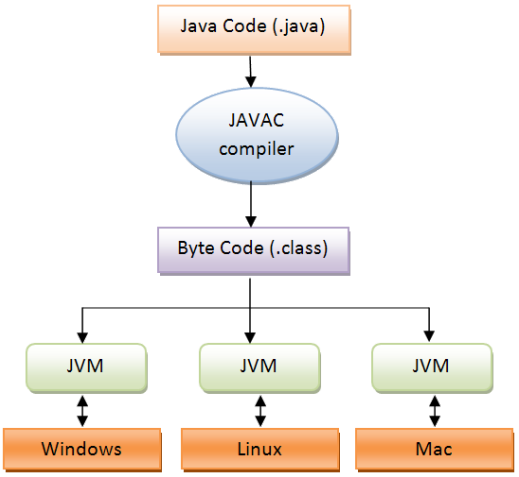
Arrancando con Java

El objetivo de este documento es brindar a cualquiera que conozca el paradigma de objetos una vista rápida de las características y de las preguntas más comunes que suelen surgir al momento de desarrollar en este lenguaje. Y una serie de links para profundizar los temas una vez dominados estos conceptos principales.

¿Cómo funciona?

Cuando programamos en Java, escribimos el código fuente en archivos con extensión .java (escribo las clases, interfaces y demás elementos de nuestro software), eso se compila a bytecode (para esto necesitamos instalar la JDK que es la que trae el compilador) en unos archivos .class que puede correr la Java Virtual Machine (JVM).



¿Qué necesito para empezar a programar?

- **JDK**: Java Development Kit (compilador, debugger, etc..)
- **Eclipse**: Entorno de desarrollo Integrado

Podes encontrar un instructivo de como instalar las herramientas en:
http://wiki.ugbar.org/wiki/articles/preparacion-de-un-entorno-de-desarrollo-java.html
(Apartados del 1 al 5)

Importante: ¿Cómo sé si instale correctamente Java?

En la terminal (linux / windows) ejecuto el comando
java -version

Debería mostrar algo similar a lo siguiente:
java version "1.8.0_161"
.....

¿Y el lenguaje como es?

Tipado

Estático y nominal ¿Qué es?

Abstracciones principales

Clases

```
public class Pajaro {
  private int energia; // atributo
  // métodos
  public void volar() {
    this.energia = this.energia - 10;
  }

  public void comer() {
    this.energia = this.energia + 300;
  }
}
```

- ¿Qué es ese *int*?
- Bueno, Java no es muy puro que digamos con respecto a que todo es un objeto. Existen los tipos de datos primitivos (int, float, double, etc) que no entienden mensajes ni pueden referenciar a null.
Pero.. Se dieron cuenta que estaría bueno tener sus versiones en objetos, entonces crearon unas clases *"wrappers"* de estos tipos de datos primitivos (Integer, Float, etc.).

Lo interesante de los Integer, Float, etc es que:

- No entienden mensajes. ¡Buuuuu!
- Pero podemos utilizarlos donde se esperaría un objeto (por ejemplo, podemos agregarlos a una colección)
- Las referencias tipadas de esta forma admiten referenciar a null.

- ¿Cómo uso los Integer por ejemplo?

```
Integer numero = new Integer(10); //Fácil, instancio un integer con el valor de 10
Integer otroNumero = 20; // Asigno un int a un Integer, Java lo resuelve por mi
Integer sinNumero = null; // Referencio a la nada.
```

- ¿Qué es ese `this`.*algo*?

`this` es una referencia al objeto actual, entonces en ese código por ejemplo cuando hace `this.energia` está haciendo referencia al atributo energía del objeto actual.

Si un pájaro cuando come vuela, entonces una posible implementación de comer podría ser:

```
public void comer() {
  this.energia = this.energia + 300;
  this.volar();
}
```

- ¿Qué onda esto de los public/private?

Bueno… rápidamente Java tiene modificadores que indican quién puede tener acceso a las clases/atributos/métodos, los más usados son:

	Misma clase	**Clase y subclases**	**Otros**
private	Accede	No accede	No accede
protected	Accede	Accede	No accede
public	Accede	Accede	Accede

Che, todo bien, pero mi sistema en objetos, son objetos que interactúan entre sí **¿cómo consigo un objeto?** Fácil, le hago un `new`.

```
new Pajaro()
```

- Y si pajaro tiene un nombre y quiero crearlo directamente con el nombre. ¿Cómo hago?
Fácil, podés redefinir la forma en que se crea un objeto, creando un constructor de más parámetros o que tenga cierto comportamiento.
Para cumplir lo que me pedís, podes hacer:

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

Y lo usas como:

```
Persona unaPersona = new Persona("Pepita");
```

- Che, ¿pero no podría haber hecho un método de clase que instancie a la Persona y luego le setee el nombre? Hablando de eso ¿Cómo hago un método de clase?
Tanto un método de clase como una variable de clase en lo único que se diferencian con las de instancia, es en que tienen el modificador ***static.***
Un ejemplo sería:

Variable de clase

```
private static String FOO = "Bar"
```

Método de clase

```
private static Persona deNombre(String nombre) {
  Persona persona = new Persona();
  persona.setNombre(nombre);
```

```
  return persona;
}
Persona p = Persona.deNombre("Bar")
```

Esquema de Herencia

Utiliza herencia simple.

```
public class Zorzal extends Pajaro {
  public void cantar() {
    //Implementación de cantar
  }
}
```

- ¿Y tengo clases abstractas?
- Sii!! Se definen poniendo el modificador `abstract`, un ejemplo sería:

```
public abstract class Pajaro {
  public abstract void cantar();
  public int cantidadDePicos() {
   return 1;
  }
}
```

El `abstract class` indica que la clase es abstracta.
Agregar `abstract` a un método indica que alguna de las subclases va a tener que definirlo.

- Tengo que redefinir un método y reusar el método del padre, ¿Qué hago?
- En vez de usar `this` podes usar `super`.

Interfaz/contrato

Son parte del código (consecuencia de su tipado) y definen los mensajes que tiene que entender la clase que implemente dicha interfaz.

```
public interface Volador {
  public void volar();
}

public class Pajaro implements Volador {
  public void volar() {
    //Implementación
  }
}
```

- ¿Cuántas interfaces puede implementar una clase?
- Las que quieras!

Collections

Java tiene principalmente las siguientes abstracciones:

Collection

Es la interfaz más general para las colecciones, sirve para modelar los mensajes que debería cumplir todos las implementaciones de colecciones, sin importar si tiene orden o repetidos, por eso entiende los mensajes más comunes de colecciones, algunos son: ***add(..), remove(..), size()***, etc..
Link a la interfaz: https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

List

Es una interfaz que extiende la interfaz ***Collection***, agregando todos los mensajes que tienen que ver a colecciones que tienen un orden (indexado), por ejemplo: ***get(int index)***, ***remove(int index), sublist(int fromIndex, int toIndex)***, etc.
Link a la interfaz: https://docs.oracle.com/javase/8/docs/api/java/util/List.html
Implementaciones más utilizadas: LinkedList, ArrayList

También hay una forma de obtener una lista de elementos de forma bien sencilla, que es haciendo:

```
List myLista = Arrays.asList(objeto1, objeto2, etc...)
```

La única consideración a tener en cuenta sobre la lista que retorna ***Arrays.asList(..)*** es que es inmutable.

Set

Esta interfaz no agrega nuevos mensajes que deberían cumplir las colecciones que la implementen, pero sirve para indicar que las implementaciones de dicha interfaz no van a tener repetidos. La decisión de si va a poner los elementos en la lista la va a hacer en base a la equivalencia de objetos.
Link a la interfaz: https://docs.oracle.com/javase/8/docs/api/java/util/Set.html
Algunas implementaciones: HashSet, TreeSet

Map

Esta interfaz no extiende de Collection, porque lo que modela son los Dictionary/Map/Array-Asociativo, lo llaman de muchos nombres.
Básicamente es una estructura que guarda claves asociadas a valores.
Por ejemplo le podemos decir: ***instanciaDeMap.put(key, value)*** y obtener el valor en base a la clave, haciendo: ***instanciaDeMap.get(key)***
Link a la interfaz: https://docs.oracle.com/javase/8/docs/api/java/util/Map.html
Algunas implementaciones: HashMap, TreeMap

- Che, instancié una lista, le agregué una instancia de Persona y quiero obtenerla y me devuelve un Object ¿qué onda, tengo que castearlo? Este código fue el que escribí:

```
List personas = new ArrayList();
personas.add(new Persona());
Persona elPrimero = personas.get(0); //Aca no me compila
```

- No, no casteás!! Si te fijás en la documentación de ***get(..)*** lo que retorna no es un Object, sino un **E**, y si te fijás arriba de todo en la documentación dice:

Type Parameters:

E - the type of elements in this list

- ¿Y qué significa que E sea un parámetro de tipo?
- Significa que E es una especie de variable que va a representar un tipo y que debería ser definida cuando declares la lista que vas a usar. En este caso como no hay nada definido asume que el tipo para E es Object.

Entonces como vos queres que ese "***E"*** sea una persona, porque la lista que vos estas usando es una lista de personas, deberías hacer:

```
List<Persona> personas = new ArrayList<>();
personas.add(new Persona());
Persona elPrimero = personas.get(0);
```

Cuando haces la declaración de la lista de esa forma, lo que le decís es que todos los **"E"** ahora son ***Persona***, y listo :)

A esto se lo conoce como Generics, si te quedaste con ganas de saber qué más se puede hacer con esto, te dejo un link:
https://docs.oracle.com/javase/tutorial/extra/generics/intro.html

- Estuve viendo la interfaz de Collection, no encuentro el filter/map, etc ¿Hago un for?
- Noo!!! Desde Java8, las colecciones ya soportan esos mensajes, pero por una decisión de diseño, decidieron agregarlos a otro objeto que se llama ***Stream*** y se lo podes pedir a las colecciones mandándole el mensaje ***stream()***

- Daleee, mostrame un ejemplo!!!
- Ok, esta bien.. un ejemplo sería:
```
personas.stream().filter(persona -> persona.esMayorDeEdad())
```
Tené en cuenta cuenta que lo que devuelve el ***filter(..)*** sigue siendo un Stream, entonces para obtener de nuevo una lista tenes que usar el mensaje ***collect(..)***.
Por ejemplo para después de filtrar pasarlo de nuevo a una lista, se puede hacer:

```
personas
  .stream()
  .filter(persona -> persona.esMayorDeEdad())
  .map(persona -> persona.getNombre())
  .collect(Collectors.toList())
```

Podés encontrar más mensajes:
- En la guía de lenguajes: link
- En la Api de Stream: link

-¿Entonces además hay lambdas?
- Sii!!! Desde Java 8 se incorporan las lambdas, acá tenes un apunte copado sobre cómo se usan: link

Annotations

Sirven para agregar metadata a los elementos de una clase (definición de la clase/atributo/métodos), un ejemplo clásico es cuando se indica que se sobrescribe un método:

```
public class Zorzal extends Pajaro {
    @Override
    public void cantar(){
      // implementación
    }
}
```

Pueden leer un poco más de información acá: link

Enums

Sí, tiene enums, un ejemplo de definición sería:

```
public enum Dia {
    LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO
}
```

- Hay más… pueden definir comportamiento!!! :)

```
public enum PuntoCardinal {
    public abstract PuntoCardinal getOpuesto();
    NORTE {
        @Override
        public PuntoCardinal getOpuesto() { return SUR; }
    },
    SUR {
        @Override
        public PuntoCardinal getOpuesto() { return NORTE; }
    },
    ESTE {
        @Override
        public Direction getOpuesto() { return OESTE; }
    },
    OESTE {
        @Override
        public PuntoCardinal getOpuesto() { return ESTE; }
    };
}
```

Cada **PuntoCardinal** es un valor del enumerado, pero además ese valor entiende un mensaje. Por ejemplo, si quisiéramos saber el opuesto del Norte, en nuestro código hariamos:

```
PuntoCardinal unPunto = PuntoCardinal.NORTE;
unPunto.getOpuesto();
```

Excepciones

Tiene dos tipos: chequeadas y no chequeadas, las que más vamos a usar son las no chequeadas.

¿Qué es que sea chequeada/no chequeada?

- Las chequeadas heredan de Exception y en las firmas de los métodos que van a dejar propagar la excepción (sea porque la lanzan o no la atrapan) deben indicar qué puede lanzar una excepción de ese tipo.
  Ejemplo de la firma de un método que deja propagar una excepción chequeda llamada SaldoInsuficienteException

  ```
  public void comprar() throws SaldoInsuficienteException
  ```

- Las no chequeadas heredan de RuntimeException y no es obligatorio definir en la firma que el método las puede lanzar.

¿Cómo hago para crear una excepción no chequeda?

```
public class SaldoInsuficienteException extends RuntimeException {
  public SaldoInsuficienteException() {
    super();
  }
  public SaldoInsuficienteException(String message) {
    super(message);
  }
}
```

- ¿Cómo hago para lanzar una excepción?
Si cree una excepción no chequeada SaldoInsuficienteException, se lanzaría de la siguiente forma:

```
throw new SaldoInsuficienteException()
```

- ¿Cómo hago para atrapar una excepción?

```
public void comprar() {
  try {
    // código que puede lanzar una SaldoInsuficienteException
  } catch (SaldoInsuficienteException e) {
    // Hago algo, por eso la atrape ;)
  } finally {
    // Se ejecuta siempre
  }
}
```

Para más información para saber como lanzar/atrapar excepciones y como crear las tuyas propias, te recomendamos leer este apunte: link.

Identidad vs Igualdad

Se puede comparar si un objeto es idéntico a otro haciendo: `objeto1 == objeto2`
Se puede comparar si un objeto es igual a otro haciendo: `objeto1.equals(objeto2)`

- ¿Y qué hago si no quiere usar el comportamiento por defecto de la igualdad?
- Redefinis el *equals(..)*, pero no solo el *equals(..)* sino el *hashCode(..)* también.

- **¿Por qué?**
- el hashCode es un método que se usa para optimizar la búsqueda por ***equals(..)*** y se basa en lo que es una función de hash que recibe un elemento y retorna un número que lo represente. El chiste de esto es que la función de hash no se base en todos los campos en los que se va a basar el ***equals(..)***, así es más liviana.

Ejemplo: si el equals de una persona se basa en el nombre, la edad, el dni y el sexo, el hashcode podría basarse solo en el dni.

- Pero si se basa en menos cosas, que pasa cuando dos objetos tienen el mismo hashCode ¿no daría que son iguales?
- No, porque si los ***hashCode(..)*** le dieran iguales entonces va a utilizar el ***equals(..)***. En el caso que el ***hashCode(..)*** sea distintos ni siquiera va a usar el ***equals(..)***.

*Es importante recalcar que existe un contrato entre **equals()** y **hashcode()** que debe ser respetado. Si dos objetos son equivalentes, entonces tienen el mismo hashcode. Ahora bien, si dos objetos tienen el mismo hashcode, no necesariamente son equivalentes.*

- Voy entendiendo, ¿y cómo lo defino?
- Momento.. ¿estás seguro de que lo queres hacer? ¿Realmente es necesario? Mirá que raras veces lo queres definir.

- Si, ya lo pensé mucho, lo consulte y estoy seguro que sirve
- Bueno, entonces podés usar una biblioteca que se llama Apache-Commons-Lang que te da dos clases para eso EqualsBuilder y HashCodeBuilder

JUNIT

JUnit es el framework por defecto para testear nuestros objetos en Java. Para utilizarlo solamente tenemos que añadir la librería a nuestro proyecto, y esto se puede hacer muy fácilmente agregando la dependencia en maven.
Ejemplo de una clase de Test:

```
public class ClienteTest {
  private Cliente cliente;

  @Before
  public void init() {
    cliente = new Cliente("juan");
    cliente.setDeuda(0);
  }

  @Test
  public void noDeberiaSerMoroso() {
    Assert.assertFalse(cliente.esMoroso());
  }

  @Test
  public void deberiaSerMoroso() {
    cliente.setDeuda(10);
    Assert.assertTrue(cliente.esMoroso());
  }

  @Test(expected = NoPuedeComprarPorMorosoException.class)
  public void noDeberiaPoderComprarSiEsMoroso() {
    cliente.setDeuda(10);
    Assert.assertTrue(cliente.comprar(100));
  }

}
```

Estos test pueden ser corridos desde nuestro IDE preferido o incluso de herramientas como maven/gradle, etc.

Las anotaciones básicas para nuestros test son:
`@Before`
El método que tenga esta annotation va a ejecutarse antes de cada test.

`@After`
El método que tenga esta annotation va a ejecutarse después de cada test.

`@Test`
El método que tenga esta annotation será considerado un test y será satisfactorio si pasa todas las aserciones.

```
@Test(expected = Clase.class)
```

El método que tenga esta annotation será considerado un test y es correcto si lanza una excepción de tipo Clase.

- ¿ Que *asserts* hay disponibles?
Muchos! En este link, podrás encontrar un listado de los existentes en Junit
http://junit.sourceforge.net/javadoc/org/junit/Assert.html

Demo de como trabajar con el IDE

Acá tenes un video para ver cómo hacer esto en el IDE: link