 1.1);
  return d.intValue();
 }
 public String getPuesto() {
   return this.puesto;
 }
}
```

POLIMORFISMO (PASO 3)

```
Empleado miEmpleado;
miEmpleado = new Encargado();
miEmpleado.setSueldoBase(300);

sueldo = miEmpleado.getSueldo();
puesto = miEmpleado.getPuesto(); //ERROR
```

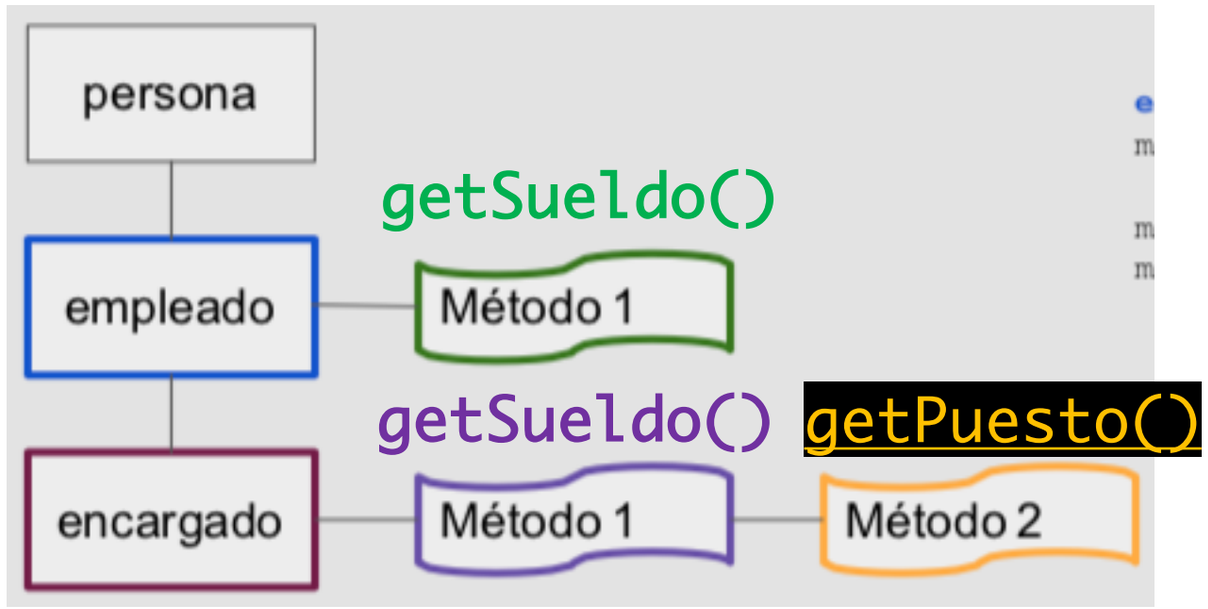


- ✓ Al crear la referencia a la clase base (**Empleado**) y el objeto a la clase derivada (**Encargado**), ocurre:
  - En tiempo de compilación, a la hora de **vincular** el Método 2 (`getPuesto()`), Java da error porque la referencia del objeto es una clase que no tiene ese método (clase base **Empleado**). Es lo que se llama **vinculación temprana** que se realiza en **tiempo de compilación**.
  - En tiempo de ejecución, Java evalúa el Método 1 (`getSueldo()`) que se ha sobre-escrito y detecta que debe llamar al de la clase derivada **Encargado**. Es lo que se llama **vinculación tardía** y se realiza en **tiempo de ejecución**.

Los **métodos definidos como final** siempre tendrán un **vinculación temprana**, es decir, se evalúan en **tiempo de compilación**.

POLIMORFISMO (PASO 4)

- ✓ Para solucionar este problema podemos utilizar:



CAST EXPLÍCITO

- Obligamos al compilador a transformar obligatoriamente el objeto inicialmente referenciado como de la clase base, a objeto de la clase derivada que sí tiene el método necesitado.

```
//Paso 4: Cast Explícito

puesto = ((Encargado)miEmpleado).getPuesto();
System.out.println("Su puesto  es: " + puesto);
```

GESTIÓN DE FECHAS

- ✓ JAVA ofrece varias opciones para obtener y gestionar las fechas.
- ✓ Podemos utilizar una **clase que represente una fecha** u otra **clase que represente un calendario** del que obtendremos alguna fecha.

CLASE DATE

- Clase `Date` predefinida en el **API** de Java que representa las fechas.
- `Date` es un tipo de Objeto de Java (como tipo **String**) que pertenece al paquete **java.util** (Hay dos clases Date en el API Java en paquetes diferentes)
- La clase `Date` representa un instante específico en el tiempo, con una precisión de milisegundos. Es una variable de tipo Long que registra los milisegundos transcurridos desde el 1 de enero de 1970.

```
Date fecha = new Date();
System.out.println("1 Fecha: " + fecha.getTime());
System.out.println("2 Fecha: " + fecha);
```

1 Fecha: 1647449098354
2 Fecha: Wed Mar 16 17:46:25 CET 2022

GESTIÓN DE FECHAS

CLASE DATE

- Para **formatear** el objeto `Date` podemos utilizar una clase auxiliar, **SimpleDateFormat**, del paquete **java.text.SimpleDateFormat** y su método **format()** indicando el formato de la fecha que queremos.

```
Date fecha = new Date();
SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/YYYY");
System.out.println("3 Fecha: " + sdf.format(fecha));
```

3 Fecha: 16/03/2022

```
Date fecha2 = new Date();
System.out.println("4 Fecha: " + sdf.format(fecha2));
```

Devuelve un ***String***

- La clase **Date** tiene muchos **métodos obsoletos** y es mejor utilizar otras clases de Fechas del API de Java para crear fechas, formatearlas, compararlas, etc.

GESTIÓN DE FECHAS

GREGORIAN CALENDAR

- Clase predefinida del **API** de Java.
- Clase que construye una fecha, con el día de la semana, el mes y el año.
- Hereda de **clase abstracta** `Calendar`
- Al constructor `GregorianCalendar` le podemos pasar diferentes parámetros:

Sobrecarga de constructores

- Constructor por defecto. Utiliza el tiempo actual para construir la clase: `GregorianCalendar ()`
- Establecer una fecha pasándole por parámetro año, mes y día:

```
GregorianCalendar (int year, int month, int dayOfMonth)
```

Empieza a contar los meses desde 0 (enero) hasta 11 (diciembre)

Calendario Gregoriano

GESTIÓN DE FECHAS

GREGORIAN CALENDAR: MÉTODOS

```
//Crear un objeto de tipo GregorianCalendar

GregorianCalendar calendario = new GregorianCalendar();
System.out.println("La fecha es: " + calendario.getTime());
```

java.util

Class GregorianCalendar

java.lang.Object
  java.util.Calendar
    java.util.GregorianCalendar

All Implemented Interfaces:

Serializable, Cloneable, Comparable<Calendar>

getTime() es un método de la clase **Calendar**

Devuelve un objeto ***Date***

Date

getTime()

Returns a Date object representing this Calendar's time value (millisecond offset from the Epoch").

GESTIÓN DE FECHAS

CLASE GREGORIANCALENDAR CON FORMATO

```
SimpleDateFormat sdf1 = new SimpleDateFormat("dd/MM/YYYY");
SimpleDateFormat sdf2 = new SimpleDateFormat("dd/MM/YYYY HH:mm");
SimpleDateFormat sdf3 = new SimpleDateFormat("HH:mm");

GregorianCalendar fecha1 = new GregorianCalendar();
GregorianCalendar fecha2 = new GregorianCalendar(1920,2,13);
```

Objetos GregorianCalendar

```
System.out.println("\nFecha 1: "+ sdf1.format( fecha1.getTime() ));
System.out.println("\nFecha 2: "+ sdf1.format( fecha2.getTime() ));

System.out.println("\nFecha 1 con horas: "+ sdf2.format(fecha1.getTime()) );
System.out.println("\nFecha 1 solo horas: "+ sdf3.format(fecha1.getTime()) );
```

Objetos Date

Objetos String

```
Fecha 1: 16/03/2022
Fecha 2: 13/03/1920
Fecha 1 con horas: 16/03/2022 19:20
Fecha 1 solo horas: 19:20
```

ACCESO A MÉTODOS DE LA CLASE PADRE

- ✓ El método `super()` nos sirve para acceder a los métodos de la clase que **hereda la subclase**.
- ✓ Su utilidad principal reside en que nos permite sobreescribir un método, asumiendo la funcionalidad implementada en el padre.

```
public class clasePadre {

        int i = 0;

        public void suma(int j){
                i = i + j;
        }

}
```

```
public class claseHijo extends clasePadre{

        public void suma(int j){
                j = j + 10;
                super.suma(j);
        }

}
```