Fundamental Programming Structures in Java: Control Flow, Arrays, Vectors and ArrayList

In this class, we will cover:

- Blocks and scope
- Writing decision-making statements
- Writing loops
- Declaring and accessing arrays, vectors and arraylist

Blocks and Scope

- **All classes and methods contain code within curly brackets called blocks.**
- **Where is the error in this class.**

```
class ScopeExample {
    public static void main(String[] args) {
            //  aNumber comes into existence
            int aNumber = 22;
            System.out.println("Number is " + aNumber);
            {
                    //  anotherNumber comes into existence
                    int anotherNumber = 99;
                    System.out.println("aNumber is " + aNumber);
                    System.out.println("anothreNumber is " +

anotherNumber);
            } //  End of block - anotherNumber ceases to exist
            System.out.println("aNumber is " +  aNumber);
            System.out.println("anotherNumber is " +
anotherNumber);
    } //  End of outer block - anumber ceases to exist
```

Writing Decision-Making Statements

- Decision-Making Statements
  - Determine whether a condition is true, and take some action based on determination
  - What are the three ways to write a decision-making statement in Java?

Three ways to write decision-making statements:

- if statement
- switch statement
- conditional operator

Writing Decision-Making Statements

- Writing *if* Statements
  - *if* statement:
    - Interrogates logical expression enclosed in parentheses
      - boolean expressions can use just one word
      - example:

```
boolean isFound=true;
if (isFound)
    System.out.println("The object is found.");
```

    - Determines whether it is true or false
    - Uses logical operators to compare values

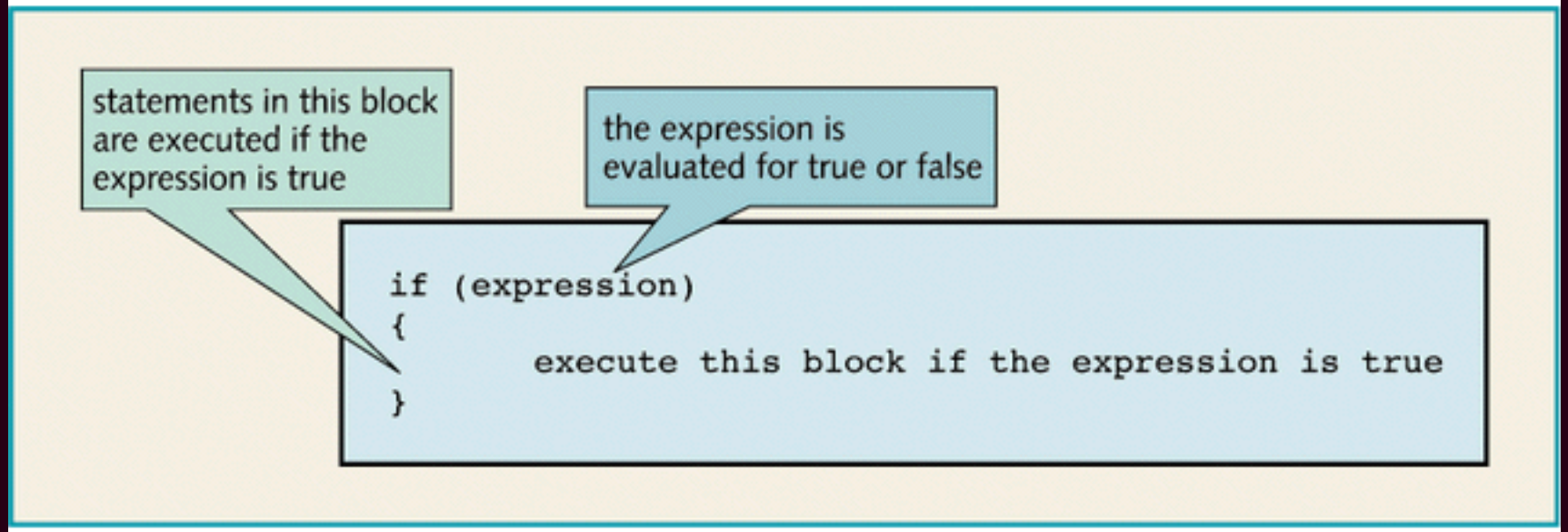

Figure 2-12 if statement format

Table 2-6 Java Logical Operators

Operator	Description
&&	And
==	equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
!	Not
!=	not equal to
||	Or

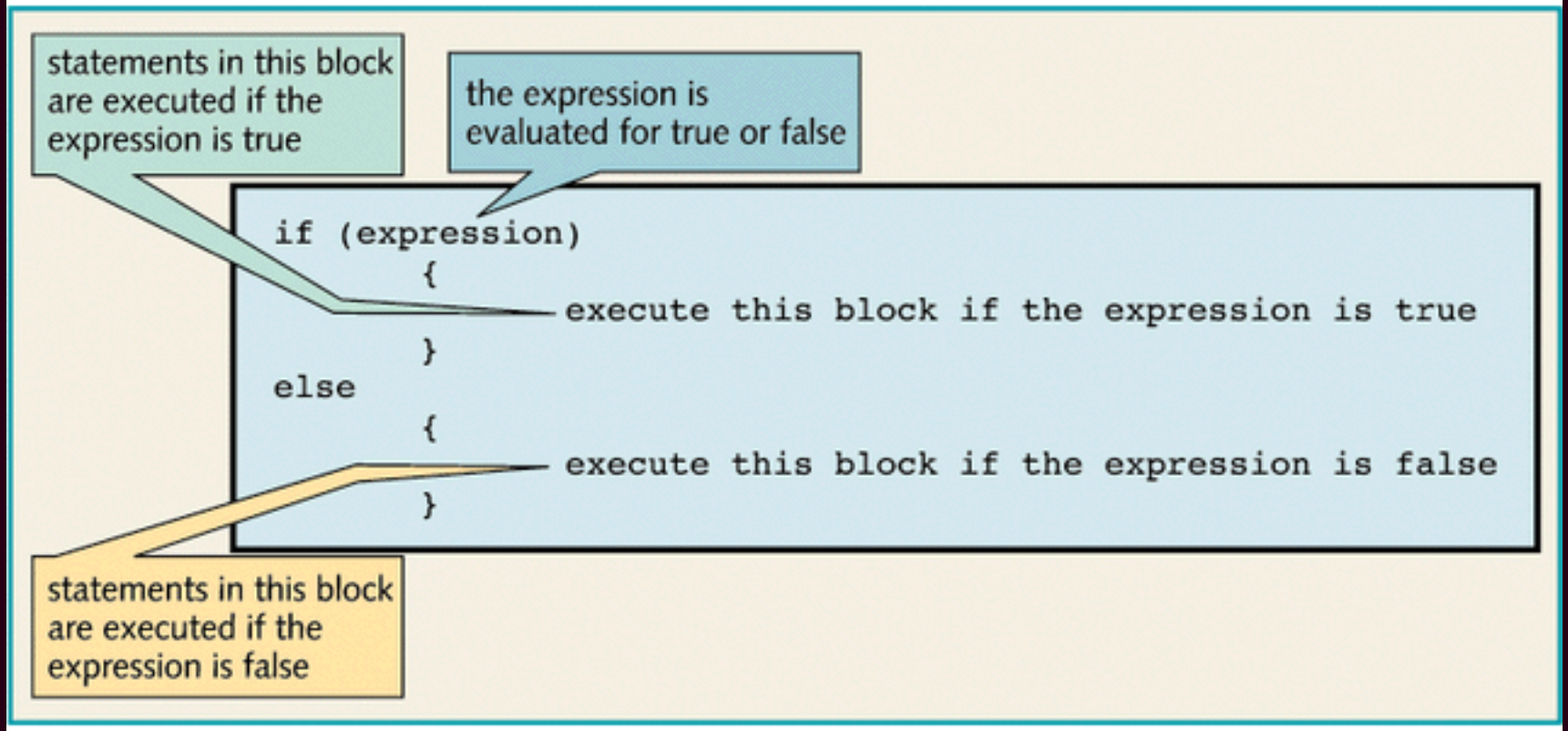

Figure 2-13 if-else statement format

Writing Decision-Making Statements

- Writing *if* Statements
  - You don't need the brackets for only one statement.
  - *if* statements can contain compound expressions
    - Two expressions joined using logical operators
      - OR → ||
      - AND → &&
  - Nested *if* statement
    - *if* statement written inside another *if* statement
    - the *else* statement always corresponds with the closest *if* statement

Writing Decision-Making Statements

- Using the Conditional Operator
  - Conditional operator (?)
    - Provides a shortcut to writing an if-else statement
    - Structure:
      - variable = expression ? value1:value2;
    - example

```
int smallerNumber = (a < b) ? a : b;
```

Writing Decision-Making Statements

```
1
2  import javax.swing.*;
3
4  public class Java1 {
5
6      public static void main(String[] args) {
7          int age = 21;
8          System.out.println(age > 50 ? "You are old": "You are young");
9      }
10 }
```

Writing Decision-Making Statements

- Writing *switch* Statements
  - Acts like a multiple-way *if* statement
  - Transfers control to one of several statements or blocks depending on the value of a variable
  - Used when there are more than two values to evaluate
  - Restrictions:
    - Each case evaluates a single variable for equality only
    - Variable being evaluated must be: char, byte, short, or int

Example of Switch Statement

```
char eventType;
…
switch (eventType)
{
      case ‘A’:
          eventCoordinator = “Dustin”;
          break;
      case ‘B’:
         eventCoordinator = “Heather”;
          break;
      case ‘C’:
         eventCoordinator = “Will”;
          break;
      default:
         eventCoordinator = “Invalid Entry”;
}
```

Writing Loops

- Loops
  - Provides for repeated execution of one or more statements until a terminating condition is reached
  - Three basic types:
    - while
    - do
    - for
  - What is the difference between the while and do loops?

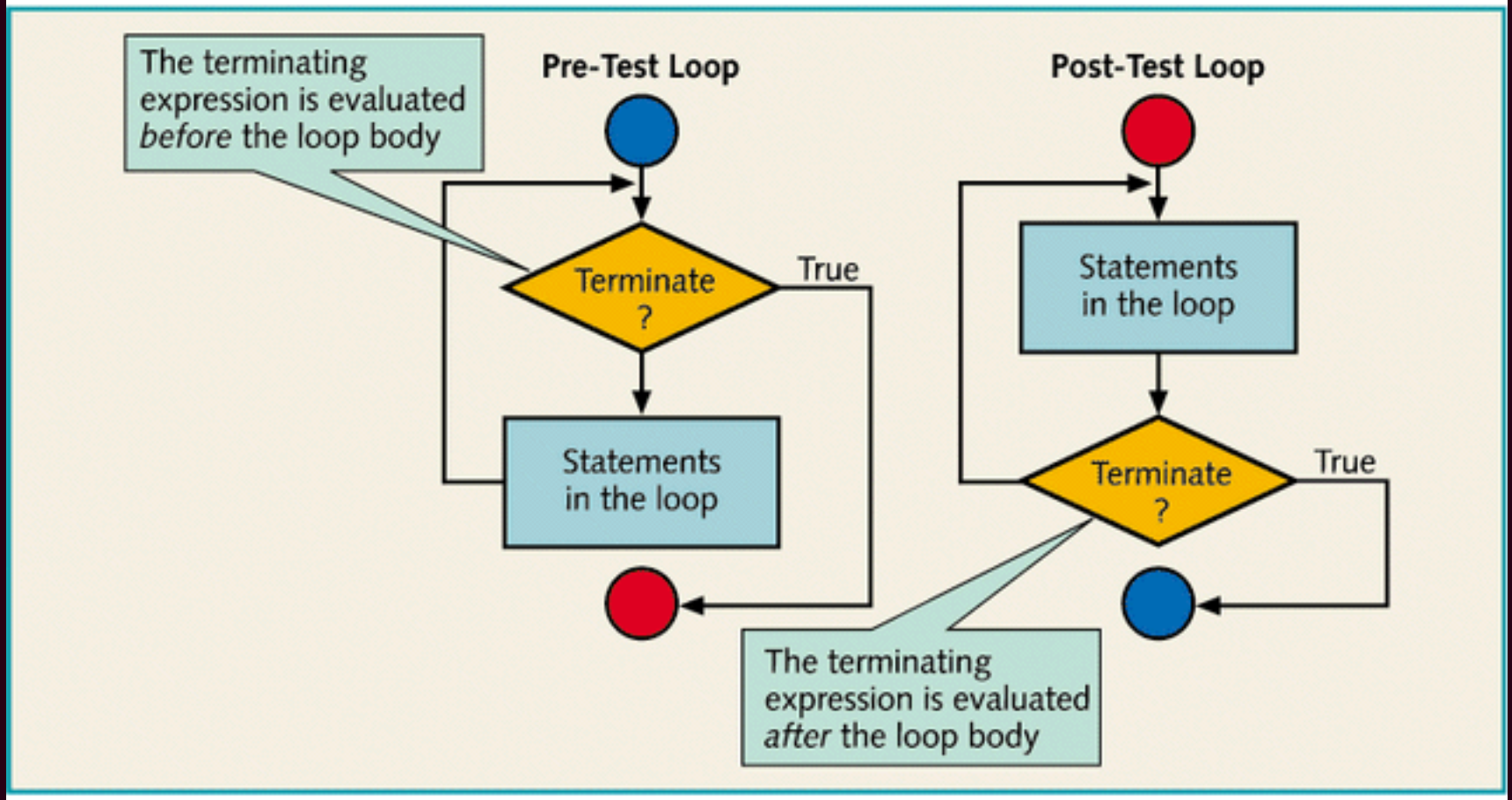

Figure 2-16 Loop structures

Writing Loops

- Writing *while* Loops
  - Loop counter
    - Counts number of times the loop is executed
  - Two kinds of loops
    - Pre-test loop
      - Tests terminating condition at the beginning of the loop
    - Post-test loop
      - Tests terminating condition at the end of the loop
  - Example of a while loop

```
while (count <= 10)  {
        System.out.println (count = + count);
        count++;
}
```

Writing Loops

- Writing *do* Loops
  - Loop counter
    - Counts number of times the loop is executed
  - Post-test loop
    - Tests terminating condition at the end of the loop
    - Forces execution of statements in the loop body at least once
  - Example:
    - do {
      System.out.println(“count = “ + count);
      count++;
      }
      while (count <= 10);

Writing Loops

- Writing *for* Loops
  - Loop counter
    - Counts number of times the loop is executed
  - Pre-test loop
    - Tests terminating condition at the beginning of the loop
  - Includes counter initialization and incrementing code in the statement itself
  - Example:
    - for (int count=1; count<=10; count++) {
      System.out.println(“count = “ + count);
      }

Writing Loops

- Writing Nested Loops
  - A loop within a loop
  - Useful for processing data arranged in rows and columns
  - Can be constructed using any combinations of *while*, *do*, and *for* loops

Declaring and Accessing Arrays

- Arrays
  - Allows the creation of a group of variables with the same data type
  - Consist of elements:
    - Each element behaves like a variable
  - Can be:
    - One dimensional
    - Multi-dimensional

Declaring and Accessing Arrays

- Using One-Dimensional Arrays
  - Keyword
    - new
      - Used to create a new array instance
      - int testScores[] = new int[10];
  - Use brackets ([]) and indices to denote elements:
    - testScores[5] = 75;
  - Note: Arrays in Java begin with 0 not 1
  - Used in the main method of applications.

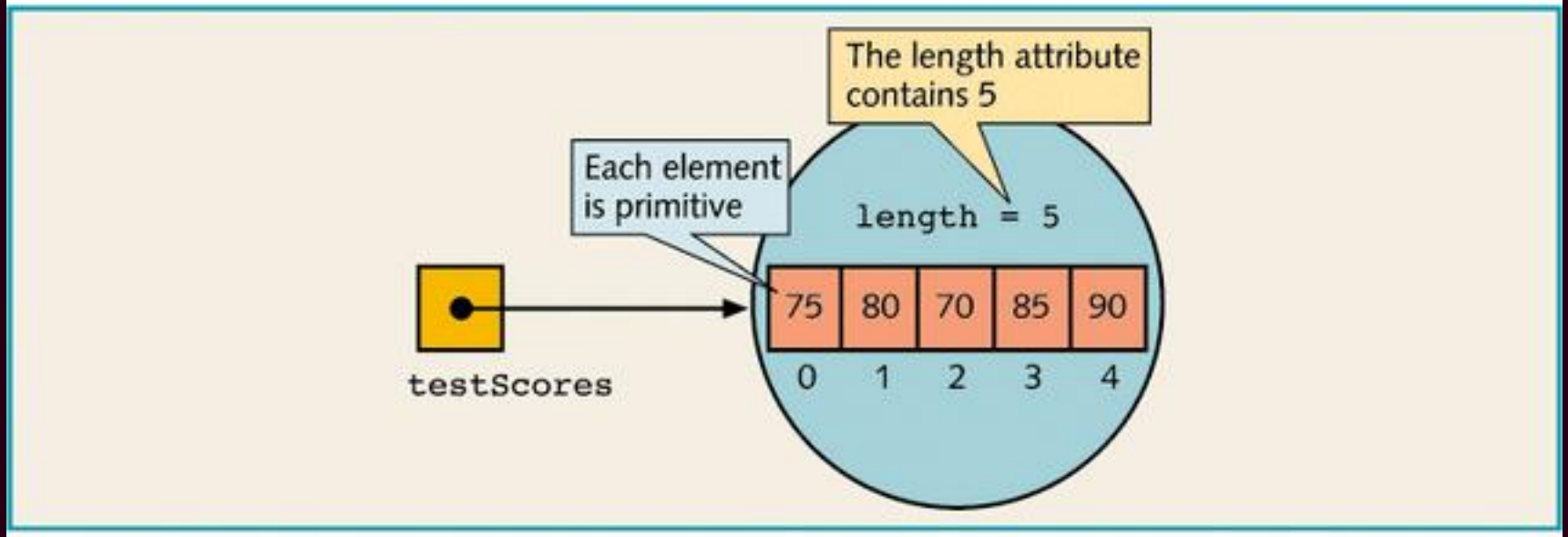

Figure 2-21 A five-element `int` array

Declaring and Accessing Arrays

- Using Multidimensional Arrays
  - Array of arrays
    - Three dimensions → cube
    - Four dimensions → ???
  - Each dimension has its own set of brackets:
    - testScoreTable[5][5] = 75;

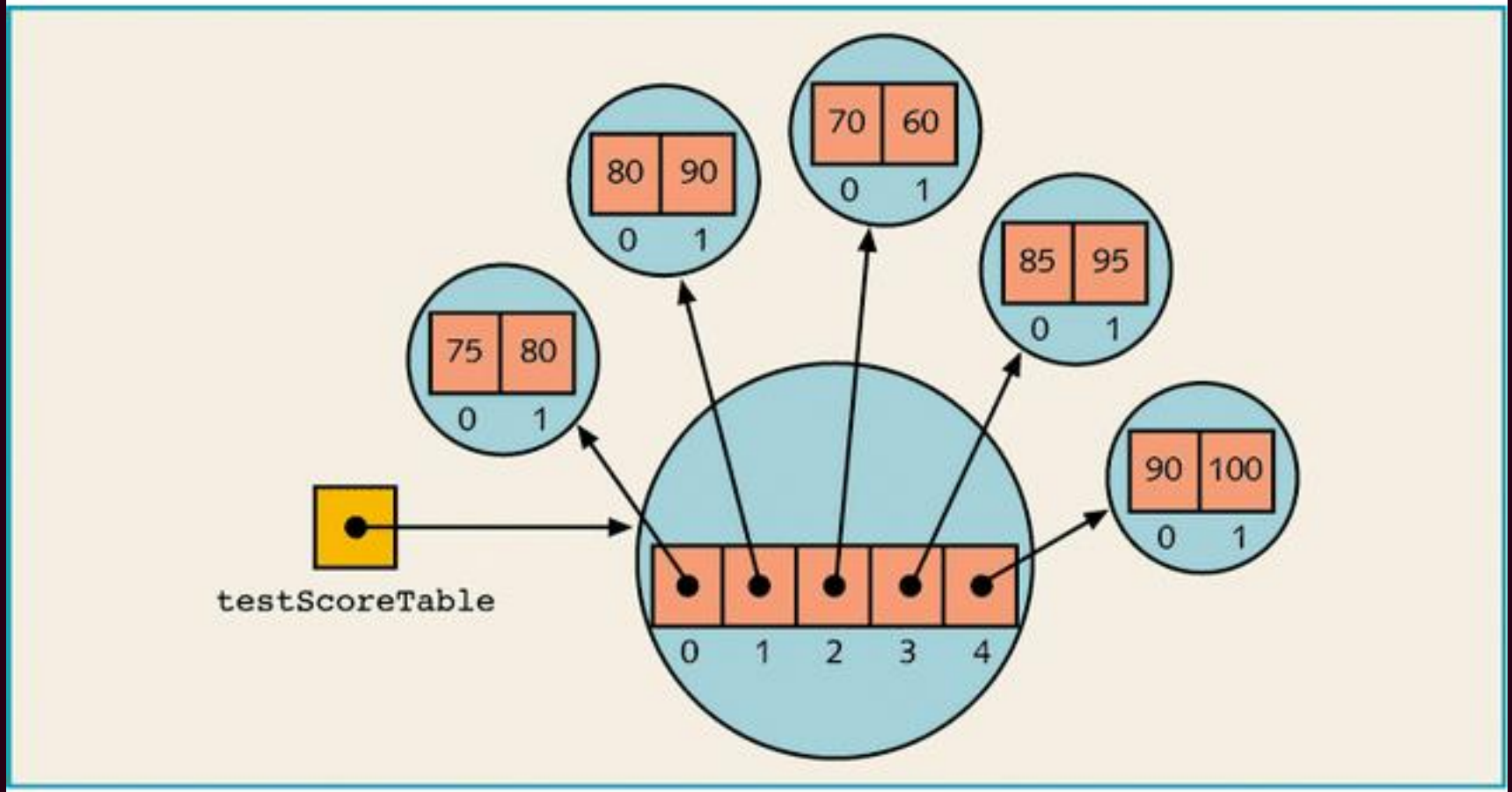

Figure 2-24 An array of arrays

Vectors

- Similar to arrays but do not need to declare size
- Can contain a list of different object types
- Structure
    - Vector vec = new Vector();
- Vector class has many helpful methods such as:
  - isEmpty()
  - indexOf(Object arg)
  - contains(Object arg)
  - See java docs for more info

Vectors

- Use the Enumeration class to iterate through a vector
  - Example:

```
Vector vec = new Vector();
vec.addElement(“one”);
vec.addElement(“two”);
vec.addElement(“three”);
Enumeration e = vec.elements();
while (e.hasMoreElements()) {
    System.out.println(“element = “ + e.nextElement());
}
```

  - Result:

```
element = one
element = two
element = three
```

ArrayList

- ArrayList is
  - a class in the standard Java libraries that can hold any type of object
  - an object that can grow and shrink while your program is running (unlike arrays, which have a fixed length once they have been created)
- In general, an ArrayList serves the same purpose as an array, except that an ArrayList can change length while the program is running

ArrayList

- The class ArrayList is implemented using an array as a private instance variable
  - When this hidden array is full, a new larger hidden array is created and the data is transferred to this new array

ArrayList

- In order to make use of the ArrayList class, it must first be imported
  - import java.util.ArrayList;

- An ArrayList is created and named in the same way as object of any class, except that you specify the base type as follows:

```
ArrayList<BaseType> aList =  new ArrayList<BaseType>();
```

ArrayList

Syntax

To construct an array list: `new ArrayList<typeName>()`

To access an element:
arraylistReference`.get(index)`
arraylistReference`.set(index, value)`

Variable type — Variable name — An array list object of size 0

```
ArrayList<String> friends = new ArrayList<String>();
```

```
friends.add("Cindy");
String name = friends.get(i);
friends.set(i, "Harry");
```

The add method appends an element to the array list, increasing its size.

Use the get and set methods to access an element.

The index must be ≥ 0 and < friends.size().

ArrayList

- Constructs an empty array list that can hold string.
  - ArrayList<String> names = new ArrayList<String>();
- Adds elements to the end.
  - names.add("Ann");
  - names.add("Cindy");
- Prints [Ann, Cindy].
  - System.out.println(names);
- Inserts an element at index 1. names is now [Ann, Bob, Cindy].
  - names.add(1, "Bob");
- Removes the element at index 0. names is now [Bob, Cindy].
- names.remove(0);

ArrayList

- Replaces an element with a different value. names is now [Bill, Cindy].
- names.set(0, "Bill"); (names is now [Bill, Cindy])
- Gets an element.
- String name = names.get(i);
- Gets the last element.
- String last = names.get(names.size() - 1);

ArrayList

- Constructs an array list holding the first ten squares.

ArrayList

```
ArrayList<Integer> squares =  new ArrayList<Integer>();
for (int i = 0; i < 10; i++)
        {
                squares.add(i * i);
        }
```

Array vs ArrayList

Array	ArrayList
Arrays are static in nature. Arrays are fixed length data structures. You can't change their size once they are created.	ArrayList is dynamic in nature. Its size is automatically increased if you add elements beyond its capacity.
Arrays can hold both primitives as well as objects.	ArrayList can hold only objects.
Arrays can be iterated only through for loop or for-each loop.	ArrayList provides iterators to iterate through their elements.
Arrays can be multi-dimensional.	ArrayList can't be multi-dimensional.

ArrayList vs Victor

ArrayList	Victor
Every method presented in the ArrayList is Asynchronous	Every method presented in Vector is synchronous.
ArrayList is not threads safe because multiple threats are allowed to operate at once.	Vector is threads safe because only one thread is allowed to operate at a time.
High performance because the threats are not required to wait.	Low performance comparing to ArrayList because Vector can only handle one threat at time.

ArrayList vs Victor

Table 3 Comparing Array and Array List Operations

Operation	Arrays	Array Lists
Get an element.	`x = values[4];`	`x = values.get(4);`
Replace an element.	`values[4] = 35;`	`values.set(4, 35);`
Number of elements.	`values.length`	`values.size()`
Number of filled elements.	`currentSize` (companion variable, see Section 7.1.4)	`values.size()`
Remove an element.	See Section 7.3.6.	`values.remove(4);`
Add an element, growing the collection.	See Section 7.3.7.	`values.add(35);`
Initializing a collection.	`int[] values = { 1, 4, 9 };`	No initializer list syntax; call `add` three times.