Object-oriented programming

Madhavan Mukund

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

Programming Concepts using Java

Week 1

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

- Objects are like abstract datatypes
- Uniform way of encapsulating different combinations of data and functionality
- 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