

1 | Compiladores e intérpretes

Meta

Que el alumno aprenda a utilizar un compilador para traducir código, detectar y corregir errores sintácticos y semánticos; y un intérprete para ejecutar `bytecode`.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Invocar al compilador de Java, `javac` desde una terminal para generar el `bytecode` ejecutable por la máquina virtual.
- Invocar a la máquina virtual de Java con el comando `java`, para ejecutar al código de una clase que contenga un método `main`.
- Empaquetar los archivos resultantes utilizando el comando `jar`.
- Ejecutar código en el archivo `.jar`.
- Generar la documentación del paquete utilizando `javadoc`.
- Utilizar la herramienta `ant` para compilar y empaquetar código, generar documentación y ejecutar un programa, utilizando un archivo `build.xml` provisto.

Código Auxiliar 1.1: Compiladores e intérpretes

<https://github.com/computacion-ciencias/icc-compiladoreseinterpretes>

1. Compiladores e intérpretes

Antecedentes

En esta práctica comenzarás a trabajar con el lenguaje de programación Java. Java es conocido como un lenguaje de alto nivel de abstracción pues fue diseñado para parecerse un poco más al lenguaje natural con el que nos comunicamos las personas; concretamente, al *inglés*. La computadora no entiende ese lenguaje, hay que traducirlo a un código que le resulte más familiar: el lenguaje *máquina*. La traducción de Java a lenguaje máquina se lleva a cabo en dos etapas, con la ayuda de dos programas:

1. Un *compilador* traduce Java a *bytecode*, un lenguaje ya muy parecido al lenguaje máquina de varias computadoras.
2. Un *intérprete*, específico para el sistema operativo en la computadora que ejecutará el programa (como *Windows* o *Linux*), traduce el bytecode a *lenguaje de máquina*. También se le conoce como la *máquina virtual de Java*.

Veamos formalmente cómo se definen ambos programas:

Definición 1.1: Compilador

“Un compilador es un programa que lee un programa escrito en un lenguaje, el lenguaje *fuente*, y lo traduce a un programa equivalente en otro lenguaje, el lenguaje *objeto*. Como parte importante de este proceso de traducción, el compilador informa a su usuario de la presencia de [algunos] errores en el programa fuente.”^a

^aAho, Sethi y Ullman 1998, pp. 1.

Definición 1.2: Intérprete

Un intérprete es un programa que, una vez cargado en la memoria de una computadora y al ejecutarse, procede como sigue:

1. Toma un enunciado del programa en lenguaje de alto nivel, llamado código fuente.
2. Traduce ese enunciado y lo ejecuta.
3. Repite estas dos acciones hasta que alguna instrucción le indique que pare, o bien tenga un error fatal en la ejecución.

Se llama *JDK* (*Java Development Kit*) al conjunto de herramientas para desarrollar y probar programas escritos en el lenguaje de programación Java. Entre ellos se encuentra

el compilador `javac` y el generador de documentación `javadoc`. La JVM (*Java Virtual Machine*) es el ambiente virtual donde se ejecuta el bytecode y contiene al intérprete `java`.

Repositorios de código

La mejor forma de no perder tu código es tener una copia en internet. Por este motivo, utilizarás `git` para guardar y entregar tu trabajo.

Actividad 1.1

Si estás siguiendo este manual como parte de un curso, se te asignará un repositorio con una dirección personalizada. Si estás siguiendo este manual por tu cuenta deberás crear una copia del repositorio indicado arriba (Código auxiliar 1.1) y trabajar con tu copia. Abre una terminal (o consola de comandos) y clona tu repo en el directorio donde desees trabajar, ingresa a la carpeta correspondiente. Los comandos para clonar un repositorio e ingresar a su directorio son de la forma:

```
$ git clone <url>
$ cd <nombre-del-repo>
```

Por ejemplo, si la dirección `url` es:

```
https://github.com/computacion-ciencias/icc-cei-any
```

deberás ejecutar:

```
$ git clone https://github.com/computacion-ciencias/icc-cei-any
$ cd icc-compiladoreseinterpretes-any
```

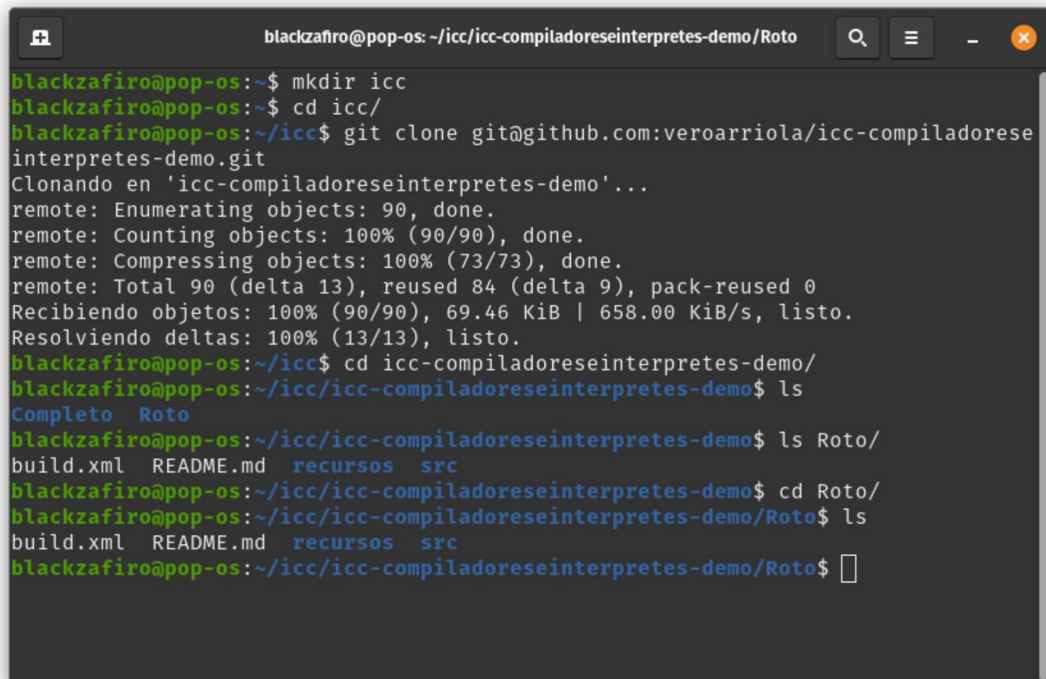
La figura Figura 1.1 muestra cómo se puede ver tu consola al realizar estas actividades, observa que el nombre del repositorio debe ser distinto.

Desarrollo

Los programas de la JDK

En el interior encontrarás dos carpetas: `Roto` y `Completo`. La primera parte de esta práctica se realizará con los archivos que se encuentran dentro de `Roto`.

1. Compiladores e intérpretes

A terminal window titled 'blackzañiro@pop-os: ~/icc/icc-compiladoreseinterpretes-demo/Roto'. The user runs 'mkdir icc', 'cd icc/', and 'git clone git@github.com:veroarriola/icc-compiladoreseinterpretes-demo.git'. The output shows the cloning progress: 'Clonando en \'icc-compiladoreseinterpretes-demo\'...', 'remote: Enumerating objects: 90, done.', 'remote: Counting objects: 100% (90/90), done.', 'remote: Compressing objects: 100% (73/73), done.', 'remote: Total 90 (delta 13), reused 84 (delta 9), pack-reused 0', 'Recibiendo objetos: 100% (90/90), 69.46 KiB | 658.00 KiB/s, listo.', 'Resolviendo deltas: 100% (13/13), listo.'. Then the user runs 'cd icc-compiladoreseinterpretes-demo/' and 'ls', showing 'Completo Roto'. Finally, the user runs 'cd Roto/' and 'ls', showing 'build.xml README.md recursos src'.

```
blackzañiro@pop-os: ~$ mkdir icc
blackzañiro@pop-os: ~$ cd icc/
blackzañiro@pop-os: ~/icc$ git clone git@github.com:veroarriola/icc-compiladoreseinterpretes-demo.git
Clonando en 'icc-compiladoreseinterpretes-demo'...
remote: Enumerating objects: 90, done.
remote: Counting objects: 100% (90/90), done.
remote: Compressing objects: 100% (73/73), done.
remote: Total 90 (delta 13), reused 84 (delta 9), pack-reused 0
Recibiendo objetos: 100% (90/90), 69.46 KiB | 658.00 KiB/s, listo.
Resolviendo deltas: 100% (13/13), listo.
blackzañiro@pop-os: ~/icc$ cd icc-compiladoreseinterpretes-demo/
blackzañiro@pop-os: ~/icc/icc-compiladoreseinterpretes-demo$ ls
Completo  Roto
blackzañiro@pop-os: ~/icc/icc-compiladoreseinterpretes-demo$ ls Roto/
build.xml  README.md  recursos  src
blackzañiro@pop-os: ~/icc/icc-compiladoreseinterpretes-demo$ cd Roto/
blackzañiro@pop-os: ~/icc/icc-compiladoreseinterpretes-demo/Roto$ ls
build.xml  README.md  recursos  src
blackzañiro@pop-os: ~/icc/icc-compiladoreseinterpretes-demo/Roto$
```

Figura 1.1 Clonando un repositorio.

Actividad 1.2

Usa el comando `cd` para entrar al directorio `Roto/src` y una vez dentro ejecuta el comando siguiente:

```
1 $ javac icc/mundo/Usopersonaje.java
```

Este comando manda llamar al programa compilador `javac` y le solicita que compile al archivo `Usopersonaje.java` y a todos aquellos que éste manda llamar. El código de este paquete tiene unos pocos errores, por lo que no logra terminar la traducción. En lugar de ello, te envía los mensajes que estás viendo indicando dónde se perdió y sugiriendo cuál podría ser el error. Algunos errores son sencillos y para resolverlos sólo tienes que seguir las sugerencias del compilador, otros no son tan fáciles de inferir, sólo el programador (o sea tú) puede saber dónde está lo que hay que arreglar.

Todos los errores de este ejercicio son de los fáciles. Por ejemplo, lee con cuidado la línea:

```
1 icc/mundo/Usopersonaje.java:36: error: ';' expected
2     personaje.salta()
```

3

Literalmente te está diciendo que falta un `;` después de `salta()` en la línea de código 36, del archivo `UsoPersonaje.java`. Es el error gramatical más sencillo que se puede cometer.

Abre ahora ese archivo en algún editor de texto, de preferencia alguno con funcionalidad para editar código. La Figura 1.2 muestra cómo se puede ver uno de estos editores. No uses una IDE (*Integrated development environment*)^a, no es necesario usar un programa tan complejo y pesado para lo que vas a hacer en esta práctica y sólo causaría confusión.

Una vez en el editor ve a la línea 36, bastará con agregar un punto y coma al final para arreglar este error. Ojo, aunque en tu terminal apareció otro mensaje de error parecido (es el tercero), no intentes aún hacer nada al respecto hasta que hayas atendido el segundo mensaje. Este puede ser un poco más complicado de entender:

```
1 icc/mundo/UsoPersonaje.java:38: error: not a statement
2      personaje.espera);
```

Traduciremos *statement* como *enunciado*. Este error te está diciendo que lo que está escrito en la línea 38 no tiene la forma de un enunciado válido de Java. Aunque no tengas conocimiento de Java, trata de detectar el patrón. Lee con cuidado el código, notarás cierta regularidad en la forma en que se escriben los comandos. Deduce qué está mal con la línea 38 y trata de arreglarlo.

Vuelve a invocar al compilador como hiciste anteriormente. Ya no debería haber errores, si no es el caso regresa al paso anterior e intenta arreglar el problema de otro modo hasta que logres compilar sin errores. Observa que, de los tres mensajes de error originales, sólo fue necesario atender a dos: el tercer mensaje también era consecuencia del segundo error.

^aLas IDEs son ambientes para desarrollo con funciones muy ricas y complejas para automatizar parte del trabajo del programador, pero así como no es seguro aprender a manejar en un auto deportivo, empezar a programar usando una de estas puede hacer que todo parezca más complicado de lo que es, o más simple de lo que debería, según sea el caso.

Acostúmbrate entonces, cuando compiles tu código y el compilador reporte errores, a leerlos cuidadosamente de arriba hacia abajo, velos resolviendo uno por uno y compila en cada ocasión. Frecuentemente arreglar un error también resuelve otros reportados después de él.

Actividad 1.3

Una vez que ya no haya errores, notarás que aparecieron archivos con terminación `.class` dentro del directorio `icc/mundo`. Notarás que esos archivos tienen los mis-

1. Compiladores e intérpretes



```
1 /**
2  * Se permite acceder a este código para uso personal, con fines didácticos.
3  * No está permitido realizar trabajos derivados directamente de él o
4  * publicar soluciones a los problemas aquí planteados, para
5  * no afectar a otros estudiantes que lo utilicen en sus cursos.
6  */
7
8 package icc.mundo;
9
10 import java.io.IOException;
11
12 /**
13  * Clase que muestra cómo pedir a un personaje que realice acciones en
14  * un mundo sencillo.
15  */
16 public class UsoPersonaje {
17
18     /**
19      * Punto de entrada al programa.
20      * @param args
21      * @throws IOException
22      */
23     public static void main(String[] args) {
24
25         // Crea un objeto de tipo Mundo
26         Mundo mundo = new Mundo();
27
28         // Crea un personaje con forma de Cyborg.
29         Personaje personaje = new Personaje("Cyborg");
30
31         // Agrega el personaje a la ventana.
32         mundo.muestraPersonaje(personaje);
33
34         // Instrucciones para que las ejecute el personaje.
35         personaje.corre();
36         personaje.salta();
37         personaje.corre();
38         personaje.espera();
39     }
40 }
```

Figura 1.2 Editor de texto con formato para código. Las palabras reservadas del lenguaje, entre otros elementos de interés, se resaltan con colores.

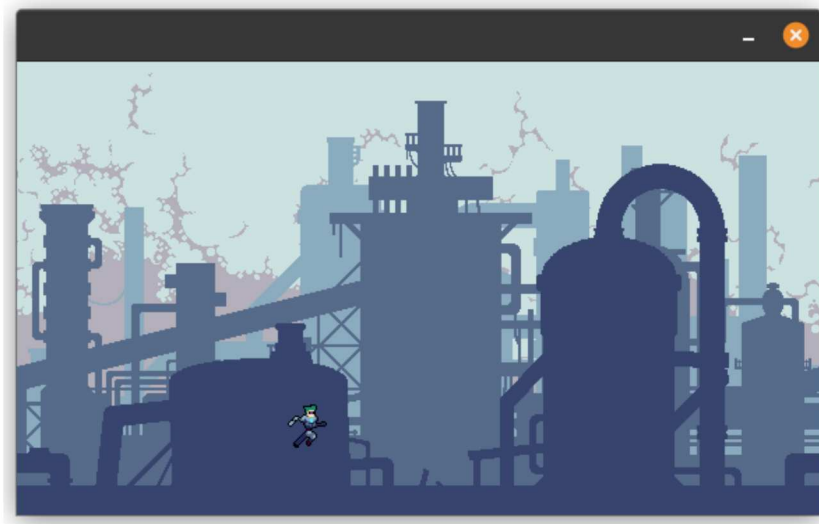


Figura 1.3 Programa en ejecución.

mos nombres que los archivos `.java` originales, casi siempre. Estos nuevos archivos contienen *bytecode*. Si intentas abrirlos con un editor de texto obtendrás una serie de símbolos ininteligibles, eso si no trabas a tu editor.

Entregable: Escribe exacta y explícitamente qué archivos fueron creados y dónde.

Actividad 1.4

Aún dentro del directorio `Roto/src`, ahora invoca a la máquina virtual de Java para que interprete el código que generaste.

```
1 $ java icc.mundo.UsoPersonaje
```

Deberás ver una ventana como la de la Figura 1.3 mostrando una animación sencilla, mientras que en la terminal se imprimen las acciones que está realizando. Deberás cerrar la ventana para que termine la ejecución del programa.

Actividad 1.5

Intenta invocar a la máquina virtual con los nombres de otros archivos `.class`, a lo más un par serán suficientes. ¿Qué sucede? Lee lo que devuelve la consola y abre los archivos `.java` correspondientes que necesites.

Entregable: Describe qué sucedió al intentar lo anterior. ¿Qué tiene el archivo `UsoPersonaje.java` que permite invocar su `.class` con `java`?

1. Compiladores e intérpretes

Actividad 1.6

Aún dentro del directorio `src`, ahora crearemos un archivo comprimido con sólo el código ejecutable y las imágenes que usa el programa:

```
1 $ jar cvfe Mundo.jar icc.mundo.UsoPersonaje icc/mundo/*.class -C
   ↪ .. recursos
```

Puedes ejecutar el programa almacenado en el `.jar` con:

```
1 $ java -cp Mundo.jar icc.mundo.UsoPersonaje
```

La opción `-cp` es una abreviatura de `classpath` y se usa para indicar a Java dónde buscar los archivos `.class`.

Este archivo comprimido ya puede ser utilizado en otra computadora que tenga la máquina virtual, por lo que puedes usar este formato para distribuir los programas que hagas.

Para más información sobre el uso de `jar`, visita el [tutorial oficial](#).

Actividad 1.7

Finalmente, ejecuta el comando siguiente:

```
1 $ javadoc icc.mundo
```

Esto generará una serie de archivos `.html` en el directorio donde te encuentras y junto a los archivos `.java` dentro de `icc/mundo`. Desde tu navegador de archivos puedes usar el navegador de internet para abrir el que se llama `index.html`. ¿Qué observas? Aquí están todos los comentarios escritos entre `/** */` en los archivos de código de las clases `public` con las que estás trabajando, pero en un formato más amigable para el lector. Esta es la documentación que le darás a tus usuarios cuando entregues tus trabajos.

Usando una herramienta auxiliar: ant

Aunque generaste todo lo necesario, el código compilado y la documentación se mezcló con los archivos con el código fuente. Al entregar un trabajo esto se ve desordenado y también te hará a ti más complicado identificar y encontrar tus archivos. Además, al aumentar la complejidad de tu proyecto, la línea de comandos se vuelve más complicada de manejar. **Borra todos esos archivos extra antes de proseguir a la siguiente**

sección. Jeje, está bien, no te abandonaremos con la tarea. Podrías borrar los archivos uno por uno con un navegador de archivos, pero sería tedioso y tendrías que fijarte bien para no borrar los archivos de código. Bueno, igualmente deberás tener cuidado, pero los comandos siguientes ayudarán:

Actividad 1.8

Estado aún dentro del directorio `src` ejecuta:

Listado 1.1: Borrando la documentación

```
1 $ rm *.html *.css *.js *.zip
2 $ rm element-list
3 $ rm -r jquery/
4 $ rm -r resources/
5 $ rm icc/mundo/*.html
```

Listado 1.2: Borrando el código generado

```
1 $ rm Mundo.jar
2 $ rm icc/mundo/*.class
```

Si bien es posible arreglar este desorden aún sólo con los comandos de Java es mucho más sencillo utilizar una herramienta auxiliar: `ant`.

El programa `ant` utiliza un archivo de configuración llamado `build.xml`, que le indica cómo realizar las acciones que le vamos a solicitar. Habrás notado la presencia de uno de estos archivos dentro de la carpeta `Roto`, ahora lo vas a utilizar.

El archivo `build.xml` está escrito en *lenguaje de anotación extensible* (*eXtensible Markup Language XML*), este formato permite organizar información por medio de *etiquetas* de la forma `<etiqueta>` y sirve para muchos usos ¹. Su elemento distintivo es el uso de estas etiquetas anidadas para abrazar piezas de información: cada bloque inicia con una etiqueta de apertura `<etiqueta>` y al final se concluye con la de cierre `</etiqueta>`. En general, es posible poner más información o etiquetas dentro de cada par de etiquetas, donde la etiqueta que contiene a todas las demás se llama *etiqueta raíz*. Así mismo, una etiqueta también puede tener *atributos*. Un ejemplo genérico de esta gramática se muestra en el Listado 1.3.

Listado 1.3: Forma genérica de un archivo XML

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <etiqueta atributo="valor" atributo2="valor2">
3   <etq otroatributo="otrovalor">
4     </etq>
```

¹Valdés y Gurovich 2008

1. Compiladores e intérpretes

```

5  <etq>
6  <!-- Comentario para explicar algo -->
7  </etq>
8  <!-- Abre y cierra etiqueta en versión abreviada -->
9  <etq />
10 </etiqueta>

```

Para `ant` se ha definido un conjunto de etiquetas con el propósito de indicar procesos del JDK como *compilar*, *ejecutar* y *borrar archivos auxiliares*. El Listado 1.4 muestra un fragmento del archivo `build.xml`. La etiqueta que contiene a todas las demás, en este caso `<project>`, es la etiqueta *raíz* de este archivo. La etiqueta `project` tiene dos atributos que nos interesan particularmente: `name` que será el nombre de nuestro proyecto de Java y `default` que nos servirá para indicar la acción que realizará `ant` por defecto.

Listado 1.4: Fragmento de `build.xml`

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <project name="práctica" default="compile" basedir=".">
3    <target name="compile">
4      ...
5    </target>
6
7    <target name="run" depends="compile">
8      ...
9    </target>
10
11    ...
12 </project>

```

Las acciones que realizará `ant` se configuran mediante etiquetas `<target>`, donde se indica cómo invocar a los programas de la JDK. De esta manera, en el archivo `build.xml` se definen los comandos que tienes disponibles para trabajar con el proyecto que lo contiene. En este caso son:

<code>compile</code>	Compila la práctica.
<code>run</code>	Ejecuta la práctica, compilándola si no ha sido compilada.
<code>doc</code>	Genera la documentación JavaDoc de la práctica.
<code>clean</code>	Limpia la práctica de <i>bytecode</i> , documentación, o ambos.

Actividad 1.9

Regresa al directorio `Roto` justo antes de `src`. Si aún te encuentras en `src` lo puedes hacer con el comando `cd` pidiéndole que te transfiera al directorio padre:

```
1 $ cd ..
```

Ahora compilarás tu código desde ahí usando `ant`. Observa que el archivo `build.xml`

está justo a un lado. Escribe:

```
1 $ ant compile
```

Como ya corregiste los errores en `UsoPersonaje.java`, el proceso debe terminar sin errores. Notarás que se crearon algunas cosas.

Entregable: ¿Dónde quedaron ahora los archivos `.class`?

Actividad 1.10

Ejecuta tu código con

```
1 $ ant run
```

¿Qué sucede?

Entregable: ¿A qué comando de los que usaste en la sección anterior, sin `ant`, correspondería?

Actividad 1.11

¿Recuerdas el desorden que provocó la documentación anteriormente? Prueba ahora con

```
1 $ ant doc
```

Actividad 1.12

El comando siguiente removerá todo lo que se generó al compilar con `ant` y será tu mejor amigo en el futuro.

```
1 $ ant clean
```

Después ejecuta de nuevo `ant compile` observando en la terminal el número de archivos que compila.

Una vez hecho esto ejecuta:^a

```
1 $ touch src/icc/mundo/UsoPersonaje.java
```

Si quieres conocer más detalles sobre lo que hace `touch` escribe:

1. Compiladores e intérpretes

```
1 $ man touch
```

Ahora compila de nuevo con `ant compile` y observa cuántos y cuáles archivos se compilan. ¿Qué notas diferente?

Entregable: ¿Qué mecanismo crees que utilice Ant para determinar qué compilar?

^aEl equivalente en Windows es `copy /b src\icc\mundo\UsoPersonaje.java +`

Como seguramente compilarás a menudo, pusimos a `compile` como la tarea por defecto. Entonces puedes compilar escribiendo simplemente:

```
1 $ ant
```

Pero si prefieres alguna otra tarea, sólo debes modificar el atributo `default` de la etiqueta `project` en el archivo `build.xml`.

Estructura de un programa

Para esta parte utilizarás el código dentro de la carpeta `Completo`. El objetivo de esta sección es explicar cómo luce un programa sencillo en Java y descifrar algo del código que viste en los archivos de los ejercicios anteriores.

Juntos, el compilador `javac` y el intérprete `java` transforman el código que escribiste en secuencias de comandos que la computadora puede ejecutar como un programa.

Los programas de Java son muy parecidos a los programas que utilizas en la consola de Linux, como `ls`, `cd`, `pwd`, `more`, etc. En particular, también les puedes enviar parámetros al invocarlos, como harías al llamar `ls -al` o `diff archivo.txt archivo2.txt`.

Actividad 1.13

Entra a la carpeta `Completo` e invoca `ant`. Te darás cuenta de que generó un archivo extra: `Mundo.jar`. Ignóralo por el momento y entra a la carpeta `build`. Usa los conocimientos adquiridos en las secciones anteriores para ejecutar desde aquí el programa `Mundo`. ¿Qué mensaje aparece?

Ahora ejecuta:

```
1 $ java icc.mundo.UsoPersonaje Punk
```

Entregable: ¿Qué obtienes cuando no pasas argumentos? ¿Qué obtienes cuando

agregas alguno de los personajes sugeridos (cuidado el uso de mayúsculas y minúsculas importa)?

Los archivos `.class` que se encuentran dentro de `build` fueron empaquetados en el archivo `Mundo.jar`, por lo que es posible ejecutar el mismo programa utilizando sólo el `.jar`.

Actividad 1.14

Regresa al directorio que contiene el archivo `Mundo.jar` y ejecuta:

```
1 $ java -jar Mundo.jar Punk
```

Entregable: Prueba con diferentes argumentos y reporta lo que hace el programa.

Actividad 1.15

Ahora abre el código en `src/icc/mundo/UsoPersonaje.java` y lee cuidadosamente su documentación. Modifica el texto que indica las opciones a utilizar. Tomaremos como válida cualquier modificación que hagas al texto que aparece cuando no envías argumentos.

Entregable: Tendrás que entregar el archivo con las modificaciones requeridas en esta actividad como parte de esta práctica. Comenta tus ejemplos para que recuerdes qué hace cada modificación.

Actividad 1.16

Lee los dos archivos `build.xml` utilizados en esta práctica y observa en qué se parecen y en qué difieren.

Entregable: ¿Qué objetivos (*target*) reconoce cada archivo? ¿Qué pasos ejecutará cada uno de los objetivos (observa el atributo llamado *depends*)?

Entregables

Lo que deberás entregar para esta práctica es:

1. Archivo `UsoPersonaje.java` corregido.

1. Compiladores e intérpretes

2. Archivo `src/icc/entrada/Entrada.java`, documenta los ejemplos que agregaste, tanto en línea con `//` cuando sea necesario, y en comentarios de `javadoc` para funciones y la clase. No es necesario entregar la carpeta `doc` porque esa la podemos generar con `ant doc`.
3. Respuestas a las actividades:
 - a) 1.3
 - b) 1.5
 - c) 1.9, 1.10 y 1.12
 - d) 1.13 y 1.14
 - e) 1.16
4. La respuesta, con su justificación, a la siguiente pregunta: Los errores que tuviste que corregir, ¿de qué tipo son, sintácticos o semánticos?