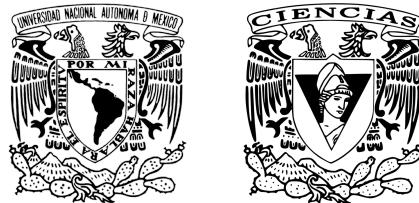


UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS



Práctica 01:
Compilador

Pablo A. Trinidad Paz

Trabajo presentado como parte del curso de **Introducción a Ciencias de la Computación**
impartido por la profesora **Verónica Esther Arriola Ríos**.

24 de agosto de 2018

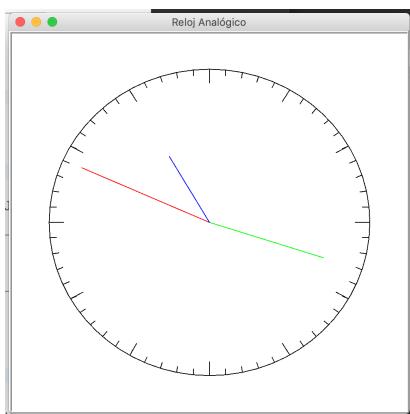
Los programas de la JDK

Actividad 1.2 Escribe exactamente que archivos fueron creados y donde:

Dentro de Reloj/src/icc/practica1 los siguientes archivos fueron creados:

1. ClaseReloj.class
2. PanelDeReloj.class
3. Reloj.class
4. TiempoSistema.class
5. UsoReloj.class
6. VistaReloj.class
7. VistaRelojAnalogico.class
8. VistaRelojAnalogico\$1.class

Actividad 1.3 Ejecuta UsoReloj



Actividad 1.4 Intenta invocar a la máquina virtual con los nombres de otros archivos .class. ¿Qué sucede? Lee lo que devuelve la consola y abre los archivos .java correspondientes que necesites. ¿Qué tiene el archivo UsoReloj.java que permite invocar su .class con java?

Al tratar de ejecutar otros archivos diferentes a `UsoReloj.class`, Java busca un método principal, `Main method`, que le especifique qué operaciones tiene que ejecutar para iniciar. Lo que tiene diferente el archivo `UsoReloj.java` que permite que la ejecución de la aplicación suceda es la implementación del método `main` en la línea 16, que además está definido como público, estático y que no regresa ningún valor al terminar la ejecución.

Actividad 1.6 Generar la documentación de `practica1`. ¿Qué observas?

Se generaron archivos para servir un sitio web estático como: HTMLs, CSSs y archivos de JavaScript. Al acceder al website de los documentos podemos ver el listado de clases, descripción de cada una de ellas y dentro de cada una sus métodos, atributos, etc.

El website luce así:

The screenshot shows a browser window titled "icc.practica1". The address bar indicates the URL is "file:///Users/pablontrinidad/Documents/UNAM/1er/ICC/icc/practica1/package-summary.html". The page content is the generated Java documentation for the package `icc.practica1`.

Interface Summary

Interface	Description
<code>Reloj</code>	Interfaz de Reloj.
<code>VistaReloj</code>	Interfaz para definir qué servicios debe proveer la representación gráfica de un Reloj.

Class Summary

Class	Description
<code>ClaseReloj</code>	Clase para representar un Reloj.
<code>TiempoSistema</code>	Clase que asigna la hora del sistema a su reloj.
<code>UsoReloj</code>	La clase UsoReloj sirve (como su nombre indica), para hacer uso de los tipos Reloj y VistaReloj, definidos por las interfaces respectivas.
<code>VistaRelojAnalogico</code>	Clase para representar gráficamente un Reloj analógico.

At the bottom of the page, there is a navigation bar with links: PACKAGE, CLASS, TREE, DEPRECATED, INDEX, HELP, PREV PACKAGE, NEXT PACKAGE, FRAMES, NO FRAMES, ALL CLASSES, and a SEARCH bar.

Usando una herramienta auxiliar: ant

Las siguientes respuestas corresponden a las actividades, ejercicios y preguntas de la **Práctica 1** del Manual de Prácticas de Introducción a las Ciencias de la Computación escrito por **Canek Peláez V.** y **Elisa Viso G..**

Actividad 1.1 Al invocar el comando `javac` Anota todas las opciones que se pueden pasar al compilador.

- `@<filename>`: Read options and filenames from file
- `-Akey[=value]`: Options to pass to annotation processors
- `--add-modules <module>(<module>)*`: Root modules to resolve in addition to the initial modules, or all modules on the module path if `<module>` is ALL-MODULE-PATH.
- `--boot-class-path <path>, -bootclasspath <path>`: Override location of bootstrap class files
- `--class-path <path>, -classpath <path>, -cp <path>`: Specify where to find user class files and annotation processors

- **-d <directory>**: Specify where to place generated class files
- **-deprecation**: Output source locations where deprecated APIs are used
- **-encoding <encoding>**: Specify character encoding used by source files
- **-endorseddirs <dirs>**: Override location of endorsed standards path
- **-extdirs <dirs>**: Override location of installed extensions
- **-g**: Generate all debugging info
- **-g: lines,vars,source**: Generate only some debugging info
- **-g:none**: Generate no debugging info
- **-h <directory>**: Specify where to place generated native header files
- **--help, -help**: Print this help message
- **--help-extra, -X**: Print help on extra options
- **-implicit: none, class**: Specify whether or not to generate class files for implicitly referenced files
- **-J<flag>**: Pass `flag` directly to the runtime system
- **--limit-modules <module>(<module>)***: Limit the universe of observable modules
- **--module <module-name>, -m <module-name>**: Compile only the specified module, check timestamps
- **--module-path <path>, -p <path>**: Specify where to find application modules
- **--module-source-path <module-source-path>**: Specify where to find input source files for multiple modules
- **--module-version <version>**: Specify version of modules that are being compiled
- **-nowarn**: Generate no warnings
- **-parameters**: Generate metadata for reflection on method parameters
- **-proc: none,only**: Control whether annotation processing and/or compilation is done.
- **-processor <class1>[,<class2>,<class3>...]**: Names of the annotation processors to run; bypasses default discovery process
- **--processor-module-path <path>**: Specify a module path where to find annotation processors
- **--processor-path <path>, -processorpath <path>**: Specify where to find annotation processors
- **-profile <profile>**: Check that API used is available in the specified profile
- **--release <release>**: Compile for a specific VM version. Supported targets: 10, 6, 7, 8, 9
- **-s <directory>**: Specify where to place generated source files
- **-source <release>**: Provide source compatibility with specified release
- **--source-path <path>, -sourcepath <path>**: Specify where to find input source files
- **--system <jdk>|none**: Override location of system modules
- **-target <release>**: Generate class files for specific VM version

- **--upgrade-module-path <path>**: Override location of upgradeable modules
- **-verbose**: Output messages about what the compiler is doing
- **--version, -version**: Version information
- **-Werror**: Terminate compilation if warnings occur

Ejercicio 3 Ejecutar ant clean y touch src/icc/practica1/UsoReloj.java y explicar la diferencia de correr ant antes y después de haber sobreescrito UsoReloj.java

Durante la primera ejecución de **ant**, la salida del programa decía que estaba compilando 6 archivos fuente mientras que en la segunda ejecución de **ant** la salida decía que únicamente estaba compilando un archivo:

```
hubble:Reloj:% ant
Buildfile: /Users/pablotrinidad/Documents/UNAM/1er/ICC/icc/psets/P01/P01/Reloj/build.xml

compile:
    [mkdir] Created dir: /Users/pablotrinidad/Documents/UNAM/1er/ICC/icc/psets/P01/P01/Reloj/build
    [javac] Compiling 6 source files to /Users/pablotrinidad/Documents/UNAM/1er/ICC/icc/psets/P01/P01/Reloj/build

BUILD SUCCESSFUL
Total time: 1 second
hubble:Reloj:% touch src/icc/practica1/UsoReloj.java
hubble:Reloj:% ant
Buildfile: /Users/pablotrinidad/Documents/UNAM/1er/ICC/icc/psets/P01/P01/Reloj/build.xml

compile:
    [javac] Compiling 1 source file to /Users/pablotrinidad/Documents/UNAM/1er/ICC/icc/psets/P01/P01/Reloj/build

BUILD SUCCESSFUL
Total time: 1 second
```

Este comportamiento se debe a que el archivo **UsoReloj.java** es el archivo principal que desencadena la ejecución completa de la aplicación y logra esto haciendo uso de los otros 5 archivos restantes.

Al momento en que nosotros usamos **touch UsoReloj.java**, el contenido del archivo **UsoReloj.java** es borrado y queda un archivo vacío. Después, cuando ejecutamos **ant**, o en su defecto **javac**, estos inician el proceso de compilación en un sólo archivo (al igual que en el caso previo) pero con la diferencia de que este nuevo, y vacío, archivo nunca hace referencia a los otros 5.

Pregunta 1 ¿Qué errores encontraste al compilar esta práctica? Explica en qué consisten.

- **not a statement**: Que hacía referencia a que una construcción del tipo **objeto.metodo()** no es correcta y debía ser cambiada por algo del tipo **objeto.metodo()**.
- **error: 'symbol' expected**: Donde **symbol** podría ser un **;**, **,**, **(** o cualquier elemento de la sintaxis de Java que fuera requerido en la expresión formada pero que resultaba faltante o se encontraba sustituido por uno incorrecto.
- **cannot find symbol**: Que significaba que se estaba haciendo uso de una variable que no había sido definida anteriormente. En el ejemplo de la práctica, el error estaba en haber escrito **relog** en lugar de **reloj**.

- method X cannot be applied to given types: Que significaba que un método de una clase estaba esperando recibir ciertos parámetros de cierto tipo y recibió otros diferentes a los especificados. En el ejemplo de la práctica, el error estaba cuando se llamaba el método `espera` desde una instancia de la clase `VistaRelojAnalogico` sin pasarle el entero positivo que el método estaba esperando.

Pregunta 2 **Los errores que encontraste, ¿de qué tipo crees que sean, sintácticos o semánticos? Justifica tu respuesta.**

Ambos. Sintácticos porque en algunos casos hacían falta símbolos para delimitar el fin de una sentencia (;) o la llamada a un método () pero de la misma manera había errores semánticos, es decir, relacionados al significado del contenido escrito. Por ejemplo: Llamar a la variable inexistente `relog` o ejecutar el método `espera` de la clase `VistaRelojAnalogico` sin mandarle los argumentos necesarios para que se ejecutara.

Pregunta 3 **¿Cuántos archivos en bytecode (los que tienen extensión .class) se generaron?**

8 archivos .class

```
hubble:src:% ls icc/practical/*.class | wc -l
8
hubble:src:% ls icc/practical/*.class
icc/practical/ClaseReloj.class          icc/practical/TiempoSistema.class      icc/practical/VistaRelojAnalogico$1.class
icc/practical/PanelDeReloj.class        icc/practical/UsoReloj.class           icc/practical/VistaRelojAnalogico.class
icc/practical/Reloj.class               icc/practical/VistaReloj.class
hubble:src:% |
```

Pregunta 4 **¿Cuál crees que sea la explicación del comportamiento de Ant después de hacer el ejercicio 3? Justifica tu respuesta.**

Como mencioné en la respuesta al **Ejercicio 3**, se debe a que el archivo principal del cual parte la compilación es diferente antes y después de hacer el `touch`. Antes, el archivo hace uso de los otros 5 mientras que después, el archivo está vacío y no usa ninguna otra clase.

Estructura de un programa

Actividad 1.7 **Ejecuta el programa Entrada ¿Qué aparece? Ahora, ejecutalo enviando argumentos ¿Qué obtienes?**

En la primera ejecución el programa regresa el mensaje "No se recibieron indicaciones." en la segunda ejecución el programa regresó la lista de argumentos enviados precedidos por el mensaje "Argumento n:" donde n será el número del argumento –1

Actividad 1.8 **Experimenta con el archivo Entrada.jav y reporta lo que hace el programa.**

Ejecutar `java -jar arg1 arg2 arg3` generará errores ya que el primer argumento después de `-jar` debe la ubicación del jarfile.

Agregando el argumento `Entrada.jar` entre `-jar` y el listado de argumentos `arg1 arg2 arg3` el comando se ejecuta de manera correcta y podemos observar el mismo comportamiento de listado de argumentos que se describe en la **Actividad 1.7**.

```
hubble:Entrada:% java -jar Entrada.jar a b c
Argumento 0: a
Argumento 1: b
Argumento 2: c
hubble:Entrada:% |
```

Actividad 1.10 Lee los dos archivos build.xml utilizados en esta práctica y observa en qué se parecen y en qué difieren. ¿Qué objetivos reconoce cada archivo? ¿Qué pasos ejecutará cada uno de los objetivos (observa el atributo llamado depends)?

Se parecen en que ambos tienen la misma estructura de tags. Por ejemplo, ambos comparten las etiquetas `xml` y `project`. Además, ambos tienen `targets` definidos que permiten compilar, correr, limpiar y generar la documentación del proyecto.

Difieren en que el XML del programa Entrada define más propiedades usando el tag `property` mientras que la clase del reloj no lo hace. También, se define un `target` adicional en Entrada para distribuir el paquete. Existen más comentarios para el XML de Entrada y de la misma manera la tarea default está definida como la sucesión de `clean`, `build` y `dist`.

Cada XML apunta a su respectivo proyecto y cada target "paso" difiere ligeramente de otro.

Acerca de los `depends`, especifican que `target` tiene que ser ejecutado antes de que ese puede correr. Por ejemplo, `run` requiere que `compile` suceda antes.