Advanced Object-Oriented Programming

Fundamental Programming Structures in Java

Dr. Kulwadee Somboonviwat

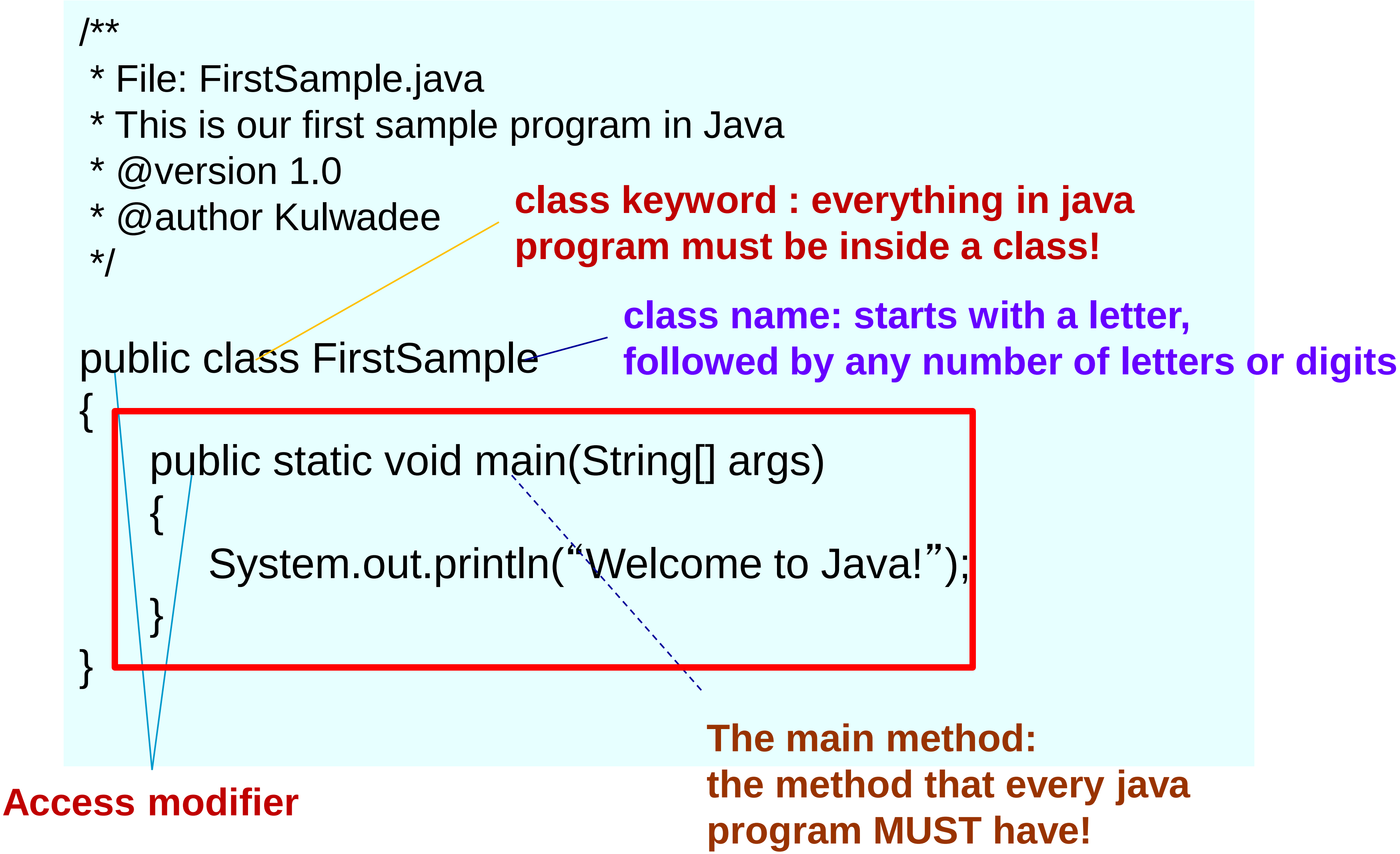
International College, KMITL

kskulwad@kmitl.ac.th

Fundamental Programming Structures

- A Simple Java Program
- Comments
- Data Types
- Variables
- Operators
- Strings
- Input and Output
- Control Flow
- Arrays
- Methods

A Simple Java Program



```
/**
 * File: FirstSample.java
 * This is our first sample program in Java
 * @version 1.0
 * @author Kulwadee
 */

public class FirstSample
{
    public static void main(String[] args)
    {
        System.out.println(“Welcome to Java!”);
    }
}
```

class keyword : everything in java program must be inside a class!

class name: starts with a letter, followed by any number of letters or digits

The main method: the method that every java program MUST have!

Access modifier

A Simple Java Program : output a line of message to console

```
System.out.println( “Welcome to Java! ”);
```

```
Object.method(parameters)
```

Now.. Let’ s compile and run our first program!

```
C:\> javac FirstSample.java

C:\> dir FirstSample.*

FirstSample.java      FirstSample.class

C:\> java FirstSample

Welcome to Java!
```

Fundamental Programming Structures

- A Simple Java Program
- Comments
- Data Types
- Variables
- Operators
- Strings
- Input and Output
- Control Flow
- Arrays
- Methods

Comments

- Comments do not show up in the executable program
  - Single-line comment delimiter: //
  - Multi-line comment delimiter: /* and */
  - Javadoc comment delimiter: /** and */

    * This type of comment is used in automatic document generation

Fundamental Programming Structures

- A Simple Java Program
- Comments
- Data Types
- Variables
- Operators
- Strings
- Input and Output
- Control Flow
- Arrays
- Methods

Data Types

- Java is a strongly typed language.
  - Every variable must have a declared type
- Eight primitive types in Java
  - 4 integer types
  - 2 floating-point types
  - 1 character type
  - 1 boolean type

Primitive Data Types (1/3)

Type	Description	Size
int	The integer type, with range -2,147,483,648 . . . 2,147,483,647	4 bytes
byte	The type describing a single byte, with range -128 . . . 127	1 byte
short	The short integer type, with range -32768 . . . 32767	2 bytes
long	The long integer type, with range -9,223,372,036,854,775,808 . . . 9,223,372,036,854,775,807	8 bytes

Primitive Data Types (2/3)

Type	**Description**	**Size**
double	The double-precision floating-point type, with a range of about ±10^{308} and about 15 significant decimal digits	8 bytes
float	The single-precision floating-point type, with a range of about ±10^{38} and about 7 significant decimal digits	4 bytes

Primitive Data Types (3/3)

Type	**Description**	**Size**
char	The character type, representing code units in the **Unicode encoding scheme**	2 bytes
boolean	The type with the two truth values false and true	1 bit

Fundamental Programming Structures

- A Simple Java Program
- Comments
- Data Types
- Variables
- Operators
- Strings
- Input and Output
- Control Flow
- Arrays
- Methods

Types and Variables

Syntax 2.1: Variable Definition

typeName variableName = value;

or

typeName variableName;

Example:

```
String greeting = "Hello, AOOP!";
double salary = 65000.0;
```

Purpose:

To define a new variable of a particular type and optionally supply an initial value

Identifiers

- Identifier: **name of a variable, method, or class**
- Rules for identifiers in Java:
  - Can be made up of letters, digits, and the underscore (_) character
  - Cannot start with a digit
  - Cannot use other symbols such as ? or %
  - Spaces are not permitted inside identifiers
  - You cannot use reserved words
  - They are **case sensitive**
- **Convention:**
  - variable names start with a lowercase letter
  - class names start with an uppercase letter

Number Types

- **int**: integers, no fractional part

  1, -4, 0

- **double**: floating-point numbers (double precision)

  0.5, -3.11111, 4.3E24, 1E-14

- A numeric computation *overflows* if the result falls outside the range for the number type

```
int n = 1000000;
System.out.println(n * n); // prints -727379968
```

Number Types: Floating-point

- Rounding errors occur when an exact conversion between numbers is not possible

```
double f = 4.35;
System.out.println(100 * f); // prints 434.99999999999994
```

- Java: Illegal to assign a floating-point expression to an integer variable

```
double balance = 13.75;
int dollars = balance; // Error
```

- **Casts: used to convert a value to a different type**

```
int dollars = (int) balance; // OK
```

  - Cast discards fractional part.
  - Math.round converts a floating-point number to nearest integer

```
long rounded = Math.round(balance);
// if balance is 13.75, then rounded is set to 14
```

Cast

Cast: used to convert a value to a different type

→ discard fractional part

Syntax 2.2: Cast

(typeName) expression

Example:

(int) (balance * 100)

Purpose:

To convert an expression to a different type

Constants: final

- A final variable is a constant
- Once its value has been set, it cannot be changed
- Named constants make programs easier to read and maintain
- Convention: use all-uppercase names for constants

```
final double QUARTER_VALUE = 0.25;
final double DIME_VALUE = 0.1;
final double NICKEL_VALUE = 0.05;
final double PENNY_VALUE = 0.01;

payment = dollars + quarters * QUARTER_VALUE +
            dimes * DIME_VALUE +
            nickels * NICKEL_VALUE +
            pennies * PENNY_VALUE;
```

Constants: static final

- If constant values are needed in several methods, declare them together with the instance fields of a class and tag them as static and final
- Give static final constants public access to enable other classes to use them

```
public class Math
{
. . .
public static final double E = 2.7182818284590452354;
public static final double PI = 3.14159265358979323846;
}


double circumference = Math.PI * diameter;
```

Constant Definition

Syntax 2.3: Constants

In a method:

final *typeName variableName* = *expression* ;

In a class:

accessSpecifier static final *typeName variableName* = *expression*;

Example:

final double NICKEL_VALUE = 0.05;
public static final double LITERS_PER_GALLON = 3.785;

Purpose:

To define a constant in a method or a class

Fundamental Programming Structures

- A Simple Java Program
- Comments
- Data Types
- Variables
- Operators
- Strings
- Input and Output
- Control Flow
- Arrays
- Methods

Operators

- Assignment (=), Increment (++), Decrement (--)
- Arithmetic Operators

  + - * / %

- Relational Operators

  < <= > >= == !=

- Logical Operators

  ! && || ^

Assignment, Increment, Decrement

- Assignment **is not the same as mathematical equality**:

  items = items + 1;
- Increment

  items++ is the same as items = items + 1
- Decrement

  items-- subtracts 1 from items

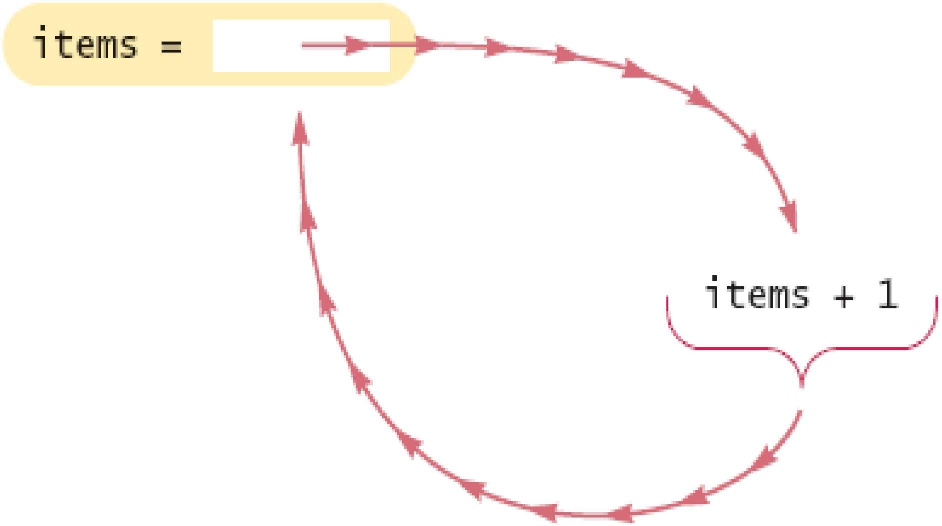


Figure 1

Incrementing a Variable

Arithmetic Operations

- / is the division operator

  If both arguments are integers, the result is an integer.
  The remainder is discarded

  7.0 / 4 yields 1.75
  7 / 4 yields 1

- Get the remainder with % (pronounced "modulo")

  7 % 4 is 3

The Math class

- Math class: contains methods like sqrt and pow
- To compute x^n, you write Math.pow(x, n)
- To take the square root of a number, use the Math.sqrt; for example, Math.sqrt(x)

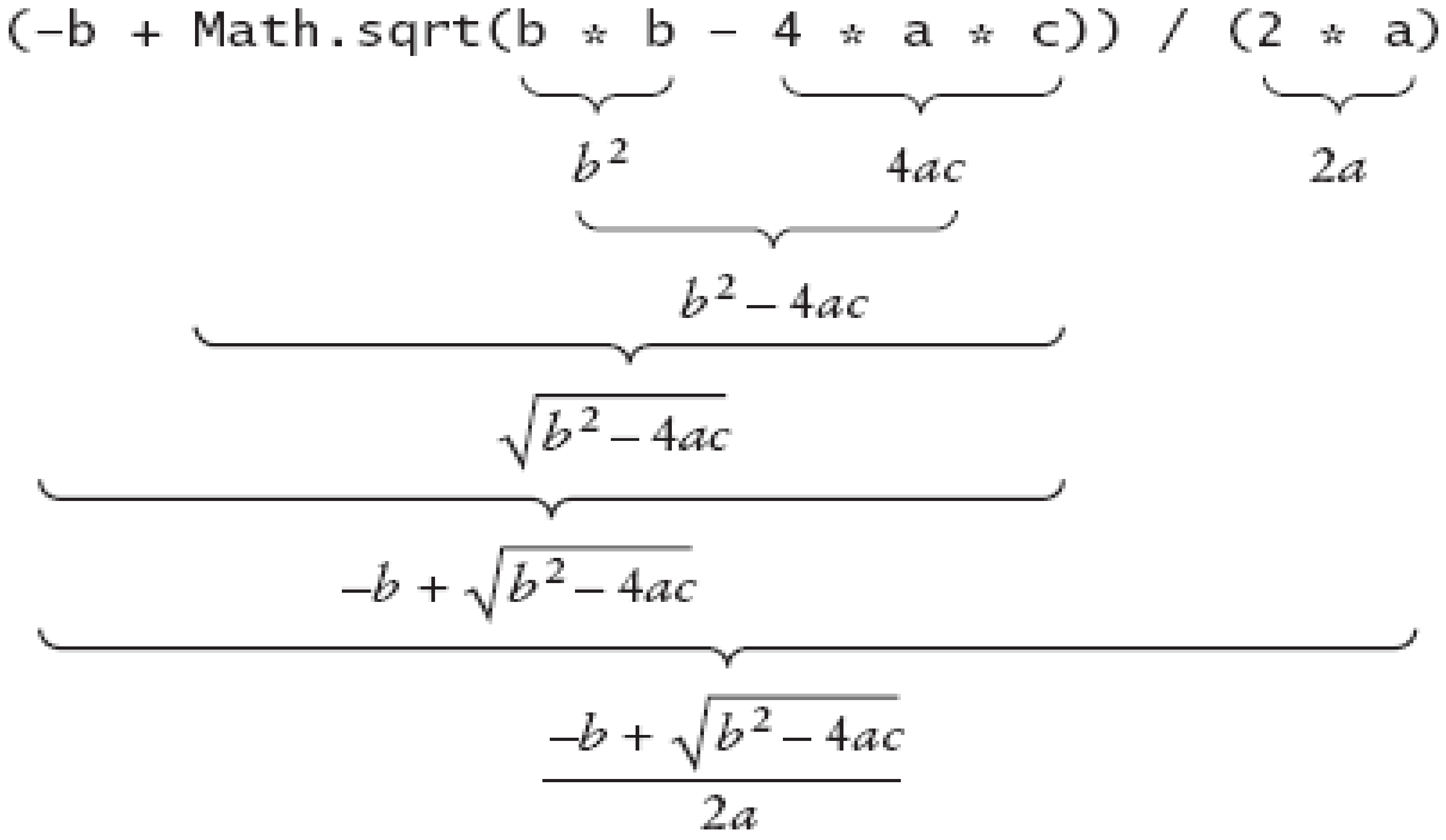
- In Java,

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

can be represented as

(-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a)

The Math class



```
(-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a)
```

Figure 2 Analyzing an Expression

Mathematical Methods in Java

Math.sqrt(x)	square root
Math.pow(x, y)	power x^y
Math.exp(x)	e^x
Math.log(x)	natural log
Math.sin(x), Math.cos(x), Math.tan(x)	sine, cosine, tangent (x in radian)
Math.round(x)	closest integer to x
Math.min(x, y), Math.max(x, y)	minimum, maximum

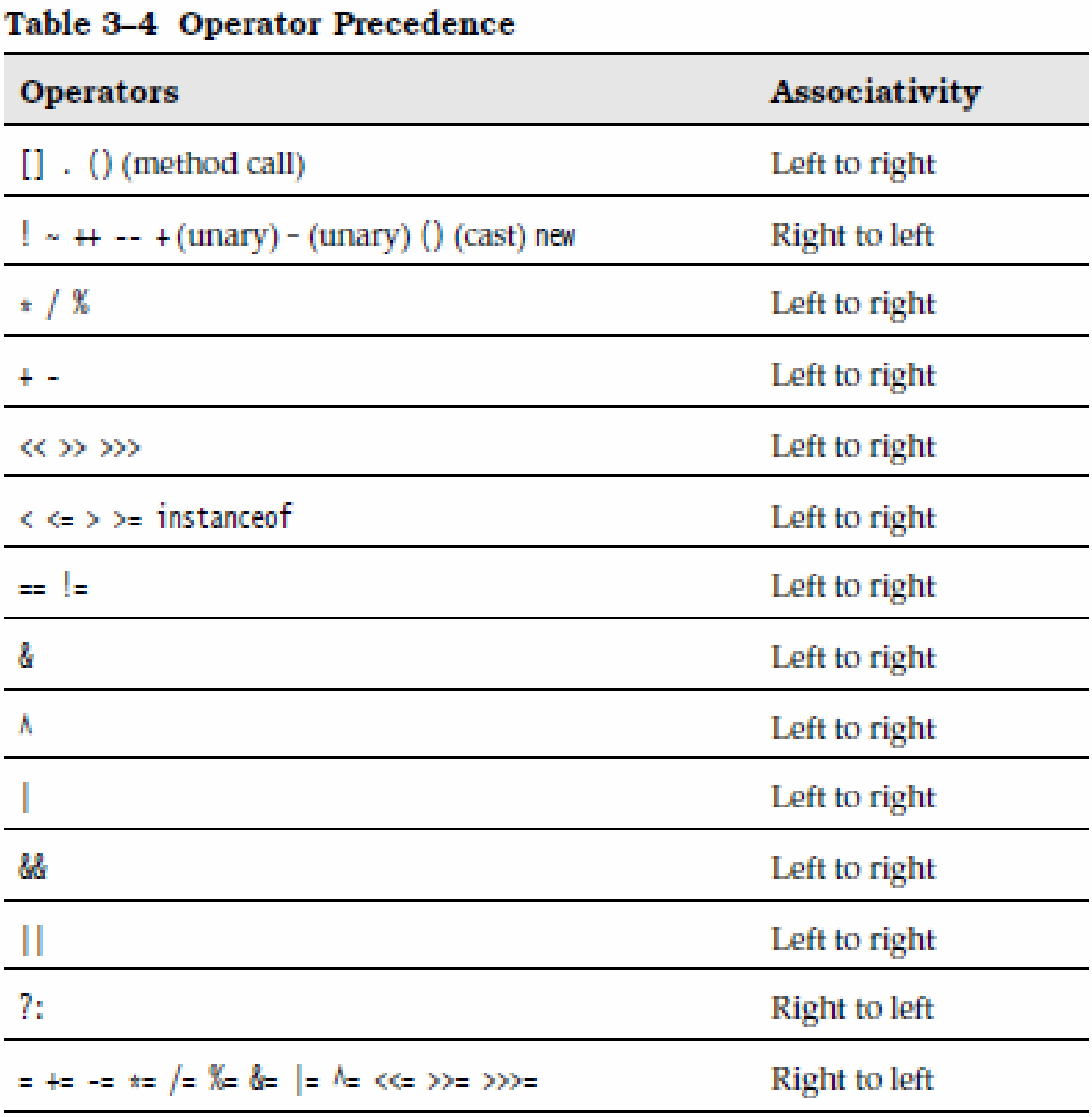


Table 3–4 Operator Precedence

Operators	Associativity
[] . () (method call)	Left to right
! ~ ++ -- + (unary) - (unary) () (cast) new	Right to left
* / %	Left to right
+ -	Left to right
<< >> >>>	Left to right
< <= > >= instanceof	Left to right
== !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
?:	Right to left
= += -= *= /= %= &= |= ^= <<= >>= >>>=	Right to left

Fundamental Programming Structures

- A Simple Java Program
- Comments
- Data Types
- Variables
- Operators
- Strings
- Input and Output
- Control Flow
- Arrays
- Methods

String

- A string is a sequence of characters
- Strings are objects of the String class
- String constants:

  "Hello, World!"

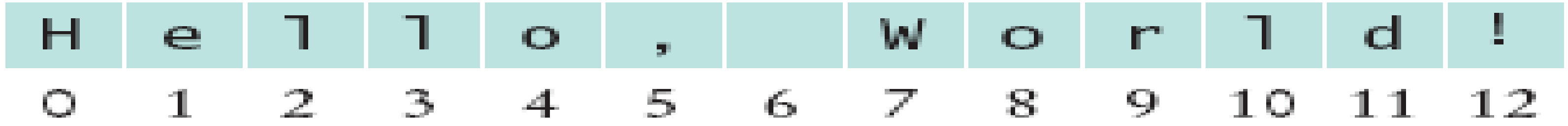
- String variables:

  String message = "Hello, World!";

- String length:

  int n = message.length();

- Empty string: ""



H	e	l	l	o	,		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11	12

Figure 3 String Positions

String Operations (1)

- **Concatenation**
  - Use the + operator:

```
String name = "Dave";
String message = "Hello, " + name;
// message is "Hello, Dave"
```

  - If one of the arguments of the + operator is a string, the other is converted to a stringString a = "Agent";

```
int n = 7;
String bond = a + n; // bond is Agent7
```

String Operations (2)

- **Substring**

```
String greeting = "Hello, World!";
String sub = greeting.substring(0, 5);
// sub is "Hello“
```

  - Supply start and “past the end” position

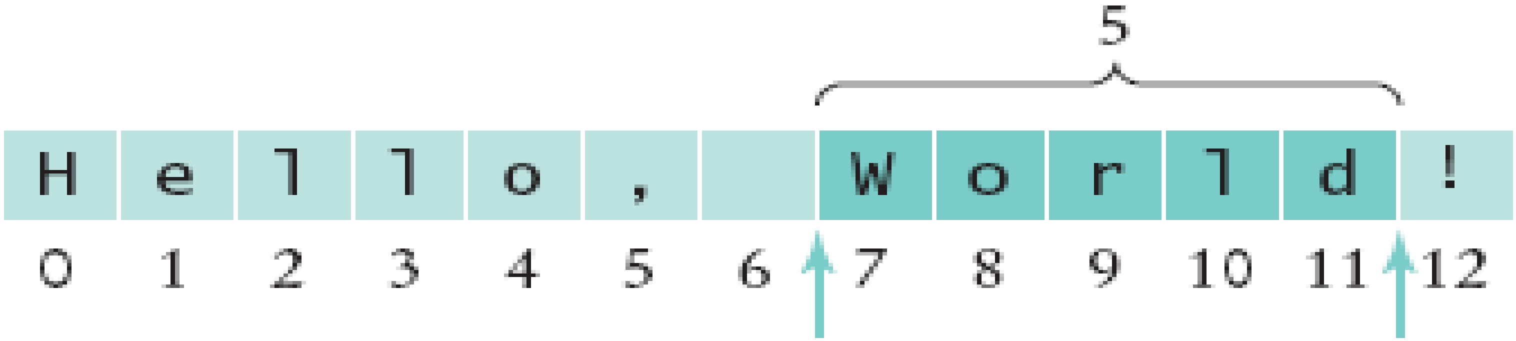


Figure 4 Extracting a Substring

String Operations (3)

- **Testing Strings for Equality**

  - use the **equals** method

    s.equals(t)

  - Do not use == to test if two strings are equal!!

  it only determines if the strings are stored in the same location or not.

```
String greeting = “hello”;
if (greeting.equals(“hello”))
{
  System.out.println(“they are equal!”);
}
else
{
  System.out.println(“they aren’t equal!”);
}
```

```
String greeting = “hello”;
if (greeting == “hello”))
{
      // probably true
}
```

Fundamental Programming Structures

- A Simple Java Program
- Comments
- Data Types
- Variables
- Operators
- Strings
- Input and Output
- Control Flow
- Arrays
- Methods

Writing Output

- for simple stand-alone java program,

```
System.out.println(data)
```

System.out (standard output) :
a static ***PrintStream*** object declared in class System (java.lang.System)

println method
Print an object (i.e. data) to the standard output stream

Reading Input

- System.in has minimal set of features–it can only read one byte at a time
- In Java 5.0, Scanner class was added to read keyboard input in a convenient manner

```
import java.util.Scanner;
Scanner in = new Scanner(System.in);
System.out.print("Enter quantity: ");
int quantity = in.nextInt();
```

Note:

nextDouble reads a double

nextLine reads a line (until user hits Enter)

nextWord reads a word (until any white space)

Fundamental Programming Structures

- A Simple Java Program
- Comments
- Data Types
- Variables
- Operators
- Strings
- Input and Output
- Control Flow
- Arrays
- Methods

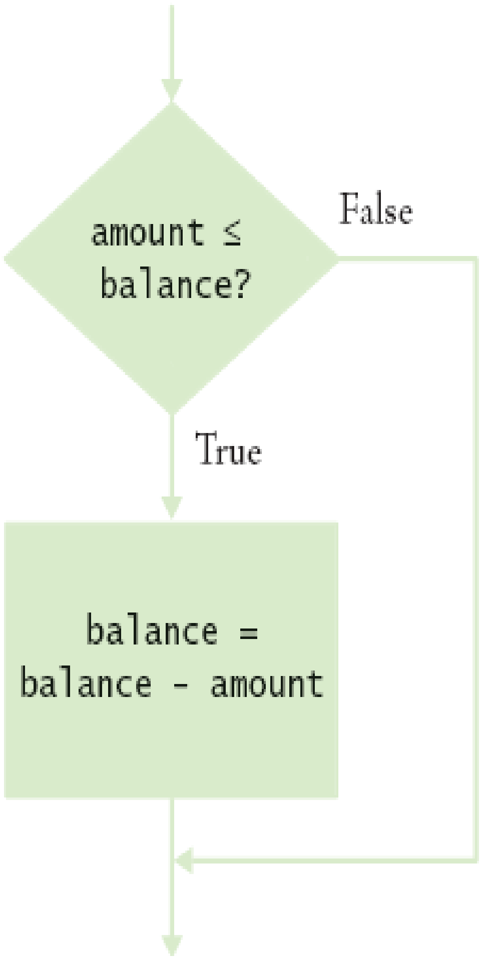
Control Structures

- Java supports both **conditional statements** and **loops** to determine the control flow of a program

  - Conditional statements
    - If-statement
    - Switch-statement
  - Loops
    - While-statement
    - Do-While-statement
    - For-statement

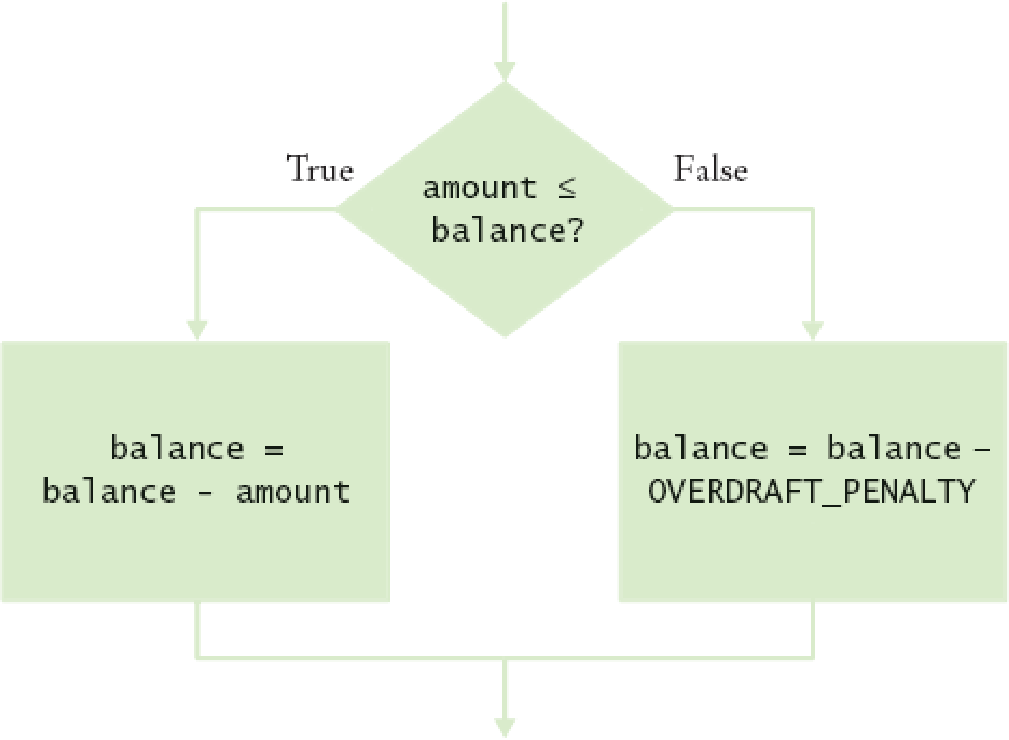
Decisions

- if statement



Decisions

- if/else statement


True
amount ≤ balance?
False
balance = balance - amount
balance = balance − OVERDRAFT_PENALTY

if statement

Syntax 2.4: if statement

```
if (condition)
{
  statement
}
```

```
if (condition)
{
  statement1
}
else
{
  statement2
}
```

Example:

```
if (amount <= balance) balance = balance - amount;
if (amount <= balance)
          balance = balance - amount;
else
          balance = balance - OVERDRAFT_PENALTY;
```

Purpose:

To execute a statement when a condition is true or false

Exercise: implement this loop in Java

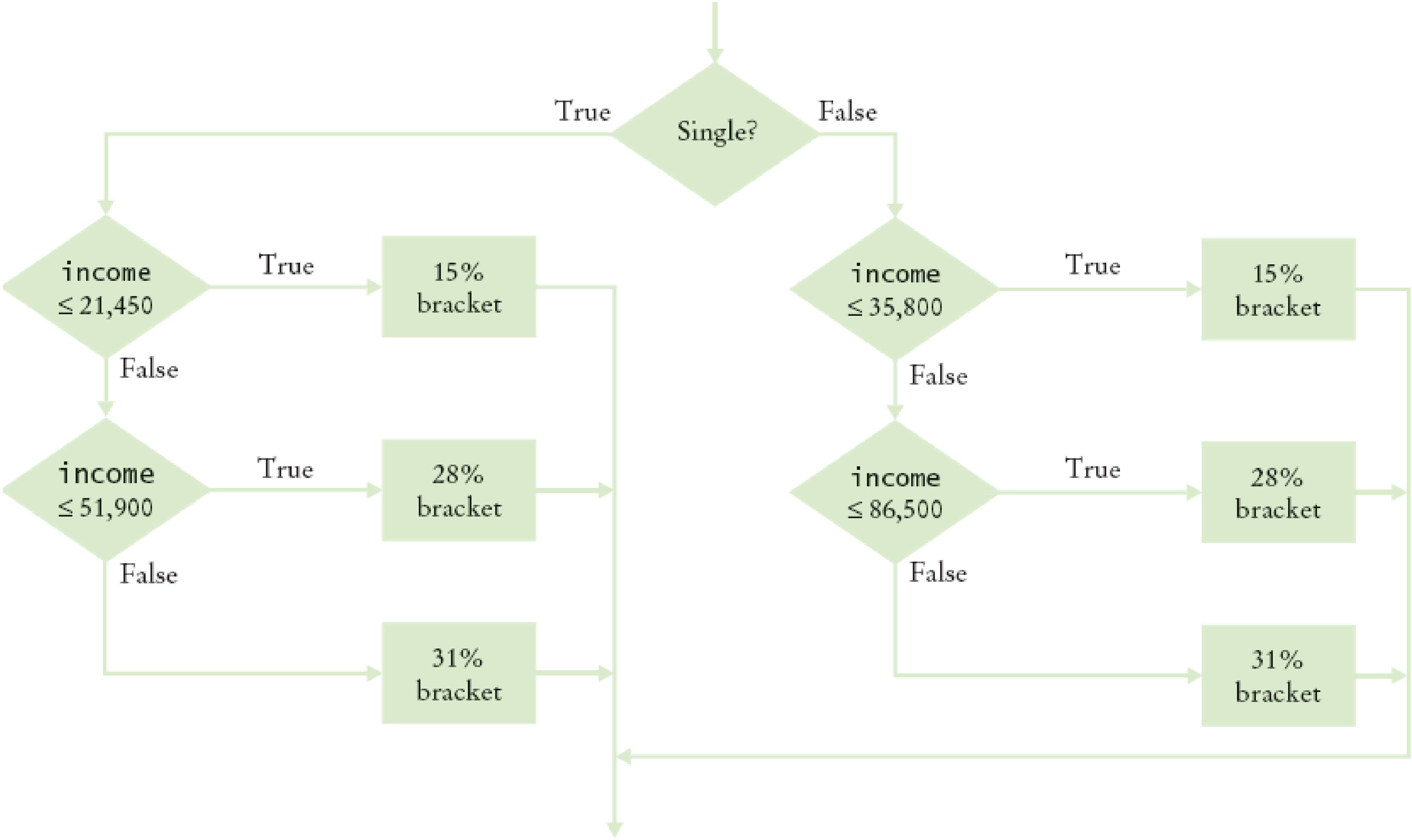


Figure 5 Income Tax Computation Using 1992 Schedule

while loop

- Executes a block of code repeatedly
- A condition controls how often the loop is executed

```
while (condition)
        statement;
```

- Most commonly, the statement is a block statement (set of statements delimited by { })

while loop

Calculating the Growth of an Investment
Invest $10,000, 5% interest, compounded annually

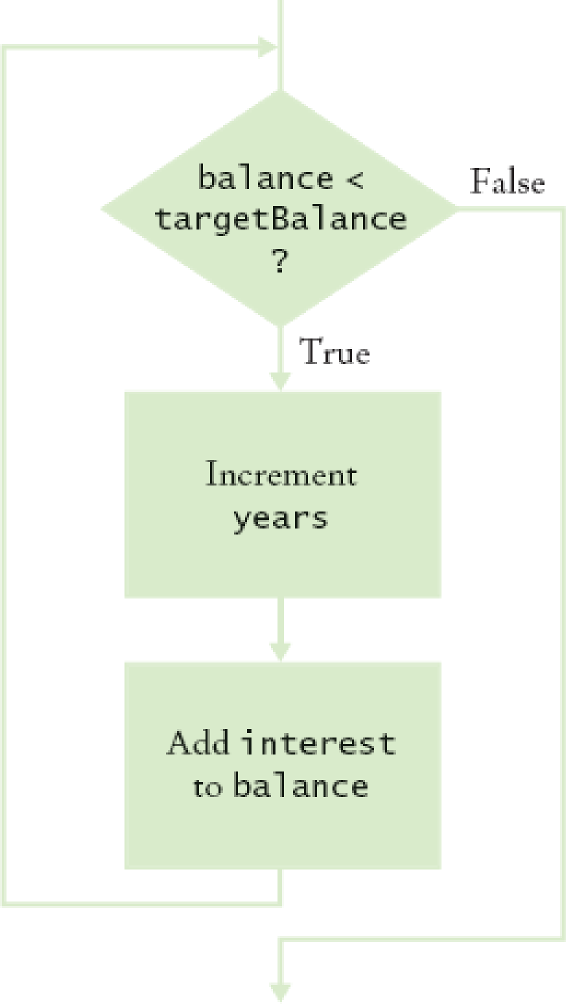
Year	Balance
0	$10,000
1	$10,500
2	$11,025
3	$11,576.25
4	$12,155.06
5	$12,762.82

When has the bank account reached a target balance of $500,000 ?

while loop

Calculating the Growth of an Investment
Invest $10,000, 5% interest, compounded annually

When has the bank account reached a target balance of $500,000 ?



while statement

Syntax 2.5: while statement

while (*condition*)
  statement

Example:

```
while (balance < targetBalance)
{
        year++;
        double interest = balance * rate / 100;
        balance = balance + interest;
}
```

Purpose:
To repeatedly execute a statement as long as a condition is true

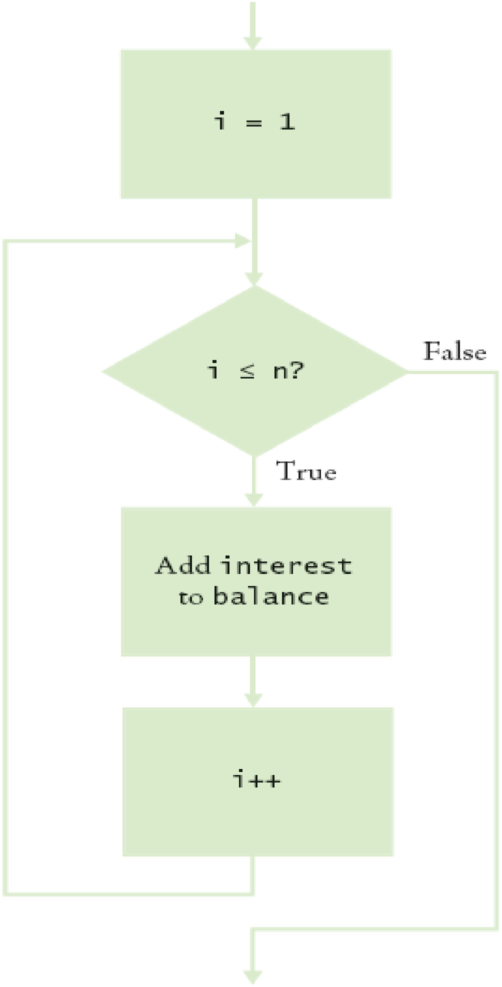
for loop

for (*initialization*; *condition*; *update*)
    statement

Example:

```
for (int i = 1; i <= n; i++)
   {
     double interest = balance * rate / 100;
     balance = balance + interest;
   }
```

for loop



```
for (int i = 1; i <= n; i++)
{
            double interest = balance * rate / 100;
            balance = balance + interest;
}
```

for statement

Syntax 2.6: for statement

for (*initialization*; *condition*; *update*)
  statement

Example:

```
 for (int i = 1; i <= n; i++)
{
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

Purpose:

To execute an initialization, then keep executing a statement and updating an expression while a condition is true

Fundamental Programming Structures

- A Simple Java Program
- Comments
- Data Types
- Variables
- Operators
- Strings
- Input and Output
- Control Flow
- Arrays
- Methods

The AnalzeNumbers Problem

Consider a *pseudocode* of a program **AnalyzeNumbers**

```
read any numbers from keyboard

compute their average

count how many input numbers are above the average
```

How to solve this problem
if you don't use an array ?

Why the problem is hard ?

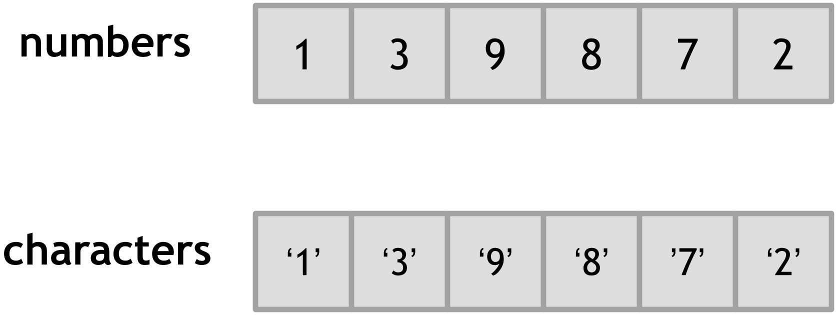
- We need each input value twice:
  - Compute the average
  - Count how many were above the average

- We could assume a maximum number of inputs (e.g. 100 values), and read each value into a variable
  - Solved, but too many variables to declare!

```
import java.util.Scanner;
public class AnalyzeNumber {
 public static void main(String[] args) {
   final int NUMBER_ELEMENTS = 100;
   double sum = 0.0;
   // declare 100 variables to keep the input
   double n1=0.0, n2=0.0, n100=0.0;
   Scanner in = new Scanner(System.in);
   for ( int i = 0; i < NUMBER_ELEMENTS; i++) {
    System.out.print("Enter a new number: ");
    switch ( i ) {
     case  0:   n1   = in.nextDouble(); sum += n1;  break;
     // . . .
     case 99:   n100 = in.nextDouble(); sum += n100; break;
    }
   }
   double average = sum / NUMBER_ELEMENTS;
   int count=0;
   // count numbers that are above average, another switch .. case
   if (n1 > average) count++;
   if (n2 > average) count++;
   // . . .
   if (n100 > average) count++;
   System.out.println("# of elements above average : " + count);
 }
}
```

Array

- **Array** is a *data structure* that stores a fixed-size sequential collection of elements of the same type



numbers	1	3	9	8	7	2
characters	‘1’	‘3’	‘9’	‘8’	’7’	‘2’

```
int[] numbers = new int[6];

char[] characters = new char[6];
```

Java Array Data Type

- **Declare** Array variables
- **Create** Arrays
- Array **Initializers**
- Array **Size** and **Index**
- Iterate Arrays
- Copying Arrays
- Passing/Returning Arrays to/from Methods
- Variable-length Argument Lists
- Multidimensional Arrays

Declaring Array [Reference] variables

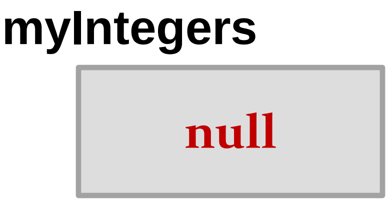
syntax: elementType[] arrayRefVar;

```
// elementType is a primitive type
double[]  myDoubles;
int[]        myIntegers;
char[]      myChars;

// elementType is a standard Java class
String[]   myArgs;
Date[]     myDates;

// elementType is a user-defined class
Employee[]   empList;
Card[]           deck;
```

What happens when you declare an array reference variable ?



Reference variables store null / address / handle of an array object.

Creating an Array

syntax: **arrayRefVar = new elementType[arraySize];**

elementType[] arrayRefVar = new elementType[arraySize];

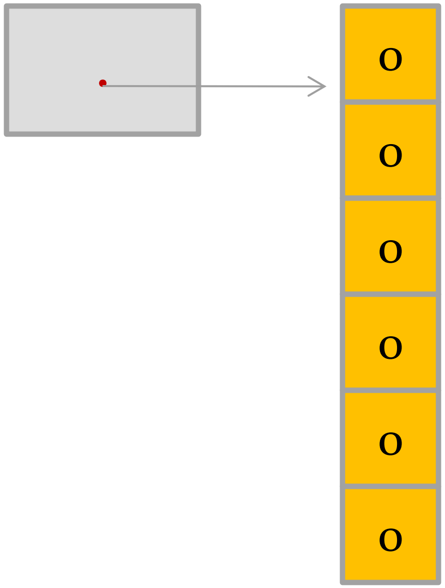
```
// elementType is a primitive type
int[]      myIntegers;
myIntegers = new int[4];

double[]  myDoubles;
myDoubles = new double[5];
double[] numbers = new double[100];

// elementType is a class
String[] strList = new String[8];
Employee[]   empList = new
Employee[10];
```

What happens when you create an array ?

myIntegers = new int[6];



*The array object is kept in a memory area called a **heap***

Array Initializers

syntax: **elementType[] arrayRefVar = { value0, value1, ..., valuek}**

```
double[]  myNumbers = { 1.9, 2.9, 3.4, 3.5, 4.8, 52.0, 49.1 };
String[] monthNames = {“Jan”,”Feb”,”Mar”,”Apr”,”May”,”Jun”,
                      ”Jul”,”Aug”,”Sep”,“Oct”,“Nov”,”Dec”};
```

```
Employee[]  empList =
        {new Employee(“emp1”), new Employee(“emp2”) };
```

Array Size and Index

syntax: **arrayRefVar.length // get the number of elements**

arrayRefVar[***index***] **// access the element at *index***

```
double[]  myNumbers = { 1.9, 2.9, 3.4, 3.5, 4.8, 52.0, 49.1 };


System.out.println(“Array size = “ + myNumbers.length );
System.out.println(“The first element of this array is “ + myNumbers[0] );
System.out.println(“The third element of this array is “ +
myNumbers[2] );
```

Idiom for Array Processing

- **for loop**

```
int[] nums = {1,8,9,2,5};
for (int i = 0; i < nums.length; i++)
        System.out.println(nums[i]);
```

- **for-each loop**

- don’t have to use index variable
- avoid ArrayIndexOutOfBoundsException

```
int[] nums = {1,8,9,2,5};
for (int elem : nums)
        System.out.println(elem);
```

The AnalyzeNumbers Solution (using Arrays)

```
import java.util.Scanner;
public class AnalyzeNumbers {
  public static void main(String[] args) {
    final int NUMBER_ELEMENTS = 5;
    double[] numbers = new double[NUMBER_ELEMENTS];
    double sum = 0;
    Scanner in = new Scanner(System.in);
    for ( int i = 0; i < numbers.length; i++) {
      System.out.print("Enter a new number: ");
      numbers[i] = in.nextDouble();
      sum += numbers[i];
    }
    double average = sum / NUMBER_ELEMENTS;
    int count = 0;
    for ( double elem : numbers )
        if (elem > average) count++;
    System.out.println("Average is " + average);
    System.out.println("# of elements above average : " + count);
  }
}
```

Out-of-Bounds

- Legal array indexes: [0 …. . Array.length – 1]
  - Accessing any index outside this range will throw an **ArrayIndexOutOfBoundsException**

```
int[] myArray = new int[8];

System.out.println(myArray[0]);    // OK

System.out.println(myArray[7]);    // OK

System.out.println(myArray[-1]);   // exception

System.out.println(myArray[8]);    // exception
```

Copying Arrays

Is this the correct way to **copy** arrays?

```
int[] list1 = {1, 2, 3, 4 };

int[] list2;

list2        =       list1
```

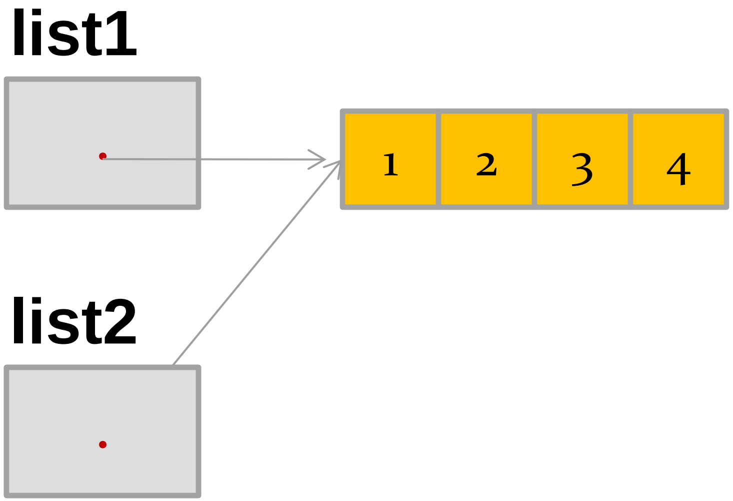
Copying Arrays

Is this the correct way to **copy** arrays?

```
int[] list1 = {1, 2, 3, 4 };

int[] list2;

list2       =      list1
```



Copying Arrays : correct ways (1/3)

- Use a for-loop

```
int[] srcArray = {2, 3, 1, 5, 10};
int[] dstArray = new int[srcArray.length];
for (int i = 0; i < srcArray.length; i++)
{
            dstArray[i] = srcArray[i];
}
```

Copying Arrays : correct ways (2/3)

- Use **System.arraycopy()**

```
int[] srcArray = {2, 3, 1, 5, 10};
int[] dstArray = new int[srcArray.length];
System.arraycopy(
                    srcArray,          /* source array reference var. */
                    0,                 /* starting position of the source array */
                    dstArray,          /* target array reference var. */
                    0,                 /*  starting position of the target array */
                    srcArray.length    /* number of elements to copy */
);
```

Copying Arrays : correct ways (3/3)

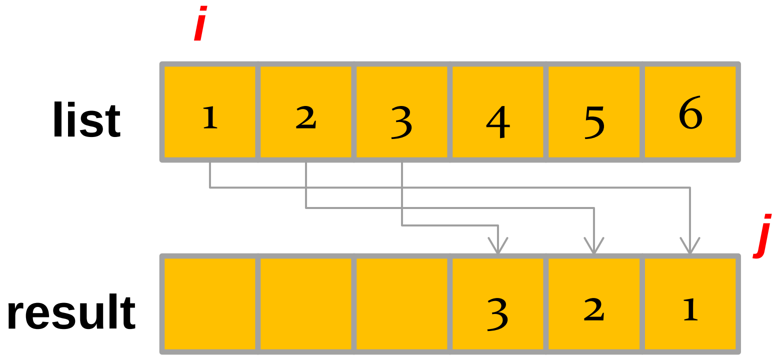
- Use **java.util.Arrays.copyOf()**

```
int[] srcArray = {2, 3, 1, 5, 10};
int[] dstArray = java.util.Arrays.copyOf(
        srcArray,               /* source array reference */
        srcArray.length         /* number of elements to copy */
);
```

Passing/Returning Arrays to/from Methods

Example: Reversing an Array

```
public static int[] reverse(int[] list)
{
        int[] result = new int[list.length];
        for (int i = 0, j = result.length-1;   i < list.length;   i++, j--)
                result[j] = list[i];
        return result;
}
```



Variable-Length Argument Lists

syntax: **typeName... parameterName**

Example: print maximum number in an array

```
public static void printMax(int… numbers)
// int… means any numbers of int arguments
{
    if (numbers.length == 0) return;   // no argument
    int max = numbers[0];
    for (int n : numbers)    if (n > max) max = n;
    System.out.println(“The maximum number is “ + max);
}
```

```
printMax(3,4,8,3, 9);                                  // output 9
int[] nums = {3,4,8,3,11};   printMax(nums);           // output 11
```

Variable-Length Argument Lists

syntax: **typeName[] parameterName**

Example: print maximum number in an array

```
public static void printMax2(int[] numbers)
// int[] means array of integers of any lengths!!!
{
    if (numbers.length == 0) return;   // no argument
    int max = numbers[0];
    for (int n : numbers)
                  if (n > max) max = n;

    System.out.println(“The maximum number is “ + max);
}
```

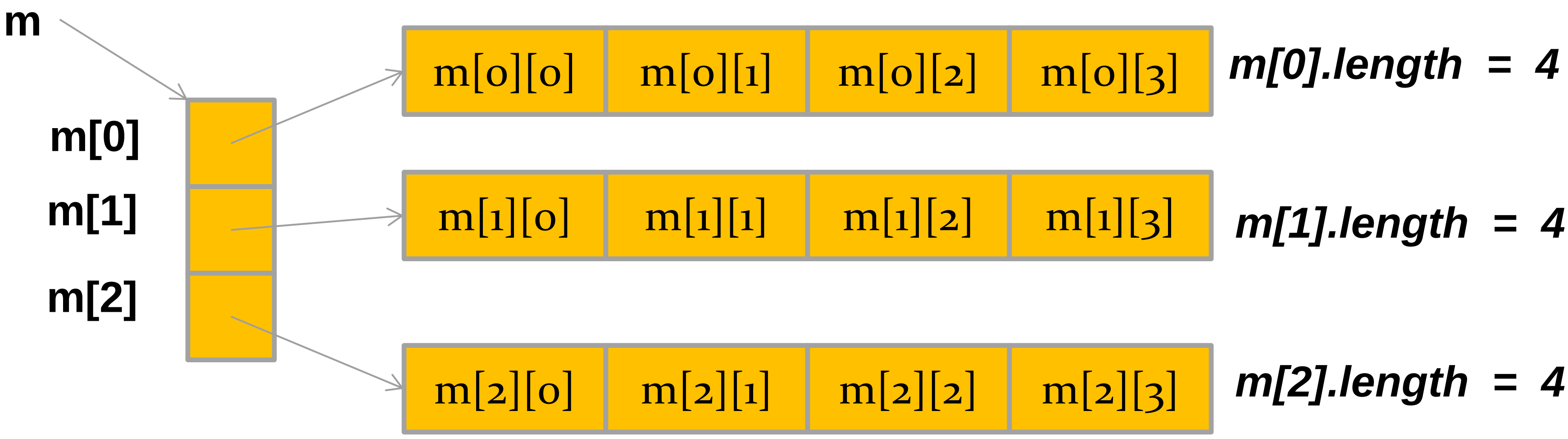
~~printMax(3,4,8,3, 9); // !!! ERROR~~ --- WHY ?

int[] nums = {3,4,8,3,11}; printMax(nums); // OK: output 11

Multidimensional Arrays

syntax: **elementType[][] arrayRefVar;**

```
int[][] m = new int[3][4];
```



m

m[0]

m[1]

m[2]

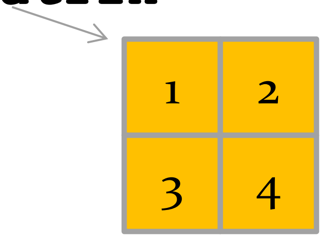
m[0][0]	m[0][1]	m[0][2]	m[0][3]	*m[0].length = 4*
m[1][0]	m[1][1]	m[1][2]	m[1][3]	*m[1].length = 4*
m[2][0]	m[2][1]	m[2][2]	m[2][3]	*m[2].length = 4*

m.length = 3

Multidimensional Arrays

syntax: **elementType[][] arrayRefVar** = { {}, {}, {} };

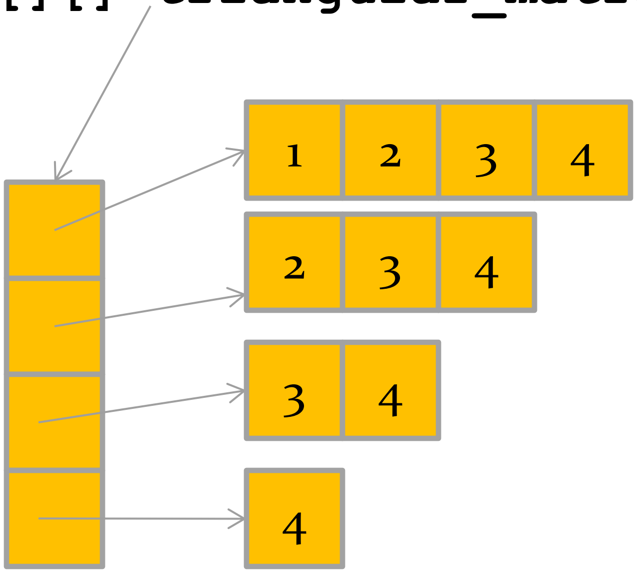
```
int[][] matrix  =  { {1,2},{3,4} };
```



1 2

3 4

```
int[][] triangular_matrix  =  { {1,2,3,4},
                                {2,3,4},
                                {3,4}
                                {4} };
```



1 2 3 4

2 3 4

3 4

4

Processing Multidimensional Arrays

```
int[][] matrix  =  new int[3][4];

// init each element with random integer (<100)
for (int row=0; row<matrix.length; row++)
        for (int column=0; j<matrix[row].length; column++)
                matrix[row][column] = (int)(Math.random() * 100);

// print all element in tabular format
for (int[] row : matrix) {
        for (int elem : row)
                System.out.print(elem + “ “);
        System.out.println();
}
```

Limitations of Array

- Cannot resize an existing array

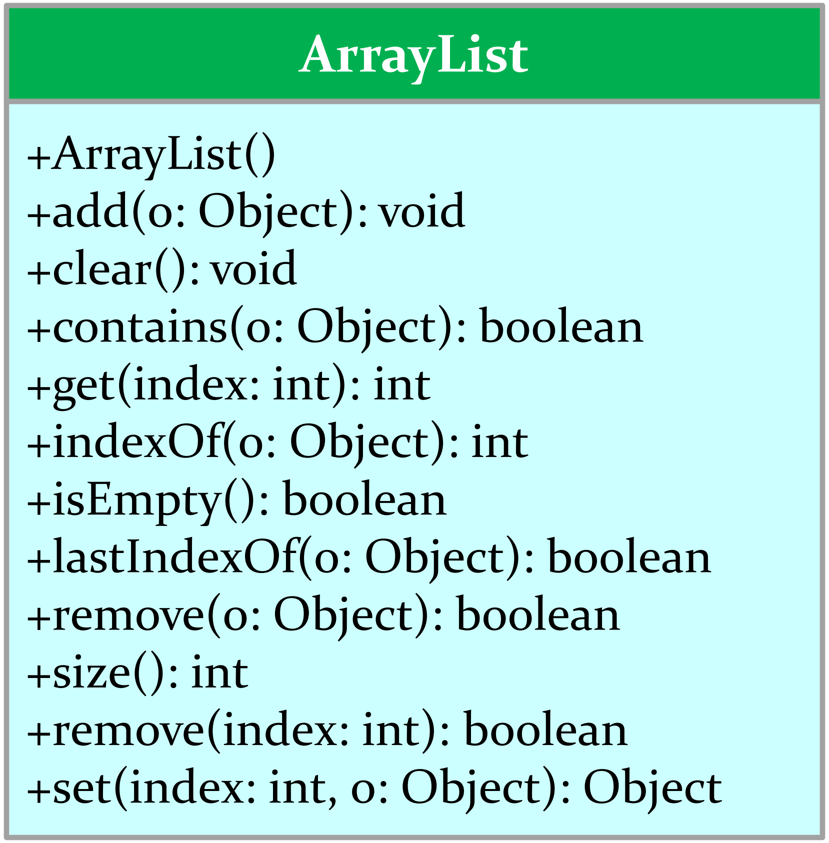
```
int[] a = new int[4];
a.length = 10;   // error!
```

- Cannot compare arrays with == or equals:

```
int[] a1 = {42, -7, 1, 15};
int[] a2 = {42, -7, 1, 15};
if (a1 == a2) { .... }              // false!
if (a1.equals(a2) { .... }      // false!
```

The ArrayList Class

- The size of an Array is fixed once created
- Java provides the java.util.*ArrayList* class for storing any number of objects



ArrayList

+ArrayList()
+add(o: Object): void
+clear(): void
+contains(o: Object): boolean
+get(index: int): int
+indexOf(o: Object): int
+isEmpty(): boolean
+lastIndexOf(o: Object): boolean
+remove(o: Object): boolean
+size(): int
+remove(index: int): boolean
+set(index: int, o: Object): Object

Study the ArrayList class by reading the Java API documentation. For each method listed here, please find:

- Description
- Usage example

```
// CD.java
class CD {
  private String artist;
  private String album;
  CD(String artist, String album) { this.artist = artist; this.album = album;}
  public String getArtist() { return artist; }
  public String getAlbum() { return album; }
  @Override
  public String toString() {  return album + “ by “ + artist;  }
}

/* Example: Using ArrayList to keep track of CD objects */
// CDCollection.java
import java.util.ArrayList;
class CDCollection {
  private ArrayList<CD> myCds = new ArrayList<CD>();
  CDCollection() { }
  void addCD(CD newCD) {
    myCds.add(newCD);
  }
  void printCollectionInfo() {
    for (CD cd: myCds)    System.out.println(cd);
  }
}
```

Example Application of Array

Array for Tallying

A multi-counter problem

- Problem: Write a method `mostFrequentDigit` that returns the digit value that occurs most frequently in a number.
  - Example: The number 669260267 contains:
    one 0, two 2s, four 6es, one 7, and one 9.
    `mostFrequentDigit(669260267)` returns 6.
  - If there is a tie, return the digit with the lower value.
    `mostFrequentDigit(57135203)` returns 3.

Solution 1

- Declare 10 counters for 10 digits:

  counter0, counter1, counter2, …, counter9

Solution 2: Use a counter array

```
int[] counters = new int[10];
int inputNumber = scanner.nextInt();
while (inputNumber > 0) {
    int digit = inputNumber % 10;
    counters[digit]++;
    inputNumber /= 10;
}
```

inputNumber= 26206676

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	1	0	2	0	0	0	4	1	0	0

Fundamental Programming Structures

- A Simple Java Program
- Comments
- Data Types
- Variables
- Operators
- Strings
- Input and Output
- Control Flow
- Arrays
- Methods

Why use methods?

Suppose that you need to find the area of three circles, you may come up with this code

```
// circle1, radius = 1.0
double radius1 = 1.0;
double area1 = (22.0/7.0)*(1.0)*(1.0);

// circle2, radius = 2.0
double radius2 = 2.0;
double area2 = (22.0/7.0)*(2.0)*(2.0);

// circle3, radius = 7.0
double radius3 = 7.0;
double area3 = (22.0/7.0)*(7.0)*(7.0);
```

Why use methods?

Methods can be used to

- reduce redundant code and enable code reuse
- modularize code (divide a large problem into sub-problems)

```
/* create a method to find an area of a circle */
public static double area(double radius) {
   final double PI = 3.14159;
   double area = PI*(radius)*(radius);
   return area;
}
public static void main(String[] args) {
   double r1=1.0, r2=2.0, r3=7.0;
   System.out.println(“Area of circle1 is “ + area(r1));
   System.out.println(“Area of circle1 is “ + area(r2));
   System.out.println(“Area of circle1 is “ + area(r3));
}
```

Defining and Calling a method

Syntax for defining a method

```
modifier returnValue methodName(list of parameters)
{
    // method body
}
```

Syntax for calling a method

```
methodName(argumentList);
```

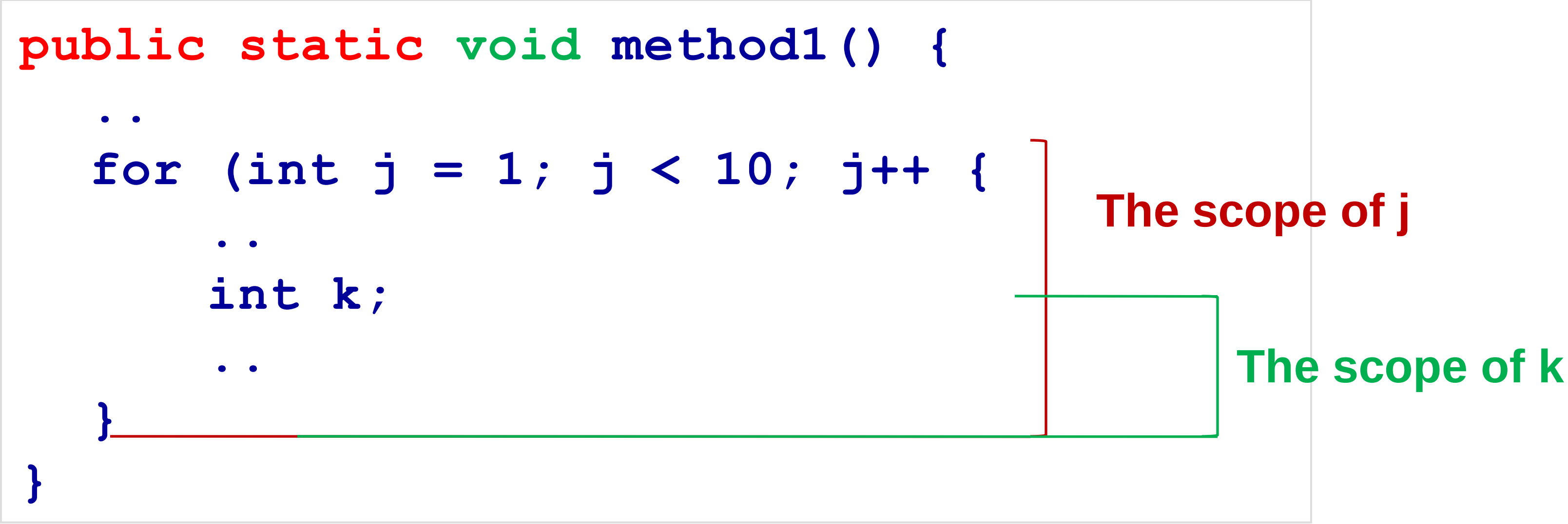
```
public static double area(double radius)
{
  final double PI = 3.14159;
  double area = PI*(radius)*(radius);
  return area;
}
```

```
// call the method
area(3.0);
area(5.0);
```

Scope of Variables

Scope of a variable is the part of the program where the variable can be referenced.

- A variable defined inside a method is called a *local variable*
- the scope of a local variable starts from its declaration to the end of the block that contains the variable.



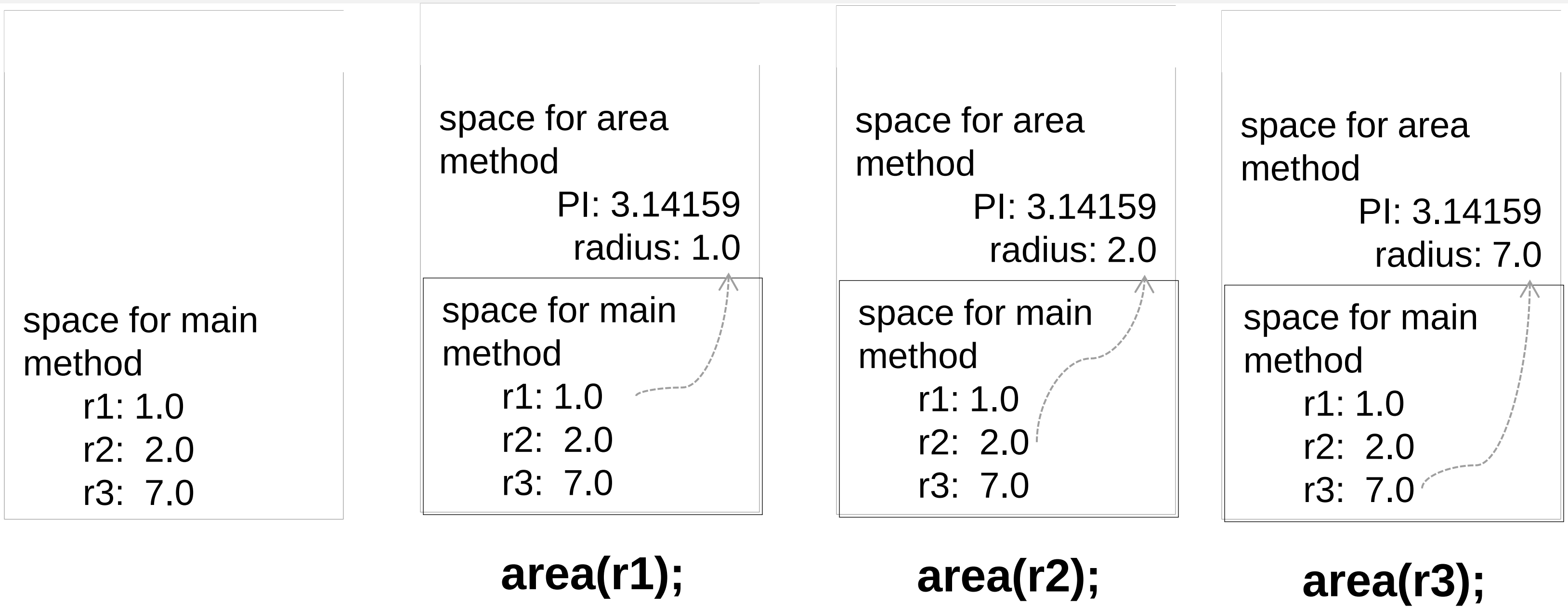
```
public static void method1() {
   ..
   for (int j = 1; j < 10; j++ {
        ..
        int k;
        ..
   }
}
```

The scope of j

The scope of k

Call Stacks and Parameter Passing

- Each time a method is invoked, the system stores parameters and variables in a memory area, called a *stack*.
- *The stack* stores elements in last-in, first-out fashion



In Java, parameters are **pass-by-value.**

- The value of the argument variables is passed to the parameters.
- The variable is not affected by changes mad inside the method.

Summary: Legacy Language Features

- You should now be able to solve various programming problems using the following building blocks.
  - statements and expressions
  - decisions
  - loops
  - arrays
  - Methods

- These legacy language features are not enough for productive *large-scale* software systems development
  - object-oriented features of Java