Intermediate object-oriented programming

Joachim Vandekerckhove

Input validation

Input Validation

- Input validation is the process of checking if input data is valid before using it.
- It helps ensure that your program runs correctly and securely.
- Common types of input validation include range checking, type checking, format checking, and existence checking.

Benefits of Input Validation

- Helps prevent unexpected behavior.
- Improves code readability and maintainability.
- Reduces security risks.
- Provides better user experience by giving informative error messages.

Common Types of Input Validation

- Range checking: Ensures input falls within acceptable range of values.
- Type checking: Ensures input is of the expected type.
- Format checking: Ensures input conforms to a specific format (e.g., phone numbers or email addresses).
- Existence checking: Ensures input is not empty or null.

Basic Input Validation Example

```
classdef BankAccount
    properties
        balance
    end
    methods
        function obj = BankAccount(balance)
            if balance < 0
                error("Balance cannot be negative.")
            end
            obj.balance = balance;
        end
    end
end
```

Handling Invalid Inputs Gracefully

- Provide informative error messages to the user.
- Log error messages for developers to review.
- Raise exceptions with descriptive error messages.
- Ensure that the program does not continue with invalid input.

Input Validation in the Context of OOP

- Input validation can be incorporated into methods and class constructors.
- It's a good practice to perform input validation as close to the source of input as possible.
- Encapsulation allows you to define rules for valid state and prevent invalid input from corrupting the state of the object.

Corruption

Corrupted State

A corrupted state is a situation where an object's data is in an inconsistent or invalid state, often caused by an error in the program or a violation of the object's invariants.

- An object is in a corrupted state when its internal data is inconsistent with the object's intended behavior
- A corrupted state can lead to bugs and errors in the program
- Preventing corrupted states is an important part of writing correct, reliable code

Example: Workflow that leads to a Corrupted State

Consider a bank account object, with a balance attribute and a deposit and withdraw method.

```
classdef BankAccount
  properties
    balance
  end
  methods
    function obj = BankAccount(balance)
      if balance < 0, error("Balance must be positive."), end
      obj.balance = balance;
    end
    function obj = deposit(obj, amount)
      obj.balance = obj.balance + amount;
    end
    function obj = withdraw(obj, amount)
      if amount > obj.balance, error("Overdrawing error!"), end
      obj.balance = obj.balance - amount;
    end
  end
end
```

Example: Workflow that leads to a Corrupted State

Suppose we have the following workflow:

1. Create a new bank account with initial balance of 100 dollars
2. Make a deposit of -50 dollars
3. That's arguably not a deposit
4. This workflow violates the invariants of the bank account object and results in a corrupted state, where the balance is negative.

Example: External Function Bypassing Logger Method

Let's add a "history" attribute and a "logger" method to keep track of all the transactions made on the account:

```
classdef BankAccount
    properties
        balance
        history
    end

    methods
        function obj = BankAccount(balance)
            obj.balance = balance;
            obj.history = {};
        end

        function obj = deposit(obj, amount)
            obj.balance = obj.balance + amount;
            obj.history{end+1} = {'deposit', amount};
        end
```

Example: External Function Bypassing Logger Method

```
        function obj = withdraw(obj, amount)
            if amount > obj.balance
                error('Overdrawing not allowed!');
            end
            obj.balance = obj.balance - amount;
            obj.history{end+1} = {'withdraw', amount};
        end

        function logger(obj)
            for i = 1:length(obj.history)
                fprintf('%s %d\n', ...
                    obj.history{i}{1}, ...
                    obj.history{i}{2});
            end
        end
    end
end
```

Example: External Function Bypassing Logger Method

Now suppose an *external* function edits the balance attribute of the BankAccount object directly, bypassing the deposit and withdraw methods and the logger method:

```
>> account = BankAccount(0);
>> account.balance = 1e9
```

This bypasses the logger method and leads to a corrupted state where the transaction history no longer reflects the actual transactions made on the account.

Private vs. Public

Private vs. Public in OOP

Public methods and attributes are accessible from outside the class

Private methods and attributes are only accessible within the class

```
classdef MyClass
    properties
        public_attr = 0;
    end
    properties (Access = private)
        private_attr = 1;
    end
    methods
        function obj = MyClass(), end
        function public_method(obj), end
    end
    methods (Access = private)
        function private_method(obj), end
    end
end
```

Private vs Public Methods and Attributes

- MATLAB lets you define methods or attributes as private
- Private methods and attributes are available for internal use only, and cannot be accessed from outside the class
- Public methods and attributes can be freely accessed from outside the class

Private Attributes in the BankAccount Class

```
classdef BankAccount
    properties (Access = private)
        balance
    end
    methods
        function obj = BankAccount(initial_balance)
            if initial_balance < 0
                error('Initial balance cannot be negative.'), end
            obj.balance = initial_balance;
        end
        function balance = get_balance(obj)
            balance = obj.balance;
end, end, end
```

- The ‘balance‘ attribute in ‘BankAccount‘ is now private
- It is accessed using the ‘get_balance()‘ method, which is public
- External code can’t modify the ‘balance‘ attribute directly
- This ensures that the balance is always valid and prevents external code from corrupting the object’s state

Public Methods in the BankAccount Class

```
classdef BankAccount
    properties(Access = private)
        balance
    end

    methods
        function obj = BankAccount(initial_balance)
            if initial_balance < 0
                error('Initial balance cannot be <0')
            end
            obj.balance = initial_balance;
        end

        function balance = get_balance(obj)
            balance = obj.balance;
        end
```

Public Methods in the BankAccount Class

```
        function deposit(obj, amount)
            if amount < 0
                error('Deposit amount cannot be <0')
            end
            obj.balance = obj.balance + amount;
        end

        function withdraw(obj, amount)
            if amount < 0
                error('Withdrawal amount cannot be <0')
            end
            if obj.balance < amount
                error('Insufficient funds')
            end
            obj.balance = obj.balance - amount;
        end
    end
end
```

Public Methods in the BankAccount Class

- The ‘deposit()‘ and ‘withdraw()‘ methods in ‘BankAccount‘ are public
- They allow external code to modify the object’s state in a controlled manner
- They perform validation to ensure that the object remains in a valid state
- External code cannot access the ‘balance‘ attribute directly, so it cannot corrupt the object’s state

Operator overloading

Operator Overloading in Object-Oriented Programming

Operator overloading is a technique in object-oriented programming that allows us to define the behavior of operators (+, -, *, /, etc.) for user-defined objects.

In MATLAB, operator overloading is achieved by defining methods (with specific names) in a class that correspond to the operator being used.

For example, the addition operator (+) can be overloaded by defining the ‘plus()‘ method in a class.

Consider the following example:

Operator Overloading in Object-Oriented Programming

Suppose we have a class that represents a 2D vector. We can overload the addition operator (+) to allow adding two vectors.

```
classdef Vector
    properties
        x
        y
    end
    methods
        function obj = Vector(x, y)
            obj.x = x; obj.y = y; end
        function out = plus(v1, v2)
            out = Vector(v1.x + v2.x, v1.y + v2.y); end
    end
end
```

Here, the ‘plus()‘ method takes another ‘Vector‘ object as an input, adds the corresponding components of the two vectors, and returns a new ‘Vector‘ object that represents the sum of the two vectors.

Using Overloaded Operators

Let's create two 'Vector' objects and add them together using the overloaded addition operator (+):

```
>> v1 = Vector(1, 2);
>> v2 = Vector(3, 4);
>> v3 = v1 + v2
```

Output:

```
v3 =
  Vector with properties:
    x: 4
    y: 6
```

Here, we create two 'Vector' objects, 'v1' and 'v2', and add them together using the overloaded addition operator '+'. The resulting 'Vector' object, 'v3', has components '(4, 6)'.

Operator Overloading in Object-Oriented Programming

We can also overload other operators, such as the multiplication operator (.*), to perform scalar multiplication of a vector.

```
classdef Vector
    properties
        x
        y
    end
    methods
        function obj = Vector(x, y)
            obj.x = x; obj.y = y; end
        function out = plus(v1, v2)
            out = Vector(v1.x + v2.x, v1.y + v2.y); end
        function out = times(obj, k)
            out = Vector(obj.x .* k, obj.y .* k); end
end, end
```

By overloading operators, we can make our code more intuitive and expressive, and allow our user-defined objects to behave more like built-in types.

Incomplete List of Overloaded Operators in MATLAB

Operator	Function	Description
`+`	`plus`	Addition
`-`	`minus`	Subtraction
`*`	`mtimes`	Matrix multiplication
`/`	`mrdivide`	Matrix right division
`.*`	`times`	Multiplication
`./`	`rdivide`	Right division
`.^`	`power`	Exponentiation
`==`	`eq`	Equal to
`~=`	`ne`	Not equal to
`>`	`gt`	Greater than
`>=`	`ge`	Greater than or equal to
`<`	`lt`	Less than
`<=`	`le`	Less than or equal to