Coding in Java vs. RIS

To some extent programming is programming, no matter which language you use. Java is a text-based, object-oriented language while RIS is a visual, procedural language, which makes the two very different. But they do have many similarities. This section displays RIS code, which you have learned in previous chapters, and Java code. Some of the Java code will be complete programs while other code will be code fragments. You can cut and paste the Java code into your editor, compile it and download it to your robot.

At left is a complete RIS program. At right is the equivalent Java program. Both programs can be downloaded to the RCX and both do nothing.

class DoNothing {
    static void main(String[] args) {
    }
}

At left is a complete RIS program. Its name is Forward and it has two statements, a small block that sets the direction of the motors and a second block that turns on the A & C motors. The Java seems more complex. It starts with an import statement that makes RCX objects available to our program. The class statement defines a Forward object, whose “objectness” we ignore in this simple program. We define one method, main, which is our entry point for our program. main has a parameter, an array of Strings, which we also do not use.

import josx.platform.rcx.*;
class Forward {
    static void main(String[] args) {
        Motor.A.forward();
        Motor.C.forward();
    }
}
Forward, the main program, at right, has one statement, a big block Forward. Forward starts both motors in the forward direction for one section. Big blocks are subroutines, the equivalent to a Java method. The Forward big block is expanded at far right. The algorithm for Forward has four steps: set the motors’ direction, turn the motors on, wait for one second and then turn the motors off. The Java code, below, does the same thing. The entry point is main and there is a single statement, goForward(1000). Java times are in milliseconds so 1000 milliseconds is one second. The method goForward has the same statements as the Forward big block. The Motor.A.forward method both sets the direction and turns the motor on. Thread.sleep suspends execution for the specified time just as the Wait For block does in the RIS code. Finally, the Motor.A.flt statement turns off the motors. Again, goForward is static; it can be used without an associated object.

```java
import josx.platform.rcx.*;

class Forward {
    static void goForward(int time) {
        Motor.A.forward();
        Motor.C.forward();
        try {
            Thread.sleep(time);
        } catch (InterruptedException e) {}
        Motor.A.flt();
        Motor.C.flt();
    }

    static void main(String[] args) {
        goForward(1000);
    }
}
```

This is a full program that you can cut and paste into your editor.
Declaring Variables

Declaring and assigning values to variables in RIS is done through a set variable small block. But RIS, like Java, requires declaring variables before they can be used. In RIS, you pick a variable to assign a value to through the set window. You can pick an existing variable or create a new one and then assign a value to it. RIS has only one variable type, an integer that is displayed as a floating point number with only one decimal place and a range from –3276.8 to 3276.7.

```java
boolean flg = true;
int intVar = 0;
float floatVar = 0.0;
char charVar = 'a';
```

Java has 8 different primitive types: boolean for true or false, 4 sizes of integers (byte, short, int and long), two sizes of floating point numbers (float and double) and char for storing letters. The biggest two, long and double, are not supported by LeJOS because there is no practical reason for such large numbers in a small robot. Variables should be declared as close as possible to their first use.

```java
{ 
  int idx = 0;
  if (idx == 0) {
    int jdx = 0;
    // idx & jdx exist
  }
  // idx exists
  // jdx does not exist
}
```

Variables exist only within the block where they are declared and within blocks contained within that block. The code fragment at left illustrates this principle.
**Conditional Execution**

**if statement**

Conditional execution, the if statement, refers to executing code fragments depending upon the values of variables in the program. At right is the RIS code for conditional execution and the Java code is at left. In RIS code, if the condition is true, the code in the Yes path is executed, otherwise the code in the No path is executed. In Java, if the condition expression evaluates to a boolean true, code fragment 1 is executed, otherwise code fragment 2 is executed. In both languages you can nest if statements, that is, one of the code fragments can be another if statement. In RIS code, because of its visual nature, it is clear which yes and no fragments are in a given if statement; in Java it may not so clear because you are not required to have an else with each if as you are in RIS.

The rule in Java for matching else’s with if’s is simple; each else goes to closest preceding, unmatched if at the same block level. That is, if you enclose an if statement inside a block, place it between “{“ and “}”, then it doesn’t match with an else outside the block. The example at right demonstrates if’s and else’s matching. Notice that each if matches the first else that follows it except for the c4 if. That is because the c4 if enclosed within a block and there is no else within the block.

```java
if (condition) {
    Code-fragment 1
} else {
    Code-fragment 2
}
```
switch statement
Java has two conditional statements that RIS does not have: switch and try-catch. The switch statement appears at right. The statement works by evaluating the caseVar, which may be a simple variable or an expression, and goes to case label that has the literal value of the caseVar. For example, if caseVar had the value 1, code fragment 1 would be executed. The break statement that follows the code fragment causes execution to jump to the end of the switch statement. Without the break, execution continues through the rest of the cases until a break is encountered. The default is executed whenever no case label matches the caseVar.

try-catch-finally statement
The try-catch-finally statement is a modern innovation. The block between the try and the catch are checked for runtime errors. In this case, an array of five ints is created. The program then tries to assign a value into the array position 6. Since not enough space has been allocated, this causes an error. The catch block is executed if and only if the specified error has occurred. A try can have multiple catches if each catch specifies a different type of error. User defined methods can specify different errors that they can throw and any method that specifies an error in its definition must be inside a try-catch statement. Finally, the finally clause of a try-catch-finally statement is optional. If a finally clause is given, its code is always executed, whether or not an error occurs.
**Loops**

The next section compares RIS code fragments to their equivalent Java code fragments. You can experiment with these fragments by creating the called method, e.g. `turnLeft`, in the Forward class file and replacing the `goForward(1000)` method call with the respective code fragment.

These fragments are flow-control statements. If-then, conditionals, allow the execution of a program to proceed along different paths. Loops are the second type of flow-control statements and there are three distinct types of loops: for, while and do until. The repeat statement in RIS translates into the three different types of loops.

**for loop**

The for loop is equivalent to the RIS repeat with a count in the bottom, which is pictured at left, and at right, the Java code is listed. In Java, the for statement has three parts that are contained within the parentheses and are separated by semicolons. The first part is the initialization part, the second is the test part and the third section is the increment part. The initialization part usually contains the declaration of one or more variables, the loop variables, and their initialization to 0 (note that this is the usual case and not, in any way, required). The test section compares the loop variable to some control value, in this case 3. The increment section changes the value of the loop variable after all the statements within the body of the loop have been executed. The for loop is the most complex of the three different loop types.
while loop

The while loop is equivalent to the RIS repeat with a while in the bottom. With a while loop, the condition is tested first. If the value is true then the body of the loop is executed. The body continues to be executed as long as the condition remains. When it becomes false, the next statement after the body of the loop is executed. Note that the body of a while loop may, or may not, be executed depending upon whether or not the condition evaluates to true when the while is executed the first time.

```java
Static void spinLeft(int tim) {
    Motor.A.backward();
    Motor.C.forward();
    try {
        Thread.sleep(tim);
    } catch (Exception e) {
    }
    Motor.A.stop();
    Motor.C.stop();
}

public static void main(String[] args) {
    while (Sensor.S1.readBooleanValue()) {
        spinLeft(1000);
    }
}
```

do until loop

The do until loop is equivalent to the RIS repeat with an until in the bottom. With a do until loop, the condition is not tested first; the body of the loop is always executed at least once.

```java
Static void spinRight(int tim) {
    Motor.A.forward();
    Motor.C.backward();
    try {
        Thread.sleep(tim);
    } catch (Exception e) {
    }
    Motor.A.stop();
    Motor.C.stop();
}

public static void main(String[] args) {
    do {
        spinRight(1000);
    } until (Sensor.S1.readBooleanValue())
}
```
The repeat forever loop in RIS does not have a unique Java equivalent; it is just a special case of a while loop. In this case, the logical literal true replaces a condition. This is called an infinite loop because it never exits normally; there must be some kind of test inside the loop that interrupts the normal flow of the program.

```java
Static void turnRight(int tim) {
    Motor.A.forward();
    Motor.C.stop();
    try {
        Thread.sleep(tim);
    } catch (Exception e) {
    }
    Motor.A.stop();
    Motor.C.stop();
}

public static void main(String[] args) {
    while (true) {
        turnRight(1000);
    }
}
```

break, continue, return & throw

There are four statements that alter the flow inside of a loop: break, continue, return and throw. break terminates the loop and starts executing the statement immediately after the body of the loop. continue skips to the end of the body of the loop and performs the loop test. Note that in a for loop, the increment part of the for statement is executed before the test is performed. return acts as a return does anywhere else; it exits the current method and continues execution in the method that called the method that contains the loop. Finally, throw throws an exception that must the caught in the catch part of a try-catch statement.
Exercises

1. The three different loop types, for, while and do until, can all be used to do the same things. Using a for statement, write code that is equivalent to a while and a do until. Do the same for the other cases. You should wind up with six different code fragments.