Chapter 1. Basics of OOP

1. Introduction

Object-Oriented Programming (OOP) is a programming paradigm that allows structuring programs in a more natural and modular way by representing real-world entities as objects. These objects interact with each other to perform specific tasks. Unlike procedural programming, which focuses on the sequence of instructions, OOP focuses on managing and interacting with objects.

One of the main advantages of OOP is that it allows for better code organization, making it easier to maintain, extend, and reuse. This chapter explores the fundamental concepts of OOP and how they have evolved over time to meet the growing needs of developers.

2. Fundamental Concepts of OOP

a. A Brief History of OOP

OOP emerged in the 1960s with languages like Simula, which was designed for simulations and introduced concepts such as objects and classes. However, OOP truly took off with languages like Smalltalk in the 1970s and later with the advent of C++ in the 1980s, which integrated OOP within a structured programming language. Today, popular languages like Python, Java, and C# are widely used for object-oriented programming, and OOP principles are at the core of most modern applications.

b. Procedural Programming vs Object-Oriented Programming

Procedural programming is based on executing sequences of instructions (or functions) to process data. Data is often manipulated through global and local variables, which can make state management and code maintenance complex as the application grows.

In contrast, object-oriented programming focuses on creating objects that contain both data and methods to manipulate that data. This approach is more intuitive as it better reflects the reality of the world, where objects interact with one another. The main advantage of OOP over procedural programming is a clearer organization of code and the ability to model complex problems more naturally.

Classical programming, as studied through languages like C and Pascal, defines a program as a set of data manipulated by procedures and functions. In this paradigm:

- ✓ Data constitutes the passive part of the program.
- ✓ Procedures and functions constitute the active part.

Programming in this case involves:

- ✓ Defining a set of variables (e.g., structures, arrays).
- ✓ Writing procedures to manipulate these variables without explicitly associating them.

Executing a program is then reduced to calling these procedures in a sequence defined by the order of instructions and providing them with the necessary data to perform their tasks.

In this approach, data and procedures are treated independently of each other, without considering the close relationships that unite them.

The following questions arise in this case:

i. Is this separation (data, procedures) useful?
ii. Why prioritize procedures over data (**What do we want to do?**)?
iii. Why not consider programs primarily as sets of computer objects characterized by the operations they can perform?

Object-oriented languages were created to address these questions. They are based on understanding a single category of computational entities: objects.

An object incorporates both static and dynamic aspects within the same concept. With objects, data becomes the focus. The first question to answer is: "**What are we talking about?**"

A program is composed of a set of objects, each containing a part for procedures and a part for data. Objects interact by sending messages.

c. Code Reusability

One of the main benefits of OOP is code reusability through mechanisms like inheritance and encapsulation.

Inheritance: An object can inherit properties and methods from another class, allowing the creation of new classes without rewriting code. This makes it easier to extend and modify existing code without affecting already functioning parts.

Encapsulation: Objects hide their implementation details inside a class. This allows interaction with objects via a public interface while maintaining data integrity and protecting internal variables from improper manipulation. Code reusability becomes safer and easier to maintain.

d. Introduction to Modularity

Modularity in OOP refers to organizing code into small units called classes and objects. Each class represents a specific concept of the problem being solved, and each object is an instance of that class. By structuring the code this way, it becomes easier to maintain and update different parts of the application independently.

Modularity allows for:

- Isolating problems: Each class has a clearly defined responsibility, making it easier to manage bugs and changes.
- Easier testing: By testing classes independently, we can ensure that each part of the program works correctly without interfering with others.
- Reducing complexity: Each module can be designed and developed independently and then integrated into the system. This allows work to be done on specific parts without disrupting the entire application.

3. Objects and Classes

a. Notions of Object

In Object-Oriented Programming (OOP), objects are the fundamental building blocks. An object is an instance of a class and represents a specific entity in the real world or a specific concept in the software being developed. Objects combine state and behavior:

State: The state of an object is represented by its attributes (also called properties or fields). These attributes define the characteristics of an object, such as the color, size, or value of the object.

Behavior: The behavior of an object is defined by its methods (or functions). These methods perform operations or actions that can change the object's state or interact with other objects.

For example, consider a Car object. The state could include attributes such as color, model, and speed, while the behavior could involve methods like accelerate(), brake(), or turn(). Objects communicate with each other, modifying their state and interacting through methods.

An object is a software entity:

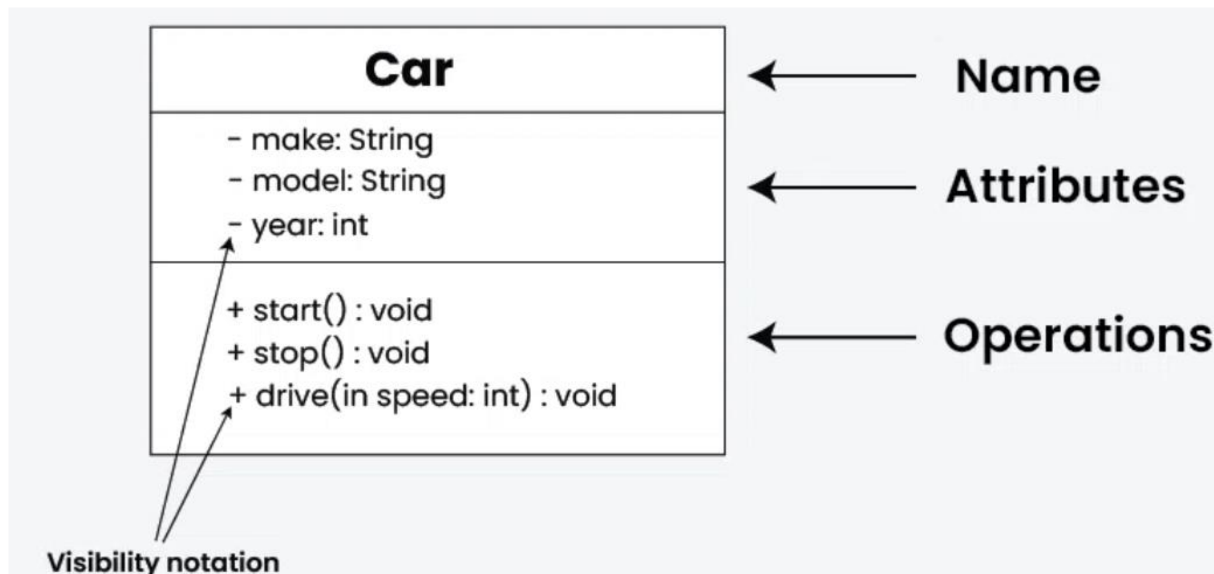
- Having an identity.
- Capable of maintaining a state, i.e., a set of information stored in internal variables.
- Responding to specific messages by triggering appropriate internal actions that change the object's state. These operations are called methods. They are functions associated with objects that define their behavior.

b. Notions of Class

A class is a new data type which is used to define objects. A class serves as a plan, or a template. It defines the common structure (attributes) and behavior (methods) that the objects of that class will have. A class is essentially a type for creating objects, and each object instantiated from a class is a member or instance of that class.

Classes allow you to define multiple objects with similar characteristics and behaviors. Once a class is defined, you can create as many objects (instances) as needed.

For example, the Car class mentioned earlier would have a structure that specifies attributes like color, model, and speed, as well as behaviors like accelerate (), brake (), and turn (). Creating an object of the Car class would involve instantiating it, giving it specific values for these attributes, and calling its methods to interact with it.



c. Attributes

Attributes (or fields) are the properties that define the state of an object. They store data about the object and are usually defined inside the class. An object's state is represented by these attributes.

Attributes are typically initialized when an object is created (**during instantiation**) and can be modified or accessed through methods. They are usually associated with an object's data but may also serve as internal variables that influence its behavior.

For example, a Person class could have attributes like name, age, and address. When you create a new Person object, you define values for these attributes, which represent the state of that specific person.

Attributes can also have different levels of access control, such as **public**, **protected**, **private and friendly (without modifier)** attributes, which regulate how the data can be accessed or modified from outside the class.

d. Notion of Message

In OOP, messages are used to communicate between objects. A message is typically a method call from one object to another, requesting the execution of a particular action. This concept is fundamental in OOP because it allows objects to collaborate and interact with one another.

When an object wants another object to perform a task, it sends a message in the form of a method invocation. The sender object will invoke a method (behavior) on the receiver object, possibly passing arguments and receiving a result.

- **Methods:** A method is a function or procedure linked to an object that is triggered upon receiving a specific message: the triggered method corresponds precisely to the received message. The list of methods defined within an object constitutes the object's interface for the user: these are the messages the object can understand when sent to it, triggering the corresponding methods.

- **Signature:** The signature of a method represents the specification of its name, the type of its arguments, and the type of data it returns.

For example, in a BankAccount class, an object may send a message to another object like account.deposit(amount) or account.withdraw(amount). The sending object is requesting the receiver object to execute a behavior based on the method it defines.

Messages are a key concept in OOP, and they highlight how objects interact, exchange data, and trigger actions within the system.

Conclusion : Classes, Objects, Instances

A class is a set of objects that share:

- ✓ The same methods
- ✓ The same types of attributes

An instance of a class is a specific object of the class that can activate the class methods and has specific values for its attributes. An object is defined as an instance of a class. The class represents for the object what the type represents for a variable. An object is characterized by:

- ✓ Its identity (OID): A unique and invariant value that characterizes the object.
- ✓ Its behavior: Defined through the methods applicable to this object, characterizing how it acts and reacts in its environment.
- ✓ Its state: Constituting the set of attribute values of this object.

A class is an abstract type that encapsulates data and processing. It acts as a mold that allows the creation of as many instances as desired. These instances are objects of the class to which messages can be sent, activating the corresponding methods.

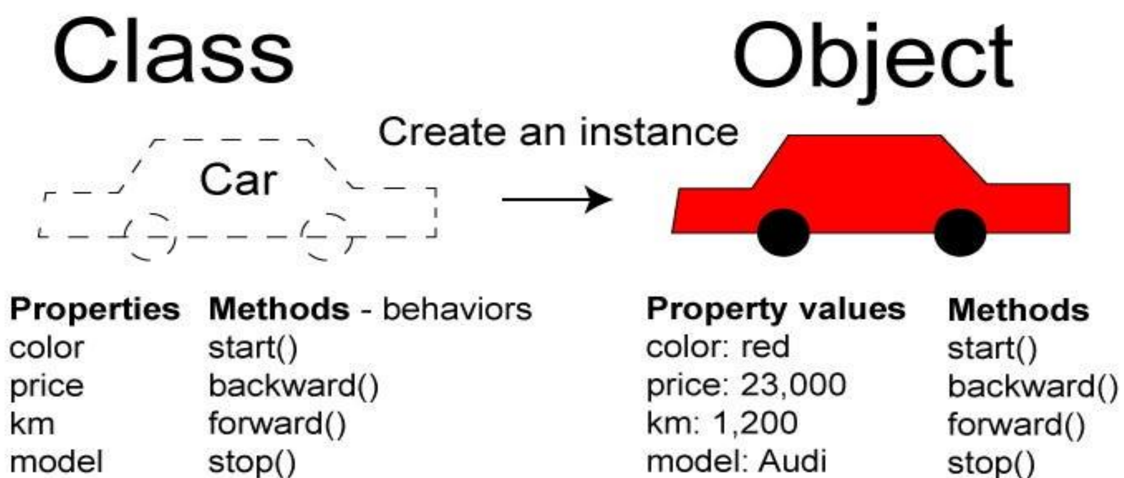


Figure 1: Classes, Objects, Instances

Class	Vs Object
A class is a model.	An object is an instance of the model.
A class is a software entity.	An object is a dynamically created data structure.
A class exists in the source code.	An object exists in memory during execution.

e. Problem Solving through Message Exchange

In OOP, problem-solving is often achieved by objects communicating with one another through the exchange of messages. Objects perform tasks by invoking methods on other objects, which encapsulates functionality and keeps the system modular. This approach allows for more efficient and organized problem-solving as it decouples the behavior of objects and provides clear interfaces for communication.

For example, consider a simple **Order Management System** where different objects such as **Customer**, **Order**, and **Payment interact**:

The **Customer** object sends a message to the **Order** object, requesting the creation of a new order.

The **Order** object sends a message to the **Payment object**, requesting payment processing.

If the payment is successful, the **Payment** object sends a message back to the **Order** object to confirm the order completion.

By structuring the problem as a series of message exchanges between objects, we can modularize the solution and make it easier to maintain, extend, and understand. Each object only needs to know how to handle its own responsibilities, not how the other objects are implemented.

f. Basic Concepts of OOP: OOP is based on the following key concepts:

- ✓ Encapsulation
- ✓ Abstraction
- ✓ Classes and objects
- ✓ Inheritance
- ✓ Polymorphism

f.1 Encapsulation

Encapsulation is the principle that an object contains its own attributes and methods. The implementation details of an object are hidden from other objects in the system. This is referred to as the encapsulation of data and behavior. Three access modes for an object's attributes are specified:

Public mode: Attributes are directly accessible by the object itself or by other objects. This is the least restrictive level of protection.

Private mode: Attributes are inaccessible from other objects; only the object's methods can access them. This is the highest level of protection.

Protected mode: This level of protection is closely associated with the concept of inheritance (covered later in the course).

f.2 Abstraction

Abstraction focuses on the essential characteristics of an object from the observer's point of view. Abstraction is a principle that involves ignoring certain aspects of a subject that are not important for the problem to focus on those that are.

f.3 Inheritance in Object-Oriented Programming (OOP)

Inheritance is one of the fundamental concepts of Object-Oriented Programming (OOP). It allows a class to inherit properties and behaviors (attributes and methods) from another class. This enables code reusability, modularity, and a hierarchical class structure, allowing new classes to build on existing ones.

In inheritance, the class that is inherited from is called the **base class** or **parent class**, and the class that inherits from it is called the **derived class** or **child class**. The child class inherits all the features (methods and attributes) of the parent class, but it can also have additional features or override the inherited ones.

f.4 Polymorphism in Object-Oriented Programming (OOP)

The term "polymorphism" is derived from Greek, meaning "many forms.". Polymorphism is one of the core principles of object-oriented programming. It allows objects of different classes to be treated as objects of a common parent class, enabling flexibility and reusability in code.

f.4.1 Key Concepts of Polymorphism

Definition: Polymorphism enables a single interface to be used with different underlying forms (data types, methods, or objects). It is implemented through:

- ✓ Method Overloading
- ✓ Method Overriding

4. Introduction to Java language

Java is a general-purpose programming language that synthesizes the main existing languages at the time of its creation in 1995 by Sun Microsystems. It supports object-oriented programming (similar to SmallTalk and, to a lesser extent, C++), modularity (ADA language), and uses a syntax very similar to that of the C language.

In addition to its object-oriented nature, Java has the advantage of being modular (allowing the writing of generic code, i.e., reusable across different applications), rigorous (most errors occur at compile-time, not runtime), and portable (the same compiled program can run on different environments).On the downside, Java applications tend to be slower at execution compared to applications written in languages like C.

4.1 Java Environment

Java is an interpreted language, which means that a compiled program is not directly executable by the operating system. Instead, it must be interpreted by another program known as the interpreter. Figure 6 illustrates this process.

2 Writes
3 Java file say HelloWorld.java
public class HelloWorld {
}
1 Developer
4 Is compiled By
5 Java Compiler javac
6 Generates
compiler
010101010101010101010
010101010101010101010
010101010101010101010
010101010101010101010
010101010101010101010
7 bytes codes say HelloWorld.class
8 Is Executed By
JVM
9
10 On
Hello World
11 Operating System

Figure 6. Java Programming Language Work

A Java programmer writes their source code, in the form of classes, in files with the .java extension. This source code is then compiled by the javac compiler into a language called bytecode and the result is saved in a file with a .class extension. The bytecode produced is not directly usable. It must be interpreted by the Java Virtual Machine (JVM), which then transforms the compiled code into machine code that the operating system can understand. This is why Java is a portable language: the bytecode remains the same regardless of the execution environment. In 2009, Sun Microsystems was acquired by Oracle Corporation, which now provides the Java SE (Standard Edition) development tools contained in the Java Development Kit (JDK). As of the writing of this guide, the latest stable version is JDK 1.7.

4.2 Compilation

Compilation is done using the javac command followed by one or more filenames containing the source code of Java classes. For example, **javac MyProg.java** compiles the MyProg class, whose source code is located in the MyProg.java file. Compilation often requires specifying certain parameters to proceed correctly, especially when the source code refers to other classes located in directories other than the directory of the compiled code. In this case, the -classpath option must be added, followed by the directories (separated by a '**;**' on Windows and a '**:**' on Unix) containing the referenced classes. For example: javac -classpath /prog/exos1:/cours MyProg.java will compile the MyProg.java file if it refers to other classes located in the /prog/exos1 and /cours directories. The result of this compilation is a file named MyProg.class, which contains the bytecode corresponding to the compiled source. This file is created by default in the directory where the compilation occurred. However, it is strongly recommended not to mix source code files with bytecode files. A destination directory where the MyProg.class file will be created can be specified with the -d option. For example: javac -d /prog/exos1 -classpath /cours MyProg.java.

4.3 Interpretation

The bytecode obtained through compilation can only be executed using the interpreter. Execution is done with the java command followed by the name of the class to execute (without the .class extension). As with compilation, it may be necessary to reference classes from other directories.

a. Types and Control Structures in Java

The C language served as the basis for the syntax of the Java language:

The end of an instruction is marked by a semicolon ";" : a = c + c;

- Comments (which are ignored by the compiler) are placed between the symbols /* and */, or begin with the // symbol and end at the end of the line:

int a; // this comment is on one line Or /* This comment spans 2 lines */ int a;

- Identifiers for variables or methods may contain characters from {a..z}, {A..Z}, $, _, as well as {0..9}, provided they are not the first character of the identifier. The identifier must also not be a reserved word in the language (such as int or for). Example: my_integer and ok4all are valid identifiers, but my-integer and 4all are not valid identifiers.

a.1 Data Types

a.1.1 Primitive Types

The following table 1 lists the primitive data types in Java. In addition to these primitive types, the term void is used to specify an empty return or the absence of parameters in a method. It is important to note that each primitive type has a corresponding class that encapsulates an attribute of that primitive type. For example, the Integer class encapsulates an attribute of type int, allowing operations and manipulations that would be impossible on a simple int variable.
Unlike the C language, Java is very strict with data typing. It is forbidden to assign a variable of one type to a variable of a different type unless the second variable is explicitly transformed. For example:

```
int a;
double b=2.5;
a=b;// compilation error : cannot convert from double to int
```

Correction should be:

```
int a;
double b=2.5;
a=(int)b;
```

Type	Class	Value	Scope	Default
boolean	Boolean	true or false	N/A	false
byte	Byte	Signed integer	{-128…128}	0
char	Character	character	{\uoooo…\uffff}	\uoooo
short	Short	Signed integer	{-32786..32767}	0
int	Integer	Signed integer	{-2147483648…2147483647}	0
long	Long	Signed integer	{-2^{31}…2^{31}-1}	0
float	Float	Signed real	{$-3{,}4028234^{38}$,…,$3{,}4028234^{38}$} {$-1{,}40239846^{-45}$,…,$1{,}40239846^{-45}$}	0.00
double	Double	Signed real	{$-1{,}797693134^{308}$,..,$1{,}797693134^{308}$} {$-4{,}94065645^{-324}$…$-4{,}94065645^{-324}$}	0.00

Table 1. Primitive data types in Java

a.1.2 Arrays and Matrices

A variable is declared as an array when square brackets are present either after its type or after its identifier. The following two syntaxes are accepted to declare an array of integers (although the first, which is not allowed in C, is more intuitive):

```
int[] myArray;
int myArray2[];
```

An array always has a fixed size, which must be specified before assigning values to its indices, as follows:

```
int[] myArray = new int[20];
```

Moreover, the size of this array is available in a length variable belonging to the array, and can be accessed via **myArray.length**. You can also create matrices or multi-dimensional arrays by multiplying the square brackets (e.g., int[][] myMatrix;). Like in C, elements of an array are accessed by specifying an index in square brackets (myArray[3] is the fourth integer of the array), and an array of size n stores its elements at indices ranging from 0 to n-1.

a.1.3 Strings

Strings are not considered a primitive type or an array in Java. A special class, called String, is used, which is provided in the java.lang package. The + operator can be used to concatenate two strings:

```
String s1 = "hello";
String s2 = "world";
String s3 = s1 + " " + s2;
// After these instructions, s3 equals "hello world"
```

The initialization of a string is written as:

```
String s = new String(); // for an empty string
String s2 = new String("helloworld"); // for a string with value "helloworld"
```

A set of methods from the java.lang.String class allows performing operations or tests on a string (refer to the String class documentation).

a.2 Operators

An expression is a combination of variables, constants and operators written according to some rules. An expression evaluates to a value that can be assigned to variables and can also be used wherever that value can be used.

Individual constant, variables, array elements function references can be joined together by various operators to form expressions. **Java** includes a large number of operators, which fall into several different categories. In this section we examine some of these categories in detail. Specifically, we will see how **arithmetic** operators; **unary** operators, r**elational** and **logical** operators, **assignment** operators and the **conditional** operator are used to form expressions.

The data items that operators act upon are called operands. Some operators require two operands, while others act upon only one operand. Most operators allow the individual operands to be expressions. A few operators permit only single variables as operands. Operators are used to compute and compare values, and test multiple conditions. They can be classified as:

1. Arithmetic operators
2. Assignment operators
3. Unary operators Notes
4. Comparison operators

5. Shift operators
6. Bit-wise operators
7. Logical operators
8. Conditional operators

1. Arithmetic Operators

There are five arithmetic operators in JAVA. They are

Operator	Function
+	addition
–	subtraction
*	multiplication
/	division
%	remainder after integer division

The % operator is sometimes referred to as the modulus operator.
There is no exponentiation operator in JAVA. However, there is a library function (pow) to carry out exponentiation. Alternatively, you can write your own function to compute exponential value.

2. Assignment Operators

Operator	Description	Example	Explanation
=	Assign the value of the right operand to the left	a = b	Assigns the value of b to a
+=	Adds the operands and assigns the result to the left operand	a +=b	Adds the of b to a The expression could also be written as a = a+b
-=	Subtracts the right operand from the left operand and stores the result in the left operand	a -=b	Subtracts b from a Equivalent to a = a-b
*=	Multiplies the left operand by the right operand and stores the result in the left operand	a*=b	Multiplies the values a and b and stores the result in a Equivalent to a = a*b
/=	Divides the left operand by the right operand and stores the result in the left operand	a/=b	Divides a by b and stores the result in a Equivalent to a = a/b
%=	Divides the left operand by the right operand and stores the remainder in the left operand	a%=b	Divides a by b and stores the remainder in a

3. Unary Operators

Operator	Description	Example	Explanation
++	Increases the value of the operand by one	a++	Equivalent a = a+1
--	Decreases the value of the operand by one	a--	Equivalent to a = a-1

The increment operator, ++, can be used in two ways - as a prefix, in which the operator precedes the variable.

++var;

In this form the value of the variable is first incremented and then used in the expression as
illustrated below:

var1=20;

var2 = ++var1;

This code is equivalent to the following set of codes:

var1=20;

var1 = var1+1;
var2 = var1;
In the end, both variables var1 and var2 store value 21.
The increment operator ++ can also be used as a postfix operator, in which the operator follows the variable.
var++;
In this case the value of the variable is used in the expression and then incremented as illustrated below:
var1 = 20;
var2 = var1++;
The equivalent of this code is:
var1 = 20;
var2=var1;
var1 = var1 + 1; // Could also have been written as var1 += 1;
In this case, variable var1 has the value 21 while var2 remains set to 20.
In a similar fashion, the decrement operator can also be used in both the prefix and postfix forms.
If the operator is to the left of the expression, the value of the expression is modified before the assignment. Conversely, when the operator is to the right of the expression, the Java statement, var2 = var1++; the original of var1 is assigned to var2. In the statement, var2 = ++var1; the incremented value of var1 is assigned to var2.

4. **Comparison Operators**

Comparison operators evaluate to true or false.

Operator	Description	Example	Explanation
==	Evaluates whether the operands are equal.	A==b	Returns true if the values are equal and false otherwise
!=	Evaluates whether the operands are not equal	a!=y	Returns true if the values are not equal and false otherwise
>	Evaluates whether the left operand is greater than the right operand	a>b	Returns true if a is greater than b and false
<	Evaluates whether the left operand is less than the right operand	a<b	Returns true if a is greater than or equal to b and false otherwise
>=	Evaluates whether the left operand is greater than or equal to the right operand	a>=b	Returns true if a is greater than or equal to b and false otherwise
<=	Evaluates whether the left operand is less than or equal to the right Operand	A<=b	Returns true if a is less than or equal to b and false otherwise.

5. **Shift Operators**

Data is stored internally in binary format (in the form of bits). A bit can have a value of one or zero. Eight bits form a byte. Shift operators work on individual bits in a byte. Using the shift operator involves moving the bit pattern left or right. You can use them only on integer data type and not on the char, bool, float, or double data types.

Operator	Description	Example
>>	Shifts bits to the right, filling sign bit at the left	a=10 >> 3
<<	Shifts bits to the left, filling zeros at the right	a=10 << 3

6. Bit-wise Operators

Operator	Description	Example	Explanation
& (AND)	Evaluates to a binary value after a bit-wise AND on the operands	a & b	AND results in a 1 if both the bits are 1, any other combination results in a 0
! (OR)	Evaluates to binary value after a two operands	a ! b	OR results in a 0 when both the bit-wise OR on the bits are 0, any other combination results in a 1.
^ (XOR)	Evaluates to a binary value after a bit-wise XOR on the two operands	a ^ b	XOR results in a 0 if both the bits are of the same value and 1 if the bits have different values.
~	Converts all 1 bits to 0s and (inversion)		

7. Logical Operators

Use logical operators to combine the results of Boolean expressions.

Operator	Description	Example	Explanation
&&	Evaluates to true, if both the conditions evaluate to true, false otherwise	a>6&&y<20	The result is true if condition 1 (a>6) and condition 2 (y<20) are both true. If one of them is false, the result is false.
| |	Evaluate to true, if at least one of the conditions evaluates to true and false if none of the conditions evaluate to true.	a>6 | | y < 20	The result is true if either condition1 (a>6) and condition2 (y<20) or both evaluate to true. If both the conditions are false, the result is false.

8. Conditional Operators

Operator	Description	Example	Explanation
(condition) va11, va12	Evaluates to va11 if the condition returns true and va12 if the condition returns false	a = (b>c) ? b:c	A is assigned the value in b, if b is greater than c, else a is assigned the value of c.

a.3 Control Structures

Control structures allow executing a block of instructions either multiple times (iterative instructions) or depending on the value of an expression (conditional or multiple-choice instructions). In all these cases, a block of instructions is:

Either a single instruction; Or a sequence of instructions starting with an opening brace { and ending with a closing brace }.

a.3.1 Conditional Statements

Syntax:

if(<condition>) <block1> [else <block2>]

The <condition> must return a boolean value. If it is true, <block1> is executed; otherwise, <block2> is executed.

Example:

```
if (a == b) {
    a = 50;
    b = 0;
} else {
    a = a - 1;
}
```

a.3.2 Iterative Statements

Iterative statements allow executing a block of instructions multiple times, until a given condition becomes false. The three types of iterative statements are as follows:

a.3.2.1 While...Do...

The execution of this statement follows these steps:

1. The condition (which must return a boolean value) is evaluated. If it is true, step 2 is executed; otherwise, step 4 is executed.
2. The block is executed.
3. Go back to step 1.
4. The loop ends, and the program continues execution by interpreting the instructions following the block.

Syntax:

while(<condition>) <block>

Example: while(a != b) a++;

a.3.2.2 Do...While...

The execution of this statement follows these steps:

1. The block is executed.
2. The condition (which must return a boolean value) is evaluated. If it is true, return to step 1; otherwise, proceed to step 3.
3. The loop ends, and the program continues execution by interpreting the instructions following the block.

Syntax:

do <block> **while**(<condition>);

Example: do a++ while (a != b);

a.3.2.3 For…

This loop consists of three parts: (i) initialization (local variable declarations within the loop are allowed here), (ii) a stop condition, and (iii) a set of instructions to execute after each iteration (each instruction is separated by a semi colon). The execution of this statement follows these steps:

1. Initializations are performed.
2. The condition (which must return a boolean value) is evaluated. If it is true, proceed to step 3; otherwise, go to step 6.

3. The main block is executed.
4. Instructions to execute after each iteration are executed.
5. Return to step 2.
6. The loop ends, and the program continues execution by interpreting the instructions following the main block.

Syntax:

for(<init>; <condition>; <post_iteration_instr>) <block>

```
for(int i = 0, j = 49; (i < 25) && (j >= 25); i++, j--) {
   if(tab[i] > tab[j]) {
      int temp = tab[j];
   }
}
```

a.3.3 Break and Continue Statements

The **break** statement is used to immediately exit a block of instructions (without processing the remaining instructions in the block). In the case of a loop, the **continue** statement can also be used, with the following difference:

break: Execution continues after the loop (as if the stop condition became true).

continue: Execution of the block stops, but not of the loop. A new iteration of the block starts if the stop condition is still true.

Example:

```
for(int i = 0, j = 0; i < 100; i++) {
   if(i > tab.length) {
      break;
   }
   if(tab[i] == null) {
      continue;
   }
   tab2[j] = tab[i];
   j++;
}
```

b. Classes and Instantiation

An object is an instance of a class and is referenced by a variable that holds a state (or value). To create an object, it is necessary to declare a variable of the class type to instantiate, and then call a constructor of that class. The following example illustrates the creation of an object of the Car class in Java:

```
Car  my_car;
my_car = new Car();
```

The use of parentheses in the initialization shows that a method is called for instantiation. This method is a constructor of the Car class. If the called constructor requires input parameters, they must be specified within these parentheses (just like in a normal method call). The instantiation of an object of the Rectangle class, using the constructor provided in the example below, could be written as:

```
Rectangle my_rectangle = new Rectangle(15, 5);
```

c. Access to Variables and Methods

To access a variable associated with an object, the object that contains it must be specified. The symbol "**.**" is used to separate the object identifier from the variable identifier. A copy of the length of a rectangle into an integer temp is written as:

int temp = my_rectangle.longueur;

The same syntax is used to call a method of an object. For example:

my_rectangle.deplace(10, -3);

For such a call to be possible, three conditions must be met:

1. The variable or method being called must exist.
2. A variable pointing to the targeted object must exist and be instantiated.
3. The object, within which the call is made, must have permission to access the method or variable .

To reference the "current" object (the one in which the code line is located), **Java** provides the keyword **this**. It does not need to be instantiated and is used like a variable pointing to the current object. The **this** keyword is also used to call a constructor of the current object. These two uses of **this** are illustrated in the following example:

```
class Square {
   private double sideLength; // Instance variable to store the side length of the square

   // Default constructor
   public Square() {
      this(1.0); // Calls the parameterized constructor with a default value
      System.out.println("Default constructor called.");
   }

   // Parameterized constructor
   public Square(double sideLength) {
      this.sideLength = sideLength; // Using 'this' to assign the parameter to the instance
variable
      System.out.println("Parameterized constructor called. Side length set to: " +
this.sideLength);
   }

   // Method to calculate the area
   public double calculateArea() {
      return this.sideLength * this.sideLength; // Using 'this' to refer to the instance variable
   }

   // Method to display details
   public void displayDetails() {
      System.out.println("Square with side length: " + this.sideLength);
      System.out.println("Area: " + this.calculateArea()); // Using 'this' to call another
method
   }
}
```

```
public class Main {
   public static void main(String[] args) {
      Square defaultSquare = new Square(); // Calls the default constructor
      defaultSquare.displayDetails();

      Square customSquare = new Square(4.5); // Calls the parameterized constructor
      customSquare.displayDetails();
   }
}}
```

d. References and Passing Parameters

In **Java**, when you pass a parameter to a method, it can either be passed by value or by reference. However, in Java, all parameters are passed by value. The difference between passing by value and passing by reference is important to understand when working with methods.

d.1 Passing Primitive Types (By Value)

When a primitive type is passed to a method, a copy of the value is created and passed. Therefore, any modifications to the parameter inside the method will not affect the original variable.

Example: public void modify(int x) {   x = 10;}

```
int a = 5;
modify(a); // After the method call, a remains 5, not 10.
```

In the example above, the value of a remains unchanged after the method call because it was passed by value.

d.2 Passing Objects (By Reference)

When an object is passed to a method, the reference to the object is passed, not the actual object itself. This means that the method can modify the object's fields. However, if the reference itself is reassigned to a new object, this change will not affect the original reference outside the method.

Example:

```
class Person {  String name;

public void modifyName(Person p) {  p.name = "John";}}

Person person = new Person();
person.name = "Alice";
modifyName(person);
// After the method call, person.name is "John".
```

In the example above, the name field of the person object is modified because the reference to the object is passed to the method. However, if the method had reassigned p to a new Person object, the original person object outside the method would remain unaffected.

d.3 Returning Objects from Methods

A method can also return an **object**. Since objects are passed by reference, returning an object from a method passes a reference to the calling code.

Example:

```
public Person createPerson() {   return new Person();}
```

```
Person person = createPerson();
// The person variable now holds a reference to the newly created Person object.
```
In this example, the createPerson() method returns a reference to a new Person object, which is assigned to the person variable.

d.4 Passing Arrays

Arrays in Java are also objects, so when you pass an array to a method, a reference to the array is passed. As a result, the method can modify the contents of the array, but reassigning the array itself will not affect the original array outside the method.

Example:

```
public void modifyArray(int[] arr) {     arr[0] = 10;}

int[] numbers = {1, 2, 3};
modifyArray(numbers);
// After the method call, numbers[0] is 10.
```
In this case, the method modifies the contents of the array, as arrays are passed by reference.

d.5 Conclusion

In summary, **Java** always passes parameters by value, but when dealing with objects or arrays, the value is the reference to the object, meaning the method can modify the object's fields. However, reassigning the reference itself inside the method does not affect the original reference outside the method.

e. Input/Output

e.1. Scanner / Input

Reading data from the keyboard via the console using the Scanner class.

```
public class Faculty {
            public static void main(String[] args) {
            Scanner sc = new Scanner(System.in);
                System.out.println("answer Yes or No:");
                int n1 = sc.nextInt();
                Double d1 = sc.nextDouble();
                float f1 = sc.nextFloat();
                String s1 = sc.next();
             System.out.println("integer = " + n1);
             System.out.println("double = " + d1);
             System.out.println("float = " + f1);
             System.out.println("String = " + s1);

             sc.close();}}
```

In this example, the Scanner class is used to read different types of data from the console, such as an integer (nextInt()), a double (nextDouble()), a float (nextFloat()), and a string (next()).

e.2. printf and println for Output

Displaying data on the console using printf and println methods.

```
import java.util.Calendar;
import java.util.Locale;
```

```
public static void main(String[] args) {
    // Output using printf
    double a = 5.6d;
    double b = 2d;
    String mul = "multiply ";
    String eq = "equal";

    System.out.printf(Locale.ENGLISH,
            "%3.2f X %3.2f = %6.4f \n", a, b, a * b);

    System.out.printf(Locale.FRENCH,
         "%3.2f %s %3.2f %s %6.4f \n", a, mul, b, eq, a * b);

    System.out.format(
       "Aujourd'hui %1$tA, %1$te %1$tB," +
        " il est: %1$tH h %1$tM min %1$tS \n",
            Calendar.getInstance());
      // System.out.flush();}
```

Output:

```
5.60 X 2.00 = 11.2000
5,60 multiply 2,00 equal 11,2000
Today Saturday , 10 October , it is : 15 h 31 min 01
```

In this example, the printf and format methods are used to display formatted data on the console. printf can format numbers, strings, and dates, while System.out.format allows custom formatting with Locale support.

f. Default Constructor and Other Constructors

Constructors are special methods in a class that are called when an object is created. They initialize the object's state. below an explanation of the default constructor and other types of constructors:

f.1Default Constructor

Definition:

- A constructor that takes no parameters.
- If you don't define any constructor in a class, Java automatically provides a default constructor with no arguments.

Characteristics:

- It initializes the object with default values (e.g., 0 for integers, null for objects, false for boolean).
- If you define any constructor, the default constructor is not provided automatically.

Example:

```
class Example {
   int number;
   String text;

   // No explicit constructor defined here, so Java provides a default one.
}
```

```
public class Main {
   public static void main(String[] args) {
      Example obj = new Example(); // Calls the default constructor
      System.out.println("Number: " + obj.number); // Outputs: 0
      System.out.println("Text: " + obj.text);    // Outputs: null
   }}
```

f.2 Parameterized Constructor

Definition:

A constructor that accepts arguments to initialize the object with specific values.

Purpose:

Useful for setting up initial values for the object's fields when it is created.

Example:

```
class Example {
   int number;
   String text;

   // Parameterized constructor
   Example(int number, String text) {
      this.number = number; // Assign parameter to the field
      this.text = text;
   }
}

public class Main {
   public static void main(String[] args) {
      Example obj = new Example(42, "Hello");
      System.out.println("Number: " + obj.number); // Outputs: 42
      System.out.println("Text: " + obj.text);    // Outputs: Hello
   }}
```

f.3 Custom Default Constructor

If you define a default constructor explicitly, it replaces the implicit default constructor provided by Java.

```
class Example {
   int number;
   String text;

   // Explicit default constructor
   Example() {
      number = 0;
      text = "Default Text";
   }
}

public class Main {
   public static void main(String[] args) {
      Example obj = new Example();
```

```
        System.out.println("Number: " + obj.number); // Outputs: 0
        System.out.println("Text: " + obj.text);    // Outputs: Default Text
    }
}
```

f.4 Constructor Overloading Example

```
class Example {
    int number;
    String text;

    // Default constructor
    Example() {
        number = 0;
        text = "Default";
    }

    // Parameterized constructor
    Example(int number, String text) {
        this.number = number;
        this.text = text;
    }
}

public class Main {
    public static void main(String[] args) {
        Example defaultObj = new Example(); // Calls default constructor
        Example paramObj = new Example(100, "Parameterized"); // Calls parameterized
constructor

        System.out.println("Default Obj -> Number: " + defaultObj.number + ", Text: " +
defaultObj.text);
        System.out.println("Param Obj -> Number: " + paramObj.number + ", Text: " +
paramObj.text);
    }}
```

Conclusion

1. If you define any constructor, Java does not provide a default constructor.
2. If you want both a parameterized constructor and a no-argument constructor, you must explicitly define both.
3. Constructors can be overloaded (multiple constructors with different parameter lists).
4. A constructor is special method
5. It has three characteristics:
    - ✓ it has no return type (not even void)
    - ✓ A constructor name must match the class name
    - ✓ It is not invoked in the same way.
6. The constructor is executed when the object is created
7. Useful for setting up initial values for the object's fields when it is created
8. Constructors are commonly overloaded in a class to provide multiple ways to initialize objects with different sets of parameters.

g. Destructors

In Java, destructors are not explicitly defined as in some other programming languages like C++. Java handles object destruction through garbage collection. The garbage collector automatically frees up memory when objects are no longer referenced or accessible.
However, Java provides a special method called finalize(), which was intended to be used for cleanup operations before an object is garbage collected. But it is not recommended to rely on finalize() for resource management, as it's unpredictable when it will be called.

Below an example of the finalize() method:

```
public class MyClass {
   @Override
   protected void finalize() throws Throwable {
      try {
         // Cleanup code, such as closing resources
         System.out.println("Object is being destroyed");
      } finally {
         super.finalize(); // Call the superclass's finalize method
      }
   }
}
```

However, modern Java best practices suggest using try-with-resources or explicit resource management (like closing files, streams, or database connections) instead of relying on finalize().

Example with try-with-resources:

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
   String line = br.readLine();
   // process the line
} catch (IOException e) {
   e.printStackTrace();
}
// The BufferedReader is automatically closed at the end of the try block
```

In this case, resources are automatically cleaned up when they are no longer needed.

Chapter end