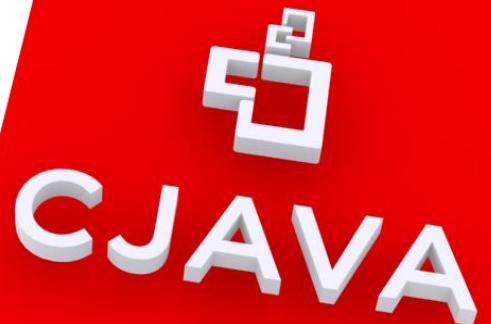


Edwin Maraví
emaravi@cjavaperu.com





0101010101010101
001010101010101010
0101010101010101
0101010101010101

01010101010101
001010101010101010
0101010101010101

01010101010101
001010101010101010
0101010101010101

Misión

Nuestro equipo trabaja para integrar la tecnología Java en la sociedad como solución a todas sus necesidades.





CJAVA
siempre para apoyarte

010101010101010101010101010101
001010101010101010101010101010101
010101010101010101010101010100101
01010101010101010101
1010101010101010
0101010101010101010101010101010101
101010101010101010101001010010101010101
01010101010101

Visión

Poder aportar al desarrollo del País usando tecnología Java.



Servicios Académicos

Programer (80 horas - Certificación Java 11)

[Certificado: Java Programer]

Developer (80 horas - Spring FrameWork y Angular)

[Certificado: Java Developer]

Expert (80 horas – Microservicios y DevOps)

[Certificado: Java Expert]

Architect (80 horas)

[Certificado: Java Arquitect]

Carrera (12 meses)

[Diploma: Carrera Java]

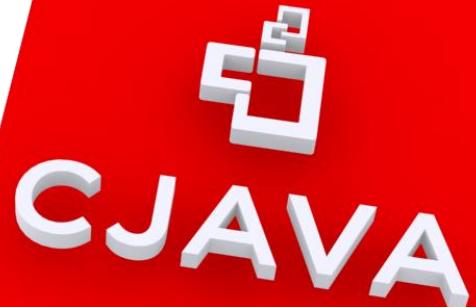
Programmer



Quienes Somos

Somos una organización orientada a **desarrollar, capacitar e investigar tecnología JAVA** a través de un prestigioso staff de profesionales a nivel nacional.





Contáctenos

- Av. Arenales 395 oficina 405
Santa Beatriz - Lima 01 - Perú
- Teléfono: 433-6948
- RPC / WhatsApp: 932 656 459
- Email: info@cjavaperu.com

Síguenos

- [/cjava.peru.1](https://www.facebook.com/cjava.peru.1)
- [@cjava_peru](https://twitter.com/cjava_peru)
- [/cjavaperu](https://www.linkedin.com/company/cjavaperu/)



Programación Orientada a Objetos.



Objetivos

Al finalizar esta lección, debería estar capacitado para lo siguiente:

- Usar la encapsulación en el diseño de clases Java
- Modelar problemas de negocio con clases Java
- Convertir las clases en inmutables
- Crear y usar subclases Java
- Sobrecargar métodos
- Usar métodos de argumentos variables



Encapsulación

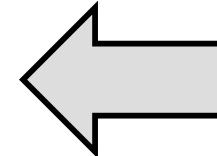
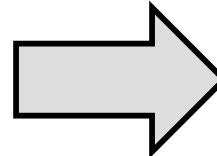
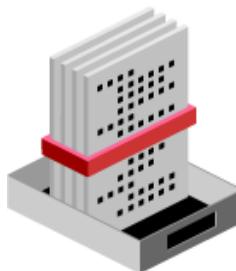
El término *encapsulación* significa incluir en una cápsula o envolver algo alrededor de un objeto para cubrirlo. En la programación orientada a objetos, la encapsulación cubre, o envuelve, el funcionamiento interno de un objeto Java.

- El usuario del objeto no puede ver las variables de datos o los campos.
- Los métodos, las funciones de Java, proporcionan un servicio explícito al usuario del objeto, pero ocultan la implantación.
- Mientras los servicios no cambien, la implantación se puede modificar sin que esto afecte al usuario.



Encapsulación: Ejemplo

¿Qué datos y operaciones encapsularía en un objeto que represente a un empleado?



ID de empleado
Nombre
Número de la Seguridad Social
Salario

Cambio de nombre
Subida de salario



Encapsulación: datos privados, métodos públicos

Una forma de ocultar los detalles de implantación es declarar todos los campos como private.

```
1 public class CheckingAccount {  
2     private int custID;  
3     private String name;  
4     private double amount;  
5     public CheckingAccount {  
6         }  
7     public void setAmount (double amount) {  
8         this.amount = amount;  
9     }  
10    public double getAmount () {  
11        return amount;  
12    }  
13    //... other public accessor and mutator methods  
14 }
```

La declaración de campos como private evita que se pueda acceder directamente a estos datos desde una instancia de clase.

// illegal!
ca.amount =
1_000_000_000.00;



Modificadores de acceso públicos y privados

- La palabra clave public, que se aplica a campos y métodos, permite a cualquier clase de cualquier paquete acceder al campo o al método.
- La palabra clave private, que se aplica a campos y métodos, permite el acceso solo a otros métodos de la propia clase.

```
CheckingAccount chk = new CheckingAccount ();
chk.amount = 200; // Compiler error - amount is a private
field
chk.setAmount (200); // OK
```

- La palabra clave private también se puede aplicar a un método para ocultar un detalle de implantación.

```
// Called when a withdrawal exceeds the available funds
private void applyOverdraftFee () {
    amount += fee;
}
```

Revisión de la clase Employee

La clase Employee utiliza actualmente el acceso de tipo public para todos sus campos. Para encapsular los datos, convierta los campos en private.

```
package come.example.model;
public class Employee {
    private int empId;
    private String name;
    private String ssn;
    private double salary;
    //... constructor and methods
}
```

Paso 1 de la encapsulación:
ocultar los datos (campos).



Asignación de nombres de métodos: recomendaciones

Si bien los campos ahora están ocultos mediante el acceso private, hay algunos problemas con la clase Employee actual.

- Los métodos setter (actualmente acceso de tipo public) permiten a cualquier otra clase cambiar el ID, el SSN y el salario (aumentarlo o reducirlo).
- La clase actual no representa realmente las operaciones definidas en el diseño de la clase Employee original.
- Dos recomendaciones para los métodos:
 - Oculte todos los detalles de implantación que pueda.
 - Asigne al método un nombre que identifique claramente su uso o funcionalidad.
- En el modelo original de la clase Employee se han realizado las operaciones de cambio de nombre y subida de salario.



Clase Employee refinada

```
1 package com.example.domain;
2 public class Employee {
3     // private fields ...
4     public Employee () {
5     }
6     // Remove all of the other setters
7     public void setName(String newName) {
8         if (newName != null) {
9             this.name = newName;
10        }
11    }
12
13    public void raiseSalary(double increase) {
14        this.salary += increase;
15    }
16 }
```

Paso 2 de la encapsulación: estos nombres de métodos tienen sentido en el contexto de una clase Employee.



Haga que las clases sean lo más inmutables posibles

```
1 package com.example.domain;
2 public class Employee {
3     // private fields ...
4     // Create an employee object
5     public Employee (int empId, String name,
6                       String ssn, double salary) {
7         this.empId = empId;
8         this.name = name;
9         this.ssn = ssn;
10        this.salary = salary;
11    }
12
13    public void setName(String newName) { ... }
14
15    public void raiseSalary(double increase) { ... }
16 }
```

Paso 3 de la encapsulación: elimine el constructor por defecto; implante un constructor para definir el valor de todos los campos.



Creación de subclases

Ha creado una clase Java para modelar los datos y las operaciones de un objeto Employee. Ahora suponga que desea especializar los datos y las operaciones para describir un objeto Manager.

```
1 package com.example.domain;
2 public class Manager {
3     private int empId;
4     private String name;
5     private String ssn;
6     private double salary;
7     private String deptName;
8     public Manager () { }
9     // access and mutator methods...
10 }
```

un momento...
este código resulta muy familiar....



Subclases

En un lenguaje orientado a los objetos como Java, las subclases se usan para definir una nueva clase en relación con una existente.

Employee
private int empld
private String name
private String ssn
private double salary
public Employee(int empld, String name, String ssn, double salary) {}
public void setName(String newName) {}
public void raiseSalary(double increase) {}
«accessor methods»

superclase: Employee
(clase "principal")

esto significa "hereda"

Manager
private String deptName
public Manager (int empld, String name, String ssn, double salary, String dept) {}
public String getDeptName() {}

subclase: Manager,
es un Employee
(clase "secundaria")



Subclase Manager

```
1 package com.example.domain;
2 public class Manager extends Employee {
3     private String deptName;
4     public Manager (int empId, String name,
5                     String ssn, double salary, String dept) {
6         super (empId, name, ssn, salary);
7         this.deptName = dept;
8     }
9
10    public String getDeptName () {
11        return deptName;
12    }
13    // Manager also gets all of Employee's public methods!
14 }
```

La palabra clave `super` se usa para llamar al constructor de la clase principal. Debe ser la primera sentencia del constructor.



Los constructores no se heredan

Si bien una subclase hereda todos los métodos y campos de una clase principal, no hereda los constructores. Hay dos formas de obtener un constructor:

- Escribir su propio constructor.
- Usar el constructor por defecto.

– Si no declara un constructor, se le proporcionará un constructor sin argumentos por defecto.

– Si declara su propio constructor, el constructor por defecto ya no se proporcionará.



Uso de super

Para crear una instancia de una subclase, normalmente resulta más fácil llamar al constructor de la clase principal.

- En su constructor, Manager llama al constructor de Employee.

```
super (empId, name, ssn, salary);
```

- La palabra clave super se usa para llamar al constructor de un principal.
- Debe ser la primera sentencia del constructor.
- Si no se proporciona, se inserta una llamada por defecto a super().
- La palabra clave super también se puede usar para llamar al método de un principal o para acceder a un campo (no privado) de un principal.

Creación de un objeto Manager

La operación de creación de un objeto Manager es similar a la de creación de un objeto Employee:

```
Manager mgr = new Manager (102, "Barbara Jones",  
"107-99-9078", 109345.67, "Marketing");
```

- Todos los métodos Employee están disponibles para Manager:

```
mgr.raiseSalary (10000.00);
```

- La clase Manager define un nuevo método para obtener el valor Department Name:

```
String dept = mgr.getDeptName();
```



¿Qué es el polimorfismo?

El término *polimorfismo*, en su definición estricta, significa “muchas formas”.

```
Employee emp = new Manager();
```

- Esta asignación es perfectamente válida. Un empleado puede ser un superior.
- Sin embargo, el siguiente código no se compila:

```
emp.setDeptName ("Marketing"); // compiler error!
```

- El compilador Java reconoce la variable emp solo como un objeto Employee. Debido a que la clase Employee no tiene un método setDeptName, muestra un error.

Sobrecarga de métodos

Su diseño puede llamar a varios métodos de la misma clase con el mismo nombre, pero con distintos argumentos.

```
public void print (int i)
public void print (float f)
public void print (String s)
```

- Java le permite reutilizar un nombre de método para más de un método.
- Se aplican dos reglas a los métodos sobrecargados:
 - Las listas de argumentos *deben* ser distintas.
 - Los tipos de retorno *pueden* variar.
- Por tanto, el siguiente ejemplo no es válido:

```
public void print (int i)
public String print (int i)
```

Métodos con argumentos variables

Una variación a la sobrecarga de métodos es cuando necesita un método que tome cualquier número de argumentos del mismo tipo:

```
public class Statistics {  
    public float average (int x1, int x2) {}  
    public float average (int x1, int x2, int x3) {}  
    public float average (int x1, int x2, int x3, int x4) {}  
}
```

- Estos tres métodos sobrecargados comparten la misma funcionalidad.
Estaría bien reducir estos métodos a uno solo.

```
Statistics stats = new Statistics ();  
float avg1 = stats.average(100, 200);  
float avg2 = stats.average(100, 200, 300);  
float avg3 = stats.average(100, 200, 300, 400);
```



Métodos con argumentos variables

- Java proporciona una función denominada *varargs* o *argumentos variables*.

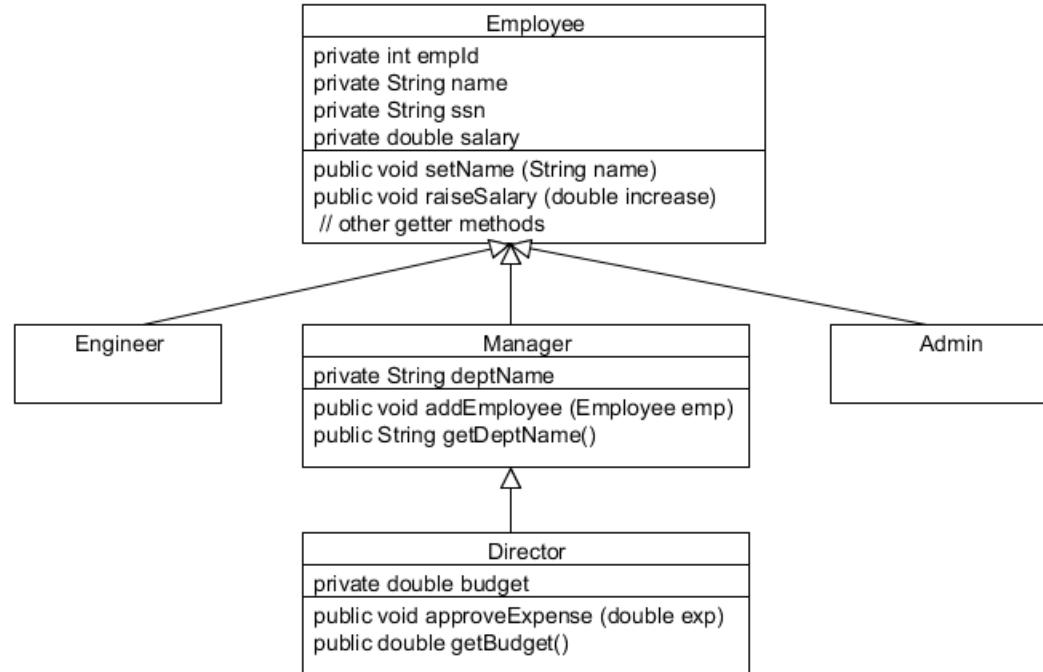
```
1 public class Statistics {  
2     public float average(int... nums) {  
3         int sum = 0;  
4         for (int x : nums) { // iterate int array nums  
5             sum += x;  
6         }  
7         return ((float) sum / nums.length);  
8     }  
9 }
```

- Tenga en cuenta que el argumento `nums` es realmente un objeto de matriz de tipo `int[]`. Esto permite al método iterarse y permitir cualquier cantidad de elementos.



Herencia única

El lenguaje de programación Java permite que una clase solo amplíe otra clase. A esto se le denomina *herencia única*.

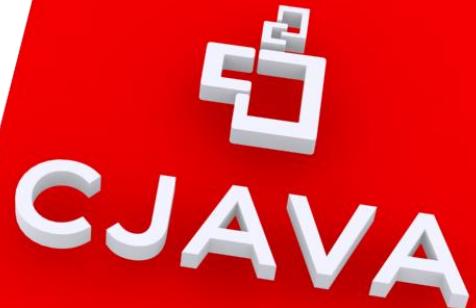




Resumen

En esta lección, debe haber aprendido a hacer lo siguiente:

- Crear clases Java simples
- Usar la encapsulación en el diseño de clases Java
- Modelar problemas de negocio con clases Java
- Convertir las clases en inmutables
- Crear y usar subclases Java
- Sobrecargar métodos
- Usar métodos de argumentos variables



Programación Orientada a Objetos Avanzada.



Objetivos

Al finalizar esta lección, debería estar capacitado para lo siguiente:

- Usar niveles de acceso: private, protected, el nivel por defecto y public.
- Sustituir métodos
- Sobrecargar constructores y otros métodos de la forma adecuada
- Usar el operador instanceof para comparar tipos de objeto
- Usar la llamada al método virtual
- Usar conversiones ascendentes y descendentes
- Sustituir métodos de la clase Object para mejorar la funcionalidad de la clase



Uso del control de acceso

Ha visto las palabras clave public y private. Hay cuatro niveles de acceso que se pueden aplicar a los métodos y los campos de datos.

En la siguiente tabla se muestra el acceso a un campo o método marcado con el modificador de acceso en la columna izquierda.

Modifier (keyword)	Same Class	Same Package	Subclass in Another Package	Universe
private	Yes			
default	Yes	Yes		
protected	Yes	Yes	Yes *	
public	Yes	Yes	Yes	Yes

Las clases pueden ser por defecto (sin modificador) o de tipo public.



Control de acceso protegido: ejemplo

```
1 package demo;
2 public class Foo {
3     protected int result = 20;           ← declaración que
4     int other = 25;                   permite
5 }
```

subclases

```
1 package test;
2 import demo.Foo;
3 public class Bar extends Foo {
4     private int sum = 10;
5     public void reportSum () {
6         sum += result;
7         sum += other;               ← error del
8     }                                compilador
9 }
```



Sombra de campos: ejemplo

```
1 package demo;  
2 public class Foo2 {  
3     protected int result = 20;  
4 }
```

```
1 package test;  
2 import demo.Foo2;  
3 public class Bar2 extends Foo2 {  
4     private int sum = 10;  
5     private int result = 30; El campo result es una sombra del campo principal.  
6     public void reportSum() {  
7         sum += result;  
8     }  
9 }
```

Control del acceso: recomendación

Una buena práctica al trabajar con campos es hacer que sean tan poco accesibles como sea posible y especificar claramente el uso de los campos en los métodos.

```
1 package demo;
2 public class Foo3 {
3     private int result = 20;
4     protected int getResult() { return result; }
5 }
```

```
1 package test;
2 import demo.Foo3;
3 public class Bar3 extends Foo3 {
4     private int sum = 10;
5     public void reportSum() {
6         sum += getResult();
7     }
8 }
```



Sustitución de métodos

Considere un requisito para proporcionar una cadena que represente algunos detalles sobre los campos de la clase Employee.

```
1 public class Employee {  
2     private int empId;  
3     private String name;  
4     // ... other fields and methods  
5     public String getDetails () {  
6         return "Employee id: " + empId +  
7                 " Employee name:" + name;  
8     }  
9 }
```



Sustitución de métodos

En la clase Manager, mediante la creación de un método con la misma firma que el método de la clase Employee, está *sustituyendo* el método getDetails:

```
1 public class Manager extends Employee {  
2     private String deptName;  
3     // ... other fields and methods  
4     public String getDetails () {  
5         return super.getDetails () +  
6                 " Department: " + deptName;  
7     }  
8 }
```

Una subclase puede llamar a un método principal usando la palabra clave `super`.

Llamada a un método sustituido

- Con los ejemplos anteriores de Employee y Manager:

```
Employee e = new Employee (101, "Jim Smith", "011-12-2345",
100_000.00);
Manager m = new Manager (102, "Joan Kern", "012-23-4567",
110_450.54, "Marketing");
System.out.println (e.getDetails());
System.out.println (m.getDetails());
```

- Se llama al método `getDetails` correcto de cada clase:

```
Employee id: 101 Employee name: Jim Smith
Employee id: 102 Employee name: Joan Kern Department: Marketing
```

Llamada al método virtual

- ¿Qué sucede si tiene lo siguiente?

```
Employee e = new Manager (102, "Joan Kern", "012-23-4567",  
110_450.54, "Marketing");  
System.out.println (e.getDetails());
```

- Durante la ejecución, se determina que el tipo de tiempo de ejecución del objeto es un objeto Manager:

```
Employee id: 102 Employee name: Joan Kern Department: Marketing
```

- El compilador no tiene fallos porque la clase Employee tiene un método getDetails y, en tiempo de ejecución, al método que se ejecuta se le hace referencia desde un objeto Manager.
- Se trata de un aspecto de polimorfismo denominado *llamada al método virtual*.



Accesibilidad de los métodos sustituidos

Un método sustituido no puede ser menos accesible que el método de la clase principal.

```
public class Employee {  
    //... other fields and methods  
    public String getDetails() { ... }  
}
```

```
public class Manager extends Employee {  
    //... other fields and methods  
    private String getDetails() { //... } // Compile time error  
}
```



Aplicación del polimorfismo

Suponga que se le solicita que cree una nueva clase que calcule las acciones otorgadas a los empleados según su salario y su rol (superior, ingeniero o administrador):

```
1 public class EmployeeStockPlan {  
2     public int grantStock (Manager m) {  
3         // perform a calculation for a Manager  
4     }  
5     public int grantStock (Engineer e) {  
6         // perform a calculation for an Engineer  
7     }  
8     public int grantStock (Admin a) {  
9         // perform a calculation for an Admin  
10    }  
11    //... one method per employee type  
12}
```

*no muy
orientado a
objetos*



Aplicación del polimorfismo

Una buena práctica consiste en transferir parámetros y escribir métodos que usen el formato más genérico del objeto posible.

```
public class EmployeeStockPlan {  
    public int grantStock (Employee e) {  
        // perform a calculation based on Employee data  
    }  
}
```

```
// In the application class  
EmployeeStockPlan esp = new EmployeeStockPlan () :  
Manager m = new Manager ();  
int stocksGranted = grantStock (m);  
...
```

Uso de la palabra clave instanceof

El lenguaje Java proporciona la palabra clave instanceof para determinar un tipo de clase de objeto en tiempo de ejecución.

```
1 public class EmployeeRequisition {  
2     public boolean canHireEmployee(Employee e) {  
3         if (e instanceof Manager) {  
4             return true;  
5         } else {  
6             return false;  
7         }  
8     }  
9 }
```

Conversión de referencias de objetos

Después de usar el operador instanceof para verificar que el objeto recibido como argumento es una subclase, puede acceder a toda la funcionalidad del objeto convirtiendo la referencia:

```
1 public void modifyDeptForManager (Employee e, String dept) {  
2     if (e instanceof Manager) {  
3         Manager m = (Manager) e;  
4         m.setDeptName (dept);  
5     }  
6 }
```

Sin la conversión a Manager, el método setDeptName no se compilaría.



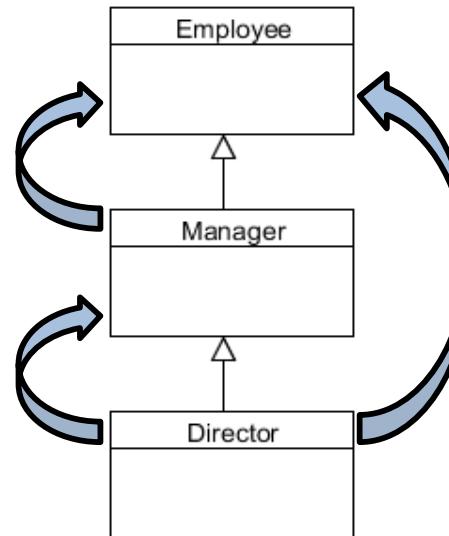
Conversión de reglas

Las conversiones ascendentes siempre están permitidas y en ellas no se necesita un operador cast.

```
Director d = new Director();  
Manager m = new Manager();
```

```
Employee e = m;  
// OK
```

```
Manager m = d;  
// OK
```



```
Employee e = d;  
// OK
```



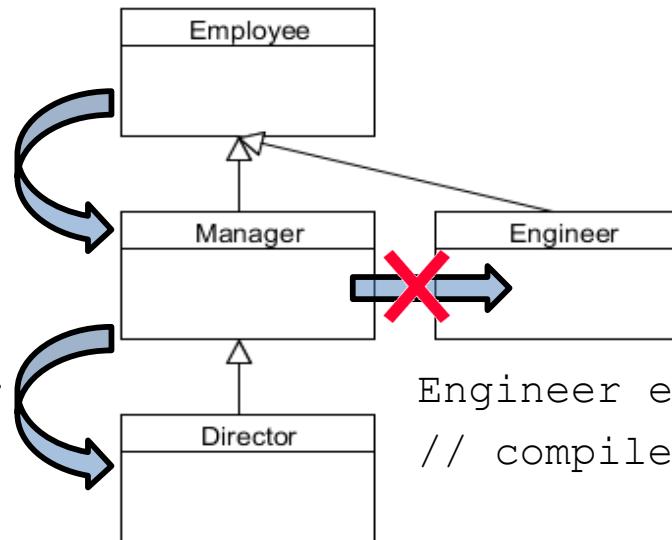
Conversión de reglas

En el caso de las conversiones descendentes, el compilador debe aceptar que la conversión es, al menos, posible.

```
Employee e = new Manager();  
Manager m = new Manager();
```

```
Manager m = (Manager)e;  
// Would also work if  
// e was a Director obj
```

```
Director d = (Director)m;  
// fails at run time
```



```
Engineer eng = (Engineer)m;  
// compiler error
```

Sustitución de métodos de objeto

Una de las ventajas de la herencia única es que cada una de las clases tiene un objeto principal por defecto. La clase raíz de cada clase Java es `java.lang.Object`.

- No es necesario que declare que la clase amplía `Object`. El compilador se encarga de esa tarea.

```
public class Employee { //... }
```

es equivalente a:

```
public class Employee extends Object { //... }
```

- La clase raíz contiene varios métodos que no son finales, pero hay tres que son importantes para pensar en la sustitución:
 - `toString`, `equals` y `hashCode`



Método Object `toString`

Al método `toString` se le llama siempre que se transfiera una instancia de la clase a un método que tome un objeto `String`, como `println`:

```
Employee e = new Employee (101, "Jim Kern", ...)  
System.out.println (e);
```

- Puede utilizar `toString` para proporcionar información de la instancia:

```
public String toString () {  
    return "Employee id: " + empId + "\n"  
        "Employee name:" + name;  
}
```

- Este enfoque para obtener detalles sobre la clase es mejor que crear su propio método `getDetails`.



Método object equals

El método Object equals solo compara referencias de objetos.

- Si hay dos objetos x e y en cualquier clase, x es igual a y si y solo si x e y hacen referencia al mismo objeto.
- Ejemplo:

```
Employee x = new Employee (1,"Sue","111-11-1111",10.0);
Employee y = x;
x.equals (y); // true
Employee z = new Employee (1,"Sue","111-11-1111",10.0);
x.equals (z); // false!
```

- Ya que lo que realmente se desea es probar el contenido del objeto Employee, es necesario sustituir el método equals:

```
public boolean equals (Object o) { ... }
```

Sustitución de equals en Employee

Un ejemplo de sustitución del método equals en la clase Employee compara todos los campos para ver si tienen igualdad:

```
1 public boolean equals (Object o) {  
2     boolean result = false;  
3     if ((o != null) && (o instanceof Employee)) {  
4         Employee e = (Employee)o;  
5         if ((e.empId == this.empId) &&  
6             (e.name.equals(this.name)) &&  
7             (e.ssn.equals(this.ssn)) &&  
8             (e.salary == this.salary)) {  
9                 result = true;  
10            }  
11        }  
12        return result;  
13    }
```



Sustitución de object hashCode

El contrato general de Object indica que si dos objetos se consideran iguales (con el método equals), el código hash del entero devuelto para los dos objetos también debe ser igual.

```
1// Generated by NetBeans
2 public int hashCode() {
3     int hash = 7;
4     hash = 83 * hash + this.empId;
5     hash = 83 * hash + Objects.hashCode(this.name);
6     hash = 83 * hash + Objects.hashCode(this.ssn);
7     hash = 83 * hash +
8             (int) (Double.doubleToLongBits(this.salary) ^
9                     (Double.doubleToLongBits(this.salary) >>> 32));
10    return hash;
11 }
```



Resumen

En esta lección, debe haber aprendido a hacer lo siguiente:

- Usar niveles de acceso: private, protected, el nivel por defecto y public
- Sustituir métodos
- Sobrecargar constructores y otros métodos de la forma adecuada
- Usar el operador instanceof para comparar tipos de objeto
- Usar la llamada al método virtual
- Usar conversiones ascendentes y descendentes
- Sustituir métodos de la clase Object para mejorar la funcionalidad de la clase.



Objetivos

Al finalizar esta lección, debería estar capacitado para lo siguiente:

- Diseñar clases base de uso general mediante clases abstractas
- Crear clases y subclases Java abstractas
- Modelar problemas de negocio mediante las palabras clave static y final
- Implantar el patrón de diseño singleton
- Distinguir entre clases de nivel superior y anidadas



Modelación de problemas de negocio con clases

La herencia (o creación de subclases) es una función esencial del lenguaje de programación Java. La herencia permite la reutilización de código mediante:

- Herencia de métodos: las subclases evitan la duplicación del código al heredar las implementaciones de métodos.
- Generalización: el código que está diseñado para basarse en el tipo más genérico posible es más fácil de mantener.

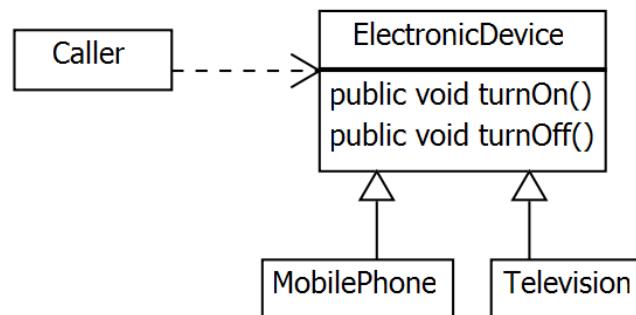


Diagrama de herencia de clases

Activación de la generalización

La codificación en un tipo base común permite introducir nuevas subclases con pocas modificaciones o ninguna de cualquier código que dependa del tipo base más genérico.

```
ElectronicDevice dev = new Television();  
dev.turnOn(); // all ElectronicDevices can be turned on
```

Use siempre el tipo de referencia más genérico posible.



Identificación de la necesidad de clases abstractas

Las subclases no tienen que heredar una implantación de método si el método está especializado.

```
public class Television extends ElectronicDevice {  
  
    public void turnOn() {  
        changeChannel(1);  
        initializeScreen();  
    }  
    public void turnOff() {}  
  
    public void changeChannel(int channel) {}  
    public void initializeScreen() {}  
  
}
```



Definición de clases abstractas

Una clase se puede declarar como abstracta mediante el modificador de nivel de clase `abstract`.

```
public abstract class ElectronicDevice { }
```

- Una clase abstracta puede tener una subclase.

```
public class Television extends ElectronicDevice { }
```

- Una clase abstracta no se puede instanciar.

```
ElectronicDevice dev = new ElectronicDevice(); // error
```



Definición de métodos abstractos

Un método se puede declarar como abstracto mediante el modificador de nivel de método `abstract`.

```
public abstract class ElectronicDevice {  
  
    public abstract void turnOn();  
    public abstract void turnOff();  
}
```

Sin corchetes

Un método abstracto:

- No puede tener un cuerpo de método.
- Se debe declarar en una clase abstracta.
- Se sobrescribe en subclases.

Validación de clases abstractas

Las siguientes reglas adicionales se aplican cuando se usan las clases y los métodos abstractos:

- Una clase abstracta puede tener cualquier número de métodos abstractos y no abstractos.
- Al heredar de una clase abstracta, debe realizar una de las siguientes acciones:
 - Declarar la clase secundaria como abstracta.
 - Sustituya todos los métodos abstractos heredados de la clase principal. De lo contrario, se producirá un error de tiempo de compilación.

```
error: Television is not abstract and does not override  
abstract method turnOn() in ElectronicDevice
```



Palabra clave static

El modificador static se usa para declarar campos y métodos como recursos de nivel de clase. Los miembros de clase estáticos:

- Se pueden usar sin instancias de objeto.
- Se usan cuando un problema se soluciona mejor sin objetos.
- Se usan cuando hay objetos del mismo tipo que deben compartir campos.
- *No se deben usar para no usar las funciones orientadas a objetos de Java a menos que haya un motivo justificado.*



Métodos estáticos

Los métodos estáticos son métodos que se pueden llamar incluso si la clase en la que se hayan declarado no se ha instanciado. Los métodos estáticos:

- Se denominan métodos de clase.
- Son útiles para las API que no están orientadas a objetos.
 - `java.lang.Math` contiene muchos métodos estáticos.
- Se suelen usar en lugar de los constructores para realizar tareas relacionadas con la inicialización de objetos.
- No pueden acceder a miembros no estáticos de la misma clase.
- Se pueden ocultar en subclases, pero no se pueden sustituir.
 - Sin llamada a método virtual.



Implantación de métodos estáticos

```
public class StaticErrorClass {  
    private int x;  
  
    public static void staticMethod() {  
        x = 1; // compile error  
        instanceMethod(); // compile error  
    }  
  
    public void instanceMethod() {  
        x = 2;  
    }  
}
```



Llamada a métodos estáticos

```
double d = Math.random();
StaticUtilityClass.printMessage();
StaticUtilityClass uc = new StaticUtilityClass();
uc.printMessage(); // works but misleading
sameClassMethod();
```

Al llamar a los métodos estáticos, debería:

- Cualificar la ubicación del método con un nombre de clase si el método se encuentra en otra clase distinta a la del emisor de la llamada
 - No es necesario para métodos de la misma clase
- Evitar el uso de una referencia de objeto para llamar a un método estático



Variables estáticas

Las variables estáticas son variables a las que se puede acceder incluso aunque la clase en la que se hayan declarado no se haya instanciado. Las variables estáticas:

- Se denominan variables de clase.
- Se limitan a una sola copia por JVM.
- Son útiles para contener datos compartidos.
 - Los métodos estáticos almacenan datos en variables estáticas.
 - Todas las instancias de objetos comparten una sola copia de cualquier variable estática.
- Se inicializan cuando la clase contenedora se carga por primera vez.



Definición de variables estáticas

```
public class StaticCounter {  
    private static int counter = 0;  
  
    public StaticCounter() {  
        counter++;  
    }  
  
    public static int getCount() {  
        return counter;  
    }  
}
```



Solo una copia en
la memoria



Uso de variables estáticas

```
double p = Math.PI;
```

```
new StaticCounter();
new StaticCounter();
System.out.println("count: " + StaticCounter.getCount());
```

Al acceder a las variables estáticas, debería:

- Cualificar la ubicación de la variable con un nombre de clase si la variable se encuentra en otra clase distinta a la del emisor de la llamada
 - No es necesario para variables de la misma clase
- Evitar el uso de una referencia de objeto para acceder a una variable estática



Importaciones estáticas

Una sentencia de importación estática hace que los miembros estáticos de una clase estén disponibles con su nombre simple.

- Con cualquiera de las siguientes líneas:

```
import static java.lang.Math.random;  
import static java.lang.Math.*;
```

- La llamada al método Math.random() se podría escribir como:

```
public class StaticImport {  
    public static void main(String[] args) {  
        double d = random();  
    }  
}
```



Métodos finales

Un método se puede declarar como final. Los métodos finales no se pueden sobrescribir.

```
public class MethodParentClass {  
    public final void printMessage() {  
        System.out.println("This is a final method");  
    }  
}
```

```
public class MethodChildClass extends MethodParentClass {  
    // compile-time error  
    public void printMessage() {  
        System.out.println("Cannot override method");  
    }  
}
```



Clases finales

Una clase se puede declarar como final. Las clases finales no se pueden ampliar.

```
public final class FinalParentClass { }
```

```
// compile-time error
public class ChildClass extends FinalParentClass { }
```



Variables finales

El modificador final se puede aplicar a las variables. Las variables finales no pueden cambiar sus valores una vez inicializadas. Las variables finales pueden ser:

- Campos de clase
 - Los campos finales con expresiones de constantes de tiempo de compilación son variables de constantes.
 - Los campos estáticos se pueden combinar con los finales para crear una variable siempre disponible y que nunca cambia.
- Parámetros del método
- Variables locales

Nota: las referencias finales siempre deben hacer referencia al mismo objeto, pero el contenido de ese objeto se puede modificar.



Declaración de variables finales

```
public class VariableExampleClass {  
    private final int field;  
    private final int forgottenField;  
    private final Date date = new Date();  
    public static final int JAVA_CONSTANT = 10;  
  
    public VariableExampleClass() {  
        field = 100;  
        // compile-time error - forgottenField  
        // not initialized  
    }  
  
    public void changeValues(final int param) {  
        param = 1; // compile-time error  
        date.setTime(0); // allowed  
        date = new Date(); // compile-time error  
        final int localVar;  
        localVar = 42;  
        localVar = 43; // compile-time error  
    }  
}
```



Cuándo evitar las constantes

Las variables public, static y final pueden ser muy útiles, pero hay un patrón de uso concreto que se debería evitar. Las constantes pueden proporcionar una falsa sensación de validación de los datos introducidos o de comprobación del rango de valores.

- Piense en un método que solo deba recibir uno de los tres valores posibles:

```
Computer comp = new Computer();  
comp.setState(Computer.POWER_SUSPEND);
```

Se trata de una constante
int que es igual que 2.

- Las siguientes líneas de código se seguirían compilando:

```
Computer comp = new Computer();  
comp.setState(42);
```



Enumeraciones Typesafe

Java 5 ha incluido una enumeración typesafe al lenguaje. Las enumeraciones:

- Se crean con una variación de una clase Java
- Proporcionan una comprobación de rangos en tiempo de compilación

```
public enum PowerState {  
    OFF,  
    ON,  
    SUSPEND;  
}
```

Estas son las referencias a los tres únicos objetos PowerState que pueden existir.

Una enumeración se puede utilizar de la siguiente forma:

```
Computer comp = new Computer();  
comp.setState(PowerState.SUSPEND);
```

Este método toma una referencia PowerState.



Uso de enumeraciones

Las referencias a enumeraciones se pueden importar de forma estática.

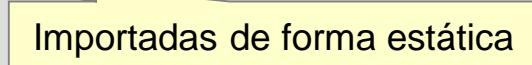
```
import static com.example.PowerState.*;  
  
public class Computer extends ElectronicDevice {  
    private PowerState powerState = OFF;  
    //...  
}
```



PowerState.OFF

Las enumeraciones se pueden usar como expresión en una sentencia switch.

```
public void setState(PowerState state) {  
    switch(state) {  
        case OFF:  
        //...  
    }  
}
```



Importadas de forma estática



Enumeraciones complejas

Las enumeraciones pueden tener campos, métodos y constructores privados.

```
public enum PowerState {  
    OFF("The power is off"),  
    ON("The usage power is high"),  
    SUSPEND("The power usage is low");  
  
    private String description;  
    private PowerState(String d) {  
        description = d;  
    }  
    public String getDescription() {  
        return description;  
    }  
}
```

Llame a un constructor PowerState para inicializar la referencia public static final OFF.

El constructor no puede ser del tipo public ni protected.



Patrones de diseño

Los patrones de diseño:

- Son soluciones reutilizables a problemas de desarrollo de software comunes.
- Están documentados en catálogos de patrones.
 - *Design Patterns: Elements of Reusable Object-Oriented Software* (Patrones de diseño: elementos del software reutilizable orientado a objetos), de Erich Gamma et al. (conocido como “Gang of Four”, la banda de los cuatro, por sus cuatro autores)
- Forman un vocabulario para hablar sobre el diseño.



Patrón Singleton

El patrón de diseño singleton detalla una implantación de clase que solo se puede instanciar una vez.

```
public class SingletonClass {  
    1 private static final SingletonClass instance =  
        new SingletonClass();  
  
    2 private SingletonClass() {}  
  
    3 public static SingletonClass getInstance() {  
        return instance;  
    }  
}
```



Clases anidadas

Una clase anidada es una clase declarada dentro del cuerpo de otra clase. Las clases anidadas:

- Tienen varias categorías.
 - Clases internas
 - Clases de miembros
 - Clases locales
 - Clases anónimas
 - Clases anidadas estáticas
- Se suelen usar en aplicaciones con elementos de interfaz gráfica de usuario (GUI).
- Pueden limitar el uso de una "clase helper" a la clase delimitadora de niveles



Clase interna: ejemplo

```
public class Car {  
    private boolean running = false;  
    private Engine engine = new Engine();  
  
    private class Engine {  
        public void start() {  
            running = true;  
        }  
    }  
  
    public void start() {  
        engine.start();  
    }  
}
```



Clases internas anónimas

Una clase anónima se usa para definir una clase sin nombre.

```
public class AnonymousExampleClass {  
    public Object o = new Object() {  
        @Override  
        public String toString() {  
            return "In an anonymous class method";  
        }  
    };  
}
```



Resumen

En esta lección, debe haber aprendido a hacer lo siguiente:

- Diseñar clases base de uso general mediante clases abstractas
- Crear clases y subclases Java abstractas
- Modelar problemas de negocio mediante las palabras clave static y final
- Implantar el patrón de diseño singleton
- Distinguir entre clases de nivel superior y anidadas.



Programación Orientada a Objetos Avanzada.



Objetivos

Al finalizar esta lección, debería estar capacitado para lo siguiente:

- Modelar problemas de negocio mediante interfaces
- Definir una interfaz Java
- Seleccionar entre herencia de interfaz y herencia de clase
- Ampliar una interfaz
- Refactorizar código para implantar el patrón DAO

Implantación de sustitución

La capacidad de describir tipos abstractos es una potente función de Java. La abstracción permite:

- Facilidad de mantenimiento
 - Clases con errores lógicos que se pueden sustituir con clases nuevas y mejoradas.
- Implantación de sustitución
 - El paquete `java.sql` describe los métodos que usan los desarrolladores para comunicarse con las bases de datos, pero la implantación es específica del proveedor.
- División del trabajo
 - Describir la API de negocio que necesita la interfaz de usuario de una aplicación permite a dicha interfaz de usuario y a la lógica de negocio desarrollarse de forma conjunta.



Interfaces Java

Las interfaces Java se usan para definir tipos abstractos. Las interfaces:

- Son similares a las clases abstractas que solo contienen métodos abstractos públicos.
- Describen los métodos que debe implantar una clase.
 - Los métodos no deben tener una implantación {corchetes}.
- Pueden contener campos constantes.
- Se pueden usar como tipo de referencia.
- Son un componente esencial de muchos patrones de diseño.



Desarrollo de interfaces Java

Las interfaces públicas de nivel superior se declaran en su propio archivo .java. Las interfaces se implantan en lugar de ampliarse.

```
public interface ElectronicDevice {  
    public void turnOn();  
    public void turnOff();  
}
```

```
public class Television implements ElectronicDevice {  
    public void turnOn() { }  
    public void turnOff() { }  
    public void changeChannel(int channel) { }  
    private void initializeScreen() { }  
}
```



Campos constantes

Las interfaces pueden tener campos constantes.

```
public interface ElectronicDevice {  
    public static final String WARNING =  
        "Do not open, shock hazard";  
    public void turnOn();  
    public void turnOff();  
}
```



Referencias a la interfaz

Puede utilizar una interfaz como tipo de referencia. Al usar un tipo de referencia de interfaz, debe usar solo los métodos señalados en la interfaz.

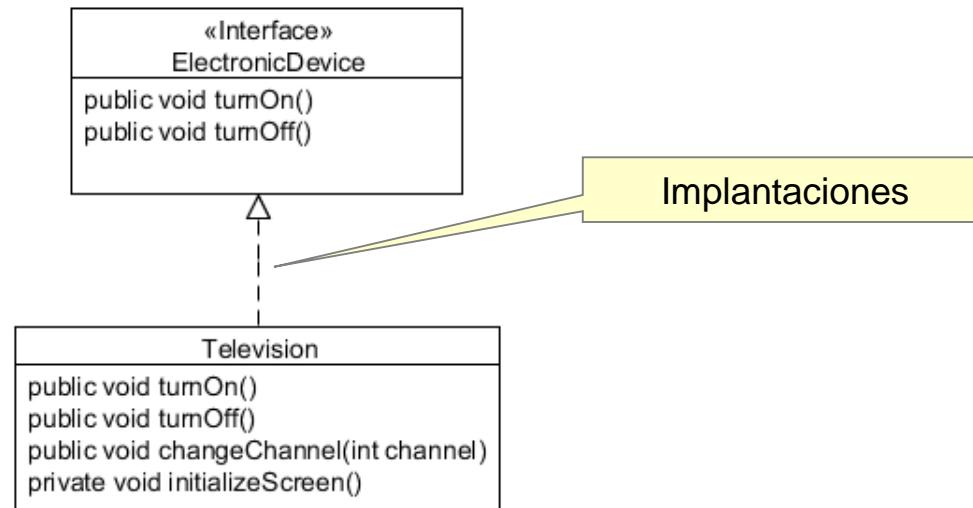
```
ElectronicDevice ed = new Television();  
ed.turnOn();  
ed.turnOff();  
ed.changeChannel(2); // fails to compile  
String s = ed.toString();
```



Operador instanceof

Puede usar instanceof con las interfaces.

```
Television t = new Television();
if (t instanceof ElectronicDevice) { }
```



Televisión es una instancia de un objeto `ElectronicDevice`.



Interfaces de marcador

- Las interfaces de marcador definen un tipo, pero no señalan ningún método que deba implantar una clase.

```
public class Person implements java.io.Serializable { }
```

- El objetivo de estos tipos de interfaces solo es comprobar los tipos.

```
Person p = new Person();
if (p instanceof Serializable) {
}
```



Conversión en tipos de interfaz

Puede realizar la conversión en un tipo de interfaz.

```
public static void turnObjectOn(Object o) {  
    if (o instanceof ElectronicDevice) {  
        ElectronicDevice e = (ElectronicDevice)o;  
        e.turnOn();  
    }  
}
```

Uso de tipos de referencia genéricos

- Utilice el tipo de referencia más genérico siempre que sea posible:

```
EmployeeDAO dao = new EmployeeDAOMemoryImpl();  
dao.delete(1);
```

EmployeeDAOMemoryImpl implanta
EmployeeDAO.

- Al usar un tipo de referencia de interfaz, puede usar una clase de implantación distinta sin correr el riesgo de interrumpir las siguientes líneas de código:

```
EmployeeDAOMemoryImpl dao = new EmployeeDAOMemoryImpl();  
dao.delete(1);
```

Es posible que solo use métodos
EmployeeDAOMemoryImpl aquí.



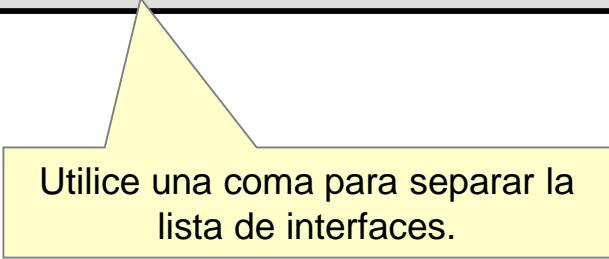
Implantación y ampliación

- Las clases pueden ampliar una clase principal e implantar una interfaz:

```
public class AmphibiousCar extends BasicCar implements  
MotorizedBoat { }
```

- También puede implantar varias interfaces:

```
public class AmphibiousCar extends BasicCar implements  
MotorizedBoat, java.io.Serializable { }
```



Utilice una coma para separar la
lista de interfaces.



Ampliación de interfaces

- Las interfaces pueden ampliar otras interfaces:

```
public interface Boat { }
```

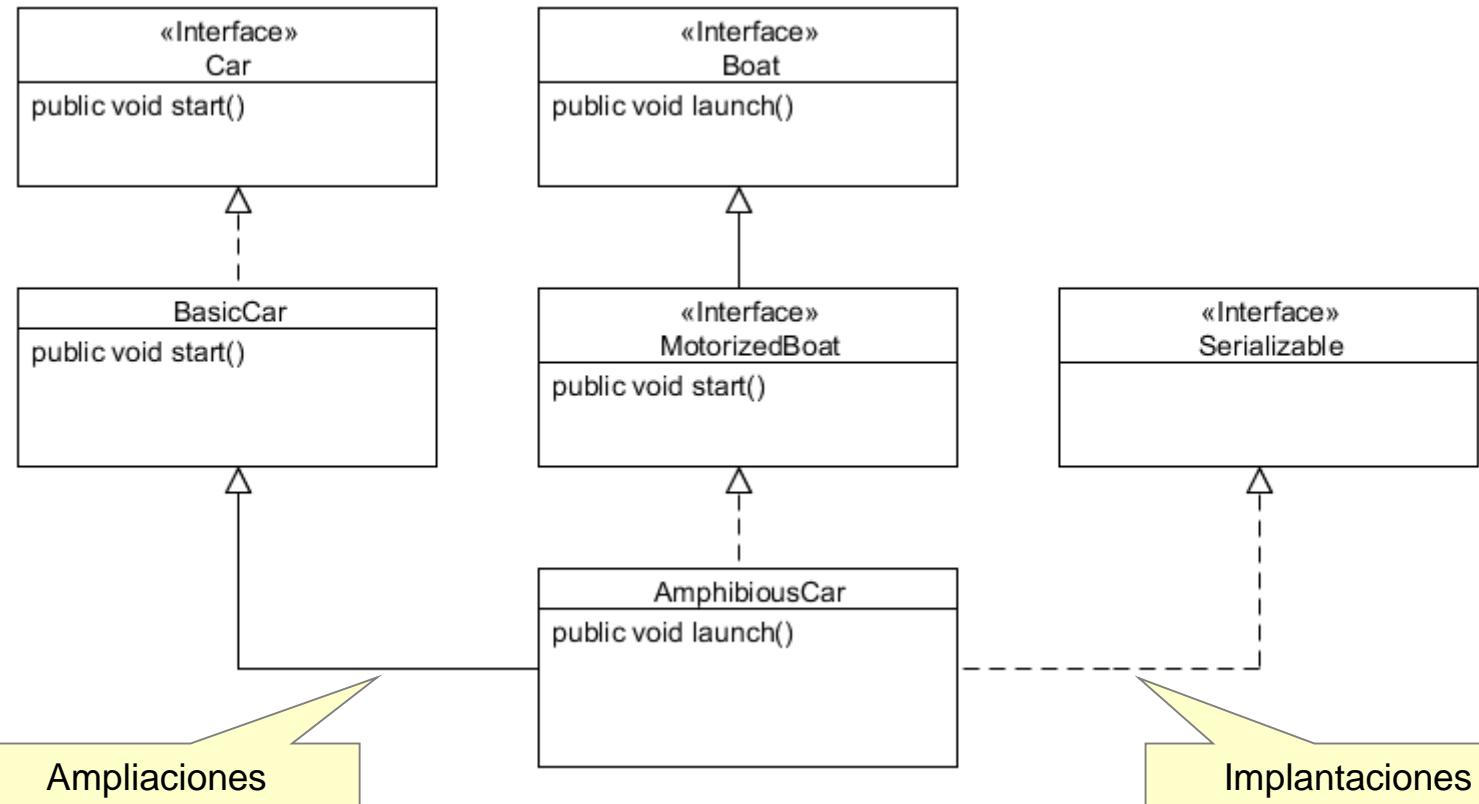
```
public interface MotorizedBoat extends Boat { }
```

- Al implantar MotorizedBoat, la clase AmphibiousCar debe cumplir el contrato señalado tanto por MotorizedBoat como por Boat:

```
public class AmphibiousCar extends BasicCar implements  
MotorizedBoat, java.io.Serializable { }
```



Interfaces de jerarquías de herencia



Diseño de patrones e interfaces

- Uno de los principios del diseño orientado a objetos es:
“Programe para una interfaz, no para una implantación.”
- Se trata de un tema común en muchos patrones de diseño.
Este principio desempeña un rol en:
 - El patrón de diseño DAO
 - El patrón de diseño de fábrica

Patrón DAO

El patrón de objeto de acceso a datos (DAO) se usa al crear una aplicación que debe mantener información. El patrón DAO:

- Separa el dominio de problemas del mecanismo de persistencia.
- Usa una interfaz para definir los métodos usados para la persistencia. Una interfaz permite sustituir la implantación de la persistencia por:
 - DAO basados en memoria como solución temporal
 - DAO basados en archivos para una versión inicial
 - DAO basados en JDBC para soportar la persistencia de la base de datos
 - DAO basados en la API de persistencia Java (JPA) para soportar la persistencia de la base de datos

Antes del patrón DAO

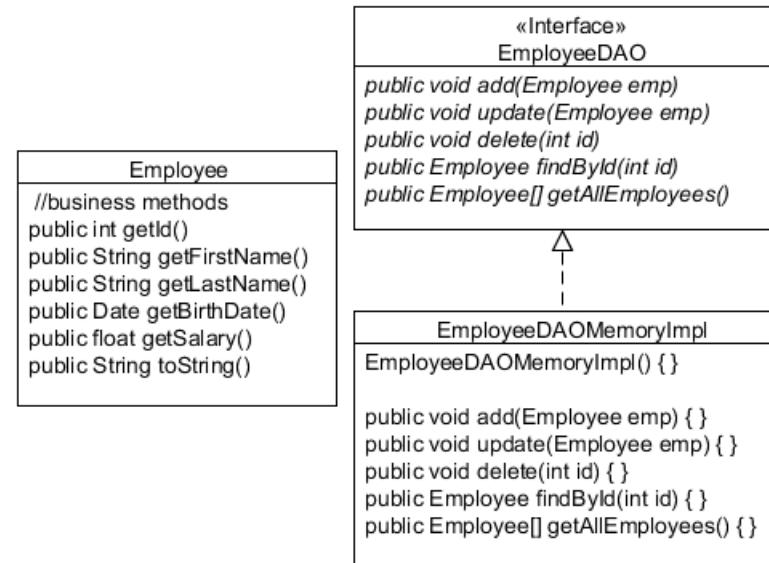
Observe la combinación de métodos de persistencia y métodos de negocio.

Employee
public int getId()
public String getFirstName()
public String getLastName()
public Date getBirthDate()
public float getSalary()
public String toString()
 //persistence methods
public void save()
public void delete()
<u>public static Employee findById(int id)</u>
<u>public static Employee[] getAllEmployees()</u>

Antes del patrón DAO

Después del patrón DAO

El patrón DAO extrae la lógica de persistencia de las clases de dominios y las traslada a clases distintas.

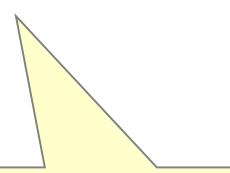


Después de la refactorización del patrón DAO

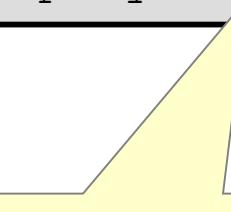
La necesidad del patrón de fábrica

El patrón DAO depende del uso de interfaces para definir una abstracción. El uso de un constructor de implantación DAO le ata a una implantación concreta.

```
EmployeeDAO dao = new EmployeeDAOMemoryImpl();
```



Al usar un tipo de interfaz, las posteriores líneas no estarán ligadas a una sola implantación.



Esta llamada al constructor está ligada a una implantación y aparecerá en muchos lugares de una aplicación.



Uso del patrón de fábrica

Al usar una fábrica, se evita que la aplicación tenga que estar totalmente acoplada a una implantación de DAO concreta.

```
EmployeeDAOFactory factory = new EmployeeDAOFactory();  
EmployeeDAO dao = factory.createEmployeeDAO();
```

La implantación
EmployeeDAO está oculta.



Fábrica

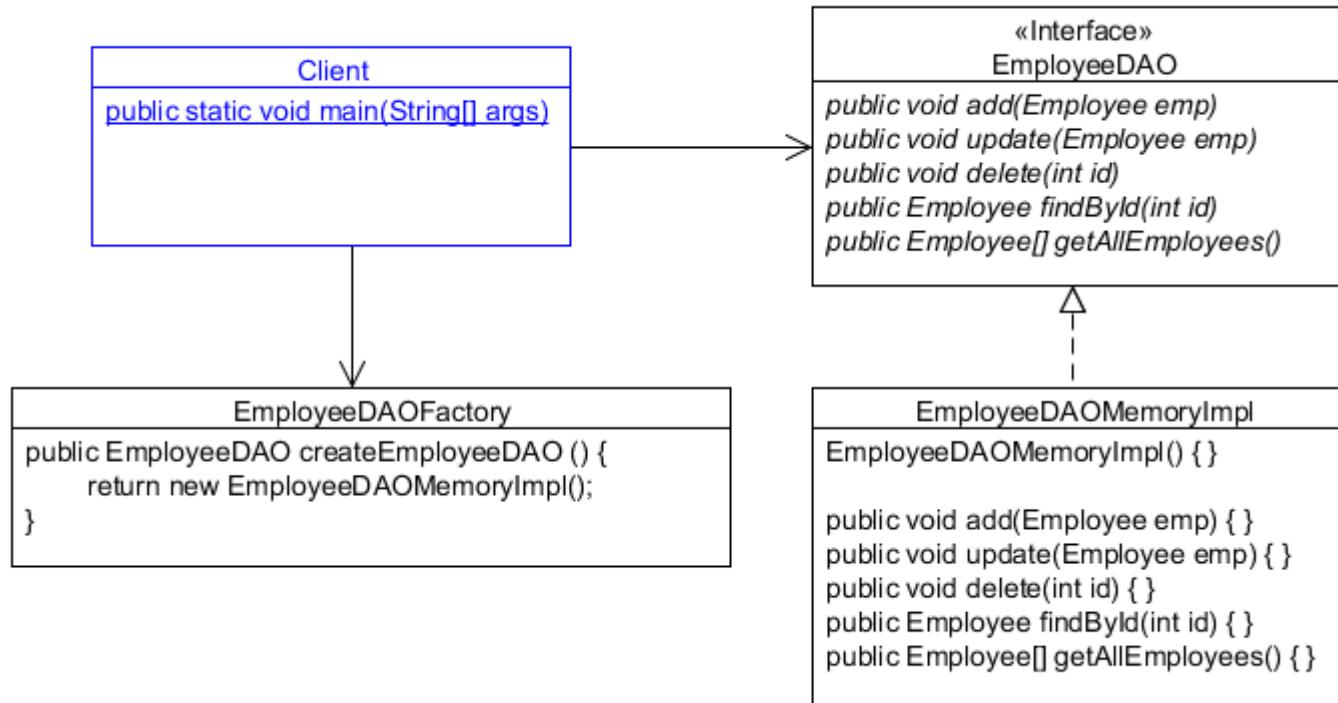
La implantación de la fábrica es el único punto de la aplicación que debe depender de clases DAO concretas.

```
public class EmployeeDAOFactory {  
    public EmployeeDAO createEmployeeDAO() {  
        return new EmployeeDAOMemoryImpl();  
    }  
}
```

Devuelve una referencia escrita como interfaz



Combinación de DAO y fábrica



Los clientes dependen solo de DAO abstractos



Reutilización del código

La duplicación del código (copiar y pegar) puede conllevar problemas de mantenimiento. No desea corregir el mismo bug una y otra vez.

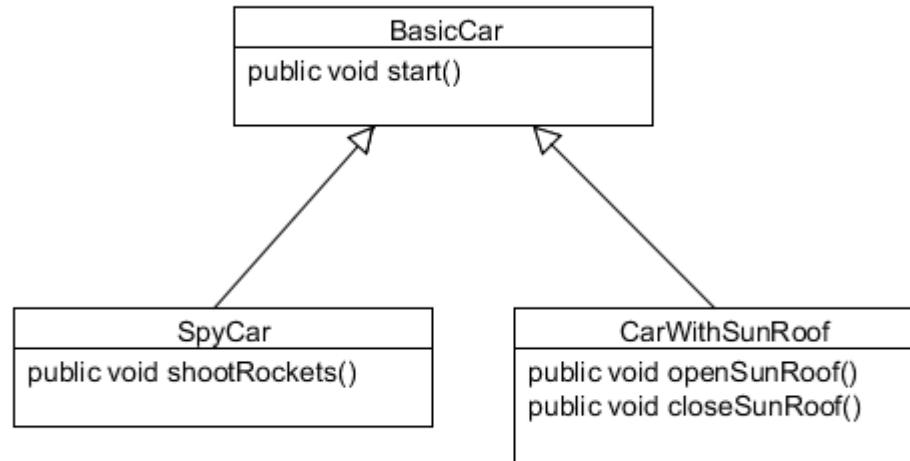
- “No se repita .” (principio DRY, del inglés Don't Repeat Yourself)
- Reutilice el código de la forma correcta:
 - Refactorice rutinas de uso común en bibliotecas.
 - Mueva el comportamiento que comparten las clases hermanas a su clase principal.
 - Cree nuevas combinaciones de comportamientos combinando varios tipos de objetos (composición).



Dificultades en el diseño

La herencia de clases permite reutilizar código, pero no es algo muy modular.

- ¿Cómo se crea un elemento SpyCarWithSunRoof?



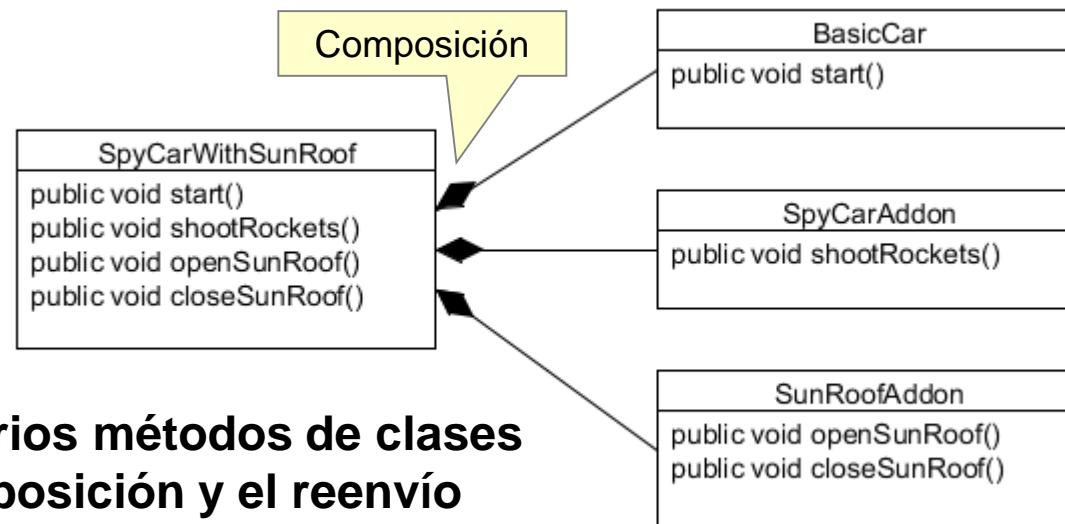
Implantaciones de métodos situadas en distintas clases



Composición

La composición de objetos permite crear objetos más complejos. Para implantar la composición:

1. Cree una clase con referencias a otras clases.
2. Agregue los mismos métodos de firma que se reenvían a los objetos a los que se hace referencia.



**Combinación de varios métodos de clases
mediante la composición y el reenvío**



Implantación de la composición

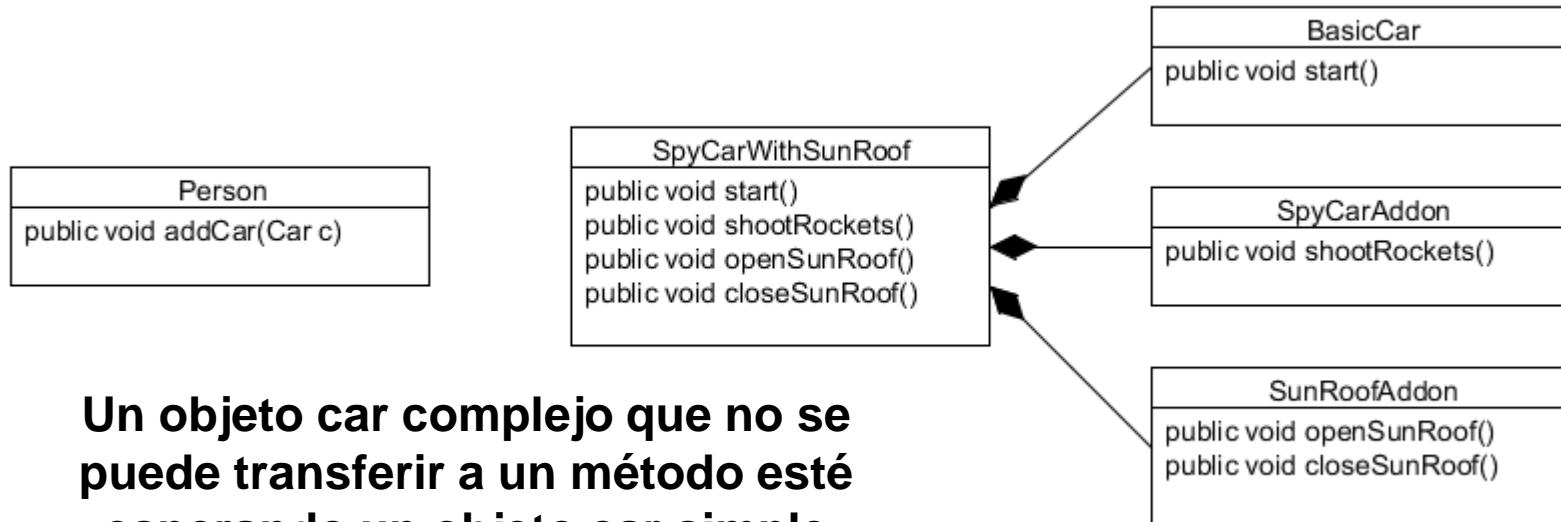
```
public class SpyCarWithSunRoof {  
    private BasicCar car = new BasicCar();  
    private SpyCarAddon spyAddon = new SpyCarAddon();  
    private SunRoofAddon roofAddon = new SunRoofAddon();  
  
    public void start() {  
        car.start();  
    }  
  
    // other forwarded methods  
}
```

Reenvío de métodos



Polimorfismo y composición

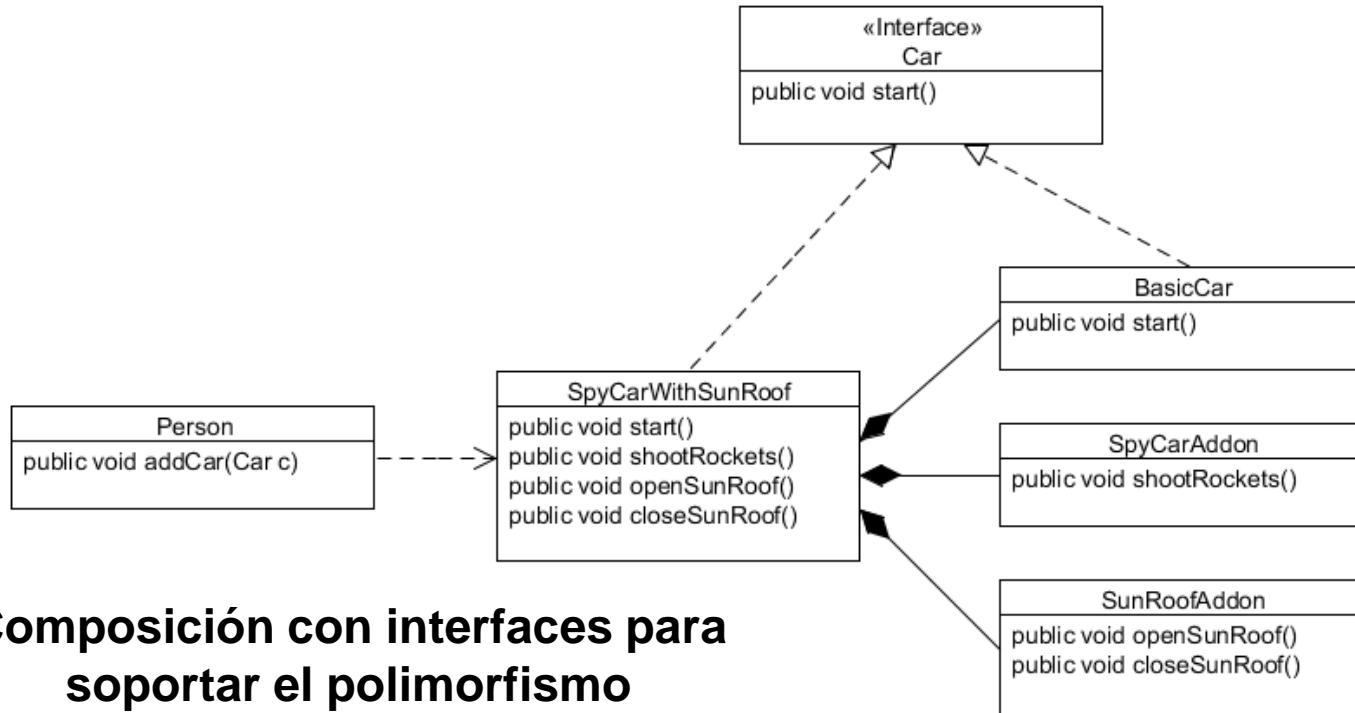
El polimorfismo nos debe permitir transferir cualquier tipo de elemento Car al método addCar. La composición no permite el polimorfismo, a menos que...



Un objeto car complejo que no se puede transferir a un método esté esperando un objeto car simple

Polimorfismo y composición

Utilice interfaces para que todas las clases delegadas soporten el polimorfismo.



**Composición con interfaces para
soportar el polimorfismo**



Resumen

En esta lección, debe haber aprendido a hacer lo siguiente:

- Modelar problemas de negocio mediante interfaces
- Definir una interfaz Java
- Seleccionar entre herencia de interfaz y herencia de clase
- Ampliar una interfaz
- Refactorizar código para implantar el patrón DAO



Generics y Collections



Objetivos

Al finalizar esta lección, debería estar capacitado para:

- Crear una clase genérica personalizada
- Utilizar el diamante de inferencia de tipo para crear un objeto
- Crear una recopilación sin utilizar genéricos
- Crear una recopilación mediante el uso de genéricos
- Implantar una ArrayList
- Implantar una interfaz Set
- Implantar un HashMap
- Implantar una pila mediante el uso de Deque
- Utilizar tipos enumerados



Genéricos

- Proporcionan una seguridad flexible al código.
- Mueven muchos errores comunes de tiempo de ejecución a tiempo de compilación.
- Proporcionan un código más limpio y fácil de escribir.
- Reducen la necesidad de conversión de recopilaciones.
- Se utilizan frecuentemente en la API Collections de Java.



Clase de caché simple sin genéricos

```
public class CacheString {  
    private String message = "";  
  
    public void add(String message) {  
        this.message = message;  
    }  
  
    public String get() {  
        return this.message;  
    }  
}
```

```
public class CacheShirt {  
    private Shirt shirt;  
  
    public void add(Shirt shirt) {  
        this.shirt = shirt;  
    }  
  
    public Shirt get() {  
        return this.shirt;  
    }  
}
```



Clase de caché genérica

```
1 public class CacheAny <T>{
2
3     private T t;
4
5     public void add(T t) {
6         this.t = t;
7     }
8
9     public T get() {
10        return this.t;
11    }
12 }
```



Fucionamiento de los genéricos

Compare los objetos de tipo restringido con las alternativas genéricas.

```
1 public static void main(String args[]) {  
2     CacheString myMessage = new CacheString(); // Type  
3     CacheShirt myShirt = new CacheShirt(); // Type  
4  
5     //Generics  
6     CacheAny<String> myGenericMessage = new CacheAny<String>();  
7     CacheAny<Shirt> myGenericShirt = new CacheAny<Shirt>();  
8  
9     myMessage.add("Save this for me"); // Type  
10    myGenericMessage.add("Save this for me"); // Generic  
11  
12 }
```



Genéricos con diamante de inferencia de tipo

- Sintaxis.
 - No es necesario repetir los tipos en la parte derecha de la sentencia.
 - Los paréntesis angulares indican el reflejo de los parámetros de tipo.
- Simplifica las declaraciones genéricas.
- Ahorra la introducción de datos.

```
//Generics
CacheAny<String> myMessage = new CacheAny<>();
```

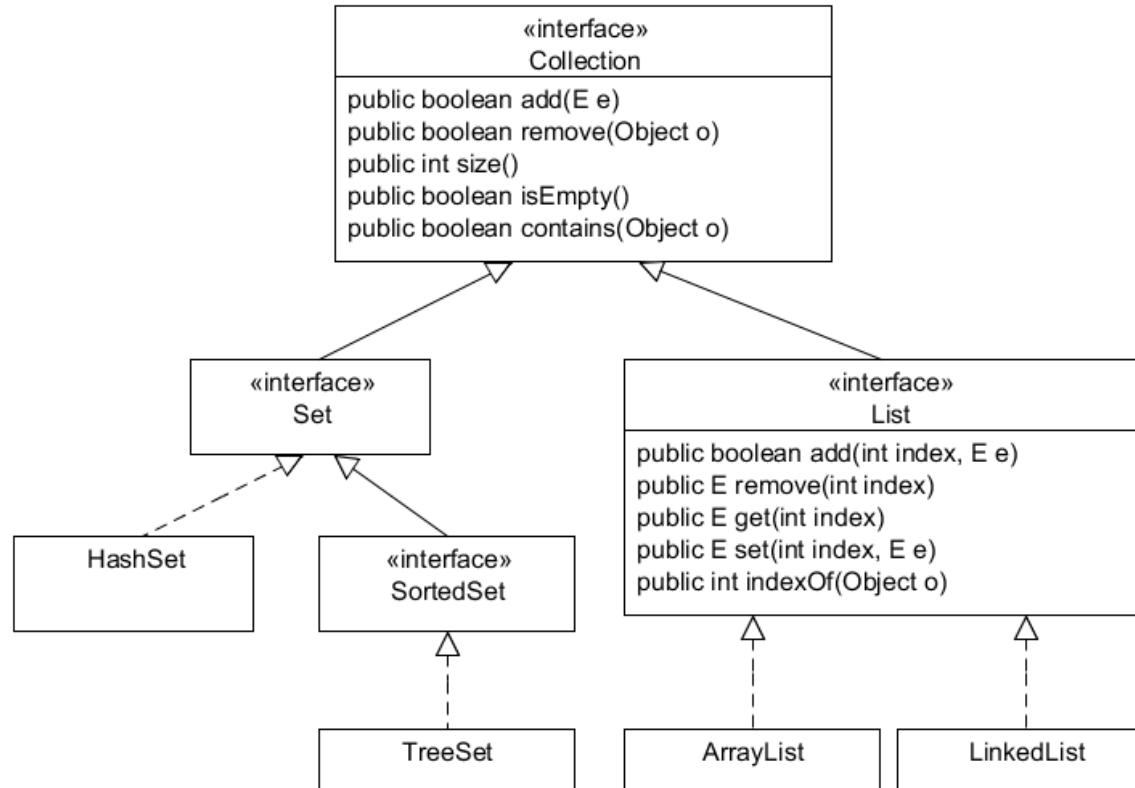


Recopilaciones

- Una recopilación es un objeto único diseñado para gestionar un grupo de objetos.
 - Los objetos de una recopilación se denominan *elementos*.
 - *No se permite el uso de primitivos en una recopilación.*
- Distintos tipos de recopilaciones implantan varias estructuras de datos comunes:
 - Pilas, colas, matrices dinámicas o elementos hash.
- La API Collections se basa, en gran medida, en los genéricos para su implantación.



Tipos de recopilaciones





Interfaz List

- List es una interfaz que define el comportamiento de una lista genérica.
 - Recopilación ordenada de elementos
- List incluye los siguientes comportamientos:
 - Adición de elementos en un índice específico
 - Adición de elementos al final de la lista
 - Obtención de un elemento basado en un índice
 - Eliminación de un elemento basado en un índice
 - Sobrescritura de un elemento basado en un índice
 - Obtención del tamaño de la lista
- Utilice List como un objeto de instancia para ocultar los detalles de la implantación.



Clase de implantación ArrayList

- Se trata de una matriz aumentable de forma dinámica.
 - La lista crece automáticamente si los elementos exceden el tamaño inicial.
- Tiene un índice numérico.
 - El índice accede a los elementos.
 - Los elementos se pueden insertar según el índice.
 - Los elementos se pueden sobrescribir.
- Permite los elementos duplicados.



ArrayList sin genéricos

```
1public class OldStyleArrayList {  
2    public static void main(String args[]) {  
3        List partList = new ArrayList(3);  
4  
5        partList.add(new Integer(1111));  
6        partList.add(new Integer(2222));  
7        partList.add(new Integer(3333));  
8        partList.add("Oops a string!");  
9  
10       Iterator elements = partList.iterator();  
11       while (elements.hasNext()) {  
12           Integer partNumberObject = (Integer) (elements.next()); // error?  
13           int partNumber = partNumberObject.intValue();  
14  
15           System.out.println("Part number: " + partNumber);  
16       }  
17   }  
18 }
```

Java example using syntax prior to Java 1.5.

Runtime error:
ClassCastException



ArrayList genérica

```
1  public class GenericArrayList {  
2      public static void main(String args[]) {  
3          List<Integer> partList = new ArrayList<>(3);  
4  
5          partList.add(new Integer(1111));  
6          partList.add(new Integer(2222));  
7          partList.add(new Integer(3333));  
8          partList.add("Bad Data"); // compile error now  
9  
10         Iterator<Integer> elements = partList.iterator();  
11         while (elements.hasNext()) {  
12             Integer partNumberObject = elements.next();  
13             int partNumber = partNumberObject.intValue();  
14  
15             System.out.println("Part number: " + partNumber);  
16         }  
17     }  
18 }
```

Java example using
SE 7 syntax.

No cast required.



ArrayList genérica: Iteración y empaquetado

```
for (Integer partNumberObj:partList) {  
    int partNumber = partNumberObj; // Demos auto  
    unboxing  
    System.out.println("Part number: " + partNumber);  
}
```

- El bucle for mejorado o el bucle for-each proporcionan un código más limpio.
- La conversión no se realiza debido al empaquetado automático y el desempaquetado.



Empaqueado automático y desempaqueado

- Simplifies syntax
- Produces cleaner, easier-to-read code

```
1 public class AutoBox {  
2     public static void main(String[] args){  
3         Integer intObject = new Integer(1);  
4         int intPrimitive = 2;  
5  
6         Integer tempInteger;  
7         int tempPrimitive;  
8  
9         tempInteger = new Integer(intPrimitive);  
10        tempPrimitive = intObject.intValue();  
11  
12        tempInteger = intPrimitive; // Auto box  
13        tempPrimitive = intObject; // Auto unbox
```



Interfaz Set

- Set es una lista que contiene solo elementos únicos.
- Una interfaz Set no tiene índice.
- No se permite el uso de elementos duplicados en una interfaz Set.
- Puede iterar con elementos para acceder a ellos.
- TreeSet ofrece una implantación ordenada.



Interfaz Set: ejemplo

Una interfaz Set es una recopilación de elementos únicos.

```
1 public class SetExample {  
2     public static void main(String[] args) {  
3         Set<String> set = new TreeSet<>();  
4  
5         set.add("one");  
6         set.add("two");  
7         set.add("three");  
8         set.add("three"); // not added, only unique  
9  
10        for (String item:set){  
11            System.out.println("Item: " + item);  
12        }  
13    }  
14 }
```



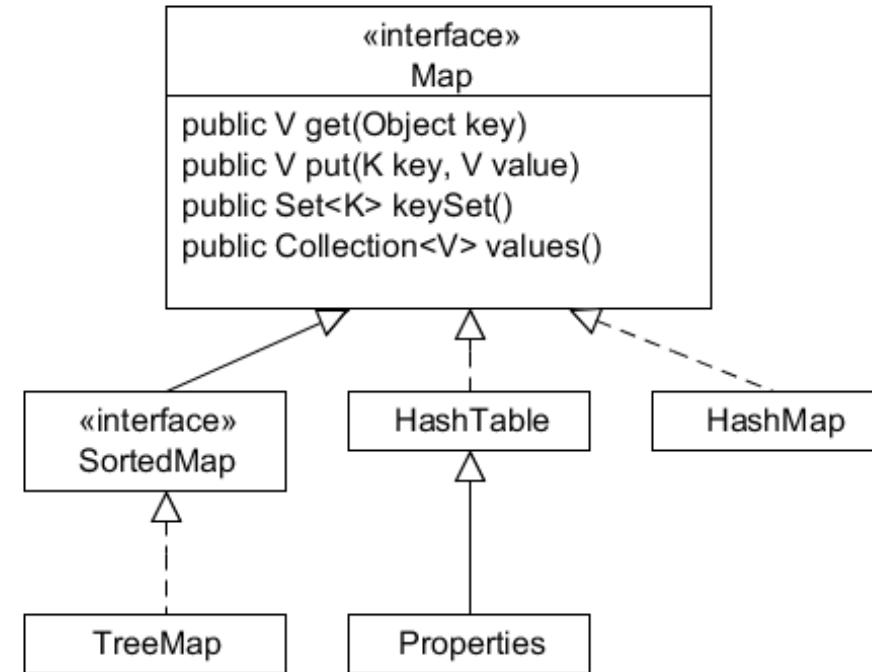
Interfaz Map

- Recopilación que almacena varios pares clave-valor.
 - Clave: identificador único de cada elemento de una recopilación
 - Valor: valor almacenado en el elemento asociado a la clave
- Se denomina “matriz asociativa” en otros lenguajes.

Key	Value
101	Blue Shirt
102	Black Shirt
103	Gray Shirt



Tipos de Map





Interfaz Map: ejemplo

```
1 public class MapExample {  
2     public static void main(String[] args){  
3         Map <String, String> partList = new TreeMap<>();  
4         partList.put("S001", "Blue Polo Shirt");  
5         partList.put("S002", "Black Polo Shirt");  
6         partList.put("H001", "Duke Hat");  
7  
8         partList.put("S002", "Black T-Shirt"); // Overwrite value  
9         Set<String> keys = partList.keySet();  
10  
11         System.out.println("== Part List ==");  
12         for (String key:keys){  
13             System.out.println("Part#: " + key + " " +  
14                             partList.get(key));  
15         }  
16     }  
17 }
```



Interfaz Deque

Recopilación que se puede utilizar como pila o cola.

- Significa “cola de dos extremos” (y se pronuncia “deck”).
- Una cola proporciona operaciones FIFO (primero en entrar, primero en salir).
 - Métodos add(e) y remove()
- Una pila proporciona operaciones LIFO (último en entrar, primero en salir).
 - Métodos push(e) y pop()



Pila con Deque: ejemplo

```
1 public class TestStack {  
2     public static void main(String[] args) {  
3         Deque<String> stack = new ArrayDeque<>();  
4         stack.push("one");  
5         stack.push("two");  
6         stack.push("three");  
7  
8         int size = stack.size() - 1;  
9         while (size >= 0 ) {  
10             System.out.println(stack.pop());  
11             size--;  
12         }  
13     }  
14 }
```

Ordenación de recopilaciones

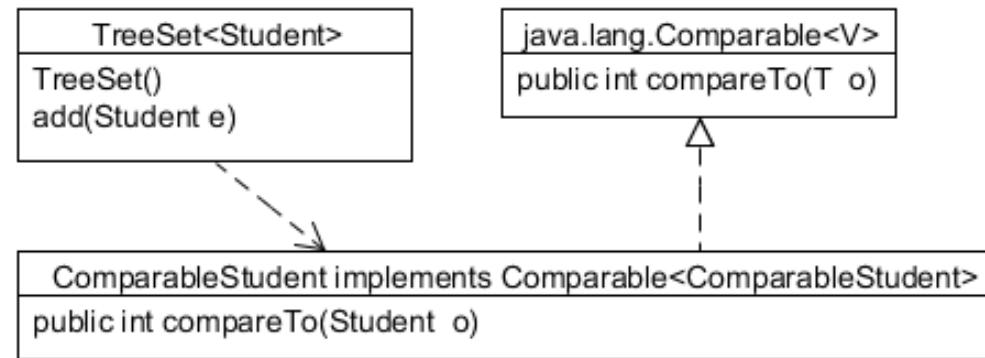
- Las interfaces Comparable y Comparator se utilizan para ordenar recopilaciones.
 - Las dos se implantan mediante el uso de genéricos.
- Uso de la interfaz Comparable:
 - Sustituye al método compareTo.
 - Proporciona una única opción de ordenación.
- Uso de la interfaz Comparator:
 - Se implanta mediante el uso del método compare.
 - Permite crear varias clases Comparator.
 - Permite crear y utilizar distintas opciones de ordenación.



Interfaz Comparable

Uso de la interfaz Comparable:

- Sustituye al método compareTo.
- Proporciona una única opción de ordenación.



Sorting logic is inside of the Student class.
Benefit: Sorting can leverage private fields to determine order.
Drawback: Only one ordering behavior is possible.



Comparable: ejemplo

```
1 public class ComparableStudent implements Comparable<ComparableStudent>{
2     private String name; private long id = 0; private double gpa = 0.0;
3
4     public ComparableStudent(String name, long id, double gpa) {
5         // Additional code here
6     }
7     public String getName() { return this.name; }
8     // Additional code here
9
10    public int compareTo(ComparableStudent s) {
11        int result = this.name.compareTo(s.getName());
12        if (result > 0) { return 1; }
13        else if (result < 0){ return -1; }
14        else { return 0; }
15    }
16 }
```

Prueba de Comparable: ejemplo

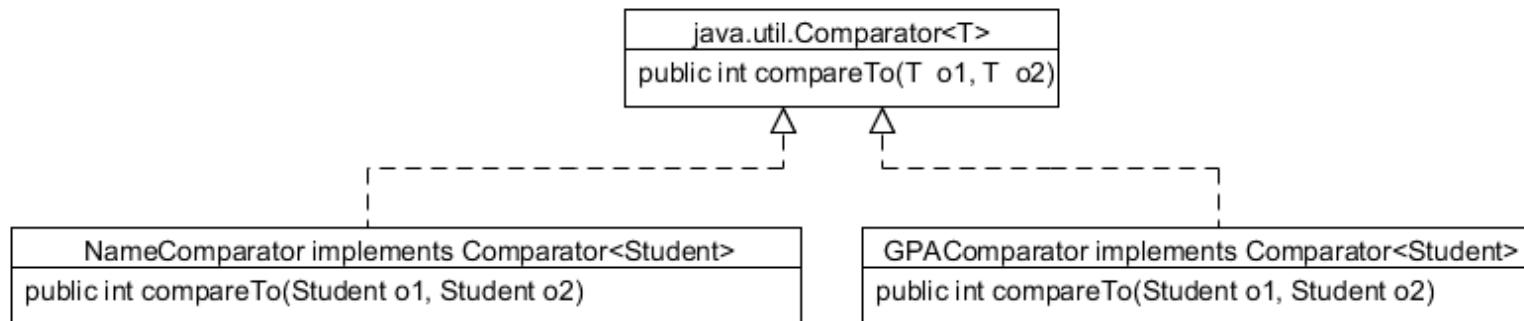
```
public class TestComparable {  
    public static void main(String[] args) {  
        Set<ComparableStudent> studentList = new TreeSet<>();  
  
        studentList.add(new ComparableStudent("Thomas Jefferson", 1111, 3.8));  
        studentList.add(new ComparableStudent("John Adams", 2222, 3.9));  
        studentList.add(new ComparableStudent("George Washington", 3333, 3.4));  
  
        for(ComparableStudent student:studentList){  
            System.out.println(student);  
        }  
    }  
}
```



Interfaz Comparator

Uso de la interfaz Comparator:

- Se implanta mediante el uso del método compare.
- Permite crear varias clases Comparator.
- Permite crear y utilizar distintas opciones de ordenación.



Sorting logic is outside of the Student class.

Benefit: Changeable ordering behaviors are possible.

Drawback: Sorting can not leverage private fields to determine order.



Comparator: ejemplo

```
public class StudentSortName implements Comparator<Student>{
    public int compare(Student s1, Student s2){
        int result = s1.getName().compareTo(s2.getName());
        if (result != 0) { return result; }
        else {
            return 0; // Or do more comparing
        }
    }
}
```

```
public class StudentSortGpa implements Comparator<Student>{
    public int compare(Student s1, Student s2){
        if (s1.getGpa() < s2.getGpa()) { return 1; }
        else if (s1.getGpa() > s2.getGpa()) { return -1; }
        else { return 0; }
    }
}
```

Here the compare logic is reversed and results in descending order.



Prueba de Comparator: ejemplo

```
1 public class TestComparator {  
2     public static void main(String[] args){  
3         List<Student> studentList = new ArrayList<>(3);  
4         Comparator<Student> sortName = new StudentSortName();  
5         Comparator<Student> sortGpa = new StudentSortGpa();  
6  
7         // Initialize list here  
8  
9         Collections.sort(studentList, sortName);  
10        for(Student student:studentList){  
11            System.out.println(student);  
12        }  
13  
14        Collections.sort(studentList, sortGpa);  
15        for(Student student:studentList){  
16            System.out.println(student);  
17        }  
18    }  
19 }
```



Resumen

En esta lección, debe haber aprendido a hacer lo siguiente:

- Crear una clase genérica personalizada
- Utilizar el diamante de inferencia de tipo para crear un objeto
- Crear una recopilación sin utilizar genéricos
- Crear una recopilación mediante el uso de genéricos
- Implantar una ArrayList
- Implantar una interfaz Set
- Implantar un HashMap
- Implantar una pila mediante el uso de Deque
- Utilizar tipos enumerados.

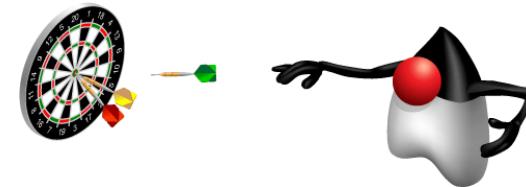


Interfaces and Lambda Expressions



Objectives

- After completing this lesson, you should be able to do the following:
 - Define a Java interface
 - Choose between interface inheritance and class inheritance
 - Extend an interface
 - Define a lambda expression





Java Interfaces

- Java interfaces are used to define abstract types.
Interfaces:
 - Are similar to abstract classes containing only public abstract methods
 - Outline methods that must be implemented by a class
 - Methods must not have an implementation {braces}.
 - Can contain constant fields
 - Can be used as a reference type
 - Are an essential component of many design patterns

A Problem Solved by Interfaces

- **Given:** A company sells an assortment of products, very different from each other, and needs a way to access financial data in a similar manner.
 - Products include:
 - Crushed Rock
 - Measured in pounds
 - Red Paint
 - Measured in gallons
 - Widgets
 - Measured by Quantity
 - Need to calculate per item
 - Sales price
 - Cost
 - Profit



CrushedRock Class

The CrushedRock class before interfaces

```
public class CrushedRock {  
    private String name;  
    private double salesPrice = 0;  
    private double cost = 0;  
    private double weight = 0; // In pounds  
  
    public CrushedRock(double salesPrice, double cost,  
        double weight){  
        this.salesPrice = salesPrice;  
        this.cost = cost;  
        this.weight = weight;  
    }  
}
```

The SalesCalcs Interface

The SalesCalcs interface specifies the types of calculations required for our products.

Public, top-level interfaces are declared in their own .java file.

```
public interface SalesCalcs {  
    public String getName();  
    public double calcSalesPrice();  
    public double calcCost();  
    public double calcProfit();  
}
```



Adding an Interface

The updated CrushedRock class implements SalesCalcs.

```
public class CrushedRock implements SalesCalcs{
    private String name = "Crushed Rock";
    ... // a number of lines not shown
    @Override
    public double calcCost(){
        return this.cost * this.weight;
    }

    @Override
    public double calcProfit(){
        return this.calcSalesPrice() - this.calcCost();
    }
}
```



Interface References

Any class that implements an interface can be referenced by using that interface.

Notice how the calcSalesPrice method can be referenced by the CrushedRock class or the SalesCalcs interface.

```
CrushedRock rock1 = new CrushedRock(12, 10, 50);
SalesCalcs rock2 = new CrushedRock(12, 10, 50);
System.out.println("Sales Price: " + rock1.calcSalesPrice());
System.out.println("Sales Price: " + rock2.calcSalesPrice());
```

Output

```
Sales Price: 600.0
Sales Price: 600.0
```

Interface Reference Usefulness

Any class implementing an interface can be referenced by using that interface. For example:

```
SalesCalcs[] itemList = new SalesCalcs[5];  
ItemReport report = new ItemReport();  
  
itemList[0] = new CrushedRock(12.0, 10.0, 50.0);  
itemList[1] = new CrushedRock(8.0, 6.0, 10.0);  
itemList[2] = new RedPaint(10.0, 8.0, 25.0);  
itemList[3] = new Widget(6.0, 5.0, 10);  
itemList[4] = new Widget(14.0, 12.0, 20);  
  
System.out.println("==Sales Report==");  
for(SalesCalcs item:itemList){  
    report.printItemData(item);  
}
```



Interface Code Flexibility

A utility class that references the interface can process any implementing class.

```
public class ItemReport {  
    public void printItemData(SalesCalcs item) {  
        System.out.println("--" + item.getName() + " Report--");  
        System.out.println("Sales Price: " + item.calcSalesPrice());  
        System.out.println("Cost: " + item.calcCost());  
        System.out.println("Profit: " + item.calcProfit());  
    }  
}
```

default Methods in Interfaces

Java 8 has added `default` methods as a new feature:

```
public interface SalesCalcs {  
    ... // A number of lines omitted  
    public default void printItemReport() {  
        System.out.println("--" + this.getName() + " Report--");  
        System.out.println("Sales Price: " + this.calcSalesPrice());  
        System.out.println("Cost: " + this.calcCost());  
        System.out.println("Profit: " + this.calcProfit());  
    }  
}
```

`default` methods:

- Are declared by using the keyword `default`
- Are fully implemented methods within an interface
- Provide useful inheritance mechanics



default Method: Example

Here is an updated version of the item report using default methods.

```
SalesCalcs[] itemList = new SalesCalcs[5];  
  
itemList[0] = new CrushedRock(12, 10, 50);  
itemList[1] = new CrushedRock(8, 6, 10);  
itemList[2] = new RedPaint(10, 8, 25);  
itemList[3] = new Widget(6, 5, 10);  
itemList[4] = new Widget(14, 12, 20);  
  
System.out.println("==Sales Report==");  
for(SalesCalcs item:itemList){  
    item.printItemReport();  
}
```



static Methods in Interfaces

- Java 8 allows `static` methods in an interface. So it is possible to create helper methods like the following.

```
• public interface SalesCalcs {  
•     ... // A number of lines omitted  
•     public static void printItemArray(SalesCalcs[] items){  
•         System.out.println(reportTitle);  
•         for(SalesCalcs item:items){  
•             System.out.println("--" + item.getName() + " Report--");  
•             System.out.println("Sales Price: " + item.calcSalesPrice());  
•             System.out.println("Cost: " + item.calcCost());  
•             System.out.println("Profit: " + item.calcProfit());  
•         }  
•     }  
• }
```



Constant Fields

Interfaces can have constant fields.

```
public interface SalesCalcs {  
    public static final String reportTitle="\n==Static List Report==";  
    ... // A number of lines omitted
```



Extending Interfaces

Interfaces can extend interfaces:

```
public interface WidgetSalesCalcs extends SalesCalcs{  
    public String getWidgetType();  
}
```

So now any class implementing WidgetSalesCalc must implement all the methods of SalesCalcs in addition to the new method specified here.



Implementing and Extending

Classes can extend a parent class and implement an interface:

```
public class WidgetPro extends Widget implements WidgetSalesCalcs{  
    private String type;  
  
    public WidgetPro(double salesPrice, double cost, long quantity, String type) {  
        super(salesPrice, cost, quantity);  
        this.type = type;  
    }  
  
    public String getWidgetType() {  
        return type;  
    }  
}
```



Anonymous Inner Classes

Define a class in place instead of in a separate file

Why would you do this?

- Logically group code in one place
- Increase encapsulation
- Make code more readable

StringAnalyzer interface

```
public interface StringAnalyzer {  
    public boolean analyze(String target, String searchStr);  
}
```

A single method interface

- **Functional Interface**

Takes two strings and returns a boolean



Anonymous Inner Class: Example

Example method call with concrete class

```
20     // Call concrete class that implements StringAnalyzer
21     ContainsAnalyzer contains = new ContainsAnalyzer();
22
23     System.out.println("====Contains====");
24     Z03Analyzer.searchArr(strList01, searchStr, contains);
```

Anonymous inner class example

```
22     Z04Analyzer.searchArr(strList01, searchStr,
23             new StringAnalyzer(){
24                 @Override
25                 public boolean analyze(String target, String searchString){
26                     return target.contains(searchString);
27                 }
28             });

```

The class is created in place.

String Analysis Regular Class

Class analyzes an array of strings given a search string

- Print strings that contain the search string
- Other methods could be written to perform similar string test

Regular Class Example method

```
1 package com.example;  
2  
3 public class AnalyzerTool {  
4     public boolean arrContains(String sourceStr, String searchStr){  
5         return sourceStr.contains(searchStr);  
6     }  
7 }  
8
```

String Analysis Regular Test Class

Here is the code to test the class, Z01Analyzer

```
4  public static void main(String[] args) {  
5      String[] strList =  
6          {"tomorrow", "toto", "to", "timbukto", "the", "hello", "heat"};  
7      String searchStr = "to";  
8      System.out.println("Searching for: " + searchStr);  
9  
10     // Create regular class  
11     AnalyzerTool analyzeTool = new AnalyzerTool();  
12  
13     System.out.println("==Contains==");  
14     for(String currentStr:strList){  
15         if  (analyzeTool.arrContains(currentStr, searchStr)){  
16             System.out.println("Match: " + currentStr);  
17         }  
18     }  
19 }
```

String Analysis Interface: Example

What about using an interface?

```
3 public interface StringAnalyzer {  
4     public boolean analyze(String sourceStr, String searchStr);  
5 }
```

`StringAnalyzer` is a single method functional interface.

Replacing the previous example and implementing the interface looks like this:

```
3 public class ContainsAnalyzer implements StringAnalyzer {  
4     @Override  
5     public boolean analyze(String target, String searchStr) {  
6         return target.contains(searchStr);  
7     }  
8 }
```

String Analyzer Interface Test Class

```
4  public static void main(String[] args) {
5      String[] strList =
6
7      {"tomorrow", "toto", "to", "timbukto", "the", "hello", "heat"};
8      String searchStr = "to";
9      System.out.println("Searching for: " + searchStr);
10
11     // Call concrete class that implements StringAnalyzer
12     ContainsAnalyzer contains = new ContainsAnalyzer();
13
14     System.out.println("====Contains====");
15     for(String currentStr:strList){
16         if (contains.analyze(currentStr, searchStr)){
17             System.out.println("Match: " + currentStr);
18         }
19     }
```



Encapsulate the for Loop

An improvement to the code is to encapsulate the forloop:

```
3 public class Z03Analyzer {  
4  
5     public static void searchArr(String[] strList, String searchStr, StringAnalyzer  
analyzer){  
6         for(String currentStr:strList){  
7             if (analyzer.analyze(currentStr, searchStr)){  
8                 System.out.println("Match: " + currentStr);  
9             }  
10        }  
11    }  
// A number of lines omitted
```



String Analysis Test Class with Helper Method

With the helper method, the main method shrinks to this:

```
13 public static void main(String[] args) {  
14     String[] strList01 =  
15         {"tomorrow", "toto", "to", "timbukto", "the", "hello", "heat"};  
16     String searchStr = "to";  
17     System.out.println("Searching for: " + searchStr);  
18  
19     // Call concrete class that implements StringAnalyzer  
20     ContainsAnalyzer contains = new ContainsAnalyzer();  
21  
22     System.out.println("====Contains====");  
23     Z03Analyzer.searchArr(strList01, searchStr, contains);  
24 }
```



String Analysis Anonymous Inner Class

Create anonymous inner class for third argument.

```
19     // Implement anonymous inner class
20     System.out.println("==Contains==");
21     Z04Analyzer.searchArr(strList01, searchStr,
22         new StringAnalyzer(){
23             @Override
24             public boolean analyze(String target, String
searchStr) {
25                 return target.contains(searchStr);
26             }
27         });
28 }
```

String Analysis Lambda Expression

Use lambda expression for the third argument.

```
13  public static void main(String[] args) {  
14      String[] strList =  
15          {"tomorrow", "toto", "to", "timbukto", "the", "hello", "heat"};  
16      String searchStr = "to";  
17      System.out.println("Searching for: " + searchStr);  
18  
19      // Lambda Expression replaces anonymous inner class  
20      System.out.println("==Contains==");  
21      Z05Analyzer.searchArr(strList, searchStr,  
22          (String target, String search) -> target.contains(search));  
23  }
```



Lambda Expression Defined

Argument List	Arrow Token	Body
(int x, int y)	->	x + y

Basic Lambda examples

```
(int x, int y) -> x + y  
(x, y) -> x + y
```

```
(x, y) -> { System.out.println(x + y); }
```

```
(String s) -> s.contains("word")  
s -> s.contains("word")
```





What Is a Lambda Expression?

```
(t,s) -> t.contains(s)
```

ContainsAnalyzer.java

```
public class ContainsAnalyzer implements StringAnalyzer {  
    public boolean analyze(String target, String  
    searchStr) {  
        return target.contains(searchStr);  
    }  
}
```



What Is a Lambda Expression?

ContainsAnalyzer.java

```
public class ContainsAnalyzer implements StringAnalyzer {  
    public boolean analyze(String target, String searchStr) {  
        return target.contains(searchStr);  
    }  
}
```

(t, s) -> t.contains(s)

Both have parameters





What Is a Lambda Expression?

ContainsAnalyzer.java

```
public class ContainsAnalyzer implements StringAnalyzer {  
    public boolean analyze(String target, String searchStr) {  
        return target.contains(searchStr);  
    }  
}
```

(t, s) -> t.contains(s)

Both have parameters

Both have a body
with one or more
statements



Lambda Expression Shorthand

Lambda expressions using shortened syntax

```
20    // Use short form Lambda
21    System.out.println("==Contains==");
22    Z06Analyzer.searchArr(strList01, searchStr,
23        (t, s) -> t.contains(s));
24
25    // Changing logic becomes easy
26    System.out.println("==Starts With==");
27    Z06Analyzer.searchArr(strList01, searchStr,
28        (t, s) -> t.startsWith(s));
```

The `searchArr` method arguments are:

```
public static void searchArr(String[] strList, String searchStr,
    StringAnalyzer analyzer)
```



Lambda Expressions as Variables

Lambda expressions can be treated like variables.
They can be assigned, passed around, and reused.

```
19     // Lambda expressions can be treated like variables
20     StringAnalyzer contains = (t, s) -> t.contains(s);
21     StringAnalyzer startsWith = (t, s) -> t.startsWith(s);
22
23     System.out.println("==Contains==");
24     Z07Analyzer.searchArr(strList, searchStr,
25         contains);
26
27     System.out.println("==Starts With==");
28     Z07Analyzer.searchArr(strList, searchStr,
29         startsWith);
```



Summary

In this lesson, you should have learned how to:

- Define a Java interface
- Choose between interface inheritance and class inheritance
- Extend an interface
- Define a Lambda Expression





Quiz

- All methods in an interface are:
 - a.final
 - b.abstract
 - c.private
 - d.volatile



Quiz

When a developer creates an anonymous inner class, the new class is typically based on which one of the following?

- a. enums
- b. Executors
- c. Functional interfaces
- d. Static variables



Quiz

Which is true about the parameters passed into the following lambda expression?

(t, s) -> t.contains(s)

- a. Their type is inferred from the context.
- b. Their type is executed.
- c. Their type must be explicitly defined.
- d. Their type is undetermined.



CJAVA

siempre para apoyarte

#CJavaNoPara

Gracias