Java para programadores objetosos

ALGORITMOS²

por
Leo Gassman

Versión 2.1
Agosto 2009

Indice

OBJETIVO **3**

INTRODUCCIÓN A JAVA **3**

INSTALACIÓN DE LA JDK. 3
INSTALACIÓN DE ECLIPSE 3
ARMADO DE UN PROYECTO JAVA. 4

ESCRIBIENDO NUESTRAS CLASES. **4**

MODIFICADORES DE VISIBILIDAD. 5
AGREGANDO COMPORTAMIENTO. 5
ESCRIBIENDO UN MAIN 6
HERENCIA 7
AGREGANDO NUEVOS PACKAGES, ATRIBUTOS Y CONSTRUCTORES. 9
INTERFACES 10
COLECCIONES 13
TOSTRING 14
EQUALS 16
HASHCODE 18
ANNOTATIONS 18
COMENTARIOS Y JAVADOC 22

EXCEPCIONES **26**

MIS PRIMEROS BUGS. 26
ACCIONES SOBRE EXCEPCIONES 29
PROPAGACIÓN DE LA EXCEPCIÓN 31
MANEJO DEL ERROR 31
EXCEPCIONES ANIDADAS 33

Objetivo

El presente documento pretende facilitar la transición de un programador hacia el mundo de java. Se requiere tener nociones sobre programación orientada a objetos. No pretende ser un manual de referencia de java. 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.

Introducción a java

Java es un lenguaje orientado a clases y tipado. Para que un programa realizado en java 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 cositas, 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.

Instalación de la jdk.

Además de la VM, para poder desarrollar en java, 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: http://java.sun.com/javase/downloads/index.jsp
Al escribir este documento, la última versión estable es la jdk 6 update 7.
Bajar e instalar para el sistema operativo que corresponda.

Instalación de eclipse

Teóricamente, con la jdk y un editor de texto, ya alcanza para desarrollar programas. Sin embargo, realizarlo de este modo implica mucho esfuerzo a pulmón. Nosotros vamos a usar un IDE que nos simplificará las cosas. El IDE elegido es el Eclipse, uno de los mejores y más populares del mercado. Esto se baja de http://www.eclipse.org/downloads/
Hay varias distribuciones distintas de eclipse. Lo que varía entre las mismas son los plugins que traen incorporados. El eclipse está preparado para que se le puedan anexar plugins extendiendo su funcionalidad. Vamos a bajar el eclipse classic, que es una distribución básica, sin embargo el documento podría ser seguido con otra distribución.
La última versión estable al escribir este documento es la 3.4. 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. El eclipse puede verse de diferentes maneras para que quede más cómodo según la tarea que se quiera realizar. Cada una de estas formas se llama "Perspective". A su vez, cada ventana que se ve dentro de una perspective, se llama view. Las perspectivas que vamos a usar a lo largo de este tutorial, son la "java browser" y la "resource". En la resource, uno puede ver los archivos que intervienen en el proyecto de la forma en que se almacenan en el file system, mientras que en la java browser lo que se muestra es lo que representa cada archivo. De esta

forma, en la primera trabajamos con "folders" y archivos .class, .java, etc, mientras que en la segunda trabajamos con classes, types, packages, y conceptos relacionados con la programación java. Para cambiar de perspectiva se puede ir a Windows -> open perspective. O desde los íconos que figuran en la parte superior de la pantalla.

Ahora hay que decirle al eclipse cual es la VM que va a utilizar. Para eso, se debe ir a preference -> java -> Installed jre , hay que agregar la VM de la jdk.
Si se instaló en Windows, y no se modificó ninguna opción, sería: C:\Archivos de programas\Java\jdk1.6.0_07.
Si hay alguna otra, se recomienda removerla.

Armado de un proyecto java.

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 java" 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. Por eso cuando se habla de las ventajas de java, se dice que los programas son multiplataformas, porque cualquier archivo .class creado en una JVM puede ser leído por otra JVM. (Por ejemplo, un .class puede ser leido tanto por la JVM de Windows , como la de Linux).
¿Que fue lo que sucedió al crear este proyecto?
En la perspectiva de resources, show -> view- > navigator.
Se puede ver que hay dos carpetas: "src", dónde se guardan los archivos fuentes, y "bin" dónde se guardan los archivos compilados (a veces esta carpeta se llama "classes").
También se pueden ver dos archivos: ".project" y ".classpath" que son usados internamente por el eclipse. Normalmente, no deberíamos modificar directamente estos archivos.
En la perspectiva "java browsing" no se ve la carpeta "bin", porque es algo que no le interesa. Si se ve la carpeta src, y un montón de archivos .jar.
Un archivo .jar es una librería compuesta por un conjunto de clases. Los jars que aparecen en nuestro proyecto son los necesarios para que nuestras propias clases puedan ser ejecutadas. ¿Puedo agregar más jars? Sí, pero no es algo que haremos ahora.
Aparecen además dos ventanas más, la de package y la de types. En la de types aparecen las clases e interfaces (hay algunos tipos más que no veremos). ¿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.

Escribiendo nuestras clases.

Cómo no podía ser de otra manera, nuestro ejemplo será un "Hola Mundo", pero introduciendo conceptos de objetos (Ok. Alguien puede decir que voy a sobrediseñar la solución de este problema tan simple, pero no olvidemos que es a fines didácticos).

Entonces, primero haremos 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 en ese package llamadas Recepcionista y Mundo (clic derecho sobre el package, new Class). Por convención, los nombres de package van con minuscula y las clases Comienzan con mayúscula y en cada palabra nueva, se usa mayúscula.
Entonces, después de esto, en la perspectiva de java browser puedo ver un package, y dos clases. Si cambio a la perspectiva de resource, veo que me creo una estructura de carpetas src\ar\edu\utn\frba\tadp\holamundo con dos archivos Mundo.java y Recepcionista.java; y replicó la estructura de carpetas en bin, pero con Mundo.class y Recepcionista.class.
Cómo un package, en el file system es una estructura de subdirectorios, puede pensarse que ar.edu.utn.frba.tadp.holamundo es un subpackage de ar.edu.utn.frba.tadp. Este pensamiento tiene sentido para que nosotros ordenemos nuestras clases con algún criterio, pero para la JVM, ambos packages son independientes y no tienen relación alguna.

Modificadores de visibilidad.

Si observamos el código que se escribió automáticamente, podemos deducir cómo es la declaración del package y de la clase. Vemos también que cuando declaramos la clase, se le agrega la palabra "public". ¿Qué es eso?, es un modificador de visibilidad. Las clases, métodos y atributos tienen un alcance y no pueden ser vistas desde cualquier parte del programa. Algo "private", sólo se puede ver desde esa clase. Si no ponemos nada (default) se puede ver desde todo el package al cual pertenece. La visibilidad "protected" es similar a la de package, y además puede ser visto por las subclases.
Es una buena práctica que todos los atributos de los objetos sean "private".

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?

Mundo.java:
```
package ar.edu.utn.frba.tadp.holamundo;


public class Mundo {

	public String getNombre() {
		return "Mundo";
	}

}
```

Recepcionista.java
```
package ar.edu.utn.frba.tadp.holamundo;


public class Recepcionista {

	public void saludar(Mundo mundo) {
```

```
        System.out.println("hola " + mundo.getNombre());
    }

}
```

¿Qué cosas nuevas aparecen?
Para definir un método, se debe poner el modificador, el tipo del retorno, el nombre del método y entre paréntesis el tipo y nombre de los parámetros (separados por coma si hubiera).
Como lo puede sospechar, java es tipado! Entonces, a las variables y parámetros se le debe indicar el tipo.
¿Qué es System.out? La clase System es una clase donde 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.

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).
Para evitar que se nos mezcle nuestra clase de test con nuestras clases de negocio, agregaremos una nueva carpeta de sources para escribir test. Entonces, hacemos clic derecho sobre el proyecto en la vista projects, new -> source folder (no confundir con folder). La llamaremos test. Y aceptamos.
Vamos a crear el mismo package en la carpeta test, y luego la clase Test en dicho package.
Fíjese que si en la vista java browser, hacen clic en el proyecto, y luego en el package ar.edu.utn.frba.tadp.holamundo, van a ver las tres clases, porque a pesar de que se encuentran en distintas carpetas físicas, los tres pertenecen al mismo package. Entonces, podemos hacer referencia a las cosas que sólo tienen visibilidad de package desde nuestra clase Test, y cuando haya que hacer el deploy de la aplicación será fácil dejar afuera las clases creadas para testing.
Nuestra clase Test, deberá tener un método Main de la siguiente forma.

```
package ar.edu.utn.frba.tadp.holamundo;

public class Test {

    public static void main(String[] args) {
        Recepcionista  recepcionista;
        recepcionista = new Recepcionista();
        Mundo mundo;
        mundo = new Mundo();
        recepcionista.saludar(mundo);
    }
}
```

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 veremos el resultado.

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

```
package ar.edu.utn.frba.tadp.holamundo;

public class Test {

	public static void main(String[] args) {
		Recepcionista  recepcionista = new Recepcionista();
		Mundo mundo = new Mundo();
		recepcionista.saludar(mundo);
	}
}
```

Aquí la declaración de la variable y la asignación ocurren en la misma línea.
Pero aún podemos achicarlo más, pues cómo las variables recepcionista y mundo no se vuelven a usar, pierde sentido que la tengamos que declarar. Podemos cambiar nuestro método test de la siguiente forma.

```
package ar.edu.utn.frba.tadp.holamundo;

public class Test {

	public static void main(String[] args) {
			new Recepcionista().saludar(new Mundo());
	}
}
```

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

Recepcionista.java

```
package ar.edu.utn.frba.tadp.holamundo;


public abstract class Recepcionista {

	public void saludar(Mundo mundo) {
		System.out.println(this.armarSaludo(mundo));
	}

	protected abstract 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. Por eso, el método armarSaludo está definido como abstract.

Para armar la subclase, si hacemos clic derecho en Recepcionista y luego new class, ya nos armará la relación de herencia automáticamente. Si no lo hacemos así, podemos escribir el código que arma la relación nosotros mismos.
El método armarSaludo es protected porque debe ser visto desde las subclases, y no es público porque los usuarios de recepcionista no deben llamarlo directamente.
La palabra this, significa que el receptor del método es el mismo objeto.

Las clases quedarían de la siguiente forma:

RecepcionistaClasico.java
```
package ar.edu.utn.frba.tadp.holamundo;

public class RecepcionistaClasico extends Recepcionista {

	@Override
	protected String armarSaludo(Mundo mundo) {
		return "hola " + mundo.getNombre();
	}

}
```

RecepcionistaFormal.java
```
package ar.edu.utn.frba.tadp.holamundo;

public class RecepcionistaFormal extends Recepcionista {

	@Override
	protected String armarSaludo(Mundo mundo) {
		return "Buen día estimado " + mundo.getNombre();
	}

}
```

Para marcar la relación de herencia, en la declaración de la clase, se pone extends ClasePadre.
El @Override, es una annotation. Una annotation es información adicional que se le puede agregar a un método, atributo o clase, para que alguien pueda leerla en otro momento. La annotation Override la lee el eclipse para saber que ese método debe sobrescribir a uno de 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.
Si no escribimos la annotation, funciona igual, pero ya no nos avisa si hay cambios en la superclase.
Por default, las clases extienden de Object.

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

```
package ar.edu.utn.frba.tadp.holamundo;

public class Test {
```

```
	public static void main(String[] args) {
		Mundo mundo = new Mundo();
		new RecepcionistaClasico().saludar(mundo);
		new RecepcionistaFormal().saludar(mundo);
	}
}
```

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

Agregando nuevos packages, atributos y constructores.

Vamos a extender nuestro sistemita, para que salude tambien a las Personas.
Vamos a hacer una clase persona, en un nuevo package.
Creamos el package ar.edu.utn.frba.tadp.entes en la carpeta de resource. Y allí la siguiente clase:
Persona.java

```
package ar.edu.utn.frba.tadp.entes;

public class Persona {

	private String nombre;

	public Persona(String nombre) {
		this.nombre = nombre;
	}

	public String getNombre() {
		return this.nombre;
	}

}
```

Hay dos cosas nuevas aquí. La primera, es que estamos definiendo un Atributo nombre para la persona. Sigue la misma convención que los métodos.
La segunda, es que hay un método que no es igual a los métodos comunes: no define que devuelve, y se llama igual que la clase. Ese método es un constructor. Antes no habíamos escrito ningún constructor, porque nuestros constructores al no tener parámetros, java los agrega automáticamente. Nuestro constructor recibe ahora un nombre por parámetro y se lo guarda en el atributo nombre. Luego, cada vez que alguien le pregunte el nombre, lo devolverá.
Se puede tener varios constructores. Al agregar un constructor, java deja de poner automáticamente el constructor sin parámetro, 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) . 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.

Persona.java

```
package ar.edu.utn.frba.tadp.entes;

public class Persona {

	private String nombre;
```

```
	public Persona() {
		super();
	}

	public Persona(String nombre) {
		this();
		this.nombre = nombre;
	}

	public String getNombre() {
		return this.nombre;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

}
```

Interfaces

Vamos a agregar la posibilidad de que Recepcionista pueda saludar a Mundos y a Personas. Fíjese que Recepcionista, sólo necesita que el objeto a saludar entienda el Mensaje getNombre(). Una forma de lograrlo, es armar una superclase común entre Mundo y Persona. Pero esto no es siempre posible, porque como java no tiene herencia múltiple, si Mundo y Persona ya pertenecieran a otras jerarquías, no se podría realizar. La forma de solucionarlo es armando una interface que defina el mensaje getNombre, y hacer que Mundo y Recepcionista, la implementen.
Una interface en java, es un archivo .java similar al que define una clase.

Nombrable.java

```
package ar.edu.utn.frba.tadp.entes;

public interface Nombrable {

	public String getNombre();
}
```

La clases que las implementan, lo hacen de la siguiente forma.

Persona.java

```
package ar.edu.utn.frba.tadp.entes;

public class Persona implements Nombrable{

	private String nombre;

	public Persona() {
		super();
	}

	public Persona(String nombre) {
		this();
		this.nombre = nombre;
```

```
	}

	public String getNombre() {
		return this.nombre;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

}
```

Mundo.java

```
package ar.edu.utn.frba.tadp.holamundo;

import ar.edu.utn.frba.tadp.entes.Nombrable;


public class Mundo implements Nombrable {

	public String getNombre() {
		return "Mundo";
	}

}
```

Note que en estas líneas hay un elemento nuevo, la línea que dice:

```
import ar.edu.utn.frba.tadp.entes.Nombrable;
```

¿Por qué es esto?. Dijimos al principio, que el nombre de una clase (y también el de las interfaces), es el nombre de package "."nombre corto. Cuando desde una clase se hace referencia a otra que está en el mismo package se puede usar el nombre corto sin problemas. Cuando la clase o interface que queremos usar está en otro package (como en este caso, Nombrable está en un package distinto al de Mundo), para usar el nombre corto debemos importarlo.
Otra forma de hacer que Mundo implemente Nombrable sin usar el import es:

```
package ar.edu.utn.frba.tadp.holamundo;


public class Mundo implements ar.edu.utn.frba.tadp.entes.Nombrable {

...
```

Nombrable, al ser una interface, define también un tipo de dato. Veamos como quedan las clases de Recepcionistas. Para hacer el cambio, podemos cambiar las tres clases una a una, o podemos usar el refactor de eclipse, para cambiar la superclase, y que solitas las subclases se den cuenta. Entonces, vamos a la clase recepcionista, hacemos clic sobre el metodo armarSaludo y apretamos alt + shift + c (que es el short cut a "change method signature") y reemplazamos Mundo por Nombrable. Después repetimos la operación para el método saludar.

Recepcionista.java

```
package ar.edu.utn.frba.tadp.holamundo;

import ar.edu.utn.frba.tadp.entes.Nombrable;
```

```
public abstract class Recepcionista {

	public void saludar(Nombrable nombrable) {
		System.out.println(this.armarSaludo(nombrable));
	}

	protected abstract String armarSaludo(Nombrable nombrable);

}
```

RecepcionistaClasico.java

```
package ar.edu.utn.frba.tadp.holamundo;

import ar.edu.utn.frba.tadp.entes.Nombrable;

public class RecepcionistaClasico extends Recepcionista {

	@Override
	protected String armarSaludo(Nombrable nombrable) {
		return "hola " + nombrable.getNombre();
	}

}
```

RecepcionistaFormal.java

```
package ar.edu.utn.frba.tadp.holamundo;

import ar.edu.utn.frba.tadp.entes.Nombrable;

public class RecepcionistaFormal extends Recepcionista {

	@Override
	protected String armarSaludo(Nombrable nombrable) {
		return "Buen día estimado " + nombrable.getNombre();
	}

}
```

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

```
package ar.edu.utn.frba.tadp.holamundo;

import ar.edu.utn.frba.tadp.entes.Nombrable;
import ar.edu.utn.frba.tadp.entes.Persona;

public class Test {

	public static void main(String[] args) {
		Recepcionista recepcionista = makeRecepcionista();
		Nombrable nombrable = new Mundo();
		recepcionista.saludar(nombrable);
		nombrable = new Persona("José");
		recepcionista.saludar(nombrable);

	}
```

```
	private static Recepcionista makeRecepcionista() {
		return new RecepcionistaClasico();
	}
}
```

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.

Colecciones

Veremos brevemente el uso de una colección, En java hay una interface llamada Collection, que tiene los métodos add y remove para agregar y sacar objetos. Tiene un método iterator, que devuelve un objeto que es capaz de recorrerlo. Una de las implementaciones se llama ArrayList, que implementa en realidad una subinterface de Collection, llamada List, (la herencia entre interfaces se escribe igual que entre las clases, con la diferencia de que una interface puede extender de varias simplemente separándolas por comas).
En nuestro test, vamos a crear una colección de Nombrables, vamos a recorrerlos para saludarlos. Hay mucha más información sobre colecciones, pero queda afuera del alcance de este documento[1].

```
package ar.edu.utn.frba.tadp.holamundo;

import java.util.ArrayList;
import java.util.Collection;

import ar.edu.utn.frba.tadp.entes.Nombrable;
import ar.edu.utn.frba.tadp.entes.Persona;

public class Test {

	public static void main(String[] args) {
		Collection<Nombrable> nombrables = getNombrables();
		Recepcionista recepcionista = makeRecepcionista();
		for(Nombrable nombrable : nombrables) {
			recepcionista.saludar(nombrable);
		}


	}

	private static Collection<Nombrable> getNombrables() {
		Collection<Nombrable> nombrables = new ArrayList<Nombrable>();
		nombrables.add(new Mundo());
		nombrables.add(new Persona("Juan"));
		nombrables.add(new Persona("José"));
		return nombrables;

	}

	private static Recepcionista makeRecepcionista() {
		return new RecepcionistaClasico();
	}
```

[1] Para más información de colecciones, puede consultar el apunte de la cátedra

```
}
```

toString

Vamos a modificar Persona para que tenga una dirección. En lugar de tener un atributo calle y otro número en la Persona, vamos a generar la abstracción de Dirección. Entonces las Personas tendrán una dirección, que podrán devolver. Y vamos a ir agregando cierto comportamiento básico con las direcciones. Por el momento, le podemos preguntar a una persona dónde vive.

Direccion.java

```
package ar.edu.utn.frba.tadp.entes;

public class Direccion {

	private String calle;
	private int numero;

	public Direccion(String calle, int numero) {
		this.calle = calle;
		this.numero = numero;
	}
}
```

Persona.java

```
package ar.edu.utn.frba.tadp.entes;

public class Persona implements Nombrable {

	private String nombre;
	private Direccion direccion;

	public Persona() {
		super();
	}

	public Persona(String nombre, Direccion direccion) {
		this();
		this.nombre = nombre;
		this.direccion = direccion;
	}

	public String getNombre() {
		return this.nombre;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public Direccion getDireccion() {
		return direccion;
	}

	public void setDireccion(Direccion direccion) {
		this.direccion = direccion;
	}
}
```

Test.java

```
package ar.edu.utn.frba.tadp.holamundo;

import ar.edu.utn.frba.tadp.entes.Direccion;
import ar.edu.utn.frba.tadp.entes.Persona;

public class Test {

	public static void main(String[] args) {
		Persona persona = new Persona("Leo", new Direccion("Esteban de luca",
		1322));
		System.out.println("La persona " + persona + " vive en " +
		persona.getDireccion());
	}

}
```

Al ejecutar el main la consola sale algo como:

```
La persona ar.edu.utn.frba.tadp.entes.Persona@42e816 vive en
ar.edu.utn.frba.tadp.entes.Direccion@9304b1
```

¿Por qué el test compiló, si el operador "+" opera entre String y tanto persona como dirección no lo son? En la clase Object hay un método toString(), lo cual significa que cualquier objeto sabe responder a ese mensaje. El operador simplemente opera contra el toString de Juan. Hay varias herramientas en el mundo java (debugging y logueos) que se basan en el toString, entonces, es importante que nuestra clase redefina este método para que diga algo sensato, ya que la implementación de object no es del todo feliz.

Modificamos entonces, las clases Persona y Direccion.

Persona.java

```
package ar.edu.utn.frba.tadp.entes;

public class Persona implements Nombrable {

	…


	@Override
	public String toString() {
		return this.getNombre();
	}
}
```

Direccion.java

```
package ar.edu.utn.frba.tadp.entes;

public class Direccion {

…

	@Override
	public String toString() {
		return this.calle + " " + this.numero;
	}
}
```

Aquí también vemos como el operador "+" puede resolver tipos básicos como enteros.

Ahora la consola del mismo main sería:

```
La persona Leo vive en Esteban de luca 1322
```

Equals

Vamos a agregar un mensaje a la persona para que nos cuente si vive en cierto lugar.
El test con el que lo vamos a probar es:

```
public static void main(String[] args) {
	Direccion direccion = new Direccion("Esteban de Luca", 1322);
	Persona persona = new Persona("Leo", direccion);
	if(persona.viveEn(direccion)){
		System.out.println(persona + " vive en " + direccion);
	}
	else {
		System.out.println(persona + " no vive en " + direccion +", vive en " +
		persona.getDireccion());
	}
}
```

Nuestra primera versión para resolver el mensaje viveEn es:

```
package ar.edu.utn.frba.tadp.entes;

public class Persona implements Nombrable {

…
	public boolean viveEn(Direccion direccion) {
		return this.direccion == direccion;
	}
}
```

Al ejecutarlo, la salida es:

```
Leo vive en Esteban de Luca 1322
```

Para resolver el problema, lo que hicimos es comparar si la dirección que nos pasaron por parámetro es la misma instancia que la dirección a la cual tengo como atributo. Es decir el operador == se fija que dos referencias apunten al mismo objeto.

En ciertas ocaciones, no se tiene la misma instancia para pasar por parámetro, y se construye un objeto que sea igual [2]. Suponemos que la persona se construye en cierto lugar y tenemos que saber si vive en Esteban de luca 1322. Al modificar el main para que eso ocurra, el test comienza a fallar.
Test.java

```
…
public static void main(String[] args) {
	Direccion direccion = new Direccion("Esteban de Luca", 1322);
	Persona persona = createPersona();
	if(persona.viveEn(direccion)){
		System.out.println(persona + " vive en " + direccion);
	}
	else {
		System.out.println(persona + " no vive en " + direccion +", vive en " +
		persona.getDireccion());
	}
}

private static Persona createPersona() {
	return new Persona("Leo", new Direccion("Esteban de Luca",1322));
```

[2] Lea el apunte de la cátedra sobre value objects acerca de estas ocaciones

```
}
...
```

Ahora que el objeto direccion que está como atributo y el que está como parámetro no son la misma instancia, la salida es:

```
Leo no vive en Esteban de Luca 1322, vive en Esteban de Luca 1322
```

Para evitar este problema, java propone un mecanismo a través de un método llamado equals. Este método está definido en la clase Object, por lo tanto todos los objetos lo tienen. La implementación de object hace exactamente lo mismo que el operador "==", por lo tanto, al cambiar la clase persona para que use equals en lugar de == no soluciona el problema. Sin embargo, al utilizar un método en lugar de un operador, da la posibilidad a aquellas clases que necesitan contestar la pregunta de otra forma puedan redefinirlo.[3]

Persona.java

```
package ar.edu.utn.frba.tadp.entes;

public class Persona implements Nombrable {
...
	public boolean viveEn(Direccion direccion) {
		return this.direccion.equals(direccion);
	}
}
```

Direccion.java

```
package ar.edu.utn.frba.tadp.entes;

public class Direccion {

...
	@Override
	public boolean equals(Object obj) {
		if (!(obj instanceof Direccion)) {
			return false;
		}
		Direccion otraDireccion = (Direccion) obj;
		return this.calle.equals(otraDireccion.calle) && this.numero ==
		otraDireccion.numero;
	}
}
```

Con esta implementación la salida es:

```
Leo vive en Esteban de Luca 1322
```

Analicemos un poco ese método equals.
¿Por qué el método recibe un Object en lugar de otra dirección?
El mecanismo de comparación está definido para todos los objetos. Hay muchos componentes que se basan en este mecanismo que desconocen absolutamente la existencia de la clase Direccion. Por ejemplo, puedo poner una dirección en una colección y luego preguntarle si la contiene. La colección entonces envía el mensaje equals declarado en Object a la dirección.
¿Por qué el instanceof? Este equals puede ser llamado con cualquier objeto por parámetro, entonces sólo si lo que viene es una dirección existe la posibilidad de que sean iguales. El instanceof es un operador que indica si el objeto puede ser asignado a una variable del tipo. Es poco recomendado porque su uso atenta contra el polimorfismo y contra una buena distribución de responsabilidades, sin embargo, el equals es un caso dónde sí es correcto usarlo.
Otro tema no trivial es la línea

[3] Para saber un poco más de las características de las clases que necesitan redefinir el equals, puede consultar el apunte sobre value objects de la cátedra

```
Direccion otraDireccion = (Direccion) obj;
```

Cómo obj es una variable de tipo Object, sólo me deja enviar mensajes definidos en ese tipo. Sin embargo, yo estoy seguro que hace referencia a una instancia de tipo Direccion (porque más arriba hice el instanceof), entonces, puedo castear el objeto a ese tipo. El casteo es una indicación que el desarrollador le hace al compilador para asegurarle que un objeto cumple con cierto tipo y a partir de ahí poder mandarle mensajes que antes no eran visibles.
¿Por qué usé equals para la calle e "==" para el número? Es correcto al decir que la igualdad de un objeto depende de la igualdad de algún atributo, usar el equals en el atributo. Ahora cuando ese atributo es de un tipo básico, la comparación se realiza con ==.
Para dar por concluido la redefinición del equals, es necesario redefinir otro método llamado hashCode.

HashCode

El método hashCode está definido en Object y es usado para realizar optimizaciones. Por este motivo existe una restricción que dice que si dos objetos son equals, entonces ambos deben devolver el mismo número en el método hashCode. Un claro ejemplo donde se ve el uso de hashCode es con la clase HashSet. Esta clase es una colección que tiene internamente una tabla de hash, entonces al preguntarle si contiene un objeto, primero reduce la cantidad de objetos a evaluar preguntando el hashCode, para luego preguntar por equals.
Vamos a probar el siguiente ejemplo y veremos el problema de haber redefinido el equals sin haber redefinido el hashCode.

```java
public static void main(String[] args) {
      Direccion direccion = new Direccion("Esteban de Luca", 1322);
      Direccion otraInstancia = new Direccion("Esteban de Luca", 1322);
      System.out.println("direccion.equals(otraInstancia): "
                  + direccion.equals(otraInstancia));
      Collection<Direccion> coleccion = new HashSet<Direccion>();
      coleccion.add(direccion);
      System.out.println("coleccion.contains(otraInstancia): "
                  + coleccion.contains(otraInstancia));
}
```

La salida de este main es:

```
direccion.equals(otraInstancia): true
coleccion.contains(otraInstancia): false
```

Es decir, tengo dos instancias iguales, pero la colección no se da cuenta que lo son.

Para poder redefinir el equals evitando este problema, se sigue una regla simple: redefinir el hashCode haciendo que dependa de los mismos atributos de los cuales depende el equals. Si una dirección es igual a otra si son la misma calle y número, un hashCode válido puede ser:

```java
      @Override
      public int hashCode() {
            return this.calle.hashCode() + this.numero;
      }
```

Annotations

Una annotation es metadata que se le puede agregar a los elementos que componen una clase, o más genéricamente un tipo. Para ver todos los elementos que se pueden anotar, revise el Enum ElementType.

Vamos a modificar nuestro programa para informe por pantalla si está saludando a un mundo o a una persona. Saber si se trata de un mundo o de una persona, se puede saber preguntando por la clase. Lo que no se puede saber es el género de lo que representa la clase, es decir si es "un mundo" o "una mundo". Entonces vamos a usar una annotation para agregar esta información.
De manera similiar a crear una clase, hacemos clic derecho sobre el package ar.edu.utn.frba.tadp.entes, new Annotation, y la llamamos Articulo.
En una annotation, solo podemos poder información que sea de tipo básico, String, Class o enums. (Los enums son un tipo de dato con un conjunto constante de valores) .
Cómo nosotros queremos agregar un solo valor, llamaremos a ese valor "value". Ponerle ese nombre hace que sea más corta la forma de utilizarlo, pues a la hora de usar la annotation si no se indica ningún nombre java por defecto usará value.

La annotation articulo nos quedará de la siguiente forma:

```
package ar.edu.utn.frba.tadp.entes;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Articulo {

      String value();

}
```

¿Qué significa @Target(ElementType.*TYPE*) ? Para crear nuestra annotation, estamos haciendo uso de otra annotation llamada Target, para indicar que esa annotation será usada sólo a nivel de tipo. Con lo cual si la usaramos para un atributo el compilador no nos dejará avanzar.
Si hubieramos obviado el uso de esta annotation, lo único que hubiera pasado es que se podría usar la misma en cualquier lugar.
Fíjese que ElementType es un enum, con todos los valores posibles de los lugares dónde se puede usar una annotation.

La annotation del target era fácil deducirlo, pero ¿y la Retention?. Las annotation tienen un tiempo de vida por el cual estarán disponbles. Hay tres niveles. Source: son annotation que viven en el .java y el compilador descartará al generar el binario, por ejemplo la @Override. Class: es el valor default, la annotation está presente en el archivo .class pero no mientras corre la JVM. Y la última es Runtime, en este caso el valor estará disponible en tiempo de ejecución.

Anotemos ahora Mundo y Persona indicando que articulo debería usarse en cada caso.

Mundo.java

```
package ar.edu.utn.frba.tadp.holamundo;

import ar.edu.utn.frba.tadp.entes.Articulo;
import ar.edu.utn.frba.tadp.entes.Nombrable;

@Articulo("un")
public class Mundo implements Nombrable {
```

```
…

}
```

Persona.java

```
package ar.edu.utn.frba.tadp.entes;

@Articulo("una")
public class Persona implements Nombrable {

…

}
```

Y modifiquemos la clase recepcionista para que haga uso de esta annotation

```
package ar.edu.utn.frba.tadp.holamundo;

import ar.edu.utn.frba.tadp.entes.Articulo;
import ar.edu.utn.frba.tadp.entes.Nombrable;

public abstract class Recepcionista {

	public void saludar(Nombrable nombrable) {
		this.log(nombrable);
		System.out.println(this.armarSaludo(nombrable));
	}

	private void log(Nombrable nombrable) {
		System.out.println("saludando a " + getArticulo(nombrable) + " " +
		nombrable.getClass().getSimpleName());
	}

	private String getArticulo(Nombrable nombrable) {
		return nombrable.getClass().
				getAnnotation(Articulo.class).value();
	}

	protected abstract String armarSaludo(Nombrable nombrable);

}
```

Fíjese que la annotation se le pide al class. De haber agregado una annotation en un atributo, entonces a la clase se le debe pedir el atributo (o field) y a el field la annotation. De la misma forma con un método.

Corramos nuestros main de test para ver la salida:

```
package ar.edu.utn.frba.tadp.holamundo;

import ar.edu.utn.frba.tadp.entes.Nombrable;
import ar.edu.utn.frba.tadp.entes.Persona;

public class Test {

	public static void main(String[] args) {
		Recepcionista recepcionista = makeRecepcionista();
		Nombrable nombrable = new Mundo();
		recepcionista.saludar(nombrable);
		System.out.println();
		nombrable = new Persona("José");
```

```
            recepcionista.saludar(nombrable);

      }

      private static Recepcionista makeRecepcionista() {
            return new RecepcionistaClasico();
      }
}
```

Consola:

```
saludando a un Mundo
hola Mundo

saludando a una Persona
hola José
```

Para ver como se usa cuando quiero agregar más de un valor a una annotation, vamos a hace que a las personas le diga "sr", mientras que al mundo no le diga nada. (Nótese que es un ejemplo didáctico, no pretendemos discriminar a las damas).

Entonces modificaremos nuestra annotation de la siguiente manera:

Articulo.java

```java
package ar.edu.utn.frba.tadp.entes;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Articulo {

      String cardinal();
      String titulo();

}
```

Mundo.java

```java
package ar.edu.utn.frba.tadp.holamundo;

import ar.edu.utn.frba.tadp.entes.Articulo;
import ar.edu.utn.frba.tadp.entes.Nombrable;

@Articulo(cardinal="un", titulo="")
public class Mundo implements Nombrable {
…
```

Persona.java

```java
package ar.edu.utn.frba.tadp.entes;

@Articulo(cardinal = "una", titulo = "Sr.")
public class Persona implements Nombrable {
…
```

En Recepcionista debemos cambiar para que en vez de usar "value()" use "cardinal()"

```
package ar.edu.utn.frba.tadp.holamundo;


public abstract class Recepcionista {
…

	private String getArticulo(Nombrable nombrable) {
		return nombrable.getClass()
			.getAnnotation(Articulo.class).cardinal();
	}
…
```

Y vamos a cambiar nuestro recepcionista clásico para que use el título.

```
package ar.edu.utn.frba.tadp.holamundo;

import ar.edu.utn.frba.tadp.entes.Articulo;
import ar.edu.utn.frba.tadp.entes.Nombrable;

public class RecepcionistaClasico extends Recepcionista {

	@Override
	protected String armarSaludo(Nombrable nombrable) {
		return "hola " + getTitulo(nombrable) + " " + nombrable.getNombre();
	}

	private String getTitulo(Nombrable nombrable) {
		return nombrable.getClass().
		getAnnotation(Articulo.class).titulo();
	}

}
```

Y ahora nuestra salida por consola es:

```
saludando a un Mundo
hola  Mundo

saludando a una Persona
hola Sr. José
```

Comentarios y Javadoc

En java existen los clásicos comentarios heredados de c:

```
	//La doble barra comenta la línea entera

	/* esto es un comentario que puede extenderse
	   varias líneas */
```

Sin embargo agrega un tipo especial de comentario llamado javadoc, que comienzan con "/**" y terminan con "* /"

```
	/**
	 * Esto es un comentario javadoc
	 */
```

Este tipo de comentario tiene un motivo especial, que es el de **documentar** todo lo que sea relevante **al desarrollador que va a utilizar la clase (interface, o cualquier archivo java)**, allí se nombra el motivo de existencia del componente y los contratos implícitos. Un contrato implícito es aquel que no puede comprobar el compilador, pero que es necesario cumplir, por ejemplo, que tal atributo de un constructor no puede ser *null*.

Existen herramientas que procesan los comentarios *javadoc* para generar una presentación útil de la clase. El más conocido es el comando *javadoc* que viene con el *jdk* que genera documentación html. También en el eclipse al apretar F2 sobre algún componente lo muestra en una ventana.

El *javadoc* se escribe especialmente para cada parte de un archivo java: Un comentario que explica para que sirve la clase, uno para cada método, y a veces se documenta los atributos cuándo éstos son relevantes. ¿Por qué no es común comentar los atributos? Porque la idea es documentar la interfaz que va a usar un desarrollador. Si nuestra clase maneja correctamente el nivel de abstracción y presenta una interfaz sensata, sus atributos van a estar correctamente encapsulados y por lo tanto no sería importante para un usuario de la clase. Por esa característica de ser documentación, es importante que se agregue información que sea útil y no simplemente decir lo que puedo deducir al ver la firma del método. Por ejemplo, agregar un *javadoc* para un *get* o un *set* no es útil, porque todo lo que necesito está en el nombre del método. A veces el nombre y tipo del parámetro ya es lo suficientemente descriptivo cómo para que no sea necesario especificarlo.

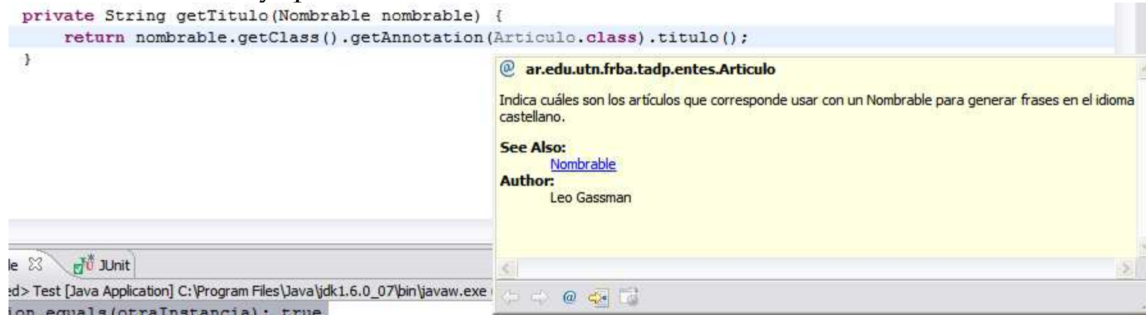
Vamos a ir documentando algunos de los componentes que realizamos con *javadoc*.
Vamos a empezar a documentar la *annotation* artículo.

```
package ar.edu.utn.frba.tadp.entes;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * Indica cuáles son los artículos que corresponde usar con un
 * Nombrable para generar frases en el idioma castellano.
 * @author Leo Gassman
 * @see Nombrable
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Articulo {
      /**
       * Dependiendo de si es masculino o femenino,
       * se admiten las formas "un" o "una"
       */
      String cardinal();

      /**
       * Es un título agregado al nombre
       */
      String titulo();

}
```

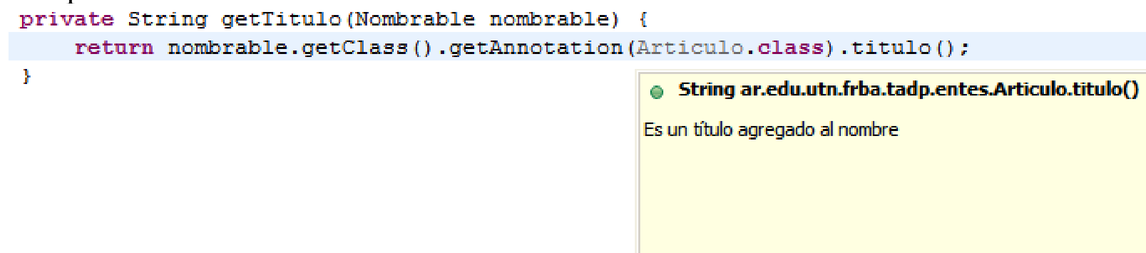
En este ejemplo se ve claramente dónde está la documentación de la *annotation*, y dónde está la documentación de cada atributo.
Hay otra cosa extraña que se observa. Hay unos comentarios que empiezan con "@". Estos son llamados tags y están predefinidos. Cada uno dice algo particular que podría ser útil al usuario. Estos tags son interpretados por las herramientas de presentación para formatear la información. En este caso usamos los tags *author* para indicar quien escribió la clase y *see* para decir que esta clase está relacionada con la Interface

Nombrable. Las herramientas de presentación suelen generar un link para navegar hacia el javadoc del componente relacionado.

Para ver cómo puede resultar útil el javadoc, podemos ir a la clase Recepcionista clásico, poner el cursor sobre Articúlo.class y apretar f2



Y al poner el cursor sobre *titulo()*



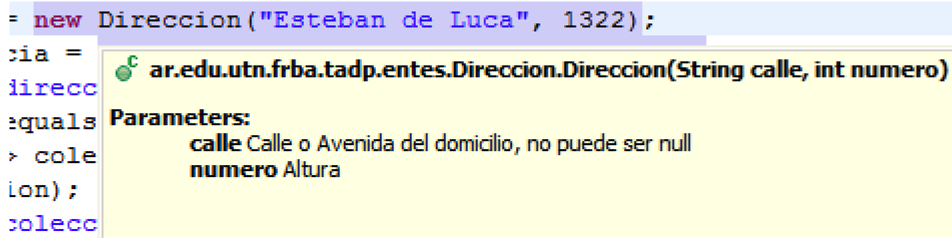
Hasta ahora escribimos comentarios que tienen que ver con la responsabilidad del componente. Sin embargo también es útil para decir ciertas características técnicas o contratos que cumple

```
package ar.edu.utn.frba.tadp.entes;

/**
 * Es un Value Object (inmutable) que representa
 * una dirección física
 * @author Leo Gassman
 */
public class Direccion {

	private String calle;
	private int numero;

	/**
	 * @param calle Calle o Avenida del domicilio, no puede ser null
	 * @param numero Altura
	 */
	public Direccion(String calle, int numero) {
		this.calle = calle;
		this.numero = numero;
	}
```

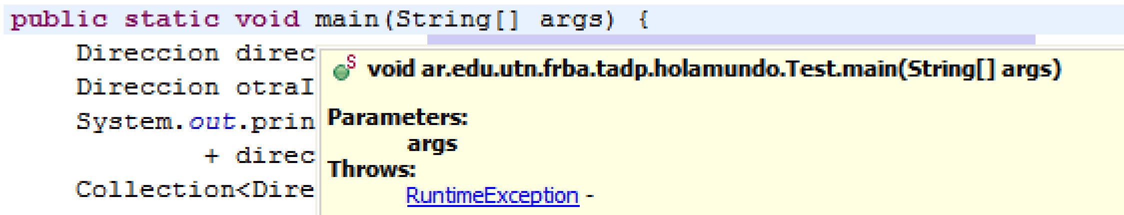
Nótese que ahora en estos comentarios se ve un contrato implícito: que calle no puede ser null, porque la clase no está preparada para funcionar con este valor. (Si no me cree puede generar una Dirección sin calle y compararla contra otra). Y las direcciones son *Value Objects*[4] y por lo tanto inmutables. Lo cuál nos permite tomar ciertas decisiones en la forma de usarlas.

Por otro lado introducimos un *tag* nuevo que es *param*, para dar indicaciones sobre los parámetros.

[4] Puede consultar el apunte de *Value Objects* de la cátedra



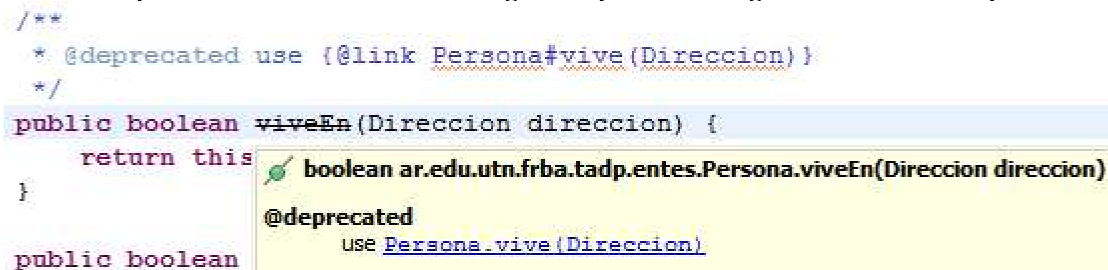
Otro *tag* útil es el *throws,* que indica que un método puede salir por un error e indica cuál puede ser, en la próxima sección hablaremos un poco de excepciones.



Por último, hay un tag llamado *deprecated* que se utiliza para indicar que un componente o método no debe utilizarse porque es algo viejo que existe por compatibilidad hacia atrás. Significa que hay una forma mejor de cumplir con la responsabilidad que hay detrás del componente o método, pero aún no puedo borrar la forma vieja porque se está usando. Es importante indicar al momento de deprecar cual es la nueva forma de realizar lo que se necesitaba. Vamos a deprecar el método viveEn de Persona, porque por convención no queremos utilizar artículos en los nombres de los mensajes.

```
/**
 * @deprecated use {@link Persona#vive(Direccion)}
 */
public boolean viveEn(Direccion direccion) {
	return this.vive(direccion);
}

/**
 * @return si es la dirección de la Persona
 */
public boolean vive(Direccion direccion) {
	return this.direccion.equals(direccion);
}
```

Note cómo al deprecar el eclipse tacha el nombre del método para resaltar que no conviene usarlo. Por otro lado, aparece el tag *link* que sirve para que las herramientas generen un vínculo con otro javadoc. También aparece en el método vive el tag *return* para decir algo acerca sobre lo que devuelve el método



```
/**
 * @return si es la dirección de la Persona
 */
public boolean vive(Direccion direccion) {
     return this
}
```

boolean ar.edu.utn.frba.tadp.entes.Persona.vive(Direccion direccion)

Returns:
si es la dirección de la Persona

Para terminar de ver la utilidad de javadoc, es interesante ver los generados para las mismas clases de java, en particular mire el javadoc de equals y hashcode de *Object*, y el de la interface Set, pués no es solamente información útil para quien va a llamar a esos métodos o usar un Set, si no para aquellos que quieran extender o implementar un *Object* o un *Set*.

Dijimos más atrás que existe un comando que convierte el javadoc en html. El eclipse sabe utilizar este comando, entonces vamos a generar el html.
Clic derecho sobre el proyecto -> export -> java -> javadoc -> next
Se puede elegir un destino, por default va a crear una carpeta doc dentro del workspace. Dejamos esta opción y clic en fisish. Y le decimos que sí a la nueva ventana de confirmación.
Ahora con un browser se puede navegar la documentación del proyecto.

C:\Documents and Settings\Leo\workspace\mi primer proyecto java\doc\index.html

All Classes

Packages
ar.edu.utn.frba.tadp.en
ar.edu.utn.frba.tadp.ho

All Classes
Articulo
Direccion
Mundo
Nombrable
Persona
Recepcionista
RecepcionistaClasico
RecepcionistaFormal
Test

Overview Package Class Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

ar.edu.utn.frba.tadp.entes
Class Direccion

```
java.lang.Object
  └ar.edu.utn.frba.tadp.entes.Direccion
```

```
public class Direccion
extends java.lang.Object
```

Es un Value Object (inmutable) que representa una dirección fisica

Author:
Leo Gassman

Constructor Summary

```
Direccion(java.lang.String calle, int numero)
```

Excepciones[5]

Mis primeros bugs.

Vamos a correr el siguiente main

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

[5] En esta sección se abarca el tema en forma básica. Para más detalles se puede consultar el apunte de excepciones de la cátedra

```
	Persona persona = new Persona();
	Direccion direccion = new Direccion("Esteban de Luca", 1322);
	if(persona.vive(direccion)) {
		System.out.println(persona + "vive en " + direccion);
	}
	else {
		System.out.println(persona + "no vive en " + direccion +", vive en" +
		persona.getDireccion());
	}
}
```

En teoría este main debería imprimir si una persona vive o no en una dirección, pero sin embargo por consola pasa lo siguiente:

```
Exception in thread "main" java.lang.NullPointerException
	at ar.edu.utn.frba.tadp.entes.Persona.vive(Persona.java:50)
	at ar.edu.utn.frba.tadp.holamundo.Test.main(Test.java:15)
```

Esto ocurre cuando hubo un error en el programa, y lo que uno ve por pantalla es lo que se llama stack trace. En un stack trace se ve dos cosas principalmente: la excepción ocurrida y cuál es el camino de los llamados a métodos que la provocaron. En nuestro caso la excepción es NullPointerException, lo cual significa que se envió un mensaje a una referencia que estaba nula (probablemente falta una inicialización). A veces la excepción viene acompañada también de un mensaje que agrega información útil para entender que es lo que ocurre.
Para detectar el lugar dónde está el problema, hay que ver el camino de llamados. El stack se lee de abajo hacia arriba: el método main de la clase Test llamó en la línea 15 al método vive de la clase Persona que en la línea 50 lanzó una NullPointerException.
Entonces, vamos a la línea 50 de Persona (con el eclipse basta al hacer clic en el stack trace sobre Persona.java:50) y vemos la siguiente línea:

```
	return this.direccion.equals(direccion);
```

Sabemos ahora que hay una referencia nula, ¿Quién es? ¿La Persona this? ¿El atributo direccion? ¿El parámetro direccion? ¿La referencia nula es dentro del equals?
La Persona que recibe el mensaje no puede ser nula, de lo contrario no se hubiera empezado a ejecutar el método.
Si el error hubiera sido dentro del equals, debería haber aparecido una línea más dentro del stack trace.
El problema no puede ser el argumento porque ningún mensaje es enviado a él, sólo se usa para pasarlo dentro del equals.
El problema está entonces en el atributo this.dirección. No sólo porque los demás no pueden ser, si no porque el equals es el único mensaje que se dispara en esa línea, Recordenos que la excepción NullPointerException significa que el receptor de un mensaje es nulo.
Para corroborar nuestras sospechas, revisamos que al instanciar la Persona en el main, no le dijimos en ningún momento cuál es la Dirección, es más ¡tampoco le asignamos el nombre! Corregimos agregando los seteos y corremos de nuevo:

```
public static void main(String[] args) {
	Persona persona = new Persona();
	Direccion direccion = new Direccion("Esteban de Luca", 1322);
	persona.setDireccion(direccion);
	persona.setNombre("Leo");
	if(persona.vive(direccion)) {
		System.out.println(persona + " vive en " + direccion);
	}
	else {
		System.out.println(persona + " no vive en " + direccion +", vive en" +
		persona.getDireccion());
```

```
        }
}
```

La consola dice:

```
Leo vive en Esteban de Luca 1322
```

Otros errores muy comunes:
Castear un objeto a un tipo que no corresponde, suele ocurrir mucho al trabajar con colecciones:

```
public class Test {
        static Direccion direccion =
                new Direccion("Esteban de Luca", 1322);

        public static void main(String[] args) {

                Collection<Nombrable> nombrables = getNombrables();
                for (Nombrable nombrable : nombrables) {
                        if(((Persona)nombrable).vive(direccion)) {
                                System.out.println(nombrable + " vive en mi casa");
                        }
                }

        }

        private static Collection<Nombrable> getNombrables() {
                Collection<Nombrable> nombrables = new ArrayList<Nombrable>();
                nombrables.add(new Persona("Leo", direccion));
                nombrables.add(new Mundo());
                nombrables.add(new Persona("fulano", direccion));
                return nombrables;

        }
}
```

```
Leo vive en mi casa
Exception in thread "main" java.lang.ClassCastException: ar.edu.utn.frba.tadp.holamundo.Mundo cannot be cast to ar.edu.utn.frba.tadp.entes.Persona
        at ar.edu.utn.frba.tadp.holamundo.Test.main(Test.java:17)
```

Esto se puede ver como que en el main, línea 17, Ocurrió una ClassCastException, se tenía una referencia a una variable de tipo Mundo y se quiso ver como Persona. Si revisamos es verdad, porque en la colección de Nombrables había mezclados mundos y Personas. Arreglar este error ya es un poco más complicado porque hay que cambiar una decisión de diseño: tener Personas y Mundos mezclados en esa colección que se usa para verificar si hay mas cosas que viven en mi casa, o agregar el mensaje vive en la interface Nombrable para que todos puedan contestar.

Cuando uno quiere correr algo sobre código que no compila, también ocurre una excepción.

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

    Collection<Nombrable> nombrables = getNombrables();
    for (Nombrable nombrable : nombrables) {
        if(((Persona)nombrable).vive(direccion)) {
            Esto no compila
            System.out.println(nombrable + " vive en mi casa");
```

…
La consola muestra:
```
Exception in thread "main" java.lang.Error: Unresolved compilation problems: 
	Esto cannot be resolved to a type
	Syntax error on token "compila", ; expected

	at ar.edu.utn.frba.tadp.holamundo.Test.main(Test.java:18)
```

Acciones sobre excepciones

En java las excepciones son objetos que extienden de una clase llamada Exception. Sin embargo hay una subclase particular llamada RuntimeException. El compilador concede ciertas libertades a las excepcions Runtime y sus subclases. Hasta el momento, las excepciones que vimos son runtime (también llamadas no chequeadas), por lo que el compilador no nos molestó para nada. Usaremos ahora una excepción chequeada.
Vamos a hacer entonces una clase Cartero, que tiene que generar un archivo con las direcciones de un conjunto de personas, para visitar y llevarle correspondencia.
Vamos a genear un package correo.

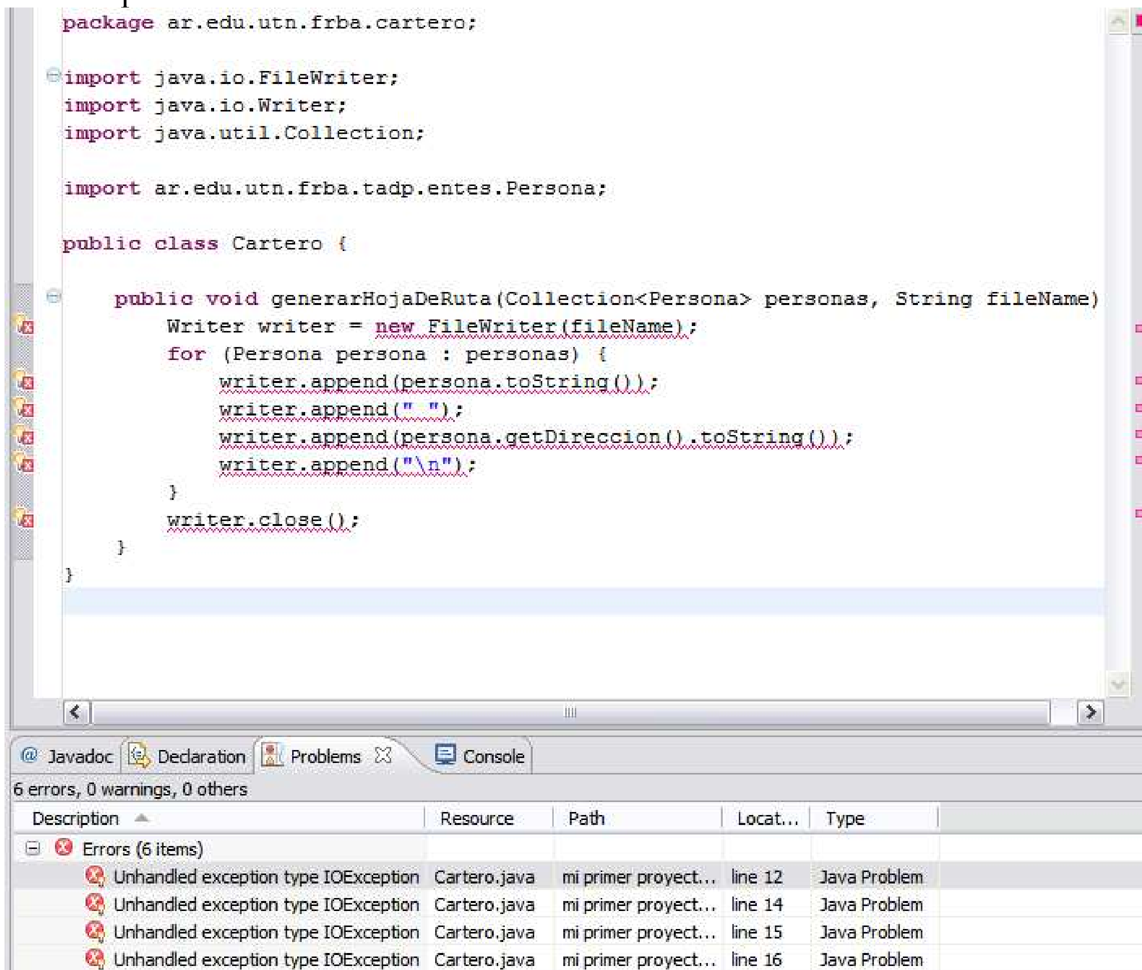
Test.java
```java
package ar.edu.utn.frba.tadp.holamundo;

import java.util.ArrayList;
import java.util.Collection;

import ar.edu.utn.frba.cartero.Cartero;
import ar.edu.utn.frba.tadp.entes.Direccion;
import ar.edu.utn.frba.tadp.entes.Persona;

public class Test {

	public static void main(String[] args) {
		new Cartero().generarHojaDeRuta(getPersonas(), "c:\\direcciones.txt");[6]
	}

	private static Collection<Persona> getPersonas() {
		Collection<Persona> personas = new ArrayList<Persona>();
		personas.add(new Persona("Leo", new Direccion("Esteban de Luca", 
		1322)));
		personas.add(new Persona("Fulano", new Direccion("Echeverría", 978)));
		personas.add(new Persona("Mengano", new Direccion("Colón", 2183)));
		return personas;
	}
}
```

Cartero.java
```java
package ar.edu.utn.frba.cartero;

import java.io.FileWriter;
import java.io.Writer;
import java.util.Collection;

public class Cartero {
```

[6] Al igual que en C, el carácter '\' es de control, y por eso es necesario utilizar '\\'

```
	public void generarHojaDeRuta(Collection<Persona> personas, String fileName)
{
		Writer writer = new FileWriter(fileName);
		for (Persona persona : personas) {
			writer.append(persona.toString());
			writer.append(" ");
			writer.append(persona.getDireccion().toString());
			writer.append("\n");[7]
		}
		writer.close();
	}
}
```

Al escribir esto, el eclipse subraya en rojo varias líneas. Abrimos la vista de problemas Windows -> show views -> problems. Y nos encontramos con:



Se queja de que Writer y FileWriter declara que puede ocurrir una excepción de tipo IOException y nosotros no estamos tomando ninguna decisión con respecto a eso.

Hay tres decisiones que uno puede tomar ante la ocurrencia de una excepción.

1) Que se arregle el que me llamó, yo no hago nada, simplemente declaro que mi método puede tirar excepción y entonces va a subir el problema por el stack trace hasta que alguien la trate o la máquina virtual nos corte la ejecución. Esta aproximación es la misma que usabamos antes cuando el compilador no nos molestaba: si en persona ocurría una NullPointerException se propagaba al main, quien no tomaba acción alguna y entonces la VM cancelaba el programa.

[7] Al igual que en C el caracter '\n' indica salto de línea

2) Manejar el problema, esto ocurre si se sabe que hacer como alternativa. En nuestro caso, para explorar esta opción, diremos que si no puede escribirse el archivo, escriba la información por consola.[8]
3) Una combinación de ambos: el manejo de la excepción es generar una nueva para agregar más información

Propagación de la excepción

Consiste en declarar que nuestro método tira una excepción en la firma del mismo:

```
public class Cartero {

	public void generarHojaDeRuta(Collection<Persona> personas, String fileName)
	throws IOException{
	…
	}
}
```

Al hacer esto Cartero comienza a compilar, sin embargo existe un error en nuestro main, porque ahora estamos usando un método que declara que puede terminar con excepción, y no estamos tomando ninguna acción al respecto.

```
public class Test {

    public static void main(String[] args) {
        new Cartero().generarHojaDeRuta(getPersonas(), "c:\\direcciones.txt");
    }
```

Siguiendo la política de que no sabemos que hacer en caso de error, vamos a hacer que el main declare que puede ocurrir la misma, para que la VM se haga cargo si ocurriera el problema.

```
	public static void main(String[] args) throws IOException {
		new Cartero().generarHojaDeRuta(getPersonas(), "c:\\direcciones.txt");
	}
```

Ahora sí podemos ejecutar el programa y la salida es un archivo en el disco que contiene:

Leo Esteban de Luca 1322
Fulano Echeverría 978
Mengano Colón 2183

Manejo del error

En este caso suponemos que es lo que el método hoja de ruta debe hacer al ocurrir el problema, por lo tanto eliminamos las declaraciones de la excepción en el método. Luego, nos valemos del bloque try/catch. Esta construcción del lenguaje permite decir "intente ejecutar lo que está en el try pero que si ocurre un error de un tipo determinado entonces ejecute el catch". Vea cómo queda el ejemplo luego de realizar cierto emprolijamiento de métodos. El método grabarArchivo aún declara error, porque no sabe que hacer cuando esto ocurre. El que tiene el conocimiento es el método generarHojaDeRuta.

```
public class Cartero {
```

[8] Recuerde que es un ejemplo didáctico

```
	public void generarHojaDeRuta(Collection<Persona> personas, String fileName)
	{
		try {
			this.grabarArchivo(personas, fileName);
		}
		catch (IOException e) {
			this.imprimirPantalla(personas);
		}
	}

	private void imprimirPantalla(Collection<Persona> personas) {
		for (Persona persona : personas) {
			System.out.print(this.item(persona));
		}
	}

	private void grabarArchivo(Collection<Persona> personas, String
	fileName)throws IOException {
		Writer writer = new FileWriter(fileName);
		for (Persona persona : personas) {
			writer.append(this.item(persona));
		}
		writer.close();
	}

	private String item(Persona persona) {
		return persona + " " + persona.getDireccion() + "\n";
	}
}
```

Ahora nuestro programa funciona normalmente y el main no se enteró de que puede ocurrir una IOException. Para ver funcionando a nuestro bloque catch, vamos a forzar que el método grabarArchivo arroje la excepción. Una forma es haciendo que ocurra realmente un problema en el disco, por ejemplo usar una dirección inválida. Pero otra forma más interesante a fines didácticos es generando nosotros un error dentro del método grabar archivo. Esto se hace con la palabra reservada *throw*, que indica que el objeto a continuación es una excepción que tiene que ser lanzada.

```
	private void grabarArchivo(Collection<Persona> personas, String fileName)
			throws IOException {
		Writer writer = new FileWriter(fileName);
		for (Persona persona : personas) {
			writer.append(this.item(persona));
		}
		writer.close();
		throw new IOException();
	}
```

Luego de esta modificación, por consola se puede observar

```
Leo Esteban de Luca 1322
Fulano Echeverría 978
Mengano Colón 2183
```

Hay otro bloque llamado *finally* que viene de la mano con el *try/catch*. El mismo se utiliza cuando hay algo que queremos hacer independientemente de si ocurre un problema o no. En este documento no lo utilizaremos.

Excepciones anidadas

La tercera opción persigue la idea de ir elevando el nivel de abstracción del error. Por lo tanto cuando ocurre un error grabando el archivo, no quiero que el método me diga "tengo un problema de entrada y salida" si no que me diga algo así cómo "No se puede generar la hoja de ruta". Entonces, para lo siguiente vamos a generar una excepción nuestra llamada HojaDeRutaException. Para eso basta con generar una clase y hacerla extender de Exception (si quiero que sea chequeada) o de RuntimeException (si quiero que sea no chequeada).

```
package ar.edu.utn.frba.cartero;

public class HojaDeRutaException extends RuntimeException {

	public HojaDeRutaException(String message, Throwable cause) {
		super(message, cause);
	}

}
```

Las excepciones tienen básicamente dos atributos: un mensaje de error, y una causa que es otra excepción. Así queda nuestro ejemplo del cartero.

```
public class Cartero {

	public void generarHojaDeRuta(Collection<Persona> personas, String fileName)
	{
		try {
			this.grabarArchivo(personas, fileName);
		}
		catch (IOException e) {
			throw new HojaDeRutaException("No se pudo generar la"
			+ "hoja de ruta en la dirección: " + fileName, e);
		}
	}
```

Vamos a forzar el error esta vez usando una dirección inválida de archivo. Y vamos a ver cómo la excepción se propaga hasta la VM y el stack trace que se genera.

```
public static void main(String[] args) {
	new Cartero().generarHojaDeRuta(getPersonas(), "estaDireccionEsInvalida");
}
```

Stack trace:

```
Exception in thread "main" ar.edu.utn.frba.cartero.HojaDeRutaException: No se pudo generar
la hoja de ruta en la dirección: HH:\estaDireccionEsInvalida
	at ar.edu.utn.frba.cartero.Cartero.generarHojaDeRuta(Cartero.java:17)
	at ar.edu.utn.frba.tadp.holamundo.Test.main(Test.java:13)
Caused by: java.io.FileNotFoundException: HH:\estaDireccionEsInvalida (The filename,
directory name, or volume label syntax is incorrect)
	at java.io.FileOutputStream.open(Native Method)
	at java.io.FileOutputStream.<init>(FileOutputStream.java:179)
	at java.io.FileOutputStream.<init>(FileOutputStream.java:70)
	at java.io.FileWriter.<init>(FileWriter.java:46)
	at ar.edu.utn.frba.cartero.Cartero.grabarArchivo(Cartero.java:23)
	at ar.edu.utn.frba.cartero.Cartero.generarHojaDeRuta(Cartero.java:14)
	... 1 more
```

¡No asustarse! El stack trace a simple vista parece feo pero si aprendemos a entenderlo puede ser un gran amigo. ¿Qué es lo que está diciendo?
Primero leemos la excepción sin tener en cuenta los métodos por los que pasó. Esto nos da una idea del problema sin entrar en el detalle de dónde ocurrió:

```
No se pudo generar la hoja de ruta en la dirección: HH:\estaDireccionEsInvalida
…
Caused by: java.io.FileNotFoundException: HH:\estaDireccionEsInvalida (The filename, directory name, or volume label syntax is incorrect)
…
```

¡Ahora se entiende un poco mejor! Por suerte los mensajes de las excepciones fueron escritos a conciencia y dan una buena idea de lo que ocurre: No se pudo generar la hoja de ruta porque la dirección dónde se quiso generar es sintácticamente incorrecta. Y además es tan amable de decirnos cual es la dirección del problema.

Ahora veamos cómo con este stack trace puedo llegar hasta el problema. La primera línea que hace referencia a un método en cada excepción es el lugar dónde se lanzó esa excepción, es decir la línea dónde están los throws:

```
ar.edu.utn.frba.cartero.Cartero.generarHojaDeRuta(Cartero.java:17)
```

es:

```
throw new HojaDeRutaException("No se pudo generar la hoja de ruta en la dirección: " + fileName, e);
```

y `at java.io.FileOutputStream.open(Native Method)`[9]
es… ¡Oh! ¡Diantres!! No tengo disponible el código dónde ocurre realmente la excepción. ¿Qué hago? En ese caso empiezo a bajar en la sección del stack trace correspondiente a esta excepción hasta encontrar una línea que tenga que ver con lo que yo escribí, es decir, una clase mía y no de terceros porque ahí no puedo tomar ninguna acción.
En nuestro caso la primera línea que tiene sentido es:

```
at ar.edu.utn.frba.cartero.Cartero.grabarArchivo(Cartero.java:23)
```

correspondiente a:

```
Writer writer = new FileWriter(fileName);
```

Bien, de estas dos líneas se deduce que, la excepción de no poder generar la hoja de ruta es causa de que me explota la aplicación al instanciar un FileWriter. Por el mensaje de error me doy cuenta que el problema es que ese *fileName* está mal formado, por lo tanto tengo que rastrear en el stack hasta ver de dónde sale este parámetro:

```
at ar.edu.utn.frba.cartero.Cartero.generarHojaDeRuta(Cartero.java:14)
```

me lleva a :

```
this.grabarArchivo(personas, fileName);
```

Acá también *fileName* es un parámetro, por lo tanto aún no tengo el punto dónde arreglar. Cómo es la última línea de esta excepción, sigo navegando en el bloque de la primera, siguiendo la pista de dónde viene el *fileName*:

```
at ar.edu.utn.frba.tadp.holamundo.Test.main(Test.java:13)
new Cartero().generarHojaDeRuta(getPersonas(), "HH:\\estaDireccionEsInvalida");
```

Finalmente encontré el lugar dónde se genera el valor de *fileName,* causa de todos los males. Cambio este valor por un valor válido y nuevamente nuestro programa funciona correctamente.

[9] Un método nativo no está escrito en java, si no que la máquina virtual hace un llamado al sistema operativo.