

## Mango Eclipse plugin High School Code example

Here is an example of the Mango plugin at work inside the Eclipse workbench, analyzing the method [CarRecall.main\(\)](#). The code is shown in the familiar Eclipse view. The red dot in the gutter represents the loop [1]. Opening the red dot to causes a green "Tr" to appear in the Mango Explorer view [2]. As you can see, the "Tr" is now fully opened, revealing all possible paths through the loop as a hierarchy. Double-clicking on the selected "Tr 1.2" [3] reveals that particular case as a view, which appears below the code [4]. The view is "functional", just as the code is "procedural". In this functional view there are input state assumptions and resulting output state. Some of the assumptions are boring and are in yellow. The interesting assumptions are in brown [5] , and indicate that the input is less than 189, but not equal to 179, 119, or 0. In this case, as you can see front the output state, the car is not defective [6]. Now go back and look at the code. This is clearly an intended outcome. Viewing all the possible outcomes from the functional point of view increases confidence in the correctness of the code.

[CarRecall](#) was written by a high school student in an introductory Java course. With this tool, students can see the relationship between procedural and functional descriptions. There is also the potential for professional programmers to use the Mango Eclipse plugin during development as a debugging aid.

## Car Recall Example

The screenshot displays the Mango Explorer IDE with a Java project. The main editor shows the `CarRecall.java` file with the following code:

```

public class CarRecall {
    public static void main(String[] args){
        int modelNum;
        Scanner input = new Scanner (System.in);
        do{
            System.out.println("Enter the car's model number or 0 to quit: ");
            modelNum=input.nextInt();

            //test if car is defective
            if ((modelNum==119)|| (modelNum==179)
                ||((modelNum>=189)&&(modelNum<=195))
                ||(modelNum==221)|| (modelNum==780))
            {
                System.out.println("Your car is defective. It must be repaired.");
            }
            else if (modelNum==0)
            {
                System.out.println ("Program terminated.");
            }
            else
            {
                System.out.println ("Your car is not defective.");
            }
        }while (modelNum!=0);
    }
}

```

The left sidebar shows a project tree with a call stack for `firstYearCode.CarRecall.main`. The bottom console window shows the following output:

```

firstYearCode.CarRecall.main([Ljava/lang/String;):V loops loop at 37 System.out.println("Enter the car's model number or 0 to quit
input.nextInt()I resolves to java.util.Scanner.nextInt()I
input.nextInt()I does not throw invocation exception
input is defined
Integer_scan[input.counter + 1] is defined
Integer_scan[input.counter + 1].value does not equal 119
Integer_scan[input.counter + 1].value does not equal 179
Integer_scan[input.counter + 1].value is less than 189
Integer_scan[input.counter + 1].value does not equal 0
out.println(Ljava/lang/String;):V resolves to java.io.PrintStream.println(Ljava/lang/String;):V
out.println(Ljava/lang/String;):V does not throw invocation exception
out is defined

```

The 'output state' window shows the following details:

- heap**
  - <reveal> at #0: `#0.value=Enter the car's model number or 0 to quit:`
  - <opVar> at unresolved location: `op0.buffer=op0.buffer + Enter the car's model number or 0 to quit: <nl>`
  - <localVar> at unresolved location: `input.counter=input.counter + 1`
  - <reveal> at unresolved location: `outinputStaticArea.buffer=out.buffer^ + Your car is not defective.<nl>`
- stack**
  - `modelNum = Integer_scan[input.counter + 1].value`
  - `op0 = outinputStaticArea`