CMPS 251

Lecture 06

Arrays and ArrayLists

CSE@QU

Summary: Lecture 05 (OOP)

- Defining Class
- Instantiation of objects
- Variables
  - Instance variables
  - Local variables
  - Static/class variables
  - this. Variable (object reference)
- Methods
  - Constructor
  - Static/class method
  - Public method
  - Setter and getter methods
  - Method overloading
- Access Modifiers
  - Public
  - Private
  - Protected
  - Default

You must have a VERY good understanding of these important aspect of OOP

Outline

- Arrays
- Arrays of Objects
- Array Class
- ArrayList Class
- Exception Handling

Basic Concepts of Arrays

- Arrays are used to store multiple values in **a single variable**, instead of declaring separate variables for each value.
- A group of variables/elements of the same type
- Arrays are objects
  - Created with the **new** keyword
  - Memory allocation of an array is contiguous (elements next to each others, not randomly placed in memory)
- The array size is fixed/constant
  - Cannot be resized
  - The number of elements in the array can be retrieved using the instance variable **length**
- An array can be of any primitive or object type

Declaring and creating arrays

- Array are objects created with keyword new.

```
int[] c = new int[ 12 ];
```

- Or,

```
int[] c; // declare the array variable
c = new int[ 12 ]; // creates the array
```

- Multiple arrays declaration in one statement,

```
int[] a, b, c;
```

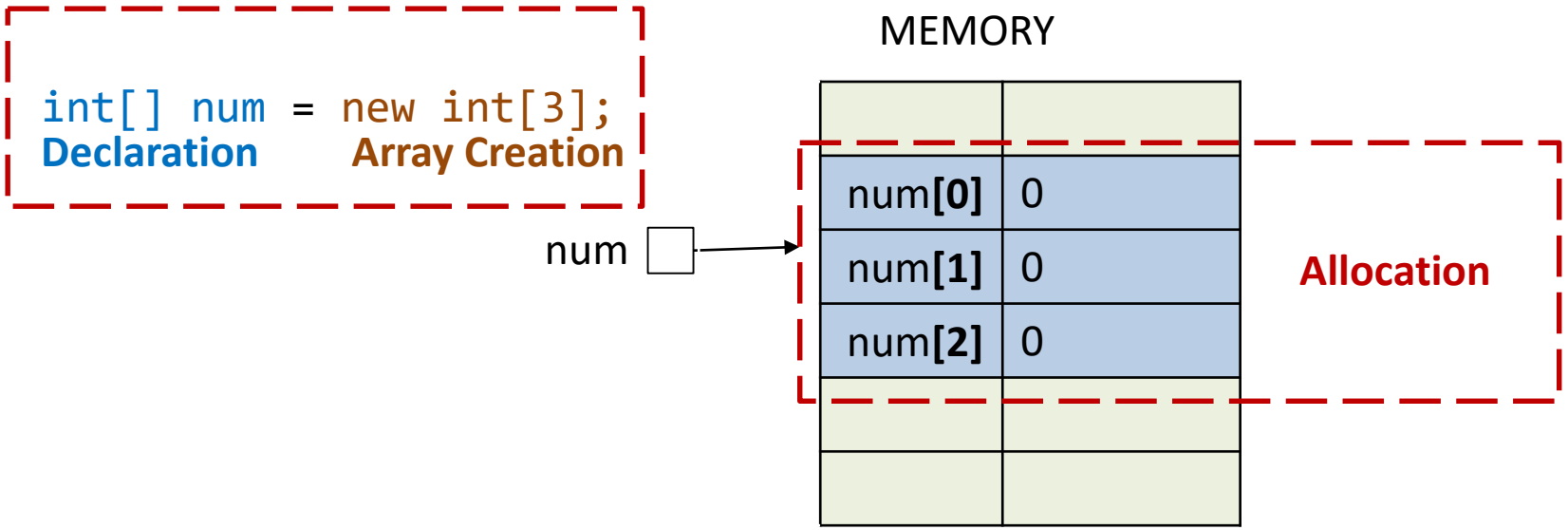
- Declaring arrays and variables in the same statement,

```
int a[], b, c = 3;
```

  - Array length is determined by the number of elements in the initializer list.

```
int[] n = { 10, 20, 30, 40, 50 };
```

An array object stores multiple values of the same type



- Array = **fixed-length** data structure storing values of the same type
- Array elements are **auto initialized** with the type's default value:
  - 0 for the numeric primitive-type elements, false for boolean elements and null for references

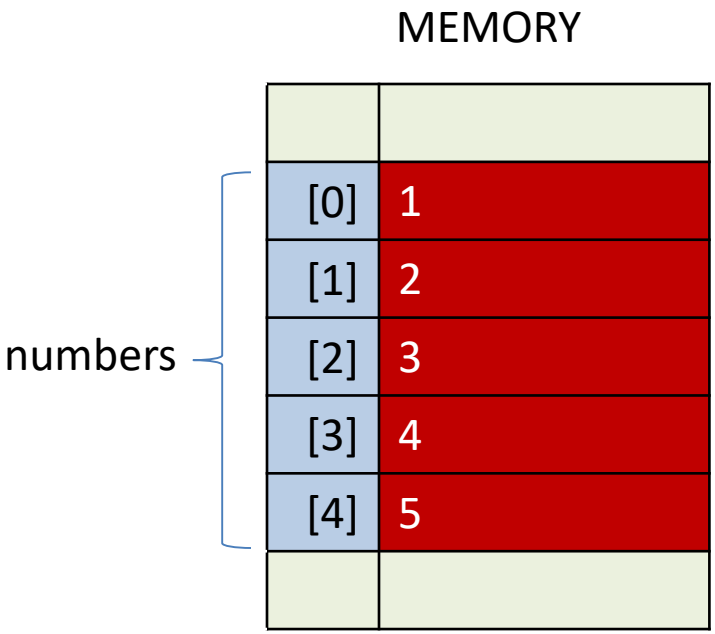
Array stores values of the same type

```
int[] number = new int[100];          // stores 100 integers

double[] salesTax = new double[10];  // stores 10 doubles

char[] alphabet = new char[26];       // stores 26 characters
```

- The array **size** determines the number of elements in the array.
- The **size** must be specified in the array declaration and it cannot change once the array is created

You may initialize an array explicitly

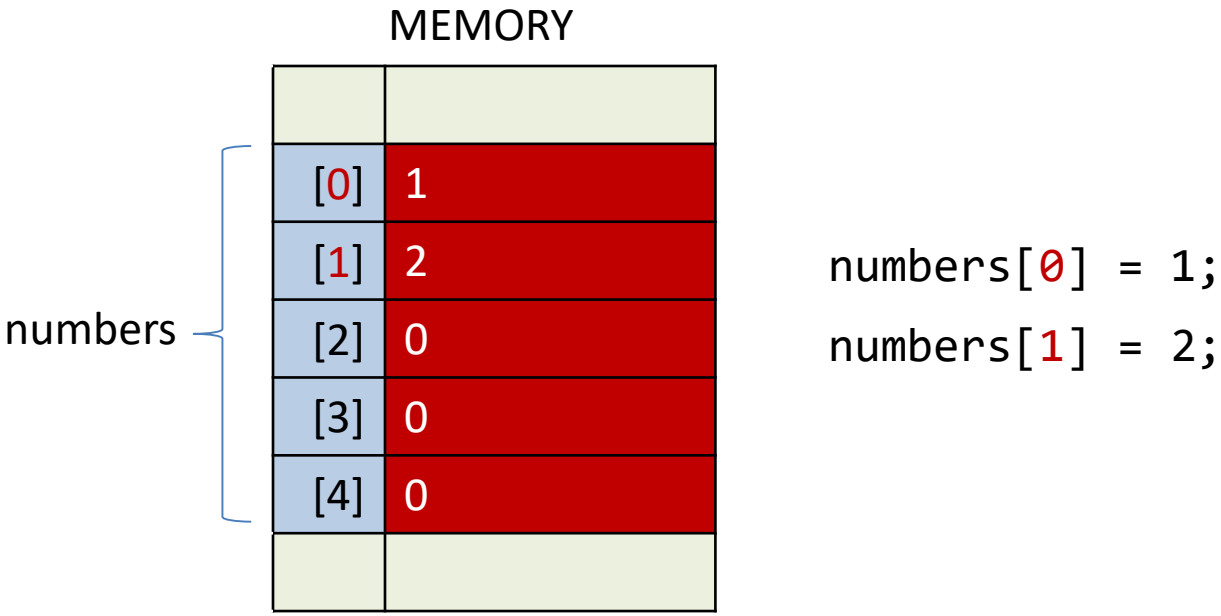
```
int[] numbers = {1, 2, 3, 4, 5};  // Array initializer
```



Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Array elements are indexed

```
int[] numbers = new int[5];
```



- Array index range is 0 to array size -1

String[] b = new String[100], x = new String[20], z = new String[30];
// declaring several arrays in a single statement

String[] b = new String[100] == String b[] = new String[100]; //same

Arrays can be attributes

```
public class Student {
    private int[] grades;
    ...
}
```

Arrays can be local variables

```
public void getSalaryEmployees() {
    double[] salary;
    ...
}
```

Arrays can be parameters

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

Arrays can be return values

```
public String[] getNames() {
    ...
}
```

Example - Method that returns an array

```
public int[] initArray(int size, int initValue) {
   int[] array = new int[size];

//array.length finds out the total number of elements
   for (int i = 0; i < array.length; i++) {
      array[i] = initValue;
   }

   return array;
}
```

```
//example
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars[0]);
// displays Volvo
 System.out.println(cars.length); // displays 4
```

Arrays are objects, thus

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

b = a;                  // makes b and a refer to the same
                        // memory location
```

- Arrays are objects so they are **reference types**.

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda", "Toyota"};
System.out.println(cars.length);
// Outputs 5
```

Arrays are objects, thus

```
int[] a = {1, 2, 3};
int[] b = {1, 2, 3};
if (a == b) {...}      // evaluates to false
                       // since a and b refer to two
                       // different memory locations
```

- You can loop through the array elements with the for loop, and use the length property to specify how many times the loop should run.
- The following example outputs all elements in the **cars** array:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (int i = 0; i < cars.length; i++) {
  System.out.println(cars[i]);
}
```

Example - Method that tests for array equality

```
public boolean areEqual(int[] array1, int[] array2) {
   if (array1.length != array2.length) {
      return false;
   } else {
      for(int i = 0; i < array1.length; i++) {
         if(array1[i] != array2[i])
             return false;
      }// end for
   }// end if
   return true;
}
```

Enhanced for loop

- The enhanced for loop (also called a "for each" loop) allows you to iterate through the elements of an array or a list without using a counter.
- The syntax of an enhanced for statement is:

```
for (var item : arrayName) {
    statement;
}
```

```
//outputs all elements in the cars array, using a "for-each" loop
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
    for (String i : cars) {  //for-each loop
      System.out.println(i);
      }
//read like this: for each String element (called i - as in index) in cars,
print out the value of i.
```

Note: **for-each** method for loop is easier to write, it does not require a counter (using the length property), and it is more readable.

Array Search

```
// Returns true if array contains item, false otherwise.
private boolean contains(String[] items, String element) {
        // Using enhanced for loop to iterate through the
array
       for(var item : items) {
            if (item.equalsIgnoreCase(element)) {
               return true;
            }
        }   // end for
        return false;
}
c[a + b] += 2; //is a correct expression.
```

Array of Objects Example

```
Book[] books = new Book[2];

Book b = new Book("Harry Potter");
books[0] = b;

Book c = new Book("Hunger Games");
books[1] = c;

for (int i = 0; i < books.length; i++) {
      System.out.println(books[i].getTitle());
}
// A simpler for loop  (called for each)
for (Book temp : books) {
      System.out.println(temp.getTitle());
}
```

The Arrays class and its API

- **`Arrays`** class
  - Must import java.util.Arrays;
  - Provides `static` methods for common array manipulations.
  - Methods include
    - `sort` for sorting an array (ascending order by default)
    - `binarySearch` for searching a sorted array
    - `equals` for comparing arrays
    - `fill` for placing values into an array.
  - Methods are overloaded for primitive-type arrays and for arrays of objects.
- **`System`** class `static` `arraycopy` method.
  - Copies contents of one array into another.

Built-in Array Methods

```
Arrays.sort(a);
```
- *sorts the array*

```
Arrays.sort(b, 4, 10);
```
- *sorts the range of elements indexed 4 to 10 of the array.*

```
Arrays.fill(c, 5);
```
- *fills all elements with the value 5*

```
Arrays.fill(c, 7, 11, 33);
```
- *fills the range of elements indexed 7 to 11 with the value 33*

```
int[] d = Arrays.copyOf(a, 10);
```
- *produces array containing the first 10 elements of a.*

```
int[] e = Arrays.copyOf(a, 20);
```
- *produces array containing the first 20 elements of a. if array has less zeros are the rest of elements.*

```
int[] f = Arrays.copyOfRange(a, 5, 10);
```
- *produces array containing the range of elements indexed 5 to 10 of a.*

```
if(Arrays.equals(a, b))
```
- *Checks for elements equality of the arrays a and b. Returns true or false*

```
System.arraycopy(a, 2, b, 5, 4);
```
- *Copies 4 elements of a starting from index 2 placing them in b starting at index 5*

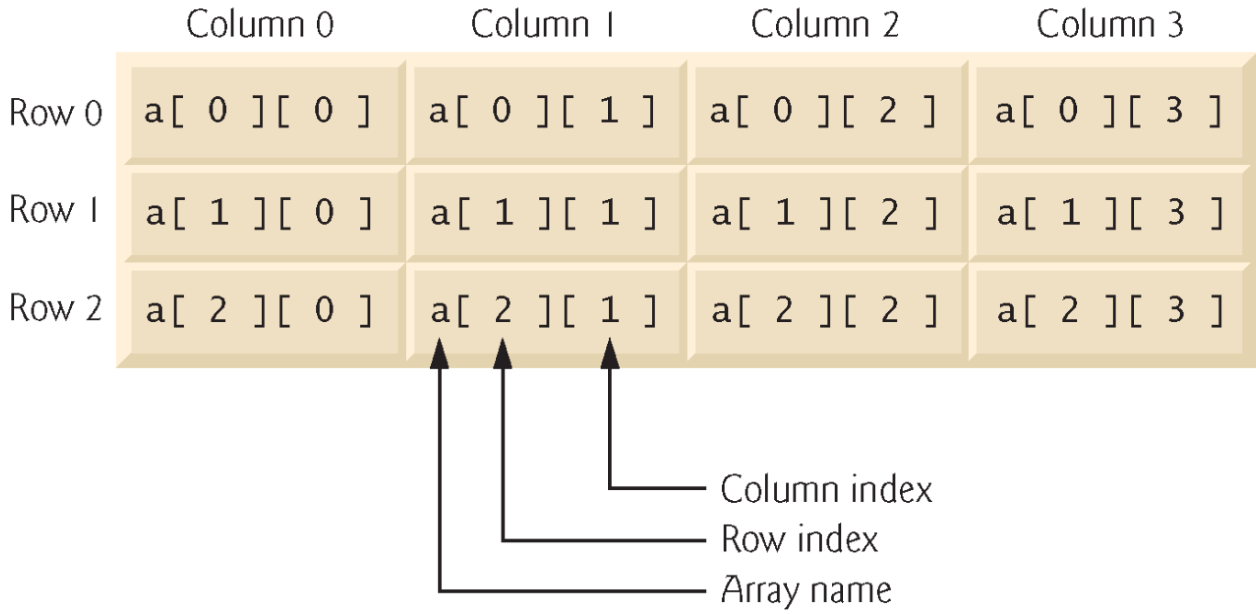
```
System.arraycopy(a, 0, b, 0, a.length);
```
- *Copies all elements of a placing them in b starting at index 0*

Searching Array

```
//search for x in a
int searchIndex, x=26;
searchIndex = Arrays.binarySearch(a, x);
if(searchIndex<0)
   System.out.println(x+" is NOT found\n");
else
   System.out.println(x+" is found at location "+searchIndex+"\n");


//search for x in the range of locations indexed 5 to 10 of the array
a
int searchIndex, x=33;
searchIndex = Arrays.binarySearch(a,5,10,x);
if(searchIndex<0)
   System.out.println(x+" is NOT found\n");
else
   System.out.println(x+" is found at location "+searchIndex+"\n");
```

Multidimensional Arrays

- **Two-dimensional arrays** are often used to represent tables of values with data arranged in *rows* and *columns*.
- Example: two-dimensional arrays with 3 rows and 4 columns
- `int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} }`



- Java is considered "row major", meaning that it does rows first. This is because a 2D array is an "array of arrays".
- There are no multi-dimension arrays. There are arrays of arrays.

Multidimensional Arrays (Cont.)

- A multidimensional array **b** with 3 rows and 4 columns

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

- A two-dimensional array **b** with 2 rows and 3 columns could be declared and initialized with nested array initializers as follows:

```
int[][] b = {{1, 2, 9}, {3, 4, 8}};
```

  - The initial values are *grouped by row* in braces.
  - The number of nested array initializers (represented by sets of braces within the outer braces) determines the number of *rows*.
  - The number of initializer values in the nested array initializer for a row determines the number of *columns* in that row.

ArrayLists

- Problem with arrays
  - You must know the array size when you create the array
  - Array size cannot change once created.

- Solution:
  - Use **ArrayList**: they stretch as you add elements to them or shrink as you remove elements from them
  - Similar to arrays + allow **Dynamic resizing**

ArrayList Class

- **`ArrayList<T>`** in **package java.util** can dynamically change its size to accommodate more elements.
  - **T** is a placeholder for the type of element stored in the collection.
  - This is similar to specifying the type when declaring an array, except that only nonprimitive types can be used with these collection classes.

ArrayList methods

- Create empty list

  `new ArrayList<>()`
- Add entry to end

  `add(value)`
- Retrieve element at index

  `get(index)`
- Check if element exists in list

  `contains(element)`
- Remove element

  `remove(index)` or `remove(element)`
- Get the number of elements

  `size()`
- Remove all elements

  `clear()`

ArrayList Example

```
import java.util.ArrayList; // Don't forget this import

public class ListTest2 {
  public static void main(String[] args) {
    ArrayList<String> entries = new ArrayList<String>();
    double d;
    while((d = Math.random()) > 0.1) {
      entries.add("Value: " + d);
    }
    for(String entry: entries) {
      System.out.println(entry);
    }
  }
}
```

This tells Java that the list will contain only strings.

ArrayList Example

```
public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    cars.get(2); //Ford
    cars.set(2,“Toyota”); // replace “Ford” by “Toyota”
    cars.remove(0); //”Volvo” removed
    cars.size(); //to find out how many elements an ArrayList has
    for (String i : cars) {  //for-each loop
      System.out.println(i);
    }
    cars.clear(); //Remove all elements from ArrayList

  }
}
```

Other Types in ArrayList

- Elements in an ArrayList are actually objects.
- In the examples in the previous slide, we created elements (objects) of type "String".
- Remember that a String in Java is an object (not a primitive type).
- To use other types, such as `int`, you must specify an equivalent wrapper class: `Integer`.
- For other primitive types, use: `Boolean` for boolean, `Character` for char, `Double` for double, etc.

ArrayList Example with Integer

```
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(10);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(25);
    myNumbers.set(1, 100); //replace 15 by 100
    myNumbers.get(1); //100

    for (int i : myNumbers) {
      System.out.println(i);
    }
  }
}
```

Sort an ArrayList of String

- Another useful class in the `java.util` package is the `Collections` class, which include the `sort()` method for sorting lists alphabetically or numerically:

```
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class

public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    Collections.sort(cars);  // Sort cars
    for (String i : cars) {
      System.out.println(i);
    }
  }
}
```

Sort an ArrayList of Integers

```
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(33);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(34);
    myNumbers.add(8);
    myNumbers.add(12);

    Collections.sort(myNumbers);  // Sort myNumbers

    for (int i : myNumbers) {
      System.out.println(i);
    }
  }
}
```

ArrayList Example

```
ArrayList<Book> books = new ArrayList<Book>();

Book b = new Book("Harry Potter");
books.add(b);

Book c = new Book("Hunger Games");
books.add(c);

for(int i = 0; i < books.size(); i++) {
        Book temp = books.get(i);
        System.out.println(temp.getTitle());
}

//alternative solution
for(Book temp: books) {
        System.out.println(temp.getTitle());
}

books.set(0, new Book("The Man and the Sea");
//replaces item at position 0
books.remove(0);
```

Variable-Length Argument Lists

- Variable-length argument lists can be used to create methods that receive an **unspecified** number of arguments.
  - Parameter type followed by an **ellipsis (`...`)** indicates that the method receives a variable number of arguments of that particular type.
- A variable-length argument list **is treated as an array** within the method body. The number of arguments in the array can be obtained using the array's length attribute.

Variable-Length Argument Lists - Example

```
// Variable-Length Argument Lists - Example
public static double average(double... numbers) {
    double total = 0.0;
    for(var num : numbers) {
        total += num;
    }
    return total / numbers.length;
}

public static void main(String[] args) {
    double avg = average(4, 6, 2);
    System.out.println(avg);
}
```

Reference Variables

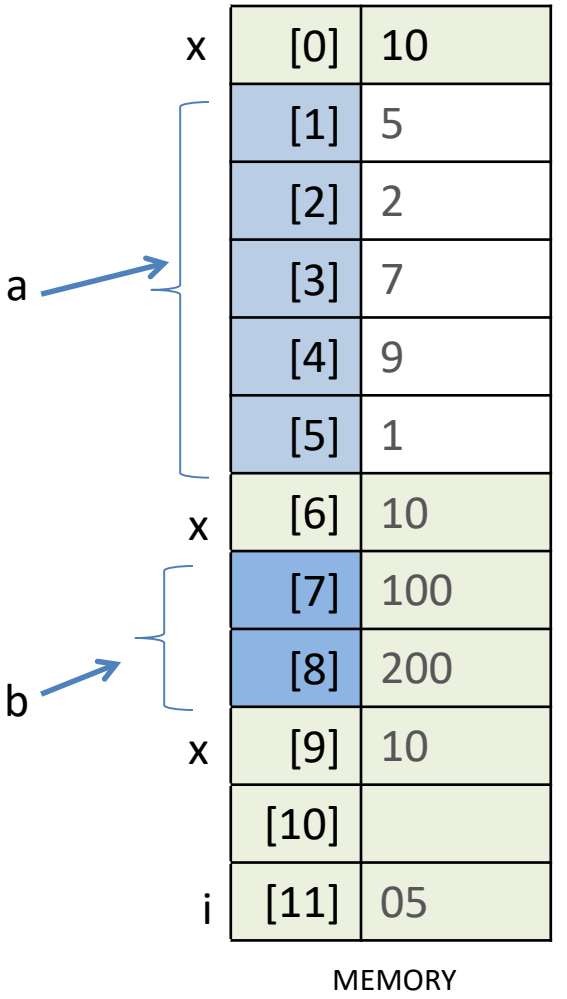
- All of the variables provided by Java are Reference variable type.
- Objects instantiated from any class are Reference variables.
- The variables with eight primitive data type are not Reference variables
- In practice, any object instanced from a class is a reference variable.
- The value of a primitive variable is stored directly in the variable, whereas the value of a reference variable is a reference to the variable's data
- The value of a reference variable — i.e., the reference — points to a location in the memory that contains information relating to the given variable.

Array Variable

- A variable of array type holds a reference to an object.
- Declaring a variable of array type does not create an array object or allocate any space for array components.
- It creates only the variable itself, which can contain a reference to an array.
- but array objects do not really belong to a class of their own. An array object inherits all of the variables and methods of the Object class.

Reference Variables in Picture

```
int[] a = {5,2,7,9,1};
int[] b = {100, 200};
// b = a;
```



myVar

yourVar

```
public class RefVariable {
int x = 10;
public static void main(String[] args) {
        RefVariable myVar   = new RefVariable();
        RefVariable yourVar = new RefVariable();

        myVar.x = 50;
        yourVar = myVar;

        yourVar.x = 300;
        yourVar.x  = myVar.x;

        int i = 05;
```

Example of Reference Variables

```
import java.util.ArrayList;
public class RefVariable {
int x = 10;
public static void main(String[] args) {
        RefVariable myVar = new RefVariable();
        RefVariable yourVar = new RefVariable();

        System.out.println("myVar.x is: "+myVar.x +", yourVar.x is: "+yourVar.x);
        System.out.println();

        myVar.x = 50;
        yourVar = myVar;
        System.out.println("myVar.x is: "+myVar.x +", yourVar.x is: "+yourVar.x);
        System.out.println();

        yourVar.x = 300;
        yourVar.x  = myVar.x;
        System.out.println("myVar.x is: "+myVar.x +", yourVar.x is: "+yourVar.x);
        System.out.println();
        //Array example
         int[] a = {5,2,7,9,1};
         int[] b = {100, 200};
         b = a;
           for(int i= 0; i < a.length; i++) {
              System.out.println("a: "+a[i]);
              System.out.println("b: "+b[i]);
           } } }
```

What is an Exception?

- An exception indicates a problem that occurs while a program executes.
- When the Java Virtual Machine (JVM) or a method detects a problem, such as an *invalid array index* or an *invalid method argument*, it **throws an exception**.
- e.g., trying to access an array element outside the bounds of the array.
  - Java doesn't allow this.
  - JVM checks that array indices to ensure that they are **>= 0** and **< the array's size**. This is called **bounds checking**.
  - If a program uses an invalid index, JVM throws an exception to indicate that an error occurred in the program at execution time.

Handling Exceptions

- Exception handling helps you create **fault-tolerant programs** that can resolve (or handle) exceptions.
- To handle an exception, place any code that might throw an exception in a **try statement**.
- The try statement allows you to define a block of code to be tested for errors while it is being executed.
- The **catch block** contains the code that *handles* the exception.
- The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.
  - You can have many catch blocks to handle different *types* of exceptions that might be thrown in the corresponding try block
  - An exception object's .**toString** or .**getMessage** method returns the exception's error message

Handling Exceptions – Example 1

```
try {
    int nums[] = {3, 5, 9};
    System.out.println(nums[3]);
    System.out.println("nums array size: " +
    nums.length);
}
catch (IndexOutOfBoundsException ex){
    System.err.println(ex.getMessage());
}
```

- The program attempts to access an element *outside* the bounds of the array
  - the array has only 3 elements (with an index 0 to 2).
- JVM throws **`ArrayIndexOutOfBoundsException`** to notify the program of this problem.
- At this point the **`try` block** terminates and the **`catch` block** begins executing
  - if you declared any local variables in the try block, they're now out of scope.

Handling Exceptions – Example 2

```
try {
    int[] nums = null;
    System.out.println("nums array size: " + nums.length);
}
catch (NullPointerException ex){
    System.err.println(ex.toString());
}
```

- A `NullPointerException` occurs when you try to call a method on a **null reference**.
- Ensuring that references are not null before you use them to call methods prevents `Null Pointer Exceptions`.

Handling Exceptions - Example 3

```
//This will generate an error, because myNumbers[10] does not exist.
public class Main {
  public static void main(String[ ] args) {
    int[] myNumbers = {1, 2, 3};
    System.out.println(myNumbers[10]); // error!
  }
}
```

//we can use try...catch to catch the error and execute some code to handle it

```
public class Main {
  public static void main(String[ ] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    }
  }
}
```

Handling Exceptions with Finally – Example 4

The finally statement lets you execute code, after try...catch, regardless of the result:

```
public class Main {
  public static void main(String[] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    } finally {
      System.out.println("The 'try catch' is finished.");
    }
  }
}
```

The output will be:

```
Something went wrong.
The 'try catch' is finished
```