Worksheet 2

Our first mini-game

We will now create our first command-line program, a very simple game in which the player must guess a number.

A functional interface

Remember the conditional statement we wrote in the first worksheet to print whether a `guess` was greater or lower than a `secret` ? It was enough to try it just for once, but it wouldn't be very convenient if we had to evaluate it several times, having to alter the `guess` variable in between. Let's wrap it into a proper procedure (just write it in a text file, evaluate it and try it in jshell, it'll become a class very soon I promise).

- what inputs would this procedure take? what are their types (we can just keep the same names we already had for the variables)?
- what should be its return type (hint, it's a *procedure*, not a *function*)
- pick a proper name for this new function and implement it (its body should just be the same code you wrote in the first worksheet, although today's class should allow you to write something a bit more elegant)

Towards object-oriented programming

During play, the secret isn't supposed to change (or the game would become suddenly much more difficult...). It is therefore a bit awkward to have to pass the `secret` argument every time the function is called. Let's create a class to hold the contextual information that this `secret` is and make it available to our function!

- create a new class called `Game` in a new file called `Game.java` in an empty directory.
- make sure it has an `int` field to hold the `secret` that will replace the corresponding variable
- implement its constructor
- now reuse and adapt the code of the previous function to become a method in this new `Game` class.

Please note that even though a bit long, classes can still be evaluated by `jshell`. With all the structure we have designed so far, playing should have become as easy as:

1. Starting a game with `Game game = new Game(<SOME SECRET HERE>);`
2. Try a number with `game.guess(13)` (assuming you have named your function `guess`)
3. Repeating step 2. adjusting your guess based on the feedback provided.

Once it works in `jshell`, make sure the class file also compiles with `javac`.

Ok, not bad but usually when we play a game we don't have to manually type commands to a Java prompt. Please meet the `Scanner` class (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Scanner.html)

- read about it, look at the examples provided
- try and reproduce them in jshell
- pay attention to what happens when the input doesn't represent a proper integer: what feature covered in today's class could help us handle that?
- once you're confortable enough with this new tool, turn it into a new method of your `Game` class called `prompt`. Does it need to access any context? Should you make it `static` or not?
- get rid of `guess`'s last parameter and call the new `prompt` method instead to declare the (local) variable

Keep checking everything still compiles (hint: need an import? look at the documentation). Next step will be to have Java call this method automatically in a loop but... wait! How can we know when to stop?

- alter the type of your method to provide enough information to its caller to know whether to stop or continue
- check it works by writing a very simple (so simple it may look suspicious) loop using it in `jshell`

A command-line program

Now that we have all the logics for the game in a class, let's expose it as a command-line program to start the game.

- Create an empty `Main` class in a new `Main.java` file holding the structure for an empty program (draw inspiration from the last section of the slides). Verify you can compile it with `javac` and run it with `java`
- Running the game should be just: instanciating a new `Game`, then using it in the same loop you have been practicing at the end of the previous part. Implement the `main` function accordingly

We have made a very simple game!

Suggested improvements

- the secret value is so far built into your game, once someone has guessed it it has solved the game once and for all. Let's make the game a little bit more interesting by having *the computer* choose the `secret` itself. Please meet the `Random` class (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Random.html)
- again: try in `jshell`, look at the results returned, how could we leverage that into something usable for our game?
- replace your built-in secret by a random one

One last thing: the range on which you play is still arbitrary. Let's replace that by a parameter on the command line.

- modify the `main` function of your class to
  - make sure only one argument is passed to your program
  - convert it to an `int`, not forgetting to handle the errors which may arise
  - use this value as the upper bound for your random number

We're done! We have an actually playable game, and it doesn't require its users to type Java or to know anything about the `jshell`.

Extra fun for the Java enthuasiast

Namespace occupation

First draft

The euclidian norm in a 2D-space ("plane") can be defined as a function of the coordinates (x_A, y_A) of a given point A.

$$|A| = \sqrt{x_A^2 + y_A^2}$$

Implement this as a function with signature: `double norm(double x, double y)`. You may have to browse the documentation to find functions to compute the square and square root of a given input number.

Test it on some values where you know the result in advance: the origin `(0, 0)`, the unit on respectively the x and y axis `(1, 0)` and `(0, 1)`, the opposite corner from the origin on the unit square `(1, 1)` (remember that $\sqrt{2} \approx 1.414$).

Now, let's define the same function but for a 3D-space. Can you keep the same name ? What do you suggest ?

Sharing the namespace

2D-Points

- Declare a simple class `Point2D` to model a geometric point in the plane (2D space). What fields must it contain ? What type will you use for them ?
- Add a `norm` method to compute the same metric as previously. How will the signature differ from the signature of the function defined above ?

3D-Points

- Now create a new class to represent points in a 3D space.
- Add the corresponding `norm` method to it too. Is there any naming conflict ?

Pass by value / pass by reference

- Declare a `short` value `age` and assign it to `4`.
- Create a (`void`) procedure called `increment` that increases the value of its only (`short`) argument by `1` (modifies it without returning it).
- Apply it on `age`. Did you get any error? Was `age` modified?

A native value is nothing more than a (temporary) value in the processor. There's no memory location available. When a function is called in Java, it is passed a copy of its arguments. They are "passed by value". Passing by value is simpler and closer to the mathematical intuition of functions, but it can be costly for huge data structures.

We're now going to define a special wrapper for the `short` type where we have access to the inner value : `RWShort` ("read-write short"). This class should have one field of type `short` named `value`.

- Declare a `RWShort` value called `rwAge` holding the same value `4`.
- Modify `increment` so that it now works on `RWShort` and not `short`
- Use it on the `rwAge`. Is it modified now ? Why ?

Java passes arguments by value; but since the value of objects are their reference in memory, this actually amounts to passing by reference ! We get the best of both worlds. Again, understanding the duality in Java datastructures between "what is directly a number" and "what is a complex data implented by a graph of number values" helps us understand why things are working the way they do.

Automatically numbering instances with `static`

- Declare a class called `Serial` having only one `long` field and one constructor taking as argument the number to "wrap" as a `Serial`.
- Now modify it to add a `static` field initialized to `0`.
- Remove the argument to the constructor, and use the value of the `static` field instead, while incrementing it. The `static` field is keeping track of the highest number and "magically" guarantees to generate distinct numbers.

- Complete the class with an `.olderThan` method that takes another `Serial` as argument and is able to return `true` if and only if it was created after the target object (the one on which the method is applied).

This pattern can be very useful to keep track of objects created throughout an application.