




Fundamentals of the Java Programming Language (SL-110)



OptaSmart IT Learning Services ©



Módulo 1 – Tecnología Java



OptaSmart IT Learning Services ©

Origen de Java

- Creado en 1991 por James Gosling y otros en Sun Microsystems.
- Fue lanzado en 1994.
- Inicialmente fue creado para reemplazar a C++
- Se desarrolló con la intención de usarlo en aparatos electrodomésticos inteligentes.

OptaSmart IT Learning Services ©

Objetivos durante la creación de Java

- Usar Orientación a Objetos
- El mismo programa debe poder ser ejecutado en distintas plataformas
- Proveer soporte para usar redes de computadoras
- Estar diseñado para ejecutar código desde lugares remotos en forma segura
- Tomar prestadas las partes buenas de otros lenguajes OO como C++

OptaSmart IT Learning Services ©

Ediciones de Java

- Estándar (J2SE)
 - 1.0, 1.1, 1.2, 1.3, 1.4, 5.0
- Enterprise (J2EE)
 - 1.2, 1.3, 1.4
- Micro (J2ME, para PDAs y teléfonos celulares)

OptaSmart IT Learning Services ©

Características de Java

- Lenguaje de 3ra. Generación
- Lenguaje Compilado
- Utiliza una Máquina Virtual
- Es sensible a mayúsculas/minúsculas

OptaSmart IT Learning Services ©

Orientación a objetos - 1

- Java es Orientado a Objetos, no es *imperativo*.

Programa en Lenguaje Imperativo:	Lista de instrucciones que una computadora debe ejecutar.
Programa en Lenguaje Orientado a Objetos:	Colección de Objetos que cooperan entre sí.

OptaSmart IT Learning Services ©

Orientación a objetos - 2

- Software diseñado de tal manera que los varios tipos de datos involucrados estén combinados con las operaciones que les sean relevantes.
- Datos y código se unen en entidades llamadas **objetos**.
- El objetivo es separar las cosas que cambian de las que no. Por lo general, un cambio a una estructura de datos requiere un cambio al código que opera sobre esa estructura o viceversa.

OptaSmart IT Learning Services ©

Orientación a objetos - 3

- Esta separación en objetos coherentes provee un diseño de software más sólido
→ los proyectos grandes no son tan difíciles de manejar.
- Otro objetivo de la OO es desarrollar objetos más genéricos de al manera que puedan ser reutilizados entre distintos proyectos.

OptaSmart IT Learning Services ©

Orientación a objetos – 4

- 3 principales características de los objetos:
 - Herencia
 - Polimorfismo
 - Encapsulación
- Un objeto puede representar un sistema o un componente del mismo.
- Una clase es un **prototipo** de un objeto.

OptaSmart IT Learning Services ©

Orientación a objetos - 5

- En un sistema determinado, puede existir la clase Gato. El prototipo define que los objetos de la clase Gato cuentan con 4 patas y pueden realizar operaciones como comer, dormir y brincar.

```
class Gato {  
    int patas = 4;  
    void comer() {}  
    void dormir() {}  
    void brincar() {}  
}
```

OptaSmart IT Learning Services ©

Herencia

- La clase Gato desciende de la clase Mamífero. Puede **heredar** características (atributos) o comportamientos (métodos). De esta manera se reutiliza código.

```
class Mamifero {  
    int patas = 4;  
    void comer() {}  
    void dormir() {}  
}  
  
class Gato extends Mamifero {  
    void brincar() {}  
}
```

OptaSmart IT Learning Services ©

Polimorfismo

- El comportamiento de la clase Gato no tiene que ser siempre el mismo.


```
class Gato {  
    void comer(Pollo pollo) {}  
    void comer(Pescado pescado) {}  
}
```

OptaSmart IT Learning Services ©


Encapsulamiento

- Separación entre la representación de los datos y las aplicaciones que usan los datos en un nivel lógico.
- Característica de los lenguajes de programación que favorece el encubrimiento de la información.

OptaSmart IT Learning Services ©



Módulo 2 – Analizar un problema y diseñar una solución



OptaSmart IT Learning Services ©

Fases del desarrollo

- Incepción
- Elaboración
- Construcción
- Transición

OptaSmart IT Learning Services ©

Análisis basado en casos de uso

- Un sistema existe para servir a sus usuarios (personas, otros sistemas)
- La interacción entre un sistema y sus usuarios es un caso de uso
- Los casos de uso capturan requerimiento funcionales

OptaSmart IT Learning Services ©

Análisis basado en la arquitectura

- La arquitectura son las diferentes formas de ver un sistema
 - Según los requerimientos del usuario
 - Según la plataforma en que correrá
 - Según los componentes que reutilizará
 - Según la forma en que será distribuido
 - Según requerimientos no-funcionales (seguridad, rapidez)
- La arquitectura le da forma al sistema

OptaSmart IT Learning Services ©

El análisis es iterativo e incremental

- Es práctico dividir los sistemas en piezas que serán realizadas en mini-proyectos
- Cada mini-proyecto es una **iteración**
- Cada iteración genera un avance o un **incremento**

OptaSmart IT Learning Services ©

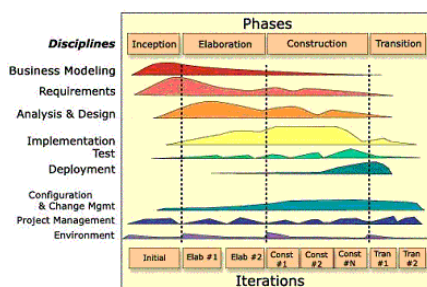
Unified Modeling Language (UML)

- Diagrama de Casos de Uso
- Diagrama de Clases
- Diagrama de Secuencia
- Diagrama de Actividades
- Diagrama de Colaboración
- Diagrama de Estados
- Diagrama de Componentes
- Diagrama de Distribución



OptaSmart IT Learning Services ©

Unified Software Development Process



OptaSmart IT Learning Services ©

Disciplinas

- Requerimientos
- Análisis
- Diseño
- Implementación
- Pruebas

OptaSmart IT Learning Services ©

Proceso basado en modelos de objetos

- Modelo de casos de uso
- Modelo de análisis
- Modelo de diseño
- Modelo de distribución (deployment)
- Modelo de pruebas

OptaSmart IT Learning Services ©



Módulo 3 – Desarrollando y probando un programa Java



OptaSmart IT Learning Services ©

Sintácticamente, una clase contiene 4 partes

- Paquete (package) al que pertenece
- Clases utilizadas (imports)
- Atributos (variables)
- Métodos (comportamiento, operaciones)

OptaSmart IT Learning Services ©

Programas en Java

- En Java, en todos los archivos se representa por lo menos a una clase.
- Una clase puede utilizar clases que fueron definidas en el mismo o en otros archivos.
- Un programa en Java es también una clase, pero con un método especial que sirve para indicarle a la Máquina Virtual que se trata de un programa.

OptaSmart IT Learning Services ©

HolaMundo.java

```
class HolaMundo {  
    static void main (String  
        args[]) {  
        System.out.println("Hola  
        Mundo!");  
    }  
}
```

OptaSmart IT Learning Services ©

javac & java

- `javac` es el comando incluido en el SDK (Software Development Kit) y sirve para compilar una clase/programa en Java.
- `java` es otro comando que se incluye en el SDK y en el JRE (Java Runtime Environment); sirve para ejecutar una programa de Java

OptaSmart IT Learning Services ©

Compilar una clase

- `javac HolaMundo.java`
- El comando anterior compila la clase y genera un archivo `HolaMundo.class`. Si la clase `HolaMundo` contiene un método `main` correcto, es un programa y puede ser ejecutado por el comando `java`.
- `java HolaMundo`

OptaSmart IT Learning Services ©

Argumentos desde la línea de comandos

```
class Argumentos {  
    public static void main (String []  
        args) {  
        System.out.println("1er  
        argumento: " + args[0]);  
    }  
}
```

- `javac Argumentos.java`
- `java Argumentos Hola`
 1er argumento: Hola

OptaSmart IT Learning Services ©

Classpath

- Al utilizar javac o java se pueden especificar archivos o directorios dentro de los cuales Java buscará clases requeridas por los archivos a compilar o ejecutar, respectivamente.

- `javac -classpath c:\java\ejemplos HolaMundo.java`
- `java -classpath c:\java\pruebas.jar Prueba`

OptaSmart IT Learning Services ©



Módulo 4 - Variables



OptaSmart IT Learning Services ©

Variables

- Utilizadas en los programas para guardar datos.
- Usos de las variables
 - ☐ Estados
 - ☐ Mensajes
 - ☐ Banderas
- Sintaxis de una variable
 - ☐ `<tipo> nombreVariable;`
 - ☐ `<tipo> nombreVariable = 234;`
 - ☐ `<Clase> nombreVariable = new <Clase>();`

OptaSmart IT Learning Services ©

Tipos de datos primitivos

- boolean
- byte
- char
- short
- int
- long
- float
- double

OptaSmart IT Learning Services ©

Modificadores en la declaración de una variable

- Acceso (`private`, `protected`, `public`, `<default>`)
- `final` (no cambiará)
- `static` (una sola copia para todas las instancias de la clase)
- `transient` (excluir de serialización)
- `volatile` (alerta para programas multi-hilo)

OptaSmart IT Learning Services ©

Modificar el valor de una variable

- Asignación
`int variable = 0;`
- Operaciones aritméticas
`int variable = 10;`
`variable = variable * 12;`
- Operadores de corrimiento de bits
`int variable = 123456;`
`variable = variable << 1;`

OptaSmart IT Learning Services ©

Transformaciones de Tipo (Casting)

■ Implícito (promoción de tipo)

```
int a = 100;  
long b = a;           //un int cabe en un long
```

■ Explícito

```
double a = 100.001;  
int b = (int) a;       //el double perderá info
```

OptaSmart IT Learning Services ©



Módulo 5 - Objetos



OptaSmart IT Learning Services ©

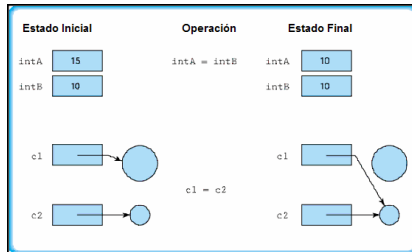
Objetos

- Unión de variables y métodos relacionados.
- Se pueden inicializar al momento de declararse:

```
Object a = new Object();
```

OptaSmart IT Learning Services ©

Memoria - Objetos vs. Datos primitivos



OptaSmart IT Learning Services ©

La clase `java.util.String`

■ Declaración

- `String s = new String();`
- `String s = new String("texto");`
- `String s = "texto";`

■ Inmutable

- `String s = "texto";`
- `s = s + " libre"; //crea un nuevo String`

OptaSmart IT Learning Services ©

API (Application Program Interface)

- Las clases que componen la versión estándar de Java se encuentran documentadas en:

<http://java.sun.com/j2se/1.4.2/docs/api/index.html>

OptaSmart IT Learning Services ©






Módulo 6 – Operadores y Controladores de Flujo




OptaSmart IT Learning Services ©



Operadores

- Operadores Relacionales
 - Comparan dos valores y determinan la relación entre ellos.
- Operadores Condicionales
 - Especifican **cómo** comparar dos valores.

OptaSmart IT Learning Services ©



Operadores Relacionales

Operador	Uso	Descripción
>	op1 > op2	Regresa true si op1 es mayor que op2
>=	op1 >= op2	Regresa true si op1 es mayor o igual a op2
<	op1 < op2	Regresa true si op1 es menor que op2
<=	op1 <= op2	Regresa true si op1 es menor o igual a op2
==	op1 == op2	Regresa true si op1 y op2 son iguales
!=	op1 != op2	Regresa true si op1 y op2 no son iguales

OptaSmart IT Learning Services ©

Operadores Condicionales

Operador	Uso	Descripción
&&	op1 && op2	Regresa true si op1 y op2 son true; condicionalmente evalúa op2
	op1 op2	Regresa true si op1 u op2 son true; condicionalmente evalúa op2
!	!op	Regresa true si op es false
&	op1 & op2	Regresa true si op1 y op2 son boolean y true; siempre evalúa op1 y op2; si ambos operandos son números, realiza operación AND bit a bit
	op1 op2	Regresa true si op1 y op2 son boolean y si op1 u op2 es true; siempre evalúa op1 y op2; si ambos operandos son números, realiza operación OR inclusivo bit a bit
^	op1 ^ op2	Regresa true si op1 y op2 son diferentes — esto es, si cualquiera de los dos operandos, pero no ambos, es true

OptaSmart IT Learning Services ©

Controladores de Flujo

- Un programa tiene enunciados (statements). Sin controladores de flujo (control flow statements) los enunciados se ejecutan en estricto orden.
- Los controladores de flujo pueden servir para
 - Ejecutar enunciados condicionalmente
 - Ejecutar repetidamente bloques de enunciados
 - Cambiar el orden normal y secuencial de ejecución

OptaSmart IT Learning Services ©

if, if-else

- ```
if (expresión) {
 enunciado(s)
}
```
- ```
if (expresión) {
    enunciado(s)
} else {
    enunciado(s) alternativo(s)
}
```

OptaSmart IT Learning Services ©

switch

```
switch (expresión) {  
    case cond1: bloque_1;  
    case cond2: blo  que_2;  
    ...  
    case condn: bloque_n;  
    default: bloque_default;  
}
```

OptaSmart IT Learning Services ©



Módulo 7- Constructores de ciclos



OptaSmart IT Learning Services ©

Constructores de ciclos

- Java tiene la habilidad de hacer ciclos en el código de ejecución hasta que se cumpla una condición de terminación.
- Durante cada iteración del ciclo, el mismo bloque de enunciados es ejecutado.
- Cuando la condición de terminación evalúa a `false`, el ciclo termina y la ejecución continúa con el enunciado que está después del mismo.

OptaSmart IT Learning Services ©

Ciclos en Java – 1

- `while`
ejecuta un bloque de enunciados **mientras** se cumpla una condición
- `do-while`
ejecuta un bloque de enunciados y lo vuelve a ejecutar mientras se cumpla una condición
- `for`
provee una manera compacta de iterar sobre un rango de valores

OptaSmart IT Learning Services ©

Ciclos en Java – 2

- Todos los ciclos en Java se pueden expresar como ciclos `while`.
- Algunos ciclos son escritos en forma más concisa en ciclos `do-while` o `for`.

OptaSmart IT Learning Services ©

`while`

```
while (expresión){  
    enunciado(s)  
}
```

1. El enunciado `while` evalúa la expresión, que debe regresar un valor `true`.
2. Si la expresión regresa `true` se ejecutan los enunciados del bloque.
3. El enunciado `while` continúa probando la expresión y ejecutando los enunciados hasta que la expresión regresa `false`.

OptaSmart IT Learning Services ©

while – Ejemplo 1

```
while (true) {  
    System.out.println("ciclo infinito");  
}
```

- El ciclo se ejecutará para siempre ya que la condición nunca será `false`.
- En ocasiones es útil tener un ciclo infinito; en este caso, solamente imprime "ciclo infinito" hasta que la JVM arroja una excepción.

OptaSmart IT Learning Services ©

while – Ejemplo 2

```
int contador = 0;  
while (contador < 5) {  
    System.out.println("iteración: " + contador);  
    contador++;  
}
```

- Típico ciclo `while`. Se crea una variable y se inicializa en 0. Mientras la variable sea menor a 5:
 - ☐ Se imprimirá iteración: <valor de la variable>
 - ☐ Se incrementará en uno el valor de la variable.
- Este ciclo terminará cuando la variable sea mayor o igual a 5 (no sea menor a 5).

OptaSmart IT Learning Services ©

while – Ejemplo 3

```
String colores[] = {"rojo", "azul", "verde"};  
int contador = 0;  
while (contador < colores.length) {  
    System.out.println("Elemento " + contador + " es " +  
        colores[contador]);  
    contador++;  
}
```

- Otro uso común del `while`: iterar sobre un arreglo y realizar operaciones sobre los elementos del arreglo.
- Mientras `contador` sea menor que el tamaño del arreglo:
 - ☐ Imprimirá Elemento <contador> es <color>
 - ☐ Se incrementará en uno el valor de `contador`
- Este ciclo terminará después de haber recorrido todo el arreglo.

OptaSmart IT Learning Services ©

do-while

```
do {  
    enunciado(s)  
} while (expresión);
```

1. El enunciado do-while **primero** ejecuta el bloque de enunciados.
 2. El bloque se sigue ejecutando mientras la expresión evalúe a true.
- En un enunciado do-while el bloque de enunciados se ejecuta por lo menos una vez.

OptaSmart IT Learning Services ©

while vs. do-while (iguales)

```
int contador = 0;  
while (contador < 5) {  
    System.out.println("iteracion: " + contador);  
    contador++;  
}
```

vs.

```
int contador = 0;  
do {  
    System.out.println("iteracion: " + contador);  
    contador++;  
} while(contador < 5);
```

OptaSmart IT Learning Services ©

while vs. do-while (diferentes)

```
int contador = 0;  
while (contador < 1) {  
    System.out.println("iteracion: " + contador);  
    contador++;  
}
```

vs.

```
int contador = 0;  
do {  
    System.out.println("iteracion: " + contador);  
    contador++;  
} while(contador < 1);
```

OptaSmart IT Learning Services ©

for

- Sintaxis compacta para iterar sobre un rango de valores, casi siempre un arreglo o una colección de objetos.
- Forma general:

```
for (inicialización; terminación; incremento) {  
    enunciado(s)  
}
```

OptaSmart IT Learning Services ©

```
for (inicialización; terminación;  
    incremento) {enunciado(s)}
```

- Es lo primero que se ejecuta y se ejecuta una sola vez.
- Permite la declaración e inicialización de variables que estarán disponibles dentro del ciclo `for`.
- Casi siempre es usado para declarar contadores `int` que servirán para saber en qué iteración se encuentra actualmente.

OptaSmart IT Learning Services ©

```
for (inicialización; terminación;  
    incremento) {enunciado(s)}
```

- Expresión condicional que debe evaluar a `false` eventualmente.
- Se evalúa al inicio de cada iteración.
 - ☐ Si evalúa a `true`, el ciclo continua y ejecuta los enunciados.
 - ☐ Si evalúa a `false`, el ciclo termina.
 - ☐ Si nunca evalúa a `false`, el ciclo continua indefinidamente.

OptaSmart IT Learning Services ©

```
for (inicialización; terminación;  
incremento) {enunciado(s)}
```

- Se ejecuta al final de cada iteración.
- Casi siempre incrementa los contadores que fueron declarados al inicio del ciclo.
- El enunciado de incremento debe acercar al ciclo a su terminación.

OptaSmart IT Learning Services ©

```
for (inicialización; terminación;  
incremento) {enunciado (s)}
```

- Se ejecutan en cada iteración, después de revisar la condición de terminación.
- Por lo general, en estos enunciados se accede al siguiente elemento del arreglo o colección y se procesa dicho elemento.

OptaSmart IT Learning Services ©





Módulo 8 – Métodos



OptaSmart IT Learning Services ©

Métodos

- Una clase tiene atributos (variables) y comportamientos (métodos).
- Sin métodos un objeto no puede hacer nada. Sólo esperar pasivamente a que otros objetos manipulen sus atributos.
- Un buen diseño orientado a objetos exige que los datos estén lo más escondido posible.

OptaSmart IT Learning Services ©

Encapsulamiento

- Si los datos de una clase están escondidos, otras clases no los pueden manipular directamente.
- Si otras clases quieren actualizarlos, deben invocar algún método que sí tenga acceso a ellos.
- Esconder datos es conocido como **encapsulamiento** y es una práctica recomendada de la Orientación a Objetos.

OptaSmart IT Learning Services ©

Ventajas de usar métodos

- Permite pedirle a un objeto que realice una acción.
- Define exactamente lo que un objeto puede hacer.
- Permite extraer e ingresar valores.
- Hace que los programas sean más fáciles de leer y mantener.
- Hace que el desarrollo y el mantenimiento sean más rápidos.
- Permite reutilizar software.

OptaSmart IT Learning Services ©

Método Trabajador y Método Invocador – 1

- Hasta ahora, los programas han sido así:

```
public class TodoEnMain {  
    public static void main (String args[]) {  
        int int1 = 42;  
        int int2 = 24;  
        System.out.println(int1 + int2);  
    }  
}
```

- Esta manera es simple pero
 - Impráctico para aplicaciones grandes
 - No usa Orientación a Objetos
 - Tiene otras desventajas

OptaSmart IT Learning Services ©

Método Trabajador y Método Invocador – 2

- Este ejemplo divide en dos el programa anterior:

```
public class Uno {  
    public static void main (String args[]) {  
        Dos dos = new Dos();  
        dos.metodoTrabajador();  
    }  
}  
  
public class Dos {  
    public void metodoTrabajador() {  
        int int1 = 42;  
        int int2 = 24;  
        System.out.println(int1 + int2);  
    }  
}
```

OptaSmart IT Learning Services ©

Método Trabajador y Método Invocador – 3

- Un método invocador le pide a un objeto que haga algo, usando uno de los métodos del objeto.
 - Además de invocar al método trabajador, un método invocador realiza tareas normales.
- El objeto contiene métodos trabajadores:
 - Pueden recibir información
 - Pueden regresar información

OptaSmart IT Learning Services ©

Declarando métodos

- Los métodos trabajadores y los métodos invocadores tienen la misma estructura sintáctica.

```
modificadores [tipo_de_retorno] identificador_del_metodo
([argumentos]) {
    cuerpo_del_metodo
}
```

- La combinación del nombre de un método con sus parámetros (número, orden y tipo) es conocida como firma (signature).

OptaSmart IT Learning Services ©

Invocando a un método

- Típicamente se llama desde otro método.
- El método invocador hace una pausa y el método trabajador entra en su lugar.
- El método invocador continúa donde se quedó.
- El método invocador y el trabajador pueden estar en la misma clase o en diferentes clases.

OptaSmart IT Learning Services ©

Pasando argumentos

- Incluirlos en los paréntesis en el método invocador.
- Se pueden pasar literales o variables.
- Enlistarlas en el mismo orden que en la declaración del método.

OptaSmart IT Learning Services ©

Recibiendo valores de retorno

- El valor de retorno llega al mismo lugar del código donde se invocó al método.
`math.add(2, 4)`
- La expresión en la cual se mandó llamar el método es igual al valor de retorno.
`int suma = math.add(2, 4);`
- Se puede combinar la llamada y el uso del valor de retorno en una sola línea usando el valor de retorno.
`if (add(2, 4) < dias)`

OptaSmart IT Learning Services ©

Métodos de Objeto vs. Métodos Estáticos

- Los métodos de objeto necesitan que se haya creado un objeto.
- Los métodos estáticos
 - Se declaran igual que los métodos de objeto, pero se agrega el modificador `static`.
 - Se pueden ejecutar sin un objeto.
 - Están relacionados con una clase, pero no operan sobre las variables asociadas con un objeto en particular.

OptaSmart IT Learning Services ©

Invocación de un método estático

- Sintaxis:
`Clase.metodo();`
- Ejemplo:
`Calendario.regresaDia();`

OptaSmart IT Learning Services ©

Cuándo declarar un método estático

- Define un método como estático cuando:
 - No es importante el objeto individual sobre el cual se realiza la operación
 - Es importante que las acciones del método se realicen antes de crear objetos.
 - Las responsabilidades que cumple el método no pertenecen lógicamente a un objeto.

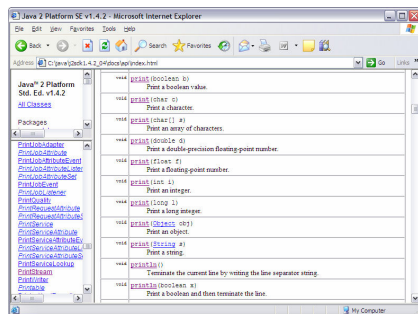
OptaSmart IT Learning Services ©

Sobrecarga de métodos (overloading)

- Varios métodos realizan la misma tarea pero reciben distintos parámetros.
- Pueden tener el mismo nombre mientras su firma sea distinta.

OptaSmart IT Learning Services ©

Ejemplos de métodos sobrecargados



Repaso

- Describir por qué usar múltiples métodos ayuda a la Orientación a Objetos.
- Escribir una declaración de un método distinto a main.
- Escribir un método que llama a otro método.
- Pasar parámetros a un método y recibir un valor de retorno.
- Definir métodos estáticos y de objeto.
- Explicar que es un método sobrecargado y por qué son útiles.
- Escribir código usando métodos sobrecargados.

OptaSmart IT Learning Services ©



Módulo 9 – Implementing Encapsulation and Constructors

OptaSmart IT Learning Services ©

Encapsulamiento – 1

- Los objetos tienen como *miembros* a sus operaciones y atributos.
- Los miembros pueden tener varios niveles de visibilidad.
- Todas o casi todas las variables deben mantenerse *privadas*.
- Las variables deben ser modificadas por métodos de su misma clase.

OptaSmart IT Learning Services ©

Encapsulamiento – 2

- En la Orientación a Objetos, el encapsulamiento hace posible que variables y métodos estén protegidos contra acceso o ejecución desde otros objetos.
- En Java, esto se logra con el uso de modificadores de acceso: `public`, `private`, `protected` y un modificador default de acceso.

OptaSmart IT Learning Services ©

Encapsulamiento – 3

- Estos modificadores de acceso se pueden usar en la declaración de cualquier método o variable, por ejemplo hemos visto la declaración del método `main`:

```
□ public static void main(String args[])
```
- Los modificadores de acceso van al inicio de la declaración.
- Si no hay modificador de acceso entonces el modificador default aplica.

OptaSmart IT Learning Services ©

Modificadores de acceso – 1

	Misma clase	Sub-clase	Mismo paquete	Universo
<code>public</code>	Sí	Sí	Sí	Sí
<code>private</code>	Sí	No	No	No
<code>protected</code>	Sí	Sí	No	No
default	Sí	No	Sí	No

OptaSmart IT Learning Services ©

Modificadores de acceso – 2

- `public`
 - puede ser accesado por
 - cualquier objeto.
- `private`
 - puede ser accesado por
 - la misma clase
 - no puede ser accesado por
 - diferentes clases

OptaSmart IT Learning Services ©

Modificadores de acceso – 3

- `protected`
 - puede ser accesado por
 - la misma clase
 - una sub-clase
 - el mismo paquete
 - no puede ser accesado por
 - diferentes paquetes si no son sub-clases
- `default`
 - puede ser accesado por
 - la misma clase
 - el mismo paquete
 - no puede ser accesado por
 - diferentes paquetes

OptaSmart IT Learning Services ©

Ventajas del encapsulamiento

- Cada objeto protege sus datos.
- La implementación es controlada por la persona que programa la clase.
- Los objetos son manejados a través de sus operaciones (y no directamente sobre sus datos).
- La implementación puede cambiar sin cambiar la interfaz.

OptaSmart IT Learning Services ©

Métodos get y métodos set

- Cuando las variables son privadas, se accede a ellas a través de métodos que sean miembros de la misma clase.
- Para obtener un valor, se usa un método `get`.
- Para asignar un valor, se usa un método `set`.

OptaSmart IT Learning Services ©

Métodos get y set – Implementación

```
public class ArtículoEncapsulado {  
    private int idArticulo;  
    public void setIdArticulo(int nuevoValor)  
    {  
        idArticulo = nuevoValor;  
    }  
    public int getIdArticulo() {  
        return idArticulo;  
    }  
}
```

OptaSmart IT Learning Services ©

Alcance (scope) de una variable

- No todas las variables se encuentran disponibles durante todo un programa.
- Scope de una variable → dónde puede ser usada dicha variable.

OptaSmart IT Learning Services ©

Scope de una variable

```
public class NombreEdad {  
    // empieza scope de int edad  
    private int edad = 32;  
    public void print () {  
        // empieza scope de String nombre  
        String nombre = "Roman Martinez";  
        System.out.println("Mi nombre es " + nombre +  
            " y tengo " + edad + " años.");  
    } // termina scope de String nombre  
  
    public String indicarNombre() {  
        return nombre; // esto causara un error  
    }  
} // termina scope de int edad
```

OptaSmart IT Learning Services ©

Constructores

- Este bloque de código se ejecuta al crear un nuevo objeto usando `new`.
- Un constructor tiene el mismo nombre que su clase y no tiene valor de retorno.
- Los constructores permiten especificar valores para atributos al momento de crear un objeto.

OptaSmart IT Learning Services ©

Constructores - Implementación

```
public class Sombrero {  
    private String tipo;  
    public Sombrero(String tipoSombrero) {  
        tipo = tipoSombrero;  
    }  
}  
  
class Construye {  
    Sombrero sombrero = new  
        Sombrero("De Copa");  
}
```

OptaSmart IT Learning Services ©

Constructores disponibles

- Los constructores creados explícitamente
 -
- El constructor default.

OptaSmart IT Learning Services ©

El constructor default – 1

- Todos los programas vistos hasta ahora han tenido un constructor default.
- Si una clase **no** tiene constructor, Java agrega un constructor default.
- Cuando se usa `new` para instanciar un objeto de una clase sin constructor, `new` manda llamar al constructor default de la clase.
 - `Camiseta camiseta = new Camiseta();`
- El constructor default será insertado por el compilador.

OptaSmart IT Learning Services ©

El constructor default – 2

```
public class Camiseta {  
    String talla;  
  
    // constructor default  
    // (no aparece en el código)  
    public Camiseta() {  
    }  
}
```

OptaSmart IT Learning Services ©

Capacidades de los constructores

- Muy flexibles ya que pueden ser sobrecargados al igual que los métodos.
- Pueden asignar valores, pasar parámetros.

OptaSmart IT Learning Services ©

Declaración y uso de constructores

- Declarando

```
[modificadores] class ClaseUno {
    ClaseUno([argumentos]) {
        [enunciados_de_inicializacion];
    }
}
```
- Usando

```
[modificadores] class ClaseDos {
    ClaseUno referencia = new ClaseUno([valores]);
}
```

OptaSmart IT Learning Services ©

Sobrecarga de constructores

```
public class Camisa {
    String tipo;
    char talla;
    //Constructor sin argumentos
    Camisa() {
        tipo = "Camisa Oxford";
    }

    //Constructor con un argumento
    Camisa(String tipoCamiseta) {
        tipo = tipoCamiseta;
    }
}
```

OptaSmart IT Learning Services ©

Ejercicio

- Crear una clase Cliente con un atributo nombre. Si no se especifica un nombre al crear una instancia de Cliente, el nombre debe ser "Nuevo Cliente".

OptaSmart IT Learning Services ©



Module 10 - Creating and Using Arrays

OptaSmart IT Learning Services ©

Arreglos

- Los arreglos son objetos que pueden almacenar muchas variables del mismo tipo.
- Los arreglos pueden guardar primitivos o referencias a objetos, pero el arreglo en sí siempre es un objeto aunque guarde primitivos.

OptaSmart IT Learning Services ©

Trabajando con arreglos

- Crear una variable que sea referencia a un arreglo (*declaración*)
- Crear un objeto arreglo (*construcción*)
- Popular un arreglo con elementos (*inicialización*)

OptaSmart IT Learning Services ©

Declaración de arreglos

- Los arreglos son declarados indicando el tipo de elemento que guardarán (pueden ser objetos o primitivos), seguido de corchetes a la izquierda o derecha del identificador.
- Arreglos de primitivos
 - `int[] clave;`
 - `int clave[];`
- Arreglos de objetos
 - `String[] strings;`
 - `String strings[];`

OptaSmart IT Learning Services ©

Declaración de arreglos multidimensionales

- Declaración de arreglos multidimensionales (arreglos de arreglos)
 - `String[][][] nombres;`
 - `String[] gerentes[];`

OptaSmart IT Learning Services ©

¿Cómo se construye?

- Construcción → crear el objeto en memoria → hacer `new` del tipo del arreglo.
- Para crear el objeto arreglo, Java debe saber cuánto espacio requiere en memoria, en la construcción se debe especificar el tamaño del arreglo.
- El tamaño del arreglo es la cantidad de elementos que guardará.

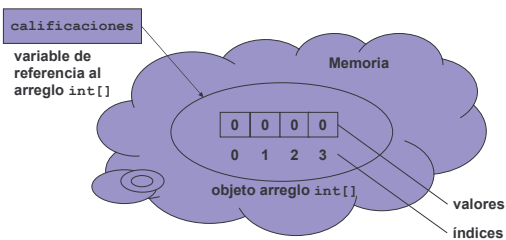
OptaSmart IT Learning Services ©

Ejemplo

- La manera más sencilla de construir un arreglo es usar la palabra reservada `new` seguida del tipo del arreglo, con unos corchetes especificando cuántos elementos de ese tipo guardará el arreglo.
- Ejemplo:
 - `int[] calificaciones; //declaración`
 - `calificaciones = new int[4]; //construcción`
 - `int[] calificaciones = new int[4];`

OptaSmart IT Learning Services ©

¿Cómo se ve en memoria?



OptaSmart IT Learning Services ©

Construcción de arreglos multidimensionales

- Los arreglos multidimensionales son simplemente arreglos de arreglos.
- Un arreglo bidimensional de tipo `int` es en realidad un objeto de tipo `int[]`, donde cada elemento en el arreglo guarda una referencia a otro arreglo `int`. La segunda dimensión es la que guarda los valores primitivos `int`.
- `int[][] codigos = new int[3][];`

OptaSmart IT Learning Services ©

Inicialización

- Inicialización → poner elementos en el arreglo.
- Elementos
 - Primitivos (2, false, 'b', etc.)
 - Referencias a objetos
- Si un arreglo guarda referencias a objetos, esas referencias pueden apuntar a `null` y no a objetos reales.

OptaSmart IT Learning Services ©

Acceso a los elementos de un arreglo

- Los elementos de un arreglo pueden ser accedidos con un número índice.
- El número índice empieza siempre en cero → para un arreglo de 10 elementos los índices van del 0 al 9.
- Si se crea un arreglo de 3 Animales
 - `Animal[] mascotas = new Animal[3];`

OptaSmart IT Learning Services ©

Inicializar elementos de un arreglo

- Ahora tenemos un objeto arreglo en memoria con 3 referencias `null` de tipo `Animal`, pero aun no hay objetos `Animal`.
- El siguiente paso es crear objetos `Animal` y asignarlos a las posiciones del arreglo `mascotas`:

```
□ mascotas[0] = new Animal();  
  mascotas[1] = new Animal();  
  mascotas[2] = new Animal();
```

OptaSmart IT Learning Services ©

Inicializar elementos con un ciclo

- Los objetos arreglo tienen una sola variable pública llamada `length` que contiene el número de elementos en el arreglo.
- La variable `length` se puede usar para recorrer el arreglo en un ciclo e inicializarlo:

```
□ Perro[] perros = new Perro[6];  
  for (int i = 0; i < perros.length; i++)  
  {  
    perros[i] = new Perro();  
  }
```

OptaSmart IT Learning Services ©

El arreglo `String[] args`

- Cuando se ejecuta un programa Java, éste puede recibir parámetros desde la línea de comandos.
- Los parámetros son almacenados en el único argumento del método `main`:

```
□ public static void main(String[]  
  args) {}
```

OptaSmart IT Learning Services ©

Enviando parámetros a un programa

- ```
public void static main(String[] args) {
 System.out.println("Hola " + args[0] +
 args[1]);
}
```
- ```
java Programa Homero Simpson  
Hola Homer Simpson
```

OptaSmart IT Learning Services ©



Module 11 - Implementing Inheritance

OptaSmart IT Learning Services ©

Herencia – 1

- Los objetos son definidos en términos de clases.
- Si se conoce la clase de un objeto, se obtiene mucha información sobre el objeto.
- Si no sabemos qué es un *penny-farthing*, pero nos dicen que es una bicicleta, ya sabemos que tiene dos llantas, manubrio y pedales.
- La Orientación a Objetos lleva esto más lejos: una clase se puede definir en términos de *otras* clases.

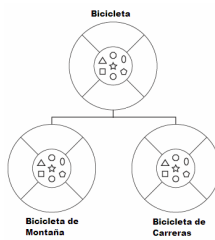
OptaSmart IT Learning Services ©

Herencia – 2

- Por ejemplo, las bicicletas de montaña y las de carreras son tipos de bicicletas.
- En la terminología de OO, las bicicletas de montaña y las de carreras son **subclases** de la clase bicicleta.
- La clase bicicleta es la **superclase** de las bicicletas de montaña y las de carreras.

OptaSmart IT Learning Services ©

Ejemplo: Bicicleta



OptaSmart IT Learning Services ©

¿Qué es la herencia?

- Un objeto tiene:
 - Datos → atributos
 - Métodos → comportamiento
- Un objeto que hereda ya tiene los datos y métodos de sus ancestros.
- ES-UN (is-a)
 - La herencia implementa la relación ES-UN. Es similar a una taxonomía: un perro ES-UN mamífero.

OptaSmart IT Learning Services ©

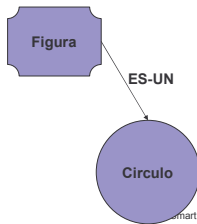
¿Por qué usar subclases?

- Reutilizar código.
- Si las subclases no redefinen el método de una superclase, se usa el de la superclase.
- Categorías de objetos con una relación conceptual pueden ser asociadas usando su jerarquía.
- Un programador puede especificar superclases (clases abstractas) que definen comportamientos comunes, pero la implementación se deja a otro programador.

OptaSmart IT Learning Services ©

Heredando

- Para heredar, usar la palabra `extends`.
- ```
public class Circulo extends Figura {
```



OptaSmart IT Learning Services ©

---

---

---

---

---

---

---

## Acceso

- La subclase puede acceder a los miembros `public` y `protected` de la superclase.
- La subclase **no** puede acceder a los miembros `private` de la superclase.
- Los métodos de la superclase pueden ser sobrescritos cuando son inapropiados para una subclase.

OptaSmart IT Learning Services ©

---

---

---

---

---

---

---

## Conversiones dentro de la jerarquía de la herencia

- Default es en un solo sentido:
  - Los objetos de la superclase no pueden referenciar elementos que sólo se encuentran en una subclase.
- Un objeto de una subclase puede ser tratado como objeto de su correspondiente superclase. Lo opuesto no es verdadero (sin un cast explícito).

```
class Circulo extends Figura {}
Figura f = new Circulo();
```
- Un cast explícito puede hacer que un objeto sea tratado como objeto de una subclase.

```
Figura f = new Circulo();
Circulo c = (Circulo) f;
```

OptaSmart IT Learning Services ©

---

---

---

---

---

---

---

---

## Polimorfismo

- Habilidad de aparecer en más de una forma.
- En OO: La habilidad de un lenguaje de programación para procesar objetos en forma diferente dependiendo de su tipo o clase.

OptaSmart IT Learning Services ©

---

---

---

---

---

---

---

---

## Clases abstractas y Clases concretas

- Clases abstractas
  - **Definen** métodos comunes
  - No pueden ser instanciadas
- Clases concretas (default en Java)
  - Deben **implementar** métodos comunes
  - Pueden ser instanciadas

OptaSmart IT Learning Services ©

---

---

---

---

---

---

---

---

## Clases abstractas y polimorfismo

- Clases genéricas
  - Jerarquías completas pueden ser procesados sin hacer referencia a clases concretas
- Polimorfismo
  - Muchas *formas* pueden ser manipuladas por métodos con el mismo nombre
- Extensibilidad
  - Nuevos objetos, métodos y funcionalidad pueden agregarse fácilmente sin impactar grandes secciones de código

OptaSmart IT Learning Services ©

---

---

---

---

---

---

---

---

## Interfaces

- Interfaces se usan en lugar de clases abstractas
  - cuando la jerarquía no contiene un método que se desea agregar
  - se desea proveer herencia múltiple
- La definición de la interfaz contiene un conjunto de métodos públicos y abstractos
- Un objeto que implementa una interfaz debe definir cada método en la interfaz

```
class Murcielago implements Volador {}
class Aguila implements Volador {}
```

OptaSmart IT Learning Services ©

---

---

---

---

---

---

---

---

## Interfaces vs. Clases Abstractas

| Características        | Interfaces                                | Clases abstractas                                     |
|------------------------|-------------------------------------------|-------------------------------------------------------|
| Herencia múltiple      | Clase puede implementar varias interfaces | Clase sólo puede extender a una clase abstracta       |
| Implementación default | Interfaz no puede proveer ningún código   | Clase abstracta puede proveer código que será default |
| Sobrecarga             | Clase debe sobrecargar todos los métodos  | Opcional para subclases                               |

OptaSmart IT Learning Services ©

---

---

---

---

---

---

---

---