



**CSEg2202**

# **Object Oriented Programming**

---

## **CHAPTER - 2**

# **Object and Class**

Pr.by - Amir Ibrahim  
2023

# Identifier

- Identifiers are names given to elements in a program, such as **variables, functions, constants, classes, objects, arrays, maps**, etc.
- Identifiers help programmers to refer to the elements and manipulate them in the code.
- Identifiers must follow certain rules and conventions depending on the programming language.
- Identifiers should be meaningful and descriptive to make the code more readable and maintainable.

# Java –Identifier naming rules

- ✓ Can contain **letters, digits, underscores** ( ) and **dollar signs** ( )
- ✓ Should begin with a letter, an underscore ( ) or a dollar sign ( ).
- ✓ Should not contain **special characters** ( !, @, #, %, ^, &, \*, >, ?, / ).
- ✓ Should not contain **space**.
- ✓ Are **case sensitive**, meaning that myVar and myvar are different identifiers.

# Java –Identifier naming rules

- ✓ Should not be Java **reserved** or **keywords**, such as int, double, class, etc.
- ✓ Should be **descriptive** and **meaningful**, but not too long.
- ✓ Should follow the **Java naming conventions**, such as using camelCase for variables and methods, and PascalCase for classes and interfaces.

# Java –Keywords

Java **keywords** or **reserved** words are words that have a special meaning or function in the Java programming language. They cannot be used as identifiers for variables, methods, classes or any other elements.

abstarct	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

# Java –Identifier example

Valid Identifiers	Invalid identifiers
minutesPerHour	sum_&_difference (contains a special character)
_nextPage	123myVariable (starts with a digit)
\$test_variable	a+c (contains a special character)
numbers123	variable-2 (contains a special character)
First_Name	my Variable (contains a space)
B5	while (while is a keyword)

# Variables

- Variables are names given to memory locations that store data values during program execution.
- Variables have a **data type** that specifies the type and size of the value they can hold.
- Variables can be declared using the **syntax**:

*dataType variableName = value;*

- Example : *int counter=0;*
- Variable name should follow valid identifier naming rule.

# Variables

The Java programming language defines the following kinds of variables:

1. Instance Variables (Non-Static Fields)
2. Class Variables (Static Fields)
3. Local Variables
4. Parameters



# Variables

## **1. Instance Variables (Non-Static Fields) :**

- Declared within a class but outside of any method or block of code.
- They are accessible to all methods and constructors in the class.

# Variables

## 2. Class Variables (Static Fields)

- Declared with the **static** keyword.
- They are shared by all objects of the class and are only initialized once, when the class is loaded.

# Variables

## 3. Local variables

- Declared within a method or block of code.
- They are only accessible within the method or block in which they are declared.

# Variables

## 4. Parameters

- Used to pass information between methods or classes.
- Can be declared in the method signature.
- Can also be declared in a constructor.

# Java – Data types

Java has two types of data types: primitive and reference.

- ✓ **Primitive data types** : are the basic building blocks of data in Java.
  - They are used to store simple values, such as numbers, characters, and Boolean values.
- ✓ **Reference (Non Primitive) data types** : are used to store references to objects.
  - Objects are more complex than primitive values, and they can contain multiple data fields and methods. Includes Classes, Interfaces, and Arrays.

# Java – Primitive data types

Data type	Description	Range	Default value
boolean	A Boolean value, either true or false.		false
byte	A signed 8-bit integer.	-128 to 127	0
short	A signed 16-bit integer.	-32768 to 32767	0
int	A signed 32-bit integer.	-2147483648 to 2147483647	0
long	A signed 64-bit integer.	-9223372036854775808 to 9223372036854775807	0L
float	A single-precision 32-bit floating-point number.	$\pm 3.402823466E+38$	0.0f
double	A double-precision 64-bit floating-point number.	$\pm 1.7976931348623157E+308$	0.0d
char	A single character, represented by its Unicode code point.		'\u0000'

# Java – Default value

- Fields that are declared but not initialized will be set to a reasonable default by the compiler.
- Local variables are slightly different; the compiler never assigns a default value to an uninitialized local variable.
- If you cannot initialize your local variable where it is declared, make sure to assign it a value before you attempt to use it.
- Accessing an uninitialized local variable will result in a **compile-time error**.

# Java – Reference data types

Data type	Description
Object	The base class for all other reference types.
Class	A blueprint for creating objects.
Interface	A contract that defines a set of methods that must be implemented by a class.
String	A sequence of characters.
Array	A collection of objects of the same type.
Exception	An object that represents an error condition.



# Java – Literal

- **Literal** is a constant value that appears directly in the source code.
- Literals can be used to represent numbers, characters, strings, and boolean values.

## **Boolean literals**

- A boolean literal in Java is a literal that represents a logical value of either true or false.
- It is declared using the keyword boolean.
- Boolean literals can also use the values of “0” and “1.”

# Java – Literal

## Integer literal :

```
int decVal = 26;          // The number 26, in decimal
```

```
int hexVal = 0x1a;        // The number 26, in hexadecimal
```

```
int binVal = 0b11010;     // The number 26, in binary
```

```
long longVal= 243L        // long
```

# Java – Literal

## **Floating-Point Literals:**

```
double d1 = 123.4;
```

```
// same value as d1, but in scientific notation
```

```
double d2 = 1.234e2;
```

```
float f1  = 123.4f;
```

# Java – Literal

## Character and String Literals

Literals of types char and String may contain any Unicode (UTF-16) characters.

- 'single quotes' for char literals

(Example : 'a', '1', '\$', '\u0108' )and

- "double quotes" for String literals

(Examples : "Hello, world! ", "1234567890" , "This is a string. " ,  
"S\u00ED Se\u00F1or“)

# Java – Escape Sequence characters

- An **escape sequence character** is a special character in Java that starts with a backslash (\) and is followed by another character.
- Escape sequence characters are used to represent characters that cannot be typed directly, such as newline (\n), tab (\t), or quotation marks (\")

# Java – Escape Sequence characters

`\b` (backspace),

`\t` (tab),

`\n` (line feed),

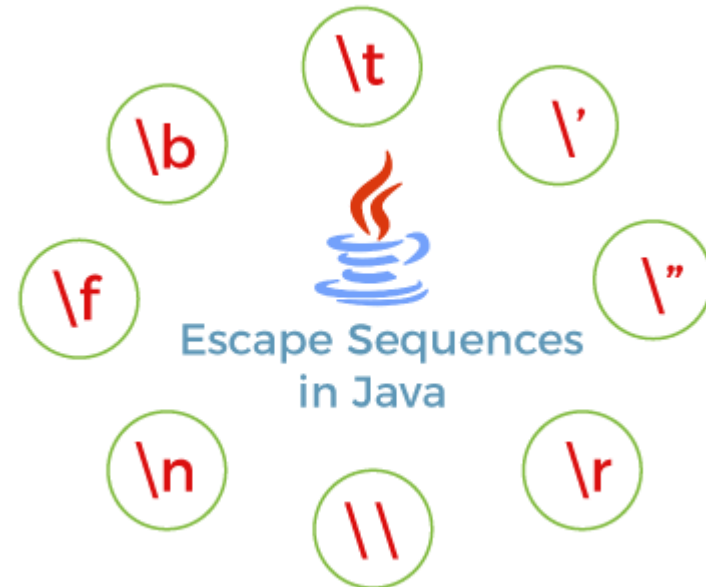
`\f` (form feed),

`\r` (carriage return),

`\"` (double quote),

`'` (single quote), and

`\\` (backslash).



# Java – Escape Sequence characters

```
String str = "Hello\tEthiopia";  
System.out.println(str);  
String str1 = "Welcome \nto \nJava.";   
System.out.println(str1);  
String str2 = "Basketball\\Football";  
System.out.println(str2);  
String str3 = "Carriage\rReturn";  
System.out.println(str3);  
String str4 = "The squirrel\'s";  
System.out.println(str4);  
String str5 = "\"Great man\"";  
System.out.println(str5);
```

output

```
Hello    Ethiopia  
Welcome  
to  
Java.  
Basketball\Football  
Return  
The squirrel's  
"Great man"
```

# Java – Comments

- **Comments** are annotations that can be added to the source code to provide additional information or explanation.
- They are ignored by the compiler and do not affect the execution of the program. Java comments can be used for various purposes, such as:
  - ✓ Documenting the functionality and purpose of the code
  - ✓ Making the code more readable and understandable
  - ✓ Debugging or testing the code by temporarily disabling some statements.
  - ✓ Generating documentation using tools like Javadoc



# Java – Comments

There are three types of java comments

## **1. Single-line comments:**

- They start with two forward slashes (//) and comment only one line of code.
- They are widely used for short comments.

# Java – Comments

## 2. Multi-line comments:

- They start with `/*` and end with `*/` and comment multiple lines of code.
- They can be used to explain complex code snippets or comment out multiple lines of code at a time.

# Java – Comments

## 3. Documentation comments:

- Java **documentation comments** are special comments that are used to generate HTML documentation for Java source code using the Javadoc tool.
- They are also known as doc comments or Javadoc comments. They start with `/**` and end with `*/`, and can have multiple lines.
- They help to create documentation API using the javadoc tool.
- They can use various tags and HTML elements to depict parameters, headings, author names, etc.

# Java – Comments

```
/**
 * This is a documentation comment for the class Example
 * @author John Doe
 * @version 1.0
 */
public class Example {
    public static void main(String[] args) {
        // This is a single-line comment
        System.out.println("Hello World"); // This is another single-line comment

        /* This is a multi-line comment
           It can span multiple lines
           It can also be used to comment out code
           System.out.println("This will not be executed");
        */
    }
}
```

# Java – Operators

- ✓ **Operators** are special symbols that perform specific operations on one, two, or three operands, and then return a result.
- ✓ **Operator precedence** is a rule in programming languages that determines the order in which operators are evaluated in an expression.
- ✓ Operators with higher precedence are evaluated before operators with lower precedence.
- ✓ The operators in the following table are listed according to precedence order.

# Java – Operators

Operators	Precedence
postfix	expr++ expr--
unary	++expr --expr +expr -expr ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	
logical AND	&&
logical OR	
ternary	? :
assignment	= += -= *= /= %= &= ^=  = <<= >>= >>>=

# Java – Operators

## Simple Assignment Operator

=      Simple assignment operator

## Arithmetic Operators

- +      Additive operator (also used for String concatenation)
- Subtraction operator
- \*      Multiplication operator
- /      Division operator
- %      Remainder operator

# Java – Operators

```
// Declare and initialize some variables
int x = 10;
int y = 20;
int z = 3;

// Perform and print some arithmetic operations
System.out.println("x = " + x);
System.out.println("y = " + y);
System.out.println("z = " + z);

System.out.println("x + y = " + (x + y)); // Addition
System.out.println("x - y = " + (x - y)); // Subtraction
System.out.println("x * y = " + (x * y)); // Multiplication
System.out.println("x / y = " + (x / y)); // Division
System.out.println("x % z = " + (x % z)); // Modulus
```

output

```
x = 10
y = 20
z = 3
x + y = 30
x - y = -10
x * y = 200
x / y = 0
x % z = 1
```



# Java – Operators

## Unary Operators

- + Unary plus operator; indicates positive value
- Unary minus operator; negates an expression
- ++ Increment operator; increments a value by 1
- Decrement operator; decrements a value by 1
- ! Logical complement operator; inverts the value of a boolean

# Java – Operators

```
int x = +10; // unary plus operator
System.out.println("x = " + x);

int y = -10; // unary minus operator
System.out.println("y = " + y);

boolean cond = true;
cond = !cond; // logical complement operator
System.out.println("cond = " + cond);

int z = 10;
z++; // postfix increment operator
System.out.println("z++ = " + z);
++z; // prefix increment operator
System.out.println("++z = " + z);

int w = 10;
w--; // postfix decrement operator
System.out.println("w-- = " + w);
--w; // prefix decrement operator
System.out.println("--w = " + w);
```

output

```
x = 10
y = -10
cond = false
z++ = 11
++z = 12
w-- = 9
--w = 8
```

# Java – Operators

## Equality and Relational Operators

`==`    Equal to

`!=`    Not equal to

`>`    Greater than

`>=`    Greater than or equal to

`<`    Less than

`<=`    Less than or equal to

# Java – Operators

```
int x = 10;
int y = 20;

// Perform and print some relational operations
System.out.println("x = " + x);
System.out.println("y = " + y);

System.out.println("x == y: " + (x == y)); // Equal to
System.out.println("x != y: " + (x != y)); // Not equal to
System.out.println("x > y: " + (x > y)); // Greater than
System.out.println("x < y: " + (x < y)); // Less than
System.out.println("x >= y: " + (x >= y)); // Greater than or equal to
System.out.println("x <= y: " + (x <= y)); // Less than or equal to
```

output

```
x = 10
y = 20
x == y: false
x != y: true
x > y: false
x < y: true
x >= y: false
x <= y: true
```

# Java – Operators

## Logical Operators

&&      Conditional-AND

||        Conditional-OR

!        Negation -NOT

## Conditional operator

?:       Ternary (shorthand for if-then-else statement)

## Type Comparison Operator

instanceof      Compares an object to a specified type

# Java – Operators

```
//Variables Definition and Initialization
```

```
boolean bool1 = true, bool2 = false;
```

```
//Logical AND
```

```
System.out.println("bool1 && bool2 = " + (bool1 && bool2));
```

```
//Logical OR
```

```
System.out.println("bool1 || bool2 = " + (bool1 || bool2));
```

```
//Logical Not
```

```
System.out.println("! (bool1 && bool2) = " + !(bool1 && bool2));
```

output

```
bool1 && bool2 = false
```

```
bool1 || bool2 = true
```

```
! (bool1 && bool2) = true
```

# Java – Operators

```
//Variables Definition and Initialization
```

```
int x = 10, y = 20, z = 15;
```

```
//Ternary operator to compare and return the largest of two variables
```

```
int maxOfTwo = x > y ? x : y;
```

```
System.out.println("The largest of x and y is: " + maxOfTwo);
```

```
//Ternary operator to compare and return the largest of three variables
```

```
int maxOfThree = x > y ? (x > z ? x : z) : (y > z ? y : z);
```

```
System.out.println("The largest of x, y and z is: " + maxOfThree);
```

output

The largest of x and y is: 20

The largest of x, y and z is: 20

# Java – Operators

```
// superclass
class Animal {
}

// subclass
class Dog extends Animal {
}

public class InstanceOfExample {
    public static void main(String[] args) {
        // create an object of the subclass
        Dog d1 = new Dog();

        // checks if d1 is an instance of the subclass
        System.out.println(d1 instanceof Dog); // prints true

        // checks if d1 is an instance of the superclass
        System.out.println(d1 instanceof Animal); // prints true
    }
}
```

output

true

true



# Java – Operators

## Bitwise and Bit Shift Operators

~      Unary bitwise complement

<<    Signed left shift

>>    Signed right shift

>>>   Unsigned right shift

&      Bitwise AND

^      Bitwise exclusive OR

|      Bitwise inclusive OR

# Java – Operators

```
//Variables Definition and Initialization
```

```
int x = 10, y = 6;
```

```
//Bitwise AND
```

```
System.out.println("x & y = " + (x & y));
```

```
//Bitwise OR
```

```
System.out.println("x | y = " + (x | y));
```

```
//Bitwise XOR
```

```
System.out.println("x ^ y = " + (x ^ y));
```

```
//Bitwise Complement
```

```
System.out.println("~x = " + ~x);
```

```
//Left Shift
```

```
System.out.println("x << 2 = " + (x << 2));
```

```
//Right Shift
```

```
System.out.println("y >> 1 = " + (y >> 1));
```

output

x & y = 2

x | y = 14

x ^ y = 12

~x = -11

x << 2 = 40

y >> 1 = 3

# Basic Java program structure

- Java Program Structure is the way a Java program is organized and written.
- It consists of different sections that have specific roles and functions.
- A basic Java program structure involves the following sections:
  - ✓ Documentation section
  - ✓ Package declaration
  - ✓ Import statements
  - ✓ Interface section
  - ✓ Class definition
  - ✓ Main method

# Basic Java program structure

## Documentation section

- ✓ It is an optional section that contains basic information about the program, such as author name, date, version, description, etc.
- ✓ It uses comments to write the information.

## Package declaration

- ✓ It is an optional section that declares the package name in which the class is placed.
- ✓ It helps to organize the classes into different modules and directories.
- ✓ It uses the keyword **package** to declare the package name..

# Basic Java program structure

## Import statements

- ✓ They are used to import the classes or interfaces from other packages that are needed in the program.
- ✓ They use the keyword **import** to import the classes or interfaces.

## Interface section

- ✓ It is an optional section that declares one or more interfaces that are implemented by the class.
- ✓ An interface is a collection of abstract methods and constants that define a common behavior for a class.

# Basic Java program structure

## Class definition

- ✓ It is a mandatory section that defines the class name, variables, methods and constructors of the program.
- ✓ A class is a blueprint for creating objects that have state and behavior. It uses the keyword **class** to define a class.

## Main method

- ✓ It is a mandatory method that is the entry point of any Java program.
- ✓ It contains the instructions for executing the program.
- ✓ It uses the keyword **public static void main(String[] args)** to declare the main method.

# Basic Java program structure

```
// This is a simple program that prints hello world
public class HelloWorld {
    // This is the main method that runs the program
    public static void main(String[] args) {
        // This is a statement that prints hello world to the screen
        System.out.println("Hello World");
    }
}
```

# Expressions, Statements, and Blocks

## Expressions

- An **expression** is a construct made up of variables, operators, and method invocations, that evaluates to a single value.
- The data type of the value returned by an expression depends on the elements used in the expression.
- **Example:** of expression highlighted below



# Expressions, Statements, and Blocks

## Expressions

```
int cadence = 0;  
anArray[0] = 100;  
System.out.println("Element 1 at index 0: " + anArray[0]);  
  
int result = 1 + 2; // result is now 3  
if (value1 == value2) {  
    System.out.println("value1 == value2");  
}
```

# Expressions, Statements, and Blocks

## Statements

- Statements are roughly equivalent to sentences in natural languages.
- A statement forms a complete unit of execution.
- The following types of expressions can be made into a statement by terminating the expression with a semicolon (;).
  - ✓ Assignment expressions
  - ✓ Any use of ++ or --
  - ✓ Method invocations
  - ✓ Object creation expressions

# Expressions, Statements, and Blocks

## Statements

- Such statements are called expression statements. Here are some examples of expression statements.

```
// assignment statement
aValue = 8933.234;
// increment statement
aValue++;
// method invocation statement
System.out.println("Hello World!");
// object creation statement
Bicycle myBike = new Bicycle();
```

# Expressions, Statements, and Blocks

## Statements

- In addition to expression statements, there are two other kinds of statements: declaration statements and control flow statements.

- A declaration statement declares a variable.

```
// declaration statement
```

```
double aValue = 8933.234;
```

- Control flow statements regulate the order in which statements get executed.

# Expressions, Statements, and Blocks

## Block

- A block is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed.

```
class BlockDemo {  
    public static void main(String[] args) {  
        boolean condition = true;  
        if (condition) { // begin block 1  
            System.out.println("Condition is true.");  
        } // end block one  
        else { // begin block 2  
            System.out.println("Condition is false.");  
        } // end block 2  
    }  
}
```

# Control flow structures

**Conditional Branches** :are used to choose between two or more paths. There are three types of conditional branches in Java:

- **if/else/else if statements:** These statements allow you to check for a condition and execute different code depending on whether the condition is true or false.
- **Ternary operator:** The ternary operator is a shortcut for writing an if/else statement. It takes three operands: a condition, a value if the condition is true, and a value if the condition is false.
- **switch statements:** Switch statements allow you to choose between multiple paths based on the value of a variable.

# Control flow structures

**Loops** are used to repeatedly execute a block of code. There are three types of loops in Java:

- **for loops:** For loops allow you to iterate over a range of numbers or a collection of objects.
- **while loops:** While loops allow you to execute a block of code as long as a condition is true.
- **do-while loops:** Do-while loops are similar to while loops, but the block of code is executed at least once, even if the condition is false.

# Control flow structures

**Branching Statements** are used to alter the flow of control in loops. There are three types of branching statements in Java:

- **break statements:** Break statements are used to exit a loop.
- **continue statements:** Continue statements are used to skip the current iteration of a loop and continue with the next iteration.
- **return :** used to exit from a method, with or without value.



# Control flow structures

## The if-then Statement

It tells your program to execute a certain section of code only if a particular test evaluates to true.

the **opening and closing braces are optional**, provided that the "then" clause contains only one statement:

```
public void applyBrakes() {  
    // Check if the bicycle is moving  
    if (isMoving) {  
        // Decrease the current speed  
        currentSpeed--;  
    }  
}
```

# Control flow structures

## The if-then-else Statement

The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false.

```
void applyBrakes() {  
    if (isMoving) {  
        currentSpeed--;  
    } else {  
        System.err.println("The bicycle has already stopped!");  
    }  
}
```

# Control flow structures

## The if-then-else Statement

```
public class GradeGenerator {  
    public static void main(String[] args) {  
        double result = 85.5;  
        char grade;  
        if (result >= 90) {  
            grade = 'A';  
        } else if (result >= 80) {  
            grade = 'B';  
        } else if (result >= 70) {  
            grade = 'C';  
        } else if (result >= 60) {  
            grade = 'D';  
        } else {  
            grade = 'F';  
        }  
        System.out.println("The grade is: " + grade);  
    }  
}
```

# Control flow structures

## The switch Statement

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

# Control flow structures

## The switch Statement

```
public class MonthGenerator {
    public static void main(String[] args) {
        int month = 6;
        String monthName;
        switch (month) {
            case 1:
                monthName = "January";
                break;
            case 2:
                monthName = "February";
                break;
            case 3:
                monthName = "March";
                break;
            case 4:
                monthName = "April";
                break;
            case 5:
                monthName = "May";
                break;
            case 6:
                monthName = "June";
                break;
            case 7:
                monthName = "July";
                break;
            case 8:
                monthName = "August";
                break;
            case 9:
                monthName = "September";
                break;
            case 10:
                monthName = "October";
                break;
            case 11:
                monthName = "November";
                break;
            case 12:
                monthName = "December";
                break;
            default:
                monthName = "Invalid month";
        }
        System.out.println("The month name is: " + monthName);
    }
}
```

# Control flow structures

## The while and do-while Statements

The difference between while and do while loops in Java is that

- **while loop** checks the condition before executing the statements inside the loop, whereas
- **do while loop** checks the condition after executing the statements inside the loop.

This means that while loop may execute zero times if the condition is false initially, but do while loop will execute at least once regardless of the condition.

# Control flow structures

## The while and do-while Statements

The syntax for while loop

```
while (testExpression) {  
    // body of loop  
}
```

The syntax for do-while loop

```
do {  
    // body of loop  
} while (testExpression);
```

# Control flow structures

```
class Main {  
    public static void main(String[] args) {  
        // declare variables  
        int i = 1;  
        int j = 1;  
  
        // while loop example  
        System.out.println("Using while loop:");  
        while (i > 0) { // condition is false initially  
            System.out.println(i); // this statement is never executed  
            i++;  
        }  
  
        // do while loop example  
        System.out.println("Using do while loop:");  
        do {  
            System.out.println(j); // this statement is executed once  
            j++;  
        } while (j > 0); // condition is false after the first iteration  
    }  
}
```

output

Using while loop:

Using do while loop:

1



# Control flow structures

## The for Statement

The **for loop** statement provides a compact way to iterate over a range of values.

```
for (initialization; condition; increment/decrement) {  
    // body of loop  
}
```

# Control flow structures

## The for Statement

When using this version of the for statement, keep in mind that:

- ✓ **The initialization** expression initializes the loop; it's executed once, as the loop begins.
- ✓ When the **condition** expression evaluates to false, the loop terminates.
- ✓ The **increment/decrement** expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment or decrement a value.

# Control flow structures

```
// Program to display all multiples of 9 under 100
class Main {
    public static void main(String[] args) {
        // declare variables
        int n = 9; // the number to find the multiples of
        int limit = 100; // the upper limit

        // for loop from 1 to limit/n
        for (int i = 1; i <= limit/n; i++) {
            // print the multiple of n
            System.out.println(n * i);
        }
    }
}
```

output

9  
18  
27  
36  
45  
54  
63  
72  
81  
90  
99

---

# Control flow structures

## The break Statement

- The break statement in Java is used to terminate a loop or a switch statement immediately.
- It breaks the current flow of the program at a specified condition .
- The break statement has two forms: labeled and unlabeled.

Unlabeled

```
break;
```

Labeled

```
break label;
```

# Control flow structures

```
// Example 1: break statement in a for loop
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break; // terminate the loop when i is 5
    }
    System.out.println(i);
}
```

```
// Example 2: break statement in a while loop
int j = 0;
while (j < 10) {
    System.out.println(j);
    j++;
    if (j == 4) {
        break; // terminate the loop when j is 4
    }
}
```

```
// Example 3: labeled break statement in a nested loop
outer: // label for outer loop
for (int k = 0; k < 5; k++) {
    inner: // label for inner loop
    for (int l = 0; l < 5; l++) {
        if (l == 3) {
            break outer; // terminate the outer loop when l is 3
        }
        System.out.println("k=" + k + ", l=" + l);
    }
}
```

# Control flow structures

## The continue Statement

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

Unlabeled

```
continue;
```

Labeled

```
continue label;
```

# Control flow structures

```
// Example 1: continue statement in a for loop
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        continue; // skip the rest of the loop body when i is 5
    }
    System.out.println(i);
}
```

```
// Example 3: labeled continue statement in a nested loop
outer: // label for outer loop
for (int k = 0; k < 5; k++) {
    inner: // label for inner loop
    for (int l = 0; l < 5; l++) {
        if (l == 3) {
            continue outer; // skip the rest of the outer loop body when l is 3
        }
        System.out.println("k=" + k + ", l=" + l);
    }
}
```

```
// Example 2: continue statement in a while loop
int j = 0;
while (j < 10) {
    j++;
    if (j == 4) {
        continue; // skip the rest of the loop body when j is 4
    }
    System.out.println(j);
}
```

# Control flow structures

## The return Statement

- The return statement exits from the current method, and control flow returns to where the method was invoked.
- The return statement has two forms: one that returns a value, and one that doesn't.

```
return ++count;
```

- The data type of the returned value must match the type of the method's declared return value. When a method is declared void, use the form of return that doesn't return a value.

```
return;
```



# Array

- An array in Java is a group of **like-typed** variables referred to by a common name.
- Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.
- Arrays are **dynamically allocated** in Java, which means they are created at run time and their size can change.
- Arrays are stored in **contiguous memory** locations, which means they are placed next to each other in memory

# Array

## Declaring and Initializing Arrays

- To declare an array, define the variable type with square brackets:  
**String[] cars;**
- To initialize an array, you can place the values in a comma-separated list, inside curly braces: **String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};**
- Alternatively, you can specify the size of the array and assign values to each element using the index:

```
int[] myNum = new int[4];  
myNum[0] = 10;  
myNum[1] = 20;
```

# Array

## Accessing and Modifying Array Elements

- You can access an array element by referring to the index number:  
**`System.out.println(cars[0]);`** // Outputs Volvo
- Note that array indexes start with 0: [0] is the first element, 1 is the second element, etc.
- To change the value of a specific element, refer to the index number:  
`cars[0] = "Opel";` // Now outputs Opel instead of Volvo

# Array

## Array Length and Copying

- To find out how many elements an array has, use the length property:  
`System.out.println(cars.length); // Outputs 42`
- To copy an array, you can use the `clone()` method or the `System.arraycopy()` method.
- The `clone()` method returns a new array object with the same elements as the original array: `int[] copy = myNum.clone();`
- The `System.arraycopy()` method copies a source array to a destination array: `System.arraycopy(myNum, 0, copy, 0, myNum.length);`

# Array

## Arrays Class

- The Arrays class in the java.util package provides various methods for manipulating arrays (such as sorting and searching).
- Some of the methods in the Arrays class are:
- `Arrays.sort(array)`: Sorts the specified array into ascending order.
- `Arrays.binarySearch(array, key)`: Searches the specified array for the specified value using the binary search algorithm.

# Array

## Arrays Class

- `Arrays.equals(array1, array2)`: Returns true if the two specified arrays are equal to one another.
- `Arrays.fill(array, value)`: Assigns the specified value to each element of the specified array.
- `Arrays.asList(array)`: Returns a fixed-size list backed by the specified array.

```
// declare an array of doubles
double[] data;

// allocate memory for 5 elements
data = new double[5];

// assign values to each element
data[0] = 1.2;
data[1] = 3.4;
data[2] = 5.6;
data[3] = 7.8;
data[4] = 9.0;

// print the array elements using a for loop
for (int i = 0; i < data.length; i++) {
    System.out.println(data[i]);
}
```

```
// declare and initialize an array of strings
String[] items = {"item 1", "item 2", "item 3"};

// print the array elements using a for loop
for (int i = 0; i < items.length; i++) {
    System.out.println(items[i]);
}
```

```
// declare and initialize an array of integers
int[] numbers = {10, 20, 30, 40, 50};

// print the array elements using the Arrays.toString() method
System.out.println(Arrays.toString(numbers));
```

# Classes and objects

- A **class** is like a blueprint or a template for creating objects.
- A class defines the attributes (data) and behaviors (methods) of the objects of that class.
- An **object** is an instance of a class.
- It has its own state (values of the attributes) and can perform actions (invoke methods) defined by the class.
- A class can be `Car`, and an object can be `myCar`, which is a specific car with a certain color, model, speed, etc.



# Classes and objects

- **For example**, in real life, a car is an object that has **attributes** like color, weight, model, etc. and **methods** like drive, brake, etc.
- A class named Car can define these attributes and methods for all car objects.

# Classes and objects

## Declaring Classes

- A class is defined using the keyword `class`, followed by the name of the class and a pair of curly braces `{ }` that enclose the body of the class.
- The name of the class should start with a capital letter and follow the camel case convention (e.g. `MyClass`, `Student`, `BankAccount`).
- The body of the class can contain fields (variables that store data) and methods (functions that perform actions).

# Classes and objects

## Declaring Classes

The class body (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class:

- ✓ constructors for initializing new objects,
- ✓ declarations for the fields that provide the state of the class and its objects, and
- ✓ methods to implement the behavior of the class and its objects.

# Classes and objects

## Declaring Classes

```
class MyClass {  
    // field, constructor, and  
    // method declarations  
}
```

```
class Car {  
    // fields  
    String color;  
    String model;  
    String brand;  
  
    // methods  
    void start() {  
        // code to start the car  
    }  
  
    void accelerate(int amount) {  
        // code to increase the speed by amount  
        // speed += amount; // this line is no longer valid  
    }  
  
    void stop() {  
        // code to stop the car  
        // speed = 0; // this line is no longer valid  
    }  
}
```

# Classes and objects

## How to create an object?

- An object is created using the keyword **new**, followed by the **name of the class** and a pair of parentheses () that can optionally contain arguments (values passed to the constructor method).

## How to access each fields and methods?

- The fields and methods of an object can be accessed using the **dot operator** ( . ) followed by the name of the field or method.
- The fields can be read or modified, and the methods can be invoked with or without arguments.

# Classes and objects

**Example :** for the car class created above the object can be created and the fields and methods can be accessed as follows.

```
// create an object of Car class using default constructor
Car myCar = new Car();

// access and modify the fields using dot operator
myCar.color = "yellow";
myCar.model = "Truck";
myCar.brand = "Tesla";

// access and invoke the methods using dot operator
myCar.start();
myCar.accelerate(10);
myCar.stop();
```

# Classes and objects

## Example

```
// Define a class person with some fields and methods
class person {
    // A field to store the name of the person
    String name;

    // A field to store the age of the person
    int age;

    // A field to store the gender of the person
    char gender;

    // A method to set the fields of the person
    void setDetails(String name, int age, char gender) {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }
}
```

```
// A method to print the details of the person
void printDetails() {
    System.out.println("The name of this person is " + name);
    System.out.println("The age of this person is " + age);
    System.out.println("The gender of this person is " + gender);
}

// A method to calculate the body mass index (BMI) of the person
// Assuming height and weight are given in meters and kilograms respectively
double calculateBMI(double height, double weight) {
    return weight / (height * height);
}

// A method to greet another person
void greet(person other) {
    System.out.println("Hello, " + other.name + ". I am " + name + ".");
}
}
```

# Classes and objects

## Example (cont...)

```
// Define a main class with a main method
```

```
public class Main {
```

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

```
        // Create two objects of class person
```

```
        person tesfaye = new person();
```

```
        person ahmed = new person();
```

```
        // Set the details of both objects using the setDetails method
```

```
        tesfaye.setDetails("Tesfaye", 27, 'M');
```

```
        ahmed.setDetails("Ahmed", 32, 'M');
```

```
        // Call the printDetails method of both objects
```

```
        tesfaye.printDetails();
```

```
        ahmed.printDetails();
```

```
        // Call the calculateBMI method of both objects with some arguments
```

```
        double bmiTesfaye = tesfaye.calculateBMI(1.7, 60);
```

```
        double bmiAhmed = ahmed.calculateBMI(1.9, 90);
```

```
        // Print the BMI values of both objects
```

```
        System.out.println("The BMI of Tesfaye is " + bmiTesfaye);
```

```
        System.out.println("The BMI of Ahmed is " + bmiAhmed);
```

```
        // Call the greet method of one object with another object as an argument
```

```
        tesfaye.greet(ahmed);
```

```
    }
```

```
}
```



# Access modifiers

**Access modifiers** in Java are keywords that specify the visibility and accessibility of classes, methods, and variables.

They are an important feature of object-oriented programming that allows you to control how your code is accessed by other classes and methods.

There are **four** types of access modifiers in Java:

- 1. public:** The public access modifier makes the class, method, or variable accessible from anywhere in the program. It has the widest scope of accessibility.

# Access modifiers

**2. private:** The private access modifier makes the class, method, or variable accessible only within the same class. It has the narrowest scope of accessibility and is used to hide the implementation details of a class.

**3. protected:** The protected access modifier makes the class, method, or variable accessible within the same package and also by subclasses in other packages. It has a wider scope than private but narrower than public and is used to support inheritance.

# Access modifiers

**4. default:** The default access modifier (also known as package-private) makes the class, method, or variable accessible only within the same package. It has a wider scope than private but narrower than protected and public. It is applied when no explicit access modifier is specified.

# Access modifiers

Modifier	Within Class	Within Package	Outside Package by Subclass	Outside Package
public	Yes	Yes	Yes	Yes
private	Yes	No	No	No
protected	Yes	Yes	Yes	No
default	Yes	Yes	No	No

# Methods

- A **method** in Java is a block of code that performs a specific task.
- You can pass data, known as parameters, into a method.
- Methods are used to perform certain actions, and they are also known as functions.
- A method must be declared within a class.
- It is defined with the name of the method, followed by parentheses ().
- Java provides some pre-defined methods, such as `System.out.println ()`, but you can also create your own methods to perform certain actions.

# Methods

- A method can have parameters that are values passed to it, and a return type that is the value it produces.
- A method must be declared within a class, and it is called by its name followed by parentheses ().
- A method can be public, private, static, or other modifiers that affect its access and behavior.

# Methods

To create a method, we use the following syntax:

```
modifier static returnType methodName(parameter1, parameter2, ...) {  
    // method body  
}
```

To call a method, we use the following syntax:

```
methodName(argument1, argument2, ...);
```

# Methods

- **returnType** specifies what type of value a method returns.
- For example, if a method has an **int** return type then it returns an integer value.
- If the method does not return a value, its return type is **void**.
- **methodName** is an identifier that is used to refer to the particular method in a program.
- The method body includes the programming statements that are used to perform some tasks.
- The **method body** is enclosed inside the curly braces { }



# Methods

## Example

```
class Main {  
    // declare a method  
    public void sayHello() {  
        // code to be executed  
        System.out.println("Hello, world!");  
    }  
  
    public static void main(String[] args) {  
        // create an object of Main  
        Main obj = new Main();  
  
        // call the method  
        obj.sayHello();  
    }  
}
```

output

Hello, world!

# Methods

## Example

```
class Main {  
    // create a method  
    public int addNumbers(int a, int b) {  
        int sum = a + b;  
        // return value  
        return sum;  
    }  
  
    public static void main(String[] args) {  
        int num1 = 25;  
        int num2 = 15;  
  
        // create an object of Main  
        Main obj = new Main();  
  
        // calling method  
        int result = obj.addNumbers(num1, num2);  
  
        System.out.println("Sum is: " + result);  
    }  
}
```

output

Sum is: 40

# Method overloading

- **Method overloading** in Java is a feature that allows us to define multiple methods with the same name but different parameters.
- This way, we can use the same method name for different scenarios and avoid creating multiple methods with different names that do the same thing.
- For example, suppose we want to write a method that calculates the area of different shapes.
- We can use method overloading to define one method named area that takes different parameters depending on the shape.

# Method overloading

In this example, we have declared three static methods with the same name `area` but different parameters. The first method takes one parameter `radius` and returns the area of a **circle**. The second method takes two parameters `length` and `width` and returns the area of a **rectangle**. The third method takes three parameters `base`, `height` and `angle` and returns the area of a **triangle**.

```
class Shape {  
    // declare a static method for circle  
    public static double area(double radius) {  
        // code to be executed  
        return Math.PI * radius * radius;  
    }  
  
    // declare a static method for rectangle  
    public static double area(double length, double width) {  
        // code to be executed  
        return length * width;  
    }  
  
    // declare a static method for triangle  
    public static double area(double base, double height, double angle) {  
        // code to be executed  
        return 0.5 * base * height * Math.sin(angle);  
    }  
}
```

# Method overloading

The compiler determines which method to invoke based on the number and type of arguments passed.

```
class Main {  
    public static void main(String[] args) {  
        // call the static method for circle  
        double circleArea = Shape.area(5.0);  
        System.out.println("The area of circle is: " + circleArea);  
  
        // call the static method for rectangle  
        double rectangleArea = Shape.area(10.0, 20.0);  
        System.out.println("The area of rectangle is: " + rectangleArea);  
  
        // call the static method for triangle  
        double triangleArea = Shape.area(15.0, 12.0, Math.PI / 6);  
        System.out.println("The area of triangle is: " + triangleArea);  
    }  
}
```

output

```
The area of circle is: 78.53981633974483  
The area of rectangle is: 200.0  
The area of triangle is: 64.9519052838329
```

# this keyword

The **this** keyword in Java has various uses, such as:

- Refer to current class instance variables
- Invoke current class constructor
- Invoke current class method
- Return the current class object
- Pass an argument in the method call
- Access the outer class instance from within the inner class

# this keyword

Refer to current class instance variables:

```
public class Main {  
    int x; // instance variable  
    public Main(int x) { // parameter  
        this.x = x; // assign parameter value to instance variable  
    }  
}
```

Invoke current class constructor:

```
public class Main {  
    int x;  
    public Main() { // no-argument constructor  
        this(10); // invoke parameterized constructor with value 10  
    }  
    public Main(int x) { // parameterized constructor  
        this.x = x;  
    }  
}
```

Invoke current class method:

```
public class Main {  
    public void display() {  
        System.out.println("Hello");  
    }  
    public void greet() {  
        this.display(); // invoke display() method of same class  
        System.out.println("World");  
    }  
}
```

# this keyword

Return the current class object:

```
public class Main {  
    int x;  
    public Main(int x) {  
        this.x = x;  
    }  
    public Main add(Main m) {  
        this.x = this.x + m.x; // add x values of two objects  
        return this; // return current object  
    }  
}
```

Pass an argument in the method call:

```
public class Main {  
    int x;  
    public Main(int x) {  
        this.x = x;  
    }  
    public void print(Main m) {  
        System.out.println(m.x); // print x value of argument object  
    }  
    public void show() {  
        this.print(this); // pass current object as argument to print() method  
    }  
}
```



# this keyword

Access the outer class instance from within the inner class:

```
public class Outer {  
    int x = 10; // outer class instance variable  
    class Inner {  
        int y = 20; // inner class instance variable  
        public void show() {  
            System.out.println(Outer.this.x + y); // access outer class instance variable with  
Outer.this  
        }  
    }  
}
```

# this keyword

Access the outer class instance from within the inner class:

```
public class Outer {  
    int x = 10; // outer class instance variable  
    class Inner {  
        int y = 20; // inner class instance variable  
        public void show() {  
            System.out.println(Outer.this.x + y); // access outer class instance variable with  
Outer.this  
        }  
    }  
}
```

# Constructor

- A **constructor** in Java is a special method that is used to initialize objects of a class.
- The constructor is invoked when an object of a class is created using the new keyword.
- The constructor can be used to set initial values for the object attributes, such as fields or properties.
- Constructors can also perform some validation or logic before assigning values to the object attributes.
- Constructors can also invoke other constructors or methods to reuse code and avoid duplication.

# Constructor

There are two types of constructors in Java: default constructor and parameterized constructor.

## 1. **default constructor**

- ✓ Is a constructor that has no parameters and no explicit code.
- ✓ It is provided by the Java compiler if the class does not have any other constructor.
- ✓ The default constructor assigns default values to the object attributes, such as 0 for numeric types, null for reference types, and false for boolean types.

# Constructor

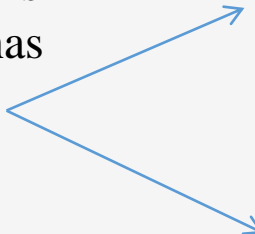
There are two types of constructors in Java: default constructor and parameterized constructor.

## **2. parameterized constructor**

- ✓ is a constructor that has one or more parameters and some explicit code.
- ✓ It is defined by the programmer to initialize the object attributes with specific values.
- ✓ The parameterized constructor can also call another constructor of the same class using the `this` keyword, or call a constructor of the superclass using the `super` keyword.

# Constructor

The syntax of a constructor in Java is similar to a method, except that it has the same name as the class and no return type.



```
// A class named Person
public class Person {
    // A field named name
    private String name;

    // A default constructor
    public Person() {
        // Assign a default value to name
        this.name = "Unknown";
    }

    // A parameterized constructor
    public Person(String name) {
        // Assign the parameter value to name
        this.name = name;
    }

    // A method to get the name
    public String getName() {
        return this.name;
    }
}
```

# Constructor

To create an object of a class using a constructor, we use the new keyword followed by the class name and parentheses. If the constructor has parameters, we pass the arguments inside the parentheses. For example:

```
// Create an object of Person using the default constructor
Person p1 = new Person();

// Create an object of Person using the parameterized constructor
Person p2 = new Person("Alice");

// Print the names of p1 and p2
System.out.println(p1.getName()); // Unknown
System.out.println(p2.getName()); // Alice
```

# Constructor vs method

Constructors are different from methods in Java in several ways:

- ✓ Constructors are called only once when an object is created, while methods can be called multiple times on the same or different objects.
- ✓ Constructors do not have a return type, while methods must have a return type or void.
- ✓ Constructors must have the same name as the class, while methods can have any valid identifier as their name.



# Constructor vs method

Constructors are different from methods in Java in several ways:

- ✓ Constructors cannot be inherited, overridden, or abstract, while methods can be.
- ✓ Constructors can only be accessed by using the new keyword, while methods can be accessed by using the object reference or the class name (for static methods).

# Constructor vs method

Example :

```
public class Car {  
    private String model;  
    private String color;  
    private String brand;  
    public Car(String model, String color, String brand) {  
        this.model = model;  
        this.color = color;  
        this.brand = brand;  
    }  
    public void start() {  
        System.out.println("The car is starting.");  
    }  
    public void accelerate() {  
        System.out.println("The car is accelerating.");  
    }  
    public void stop() {  
        System.out.println("The car is stopping.");  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        // create three car objects with different attributes  
        Car car1 = new Car("Model 3", "Red", "Tesla");  
        Car car2 = new Car("Camry", "White", "Toyota");  
        Car car3 = new Car("Mustang", "Black", "Ford");  
  
        // call the methods on each car object  
        car1.start();  
        car1.accelerate();  
        car1.stop();  
  
        car2.start();  
        car2.accelerate();  
        car2.stop();  
  
        car3.start();  
        car3.accelerate();  
        car3.stop();  
    }  
}
```

# Classes and objects

class

Car

data  
member

model  
color  
brand

method

start()  
accelerate()  
stop()



mode - Model 3  
color - Red  
brand - Tesla



mode - Camry  
color - White  
brand - Toyota



mode - Mustang  
color - Black  
brand - Ford

# Static members

- **Static members** in Java are those members of a class that belong to the class itself and not to any instance of the class.
- Static members are declared with the **static** keyword before their names. Static members can be fields, methods, blocks, or nested classes.

# Static members

**1. Static fields** are variables that have only one copy shared among all instances of the class.

- They are also known as **class variables**.
- Static fields are initialized when the class is loaded and can be accessed directly using the class name without creating an object.

# Static members

In this example, the PI field is a static field that belongs to the Math class and has a constant value of 3.14159.

The field can be accessed from any other class using the Math.PI.

```
class Math {  
    // declare a static field  
    public static final double PI = 3.14159;  
}  
  
class Main {  
    public static void main(String[] args) {  
        // access the static field without creating an object  
        System.out.println("The value of PI is: " + Math.PI);  
    }  
}
```

# Static members

2. **Static methods** are methods that can be invoked without creating an object of the class.
- They are also known as **class methods**.
  - Static methods can only access static fields and other static methods.
  - They cannot use the `this` or `super` keywords.
  - To call a static method, we can use the class name followed by a dot (.) and the method name.

# Static members

In this example, the square method is a static method that belongs to the Math class and takes one parameter x and returns its square.

The method can be called from any other class using the Math.square syntax

```
class Math {  
    // declare a static method  
    public static int square(int x) {  
        // code to be executed  
        return x * x;  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        // call the static method without creating an object  
        int y = Math.square(5);  
        System.out.println("The square of 5 is: " + y);  
    }  
}
```



# Static members

3. **Static blocks** are blocks of code that are executed only once when the class is loaded.
- They are used to initialize static fields or perform any other one-time operations.
  - Static blocks are declared with the **static** keyword before the curly braces { }.

# Static members

In this example, the PI field is initialized in a static block that is executed only once when the Math class is loaded. The block also prints a message on the screen

```
class Math {  
    // declare a static field  
    public static final double PI;  
  
    // declare a static block  
    static {  
        // code to be executed  
        PI = 3.14159;  
        System.out.println("Static block executed");  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        // access the static field without creating an object  
        System.out.println("The value of PI is: " + Math.PI);  
    }  
}
```

# Static members

- 4. **Static nested classes** are classes that are defined inside another class with the static keyword.
  - They are also known as **static inner classes**.
  - Static nested classes can access only static members of the outer class and do not have a reference to an instance of the outer class.
  - To create an object of a static nested class, we can use the outer class name followed by a dot (.) and the nested class name.

# Static members

In this example, the Inner class is a static nested class that belongs to the Outer class and has a method display.

The object of the nested class is created using the Outer.Inner syntax and the method is called using the obj.display

```
class Outer {  
    // declare a static nested class  
    public static class Inner {  
        // code to be executed  
        public void display() {  
            System.out.println("This is a static nested class");  
        }  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        // create an object of the static nested class  
        Outer.Inner obj = new Outer.Inner();  
        // call the method of the nested class  
        obj.display();  
    }  
}
```

# THE END



**Adama Science and Technology University**  
School of Electrical Engineering And Computing

Prepared by : **Amir Ibrahim**  
2023 G.C