CONTROL STATEMENT

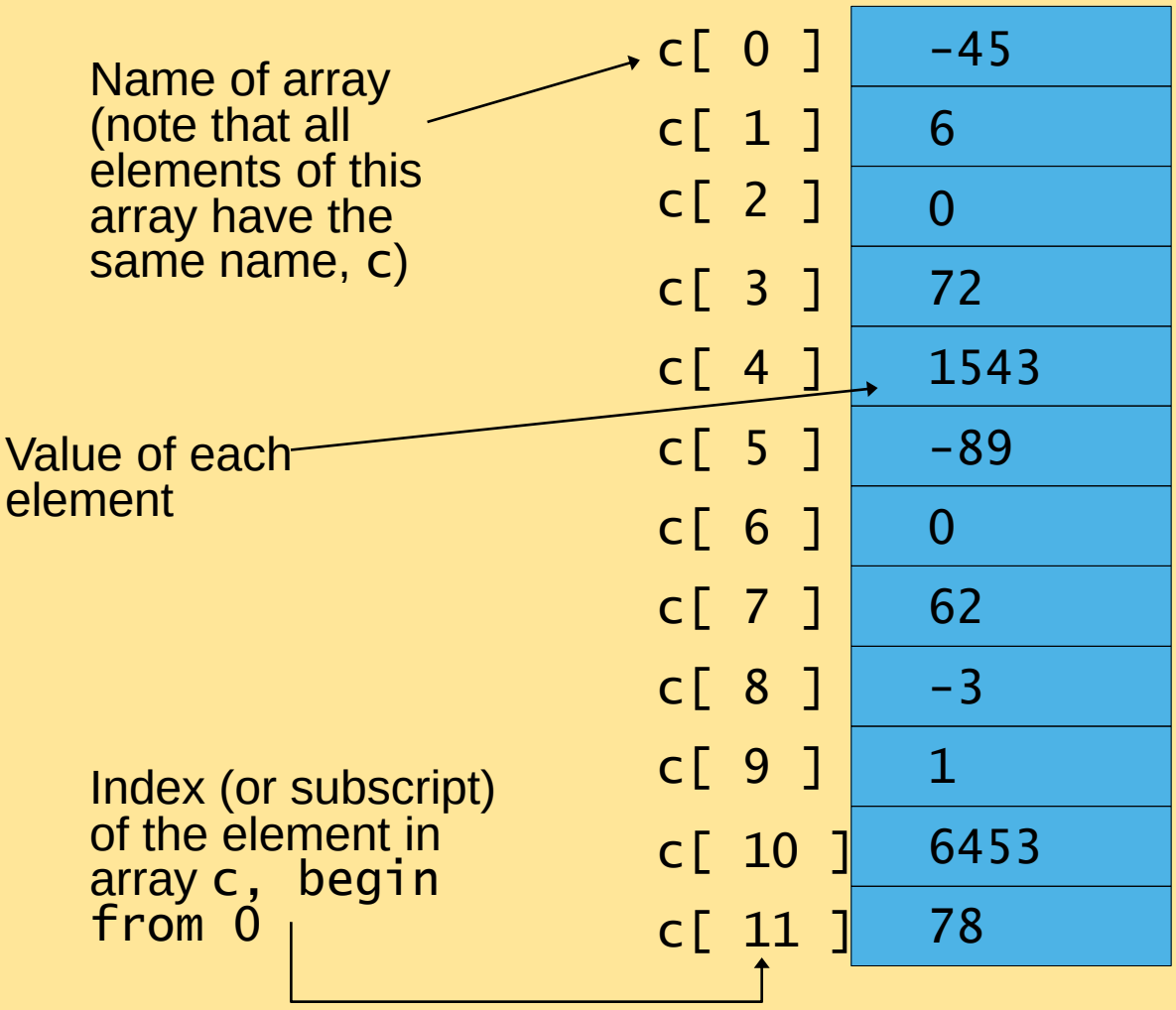
Instructor: DieuNT1

Table of contents

Java Arrays

- An array is a container that holds data (values) of one single type.
  - ✓ For example, you can create an array that can hold 100 values of int type.
- Array is a fundamental construct in Java that allows you to store and access large number of values conveniently.
- **Arrays:**
  - ✓ **Data structures**
  - ✓ Related data items of **same type**
  - ✓ Remain same size once created
    - *Fixed-length* entries

Array structure



Array Declarations

- **Syntax:** Three ways to declare an array are

```
datatype[] identifier;
datatype[] identifier = new datatype[size];
datatype[] identifier = {value1,value2,…valueN};
```

- You can also place the square brackets after the array's name:

```
datatype identifier[];//this form is discouraged
```

- **Example:**

```
byte[] bArray;
float[] fArray = new float[20];
int[] iArray = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
```

Array Declarations

- Examine array **bArray, fArray, iArray:**
  - ✓ **bArray, fArray, iArray** is the array *name*
  - ✓ **fArray.**length accesses array c's *length*
  - ✓ **iArray** has 10 *elements*:

    iArray[0], **iArray** [1], … , **iArray** [9]
    - The *value* of **iArray** [0] is 32

Array Index

- Also called subscript
- Position number in square brackets
- Always begin from zero
  - ✓ Must >= 0 and < array's length
- **Example:**

```
fArray[0] = 12.5f;
for (int counter = 0; counter < iArray.length; counter++){
 output += counter + "\t" + iArray[counter] + "\n";
}
```

Multidimensional Arrays

- Multidimensional arrays
  - ✓ Tables with rows and columns
    - Two-dimensional array
    - Declaring two-dimensional array `b[2][2]`

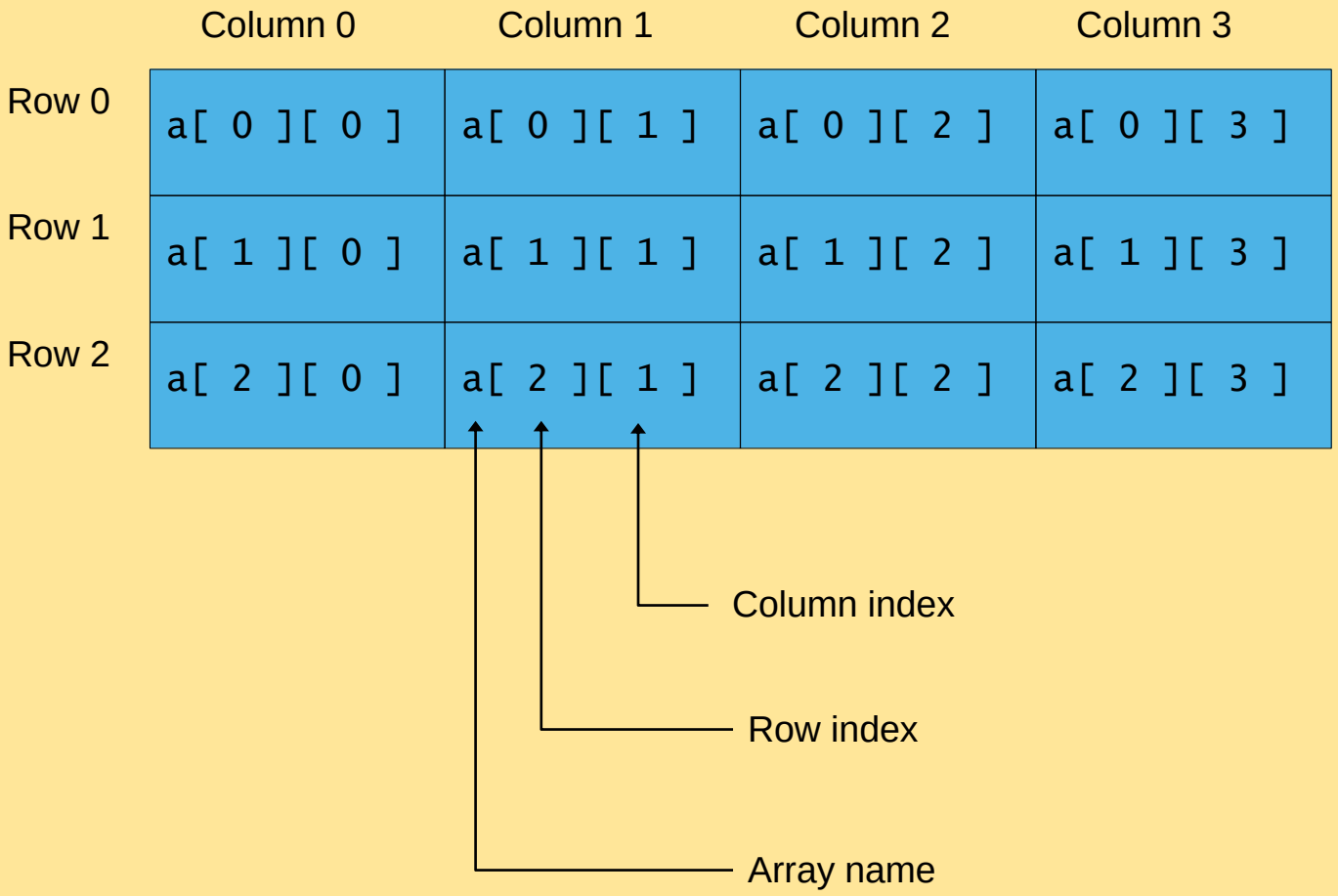
      ```
      int[][] b = { { 1, 2 }, { 3, 4 } };
      ```

      - 1 and 2 initialize `b[0][0]` and `b[0][1]`
      - 3 and 4 initialize `b[1][0]` and `b[1][1]`
    - 3-by-4 array

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

Multidimensional Arrays

- **Two-dimensional array structure**



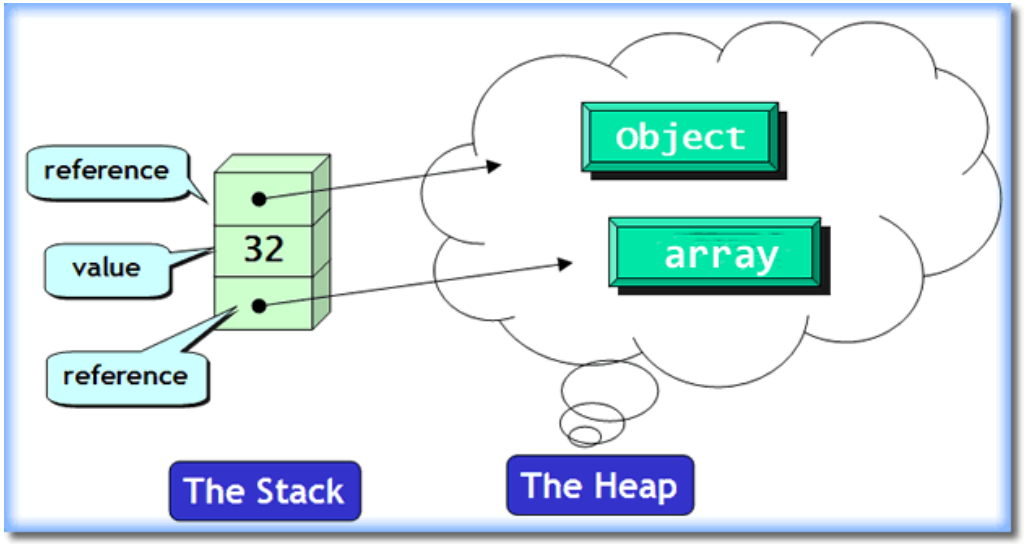
Section 2

Heap Space vs Stack Memory

Introduction

- To run an application in an optimal way, JVM divides memory into stack and heap memory.
  - ✓ **Declare new variables and objects, call new method, declare a *String* or perform similar operations**

    ➔ JVM designates memory to these operations from either Stack Memory or Heap Space.



Stack Memory

- **Stack Memory** in Java is used for static memory allocation and the execution of a thread.
- It contains ***primitive values*** that are specific to a method and ***references*** to objects that are in a heap, referred from the method.
- Access to this memory is in Last-In-First-Out (LIFO) order.
  - ✓ It grows and shrinks as new methods are called and returned respectively
  - ✓ Variables inside stack exist only as long as the method that created them is running
  - ✓ It's automatically allocated and deallocated when method finishes execution
  - ✓ If this memory is full, Java throws ***java.lang.StackOverFlowErro***r
  - ✓ Access to this memory is fast when compared to heap memory
  - ✓ This memory is threadsafe as each thread operates in its own stack

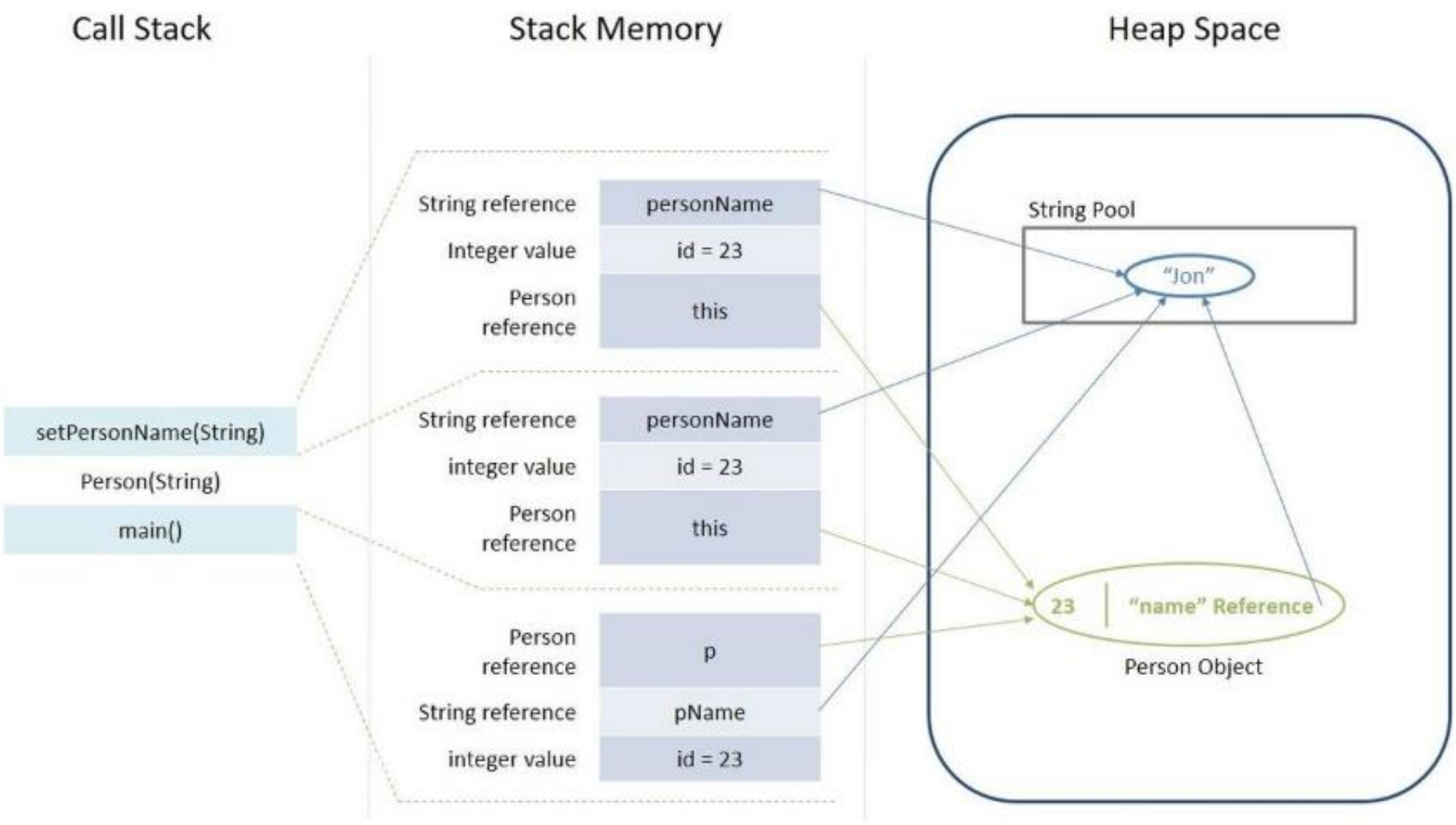
Heap Space

- Heap space in Java is used for ***dynamic memory allocation for Java objects*** and JRE classes at the runtime.
- ***New objects*** are always created in heap space and the references to this objects are stored in stack memory.
  - ✓ If heap space is full, Java throws j***ava.lang.OutOfMemoryError***
  - ✓ Access to this memory is relatively slower than stack memory
  - ✓ This memory, in contrast to stack, isn't automatically deallocated. It needs Garbage Collector to free up unused objects so as to keep the efficiency of the memory usage
  - ✓ Unlike stack, a heap isn't threadsafe and needs to be guarded by properly synchronizing the code

Heap Space vs Stack Memory

```
 3  class Person {
 4    int personId;
 5    String personName;
 6
 7    public int getPersonId() {..
10
11    public void setPersonId(int personId) {..
14
15    public String getPersonName() {..
18
19    public void setPersonName(String personName) {..
22
23    public Person(int personId, String personName) {..
28
29  }
30
31  public class Driver {
32    public static void main(String[] args) {
33      int id = 23;
34      String pName = "Jon";
35      Person p = null;
36      p = new Person(id, pName);
37    }
38  }
39
```

Heap Space vs Stack Memory



Section 3

Parameters

Parameters

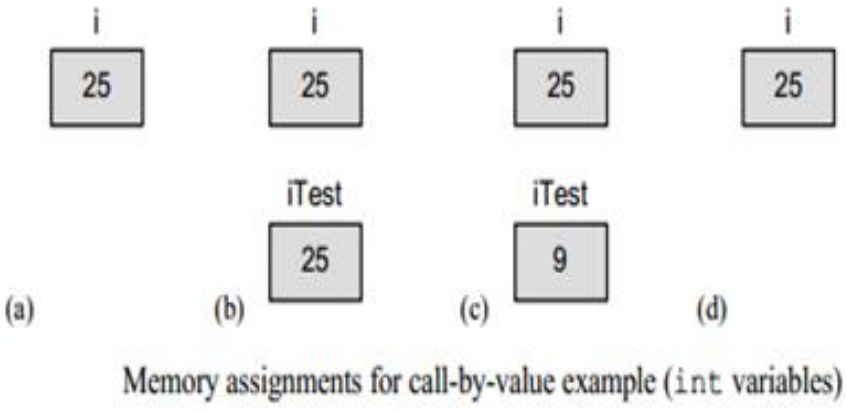
- Parameters (also called arguments) is variable that declare in the method definition.
- Parameters are always classified as "variables" not "fields".
- Two ways to pass arguments to methods
  - ✓ Pass-by-value
  - ✓ Pass-by-reference

Value and Reference Parameters

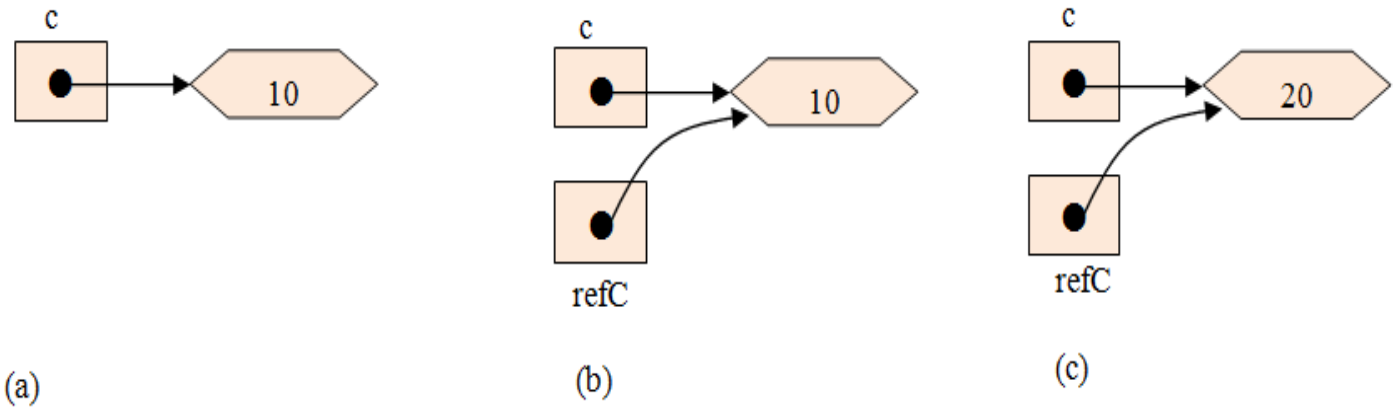
- Pass-by-value
  - ✓ Copy of argument's value is passed to called method
  - ✓ In Java, every primitive[nguyên thủy] is pass-by-value
- Pass-by-reference
  - ✓ Caller gives called method direct access to caller's data
  - ✓ Called method can manipulate this data
  - ✓ Improved performance over pass-by-value
  - ✓ In Java, every object is pass-by-reference
    - In Java, arrays are objects
      - Therefore, arrays are passed to methods by reference

Value and Reference Parameters(2)

- **Example**: reference **DemoPassByReference.java**
- **Ananys:**



Memory assignments for call-by-value example (`int` variables)



Passing Arrays to Methods

- To pass an array argument to a method
  - ✓ Create a method

```
public void modifyArray(int[] arr)
```

  - ✓ Array hourlyTemperatures is declared as

```
int[] hourlyTemperatures = new int[24];
```

  - ✓ Passes array `hourlyTemperatures` to method modifyArray

```
modifyArray(hourlyTemperatures);
```

- Example: reference **PassArray.java**

Passing Arrays to Methods

```
public class PassArray {
  // initialize applet
  public static void main(String args[]) {
    int array[] = { 1, 2, 3, 4, 5 };

    String output = "Effects of passing entire array by reference:\n"
        + "The values of the original array are:\n";

    // append original array elements to String output
    for (int counter = 0; counter < array.length; counter++)
      output += "   " + array[counter];

    modifyArray(array); // array passed by reference

    output += "\n\nThe values of the modified array are:\n";

    // append modified array elements to String output
    for (int counter = 0; counter < array.length; counter++)
      output += "   " + array[counter];

    output += "\n\nEffects of passing array element by value:\n"
        + "array[3] before modifyElement: " + array[3];

    modifyElement(array[3]); // attempt to modify array[ 3 ]

    output += "\narray[3] after modifyElement: " + array[3];

    System.out.println(output);

  } // end method init
```

Passing Arrays to Methods

```
  // multiply each element of an array by 2
  public static void modifyArray(int array2[]) {
    for (int counter = 0; counter < array2.length; counter++)
      array2[counter] *= 2;
  }

  // multiply argument by 2
  public static void modifyElement(int element) {
    element *= 2;
  }

} // end class PassArray
```

```
Effects of passing entire array by reference:
The values of the original array are:
  1   2   3   4   5

The values of the modified array are:
  2   4   6   8   10

Effects of passing array element by value:
array[3] before modifyElement: 8
array[3] after modifyElement: 8
```

Section 4

Flow Control

Flow Control

- **Decision-making**
  - ✓ if-else statement
  - ✓ switch-case statement
- **Loops**
  - ✓ while loop
  - ✓ do-while loop
  - ✓ for loop
- **Branching**
  - ✓ break
  - ✓ continue
  - ✓ return

Control Flow

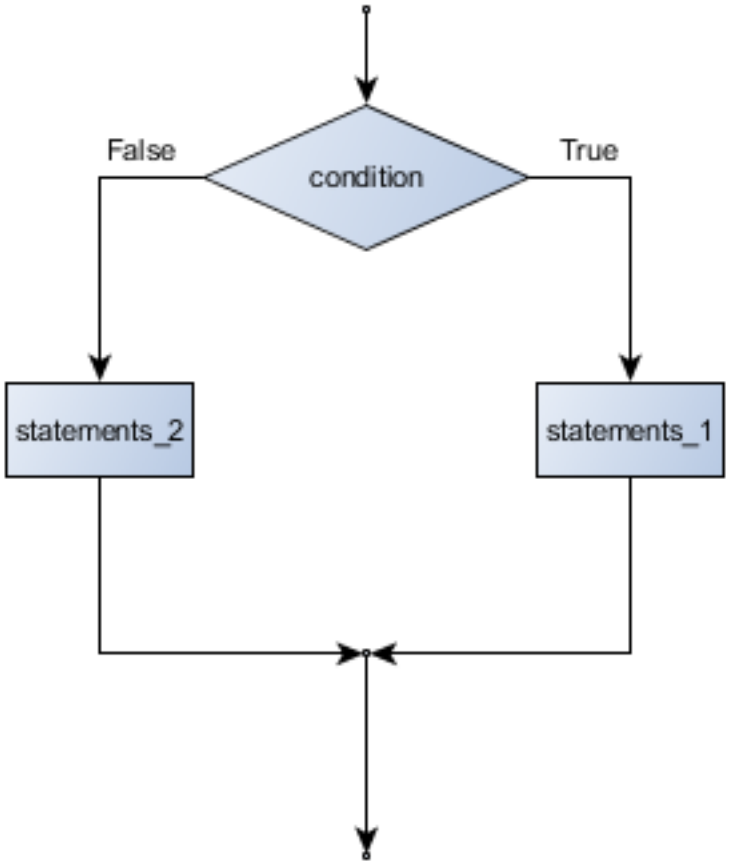
- All application development environments provide a decision making process called control flow statements that direct the application execution.
- Flow control enables a developer to create an application that can examine[kiểm tra] the existing conditions, and decide a suitable course of action.
- Loops or iteration are an important programming construct that can be used to repeatedly execute a set of actions.
- Jump statements allow the program to execute in a non-linear fashion.

if-else statement

- **Syntax:**

```
if (condition){
 action1;
} else {
 action2;
}
```

- **Note:**
  - ✓ "**else**" is optional
  - ✓ Alternative way to if-else is conditional operator (?:)



if-else statement

- **Example:**

```
public class CheckNum {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int num = 10;
        if (num % 2 == 0) {
        System.out.println(num + " is an even number");
        } else {
        System.out.println(num + " is an odd number");
        }
    }
}
```

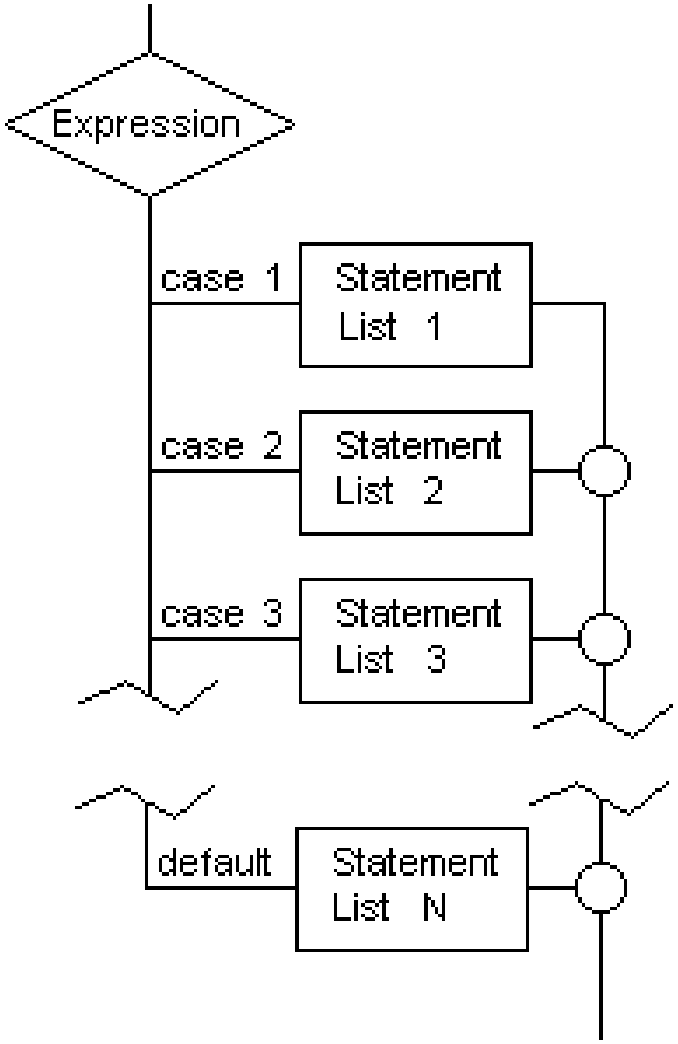
switch – case statement

- Unlike if-then and if-then-else statements, the switch statement can have a number of possible execution paths.
- A switch works with the byte, short, char, and int primitive data types.
- It also works with enumerated types, the String class, and a few special classes that wrap certain primitive types: Character, Byte, Short, and Integer (discussed in Numbers and Strings).

switch – case statement

- **Syntax:**

```
switch(expression) {
    case value_1:
      statement_1; [ break;]
    case value_2:
      statement_2; [ break;]
    …
    case value_n:
      statement_n; [ break;]
    default:
      statement_n+1; [break;]
}
```



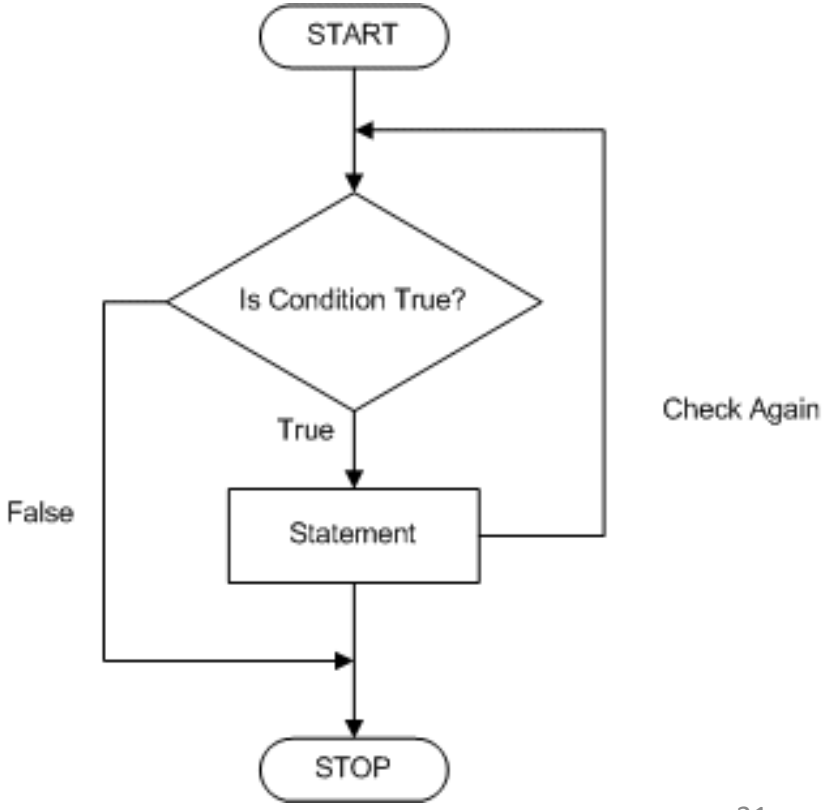
switch – case statement

```
public class SwitchDemo2 {
    public static void main(String[] args) {

        int month = 2;
        int year = 2000;
        int numDays = 0;

        switch (month) {
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
                numDays = 31;
                break;
            case 4:
            case 6:
            case 9:
            case 11:
                numDays = 30;
                break;
            case 2:
                if ( ((year % 4 == 0) && !(year % 100 == 0))
                     || (year % 400 == 0) )
                    numDays = 29;
                else
                    numDays = 28;
                break;
        }
        System.out.println("Number of Days = " + numDays);
    }
}
```

while Loop

- `while` loops are used for situations when a loop has to be executed as long as certain condition is True.
- The number of times a loop is to be executed is not pre-determined, but depends on the condition.
- **The syntax is:**

```
while (condition) {
    action statements;
}
```



while Loop

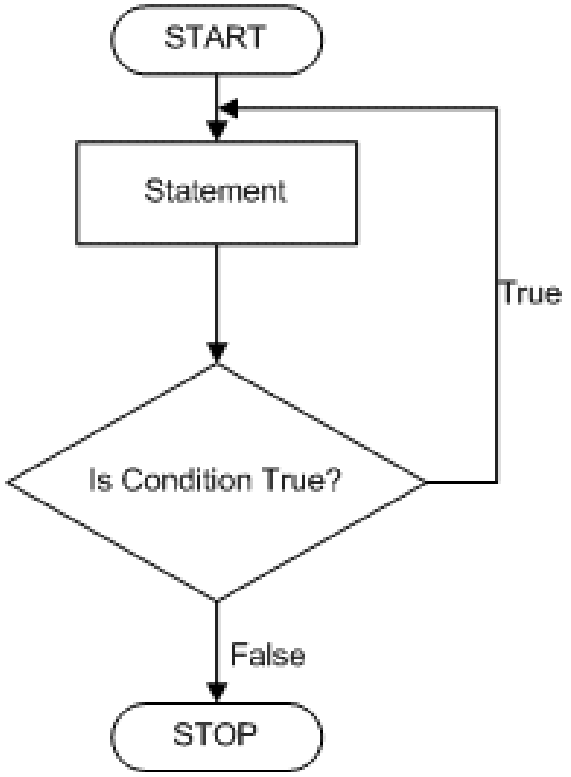
- **Example:**

```
public class FactDemo {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int num = 5, fact = 1;
        while (num >= 1) {
        fact *= num;// fact = fact * num;
        num--;
        }
        System.out.println("The factorial of 5 is : " +
          fact);
    }
}
```

do – while Loop

- The `do-while` loop executes certain statements till the specified condition is True.
- These loops are similar to the `while` loops, except that a `do-while` loop executes at least once, even if the specified condition is False.
- **The syntax is:**

```
do {
    action statements;
} while (condition);
```



do – while Loop

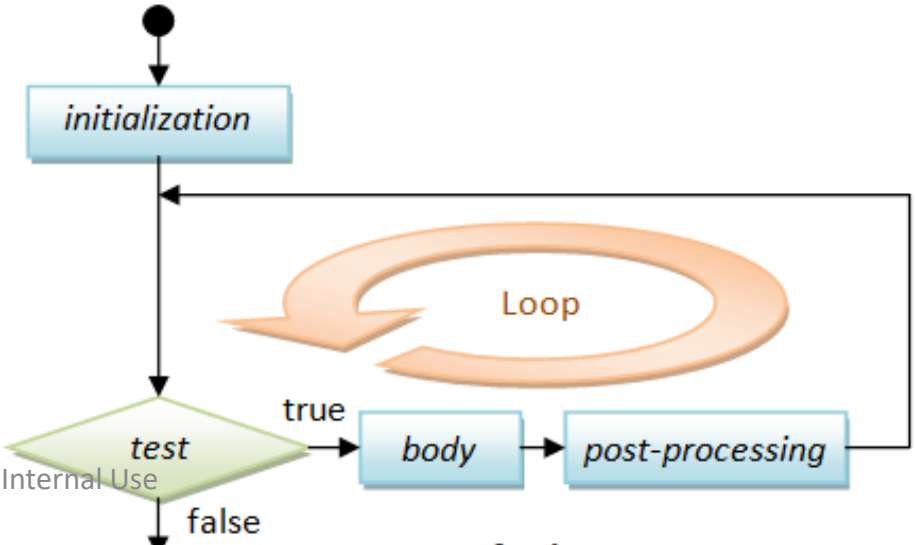
- **Example:**

```
public class DoWhileDemo {
    public static void main(String[] args) {
        int count = 1, sum = 0;
        do {
        sum += count;
        count++;
        } while (count <= 100);
        System.out.println("The sum of first 100 numbers is
          : " + sum);
    }
}
```

for Loop

- All loops have some common features: a counter variable that is initialized before the loop begins, a condition that tests the counter variable and a statement that modifies the value of the counter variable.
- The `for` loop provides a compact format for incorporating these features.
- Syntax:

```
for (initialization;loopContinuationCondition; increment)
{
   statement;
}
```



for Loop

- Example:

```
public class ForDemo {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int count = 1, sum = 0;
        for (count = 1; count <= 10; count += 2) {
        sum += count;
        }
        System.out.println("The sum of first 5 odd numbers is : " +
          sum);
    }
}
```

Break Statements

- The break statement has two forms: labeled and unlabeled.
- Use unlabeled break to terminate a switch, for, while, or do-while loop
- Use labeled break to terminates an outer statement
- Example:

```
public class BreakDemo {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        for (int count = 1; count <= 100; count++) {
        if (count == 10) {
        break;
        }
        System.out.println("The value of num is : " + count);
        }
        System.out.println("The loop is over");
    }
}
```

Continue statement

- The continue statement skips the current iteration of a for, while , or do-while loop.
- The unlabeled form skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop.
- The labeled continue statement skips the current iteration of an outer loop marked with the given label.

Continue statement

- **Example:**

```
public class ContinueDemo {
    public static void main(String[] args) {
        String searchMe = "peter piper picked a peck of pickled
          peppers";
        int max = searchMe.length();
        int numPs = 0;
        for (int i = 0; i < max; i++) {
        // interested only in p's
          if (searchMe.charAt(i) != 'p') {
            continue;
          }
          numPs++;
        }
        System.out.println("Found " + numPs + " p's in the
          string.");
    }
}
```

Return statement

- The return statement exits from the current method, and control flow returns to where the method was invoked.
- The return statement has two forms:
  - ✓ Returns a value: `return ++count;`
  - ✓ Doesn't returns a value (void): `return;`
- The data type of the returned value must match the type of the method's declared return value.

SUMMARY

- Arrays
- Heap Space and Stack Memory
- Parameters
- Flow Control

Thank you