Tecnología de la Programación I

Presentación de la Práctica 2 (parte III)

(Basado en la práctica de Alberto Núñez y Miguel V. Espada)

Ana M. González de Miguel (ISIA, UCM)

Índice

1. Serialización del Juego

- El término **serialization** (serialización o secuenciación) se refiere a la conversión del estado de ejecución de un programa, o parte del programa, en un flujo de bytes, habitualmente con el objetivo de guardarlo en fichero. El término **deserialization** se refiere al proceso inverso de reconstruir el estado de ejecución de un programa, o parte del programa, a partir de un flujo de bytes.
- Aquí nos interesa producir un flujo de bytes que representa el estado actual del juego — no nos hace falta representar el estado completo — con el objetivo de escribir este estado en, y leer este estado de, un fichero de texto.
- Los términos **stringification/destringification** para referirse a la serialización /deserialización que trabaja con un flujo de texto.

- ✓ En esta versión de la práctica vamos a permitir dos modos de pintado: **formatted** y **stringified**.
- ✓ El modo formatted es el que vimos en la práctica anterior, donde el juego se muestra como un tablero, mientras que en el modo stringified mostramos más información como texto plano sin formato.
- ✓ Esta nueva forma de pintado nos servirá para hacer debug del juego y, como veremos en la siguiente sección, para poder salvar el juego en un fichero.

- ✓ También crearemos un nuevo comando *stringify* que envía el estado del juego serializado como texto a la salida estándar para que se muestre en pantalla. El formato de esta serialización como texto será el siguiente:
  - La primera línea será siempre — *Space Invaders v2.0* —
  - La segunda línea es una línea en blanco
  - La tercera línea dará información sobre el *Game*: *G;totalCycles*
  - La cuarta línea dará información de *Level*: *L;level*
  - Después, en cada línea, daremos información sobre los objetos de juego que estén en el tablero:
    - Ovni: *O;x,y;live*
    - Regular alien: *R;x,y;live;cyclesNextAlienMove;dir*
    - Destroyer alien: *D;x,y;live;cyclesNextAlienMove;dir*
    - Explosive alien: *E;x,y;live;cyclesNextAlienMove;dir*

- Bomb: *B;x,y*
- Missile: *M;x,y*
- Supermissile: *X;x,y*
- UCMShip (player): *P;x,y;live;points;superpower;missiles*

✓ Este comando *stringify* no cambia el estado del juego. Simplemente vuelca la información en la consola. Por eso su metodo *execute* devuelve false.

✓ Para implementar esta funcionalidad se recomienda hacer los siguientes cambios.

- ✓ Primero, renombraremos el *GamePrinter* de la práctica anterior a *BoardPrinter* y haremos que herede de una nueva clase abstracta *GamePrinter*. Esta clase abstracta tiene un único atributo *Game* y un método mutador *setGame*.
- ✓ Segundo, programamos la nueva clase *Stringifier* que también heredará de la clase abstracta *GamePrinter*. Esta clase necesita una constructora con un parámetro *Game* y un método *toString* que llame a *game.stringify()*. Este método de *Game* tendrá que poner información en un String y añadir *board.stringify()* desde donde se llame al método *stringify* de cada objeto.
- ✓ Tercero, para desacoplar la vista de la lógica de la aplicación haremos que el *Game* no dependa del *GamePrinter*, así que eliminaremos cualquier referencia a *GamePrinter* en la clase *Game*.

- ✓ Cuarto, en cada bucle del juego, cuando haya que repintar, el controlador ejecutará la instrucción:

```
System.out.println(printer);
```

- ✓ El tipo declarado del atributo *printer* de la clase *Controller* será *GamePrinter y*, normalmente, el tipo real será *BoardPrinter* para que la impresión salga como en la primera práctica.
- ✓ Finalmente, crearemos un nuevo comando para implementar la serialización *StringifyCommand* que añadiremos a la lista de comandos disponibles. El método *execute* de la clase *StringifyCommand* contendrá la misma línea de código que el controlador pero esta vez, el tipo real será *Stringifier*.

- ✓ Para gestionar los diferentes *printers*, podríamos crear una clase *PrinterGenerator*, muy parecido al *CommandGenerator* utilizado para gestionar los comandos.
- ✓ Sin embargo, en este caso, al contrario del caso de los comandos, no es el usuario externo quien proporciona el tipo de *printer* que se quiere usar y, por tanto, esta información no viene en forma de cadena de caracteres que se tendrá que pasar a un método *parse*.
- ✓ Vamos a gestionar los diferentes *printers* con un *enum*, cuyo código se proporciona en los anexos:

```
public enum PrinterTypes {
   BOARDPRINTER("boardprinter", "prints the game formatted as a board
of dimension: ", new BoardPrinter()),
   STRINGIFIER(“stringifier", "prints the game as plain text", new
Stringify());

   ...
   private GamePrinter printerObject;

private PrinterTypes(String name, String text, GamePrinter printer) {
   ...
}
public static String printerHelp(Game game) {
   ...
}
public GamePrinter getObject() {
   return printerObject;
}
}
```

- Esta clase se usará en las clases *Controller* y *StringifyCommand* para seleccionar el tipo de printer e imprimir el estado del juego con el formato correspondiente. Aprovechando la existencia del método *printerHelp*, vamos a añadir un nuevo comando *ListPrintersCommand* que nos liste los printers disponibles, de la siguiente manera:

```
Command > listPrinters
boardprinter : prints the game formatted as a board of dimension: 8x9
stringifier : prints the game as plain text
```

- El método *execute* de este comando simplemente hace la siguiente llamada y devuelve false porque no cambia el estado del juego:

```
System.out.println(PrinterTypes.printerHelp(game));
```

2. Manejo de Excepciones

- ✓ El tratamiento de excepciones en Java resulta muy útil para controlar determinadas situaciones del juego en tiempo de ejecución como, por ejemplo, mostrar información relevante al usuario sobre la ejecución de un comando.
- ✓ En la práctica anterior, cada comando invocaba el método correspondiente – de la clase *Controller* o de la clase *Game* – para poder llevar a cabo las operaciones necesarias.
- ✓ Por ejemplo, en el caso de no poder disparar el misil, se devolvía un valor de tipo boolean con valor false. De esta forma, el juego podía controlar que el misil no se volvía a añadir, **pero no se indicaba el motivo exacto**.

- ✓ En esta práctica vamos a trabajar con el manejo de excepciones, de forma que cada clase pueda lanzar y procesar determinadas excepciones para tratar determinadas situaciones durante el juego.
- ✓ En algunos casos, estas situaciones consistirán únicamente en proporcionar un **mensaje** al usuario, mientras que en otras su tratamiento será más complejo.
- ✓ Cabe destacar que en esta sección no se van a tratar las excepciones relativas a los ficheros, las cuales serán explicadas en detalle en la sección siguiente.

- ✓ Inicialmente se van a manejar las excepciones lanzadas por el sistema, es decir, aquellas que no son creadas ni lanzadas por el usuario. Al menos, debe tratarse la siguiente excepción:
  - *NumberFormatException*, que será lanzada cuando ocurra un error al transformar un número de formato *String* a formato *int*.
- ✓ Además, se deberán crear dos excepciones: *CommandParseException* y *CommandExecuteException*.
- ✓ La primera de ellas tratará los errores producidos al parsear un comando, es decir, aquéllos producidos durante la ejecución del método *parse*, tales como "comando desconocido" o "parámetros incorrectos".
- ✓ La segunda se utilizará para tratar las situaciones de error al ejecutar el método *execute* de un comando como, por ejemplo, que la nave del usuario no puede desplazarse en la dirección introducida por el usuario.

- ✓ Una de las principales modificaciones que realizaremos al incluir el manejo de excepciones en el juego consistirá en eliminar la comunicación directa entre los comandos y el controlador.
- ✓ De esta forma, no será necesario que cada comando indique al controlador cuándo se ha producido un error o cuándo se debe actualizar el tablero, sino que bastará con controlar las excepciones que puedan producirse durante la ejecución del comando.
- ✓ Además, puesto que ahora se van a tratar las situaciones de error tanto en el procesamiento como en la ejecución de los comandos, los mensajes de error mostrados al usuario serán mucho más descriptivos que en la práctica anterior.

- ✓ Básicamente, los cambios a realizar serán los siguientes:
  - Asegurar que el controlador puede poder capturar las excepciones lanzadas por los métodos *execute* y *parse* de la clase *Command*.
  - Incluir el tratamiento de excepciones en las clases correspondientes para que puedan ser capturadas por el controlador y el mensaje correspondiente impreso por pantalla. Recuerda que no hace falta incluir las excepciones de sistema que son subclases de *RuntimeException* (p.ej. *NumberFormatException*) en las cláusulas throws.
- ✓ Ahora todos los mensajes se imprimen desde el método *run* del controlador cuyo cuerpo se asemejará al código siguiente:

```
printGame();
while (!game.isFinished()){
   System.out.print(prompt);
   String[] words = in.nextLine().trim().split( "\\s+");
   try {
      Command command = CommandGenerator.parseCommand(words);
      if (command != null) {
         if (command.execute(game))
            printGame();
      } else
         System.out.println(unknownCommandMsg);
   } catch (CommandParseException | CommandExecuteException ex {
      System.out.format(ex.getMessage() + " %n %n");
   }
}
```

- ✓ **Mejorar la calidad del código**.
  - Una excepción pasa por una cadena de llamadas desde el método donde se lanza hasta el método donde se recoge. Surge la pregunta: ¿en cuál de los métodos se debería lanzar y en cuál recoger? Por ejemplo, supongamos que un método m1 llama a un método m2 que llama a un método m3 y se puede producir un error en la ejecución del método m3 que queremos tratar en el método m1. Cuando ocurre el error en cuestión, hay dos políticas posibles:
    - m3 devuelve una indicación de error (no lanza una excepción), digamos el valor false, a m2, m2 lanza una excepción al recibir el valor de retorno false de m3, m1 recoge la excepción.
    - m3 lanza una excepción cuando ocurre el error, m2 no recoge la excepción, m1 recoge la excepción.
  - La implementación más coherente con la filosofía de las excepciones es la segunda. Observa que en consecuencia no se debe devolver un valor booleano cuando ocurre un error sino lanzar una excepción, lo que implica modificar el interfaz *IPlayerController*.

- Hay circunstancias en las que es buena práctica recoger una excepción de bajo nivel para a continuación lanzar una excepción de alto nivel que envuelve la de bajo nivel (y que contiene un mensaje menos específico que el de la excepción de bajo nivel).
- Si ya has implementado la segunda opción del punto anterior, sería buena práctica lanzar una excepción de bajo nivel donde ocurre un error en la ejecución de un comando y recogerla en el método *execute*() del comando correspondiente para a continuación lanzar un *CommandExecuteException* que la envuelve. En este caso hay que hacer una constructura en *CommandExecuteException* en la que además del mensaje se pase la excepción que envuelve (la causa del error).

- ✓ **Ejemplos de ejecución**. En esta sección se muestran algunos ejemplos que tratan las excepciones anteriormente descritas:
  - La ejecución con un número incorrecto de parámetros (0 o más que 2) produce el mensaje siguiente:

    ```
    Usage: Main <EASY|HARD|INSANE> [seed]
    ```

  - La ejecución con el parámetro *difficult* produce el mensaje siguiente:

    ```
    Usage: Main <EASY|HARD|INSANE> [seed]: level must be one of:
        EASY, HARD, INSANE
    ```

  - La ejecución con los parámetros *easy XL* produce el mensaje siguiente (después de capturar una *RuntimeException*):

    ```
    Usage: Main <EASY|HARD|INSANE> [seed]: the seed must be a number
    ```

- ✓ Otros ejemplos son:

```
Command > shockwave

Failed to shoot

Cause of Exception: pr2.exceptions.NoShockwaveException: Cannot
release shockwave: no shockwave available

Command > shoot

Failed to shoot

Cause of Exception: pr2.exceptions.MissileInFlightException: Cannot
fire missile: missile already exists on board

Command > move left 1

Failed to move

Cause of Exception: pr2.exceptions.OffWorldException: Cannot perform
move: ship too near border
```

3. Ficheros

- ✓ Seguidamente, añadiremos una nueva funcionalidad que consistirá en poder guardar y cargar partidas almacenadas en ficheros de texto.
- ✓ Para permitir guardar el estado de una partida en un fichero, así como cargar el estado de una partida previamente guardada, se van a crear dos comandos nuevos, *SaveCommand* y *LoadCommand*, tal que:
  - *save fileName* guardará el estado actual de la partida en el fichero *fileName*. Hay que tener en cuenta que *fileName* es 'case-sensitive'.
  - *load fileName* cargará la partida almacenada en el fichero *fileName*.

- ✓ Se debe tener en cuenta que durante una misma partida se pueden guardar varios estados del juego, es decir, que se debe poder permitir al usuario guardar varios estados de la partida durante el transcurso de la misma utilizando, o no, ficheros diferentes.
- ✓ En el caso de que ya exista un fichero con el mismo nombre, se sobrescribirá el mismo.
- ✓ Vamos a utilizar la funcionalidad de *Stringifier* que hemos descrito en el apartado anterior para guardar la información en un fichero.
- ✓ El manejo de ficheros implica el uso de nuevos tipos de **excepciones**, tanto del sistema, como las definidas por el usuario.

- Por ello, se deberá crear una nueva excepción *FileContentsException* y lanzarla en caso de detectar un problema en el contenido del fichero, como por ejemplo, leer un level desconocido.
- En Java existen muchos mecanismos para manejar ficheros. En esta práctica usaremos **flujos de caracteres** en lugar de flujos de bytes.
- En particular, recomendamos el uso de *BufferedWriter* y *FileWriter* para escribir en un fichero, y *BufferedReader* y *FileReader* para leer de un fichero.
- Además, se recomienda el uso de bloques **try-with-resources** para el código donde se abre el fichero, capturando *IOException* en cada uno de los métodos *execute*() de las clases *LoadCommand* y *SaveCommand*.

- ✓ Para poder guardar el estado de una partida:
  - El método *execute*() de la clase *SaveCommand* debe:
    - Para simplificar, no hace falta comprobar que el texto proporcionado por el usuario pueda ser un nombre de fichero válido, donde esta noción depende del sistema operativo, ni que el programa tenga permisos de escritura. Si ocurre esto, el sistema lanzará una excepción.
    - Añadir la extensión '.dat' al nombre sin extensión proporcionado por el usuario. Si un fichero con este nombre ya existe, se sobreescribirá.
    - Crear un *FileWriter* y un *BufferedWriter* que lo envuelve.
    - Utilizar el *Stringifier* para generar un *String* con los datos del juego y volcarlos en el fichero.
  - En caso de éxito, se debe imprimir el mensaje:

```
"Game successfully saved in file
    <nombre_proporcionado_por_el_usuario>.dat. Use the load
    command to reload it"
```

- ✓ Para poder cargar el estado de una partida:
  - El método *execute*() de la clase *LoadCommand* debe:
    - Para simplificar, no hace falta comprobar que el texto proporcionado por el usuario pueda ser un nombre de fichero válido, donde esta noción depende del sistema operativo, ni que el fichero se pueda leer. Si ocurre alguno de estos problemas al cargar un fichero, el sistema lanzará una excepción. Para tratar el caso de un nombre de fichero que no corresponde a ningún fichero, se puede recoger la excepción *FileNotFoundException* un tipo de *IOException*.
    - Crear un *FileReader* y un *BufferedReader* que lo envuelve.
    - Leer la cabecera, que consta del mensaje "— Space Invaders v2.0 —". En un contexto real, probablemente también aceptaría cabeceras de algunas versiones anteriores.
    - Leer una línea en blanco.
    - Invocar un método llamado *load*() de la clase *Game*, pasándole el *BufferedReader*

- En el cuerpo del método *load* de la clase *Game* se lee y se almacena el número de ciclos y el nivel.
- A continuación, el método *load* de la clase *Game* contiene el código siguiente:

```
loading = false;
line = inStream.readLine().trim();
while(line != null && !line.isEmpty() ) {
   GameObject gameObject = GameObjectGenerator.parse(line, this,
verifier);
   if (gameObject == null) {
       throw new FileContentsException("invalid file, " +
"unrecognised line prefix");
   }
   board.add(gameObject);
   line = inStream.readLine().trim();
}
```

- El parecido entre este fragmento de código y la parte del código del método *run* de la clase *Controller* en la que se llama al método *parseCommand* de la clase *CommandGenerator* no es accidental. El parecido resulta más evidente al ver el código de la clase *GameObjectGenerator*:

```
public class GameObjectGenerator {

   private static GameObject[] availableGameObjects = {
     new UCMShip(),
     new Ovni(),
     new RegularAlien(),
     new DestroyerAlien(),
     new ExplosiveAlien(),
     new ShockWave(),
     new Bomb(),
     new Missile(),
     new Supermissile()
  };
```

```
public static GameObject parse(String stringFromFile, Game game,
FileContentsVerifier verifier) throws FileContentsException {
    GameObject gameObject = null;
    for (GameObject go: availableGameObjects) {
       gameObject = go.parse(stringFromFile, game, verifier);
       if (gameObject != null)
          break;
    }
    return gameObject;
}
```

- ¿Para qué sirve el atributo *loading* que aparece en el código del método *load* de la clase *Game*? Si una subclase de *GameObject* quiere reinicializar atributos estáticos cuando se carga un estado de juego desde fichero, querrá saber cuándo empieza la carga de las instancias de esta clase. Esto puede conseguirse manipulando el valor del atributo *loading* mediante unos métodos que debe proporcionar el *game* para este fin.
- En caso de ocurrir un problema a la hora de cargar un fichero, el método *execute*() de la clase *LoadCommand* debe recoger la excepción lanzada y envolverla en un *CommandExcecuteException*.
- El método *load*() de *Game*, así como los métodos llamados desde este método, pueden comprobar la coherencia de los datos leídos y, en caso de haber un problema, lanzar un FileContentsException, que será recogida por el método *execute*() de la clase *LoadCommand*. Para facilitar esta tarea, en los anexos proporcionamos el código de la clase *FileContentsVerifier*.

- En caso de haber podido cargar el estado del juego de fichero con éxito, se debe imprimir el mensaje:

```
"Game successfully loaded from file
   <nombre_proporcionado_por_el_usuario>".
```

- La relación entre cada instancia de la clase *Bomb* y una instancia de la clase *DestroyerShip* (su creador) es una parte del estado del juego. Al escribir el estado del juego en fichero, si no se almacena la información de la relación entre bombas y destroyers, al leer el estado del juego desde fichero y seguir jugando, el comportamiento del juego no será correcto. Hay distintas maneras de almacenar esta información.

- ✓ **Calidad del código.** Al cargar un fichero con datos inválidos, la aplicación no debería parar su ejecución (o bloquearse) y tampoco debería quedarse en un estado incoherente. Para evitar que se produzcan estados incoherentes, el método *load* de *Game* debería almacenar los datos del *game* que se quieren cargar en una copia de seguridad (en memoria) antes de empezar a cargarlos.
- ✓ De este modo, cuando se produzca una excepción debido a que el contenido del fichero sea incorrecto, en el método *load* de *Game*, se podrá recoger esta excepción, reponer el estado anterior del game a partir de la copia de seguridad y a continuación relanzar la misma excepción (para que se recoja más arriba en la pila de llamadas).

4. Anexos de la Práctica 2

```
public enum PrinterTypes {
   BOARDPRINTER("boardprinter", "prints the game formatted as a board
of dimension: ", new BoardPrinter()),
   STRINGIFIER(“stringifier", "prints the game as plain text", new
Stringifier());

   private String printerName;
   private String helpText;
   private GamePrinter printerObject;

   private PrinterTypes(String name, String text, GamePrinter printer)
   {
     printerName = name;
     helpText = text;
     printerObject = printer;
   }
```

```
public static String printerHelp(Game game) {
    String helpString = "";
    for (PrinterTypes printer : PrinterTypes.values())
       helpString += String.format("%s : %s%s%n", printer.printerName,
printer.helpText, (printer == BOARDPRINTER ? Game.DIM_X + " x " +
Game.DIM_Y : "" ) );
    return helpString;
}

public GamePrinter getObject() {
   return printerObject; }
}
```

```
public class GameObjectGenerator {
    private static GameObject[] availableGameObjects = {
      new UCMShip(),
      new Ovni(),
      new RegularAlien(),
      new DestroyerAlien(),
      new ExplosiveAlien(),
      new ShockWave(),
      new Bomb(),
      new Missile(),
      new Supermissile()
   };
   public static GameObject parse(String stringFromFile, Game game,
FileContentsVerifier verifier) throws FileContentsException {
      GameObject gameObject = null;
      for (GameObject go: availableGameObjects) {
         gameObject = go.parse(stringFromFile, game, verifier);
         if (gameObject != null) break;
      } return gameObject; } }
```

```
public class FileContentsVerifier {
    public static final String separator1 = ";";
    public static final String separator2 = ",";
    public static final String labelRefSeparator = " - ";
    private String foundInFileString = "";

    private void appendToFoundInFileString(String linePrefix) {
      foundInFileString += linePrefix;
    }
    // Don't catch NumberFormatException.
    public boolean verifyCycleString(String cycleString) {
       String[] words = cycleString.split(separator1);
       appendToFoundInFileString(words[0]);
       if (words.length != 2
           ||!verifyCurrentCycle(Integer.parseInt(words[1])))
         return false;
       return true;
    }
```

```
public boolean verifyLevelString(String levelString) {
     String[] words = levelString.split(separator1);
     appendToFoundInFileString(words[0]);
     if (words.length != 2 || !verifyLevel(Level.parse(words[1])))
        return false;
     return true;
}
// Don't catch NumberFormatException.
public boolean verifyOvniString(String lineFromFile, Game game, int
armour) {
     String[] words = lineFromFile.split(separator1);
     appendToFoundInFileString(words[0]);
     if (words.length != 3)
        return false;
     String[] coords = words[1].split(separator2);
     if ( !verifyCoords(Integer.parseInt(coords[0]),
Integer.parseInt(coords[1]), game) ||
!verifyLives(Integer.parseInt(words[2]), armour) )
        return false;
     return true; }
```

```
// Don't catch NumberFormatException.
public boolean verifyPlayerString(String lineFromFile, Game game, int
armour) {
     String[] words = lineFromFile.split(separator1);
     appendToFoundInFileString(words[0]);
     if (words.length != 5)
        return false;
     String[] coords = words[1].split(separator2);
     if ( !verifyCoords(Integer.parseInt(coords[0]),
          Integer.parseInt(coords[1]), game) ||
          !verifyLives(Integer.parseInt(words[2]), armour) ||
          !verifyPoints(Integer.parseInt(words[3])) ||
          !verifyBool(words[4]) )
       return false;
    return true;
}
```

```
// Don't catch NumberFormatException.
public boolean verifyAlienShipString(String lineFromFile, Game game,
int armour) {
     String[] words = lineFromFile.split(separator1);
     appendToFoundInFileString(words[0]);
     if (words.length != 5)
        return false;
     String[] coords = words[1].split (separator2);
     if ( !verifyCoords(Integer.parseInt(coords[0]),
Integer.parseInt(coords[1]), game) ||
          !verifyLives(Integer.parseInt(words[2]), armour) ||
          !verifyCycleToNextAlienMove(Integer.parseInt(words[3]),
game.getLevel()) ||
          !verifyDir(Move.parse(words[4])) ) {
               return false;
          }
     return true;
}
```

```
// Don't catch NumberFormatException.
public boolean verifyWeaponString(String lineFromFile, Game game) {
    String[] words = lineFromFile.split(separator1);
    if (words.length != 2)
        return false;
    appendToFoundInFileString(words[0]);
    String[] coords = words[1].split(separator2);
    if (!verifyCoords(Integer.parseInt(coords[0]),
Integer.parseInt(coords[1]), game) )
       return false;
    return true;
}
public boolean verifyRefString(String lineFromFile) {
    String[] words = lineFromFile.split(labelRefSeparator);
    if (words.length != 2 || !verifyLabel(words[1]))
      return false;
    return true;
}
```

```
public static boolean verifyLabel(String label) {
    return Integer.parseInt(label) > 0;
}
public static boolean verifyCoords(int x, int y, Game game) {
    return game.isOnBoard(x, y);
}
public static boolean verifyCurrentCycle(int currentCycle) {
    return currentCycle >= 0;
}
public static boolean verifyLevel(Level level) {
    return level != null;
}
public static boolean verifyDir(Move dir) {
    return dir != null;
}
public static boolean verifyLives(int live, int armour) {
    return 0 < live && live <= armour;
}
```

```
public static boolean verifyPoints(int points) {
    return points >= 0;
}
public static boolean verifyCycleToNextAlienMove(int cycle, Level
level) {
   return 0 <= cycle && cycle <= level.getNumCyclesToMoveOneCell();
}
// parseBoolean converts any string different from "true" to false.
public static boolean verifyBool(String boolString) {
   return boolString.equals("true") || boolString.equals("false");
}
public boolean isMissileOnLoadedBoard() {
   return foundInFileString.toUpperCase().contains("M");
}
// Use a regular expression to verify the string of concatenated
prefixes found
public boolean verifyLines() {
   // TO DO: compare foundInFileString with a regular expression
return true;
}
```

```
public String toString() {
   // TO DO
   return "";
}
```