Fundamentals of Java

Session: 8

Arrays and Strings

Objectives

- Describe an array
- Explain declaration, initialization, and instantiation of a single-dimensional array
- Explain declaration, initialization, and instantiation of a multi-dimensional array
- Explain the use of loops to process an array
- Describe ArrayList and accessing values from an ArrayList
- Describe String and StringBuilder classes
- Explain command line arguments
- Describe Wrapper classes, autoboxing, and unboxing

Introduction

- Consider a situation where in a user wants to store the marks of ten students.
- For this purpose, the user can create ten different variables of type integer and store the marks in them.
- What if the user wants to store marks of hundreds or thousands of students?
- In such a case, one would need to create as many variables.
- This can be a very difficult, tedious, and time consuming task.
- Here, it is required to have a feature that will enable storing of all the marks in one location and access it with similar variable names.
- Array, in Java, is a feature that allows storing multiple values of similar type in the same variable.

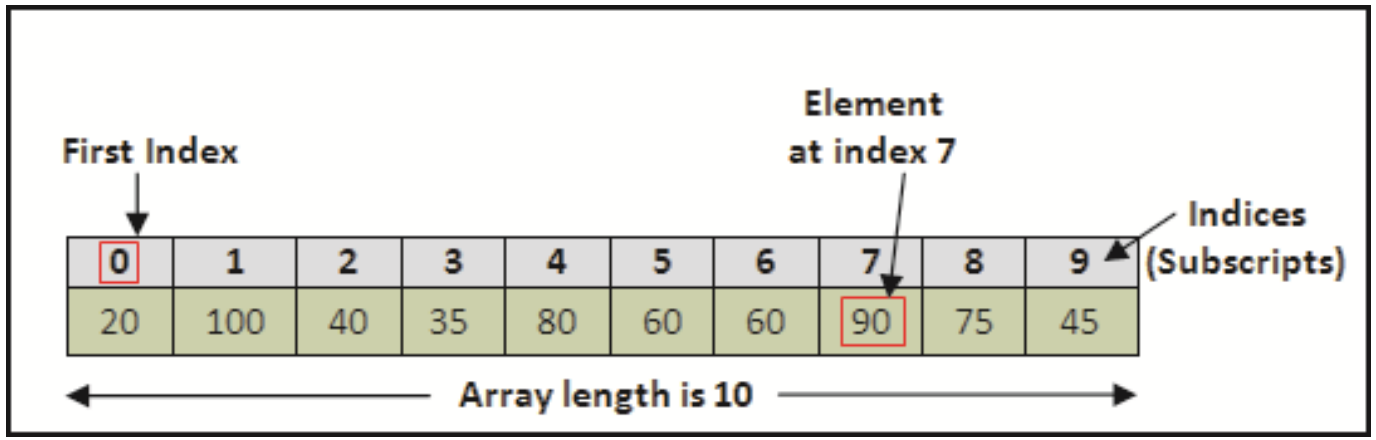
Introduction to Arrays 1-3

An array is a special data store that can hold a fixed number of values of a single type in contiguous memory locations.

It is implemented as objects.

The size of an array depends on the number of values it can store and is specified when the array is created.

After creation of an array, its size or length becomes fixed. Following figure shows an array of numbers:



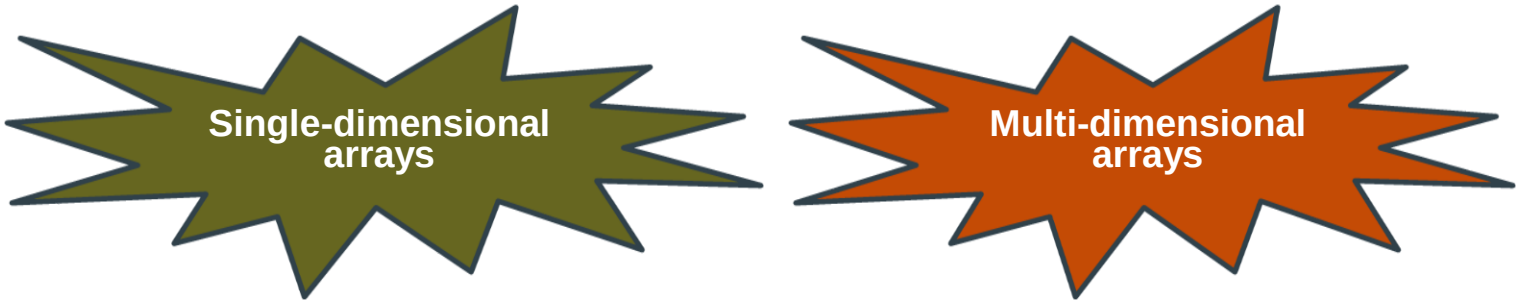
- The figure displays an array of ten integers storing values such as, 20, 100, 40, and so on.

Introduction to Arrays 2-3

- Each value in the array is called an element of the array.
- The numbers 0 to 9 indicate the index or subscript of the elements in the array.
- The length or size of the array is 10. The first index begins with zero.
- Since the index begins with zero, the index of the last element is always length - 1.
- The last, that is, tenth element in the given array has an index value of 9.
- Each element of the array can be accessed using the subscript or index.
- Array can be created from primitive data types such as `int`, `float`, `boolean` as well as from reference type such as object.
- The array elements are accessed using a single name but with different subscripts.
- The values of an array are stored at contiguous locations in memory.
- This induces less overhead on the system while searching for values.

Introduction to Arrays 3-3

- The use of arrays has the following benefits:
  - Arrays are the best way of operating on multiple data elements of the same type at the same time.
  - Arrays make optimum use of memory resources as compared to variables.
  - Memory is assigned to an array only at the time when the array is actually used. Thus, the memory is not consumed by an array right from the time it is declared.
- Arrays in Java are of the following two types:



Declaring, Instantiating, and Initializing Single-dimensional Array 1-8

A single-dimensional array has only one dimension and is visually represented as having a single column with several rows of data.

Each element is accessed using the array name and the index at which the element is located.

- Following figure shows the array named marks and its elements with their values and indices:

marks[4]	
Element	Value
marks[0]	65
marks[1]	47
marks[2]	75
marks[3]	50

- The size or length of the array is specified as 4 in square brackets '[]'.
- **`marks[0]`** indicates the first element in the array.
- **`marks[3]`**, that is, **`marks[length-1]`** indicates the last element of the array.
- Notice that there is no element with index 4.

Declaring, Instantiating, and Initializing Single-dimensional Array 2-8

- An attempt to write **`marks[4]`** will issue an exception.

An exception is an abnormal event that occurs during the program execution and disrupts the normal flow of instructions.

- Array creation involves the following tasks:

Declaring an Array

- Declaring an array notifies the compiler that the variable will contain an array of the specified data type. It does not create an array.
- The syntax for declaring a single-dimensional array is as follows:

Syntax

```
datatype[] <array-name>;
```

where,

`datatype`: Indicates the type of elements that will be stored in the array.

`[]`: Indicates that the variable is an array.

`array-name`: Indicates the name by which the elements of the array will be accessed.

Declaring, Instantiating, and Initializing Single-dimensional Array 3-8

- For example,

```
int[] marks;
```

- Similarly, arrays of other types can also be declared as follows:

```
byte[] byteArray;
float[] floatsArray;
boolean[] booleanArray;
char[] charArray;
String[] stringArray;
```

Instantiating an Array

- Since array is an object, memory is allocated only when it is instantiated.
- The syntax for instantiating an array is as follows:

Syntax

```
datatype[] <array-name> = new datatype[size];
```

where,

`new`: Allocates memory to the array.

Declaring, Instantiating, and Initializing Single-dimensional Array 4-8

`size`: Indicates the number of elements that can be stored in the array.

- For example,

```
int[] marks = new int[4];
```

Initializing an Array

- Since, array is an object that can store multiple values, array needs to be initialized with the values to be stored in it.
- Array can be initialized in the following two ways:

During creation:

- To initialize a single-dimensional array during creation, one must specify the values to be stored while creating the array as follows: `int[ ] marks = {65, 47, 75, 50};`
- Notice that while initializing an array during creation, the `new` keyword or size is not required.
- This is because all the elements to be stored have been specified and accordingly the memory gets automatically allocated based on the number of elements.

Declaring, Instantiating, and Initializing Single-dimensional Array 5-8

After creation:

- A single-dimensional array can also be initialized after creation and instantiation.
- In this case, individual elements of the array need to be initialized with appropriate values.
- For example,
  - `int[] marks = new int[4];`
  - `marks[0] = 65;`
  - `marks[1] = 47;`
  - `marks[2] = 75;`
  - `marks[3] = 50;`
- Notice that in this case, the array must be instantiated and size must be specified.
- This is because, the actual values are specified later and to store the values, memory must be allocated during creation of the array.

- Another way of creating an array is to split all the three stages as follows:

```
int marks[]; // declaration
marks = new int[4]; // instantiation
marks[0] = 65; // initialization
```

Declaring, Instantiating, and Initializing Single-dimensional Array 6-8

- Following code snippet demonstrates an example of single-dimensional array:

```
package session8;
public class OneDimension {
  //Declare a single-dimensional array named marks
  int marks[]; // line 1

  /**
   * Instantiates and initializes a single-dimensional array
   *
   * @return void
   */
  public void storeMarks() {
    // Instantiate the array
    marks = new int[4]; // line 2
    System.out.println("Storing Marks. Please wait...");

    // Initialize array elements
    marks[0] = 65; // line 3
    marks[1] = 47;
    marks[2] = 75;
    marks[3] = 50;
  }
```

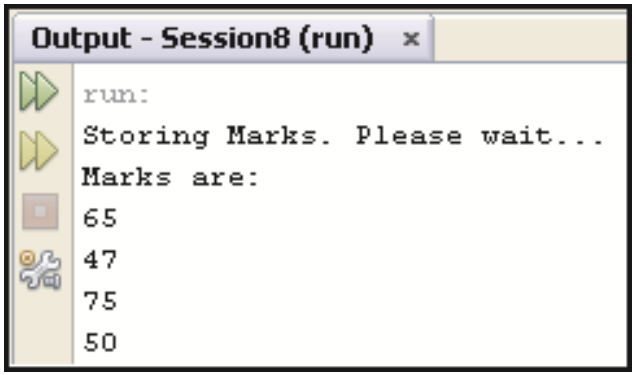
Declaring, Instantiating, and Initializing Single-dimensional Array 7-8

```
/**
 * Displays marks from a single-dimensional array
 *
 * @return void
 */
public void displayMarks() {
  System.out.println("Marks are:");

  // Display the marks
  System.out.println(marks[0]);
  System.out.println(marks[1]);
  System.out.println(marks[2]);
  System.out.println(marks[3]);
}

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {

//Instantiate class OneDimension
OneDimension oneDimenObj = new OneDimension(); //line 4
```

Declaring, Instantiating, and Initializing Single-dimensional Array 8-8

```
    //Invoke the storeMarks() method
    oneDimenObj.storeMarks(); // line 5

    //Invoke the displayMarks() method
    oneDimenObj.displayMarks(); // line 6
    }
}
```

- The class **OneDimension** consists of an array named **marks[]** declared in line 1.
- To instantiate and initialize the array elements, the method **storeMarks()** is created.
- To display the array elements, the **displayMarks()** method is created.
- Following figure shows the output of the code:



Declaring, Instantiating, and Initializing Multi-dimensional Array 1-7

A multi-dimensional array in Java is an array whose elements are also arrays. This allows the rows to vary in length.

- The syntax for declaring and instantiating a multi-dimensional array is as follows:

Syntax

```
datatype[][] <array-name> = new datatype [rowsize][colsize];
```

where,

`datatype`: Indicates the type of elements that will be stored in the array.

`rowsize and colsize`: Indicates the number of rows and columns that the array will contain.

`new`: Keyword used to allocate memory to the array elements.

- For example,

```
int[][] marks = new int[4][2];
```

- The array named `marks` consists of four rows and two columns.

Declaring, Instantiating, and Initializing Multi-dimensional Array 2-7

- A multi-dimensional array can be initialized in the following two ways:

During creation

- To initialize a multi-dimensional array during creation, one must specify the values to be stored while creating the array as follows:

```
int[][] marks = {{23,65}, {42,47}, {60,75}, {75,50}};
```

- While initializing an array during creation, the elements in rows are specified in a set of curly brackets separated by a comma delimiter.
- Also, the individual rows are separated by a comma separator.
- This is a two-dimensional array that can be represented in a tabular form as shown in the following figure:

Rows	Columns	
	0	1
0	23	65
1	42	47
2	60	75
3	75	50

Declaring, Instantiating, and Initializing Multi-dimensional Array 3-7

After creation

A multi-dimensional array can also be initialized after creation and instantiation.

In this case, individual elements of the array need to be initialized with appropriate values.

Each element is accessed with a row and column subscript.

- For example,

```
int[][] marks = new int[4][2];
marks[0][0] = 23;  // first row, first column
marks[0][1] = 65;  // first row, second column
marks[1][0] = 42;
marks[1][1] = 47;
marks[2][0] = 60;
marks[2][1] = 75;
marks[3][0] = 75;
marks[3][1] = 50;
```

Declaring, Instantiating, and Initializing Multi-dimensional Array 4-7

- The element **23** is said to be at position (0,0), that is, first row and first column.
- Therefore, to store or access the value 23, one must use the syntax **`marks[0][0]`**.
- Similarly, for other values, the appropriate row-column combination must be used.
- Similar to row index, column index also starts at zero. Therefore, in the given scenario, an attempt to write **`marks[0][2]`** would result in an exception as the column size is 2 and column indices are 0 and 1.
- Following code snippet demonstrates an example of two-dimensional array:

```
package session8;
public class TwoDimension {

  //Declare a two-dimensional array named marks
  int marks[][]; //line 1

  /**
   * Stores marks in a two-dimensional array
   *
   * @return void
   */
  public void storeMarks() {
```

Declaring, Instantiating, and Initializing Multi-dimensional Array 5-7

```
    // Instantiate the array
    marks = new int[4][2]; // line 2
    System.out.println("Storing Marks. Please wait...");

    // Initialize array elements
    marks[0][0] = 23; // line 3
    marks[0][1] = 65;
    marks[1][0] = 42;
    marks[1][1] = 47;
    marks[2][0] = 60;
    marks[2][1] = 75;
    marks[3][0] = 75;
    marks[3][1] = 50;
  }

  /**
   * Displays marks from a two-dimensional array
   *
   * @return void
   */
  public void displayMarks() {
```

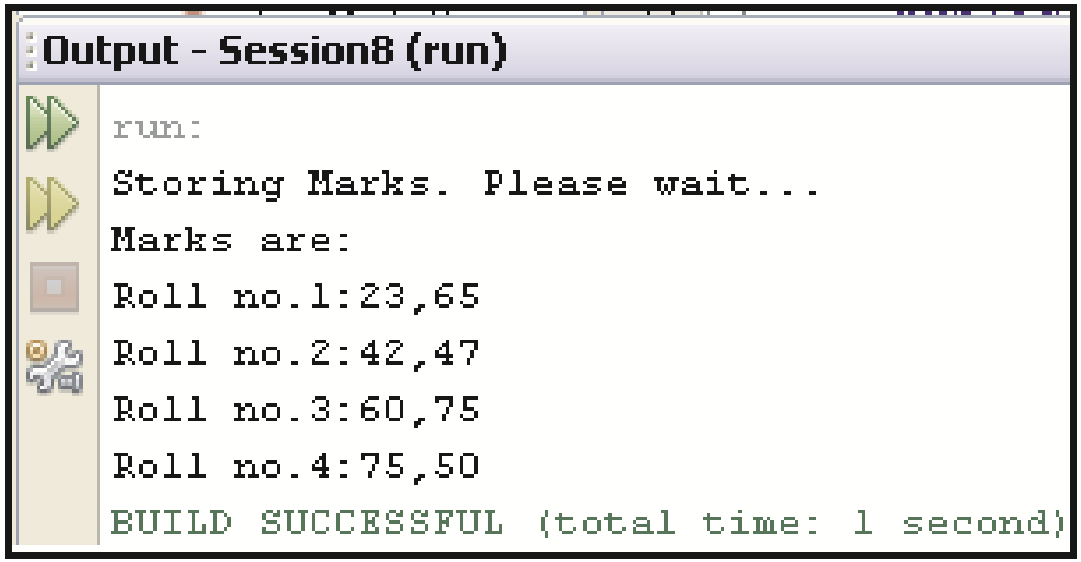
Declaring, Instantiating, and Initializing Multi-dimensional Array 6-7

```
/**
 * Displays marks from a two-dimensional array
 *
 * @return void
 */
public void displayMarks() {

  System.out.println("Marks are:"); // Display the marks
  System.out.println("Roll no.1:" + marks[0][0]+ "," + marks[0][1]);
  System.out.println("Roll no.2:" + marks[1][0]+ "," + marks[1][1]);
  System.out.println("Roll no.3:" + marks[2][0]+ "," + marks[2][1]);
  System.out.println("Roll no.4:" + marks[3][0]+ "," + marks[3][1]);
}

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {

  //Instantiate class TwoDimension
  TwoDimension twoDimenObj = new TwoDimension(); // line 4
```

Declaring, Instantiating, and Initializing Multi-dimensional Array 7-7

```
        //Invoke the storeMarks() method
        twoDimenObj.storeMarks();

        //Invoke the displayMarks() method
        twoDimenObj.displayMarks();
    }
}
```

- Following figure shows the output of the code, that is, marks of four students are displayed from the array **`marks[][]`**:



Using Loops to Process and Initialize an Array 1-6

- A user can use loops to process and initialize an array.
- Following code snippet depicts the revised **`displayMarks()`** method of the single-dimensional array named **`marks[]`**:

```
...
public void displayMarks() {
  System.out.println("Marks are:");

  // Display the marks using for loop
  for(int count = 0; count < marks.length; count++) {
     System.out.println(marks[count]);
  }
}
...
```

- In the code, a `for` loop has been used to iterate the array from zero to **`marks.length`**.
- The property, `length`, of the array object is used to obtain the size of the array.
- Within the loop, each element is displayed by using the element name and the variable **`count`**, that is, **`marks[count]`**.

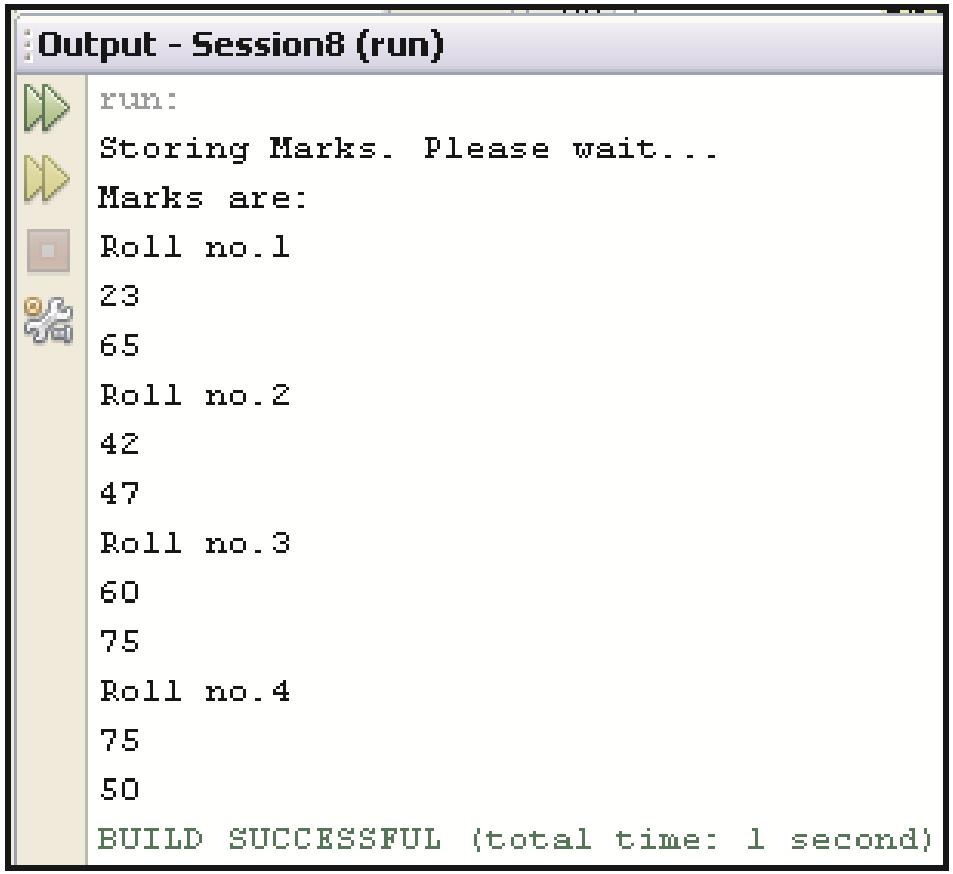
Using Loops to Process and Initialize an Array 2-6

- Following code snippet depicts the revised **displayMarks()** method of the two-dimensional array **marks[][]**:

```
...
public void displayMarks(){

  System.out.println("Marks are:");

  // Display the marks using nested for loop

  // outer loop
  for (int row = 0; row < marks.length; row++) {

    System.out.println("Roll no." + (row+1));

    // inner loop
    for (int col = 0; col < marks[row].length; col++) {
      System.out.println(marks[row][col]);
    }
  }
}
...
```

Using Loops to Process and Initialize an Array 3-6

- The outer loop keeps track of the number of rows and inner loop keeps track of the number of columns in each row.
- Following figure shows the output of the two-dimensional array **`marks[][]`**, after using the `for` loop:



Using Loops to Process and Initialize an Array 4-6

- One can also use the enhanced `for` loop to iterate through an array.
- Following code snippet depicts the modified **`displayMarks()`** method of single-dimensional array **`marks[]`** using the enhanced `for` loop:

```
...
public void displayMarks() {

  System.out.println("Marks are:");

  // Display the marks using enhanced for loop
  for(int value:marks) {

    System.out.println(value);
  }
}
...
```

- The loop will print all the values of **`marks[]`** array till **`marks.length`** without having to explicitly specify the initializing and terminating conditions for iterating through the loop.

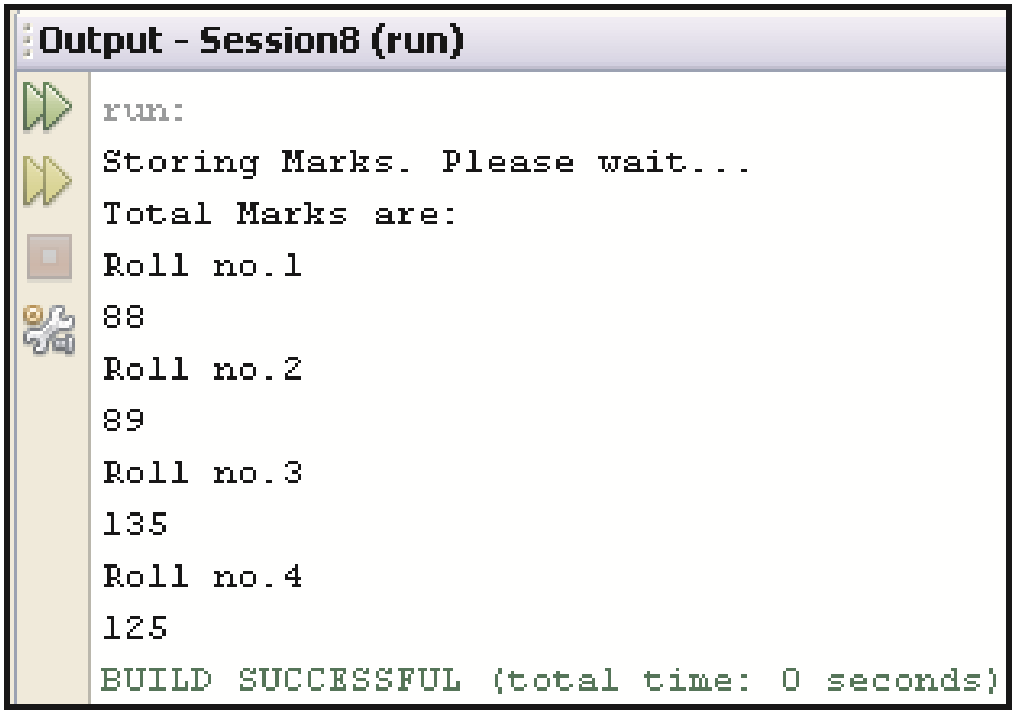
Using Loops to Process and Initialize an Array 5-6

- Following code snippet demonstrates the calculation of total marks of each student by using the `for` loop and the enhanced `for` loop together with the two-dimensional array **`marks[][]`**:

```
...
public void totalMarks() {

  System.out.println("Total Marks are:");

  // Display the marks using for loop and enhanced for loop
  for (int row = 0; row < marks.length; row++) {
    System.out.println("Roll no." + (row+1));
    int sum = 0;

    // enhanced for loop
    for(int value:marks[row]) {
      sum = sum + value;
    }
    System.out.println(sum);
  }
}
...
```

Using Loops to Process and Initialize an Array 6-6

- The enhanced `for` loop is used to iterate through the columns of the row selected in the outer loop using **`marks[row]`**.
- The code **`sum = sum + value`** will add up the values of all columns of the currently selected row.
- The selected row is indicated by the subscript variable **`row`**.
- Following figure shows the sum of the values of the two-dimensional array named **`marks[][]`** using `for` loop and enhanced `for` loop together:



Initializing an ArrayList 1-8

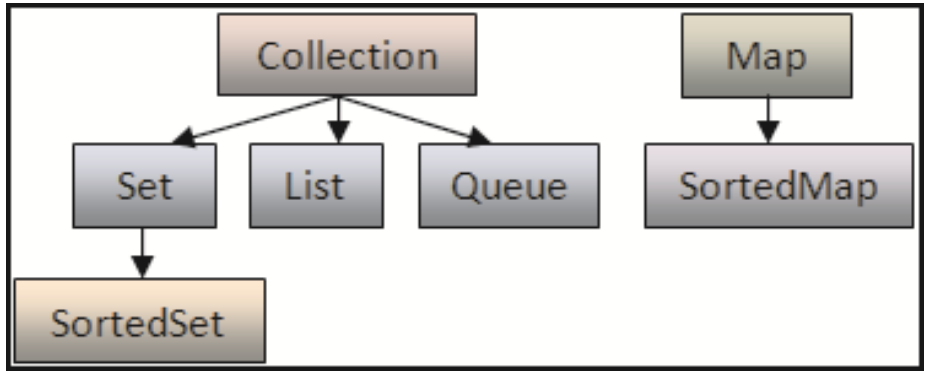
One major disadvantage of an array is that its size is fixed during creation. The size cannot be modified later.

To resolve this issue, it is required to have a construct to which memory can be allocated based on requirement.

Also, addition and deletion of values can be performed easily. Java provides the concept of collections to address this problem.

A collection is a single object that groups multiple elements into a single unit.

- The core `Collection` interfaces that encapsulate different types of collections are shown in the following figure:



Initializing an ArrayList 2-8

- The general-purpose implementations are summarized in the following table:

Interfaces	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set	HashSet	-	TreeSet	-	LinkedHashSet
List	-	ArrayList	-	LinkedList	-
Queue	-	-	-	-	-
Map	HashMap	-	TreeMap	-	LinkedHashMap

- The `ArrayList` class is a frequently used collection that has the following characteristics:

It is flexible and can be increased or decreased in size as needed.

Provides several useful methods to manipulate the collection.

Insertion and deletion of data is simpler.

It can be traversed by using `for` loop, enhanced `for` loop, or other iterators.

Initializing an ArrayList 3-8

- `ArrayList` extends `AbstractList` and implements the interfaces such as `List`, `Cloneable`, and `Serializable`.
- The capacity of an `ArrayList` grows automatically.
- It stores all elements including null.
- The `Arraylist` collection provides methods to manipulate the size of the array.

- Following table lists the constructors of `ArrayList` class:

Constructor	Description
`ArrayList( )`	Creates an empty array list.
`ArrayList(Collection c)`	Creates an array list initialized with the elements of the collection `c`.
`ArrayList(int capacity)`	Creates an array list with a specified initial capacity. Capacity is the size of the underlying array used to store the elements. The capacity can grow automatically as elements are added to an array list.

Initializing an ArrayList 4-8

- `ArrayList` consists of several methods for adding elements.
- These methods can be broadly divided into following two categories:

Methods that append one or more elements to the end of the list.

Methods that insert one or more elements at a position within the list.

- Following table lists the methods of `ArrayList` class:

Method	Description
`void add(int index, Object element)`	Inserts the specified element at the given index in this list. If `index>=size()` or `index<0`, it throws `IndexOutOfBoundsException`.
`boolean add(Object o)`	Appends the specified element to the end of this list.
`boolean addAll(Collection c)`	Appends all elements in the specified collection to the end of this list. If the specified collection is `null`, it throws `NullPointerException`.
`boolean addAll(int index, Collection c)`	Inserts all of the elements in the specified collection into this list, starting at the specified index. If the collection is `null`, it throws `NullPointerException`.
`void clear()`	Removes all of the elements from this list.
`Object clone()`	Returns a copy of the `ArrayList`.

Initializing an ArrayList 5-8

Method	Description
`boolean contains(Object o)`	Returns true if and only if the list contains the specified element.
`void ensureCapacity(int minCapacity)`	Increases the capacity of the `ArrayList`, if required, to ensure that it can store at least as many number of elements as indicated by the minimum capacity.
`Object get(int index)`	Returns the element at the specified index in this list. If `index>=size()` or `index<0`, it throws `IndexOutOfBoundsException`.
`int indexOf(Object o)`	Returns the index of the first occurrence of the specified element in the list. If the element is not found, it returns `-1`.
`int lastIndexOf(Object o)`	Returns the index of the last occurrence of the specified element in this list. If the element is not found, it returns `-1`.
`Object remove(int index)`	Removes the element at the specified index in this list. If `index >= size()` or `index < 0`, it throws `IndexOutOfBoundsException`.
`protected void removeRange(int fromIndex, int toIndex)`	Removes all the elements between `fromIndex`, inclusive and `toIndex`, exclusive of the list.
`Object set(int index, Object element)`	Replaces the element at the specified index in this list with the newly specified element. If `index >= size()` or `index < 0`, it throws `IndexOutOfBoundsException`.

Initializing an ArrayList 6-8

Method	Description
`int size()`	Returns the number of elements in this list.
`Object[] toArray()`	Returns an array containing all of the elements in the list in the correct order. If the array is `null`, it throws `NullPointerException`.
`Object[] toArray(Object[] a)`	Returns an array containing all of the elements in the list in the correct order. The type of the returned array is same as that of the specified array.
`void trimToSize()`	Trims the capacity of the `ArrayList` to the list's actual size.

- To traverse an `ArrayList`, one can use one of the following approaches:
  - A `for` loop
  - An enhanced `for` loop
  - `Iterator`
  - `ListIterator`

Initializing an ArrayList 7-8

`Iterator` interface provides methods for traversing a set of data.

It can be used with arrays as well as various classes of the Collection framework.

- The `Iterator` interface provides the following methods for traversing a collection:

`next()`	• This method returns the next element of the collection.
`hasNext()`	• This method returns true if there are additional elements in the collection.
`remove()`	• This method removes the element from the list while iterating through the collection.

- There are no specific methods in the `ArrayList` class for sorting.
- However, one can use the `sort()` method of the `Collections` class to sort an `ArrayList`.

Initializing an ArrayList 8-8

- The syntax for using the `sort()` method is as follows:

Syntax

```
Collections.sort(<list-name>);
```

- Following code snippet demonstrates instantiation and initialization of an `ArrayList`:

```
ArrayList marks = new ArrayList(); // Instantiate an ArrayList
marks.add(67); // Initialize an ArrayList
marks.add(50);
```

Accessing Values in an ArrayList 1-5

- An `ArrayList` can be iterated by using the `for` loop or by using the `Iterator` interface.
- Following code snippet demonstrates the use of `ArrayList` named **`marks`** to add and display marks of students:

```
package session8;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
public class ArrayLists{

  // Create an ArrayList instance
  ArrayList marks = new ArrayList(); // line 1

  /**
   * Stores marks in ArrayList
   *
   * @return void
   */
  public void storeMarks(){
    System.out.println("Storing marks. Please wait...");
    marks.add(67); // line 2
```

Accessing Values in an ArrayList 2-5

```
    marks.add(50);
    marks.add(45);
    marks.add(75);
  }

  /**
  * Displays marks from ArrayList
  *
  * @return void
  */
  public void displayMarks() {

    System.out.println("Marks are:");

    // iterating the list using for loop
    System.out.println("Iterating ArrayList using for loop:");
    for (int i = 0; i < marks.size(); i++) {
      System.out.println(marks.get(i));
    }
    System.out.println("-------------------------------------");
```

Accessing Values in an ArrayList 3-5

```
    // Iterate the list using Iterator interface
    Iterator imarks = marks.iterator(); // line 3
    System.out.println("Iterating ArrayList using Iterator:");
    while (imarks.hasNext()) { // line 4
      System.out.println(imarks.next()); // line 5
    }
    System.out.println("-----------------------------------");

    // Sort the list
    Collections.sort(marks); // line 6
    System.out.println("Sorted list is: " + marks);
  }

  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {

    //Instantiate the class OneDimension
    ArrayLists obj = new ArrayLists(); // line 7
```

Accessing Values in an ArrayList 4-5

```
        //Invoke the storeMarks() method
        obj.storeMarks();

        //Invoke the displayMarks() method
        obj.displayMarks();
    }
}
```

- The `Iterator` object **`imarks`** is instantiated in line 3 and attached with the marks `ArrayList` using **`marks.iterator()`**.
- It is used to iterate through the collection.
- The `Iterator` interface provides the `hasNext()` method to check if there are any more elements in the collection as shown in line 4.
- The method, `next()` is used to traverse to the next element in the collection.
- The retrieved element is then displayed to the user in line 5.
- The static method, `sort()` of `Collections` class is used to sort the `ArrayList` **`marks`** in line 6 and print the values on the screen.

Accessing Values in an ArrayList 5-5

- Following figure shows the output of the code:

```
Output - Session8 (run)
run:
Storing marks. Please wait...
Marks are:
Iterating ArrayList using for loop:
67
50
45
75
---------------------------------------
Iterating ArrayList using Iterator:
67
50
45
75
---------------------------------------
Sorted list is: [45, 50, 67, 75]
```

The values of an `ArrayList` can also be printed by simply writing `System.out.println("Marks are:"+ marks)`.

In this case, the output would be: `Marks are:[67, 50, 45, 75]`.

Introduction to Strings

- Consider a scenario, where in a user wants to store the name of a person.
- One can create a character array as shown in the following code snippet:

```
char[] name = {'J','u','l','i','a'}
```

- Similarly, to store names of multiple persons, one can create a two-dimensional array.
- However, the number of characters in an array must be fixed during creation.
- This is not possible since the names of persons may be of variable sizes.
- Also, manipulating the character array would be tedious and time consuming.
- Java provides the `String` data type to store multiple characters without creating an array.

Strings 1-3

String literals such as "Hello" in Java are implemented as instances of the String class.

Strings are constant and immutable, that is, their values cannot be changed once they are created.

String buffers allow creation of mutable strings.

A simple String instance can be created by enclosing a string literal inside double quotes as shown in the following code snippet:

```
...
String name = "Mary";
// This is equivalent to:
char name[] = {'M', 'a', 'r', 'y'};
...
```

- An instance of a String class can also be created using the new keyword, as shown in code snippet:

```
String str = new String();
```

- The code creates a new object of class String, and assigns its reference to the variable **str**.

Strings 2-3

- Java also provides special support for concatenation of strings using the plus (+) operator and for converting data of other types to strings as depicted in the following code snippet:

```
...
String str = "Hello"; String str1 = "World";
// The two strings can be concatenated by using the operator '+'
System.out.println(str + str1);

// This will print 'HelloWorld' on the screen
...
```

- One can convert a character array to a string as depicted in the following code snippet:

```
char[] name = {'J', 'o', 'h', 'n'}; String empName = new String(name);
```

The `java.lang.String` class is a `final` class, that is, no class can extend it.

The `java.lang.String` class differs from other classes, in that one can use '+=' and '+' operators with `String` objects for concatenation.

Strings 3-3

- If the string is not likely to change later, one can use the `String` class.
- Thus, a `String` class can be used for the following reasons:

String is immutable and so it can be safely shared between multiple threads.

The threads will only read them, which is normally a thread safe operation.

- If the string is likely to change later and it will be shared between multiple threads, one can use the `StringBuffer` class.
- The use of `StringBuffer` class ensures that the string is updated correctly.
- However, the drawback is that the method execution is comparatively slower.
- If the string is likely to change later but will not be shared between multiple threads, one can use the `StringBuilder` class.
- The `StringBuilder` class can be used for the following reasons:

It allows modification of the strings without the overhead of synchronization.

Methods of `StringBuilder` class execute as fast as, or faster, than those of the `StringBuffer` class

Working with String Class 1-7

- Some of the frequently used methods of `String` class are as follows:

`length(String str)`

- The `length()` method is used to find the length of a string. For example,
  - `String str = "Hello";`
  - `System.out.println(str.length()); // output: 5`

`charAt(int index)`

- The `charAt()` method is used to retrieve the character value at a specific index.
- The index ranges from zero to `length() - 1`.
- The index of the first character starts at zero. For example,
  - `System.out.println(str.charAt(2)); // output: 'l'`

`concat(String str)`

- The `concat()` method is used to concatenate a string specified as argument to the end of another string.
- If the length of the string is zero, the original String object is returned, otherwise a new `String` object is returned.
  - `System.out.println(str.concat("World"));`
    `// output: 'HelloWorld'`

Working with String Class 2-7

compareTo(String str)

- The `compareTo()` method is used to compare two `String` objects.
- The comparison returns an integer value as the result.
- The comparison is based on the Unicode value of each character in the strings.
- The result will return a negative value, if the argument string is alphabetically greater than the original string.
- The result will return a positive value, if argument string is alphabetically lesser than the original string and the result will return a value of zero, if both the strings are equal.
- For example,
  - `System.out.println(str.compareTo("World"));`
    `// output: -15`
- The output is **15** because, the second string **"World"** begins with '**W**' which is alphabetically greater than the first character '**H**' of the original string, **str**.
- The difference between the position of '**H**' and '**W**' is **15**.
- Since '**H**' is smaller than '**W**', the result will be **-15**.

indexOf(String str)

- The `indexOf()` method returns the index of the first occurrence of the specified character or string within a string.
- If the character or string is not found, the method returns `-1`. For example,
  - `System.out.println(str.indexOf("e"));  // output: 1`

Working with String Class 3-7

lastIndexOf(String str)

- The `lastIndexOf()` method returns the index of the last occurrence of a specified character or string from within a string.
- The specified character or string is searched backwards that is the search begins from the last character. For example,
  - `System.out.println(str.lastIndexOf("l")); // output: 3`

replace(char old, char new)

- The `replace()` method is used to replace all the occurrences of a specified character in the current string with a given new character.
- If the specified character does not exist, the reference of original string is returned. For example,
  - `System.out.println(str.replace('e','a')); // output: 'Hallo'`

substring(int beginIndex, int endIndex)

- The `substring()` method is used to retrieve a part of a string, that is, substring from the given string.
- One can specify the start index and the end index for the substring.
- If end index is not specified, all characters from the start index to the end of the string will be returned. For example,
  - `System.out.println(str.substring(2,5)); // output: 'llo'`

Working with String Class 4-7

toString()

- The `toString()` method is used to return a `String` object.
- It is used to convert values of other data types into strings. For example,
  - `Integer length = 5;`
  - `System.out.println(length.toString()); // output: 5`
- Notice that the output is still 5. However, now it is represented as a string instead of an integer.

trim()

- The `trim()` method returns a new string by trimming the leading and trailing whitespace from the current string. For example,
  - `String str1 = " Hello ";`
  - `System.out.println(str1.trim()); // output: 'Hello'`
- The `trim()` method will return '**`Hello`**' after removing the spaces.

Working with String Class 5-7

- Following code snippet demonstrates the use of `String` class methods:

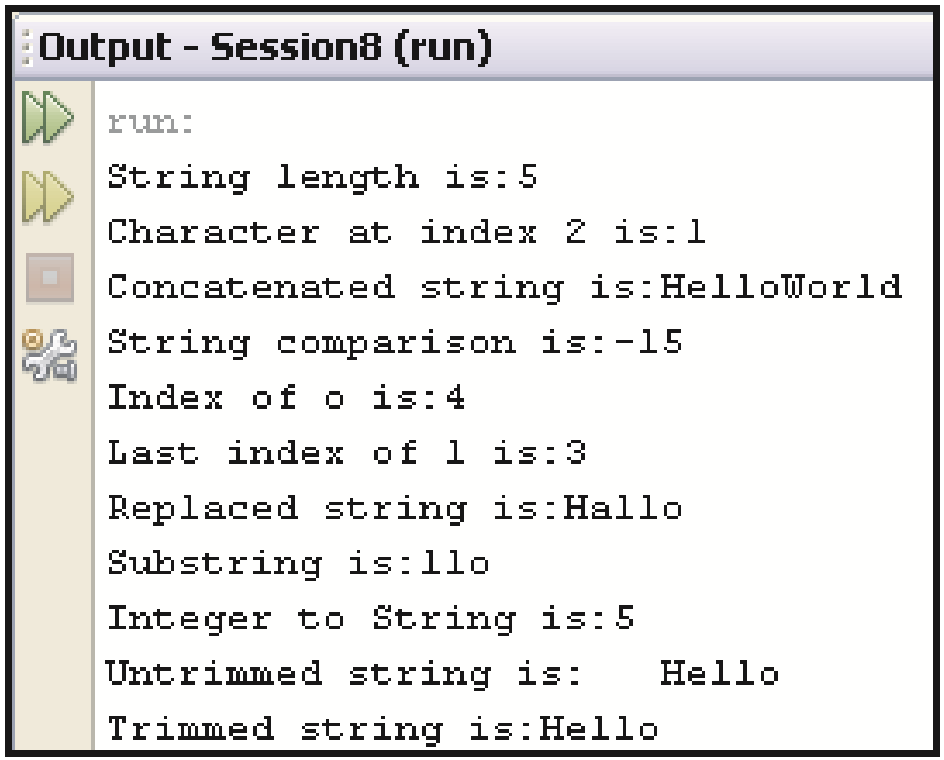
```
public class Strings {

  String str = "Hello"; // Initialize a String variable
  Integer strLength = 5; // Use the Integer wrapper class

  /**
  * Displays strings using various String class methods
  *
  * @return void
  */
  public void displayStrings(){
    // using various String class methods
    System.out.println("String length is:"+ str.length());
    System.out.println("Character at index 2 is:"+ str.charAt(2));
    System.out.println("Concatenated string is:"+ str.concat("World"));
    System.out.println("String comparison is:"+ str.compareTo("World"));
    System.out.println("Index of o is:"+ str.indexOf("o"));
    System.out.println("Last index of l is:"+ str.lastIndexOf("l"));
    System.out.println("Replaced string is:"+ str.replace('e','a'));
    System.out.println("Substring is:"+ str.substring(2, 5));
```

Working with String Class 6-7

```
    System.out.println("Integer to String is:"+ strLength.toString()) ;
    String str1=" Hello ";
    System.out.println("Untrimmed string is:"+ str1);
    System.out.println("Trimmed string is:"+ str1.trim());
  }

  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {

    //Instantiate class, Strings
    Strings objString = new Strings(); // line 1

    //Invoke the displayStrings() method
    objString.displayStrings();
  }
}
```

Working with String Class 7-7

- Following figure shows the output of the `Strings.java` class:



Working with StringBuilder Class 1-2

- `StringBuilder` objects are similar to `String` objects, except that they are mutable and flexible.
- Internally, the system treats these objects as a variable-length array containing a sequence of characters.
- The length and content of the sequence of characters can be changed through methods available in the `StringBuilder` class.
- For concatenating a large number of strings, using a `StringBuilder` object is more efficient.
- The `StringBuilder` class also provides a `length()` method that returns the length of the character sequence in the class.
- Unlike strings a `StringBuilder` object also has a property `capacity` that specifies the number of character spaces that have been allocated.
- The capacity is returned by the `capacity()` method and is always greater than or equal to the length.
- The capacity will automatically expand to accommodate the new strings when added to the string builder.
- `StringBuilder` object allows insertion of characters and strings as well as appending characters and strings at the end.

Working with StringBuilder Class 2-2

- The constructors of the `StringBuilder` class are as follows:

`StringBuilder()`	• Default constructor that provides space for 16 characters.
`StringBuilder(int capacity)`	• Constructs an object without any characters in it. • However, it reserves space for the number of characters specified in the argument, capacity.
`StringBuilder (String str)`	• Constructs an object that is initialized with the contents of the specified string, str.

Methods of StringBuilder Class 1-5

- The `StringBuilder` class provides several methods for appending, inserting, deleting, and reversing strings as follows:

`append()`

- The `append()` method is used to append values at the end of the `StringBuilder` object.
- This method accepts different types of arguments, including `char`, `int`, `float`, `double`, `boolean`, and so on, but the most common argument is of type `String`.
- For each `append()` method, `String.valueOf()` method is invoked to convert the parameter into a corresponding string representation value and then the new string is appended to the `StringBuilder` object.
- For example,
  - `StringBuilder str = new StringBuilder("JAVA ");`
  - `System.out.println(str.append("SE ");`
    `// output: JAVA SE`
  - `System.out.println(str.append(7); // output: JAVA SE 7`

Methods of StringBuilder Class 2-5

insert()

- The `insert()` method is used to insert one string into another.
- It calls the `String.valueOf()` method to obtain the string representation of the value.
- The new string is inserted into the invoking `StringBuilder` object.
- The `insert()` method has several versions as follows:
  - `StringBuilder insert(int insertPosition, String str)`
  - `StringBuilder insert(int insertPosition, char ch)`
  - `StringBuilder insert(int insertPosition, float f)`
- For example,

```
StringBuilder str = new StringBuilder("JAVA 7 ");
System.out.println(str.insert(5, "SE");
  // output: JAVA SE 7
```

Methods of StringBuilder Class 3-5

delete()

- The `delete()` method deletes the specified number of characters from the invoking `StringBuilder` object.
- For example,
  - `StringBuilder str = new StringBuilder("JAVA SE 7");`
  - `System.out.println(str.delete(4,7); // output: JAVA 7`

reverse()

- The `reverse()` method is used to reverse the characters within a `StringBuilder` object.
- For example,
  - `StringBuilder str = new StringBuilder("JAVA SE 7");`
  - `System.out.println(str.reverse());`
    `// output: 7 ES AVAJ`

Methods of StringBuilder Class 4-5

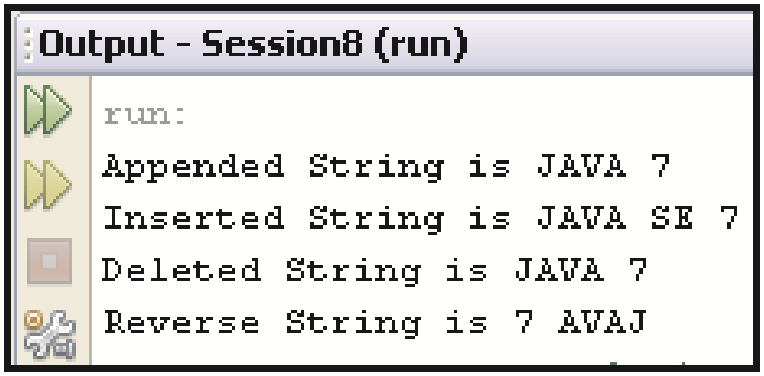
- Following code snippet demonstrates the use of methods of the `StringBuilder` class:

```
package session8;
public class StringBuilders {

  // Instantiate a StringBuilder object
  StringBuilder str = new StringBuilder("JAVA ");

  /**
   * Displays strings using various StringBuilder methods
   *
   * @return void
   */
  public void displayStrings(){

    // Use the various methods of the StringBuilder class
    System.out.println("Appended String is "+ str.append("7"));
    System.out.println("Inserted String is "+ str.insert(5, "SE "));
    System.out.println("Deleted String is "+ str.delete(4,7));
    System.out.println("Reverse String is "+ str.reverse());
  }
```

Methods of StringBuilder Class 5-5

```
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        //Instantiate the StringBuilders class
        StringBuilders objStrBuild = new StringBuilders(); // line 1

        //Invoke the displayStrings() method
        objStrBuild.displayStrings();
    }
}
```

- Following figure shows the output of the `StringBuilders.java` class:



String Arrays 1-3

Sometimes there is a need to store a collection of strings.

`String` arrays can be created in Java in the same manner as arrays of primitive data types.

For example, `String[] empNames = new String[10];`

This statement will allocate memory to store references of 10 strings.

However, no memory is allocated to store the characters that make up the individual strings.

Loops can be used to initialize as well as display the values of a `String` array.

- Following code snippet demonstrates the creation of a `String` array:

```
package session8;
public class StringArray {

  // Instantiate a String array
  String[] empID = new String[5];
```

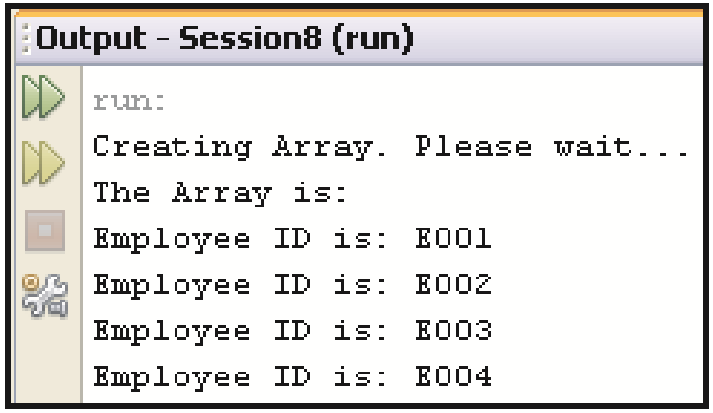
String Arrays 2-3

```
  /**
   * Creates a String array
   * @return void
   */
  public void createArray() {
    System.out.println("Creating Array. Please wait...");
    // Use a for loop to initialize the array
    for(int count = 1; count < empID.length; count++){
    empID[count]= "E00"+count; // storing values in the array
    }
  }
  /**
  * Displays the elements of a String array
  * @return void
  */
  public void printArray() {
    System.out.println("The Array is:");
    // Use a for loop to print the array
    for(int count = 1; count < empID.length; count++){
    System.out.println("Employee ID is: "+ empID[count]);
  }
}
```

String Arrays 3-3

```
  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    //Instantiate class Strings
    StringArray objStrArray = new StringArray(); // line 1
    //Invoke createArray() method
    objStrArray.createArray();
    //Invoke printArray() method
    objStrArray.printArray();
  }
}
```

- Following figure shows the output of the **`StringArray.java`** class:



Command Line Arguments 1-7

A user can pass any number of arguments to a Java application at runtime from the OS command line.

The `main()` method declares a parameter named `args[]` which is a `String` array that accepts arguments from the command line.

These arguments are placed on the command line and follow the class name when it is executed.

- For example,

```
java EmployeeDetail Roger Smith Manager
```

- Here, **`EmployeeDetail`** is the name of a class.
- **`Roger`**, **`Smith`**, and **`Manager`** are command line arguments which are stored in the array in the order that they are specified.

When the application is launched, the runtime system passes the command line arguments to the application's `main()` method using a `String` array, `args[]`.

The array of strings can be given any other name.

The `args[]` array accepts the arguments and stores them at appropriate locations in the array.

Command Line Arguments 2-7

- The length of the array is determined from the number of arguments passed at runtime.
- The arguments are separated by a space.
- The basic purpose of command line arguments is to specify the configuration information for the application.
- The `main()` method is the entry point of a Java program, where objects are created and methods are invoked.
- The `static main()` method accepts a `String` array as an argument as depicted in the following code snippet:

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

- The parameter of the `main()` method is a `String` array that represents the command line arguments.
- The size of the array is set to the number of arguments specified at runtime.
- All command line arguments are passed as strings.

Command Line Arguments 3-7

- Following code snippet demonstrates an example of command line arguments:

```
package session8;
public class CommandLine {
  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {

    // Check the number of command line arguments
    if(args.length==3) {

      // Display the values of individual arguments
      System.out.println("First Name is "+args[0]);
      System.out.println("Last Name is "+args[1]);
      System.out.println("Designation is "+args[2]);
    }
    else {
      System.out.println("Specify the First Name, Last Name, and
      Designation");
    }
  }
}
```

Command Line Arguments 4-7

- To run the program with command line arguments at command prompt, do the following:

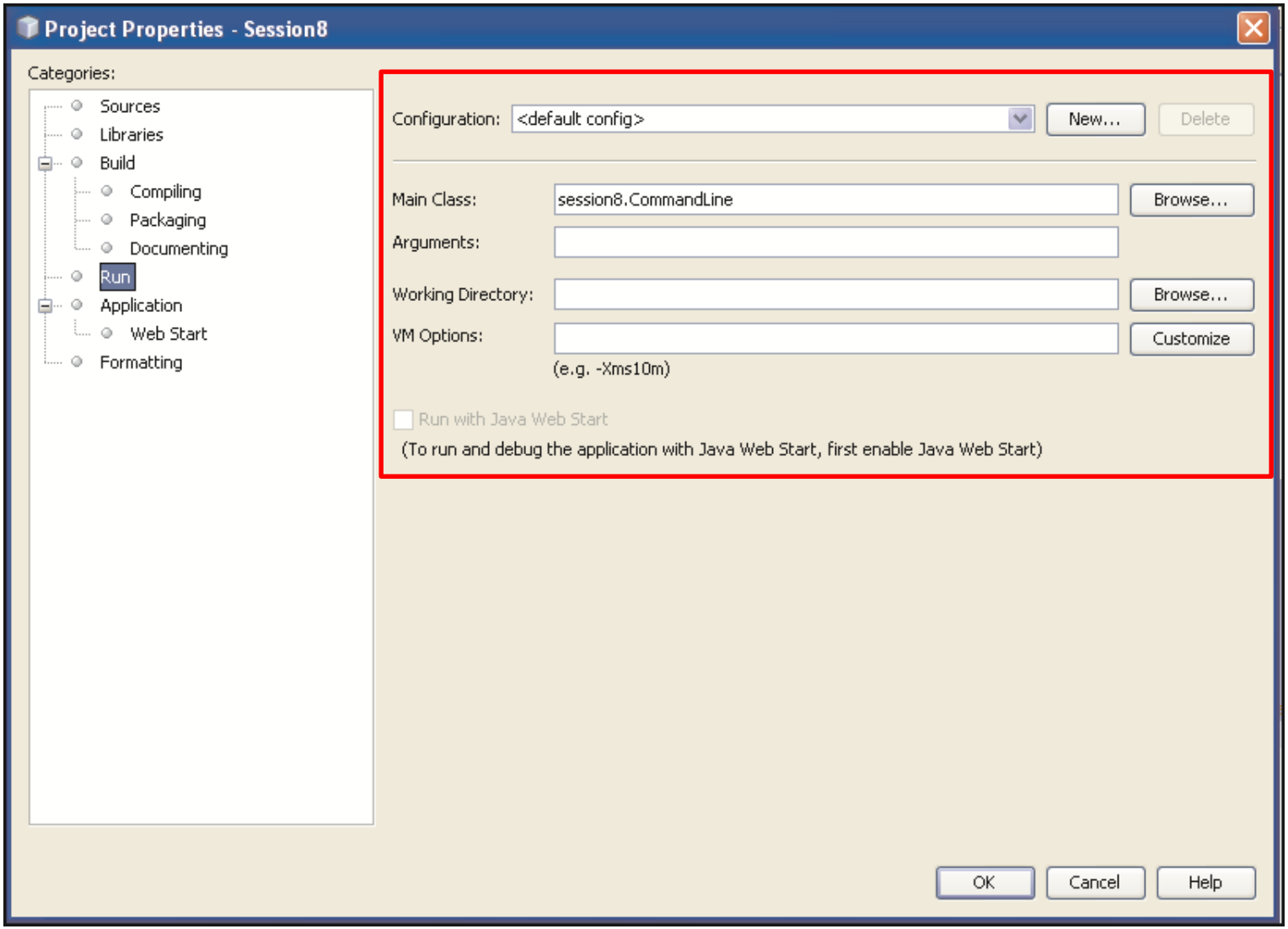
1. Open the command prompt.
2. Compile the Java program by writing the following statement:
   - `javac CommandLine.java`
3. Execute the program by writing the following statement:
   - `java CommandLine Roger Smith Manager`

- To run the program with command line arguments using NetBeans IDE, perform the following steps:

1. Right-click the project name in the **Projects** tab and click **Properties**.
   - The **Project Properties** dialog box is displayed.
2. Click **Run** in the **Categories** pane to the left.
   - The runtime properties are displayed in the right pane.

Command Line Arguments 5-7

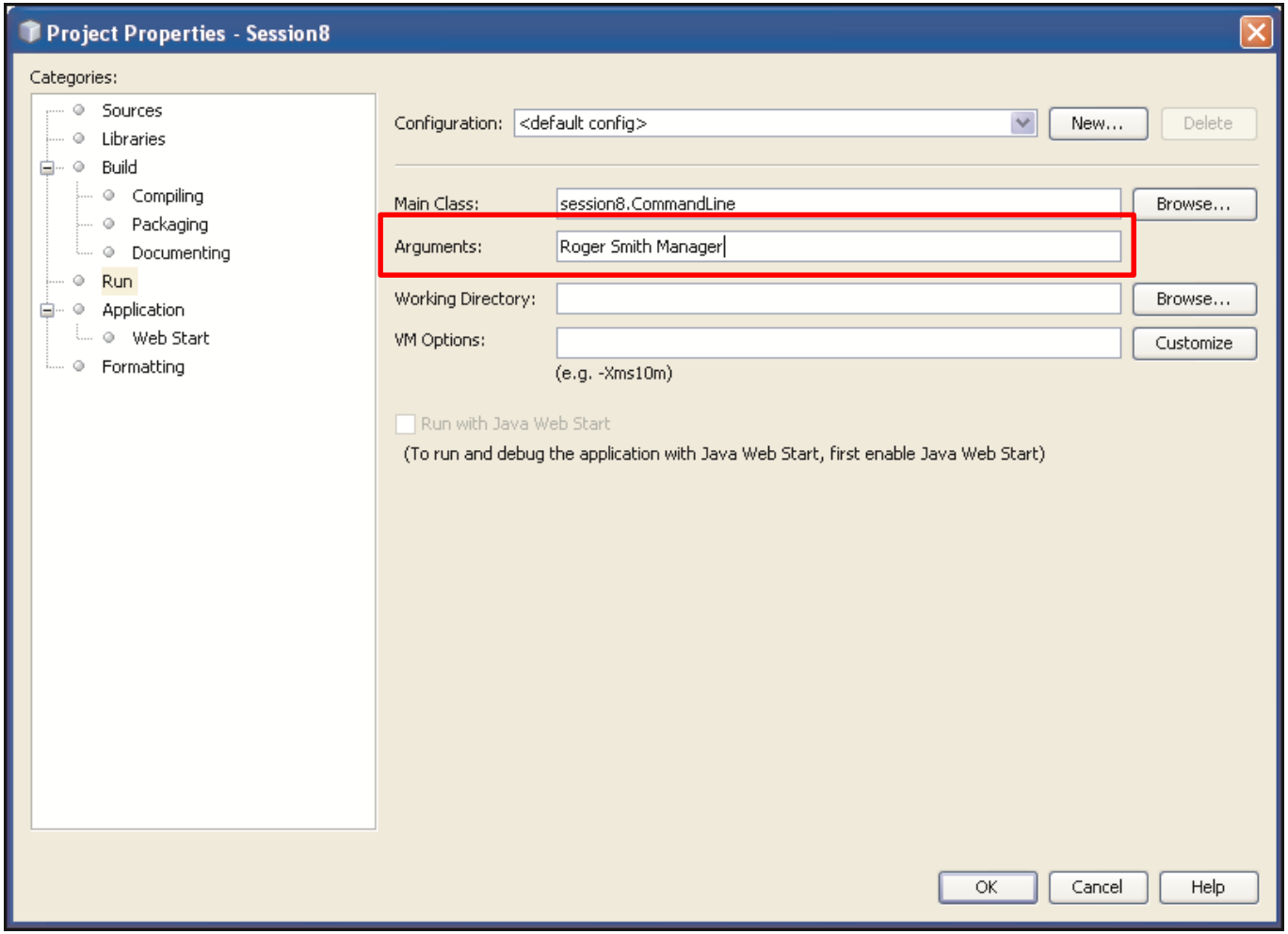
- The runtime properties are shown in the following figure:



Command Line Arguments 6-7

3

- Type the arguments `Roger`, `Smith`, and `Manager` in the **Arguments** box as shown in the following figure:



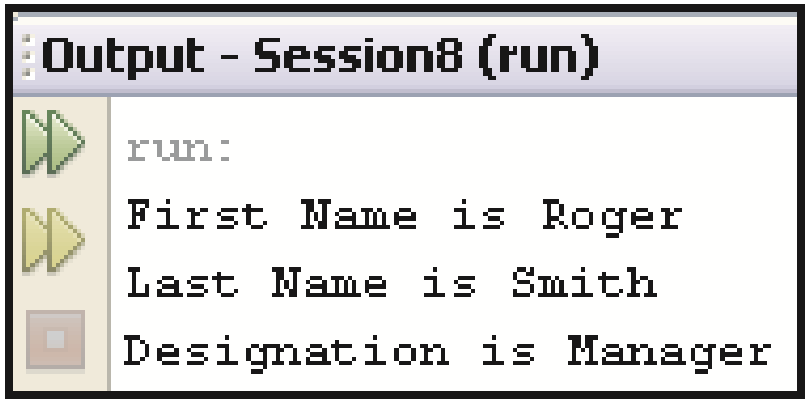
Command Line Arguments 7-7

4

- Click **OK** to close the **Project Properties** dialog box.

5

- Click **Run** on the toolbar or press **F6**.
- The command line arguments are supplied to the `main()` method and printed as shown in the following figure:



Wrapper Classes 1-11

Java provides a set of classes known as wrapper classes for each of its primitive data type that 'wraps' the primitive type into an object of that class.

In other words, the wrapper classes allow accessing primitive data types as objects.

The wrapper classes for the primitive data types are: `Byte`, `Character`, `Integer`, `Long`, `Short`, `Float`, `Double`, and `Boolean`.

The wrapper classes are part of the `java.lang` package.

- The primitive types and the corresponding wrapper types are listed in the following table:

Primitive type	Wrapper class
byte	Byte
char	Character
float	Float
double	Double
int	Integer
long	Long
short	Short
boolean	Boolean

Wrapper Classes 2-11

What is the need for wrapper classes?

- The use of primitive types as objects can simplify tasks at times.
- For example, most of the collections store objects and not primitive data types.
- Many of the activities reserved for objects will not be available to primitive data types.
- Also, many utility methods are provided by the wrapper classes that help to manipulate data.
- Wrapper classes convert primitive data types to objects, so that they can be stored in any type of collection and also passed as parameters to methods.
- Wrapper classes can convert numeric strings to numeric values.

Wrapper Classes 3-11

valueOf()

- The `valueOf()` method is available with all the wrapper classes to convert a type into another type.
- The `valueOf()` method of the `Character` class accepts only `char` as an argument.
- The `valueOf()` method of any other wrapper class accepts either the corresponding primitive type or `String` as an argument.

typeValue()

- The `typeValue()` method can also be used to return the value of an object as its primitive type.

- Some of the wrapper classes and their methods are listed in the following table:

Wrapper Class	Methods	Example

`Byte`	`byteValue()` – returns a `byte` value of the invoking object.
	
	`parseByte()` – returns the `byte` value from a string storing a `byte` value.
		`byte byteVal = Byte.byteValue();`
		
		`byte byteVal = Byte.parseByte("45");`

Wrapper Classes 4-11

Wrapper Class	Methods	Example

`Character`	`isDigit()` – checks if a character is a digit.
	
	`isLowerCase()` – checks if a character is a lower case alphabet.
	
	`isLetter()` – checks if a character is an alphabet.
		`if(Character.isDigit('4')`
		`System.out.println("Digit");`
		
		`if(Character.isLetter('L')`
		`System.out.println("Letter");`

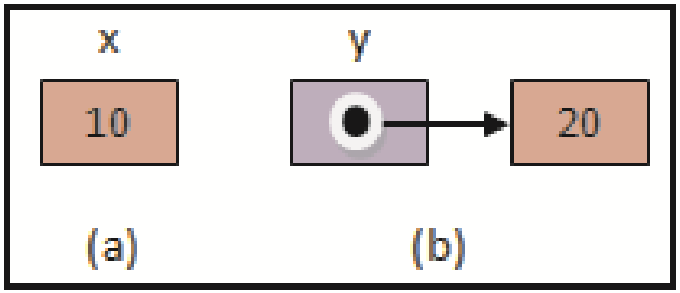
`Integer`	`intValue()` – returns the `Integer` value as a primitive type `int`.
	
	`parseInt()` - returns the `int` value from a string storing an integer value.
		`int intVal = Integer.intValue();`
		
		`int intVal=Integer.parseInt("45");`

Wrapper Classes 5-11

- The difference between creation of a primitive type and a wrapper type is as follows:

Primitive type	• `int x = 10;`
Wrapper type	• `Integer y = new Integer(20);`

- The first statement declares and initializes the `int` variable **x** to **10**.
- The second statement instantiates an `Integer` object **y** and initializes it with the value **20**.
- In this case, the reference of the object is assigned to the object variable **y**.
- The memory assignment for the two statements is shown in the following figure:



- It is clear from the figure that **x** is a variable that holds a value whereas **y** is an object variable that holds a reference to an object.

Wrapper Classes 6-11

The six methods of type `parseXxx()` available for each numeric wrapper type are in close relation to the `valueOf()` methods of all the numeric wrapper classes including `Boolean`.

The two type of methods, that is, `parseXxx()` and `valueOf()`, take a `String` as an argument.

If the `String` argument is not properly formed, both the methods throw a `NumberFormatException`.

These methods can convert `String` objects of different bases if the underlying primitive type is any of the four integer types.

The `parseXxx()` method returns a named primitive whereas the `valueOf()` method returns a new wrapped object of the type that invoked the method.

Wrapper Classes 7-11

- Following code snippet demonstrates the use of `Integer` wrapper class to convert the numbers passed by user as strings at command line into integer types to perform the calculation based on the selected operation:

```
package session8;
public class Wrappers {
  /**
  * Performs calculation based on user input
  *
  * @return void
  */
  public void calcResult(int num1, int num2, String choice){
    // Switch case to evaluate the choice
    switch(choice) {
     case "+": System.out.println("Result after addition is: "+
        (num1+num2));
        break;
     case "-": System.out.println("Result after subtraction is: "+
        (num1-num2));
        break;
     case "*": System.out.println("Result after multiplication is: "+
        (num1*num2));
        break;
```

Wrapper Classes 8-11

```
    case "/": System.out.println("Result after division is: " +
      (num1/num2));
      break;
   }
}

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {

  // Check the number of command line arguments
  if(args.length==3){

    // Use the Integer wrapper to convert String argument to int type
    int num1 = Integer.parseInt(args[0]);
    int num2 = Integer.parseInt(args[1]);

    // Instantiate the Wrappers class
    Wrappers objWrap = new Wrappers();
```

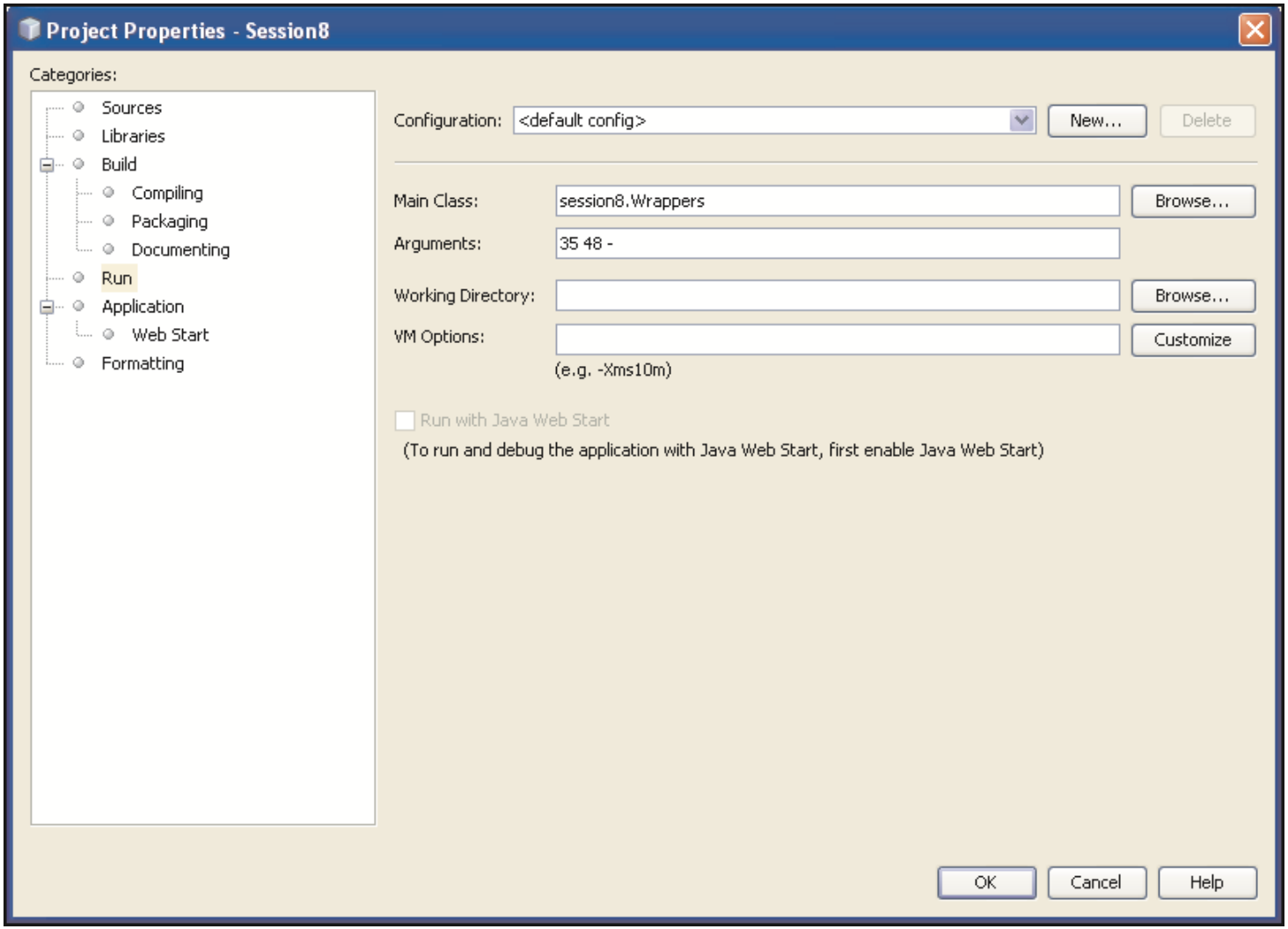
Wrapper Classes 9-11

```
        // Invoke the calcResult() method
        objWrap.calcResult(num1, num2, args[2]);
      }
      else{
        System.out.println("Usage: num1 num2 operator");
      }
    }
}
```

- The class **`Wrappers`** consists of the **`calcResult()`** method that accepts two numbers and an operator as the parameter.
- The `main()` method is used to convert the `String` arguments to `int` type by using the `Integer` wrapper class.
- Next, the object, **`objWrap`** of **`Wrappers`** class is created to invoke the **`calcResult()`** method with three arguments namely, **`num1`**, **`num2`**, and **`args[2]`** which is the operator specified by the user as the third argument.

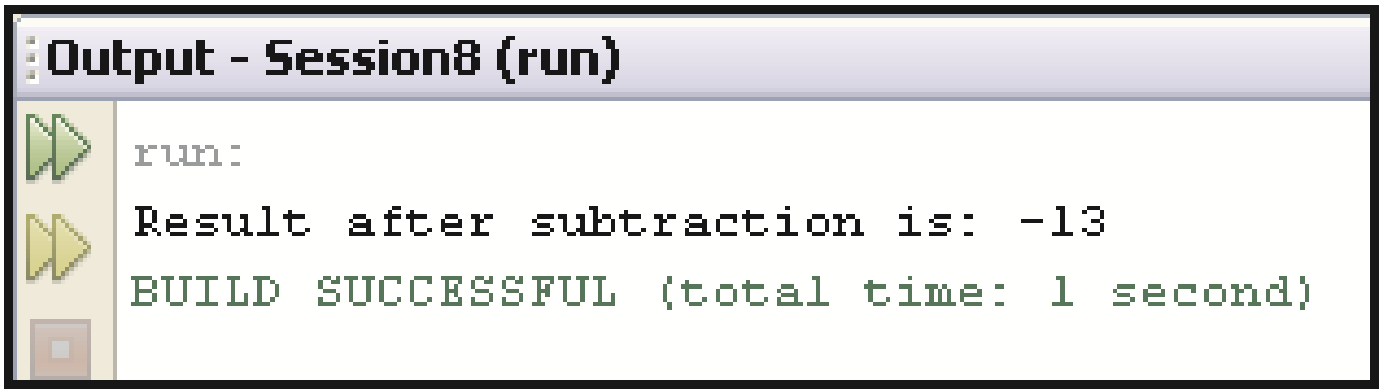
Wrapper Classes 10-11

- To run the class, specify the command line values as **35**, **48**, and **–** in the **Arguments** box as shown in the following figure:



Wrapper Classes 11-11

- When the program is executed, the first two arguments are stored in the `args[0]` and `args[1]` elements and then, converted to integers.
- The third argument is stored in the `args[2]` element.
- It is the operator to be applied on the numbers.
- The output of the code is shown in the following figure:



Autoboxing and Unboxing 1-3

Autoboxing

- The automatic conversion of primitive data types such as `int`, `float`, and so on to their corresponding object types such as `Integer`, `Float`, and so on during assignments and invocation of methods and constructors is known as autoboxing.
- For example,

```
ArrayList<Integer> intList = new ArrayList<Integer>();
intList.add(10); // autoboxing
Integer y = 20; // autoboxing
```

Unboxing

- The automatic conversion of object types to primitive data types is known as unboxing.
- For example,

```
int z = y; // unboxing
```

Autoboxing and unboxing helps a developer to write a cleaner code.

Using autoboxing and unboxing, one can make use of the methods of wrapper classes as and when required.

Autoboxing and Unboxing 2-3

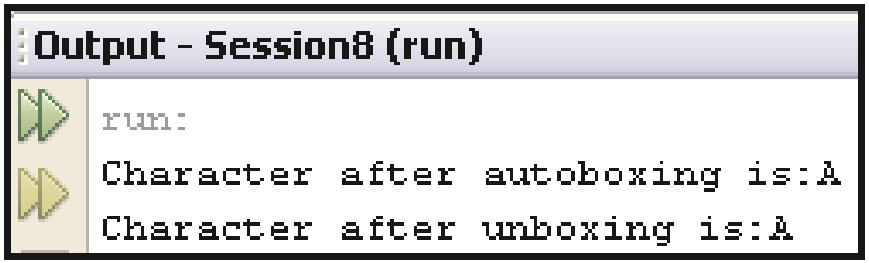
- Following code snippet demonstrates an example of autoboxing and unboxing:

```
package session8;
public class AutoUnbox {

  /**
   * @param args the command line arguments
   */
  public static void main(String args[]) {
    Character chBox = 'A'; // Autoboxing a character
    char chUnbox = chBox; // Unboxing a character

    // Print the values
    System.out.println("Character after autoboxing is:"+ chBox) ;
    System.out.println("Character after unboxing is:" + chUnbox);
  }
}
```

- The class **AutoUnbox** consists of two variable declarations **chBox** and **chUnbox**.
- **chBox** is an object type and **chUnbox** is a primitive type of character variable.

Autoboxing and Unboxing 3-3

- Following figure shows the output of the code:



- The figure shows that both primitive as well as object type variable give the same output.
- However, the variable of type object, that is **chBox**, can take advantage of the methods available with the wrapper class `Character` which are not available with the primitive type `char`.

Summary

- An array is a special data store that can hold a fixed number of values of a single type in contiguous memory locations.
- A single-dimensional array has only one dimension and is visually represented as having a single column with several rows of data.
- A multi-dimensional array in Java is an array whose elements are also arrays.
- A collection is an object that groups multiple elements into a single unit.
- Strings are constant and immutable, that is, their values cannot be changed once they are created.
- StringBuilder objects are similar to String objects, except that they are mutable.
- Java provides a set of classes known as Wrapper classes for each of its primitive data type that ‘wrap’ the primitive type into an object of that class.
- The automatic conversion of primitive types to object types is known as autoboxing and conversion of object types to primitive types is known as unboxing.