Capítulo 6

Strings – Entrada/Salida – Formateo y parseo

Revista Digital - Hecha por estudiantes para estudiantes

Bienvenidos a una nueva edición de JavaWorld

Le damos la bienvenida nuevamente a nuestros amigos de Valor Creativo "¡Leo, bienvenido a la vida real nuevamente!".

En esta entrega nos estará hablando un poco sobre las expectativas en la industria de la informática. Muy útil para todos aquellos que se encuentren a un paso o dentro del ámbito de emprendedores.

Volviendo a la certificación, empezaremos a adentrarnos en la API de Java especializada para la entrada y salida de datos. Si, ahora podrás leer ese .txt con todos los teléfonos que sacaste el sábado y pasarlo a tu agenda desarrollada en Java.

Aprenderemos a persistir nuestros objetos con el modelo de serialización que nos propone de manera nativa Java, una utilidad en principio para no depender de una base de datos, pero la cual tendrá un valor agregado cuando trabajemos enviando mensajes a través de internet con este mismo método ;)

Ya estamos en la cuenta regresiva en las ediciones de JavaWorld, quedan solamente 4 capítulos. Aprovechen a sacarse sus dudas, ya sea enviando comentarios o por email (los datos pueden obtenerlos en los blogs).

Aprovechamos la oportunidad para agradecer el apoyo recibido por la comunidad y la buena onda para seguir adelante con este proyecto.

Gracias a todos.
El staff de revista JavaWorld

La clase String

Como hemos mencionado en capítulos anteriores, la clase String no puede ser modificada. Entonces, ¿cómo es que podemos asociar diferentes cadenas a una misma variable de tipo String?.

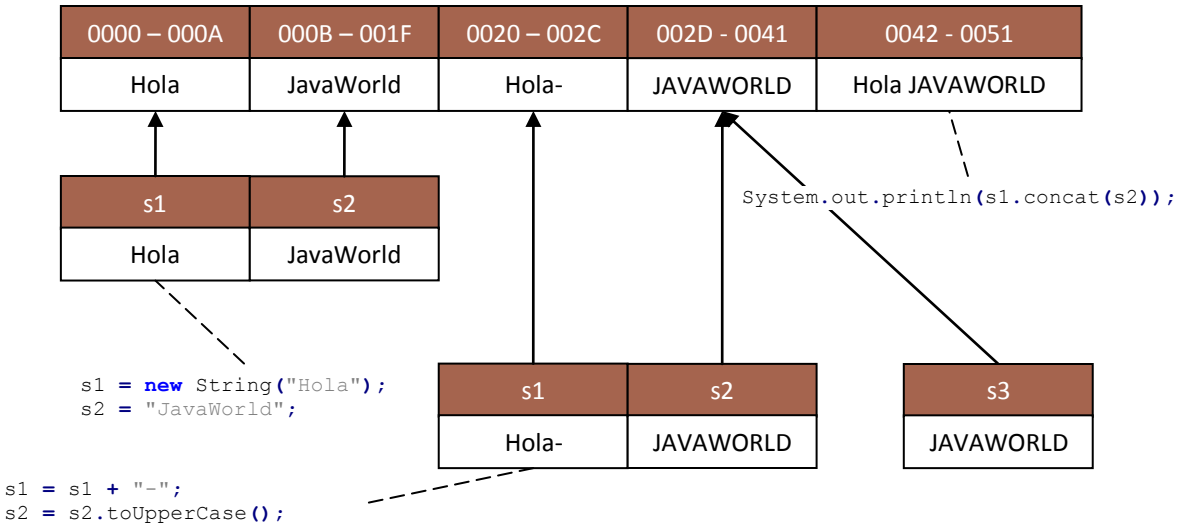
String es un objeto inmutable

Inmutable… Que palabra. ¿Y qué quiere decir?
Simple, exactamente que un objeto String no puede cambiar su valor, pero en ningún momento se menciona que una variable String no pueda apuntar a otro objeto.
Vamos a ver unos ejemplos para entenderlo mejor:

```
public class PruebasString_001 {
  static public void main(String[] args) {
    String s1 = new String("Hola");
    String s2 = "JavaWorld";

    System.out.println(s1);
    System.out.println(s2);

    s1 = s1 + "-";
    s2 = s2.toUpperCase(); //Pasa a mayúsculas la cadena

    System.out.println(s1);
    System.out.println(s2);

    System.out.println(s1.concat(s2)); //concatena dos cadenas. Equivale a s1 + s2

    String s3 = s2;
  }
}
```

```
Hola
JavaWorld
Hola-
JAVAWORLD
Hola-JAVAWORLD
```

Ahora puede que se estén preguntando “¿pero no dijo que un String era inmutable?”. Si, así es, pero reitero, nadie dice que una variable de tipo String no pueda referenciar a otro objeto.
Veamos de manera gráfica el mapeo en memoria:



Cada carácter dentro de un String se encuentra en formato UFT-16. Las posiciones en memoria mostradas anteriormente son solo para ejemplificar. El comportamiento del almacenamiento es dirigido por la JVM en conjunto con el Sistema Operativo.

Cuando se inicializan las variables, se les asignan los valores “Hola” y “JavaWorld”, aquí es cuando creamos nuestros dos primeros objetos de tipo String.
Luego, a s1 le concatenamos “-“ y guardamos el resultado en la misma variable, lo que produce que se genere un nuevo objeto, y actualizando la referencia para que apunte a este.
A s2 lo transformamos en mayúsculas. Nuevamente se genera otro objeto, y almacenamos la referencia en s2.
Y por último, generamos una salida en pantalla de s1 + s2. En este caso, también se genera un nuevo objeto, con la salvedad de que la referencia no la captura nadie. Una vez que se mostró por pantalla esta variable es candidata a ser liberada por el garbage collector.
Como podrán haber notado, en ningún momento se modificó ningún objeto que estuviera en memoria, pero si cambiaron las referencias de s1 y s2.
Por último, cuando creamos un nuevo objeto (s3), y le asignamos el valor de s2, no se crea un nuevo objeto, se asigna la misma posición de memoria.

Cada vez que generen una nueva cadena, se genera un nuevo objeto String, siempre. La única excepción es cuando asignamos el valor de una variable String a otra. En este caso, las dos apuntan a la misma posición de memoria.

String y la memoria

Dado que una cadena es algo tan común, y para poder optimizar la memoria, la JVM reserva un área especial de la memoria denominada “String constant pool” (banco de constantes String). Cuando el compilador encuentra un literal de tipo String, verifica en el banco si existe otra cadena idéntica. Si se encuentra una, directamente se referencia a esta, sino, se crea un nuevo objeto String.

La clase String contiene el modificador de no acceso **final**. Esto quiere decir que no es posible sobre escribir la clase.

Creación de cadenas

Existe una diferencia sutil entre crear un String mediante un literal y un objeto:

```
String s1 = "JavaWorld"
```

Se crea una variable de referencia (s1) y un objeto String, el cual es almacenado en el banco de constantes String y asociado a s1.

```
String s2 = new String("JavaWorld")
```

Se crea una variable de referencia (s2). Como utilizamos la palabra clave new, se crea un objeto String fuera del banco de constantes String y s2 referenciará a este. Luego, el literal "JavaWorld" es insertado en el banco.

Métodos de la clase String

A continuación se nombran los métodos más utilizados de la clase String:

Tipo de valor de retorno	Método	Descripción
char	charAt(int indice)	Devuelve el carácter ubicado en el índice especificado.
String	concat(String str)	Agrega un String al final de otro (equivalente a utilizar el +).
boolean	equalsIgnoreCase(String otroString)	Indica si dos cadenas son iguales, sin tener en cuenta mayúsculas.
int	length()	Devuelve la cantidad de caracteres que contiene la cadena.
String	replace(char caracterViejo, char caracterNuevo)	Reemplaza cada caracterViejo con el caracterNuevo.
String	substring(int inicioIndice, int finIndice)	Devuelve una nueva cadena con los caracteres comprendidos desde inicioIndice hasta finIndice.
String	toLowerCase()	Convierte los caracteres en minúsculas.
String	toString()	Devuelve el valor de un String (dado que es un String, se devuelve a sí mismo).
String	toUpperCase()	Convierte los caracteres en mayúsculas.
String	trim()	Elimina los espacios en blanco al comienzo y final de la cadena.

Para más información pueden consultar la API de Java
"http://java.sun.com/j2se/1.4.2/docs/api/java/lang/String.html"

Hay que destacar que el tipo **String** tiene un **método** length(), y los **arrays** tienen un **parámetro** length.

StringBuffer y StringBuilder

Cuando vamos a utilizar muchas cadenas podemos utilizar las clases java.lang.StringBuffer o java.lang.StringBuilder. A diferencia de String, con la cual si generaramos varias cadenas, se irían almacenando en el banco de constantes String, estas nuevas clases no se almacenan en este último, sino que aprovechan el mismo espacio de memoria.

StringBuffer vs. StringBuilder

La clase StringBuffer fue incorporada a partir de la versión 5 de Java. Posee exactamente la misma API que StringBuilder, salvo que esta última no es thread-safe (multihilo con seguridad). En otras palabras, los métodos no se encuentran sincronizados.
Java recomienda siempre que sea posible, utilizar StringBuilder, dado que es mucho más veloz que su hermana StringBuffer. Solo cuando necesitemos que sea thread-safe, utilizaremos StringBuffer.

StringBuilder no es thread-safe.
StringBuffer si es thread-safe.
Ambas comparten la misma API.

Utilizando StringBuffer y StringBuilder

```
String s1 = "JavaWorld";       //Se añade un String al banco de constantes String
s1 += ". Vamos por más";       //Se genera un nuevo String, y la referencia al anterior se pierde

//Se crea un StringBuffer (no va al banco de constantes String)
StrungBuffer s1 = new StringBuffer("JavaWorld");
s1.append(". Vamos por más");//Se modifica el mismo objeto (no se crea un nuevo String)
```

Métodos importantes en las clases StringBuffer y StringBuilder

Valor de retorno	Método	Sobrecarga	Descripción	Ejemplo
StringBuffer/ StringBuilder	append(String s)	SI	Concatena la cadena s a la cadena dentro del objeto (no es necesario atrapar la referencia para que el cambio se realice).	`StringBuffer s = new StringBuffer("Quiero ser"); s.append(" un SCJP!"); //Quiero ser un SCJP!`
StringBuffer/ StringBuilder	delete(int inicio, int fin)	NO	Elimina la subcadena dentro de la cadena del objeto desde la posición inicio a la posición fin.	`StringBuffer s = new StringBuffer("Quiero ser un SCJP!"); s.delete(10, 13); //Quiero ser SCJP!`
StringBuffer/ StringBuilder	insert(int indice, String s)	SI	Inserta una nueva cadena (s) a partir de la posición indicada por índice dentro de la cadena del objeto.	`StringBuffer s = new StringBuffer("Quiero SCJP"); s.insert(7, "ser un "); //Quiero ser un SCJP`
StringBuffer/ StringBuilder	reverse()	NO	Invierte la cadena.	`StringBuffer s = new StringBuffer("PJCS"); s.revert(); //SCJP`
String	toString()	NO	Devuelve un String con el valor del objeto.	`StringBuffer s = new StringBuffer("SCJP"); String s1 = s.toString(); //SCJP`

Es posible que se encuentren en el examen con métodos encadenados, estos simplemente se resuelven de derecha a izquierda, y el resultado de un método, es el objeto que utiliza el segundo.

Un ejemplo de equivalencias:

```
StringBuffer s = new StringBuffer("Un SCJP");
s.reverse().delete(4, 7).append("... un que?").insert(0, "Quiero ser un ");
System.out.println(s); //Quiero ser un PJCS... un que?

StringBuffer s2 = new StringBuffer("Un SCJP");
s2.reverse();
s2.delete(4, 7);
s2.append("... un que?");
s2.insert(0, "Quiero ser un ");
System.out.println(s2); //Quiero ser un PJCS... un que?
```

StringBuffer y StringBuilder, al igual que String, no requieren ningún **import**.

Navegación de archivos e I/O

El package java.io contiene una cantidad interesante de clases, pero para el examen, solo necesitaremos conocer algunas de ellas. Solo se cubren los temas referidos a la entrada/salida de caracteres y serialización.

Cuando se utilice cualquier clase del package java.io, es necesario utilizar al comienzo del código la sentencia import, o hacer referencia al nombre completo de la clase:

```
//Con el import
import java.io.File;
//... Declaración de la clase y método main
File file = new File("archivo.txt");

//Sin el import
java.io.File file = new java.io.File("archivo.txt");
```

Descripción de las clases java.io para el examen

Clase	Descripción
File	Según la API "es una representación abstracta de rutas de directorios y archivos". Aunque parezca extraño, esta clase no sirve para leer o escribir un archivo, sino que es utilizada a nivel de saber si existe un archivo, crear o eliminar un archivo, crear un nuevo directorio.
FileReader	Esta clase es utilizada para leer archivos de texto. El método read() permite un acceso a bajo nivel, obteniendo un carácter, una determinada cantidad de estos, o todos los caracteres del archivo. Esta clase generalmente es envuelta (wrapped) por otra clase, como un BufferedReader.
BufferedReader	Esta clase se utiliza para wrappear las clases de bajo nivel, como FileReader para volverlas más eficientes. En vez de leer carácter por carácter, lee una cantidad de estos y lo almacena en un buffer, cuando nosotros le pedimos el siguiente carácter, lo obtiene directamente del buffer y no de un archivo (recuerden que en Sistemas Operativos, existe un mínimo de transferencia, a esta porción se la denomina bloque, de manera que lo más óptimo sería transferir la información en un tamaño que sea un múltiplo de dicho bloque). También añade el método readLine() que permite leer una línea entera (hasta el próximo carácter de salto de línea, o final de archivo, lo que se encuentre primero).
FileWriter	Esta clase es utilizada para escribir archivos de texto. El método write() permite escribir caracteres o un String dentro de un archivo. Al igual que FileReader, esta clase es generalmente wrapped en otras clases como BufferedWriter.
BufferedWriter	Esta clase se utiliza para wrappear las clases de bajo nivel, como FileWriter para volverlas más eficientes. Esta clase escribirá de a varios caracteres para reducir los tiempos de espera por almacenamiento en disco. También provee un método llamado newLine(), el cual genera el carácter de salto de línea correspondiente a la plataforma en donde se está corriendo la aplicación.
PrintWriter	Esta clase ha sido mejorada considerablemente a partir de la versión 5 de Java. Permite realizar acciones como format(), printf(), y append().
Console	Esta clase ha sido incorporada a partir de la versión 6 de Java. Provee métodos para leer desde consola y escribir en ella.

Api de las clases java.io para el examen

Clase	Extends	Constructores	Métodos
File	Object	(File f, String s) (String s) (String s, String s)	createNewFile() : boolean delete() : boolean exists() : boolean isDirectory() : boolean isFile() : boolean list() : String[] mkdir() : boolean renameTo(File destino) : boolean
FileReader	Writer	(File f) (String s)	read() : int
BufferedReader	Reader	(Reader r)	read() : int readLine() : String
FileWriter	Writer	(File f) (String s)	close() : void flush() : void write(char[] cbuf) : void write(String cbuf) : void
BufferedWriter	Reader	(Writer wr)	close() : void flush() : void write(char[] cbuf) : void write(String cbuf) : void newLine() : void

PrintWriter	Writer	(File f) //Desde Java 5
		(String s) //Desde Java 5
		(OutputStream ou)
		(Writer wr)
			close() : void
			flush() : void
			format()
			printf()
			print()
			println()
			write()

Console			

Dentro del **package** java.io encontramos dos grandes grupos de clases, aquellas que contienen las palabras InputStream o OutputStream, y aquellas otras que contienen Read o Write.

Las primeras son utilizadas a nivel de bytes, las últimas a nivel de caracteres.

Creación de archivos a través de la clase File

Estos objetos son utilizados para representar archivos (pero no su contenido) o directorios.

```
import java.io.File; //También es válido escribir: import java.io.*

public class PruebasFiles_001 {
  static public void main(String[] args) {
    File file = new File("archivo_001.txt");
  }
}
```

Vamos a empezar con un poco de código:

Invito al lector si tiene una computadora con la JDK (no JRE) a mano a compilar y ejecutar este código (llamen a la clase PruebasFiles_001.java).

Si realizaste la tarea anterior, puede que hayas ido corriendo al directorio a buscar tu nuevo archivo recién creado… pero… ¿dónde está?, no lo encuentro por ninguna parte.

El código anterior no crea ningún archivo, sino que crea un objeto el cual tiene establecido un path (ruta), o nombre de archivo.

Pero, con dicha ruta, podemos llegar a crear un archivo. Veamos:

```
import java.io.File; //También es válido escribir: import java.io.*
import java.io.IOException; //Si utilizaste import.java.io.*, no es necesaria esta
línea

public class PruebasFiles_001 {
  static public void main(String[] args) {
    try {
      File file = new File("archivo_001.txt");

      boolean  esArchivoNuevo = false;

      //El método exists() devuelve un boolean indicando si existe o no el archivo
      System.out.println("Existe el archivo: " + file.exists());
      esArchivoNuevo = file.createNewFile();
      System.out.println("Es nuevo el archivo: " + esArchivoNuevo);
      System.out.println("Existe el archivo: " + file.exists());
    } catch(IOException ex) {
      System.out.println("Ocurrio un error: " + ex);
    }
  }
}
```

Cuando ejecutemos el programa por primera vez recibiremos lo siguiente:

```
Existe el archivo: false
Es nuevo el archivo: true
Existe el archivo: true
```

En la segunda ejecución recibiremos una respuesta diferente:

```
Existe el archivo: true
Es nuevo el archivo: false
Existe el archivo: true
```

En la primer ejecución, cuando verificamos si existe el archivo, devuelve false, luego lo creamos, y al verificar nuevamente por su existencia, indica que sí existe.

En la segunda ejecución, cuando verificamos si existe el archivo, nos encontramos con un true (lo creamos pero nunca lo destruimos). Cuando intentamos volver a crearlo, devuelve un false, esto es porque el método createNewFile() intenta crear el archivo, si pudo crearlo devuelve true, sino devuelve false, en este caso, como ya existía, no sobreescribe, sino que no produce ninguna acción. Y al verificar nuevamente por el archivo, sigue estando (que sorpresa, no?!).

Utilizando FileWriter y FileReader

En la práctica, es muy poco común utilizar estas clases sin wrappers, pero de todas maneras, veamos un ejemplo.

```
import java.io.IOException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.File;

public class PruebasFiles_002 {
  static public void main(String[] args) {
    char[] bufferLectura = new char[50];
    int tamanio = 0;

    try {
      File file = new File("archivo.txt");
      FileWriter fw = new FileWriter(file);

      fw.write("Quiero ser un SCJP");
      fw.flush();
      fw.close();

      FileReader fr = new FileReader(file);
      tamanio = fr.read(bufferLectura);
      System.out.println("Tamaño del archivo: " + tamanio + " bytes.");
      for(char c : bufferLectura) {
        System.out.print(c);
      }
      fr.close();
    } catch(IOException ex) {
      System.out.println("Ocurrio un error: " + ex);
    }
  }
}
```

```
Tamaño del archivo: 18 bytes.
Quiero ser un SCJP
```

Si abrimos el archivo, nos encontraremos con esta misma línea “Quiero ser un SCJP”.

Combinando clases de entrada/salida

Todo el package de java.io fue diseñado para utilizarse en combinación entre clases. Combinando las clases se lo suele llamar: wrapping, o decorando.

Para aquellos que se encuentren familiarizados con los patrones de diseño, les será más fácil entender que todo el modelo de clases de java.io se basa en el patrón Decorador.

Para aquellos que no tengan ni idea de que estoy hablando, les dejo el link al blog de un profesor de la cátedra de Metodologías de Sistemas, en donde explica dicho patrón.

http://metodologiasdesistemas.blogspot.com/search/label/decorador

No es necesario que conozcas el patrón (no a fines de rendir la certificación SCJP), pero para quien le interese, nunca está de más.

El decorado de un objeto siempre se realiza de los más elementales, a los más abstractos. Veamos un ejemplo:

```
import java.io.File;
import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.IOException;

public class PruebasFiles_003 {
  static public void main(String[] args) {
    File file = new File("PruebasFiles_003.txt");
    try {
      FileWriter fw = new FileWriter(file);
      BufferedWriter bf = new BufferedWriter(fw);

      bf.write("Soy una clase decoradora!");
      bf.newLine();
      bf.write("Estoy en otra la");

      bf.flush();
      bf.close();
    } catch(IOException ex){
      System.out.println("Ocurrio un error:" + ex);
    }
  }
}
```

En este caso, el BufferedWriter decora al FileWriter, y este a su vez decora a File.

En este caso, al crear el FileWriter se crea el archivo, pero no el directorio. Si el archivo esta contenido dentro de un directorio que no existe se generará una excepción IOException.

Trabajando con archivos y directorios

Básicamente, trabajar con archivos es casi lo mismo que con directorios, dado que para Java, un directorio no es más que un objeto de tipo File que referencia a un Directorio.

```
import java.io.File;
import java.io.IOException;

public class PruebasFiles_004 {
  static public void main(String[] args) {
    File dir = new File("carpeta");
    dir.mkdir(); //Genera el directorio carpeta

    //Crea un archivo que se encuentra en el directorio carpeta
    File file = new File(dir, "archivo.txt");
    try {
      file.createNewFile(); //Crea el archivo archivo.txt
    } catch(IOException ex){}
  }
}
```

También podemos listar todo el contenido de un directorio:

```
import java.io.File;
import java.io.IOException;

public class PruebasFiles_005 {//throws IOException {
  static public void main(String[] args) {
    File file = new File(".");
    String[] s = file.list();

    for (String str : s) {
      System.out.println(str);
    }
  }
}
```

Java.io.Console

A partir de Java 6 encontramos dentro del **package** java.io la clase Console. Esta sirve para obtener la instancia de la consola en que se ejecuta la aplicación, siempre que esta sea una aplicación de consola.
En caso de que no lo sea, al intentar invocar un objeto de este tipo, recibiremos un valor **null**.
Los métodos de Console:

- flush() : void
- format(String fmt, Object... args) : Console
- printf(String format, Object... args) : Console
- reader() : Reader
- readLine() : String
- readLine(String fmt, Object... args) : String
- readPassword() : **char**[]
- readPassword(String fmt, Object... args) : **char**[]
- writer() : PrintWriter

```
import java.io.Console;
import java.util.Arrays;

public class PruebasFiles_006 {
  static public void main(String[] args) {
    Console console = System.console();
    if (console != null) {
      char[] pass;
      char[] realpass = {'e', 'n', 't', 'r', 'a', 'd', 'a'};

      pass = console.readPassword("%s", "password: ");
      console.format("%s", "your password: ");
      for(char c : pass) {
        console.format("%c", c);
      }
      console.format("%s", "\n");

      if (Arrays.equals(pass, realpass)) {
        console.format("%s",  "Bienvenido...");
      } else {
        console.format("%s",  "Lo lamento... Pero no vas a entrar!");
      }
    } else { //if (console != null) {
      System.out.println("No se pudo obtener la instancia de la consola.");
    }
  }
}
```

El método format será explicado más adelante.

Serialización de objetos

La serialización de un objeto es el almacenamiento del estado de este. Recordemos que cuando nos referimos a estado, hablamos de los atributos de un objeto.
Por defecto, todos los atributos de un objeto serán serializados, a menos que estos se marquen con el modificador de no acceso **transient**.

La serialización de un objeto almacena el estado del objeto, pero no de la clase, es decir, si hay atributos **static**, estos no serán serializados.

ObjectOutputStrem y ObjectInputStream

Estas clases poseen dos métodos con los que tendremos que trabajar:

- ObjectOutputStream.writeObject();
- ObjectInputStream.readObject();

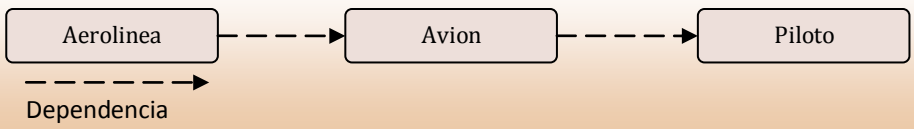
```
import java.io.File;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.io.FileOutputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.Serializable;
//import java.lang.ClassNotFoundException; //No es necesario el import

public class PruebasSerialize_1 {
  static public void main(String[] args) {
    File file = new File("PruebasSerialize_Pruebas.obj");
    //Creamos el objeto
    Pruebas prueba = new Pruebas(45, 30);

    //Lo serializamos
    try {
      FileOutputStream fs = new FileOutputStream(file);
      ObjectOutputStream ou = new ObjectOutputStream(fs);
      ou.writeObject(prueba);
      ou.close();
    } catch(IOException ex) {
      System.out.println("Ocurrio un error: " + ex);
    }

    //Establecemos el objeto a null para comprobar que realmente lo estamos leyendo.
    prueba = null;
```

```
//Lo deserializamos
    try {
      FileInputStream fi = new FileInputStream(file);
      ObjectInputStream oi = new ObjectInputStream(fi);
      prueba = (Pruebas) oi.readObject();
      oi.close();
    } catch(IOException ex) {
      System.out.println("Ocurrio un error: " + ex);
    } catch(ClassNotFoundException ex) {
      System.out.println("No es posible encontrar la clase: " + ex);
    }
    //Mostramos el objeto por pantalla para verificar que se obtuvieron los datos
    System.out.println(prueba);
  }
}
//Es necesario implementar la interface Serializable
class Pruebas implements Serializable {
  int width;
  int height;

  public Pruebas (int width, int height) {
    this.width = width;
    this.height = height;
  }

  public String toString() {
    return "width:" + width + "-height:" + height;
  }
}
```

Serializable es una interfaz constructora. No tiene métodos para implementar.

Gráfico de objetos

Supongamos que tenemos la siguiente estructura de objetos:



Y quisiéramos serializar nuestro objeto Aerolinea, podríamos pensar que hay que serializar el piloto, luego el avión, y luego la aerolínea. Imaginen si el árbol de clases se duplica.

De manera que Java ideó una solución, si serializamos un objeto que contenga dependencias a otro objeto, la serialización se produce en cadena.

Pero hay un pequeño inconveniente. Como vimos anteriormente, para poder serializar un objeto, este debe de implementar Serializable, si se genera una serialización en cadena, todas las clases afectadas deben de implementar serializable.

Ejemplo completo: http://thechroniclesofbinary.googlecode.com/files/PruebasSerialize_2.java

Compilar: javac –g PruebasSerialize_2.java

writeObject y readObject

Java posee un mecanismo que permite generar una serie de métodos privados, los cuales, si existen, serán invocados durante la serialización y deserialización.

Más patrones. Lo que estamos viendo en este caso es un ejemplo del patrón template.
http://metodologiasdesistemas.blogspot.com/search/label/decorador
Nuevamente, no es necesario conocer este patrón (no a fines de rendir la certificación SCJP)

La firma de los métodos son:

- private void writeObject(ObjectOutputStream os)
- private void readObject(ObjectInputStream is)

Veamos un ejemplo de una situación donde esto sería útil:
Supongamos que poseemos las siguientes 2 clases, pero "ClaseIncognita" es una clase cuyo fuente no tenemos (esto es muy probable cuando utilices librerías de terceros).

```
//Clase incógnita. Solo conocemos la firma de su constructor y método
ClaseIncognita(int valor);   //Firma constructor
int getValor();         //Firma método

//Y tenemos nuestra clase que utiliza ClaseIncognita
class ClaseConocida implements Serializable {
  private ClaseIncognita claseIncognita;
  private int otroValor;

  public ClaseConocida(int otroValor, ClaseIncognita clase) {
    this.otroValor = otroValor;
    claseIncognita = clase;
  }

  public int getOtroValor() {
    return otroValor;
  }

  public int getValor() {
    int valor = -1;
    if (claseIncognita != null) //Código defensivo
      valor = claseIncognita.getValor();
    return valor;
  }
}
```

Los desarrolladores de la librería indicaron que ClaseIncognita no es serializable, pero si nosotros quisiéramos serializar ClaseConocida estaríamos en problemas. Como vimos, la serialización se genera en cadena, de manera que cuando llegue a ClaseIncognita, y se encuentre con que no implementa serializable, tendremos un hermoso NotSerializableException.
Veamos como solucionamos esto:

```
//AGREGAMOS A LA CLASE ClaseConocida LOS SIGUIENTES MÉTODOS

//Implementamos los métodos readObject y writeObject
//throws IOException
private void readObject(ObjectInputStream is) {
  try {
      is.defaultReaderObject();
      claseIncognita = new ClaseIncognita(is.readInt());
  } catch(IOException ex) {
      System.out.println("Error al intentar serializar el objeto: " + ex);
  } catch(ClassNotFoundException ex) {
      System.out.println("La clase solicitada no fue localizada: " + ex);
  }
}

//throws IOException, ClassNotFoundException
private void writeObject(ObjectOutputStream os) {
  try {
    os.defaultWriteObject();
    os.writeInt(claseIncognita.getValor());
  } catch(IOException ex) {
    System.out.println("Error al intentar serializar el objeto: " + ex);
  }
}
```

Cuando se llama a la serialización, se invoca el método writeObject, el cual hace lo siguiente:

1. invoca el método defaultWriteObject. Esto le indica a la clase que se serialice normalmente.
2. Agrena un valor de tipo **int** luego de la serialización del objeto.

Cuando se llama a la deserialización, se invoca el método readObject, el cual hace lo siguiente:

1. Invoca el método defaultReadObject. Esto le indica a la clase que se deserialice normalmente.
2. Lee un carácter de tipo **int** luego de la deserialización.

Esto es posible gracias a que los streams al leer los datos llevan un puntero. Cuando nosotros serializamos estamos diciendo por ejemplo: los 48 caracteres que acabo de escribir son del objeto serializado. Luego le decimos, agregate un **int** (4 bytes).
Cuando hay que deserializarlo le decimos a la clase que lo haga de forma automática (esto quiere decir que va a leer los primeros 48 caracteres para regenerar la clase). Luego, sabemos que colocamos un **int** luego de estos, así que lo leemos.
Es muy importante respetar el orden de almacenamiento, junto con los tipos de datos (recuerda que según el tipo es el tamaño del mismo, y los streams trabajan en bytes).
Pero este código aun no funciona… ¿Perdón?
No pusimos el atributo claseIncognita con el modificador de no acceso **transient**.
Ejemplo completo: http://thechroniclesofbinary.googlecode.com/files/PruebasSerialize_3.java
Compilar: javac –g PruebasSerialize_3.java

Si la clase padre implementa la interfaz Serializable, la clase hija implícitamente será también serializable, por más que esta no la implemente. Si recordamos las propiedades de herencia y polimorfismo es fácil darnos cuenta de ello.
Si recuerdan, en capítulos anteriores hemos hablado de la manera en que se invocaba un método, y las llamadas que este producía.
En el caso de Serializable, es totalmente diferente:
Cuando un objeto es creado a raíz de una deserialización, no se invoca a ningún constructor (ni clase ni superclases), ni se le dan valores por defecto a todos sus atributos.
Si la clase hija implementa Serializable, pero no la clase padre, solo se serializan los atributos de la clase hija, y el padre inicia con los valores por defecto de cada tipo primitivo.

Fechas, números y monedas

La API de Java provee un set bastante extenso de clases para realizar tareas con fechas, números y monedas, pero para el examen solo es necesario conocer las más comunes de estas.
Estos conceptos lo que harán será adentrarnos un poco en el mundo de la internacionalización, o más bien conocido como i18n (una manera de abreviar palabras es escribiendo la primer letra y la última, y en medio, la cantidad de caracteres que hay entre medio, de manera que JavaWorld se escribiría j7d).
Para este fin debemos de conocer algunas clases de los package java.util y java.text:

package	Clase	Desc.
java.util	Date	La mayoría de los métodos de esta clase han quedado en desuso, pero es posible utilizarla como puente entre Calendar y DateFormat. Una instancia de Date representa una fecha y hora determinada, expresada en milisegundos (formato de fecha UNIX).
java.util	Calendar	Esta clase provee una gran variedad de métodos que permiten la manipulación y conversión de fechas y horas.
java.text	DateFormat	Esta clase no solo permite dar un formato a la fecha como Enero 2, 2009, sino que también permite formatear según la configuración regional.
java.text	NumberFormat	Esta clase es utilizada para formatear la moneda por configuraciones regionales.
java.util	Locale	Esta clase se utiliza en conjunto con DatoFormat y NumberFormat para formatear fechas, números y monedas para una región específica.

La clase Date

La clase Date se encuentra en desuso debido a que durante su diseño, no se tuvo en cuenta la i18n.
De todas maneras, la clase Date se encuentra en el examen por las siguientes razones:

- Es muy factible encontrarla en código antiguo.
- Es sencillo si lo que se busca es una manera rápida y simple de obtener fechas.
- Cuando se requiere solo una fecha universal.
- Es utilizada como puente entre las clases Calendar y DateFormat.

Esta clase contiene un **long** que representa los milisegundos desde el 01/01/1970 (también conocido como formato UNIX o POSIX).

Método/Constructor	Descripción	Ejemplo
Date()	**Constructor** que crea un objeto Date con la fecha actual.	`Date date = new Date();`
Date(**long** fecha)	**Constructor** que crea un objeto Date con la fecha especificada.	`Date date = new Date(1000000L);`
getTime()	Obtiene la fecha del objeto Date.	`Date date = new Date();` `long l = date.getDate();`
setTime(**long** fecha)	Establece la nueva fecha para el objeto Date.	`Date date = new Date();` `date.setDate(1234L);`

La salida estándar de Date es "[Mes] [Día] [Hora]:[Minutos]:[Segundos] MDT [Año]".
Ejemplo: Jan 08 21:12:45 MDT 2009.

La clase Calendar

Calendar es una clase abstracta, de manera que no puedes crear una instancia de Calendar con un **new**.

Calendar se encuentra diseñada con el patrón Singleton.
Para lo que deseen saber acerca del mismo pueden consultar **http://es.wikipedia.org/wiki/Singleton**.
La finalidad de este es que solo exista una única instancia de esta clase.

Calendar posee un enum para indicar si se trata de meses, días, semanas, años, y otros. Pueden ver la lista completa en la doc. de la API de Java en:
http://java.sun.com/j2se/1.5.0/docs/api/java/util/Calendar.html

Método/Constructor	Descripción	Ejemplo
Calendar.getInstance()	Constructor que devuelve una instancia de Calendar con fecha 01/01/1970.	`Calendar calendario = Calendar.getInstance();`
Calendar.getInstance(Locale region)	Constructor que devuelve una instancia de Calendar con fecha 01/01/1970 con una configuración regional específica.	`Locale region = new Locale("es");` `Calendar calendario = Calendar.getInstance(region);`
setTime(Date fecha)	Establece la fecha obteniendo la misma de un objeto Date.	`calendario.setDate(fecha);`
getDate()	Obtiene la fecha recibiendo un objeto de tipo Date.	`calendario.getDate();`
getFirstDayOfWeek()	Obtiene según la configuración regional, cual es el primer día de la semana.	`calendario.getFirstDayOfWeek();`
add(int part, int cant)	Añade la cantidad especificada a la parte indicada. Part se especifica mediante un enumerador y puede representar un día, mes, año, o cualquier otro.	`calendario.add(Calendar.MONTH, 5);`
roll(int part, int cant)	Actua similar a add, salvo que las partes mayores no se modifican. Ejemplo:si modificamos los días, por más que agrguemos 100, el més y año seguirán siendo los mismos.	`calendario.roll(Calendar.MONTH, 5);`

La clase DateFormat

Al igual que Calendar, DateFormat es una clase abstracta, de manera que solo puedes obtener una instancia de la misma a través de métodos de clase.

Constructor
DateFormat.getInstance()
DateFormat.getDateInstance()
DateFormat.getDateInstance(int format)
DateFormat.getDateTimeInstance()
DateFormat.getDateTimeInstance(int format)

Un ejemplo de formateo:

```
import java.util.Date;
import java.text.DateFormat;

//Configuración regional Calendario Griego - GMT -03:00

public class PruebasFormat_002 {
  static public void main(String[] args) {
    Date fecha = new Date();
    DateFormat[] formats = new DateFormat[6];
    formats[0] = DateFormat.getInstance();
    formats[1] = DateFormat.getDateInstance();
    formats[2] = DateFormat.getDateInstance(DateFormat.SHORT);
    formats[3] = DateFormat.getDateInstance(DateFormat.MEDIUM);
    formats[4] = DateFormat.getDateInstance(DateFormat.LONG);
    formats[5] = DateFormat.getDateInstance(DateFormat.FULL);

    for(DateFormat df : formats) {
      System.out.println(df.format(fecha));
    }
  }
}
```

```
28/12/09 17:53
28/12/2009
28/12/09
28/12/2009
28 de diciembre de 2009
lunes 28 de diciembre de 2009
```

La clase Locale

La API dice que la clase Locale es “una ubicación geográfica, políticam o región cultural”.
Para el examen, solo necesitamos saber los siguientes constructores:

- Locale(String idioma)
- Locale(String idioma, String ciudad)

Y los métodos:

- getDisplayCountry()
- getDisplayCountry(Locale ubicacion)
- getDisplayLanguage()
- getDisplayLanguage(Locale ubicacion)

El idioma representa un código establecido en la norma ISO 639.

Tanto DateFormat como NumberFormat pueden recibir como parámetro un objeto Locale en momento de la instanciación. Pero una vez instanciado este, no hay ningún método para cambiar la configuración geográfica.

La clase NumberFormat

NumberFormat también es una clase abstracta, así que no intentes ningún **new** con esta.

Método/Constructor	Descripción
NumberFormat.getInstance()	Obtiene una instancia con la configuración regional por defecto.
NumberFormat.getInstance(int region)	Obtiene una instancia con la configuración regional especificada.
NumberFormat.getCurrencyInstance()	Obtiene la instancia de la moneda para la configuración regional por defecto.
NumberFormat.getCurrencyInstance(int región)	Obtiene la instancia de la moneda para la configuración regional especificada.
getMaximunFractionDigits()	Obtiene la cantidad máxima de decimales en la fracción.
setMaximunFractionDigits(int digitos)	Establece la cantidad máxima de decimales en la fracción.
parse() : throws ParseException	Parsea un String.
setParseIntegerOnly(boolean soloInteger)	Establece si se obtiene solo la parte entera, o también la parte decimal.

Tablas de resumen

Tabla en la que se resumen todas las instanciaciones o creaciones:

Clase	Costructor o método de creación
java.util.Date	`new Date();` `new Date(long milisegundosDesde01011970);`
java.util.Calendar	`Calendar.getInstance();` `Calendar.getInstance(Locale region);`

java.util.Locale	`Locale.getDefault();`
	`new Locale(String idioma);`
	`new Locale(String idioma, String ciudad);`

java.text.DateFormat	`DateFormat.getInstance();`
	`DateFormat.getDateInstance(int estilo);`
	`DateFormat.getDateInstance(int estilo, Locale region);`

java.text.NumberFormat	`NumberFormat.getInstance();`
	`NumberFormat.getInstance(Locale region);`
	`NumberFormat.getNumberInstance();`
	`NumberFormat.getNumberInstance(Locale region);`
	`NumberFormat.getCurrencyInstance();`
	`NumberFormat.getCurrencyInstance(Locale region);`

Tabla en la que se resuelven los usos más comunes para las clases anteriores:

Caso	Pasos
Obtener la fecha y hora actuales.	`Date fecha = new Date();` `String fechaStr = fecha.toString();`

Obtener un objeto que permita realizar cálculos de fecha y hora en la configuración regional por defecto.	`Calendar cal = Calendar.getInstance();`
	`cal.add(Calendar.MONTH, 3);`
	`cal.roll(Calendar.YEAR, 1);`

Obtener un objeto que permita realizar cálculos de fecha y hora en cualquier configuración regional.	`Locale region = new Locale("en", "US");`
	`Calendar cal = Calendar.getInstance(region);`
	`cal.add(Calendar.MONTH, 3);`
	`cal.roll(Calendar.YEAR, 1);`

Obtener un objeto que permita realizar cálculos de fecha y hora, y luego formatear la salida para diferentes regiones con diferentes formatos de salida.	`Calendar cal = Calendar.getInstance();`
	`Date fecha = cal.getDate();`
	`Locale region = new Locale("en", "US");`
	`DateFormat df = DateFormat.getDateInstance(DateFormat.FULL, region);`
	`String fechaStr = df.format(fecha);`

Obtener un objeto que permita formatear números o monedas entre varias regiones.	`Locale region = new Locale("en", "US");`
	`float valor = 123.4567f;`
	`NumberFormat nf1 = NumberFormat.getInstance(region);`
	`NumberFormat nf2 = NumberFormat.getCurrencyInstance(region);`
	`String valor1 = nf1.format(valor1);`

```
String valor2 = nf2.format(valor2);
```

Búsqueda y parseo dentro de Strings

Para buscar entre grandes cadenas de texto, lo más común y utilizado son las expresiones regulares. Estas representan un lenguaje para buscar cadenas.

Todo lenguaje que tenga soporta para expresiones regulares permite buscar una determinada expresión. Estas son como pequeños programas que permiten identificar una sección de texto que cumpla con ciertas características. Lo que se hace es pasarle al motor una cadena y una expresión.

Ten en cuenta que las expresiones regulares que explicaremos a continuación no abarcan todo el abanico de posibilidades, pero son las necesarias para el examen.

Tabla para la creación de expresiones:

Cadena	Descripción	Ejemplo
Asd	Cuando no se especifican caracteres especiales (aquellos que se explican debajo), se interpreta como una cadena simple (diferencia mayúsculas de minúsculas).	Cadena: aassddasdddasd Expresión: asd Coincidencias: 6 11
\d, \s o \w	Estos meta-caracteres buscan: \d cualquier número (0 a 9) \s saltos de línea \w caracteres alfanuméricos (letras, números, o _)	Cadena: 0123aaddd1df Expresión: \d Coincidencias:0 1 2 3 9
[abc], [a-f] , [a-fA-F]	Dentro de los [] se especifica un juego y/o rango de caracteres válidos. Aquellos comprendidos entre – son los rangos. Ejemplo: supongamos que buscamos cualquier dígito, podríamos decir [0123456789], o lo que es lo mismo [0-9].	Cadena: JavaWorld Expresión: [0-9a-f] Coincidencias: 1 3 8
[^asd0-9]	La expresión [^…] actua al contrario que […]. Esta se comporta buscando cualquier carácter que no se encuentre dentro del rango especificado.	Cadena: 0123456789 Expresión: [^0-36-9] Coincidencias: 4 5
+, *, ?	Comodines de cantidad. Permiten indicar una cantidad de un grupo o carácter (un meta-carácter es interpretado como un carácter) a buscar. + indica al menos 1 o más * indica 0 o más ? indica 0 o 1 ocurrencia	Cadena: 01 a2a 223 Expresión: \d+ Coincidencias: 0 4 7
()	Indica que se busca cualquier carácter dentro del grupo. Estos se utilizan para luego poder reemplazarlo por otra expresión, pero esa parte no se explica dado que el examen no lo requiere. Se utiliza generalmente en conjunto con [] y un comodín (*, +, ?)	Cadena: gustavoalberola@JavaWorld.com Expresión: ([^@])+ Coincidencias: 0 16

Cuando se busca solo un carácter con ?, ten en cuenta que luego del último carácter hay un final de cadena, y este entra dentro de la selección por el ? como 0 ocurrencias.
Ejemplo: Cadena: bab Expresion: b? Resultados: [0,1] b [1,1] [2,3] b [3,3]
También debes de tener cuidado cuando escribes las expresiones dentro de un String, recuerda que un carácter \ es un carácter especial, deberás de escribirlo como \\;

Un ejemplo dinámico con expresiones regulares

El siguiente código recibe desde la línea de parámetros la expresión regular a buscar y la cadena dentro de la cual se busca la expresión.
El primer parámetro es la expresión regular, el segundo es la cadena.
Un ejemplo sería: java PruebasRegex_001 "\d" "012abc"

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class PruebasRegex_001 {
  //Llamar desde la consola como: java PruebasRegex_001 "ExpresionRegular" "Cadena"
  static public void main(String[] args) {

    String ptn = (args.length > 0 && !args[0].equals("")) ? args[0] : "";
    String cad = (args.length > 1 && !args[1].equals("")) ? args[1] : "";

    Pattern expresion   = Pattern.compile(ptn);
    Matcher matcher    = expresion.matcher(cad);

    System.out.print("Cadena    : ");
    for(char c : cad.toCharArray()){
      System.out.print(c + "  ");
    }
    System.out.println();

    System.out.print("Posiciones: ");
    for(int x = 0; x < cad.length(); x++) {
      if(x < 10){
        System.out.print(x + "  ");
      } else {
        System.out.print(x + " ");
      }
    }
    System.out.println();

    System.out.println("Expresion : " + matcher.pattern());
    System.out.println();

    System.out.println("Coincidencias: ");
    while(matcher.find()) {
      System.out.println("[" + matcher.start() + "," + matcher.end() + "]: " +
matcher.group());
    }
  }
}
```

Búsquedas utilizando la clase Scanner

A diferencia de Pattern, la clase Scanner no posee funcionalidad para eliminar o reemplazar, es posible buscar una expresión regular y obtener la cantidad de veces que esta aparece.

```
import java.util.Scanner;

public class PruebasRegex_002 {
  static public void main(String[] args) {
    System.out.print("input: ");
    System.out.flush();

    try {
      Scanner sc = new Scanner(System.in);
      String token;
      do {
        token = sc.findInLine(args[0]);
        System.out.println("Coincidencias: " + token);
      } while (token != null);
    } catch(Exception ex) {
      System.err.println("Error al intentar leer dentro del Scanner. " +
ex.toString());
    }
  }
}
```

Tokenizing

Sepan disculpar el nombre en inglés, pero no se me ocurría ninguna palabra que se le adecuara en español.
Tokenizing es el proceso por el cual se separa un bloque de datos en varios tokens, los cuales se encuentran divididos por uno o varios caracteres especiales, denominado parser.

Imaginemos la sentencia “aa2d3,222,333”, suponiendo que el carácter de parser es la , los tokens serían:

- aa2d3
- 222
- 333

Obtener los tokens con String.split()

El método String.split(Expresion s) recibe como parámetro una expresión regular. Lo que hará es generar un array buscando los tokens que se encuentran divididos por el parser especificado como la expresión.
Veamos un ejemplo:

```
public class PruebasTokenizing_001 {
  static public void main(String[] args) {
    String s = "hola.Java.World";
    String[] s2 = s.split("\\.");
    for(String str : s2) {
      System.out.println(str);
    }
  }
}
```

```
hola
Java
World
```

La desventaja de esta operación es que aplica a toda la cadena. Para textos grandes no es performante, y menos, si lo que solo necesitábamos era la primer coincidencia.

Obtener los tokens con la clase Scanner

Cuando sea necesario realizar una búsqueda de tokens avanzada, la clase java.util.Scanner es la solución.
Algunas de sus ventajas son:

- Puede ser construido con Strings, Streams, o Files.
- La búsqueda de tokens se realiza dentro de un bucle, pudiendo salir de este cuando lo deseemos.
- Los tokens pueden ser convertidos a un tipo primitivo inmediatamente.

Antes de ver un ejemplo, les presento algunos de sus métodos:

- nextXxx() Xxx puede ser reemplazado por el nombre de un tipo primitivo de datos. Devuelve el token formateado con el tipo de datos especificado.
- next() Devuelve el token como String.
- hasNextXxx() Xxx puede ser reemplazado por el nombre de un tipo primitivo de datos. Verifica si el siguiente token es del tipo de datos especificado.
- hasNext() Indica si hay otro token más para procesar.
- useDelimiter(String expresion) Utiliza la expresion que especifiquemos como argumento en la llamada para buscar los parsers.

Formateo con printf y format

La clase java.io.PrintStream presenta varias opciones para presentar la salida de datos de manera formateada. Es necesario conocer esta sintaxis para el examen.
Antes que nada, aclaremos el hecho de que printf y format son métodos exáctamente iguales (corre el rumor de que Sun introdujo printf para hacer felices a los programadores de c++), de manera que de aquí en más nos referiremos solamente a format, siendo equivalente la sintaxis para printf.
La firmá del método es:

Format(String cadena, Object... argumentos)

El String contenido posee ciertos caracteres especiales para el formateo de los datos. Resumimos en una tabla los posibles valores:

El formato de una expresión se forma como:
%[INDICE_ARGUMENTO$][FLAGS][ANCHO][.PRESICION]FORMATO.
Este no lleva ningún espacio, y todos los argumentos dentro de [] son opcionales.

Parámetro	Valores	Descripción	Ejemplo
INDICE_ARGUMENTO	int	Indica de los argumentos especificados, a que posición hace referencia. Si no se especifica, se administran por orden de primero (izquierda) a último (derecha). El primer argumento corresponde al índice 1.	`format("%2$d - %1$d", 12, 15); //15 - 12`
**FLAGS	-	*Alinea a la izquierda (**requiere especificación de ANCHO**).	`format("int: %-15d \r\n", 123); //123`
	+	***Requiere que FORMATO sea un número (d | f)**, incluye el signo del número (+ o -).	`format("int: %+d \r\n", 123); //+123`

	0	***Requiere que FORMATO sea un número (d | f)**, agrega ceros a la izquierda (**requiere especificación de ANCHO**),	`format("int: %05d \r\n", 123); //00123`
			`format("float: %05f \r\n", 1.23f); //1,230000`
			`format("float: %05.2f \r\n", 1.23f); //01,23`

	,	***Requiere que FORMATO sea un número (d | f)**, indica que se utilice el separador de decimales configurado para la región.	`format("float: %,f \r\n", 1.23f); //1,230000`
	(	***Requiere que FORMATO sea un número (d | f)**, encierra los valores negativos entre paréntesis.	`format("float: %(f \r\n", -1.23f); //(1,230000)`
ANCHO	int	Indica la cantidad mínima de valores a escribir. (Por defecto rellena con espacios, salvo que se especifique FLAGS = 0, y con alineación a la derecha, salvo que se especifique FLAGS = -).	`format("int: %5d \r\n", 123); //  123`
PRESICION	int	***Requiere que FORMATO sea un número de punto flotante (f)**, indica la cantidad de decimales a mostrar.	`format("float: %.2f \r\n", 1.23f); //1,23`
FORMATO	b	***El argumento debe coincidir con el tipo**. Interpreta el argumento como tipo **boolean**.	`format("char: %c", 'Z'); //char: Z`
	c	***El argumento debe coincidir con el tipo**. Interpreta el argumento como tipo **char**.	`format("int: %d", 123); //int: 123`
	d	***El argumento debe coincidir con el tipo**. Interpreta el argumento como tipo **int**.	`format("float: %f", 1.23f); //float: 1,230000`
	f	***El argumento debe coincidir con el tipo**. Interpreta el argumento como tipo **float**.	`format("boolean: %b", true); //boolean: true`
	s	***El argumento debe coincidir con el tipo**. Interpreta el argumento como tipo String.	`format("String: %s", "JavaWorld"); //String: JavaWorld`

*Si no se cumple lo resaltado en negrita al aplicar el parámetro, se generará un error en tiempo excepción dependiendo del parámetro faltante o equivocado, o error de casteo en caso del parámetro F
**Los flags son combinables, pudiendose especificar más de uno al mismo tiempo.

Lo que se viene

En el próximo capítulo aprenderemos a utilizar la sintaxis que Java nos propone para el uso de los genéricos.

Nos adentraremos en otra de las API de Java, esta vez para manejar colecciones.

Veremos las cuatro interfaces más importantes de dicha API, y haremos varios ejemplos de práctica.

Sigan conectados con más JavaWorld para la próxima.

Valor Creativo

Una mirada a la Industria del software en la Argentina

Lo más importante en un mercado es saber analizarlo. A partir de ello se encuentran las tendencias, y la información que se convierten en oportunidades de negocio.

La industria del software en el 2007 facturó unos 5.100 millones de pesos. En el 2008 unos 7.738 millones de pesos. Y en el 2009 alcanzó unos 9.700 millones. En otras palabras, un crecimiento promedio del 20% anual.

Uno de los indicadores dice que se vienen generando unos 60 mil puestos de trabajo anuales, cuota que no es satisfecha por los egresados de las carreras del sector.

Actualmente la industria de la informática factura más que la industria del vino en la Argentina.
Uno de los casos más relevantes del 2009 fue la empresa Doppler. Una empresa dedicada al marketing por correo electrónico. Con una mínima inversión inicial, la empresa en su primer año facturó unos 125 mil dólares. Al segundo año, pasó a facturar 300 mil dólares. Actualmente, cuenta con un equipo de 15 personas, y trabaja para toda Sudamérica y países de habla hispana.
En conclusión, el mercado de la informática es atractivo, económico, de bajo riesgo, pero, requiere de un alto nivel de especialización.

Nadie lo dice, pero este es el auge de la informática, y el gobierno "sin ser tendencioso", promulgó la ley 25.922 de promoción de la industria de software, que otorga bonificaciones del 60% en el impuesto a las ganancias, y del 70% en el los aportes patronales a las empresas del sector durante 10 años. La ley fue reglamentada por el decreto 1.594/04, y comenzó a regir el 17 de septiembre de 2004. Es decir, aun está vigente. Actualmente se estima que hay alrededor de 330 pymes de software que perciben los beneficios de la promoción. Para acceder al régimen hay que demostrar que la actividad principal es el desarrollo de software, o servicios informáticos, que se opera en el país y por cuenta propia. Además, hay que tener alguna certificación de la calidad (gestión de normas de la calidad ISO 9000). **Para más información www.industria.gov.ar/lpsw o al 011-4349-3683**.
Analicemos la genialidad de esta ley:

- Promueve una industria en la cual la materia prima son las personas.
- Las personas deben estar capacitadas (algo que requiere la certificación de calidad).
- La ley busca que la Argentina sea sinónimo de calidad con respecto al software.
- El 80% de la producción de estas empresas se exporta, y el gobierno se queda con el 33% de esas ventas (el gobierno no pierde nada con todos los beneficios que otorga).
- Es una industria que no consume recursos naturales, solo posee insumos como es el conocimiento y las personas. Genera ingresos millonarios con inversión mínima (un sitio + una computadora + una persona), y presenta un crecimiento sostenido del 20%.

En la actualidad se necesitan programadores. Existe una demanda del mercado que no puede ser satisfecha en cuanto a recursos humanos, y por ello las universidades lanzan carreras cortas de 2 a 3 años, para inyectarlos al mercado lo más rápido posible. También es muy común que opten por obtener certificaciones como las otorgadas por la empresa Sun (certificaciones en Java, Solaris y ahora Oracle).

Como emprendedor me pregunto "¿comienzo algún emprendimiento con la rama y tendré éxito?". La respuesta es no. Para dirigir este tipo de empresas es necesario estar capacitado en el tema. Es un ámbito en donde prevalece el más actualizado y donde el

valor se centra en el conocimiento. No cualquiera sabe programar.

Si se preguntan cuanto tiempo más va a seguir esta situación, durará por lo menos 3 años más, en donde el 2010 será el boom de esta industria.

Mi consejo

Mi consejo es que te juntes con gente que tenga las mismas ganas que vos, y miren hacia adelante. Si eres joven, los errores no son tan dolorosos.

Si querés estudiar el momento es ahora, no lo dudes. Carreras hay, y que no te importe si la carrera es una tecnicatura, ya que la experiencia hace al maestro, y lo más probable es que cuando termines de estudiar tengas que anexar algún complemento extra porque lo que estudiaste se encuentra obsoleto.

Si no querés estudiar, pero tenés el capital necesario para invertir, mi consejo es “no empieces nada si no estás seguro”. No es solo cuestión de tener el dinero para empezar un emprendimiento, sino también la gente adecuada para llevarlo a cabo. Por ello recomiendo que busquen a esos micro emprendimientos que quieren dar el salto, pero no poseen el capital para hacerlo. Un ejemplo es Twitter, una empresa que recién se muestra en la argentina, pero en Estados Unidos es furor. Una de las curiosidades es que esta empresa no facturó un dólar desde el 2007 en su creación, hasta mediados del 2009. Esto fue posible gracias a la visión de los inversionistas de riesgo que aportaron 55 millones de dólares para que sea una realidad.

Yo no digo invertir tanto, pero con “mucho menos” es posible hacer cosas muy buenas.

Hasta acá he llegado en el tema, pero tengo muchos casos de empresas que tuvieron éxito, tanto en la Argentina como en el mundo. Pero eso será en la próxima. Éxitos a todos, y felices emprendimientos. Valor creativo.

¿Tenés lo que se necesita para ser un SCJP?

Autores
Gustavo Alberola
Matias Álvarez

Diseño
Leonardo Blanco