Introduction

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 1, 5 January 2023

What this course is about

- Haskell, Python, C, C++, Java, . . . ,
  Swift, Go, Rust, . . .

What this course is about

- Haskell, Python, C, C++, Java, . . . , Swift, Go, Rust, . . .
- What is common? What are the differences?

What this course is about

- Haskell, Python, C, C++, Java, . . . ,
  Swift, Go, Rust, . . .
- What is common? What are the differences?
- Styles of programming
  - Declarative — what is to be done
  - Imperative — how to do it

What this course is about

- Haskell, Python, C, C++, Java, . . . , Swift, Go, Rust, . . .
- What is common? What are the differences?
- Styles of programming
  - Declarative — what is to be done
  - Imperative — how to do it

Sum a list of numbers

What this course is about

- Haskell, Python, C, C++, Java, . . . , Swift, Go, Rust, . . .
- What is common? What are the differences?
- Styles of programming
  - Declarative — what is to be done
  - Imperative — how to do it

Sum a list of numbers

Declarative

```
sum [] = 0
sum (x:xs) = x + sum xs
```

What this course is about

- Haskell, Python, C, C++, Java, . . . , Swift, Go, Rust, . . .
- What is common? What are the differences?
- Styles of programming
  - Declarative — what is to be done
  - Imperative — how to do it

Sum a list of numbers

Declarative

```
sum [] = 0
sum (x:xs) = x + sum xs
```

Imperative

```
sum = 0
for x in l:
  sum = sum + x
```

Imperative programming

- Variables — types, storage allocation

Imperative programming

- Variables — types, storage allocation
- Control flow

Imperative programming

- Variables — types, storage allocation
- Control flow
- Abstraction
  - Control flow — functions and procedures
  - Data — complex data structures

Imperative programming

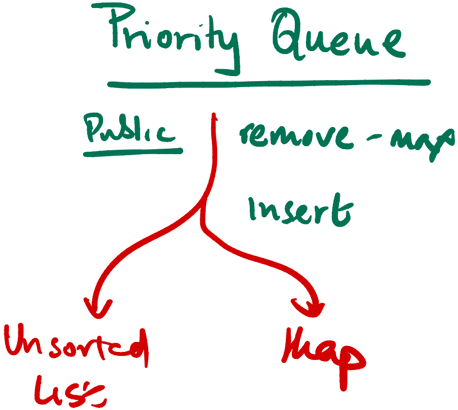
- Variables — types, storage allocation
- Control flow
- Abstraction
  - Control flow — functions and procedures
  - Data — complex data structures

Abstract datatypes

- Variables — types, storage allocation
- Control flow
- Abstraction
  - Control flow — functions and procedures
  - Data — complex data structures

Abstract datatypes

- Public interface
- Private implementation



Course outline

Imperative programming

- Object oriented programming

Course outline

Imperative programming

- Object oriented programming
- Concurrent programming
  - Language support for concurrency

Course outline

Imperative programming

- Object oriented programming
- Concurrent programming
  - Language support for concurrency
- Dealing with errors and exceptions

Course outline

Imperative programming

- Object oriented programming
- Concurrent programming
  - Language support for concurrency
- Dealing with errors and exceptions
- Event driven programming
  - Graphical user interfaces, react to mouse clicks etc

Course outline

Imperative programming

- Object oriented programming
- Concurrent programming
  - Language support for concurrency
- Dealing with errors and exceptions
- Event driven programming
  - Graphical user interfaces, react to mouse clicks etc
- Java as a concrete example language to illustrate concepts

Course outline

Imperative programming

- Object oriented programming
- Concurrent programming
  - Language support for concurrency
- Dealing with errors and exceptions
- Event driven programming
  - Graphical user interfaces, react to mouse clicks etc
- Java as a concrete example language to illustrate concepts

Declarative programming

- Haskell and relatives

Course outline

Imperative programming

- Object oriented programming
- Concurrent programming
  - Language support for concurrency
- Dealing with errors and exceptions
- Event driven programming
  - Graphical user interfaces, react to mouse clicks etc
- Java as a concrete example language to illustrate concepts

Declarative programming

- Haskell and relatives
- Foundations — λ calculus

Alonzo Church

Course outline

Imperative programming

- Object oriented programming
- Concurrent programming
  - Language support for concurrency
- Dealing with errors and exceptions
- Event driven programming
  - Graphical user interfaces, react to mouse clicks etc
- Java as a concrete example language to illustrate concepts

Declarative programming

- Haskell and relatives
- Foundations — λ calculus
- Types and type inference

Abstraction, modularity, object-oriented programming

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 1, 5 January 2023

Stepwise refinement

- Begin with a high level description of the task

```
begin
  print first thousand prime numbers
end
```

Stepwise refinement

- Begin with a high level description of the task
- Refine the task into subtasks

```
begin
  print first thousand prime numbers
end
```

```
begin
  declare table p
  fill table p with first thousand primes
  print table p
end
```

Stepwise refinement

- Begin with a high level description of the task
- Refine the task into subtasks
- Further elaborate each subtask

```
begin
  print first thousand prime numbers
end
```

```
begin
  declare table p
  fill table p with first thousand primes
  print table p
end
```

```
begin
  integer array p[1:1000]
  for k from 1 through 1000
    make p[k] equal to the kth prime number
  for k from 1 through 1000
    print p[k]
```

Stepwise refinement

- Begin with a high level description of the task
- Refine the task into subtasks
- Further elaborate each subtask
- Subtasks can be coded by different people

```
begin
  print first thousand prime numbers
end
```

```
begin
  declare table p
  fill table p with first thousand primes
  print table p
end
```

```
begin
  integer array p[1:1000]
  for k from 1 through 1000
    make p[k] equal to the kth prime number
  for k from 1 through 1000
    print p[k]
```

Stepwise refinement

- Begin with a high level description of the task
- Refine the task into subtasks
- Further elaborate each subtask
- Subtasks can be coded by different people
- Program refinement — focus on code, not much change in data structures

```
begin
  print first thousand prime numbers
end
```

```
begin
  declare table p
  fill table p with first thousand primes
  print table p
end
```

```
begin
  integer array p[1:1000]
  for k from 1 through 1000
    make p[k] equal to the kth prime number
  for k from 1 through 1000
    print p[k]
```

Data refinement

- Banking application
  - Typical functions: `CreateAccount()`, `Deposit()/Withdraw()`, `PrintStatement()`

Balances of all accounts

Data refinement

- Banking application
  - Typical functions: `CreateAccount()`, `Deposit()/Withdraw()`, `PrintStatement()`
- How do we represent each account?
  - Only need the current balance
  - Overall, an array of balances

Data refinement

- Banking application
  - Typical functions: `CreateAccount()`, `Deposit()/Withdraw()`, `PrintStatement()`
- How do we represent each account?
  - Only need the current balance
  - Overall, an array of balances
- Refine `PrintStatement()` to include `PrintTransactions()`
  - Now we need to record transactions for each account
  - Data representation also changes
  - Cascading impact on other functions that operate on accounts

Modular software development

- Use refinement to divide the solution into components

Modular software development

- Use refinement to divide the solution into components
- Build a prototype of each component to validate design

Modular software development

- Use refinement to divide the solution into components
- Build a prototype of each component to validate design
- Components are described in terms of
  - Interfaces — what is visible to other components, typically function calls
  - Specification — behaviour of the component, as visible through interface

Modular software development

- Use refinement to divide the solution into components
- Build a prototype of each component to validate design
- Components are described in terms of
  - Interfaces — what is visible to other components, typically function calls
  - Specification — behaviour of the component, as visible through interface
- Improve each component independently, preserving interface and specification

Modular software development

- Use refinement to divide the solution into components
- Build a prototype of each component to validate design
- Components are described in terms of
  - Interfaces — what is visible to other components, typically function calls
  - Specification — behaviour of the component, as visible through interface
- Improve each component independently, preserving interface and specification
- Simplest example of a component: a function
  - Interfaces — function header, arguments and return type
  - Specification — intended input-output behaviour

Modular software development

- Use refinement to divide the solution into components
- Build a prototype of each component to validate design
- Components are described in terms of
  - Interfaces — what is visible to other components, typically function calls
  - Specification — behaviour of the component, as visible through interface
- Improve each component independently, preserving interface and specification
- Simplest example of a component: a function
  - Interfaces — function header, arguments and return type
  - Specification — intended input-output behaviour
- Main challenge: suitable language to write specifications
  - Balance abstraction and detail, should not be another programming language!
  - Cannot algorithmically check that specification is met (halting problem!)

Programming language support for abstraction

- Control abstraction
  - Functions and procedures
  - Encapsulate a block of code, reuse in different contexts

Programming language support for abstraction

- Control abstraction
  - Functions and procedures
  - Encapsulate a block of code, reuse in different contexts
- Data abstraction
  - Abstract data types (ADTs)
  - Set of values along with operations permitted on them
  - Internal representation should not be accessible
  - Interaction restricted to public interface
    - For example, when a stack is implemented as a list, we should not be able to observe or modify internal elements

Programming language support for abstraction

- Control abstraction
  - Functions and procedures
  - Encapsulate a block of code, reuse in different contexts
- Data abstraction
  - Abstract data types (ADTs)
  - Set of values along with operations permitted on them
  - Internal representation should not be accessible
  - Interaction restricted to public interface
    - For example, when a stack is implemented as a list, we should not be able to observe or modify internal elements
- Object-oriented programming
  - Organize ADTs in a hierarchy
  - Implicit reuse of implementations — subtyping, inheritance

Objects

- An object is like an abstract datatype
  - Hidden data with set of public operations
  - All interaction through operations — messages, methods, member-functions, . . .

Objects

- An object is like an abstract datatype
  - Hidden data with set of public operations
  - All interaction through operations — messages, methods, member-functions, ...
- Uniform way of encapsulating different combinations of data and functionality
  - An object can hold single integer — e.g., a counter
  - An entire filesystem or database could be a single object

Objects

- An object is like an abstract datatype
  - Hidden data with set of public operations
  - All interaction through operations — messages, methods, member-functions, ...
- Uniform way of encapsulating different combinations of data and functionality
  - An object can hold single integer — e.g., a counter
  - An entire filesystem or database could be a single object
- Distinguishing features of object-oriented programming
  - Abstraction
  - Subtyping
  - Dynamic lookup
  - Inheritance

History of object-oriented programming

- Objects first introduced in Simula — simulation language, 1960s

History of object-oriented programming

- Objects first introduced in Simula — simulation language, 1960s
- Event-based simulation follows a basic pattern
  - Maintain a queue of events to be simulated
  - Simulate the event at the head of the queue
  - Add all events it spawns to the queue

```
Q := make-queue(first event)
repeat
  remove next event e from Q
  simulate e
  place all events generated
     by e on Q
until Q is empty
```

History of object-oriented programming

- Objects first introduced in Simula — simulation language, 1960s
- Event-based simulation follows a basic pattern
  - Maintain a queue of events to be simulated
  - Simulate the event at the head of the queue
  - Add all events it spawns to the queue
- Challenges
  - Queue must be well-typed, yet hold all types of events

```
Q := make-queue(first event)
repeat
  remove next event e from Q
  simulate e
  place all events generated
     by e on Q
until Q is empty
```

History of object-oriented programming

- Objects first introduced in Simula — simulation language, 1960s
- Event-based simulation follows a basic pattern
  - Maintain a queue of events to be simulated
  - Simulate the event at the head of the queue
  - Add all events it spawns to the queue
- Challenges
  - Queue must be well-typed, yet hold all types of events
  - Use a generic simulation operation across different types of events
    - Avoid elaborate checking of cases

```
Q := make-queue(first event)
repeat
  remove next event e from Q
  simulate e
  place all events generated
     by e on Q
until Q is empty
```

Abstraction

- Objects are similar to abstract datatypes
  - Public interface
  - Private implementation
  - Changing the implementation should not affect interactions with the object

Abstraction

- Objects are similar to abstract datatypes
  - Public interface
  - Private implementation
  - Changing the implementation should not affect interactions with the object
- Data-centric view of programming
  - Focus on what data we need to maintain and manipulate

Abstraction

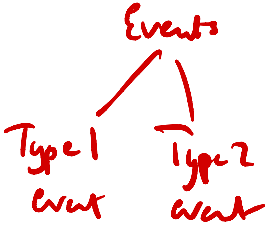
- Objects are similar to abstract datatypes
  - Public interface
  - Private implementation
  - Changing the implementation should not affect interactions with the object
- Data-centric view of programming
  - Focus on what data we need to maintain and manipulate
- Recall that stepwise refinement could affect both code and data
  - Tying methods to data makes this easier to coordinate
  - Refining data representation naturally tied to updating methods that operate on the data

Subtyping

- Recall the Simula event queue
  - A well-typed queue holds values of a fixed type
  - In practice, the queue holds different types of objects
  - How can this be reconciled?

Subtyping

- Recall the Simula event queue
  - A well-typed queue holds values of a fixed type
  - In practice, the queue holds different types of objects
  - How can this be reconciled?



- Arrange types in a hierarchy
  - A subtype is a specialization of a type
  - If `A` is a subtype of `B`, wherever an object of type `B` is needed, an object of type `A` can be used
    - Every object of type `A` is also an object of type `B`
    - Think subset — if $X \subseteq Y$, every $x \in X$ is also in Y

Subtyping

- Recall the Simula event queue
  - A well-typed queue holds values of a fixed type
  - In practice, the queue holds different types of objects
  - How can this be reconciled?
- Arrange types in a hierarchy
  - A subtype is a specialization of a type
  - If `A` is a subtype of `B`, wherever an object of type `B` is needed, an object of type `A` can be used
    - Every object of type `A` is also an object of type `B`
    - Think subset — if $X \subseteq Y$, every $x \in X$ is also in Y
- If `f()` is a method in `B` and `A` is a subtype of `B`, every object of `A` also supports `f()`
  - Implementation of `f()` can be different in `A`

Dynamic lookup

- Whether a method can be invoked on an object is a static property — type-checking

Dynamic lookup

- Whether a method can be invoked on an object is a static property — type-checking
- How the method acts is a dynamic property of how the object is implemented

Dynamic lookup

- Whether a method can be invoked on an object is a static property — type-checking
- How the method acts is a dynamic property of how the object is implemented
  - In the simulation queue, all events support a simulate method
  - The action triggered by the method depends on the type of event

Dynamic lookup

- Whether a method can be invoked on an object is a static property — type-checking
- How the method acts is a dynamic property of how the object is implemented
  - In the simulation queue, all events support a simulate method
  - The action triggered by the method depends on the type of event
  - In a graphics application, different types of objects to be rendered
  - Invoke using the same operation, each object "knows" how to render itself

Dynamic lookup

- Whether a method can be invoked on an object is a static property — type-checking
- How the method acts is a dynamic property of how the object is implemented
  - In the simulation queue, all events support a simulate method
  - The action triggered by the method depends on the type of event
  - In a graphics application, different types of objects to be rendered
  - Invoke using the same operation, each object "knows" how to render itself
- Different from overloading
  - Operation `+` is addition for `int` and `float`
  - Internal implementation is different, but choice is determined by static type

Dynamic lookup

- Whether a method can be invoked on an object is a static property — type-checking
- How the method acts is a dynamic property of how the object is implemented
  - In the simulation queue, all events support a simulate method
  - The action triggered by the method depends on the type of event
  - In a graphics application, different types of objects to be rendered
  - Invoke using the same operation, each object "knows" how to render itself
- Different from overloading
  - Operation `+` is addition for `int` and `float`
  - Internal implementation is different, but choice is determined by static type
- Dynamic lookup
  - A variable `v` of type `B` can refer to an object of subtype `A`
  - Static type of `v` is `B`, but method implementation depends on run-time type `A`

Inheritance

- Re-use of implementations

Inheritance

- Re-use of implementations
- Example: different types of employees
  - `Employee` objects store basic personal data, date of joining

Inheritance

- Re-use of implementations
- Example: different types of employees
  - `Employee` objects store basic personal data, date of joining
  - `Manager` objects can add functionality
    - Retain basic data of `Employee` objects
    - Additional fields and functions: date of promotion, seniority (in current role)

Inheritance

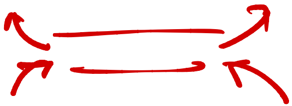
- Re-use of implementations
- Example: different types of employees
  - `Employee` objects store basic personal data, date of joining
  - `Manager` objects can add functionality
    - Retain basic data of `Employee` objects
    - Additional fields and functions: date of promotion, seniority (in current role)
- Usually one hierarchy of types to capture both subtyping and inheritance
  - `A` can inherit from `B` iff `A` is a subtype of `B`

Inheritance

- Re-use of implementations
- Example: different types of employees
  - `Employee` objects store basic personal data, date of joining
  - `Manager` objects can add functionality
    - Retain basic data of `Employee` objects
    - Additional fields and functions: date of promotion, seniority (in current role)
- Usually one hierarchy of types to capture both subtyping and inheritance
  - `A` can inherit from `B` iff `A` is a subtype of `B`
- Philosophically, however the two are different
  - Subtyping is a relationship of interfaces
  - Inheritance is a relationship of implementations

Subtyping vs inheritance

- A deque is a double-ended queue
  - Supports `insert-front()`, `delete-front()`, `insert-rear()` and `delete-rear()`



Subtyping vs inheritance

- A deque is a double-ended queue
  - Supports `insert-front()`, `delete-front()`, `insert-rear()` and `delete-rear()`
- We can implement a stack or a queue using a deque
  - Stack: use only `insert-front()`, `delete-front()`,
  - Queue: use only `insert-rear()`, `delete-front()`,

Subtyping vs inheritance

- A deque is a double-ended queue
  - Supports `insert-front()`, `delete-front()`, `insert-rear()` and `delete-rear()`
- We can implement a stack or a queue using a deque
  - Stack: use only `insert-front()`, `delete-front()`,
  - Queue: use only `insert-rear()`, `delete-front()`,
- `Stack` and `Queue` inherit from `Deque` — reuse implementation

Subtyping vs inheritance

- A deque is a double-ended queue
  - Supports `insert-front()`, `delete-front()`, `insert-rear()` and `delete-rear()`
- We can implement a stack or a queue using a deque
  - Stack: use only `insert-front()`, `delete-front()`,
  - Queue: use only `insert-rear()`, `delete-front()`,
- `Stack` and `Queue` inherit from `Deque` — reuse implementation
- But `Stack` and `Queue` are not subtypes of `Deque`
  - If `v` of type `Deque` points an object of type `Stack`, cannot invoke `insert-rear()`, `delete-rear()`
  - Similarly, no `insert-front()`, `delete-rear()` in `Queue`

Subtyping vs inheritance

- A deque is a double-ended queue
  - Supports `insert-front()`, `delete-front()`, `insert-rear()` and `delete-rear()`
- We can implement a stack or a queue using a deque
  - Stack: use only `insert-front()`, `delete-front()`,
  - Queue: use only `insert-rear()`, `delete-front()`,
- `Stack` and `Queue` inherit from `Deque` — reuse implementation
- But `Stack` and `Queue` are not subtypes of `Deque`
  - If `v` of type `Deque` points an object of type `Stack`, cannot invoke `insert-rear()`, `delete-rear()`
  - Similarly, no `insert-front()`, `delete-rear()` in `Queue`
- Interfaces of `Stack` and `Queue` are not compatible with `Deque`
  - In fact, `Deque` is a subtype of both `Stack` and `Queue`

Summary

- Solving a complex task requires breaking it down into manageable components
  - Top down: refine the task into subtasks; Bottom up: combine simple building blocks
- Modular description of components — interface and specification
  - Build prototype implementation to validate design
  - Reimplement the components independently, preserving interface and specification
- PL support for abstraction
  - Control flow: functions and procedures
  - Data: Abstract data types, object-oriented programming
- Distinguishing features of object-oriented programming
  - Abstraction: Public interface, private implementation, like ADTs
  - Subtyping: Hierarchy of types, compatibility of interfaces
  - Dynamic lookup: Choice of method implementation is determined at run-time
  - Inheritance: Reuse of implementations