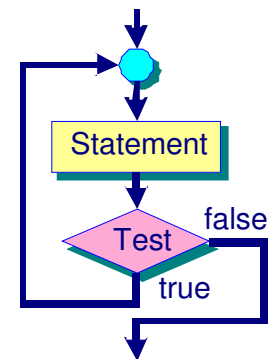
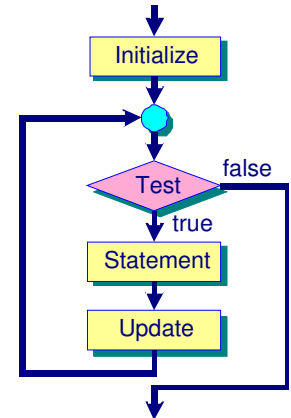
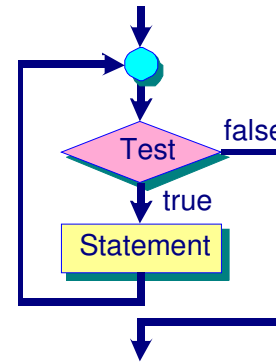


Chapter 8

Methods

Review: Looping

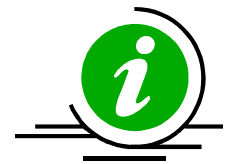
- Types of Loops
- The while Statement
- The for Statement
- The do while Statement
- The break Statement
 - causes immediate termination of the innermost enclosing loop or the switch statement
- The continue statement
 - control immediately passed to the test condition of the nearest enclosing loop (or the update step in case of for loop), and all subsequent statements after the continue statement are not executed for this particular iteration
- Nested Loops



Where are we?

- Computer Systems & Java Programming
- Java Program Development
- Data and Operators
- Console Input/Output
- Branching
- Looping
- **Methods**

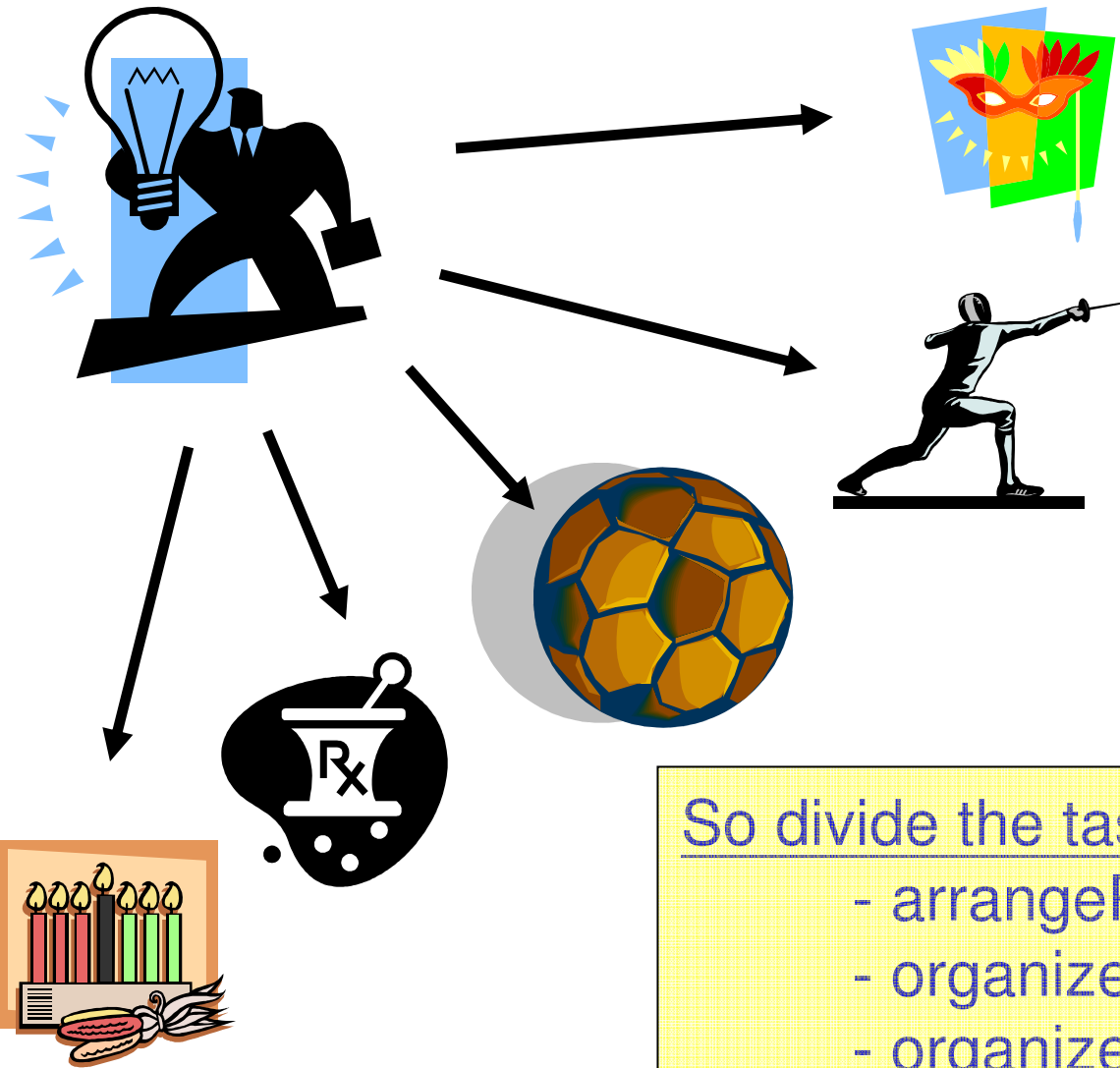
- Arrays
- Classes & Objects
- Strings & Characters
- Class Inheritance (Optional & Non-Examinable)
- Exception Handling
- File Input/Output



Methods

- **Why Methods?**
- Method Definition
- Calling a Method
- Passing Parameters and Values
- Overloading Methods
- Scope of Variables
- The Math Class
- Wrapper Classes
- Enumerated Types
- Designing Programs with Methods
- Case Study

Why Methods: Organizing a Birthday Party



We need to

- Arrange the place
- Organize food
- Organize drink
- Organize activities
- Prepare birthday cake
-

Not easy to be done by a single person!!!

So divide the tasks and do

- arrangePlace() → by A
- organizeFood() → by B
- organizeDrink() → by C

Modular Programming

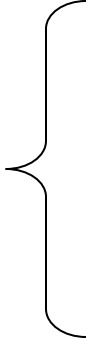
In real-life, there are many other people (e.g. lecturer, boss, colleague, etc.) need to look at or review your programs.

We need to ensure that our programs are

- Easy to read and understand (**readable**)
- Easy to be modified by others (**maintainable**)

Some suggested ways:

- Choosing meaningful variable names
- Adhering to standard naming convention
- Writing meaningful comments
- Indenting program code



method
function
subroutine
procedure

Modular programming - write your programs in a **modular manner** using methods.

Using Methods

```
public class OrganizeParty {  
    public static void main(String[] args) {
```

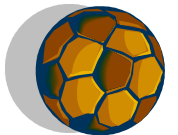
```
        arrangePlace();  
        organizeFood();  
        organizeDrink();  
        organizeActivities();  
        ...
```

```
    }
```

```
    public static void method arrangePlace() {...}  
    public static void method organizeFood() {...}  
    public static void method organizeDrink() {...}  
    public static void method organzeActivities() {...}
```

```
}
```

USE METHODS



Methods

- Why Methods?
- **Method Definition**
- Calling a Method
- Passing Parameters and Values
- Overloading Methods
- Scope of Variables
- The Math Class
- Wrapper Classes
- Enumerated Types
- Designing Programs with Methods
- Case Study

Methods

A **method** is a **self-contained unit of code** to carry out a **specific task**. A method definition is a piece of code which

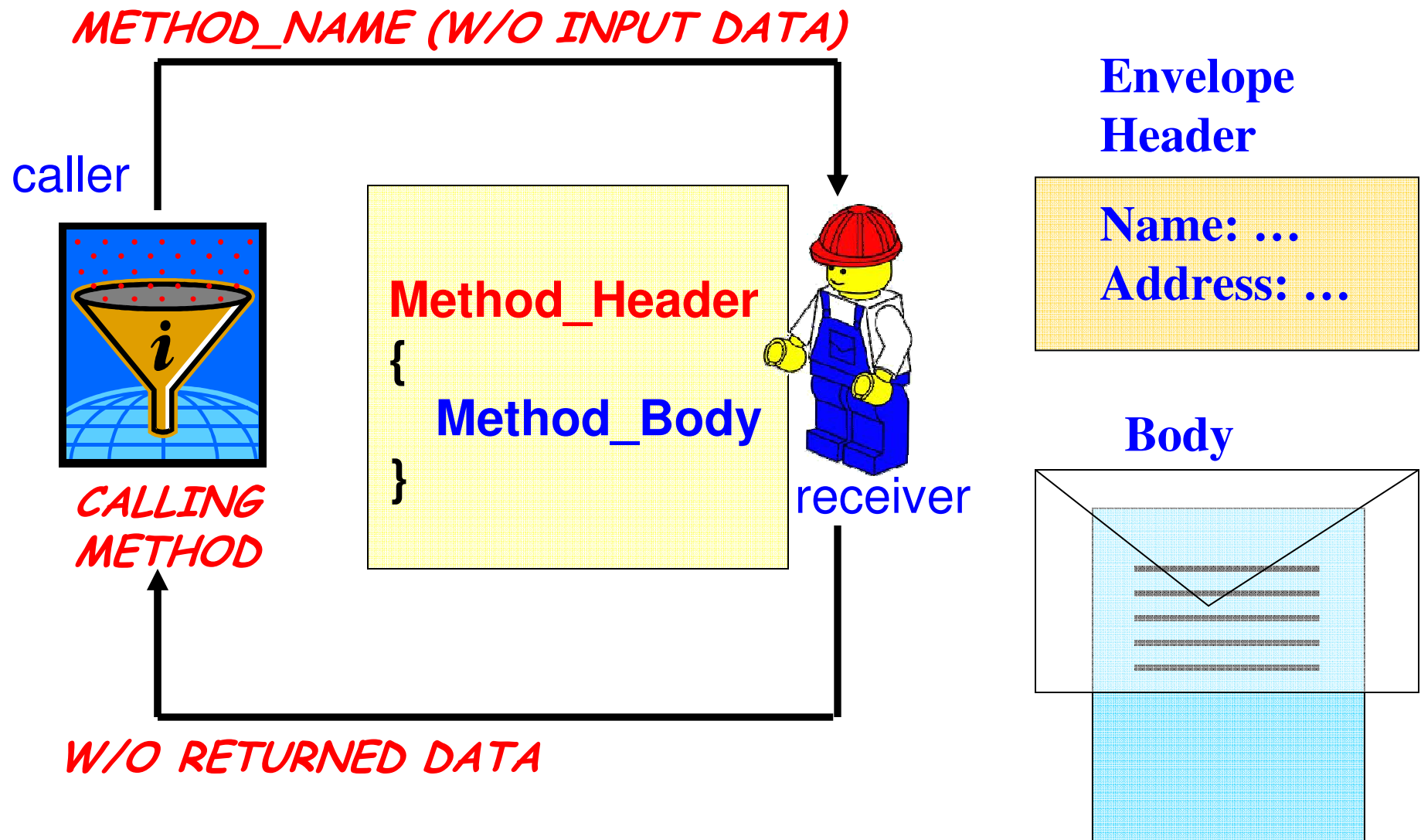
- (1) specifies the **action(s)** of the method and
- (2) specifies the **local data** used by the method.

The **main()** method definition has the following structure:

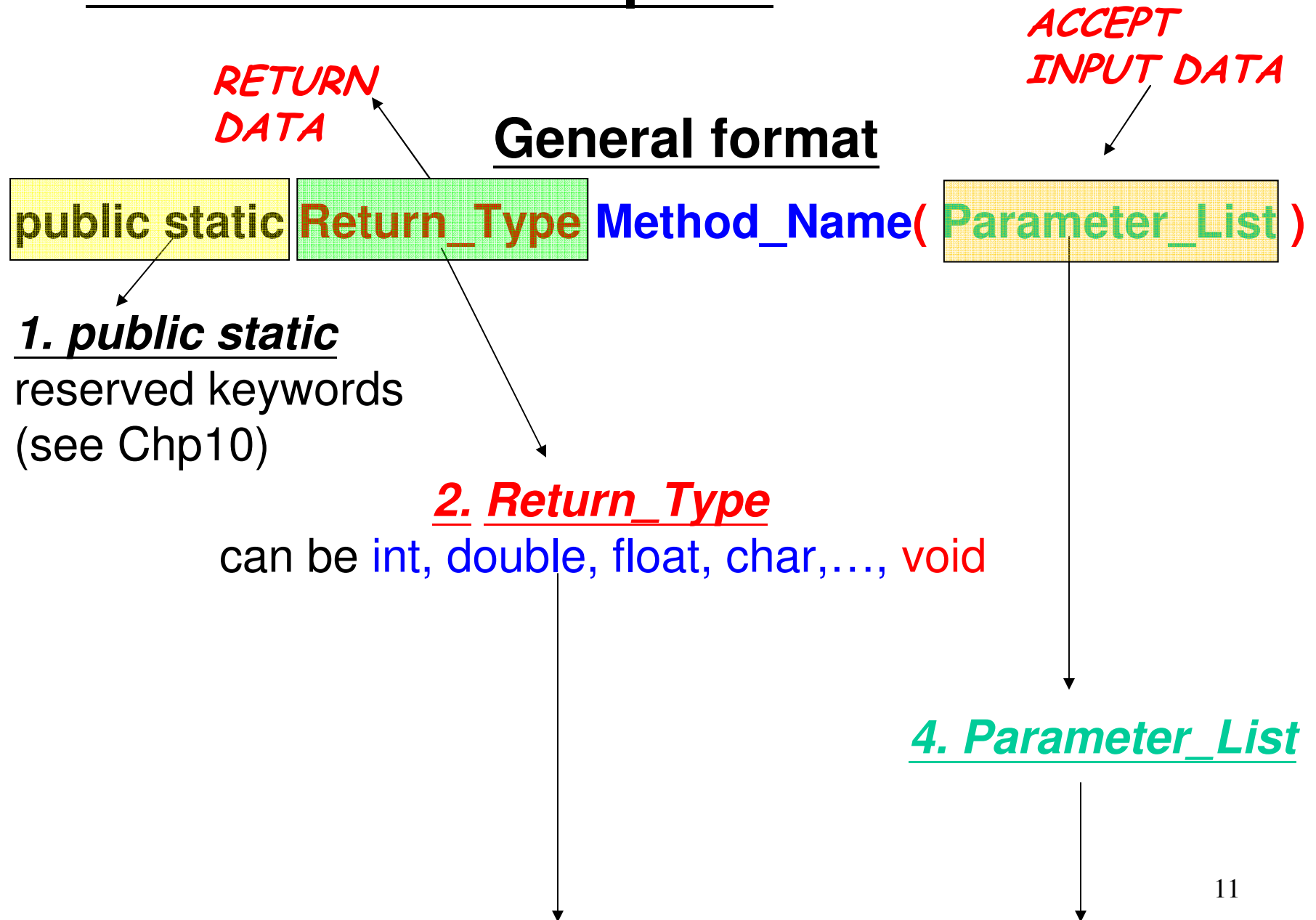
```
public class Class_Name
{
    // definitions and declarations
    public static void main(String[] args)
    {
        // main Method body
    }

    // Define helper methods here
    ...
}
```

Message Sending Concept



Method Header: 4 parts



Return_Type - example

◇ **int** -- the method will return a value of the type **int**.

```
public static int successor(int num)
{
    return num + 1;
}
```

- MUST HAVE A RETURN STATEMENT

◇ **void** -- the method will **not** return any value.

```
public static void helloNTimes(int n)
{
    int count;
    for (count = 0; count < n; count++)
        System.out.println("Hello");
    /* no return statement */
}
```

← NOTHING TO RETURN

Parameter_List

is **void** or a list of declarations for variables called **parameters**:

Type Parameter_Name[, Type Parameter_Name]

[] will be repeated zero or more times. These parameters are also called **formal parameters**. The parameters are known only **inside** the method body. They receive values from the calling method when the method is called. It provides **a mechanism for passing data between methods**.

```
public static double distance(double x, double y)
{
    return Math.sqrt(x * x + y * y);
}

public static void hello()
{
    System.out.println("Hello");
}
```

FROM CALLING METHOD

- 1/MORE FORMAL PARAMETERS

NO FORMAL PARAMETERS

Method Body

- It contains **statements** that can be declaration statements, simple statements and/or compound statements.
- The **variables** declared inside the method body are called **local variables** and are only **known within** the method.

```
public static int method(int x, int y)
{
    float fnum;
    int temp;
    .... // statements for method body
}
```

← **TWO LOCAL VARIABLES**

The return Statement

- The **return** statement is used to return a **result** computed by the method.
- Format:
return (Expression);
- It may appear in any place and in more than one place inside the method body.
- The return statement **terminates** the execution of the method and passes the control back to the calling method.

Example:

```
public static int fact(int n)
```

Method header

```
{
```

```
    int temp = 1;
```

```
    if (n < 0) {
```

```
        System.out.println("Error:no neg number.");
```

```
        return 0;
```

```
    }
```

```
    else if (n == 0)
```

```
        return 1;
```

```
    else
```

```
        for (; n > 0; n--)
```

```
            temp *= n;
```

```
        return temp;
```

```
}
```

Method body

Method call :

```
num = fact(4);
```

Method definition :

```
int fact(int n)
```

```
{
```

```
...
```

```
    return temp;
```

```
}
```

return 24

24

Note:

0! = 1

1! = 1

2! = 2 x 1

3! = 3 x 2 x 1

4! = 4 x 3 x 2 x 1....

Program Input and Output

Enter a positive number: 4

The factorial of 4 is 24

- A type **void** method may have a **return** statement to terminate the method.

```
public static void helloNTimes(int n)
{
    int count;

    if (n <= 0)
        return; /* return without any value */
    else
        for (count = 0; count < n; count++)
            System.out.println("Hello!");
}
```

Program Input and Output

Enter a number: 0

Enter a number: 2

Hello!

Hello!

Methods

- Why Methods?
- Method Definition
- **Calling a Method**
- Passing Parameters and Values
- Overloading Methods
- Scope of Variables
- The Math Class
- Wrapper Classes
- Enumerated Types
- Designing Programs with Methods
- Case Study

Calling a Method

A method call causes the method to be executed.
A method call has the following format:

Method_Name(**Argument_List**);

e.g. **helloNTimes**(4);

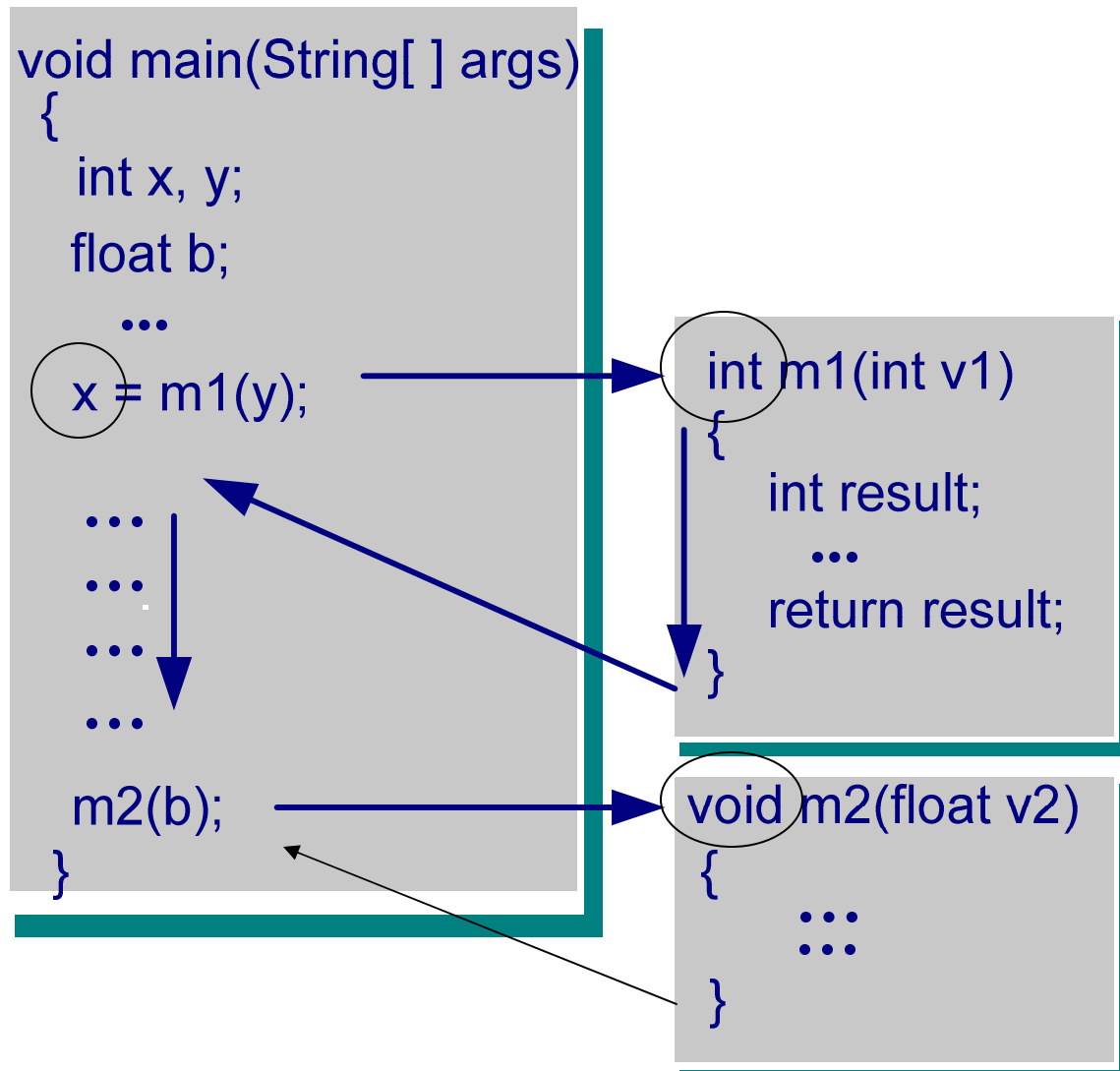
Method arguments could be constants, variables, literals, or expressions of them:

e.g. **helloNTimes**(noOfTimes);
helloNTimes(noOfTimes * 4);

We can also call a method that returns a value:

e.g. **dist** = **distance**(2.0, 4.5);

Method Calling Process





- A method is executed when it is called; **Control** goes to the first statement in the method immediately.

And **Control** goes back to the place where the method is called after

- ◇ the **last statement** of the method is executed
- ◇ OR a **return** statement is executed in the method



MUST: Understand the control flow when calling a method
i.e., trace instruction/statements step by step !!!!

Review Questions

Identify and correct the errors in the following program?

```
1 : public class Test {  
2 :     public static void main(String[] args) {  
3 :         method1(5, 6);  
4 :     }  
5 :     public static method1(int n, m) {  
6 :         n += m;  
7 :         method2(4, 5);  
8 :     }  
9 :     public static int method2(int n) {  
10 :         if (n > 0 ) return 1;  
11 :         else if (n == 0) return 0;  
12 :         else if (n < 0) return -1;  
13 :     }  
14 : }
```

How many errors here? Four!!!

Methods

- Why Methods?
- Method Definition
- Calling a Method
- **Passing Parameters and Values**
- Overloading Methods
- Scope of Variables
- The Math Class
- Wrapper Classes
- Enumerated Types
- Designing Programs with Methods
- Case Study

Passing Parameters and Returning Values

Communications between a method and the calling body is done through **parameters** and the **return value** of a method.

Need to understand the following 2 terms:

1. Parameters or (**Formal parameters**) – used when defining methods:

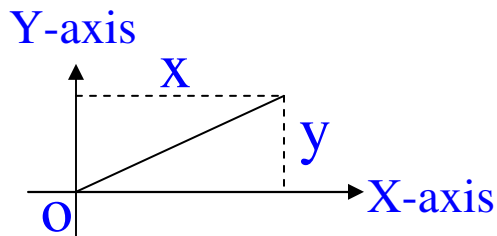
```
double distance(double x, double y);  
/* x,y – formal parameters in method definition*/
```

2. Arguments or (**Actual parameters**) –

```
a = 2.01; b=4.5;  
dist = distance(a, b);    dist = distance(3.0,5.0);  
/* a,b – arguments in the calling method*/
```


Passing Parameters and Returning Values

```
public class CallingMethods {  
    public static void main(String[] args){  
        double dist;           // local variables  
        double x = 2.0, y = 4.5; // local variables  
        double a = 3.0, b = 5.5; // local variables  
  
        dist = distance(2.0, 4.5); // arguments  
        System.out.println("The distance is " + dist);  
        dist = distance(x * y, a * b); // arguments  
        System.out.println("The distance is " + dist);  
    }  
  
    public static double distance(double x, double y)  
    {  
        // x,y - formal parameters  
        return Math.sqrt(x * x + y * y);  
    }  
}
```



Program Output

The distance is 4.924429

The distance is 18.794946

Pass by Value

- **Pass by value** refers to
 - passing parameter values into a called method, and
 - returning a resulting value (if any) to the calling method.
- **Conditions:**

(1) **types**
(2) **number**

of the **arguments** and those of the **parameters**
must **match**.

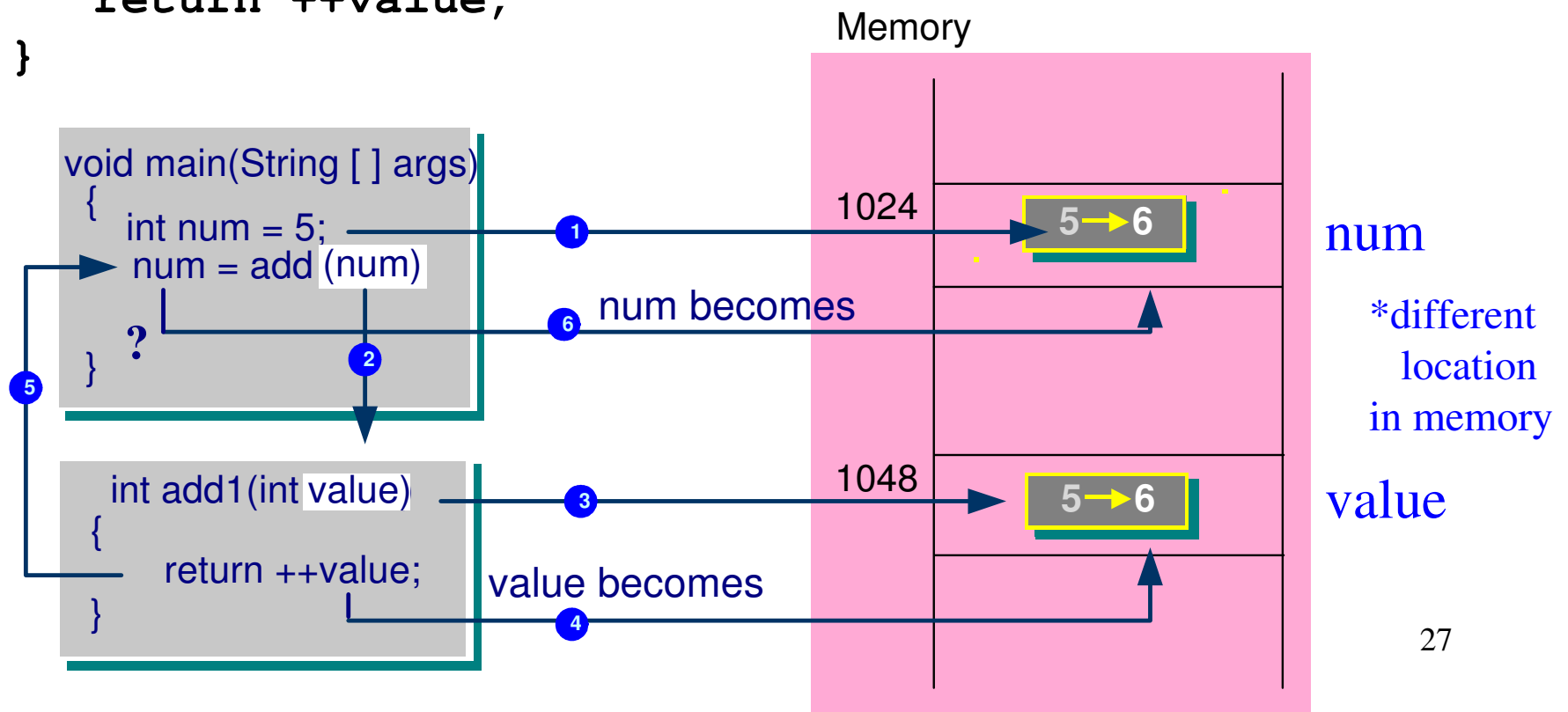
Assignment operator is used here!!!

Passing Parameters and Returning Values

Program Input and Output

The value of num is 6

```
public class PassByValue {  
    public static void main(String[] args) {  
        int num = 5;  
        num = add(num);  
        System.out.println("The value of num is " + num);  
    }  
    public static int add(int value) {  
        return ++value;  
    }  
}
```



Passing Parameters and Returning Values

Program Input and Output

The value of num is 5

```
public class PassByValue2 {  
    public static void main(String[] args) {  
        int num = 5;  
        add(num);  
        System.out.println("The value of num is " + num);  
    }  
    public static void add(int value) {  
        ++value;  
    }  
}
```

```
void main(String [ ] args)  
{  
    int num = 5;  
    add (num)  
    .....  
}
```

```
void add1(int value)  
{  
    ++value;  
}
```

Memory

1024

5

num

“num” is
not changed
in this case

1048

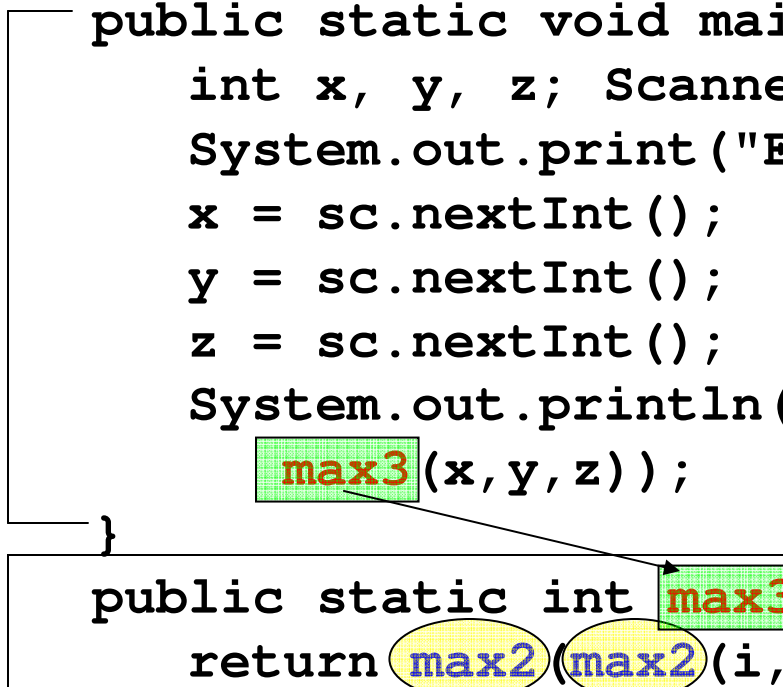
5 → 6

value

value becomes

Method Calling Other Methods

```
import java.util.Scanner;
public class CallingOtherMethods {
    public static void main(String [] args){
        int x, y, z; Scanner sc = new Scanner(System.in);
        System.out.print("Enter the three integers => ");
        x = sc.nextInt();
        y = sc.nextInt();
        z = sc.nextInt();
        System.out.println("The maximum value is " +
            max3(x,y,z));
    }
    public static int max3(int i,int j,int k) {
        return max2(max2(i,j),max2(j,k));
    }
    public static int max2(int h,int k) {
        System.out.println("Find max of " + h + " and "
            + k);
        return h>k ? h : k;
    }
}
```



NESTED METHOD CALL

Program Input and Output

```
Enter three integers => 12 24 25
Find max of 12 and 24
Find max of 24 and 25
Find max of 24 and 25
The maximum value is 25
```

Method Calling Other Methods

```
public class CallingOtherMethods {  
    public static void main(String [] args){  
        ...  
        System.out.println("The maximum value is " +  
            max3(x,y,z));  
    }  
    public static int max3(int i,int j,int k) {  
        return max2(max2(i,j),max2(j,k));  
    }  
    public static int max2(int h,int k) {  
        ...  
        return h>k ? h : k;  
    }  
}
```

The diagram illustrates the sequence of method calls in the provided code. Arrows and numbers indicate the flow of calls:

- 1**: `max3` calls `max2(i,j)`.
- 2**: `max3` calls `max2(j,k)`.
- 3**: `max2(j,k)` calls `max2(i,j)`.
- 4**: `max2(i,j)` calls `max2(h,k)`.
- 5**: `max3` calls `max2(h,k)`.

Methods

- Why Methods?
- Method Definition
- Calling a Method
- Passing Parameters and Values
- **Overloading Methods**
- Scope of Variables
- The Math Class
- Wrapper Classes
- Enumerated Types
- Designing Programs with Methods
- Case Study

Overloading of Methods

When a method is overloaded, it is designed to perform differently when it is supplied with different *signatures*

Signature of a method:

- (1) **The number of parameters;**
- (2) **The data type of each parameter.**

Advantage:

Overloading allows us to use **same method name** for

- **methods** that have **similar tasks** as long as they have **different signatures**

Without overloading, we need to use **different method names** for each similar task instead of just one.

Note: We are overloading the name of methods

Overloading of Methods

```
public class MethodOverLoading {  
    public static void main(String[] args) {  
        int x=10, y=20, z=5;  
        double i=4.5, j=5.5;
```

```
        System.out.println("findMin(x,y) with int  
        args = " + findMin(x,y));
```

```
        System.out.println("findMin(i,j) with double  
        args = " + findMin(i,j));
```

```
        System.out.println("findMin(x,y,z) with int  
        args = " + findMin(x,y,z));
```

```
    }
```

```
public static int findMin(int num1, int num2) {  
    if (num1 < num2)  
        return num1;  
    else  
        return num2;  
}
```

```
public static double findMin(double num1, double num2) {  
    if (num1 < num2)  
        return num1;  
    else  
        return num2;  
}
```

```
public static int findMin(int num1, int num2, int num3) {  
    return findMin(findMin(num1, num2), findMin(num2, num3));  
}
```

```
}
```

Program Output

findMin(x,y) with int args = 10

findMin(i,j) with double args = 4.5

findMin(x,y,z) with int args = 5

Without Overloading?

```
public static int findMin2i(int num1, int num2){  
    if (num1 < num2)  
        return num1;  
    else  
        return num2;  
}
```

```
public static double findMin2d(double num1, double num2){  
    if (num1 < num2)  
        return num1;  
    else  
        return num2;  
}
```

```
public static int findMin3i(int num1, int num2, int num3){  
    return findMin(findMin(num1, num2), findMin(num2, num3));  
}
```

```
System.out.println( "... " + findMin2i(x, y));  
System.out.println( "... " + findMin2d(i, j));  
System.out.println( "... " + findMin3i(x, y, z));
```

Review Questions

What's wrong with the following program?

```
1 : public class Test {  
2 :     public static void main(String[] args) {  
3 :         method(5);  
4 :     }  
5 :     public static void method(int x) {  
6 :     }  
7 :     public static int method(int y) {  
8 :         return y;  
9 :     }  
10 : }
```

*OK??
Why??*

Method Signature!!!!!!!

Last Note:

1. Overloading -> More convenient and better code readability!!!
2. But make sure different method signature (only parameters)
3. More example: System.out.println (also use overloading)
4. When you use overloading in your own methods,
make sure they all do a similar thing

Methods

- Why Methods?
- Method Definition
- Calling a Method
- Passing Parameters and Values
- Overloading Methods
- **Scope of Variables**
- The Math Class
- Wrapper Classes
- Enumerated Types
- Designing Programs with Methods
- Case Study

Scope and Duration of Variables

Variables have many properties: **name**, **type**, **size** and **value**.

Every variable has **scope** and **duration**

• **Scope** – section of code that the variable can be used. **Code**

• **Duration** – life time of the variable in memory **Run-time**

Local Variables – variables declared within a method.

- storage is **created** to store the value of the variable
- can only be accessed by expressions or statements **within the same method**
- **destroyed** when the method completes execution

Scope of Variables

Two scopes: block scope or class scope.

1. **Block scope** – if the variable is visible until the end of the block containing the definition, e.g. **local variables**, and method **parameters** have block scope.
2. **Class scope** – if the variable is visible from the beginning of a class definition until the end of the class definition, e.g. **class variables** (constants in a class).
(Will be discussed in Chapter 10)

Scope of Variables

Class variable

```
public class ScopeOfVariables {  
    private static int classVar = 10;  
    // classVar has class scope  
    public static void main(String[] args){  
        int localVar = 5; // localVar has block scope  
        System.out.println("main(): classVar= " + classVar);  
        System.out.println("main(): localVar= " + localVar);  
        method1();  
        System.out.println("main(): classVar= " + classVar);  
        System.out.println("main(): localVar= " + localVar);  
    }  
    public static void method1(){  
        int localVar; // localVar has block scope  
  
        localVar = 50;  
        classVar += localVar;  
        System.out.println("method1(): classVar= "+classVar);  
        System.out.println("method1(): localVar= "+localVar);  
    }  
}
```


Program Output

```
main(): classVar= 10  
main(): localVar= 5  
method1(): classVar= 60  
method1(): localVar= 50  
main(): classVar= 60  
main(): localVar= 5
```

```
public class ScopeOfVariables  
{
```

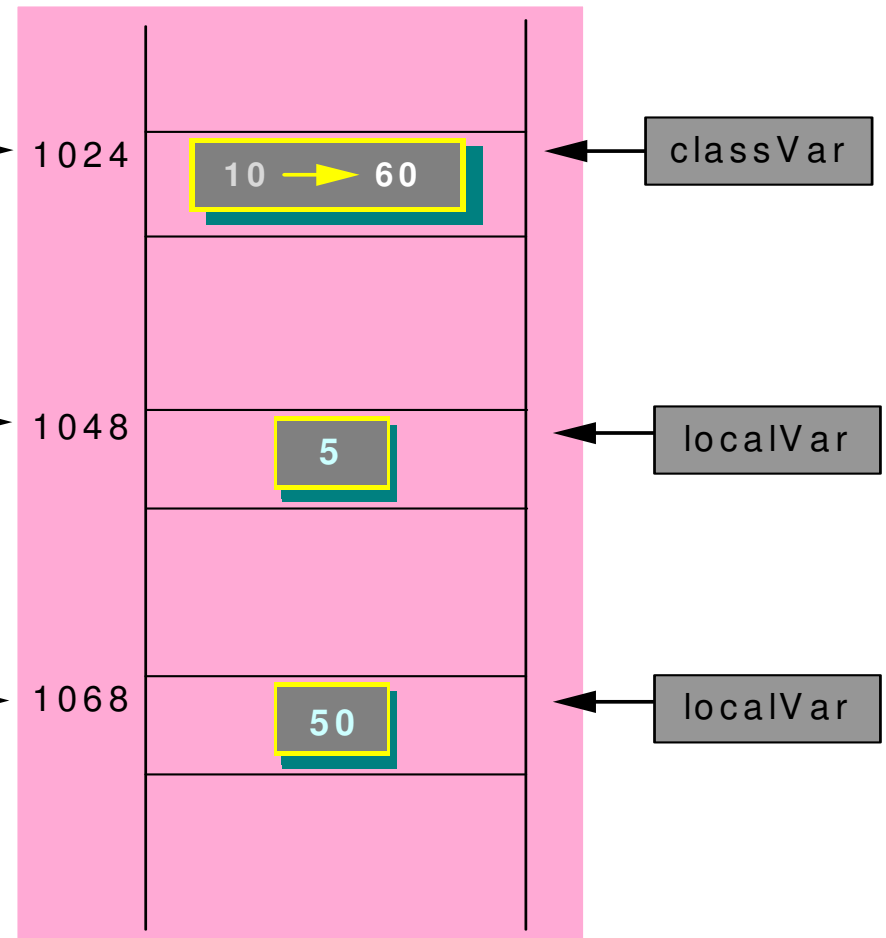
```
    private static int classVar;
```

```
    public static void main (String [ ] args)  
    {  
        int localVar = 5;  
    }
```

```
    public static void method1()  
    {  
        int localVar;  
        localVar = 50  
        classVar += localVar;  
    }
```

```
}
```

Memory



Duration of Variables

Two types of duration: static or automatic.

1. **Static Duration** – if the variable exists throughout program execution.
2. **Automatic Duration** – if the variable only exists within a block where the variable is defined.

The variables are **created** during declaration in the block, and **remains** in the memory until the block in which they are declared is exited. Once the block goes out of scope, the memory storage is released back to the OS.

Note: All variables with **class scope** have **static duration**.

Duration of Variables

Class variable

```
public class StorageDuration {  
    private static int classVar = 5;  
                        // classVar - static duration  
    public static void main(String[] args){  
        method1(classVar);  
    }  
    public static void method1(int paraVar)  
                        // paraVar - automatic duration  
    {  
        int localVar; // localVar - automatic duration  
        for (localVar = 0; localVar < paraVar; localVar++)  
            System.out.println("localVar = " + localVar +  
                               " paraVar = " + paraVar);  
        System.out.println  
        ("localVar = " + localVar);  
    }  
}
```

Program Output

```
localVar = 0 paraVar = 5  
localVar = 1 paraVar = 5  
localVar = 2 paraVar = 5  
localVar = 3 paraVar = 5  
localVar = 4 paraVar = 5  
localVar = 5
```

Note:

if we declare a variable within the initialization part of a *for* statement, the variable will be **local** to the structure of the *for* loop.

For example:

```
for (int localVar2=0; localVar2 < paraVar; localVar2++)  
{  
    System.out.println("localVar2 = " + localVar2 +  
        " paraVar = " + paraVar);  
}
```

the variable will only be visible within the structure defined by the *for* loop.

i.e. the variable **localVar2** cannot be used **outside** the *for* loop.

Methods

- Why Methods?
- Method Definition
- Calling a Method
- Passing Parameters and Values
- Overloading Methods
- Scope of Variables
- **The Math Class**
- Wrapper Classes
- Enumerated Types
- Designing Programs with Methods
- Case Study

Mathematical Methods

- The predefined **Math** class from **java.lang** package provides a set of mathematical functions such as **abs()**, **sqrt()**, **pow()**, **sin()**, **cos()**, etc.

Var:	Method call:	Result:
abs =	Math.abs(5);	// = 5
abs =	Math.abs(-5);	// = 5
sqrt =	Math.sqrt(9.0);	// = 3.0
pow =	Math.pow(2.0,4.0);	// =16.0
log =	Math.log(9.0);	// = 2.20
exp =	Math.exp(-3.2);	// = .04
min =	Math.min(3,6);	// = 3.0
min =	Math.min(3.5, 6.5);	// = 3.5
max =	Math.max(3,6);	// = 6
max =	Math.max(3.5,6.5);	// = 3.5
round =	Math.round(5.2);	// = 5
round =	Math.round(5.6);	// = 6
ceil =	Math.ceil(5.2);	// = 6.0
ceil =	Math.ceil(5.9);	// = 6.0
floor =	Math.floor(5.2);	// = 5.0
floor =	Math.floor(5.9);	// = 5.0
sin =	Math.sin(5.2);	// = -.88
cos =	Math.cos(5.2);	// = .47
tan =	Math.tan(5.2);	// = -1.89
random =	Math.random();	// = .17

Function	Description	Result Type
abs (x)	Returns the absolute value of x .	same data type as argument
sqrt (x)	Returns the square root of x .	double
pow (x, y)	Returns x raised to the y power, i.e. x^y .	double
log (x)	Returns the natural logarithm of x .	double
exp (x)	Returns the exponential number e (i.e. 2.718282) raised to the x power.	double
min (x, y)	Returns the smaller of its arguments.	same data type as argument
max (x, y)	Returns the larger of its arguments.	same data type as argument
round (x)	Returns the nearest long integer to x.	int or long
random ()	Returns a random number greater than or equal to 0.0 and less than 1.0.	double
ceil (x)	Returns the smallest integer value that is not less than x.	double
floor (x)	Returns the largest integer value that is not greater than x.	double
sin (x)	Returns the sine of x , where x is in radians.	double
cos (x)	Returns the cosine of x , where x is in radians.	double
tan (x)	Returns the tangent of x , where x is in radians.	double

Example on Math Class

```
public class MathExpressions {  
  
    public static void main(String[] args)    {  
        double sqrt, pow;  
  
        sqrt = Math.sqrt(9.0);  
        System.out.println("Math.sqrt(9.0) = " + sqrt);  
        pow = Math.pow(2.0, 4.0);  
        System.out.println("Math.pow(2.0, 4.0) = " + pow);  
  
    }  
}
```

To see in Chapter 10: Classes and Objects

NOTE: STATIC (CLASS) METHODS - You do not need to create objects in order to invoke the static methods. Compared with

***Scanner sc = new Scanner(System.in);
int x = sc.nextInt();***

Methods

- Why Methods?
- Method Definition
- Calling a Method
- Passing Parameters and Values
- Overloading Methods
- Scope of Variables
- The Math Class
- **Wrapper Classes**
- Enumerated Types
- Designing Programs with Methods
- Case Study

Wrapper Classes

Wrapper class wraps a **primitive data type** into an **object** like a container. For example, the **Integer** class represents an integer value.

E.g. create an **integer object** to store a primitive **integer value**:

Integer numObject = new Integer(100);

Wrapper classes provide **methods** for managing the associated primitive data types :

(i) byte byteValue();	(ii) double doubleValue();
(iii) float floatValue();	(iv) int intValue();
(v) long longValue();	

- returns the value of this integer as the corresponding primitive type.

The Wrapper Classes also provide other **static** (or Class) methods:

(i) static int **parseInt(String str)**

e.g.

```
int num = Integer.parseInt(str);
```

Converts an integer stored in a **String str** (e.g. “888”) into its corresponding integer value of 888.

In addition, other static methods:

(ii) static String toBinaryString(int num);

(iii) static String toHexString(int num);

(iv) static String toOctalString(int num);

- These static methods return a **string** representation of the specified integer value in the corresponding base;

Other Wrapper Classes

Primitive Data Type	Wrapper Class
<code>boolean</code>	<code>Boolean</code>
<code>byte</code>	<code>Byte</code>
<code>char</code>	<code>Character</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>
<code>void</code>	<code>Void</code>

Autoboxing and Unboxing (in Ver 1.5)

Autoboxing – convert a primitive value into its corresponding wrapper object.

```
Integer numObject1;  
int num1 = 888;  
numObject1 = num1;
```

Unboxing – extract the value from a wrapper object and assign the value into the corresponding primitive data type.

```
Integer numObject2 = new Integer(888);  
int num2;  
num2 = numObject2;
```

Will see more in **Chapter 11**: Strings Conversion

Methods

- Why Methods?
- Method Definition
- Calling a Method
- Passing Parameters and Values
- Overloading Methods
- Scope of Variables
- The Math Class
- Wrapper Classes
- **Enumerated Types**
- Designing Programs with Methods
- Case Study

Enumerated Types

Enumerated data types allow programmers to define **a set of symbolic names** to represent the variable type.

The keyword **enum** is used to declare enumerated types. For example: (step 1: declare)

```
enum Day {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
```

A variable is declared as (step 2: create a variable)

```
Day today;
```

The values are accessed through the **name of the type**:

```
Day today = Day.Tue;
```

(step 3: use the values)

The **advantage** of enumerated data type is the freedom to program using **symbolic names** which are more **readable** than numbers. Compare it with: **int today = 2 ;**

1

The enumeration values are represented **internally** by an **integer number** starting from **zero** (cannot be changed by user programs):

Sun=0 Mon=1 Tue=2 Wed=3 Thu=4 Fri=5 Sat=6

In Java, an **enumerated type** is a **class**, and the variables of an enumerated type are **object variables**.

2

Methods for the object variables:

- (1) The **ordinal** method – it returns the **integer value** associated with an enumerated type value.
- (2) The **name** method – it returns the **name** of an enumerated type value (as a String).


```

public class EnumTypes {
    enum Day {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
    public static void main(String[] args)    {
        Day yesterday, today, tomorrow;
        yesterday = Day.Mon;
        today      = Day.Tue;
        tomorrow   = Day.Wed;

        System.out.println("yesterday name = " +
            yesterday.name() );
        System.out.println("yesterday ordinal = " +
            yesterday.ordinal() );
        System.out.println("today name = " +
            today.name() );
        System.out.println("today ordinal = " +
            today.ordinal() );
        System.out.println("tomorrow name = " +
            tomorrow.name() );
        System.out.println("tomorrow
            ordinal = " +
            tomorrow.ordinal() );
    }
}

```

Program Output

```

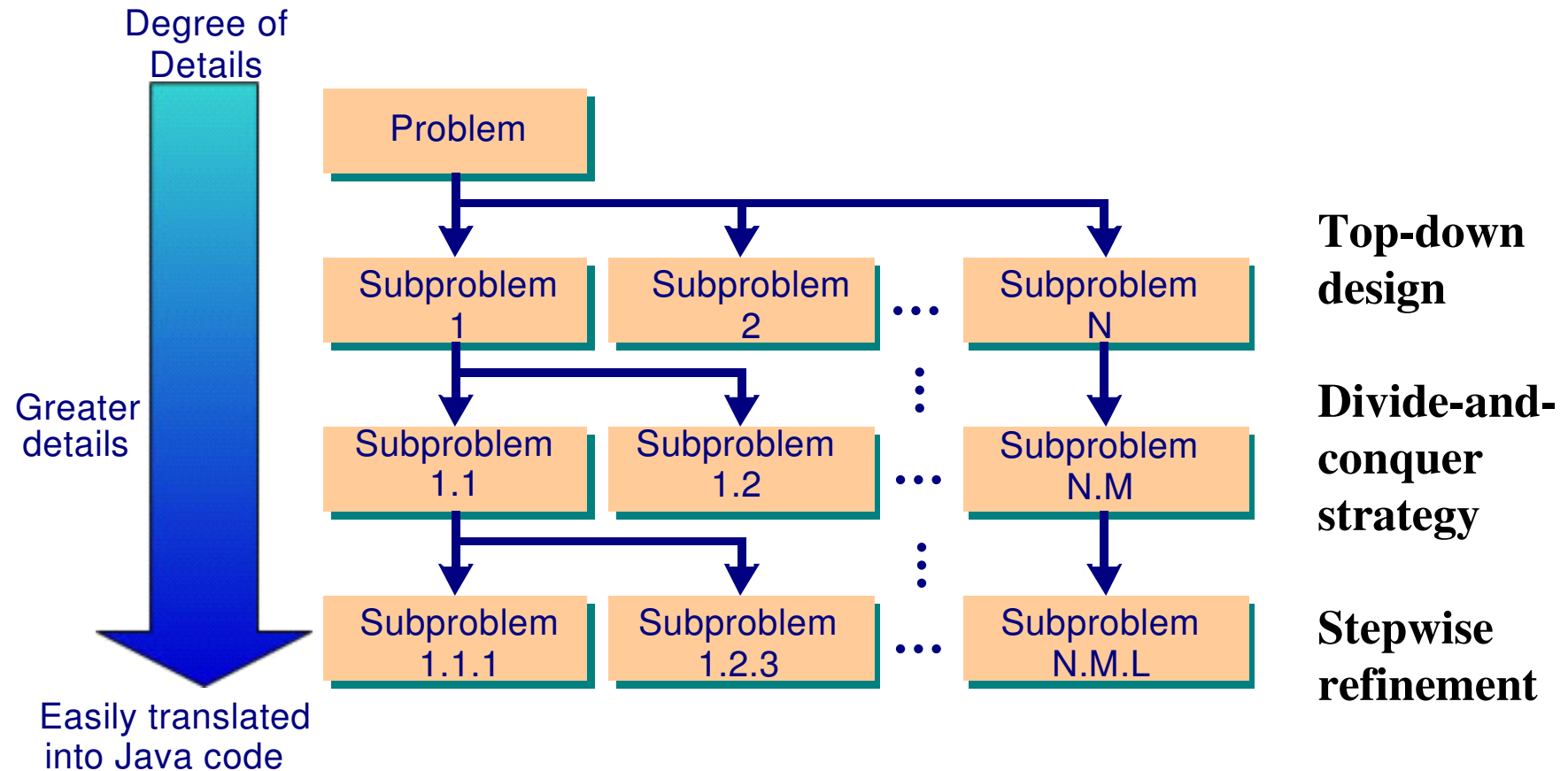
yesterday name = Mon
yesterday ordinal = 1
today name = Tue
today ordinal = 2
tomorrow name = Wed
tomorrow ordinal = 3

```

Methods

- Why Methods?
- Method Definition
- Calling a Method
- Passing Parameters and Values
- Overloading Methods
- Scope of Variables
- The Math Class
- Wrapper Classes
- Enumerated Types
- **Designing Programs with Methods**
- Case Study

Program Design Using Top-Down Approach



Philosophy of Procedural Programming (PP)

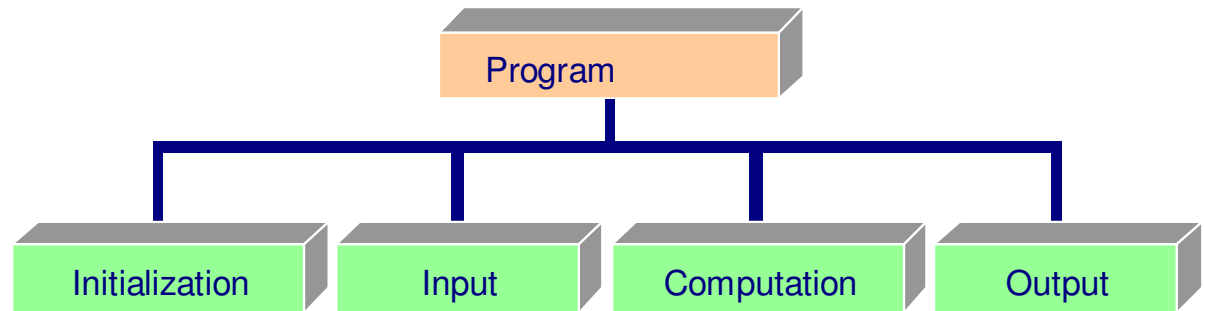
So top-down stepwise refinement means to

1. start with the high level description of the program
2. **decompose** the program (the main() method) into successively **smaller methods** until we arrive at suitably sized methods
3. then design the code for the **individual method** using stepwise refinement.
4. at each level we are only concerned with what the lower level methods will do, but not how.

Advantages:

- programs are easier to **write** and **debug**
- reduce program development time and enhance program reliability
- code reuse (reuse common methods)

Basic Program Structure



Initial Algorithm

- Initialization of variables
- Read the input data
- Perform the computation
- Print the output results

Stepwise Refinement

- further refinement of the initial algorithm, especially in **computation**

Programming Steps

1. Identify distinct sections (methods) of the program
2. For each distinct section (method), identify the **input** and **output**
3. Define the **signature** and the **return type** of the method
4. Write the code into a new **static method**
5. Place the method calls in the corresponding locations in the **main method**

Methods

- Why Methods?
- Method Definition
- Calling a Method
- Passing Parameters and Values
- Overloading Methods
- Scope of Variables
- The Math Class
- Wrapper Classes
- Enumerated Types
- Designing Programs with Methods
- **Case Study**

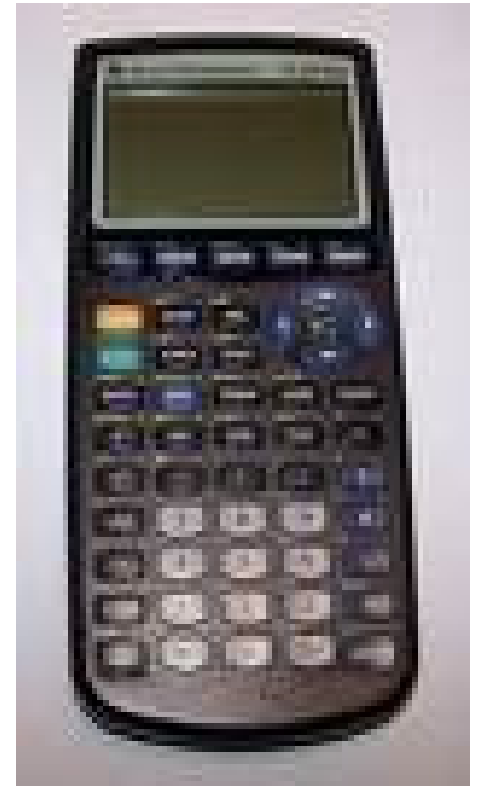
Case Study: A Simple Calculator

Problem Specification

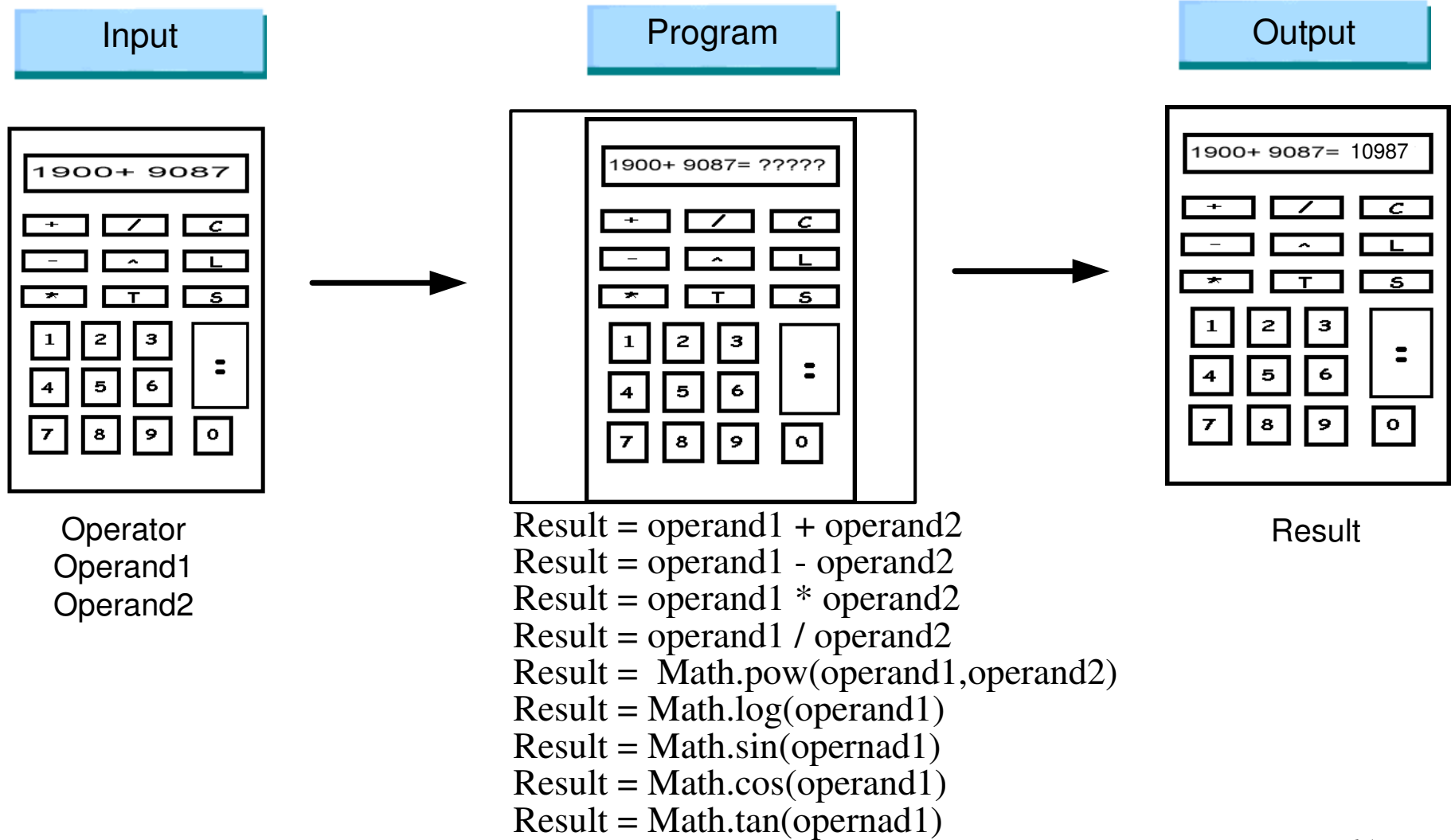
Write a program to implement a simple calculator to perform arithmetic calculations:

- addition, subtraction, division, multiplication, and
- mathematical functions such as sine, tangent and cosine.

The user will specify the required operation, and one or two operands depending on whether the operation is unary or binary in order to perform the calculation.



Problem Analysis



Problem Analysis

Required inputs:

- the operator, operand 1, operand 2

Required outputs:

- the calculation result

Formulas:

- addition: $\text{result} = \text{operand 1} + \text{operand 2}$
- subtraction: $\text{result} = \text{operand 1} - \text{operand 2}$
- multiplication: $\text{result} = \text{operand 1} * \text{operand 2}$
- division: $\text{result} = \text{operand 1} / \text{operand 2}$
- power: $\text{result} = \text{Math.pow}(\text{operand 1}, \text{operand 2})$
- log: $\text{result} = \text{Math.log}(\text{operand 1})$
- sine: $\text{result} = \text{Math.sin}(\text{operand 1})$
- cosine: $\text{result} = \text{Math.cos}(\text{operand 1})$
- tangent: $\text{result} = \text{Math.tan}(\text{operand 1})$

Program Design

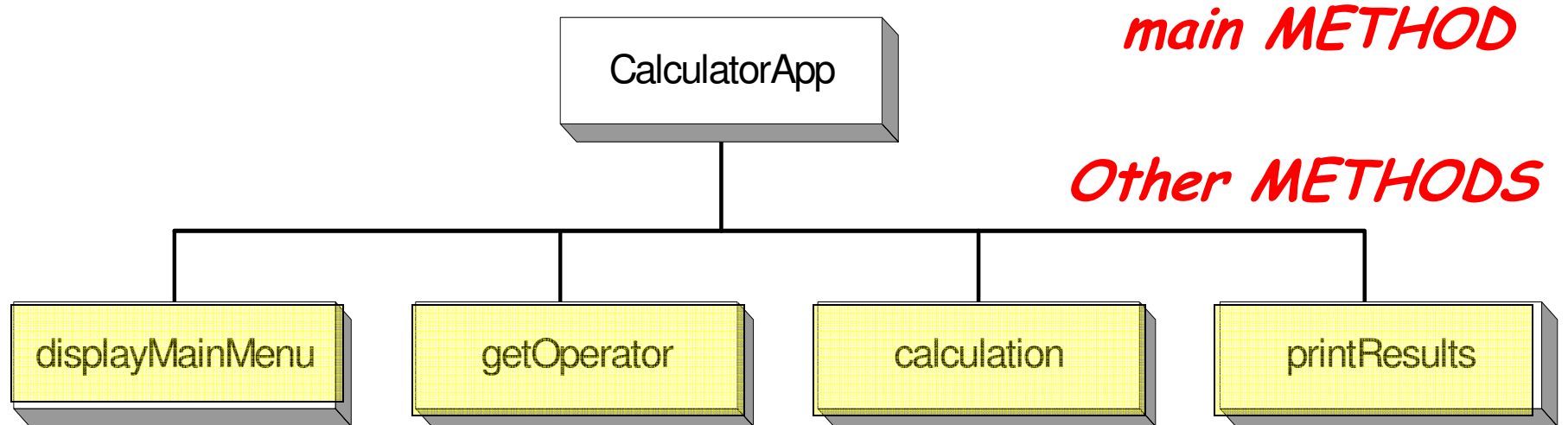
Initial Algorithm

1. Display menu.
2. Read operator.
3. Read the first operand.
4. If the operator is binary, then read the second operand.
5. Calculate the result using the operator, and the two operands.
6. Print output result.
7. Repeat steps 1 to 6 until the user quits the program.

Pseudocode

– Exercise

Program Design



Program Design

Algorithm in Pseudocode

main:

CALL displayMainMenu

DO

SET operator TO getOperator

READ operand1

IF operator is equal to '+', '*', '-', '/' OR '^'

READ operand2

ENDIF

SET result TO calculation(operator, operand1, operand2)

CALL printResults(result, operator, operand1, operand2)

PRINT "Enter 1 to repeat or 2 to quit"

READ option

WHILE option = 1

displayMainMenu:

PRINT "====Simple Calculator===="

PRINT "+ :addition (i.e. num1 + num2) "

...

PRINT "T :tangent (i.e. tan(num1)) "

```
getOperator:
```

```
    READ operator  
    RETURN operator
```

```
printResults(result, operator, operand1, operand2):
```

```
    IF operator is equal to '+', '*', '-', '/' OR '^'  
        PRINT operand1, operator, result  
    ELSE  
        PRINT result  
    ENDIF
```

```
calculation(operator, operand1, operand2):
```

```
    SET result TO 0.0
```

```
    IF operator = '+'
```

```
        COMPUTE result = operand1 + operand2
```

```
    ELSE IF operator = '-'
```

```
        COMPUTE result = operand1 - operand2
```

```
    ELSE IF operator = '*'
```

```
        COMPUTE result = operand1 * operand2
```

```
    ELSE IF operator = '/'
```

```
        COMPUTE result = operand1 / operand2
```

```
    ELSE IF operator = '^'
```

```
        SET result TO Math.pow(operand1,operand2)
```

```
    ELSE IF operator = 'L'
```

```
        SET result TO Math.log(operand1)
```

```
    ELSE IF operator = 'S'
```

```
        SET result TO Math.sin(operand1)
```

```
    ELSE IF operator = 'C'
```

```
        SET result TO Math.cos(operand1)
```

```
    ELSE IF operator = 'T'
```

```
        SET result TO Math.tan(operand1)
```

```
    ELSE
```

```
        PRINT "Invalid Operator!"
```

```
    ENDIF
```

```
    RETURN result
```

Implementation

```
import java.util.Scanner;
public class CalculatorApp {
    public static void main(String[] args) {
        char option = ' ', operator = ' ';
        double operand1 = 0, operand2 = 0, result;
        Scanner sc = new Scanner(System.in);
```

```
        displayMainMenu();
```

Use do..while loop

```
        do {
            operator = getOperator();
            System.out.println("Enter number 1: ");
            operand1 = sc.nextDouble();
            // Check whether second number is needed
            if (operator == '+' || operator == '*' ||
                operator == '-' || operator == '/'
                || operator == '^') {
                System.out.println("Enter number 2: ");
                operand2 = sc.nextDouble();
            }
            result = calculation(operator, operand1, operand2);
            printResults(result, operator, operand1, operand2);
            System.out.println("Enter 1 - repeat or 2 - quit");
            option = sc.nextInt();
        } while (option == 1);
```

```
        System.out.println("Thank You!!");
```

```
    }
```



```
// Display menu
public static void displayMainMenu() {
    System.out.println("====Simple Calculator====");
    System.out.println("Enter operation: ");
    System.out.println("+ :addition (i.e. num1 + num2");
    System.out.println("- :subtraction (i.e. num1 - num2");
    System.out.println("* :multiplication (i.e.num1*num2");
    System.out.println("/ :division (i.e. num1/num2");
    System.out.println("^ :power (i.e. pow(num1, num2)");
    System.out.println("L :logarithm (i.e. log(num1))");
    System.out.println("S :sine (i.e. sin(num1))");
    System.out.println("C :cosine (i.e. cos(num1))");
    System.out.println("T :tangent (i.e. tan(num1))");
    System.out.println("Enter Q to Quit");
    System.out.println("=====");
}
```

```
// Get the operator
public static char getOperator() {
    Scanner sc = new Scanner(System.in);
    String op = sc.next();
    return (op.charAt(0));
}

// Print the results
```

```

public static void printResults(double num, char operator,
    double operand1, double operand2) {
    if (operator=='+' || operator=='*' ||
        operator=='-' || operator=='/' || operator=='^')
        System.out.println(operand1 + " " + operator + " "
            + operand2 + " = " + num);
    else
        System.out.println("Result : " + num);
}

```

```

// Perform operations
public static double calculation(char operator, double
    operand1, double operand2) {
    double result = 0.0;
    switch (operator) {
        case '+': result = operand1 + operand2; break;
        case '-': result = operand1 - operand2; break;
        case '*': result = operand1 * operand2; break;
        case '/': result = operand1 / operand2; break;
        case '^': result = Math.pow(operand1, operand2); break;
        case 'l': result = Math.log(operand1); break;
        case 'S': result = Math.sin(operand1); break;
        case 'C': result = Math.cos(operand1); break;
        case 'T': result = Math.tan(operand1); break;
        default : System.out.println("Invalid Operator!");
    }
    return result;
}

```

```

}

```

Testing

Program input and output

=====Simple Calculator=====

+ :addition (i.e. num1 - num2)

...

Enter Q to Quit

=====

T

Enter number 1:

123

Result : 0.5179274715856552

Enter Y - repeat or Q - quit

Y

/

Enter number 1:

10

Enter number 2:

2

10.0 / 2.0 = 5.0

Enter Y - repeat or Q - quit

Q

Thank you!!

Key Terms

- modular programming
- method header, method body
- return type
- parameter list
- return value
- method signature
- pass by value
- actual parameter
- formal parameter
- method overloading
- scope of variable
- duration of variable
- top-down program design
- stepwise refinement
- divide and conquer

Further Reading

- Read Chapter 8 on “Methods” of the textbook.
- Read other case studies from the chapter.