TP3

Guillaume Lachaud

2022-10-09

1 Building a fullproof scanner

1.1 Presentation

In Java, a scanner can be used to retrieve the inputs from the user. A basic Scanner example is presented in Listing 1. Note that after a scanner has been used, it must be closed, otherwise it may leak memory. Once the scanner has closed the `System.in` (your console interface), it it not possible to open it again during the same run of the program.

```
1 Scanner sc = new Scanner(System.in);
2 System.out.println("What is your age?");
3 int answer = sc.nextInt();
4 System.out.printf("Your age is \%d\%n", answer);
5 sc.close();
```

Listing 1: Basic use of a scanner

Unfortunately, if the user provides something other than an `int`, the program will terminate with an Exception (`java.util.InputMismathException`). Other exceptions may be raised depending on the functions that are used. To determine which exceptions might be raised, you can either look at the documentation of the functions, or at the code itself. Solving the exception problem can be done in two manners.

The first one consists in using `try` and `catch` statements, as in done in Listing 2.

Using `try` and `catch` statements is not optimal because it interrupts the flow of the program. Whenever it is possible, it is more appropriate to use functions that perform the check on the input without raising exceptions. An example of such method and its use is shown in Listing 3.

```java
try{
    // statements that you want to perform
} catch(InputMismathException e) {
    // statements that you want to perform
    // if the original ones failed
}
```

Listing 2: Try and catch blocks

```java
Scanner sc = new Scanner(System.in);
    while(!sc.hasNextInt()){
        // your code
    }
```

Listing 3: Use of the `hasNextInt` function

Additionally, to avoid having to think about closing the `Scanner`, the `try` and `catch` blocks can be adapted by incorporating the `Scanner` declaration and initialization in the `try` statement, as is shown in Listing 4. In this setting, there is no need to add a `catch` block to go with the `try` block that instantiates the `Scanner`.

```java
try(Scanner sc = new Scanner(System.in)){
    // your code, e.g. try {...} catch(...) {...}
}
```

Listing 4: **try-with** statement

If the user continues to input invalid arguments, we must ensure that the program does not continue with bad input. In other words, we must force the user to provide us with good inputs. *While* the user's input is incorrect, we ask for new input. The code will look similar to the code in Listing 5 [1].

Instead of writing the same code everytime a `Scanner` is required, we can use extract the routine into a function, in a similar vein to Listing 6.

Finally, if the `Scanner` is to be used for different classes in different files, we can export the scanning task to a completely different class. The class will be similar to the one in Listing 7.

[1] You can choose to use either the **try-with** method from Listing 4 or not.

```java
boolean inputValid = false;
while(!inputValid){
    try{
    // your code
    inputValid = true;
    } catch(InputMismathException e){
    // do something else
    }
}
```

Listing 5: `while` loop to enforce proper input.

This class can be used in other classes, as is shown is Listing 8.

1.2 Exercise

Your task is to create the **SafeScanner** aforementioned, and to create other classes to use it.

1.3 Advanced use of scanner

When testing your code, instead of always writing things in the console, one can modify the `Scanner` class to allow the use of a different `InputStream` than `System.in`. See Listings 9 and 10 for an example.

demo.txt is a text file whose content is displayed in Listing 11.

Additionally, the structure of the project has been changed and now follows the one in Figure 1. *java* is declared as a *source* directory, while *resources* is declared as a resource directory.

```java
class Main{
    public static void main(String[] args){
        // your code
        Scanner sc = new Scanner(System.in);
        int answer = getInt(sc);
        // ...
        sc.close();
    }
    public static int getInt(Scanner sc){
        boolean inputValid = false;
        while(!inputValid){
            // etc.
        }
    }
}
```

Listing 6: Extracting the scanner routine into a function

```java
import java.util.Scanner;

public class SafeScanner {
  Scanner sc;
  public SafeScanner() {
    this.sc = new Scanner(System.in);
  }

  public int getInt(){
    // routine to get an integer
  }

  public void closeScanner(){
    sc.close();
  }
}
```

Listing 7: `Scanner` class

```java
public class Main {
  public static void main(String[] args) {
    SafeScanner safeScanner = new SafeScanner();
    System.out.println("What is your age?");
    int answer = safeScanner.getInt();
    System.out.printf("Your age is %d.%n", answer);
    safeScanner.closeScanner();
  }
}
```

Listing 8: How to use our newly created **SafeScanner** class

```java
import java.io.InputStream;
import java.util.Scanner;

public class SafeScanner {
  Scanner sc;

  public SafeScanner(InputStream is) {
    this.sc = new Scanner(is);
  }
  // rest of the code stays the same
}
```

Listing 9: `Scanner` with more flexible `InputStream`

```java
import java.util.Objects;

public class Main {
  public static void main(String[] args) {
    SafeScanner safeScanner = new SafeScanner(
        Objects.requireNonNull(SafeScanner.class.getResourceAsStream("demo.txt"))
        );
    System.out.println("What is your age?");
    int answer = safeScanner.getInt();
    System.out.printf("Your age is %d.%n", answer);
    safeScanner.closeScanner();
  }
}
```

Listing 10: Use of **SafeScanner** with a text file

```
32
```

Listing 11: Contents of *demo.txt* file

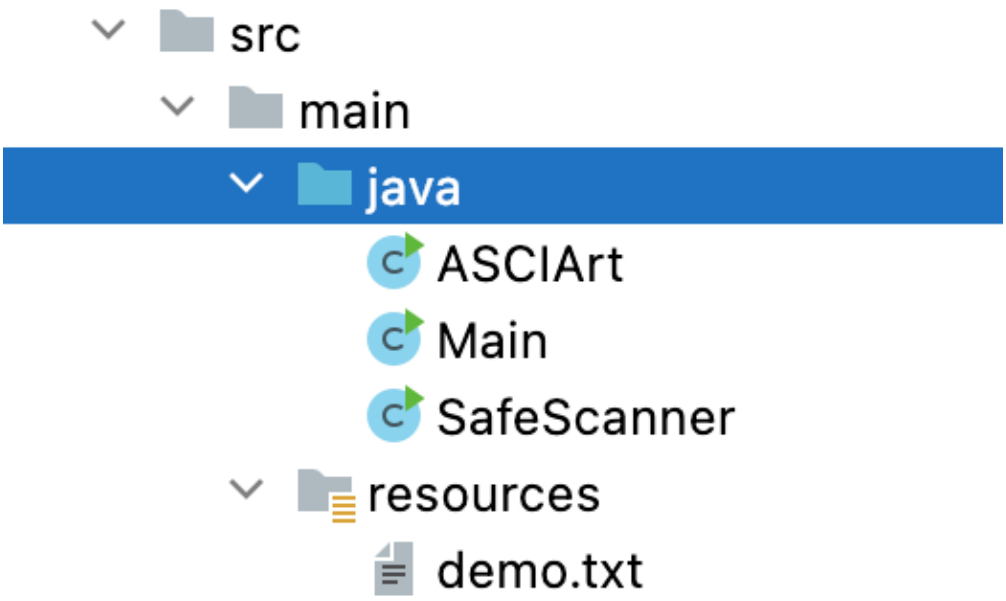
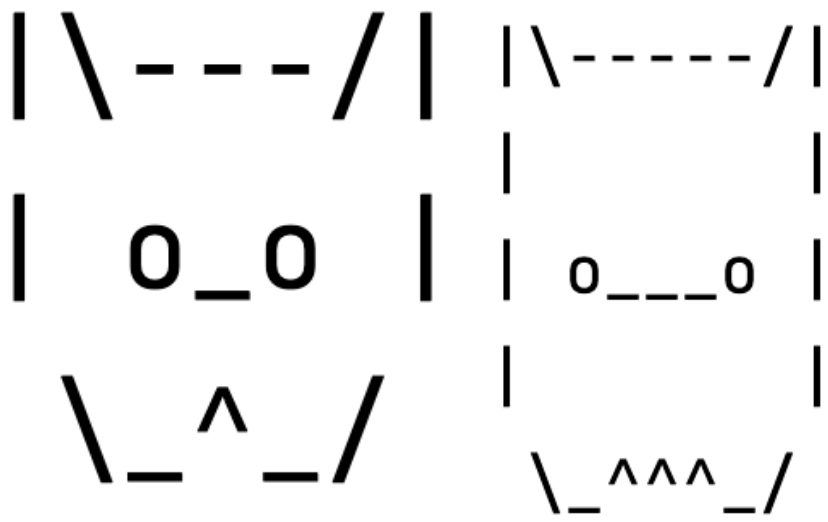


Figure 1: Project structure of our code.

2 ASCII art

The following exercise is simple: write a function that takes *height* and *width* parameters and prints a cat of the appropriate size, such as in Figure 2.



(a) Small cat (b) Bigger cat

Figure 2: Cats!