



Departamento de Programación  
Facultad de Informática  
Universidad Nacional del Comahue



# Programación Concurrente



*Presentación y Repaso*

# Presentación

## Programación concurrente (PC)

### Area: Programación Especializada

#### Correlativas:

- Introducción a la Computación
- Programación Orientada a Objetos

#### Docentes:

- Silvia Amaro,
- Valeria Zoratto
- Juan Carlos Orlando

#### Horarios: 4 hs semanales

- Teoría: Lunes 11:00 – 13:00 hs.
- Práctica: Viernes 16 -18:00 hs.



## Correlativas:

### **Introducción a la Computación (1<sup>er</sup> año)**

- Conceptos de Sistemas Operativos. Conceptos de Redes.
  - ✓ Introducción a los Sistemas Operativos
  - ✓ Administración de procesos,
  - ✓ Administración de memoria, sistema de archivos, protección

### **Programación Orientada a Objetos (2<sup>do</sup> año)**

- Aplicar el diseño y características de la programación orientada a objetos en la resolución de problemas
  - ✓ Paradigma de Objetos
  - ✓ Herencia, Polimorfismo
  - ✓ Implementar soluciones orientadas a objetos (desde su representación estática hasta su codificación)



# Temario

- Herencia en Java
- Visibilidad en Java
- Interfaces
- Tipo estático y dinámico
- Excepciones

# Ejemplo: Clase Persona

## Persona

- nombre

### Constructoras

+ Persona()  
+ Persona(String n)

### Observadoras

+ getNombre():String  
+ getDatos():void

### Modificadoras

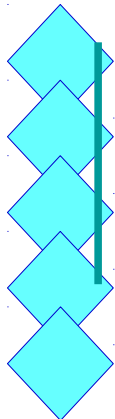
+ setNombre(String n) void

### Propias del tipo

+ mismoNombre(Persona p): boolean

# Ejemplo: Clase Persona

```
public class Persona {  
    private String nombre;  
    public Persona( ) {  
        this.nombre = null;  
    }  
    public Persona(String n) {  
        this.nombre = n;  
    }  
    public void setNombre(String nuevoNombre) {  
        this.nombre = nuevoNombre;  
    }  
    public String getNombre( ) {  
        return this.nombre;  
    }  
    public String getDatos( ) {  
        return ("Nombre: " + this.getNombre());  
    }  
    public boolean mismoNombre(Persona otraPersona) {  
        return  
(this.getNombre().equalsIgnoreCase(otraPersona.getNombre()));  
    }  
}
```



## Diagrama UML de la clase derivada Estudiante

### Persona

```
- nombre
+ Persona()
+ Persona(String n)
+ getNombre():String
+ getDatos():String
+ setNombre(String n) void
+ mismoNombre(Persona p): boolean
```

es una

### Estudiante

```
- legajo // no mutable
```

#### Constructoras

```
+ Estudiante(int l)
+ Estudiante(int l, String n)
```

#### Observadoras

```
+ getLegajo():int
+ getDatos(): String
```

#### Modificadoras

```
// modifica nombre desde la clase base
```

#### Propias del tipo

```
+ esIgual(Estudiante e): boolean
+ aCadena(): String
```

# Clase Empleado

```
public class Empleado extends Persona  
{
```

```
    private int nroEmpleado;
```

```
    public Empleado()  
{
```

```
        super();
```

```
        nroEmpleado= 0;
```

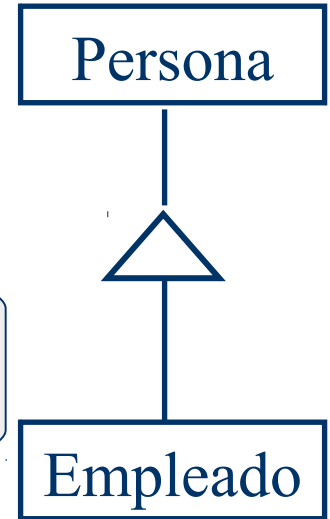
```
    }
```

```
    ...
```

Utilizar palabra  
clave **extends**

Primer acción  
del constructor

super() llama al  
constructor por defecto  
de la clase padre





# Constructor clase Empleado



```
public class Empleado extends Persona
```

```
{
```

```
.....  
public Empleado(String nuevoNombre, int nuevoNroEmpleado)
```

```
{
```

```
    super(nuevoNombre);
```

```
    nroEmpleado = nuevoNroEmpleado;
```

```
}
```

```
...
```

Usa un segundo parámetro para inicializar la variable de instancia que no está en la clase base

Pasa el parámetro *nuevoNombre* al constructor de la clase base

# Redefiniendo constructores

```
public class Estudiante extends Persona {  
    private int legajo;  
  
    public Estudiante(int leg)  
    {  
        super ();  
        this.legajo = leg;  
    }  
}
```

- Palabra clave **extends** crea la clase derivada desde la clase base, usando herencia
- La clase Estudiante establece dos constructores, uno donde se inicializa al atributo legajo con el argumento leg
  - **super** es la primera acción en un constructor de una clase derivada.
  - Si no estuviese, Java lo incluye automáticamente
  - **super()** invoca al constructor por defecto de la clase base

# Redefiniendo constructores

```
public class Estudiante extends Persona {  
    private int legajo;  
    public Estudiante(int leg, String nom)  
    {  
        super(nom) ;  
        this.legajo = leg;  
    }  
}
```

- Este constructor pasa el parámetro `nom` al constructor de la clase base
- Utiliza el segundo parámetro para inicializar la variable de instancia que no está en la clase base.

# Redefiniendo constructores

- La clase Estudiante tiene un constructor con dos parámetros: String para el atributo nombre y el atributo legajo de tipo int

```
public Estudiante(int legajoNuevo, String
nombreNuevo)
{
    super(nombreNuevo);
    this.legajo = legajoNuevo;
}
```

- El otro constructor dentro de Estudiante puede ser escrito invocando al constructor con dos argumentos dentro de la misma clase


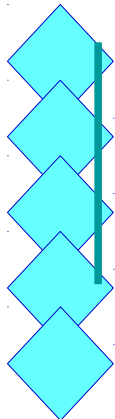
```
public Estudiante(int leg)
{
    this(leg, null);
}
```



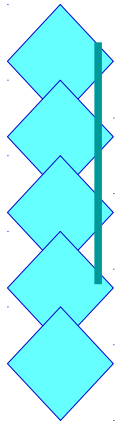
# Agregando atributos

```
private int legajo;
```

- En la clase Estudiante se agrega el atributo **legajo**
- Estudiante tiene **dos** atributos:
  - El atributo **legajo** (propio)
  - El atributo **nombre** (que heredó desde Persona)



```
public class Estudiante extends Persona {
    private int legajo;
    public Estudiante(int leg) {
        super( );
        this.legajo = leg;
    }
    public Estudiante(int leg, String nombreInicial) {
        super(nombreInicial);
        this.legajo = leg;
    }
    public int getLegajo() {
        return this.legajo;
    }
    public void setLegajo(int legajoNuevo) {
        this.legajo = legajoNuevo;
    }
    ...
}
```



```
public String getDatos() {  
    return("Nombre: " + this.getNombre( )  
        + "\nLegajo: " + this.getLegajo());  
}
```

```
public boolean esIgual(Estudiante otroEstudiante) {  
    return (this.mismoNombre(otroEstudiante)  
        && (this.getLegajo() ==  
otroEstudiante.getLegajo()));  
}
```

Puedo verificar accediendo a la variable de instancia de los objetos?

# Constructores de Empleado (subclase Persona)

Empleado tiene un constructor con dos parámetros: *String* para el atributo nombre e *int* para el atributo número de empleado

```
public Empleado (String nuevoNombre, int nuevoNroEmpleado)
{
    super(nuevoNombre);
    nroEmpleado = nuevoNroEmpleado;
}
```

Otro constructor dentro de Empleado que llama al constructor con dos argumentos nombreInicial (String) y 0 (int),

```
public Empleado (String nombreInicial)
{
    this(nombreInicial, 0);
}
```





# Resumen de Constructores

- Los constructores pueden llamar a **otros constructores**
- Se debe utilizar **super** para invocar a un constructor de la clase padre
- Se debe utilizar **this** para invocar a un constructor dentro de la clase
- Cualquiera de las dos opciones debe ser la **primera acción** realizada por el constructor
- Si se quiere invocar a ambos se debe utilizar **this** para llamar a un constructor con **super**



# Temario

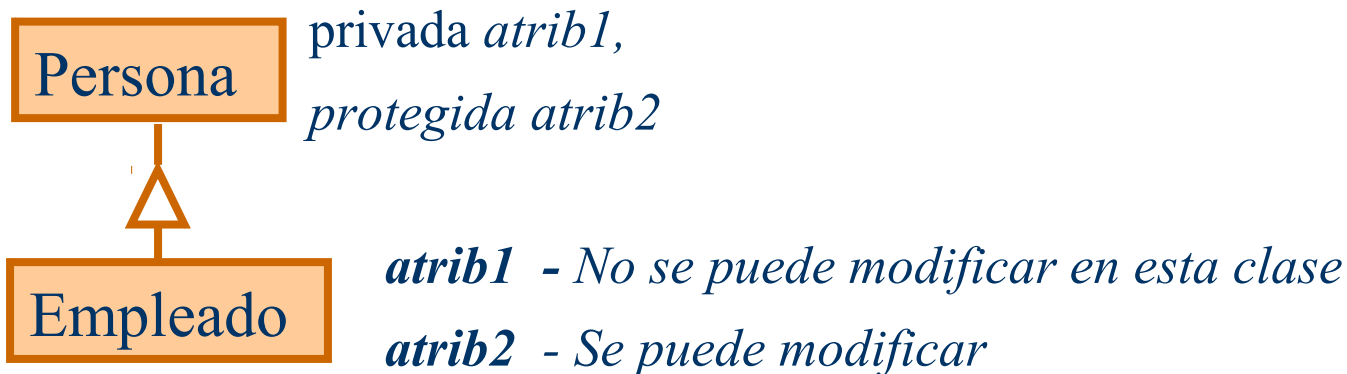
- Herencia en Java
- Visibilidad en Java
- Interfaces
- Tipo estático y dinámico
- Excepciones



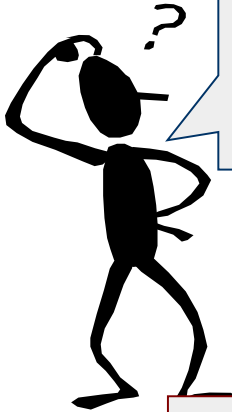
# Visibilidad

## Variables de instancia

- Públicas --- acceso fuera del ámbito de la clase
- Privadas --- acceso sólo dentro de la clase
- Protegidas --- acceso dentro de la clase y sub-clases



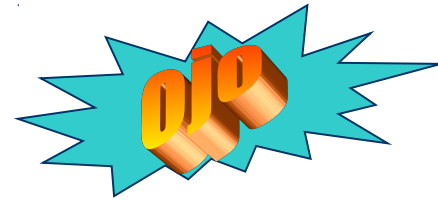
# Visibilidad



*Qué variables se utilizan en las subclases*

Las variables privadas no están disponibles en las subclases.

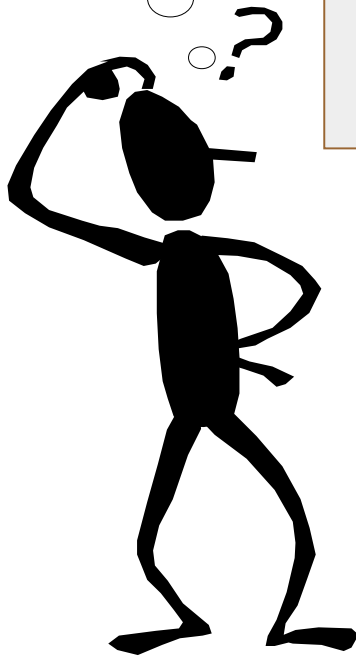
Las variables protegidas **están** disponibles en las subclases.



Los métodos privados no son heredados!

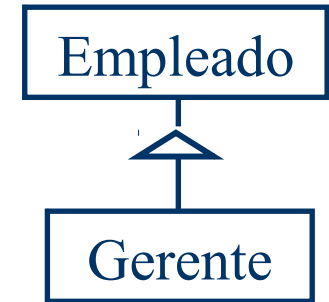
# Clase Empleado y Gerente

¿Que tipo de polimorfismo es?



```
public class Empleado
{
    private String nombre;
    private Double salario;
    ....
    public String getDatos()
    {
        return "Nombre " + nombre +
            "\n" + "Salario: " + salario;
    }....
```

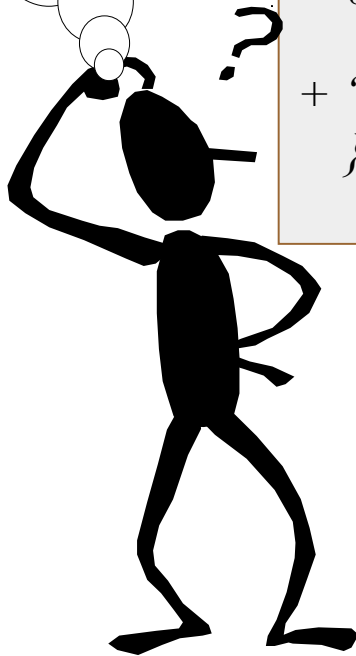
```
public class Gerente extends Empleado
    private String depto;
    ....
    public String getDatos()
    {
        return super.getDatos +
            "\n Gerente de : " + depto;
    }...
```



## Otra implementación de Empleado y Gerente

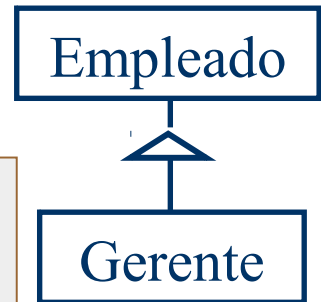
¿y ahora ...?  
¿que tipo de  
Polimorfismo es?

Es apropiado?

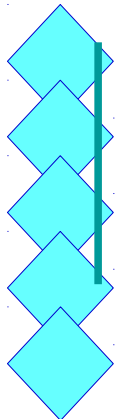


```
public class Empleado
{
    protected String nombre;
    protected double salario;
    ....
    public getDatos()
    {
        return "Nombre " + nombre + "\n"
        + "Salario: " + salario;
    }....
```

```
public class Gerente extends Empleado
    private String depto;
    ....
    public getDatos()
    {
        return "Nombre " + nombre + "\n" +
        "Gerente de : " + depto;
    }...
```



## Redefinicion/sobreescritura



Los métodos redefinidos no pueden ser menos accesibles



```
public class Padre{  
    public void haceAlgo1() {}  
}  
  
public class Hijo extends Padre {  
    private void haceAlgo1() {} ????  
}  
  
public class UsarAmbos {  
    private void haceAlgo1() {}  
    Padre p1 = new Padre();  
    Padre p2 = new Hijo();  
  
    p1.haceAlgo1();  
    p2.haceAlgo1();  
}
```



# Resumen

- Una clase derivada hereda métodos de la clase base y (a través de algunos de ellos) accede a sus atributos
- Una clase derivada puede tener atributos y métodos adicionales
- El constructor de una clase derivada debe invocar al constructor de la clase base
- Si una clase redefine un método de la clase base, la versión en la clase derivada reemplaza a la de la clase base
- Las variables de instancia y los métodos privados de una clase base no pueden ser accedidos directamente en la clase derivada
- Si A es una clase derivada de la clase B, entonces A es miembro de ambas clases, A y B

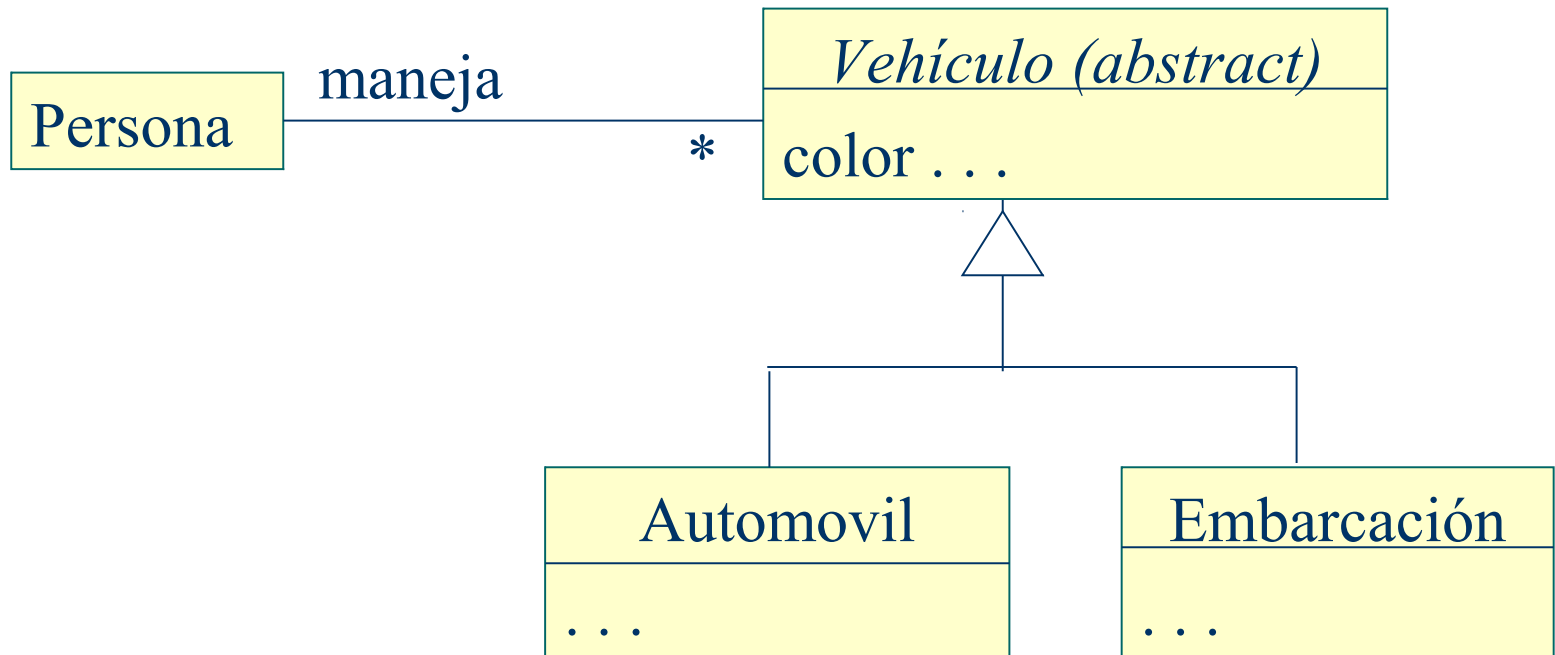




# Temario

- Herencia en Java
- Visibilidad en Java
- Interfaces
- Tipo estático y dinámico
- Excepciones

# Clases abstractas



**Los objetos reales que la persona conduce son instancias de una de las subclases concretas**



# Clases Abstractas

- En el ejemplo, la clase Vehículo no fue pensada para instanciar objetos de clase Vehículo sino como clase base para otras clases derivadas.
- Hay métodos que deben estar definidos en Vehículo (de manera abstracta) pero deben ser implementados en las clases derivadas

# Clases Abstracta – Clase Concreta

- No puede tener instancias

- Describe atributos y comportamiento **común** a sus subclases

- Puede tener **métodos abstractos**

vs

- Puede tener instancias

- Todos los **métodos** están **implementados**. Puede tener implementaciones diferentes en sus subclases

- No puede tener ningún **método abstracto**.



# Interfaces en Java

- Una interfaz en Java es una clase abstracta pura, donde todos los métodos son **abstract** (ninguno está implementado)
- También puede contener variables, pero siempre **static** y **final**



# Interfaces en Java

- Para crear una interfaz se utiliza la palabra clave **interface** y todos sus métodos son **public**.
- Para indicar que una clase implementa los métodos de una interfaz se utiliza la palabra clave **implements** (puede implementar más de una)
- La clase debe implementar TODOS los metodos de la interfaz

# Ejemplo interface

Definir la  
interfaz

```
public interface MiInterfaz {  
    /* ejemplo de interfaz */  
  
    public boolean metodo1 (int i);  
    public void método2();  
}
```

Utilizar la  
interfaz

```
public class MiClase implements MiInterfaz {  
    /* la clase debe implementar todos los  
    métodos definidos en la interfaz*/  
  
    ....  
    public boolean metodo1 (int i){  
        return true  
    };  
    public void método2(){  
        .....  
    };  
}
```



# Resumen de Conceptos

- Separa los conceptos de *subclase y subtipo*.
- Todas las clases son derivadas de una clase raiz, si no hay clase padre explicitada se utiliza *Object*.
- Si bien Java soporta *herencia simple*, también soporta múltiples interfaces es por esto que en algunas bibliografías aparece como herencia múltiple (aunque no lo es).
- Una clase puede extender *múltiples interfaces*.  
Ej. class graphicalObject **implements Storable, Graphical** {...};
- Utiliza las palabras claves *abstract* (para clase abstracta), *final* (indica que una clase no puede tener subclases).





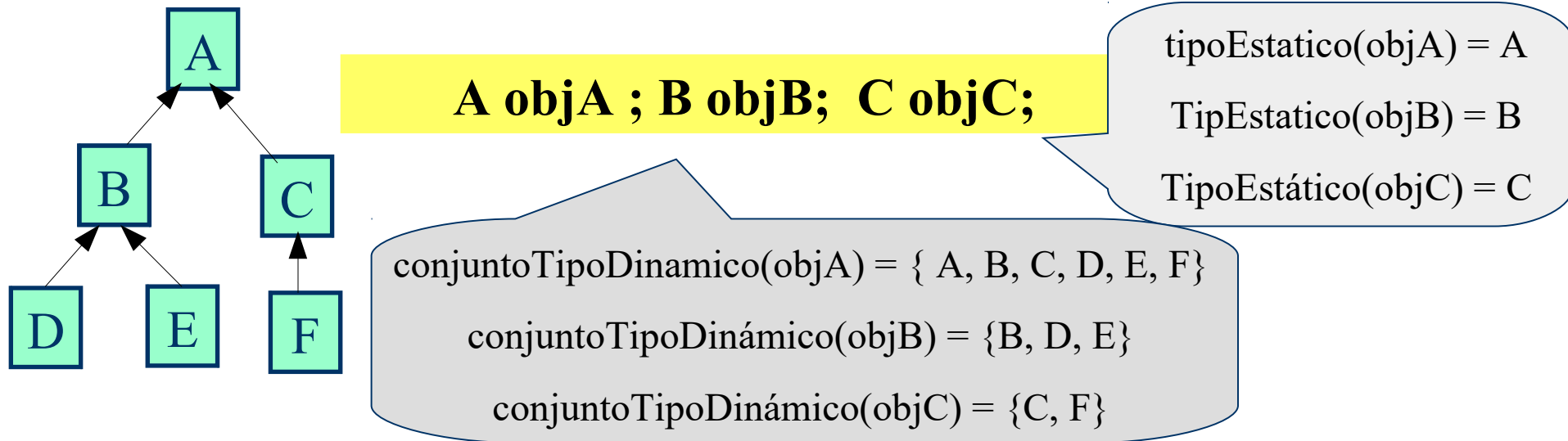
# Temario

- Herencia en Java
- Visibilidad en Java
- Interfaces
- Tipo estático y dinámico
- Excepciones

# Tipos de una variable en Java

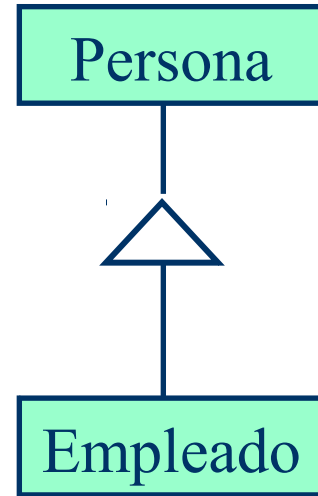
- Una variable tiene **tipo estático** y **tipo dinámico**
  - Tipo estático: asociado a la **declaración**
  - Tipo dinámico corresponde a la clase del objeto conectado a la entidad en **tiempo de ejecución**

El conjunto de tipos dinámicos es el conjunto de posibles tipos dinámicos de una entidad



# Tipos estático y dinámico en Java

```
Persona p;  
p = new Persona();  
p = new Empleado();
```



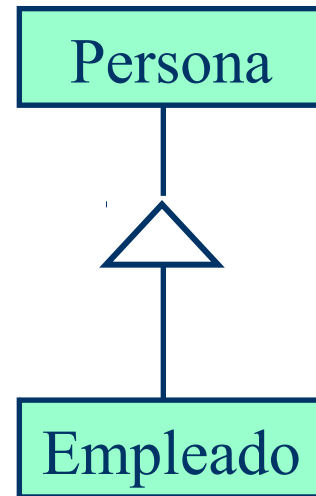
El **tipo estático** de la variable es Persona

El **tipo dinámico** de la variable puede ser Persona o Empleado

# Tipos en Java

```
Empleado jefe = new Empleado();  
Persona p = jefe;
```

Se puede asignar un objeto instancia de una subclase a cualquier variable del tipo de la superclase



Empleado es miembro de ambas clases: Persona y Empleado  
El tipo de Empleado es tanto Empleado como Persona

```
Persona p = new Persona();
```

~~Empleado jefe = p;~~

No se puede asignar a una variable del tipo de la subclase un objeto instancia de la superclase



# Por qué existen los elementos Genéricos?

- Al seleccionar un elemento de una colección, se debe convertir al tipo de elemento que se almacena en dicha colección.
- El compilador no comprueba que el **cast** sea del mismo tipo de la colección, por lo que el **cast** puede fallar en tiempo de ejecución.
- Los genéricos proporcionan una forma de determinar el tipo de una colección para el compilador, por lo que se puede comprobar

# Utilización de genéricos

Eliminar las palabras de 4 letras de una colección cualquiera

```
static void eliminar(Collection c) {  
    for (Iterator i = c.iterator(); i.hasNext(); )  
        if (((String) i.next()).length() == 4)  
            i.remove();  
}
```

*Si utilizamos genéricos:*

```
static void eliminar(Collection<String> c) {  
    for (Iterator<String> i = c.iterator(); i.hasNext(); )  
        if (i.next().length() == 4)  
            i.remove();  
}
```



# Validando datos de entrada

- Las aserciones no son buen mecanismo para validar los datos de entrada de un método público o programa
- Las aserciones, cuando no se cumplen, cortan el programa
  - Por eso las usamos para testing (antes de entregar el programa al usuario)
- ¿Qué otro mecanismo tenemos para detectar errores y que el programa pueda restablecerse?



# Temario

- Herencia en Java
- Visibilidad en Java
- Interfaces
- Tipo estático y dinámico
- Excepciones





# ¿Cómo avisar que hubo un error?

- En algunos casos se puede devolver un valor especial
  - Ejemplo: El método `indexOf(cad)` de `String` devuelve `-1` cuando no encuentra `cad` en el `String` llamado
- En otros casos no es posible
  - Ejemplo: `dividir(Entero) → Entero`  
No hay ningún valor entero para indicar que hubo un error si el segundo parámetro es `0`



# Excepciones

- Un programa **correcto** es aquel que actúa de acuerdo a su **especificación**.
- Un programa **confiable** es correcto y además tiene un comportamiento previsible, es decir actúa razonablemente no sólo en situaciones normales sino también en circunstancias anómalas, como por ejemplo fallas de hardware.
- Desde el punto de vista de la aplicación las situaciones consideradas normales dependen del diseñador que analiza el problema.



# Excepciones

- Una excepción es un **evento anormal** durante la ejecución que puede provocar que una operación falle.
- Un evento anormal no necesariamente es catastrófico y con frecuencia puede repararse de modo tal que la ejecución continúe.
- El software que previene este tipo de circunstancias se dice "tolerante a las fallas".



# Excepciones

- Se puede **reparar** la falla, **capturando** la excepción y alcanzando un estado que permita continuar la ejecución.
- A veces, el manejo de la excepción se reduce a mostrar un mensaje, porque la situación no es recuperable.
- En ese caso la operación falla y probablemente el programa se aborta.



# Excepciones

- Una excepción es una situación anormal o poco frecuente que requiere ser **capturada** y **manejada** adecuadamente.
- Las excepciones pueden ser **predefinidas por el lenguaje** o **definidas por el programador**.
- Las excepciones predefinidas son más generales y son capturadas **implícitamente** por alguna operación predefinida.



# Excepciones

- Ejemplos típicos de excepciones detectadas implícitamente son:
  - **ArithmeticException** División por 0, señalizado por la operación /
  - **ArrayIndexOutOfBoundsException** El acceso fuera de rango dentro de un arreglo, señalizado por la operación de subindexación
  - **NullPointerException** Se intenta acceder a un servicio de una variable de tipo clase pero esta no está asociada a un objeto.



# Excepciones

- En los ejemplos anteriores cuando se captura la excepción aparece un mensaje de error y el programa termina anormalmente (aborta)
- La idea es que **el programador establezca un manejador** que especifique las acciones a realizar cuando se captura una excepción.
- La acción puede ser algo tan simple como mostrar un mensaje de error diferente al predefinido o puede de alguna manera ‘salvar’ la situación anormal para reparar la excepción.



# Excepciones

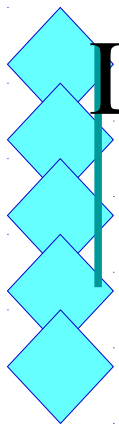
- Organizar un programa en secciones para el caso normal y para el caso excepcional
- Implementar los programas incrementalmente
  - Codificar y probar el código para la operación normal primero
  - Después agregar el código para el caso excepcional
- Tener en cuenta: las excepciones simplifican el desarrollo, prueba y mantenimiento, pero no se debe abusar de ellas.





# Terminología

- Lanzar o disparar una excepción (**throwing**)
- Manejar o capturar una excepción (**handling/catching**)
  - Se responde a una excepción ejecutando una parte del programa escrita específicamente para esa excepción
- El caso normal es manejado en un bloque **try**
- El caso excepcional es manejado en un bloque **catch**
- El bloque catch recibe un parámetro de tipo **Exception** (generalmente llamado e)
- Si se dispara una excepción, la ejecución del bloque try se interrumpe y el control pasa al bloque catch cercano al bloque try



# La terna **try-throw-catch**

## Organización básica del código

**try**

{

<código a tratar>  
*obj.metodoAux(...)*  
<más código>

}

**catch(Exception e)**

{

<código de manejo de la excepción>

}

<posiblemente más código>

...

*if(condición de prueba)*

**throw new Exception**  
**("Mensaje de error");**

...



# Flujo de Programa `try-throw-catch`

- Bloque Try

- Las sentencias encerradas en el bloque Try son las sentencias protegidas (bloque protegido).
- En el método `metodoAux`, si la condición es `true`, se lanza la excepción
  - Se corta la ejecución de `metodoAux`, y el control pasa al bloque `catch` después del bloque `try`
- Si la condición es `false`
  - La excepción no se lanza, el método se ejecuta con normalidad
  - Las sentencias restantes en el bloque `try` (aquellas que siguen el `throw` condicional) son ejecutadas

- Bloque Catch

- Se ejecuta si una excepción es lanzada. Es el bloque manejador de la excepción
- Puede terminar la ejecución con una sentencia `exit` (aborta el programa)
- Si no hace `exit`, la ejecución se reanuda después del bloque `catch`

- Las sentencias después del bloque `Catch` se ejecutan tanto si la excepción fue lanzada o no

# Ejemplo de manejo de excepciones

bloque try

sentencia throw en el método dispara la excepción

bloque catch

```
/** caramelos por persona */  
  
int contCaramelos=0, personas=0;  
double caramelos=0.0;  
  
try  
{  
    System.out.println("Ingrese cantidad de caramelos ");  
    contCaramelos = TecladoIn.readLineInt( );  
  
    contPersonas = ingresarPersonas( );  
  
    caramelosXPersona = (double)contCaramelos/  
        (double)contPersonas;  
    System.out.println(contCaramelos + " caramelos");  
    System.out.println(contPersonas + " personas");  
    System.out.println(" Hay " + caramelosXPersona  
        + " caramelos por persona");  
}  
  
catch(Exception e)  
{  
    System.out.println(e.getMessage( ));  
    System.out.println(" Ir a buscar personas");  
}  
  
System.out.println(" Fin del programa.");
```

# Ejemplo de manejo de excepciones

bloque try

sentencia throw en el método dispara la excepción

bloque catch

```
/** caramelos por persona */  
  
int contCaramelos=0, personas=0;  
double caramelos=0.0;  
  
try  
{  
    System.out.println("Ingrese cantidad de caramelos ");  
    contCaramelos = TecladoIn.readLineInt( );  
  
    contPersonas = ingresarPersonas( );  
  
    public double ingresarPersonas() throws Exception{  
        System.out.println(" Ingrese el nro.de personas:");  
        contPer = TecladoIn.readLineInt( );  
  
        if (contPer < 1)  
            throw new Exception("Excepcion: no hay personas");  
    }  
}  
  
catch(Exception e)  
{  
    System.out.println(e.getMessage( ));  
    System.out.println(" Ir a buscar personas");  
}  
  
System.out.println(" Fin del programa.");
```



# Más acerca del Bloque **catch**

- **Exception** es la clase base de todas las excepciones
- El bloque **catch** no es una definición de método (aunque parece similar)
- Cada excepción hereda el método `getMessage`
  - Este método carga el string dado al objeto-excepción cuando fue lanzada la excepción, ej.
    - `throw new Exception("Mensaje cargado");`
- Un bloque **catch** se aplica sólo sobre el bloque **try** que inmediatamente lo precede
- Si ninguna excepción es lanzada, el bloque **catch** es ignorado

# Definiendo clases de excepción propias

```
public class ExcepcionDividePorCero extends Exception
{
    public ExcepcionDividePorCero ()
    {
        super("Dividiendo por Cero!");
    }
    public ExcepcionDividePorCero (String mensaje)
    {
        super(mensaje);
    }
}
```

- Extiende (hereda) la clase Exception ya definida
- El único método que necesitamos definir es el constructor
  - Incluye un constructor que toma un argumento String
  - También un constructor por defecto con un mensaje string por defecto

# Usando la clase **ExcepcionDividePorCero**


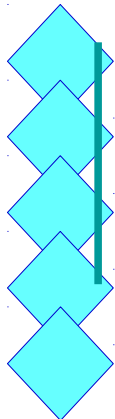
```
public void hacerEsto( ) {
    try
    {
        System.out.println("Ingrese numerador:");
        this.numerador = TecladoIn.readLineInt( );
        System.out.println("Ingrese denominador:");
        this.denominador = TecladoIn.readLineInt( );

        if (this.denominador == 0)
            throw new ExcepcionDividePorCero("Error:Division por 0" );

        double cociente =
            (double)this.numerador/(double)this.denominador;
        System.out.println(this.numerador + "/" +
            this.denominador + " = " + cociente);
    }

    catch(ExcepcionDividePorCero e)
    {
        System.out.println(e.getMessage( ));
        System.out.println("El calculo no fue realizado");
    }
}
```





# Excepciones múltiples y bloques **catch** en un Método

- Un método puede lanzar más de una excepción
- Los bloques **catch** inmediatamente después del bloque try son analizados en secuencia para identificar el tipo de excepción
- El primer bloque catch que maneja ese tipo de excepción es el único que se ejecuta
- Se deben colocar los bloques catch en orden de especificidad: los más específicos primero

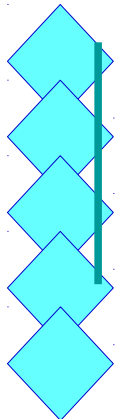
```
catch (ExcepcionDividePorCero e) {  
    // que hace si ocurre excepción divide por cero  
}  
    catch (Exception e) {  
        // aquí lo que hace si ocurre otra excepción  
    }
```



# El Bloque **finally**

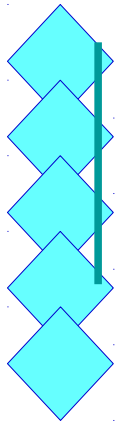
- Se puede agregar un bloque **finally** después de los bloques try/catch
- El bloque **finally** se ejecuta sin importar si el bloque catch se ejecuta
- La organización del código utilizando el bloque finally será:

```
try {bloque}
catch (...) {bloque}
finally
{
    <Código a ejecutarse se dispare o no una
    excepción>
}
```



# Tres Posibilidades para un bloque **try-catch-finally**

- El bloque try se ejecuta hasta el final sólo si ninguna excepción es lanzada.
  - El bloque finally se ejecuta después del bloque try.
- Una excepción es lanzada en el bloque try y atrapada en el manejo del bloque catch.
  - El bloque finally se ejecuta después del bloque catch.
- Una excepción es lanzada en el bloque try y no existe match en el bloque catch.
  - El bloque finally se ejecuta antes de que el método termine.
  - El código que está después del bloque catch pero no en el bloque finally no sería ejecutado en esta situación.



# Resumen

- Una excepción es un objeto descendiente de la clase Exception
- El manejo de excepciones permite diseñar código para los casos normales separados de los casos excepcionales
- Podemos usar las clases de excepción predefinidas o definir la nuestra
- Las excepciones pueden ser lanzadas por:
  - Ciertas sentencias Java
  - Los métodos de las librerías de clase
  - Un bloque try
  - Una definición de método sin bloque try, pero la invocación al método está ubicada dentro de un bloque try