Java Boot Camp

Containers, Streams, Functional and Reactive Programming

2. Functional Programming

Programming Paradigms

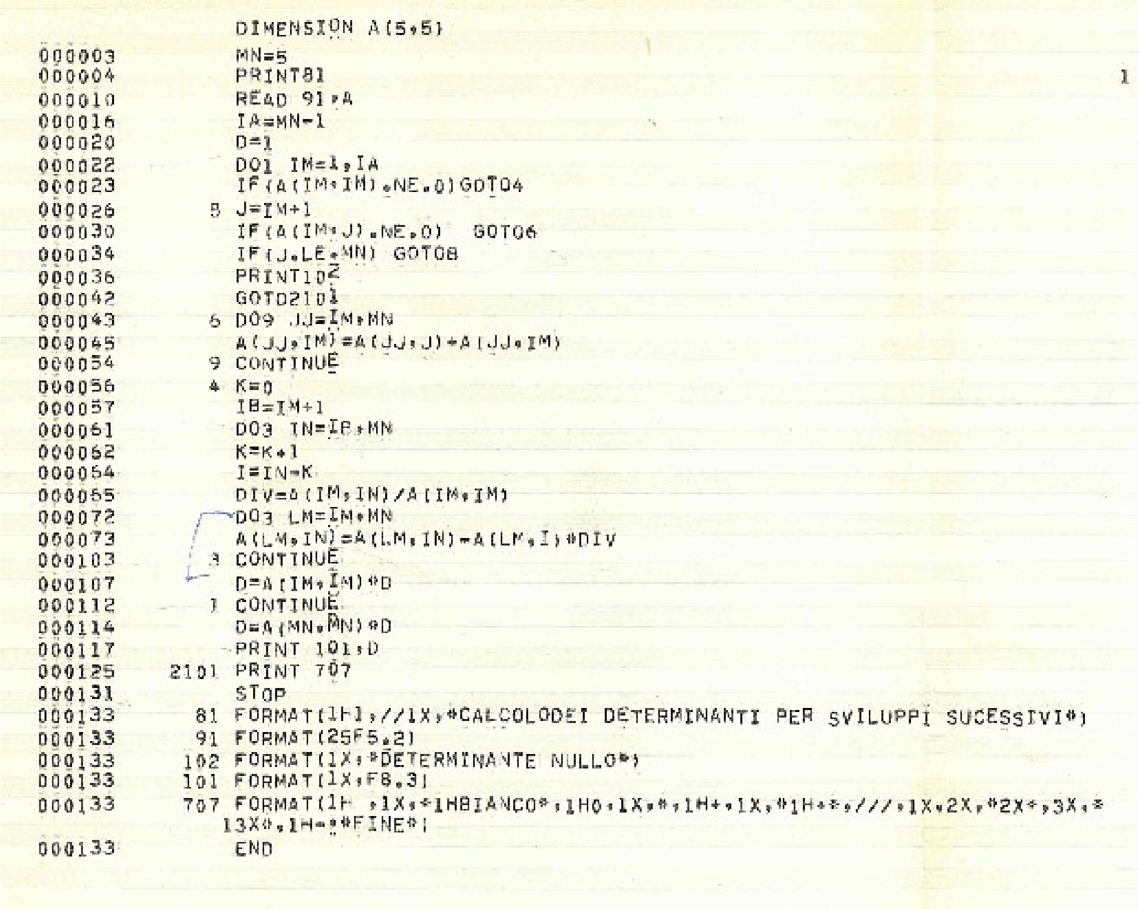
- A programming paradigm is a set of:
  - Assumptions about what the components of a program should be
  - Techniques and principles for building programs
  - Assumptions about what sort of problems are solved best by that paradigm
- There are multiple programming paradigms
  - Most have been around since the start of high level programming in the 1960s
  - A paradigm becomes “main-stream” and supported in programming languages when a set of problems arise that the paradigm is better suited to solve than the other paradigms in use
- The main-stream paradigms in use today:
  - Structured or procedural programming
  - Object Oriented Programming
  - Functional Programming

Programming Style

- Imperative programming
  - Code is a series of instructions that specify how a program execution is to be done
  - Intended to be easily compiled into assembly code
  - Procedural and OO programming tend use the imperative style
- Declarative programming
  - Code is a description of what a final result should be
  - Described as a series of transformation
  - The transformations are turned into executable code by the language
  - Requires a layer of abstraction to hide the imperative code
  - Functional and symbolic programming tend to use the declarative style

Procedural Code

- An abstraction of low level assembly code
  - Was often coupled to the hardware and OS environment it ran in
  - Eg. C was written for OS development
- In Java
  - Executable code in methods is procedural code
  - OO is more about how the code is organized and managed at run-time
  - OO programming extends procedural programming



```
                 DIMENSION A(5,5)
000003           MN=5
000004           PRINT81                                                  1
000010           READ 91,A
000016           IA=MN-1
000020           D=1
000022           DO1 IM=1,IA
000023           IF(A(IM,IM).NE.0)GOTO4
000026         5 J=IM+1
000030           IF(A(IM,J).NE.0)   GOTO6
000034           IF(J.LE.MN) GOTO8
000036           PRINT102
000042           GOTO2101
000043         6 DO9 JJ=IM,MN
000045           A(JJ,IM)=A(JJ,J)+A(JJ,IM)
000054         9 CONTINUE
000056         4 K=0
000057           IB=IM+1
000061           DO3 IN=IB,MN
000062           K=K+1
000054           I=IN-K
000065           DIV=A(IM,IN)/A(IM,IM)
000072           DO3 LM=IM,MN
000073           A(LM,IN)=A(LM,IN)-A(LM,I)*DIV
000103         3 CONTINUE
000107           D=A(IM,IM)*D
000112         1 CONTINUE
000114           D=A(MN,MN)*D
000117           PRINT 101,D
000125      2101 PRINT 707
000131           STOP
000133        81 FORMAT(1H1,//1X,*CALCOLODEI DETERMINANTI PER SVILUPPI SUCESSIVI*)
000133        91 FORMAT(25F5.2)
000133       102 FORMAT(1X,*DETERMINANTE NULLO*)
000133       101 FORMAT(1X,F8.3)
000133       707 FORMAT(1H ,1X,*1HBIANCO*,1H0,1X,*,1H+,1X,*1H+*,///,1X,2X,*2X*,3X,*
              13X*,1H-,*FINE*)
000133           END
```

OO Code

- An abstraction of low level assembly code
  - Was often coupled to the hardware and OS environment it ran in
  - Eg. C was written for OS development
- In Java
  - Methods are procedural code
  - Designed that way for easy adoption by programmers used to writing procedural code (mostly C and C++ developers)
  - Almost 30 years old and reflects the tech of the times
  - OO is about program design and organization
  - Example is from 1967 Simula code

```
Begin
   Class Glyph;
      Virtual: Procedure print Is Procedure print;;
   Begin
   End;

   Glyph Class Char (c);
      Character c;
   Begin
      Procedure print;
        OutChar(c);
   End;

   Glyph Class Line (elements);
      Ref (Glyph) Array elements;
   Begin
      Procedure print;
      Begin
         Integer i;
         For i:= 1 Step 1 Until UpperBound (elements, 1) Do
            elements (i).print;
         OutImage;
      End;
   End;

   Ref (Glyph) rg;
   Ref (Glyph) Array rgs (1 : 4);

   ! Main program;
   rgs (1):- New Char ('A');
   rgs (2):- New Char ('b');
   rgs (3):- New Char ('b');
   rgs (4):- New Char ('a');
   rg:- New Line (rgs);
   rg.print;
End;
```

Functional Code

- Uses a declarative style
  - Support for functional programming is common with most modern programming languages
  - Modeled after math notation
- One of the first languages to use functional programming was LISP
  - Dates from the 1950s
  - Earlier than FORTRAN
- Shown is a listing in APL for computing the determinant of a matrix
  - Developed in the 1960s

```
      ∇DET[⎕]∇
     ∇ Z←DET A;B;P;I
[1]    I←⎕IO
[2]    Z←1
[3]   L:P←(|A[;I])⍳⌈/|A[;I]
[4]    →(P=I)/LL
[5]    A[I,P;]←A[P,I;]
[6]    Z←-Z
[7]   LL:Z←Z×B←A[I;I]
[8]    →(0 1 ∨.=Z,1↑⍴A)/0
[9]    A←1 1 ↓A-(A[;I]÷B)∘.×A[I;]
[10]   →L
[11]  ⍝EVALUATES A DETERMINANT
     ∇
```

Programming Paradigms

- Most programming languages support more than one paradigm
  - Languages are often revised to add support for a paradigm
    - COBOL added object oriented support
    - Java added support for functional programming in Java 8
  - Modern languages like Rust, Go and Julia are designed to support multiple paradigms
- Why paradigms go mainstream
  - Most of the different paradigms have existed for over 50 years
  - They are designed to solve a particular class of problems
  - The types of problems industry deals with change over time
  - Changes often result from changes in technology and the market place
  - Existing paradigms may not be able to solve these new problems
  - A different paradigm is main-streamed that can solve the problems

Paradigm Focus

- Procedural programming
  - Was the ideal tool to support the automation of business processes in the 1960s
  - A data set was read in, algorithms used to process it, the results written out
  - Typically done in batch mode on a mainframe
  - There is still a massive installed based of COBOL and other imperative code still running business operations in the public and private sector
- Object Oriented programming
  - Ideal tool to handle distributed computing with the rise of the Internet in the 1990s
  - Enabled different nodes of computing to collaborate
- Functional programming
  - Ideal tool to handle streaming data at scale
  - The need for arose in the 2010s with the rise of big data
  - Cannot be done easily or efficiently with procedural or object oriented approaches

Functional Programming

- Dates back to Church's Lambda calculus in the 1930s
- Procedural and OO code update the state of the running program
- Functional code is based on the idea of a mathematical function
  - e.g. Square function: f(x) = x * x
  - Functions do not change the input data but transform it to a new value
  - In pure functional programming languages, variables are immutable, they just bind to new values
  - A functional program maps a set of data to a new set of data
- Complex computations are done by functional composition
  - e.g. f(g(x)) produces an output where f() takes as input the result of applying g() to an input x
  - Algorithms are expressed as a series of functions representing the steps of the algorithms
  - Each step of the algorithm is implemented as a function
  - A program can be represented as a series of function calls

Functional Programming in Java

- Functions are first class objects
  - A function body is data, called a function literal, just like “Hi There” is a string literal.
  - A function can be assigned to a variable
  - A function can be passed as an argument to another function
  - A function can be the return value from another function
- A function literal is written using Lambda notation
- Java is a strongly typed language
  - Every variable must have a type that can be checked at compile time
  - That means functions have to have types
  - Defined to be the type of return value plus the number and types of arguments
  - Similar to the signature of a method, but includes the return value

Functional Programming in Java

- The different types of functional interfaces are defined in java.util.function
- Some basic types are:
  - Function<R,T>: takes one argument of type T and returns an value of type R
  - BiFunction<R,T,U>: takes two arguments of types T and U and returns a value of type R
  - Predicate<T>: takes one argument of type T and returns a Boolean
  - Consumer<T>: takes one argument of type T and does not return a value
  - Supplier<R>: takes no arguments and returns a value of type R
- There a number of different types that are variations of the above
  - Consult the documentation
- To execute the body of the function, we use different methods, for example
  - **apply()** for functions
  - **test()** for predicates
  - **get()** for suppliers and accept() for consumers

Functional Programming in Java

```java
public class Main {

// Define several static variables to hold different function types
    public static Function<Integer,Integer> square;
    public static Predicate<Integer> isEven;
    public static BiFunction<Integer,Integer,Integer> sum;
    public static Supplier<String> today;
    // This variable is initialized
    public static Consumer<String> printLength = (s) -> System.out.print(s.length());

    public static void main(String[] args) {
        // Assign the variables values in the form of Lambda expressions
        // since the lambda expression is one line, we omit the {} by convention
        square = (x) -> x * x ;
        System.out.println("The square of 7 is " + square.apply(7));
```


Java FP Basics
Demo One
ITHACA

Functions

- Functions are stored in memory like other Java objects
  - They have addresses and types
  - Functional variables are references to Lambda expressions on the heap
  - Methods are NOT functions, they are stored completely differently
  - Functions are objects, methods are not objects
  - Function bodies can be used in the same way that other data is used in Java
- Java is not a functional language - it has functional support
- Functional languages have the following requirements
  - Variables are immutable
  - Functions are pure - they have no side effects
  - Functions do not rely on anything that an change
  - These specific requirements cannot be implemented in Java

Java FP Addressing

Demo Two

FP Basics

Lab FP1

Functions as Parameters

- Functions can be passed as parameters
  - However, the type of function that can be passed is strictly checked
  - For example, we can't pass a Predicate<T> with a parameter expecting Function<T,T>
- This a functional programming implementation of a strategy pattern
  - A specific piece of code to be executed is provided to a function at execution time
  - This is often called "meta-programming" where code can modify itself at run time
  - The decision on what specific code should be executed is decided at run time
- This makes for simpler code in many cases
  - Otherwise all possible alternatives would have to be provided at compile time with the correct option determined by some sort of test in the program environment
  - This can result in large blocks of conditional code that are hard to write, read and maintain

Java FP Parameters

Demo Three

Functions as Return Values

- This is often used to implement a dispatch table
  - During execution, depending on some condition, a number of different functions could be called
  - A dispatch table or function, checks the condition and returns the function to be executed
- Another use is to write a function at run-time
  - Recall this is referred to meta-programming
- In the previous demo, we decided what preexisting function to execute at run-time
- In following demo, we create a function to be used at run-time
  - This is useful when we don’t want to apply the specific function right away, but maybe send it off to another function as parameter later on

Java FP Return Values

Demo Four

FP Meta-programming

Lab FP2

Functional Interfaces

- A functional interface in Java is an interface with one abstract method
  - A functional interface must have exactly one abstract method that is the “exposed functionality” of that interface
  - Also called SAM or a “single abstract method” interface
  - It may have other static and default methods but usually they don’t
- The standard Java OO process for using interfaces is
  - Define the interface with an abstract method
  - Write a class that implements the interface
  - Provide an implementation that overrides the interface definition
  - This is a lot of extra coding, especially when we might use the class only once

Functional Interfaces

- Inner classes allow us to create disposable objects that implement an interface
  - Like a Lamba function, we are essentially created a class literal or class implementation and not assigning it to a variable
- Functional Interfaces have only one abstract method
  - In order to create an implementation, all we have to do is provide a Lambda to act as the body of the abstract method
  - The compiler knows which method the Lambda implements because there is only one abstract method in a functional interface
  - As long as we provide a single Lambda to override the abstract method, Java can do all the rest of the work to create a class implementation under the hood
- Many of the interfaces in the Java library are functional interfaces
  - The Runnable interface for example

Functional Interfaces

Demo Five

Runnable Lambdas

- Threads in Java require a Runnable object that either
  - is a class that extends thread and overrides the run() method
  - is a class that implements the Runnable interface and overrides the run method
- Lamba functions are inherently runnable
  - They can be assigned to variables of type runnable
  - They can be passed directly to threads
  - Because Runnable is a functional interface, it is clear what the role of the Lambda is

Runnable Interfaces

Demo Six

FP Functional Interfaces

Lab FP3

The Closure Problem

- In a pure functional language
  - Functions do not depend on anything that changes
  - Functions do not change things
- Java does not support these two properties
- In a functional language
  - There may be variables used in the body that are not either
    - declared in the function body
    - passed as parameters
  - These are call free variables
  - If they change, then the function will behave differently
  - The function result now depends on an external variable that can change

LearnQuest

The Closure Problem

- In a functional language
  - Then the function is created, it binds the free variables to the value it had when the function was created
  - In other words, it evaluates the variable at assignment time
- This is called a “closure”
  - A closure is an object containing the function body and the bound values of the free variables
  - A closure closes the function to outside changes that could be introduced by free variables

Closures

Demo Seven

Questions?

End Module


CMA CGM
CMAU 115887 0
22G1
CMA CGM
CMAU 159369 3
22G1
CMA CGM
DELMAS
DVRU 147046 5
22G1
DELMAS
DVRU 143916 1
22G1
GESU 276340 7
22G1
TRLU 308341 2
US 2210
CNC LINE
凯 航
tex