

**Topic 6:
Procedural abstraction
and data abstraction
(methods and classes)**

1. Modular decomposition
2. Java methods (subprograms)
3. Variables, parameters, and return values
4. Data abstraction and classes
5. Interface and implementation

David Keil Computer Science I 6. Methods and classes 6/09 1

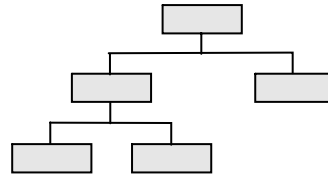
Objectives

16. Define a Java method with parameters and a return value
17. Define a Java class

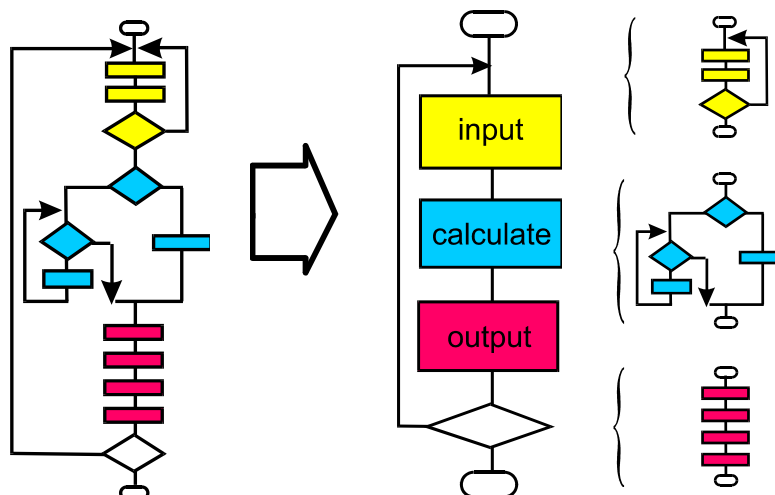
David Keil Computer Science I 6. Methods and classes 6/09 2

1. 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



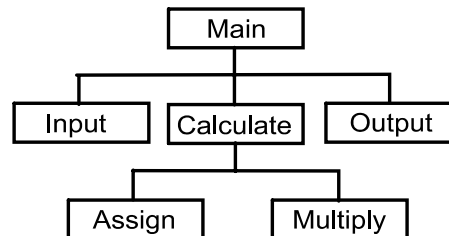
Separate modules are easier to understand.

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.

Procedural abstraction

- *Procedural abstraction* is the definition of *named subprograms* in structured design
- Statements and control structures are packaged as subprograms and the statements are replaced by calls to the subprograms
- Procedural abstraction enables modularity and reuse of code
- Defining data types, including classes, is called data abstraction, part of object-oriented design

Guidelines for writing subprograms

- A module has a single purpose
- Its purpose is documented in a comment at the top
- Code longer than a page usually should be broken down
- Experienced programmers avoid side effects (e.g., output, modification of global variables)

2. Java methods (subprograms)

- 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*

Private method example

```

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

```

Class name, part of class definition

method definitions

method call

Writing and calling Java methods

Example: Drawing a rectangle

```

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

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

private void vertical()
{ out.println("*-----*"); }

```

method calls

method definitions

Methods in the Java language

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

3. Variables, parameters, and return values

- A *variable* declared in a method is accessible only within the method that declares it
- 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

Local variables

```

int quantity = 2;
void add()
{
    int sum = 2 * quantity;
    out.println("sum = " + sum);
}
public static void main()
{
    add();
}

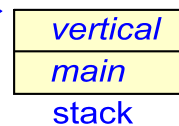
```

Class member (instance variable): accessible to all methods in class

Local variable: accessible only in this method

Local variables and scope of access

- A variable declared within a compound statement is visible 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: top →



Parameters: example

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

Actual parameters

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

Formal parameters

Output:
2 + 5 = 7

Parameters

- A way for the calling method to pass data to the called method
- Value of *actual parameter* in method call is copied to the *formal parameter* declared in called method's definition
- Formal parameter is local, is deallocated when method terminates
- Prototype must specify parameter types; definition must specify parameter names and types
- Parameters may be of any types and quantity

Parameters act like local variables

```

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

```

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

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

```

*The value of the method call is the value used in the **return** statement in the method definition.*

A return value passes data back to the calling method

- The method call is an expression
- 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

sum with return value

```
int sum(int a, int b);

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

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

Parameter and return-value types

Some examples:

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

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

char nth_char(String s, int n)
{
    return s.charAt(n);
}
```

Stubs test a top-down design

```
public static void add()
{ out.print("Calling 'add'"); }
public static void subtract()
{ out.print("Calling 'subtract'"); }
public void main()
{
    char option;
    do {
        out.print("1 Add\n 2 Subtract\n 3 Quit");
        option=in.nextChar();
        switch (option) {
            case '1': add(); break;
            case '2': subtract(); break;
        }
    } while (option != '3');
}
```

Stub method definitions

Calls to stub methods

[stub.cpp]

Recursion implements a loop

```
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();
}
```

Recursive method call

- A method that calls itself is recursive
- A *base case* (as, $age \geq 0$) triggers a simple result; a recursive case triggers a recursive call
- No base case means infinite recursion

A recursive method to add

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

```
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() [backward.cpp]  
{  
    backwards();  
}  
private void backwards()  
{  
    char ch=in.nextChar();  
    if (ch != '\n')  
        backwards();  
    out.print(ch);  
}
```

Sample I/O:
Hello
olleH

Questions:
How many *char*
variables can
store how much
data? How?

Writing methods

- A method has a single purpose
- Its purpose is documented in a comment at the top
- Code longer than a page usually should be broken down
- Experienced programmers avoid side effects (e.g., output, modification of global variables)
- Methods may be tested by *driver* programs

Discussion problems

1. Write a method that returns the absolute value of the difference between two integers.
2. Convert pseudocode in Topic-5 slide, “Invariants assure correctness”, to a method that prompts for 5 numbers and displays them.
3. Convert pseudocode in Topic-5 slide, “Convergence assures termination,” to a method that returns the floor of the base-2 logarithm of a number.

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

4. Data abstraction and classes

- *Data abstraction* is the defining 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 *implementations*

Enumerated types

- Keyword *enum* enables definition of integer types whose values are specified by constant identifiers

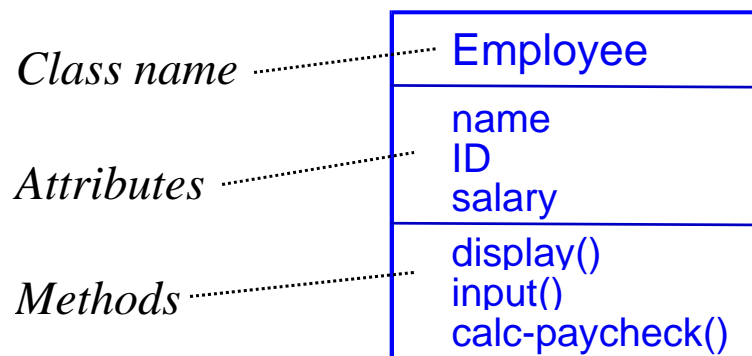
- *Example:*

```
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

UML class diagrams



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:

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

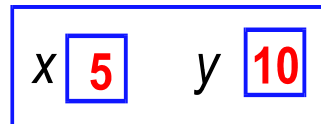
Class (data type)

- Usage:

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

Object (instance of class)

loc



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:

```
out.print(loc.x);
```

Classes associate values

- An object implements the database concept of *record* or *tuple*
- *Example:*

```
public enum Team { RedSox, Yankees };  
public class Game  
{  
    public Team home_team, visitor_team;  
    public int home_score, visitor_score;  
};
```
- The example associates elements of a set of teams with each other and with elements of the set of integers (scores)
- Instance: `Game g = new Game();`

Using a class without methods

```
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

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

Object: an instance of a class

```
employees emp1, emp2;
```

A class with methods

```
public class Employee
{
    char name[40];
    int hours;
    void input() {
        out.print("Enter name, hrs: ");
        name = in.next(); hours = in.nextInt();
    }
    void display()
    { out.print(name + " worked " + hours + " hrs"); }
}

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

Class (points to `Employee`)

Methods (points to `input()` and `display()`)

Instance of class (points to `Employee emp = new Employee();`)

Methods and objects

- A call to a method must name an object, using dot

```
Employee emp = new Employee();  
emp.display();
```

- Methods of a class have access to the members of that class's instances

```
public void display()  
{  
    out.print(name + hours);  
}
```

Methods that return objects

- If a method's return value is an object reference, encapsulation can be broken unless *clone* is used

- *Example:*

Incorrect:

```
public class Customer  
{  
    public Address get_address()  
        { return address; }  
}
```

Correct:

```
public Address get_address()  
{ return (Address)address.clone(); }
```

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

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*:
`if (x.equals(y))...`

5. 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 can hide members

```

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

```

hidden (points to `private String name;` and `private int hours;`)
valid (points to the class body `{ ... }`)
invalid (points to `emp.hours = 40;`)

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

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

```
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
- Must be called when object 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

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)

References

Cay Horstmann. *Big Java*, 3rd ed. Wiley, 2008, Ch. 3.

D. Keil. Defining and using methods. Classroom handout.

_____. Defining a class. Classroom handout.