Introduction

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 1

Programming languages

- A language is a medium for communication

Programming languages

- A language is a medium for communication
- Programming languages communicate computational instructions

Programming languages

- A language is a medium for communication
- Programming languages communicate computational instructions
- Originally, directly connected to architecture
  - Memory locations store values, registers allow arithmetic
  - Load a value from memory location M into register R
  - Add the contents of register R_1 and R_2 and store the result back in R_1
  - Write the value in R_1 to memory location M'

Programming languages

- A language is a medium for communication
- Programming languages communicate computational instructions
- Originally, directly connected to architecture
  - Memory locations store values, registers allow arithmetic
  - Load a value from memory location M into register R
  - Add the contents of register R_1 and R_2 and store the result back in R_1
  - Write the value in R_1 to memory location M'
- Tedious and error-prone

Abstraction

- Abstractions used in computational thinking
  - Assigning values to named variables
  - Conditional execution
  - Iteration
  - Functions / procedures, recursion
  - Aggregate data structures — arrays, lists, dictionaries

Abstraction

- Abstractions used in computational thinking
  - Assigning values to named variables
  - Conditional execution
  - Iteration
  - Functions / procedures, recursion
  - Aggregate data structures — arrays, lists, dictionaries
- Express such ideas in the programming language
  - Translate "high level" programming language to "low level" machine language
  - Compilers, interpreters

Abstraction

- Abstractions used in computational thinking
  - Assigning values to named variables
  - Conditional execution
  - Iteration
  - Functions / procedures, recursion
  - Aggregate data structures — arrays, lists, dictionaries
- Express such ideas in the programming language
  - Translate “high level” programming language to “low level” machine language
  - Compilers, interpreters
- Trade off expressiveness for efficiency
  - Less control over how code is mapped to the architecture
  - But fewer errors due to mismatch between intent and implementation

Styles of programming

- Imperative vs declarative

Styles of programming

- Imperative vs declarative
- Imperative
  - How to compute
  - Step by step instructions on what is to be done

Styles of programming

- Imperative vs declarative
- Imperative
  - How to compute
  - Step by step instructions on what is to be done
- Declarative
  - What the computation should produce
  - Often exploit inductive structure, express in terms of smaller computations
  - Typically avoid using intermediate variables
  - Combination of small transformations — functional programming

Imperative vs Declarative Programming, by example

- Add values in a list

Imperative vs Declarative Programming, by example

- Add values in a list
- Imperative (in Python)

```
def sumlist(l):
  mysum = 0
  for x in l:
    mysum = mysum + x
  return(mysum)
```

Imperative vs Declarative Programming, by example

- Add values in a list

- Imperative (in Python)

```
def sumlist(l):
  mysum = 0
  for x in l:
    mysum = mysum + x
  return(mysum)
```

- Declarative (in Python)

```
def sumlist(l):
  if l == []:
    return(0)
  else:
    return(l[0] + sumlist(l[1:]))
```

Imperative vs Declarative Programming, by example

- Add values in a list
- Imperative (in Python)

```
def sumlist(l):
  mysum = 0
  for x in l:
    mysum = mysum + x
  return(mysum)
```

- Intermediate values `mysum`, `x`

- Declarative (in Python)

```
def sumlist(l):
  if l == []:
    return(0)
  else:
    return(l[0] + sumlist(l[1:]))
```

Imperative vs Declarative Programming, by example

- Add values in a list
- Imperative (in Python)

```
def sumlist(l):
  mysum = 0
  for x in l:
    mysum = mysum + x
  return(mysum)
```

- Intermediate values `mysum`, `x`
- Explicit iteration to examine each element of the list

- Declarative (in Python)

```
def sumlist(l):
  if l == []:
    return(0)
  else:
    return(l[0] + sumlist(l[1:]))
```

Imperative vs Declarative Programming, by example

- Add values in a list
- Imperative (in Python)

```
def sumlist(l):
  mysum = 0
  for x in l:
    mysum = mysum + x
  return(mysum)
```

- Intermediate values `mysum`, `x`
- Explicit iteration to examine each element of the list

- Declarative (in Python)

```
def sumlist(l):
  if l == []:
    return(0)
  else:
    return(l[0] + sumlist(l[1:]))
```

- Describe the desired output by induction
  - Base case: Empty list has sum `0`
  - Inductive step: Add first element to the sum of the rest of the list

Imperative vs Declarative Programming, by example

- Add values in a list
- Imperative (in Python)

```
def sumlist(l):
  mysum = 0
  for x in l:
    mysum = mysum + x
  return(mysum)
```

- Intermediate values `mysum`, `x`
- Explicit iteration to examine each element of the list

- Declarative (in Python)

```
def sumlist(l):
  if l == []:
    return(0)
  else:
    return(l[0] + sumlist(l[1:]))
```

- Describe the desired output by induction
  - Base case: Empty list has sum `0`
  - Inductive step: Add first element to the sum of the rest of the list
- No intermediate variables

Imperative vs Declarative Programming, by example, . . .

- Sum of squares of even numbers upto `n`

Imperative vs Declarative Programming, by example, . . .

- Sum of squares of even numbers upto n
- Imperative (in Python)

```
def sumsquareeven(n):
  mysum = 0
  for x in range(n+1):
    if x%2 == 0:
      mysum = mysum + x*x
  return(mysum)
```

Imperative vs Declarative Programming, by example, . . .

- Sum of squares of even numbers upto `n`

- Imperative (in Python)

```
def sumsquareeven(n):
  mysum = 0
  for x in range(n+1):
    if x%2 == 0:
      mysum = mysum + x*x
  return(mysum)
```

- Declarative (in Python)

```
def even(x):
  return(x%2 == 0)

def square(x):
  return(x*x)

def sumsquareeven(n):
  return(
    sum(map(square,
            filter(even,
                   range(n+1)))))
```

Imperative vs Declarative Programming, by example, . . .

- Sum of squares of even numbers upto `n`

- Imperative (in Python)

```
def sumsquareeven(n):
  mysum = 0
  for x in range(n+1):
    if x%2 == 0:
      mysum = mysum + x*x
  return(mysum)
```

- Can code functionally in an imperative language!

- Declarative (in Python)

```
def even(x):
  return(x%2 == 0)

def square(x):
  return(x*x)

def sumsquareeven(n):
  return(
    sum(map(square,
            filter(even,
                   range(n+1)))))
```

Imperative vs Declarative Programming, by example, . . .

- Sum of squares of even numbers upto `n`
- Imperative (in Python)

```
def sumsquareeven(n):
  mysum = 0
  for x in range(n+1):
    if x%2 == 0:
      mysum = mysum + x*x
  return(mysum)
```

- Can code functionally in an imperative language!
- Helps identify natural units of (reusable) code

- Declarative (in Python)

```
def even(x):
  return(x%2 == 0)

def square(x):
  return(x*x)

def sumsquareeven(n):
  return(
    sum(map(square,
            filter(even,
                   range(n+1)))))
```

Names, types, values

- Internally, everything is stored a sequence of bits

Names, types, values

- Internally, everything is stored a sequence of bits
- No difference between data and instructions, let alone numbers, characters, booleans
  - For a compiler or interpreter, our code is its data

Names, types, values

- Internally, everything is stored a sequence of bits
- No difference between data and instructions, let alone numbers, characters, booleans
  - For a compiler or interpreter, our code is its data
- We impose a notion of type to create some discipline
  - Intepret bit strings as “high level” concepts
  - Nature and range of allowed values
  - Operations that are permitted on these values

Names, types, values

- Internally, everything is stored a sequence of bits
- No difference between data and instructions, let alone numbers, characters, booleans
  - For a compiler or interpreter, our code is its data
- We impose a notion of type to create some discipline
  - Intepret bit strings as “high level” concepts
  - Nature and range of allowed values
  - Operations that are permitted on these values
- Strict type-checking helps catch bugs early
  - Incorrect expression evaluation — like dimension mismatch in science
  - Incorrect assignment — expression value does not match variable type

Abstract datatypes, object-oriented programming

- Collections are important
    - Arrays, lists, dictionaries

Abstract datatypes, object-oriented programming

- Collections are important
  - Arrays, lists, dictionaries
- Abstract data types
  - Structured collection with fixed interface
  - Stack is a sequence, but only allows `push` and `pop`

Abstract datatypes, object-oriented programming

- Collections are important
  - Arrays, lists, dictionaries
- Abstract data types
  - Structured collection with fixed interface
  - Stack is a sequence, but only allows `push` and `pop`
  - Separate implementation from interface
    - Priority queue allows `insert` and `delete-max`
    - Can implement a priority queue using sorted or unsorted lists, or using a heap

Abstract datatypes, object-oriented programming

- Collections are important
  - Arrays, lists, dictionaries
- Abstract data types
  - Structured collection with fixed interface
  - Stack is a sequence, but only allows `push` and `pop`
  - Separate implementation from interface
    - Priority queue allows `insert` and `delete-max`
    - Can implement a priority queue using sorted or unsorted lists, or using a heap
- Object-oriented programming
  - Focus on data types
  - Functions are invoked through the object rather than passing data to the functions
  - In Python, `mylist.sort()` vs `sorted(mylist)`

What this course is about

- Explore concepts in programming languages
  - Object-oriented programming
  - Exception handling, concurrency, event-driven programming, . . .

What this course is about

- Explore concepts in programming languages
  - Object-oriented programming
  - Exception handling, concurrency, event-driven programming, . . .
- Use Java as the illustrative language
  - Imperative, object-oriented
  - Incorporates almost all features of interest

What this course is about

- Explore concepts in programming languages
    - Object-oriented programming
    - Exception handling, concurrency, event-driven programming, . . .
- Use Java as the illustrative language
    - Imperative, object-oriented
    - Incorporates almost all features of interest
- Discuss design decisions where relevant
    - Every language makes some compromises

What this course is about

- Explore concepts in programming languages
  - Object-oriented programming
  - Exception handling, concurrency, event-driven programming, . . .
- Use Java as the illustrative language
  - Imperative, object-oriented
  - Incorporates almost all features of interest
- Discuss design decisions where relevant
  - Every language makes some compromises
- Understand and appreciate why there is a zoo of programming languages out there

What this course is about

- Explore concepts in programming languages
  - Object-oriented programming
  - Exception handling, concurrency, event-driven programming, . . .
- Use Java as the illustrative language
  - Imperative, object-oriented
  - Incorporates almost all features of interest
- Discuss design decisions where relevant
  - Every language makes some compromises
- Understand and appreciate why there is a zoo of programming languages out there
- . . . and why new ones are still being created