


Department of Computer Science

# Managing Complexity

Dr. Chia-Ling Tsai

The slide features a decorative background of overlapping squares in shades of blue and purple on the left side.



## Outline

- Algorithm design
- Methods/Procedures
- Parameters
- Return values
- Scope

The slide has a decorative header bar with a gradient from dark blue to light grey and a small square graphic on the left.

## Algorithm Design

- Two reasons for having *procedures*
  - Controlling complexity
  - Code reuse—sometimes an entire section of detail in an algorithm is repeated
- A *structured algorithm* specifies a procedure by:
  - *Defining* the procedure *only once*, and assigning a *name* to that procedure
  - *Calling* the procedure by name as many times as it is to be applied

## Example

- Algorithm to draw the letter E
  - Print a horizontal line
  - Print a vertical segment
  - Print a horizontal line
  - Print a vertical segment
  - Print a horizontal line
- Procedures
  - To print a horizontal line
    - Do as many times as desired: print a single star
    - Move to next line
  - To print a vertical segment
    - Do as many times as desired: print a single star on a line by itself

## Methods/Procedures

- Java permits you to write procedures separately from the main program
- The implemented procedures are called *static methods*
  - *Note:* There are also other kinds of methods, used in object-oriented programming; we won't use such methods in this course
- *Call* a static method by using its name *followed by parentheses* (and a semicolon)
- *Define* a static method using the words **static**, the name of the method, and *parentheses*

```

// A program that draws the letter E using static methods
public class DrawEStructured
{
    // The program
    public static void main(String[] args)
    {
        printHorizontalLine ();
        printVerticalSegment ();
        printHorizontalLine ();
        printVerticalSegment ();
        printHorizontalLine ();
    }
}

// A method that draws a horizontal line using stars
static void printHorizontalLine ()
{
    for (int count = 1; count <= 7; count++)
    {
        System.out.print ("*");
    }
    System.out.println ();
}

// A method that draws a vertical segment using stars
static void printVerticalSegment ()
{
    for (int count = 1; count <= 3; count++)
    {
        System.out.println ("*");
    }
}
}

```

Method calls →

Method definitions →

## Another Example

- Constants must be available outside main, so their declarations are placed outside main
- But now we must use the qualifier **static**

```
// A program that draws the letter E using static methods
// Revision: Uses constants to specify sizes
public class DrawEStructuredWithConstants
{
    // Define sizes of drawing elements
    static final int WIDTH = 7;
    static final int SEGMENT_HEIGHT = WIDTH/2;

    // The program
    public static void main(String[] args)
    {
        printHorizontalLine();
        printVerticalSegment();
        printHorizontalLine();
        printVerticalSegment();
        printHorizontalLine();
    }

    // A method that draws a horizontal line using stars
    static void printHorizontalLine()
    {
        for (int count = 1; count <= WIDTH; count++)
        {
            System.out.print("*");
        }
        System.out.println();
    }

    // A method that draws a vertical segment using stars
    static void printVerticalSegment()
    {
        for (int count = 1; count <= SEGMENT_HEIGHT; count++)
        {
            System.out.println("*");
        }
    }
}
```

## Parameters

- The example yet again
- The algorithm
- Arguments
- (Formal) Parameters
- How it works
- Calling vs. defining methods

## The Example Yet Again

- Let's draw one last E, this time making the program as flexible (and useful) as possible
- The size of the letter should be up to the user
- So we should ask the user for the width, then define the height in terms of what is typed in, then draw the letter
- The best way to do this is to *pass the size information on to the procedures*
- First, we'll write this as an algorithm, then get to how to do it in Java

## The Algorithm

- Main algorithm
  - Get width information from user
  - Define segment height as half of width
  - Print a horizontal line of the given width
  - Print a vertical segment of the given segment height
  - Print a horizontal line of the given width
  - Print a vertical segment of the given segment height
  - Print a horizontal line of the given width
- To print a horizontal line having some number of stars
  - Repeat for the number of stars: print a star
  - Go to the next line
- To print a vertical segment having some number of lines
  - Repeat for the number of lines: Print a star on a new line

## Arguments

- An *argument* is a value you *send to* a method when you call that method
  - An argument is also called an *actual parameter*
- Example
  - `System.out.println` is a method whose argument is whatever string you want it to print
- Arguments are listed in parentheses after the name of the method, separated by commas if there are several arguments
- Example
  - `System.out.println("Hello there");`

## (Formal) Parameters

- A method to which you send arguments uses a special kind of *variable*, the *parameter*, or *formal parameter*, of the method
- Parameters are declared in a special part of the heading of the method, in order corresponding to the matching arguments
- Parameters are not initialized in the method – in effect, each time the method is called, a new initialization occurs, using the arguments
- Aside from the special place of declaration and not initializing them, parameters are treated just the same as variables declared in the method

```

import java.util.Scanner;

// A program that draws the letter E
public class DrawEFlexible
{
    public static void main(String[] args)
    {
        // Get width information from user
        Scanner keyboard = new Scanner(System.in);
        System.out.print("How many stars wide? ");
        int width = keyboard.nextInt();
        // Define height
        int segmentHeight = width/2;
        // Draw E
        printHorizontalLine(width);
        printVerticalSegment(segmentHeight);
        printHorizontalLine(width);
        printVerticalSegment(segmentHeight);
        printHorizontalLine(width);
    }

    // Draws a horizontal line the given number of stars wide
    static void printHorizontalLine(int numberOfStars)
    {
        for (int count = 1; count <= numberOfStars; count++)
        {
            System.out.print("*");
        }
        System.out.println();
    }

    // Draws a vertical segment using the given number of lines
    static void printVerticalSegment(int numberOfLines)
    {
        for (int count = 1; count <= numberOfLines; count++)
        {
            System.out.println("*");
        }
    }
}

```

Arguments (Actual Parameters)

(Formal) Parameters

## How It Works

- When a method is called
  - The *values* of the *arguments* are used to *initialize* the (*formal*) *parameters*
  - Another way to think of this: the values of the arguments are *copied into* the parameters (which are variables)
- Once the arguments are copied into the parameters, the method definition is used to execute the method
  - From this point on, the parameters are no different that any other variables of the method

## Multiple Parameter

```
import java.util.Scanner;

// A sample program demonstrating the use of parameters.
// Greets a number of students.
public class Greeter
{
    // The program
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.print("How many students are there? ");
        int numberOfStudents = keyboard.nextInt();
        keyboard.nextLine();
        for (int i = 1; i <= numberOfStudents; i++)
        {
            System.out.print("Name of student to greet: ");
            String nameOfStudent = keyboard.nextLine();
            System.out.print(nameOfStudent + "'s major? ");
            String major = keyboard.nextLine();

            greet(nameOfStudent, major, "Iona");
        }
    }

    // Greets student with given name, major, and school
    private static void greet(String name, String major, String school)
    {
        System.out.println("Hi, " + name + ". Nice to meet you.");
        System.out.println("I hear you're majoring in " + major + " at " + school + ".");
    }
}
```

*Technical note: Sometimes you have to "get off the line" manually*

*Copies made each time*

## Calling Vs. Defining Methods

- When you *call* a method
  - The name of the method is used, along with the arguments
  - There are *no* declarations here – we just *use values from the program*
  - Example
 

```
PrintHorizontalLine(5);
```
- When you *define* a method
  - The name of the method is included in a *heading*, which includes *declarations*
    - The method is declared to be **static**
    - Each parameter is declared with a type
  - Example
 

```
static void PrintHorizontalLine(int numberOfStars)
```

## Pass by Value

- The argument is used only to *initialize* the parameter
- Once the value of the argument is copied into the parameter, there is *no further contact*
  - This is similar to what happens when you copy the value of one variable into another
  - Example
 

```
int x = 3;
int y = 7;
y = x; // y now has value 3, a copy of that of x
x = 9; // y still has value 3
```

## Example

```
// A program that tries -- unsuccessfully -- to use a method to
// change the value of a variable
public class TryChange
{
    // The program
    public static void main(String[] args)
    {
        int x = 12;
        System.out.println("Before calling method, value is " + x);
        change(x);
        System.out.println("After calling method, value is " + x);
    }

    // A method that attempts to change the value of its parameter
    private static void change(int a)
    {
        System.out.println("At beginning of method, value is " + a);
        a = 3*a;
        System.out.println("At end of method, value is " + a);
    }
}
```

```

C:\WINDOWS\system32\cmd.exe
Before calling method, value is 12
At beginning of method, value is 12
At end of method, value is 36
After calling method, value is 12
Press any key to continue . . .

```

## Return Values

- Calling methods that return values
- Defining methods that return values
- Examples from the Java library
- Self-written example
- Defining methods that return values
- Example

## Calling Methods That Return Values

- Although a method cannot change a parameter, it may *give back* a new value as its "answer"
  - Example: a method may give back three times the value of its parameter
- Method call
  - Treat the *call* of the method *as if it were just a value in your program*
    - You can assign it to a variable
    - You can print it

## Examples From The Java Library

- The Java library has some methods that return values
  - The `Math` class has (among many other methods) `sqrt`, which gives back a square root :  

```
double x = 12;  
double y = Math.sqrt(x);
```
  - The `Integer` class has (among many other methods) `toString`, which returns a string that holds the representation of an integer in a given base  

```
int x = 245;  
String bits = Integer.toString(x, 2);
```
- We can also write methods of our own

## Self-written Example

- Calling a method we write is done just the same way as calling one from the Java library, except you don't have to start with a class name
  - Examples
    - ```
int x = 5;  
int y = 3 * x;
```
    - ```
int x = 5;  
int y = triple(x);
```
    - ```
System.out.println(2);
```
    - ```
System.out.println(triple(2));
```

## Defining Methods That Return Values

- In the *heading* of the method, specify the type of value to be returned, instead of using `void`
- In the *body* of the method
  - Declare a variable to hold the value you want to return
  - Do whatever computations you have to do
  - Use the `return` statement to give back the value

## Example

```
// A program that replaces the value of a variable by the result
// of a method applied to that variable
public class Change
{
    // The program
    public static void main(String[] args)
    {
        int x = 12;
        System.out.println("Before calling method, value is " + x);
        x = triple(x);
        System.out.println("After calling method, value is " + x);
    }

    // Method that returns three times the value of its parameter
    private static int triple(int a)
    {
        int result = 3*a;
        return result;
    }
}
```

call →

assigns answer returned by method to x

return type

variable to hold answer

definition

statement that sends result back to caller

## Method Overloading

- The signature of a method:
  - Name of the method
  - Its number and type of parameters

Overloaded methods

```
// A program to demonstrate method overloading
public class Overloading {
    public static void main(String[] args) {
        int width = 8;
        int height = 10;
        drawBox(width, height);
        int scale = 2;
        drawBox(scale);
    }
    static void drawBox(int scale)
    {
        int size = scale * 3;
        drawBox(size, size);
    }
    static void drawBox(int width, int height)
    {
        for (int i = 0; i < width; i++)
            System.out.print("*");
        System.out.println();
        for (int i = 0; i < height - 2; i++) {
            System.out.print("*");
            for (int j = 0; j < width - 2; j++)
                System.out.print(" ");
            System.out.println("*");
        }
        for (int i = 0; i < width; i++)
            System.out.print("*");
        System.out.println();
    }
}
```

## A case study

## Scope

- Concept of scope
- Local variables
- Global variables
- Global constants

## Concept Of Scope

- The *scope* of a variable is the part of the program from its declaration until the end of the *block* in which it is declared
  - The *block* is the text between { and }
  - Special case: the counter variable in a `for` loop is declared *before* the block, but its scope is the block containing the statements of the loop
- A variable has meaning only within its scope
- Variables in scopes that do not overlap may have the same name without conflict
- Java will not allow re-declarations *within* a scope

## Local Variables

- A variable whose scope is within the braces of a method is a *local variable*
  - A variable within a block such as that of a `for` loop also gets a new "life" *each* time the block is executed
- Outside the scope, the variable is unknown
- The variable becomes "alive" when the method is called, and "dies" when the method returns
  - If the method is called twice, you get *two different* lifetimes – the second is unaffected by the first
- Make the variables as local as possible
  - Declare variable in the innermost scope possible

**Example**

```

public class ScopeExample
{
    public static void main(String[] args)
    {
        int x = 1;
        doSomething(3);
        doSomethingElse();
        System.out.println("x at end of main is " + x);
    }

    static void doSomething(int a)
    {
        int x = a + 1;
        for (int i = 1; i <= 10; i++)
        {
            x = x + i;
        }
        System.out.println("x at end of doSomething is " + x);
    }

    static void doSomethingElse ()
    {
        int x = 1;
        x = x + 7;
        doSomething(6);
        System.out.println("x at end of doSomethingElse is " + x);
    }
}

```

Each of these declares a *different variable* (all have the same name) *This is not the recommended approach*

If you don't declare `x` here, Java will give an error when it is used later, in the loop

Java would not allow `x` to be re-declared within this loop, either to name the counter or to name a new variable

Each method works with its own version of `x`

```

x at end of doSomething is 59
x at end of doSomething is 62
x at end of doSomethingElse is 8
x at end of main is 1

```

Note order of execution

## Global Variables

- Sometimes you want a value to be used throughout a program
- You may define such a value *outside of all static methods* by calling the variable holding it `static`
  - Such a variable is known as a *global variable*
- A global variable is *very dangerous* – *don't ever use one*
  - Java does *not* protect you by preventing reuse of the name within methods.
- We can avoid the danger and still get the benefit of sharing information by using *parameters*

## Global Constants

- The biggest problem with global variables is that you might confuse them with local variables
- This is a problem if you *change* the value of a variable – you may be changing the wrong one
- If you want to use a global *constant*, there is no problem
- Java prevents change in the value of a constant because you use the keyword `final`
- Spelling the name using all capitals can be thought of as a way to emphasize that we are using a value globally

## Another case study – Amortization Calculator

$$MP = \frac{P * R}{1 - (1 + R)^{-N}}$$

- MP=monthly payment
- P=principal
- R=monthly interest rate
- N=total number of payment

## Amortization Schedule

- Example:
  - P=\$100,000
  - Annual interest rate (APR) = 8%
  - Duration of payment = 6 years
  - MP = \$733.76

Month	P (Principal)	I (Interest)	P + I	Total P paid	Total I paid	P balance
0	n/a	n/a	n/a	n/a	n/a	100,000.00
1	67.10	666.67	733.76	67.10	666.67	99,932.90
2	67.55	666.22	733.76	134.64	1,332.89	99,865.36
...	...	...	...	...	...	...
359	724.08	9.69	733.76	99,271.09	164,150.39	728.91
360	728.91	4.86	733.76	100,000.00	164,155.25	0.00

## Summary

- Decomposition of problem
  - Procedural programming
- Procedures
- Parameter passing
- Scope