Tema 4

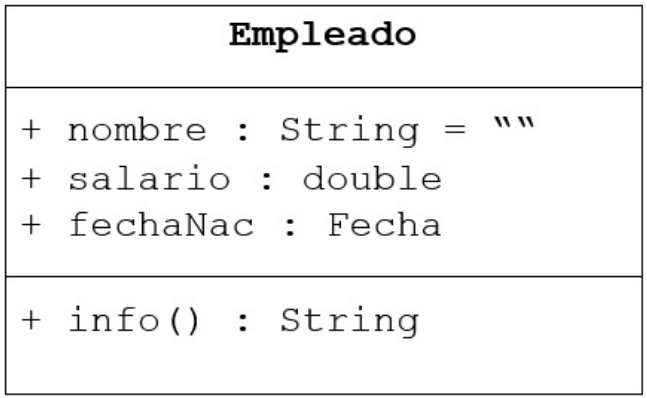
Herencia

M. En C. Laura Marina Bernal Zavala

lmbz@lasallistas.org.mx

Ejemplo para Definir Herencia

- La clase Empleado:




```
public class Empleado{
    public String nombre="";
    public double salario;
    public Fecha fechaNac;

    public String info()
    { ... }
}
```

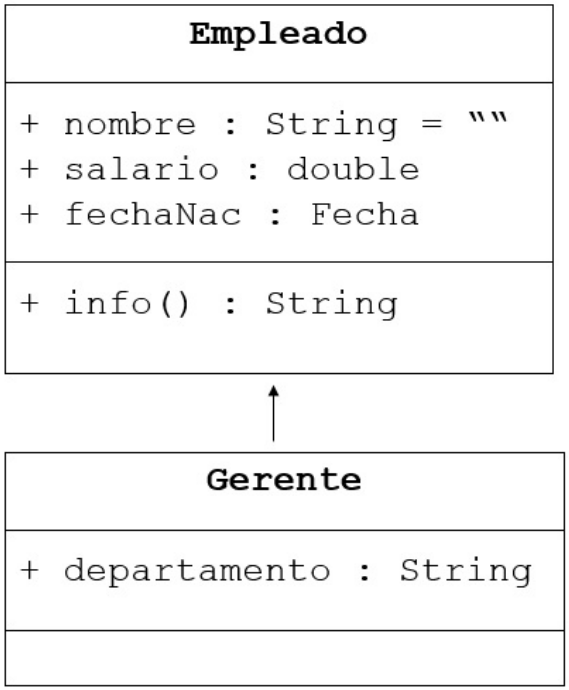
Ejemplo para Definir Herencia

- La clase Gerente:

Gerente
+ nombre : String = "" + salario : double + fechaNac : Fecha + departamento : String
+ info() : String

```
public class Gerente{
    public String nombre="";
    public double salario;
    public Fecha fechaNac;
    public String departamento;

    public String info()
    { ... }
}
```

Ejemplo para Definir Herencia




```
public class Empleado{
    public String nombre="";
    public double salario;
    public Fecha fechaNac;

    public String info()
    { ... }
}

public class Gerente extends
                          Empleado{
    public String departamento;
}
```

Herencia

- La herencia es la técnica por la que pueden crearse nuevas clases en términos de otras ya existentes
- La relación que se establece entre clases es de tipo "es un", por ejemplo, "un gerente es un empleado"
- La herencia puede aplicarse para generalizar o especializar clases

Herencia

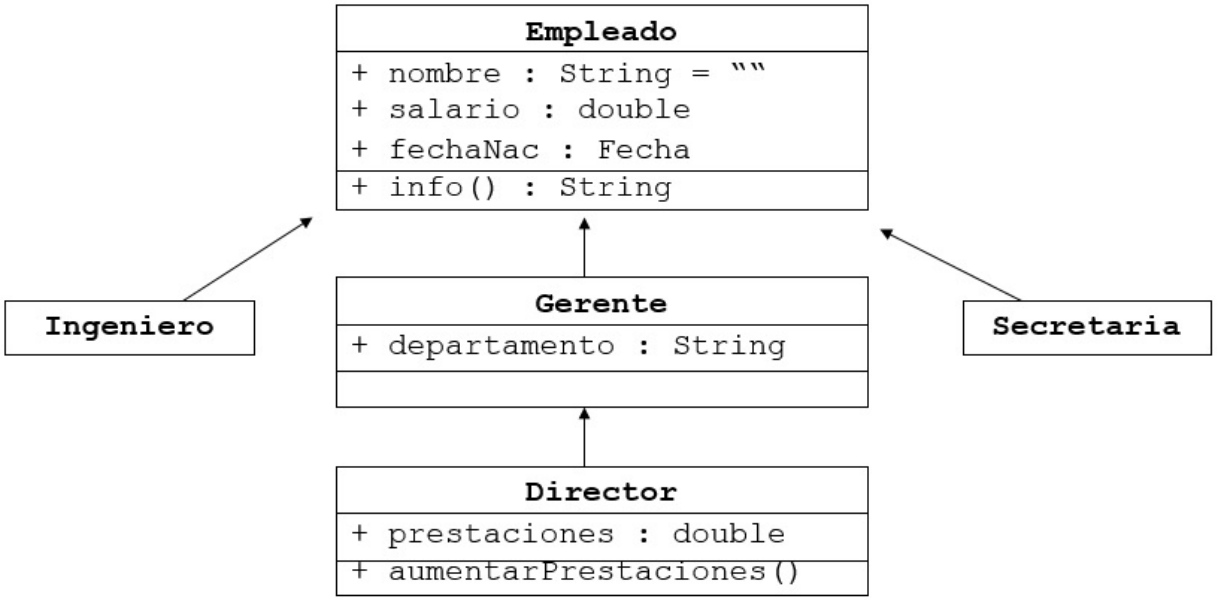
- A la clase padre se le conoce como superclases y a las hijas como subclases
- Las subclases heredan todos los métodos y atributos de la superclase, menos los constructores

Herencia Simple

- Cuando una clase hereda de una sola clase, se habla de herencia simple
- La herencia simple hace el código más fácil de modificar y más confiable
- La herencia múltiple puede simularse utilizando el concepto de interfaces
- Síntaxis de la herencia simple:

```
<modificador>* class <nombre_clase>
  [extends <superclase>]{
    <instrucción>*
}
```

Herencia Simple



Modificadores de Acceso

- Tanto los atributos como los métodos pueden llevar uno de estos modificadores
- Marcan dónde son accesibles directamente

Modificador	Misma clase	Mismo paquete	Subclase	Universo
public	sí	sí	sí	sí
protected	sí	sí	sí	no
"paquete" o "default"	sí	sí	no	no
private	sí	no	no	no

Sobreescritura de Métodos

- Una subclase puede modificar el comportamiento de su superclase
- Una subclase puede declarar un método con diferente funcionalidad que en el padre, pero con el mismo:
  - nombre
  - tipo de regreso
  - lista de argumentos

Sobreescritura de Métodos

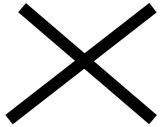
```
public class Empleado {
   protected String nombre;
   protected double salario;
   protected Fecha fechaNac;
   public String info() {
     return "Nombre: " + nombre + "\ n" + "Salario: " +
  salario;
   }
}

public class Gerente extends Empleado {
   protected String departamento;
   public String info() {
     return "Nombre: " + nombre + "\ n" +
             "Salario: " + salario +  "\n" +
             "Gerente de: " + departamento;
   }
}
```

Reglas de Sobreescritura

- Un método que sobreescribe a otro de la superclase debe cumplir las siguientes reglas:
  - El tipo de regreso debe ser idéntico que el del método que sobreescribe
  - No puede ser menos accesible que el de la superclase

Reglas de Sobreescritura

```
public class Padre {
   public void método1() { ... }
}
public class Hijo extends Padre {
   private void método1() { ... }
}
public class Prueba  {
   public void método2() {
      Padre p1 = new Hijo();
      p1.método1();
   }
}
```



La Referencia `super`

- `super` es utilizado para hacer referencia a la superclase
- Pueden accesarse tanto atributos como métodos

La Referencia super

```
public class Empleado {
   protected String nombre;
   protected double salario;
   protected Fecha fechaNac;
   public String info() {
      return “Nombre: “ + nombre + “\ n” +
  “Salario: “ + salario;
   }
}

public class Gerente extends Empleado {
   protected String departamento;
   public String info() {
      return  super.info() + "\ n“ + “Gerente de:
  “+departamento;
   }
}
```

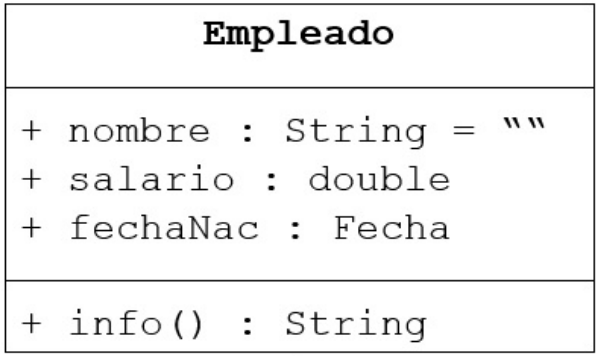
Polimorfismo

- Polimorfismo significa muchas formas
- El polimorfismo puede darse a nivel de variables y métodos
- Un método es polimórfico si se ejecutan instrucciones diferentes dependiendo del objeto sobre el que es llamado
- Una variable es polimórfica si puede hacer referencia a objetos de diferentes clases
- Un objeto tiene una sola forma

Métodos Polimórficos

- El método que se manda llamar depende del objeto sobre el que es llamado
- Al compilar sólo se verifica que el método exista en la clase a la que pertenece a referencia al objeto
- Al correr se verifica el tipo del objeto y se manda llamar el método adecuado

Métodos Polimórficos




```
Empleado e;
e = new Gerente();
e.info();
```

Verifica que en Empleado exista el método, pero realmente llama al método de Gerente

Variables Polimórficas

- Ejemplo:

```
Empleado e;

e=new Empleado();

e=new Gerente();

e=new Director();
```

La variable "e" está declarada como una referencia a objetos de tipo empleado y gracias al polimorfismo puede hacer referencia también a gerentes y directores porque son tipos específicos de empleados

Colecciones Heterogéneas

- Las colecciones de referencias a objetos del mismo tipo se conocen como colecciones homogéneas:

```
Fecha mes[]=new Fecha[31];
mes[0]=new Fecha(1,1,2000);
mes[1]=new Fecha(2,1,2000);
mes[2]=new Fecha(3,1,2000);
....
```

Colecciones Heterogéneas

- Las colecciones de referencias a objetos de diferentes tipos se conocen como heterogéneas:

```
Empleado staff[]=new Empleado[10];
staff[0]=new Gerente();
staff[1]=new Secretaria();
staff[2]=new Ingeniero();
....
```

Argumentos Polimórficos

- Dado que un gerente es un empleado:

```
//dentro de la clase Empleado
public double
  calcularImpuesto(Empleado e){
   ....
}
//en otro punto de la aplicación
Gerente g = new Gerente();
double
  impuesto1=calcularImpuesto(g);
Director d = new Director();
double
  impuesto2=calcularImpuesto(d);
```

El Operador `instanceof`

- El operador `instanceof` regresa verdadero o falso dependiendo de si el objeto referido es o no del tipo por el que se pregunta, por ejemplo:

```
public void
  calcularImpuesto(Empleado e){
   if(e instanceof Gerente){
       ...
   }

   if(e instanceof Ingeniero){
       ...
   }
}
```

Conversión de Objetos

- Para restaurar toda la funcionalidad de un objeto debe realizarse una conversión explícita
- A tiempo de compilación sólo se revisan los tipos de las referencias
- A tiempo de corrida se verifica que el tipo del objeto corresponda con el tipo de la conversión

```
public void calcularImpuesto(Empleado
  e){
   if(e instanceof Gerente){
      Gerente g = (Gerente) e;
      System.out.println("Gerente de " +

   g.regresarDepartamento() );
    }
}
```

Sobrecarga de Métodos

- Se dice que un método está sobrecargado cuando existen varios métodos con el mismo nombre, pero con diferente número, tipo u orden de argumentos
- El tipo de regreso no importa
- Ejemplo:

```
public    void   imprimir()
private   void   imprimir(int i)
public    void   imprimir(float f)
public    void   imprimir(float f, int i)
protected void   imprimir(int i, float f)
```

Sobrecarga de Constructores

- Al igual que los métodos, los constructores pueden sobrecargarse
- Ejemplo:

```
public Empleado()
public Empleado(String nombre)
public Empleado(String nombre, double
  salario)
public Empleado(String nombre, double
  salario, Fecha nac)
public Empleado(String nombre, Fecha nac)
```

- Dentro de un constructor se puede utilizar la referencia `this( )` para llamar a otro constructor (debe ir como primer línea)

Sobrecarga de Constructores

```
1 public class Empleado {
2    private static final double SALARIO_BASE
   = 10000.00;
3    private String nombre;
4    private double salario;
5    private Fecha fechaNac;
6
7    public Empleado( String nombre, double
   salario,
8                     Fecha nac) {
9      this.nombre = nombre;
10     this.salario = salario;
11     this.fechaNac = nac;
12   }
```

Sobrecarga de Constructores

```
13   public Empleado( String nombre, double
salario) {
14       this(nombre, salario, null);
15   }
16   public Empleado( String nombre, Fecha nac) {
17       this(nombre, SALARIO_BASE, nac);
18   }
19   public Empleado( String nombre) {
20       this(nombre, SALARIO_BASE);
21   }
22   // más código...
23 }
```

Invocación de Constructores

- Para invocar un constructor de la superclase, debe utilizarse `super` (como primer línea)
- Se puede llamar un constructor en especial indicando la lista de argumentos
- Si no existe `this` o `super` en un constructor, entonces el compilador añade automáticamente una llamada al constructor por default de la superclase (`super( )`), en caso de que la superclase no tenga ese constructor, marca un error

Invocación de Constructores

```
1 public class Empleado {
2    private static final double SALARIO_BASE
   = 10000.00;
3    private String nombre;
4    private double salario;
5    private Fecha fechaNac;
6
7    public Empleado( String nombre, double
   salario,
8                     Fecha nac) {
9      this.nombre = nombre;
10     this.salario = salario;
11     this.fechaNac = nac;
12   }
```

Invocación de Constructores

```
13   public Empleado( String nombre, double
salario) {
14       this(nombre, salario, null);
15   }
16   public Empleado( String nombre, Fecha nac)
{
17       this(nombre, SALARIO_BASE, nac);
18   }
19   public Empleado( String nombre) {
20       this(nombre, SALARIO_BASE);
21   }
22   // más código...
23 }
```

Invocación de Constructores

```
1  public class Gerente extends Empleado {
2   private String departamento;
3
4   public Gerente(String nombre, double salario,
     String depto) {
5       super(nombre, salario);
6       departamento = depto;
7   }
8   public Gerente(String nombre, String depto) {
9       super(nombre);
10      departamento = depto;
11  }
12  public Gerente(String depto) {
13    // inserta una llamada a super(), pero no
     existe
14      departamento = depto;
15  }
16 }
```

Construcción de Objetos

1. Se aparta memoria para el nuevo objeto
2. Se llama al constructor según los argumentos
3. Si existe `this`, llamar recursivamente e ir al paso 6
4. Llamar recursivamente a `super` (implícito o explícito)
5. Ejecutar las inicializaciones explícitas de variables
6. Ejecutar el cuerpo del constructor actual

Construcción de Objetos

```
public class Object {
   public Object() {}
}
public class Empleado extends Object {
   private String nombre;
   private double salario = 10000.00;
   private Fecha fechaNac;
   public Empleado(String n, Fecha f) {
      // super(); implícito
      nombre = n;
      fechaNac = f;
   }
```

Construcción de Objetos

```
    public Empleado(String n) {
      this(n, null);
    }
}

public class Gerente extends Empleado {
    private String departamento;
    public Gerente(String n, String d) {
       super(n);
       departamento = d;
    }
}
```

La Clase `Object`

- La clase `Object` es la raíz de la jerarquía de clases en Java
- Por default, todas las clases que existen y las que uno crea son subclases de `Object`
- Dentro de esta clase existen métodos como `equals()` y `toString()` que generalmente se sobreescriben en las subclases

Operador ==, Método `equals`

- El operador == determina si dos referencias son idénticas
- El método `equals()` en la clase `Object` utiliza el operador ==, pero puede sobreescribirse para comparar si dos objetos tienen los mismos valores en sus atributos

Método `toString()`

- Convierte un objeto en String
- Se llama automáticamente al concatenar cadenas
- Se sobreescribe este método para proveer información de los objetos

Sobreescritura de `toString()`

```
public class Empleado {
    protected String nombre;
    protected double salario;
    protected Fecha fechaNac;

    public String toString() {
      return "Nombre: " + nombre + "\ n" +
             "Salario: " + salario + "\n" +
             "Fecha de nacimiento: " + fechaNac ;
    }
}
```

Clases Envolventes

- Las clases envolventes (wrapper) sirven para tratar a los tipos primitivos como objetos

Tipos Primitivos	Clase Envolvente
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

FIN

Tema 4

Herencia

M. En C.Laura Marina Bernal Zavala

lmbz@lasallistas.org.mx