

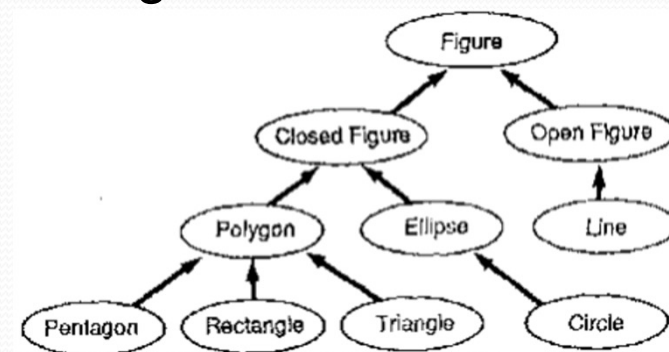
Herencia

Herencia

- **Herencia.** Una forma de crear nuevas clases basadas en clases existentes, tomando sus atributos (campos) y comportamiento (métodos).
- Es una forma de compartir / reutilizar código entre dos o más clases.
- Una clase *extiende* a otra, absorbiendo sus datos y comportamiento.
- **Superclase.** Es la clase que se está extendiendo.
- **Subclase.** clase que hereda el comportamiento de la superclase.
- La subclase hereda los campos y los métodos de la superclase.

Relación es-un y jerarquía de herencia

- **Relación es-un (*is-a*).** Cada objeto de la subclase también “*es un*” (*is a*) objeto de la superclase y puede ser tratado como uno.
 - Cada figura cerrada es una figura.
 - Cada círculo es una elipse, una figura cerrada y una figura.
- **Jerarquía de herencia.** Un conjunto de clases conectadas por relaciones es-un que pueden compartir código común.



Sintaxis de herencia

```
public class Name extends Superclass {
```

- Ejemplo:

```
public class ClosedFigure extends Figure {
```

```
    ...
```

```
}
```

- Al extender Figure, cada objeto ClosedFigure:
 - a) Recibe los métodos de Figure de forma automática.
 - b) Puede ser tratado como Figure por el código cliente.



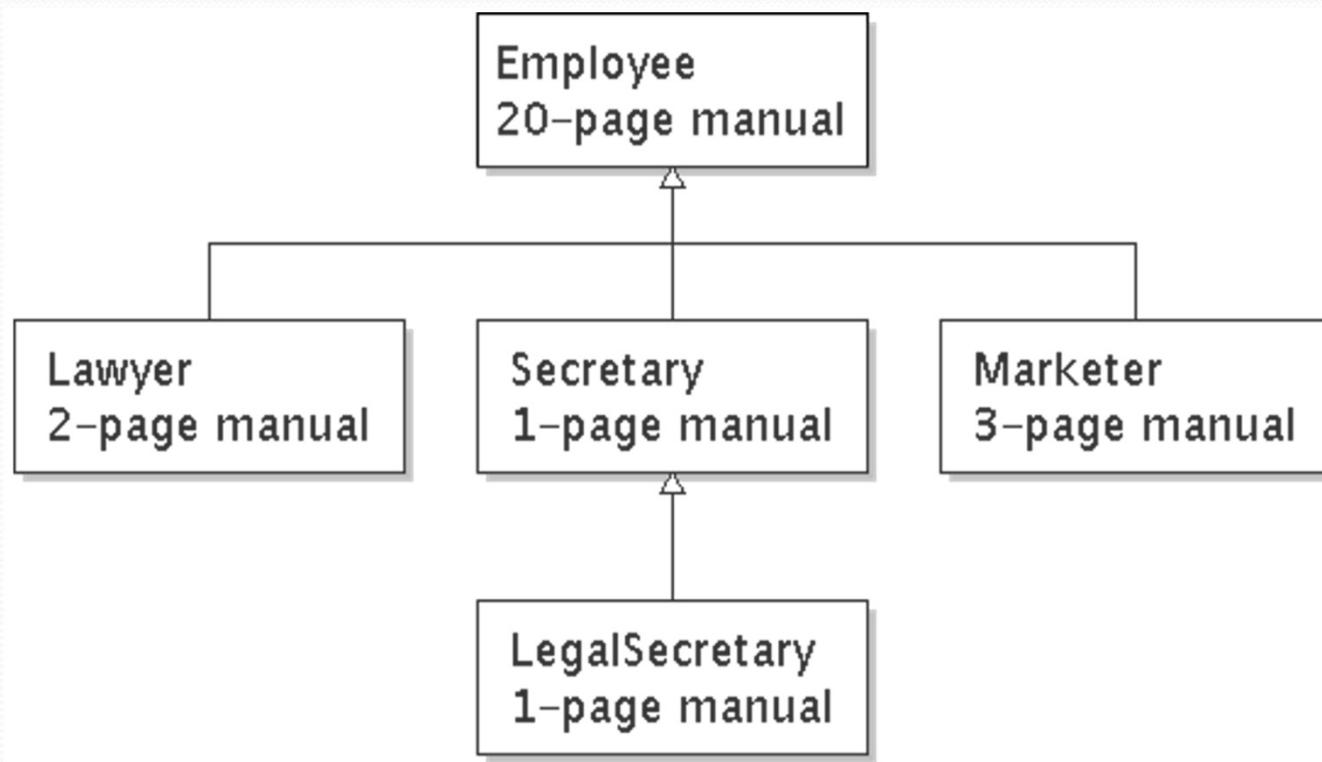
Ejemplo: firma de abogados

- Representar mediante clases los empleados de una firma de abogados.
- Hay distintos tipos de empleados: abogados, secretarios y expertos en marketing.
- Hay un tipo especializado de secretario llamado secretario legal.

Ejemplo: firma de abogados

- Reglas en común: horarios, vacaciones, beneficios, regulaciones, ...
 - Todos los empleados asisten a una orientación en común para aprender las reglas generales de la empresa.
 - Cada empleado recibe un manual de 20 páginas de reglas en común.
- También hay reglas específicas según el tipo de empleado:
 - Cada empleado recibe un manual más pequeño (1-3 páginas) de estas reglas particulares.
 - Este manual pequeño agrega algunas reglas nuevas y también cambia algunas reglas del manual grande.

Jerarquía de herencia



Separando comportamiento

- ¿Por qué no tener un manual de abogado de 22 páginas, un manual de secretario de 21 páginas, un manual de marketing de 23 páginas, etc.?
- Algunas ventajas de los manuales separados:
 - Mantenimiento: solo una actualización si cambia una regla común.
 - Localidad: encontrar rápido las reglas específicas de los abogados.

Regulaciones de empleados

- Considerar las siguientes regulaciones para empleados:
 - Los empleados trabajan 40 horas semanales.
 - Los empleados ganan \$40,000 por año, excepto los secretarios legales que ganan \$5,000 adicionales por año (\$45,000 en total) y los especialistas en marketing que ganan \$10,000 adicionales por año (\$50,000 en total).
 - Los empleados tienen 2 semanas de vacaciones pagadas por año, excepto los abogados que reciben una semana adicional (un total de 3).
 - Los empleados deben usar un formulario amarillo para solicitar una licencia, excepto los abogados que usan un formulario rosa.



Regulaciones de empleados

- Cada tipo de empleado tiene un comportamiento único:
 - Los abogados saben cómo demandar.
 - Los especialistas en marketing saben cómo hacer publicidad.
 - Los secretarios saben cómo tomar dictados.
 - Los secretarios legales saben cómo preparar documentos legales.

Clase empleado

// Clase para representar empleados en general (manual de 20 páginas).

```
public class Employee {  
    public int getHours()  
    {  
        return 40;           // 40 horas / semana  
    }  
    public double getSalary()  
    {  
        return 40000.0;      // $40,000.00 / año  
    }  
}
```


Clase empleado

```
public int getVacationDays()
{
    return 10;    // 2 semanas de vacaciones pagadas
}
public String getVacationForm()
{
    return "amarillo";    // usa la forma amarilla
}
}
```




Clase secretario redundante

- De la misma forma se puede implementar una clase para representar a los secretarios.
- Solo hay que tomar en cuenta que los secretarios saben tomar dictado.

Clase secretario redundante

// Clase redundante para representar secretarios.

```
public class Secretary {  
    public int getHours()  
    {  
        return 40;           // 40 horas / semana  
    }  
    public double getSalary()  
    {  
        return 40000.0;      // $40,000.00 / año  
    }  
}
```

Clase secretario redundante

```
public int getVacationDays()
{
    return 10;      // 2 semanas de vacaciones pagadas
}
public String getVacationForm()
{
    return "amarillo"; // usa la forma amarilla
}
```

Clase secretario redundante

// Comportamiento específico de los secretarios

```
public void takeDictation(String text)
```

```
{
```

```
    System.out.println("Tomando dictado del texto: " + text);
```

```
}
```

```
}
```


Compartir código

- takeDictation es el único comportamiento específico en Secretary.
- Sería bueno poder decir:

// Una clase para representar secretarios.

```
public class Secretary {  
    copia todo el contenido de la clase Employee;  
    public void takeDictation(String text)  
    {  
        System.out.println("Tomando dictado del texto: " + text);  
    }  
}
```

Código mejorado de secretario

// Clase para representar secretarios.

```
public class Secretary extends Employee {  
    public void takeDictation(String text)  
    {  
        System.out.println("Tomando dictado del texto: " + text);  
    }  
}
```

Código mejorado de secretario

- Ahora solo escribimos las partes únicas de cada tipo.
 - Secretary hereda los métodos `getHours`, `getSalary`, `getVacationDays` y `getVacationForm` de `Employee`.
 - El secretario agrega el método `takeDictation`.

Implementando la clase abogados

- Considerar las siguientes regulaciones para abogados:
- Los abogados tienen una semana extra de vacaciones pagadas (un total de 3).
- Los abogados usan un formulario rosa cuando solicitan licencia por vacaciones.
- Los abogados tienen un comportamiento único: saben cómo demandar.
- Problema: se quiere que los abogados hereden la *mayoría* del comportamiento de los empleados, pero se necesita reemplazar partes con un nuevo comportamiento.

Sobreposición de métodos

- **Sobreponer un método:** escribir una nueva versión de un método en una subclase que reemplaza la versión de la superclase.
- No se requiere una sintaxis especial para sobreponer un método de superclase. Simplemente se escribe una nueva versión en la subclase.
- La mayoría de los IDEs agregan o indican la anotación `@Override` al método sobrepuesto.

Sobreposición de métodos

- Ejemplo

```
public class Lawyer extends Employee {  
    // sobrepone getVacationForm en la clase Employee  
    public String getVacationForm()  
    {  
        return "rosa";  
    }  
    ...  
}
```

Clase abogado

- Con esto, ya se puede programar una clase para representar abogados (3 semanas de vacaciones, formulario rosa, pueden demandar).

Clase Lawyer

// Clase para representar abogados.

```
public class Lawyer extends Employee {  
    // sobrepone getVacationForm de la clase Employee  
    public String getVacationForm()  
    {  
        return "rosa";  
    }  
}
```


Clase Lawyer

// sobrepone getVacationDays de la clase Employee

```
public int getVacationDays()
{
    return 15;      // 3 semanas de vacaciones
}
public void sue()
{
    System.out.println("¡Te veré en la corte!");
}
}
```



Clase Marketer

- Ahora se puede programar la clase para representar a los especialistas en marketing (ganan \$10,000 extra y saben como anunciar).

Clase Marketer

// Clase para representar especialistas en marketing.

```
public class Marketer extends Employee {  
    public void advertise()  
    {  
        System.out.println("¡Actúe ahora mientras duren las existencias!");  
    }  
    public double getSalary()  
    {  
        return 50000.0;           // $50,000.00 / año  
    }  
}
```

Niveles de herencia

- Se permiten varios niveles de herencia en una jerarquía.
- Ejemplo: un secretario legal es lo mismo que un secretario regular, pero gana más dinero (\$45,000) y puede presentar informes legales.

```
public class LegalSecretary extends Secretary {  
    ...  
}
```


Clase LegalSecretary

// Clase para representar secretarios legales.

```
public class LegalSecretary extends Secretary {  
    public void fileLegalBriefs() {  
        System.out.println("¡Podría archivar todo el día!");  
    }  
    public double getSalary() {  
        return 45000.0;           // $45,000.00 / año  
    }  
}
```

Cambios al comportamiento en común

- Imaginar un cambio en toda la empresa que afecte a todos los empleados.
- Ejemplo: todos reciben un aumento de \$10,000 debido a la inflación.
 - El salario base de los empleados es ahora de \$50,000.
 - Los secretarios legales ahora ganan \$55,000.
 - Los especialistas en marketing ahora ganan \$60,000.
- Se debe modificar el código para reflejar este cambio de política.

Modificando la superclase

// Clase para representar empleados (manual de 20 páginas).

```
public class Employee {  
    public int getHours() {  
        return 40;           // trabajan 40 horas / semana  
    }  
    public double getSalary() {  
        return 50000.0;      // $50,000.00 / año  
    }  
}
```




Modificando la superclase

- ¿Eso es todo?
- Las subclases de Employee están incorrectas.
- Han sobrepuesto `getSalary` para devolver otros valores.
- Se tienen que modificar para reflejar el cambio.

Una primera solución

```
public class LegalSecretary extends Secretary {  
    public double getSalary()  
    {  
        return 55000.0;  
    }  
    ...  
}
```

Una primera solución

```
public class Marketer extends Employee {  
    public double getSalary()  
    {  
        return 60000.0;  
    }  
    ...  
}
```

- Problema: los salarios de las subclases se basan en el salario de Employee, pero el código de getSalary no refleja esto.

Llamando a métodos sobrepuestos

- Las subclases pueden llamar a los métodos que sobrepusieron usando el keyword `super`.

`super.método(parámetros)`

- Ejemplo:

```
public class LegalSecretary extends Secretary {  
    public double getSalary()  
    {  
        double salarioBase = super.getSalary();  
        return salarioBase + 5000.0;  
    }  
}
```


Clases mejoradas

```
public class Lawyer extends Employee {  
    public String getVacationForm() {  
        return "rosa";  
    }  
    public int getVacationDays() {  
        return super.getVacationDays() + 5;  
    }  
    public void sue() {  
        System.out.println("¡Le veré en la corte!");  
    }  
}
```


Clases mejoradas

```
public class Marketer extends Employee {  
    public void advertise()  
    {  
        System.out.println("¡Actúe ahora mientras duran las existencias!");  
    }  
    public double getSalary()  
    {  
        return super.getSalary() + 10000.0;  
    }  
}
```

Herencia y constructores

- Imaginar que se quiere dar a los empleados, excepto a los secretarios, más días de vacaciones conforme al tiempo que hayan estado en la empresa:
 - Por cada año trabajado, se conceden 2 días de vacaciones adicionales.
 - Los secretarios tienen 10 días de vacaciones sin importar la antigüedad.
 - Cuando se construye un objeto Employee, se pasará el número de años que el empleado ha estado en la empresa.
 - Esto requiere modificar la clase Employee y agregar un nuevo estado y comportamiento.

Clase Employee modificada

```
public class Employee {  
    private int years;  
    public Employee(int initialYears)  
    {  
        years = initialYears;  
    }  
    public int getHours()  
    {  
        return 40;  
    }  
}
```


Clase Employee modificada

```
public double getSalary()
{
    return 50000.0;
}
public int getVacationDays()
{
    return 10 + 2 * years;
}
public String getVacationForm()
{
    return "amarillo";
}
}
```


Problemas con los constructores

- Si se agrega un constructor a la clase Employee, las subclases no compilan. El error:

```
Lawyer.java:2: cannot find symbol
symbol : constructor Employee()
location: class Employee
public class Lawyer extends Employee {
    ^
```

- Explicación corta: si se escribe un constructor (que requiere parámetros) en la superclase, se debe escribir constructores en las subclases.

Explicación larga

- Los constructores no se heredan.
- Las subclases no heredan el constructor `Employee(int)`.
- Las subclases reciben un constructor por default que contiene:

```
public Lawyer()  
{  
    super(); // llama al constructor Employee()  
}
```
- Pero `Employee(int)` reemplazó al constructor `Employee()` de default.
- Ahora los constructores por default de las subclases intentan llamar a un constructor por default de `Employee` que no existe.

Llamar al constructor de la superclase

`super(parámetros);`

- Ejemplo:

```
public class Lawyer extends Employee {  
    public Lawyer(int years)  
    {  
        super(years);           // llama al constructor de Employee  
    }  
    ...  
}
```

- La llamada a `super` debe ser la primera instrucción en el constructor.

Clase Marketer modificada

// Clase para representar empleados de marketing.

```
public class Marketer extends Employee {  
    public Marketer(int years)  
    {  
        super(years);  
    }  
    public void advertise()  
    {  
        System.out.println("¡Actúe mientras haya existencias!");  
    }  
}
```

Clase Marketer modificada

```
public double getSalary()  
{  
    return super.getSalary() + 10000.0;  
}  
}
```



Clase Secretary

- La clase Secretary también se debe modificar.
- No tienen vacaciones extra por años trabajados.

Clase Secretary modificada

// Clase para representar secretarios.

```
public class Secretary extends Employee {  
    public Secretary()  
    {  
        super(0);  
    }  
    public void takeDictation(String text)  
    {  
        System.out.println("Tomando dictado del texto: " + text);  
    }  
}
```

Clase LegalSecretary

- Dado que Secretary no requiere ningún parámetro para su constructor, LegalSecretary compila sin necesidad de un constructor.
- Su constructor por default llama al constructor Secretary().

Herencia y campos

- Se intenta darle \$5,000 extra a los abogados por cada año trabajado.

```
public class Lawyer extends Employee {  
    ...  
    public double getSalary()  
    {  
        return super.getSalary() + 5000 * years;  
    }  
    ...  
}
```


Herencia y campos

- No funciona, el error es el siguiente:

```
Lawyer.java:7: years has private access in Employee
    return super.getSalary() + 5000 * years;
                                   ^
```
- No se puede acceder directamente a los campos privados desde las subclases.
 - Una razón: para que la subclases no rompan la encapsulación.
 - ¿Cómo se sortea esta limitación?
- Respuesta: agregar un *método accesor* para los campos que se necesitan en una subclase.

Clase Employee

```
public class Employee {  
    private int years;  
    public Employee(int initialYears)  
    {  
        years = initialYears;  
    }  
    public int getYears()  
    {  
        return years;  
    }  
    ...  
}
```

Clase Lawyer

```
public class Lawyer extends Employee {  
    public Lawyer(int years)  
    {  
        super(years);  
    }  
    public double getSalary()  
    {  
        return super.getSalary() + 5000 * getYears();  
    }  
    ...  
}
```


Revisando Secretary

- La clase Secretary tiene una mala solución:
 - Establece que todos los secretarios tienen 0 años porque no obtienen un bono de vacaciones por su servicio.
 - Si se llama a `getYears` en un objeto Secretary, siempre se obtiene 0.
 - No está bien; ¿Qué pasaría si se quisiera dar alguna otra recompensa a *todos* los empleados en función de los años de servicio?
- Hay que rediseñar la clase Employee para resolver el problema.
- Solución: separar los 10 días de vacaciones de base de aquellos que se otorgan en función de la antigüedad.

Clase Employee mejorada

```
public class Employee {  
    private int years;  
    public Employee(int initialYears)  
    {  
        years = initialYears;  
    }  
    public int getVacationDays()  
    {  
        return 10 + getSeniorityBonus();  
    }  
}
```

Clase Employee mejorada

// días de vacaciones dados por cada año en la compañía

```
public int getSeniorityBonus()
```

```
{
```

```
    return 2 * years;
```

```
}
```

```
...
```

```
}
```




Clase Secretary mejorada

- Secretary debe sobreponer a `getSeniorityBonus` y regresar 0.

Clase Secretary mejorada

```
public class Secretary extends Employee {  
    public Secretary(int years)  
    {  
        super(years);  
    }  
    // Secretarios no obtienen bono por sus años de servicio.  
    public int getSeniorityBonus()  
    {  
        return 0;  
    }  
}
```

Clase Secretary mejorada

```
public void takeDictation(String text)
{
    System.out.println("Tomando dictado del texto: " + text);
}
}
```


Clase LegalSecretary

- Ahora, a la clase LegalSecretary se le tiene que poner un constructor:

// Clase para representar secretarios legales.

```
public class LegalSecretary extends Secretary {  
    public LegalSecretary(int years)  
    {  
        super(years);  
    }  
    ...  
}
```



Conclusiones

- La herencia es una herramienta útil que permite polimorfismo y reusar código.
- Limitación: Java tiene herencia única. Una clase solo puede extender a una superclase.
- Solución: Java ofrece otro mecanismo de herencia llamado *interfaces*.