

# SCJP – Capitulo 6

Orientación a Objetos, sobrecarga, sobrescritura, constructores y tipos de retorno

# Agenda

- Beneficios de la Encapsulación
- Sobrecarga y Sobrescritura
- Constructores e inicialización
- Tipos de retorno

# Encapsulación

```
public class Tiempo{  
    public int hora;  
    public int minuto;  
    public int segundo;  
}  
  
public class ExploitTiempo {  
    public static void main (String[ ] args){  
        Tiempo tiempo = new Tiempo();  
        tiempo.hora = -77;  
        ...  
    }  
}
```

Asignación correcta (en sintaxis)

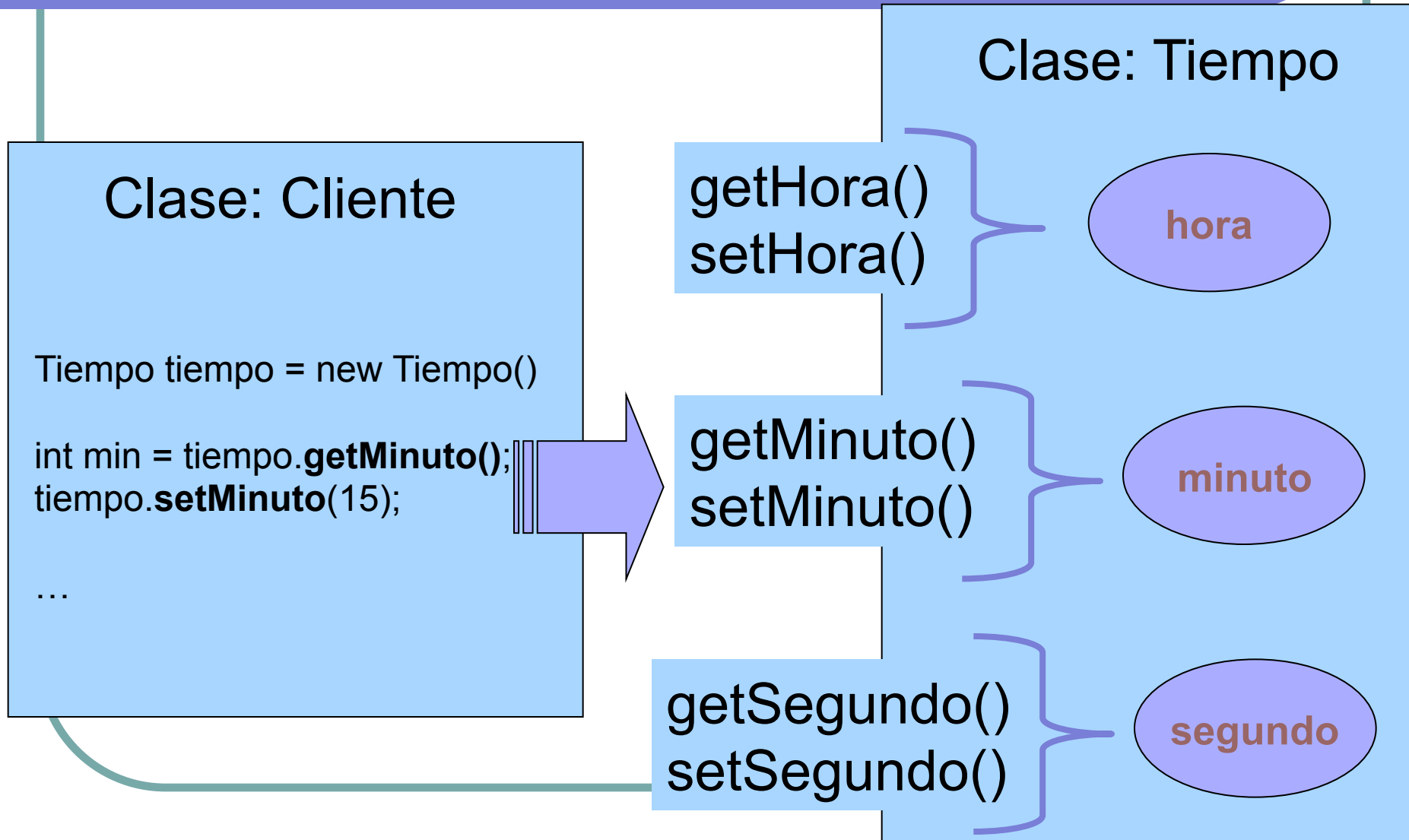
Pero sin inconsistente (semánticamente)...

# Encapsulación

- Si Ud quiere Mantenibilidad, Flexibilidad y Extensibilidad (y por supuesto que quiere).  
Diseñe con Encapsulación
  - Mantenga sus Atributos protegidos (mayormente con `private`)
  - Haga `public` los métodos para acceder a los atributos (y fuerce su código que los usen)
  - Para los métodos, use la convención de nombres de los JavaBeans: `set<NombreAtributo>`, `get<NombreAtributo>`

*Esconda los detalles de la implementación detrás de una interfaz pública*

# Encapsulación



# Encapsulación

```
public class Tiempo{  
    private int hora;  
    private int minuto;  
    private int segundo;  
    ...  
    public int getMinuto(){  
        return min;  
    }  
    public void setMinuto(int min){  
        minuto = min;  
    }  
    ...  
}
```

Mmmmm...

Pero quedo igual???

# Encapsulación

```
public class Tiempo{  
    private int hora;  
    private int minuto;  
    private int segundo;  
    ...  
    public int getMinuto(){  
        return min;  
    }  
    public void setMinuto(int min){  
        if (min < 0) restarHoras(min);  
        if (min > 59) sumarHoras(min);  
        minuto = extraerMinutos(min);  
    }  
    ...  
}
```

Ahora tiene la flexibilidad de poder cambiar la implementación sin afectar a los clientes...

... mientras no se modifique su interfaz pública...

# IS-A (es un) - Herencia

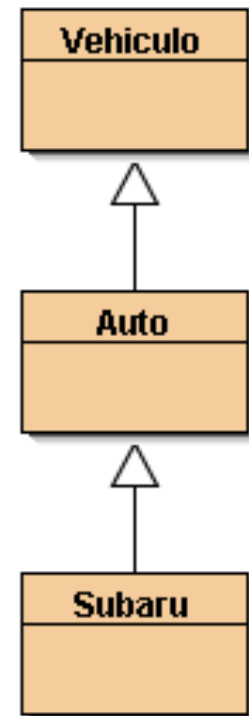
```
public class Vehiculo { }  
public class Auto extends Vehiculo { }  
public class Subaru extends Auto { }
```

Auto **es un** Vehiculo

Subaru **es un** Auto

Subaru **es un** Vehiculo

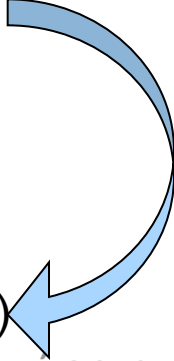
La relación **IS-A** debe pasar la validación del operador **instanceof**





# HAS-A (usa un)

```
public class Persona { }
public class Alumno extends Persona {
    private Nota nota1;
    public void setNota1(double valor){
        nota1.seValor(valor); // se delega el comportamiento al objeto Nota
    }
}
...
public Nota {
    private double valor;
    public void setValor(double)
        this.valor = valor; // aqui se puede realizar validaciones
    }
    ...
}
```



Alumno **es una** Persona

Alumno **usa una** Nota

# Sobrescribiendo Métodos

```
public class Animal {  
    public void dormir() {  
        System.out.println("acostarse");  
    }  
}  
  
public class Caballo extends Animal {  
    public void dormir() {  
        System.out.println("cerrar los ojos");  
    }  
}
```

# Sobrescribiendo Métodos

```
public class Animal {  
    public void dormir() {  
        System.out.println("acostarse");  
    }  
}  
  
public class Perro extends Animal {  
    public void dormir() {  
        System.out.print("una vuelta, ");  
        super.dormir(); // reutiliza funcionalidad del padre  
    }  
}
```

# JVM – Virtual Method Invocation

```
public class TestAnimales {  
    public static void main (String[ ] args){  
        Animal a = new Animal();  
        Animal b = new Caballo();  
        Animal c = new Perro();  
        a.dormir(); // acostarse  
        b.dormir(); // cerrar los ojos  
        c.dormir(); // una vuelta, acostarse  
    }  
}
```

# Reglas para sobrescribir métodos

- La lista de argumentos debe ser exactamente igual al del método sobrescrito.
- El tipo de retorno debe ser exactamente igual al del método sobrescrito.
- El nivel de acceso no debe ser más restrictivo que el del método sobrescrito.

# Reglas para sobrescribir métodos

- El método sobre escrito no debe lanzar nuevas o más amplias excepciones chequeadas.
- No se pueden sobre escribir métodos marcados como *final*.
- Los métodos que no pueden ser heredados no pueden ser sobrescritos.

# Sobrecargando Métodos

El siguiente método:

```
public void caracteristicas (int tamaño, String nombre,  
    float patron);
```

Puede tener las siguientes versiones sobrecargadas

```
private void caracteristicas (int tamaño, String nombre);
```

```
public int caracteristicas (int tamaño, float patron);
```

```
public void caracteristicas (float patron, String nombre)  
    throws IOException ;
```

# Reglas para sobrecargar métodos

- Debe cambiar su listado de argumentos.
- Puede cambiar su tipo de retorno.
- Puede cambiar su modificados de acceso.
- Puede declarar nuevas excepciones chequeadas.
- El método se puede recargar en la misma clase o en una subclase.



# La Sobrecarga es por el tipo

```
public UsarAnimales {  
    public char usar(Animal a) { return 'A'; }  
    public char usar(Caballo c) { return 'C'; }  
    public static void main (String[ ] args){  
        UsarAnimales ua = new UsarAnimales();  
        Animal a = new Animal();  
        Caballo c = new Caballo();  
        Animal animalRefCaballo = new Caballo();  
  
        ua.usar(a); // Retorna 'A'  
        ua.usar(c); // Retorna 'C'  
        ua.usar(animalRefCaballo); // Retorna 'A'  
    }  
}
```

Nótese que la variable local `animalRefCaballo` utiliza el método sobrescrito

`usar(Animal a)`

Al contrario de la `Virtual Method Invocation`

# Constructores

```
public class Persona {  
    String nombre;  
    int edad;  
    Persona(){  
        nombre = "Rodrigo";  
        edad = 30  
    }  
    public Persona (String nombre, int edad){  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
    ...  
}
```

La responsabilidad de un constructor es “inicializar” un objeto con un estado válido

Es invocado al momento de instancia la clase (con el operador **new**)

crea una objeto p1 con los atributos nombre = “Rodrigo”, edad=30  
`Persona p1 = new Persona();`

crea una objeto p1 con los atributos nombre = “Max”, edad=2  
`Persona p2 = new Persona(“Max”, 2);`

# El constructor por defecto

```
public class Animal { }
```

```
...
```

```
public class Animal extends Object {
```

```
    public Animal(){
```

```
        super();
```

```
    }
```

```
}
```

¿que constructor se invoca?  
Animal a = **new** Animal();

Si no se coloca una Herencia, se hereda de Object

Si no hay constructor, se crea uno por defecto

Si la primera línea del constructor no llama a otro constructor sobrecargado **this (...)**, se agrega una llamada al constructor no-args del padre **super( )**

Si el padre no tiene un constructor no-args... NO COMPILA!!!

# Reglas para constructores

- Los constructores pueden utilizar cualquier modificador de acceso (incluso *private*).
- El nombre del constructor debe ser igual al nombre de la clase.
- Los constructores no deben tener tipo de retorno.

# Reglas para constructores

- Si tu no escribes el constructor dentro de tu código de la clase, un constructor por omisión (default) será generado automáticamente por el compilador.
- El constructor por omisión (default) no tiene argumentos.
- Todo constructor debe tener en su primera sentencia una llamada a un constructor sobrecargado (`this()`) o una llamada a un constructor de la súper clase (`super()`)

# Reglas para constructores

- Una llamada `super ()` puede ser sin argumentos o puede incluir argumentos para pasar al constructor de la súper clase.
- Un constructor sin argumentos no es necesariamente el constructor por omisión (default), aunque el constructor por omisión es siempre sin argumentos, el constructor por omisión es el que te provee el compilador.

# Reglas para constructores

- Tu no puedes hacer una llamada a un método de instancia o acceder a variables de distancia hasta después de que el constructor de la super clase se ejecute.
- Tu puedes acceder a variables y métodos de clase (static) aunque sólo puede ser como parte de la llamada a `super ()` o `this ()`.

# Reglas para constructores

- Las clases abstractas (abstract) tienen constructores y esos constructores siempre son llamados cuando una subclase concreta es instanciada.
- Las interfaces no tienen constructores (las interfaces no son parte del árbol de herencias de los objetos).
- La única forma de que un constructor pueda ser llamado es desde otro constructor.



# Cadena de Constructores

```
public class Persona {  
    String nombre;  
    int edad;  
    Persona(){  
        this("Rodrigo", 30);  
    }  
    public Persona (String nombre, int edad){  
        super(); // si no existe, el compilador la coloca  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
    ...  
}
```

1ro : es invocado un constructor sobrecargado (opcional)

2do : es invocado el constructor del padre (hasta llegar a la clase Object)

3do : una vez que se inicializa Object, se comienzan a devolver inicializando todos los objetos de la jerarquía

# Tipos de Retorno : las 6 reglas

- 1. Puedes devolver **null** si el tipo de retorno es una referencia a objeto

```
public String hacerAlgo(){  
    return null;  
}
```

# Tipos de Retorno : las 6 reglas

- 2. Un Arreglo es un tipo de retorno valido

```
public String[ ] nombres() {  
    return new String[ ]{ "Pedro", "Pablo" }  
}
```

# Tipos de Retorno : las 6 reglas

- 3. Si el tipo de retorno es primitivo, puede devolver cualquier tipo que sea implícitamente convertible

```
public int getNumero(){  
    char c = 'c';  
    return c; // char es compatible con int  
}
```

# Tipos de Retorno : las 6 reglas

- 4. Si el tipo de retorno es primitivo, puede devolver cualquier tipo que sea explícitamente convertido

```
public int getNumero(){  
    float f = 32.5f;  
    return (int) f;  
}
```

# Tipos de Retorno : las 6 reglas

- 5. No puedes retornar “algo” si el tipo de retorno es **void**

```
public void hacerAlgo ( ) {  
    return “esto es lo que hago”;  
}
```

```
public void hacerAlgo ( ) {  
    return; // no se esta retornando algún valor  
}
```

# Tipos de Retorno : las 6 reglas

- 6. Si el tipo de retorno es una referencia a objeto, puede devolver cualquier objeto que sea implícitamente convertido

```
public Animal getAnimal(){  
    return new Caballo(); // Caballo IS-A Animal  
}  
public Object getObject(){  
    return new int[ ]{1, 2, 3}; // arreglo es Object  
}
```

# Tipos de Retorno : las 6 reglas

- 6. ... continuación ...

```
interface Volar{ }
```

```
class Avion implements Volar { }
```

```
...
```

```
public Volar getVolar() {
```

```
    return new Avion(); // interfaz implementada
```

```
}
```