Capítulo 2

Clases – Interfaces – Herencia - Polimorfismo

Java World

¿Querés ser un SCJP?

Revista Digital - Hecha por estudiantes para estudiantes

Agradecimientos

Quisiera darle las gracias a Matias Emiliano Alvarez Durán (quien ahora será coautor de JavaWorld) por su aporte, y por darme ese empujón que tanta falta me hacía.
Sin tu ayuda JavaWorld no hubiera visto nunca la luz.

Gracias

Paradigmas de programación

Para aquellos que son iniciados en el tema de la programación, es bueno tener en cuenta un poco la evolución en temas de cómo se programa.
Desde siempre la programación fue una serie de algoritmos que resuelven un problema, lo que cambió, fue la manera de agrupar y asociar estos algoritmos.

Paradigma estructural

Este paradigma busca como objetivo dividir un problema complejo (sistema), en varios problemas menos complejos (funciones). Este es un paradigma que ha ido quedando en desuso con el correr de los tiempos, aunque aquellos que estén dando sus primeros pasos en alguna institución, puede que lo vean en lenguajes como Pascal o C++.

Paradigma orientado a objetos

En este paradigma, cada elemento del sistema se divide en objetos. Estos son representaciones de entidades que encontramos en la vida real. Por ejemplo, si estuviéramos diseñando un sistema para un aeropuerto, sería más que factible encontrarse con las clases Aeroplano, Pasajero, Capitan.
¿Pero, y la diferencia?
En este paradigma, cada entidad es responsable de realizar las acciones que le conciernen, y solo las que le conciernen.
Supongamos que estamos diseñando un juego, y tenemos un avión, según los paradigmas hubiéramos hecho lo siguiente:

Estrucura procedural en lenguaje C

```
typedef
struct Avion {
  int cantidadPasajerosSoportada;
  float peso;
  //... otras propiedades de un avion
} Avion_1;

public void volar(avion av) {
  //... codigo
}

public void aterrizar(avion av) {
  //...codigo
}

public main() {
  volar(Avion_1)
  aterrizar(Avion_1)
}
```

Estructura orientada a objetos en Java

```
class Avion {
  int cantidadPasajerosSoportada;
  float peso;
  //... otras propiedades de un avión

  public void volar() {
    //... Codigo
  }

  public void aterrizar() {
    //... Codigo
  }
}

static public void main(String[] args) {
  Avion avion_01 = new Avion();
  avion_01.volar();
  avion_01.aterrizar();
}
```

La diferencia más grande la podemos apreciar al momento de llamar a las acciones de volar y aterrizar. En el modelo estructural se llama a una función a la cal se le pasa un avión como argumento, pero en el modelo orientado a objetos, se le dice al propio avión que vuele o aterrice.

Definición de una clase

```
public class Persona {
  //Atributos
  String nombre;
  String apellido;
  int edad;
  float peso;

  //Constructores
  public Persona(String nombre, String apellido) {
    //Código ...
  }

  //Métodos
  public void correr(){
    //Código...
  }

  public void caminar(){
    //Código...
  }

  public void hablar(){
    //Código...
  }
}
```

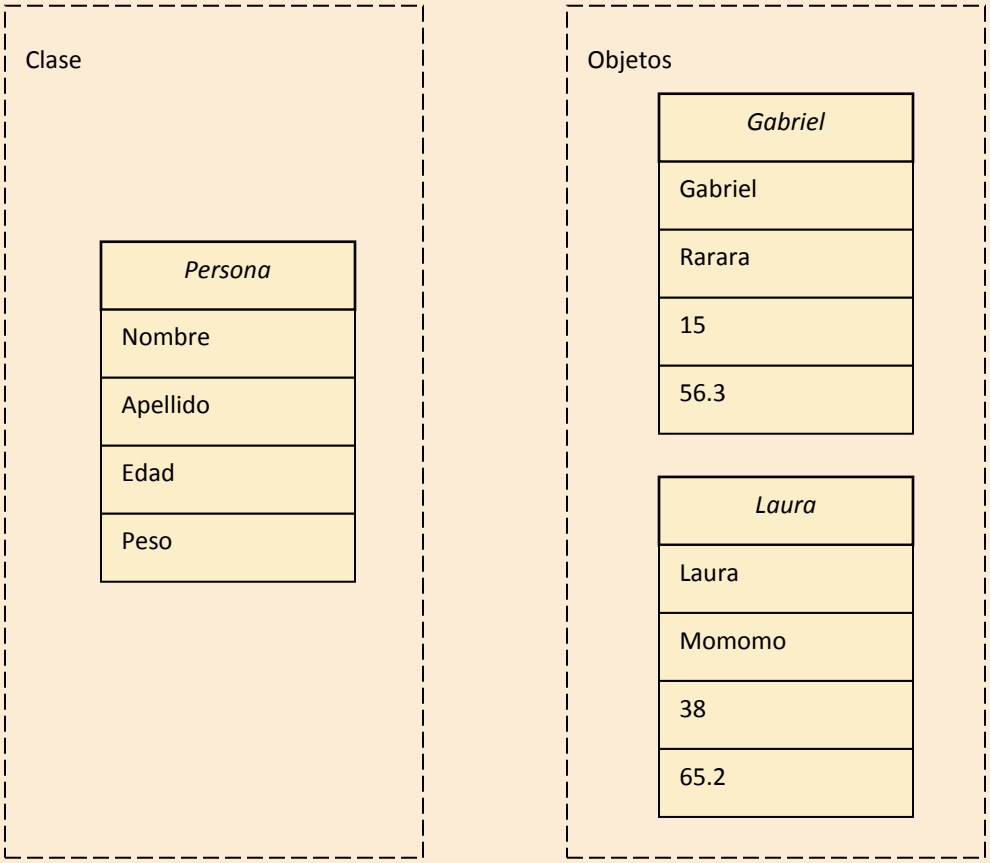
Básicamente, una clase contiene estos elementos:

- Atributos Definen el estado de un objeto
- Métodos Definen el comportamiento (pueden ser de clase o de objeto)
- Constructores Permiten crear objetos

Existen otros elementos que pertenecen a la clase, y no al objeto. Estos contienen el identificador static.
Pero no te preocupes, ya los veremos.

Diferencia entre clase y objeto

Una clase representa una especie de molde para cada objeto. Cuando nosotros creamos un objeto, lo hacemos utilizando alguna clase en particular, esto quiere decir que el objeto creado, puede realizar el comportamiento de la clase, y contiene todos sus atributos.
Vamos a verlo con un ejemplo más gráfico. Utilicemos la clase Persona que definimos anteriormente.
Basandonos en ella, podemos apreciar que cada Persona (clase), tendrá ciertas características (atributos). Por ejemplo, supongamos una persona llamada Gabriel y otra Laura.



Al definir un objeto de tipo Persona estamos diciendole a la JVM que cree una estructura con tales propiedades y tal comportamiento. En el ejemplo anterior, Laura y Gabriel también tienen la capacidad de correr(), caminar(), y hablar().

Para todos aquellos que crean necesario y/o quieran profundizar en la programación orientada a objetos les quisiera recomendar la siguiente bibliografía:

- **Head First Java** Es muy amigable, y contiene una manera de explicar todo con texto e imágenes (permite mayor absorción de los conocimientos). La desventaja es que solo se encuentra en inglés.
- **Thinking in Java** Es muy extenso y detallado, y hay una versión en castellano. La desventaja es que en algunos momentos, puede llegar a ser un tanto abrumador.
- Busquen en internet "**programacion + objetos + java**". Este es un tema más que abarcado en la web, y encontrarán mucha información al respecto.

Más conceptos de la POO

Al adentrarnos en este nuevo paradigma, no solamente cambia la manera de codificar, sino que se introducen varios conceptos que forman las bases de este, tales como encapsulamiento, polimorfismo y herencia.

Encapsulamiento

Algunos de los beneficios aportados por el paradigma de orientación a objetos (OO) son: flexibilidad y mantenimiento. Pero para lograr esto, es necesario escribir las clases de una manera en particular.
La clave para lograr esto es encapsular la lógica y el estado de la clase para que no pueda ser accedido de forma directa, y dejar de manera publica un set de métodos para poder interactuar con la clase.
Las claves son:

- Siempre mantener las variables de instancia protegidas (con un modificador de acceso, generalmente **`public`**)
- Crea métodos públicos (public) para acceder a las variables de instancia
- Para los anteriores, utiliza la convención de nombres de JavaBeans

Herencia

Cuando utilizamos la herencia, a veces es necesario saber si un objeto pertenece a una clase en concreto. Para realizar esto utilizamos el operador instanceof, este devuelve un boolean, siendo true en caso de que pertenezca a dicha clase, falso en caso contrario.

objeto **`instanceof`** clase

Es posible crear relaciones de herencia en Java extendiendo una clase. Las dos características más comunes para utilizar la herencia son:

- Para promover la reutilización del código
- Para el uso del polimorfismo

El polimorfismo nos permite tratar una subclase como si fuera una clase del tipo de su superclase. La contrapartida de esto es que solo se podrán llamar a los atributos que se encuentren disponibles para el tipo de clase declarada.

Is-A

Para el examen es necesario saber:

- Is-A indica que la clase X es del tipo Y. Esto se logra a través de **`extends`** o **`implements`**.
- Al termino Is-A también se lo puede llamar: derived, subclass, inherits, o subtype.
- A la superclase también se la conoce como “supeclase”

Has-A

Indica que una clase A tiene una variable de instancia que contiene una referencia a B.

Polimorfismo

Un objeto es polimórfico cuando cumple con al menos una condicios Is-A. Esto es cierto para todas las clases que no sean de tipo Object, ya que todas heredan de esta.
La única manera de acceder a un objeto es a través de una variable de referencia. Algunas pautas para recordar de estas:

- Una variable de referencia puede ser de un solo tipo, y una vez que fue declarada, ese tipo jamás puede modificarse.
- Una referencia es una variable, de manera que puede ser reasignada a otros objetos (a menos que se declare como **`final`**).

- Un tipo de variable de referencia determina los métodos que pueden ser invocados sobre el objeto que la variable referencia.
- Una variable de referencia puede tomar como referencia cualquier objeto del mismo tipo, o un subtipo del mismo.
- Una variable de referencia puede tener como tipo el de una interface. Si es declarada de esta manera, podrá contener cualquier clase que implemente la interfaz.

El concepto es que solo los métodos de instancia son invocados dinámicamente. Esto es logrado por la JVM, la cual siempre tiene conocimiento de que tipo de objeto es el que se está tratando, así como su árbol de herencia.

Sobre escritura

Cada vez que una clase hereda un método de su superclase, existe la posibilidad de sobrescribir el método (a menos que este haya sido declarado con el modificador de no acceso **`final`**). El beneficio de la sobre escritura es la habilidad de definir comportamiento particular para una clase específica.
Características necesarias para hacer uso de la sobre escritura:

- La lista de argumentos debe ser exactamente igual que el método a sobrecargar.
- El valor de retorno debe ser del mismo tipo, o un subtipo.
- El modificador de acceso no puede ser más restrictivo que el del método a sobre escribir.
- El modificador de acceso puede ser menos restrictivo que el método a sobre escribir.
- Los métodos de instancia solo pueden ser sobre escritos si es que son heredados por la subclase.
- El método sobre escrito puede lanzar cualquier excepción sin verificación en tiempo de ejecución, sin importar de las excepciones que hayan sido declaradas en el método de la superclase.
- El método no deberá lanzar excepciones verificadas que son nuevas o amplían a las ya definidas por el método a sobre escribir.
- El método sobre escrito puede lanzar una cantidad menor de excepciones que las definidas en el método a sobre escribir.
- No puedes sobre escribir un método que haya sido declarado como **`final`** (creo que es la 20ª vez que repetimos esto, pero, nunca es suficiente).
- No es posible sobre escribir un método que haya sido declarado como **`static`**.
- Si un método no puede ser heredado, no puede ser sobre escrito.

Invocando la versión de la superclase de un método sobre escrito

También es posible valerse del código definido en la superclase. La manera de invocarlo es con la palabra **`super`** (esta hace referencia a la superclase).

En el caso de que un método sobre escrito no declare alguna excepción del método a sobre escribir, si se utiliza como referencia el tipo de la superclase, se espera que pueda llegar a generarse la excepción.

```
public class Generic {
  public void doSomething() throws Exception {
    // . . . some code here
  }
}

public class Specific extends Generic {
  public void doSomething() {
    // . . . some code here
  }
}

public class Program {
  static public void main ( String[] args ) {
    Generic obj1 = new Specific();
    Specific obj2 = new Specific();

    obj2.doSomething();
    obj1.doSomething(); //Se genera un error de compilación, dado que no se declaro como recibir la excepción Exception

    //El código podría solucionarse con un try-catch
    try {
      obj1.doSomething();
    } catch ( Exception ex ) {
      //. . . some code here
    }
  }
}
```

Sobrecarga

La sobrecarga de métodos permite utilizar el mismo nombre de un método, el cual toma una cantidad y/o tipo de parámetros diferentes.
Las reglas para la sobrecarga son:

- Deben de modificarse la lista de parámetros.
- Pueden modificar el tipo de valor de retorno.
- Pueden cambiar el modificador de acceso.
- Pueden declarar excepciones nuevas o más abarcativas.
- Pueden ser sobrecargados en la misma clase, o en una subclase.

Básicamente, un método sobrecargado no es más que un método completamente diferente, pero que utiliza un nombre que ya existe.

Invocación de métodos sobrecargados

Para invocar un método sobrecargado, es necesario especificar los parámetros del tipo esperado según el método que queramos invocar.

```
public class Something {
  public void doSomething(int val) {
    //. . . some code here
  }

  public void doSomething(float val) {
    //. . . some code here
  }

  static public void main (String[] args){
    Something s1 = new Something();

    int valor1 = 1;
    float valor2 = 2.3;

    s1.doSomething(valor1); //Se invoca al método doSomething(int val)
    s1.doSomething(valor2); //Se invoca al método doSomething(float val)
    s1.doSomething(new Object); //Error de compilación. No existe ningún método que reciba como parámetro un tipo Object
  }
}
```

En el caso de la invocación con objetos, se vuelve un poco más complicado.

```
public class Generic {}
public class Specific extends Generic {}

public class Program {

  public void doSomething(Generic obj) {
    //. . . some code here
  }

  public void doSomething(Specific obj) {
    //. . . some code here
  }

  static public void main (String[] args) {
    Generic obj1 = new Generic();
    Specific obj2 = new Specific();
    Generic obj3 = new Specific();

    Program p = new Program();

    p.doSomething(obj1);  //Se invoca al método doSomething(Generic obj)
    p.doSomething(obj2);  //Se invoca al método doSomething(Specific obj)
    p.doSomething(obj3);  //Se invoca al método doSomething(Generic obj)
  }
}
```

En el último caso, por más que **`obj3`** sea de tipo **`Specific`**, la referencia contiene un tipo **`Generic`**, y es esta la que determina a que método se invoca.

Polimorfismo en sobre escritura y sobrecarga

El polimorfismo no determina que método invocar cuando se trata de sobrecarga, pero si cuando se trata de sobre escritura.
Tabla de diferenciación entre sobrecarga y sobre escritura:

	Sobrecarga de método	Sobre escritura de método
Parámetros	Deben cambiar.	No deben cambiar.
Valor de retorno	Puede cambiar el tipo.	No puede cambiar, salvo por alguna covariante (subclass).
Excepciones	Pueden cambiar.	Pueden reducirse o eliminarse. No se pueden lanzar nuevas excepciones u otras más abarcativas.
Modificadores de acceso	Pueden cambiar	No pueden ser más restrictivos (pueden ser menos restrictivos).
Invocación	El tipo de la referencia determina que método invocar.	El tipo del objeto determina que método invocar. Sucede en tiempo de ejecución.

Casteo de variables de referencia

En ciertas ocasiones, es necesario utilizar un método de la clase específica (subclase), pero la referencia es de la superclase. En estos casos se utiliza un casteo denominado **downcast**. El problema con este tipo de casteos, es que no generar errores en tiempo de compilación, pero pueden generarlos en tiempo de ejecución.

```
public class Generic () {
  public void doSomething() {
    // . . . some code here
  }
}

public class Specific (){
  public void doSomethingMoreSpecific() {
    // . . . some code here
  }
}

public class Program {
  static public void main (String[] args) {
    Generic g1 = new Specific();
    Generic g2 = new Generic();

    g1.doSomething();
    g1.doSomethingMoreSpecific(); //Genera un error de compilación "cannot find symbol", esto indica que la clase Generic no
contiene un método doSomethingMoreSpecific
    Specific s1 = (Specific)g1;
    s1.doSomethingMoreSpecific(); //En este caso, la instancia es de tipo Specific, de manera que conoce el método
doSomethingMoreSpecific

    g2.doSomething();
    g2.doSomethingMoreSpecific(); //Genera un error de compilación "cannot find symbol", esto indica que la clase Generic no
contiene un método doSomethingMoreSpecific
    s1 = (Specific)g2; //Se genera un error en tiempo de ejecución "java.lang.ClassCastException"

    //Para prevenir esto en tiempo de ejecución, antes de realizar el downcast verificamos que el objeto sea del tipo a castear
    if ( g2 instanceof Specific ) { //Como el objeto no es de tipo Specific, ni un subtipo del mismo, la sentencia if resulta false, y no
se entra en la condición, previniendo el error en tiempo de ejecución
      s1 = (Specific)g2;
      s1.doSomethingMoreSpecific();
//También se puede realizar un casteo directo como el siguiente
((Specific)g2).doSomethingMoreSpecific();
    }
  }
}
```

Lo único que el compilador verifica es que el tipo de la variable con el tipo a castear se encuentren en el mismo árbol de herencia. Lo que si puede generar un error de casteo es al intentar realizar un **upcasting**.

```
public class Generic {}
public class Specific extends Generic{}
public class Another {}

public class Program {
  static public void main (String[] args) {
    Specific sp = new Specific();
    Another an = new Another();

    Generic s1 = (Generic)sp;
    Generic an = (Generic)an; //Error de compilación al intentar realizar un upcasting "inconvertible types"

    //El upcasting también se puede escribir de la siguiente manera
    Generic s1 = sp; //Esta sintaxis automáticamente está realizando un upcasting
  }
}
```

También, puede darse un caso como el planteado a continuación:

```
public interface Doable {
  void doSomething();
}

public class Generic implements Doable {
  public void doSomething() {
    //. . . some code here
  }
}
public class Specific extends Generic {}
```

La clase `Specific` puede ser tanto de tipo: `Specific`, `Generic`, o `Doable`, dado que su superclase la implementa, y ya se hizo cargo de darle un cuerpo a los métodos definidos en la interfaz.

Implementar (`implements`) una interfaz

Cuando se implementa una interfaz, lo que se hace es aceptar el contrato provisto por esta, e implementarlo en la clase. Para el compilador, lo único que requiere es que se le dé una implementación legal al método, pudiendo ser esta un cuerpo vacío por ejemplo (`{}`).
Para que la implementación de una interfaz sea legal, la primer clase no abstracta del árbol de herencia debe cumplir con las siguientes reglas:

- Especificar una implementación concreta para todos los métodos definidos en el contrato de la interfaz.
- Seguir todas las reglas para la sobre escritura de métodos.
- Declarar sin excepciones controles sobre los métodos de ejecución distintos de los declarados por el método de interfaz, o subclases de los declarados por el método de interfaz.
- Mantener la misma firma del método, y devolver un valor del mismo tipo (o subtipo).

Tipos de valor de retorno válidos

Esta sección implica dos cosas:

- Lo que se puede declarar como un valor de retorno.
- Qué tipo de dato se puede devolver realmente como valor de retorno.

Declaración de tipos de valor de retorno (Return types declaration)

Esta sección indica que tipo de valor de retorno puedes escribir en la declaración del método, dependiendo si se trata de un método:

- Sobrecargado
- Sobre escrito
- Nuevo

Tipos de valor de retorno en métodos sobrecargados

El tipo de valor de retorno puede ser cualquiera (el método debe calificar para la declaración de sobrecarga).

Tipos de valor de retorno en métodos sobre escritos

Se puede especificar el mismo tipo, o un subtipo (sublass), como tipo de valor de retorno (el método debe calificar para la declaración de sobre escritura).
NOTA: esto es posible a partir de Java 5, en versiones anteriores, solo es posible utilizar el mismo tipo.

Devolviendo un valor

Solo existen seis reglas:

- Puedes devolver un **`null`** cuando el tipo es un objeto.
- Un array representa un tipo de dato valido como retorno.
- En un método con un valor de retorno de tipo primitivo, es posible devolver un tipo de dato de otro tipo, que sea casteado implícitamente al tipo de retorno.
- Idem anterior con un tipo que puede ser casteado explícitamente (se debe realizar el cast antes de devolver el valor).
- Si el tipo es **`void`**, el método no puede contener ningún **`return`**.
- Si el tipo es un objeto, se puede devolver cualquier objeto que sea del mismo tipo o que pertenezca al árbol de herencia (no superclases ni superiores). Cualquier tipo que pase la condición **`instanceof`** del tipo de valor de retorno es válido.

Constructores e instanciación

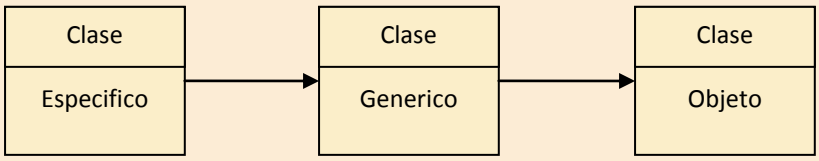
Los objetos son construidos. No es posible construir un nuevo objeto, sin antes invocar el constructor.
Los constructores representan bloques de inicialización, los cuales pueden llegar a ser invocados con el operador **new** (no es el caso de los bloques de inicialización estáticos).

Conceptos básicos de los constructores (Constructor basics)

Toda clase, incluidas aquellas que sean abstractas, deberán de tener un constructor. Pero esto no implica que el programador tenga que especificarlo.

Llamadas en cadena a constructores (Constructor chaining)

Suponiendo el siguiente árbol de herencia:



Al invocar la creación de un objeto de tipo **Especifico**, el proceso de inicialización es el siguiente:

1. Se invoca el constructor se **Especifico**. Este invoca el constructor de **Generico** (esta invocación puede ser implícita o explícita).
2. Se invoca el constructor de **Generico**. Este invoca el constructor de **Objeto** (esta invocación es siempre implícita).
3. Las variables de instancia del objeto **Objeto** reciben sus valores explícitos.
4. Se ejecuta el constructor de **Objeto**.
5. Las variables de instancia del objeto **Generico** reciben sus valores explícitos.
6. Se ejecuta el constructor de **Generico**.
7. Las variables de instancia del objeto **Especifico** reciben sus valores por defecto.
8. Se ejecuta el constructor de **Especifico**.

Reglas de los constructores (Rules for constructors)

Las siguientes reglas representan las bases de los constructores, y serán necesarias para rendir el examen.

- Los constructores pueden utilizar cualquier modificador de acceso.
- El nombre del constructor debe ser igual al de la clase.
- Los constructores no deben de tener un valor de retorno.
- Es legal tener un método que tenga el mismo nombre que la clase. Siempre que este tenga un valor de retorno, es un método.
- Si no se especifica explícitamente un constructor en la clase, implícitamente se genera un constructor que no recibe parámetros.
- Si se especifica al menos un constructor, el constructor implícito no será generado, de manera que si es necesario un constructor con argumentos, y el por defecto, es necesario especificarlo explícitamente.
- Todos los constructores tienen en la primer línea, una llamada a un constructor sobrecargado (this()), o una llamada a un constructor de la superclase (super()).

- La llamada a un constructor de la superclase puede ser con o sin argumentos.
- Un constructor sin argumentos no es necesariamente un constructor por defecto (este es el que especifica el compilador, en caso de no declarar explícitamente ningún constructor).
- No es posible acceder a una variable de instancia, o método de instancia, hasta que se ejecute el constructor de la superclase.
- Solo variables y/o métodos estáticos pueden ser llamados al invocar un constructor mediante this() o super().
- Las clases abstractas también tienen constructores, y estos son ejecutados cuando se instancie una subclase.
- Las interfaces no tienen constructores. Las interfaces no forman parte del árbol de herencia.
- La única manera de invocar un constructor es desde otro constructor.

Determinar cuándo se creará el constructor por defecto

El constructor por defecto será creado por el compilador, siempre que no se haya declarado ningún constructor en la clase.

Como es el constructor por defecto?

- Tiene el mismo modificador de acceso que la clase.
- No recibe ningún argumento (no tiene parámetros en su definición).
- Llama al constructor de la superclase sin argumentos.

Qué pasa si el constructor de la superclase requiere argumentos?

- Es necesario crear explícitamente los constructores en la clase, y dentro de la implementación de estos, llamar al constructor de la superclase pasando los argumentos necesarios.

Sobrecarga de constructores

Simplemente se comporta al igual que los métodos. Para sobrecargar un constructor basta con declarar varios de estos con diferentes parámetros.

Regla absoluta de los constructores. La primer línea de estos deberá ser una llamada a `super()` o `this()` (pueden contener parámetros en la llamada, siempre que la llamada no la realice el compilador).

Alguno de los constructores de la clase deberá de llamar a `super()`, y verificar que todos los demás constructores terminen yendo al que realiza dicha llamada.

Métodos y variables estáticos

Todos los métodos y atributos declarados con el modificador de no acceso `static` pertenecen a la clase y no al objeto (instancia de la clase).
Para saber cuándo debemos deberíamos de crear un atributo o método estático, simplemente necesitamos saber si este requiere o no acceder a algún atributo de la clase (en el caso de los métodos), o si debe ser un valor compartido para todos los objetos (en el caso de los atributos).
Características de los atributos y métodos estáticos:

- No es necesario inicializar ningún objeto para poder acceder a estos.
- Los atributos estáticos son compartidos por todos los objetos (en realidad son de la clase, y dado que el objeto pertenece a dicha clase, también tiene acceso a los atributos y métodos de esta).
- Las variables estáticas tienen la misma inicialización por defecto que las variables de clase.
- Un método estático no puede acceder a ningún miembro no estático (`main` es estático!).
- **No existe** la sobre escritura (overriding) de métodos estáticos, pero si la sobrecarga (overloading).

Accediendo a métodos y variables estáticas

Dado que podemos acceder a miembros estáticos sin siquiera haber instanciado una clase, para referenciarlos, utilizamos el nombre de la clase con el `.` para acceder.

```
public class Generic {
  static public int numero = 0;
}

public class Specific extends Generic {
  static public int numero = 1;

  static public void main (String[] args) {
    int tmp = Specific.numero;    //Nomenclatura utilizada para acceder a miembros estáticos (NombreClase.Miembro)

    Generic c = new Specific();  //Es válido por el compilador. Utiliza los miembros estáticos del tipo de dato declarado.
    tmp = c.numero();        //En este caso, el valor al que se accede es el 0
  }
}
```

En caso de utilizar un identificador que no sea el nombre de la clase (una variable de referencia), se accede a los miembros estáticos del tipo declarado de la referencia (no el tipo real contenido).

Lo que se viene

En la próxima entrega estaremos adentrándonos en las asignaciones.

Veremos que clases de literales existen en el mundo de Java, y como utilizar cada uno de ellos, así como sus compatiblidades e incompatibilidades.

Aprenderemos a inicializar y utilizar arrays, matrices, e hipermatrices. Como referenciarlos y acceder a cada uno de sus elementos.

Descubriremos bloques de inicialización de clase y objeto.

Y conoceremos sobre las clases Wrapper de Java que existen para cada tipo de valor primitivo.

¿Tenés lo que se necesita para ser un SCJP?

Autores
Gustavo Alberola
Matías Álvarez

Diseño
Leonardo Blanco