

# Chapter 6: Methods

TAO Yida

[taoyd@sustech.edu.cn](mailto:taoyd@sustech.edu.cn)

# Objectives

- ▶ What is modular programming (模块化编程)
- ▶ How to declare and use methods
- ▶ Method overloading
- ▶ Passing arguments
- ▶ Method call stack

# Problem Solving

- ▶ The programs we have written so far solve simple problems (find the max in an array of numbers).
- ▶ They are short and everything fits well in a `main` method

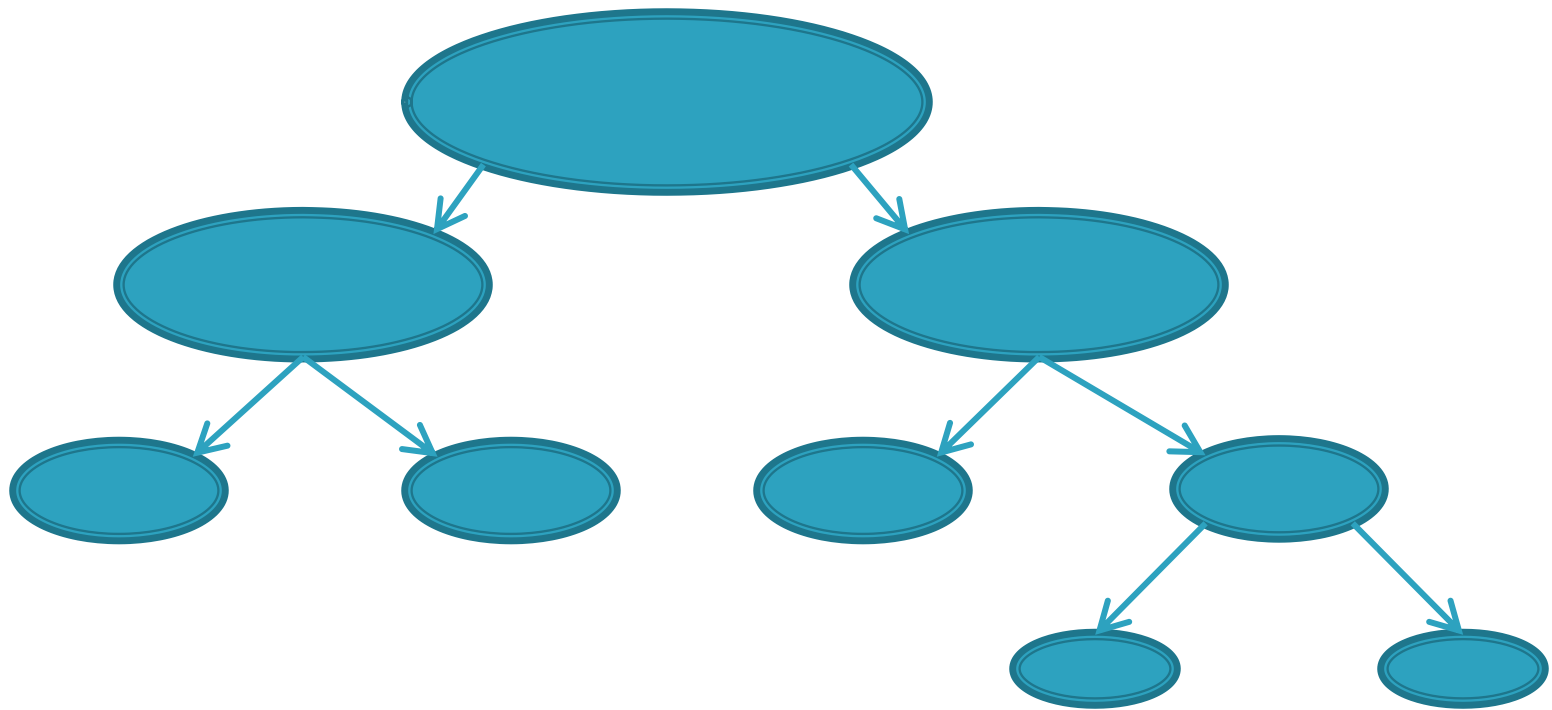


What if you are asked to **solve complex problems**, e.g., building a climate model (天气预测系统) via analyzing big data?

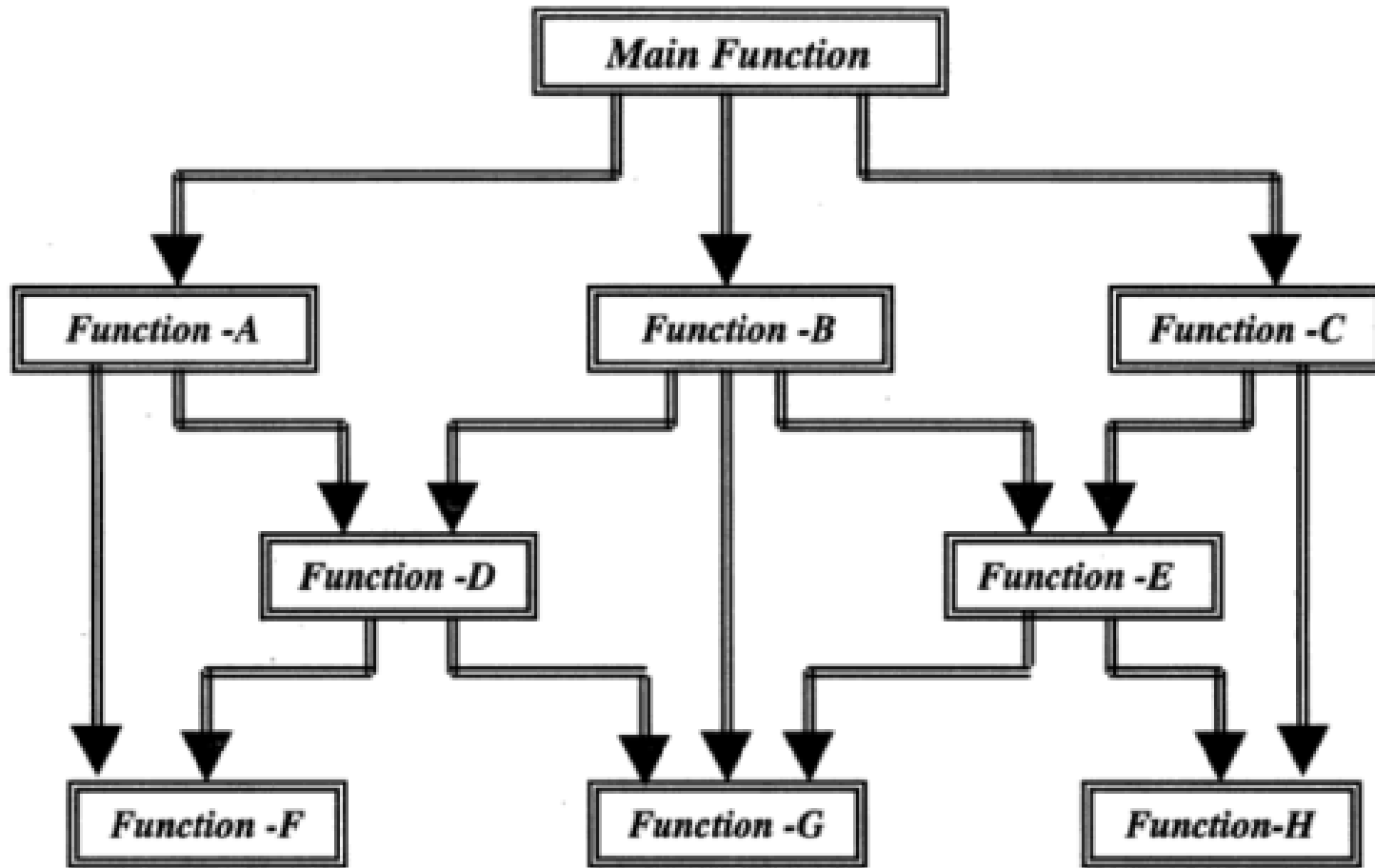
Write a giant `main` method?

# Divide and Conquer (分而治之)

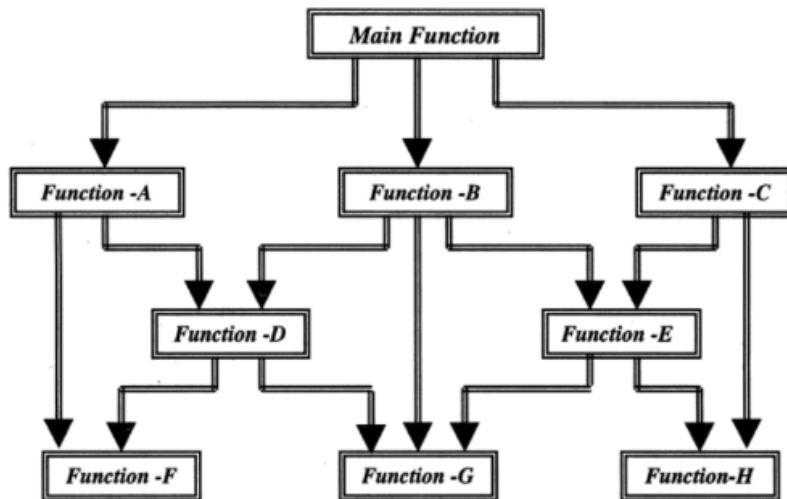
- ▶ Decompose a big/complex task into smaller one and solve each of them



# Modular Programming



# Benefits of Modular Programming



- ▶ Improved readability
- ▶ Reduce duplication and improve reusability
- ▶ Easier to update, test, and fix
- ▶ Easy collaboration

Methods facilitate the **design**, **implementation**, **operation** and **maintenance** (维护) of large programs

# Objectives

- ▶ What is modular programming (模块化编程)
- ▶ How to declare and use methods
- ▶ Method overloading
- ▶ Passing arguments
- ▶ Method call stack

# Using Methods

```
import java.util.Scanner;  
public class MaximumFinder {
```

The class defines two methods

```
    public static void main(String[] args) {  
        Scanner input = new Scanner(System.in);  
        System.out.print("enter three floating-point values: ");  
        double number1 = input.nextDouble();  
        double number2 = input.nextDouble();  
        double number3 = input.nextDouble();  
        double result = maximum(number1, number2, number3);  
        System.out.println("max is " + result);  
    }
```

```
    public static double maximum(double x, double y, double z) {  
        double max = x;  
        if(y > max) max = y;  
        if(z > max) max = z;  
        return max;  
    }
```

Find the largest of 3 double values



# Using Methods

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

```
    Scanner input = new Scanner(System.in);
```

```
    System.out.print("enter three floating-point values: ");
```

```
    double number1 = input.nextDouble();
```

```
    double number2 = input.nextDouble();
```

```
    double number3 = input.nextDouble();
```

```
    double result = maximum(number1, number2, number3);
```

```
    System.out.println("max is " + result);
```

```
}
```

You need to call it explicitly to tell it to perform its task

```
public static double maximum(double x, double y, double z) {
```

```
    double max = x;
```

```
    if(y > max) max = y;
```

```
    if(z > max) max = z;
```

```
    return max;
```

```
}
```

Method don't get called automatically after declaration.

# Declaring a Method

Modifiers + Return type + Method name + Parameters

```
public static double maximum( double x, double y, double z ) {  
    double max = x;  
    if(y > max) max = y;  
    if(z > max) max = z;  
    return max;  
}
```

**Method body** contains one or more statements that perform the method's task

The **return statement** returns a value (or just control) to the point in the program form which the method is called

# static Methods

- ▶ Sometimes a method performs a task that **does not** depend on the contents of any object
  - Method applies to the class in which it's declared
  - Known as a **static method** or a **class method**
  - Has the keyword **static** before the return type in the declaration
  - Called via the class name and a dot (.) separator

In `java.lang.Math` class:

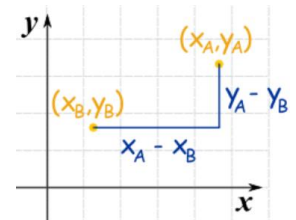
```
public static double pow(double a, double b)
```

Returns the value of the first argument raised to the power of the second argument.

# static Methods

```
public static double pow(double a, double b)
```

- ▶ Static methods can be called without the need for an object of the class to exist



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

```
double x1 = 3.14, y1 = -2.98;
```

```
double x2 = -2.71, y2 = 7.15;
```

$$c = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}$$

```
double dist = Math.pow(Math.pow(x2-x1, 2) + Math.pow(y2-y1, 2), 0.5);
```

```
System.out.println("The distance of the two points is " + dist);
```

```
}
```

Static methods can be called directly using class names.

## Many more useful static methods in `java.lang.Math` class:

Method	Description	Example
<code>abs( x )</code>	absolute value of $x$	<code>abs( 23.7 )</code> is 23.7 <code>abs( 0.0 )</code> is 0.0 <code>abs( -23.7 )</code> is 23.7
<code>ceil( x )</code>	rounds $x$ to the smallest integer not less than $x$	<code>ceil( 9.2 )</code> is 10.0 <code>ceil( -9.8 )</code> is -9.0
<code>cos( x )</code>	trigonometric cosine of $x$ ( $x$ in radians)	<code>cos( 0.0 )</code> is 1.0
<code>exp( x )</code>	exponential method $e^x$	<code>exp( 1.0 )</code> is 2.71828 <code>exp( 2.0 )</code> is 7.38906
<code>floor( x )</code>	rounds $x$ to the largest integer not greater than $x$	<code>floor( 9.2 )</code> is 9.0 <code>floor( -9.8 )</code> is -10.0
<code>log( x )</code>	natural logarithm of $x$ (base $e$ )	<code>log( Math.E )</code> is 1.0 <code>log( Math.E * Math.E )</code> is 2.0
<code>max( x, y )</code>	larger value of $x$ and $y$	<code>max( 2.3, 12.7 )</code> is 12.7 <code>max( -2.3, -12.7 )</code> is -2.3
<code>min( x, y )</code>	smaller value of $x$ and $y$	<code>min( 2.3, 12.7 )</code> is 2.3 <code>min( -2.3, -12.7 )</code> is -12.7
<code>pow( x, y )</code>	$x$ raised to the power $y$ (i.e., $x^y$ )	<code>pow( 2.0, 7.0 )</code> is 128.0 <code>pow( 9.0, 0.5 )</code> is 3.0
<code>sin( x )</code>	trigonometric sine of $x$ ( $x$ in radians)	<code>sin( 0.0 )</code> is 0.0
<code>sqrt( x )</code>	square root of $x$	<code>sqrt( 900.0 )</code> is 30.0
<code>tan( x )</code>	trigonometric tangent of $x$ ( $x$ in radians)	<code>tan( 0.0 )</code> is 0.0

# static Methods vs non-static Methods

- ▶ Unlike static method, to invoke a non-static method (instance method), we need to first create an instance of the class using the **new** keyword

```
import java.util.Random;

public class NumberGuessing {

    public static void main(String[] args) {

        Random random = new Random();
        int magicNum = random.nextInt(10);

    }

}
```

Normally, non-static methods  
are called on specific objects.

# static Methods vs non-static Methods

```
public static void main(String[] args) {  
    Scanner input = new Scanner(System.in);  
    System.out.print("enter three floating-point values: ");  
    double number1 = input.nextDouble();  
    double number2 = input.nextDouble();  
    double number3 = input.nextDouble();  
    double result = maximum(number1, number2, number3);  
    System.out.println("max is " + result);  
}
```

```
public static double maximum(double x, double y, double z) {  
    double max = x;  
    if(y > max) max = y;  
    if(z > max) max = z;  
    return max;  
}
```

Static methods in the same class  
can call each directly

# Why `main` method has to be static?

- ▶ The `main` method doesn't operate on any object
- ▶ In fact, when a program starts, there aren't any object yet.
- ▶ When you execute the Java Virtual Machine (JVM) with the `java` command, the JVM attempts to invoke the `main` method of the class you specify.
- ▶ Declaring `main` as `static` allows the JVM to invoke `main` without creating an object of the class



# Static methods in class Arrays

- ▶ Class `Arrays` helps you avoid reinventing the wheel by providing `static` methods for common array manipulations.
- ▶ These methods include `sort` for sorting an array (i.e., arranging elements into increasing order), `binarySearch` for searching an array (i.e., determining whether an array contains a specific value and, if so, where the value is located), `equals` for comparing arrays and `fill` for placing values into an array.
- ▶ These methods are overloaded for primitive-type arrays and for arrays of objects.
- ▶ You can copy arrays with class `System`'s `static` `arraycopy` method.



---

```
1 // Fig. 6.16: ArrayManipulations.java
2 // Arrays class methods and System.arraycopy.
3 import java.util.Arrays;
4
5 public class ArrayManipulations
6 {
7     public static void main( String[] args )
8     {
9         // sort doubleArray into ascending order
10        double[] doubleArray = { 8.4, 9.3, 0.2, 7.9, 3.4 };
11        Arrays.sort( doubleArray );
12        System.out.printf( "\ndoubleArray: " );
13
14        for ( double value : doubleArray )
15            System.out.printf( "%.1f ", value );
16
17        // fill 10-element array with 7s
18        int[] filledIntArray = new int[ 10 ];
19        Arrays.fill( filledIntArray, 7 );
20        displayArray( filledIntArray, "filledIntArray" );
21
22        // copy array intArray into array intArrayCopy
23        int[] intArray = { 1, 2, 3, 4, 5, 6 };
24        int[] intArrayCopy = new int[ intArray.length ];
```

---

**Fig. 6.16** | Arrays class methods. (Part I of 4.)

---

```
25 System.arraycopy( intArray, 0, intArrayCopy, 0, intArray.length );
26 displayArray( intArray, "intArray" );
27 displayArray( intArrayCopy, "intArrayCopy" );
28
29 // compare intArray and intArrayCopy for equality
30 boolean b = Arrays.equals( intArray, intArrayCopy );
31 System.out.printf( "\n\nintArray %s intArrayCopy\n",
32     ( b ? "==" : "!=" ) );
33
34 // compare intArray and filledIntArray for equality
35 b = Arrays.equals( intArray, filledIntArray );
36 System.out.printf( "intArray %s filledIntArray\n",
37     ( b ? "==" : "!=" ) );
38
39 // search intArray for the value 5
40 int location = Arrays.binarySearch( intArray, 5 );
41
42 if ( location >= 0 )
43     System.out.printf(
44         "Found 5 at element %d in intArray\n", location );
45 else
46     System.out.println( "5 not found in intArray" );
47
```

---

**Fig. 6.16** | Arrays class methods. (Part 2 of 4.)

---

```
48 // search intArray for the value 8763
49 location = Arrays.binarySearch( intArray, 8763 );
50
51 if ( location >= 0 )
52     System.out.printf(
53         "Found 8763 at element %d in intArray\n", location );
54 else
55     System.out.println( "8763 not found in intArray" );
56 } // end main
57
58 // output values in each array
59 public static void displayArray( int[] array, String description )
60 {
61     System.out.printf( "\n%s: ", description );
62
63     for ( int value : array )
64         System.out.printf( "%d ", value );
65 } // end method displayArray
66 } // end class ArrayManipulations
```

---

**Fig. 6.16** | Arrays class methods. (Part 3 of 4.)

```
doubleArray: 0.2 3.4 7.9 8.4 9.3  
filledIntArray: 7 7 7 7 7 7 7 7 7 7  
intArray: 1 2 3 4 5 6  
intArrayCopy: 1 2 3 4 5 6
```

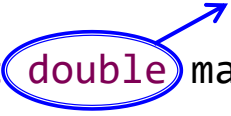
```
intArray == intArrayCopy  
intArray != filledIntArray  
Found 5 at element 4 in intArray  
8763 not found in intArray
```

**Fig. 6.16** | Arrays class methods. (Part 4 of 4.)

# Return Type of Methods

Return type: the type of data the method returns to its caller.

```
public static double maximum(double x, double y, double z) {  
    double max = x;  
    if(y > max) max = y;  
    if(z > max) max = z;  
    return max;  
}
```



A method may return

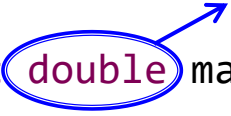
- **Nothing** (void, simply returning control back)
- **Primitive values** (e.g., an integer)
- **References** to objects, arrays

```
public void println(  
    @Nullable String x  
)
```

# Return Type of Methods

Return type: the type of data the method returns to its caller.

```
public static double maximum(double x, double y, double z) {  
    double max = x;  
    if(y > max) max = y;  
    if(z > max) max = z;  
    return max;  
}
```



If the method returns a result, the statement

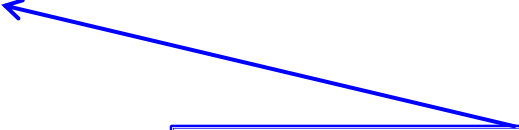
`return expression;`

evaluates the *expression*, then returns the result to the caller

```
return number1 + number2 + number3;
```

# Method Parameters

A comma-separated list of **parameters**, meaning that the method requires additional information from the caller to perform its task.



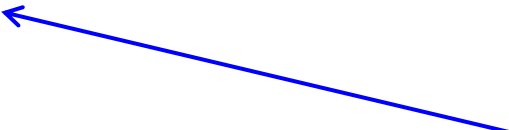
```
public static double maximum( double x, double y, double z ) {  
    double max = x;  
    if(y > max) max = y;  
    if(z > max) max = z;  
    return max;  
}
```

**Empty parentheses:** the method does not need additional information to perform its task



# Method Parameters

Each parameter must specify a **type** and an **identifier**



```
public static double maximum( double x, double y, double z ) {  
    double max = x;  
    if(y > max) max = y;  
    if(z > max) max = z;  
    return max;  
}
```

!!! A method's parameters are considered to be **local variables** of that method and can be used only in that method's body

# Method Parameters

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

```
    Scanner input = new Scanner(System.in);
```

```
    System.out.print("enter three floating-point values: ");
```

```
    double number1 = input.nextDouble();
```

```
    double number2 = input.nextDouble();
```

```
    double number3 = input.nextDouble();
```

```
    double result = maximum(number1, number2, number3);
```

```
    System.out.println("max is " + result);
```

```
}
```

A method call supplies arguments for each of the method's parameters

```
public static double maximum( double x, double y, double z ) {
```

```
    ...
```

```
}
```

One to one correspondence and the type must be consistent.

# Method Parameters

- ▶ Before any method can be called, its arguments must be evaluated to determine their values
- ▶ If an argument is a method call, the method call must be performed to determine its return value

```
Math.pow( Math.pow(x2-x1, 2) + Math.pow(y2-y1, 2) , 0.5 );
```



First argument

# Java API Packages

- ▶ Java contains many predefined classes that are grouped into categories of related classes called packages
  - `java.lang` (e.g., `java.lang.Math`) java.lang is the default package, no need to import it to access its classes
  - `java.io`
  - `java.net` To use the classes of other packages, we need to explicitly import the Class
  - `java.util` (e.g., `java.util.Random`, `java.util.Scanner`)
- ▶ Overview of the packages in Java SE
  - <https://docs.oracle.com/javase/8/docs/api/overview-summary.html>

# Importing Classes

```
import java.util.Scanner;
```

```
public class MainClass {
```

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

Scanner input = new Scanner(System.in);

Or using fully qualified method name (全名)

```
    int number1 = input.nextInt();
```

```
    int number2 = new java.util.Random().nextInt(10);
```

```
    double result = Math.max(number1, number2);
```

```
    input.close();
```

```
}
```

```
}
```

No need to specify the package name java.lang

# Objectives

- ▶ What is modular programming (模块化编程)
- ▶ How to declare and use methods
- ▶ Method overloading
- ▶ Passing arguments
- ▶ Method call stack

# Method Overloading (方法重载)

- ▶ Methods of the same name can be declared in the same class, as long as they have different sets of parameters
- ▶ Used to create several methods that perform the same/similar tasks on **different types** or **different numbers** of arguments
  - **Java compiler** selects the appropriate method to call by examining the number, types and order of the arguments in the call

static double	max(double a, double b)
static float	max(float a, float b)
static int	max(int a, int b)
static long	max(long a, long b)

```
int a = 2;  
int b = 3;  
Math.max(a, b);
```

**Which version of max?**

# Method Overloading

- ▶ Compiler distinguishes overloaded methods by their **signature**
  - A combination of the method's name and the number, types and order of its parameters.

```
public double calculateAnswer(double wingSpan, int numberOfEngines,  
                             double length, double grossTons) {  
    //do the calculation here  
    return 0.0;  
}
```

**Signature:** calculateAnswer(double, int, double, double)



# Method Overloading

- ▶ Method calls cannot be distinguished by return type. If you have overloaded methods only with different return types:
  - `int square(int a)`
  - `double square(int a)`
- ▶ and you called the method as follows
  - `square(2);`
- ▶ the compiler will be confused (since return value ignored)
- ▶ Hence, there will be compilation error for defining multiple methods with the same signature

# Method Overloading

- ▶ `System.out.println` are also overloaded

```
println()
```

Terminates the current line by writing the line separator string.

```
println(boolean x)
```

Prints a boolean and then terminate the line.

```
println(char x)
```

Prints a character and then terminate the line.

```
println(char[] x)
```

Prints an array of characters and then terminate the line.

```
println(double x)
```

Prints a double and then terminate the line.

```
println(float x)
```

Prints a float and then terminate the line.

```
println(int x)
```

Prints an integer and then terminate the line.

```
println(long x)
```

Prints a long and then terminate the line.

```
println(Object x)
```

Prints an Object and then terminate the line.

```
println(String x)
```

Prints a String and then terminate the line.

# Objectives

- ▶ What is modular programming (模块化编程)
- ▶ How to declare and use methods
- ▶ Method overloading
- ▶ Passing arguments
- ▶ Method call stack

# Passing Arguments in Method Calls

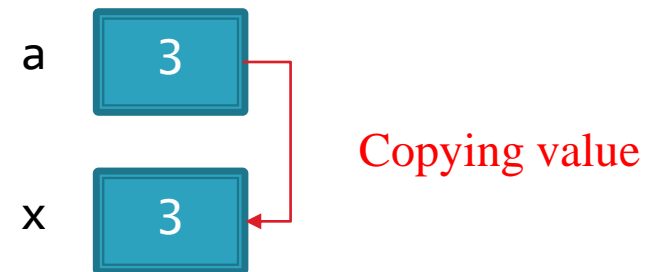
- ▶ In Java, all arguments are **passed by value**, meaning that the method gets **a copy of all parameter values**
- ▶ A method call can pass two types of values to the called method:
  - Primitive types: passing copies of primitive values
  - Reference types: passing copies of references to objects.

# Passing Primitive Type

The value is simply copied

```
public static void main(String[] args) {  
    int a = 3;  
    System.out.println("Before: " + a);  
  
    triple(a);  
    System.out.println("After: " + a);  
}  
  
public static void triple(int x) {  
    x *= 3;  
}
```

Before: 3  
After: 3



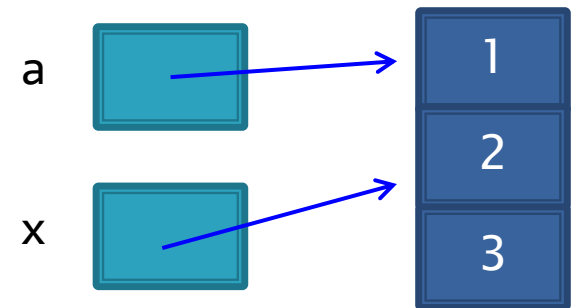
x becomes 9 after method call  
a remains unchanged

# Passing Reference Type

```
public static void main(String[] args) {  
    int[] a = {1, 2, 3};  
    System.out.println("Before: " + Arrays.toString(a));  
  
    triple(a);  
    System.out.println("After: " + Arrays.toString(a));  
}
```

Before: [1, 2, 3]  
After: [3, 6, 9]

```
public static void triple(int[] x) {  
    for(int i = 0; i < x.length; i++)  
        x[i] *= 3;  
}
```



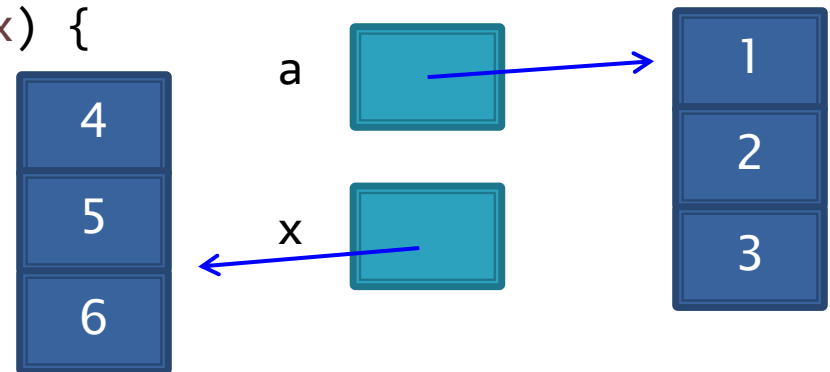
Copying value again. Difference is that the value is a memory address.

# Passing Reference Type

```
public static void main(String[] args) {  
    int[] a = {1, 2, 3};  
    System.out.println("Before: " + Arrays.toString(a));  
  
    triple(a);  
    System.out.println("After: " + Arrays.toString(a));  
}
```

Before: [1, 2, 3]  
After: [1, 2, 3]

```
public static void triple(int[] x) {  
    x = new int[]{4,5,6};  
}
```



Copying value, which is a memory address.

# What's the output?

```
public static void main(String[] args) {  
    int[] arr = {1, 2, 3};  
    System.out.println("Before: " + Arrays.toString(a));  
  
    triple(arr[0]);  
    System.out.println("After: " + Arrays.toString(a));  
}  
  
public static void triple(int x) {  
    x *= 3;  
}
```



# Passing Arguments in Method Calls

- ▶ **Passing primitive types:** the method cannot modify the content of the parameter variables passed to it
  - Previous example: should return  $x*3$ , then `a = triple(a)`
- ▶ **Passing reference types:** although an object's reference is passed by value, a method can still interact with the referenced object using the copy of the object's reference (arrays are also objects).
  - The parameter in the called method and the argument in the calling method refer to the same object in memory.

# Argument Promotion

- ▶ **Argument promotion**—converting an argument's value, if possible, to the type that the method expects to receive in its corresponding parameter

`Math.sqrt()` expects to receives a **double** argument, but it is ok to write `Math.sqrt(4)`: java converts the **int** value 4 to the **double** value 4.0

# Promotion Rules

Specify which conversions are allowed (which conversions can be performed without losing data)

Type	Valid promotions
double	None
float	double
long	float or double
int	long, float or double
char	int, long, float or double
short	int, long, float or double (but not char)
byte	short, int, long, float or double (but not char)
boolean	None (boolean values are not considered to be numbers in Java)

# Promotion Rules

Besides arguments passed to methods, the rules also apply to expressions containing values of two or more primitive types

`2 * 2.0` becomes `4.0`

```
int x = 2;  
double y = x * 2.0;  
// is x 2.0 or 2 now?
```

**Answer:** x is still of `int` type, the expression uses a temporary copy of x's value for promotion

# Variable-Length Argument Lists

- ▶ With **variable-length argument lists**, you can create methods that receive an unspecified number of arguments.
- ▶ A type followed by an **ellipsis (...)** in a method's parameter list indicates that the method receives a variable number of arguments of that particular type.
  - `public static double average(double... numbers)`
  - Can occur only once in a parameter list, and the ellipsis, together with its type, must be placed at the end of the parameter list.
- ▶ Java treats the variable-length argument list as an array of the specified type.

# Example

```
public static double average(double... numbers) {  
    double total = 0.0;  
    for(double d : numbers) total += d;  
    return total / numbers.length;  
}
```

```
numbers = {double[2]@484} [10.0, 20.0]
```

```
numbers = {double[3]@609} [10.0, 20.0, 30.0]
```

```
public static void main(String[] args) {  
    double d1 = 10.0, d2 = 20.0, d3 = 30.0;  
    System.out.printf("average of d1 and d2: %f\n", average(d1, d2));  
    System.out.printf("average of d1 ~ d3: %f\n", average(d1, d2, d3));  
}
```

```
average of d1 and d2: 15.000000  
average of d1 ~ d3: 20.000000
```

# Objectives

- ▶ What is modular programming (模块化编程)
- ▶ How to declare and use methods
- ▶ Method overloading
- ▶ Passing arguments
- ▶ **Method call stack**

# Method-Call Stack (方法调用栈)



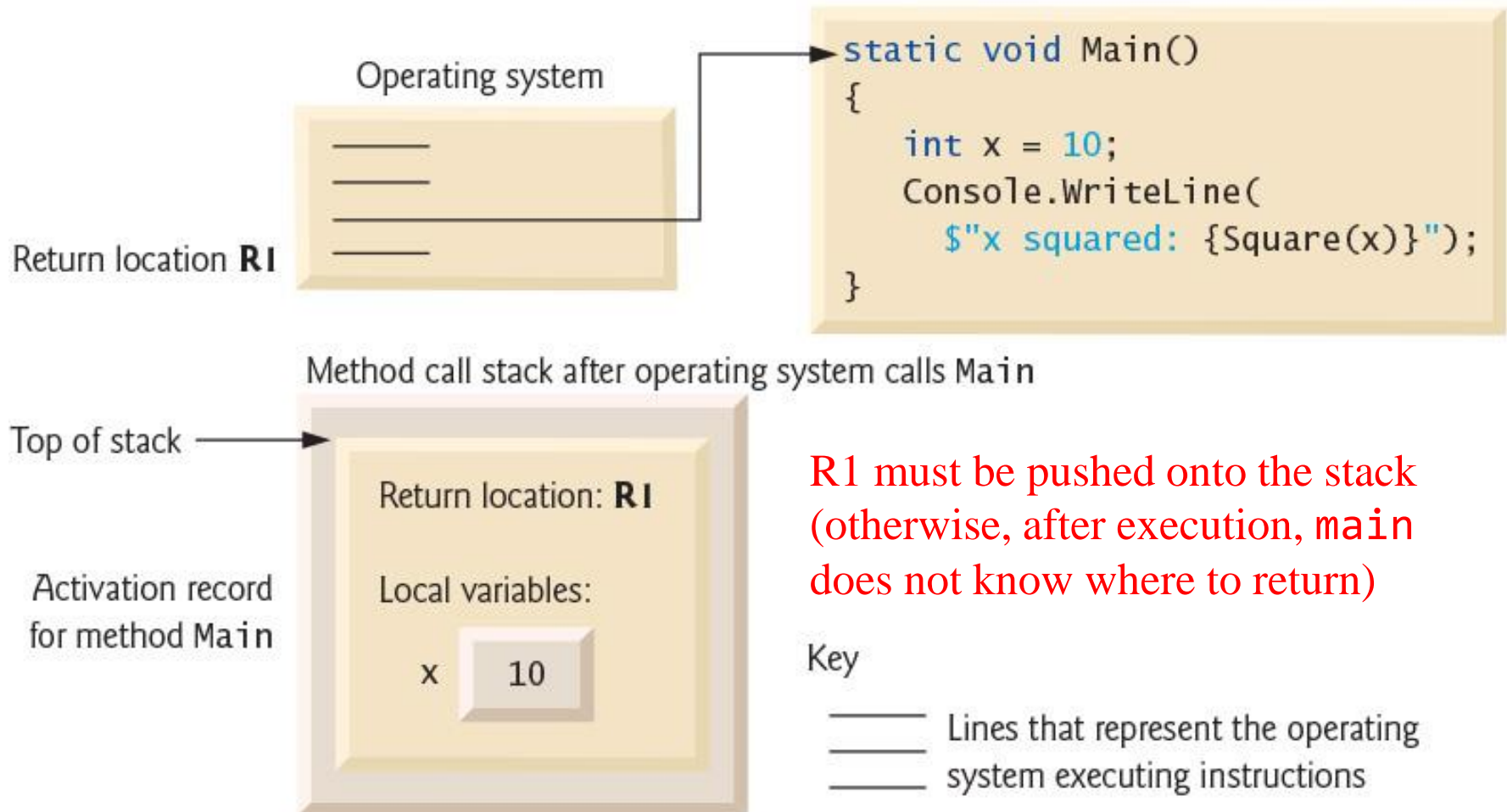
- ▶ **Stack** data structure: analogous to a pile of dishes
  - When a dish is placed on the pile, it's normally placed at the top (referred to as **pushing** onto the stack)
  - Similarly, when a dish is removed from the pile, it's always removed from the top (referred to as **popping** off the stack)
- ▶ **Last-in, first-out (LIFO)** — the last item pushed (inserted) on the stack is the first item popped (removed) from the stack (also **first in last out**)



# Method-Call Stack

- ▶ When a program calls a method, the called method must know how to return to its caller, so the **return address** of the calling method (**next instruction to execute** after method execution) is pushed onto the **method-call stack** (also known as program execution stack)

Step 1: Operating system calls Main to begin program execution



# Method-Call Stack

- ▶ If a series of method calls occurs, the successive return addresses are pushed onto the stack in last-in, first-out order
- ▶ The program-execution stack also contains the memory for the local variables used in each invocation of a method
  - Stored in the **activation record** (or **stack frame**) of the method call
  - When a method call is made, the activation record for that method call is pushed onto the method-call stack

Step 2: Main calls method Square to perform calculation

Return location **R2**

```
static void Main()
{
    int x = 10;
    Console.WriteLine(
        $"x squared: {Square(x)}");
}
```

```
static int Square(int y)
{
    return y * y;
}
```

Method call stack after Main calls Square

Top of stack →

Return location: **R2**

Local variables:

y 10

Activation record for  
method Square

Return location: **R1**

Local variables:

x 10

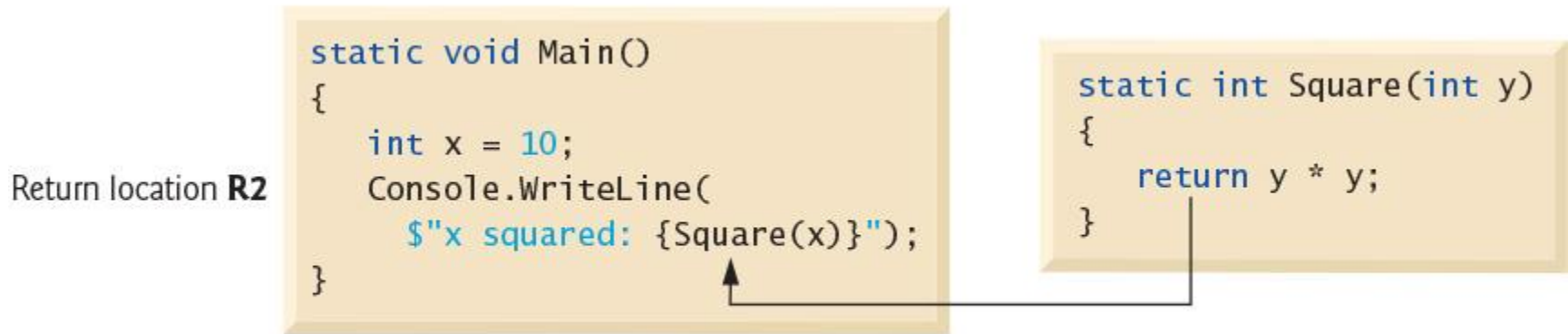
Activation record  
for method Main

**Note: The activation record of method square is at top of the stack.**

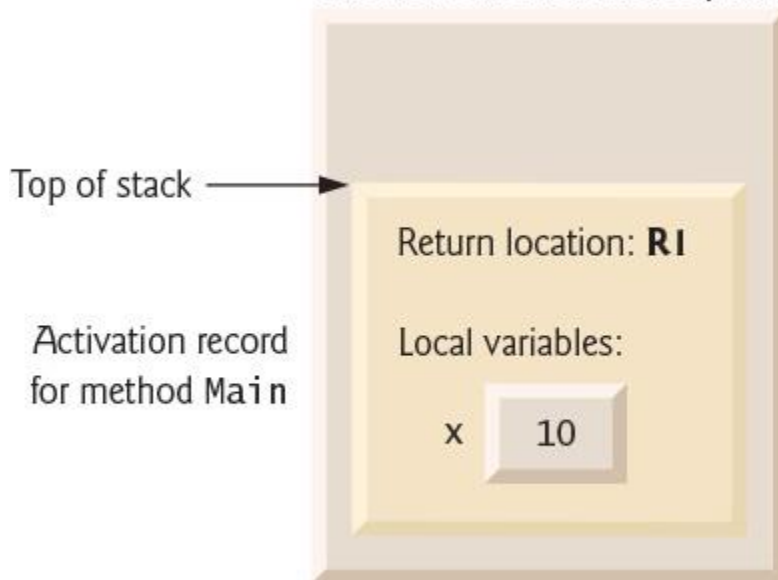
# Method-Call Stack

- ▶ When a method returns to its caller, the activation record for the method call is **popped off** the stack and those local variables are no longer known to the program

Step 3: Square returns its result to Main



Method call stack after Square returns its result to Main



The activation record for the square method call is popped off

Local variable `y` is not visible anymore, its memory will be released for other uses

# Inspecting the Call Stack

```
public class MainClass {  
    public static void main(String[] args) {  
        int x = 10;  
        caller1(x);  
    }  
  
    public static int caller1(int y) {  
        return caller2(y);  
    }  
  
    public static int caller2(int y) {  
        return caller3(y);  
    }  
  
    public static int caller3(int y) {  
        return y/0;  
    }  
}
```

```
caller3:18, MainClass1 (c  
caller2:14, MainClass1 (c  
caller1:10, MainClass1 (c  
main:6, MainClass1 (con
```

```
Exception in thread "main" java.lang.ArithmeticException Create breakpoint : / by zero  
    at com.company.MainClass1.caller3(MainClass1.java:18)  
    at com.company.MainClass1.caller2(MainClass1.java:14)  
    at com.company.MainClass1.caller1(MainClass1.java:10)  
    at com.company.MainClass1.main(MainClass1.java:6)
```

# Summary: Why Use Methods?

## ► For reusable code, reducing code duplication

- If you need to do the same thing many times, write a method to do it, then call the method each time you have to do that task.

## ► To parameterize code

- You will often use parameters that change the way the method works.

## ► For top-down programming (divide and conquer)

- You solve a big problem (the "top") by breaking it down into small problems. To do this in a program, you write a method for solving your big problem by calling other methods to solve the smaller parts of the problem, which similarly call other methods until you get down to simple methods that solve simple problems.

<https://www.leepoint.net/JavaBasics/methods/method-commentary/methcom-purpose.html>



# Summary: Why Use Methods?

## ▶ To create conceptual units

- Create methods to do something that is *one action* in your mental view of the problem. This will make it much easier for you to program.

## ▶ To simplify

- Because local variables and statements of a method can not be seen from outside the method, *they (and their complexity) are hidden* from other parts of the program, which prevents accidental errors or confusion (e.g., random number generation method)

## ▶ To ease debugging and maintenance

- You don't want to debug a main method with 100K lines of code