OBJECT-ORIENTED PROGRAMMING TECHNOLOGY

CONSTRUCTORS

Lecturer： 陈笑沙

TABLE OF CONTENTS

5.1 ABSTRACTION

The abstraction capability of code is a crucial core capability in programming, as it determines the maintainability, extensibility, and reusability of the code.

5.1 ABSTRACTION

Every powerful language provides three mechanisms:

- **Basic expression forms**: Used to represent the simplest individuals that the language is concerned with.
- **Methods of combination**: Through which one can construct compound elements from simpler things.
- **Abstraction methods**: Through which one can name compound objects and treat them as units.

—Structure and Interpretation of Computer Programs

5.1 ABSTRACTION

The philosophical essence: abstraction is logical modeling of reality

- Abstraction is the first principle of computer science, building the essential model of things by selectively ignoring details
- Similar to map drawing: retaining main roads, omitting trees and streetlights
- Examples: use coordinate system to represent city traffic, use nodes and edges to represent social network relationships

5.1 ABSTRACTION

The Three Levels of Technical Implementation

5.1 ABSTRACTION

The Three Levels of Technical Implementation

- Basic Abstraction: Code Encapsulation

5.1 ABSTRACTION

The Three Levels of Technical Implementation

- Basic Abstraction: Code Encapsulation
  - Example: Function Abstraction

5.1 ABSTRACTION

The Three Levels of Technical Implementation

- Basic Abstraction: Code Encapsulation
  - Example: Function Abstraction
- Intermediate Abstraction: Structure Modeling

5.1 ABSTRACTION

The Three Levels of Technical Implementation

- Basic Abstraction: Code Encapsulation
  - Example: Function Abstraction
- Intermediate Abstraction: Structure Modeling
  - Example: Class Abstraction

5.1 ABSTRACTION

The Three Levels of Technical Implementation

- Basic Abstraction: Code Encapsulation
  - Example: Function Abstraction
- Intermediate Abstraction: Structure Modeling
  - Example: Class Abstraction
- Advanced Abstraction: Patterns and Paradigms

5.1 ABSTRACTION

The Three Levels of Technical Implementation

- Basic Abstraction: Code Encapsulation
  - Example: Function Abstraction
- Intermediate Abstraction: Structure Modeling
  - Example: Class Abstraction
- Advanced Abstraction: Patterns and Paradigms
  - Design Patterns

5.1 ABSTRACTION

The Three Levels of Technical Implementation

- Basic Abstraction: Code Encapsulation
  - Example: Function Abstraction
- Intermediate Abstraction: Structure Modeling
  - Example: Class Abstraction
- Advanced Abstraction: Patterns and Paradigms
  - Design Patterns
  - MVC Layered Abstraction

5.1 ABSTRACTION

The Three Levels of Technical Implementation

- Basic Abstraction: Code Encapsulation
  - Example: Function Abstraction
- Intermediate Abstraction: Structure Modeling
  - Example: Class Abstraction
- Advanced Abstraction: Patterns and Paradigms
  - Design Patterns
  - MVC Layered Abstraction
  - Monad Abstraction in Functional Programming

5.1 ABSTRACTION

Abstraction Capability Evolution Path

5.1 ABSTRACTION

Abstraction Capability Evolution Path

- Beginner: Identify duplicate code

5.1 ABSTRACTION

Abstraction Capability Evolution Path

- Beginner: Identify duplicate code
- Advanced: Design domain models

5.1 ABSTRACTION

Abstraction Capability Evolution Path

- Beginner: Identify duplicate code
- Advanced: Design domain models
- Expert: Create DSLs (Domain-Specific Languages)

5.2 CLASSIFICATION

5.2 CLASSIFICATION

- Object-oriented programming design custom data types, which are self-contained, hide data composition and operations, and reflect abstraction in programming design

5.2 CLASSIFICATION

- Object-oriented programming design custom data types, which are self-contained, hide data composition and operations, and reflect abstraction in programming design
- Custom data types are not isolated (we will see inheritance later), but rather systematic hierarchical structures

5.2 CLASSIFICATION

- Object-oriented programming design custom data types, which are self-contained, hide data composition and operations, and reflect abstraction in programming design
- Custom data types are not isolated (we will see inheritance later), but rather systematic hierarchical structures
- Classification is a means of designing and dividing class objects, serving as entities for each category, distinguishing different classes, different data attributes, different scopes, and different operations

5.2 CLASSIFICATION

Example

5.2 CLASSIFICATION

Example

- `Student` class

5.2 CLASSIFICATION

Example

- `Student` class
  - Primary school student

5.2 CLASSIFICATION

Example

- `Student` class
  - Primary school student
  - Middle school student

5.2 CLASSIFICATION

Example

- `Student` class
  - Primary school student
  - Middle school student
  - College student

5.2 CLASSIFICATION

Example

- `Student` class
  - Primary school student
  - Middle school student
  - College student
    - Different majors

5.2 CLASSIFICATION

Example

- `Student` class
  - Primary school student
  - Middle school student
  - College student
    - Different majors
    - Different colleges

5.2 CLASSIFICATION

Example

- `Student` class
  - Primary school student
  - Middle school student
  - College student
    - Different majors
    - Different colleges
    - Different grades

5.3 DESIGN AND EFFICIENCY

5.3 DESIGN AND EFFICIENCY

- Software efficiency is divided into runtime efficiency and development (production) efficiency

5.3 DESIGN AND EFFICIENCY

- Software efficiency is divided into runtime efficiency and development (production) efficiency
- Runtime efficiency includes time consumption and space consumption, which are related to program design strategies, data structures, storage management, and algorithms (code optimization)

5.3 DESIGN AND EFFICIENCY

- Software efficiency is divided into runtime efficiency and development (production) efficiency
- Runtime efficiency includes time consumption and space consumption, which are related to program design strategies, data structures, storage management, and algorithms (code optimization)
- Design efficiency is related to design methods and management mechanisms

5.3 DESIGN AND EFFICIENCY

- Software efficiency is divided into runtime efficiency and development (production) efficiency
- Runtime efficiency includes time consumption and space consumption, which are related to program design strategies, data structures, storage management, and algorithms (code optimization)
- Design efficiency is related to design methods and management mechanisms
- Design methods refer to program design methods, and viewing problems from the perspective of layering and abstraction helps to clarify the data processing logic

5.3 DESIGN AND EFFICIENCY

- Software efficiency is divided into runtime efficiency and development (production) efficiency
- Runtime efficiency includes time consumption and space consumption, which are related to program design strategies, data structures, storage management, and algorithms (code optimization)
- Design efficiency is related to design methods and management mechanisms
- Design methods refer to program design methods, and viewing problems from the perspective of layering and abstraction helps to clarify the data processing logic
- Object-oriented programming design methods can concisely use the language to express layering and abstraction (development efficiency), and the class mechanism of the C++ language can well play the runtime efficiency of the program

5.4 JOSEPHUS PROBLEM REVISITED

5.4.1 PROCEDURAL APPROACH

An approach leaning towards functional programming:

Abstract the problem into a general data structure, then process it

`example/lec05/josephus1`

5.4 JOSEPHUS PROBLEM REVISITED

5.4.2 FUNCTIONAL APPROACH

Consider only the last one:

```
int josephus(int n, int k) {
    if (n == 1)
        return 0;
    int r = josephus(n - 1, k);
    return (k + r) % n;
}
```

5.4 JOSEPHUS PROBLEM REVISITED

5.4.2 FUNCTIONAL APPROACH

Consider only the last one:

```
int josephus(int n, int k) {
    return n == 1 ? 0 : (k + josephus(n - 1, k)) % n;
}
```

5.4 JOSEPHUS PROBLEM REVISITED

5.4.2 FUNCTIONAL APPROACH

5.4 JOSEPHUS PROBLEM REVISITED

5.4.2 FUNCTIONAL APPROACH

$$f(n, k) = (k + f(n - 1, k)) \bmod n$$

5.4 JOSEPHUS PROBLEM REVISITED

5.4.2 FUNCTIONAL APPROACH

$$f(n,k) = (k + f(n-1,k)) \bmod n$$
$$f(1,k) = 0$$

5.4 JOSEPHUS PROBLEM REVISITED

5.4.2 FUNCTIONAL APPROACH

$$f(n,k) = (k + f(n-1,k)) \bmod n$$
$$f(1,k) = 0$$
$$f(2,k) = (k + f(1,k)) \bmod 2 = (k+0) \bmod 2$$

5.4 JOSEPHUS PROBLEM REVISITED

5.4.2 FUNCTIONAL APPROACH

$$f(n,k) = (k + f(n-1,k)) \bmod n$$
$$f(1,k) = 0$$
$$f(2,k) = (k + f(1,k)) \bmod 2 = (k+0) \bmod 2$$
$$f(3,k) = (k + f(2,k)) \bmod 3 = ((k+0) \bmod 2 + k) \bmod 3$$

5.4 JOSEPHUS PROBLEM REVISITED

5.4.2 FUNCTIONAL APPROACH

$$f(n,k) = (k + f(n-1,k)) \bmod n$$
$$f(1,k) = 0$$
$$f(2,k) = (k + f(1,k)) \bmod 2 = (k+0) \bmod 2$$
$$f(3,k) = (k + f(2,k)) \bmod 3 = ((k+0) \bmod 2 + k) \bmod 3$$
$$f(n,k) = (((k+0) \bmod 2 + k) \bmod 3 + k \cdots) \bmod n$$

5.4 JOSEPHUS PROBLEM REVISITED

5.4.2 FUNCTIONAL APPROACH

$$f(n,k) = (((k+0) \bmod 2 + k) \bmod 3 + k \cdots) \bmod n$$

```
int josephus(int n, int k) {
    int r = 0;
    for (int i = 2; i ≤ n; i++) {
        r = (r + k) % i;
    }
    return r + 1;
}
```

5.4 JOSEPHUS PROBLEM REVISITED

5.4.2 FUNCTIONAL APPROACH

Extended reading: Several pure functional approaches (Haskell version)

```haskell
josephus n k =
  let loop xs = let d:r = drop (k-1) xs
                in d : loop (filter (/= d) r)
  in take n (loop (cycle [1..n]))
```

```haskell
josephus 1 k = 0
josephus n k = (k + josephus (n - 1) k) `mod` n
```

```haskell
josephus n k = 1 + foldl go 0 [2..n]
 where
  go r i = (r + k) `mod` i
```

5.4 JOSEPHUS PROBLEM REVISITED

5.4.3 OBJECT-ORIENTED APPROACH

`example/lec04/joseph`

5.4 JOSEPHUS PROBLEM REVISITED

EXTENDED READING: APL LANGUAGE

Uiua Language[1] Version

```
Joseph ← ◌:◌:Ö(⊙:⊂⊙(⊃(⊢|↘1)≻↻)):[]⊙::⧟⋕+1↟⊙(-1)
Joseph 41 2
```

1. https://www.uiua.org

HOMEWORK

Try to use fomular:

$$f(n,k) = ((k \bmod 2 + k) \bmod 3 + k \cdots) \bmod n$$

To write a Joseph Problem Solver class.

- It has two members: n and k.
- It has a constructor to initialize these members.
- It has a method to re-set these members.
- It has a method called **`solve`** to get the answer.