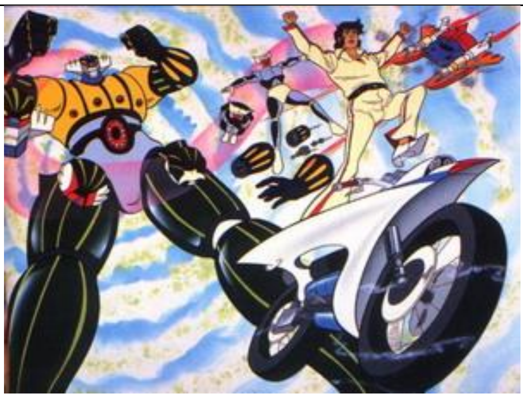
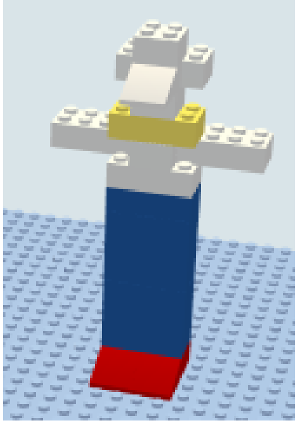
Learn Java By a Turn-Based Game

By Andrea Valente – andrea270872@gmail.com – andval.net

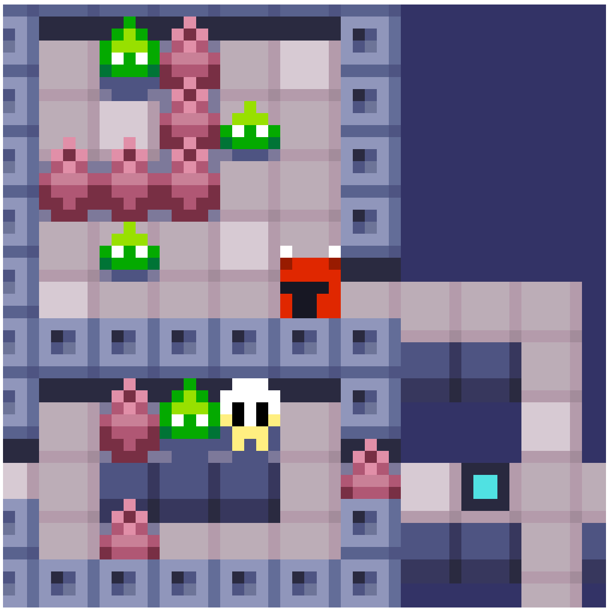
[CHAPTER 1] Programming is expressing myself

I want to (re)make something in on my computer, and because of that I NEED programming.
When I was 10 I used to watch super-robots on TV (Grendizer and Steel Jeeg being my favorites) and I constantly tried to remake them with LEGO (http://www.lego.com/). In the 1980s there were no specific pieces for making humanoid robots in the basic LEGO sets, so I was progressing in 2 ways: randomly assembling LEGO to see if I could get some useful features like a moving joint or a detachable punch rockets; at the same time I was trying to reason on the overall shape of the robots and re-build them with the available LEGO bricks. The results were kind of comic (looking back now) but I felt real pride in the exercise, and one of the results of this thinking process was that I went browsing LEGO boxes in shops looking for the pieces that I needed, instead of looking at the actual cars or houses that you were *supposed* to build with the box. Within a few years I put together a collection of various LEGO bricks in a large drawer in my bedroom: they were my toolbox and raw materials for my reverse engineering projects.

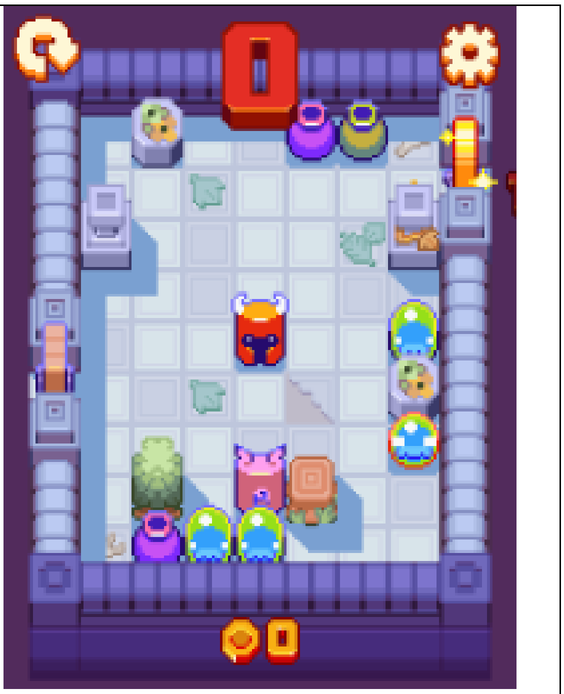
 Grendizer toy *[source https://starscreamersrants.wordpress.com]*	 *Kotetsu Jeeg poster* *[source https://dk.pinterest.com/pin/471892867179252289/]*
  BASIC Grundkasten. Ab 5 Jahre Bauanregungen inliegend Boite de base. A partir de 5 ans A l'intérieur, une brochure d'idées Confezione basic 5+. Dai 5 anni Contiene istruzioni di montaggio LEGO Aufzieh-Motor enthalten Comprend un moteur a clé LEGO Comprende un motore a molla LEGO 550 *A typical LEGO set (mid-80s)* *[source http://brickset.com/sets/year-1985]*	 *My LEGO Jeeg (early 1980s)* *[re-created with https://www.buildwithchrome.com/builder]*

Later, when I had my first computer (a Commodore 64) I found that the same way of thinking worked in programming too: you can go bottom-up and play with the basic commands to see what they do and mean, but at the same time you can think of something complete you want to achieve and break it apart in a top-down fashion, until you can re-build it.

In this case I want to remake Nitrome's Rust Bucket (https://play.google.com/store/apps/details?id=com.nitrome.rustbucket&hl=en) or Turnament (http://www.nitrome.com/games/turnament) or a game that feels like that.



Turnament snapshot



Rust Bucket snapshot

Reverse engineer a game

I have to start simpler, to be able to grasp what the game is actually doing. Also I need to start playing the game with a different goal in mind: not to win, but to test it, to uncover what the programmers and game designers put in the game, its rules, the way enemies moves and the subtle balance between what my character can do and the way the level is designed to be difficult but winnable.

First of all: **Rust Bucket is a turn-based game**, which means that nothing happens while you think about your next move. It is very much like chess, where you move and then the opponent (here the rules of the game) move. Each enemy moves in a different way, and the timing is a very important part of the game.

My remake will be called something funny that is also a pun on the original name, for example: *Oxidized Pot* (or OP for short).

The language: Java

I want to use Java for this project, but it could also be any other object-oriented programming language. For a theoretical introduction to Java just google it. As for the reverse engineering of robots with LEGO, I think I will get deeper understanding of my tools as I need it to solve the problems that arise in the projects.

I still need the equivalent of my LEGO drawer, which in programming is an IDE (Integrated Development Environment) and for Java a classic, free one is NetBeans (see https://netbeans.org/). When I download NetBeans I get a program to edit, compile and run and debug my programs, but also a Java Virtual Machine, compiler and other Java tools that are needed to be able to run my programs. These are my digital construction blocks and the tools required to express myself and realize my reverse engineering dreams (and perhaps even create new original games and applications).

The first time that I worked with an IDE like NetBeans I found it a bit weird and even simple things like opening an existing project or running my code was not that easy. To familiarize myself with new tools I find it useful to walk through a tutorial or 2. So after installing NetBeans I would probably take a quick look at a tutorial like this one: https://netbeans.org/kb/docs/java/quickstart.html .
It shows how to create a new project, and how to run it. Another tutorial that covers more or less the same topics is: http://docs.oracle.com/javase/tutorial/getStarted/cupojava/netbeans.html from Java's official documentation (a good place to look for Java-related material).

[Alternative ways]

Perhaps the Nitrome games I'm talking about are not super-interesting for some programmers… then I can still suggest something like this excellent DOOM remake: **DoomRL** http://doom.chaosforge.org/ (aka Doom the RogueLike, by Kornel Kisielewicz, part of *ChaosForge*), which is also a turn-based game, with gorgeous text-based graphics.

[Exercises]

1. Play at least the first 5 levels of Nitrome’s **Turnament** and **Rust Bucket**, in order to get the feeling of the gameplay.
2. Play a minimum of 10 games at Nitrome’s **Rust Bucket** in **Endless Mode**, to get familiar with as many of the enemies and game mechanics as possible. Take notes about the way enemies move and what strategies you adopt to defeat them.

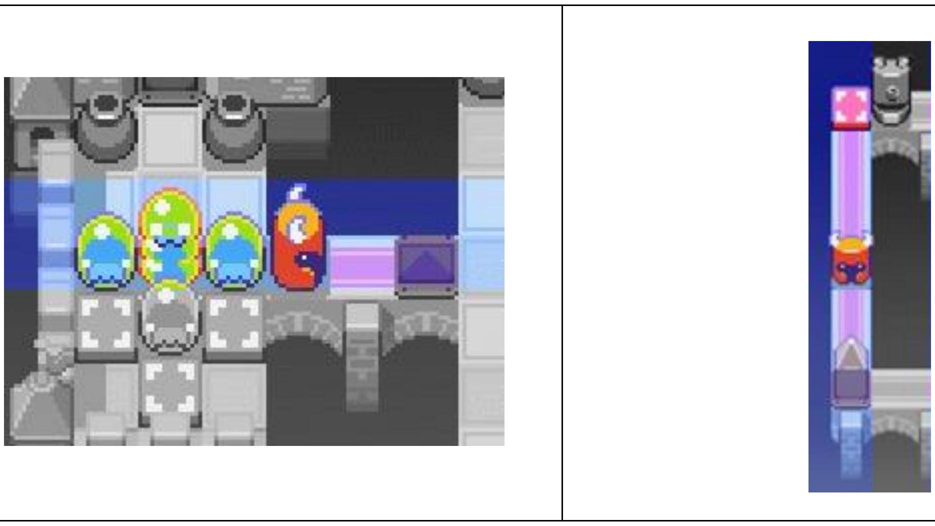
[CHAPTER 2] First challenge: find out how the original game works

Here I am in the same position as a scientist in front of an unknown gadget or phenomenon. I know that Rust Bucket works, I can play with it, but do I **understand** how it works?

I don't want to just play, I want to be able to re-make the game. So I need to understand how to re-create it without a computer. If I can do that, make the game a **board game**, I can surely teach my computer to hold the board and the pieces for me later on. And I also need to identify the rules of the game so that I perfectly simulate a game in the board game version.

With a game like Rust Bucket as the starting point, the board game version is going to be relatively easy because all works in steps already. The same exercise would be more complex for say a game like Super Mario Bros. (by Nintendo), but as for the LEGO super robots, start simple then redesign until satisfied! Also a great analysis already exists at the Rust Bucket Wiki (http://nitrome.wikia.com/wiki/Endless_Mode_(Rust_Bucket)), so I know how to call the pieces of the game.

Another trick to simplify is to think only to a portion of the game, for example focus on a single line. Even like this many interesting things can happen in the game, and it makes my life easier when defining my *Oxidized Pot* board game.

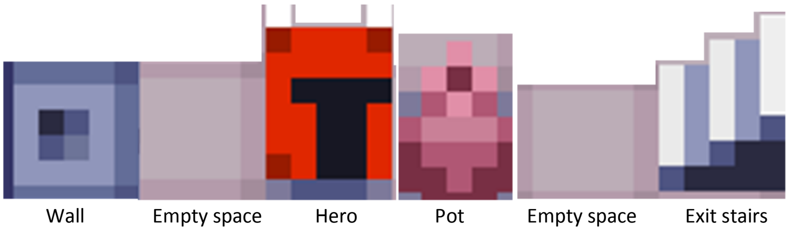


Cutting a single line out of a level in Rust Bucket
[source http://toucharcade.com/2016/01/28/rust-bucket-update/]

Good, the next question is: how can I reason about the board game? I need to be able to quickly sketch a situation and find out the consequences. For example I can imagine that the player's character (aka the hero) being near the exit with a pot to block the way, so that the only think the player needs to do to win is to smash the pot and walk into the exit… but how can I visualize/sketch this situation? (also considering that I am very VERY bad at drawing?!)

I can make up a notation and **write** the level down!

```
Wall-Empty-Hero-Pot-Empty-Exit
```



And because I want to have a quick way to scribble these single-line “levels” down, I will abbreviate the notation to something like:

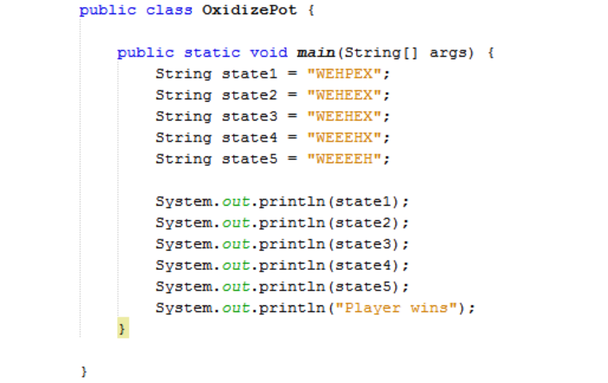
```
WEHPEX
```

In this way a sequence of characters (in programming that is called **a string**) is my representation of the situation in which the game is. And to reason about moves in the game, I can just write down a sequence of strings, like this:

`WEHPEX`	*Then the player moves right*
`WEHEEX`	*Pot brakes. Player moves right again*
`WEEHEX`	*Right*
`WEEEHX`	*Right*
`WEEEEH`	*Player wins*

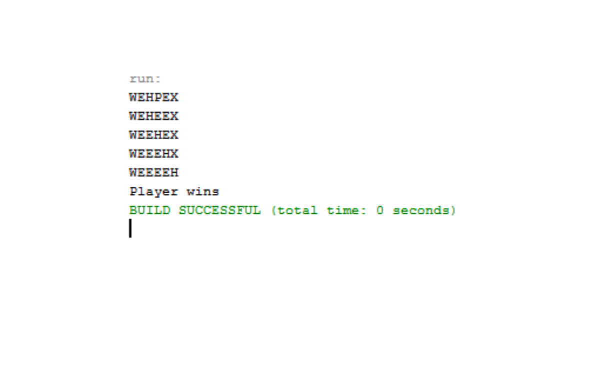
This is great: I have a notation to think about that is the situation of the game (in programming terms **the state**) and I can also quickly reason on a sequence of situations, as they might progress while I play my board game.
I can now express rules about what the game should and should not do. For example: when the hero ‘H’ moves he leaves an empty space on is old position. But if the ‘H’ wants to move on a character ‘P’ in my string, which represents a pot, the pot breaks and ‘H’ does not actually move for that turn. I need to have a representation for the state of the game before I can express rules.

Now I can now use a Java program to print the situation my board game is in and I can print sequences on the screen to give the impression that the situation is changing over time. I open NetBeans and create a Java project, call it “Oxidized Pot” and write the following:



```
public class OxidizePot {

    public static void main(String[] args) {
        String state1 = "WEHPEX";
        String state2 = "WEHEEX";
        String state3 = "WEEHEX";
        String state4 = "WEEEHX";
        String state5 = "WEEEEH";

        System.out.println(state1);
        System.out.println(state2);
        System.out.println(state3);
        System.out.println(state4);
        System.out.println(state5);
        System.out.println("Player wins");
    }

}
```

[source: NetBeans project OxidizedPot_v0]



```
run:
WEHPEX
WEHEEX
WEEHEX
WEEEHX
WEEEEH
Player wins
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ouput

So it works. It is not interactive (the program does not ask for my input, it just prints the same stuff each time I run it), but it produces a sequence of snapshots of my game, and that's a start.

What does the code mean? I basically need to tell Java:

`output` *`something`*

on the screen, where **output** should be a suitable Java a command; the *something* part is changeable, so that I can tell Java to output a text followed by a different text, for example:

`output` *`WEHPEX`*

`and then`

`output` *`WEHEEX`*

In Java the output instruction is:

`System.out.println(` *`something`* **`);`**

So to output the 2 pieces of text as above I have to write:

`System.out.println(` *`"WEHPEX"`* **`);`**

`System.out.println(` *`"WEHEEX"`* **`);`**

Placing on System.out.println instruction after the other has the effect that the first is executed first and the second… after that. The way the output instruction is written follows Java's syntax: typically a command will have a name (that should be written always in the same way, for Java to understand which command I'm referring to) and some changeable part, usually written between brackets. In this case the changeable part is a piece of text and in Java's syntax that is a sequence of characters (letters or numbers) written between 2 quotes. Usually Java commands end with a semi-column ; character, to help Java understand where each command ends and where the next one might start.

For some reasons (what I will discuss later, when the needs arises) in Java I cannot just write my commands without adding something before and after them. This is why when I create a project in NetBeans, the IDE kindly writes for me something like:

```
public class OxidizedPot{

   public static void main(String[] args){
```

before the place where I will write my commands, and then it adds something like this:

```
       }

}
```

after. For now I will just regard this as a **given fact**, the same way that I don't need to have any deeper understanding of why LEGO bricks have specific sizes and come only in certain shapes. It will become clearer as I progress and I have better grasp of the relations among my commands and general Java programming.

It is useful to notice however that whatever these 2 lines mean, the have **balanced** curly brackets: when a { opens, there is always one that closes, like this } . This suggests that somehow my code is **sandwiched** in between curly brackets. A piece of code written in between an open and closed curly bracket is called a **block** in Java. My program consists of a **class** block that contains a **main** block, that contains in turn my actual commands to output some text. And the most important thing I need to remember for now is that whatever command I write inside the main block will be executed by Java.

In my *OxidizedPot_v0* project I also use a very important feature of programming languages: I declare and use **variables**. It is very much like giving a name to something, to simplify talking about it. For instance I might want to say: "Look at the male black cat that is currently in a basket near the window and tell me if it is asleep." And later: "Look at the male black cat that is currently in a basket near the window and tell me if it has a collar". If many of my questions involve that cat, it would be more convenient to give it a name, even a made-up one, and use that to speed-up our conversation:
"See that male black cat that is currently in a basket near the window? Let's call it Bob.
Can you tell me if Bob is asleep?
Can you tell me if Bob has a collar?"

Later I might change my mind and decide to call "Bob" a different cat, and it would still make sense to ask you to check if Bob (whichever cat it is at that moment) is sleeping or not.

In Java terms, a **variable** has a **name** (here it would be Bob), a **value** (in this case the description of the specific cat I want to talk about) and a **type**, usually the same as the type of its value (in the cat example Bob is a cat-type variable, and its value is a cat-type animal). I can always replace a value with a variable in my Java commands, provided the type of the variable matches that of the value. So if it is correct to write a command like this:

```
System.out.println( "WEHPEX" );
```

then it is also correct to replace the text (a value of type String, in Java) with a String-type variable:

```
aaa = "WEHPEX";
```

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

and in this case Java will not output the text "aaa" but the value of the variable aaa, which is set to the string "WEHPEX" by the command:

```
aaa = "WEHPEX";
```

which means: please Java, give the variable aaa the value expressed by the string “WEHPEX”. This is called *assigning a value to a variable*.

Just to make my life more annoying, Java requires that I **declare** a variable before I use it, so I have to specify that aaa is not a number or another type of variable, but actually a String-type variable, that will be able to have a String-type value. In Java I do this in the following way:

```
String aaa;
```

which declares a variable with name aaa, of type String. The name of the variable is of course up to me, the programmer, and it will in most cases be more meaningful than aaa. Putting it all together, I can declare a variable, then put a value in it, output its value, change the value to a new one and output that one as well:

```
String aaa;

aaa = "WEHPEX";

System.out.println( aaa );

aaa = "WEHEEX";

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

I must remember to declare my variable exactly once: in fact I cannot use a variable that I did not declare, but when a variable is declared its type is fixed and Java will not allow me to change it later in the program, so re-declaring a variable is also considered an error by Java. After all if Bob is supposed to be a cat, I can use the name for different cats, but if I suddenly decide that Bob is a tractor our conversation will rapidly become very confused. So: declaration must be done exactly once for each variable, however putting a value in the variable (aka assigning a value) can be done as many times as needed.

Another way to think about a variable is as a box that can contain different things at different times. Also, in Java I can define as many variables as I need, and the values of these variables are stored in the memory of my computer. Of course the more boxes I use, the more space they will take up in the memory (but no worries, modern computers have very large memories).

Apart from strings, in Java I can work with numbers. For example I can declare a variable of type int (which stands for integer or whole numbers) and use it to perform calculations (like finding out the area of a square, given how long its side is):

```
int side;

side = 10;
```

```
int square;

square = side * side;

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

this program makes Java put the whole number *10* in the variable *side*, then Java computes the result of 10 times 10 and places the result in the variable *square*. Finally, Java outputs 100, which is the final value of my variable *square*. In this case I chose my variable names to help me read the code. Even if I did not use int-type variables in project **OxidizedPot_v0**, I will need both numbers and strings in the development of my game Oxidized Pot.

[Alternative ways]

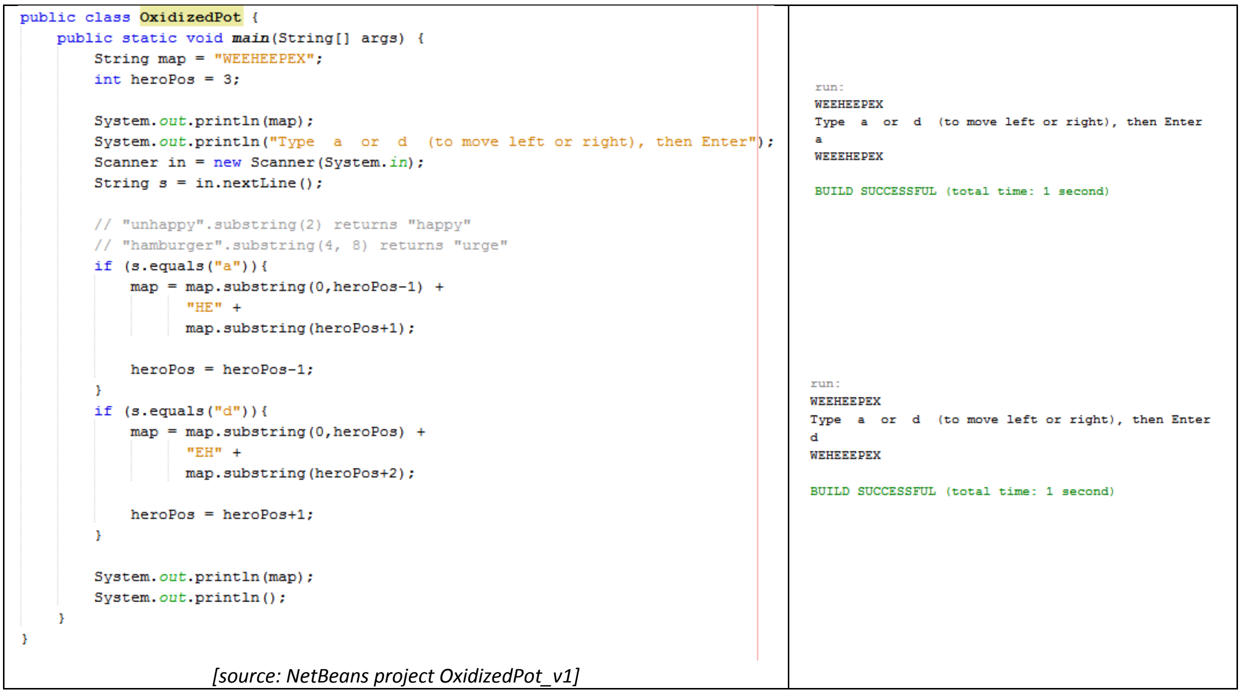
Perhaps the notation I am using to represent the state of my OP game is not visually very pleasing. I could have done it in a different way, like shown in project **OxidizedPot_v0_1**. I also renamed the variables to show that they can be totally arbitrary.

[Exercises]

1. Starting from this situation (game state):
   `WEEHPEPPEXEW`
   write down the sequence of states until the player victory. Then change the code of project **OxidizedPot_v0** so that it writes all the steps you found.
2. Starting from this situation (game state):
   `WPXEEPPHEPW`
   write down a sequence of states in which the player breaks as many pots as possible before finally reaching the X and winning. Change the code of project **OxidizedPot_v0** so that it writes all the steps you found.
3. Perhaps the idea I had to change the notation in project **OxidizedPot_v0_1** was not that good after all. Find another notation to represent the state of the game, one that is readable and better looking for you, and change the code to in project **OxidizedPot_v0_1** to print the sequence of states according to your notation.
4. Change the code in project **OxidizedPot_v0** so that the program still prints the same sequence of strings, but using only 1 variable.

[CHAPTER 3] Oxidized Pot must be playable – moving the hero

The next thing I must have: the game has to be playable, interactive, not just a sequence of strings printed on the screen. The code below is taken from the *OxidizedPot_v1* project, and shows a possible way to let the player decide in which direction the H should move (left or right).



```java
public class OxidizedPot {
    public static void main(String[] args) {
        String map = "WEEHEEPEX";
        int heroPos = 3;

        System.out.println(map);
        System.out.println("Type  a  or  d  (to move left or right), then Enter");
        Scanner in = new Scanner(System.in);
        String s = in.nextLine();

        // "unhappy".substring(2) returns "happy"
        // "hamburger".substring(4, 8) returns "urge"
        if (s.equals("a")){
            map = map.substring(0,heroPos-1) +
                  "HE" +
                  map.substring(heroPos+1);

            heroPos = heroPos-1;
        }
        if (s.equals("d")){
            map = map.substring(0,heroPos) +
                  "EH" +
                  map.substring(heroPos+2);

            heroPos = heroPos+1;
        }

        System.out.println(map);
        System.out.println();
    }
}
```

```
run:
WEEHEEPEX
Type  a  or  d  (to move left or right), then Enter
a
WEEEHEPEX

BUILD SUCCESSFUL (total time: 1 second)
```

```
run:
WEEHEEPEX
Type  a  or  d  (to move left or right), then Enter
d
WEHEEEPEX

BUILD SUCCESSFUL (total time: 1 second)
```

[source: NetBeans project OxidizedPot_v1]

A comment about comments

It is sometimes a good idea to take down few notes while (re)designing something. For example when I was building my LEGO super-robots I had a hard time keeping a record of what the robots looked like. Sometimes I would have to cannibalize the LEGO bricks from one version of a robot to the next, and that made it difficult to remember how each version actually looked like. I tried sketching simple diagrams and writing comments on them, to help me remember how I solved certain problems when building a particular version of a robot. It was annoying and time consuming (I wanted to build robots and play with them, not write notes), but it helped a bit in keeping track of my robots' development.

In programming I can do the same thing, but the good news is that I don't need to write on something else: Java allows me to write my comments directly in between the lines of my code. The question then is: if I write stuff that is not Java code in the middle of a program, how will Java react to it when I try to run the code? Usually Java (the Java compiler in fact) is VERY picky about syntax, so I might expect trouble… But in fact there is a simple solution, which is also very standard in most programming languages. Just sandwich my comments between some special symbols that tell Java not to look at the comments as code. Smart. It's a bit like *air quotes*.

	… some comment…	

[hand - source https://en.wikipedia.org/wiki/Air_quotes]

In Java a comment starts with **/*** and ends with ***/** . The comment can continue on multiple lines. When I only have a short comment that takes no more than 1 line I can use this instead **//** . The code of project *OxidizedPot_v1* shows a couple of single-line comments.

Read player input

The first thing I need is a way to tell my program:

> *"ask the player what he/she wants to do, and depending on what the player says, do it"*.

In code, that becomes something like the following:

```
someVariable = input
```

which means: Java please pause the program, wait for the player to input some text, then place the inputted text into a variable (above called *someVariable*). This is the logical way I might expect things would work… but in Java I need to prepare the ground to read user input, with the following line of code:

```
Scanner in = new Scanner(System.in);
```

I have to write it precisely as it is, just once in the beginning of my code, and after that I can read input from the player as many times as I want. The following code for example:

```
String a;

a = in.nextLine();

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

written in the **main block** of the program, reads anything that the player types (followed by ENTER), sets that into the variable a (which has type String) and after that it outputs the text, so that the player can see what he/she wrote.

A final note on the Scanner. Java requires that I add the following line before my **class block**, so at the beginning of the program:

import java.util.Scanner;

what it means is that Java needs to find the Scanner file (because I did not write the Scanner myself, but I'm borrowing it from other programmers that added it in the standard set of programs that are shipped with Java) and enables me to use Scanner to read player input.

Change a string

Now that I have a way to read player input, I need to find a way to change the string representing the state of the game from "WEEHEEPEX" to "WEHEEEPEX" if the player moves to the left, and from "WEEHEEPEX" to "WEEEHEPEX" if the player moves to the right. But these 2 rules (call them **moveLeft** and **moveRight**) should not just work with the specific string "WEEHEEPEX", instead they should work wherever the H is.

Better to consider 1 rule at the time. I start with **moveLeft**. To represent the H moving left I could:

- find where the H character is in the string -> "WEEHEEPEX" *the H is the 4th character*
- cut the string in 3 pieces: -> "WE" "EH" "EEPEX"
- change the central part so that the H moves left -> "WE" "HE" "EEPEX"
- and finally, put the 3 pieces back together -> "WEHEEEPEX"

and it is done.

In Java there is an operation that works on strings and that makes a copy of a part of a string (called **substring**). There is also another operation that can glue together 2 strings one after the other to form a longer one (called **concatenation**). By using these operations I can explain to Java how to do the steps needed for my **moveLeft** rule.

```
String map = "WEEHEEPEX";

int heroPos = 3;

map = map.substring(0,heroPos-1) +
      "HE" +
      map.substring(heroPos+1);

heroPos = heroPos-1;

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

An interesting fact about Java strings is that the 4th character has in fact index 3. In the string "abc" for example the character a is at position 1 but has index 0, and c is at position 3 but has index 2. And because indexes start from 0, the H character in the **map** string has index 3. Concatenation of strings is just + , so "ab"+"cd" gives "abcd" as result. The **substring** operation is a bit more complex to use. For example:

```
System.out.println( "abcd".substring(3) );
```

will output *d* , because **substring(3)** means *"create a new string, copy all characters from the original string starting from the character with index 3, and add all other characters until the end of the original string"*. Substring can also cut a portion of a string out and create a copy, as in this example:

```
    System.out.println( "abcd".substring(2,4) );
```

will output *cd* , because "abcd".**substring(2,4)** tells Java to create a copy of "abcd", starting from the character at index 2 and ending at index 4, so that gives c and d.

A little note about this perhaps weird assignment:

```
        heroPos = heroPos-1;
```

What is the meaning of that? I have to remember that an assignment is just a way to give a value to a variable, the one that appears to the left of the = sign (in this case *heroPos*). So here what I'm telling Java is to:

- take the value that is currently in the variable heroPos (which is 3 in my particular program)
- subtract 1 to it (which gives a value of 2)
- and place the newly calculated value into the variable to the left of the equals sign (that in this case happens to be the same variable!)

the result is that the value inside the variable heroPos is changed from 3 to 2. This makes sense in my program because I want to represent that my character moves left, from 4th position in the string to 3rd (so from index 3 to index 2).

The rule to **moveRight** is implemented in a similar way, but *mirrored*, as shown in project *OxidizedPot_v1*.

What if... ? Aka doing something only when needed

I have now a trick to read player input, and I worked out how to implement my 2 movement rules. Now the problem is that Java should **know** when to apply the moveLeft or the moveRight, depending on the player input.

What I need is a way to tell Java that if the player types something, say **a**, the H should move left, and if the player types something else, like **d**, the H should move right. What I need to express is the idea that:

```
    if playerInput is equal to a then moveLeft

    if playerInput is equal to d then moveRight
```

The Java command for *if* is in fact just **if**.

```
    String s = in.nextLine();

    if ( s.equals("a") ){

        // ... do the moveLeft stuff ...

    }
```

An if command has 2 parts: a **head** and a **body**. The head is a test that can be true or false, and the body is just a block containing other commands. The cool thing is that the program will automatically decide whether to execute the commands that in body of the **if**, on the condition that the test is true (in fact the if command is called **conditional statement** in programming terms).

When I write an if in Java I have to remember that the test must be sandwiched in between 2 round brackets: it is because of Java's syntax.

OK, but **what if the test is false?** If the player types *d*, then the string s will have value "d", and s.equals("a") will be false, because it is not true that **"d".equals("a")** . Then the body of the if will not be executed and the program will simply continue with the next command **after** the if.

IFs can be used in many other situations. For example I could have a variable that remembers the player's score as a whole number (aka *integer*). Then the following code:

```
int score = 100;

System.out.println( "let me see your score..." );

if ( score<=10 ){

      System.out.println( "you suck" );

}

if ( score>80 ){

      System.out.println( "well done" );

}

if ( score>=100 ){

      System.out.println( "terrific!" );

}

System.out.println( "... and that's all I have to say" );
```

will output:

```
let me see your score...

well done

terrific!

... and that's all I have to say
```

which means that both the second and third IF executed their bodies, since both their tests were **true**. But depending on the value of the score variable it might output something completely different.

I can sandwich this code in a main block, inside a class block, and change the first line to:

```
int score = 5;
```

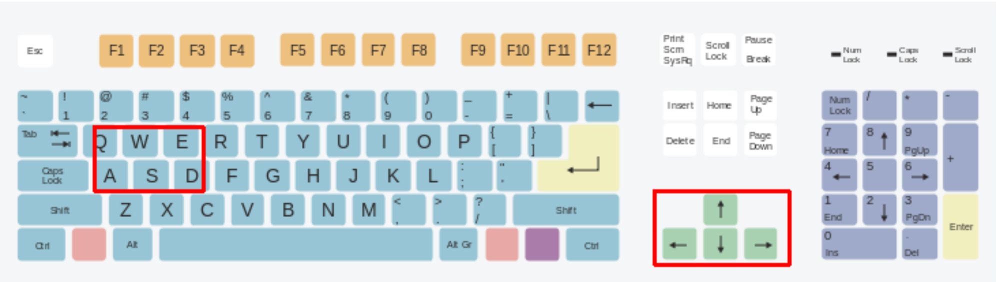
to see what happens... It should output:

```
let me see your score...

you suck

... and that's all I have to say
```

which is correct because if score is less or equals than 10, the code should output "you suck". The other tests are:

- score > 80 , which means "is score greater than 80?"
- and: score >= 100 , meaning "is score greater or equal than 100?"

Note: a and **d** are typically used in games to move left-right because of the position of the keys in a QWERTY keyboard that reminds of the way the arrow keys are placed on the right of the keyboard.



Layout of wasd keys with respect to arrow keys
[source https://en.wikipedia.org/wiki/QWERTY]

Reflections

The code in project **OxidizedPot_v1** has a typical structure that is found in many programs:

```
1. initialization of program state
2. input
3. calculations and/or decisions
4. output
```

Step 1 means variables are declared and values assigned to them, so that the state of the program represents some situation I want to work with.
Step 2 is about letting the user/player input some data: this step makes the program interactive. Without user input I can only perform calculations or execute commands in a fixed way.
In step 3 some calculations are performed, and typically that involves changing the values of the variables (aka the state of the program) according to some formulas. Step 3 can also involve conditional commands, things that I might do or not depending on the input and the values of my variables. Typically this step is implemented using IFs.

Finally, step 4 allows the program to show the results (or possibly its current state) to the user.

[Alternative ways]

Instead of cutting and putting back together my string map in the same line of code, I can use temporary variables (aka variables that exist only to help me doing something in a specific portion of my code and then I will not use them again). Project **OxidizedPot_v1_1** shows this approach.

[Exercises]

1. Create a copy of project **OxidizedPot_v1** (call it perhaps **OxidizedPot_v1_2**) so that the game prints the level in this way:

```
"WE ]-[ EEPEX"
```

both before and after player's input is processed. The "H" representing the hero has been replaced by the string "]-[". A possible execution of the program could be that the situation is printed on screen, then the player moves left and the new situation (aka game state) is printed again, like this:

```
      WE ]-[ EEPEX

-> player input: "a"

      W ]-[ EEEPEX
```

Suggestion: you might want to use a couple of variables to hold portions of the original *map* string, to help you print the new version of the game situation (check out project **OxidizedPot_v1_1**).

1.b. Practice working with string by printing the first 3 characters of the map string, followed in a new line by the last 3 characters. Add the code to achieve that at the end of the code in your **OxidizedPot_v1_2** project. The result should be something like (assuming the player moved left):

```
WEH

PEX
```

2. Make a copy of project **OxidizedPot_v1**, call it **OxidizedPot_v1_3**. In the very beginning of the **main block**, declare a int variable called *score* and set its value to 0. Now add an IF command in such a way that if the player types "+", the *score* changes to 100. The value of the variable *score* should be printed on screen each time the *map* is printed, to help the player keeping track of his/her score. It should be possible to run the game, type + and get a score of 100 without moving the "H".

[CHAPTER 4] Oxidized Pot must be playable – smashing pots and winning

So far so good. My game can now ask me (or a player) what I want to do, then depending on my input move H to the left or right, using IFs to conditionally execute blocks of commands.

What I need now are 2 things: first I want **my game to keep running** instead of asking me what I want to do once and then stop. Second, I would like to **be able to break the pot** and actually **win by reaching the exit** (which I cannot do now because there are no rules to check that my hero reached the X character in the map).

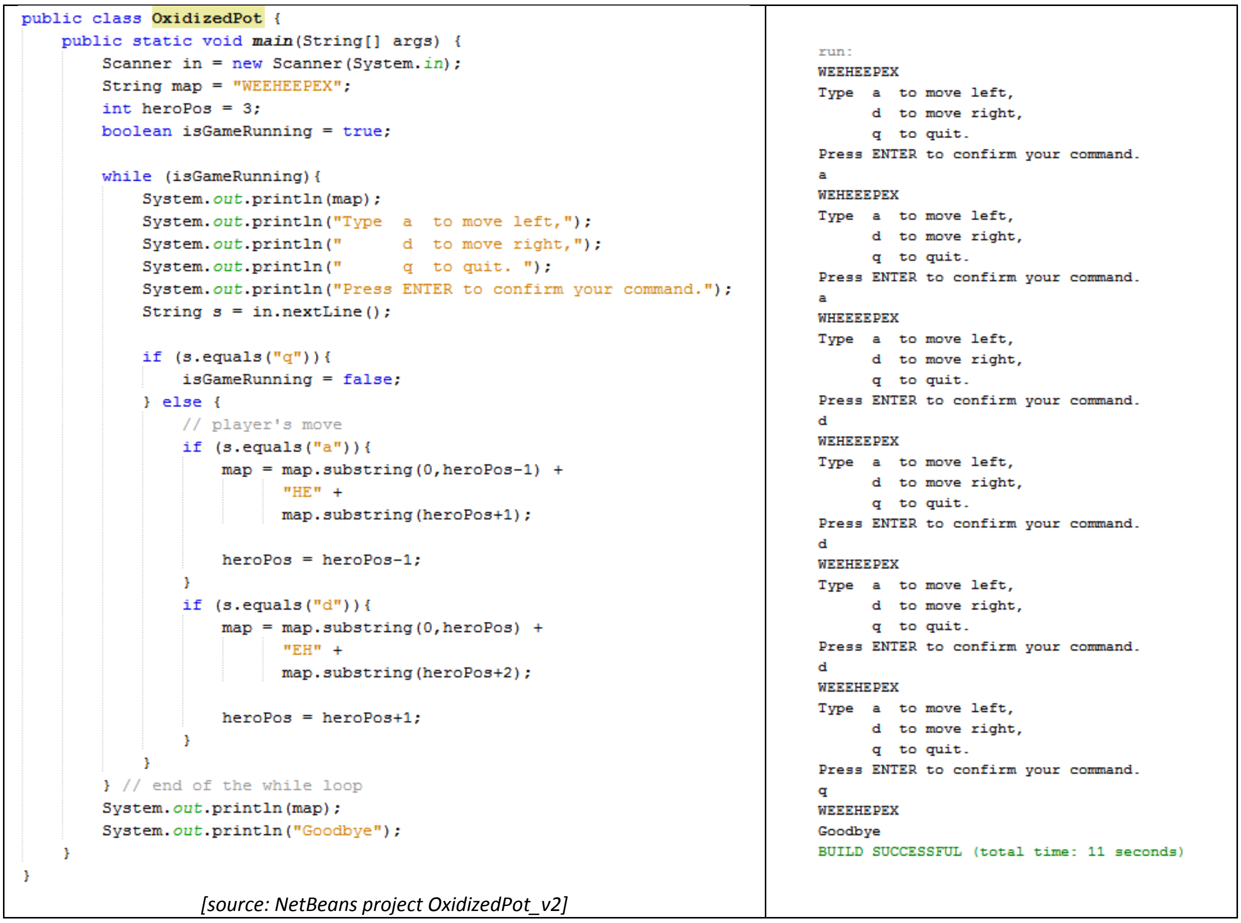
Doing something many times

The code in project *OxidizedPot_v2* shows that I can use a WHILE loop to keep the game repeating the same commands over and over, until the player decides to quit (by typing "q"). In my *OxidizedPot_v1* project I already had the code to read player input and IFs to move the H left and right in the map of the level. What I need now is a simple way to say:

> *"take these commands (that read player input and move the H)*
> *and repeat them many times"*

In programming this is can be expressed with a WHILE loop:

```
WHILE somethingIsTrue DO someCommands
```



```
public class OxidizedPot {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String map = "WEEHEEPEX";
        int heroPos = 3;
        boolean isGameRunning = true;

        while (isGameRunning){
            System.out.println(map);
            System.out.println("Type  a  to move left,");
            System.out.println("      d  to move right,");
            System.out.println("      q  to quit. ");
            System.out.println("Press ENTER to confirm your command.");
            String s = in.nextLine();

            if (s.equals("q")){
                isGameRunning = false;
            } else {
                // player's move
                if (s.equals("a")){
                    map = map.substring(0,heroPos-1) +
                          "HE" +
                          map.substring(heroPos+1);

                    heroPos = heroPos-1;
                }
                if (s.equals("d")){
                    map = map.substring(0,heroPos) +
                          "EH" +
                          map.substring(heroPos+2);

                    heroPos = heroPos+1;
                }
            }
        } // end of the while loop
        System.out.println(map);
        System.out.println("Goodbye");
    }
}
```

```
run:
WEEHEEPEX
Type  a  to move left,
      d  to move right,
      q  to quit.
Press ENTER to confirm your command.
a
WEHEEEPEX
Type  a  to move left,
      d  to move right,
      q  to quit.
Press ENTER to confirm your command.
a
WHEEEEPEX
Type  a  to move left,
      d  to move right,
      q  to quit.
Press ENTER to confirm your command.
d
WEHEEEPEX
Type  a  to move left,
      d  to move right,
      q  to quit.
Press ENTER to confirm your command.
d
WEEHEEPEX
Type  a  to move left,
      d  to move right,
      q  to quit.
Press ENTER to confirm your command.
d
WEEEHEPEX
Type  a  to move left,
      d  to move right,
      q  to quit.
Press ENTER to confirm your command.
q
WEEEHEPEX
Goodbye
BUILD SUCCESSFUL (total time: 11 seconds)
```

[source: NetBeans project OxidizedPot_v2]

The WHILE command works very much like an IF: there is a **head** with a test that needs to be true for the **body** to be executed, but after the body block is executed, the WHILE command goes back to the beginning and re-checks the test, and if it is still true the body is re-executed and so on. The loop breaks when some of the commands in the body change the variables used in the test, and make it become false. The first time that the WHILE checks the test and finds it false, the loop is interrupted, the body is **not executed** and the program continues to run from the next instruction **after** the body of the WHILE. So a WHILE is kind of a repetition of an IF:

```
WHILE somethingIsTrue DO someCommands
```

basically means:

```
IF somethingIsTrue
      THEN someCommands , go back and test the IF again
      ELSE continue with the rest of the program
```

In my Oxidized Pot game, I want to have a while loop that ends when the player types "q". When I have the while loop working I can add the commands for moving the H left and right, and my game will be much more playable. The simple while loop that stops when the player types "q" can be like this:

```
Scanner in = new Scanner(System.in);
boolean isGameRunning = true;
while (isGameRunning==true){
      System.out.println( "Type q to quit" );
      String s = in.nextLine();
      if (s.equals("q")){
            isGameRunning = false;
      }
}
System.out.println( "Game Over" );
```

The code above is available in testWHILE project, in the *alternative ways* folder. A longer version of this code is used in project *OxidizedPot_v2*. The code in *OxidizedPot_v2* also shows a variable of type *boolean*. A boolean variable can only have value **true** or **false**, so it can be naturally used in tests, both in IF and WHILE commands. A boolean variable can be used to remember if a certain fact is true in the program:

```
boolean playerAlive;
playerAlive = true;
```

```
if ( playerAlive ){

      System.out.println( "you are still alive!");

}

if ( playerAlive==false ){

      System.out.println( "oops...");

}
```

Here the variable *playerAlive* remembers in its value if the player died or not at this point of the game. The while loop in project *OxidizedPot_v2* uses the same approach: the boolean variable **isGameRunning** is declared and used to remember that the game is actually running (so its value is initially **true**). In case the player types "q", the game will change the value of **isGameRunning** to **false**, forcing the WHILE to stop repeating its body at the next repetition.

*Notice the way the variable name is written? That is called **CamelCase** and it helps reading the 2 words player and alive even when they are written without space in between https://en.wikipedia.org/wiki/CamelCase .*

Boolean values can also be created from tests, like in this example:

```
boolean whoKnows;

whoKnows = 2>3;

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

Here Java will evaluate the test 2>3 (which means "is 2 larger than 3?") and the result will be the value **true**. This means that the value of *whoKnows* is set to **true**, and the output will be *true*.

Finally, in project *OxidizedPot_v2* I use a IF-ELSE command, which I have not yet explained. IF-ELSE is not really a new command, it is more like a shorthand notation for 2 IFs one after the other. I can have in mind something like this:

if someTest is true *do* *something*

but if someTest is false *do* *somethingElse*

and I can express this already using 2 IFs, like in the following piece of code:

```
boolean gotBonus = false;

if ( gotBonus==true ){

      System.out.println( "great!" );

}

if ( gotBonus==false ){
        System.out.println( "too bad :(" );

    }
```

In Java I have a compact way to write this, using an IF-ELSE command:

```
    boolean gotBonus = false;

    if ( gotBonus==true ){

        System.out.println( "great!" );

    } else {

        System.out.println( "too bad :(" );

    }
```

These examples are available in *testIF_ELSE* project, in the *alternative ways* folder.

Testing and adding more rules

Now I have a very simple version of Oxidized Pot that works and that can be tested and improved one step at the time (aka a **horizontal prototype**). But it's a looong way to go before it feels like Rust Bucket. As for my LEGO super robot efforts, I start to feel the hidden complexity and depth of the original product. The process of reverse engineer and re-create something makes me respect the creators of the original even more then when I was just a player.

Now for some **testing**: happens if the H reaches the position of the P or the X?
The H simply keeps moving and "erases" everything else in the map ☹
How should I fix that? I also need to be able to break the pot and actually win by reaching the exit. This requires better rules and more of them. For example I need a rule that says: when you move left, if there is a P there, don't move but instead replace the P with an E; the same when moving right. Also a rule should say that the game is won if the H moves over the X. If the H tries to move on top of a W (a wall), the move should be forbidden.
The solution is in project OxidizedPot_v3, and it is another example of using IFs to check if some things are true about the values of my variables at certain times, while the code is running.

My moveLeft and moveRight rules *expect* the character near the H to be E (empty tile) but what if there is a pot there? The idea is that I can change the moveLeft and moveRight rules to take a pot and the exit into account. I start by fixing the **moveLeft** to cover a situation like `"WEPHEEEEX"` where the H is moving left but there is a pot where the H should *land* (aka the **target tile**):

- find where the H character is in the string -> `"WEPHEEEEX"` *the H is the 4th character*
- cut the string in 3 pieces -> `"WE"` `"PH"` `"EEPEX"`
- **target tile** is set to the tile where P will land if the move succeeds -> `"H"` *in this case*
- **if the target tile is** `"E"`

    change the central part so that the H moves left -> `"HE"`
- but … **if the target tile is** `"P"`

    change the central part so that the pot brakes but H does not moves -> `"EH"`

- also … **if the target tile is** `"W"`
  nothing changes, H does not moves -> `"WH"`
- and finally… **if the target tile is** `"X"`
  change the central part so that the H moves left
  AND set a variable to remember that the player won -> `"HE"`
- and finally, put the 3 pieces back together -> (in this case I get) `"WEEHEEPEX"`

In this case the final state of the game is `"WEEHEEPEX"` which represents a situation in which the pot was smashed and the hero did not move (as it should be according to the Turnament and Rust Bucket rules). The case of the H landing on an X is also interesting: when H moves to the left and lands on an X, it should move as normal, but the game should also *realize* that the **player won** and the while loop can be stopped. I can do this by using another boolean variable to remember whether the player already reached the X or not. The variable is called *reachedTheX* in *OxidizedPot_v3* and it is set to false initially, before entering the while loop. In both rules, moveLeft and moveRight, there is now an IF that changes the value of *reachedTheX* in case the H lands on X. Finally, an IF command written after the body of the while decides whether to output a victory message or not (depending on the value of the *reachedTheX* variable).

I also need a similar rule for moving right; and when the program exits the WHILE loop I need to check if the player won, and if so, output some message about that.

Version 3 of my game is playable! Minimal, but playable. My avatar, the 'H' cannot walk on walls, can smash pots and if I manage to reach the exit marked with 'X', I can even win the game. All done! Time for more testing: see the exercises at the end of the chapter.

[Alternative ways]

I could have worked in a different way. For example I can decide to use an existing Java class called StringBuilder to make my life easier when it comes to changing a single character in a string (a thing that I need to do on the map each time the player moves). The result of using StringBuilder instead of the **substring** method is shown in project **OxidizedPot_v2_1** (which is a variation on the project **OxidizedPot_v2** and is found in the *alternative_ways* folder).

Also, the code in *OxidizedPot_v3* **works**, but it is **not very nice**: I had to rewrite almost the same code inside both IFs that control which move the player wants to do. This is called **code duplication** and it is considered a bad idea. To see why consider what happens if we now have a different kind of pot represented by the character D in the state of the game, and imagine that the new pot behaves exactly like the standard P pots, but it takes 2 hits to break. I will have to insert a very similar code in 2 places and remember too keep it updated in both places every time the rules of the game become more complex. This is not very comfortable and can easily lead to introduction of errors. So better clean up the code to have as little code duplication as possible. I have done just that in project **OxidizedPot_v3_1**.
The D pots (that require 2 hits to break) are implemented in project **OxidizedPot_v3_2**.

[Exercises]

1. Let's test the game. Make a copy of the **OxidizedPot_v3** project and change the map of the level. Try the following maps, one by one, and for each see if the rules work correctly, if you can break all pots, and if there is a way to win or not:
   - `WWXPPPPHWW`
   - `WEXEEPEXPW`
   - `WXPWEEEHEW`
   - `WXPPEEEHEW`

   Be careful: *remember to change the heroPos variable too, each time you alter the map*.

2. Create a new version of the game, where the player gains 100 points each time a pot is smashed. And if the player wins (which means reaching the X) with more than 300 points, a special message should be displayed: **"good loot!"**. To achieve this, make a copy of the **OxidizedPot_v3** project; in the new project add a variable <u>score</u> of type int (so it can only contain whole numbers), that is initialized the value 0 in the beginning of the program, and gets incremented of 100 each time the player smashes a pot. Add an IF at the end of the program, after the WHILE loop, to print the special message in case the score is high enough. To test your new game change the map to be: `WEPPPEPHEEXW`, and set the heroPos to 7.
3. Modify the testWHILE project (in *alternative_ways* folder) so that the WHILE look would exit not only if the player types "q" but also if the player types "no more". Then add a few IF commands in the WHILE loop:
   - if the player types "paper" the program should ouput "scissors",
   - if the player types "rock" the program should ouput "paper",
   - and if the player types "scissors" the program should ouput "rock"

   Test your game by playing a few games. Can you be sure that you covered every possible behaviour of your game with your tests?

4. Start from the code in project **OxidizedPot_v3**. I would like to count how many turns a game lasts. Add a variable <u>steps</u>, of type int (aka an integer number) that is initialized to 0 before entering the WHILE that runs the game. Increment the variable <u>steps</u> of 1 each time the program goes through the WHILE loop and output its value together with the map at each turn.
   How many turns it takes to finish the level in **OxidizedPot_v3**?

[CHAPTER 5] My game totally needs better graphics

... Now that my game works, design (and play) a million levels!
... Improve the model of the game: add floors
... Where are the enemies?