UNIT-2

CONTROL FLOW AND ARRAY

2.1 VARIABLE AND TYPES OF VARIABLE

Variable is a named storage location that holds a value of a particular type. Variables are used to store and manipulate data within a program. Java variables can be classified into different types based on their characteristics and usage.

Here are the common types of variables in Java:

1.Local Variables:

Local variables are declared within a method, constructor, or a block of code.

They are accessible only within the scope where they are declared.

Local variables must be initialized before they are used.

Their lifespan is limited to the block in which they are declared.

EXAMPLE:

```
public class LocalVariableExample
{
  public static void main(String[] args)
 {
  int age = 25; // local variable
System.out.println("Age: " + age);
  }
}
```

2.Instance Variables (Non-Static Variables):

Instance variables are declared within a class but outside of any method, constructor, or block.

Each instance of a class has its own copy of instance variables.

They are initialized with default values if not explicitly assigned.

Instance variables are accessible throughout the class and can be accessed using the object of the class.

EXAMPLE:

```
public class InstanceVariableExample
{
String name; // instance variable


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

```
        InstanceVariableExample obj = new InstanceVariableExample();
        obj.name = "John"; // accessing instance variable using object
        System.out.println("Name: " + obj.name);
    }
}
```

3.Class Variables (Static Variables):

Class variables are declared with the static keyword within a class but outside of any method, constructor, or block.

Unlike instance variables, there is only one copy of each class variable that is shared among all instances of the class.

Class variables are initialized with default values if not explicitly assigned.

They can be accessed using the class name itself, without creating an instance of the class.

EXAMPLE:

```
public class ClassVariableExample
{
 static int count; // class variable

 public static void main(String[] args)
 {
ClassVariableExample.count = 10; // accessing class variable using class name
System.out.println("Count: " + ClassVariableExample.count);
    }
}
```

In addition to the above classifications, variables can also be categorized based on their data types:

Primitive Variables: These variables store simple data types such as int, double, boolean, etc. They hold the actual value.

EXAMPLE:

```
public class PrimitiveVariableExample
{
   public static void main(String[] args)
 {
 int age = 25; // primitive variable
 double height = 1.75;
```

```
boolean isStudent = true;


System.out.println("Age: " + age);
System.out.println("Height: " + height);
System.out.println("Is Student: " + isStudent);
 }
}
```

Reference Variables: These variables hold references to objects in memory rather than the actual object itself. They are used for accessing object properties and invoking methods.

EXAMPLE:

```
public class ReferenceVariableExample
{
 public static void main(String[] args)
{
String name = "John"; // reference variable
int[] numbers = {1, 2, 3, 4, 5};


System.out.println("Name: " + name);
  System.out.println("Numbers: " + Arrays.toString(numbers));
  }
}
```

2.2 TYPE CASTING AND CONVERSION

2.2.1 TYPE CASTING

Type casting refers to the process of converting a value from one data type to another. Java supports two types of casting: implicit casting (widening) and explicit casting (narrowing).

1.Implicit Casting (Widening):

This type of casting takes place when two data types are automatically converted. It is also known as Implicit Conversion. This happens when the two data types are compatible and also when we assign the value of a smaller data type to a larger data type.

EXAMPLE:

```
public class ImplicitCastingExample
{
   public static void main(String[] args) {
  int myInt = 10;
```

```
 double myDouble = myInt; // Implicit casting from int to double
System.out.println("myInt: " + myInt);
System.out.println("myDouble: " + myDouble);
  }
}
```

In this example, the **myInt** variable of type **int** is implicitly casted to a double when assigned to the **myDouble** variable. This is allowed because a double can hold a larger range of values than an int.

2.Explicit Casting (Narrowing):

The process of conversion of higher data type to lower data type is known as narrowing typecasting. It is not done automatically by Java but needs to be explicitly done by the programmer, which is why it is also called explicit typecasting.

EXAMPLE:

```
public class ExplicitCastingExample
{
 public static void main(String[] args) {
 double myDouble = 10.5;
 int myInt = (int) myDouble; // Explicit casting from double to int
 System.out.println("myDouble: " + myDouble);
System.out.println("myInt: " + myInt);
  }
}
```

In this example, the **myDouble** variable of type **double** is explicitly casted to an **int** using the **(int)** notation. This is allowed, but it may result in the loss of precision, as the decimal portion of the **double** value is truncated.

2.2.2 CONVERSION

Type conversion is simply the process of converting data of one data type into another. This process is known as type conversion, typecasting, or even type coercion. The Java programming language allows programmers to convert both primitive as well as reference data types.

Java provides several methods to convert values between different data types.

Here are a few examples:

1.Converting String to int:

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

```
 String numberString = "123";
int number = Integer.parseInt(numberString);
System.out.println("Number: " + number);
 }
}
```

2.Converting int to String:

```
public class IntToStringExample
{
 public static void main(String[] args)
{
   int number = 123;
 String numberString = String.valueOf(number);
System.out.println("Number: " + numberString);
 }
}
```

3.Converting String to double:

```
public class StringToDoubleExample
{
  public static void main(String[] args)
{
  String numberString = "10.5";
 double number = Double.parseDouble(numberString);
 System.out.println("Number: " + number);
  }
}
```

2.3 WRAPPER CLASS

Wrapper classes are used to provide a way to treat primitive data types as objects. Each primitive data type has a corresponding wrapper class in Java. Wrapper classes are mainly used for converting primitive types into objects, enabling them to be used in scenarios that require objects.

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Int

long	Long
float	Float
double	Double

2.3.1 Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Wrapper class Example: Primitive to Wrapper:

```
public class WrapperExample1
{
public static void main(String args[])
{
//Converting int into Integer
int a=20;
Integer i=Integer.valueOf(a);//converting int into Integer explicitly
Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally

System.out.println(a+" "+i+" "+j);
}}
```

Output:

20 20 20

2.3.2 Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing.

Wrapper class Example: Wrapper to Primitive

```
public class WrapperExample2
{
public static void main(String args[])
{
//Converting Integer to int
Integer a=new Integer(3);
int i=a.intValue();//converting Integer to int explicitly
int j=a;//unboxing, now compiler will write a.intValue() internally
```

```
System.out.println(a+" "+i+" "+j);
}}
```

Output:

3 3 3

EXAMPLE:

```
public class WrapperExample
 {
  private int value;

 public WrapperExample(int value) {
this.value = value;
 }
    public int getValue() {
  return value;
 }
  public void setValue(int value) {
    this.value = value;
  }
 public static void main(String[] args) {
WrapperExample wrapper = new WrapperExample(10);
  System.out.println("Initial value: " + wrapper.getValue());

wrapper.setValue(20);
   System.out.println("Updated value: " + wrapper.getValue());
 }
}
```

In this example, we have a **WrapperExample** class that acts as a wrapper for an integer value. It has a private instance variable **value** which holds the wrapped value.

The class provides getter and setter methods (**getValue**() and **setValue**()) to access and modify the wrapped value. In the **main()** method, we create an instance of **WrapperExample**, set an initial value of 10, and then update it to 20. Finally, we print the initial and updated values to the console.

2.4 DECISION AND CONTROL STATEMENTS

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

1.Decision Making statements

- if statements
- switch statement

2.Loop statements

- do while loop
- while loop
- for loop
- for-each loop

3.Jump statements

- break statement
- continue statement

1.Decision Making statements

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

1) **If Statement:**
   In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false.
   In Java, there are four types of if-statements given below.
   - Simple if statement
   - if-else statement
   - if-else-if ladder
   - Nested if-statement

Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax:

```
if(condition)
 {
statement 1; //executes when condition is true
}
```

EXAMPLE:

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y > 20) {
System.out.println("x + y is greater than 20");
}
}
}
```

Output:

x + y is greater than 20

if-else statement

The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

Syntax:

```
if(condition) {
statement 1; //executes when condition is true
}
else{
statement 2; //executes when condition is false
}
```

EXAMPLE:

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y < 10) {
System.out.println("x + y is less than      10");
}   else {
System.out.println("x + y is greater than 20");
}
}
```

```
}
```

Output:

x + y is greater than 20

if-else-if ladder:

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

Syntax:

```
if(condition 1) {
statement 1; //executes when condition 1 is true
}
else if(condition 2) {
statement 2; //executes when condition 2 is true
}
else {
statement 2; //executes when all the conditions are false
}
```

EXAMPLE:

```
public class Student {
public static void main(String[] args) {
String city = "Delhi";
if(city == "Meerut") {
System.out.println("city is meerut");
}else if (city == "Noida") {
System.out.println("city is noida");
}else if(city == "Agra") {
System.out.println("city is agra");
        }else {
System.out.println(city);
}
} }
```

Output:

Delhi

Nested if-statement:

In nested if-statements, the if statement can contain a if or if-else statement inside another if or else-if statement.

Syntax:

```
if(condition 1) {
statement 1; //executes when condition 1 is true
if(condition 2) {
statement 2; //executes when condition 2 is true
}
else{
statement 2; //executes when condition 2 is false
}
}
```

EXAMPLE:

```
public class Student {
public static void main(String[] args) {
String address = "Delhi, India";

if(address.endsWith("India")) {
if(address.contains("Meerut")) {
System.out.println("Your city is Meerut");
}else if(address.contains("Noida")) {
System.out.println("Your city is Noida");
}else {
System.out.println(address.split(",")[0]);
}
}else {
System.out.println("You are not living in India");
}
}  }
```

Output:

Delhi

Switch Statement:

In Java, Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
- Cases cannot be duplicate
- Default statement is executed when any of the case doesn't match the value of expression. It is optional.
- Break statement terminates the switch block when the condition is satisfied. It is optional, if not used, next case is executed.
- While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

Syntax:

```
switch (expression){
 case value1:
 statement1;
break;
   .
  .
 .
case valueN:
statementN;
break;
 default:
 default statement;
}
```

EXAMPLE:

```
public class Student{
public static void main(String[] args) {
int num = 2;
switch (num){
case 0:
System.out.println("number is 0");
break;
```

```
case 1:
System.out.println("number is 1");
break;
default:
System.out.println(num);
}
}
}
```

Output:

2

2. Loop statements

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

- for loop
- while loop
- do-while loop

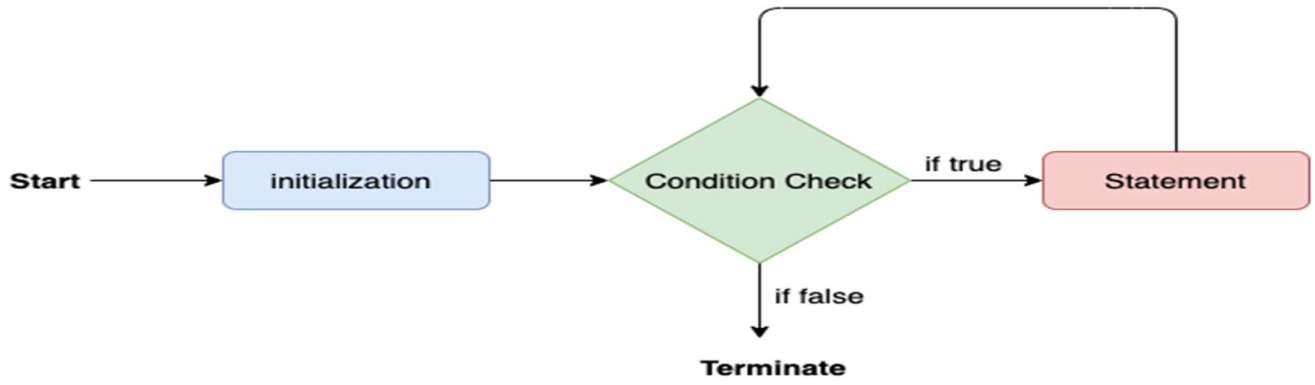
for loop

for loop is similar to C and C++. It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

Syntax:

```
for(initialization, condition, increment/decrement) {
//block of statements
}
```

The flow chart for the for-loop is given below.



EXAMPLE:

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

Output:

The sum of first 10 natural numbers is 55

for-each loop

Java provides an enhanced for loop to traverse the data structures like array or collection. In the for-each loop, we don't need to update the loop variable.

Syntax:

```
for(data_type var : array_name/collection_name){
//statements
}
```

EXAMPLE:

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
String[] names = {"Java","C","C++","Python","JavaScript"};
System.out.println("Printing the content of the array names:\n");
for(String name:names) {
System.out.println(name);
}
}
}
```

Output:

Printing the content of the array names:

Java

C

C++

Python

JavaScript

while loop

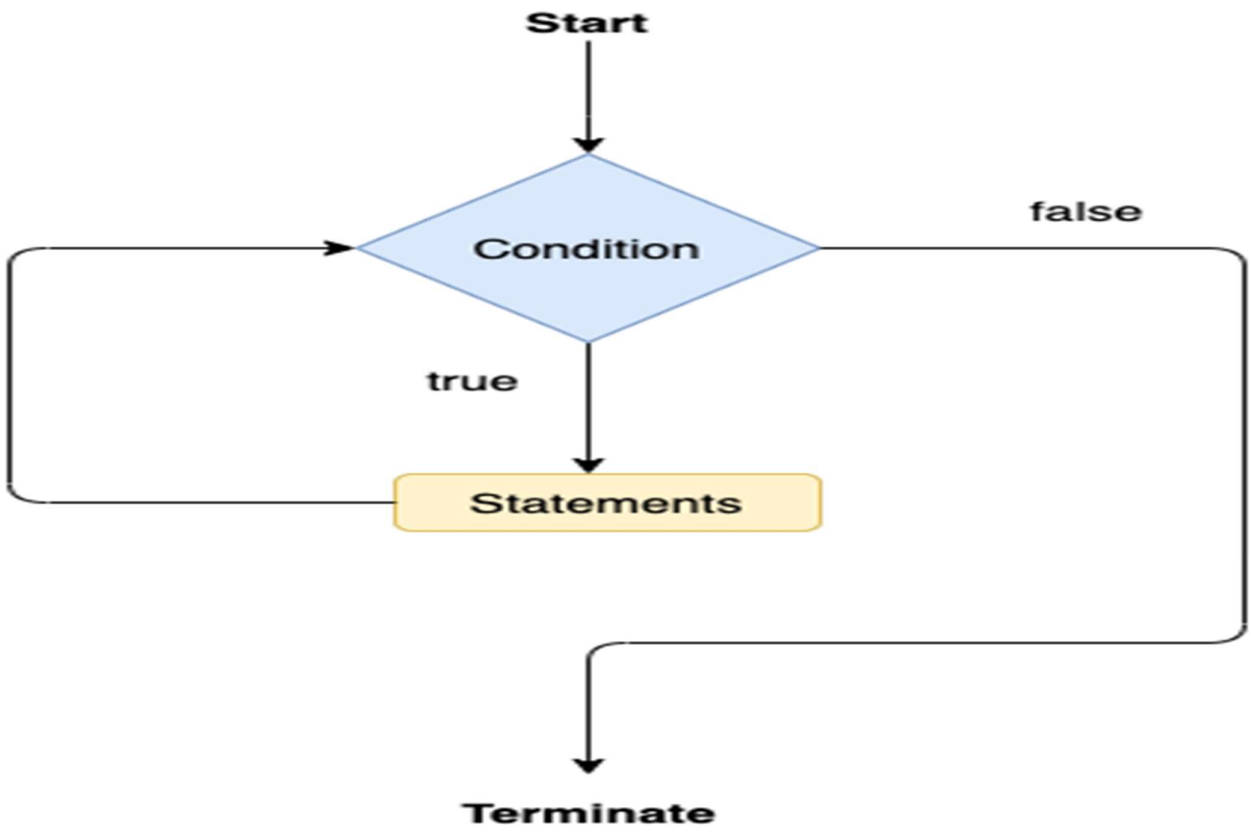
The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

Syntax:

```
while(condition){
//looping statements
}
```

The flow chart for the while loop is given in the following image.



EXAMPLE:

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
while(i<=10) {
System.out.println(i);
i = i + 2;
}
 }
 }
```

Output:

Printing the list of first 10 even numbers

0

2

4

6

8

10

do-while loop

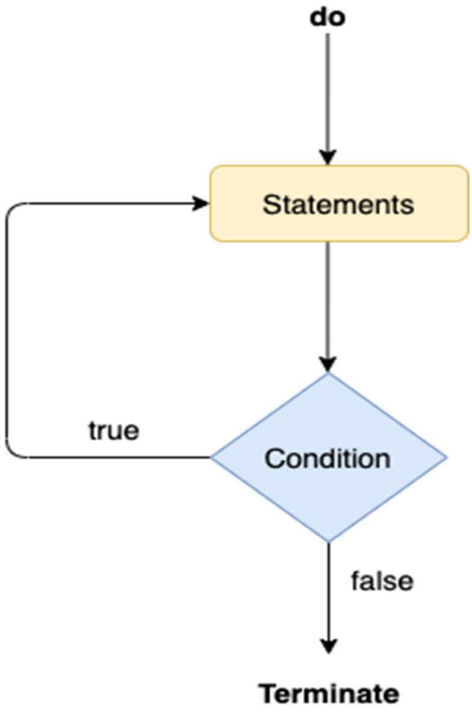
The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.

It is also known as the exit-controlled loop since the condition is not checked in advance.

Syntax:

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

The flow chart of the do-while loop is given in the following image.



EXAMPLE:

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
do {
System.out.println(i);
i = i + 2;
}while(i<=10);
}  }
```

Output:

Printing the list of first 10 even numbers

0

2

4

6

8

10

3. Jump statements

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

Break Statement

As the name suggests, the break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop.

The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

EXAMPLE:

```
public class BreakExample {


public static void main(String[] args) {
// TODO Auto-generated method stub
for(int i = 0; i<= 10; i++) {
System.out.println(i);
if(i==6) {
break;
} } } }
```

Output:

0

1

2

3

4

5

6

Continue Statement

Unlike break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

EXAMPLE:

```
public class ContinueExample {
```

```
public static void main(String[] args) {
// TODO Auto-generated method stub

for(int i = 0; i<= 2; i++) {

for (int j = i; j<=5; j++) {

if(j == 4) {
continue;
}
System.out.println(j);
}
}
}
}
```

Output:

0
1
2
3
5
1
2
3
5
2
3
5

2.5 ARRAY AND TYPES OF ARRAY

Array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

Single Dimensional Array

A single-dimensional array is the most common type of array. It stores elements in a linear fashion, where each element is accessed using its index.

Syntax:

dataType[] arr; (or)

dataType []arr; (or)

dataType arr[];

EXAMPLE:

```
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//traversing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Output:

10

20

70

40

50

Multidimensional Array

A multidimensional array is an array of arrays. It can have two or more dimensions and is represented as a matrix-like structure. Commonly used multidimensional arrays include 2D arrays and 3D arrays.

Syntax:

```
dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];
```

EXAMPLE:

```
class Testarray3{
public static void main(String args[]){
//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
//printing 2D array
for(int i=0;i<3;i++){
for(int j=0;j<3;j++){
  System.out.print(arr[i][j]+" ");
 }
 System.out.println();
}
}}
```

Output:

1 2 3

2 4 5

4 4 5

Dynamic Array (ArrayList):

Unlike regular arrays, the ArrayList class provides dynamic arrays that can grow or shrink at runtime. It is part of the Java Collections Framework and offers additional functionalities such as adding, removing, and searching elements.

EXAMPLE:

```
import java.util.ArrayList;


// Declaration and initialization of an ArrayList
ArrayList<Integer> numbers = new ArrayList< Integer >();


// Adding elements to the ArrayList
```

```
numbers.add(1);
numbers.add(2);
numbers.add(3);

// Accessing elements of the ArrayList
int firstElement = numbers.get(0);  // Accessing the first element
int size = numbers.size();          // Retrieving the size of the ArrayList
```

Other Types of Arrays:

Java also supports arrays of other data types such as boolean, char, double, etc.

EXAMPLE:

```
// Declaration and initialization of a char array
char[] characters = {'a', 'b', 'c'};

// Declaration and initialization of a boolean array
boolean[] flags = {true, false, true};

// Declaration and initialization of a double array
double[] decimals = {1.2, 3.4, 5.6};
```

Arrays in Java have a fixed size, meaning you need to specify the size when declaring them (except for ArrayList, which dynamically adjusts its size). However, you can use other data structures like ArrayList or LinkedList if you need a resizable collection.

2.6 GARBAGE COLLECTION

Garbage collection in Java is the process of automatically reclaiming memory occupied by objects that are no longer in use. It helps manage memory efficiently by freeing up resources that are no longer needed by the program.

Here's an overview of how garbage collection works in Java:

Object Lifecycle:

In Java, objects are created using the new keyword and reside in the heap memory. When an object is no longer referenced by any part of the program, it becomes eligible for garbage collection.

How can an object be unreferenced?

There are many ways:

1) By nulling a reference:

```
Employee e=new Employee();
e=null;
```

2) By assigning a reference to another:

Employee e1=new Employee();

Employee e2=new Employee();

e1=e2;//now the first object referred by e1 is available for garbage collection

3) By anonymous object:

new Employee();

EXAMPLE:

```
public class TestGarbage1{
 public void finalize(){System.out.println("object is garbage collected");}
public static void main(String args[]){
TestGarbage1 s1=new TestGarbage1();
TestGarbage1 s2=new TestGarbage1();
  s1=null;
 s2=null;
System.gc();
 }
}
```

Output:

object is garbage collected

object is garbage collected

2.7 COMMAND LINE ARGUMENTS

Command line arguments in Java are the parameters that are passed to a Java program when it is executed from the command line or terminal. These arguments provide a way to customize the behavior of the program without modifying the source code.

Here's how you can access command line arguments in Java:

Command Line Argument Format:

In Java, command line arguments are passed as strings, separated by spaces. Each argument is treated as a separate element in the args array.

Accessing Command Line Arguments:

The command line arguments are available to the Java program through the args parameter of the main() method. The args parameter is an array of strings.

```
public class CommandLineArguments {
  public static void main(String[] args) {
  // Accessing command line arguments
        for (int i = 0; i <args.length; i++) {
   System.out.println("Argument " + i + ": " + args[i]);}}}
```

In the above example, the main() method accepts an array of strings named args. The program uses a for loop to iterate over the args array and prints each argument along with its index.

Running a Java Program with Command Line Arguments:

To run a Java program with command line arguments, open the command prompt or terminal, navigate to the directory containing the compiled .class file, and use the java command followed by the program name and the arguments.

java CommandLineArguments arg1 arg2 arg3

In the above command, CommandLineArguments is the name of the Java program, and arg1, arg2, and arg3 are the command line arguments that will be passed to the program.

Example of command-line argument

```
class CommandLineExample{
public static void main(String args[]){
System.out.println("Your first argument is: "+args[0]);
}  }
```

Output:

Your first argument is: sonoo