Xtend para Programadores Objetosos

March 5, 2013

Basado en "Java para Programadores Objetosos" de Leonardo Gassman.

1 Objetivo

El presente documento pretende facilitar la transición de un programador hacia el mundo de Xtend. Se requiere tener nociones sobre programación orientada a objetos. No pretende ser un manual de referencia de Xtend. No abarca el lenguaje completamente. La utilidad de este documento radica en que al finalizar la lectura del mismo, el programador estará en condiciones de sentarse a realizar un programa sencillo. El aprendizaje se complementa con un posterior estudio del resto de los apuntes de la cátedra dónde ciertos temas se ven en mayor profundidad. Algunas de las soluciones propuestas en este documento no son las mejores en cuanto a su diseño, sin embargo fueron elegidas por su contenido didáctico.

2 Introducción a Xtend

Xtend es una extensión de Java, un orientado a clases y tipado, que ofrece una sintaxis diferente, más amigable y menos verbosa. Para que un programa realizado en Xtend pueda correr, se necesita algo llamado máquina virtual java (java virtual machine o JVM, o simplemente VM). La JVM es el ambiente dónde viven los objetos, y tiene ciertas herramientas, como un garbage collector, que nos simplificará el uso de la memoria. Éste se encarga de detectar los objetos que ya no pueden ser usados porque nadie los referencia y los elimina de la memoria automáticamente.

2.1 Instalación de la jdk

Además de la VM, para poder desarrollar en Xtend, se necesita algunas cositas más, todo esto viene en algo llamado JDK (java developer kit). Entonces, lo primero que se debe realizar es instalar el jdk en la máquina que se usará para desarrollo. La jdk se puede bajar del sitio de Sun:

??

Al escribir este documento, la última versión estable es la jdk 7. Bajar e instalar para el sistema operativo que corresponda.

2.2 Instalación de eclipse

Para poder usar la sintaxis de Xtend necesitamos usar un entorno especialmente preparado para ello, que transforme nuestros programas en algo que pueda ejecutarse en la JVM. Eclipse es un entorno de desarrollo muy avanzado, que brinda varias herramientas que facilitan el trabajo del desarrollador, como resaltado de código y autocompletado. La herramienta que vamos a usar es el Eclipse. Hay varias distribuciones distintas de Eclipse, que se diferencian en los plugins que tienen instalados al momento de la descarga. La distribución recomendada es la que puede encontrarse en **??**, pero también es posible usar otra, agregando los plugins necesarios a mano.

Para instalar el eclipse, basta con descomprimir el zip bajado del sitio. El eclipse es un programa desarrollado en java, y por lo tanto, requiere de haberse ejecutado previamente la instalación de la jvm. (Recuerde que la instalación de la jdk, incluye la instalación de la jvm). Al ejecutar eclipse.exe entonces, se abre el entorno de programación.

Al iniciar, se pide que se elija un workspace. El workspace es una carpeta dónde se guardan los proyectos y ciertas configuraciones. En ocasiones es cómodo tener más de un workspace, pero en nuestro caso elegimos el que trae por default y usaremos siempre ese.

Cerramos la ventana del "welcome" que en este tutorial no nos aporta nada. Vamos a ver que la pantalla del eclipse está dividida en diferentes secciones. En la titulada "Package Explorer" (a la izquierda) se pueden ver los proyectos con sus respectivos árboles de clases; en el centro se visualizará el código de las clases una vez hayan sido seleccionadas en el Package Explorer y en la parte inferior se puede ver la lista de errores de compilación y problemas, en caso de haber alguno.

3 Armado de un proyecto

Ahora que ya tenemos el eclipse levantado, debemos crear un proyecto java para empezar a codificar nuestro programa. Para eso, se debe hacer: File -¿ new -¿ project -¿ java project. De nombre le pondremos "mi primer proyecto xtend" y antes de aceptar, marcaremos la opción "create separates source and output folder". ¿Qué significa eso que marcamos? Java maneja principalmente dos tipos de archivos, los .java, dónde se escribe el código, y los .class, que se crean al compilar el proyecto. El eclipse suele abstraernos de ese paso, pero por prolijidad, es conveniente mantener separados las carpetas dónde escribimos el código, de la carpeta dónde se guarda el código compilado. ¿De que me sirve un archivo .class? estos archivos son interpretados por cualquier JVM.

¿Que fue lo que sucedió al crear este proyecto?

En la sección "Package Explorer" apareció un nuevo proyecto y, dentro de este, una carpeta "src", dónde se guardan los archivos fuente asociado a nuestras clases y la jerarquía de packages.

¿Qué son los packages? Son agrupaciones de clases.

Sirven para dos cosas principalmente. La primera, es armar una estructura modular o de componentes de mi aplicación, de esta forma, pongo las clases más cohesivas y acopladas en el mismo package. La segunda, es para aportar un nombre único a las clases que contiene. Los nombres de packages suelen comenzar

como una url invertida, de esta forma, se garantiza que sea un nombre único en el mundo. El nombre real de una clase, es: "nombrePackage.NombreClase". La ventaja que trae tener un nombre único de clase, es que facilita la integración entre código escrito por distinta gente.

4 Escribiendo nuestras clases

Cómo no podía ser de otra manera, nuestro ejemplo será un "Hola Mundo".

Antes que nada, vamos a crear un package. Hay muchas formas de crear packages y clases, la más fáciles son haciendo clic derecho en el lugar dónde queremos crear, y eligiendo la opción new. Entonces, en la java browser, hacemos click derecho en la carpeta src, new package, y de nombre pondremos:

ar.edu.utn.frba.tadp.holamundo

Luego, creamos dos clases Xtend en ese package llamadas Recepcionista y Mundo (clic derecho sobre el package,new Xtend Class *Ojo! No hay que confundirse! Hay que crear una Xtend Class y no una "Class" a secas...*). Por convención, los nombres de package van con minuscula y las clases Comienzan con mayúscula y en cada palabra nueva, se usa mayúscula (lo denominado "camelcase").

Entonces, después de esto, puedo ver un package, y dos clases. Es posible que, siendo las primeras clases que creamos en este proyecto, ahora figuren con errores. Estos pueden identificarse porque parte del código de las clases está resaltado en rojo, aparece una cruz roja al costado y hay una o más entradas en la pestaña "Errores". El problema del que estamos hablando se presenta porque al proyecto le faltan librerías necesarias para trabajar con Xtend y no está listo para entender el código de las clases creadas. Por suerte, el Eclipse puede solucionar esto facilmente; solo necesitamos pasar el mouse sobre el nombre de la clase, subrayado en rojo,(o ubicar el puntero sobre él y apretar Ctrl+1) y se desplegará un menú ofreciendonos una solución rápida, que consiste en agregar las librerías necesarias al proyecto. Hacemos click en ella y nuestro error debería corregirse.

5 Agregando comportamiento

Vamos a hacer que nuestro recepcionista pueda saludar al mundo. Entonces vamos a hacer que Recepcionista entienda el mensaje saludar(Mundo). Mundo sabrá devolver su nombre con el mensaje getNombre(). Nuestro recepcionista escribirá el saludo por consola. ¿Cómo quedarían nuestras clases?

```
1 class Mundo {
2     def getNombre() {
3         "Mundo"
4     }
5 }
```

```
1 class Recepcionista {
2     def saludar(Mundo unMundo){
3         println("Hola␣" + unMundo.name)
4     }
5 }
```

¿Que cosas nuevas aparecen?

Para definir un método, se utiliza la palabra clave *def*, el nombre del método y entre paréntesis el tipo y nombre de los parámetros (separados por coma si hubiera). Es importante notar que no es necesario especificar el tipo de retorno del método, dado que Xtend va a inferirlo. Esta caracteristica del lenguaje nos va a permitir omitir la especificación del tipo en ciertos lugares; sin embargo, en general, a las variables y parámetros se le debe indicar.

También podemos observar que no es necesario el uso de palabras clave para retornar un valor; Xtend simplemente va a retornar el resultado de la ultima linea del método.

Que es System::out.println?? Es un método La clase System es una clase dónde hay cosas propias del sistema, tiene un atributo público de clase llamado out, que representa a la salida estandar, al enviarle el mensaje println, entonces se escribe por pantalla. Note que el operador "+" sirve para concatenar String. Por convención, los métodos empiezan con minúscula y las clases con mayúscula. En ambos casos, si tiene más de una palabra, las palabras nuevas empiezan con mayúsculas.

6 Escribiendo un main

Para probar lo que estamos haciendo, vamos a hacer una clase de prueba, que tendrá un método main que el eclipse podrá correr (el eclipse o directamente cualquier JVM, pues el eclipse no hace más que enviarle el pedido a la JVM que configuramos). Nuestra clase Test, deberá tener un método Main de la siguiente forma.

```
1  class Test {
2      def static void main(String[] args) {
3          var Recepcionista recepcionista
4          recepcionista = new Recepcionista
5
6          var Mundo mundo
7          mundo = new Mundo
8
9          recepcionista.saludar(mundo)
10     }
11 }
```

Veamos que es esto. La palabra static, indica que es un método de clase. String[], es un array de String. Al llamar a la JVM se le puede pasar argumentos. En nuestro caso no lo usaremos.

La primera línea, es la definición de una variable de tipo Recepcionista.

En la segunda, estamos construyendo un Objeto de tipo Recepcionista, y se lo estamos asignando a la variable declarada en la línea anterior. La palabrita new, significa que se invoca un constructor. Más adelante hablaremos sobre los constructores. La asignación ocurre con el operador "=".

Las líneas 3 y 4 son iguales a las 1 y 2, pero para construir el mundo.

Finalmente, al objeto recepcionista, se le envía el mensaje saludar, con el parámetro mundo.

Para correrlo, teniendo el foco en la clase Test, hacemos Run-¿ run as java application. Y en la consola, ubicada en una solapa en la parte inferior de la ventana, veremos el resultado.

Podemos achicar la cantidad de líneas de nuestro método main, agrupando varias acciones en la misma línea.

```
class Test {
    def static void main(String[] args) {
        var recepcionista = new Recepcionista
        var mundo = new Mundo

        recepcionista.saludar(mundo)
    }
}
```

Noten como ahora que las variables están siendo inicializadas en la misma linea que se las crea, Xtend puede inducir el tipo, por lo que no hay necesidad de escribirlo.

Podemos reducir aun más la cantidad de código; dado que no es necesario reutilizar los objetos involucrados en el test, podemos usarlos directamente al crearlos, sin necesidad de guardarlos en una variable.

```
class Test {
    def static void main(String[] args) {
        new Recepcionista.saludar(new Mundo)
    }
}
```

Los objetos se instanciarán, recibirán los mensajes enviados y luego serán llevados por el Garbage Collector.

7 Herencia

Vamos a hacer cambios para que haya dos tipos de Recepcionistas. Uno que sea el Clásico Hola Mundo, y otro que sea más formal.

Ambos tienen el comportamiento en común de escribir por consola, pero difieren en cómo armar el mensaje. Entonces, dejaremos el comportamiento en común en la clase Recepcionista, y escribiremos las clases RecepcionistaClasico y RecepcionistaFormal, que armaran el String de saludo de formas distintas. La clase Recepcionista la convertiremos abstracta porque no podemos tener recepcionistas que no sean de alguno de los dos tipos que nombramos, pues no sabríamos cómo armar el mensaje. Entonces, nuestra clase Recepcionista quedaría así:

```
abstract class Recepcionista {
    def saludar(Mundo unMundo){
        System::out.println(armarSaludo(unMundo))
    }

    def String armarSaludo(Mundo mundo)
}
```

Le tuvimos que agregar la palabra abstract a la definición de la clase. Esto hace que no se pueda instanciar ningun objeto Recepcionista directamente, lo que se instancian son subclases de la misma. También el definir la clase como abstract nos habilita a que ciertos métodos sean abstractos, y por lo tanto es responsabilidad de la subclase implementarlo. Para definir un método abstracto, basta con definirlo sin un cuerpo. Un ejemplo de esto es el método abstracto armarSaludo(Mundo).

Podemos ver que, por defecto, si no ponemos un receptor para un mensaje Xtend, el objeto se lo enviará a si mismo. Si deseamos explicitarlo podemos utilizar la palabra *this*, que significa que el receptor del método es el mismo objeto.

Para armar la subclase, si hacemos clic derecho en en package deseado y luego new Xtend Class, podemos elegir a Recepcionista como superclase, en lugar de Object. Si no lo hacemos así, podemos escribir el código que arma la relación nosotros mismos.

Las clases quedarían de la siguiente forma:

```
class RecepcionistaClasico extends Recepcionista {
    override armarSaludo(Mundo mundo) {
        "Hola " + mundo.getNombre()
    }
}
```

```
class RecepcionistaClasico extends Recepcionista {
    override armarSaludo(Mundo mundo) {
        "Buen dia, estimado " + mundo.getNombre()
    }
}
```

Para marcar la relación de herencia, en la declaración de la clase, se pone *extends* ClasePadre. La palabra clave *override* se utiliza para expresar la intención de redefinir el método abstracto definido en una clase superior de la jerarquía de herencia. De esta forma, si no ocurre, el código deja de compilar. Pruebe de cambiar el nombre del método de la superclase, y verá su utilidad.

Por default, las clases heredan de Object.

Probaremos nuestras nuevas versiones cambiando nuestro main de la siguiente manera:

```
class Test {
    def static void main(String[] args) {
        var mundo = new Mundo

        new RecepcionistaClasico().saludar(mundo)
        new RecepcionistaFormal().saludar(mundo)
    }
}
```

Fíjese que es la misma instancia de Mundo la que le llega a ambos recepcionistas.

8 Agregando nuevos atributos y constructores

Vamos a extender nuestro programa, para que salude tambien a las Personas. Para eso, vamos a crear una clase persona con el siguiente código:

```
class Persona {
    String nombre

    new(String nombre) {
        this.nombre = nombre
    }

    def getNombre() {
        this.nombre
    }
}
```

Hay dos cosas nuevas aquí. La primera, es que estamos definiendo un Atributo nombre para la persona. Por convensión, los atributos de las clases solo son visibles desde dicha clase, por lo que además agregamos un *getter* (un método cuyo único fin es retornar un atributo). La convención de la cómunidad Java/Xtend es que el nombre de un getter sea el mismo nombre que el del attributo que

accede, anteponiendo la palabra *get*. De la misma forma un *setter*, un método cuyo fin es cambiar el valor de un atributo, debe llamarse como el mismo, anteponiendo la palabra *set*. Xtend se basa en estas convenciones para brindarnos un poco de azucar sintáctico: Si una clase define un getter y/o un setter de forma correcta, se podrá consultar o modificar el valor del atributo como si estuvieramos dentro de la clase (es decir, escribiendo solamente *objeto.atributo* o *objeto.atributo = nuevoValor*). En estas situaciones, pese a verse como si estuvieramos modificando una variable, Xtend ejecutará los métodos definidos, permitiendonos una sintaxis más amena sin tener que exponer directamente los atributos. Podemos apreciar la diferencia en el siguiente ejemplo:

```
class Bateria {

    int energia

    def getEnergia(){
        this.energia
    }

    def setEnergia(int energia){
        this.energia = energia
    }
}
```

Dada esta definición de Bateria, los dos scripts presentados a continuación son equivalentes:

```
var bateria = new Bateria()

bateria.setEnergia(100)

bateria.setEnergia(bateria.getEnergia() + 10)

System::out.println(bateria.getEnergia())
```

```
var bateria = new Bateria()

bateria.energia = 100

bateria.energia = bateria.energia + 10

System::out.println(bateria.energia)
```

La segunda cosa nueva es que hay un método que no es igual a los métodos comunes: no está precedido por la palabra clave *def* y su nombre es *new*. Ese método es un constructor, y es utilizado para inicializar las instancias de la clase cuando se crean. Por defecto, toda clase viene con un constructor que no recibe ningún argumento, el cual no es necesario definir (por eso no necesitamos crear ninguno hasta ahora); pero como nos gustaría que las instancias de Persona se creen ya con su nombre agregamos un constructor que lo recibe por parámetro y lo setea.

Se puede tener varios constructores, que reciban diferentes combinaciones de argumentos, pero es importante saber que, una vez le agregamos un constructor a mano a nuestras clases, Xtend ya no le generará el constructor sin parámetros por defecto; por lo tanto, si se necesita, se deberá agregar manualmente.

Cada constructor, lo primero que hace es delegar en otro, ya sea de la misma clase o de su superclase. Si no se pone nada, se delega en el constructor vacío. Para delegar se usa *super*(param1, param2, ... , paramN) o *this*(param1, param2, ... , paramN), dependiendo de si quiero ejecutar un constructor de la misma clase o de su superclase. Si no se pone ningún super o this, entonces por default usa el constructor vacío.

Agregaremos entonces, la posibilidad de construir personas sin nombre, para agregarles el nombre después:

```
class Persona {
    String nombre

    new(){
        super() // Ejecuta el constructor de la superclase.
                // Esto se hace por defecto, no hace falta
                // incluirlo.
    }

    new(String nombre) {
        this() // Ejecuta el constructor sin argumentos.
        this.nombre = nombre
    }

    def getNombre() {
        this.nombre
    }

    def setNombre(String nombre) {
        this.nombre = nombre
    }
}
```

8.1 Clases Data

Es muy habitual que queramos que nuestras clases tengan constructores que permitan setear todos sus atributos, así como también tener getters que permitan accederlos desde afuera. En estos casos podemos marcar a la clase como *@Data*. Las clases Data generaran automáticamente un constructor que recibirá todos los atributos de la clase por parámetro (en el orden en que fueron definidos), y creará getters con el mismo nombre que los atributos para todos ellos. Además, los objetos instancia de clases Data sabrán compararse entre ellos para saber si son o no iguales y convertirse en un String representativo. Entonces, podríamos definir a Persona de la siguiente manera:

```
@Data class Persona {
    String nombre

    def setNombre(String nombre) {
        this.nombre = nombre
    }
}
```

Noten que fue necesario agregar el setter a mano, dado que el @Data solo agrega los getters. También cabe resaltar que esta versión de Persona no va a tener un constructor sin argumentos, dado que el constructor default, por ser una clase Data, recibirá el nombre por parámetro.

9 Interfaces

A pesar de que ahora, tanto Persona como Mundo entienden el mensaje getNombre, podemos ver que los recepcionistas solamente pueden saludar instancias de Mundo. ¿Porqué pasa esto? Fíjese que, lo único que los recepcionistas requieren del objeto al cual van a saludar es que entienda el mensaje getNombre(); sin embargo, debido a que Xtend es un lenguaje con tipado explicito, nos vimos forzados a definirle un tipo al argumento del método. Podemos ver en el código que la versión actual de este método está esperando un objeto de clase Mundo.

Podríamos cambiar la firma para que espere un objeto de clase Persona, pero entonces ya no podriamos saludar mundos. También podríamos hacer que Persona y Mundo tuvieran una superclase común y poner ese tipo en la firma, pero dado que Persona y Mundo no tienen nada en común (no hay código que queramos que ambos compartan, ni atributos repetidos) no parece buena idea; y, además, si Mundo o Persona pertenecieran ya a diferentes jerarquías, no se podría lograr.

La forma que Xtend provée para solucionar esta situación es utilizando una construcción llamada *Interface*. Las Interfaces en Xtend son similares a las clases abstractas, con la diferencia que no pueden definir ni atributos ni métodos no abstractos. Son particularmente útiles para definir tipos, dado que una clase puede implementar tantas interfaces como desee.

Armamos entonces una interface Nombrable, que defina el mensaje getNombre:

```
public interface Nombrable {
    String getNombre();
}
```

Al momento de escribir este apunte, Xtend aun no posée una sintaxis para definir Interfaces, así que debemos utilizar las Interfaces de Java. La diferencia en la sintaxis del código anterior se debe a que en realidad estamos viendo código java.

Ahora que tenemos la Interface creada, necesitamos hacer que tanto Mundo como Recepcionista, la implementen, y modificar la definición de *saludar* en los recepcionistas para que reciban un Nombrable, en lugar de un Mundo. El código de las clases quedaría de la siguiente forma:

```
@Data class Persona implements Nombrable {
    String nombre

    def setNombre(String nombre) {
        this.nombre = nombre
    }
}
```

```
class Mundo implements Nombrable {
    override getNombre() {
        "Mundo"
    }
}
```

```
abstract class Recepcionista {
    def saludar(Nombrable unNombrable){
        System::out.println(armarSaludo(unNombrable))
    }

    def String armarSaludo(Nombrable unNombrable)
}
```

```
class RecepcionistaClasico extends Recepcionista {
    override armarSaludo(Nombrable unNombrable) {
        "Hola␣" + unNombrable.getNombre()
    }
}
```

```
class RecepcionistaFormal extends Recepcionista {
    override armarSaludo(Nombrable nombrable) {
        "Buen␣dia,␣estimado␣" + nombrable.getNombre()
```

```
    }
}
```

Y veamos la siguiente clase de Test, dónde nos podemos dar cuenta del uso del polimorfismo entre Mundo y Persona:

```
class Test {
    def static void main(String[] args) {
        var recepcionista = makeRecepcionista()
        var Nombrable nombrable = new Mundo()

        recepcionista.saludar(nombrable)

        nombrable = new Persona("Jose")

        recepcionista.saludar(nombrable)
    }

    def static Recepcionista makeRecepcionista(){
        new RecepcionistaClasico()

        // Descomentar para retornar un Recepcionista Formal
        //new RecepcionistaFormal()
    }
}
```

Note que la variable de tipo Nombrable puede albergar instancias tanto de Mundo como de Persona, y que si cambiamos la implementación de makeRecepcionista para que devuelva un RecepcionistaFormal, el método main no se entera y funciona bien.

10 Colecciones

Xtend nos provee un framework de colecciones de alto nivel muy interesante, que facilita mucho el uso de conjuntos. En el siguiente ejemplo se muestra un ejemplo de uso.

```
class Test {
    def static void main(String[] args) {
        val nombres = newArrayList("Jose","Juana","Pedro")
        val gente = nombres.map([nombre|new Persona(nombre)])
        val genteConJ = personas.filter([persona|
            persona.nombre.startsWith("J")
        ])
        personasConNombreQueEmpiezaConJ.forEach([persona|
         makeRecepcionista().saludar(persona)
        ])
    }

    def static Recepcionista makeRecepcionista(){
        new RecepcionistaClasico()
    }
}
```

Cómo se puede ver modificamos nuestro test para saludar a varias personas a la vez.

En la primer linea estamos creando un *ArrayList* utilizando una función que lo instancia y lo inicializa con los objetos que recibe como argumento. Un *ArrayList* es un tipo de colección ordenada. Xtend ofrece muchos otros tipos de colecciones y es importante elegir la clase correcta que mejor representa el problema que modelamos.

La palabra clave *val* es equivalente a *var*, con la diferencia de que la variable creada no puede cambiar su valor. Es útil para crear valores constantes.

La segunda linea *mapea* o transforma la lista de nombres en otra de personas. El método *map* recibe un *bloque* como argumento; estos objetos son, en muchos aspectos, similares a funciones. Representan una porción de código que podemos ejecutar cuando queramos. En este caso, estamos usando un bloque que recibe un argumento "nombre" y al ser ejecutado crea una nueva Persona con él. El map utiliza este tipo de bloques para crear una colección a partir de otra.

En la tercer linea filtramos la colección creada en la linea anterior para quedarnos solamente con las personas cuyo nombre empieza con "J". Para eso le enviamos el mensaje *filter*. Note que el bloque que recibe *filter* espera un elemento de la colección y devuelve *true* si el objeto debe aparecer en el resultado y *false* si no.

En la última linea se utiliza el método *forEach* para iterar la colección filtrada. Por cada elemento que contenga, *forEach* ejecutará el bloque que le pasamos. Hay que tener presente que si el bloque produce un resultado el forEach va a descartarlo sin más, así que debemos utilizarlo solamente con bloques que produzcan un efecto.

Es importante notar que *map*,*filter* y *forEach* no modifican la colección que recibe el mensaje, a menos que el bloque que les pasamos por parámetro produzca un efecto persistente, podemos estar seguros que el receptor no sufrirá cambios.

En el siguiente código podemos ver el mismo ejemplo, escrito de una forma un poco más prolija.

```
1  class Test {
2      def static void main(String[] args) {
3          newArrayList("Jose","Juana","Pedro")
4              .map(nombre | new Persona(nombre))
5              .filter(persona | persona.nombre.startsWith("J"))
6              .forEach(persona |
7                  makeRecepcionista().saludar(persona)
8              )
9      }
10
11     def static Recepcionista makeRecepcionista(){
12         new RecepcionistaClasico()
13     }
14 }
```

Se puede ver que almacenar en variables intermedias no es necesario, podemos encadenar el envio de mensajes con normalidad. También se aprecia un azucar sintáctico de Xtend: cómo sabe que lo que reciben por parámetro los métodos es un bloque, no necesitamos poner los corchetes.