6. Procedural and data abstraction

1. Procedural abstraction and Java methods
2. Local variables, parameters, return values
3. Data abstraction and Java classes
4. Encapsulation

Inquiry

- What is a *thing*?
- What is an procedure?
- How does a software development professional step back from the details of data items and computational steps?
- What is abstraction?
6. Procedural and data abstraction

**Topic objective**

6. Define and test Java methods and classes with object-oriented features

**Reading:** Chs. 5-6

**Subtopic outcomes**

6.1a Describe classes and methods*
6.1b Define a Java method*
6.2a Write method with parameters and return values*
6.2b Debug a method
6.3a Describe Java data abstraction*
6.3b Define a Java class*
6.4a Describe Java encapsulation
6.4b Write and document a class with interface and implementation
6.4c Describe class debugging concepts
6.4d Test and debug a class
1. Procedural abstraction and Java methods

• Did you ever see program code that went on for pages?
• How may the steps of a program be subdivided for modularity?
• What is a subprogram?
• How is println implemented?

Procedural abstraction

• Definition: the creation of subprograms
• Statements and control structures are packaged as named modules
• The statements are replaced by calls to the subprograms
• Procedural abstraction enables modularity and reuse of code
• Defining data types, is called data abstraction, part of object-oriented design (subtopic 6.3)
Modular decomposition

- Some solutions are too complex to be easily understood as a single unit
- A structured design can be decomposed into simpler modules
- This breaking down is called modular decomposition, implemented by procedural abstraction (writing of subprograms)
- We may continue the breakdown as needed by stepwise refinement

Modular decomposition: case study

Separate modules are easier to understand.
6. Procedural and data abstraction

Module hierarchy diagrams

- **Example:** Main invokes Input, Calculate, and Output; Calculate calls Assign and Multiply

```
Main
A. Input
B. Calculate
   1. Assign
   2. Multiply
C. Output
```

- A module hierarchy chart shows module dependencies, whereas a flowchart shows order of execution.

Private method example

```
public class HelloApp
{
    private void hello()
    {
        System.out.println("Hello");
    }

    public static void main()
    {
        hello();
    }
}
```
Java methods

- In Java, subprograms are called “methods”
- These are not general ways of doing things, but specific sequences of commands
- All Java methods are members of classes
- Every program defines at least one class with a special method, main, that executes when the program executes
- Other methods may be defined and may be called from main

Writing and calling Java methods

Example: Drawing a rectangle

```java
public static void main()
{   
    horizontal();  
    vertical();  
    vertical();  
    horizontal();
}

private void horizontal()
{   
    out.println(“*******”);  
}

private void vertical()
{   
    out.println(“*-----*”);  
}
```
Methods in the Java language

- A method call statement invokes the method and may pass parameters to it in parentheses.
- Method calls follow an object or class name, and a dot.
- A method definition spells out the method’s executable code and declares local variables.
- A method definition has:
  - a header (access specifier; type; method ID; parameters in parentheses) and
  - a body (block or compound statement).

Guidelines for writing subprograms

- A method has a single purpose.
- Its purpose is documented in a comment at the top.
- Code longer than a page is usually broken down into method definitions.
- Experienced programmers avoid side effects on variables declared outside the method.
Stubs test a top-down design

A stub simply reports that it has been called

```java
public void main()
{
    char option;
    do {
        out.print("1 Add\n 2 Sub\n 3 Quit");
        option=in.nextChar();
        switch (option) {
            case '1': add(); break;
            case '2': subtract(); break;
        }
    } while (option != '3');
} // [stub.cpp]
```

**Stub method definitions**

```java
public static void add()
{
    out.print("Calling ‘add’");
}
```

```java
public static void subtract()
{
    out.print("Calling ‘subtract’");
}
```

- Stubs are called by driver programs while being tested
Subtopic outcomes

6.1a Explain classes and methods*
6.1b Define a Java method*

2. Local variables, parameters, and return values

- What rules govern variables declared in methods?
- How do methods communicate?
- Can a method invoke itself?
A method’s ways to communicate

- A *parameter* is a value passed to a method by the method’s call
- A *return value* is passed from a method to the statement that calls it
- A *local* variable declared inside a method is inaccessible from outside

Local variables

```java
int quantity = 2;  // Class member (instance variable): accessible to all methods in class

void add()
{
    int sum = 2 * quantity;  // Local variable: accessible only in this method
    out.println("sum = " + sum);
}
```
Local variables and scope of access

- A variable declared within a compound statement is usable only there.
- When a method is called, an activation record for the call, containing local variables, is placed on top of the stack.
- When the method terminates, the stack is popped and local variables are deallocated.
- Two activation records:

```
public static void main()
{
    out.print("2 + 5 = ");
    display_sum(2,5);
}

private static void display_sum(int a, int b)
{
    out.print(a + b);
}
```

Output: 

```
2 + 5 = 7
```
Parameters

- A way for the calling method to pass data of any types and quantity to the called method
- Value of actual parameter in method call is copied to the formal parameter declared in called method’s definition
- Parameter is local; is deallocated when method terminates
- Method definition must specify parameter names and types

Parameters act like local variables

```java
Private static void display_sum(int a, int b) // Displays (a + b)
{
    out.print(a + “+” + b + “=“);
    while (b-- > 0)
        a++;
    out.print(a);
}
public static void main()
{
    int a= in.nextInt(), b= in.nextInt();
    out.print(“Enter two numbers”);
    display_sum(a, b);
    display_sum(6, 3);
}
```
Method return values: example

```java
public void main()
{
    int age = input_age();
    out.print("You are " + age + " years old");
}

private static int input_age()
{
    out.print("Your age? ");
    return in.nextInt();
}
```

A return statement passes a value back to the calling method

- The return keyword precedes the returned value in the called method
- The return value’s type must be type compatible with the method’s type, declared in header
- The return statement terminates the method call
- The returned value goes on the stack for retrieval by the calling method
- The method call is an expression whose value is the value returned
6. Methods and classes

**sum with return value**

```java
public void main()
{
    out.print("Enter past and current: ");
    int past_due=in.nextInt(),
        current=in.nextInt();
    out.print("You owe "
            + sum(past_due, current));
}

private int sum(int a, int b)
{
    return a + b;
}
```

**Parameter and return-value types**

*Some examples:*

```java
private float sum(float a, float b)
{
    return (a + b);
}

private boolean is_even(int n)
{
    return (n % 2 == 0);
}

private char nth_char(String s, int n)
{
    return s.charAt(n);
}
```
A divide-and-conquer strategy to solve problem of walking $n$ steps

**Walk**(steps)

If steps > 0
   Take a step
   Walk(steps – 1)

- This algorithm is *recursive* because it uses itself

Recursion implements a loop

```java
int input_age()
// Prompts for, returns age,
// repeats until gets valid input.
{
    out.print("Age? ");
    int age = in.nextInt();
    if (age >= 0) return age;
    else return input_age();
}
```

- A method that calls itself is recursive
- A base case (as, age >= 0) triggers a simple result; a recursive case triggers a recursive call
- Base case enables eventual termination
A recursive method to add

\[
\text{sum}(a, b) = \begin{cases} 
  b & \text{if } a = 0 \\
  \text{sum}(a-1, b+1) & \text{otherwise}
\end{cases}
\]

private int sum(int a, int b)  
// Returns a + b. Recursive.  
{  
  if (a == 0)  
    return b;  
  else  
    return sum(a-1, b+1);  
}

How recursion uses the stack

void backwards();
public void main()  
{  
  backwards();  
}  

private void backwards()  
{  
  char ch = in.nextChar();  
  if (ch != '\n')  
    backwards();  
  out.print(ch);  
}
Suggestions for writing methods

• A method has a single purpose
• Its purpose is documented in a comment at the top
• Code longer than a page may be broken down
• Experienced programmers build clean interfaces between methods
• Methods may be tested by driver programs

Subtopic outcome

6.2a Write method with parameters and return values*
6.2b Debug a method
3. Data abstraction and Java classes

- What’s an object?
- How are related data items associated in Java?
- How are classes implemented in Java?
- Is database technology relevant to object-oriented programming?

### UML class diagrams

**Class name**: Employee

**Attributes**
- name
- ID
- salary

**Methods**
- display()
- input()
- calc-paycheck()
Why program with classes?

- We may model the *state* and *behavior* of what we want to represent: e.g., persons, events, collections, displayed objects
- Classes let us model the *interactions* found in the environments we work with
- Class libraries may be *reused* conveniently
- Bookseller examples: *books*, *customers*, *transactions* (*challenge*: define classes for them)

Data abstraction

- *Defn*: the creation of new data types
- Some data types, whose instances have components, are called *compound*
- *Objects* are defined by their *attributes* and *operations*
- *Objects* are *instances of classes*
- Encapsulation separates classes’ *interfaces* from their *implementation*
Enumerated types

- Keyword `enum` enables definition of integer types whose values are specified by constant identifiers
- Example:

```java
public enum Seasons
    {Spring, Summer, Fall, Winter};
Seasons this_season = Seasons.Winter;
```

Object-oriented design

- Any concept is a candidate for a class: persons, things, places, transactions
- Relationships among classes include
  - `containment` (an address object is part of a customer object)
  - `inheritance` (scrollers and dialogs are two kinds of views)
- A class implements an abstraction; it may be instantiated by one or more objects
Objects and classes

An object is a compound data item whose attributes (data members or instance fields) may be of any types chosen by the programmer

- Example:

```java
public class locations {
    public int x, y;
}
```

- Usage:

```java
locations loc;
loc.x = 5;
loc.y = 10;
```

Classes

- Programmer may define a class identifier and use it to declare instances (objects)
- Member data items (also called “instance variables”) exist in memory only when we declare instances of a class
- We use object name, dot, and member name to refer to a class member:

```java
out.print(loc.x);
```
Classes associate values

- An object implements the database concept of record or tuple

- Example:
  ```java
  public enum Team { RedSox, Yankees }
  public class Game
  {
    public Team home_team, visitor_team;
    public int home_score, visitor_score;
  }
  ``

- Game associates teams with each other and with integers (scores)

- Instance: `Game g = new Game();`

Using a class without methods

```java
public class Employee {
    public String name;
    public int hours;
};

public static void main()
{
    Employee emp = new Employee();
    emp.name = "Dale";
    emp.hours = 35;
    out.print(emp.name + " worked " + emp.hours + " hours.");
}
```

Output: Dale worked 35 hours.
Classes normally have methods

Class: a compound type defined by data attributes and operations

```java
public class employees {
    String name;
    int hours;
    public void display()
    { out.print(name + " " + hours); }
}
```

Object: an instance of a class

```java
employees emp1, emp2;
```

---

A class with methods

```java
public class Employee {
    String name = new String;
    int hours;
    void input() {
        out.print("Enter name, hrs: ");
        name = in.next(); hours = in.nextInt();
    }
    void display() {
        out.print(name + " worked " + hours + " hrs");
    }
}
```

```java
public static void main() {
    Employee emp = new Employee();
    emp.input();
    emp.display();
}
```
Methods and objects

- A call to a method names an instance of the class, using a dot
  ```java
  Employee emp = new Employee();
  emp.display();
  ```

- Methods of a class have access to the members of that class’s instances
  ```java
  public void display()
  {
      out.print(name + hours);
  }
  ```

- A method that returns an object name returns a reference to the object, not a copy

Comparing objects

- `==` returns true iff the two operands refer to the same object – not necessarily whenever the two objects have the same attribute values

- Strings should be compared using `equals()` or `compareTo()` methods of the `String` class, not `==`, `>`, `<

- Correct example, given objects `x`, `y`:
  ```java
  if (x.equals(y))...
  ```
Implicit and explicit parameters

- Every method call has an implicit parameter: the object that calls the method, e.g., `System.out` in `System.out.println("Hello");`
- Explicit parameters are those listed in parentheses.
- The identifier `this` in a method implementation is a reference to the implicit-parameter object.

Subtopic outcomes

6.3a Describe Java data abstraction*
6.3b Define a Java class*
4. Encapsulation

• How transparent are objects?
• What is hidden in a class? Why?
• What is the process of creating objects in Java?
• What is a reference?

Interface and implementation

• Interface: public method declarations
• Implementation: private members and definitions of member methods
• A programmer using a class needs to know only its interface
• A programmer writing or maintaining a class must understand its implementation
• Interface consists of methods listed with access specifier public; members listed as private are in implementation
Encapsulation hides data

```java
public class Employee {
    public void input();
    public void display();
    private String name;
    private int hours;
    Employee emp = new Employee();
    emp.hours = 40;
    emp.input();
}
```

Public members are accessible from outside a class, private members are hidden

Cohesion and coupling

- A guideline of software development practice is to maintain strong cohesion in a single class and weak coupling among different classes
- **Cohesion:** All attributes and methods are closely related to the concept implemented by the class
- **Coupling:** Dependencies among different classes. A class depends on another if it uses instances of the other class. Two valid dependencies are containment and inheritance
Accessors and mutators

- An *accessor* is a method that does not modify data members
- Accessors are used to provide member data values, or values computed from them, to calling statements
- A *mutator* is a method that does modify member data
- In the *employees* class, *get_hours* is an accessor; *set_name* is a mutator

Constructors initialize members

```java
public class Employee {
    public Employee() {
        name = new String();
        hours = 0;
    }
    public Employee(String nm, int hrs) {
        name = new String(nm);
        hours = hrs;
    }
    public void input();
    public void display();
    private String name;
    private int hours;
};
```

- default constructor
- constructor with parameters
Constructors

- Take name of class; initialize data members
- Are called with *new* when instance is declared
- Have no return value or type
- May take parameters
- May be *overloaded*; i.e., there may be one constructor for each set of parameters the programmer desires to be able to initialize instances with

Methods that return objects

- To return an object, use *clone*
- *Example:*
  
  *Incorrect:*
  ```java
  public class Customer
  {
      public Address get_address()
      { return address; }
  }
  ```
  
  *Correct:*
  ```java
  public Address get_address()
  { return (Address)address.clone(); }
  ```
Subtopic outcome

6.4a Describe Java encapsulation
6.4b Write and document a class with interface and implementation
6.4c Describe class debugging concepts
6.4d Test and debug a class

References

_____ . Defining a class. Classroom handout.