



# Lenguajes Basados en Objetos

---

## **Introducción**

Lic.Gerardo Rossel



# Ejemplo

---

## **ObjectType Cell is**

```
var contents: Integer;  
method get(): Integer;  
method set(n: Integer);
```

```
end;
```

## **Object cell: Cell is**

```
var contents: Integer := 0;  
method get(): Integer is return self.contents end;  
method set(n: Integer) is self.contents := n end;
```

```
end;
```



# Generación de objetos

---

```
procedure newCell(m: Integer): Cell is  
  object cell: Cell is  
    var contents: Integer := m;  
    method get(): Integer is return self.contents end;  
    method set(n: Integer) is self.contents := n end;  
  end;  
  return cell;  
end;  
  
var cellInstance: Cell := newCell(0)
```



# Prototipos y Clones

---

```
var cellClone: Cell := clone cellInstance;
```

```
cellClone.contents := 3;
```

```
cellClone.get :=
```

```
    method ():Integer is
```

```
        if self.contents < 0 then
```

```
            return 0
```

```
        else
```

```
            return self.content
```

```
        end;
```

```
    end;
```



# Ejemplo: method update

---

**ObjectType** ReCell **is**

```
var contents: Integer;  
method get(): Integer;  
method set(n: Integer);  
method restore();
```

**end;**

**Object** reCell: ReCell **is**

```
var contents: Integer := 0;  
method get(): Integer is return self.contents end;  
method set(n: Integer) is  
    let x = self.get();  
    self.restore := method() is self.contents := x end;  
    self.contents := n;
```

**end;**

```
method restore() is self.contents := 0 end;
```

**end;**



## Herencia: Embeber o Delegar

---

- Hay dos aspectos ortogonales en la herencia en los lenguajes basados en prototipos:
  - Obtener los atributos del **Donante**
  - Incorporar los atributos al **Host**
- Categorías:
  - Implícita o explícita.
  - Embebido o delegado.



# Interpretación embebida de la herencia

---

aCell



contents	0
get	Código de get
set	Código de set

aReCell



contents	0
backup	0
get	Código nuevo de get
set	Código nuevo de set
restore	Código de restore



# Interpretación embebida de la herencia: versión explícita

---

**Object** cell:Cell **is**

**var** contents:Integer := 0;

**method** get():Integer **is return** self.contents **end**;

**method** set(n:Integer) **is** self.contents := n **end**;

**end**;

**Object** reCellExp:ReCell **is**

**var** contents:Integer := 0;

**var** backup: Integer := 0;

**method** get():Integer **is**  
    **return embed** cell.get();

**end**;

**method** set(n:Integer) **is**  
    **self**.backup := **self**.contents;  
    **embed** cell.set(n);

**end**;

**method** restore() **is self**.contents := **self**.backup **end**;

**end**;

**method** get **copied from** cell;



# Interpretación embebida de la herencia: versión implícita

---

**Object** cell: Cell **is**

**var** contents: Integer := 0;

**method** get(): Integer **is return** self.contents **end**;

**method** set(n: Integer) **is** self.contents := n **end**;

**end**;

**Object** reCellImp: ReCell **extends** cell **is**

**var** backup: Integer := 0;

**override** set(n: Integer) **is**

**self**.backup := **self**.contents;

**embed** cell.set(n);

**end**;

**method** restore() **is self**.contents := **self**.backup **end**;

**end**;



## Interpretación embebida de la herencia: versión implícita

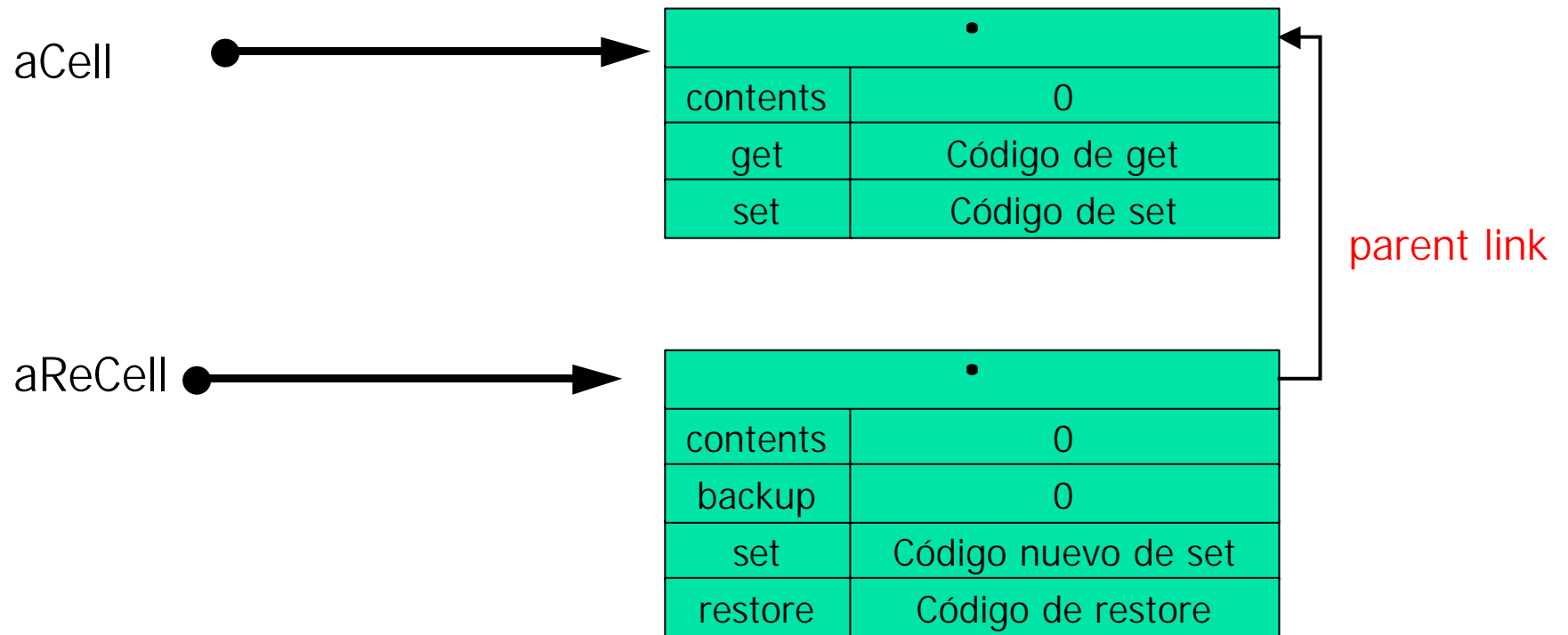
---

Alternativamente se puede definir un objeto equivalente mediante una extensión pura seguida de un method update.

```
Object reCellImp1: ReCell extends cell is  
    var backup: Integer := 0;  
    method restore() is self.contents: = self.backup end;  
end;
```

```
reCellImp1.set :=  
    method(n: Integer) is  
        self.backup := self.contents;  
        self.contents: = n;  
    end;
```

# Delegación





# Delegación (implícita)

---

```
object cell: Cell is  
    var contents: Integer := 0;  
    method get(): Integer is return self.contents end;  
    method set(n: Integer) is self.contents := n end;  
end;  
  
object reCellImp': ReCell child of cell is  
    var backup: Integer := 0;  
    override set(n: Integer) is  
        self.backup := self.contents;  
        delegate cell.set(n);  
    end;  
    method restore() is self.contents := self.backup end;  
end;
```



# Delegación (implícita)

---

```
object reCellImp: ReCell child of cell is  
  override contents: Integer := cell.contents;  
  var backup: Integer := 0;  
  override set(n: Integer) is  
    self.backup := self.contents;  
    delegate cell.set(n);  
  end;  
  method restore() is self.contents := self.backup end;  
end;
```



# Delegación (explícita)

---

```
object reCellImp: ReCell is  
  var contents: Integer := cell.contents;  
  var backup: Integer := 0;  
  method get(): Integer is return delegate cell.get() end;  
  method set(n: Integer) is  
    self.backup := self.contents;  
    delegate cell.set(n);  
  end;  
  method restore() is self.contents := self.backup end;  
end;
```



# Embeber vs. Delegar

---

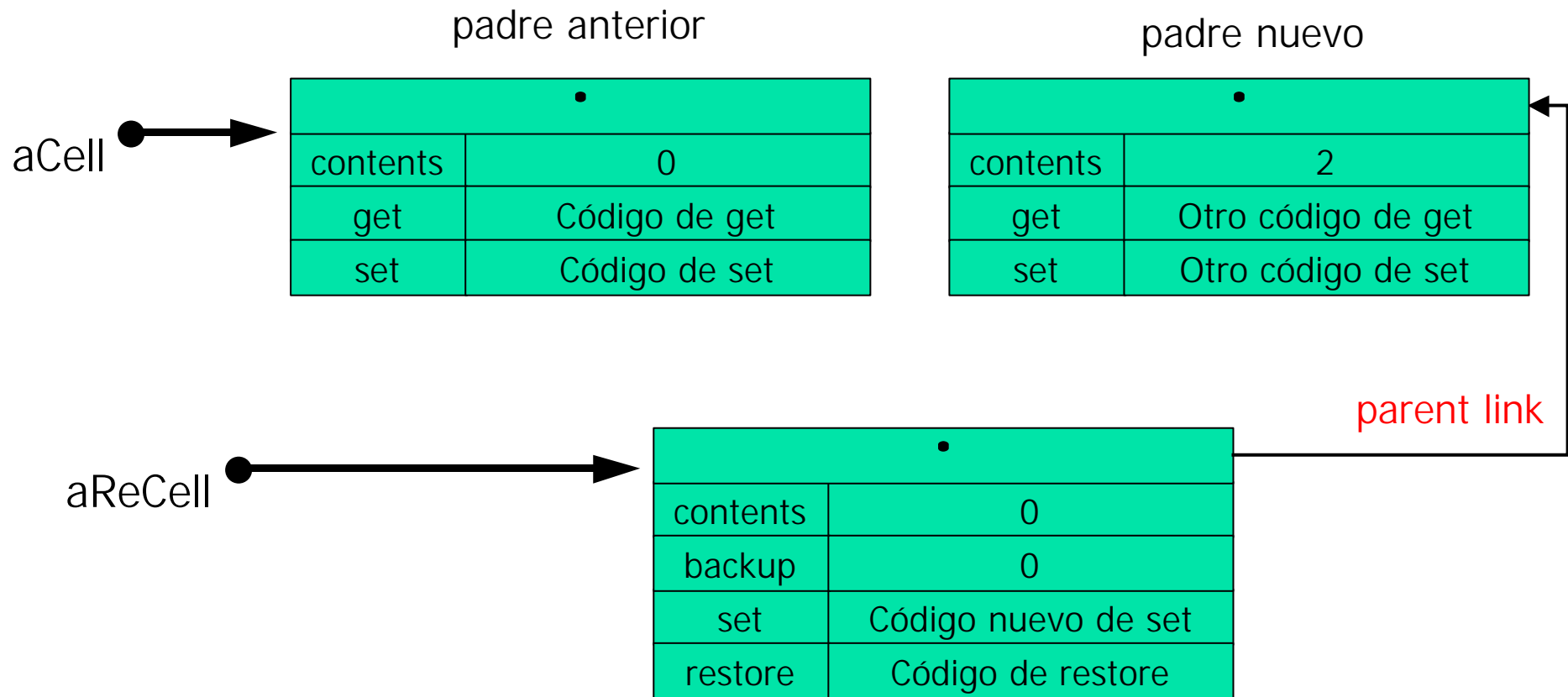
- **Delegación:**

- En delegación, los donantes pueden contener campos que pueden ser actualizados. Los cambios son vistos por los hosts.
- Idem los métodos.
- Si se permite reemplazar un parent link con otro en un objeto, resulta que todos los que hereden de dicho objeto cambian su comportamiento.

- **Embeber:**

- Se evita crear una telaraña dinámica de dependencias.
- Existen lenguajes embedding-based (ek. Kevo, Omega) que permiten la realización de cambios extendidos sin jerarquías de donates.

# Herencia Dinámica y Mode-Switching





# Herencia Dinámica y Mode-Switching

---

Utilizando reparenting technique, en lenguajes basados en delegación

```
cellNuevo: Cell
```

```
....
```

```
reparent reCellImp to cellNuevo;
```

Usando method-update

```
reCellImp.get :=
```

```
  method() : Integer is return embed cellNuevo.get end;
```

```
reCellImp.set :=
```

```
  method(n: Integer) is
```

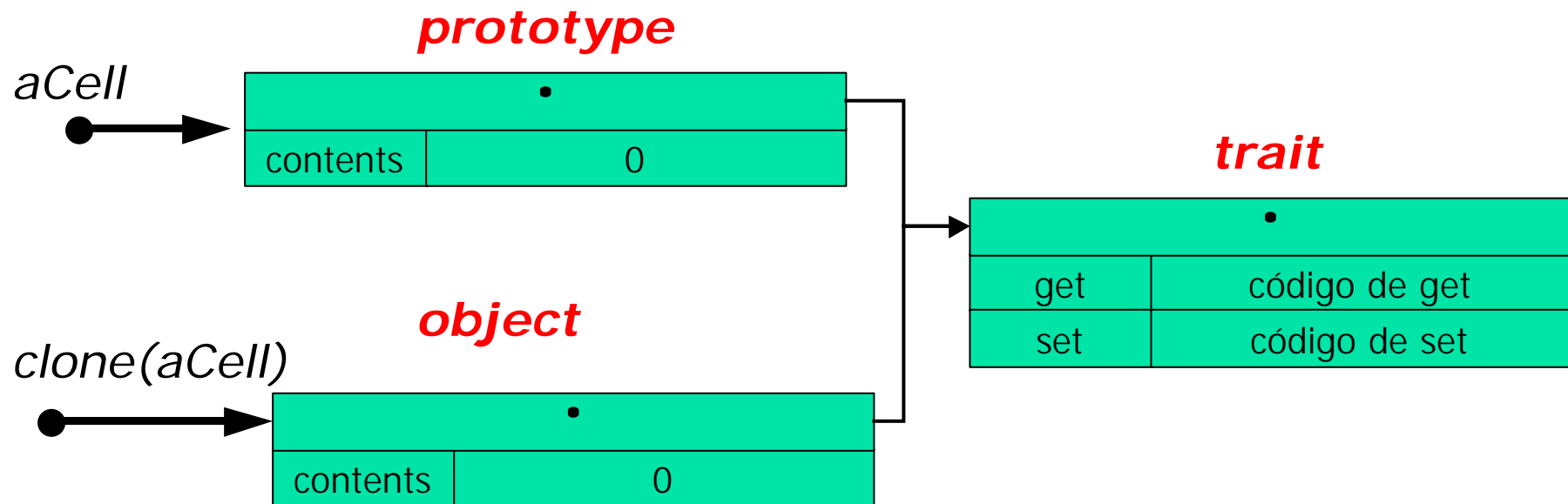
```
    self.backup := self.contents;
```

```
    embed cellNuevo.set(n);
```

```
  end;
```

# De prototipos a clases

- traits objects
- prototype objects
- normal objects





# Algunas conclusiones

---

- El logro de los lenguajes basados en objetos es hacer claro que las clases son sólo una de las formas posibles de generar objetos con propiedades comunes.
- Mode-switching (como opuesto a herencia dinámica irrestricta) puede ser tipificado más fácilmente.
- La partición traits-prototipos en lenguajes basados en delegación se ve igual que una técnica de implementación para clases.



# Bibliografía

---

- **A theory of Objects** Martin Abadi Luca Cardeli. Monographs in Computer Science- Springer 1996
- **A shared view of sharing: the treaty of Orlando** Stein L.A. – Lieberman H- Ungar D. Object Oriented concepts, applications and databases Addison-Wesley 1988
- **Kevo, a prototype-based object oriented language based on concatenation and module operation** Taivalsaari 19932 Report LACIR-92-02 Univerity of Victoria
- **Classes versus prototypes in object-oriented languages**. Borning. Proceedings of the ACM/IEEE Fall Joint Computer Conference 1986
- **An efficient implementation of Self a dynamically-typed object-oriented language based on prototypes**. Chambers, Ungar, Lee Proceedings of the ACM Conference OOPSLA 1989