Chapter 1: Basics of Object-Oriented Programming (OOP)

OOP is a powerful paradigm for software development that emphasizes the use of objects and classes to structure and organize code.

AK **by A.KHALFI**

What is Object-Oriented Programming?

- ✓ 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

Data

Objects contains data in the form of properties, representing their characteristics. Think of a car object with properties like color, model, and year.

Actions

Objects perform actions through methods, representing their behaviors. Our car object might have methods like drive, accelerate, and brake.

Advantages of OOP

Code Organization

Organized around objects, easier to manage.

Maintenance

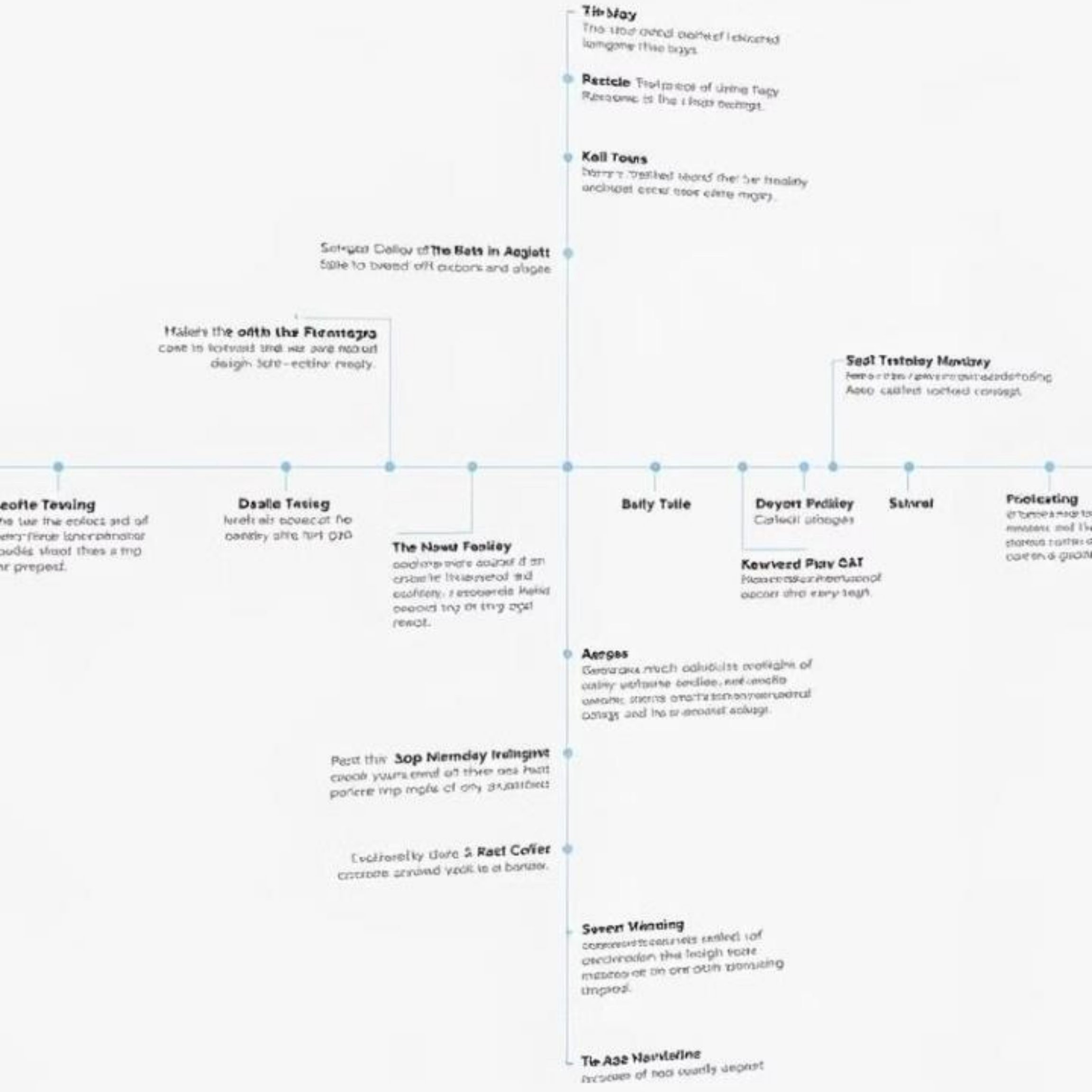
Modular design, easier updates and bug fixes.

Code Reusability

Objects and classes reused across programs. through mechanisms like inheritance and encapsulation.

Scalability

Easier to extend and add new features.



A Brief history of OOP

1

1960s: Simula

Introduced objects and classes.

2

1970s: Smalltalk

Popularized OOP.

3

1980s: C++

Integrated OOP with structured programming.

4

Today

Java, Python, C# are widely used.

Procedural vs Object-Oriented Programming

Procedural

Focuses on sequences of instructions.

Data and procedures are separate.

What do we want to do?

Object-Oriented

Focuses on objects containing data and methods.

Better reflects real-world interactions.

What are we talking about?

Classes and Objects

Classes are blueprints for objects, defining their properties and methods.

Object

An instance of a class, representing a real-world entity. Each object has its own unique set of properties and methods. The cookies you make from the cookie cutter.

Class

A template or blueprint for creating objects, defining their structure and behaviors. Think of it as a cookie cutter.

Objects and Classes

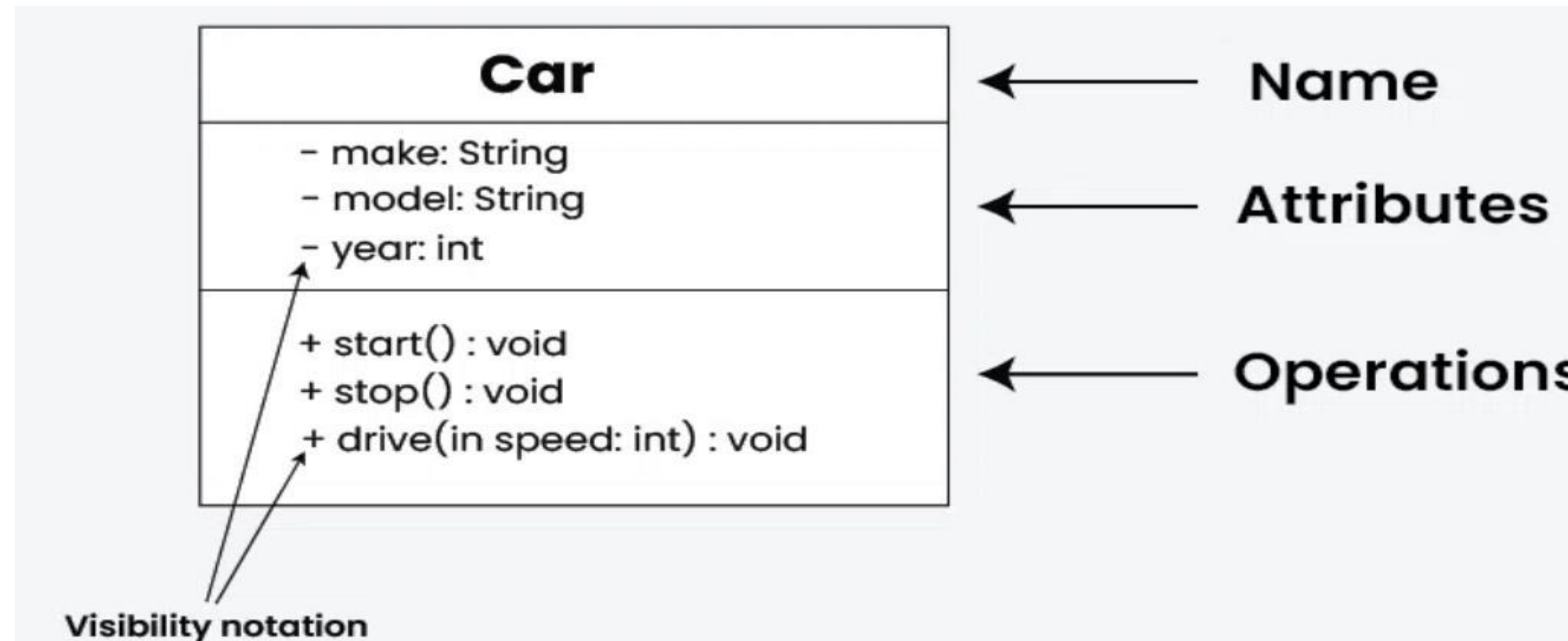
Objects

Identity : A unique and invariant value that characterizes the object.
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. Constituting the set of attribute values of this 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.

Classes

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.

Objects and Classes



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 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.

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.

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

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

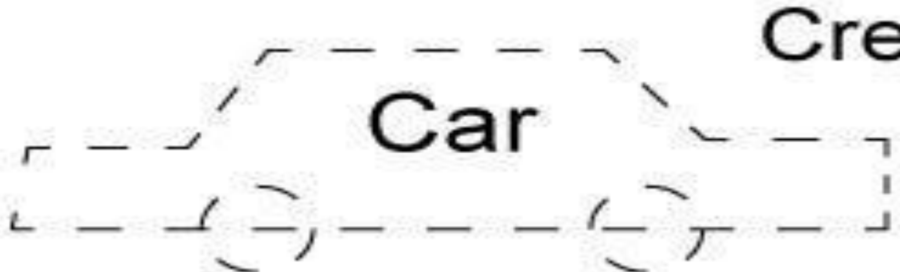
.

Classes, Objects, Instances

Class

Object

Create an instance

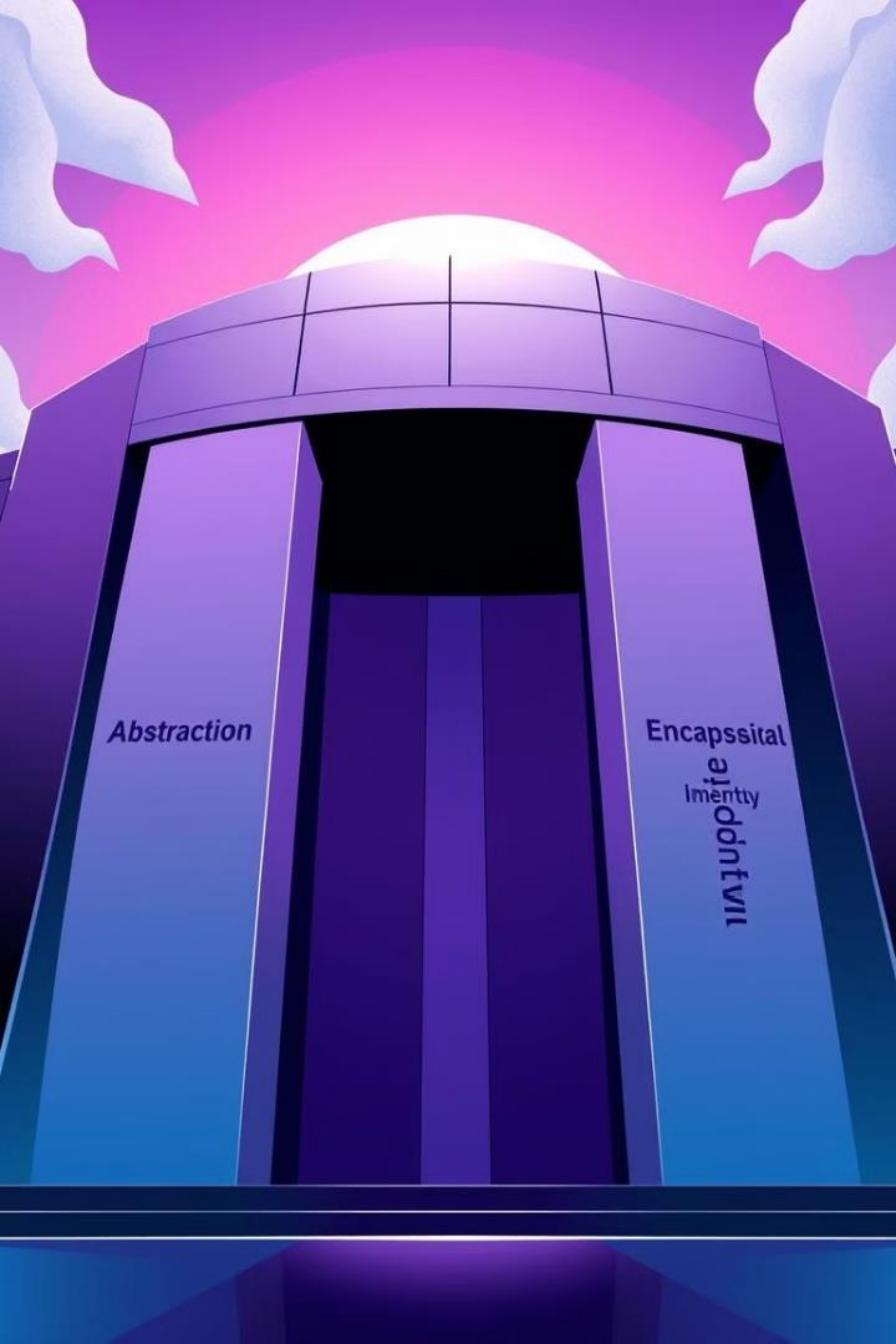


Properties	**Methods** - behaviors
color	start()
price	backward()
km	forward()
model	stop()

Property values	**Methods**
color: red	start()
price: 23,000	backward()
km: 1,200	forward()
model: Audi	stop()

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.



Basic Concepts of OOP

OOP follows five key principles that guide its structure and organization.

1 **Encapsulation**

Bundling data and methods together to protect data integrity.

2 **Abstraction**

Focusing on essential features and hiding unnecessary details.

3 **Classes and Objects**

Focusing on essential features and hiding unnecessary details.

4 **Inheritance**

Creating new classes based on existing ones, inheriting their properties and methods.

5 **Polymorphism**

The ability of objects to take on multiple forms and behave differently in different contexts

Encapsulation: Protecting Data

Encapsulation bundles data and methods together, protecting data from unauthorized access.

Data Hiding

Data is made private, accessible only through methods within the class.

Method Access

Methods are responsible for modifying and retrieving data, ensuring data integrity.

Modular Design

Encapsulation promotes modularity by making classes self-contained and easier to maintain.

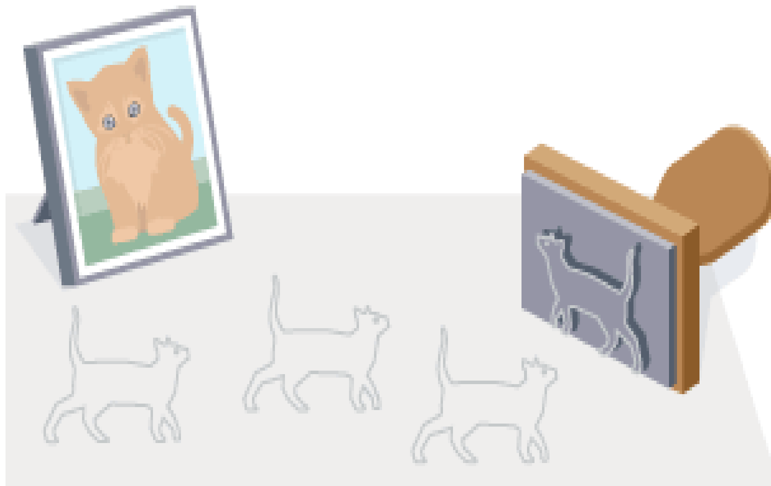
Abstraction

Abstraction is a fundamental OOP principle that focuses on **hiding complex implementation details** while **exposing only essential features** to the user.

It's like driving a car:

you use the steering wheel, pedals, and gearshift without needing to understand the internal combustion engine, transmission system, or electronic control units.

Abstraction



Core Idea:

- **Show what an object does** (interface/functionality)
- **Hide how it does it** (implementation details)
- Create a **simplified model** of reality

Abstraction

Real-World Examples:

- **Coffee Machine Abstraction :**
  - ✓What user sees: brewCoffee(), addWater()
  - ✓What's hidden: Water heating mechanism, pressure systems, bean grinding
- **Database Connection Abstraction:**
  - ✓What programmer uses: connect(), query(), disconnect()
  - ✓What's hidden: TCP/IP handshake, authentication, connection pooling

Abstraction

Benefits of Abstraction:

Reduces Complexity – Users work with simple interfaces

- Increases Reusability – Abstract components can be reused
- Enhances Security – Internal data protected from direct access
- Facilitates Maintenance – Implementation changes don't affect users
- Improves Modularity – Clear separation between interface and implementation

Abstraction vs. Encapsulation:

Abstraction: Hides complexity at design level (what to show/hide)

Encapsulation: Bundles data with methods and restricts access (how to hide)

Abstraction

Practical Use Cases:

- **Creating frameworks and libraries**
- **Designing APIs**
- **Building plug-in architectures**
- **Implementing layered architectures (MVC, etc.)**

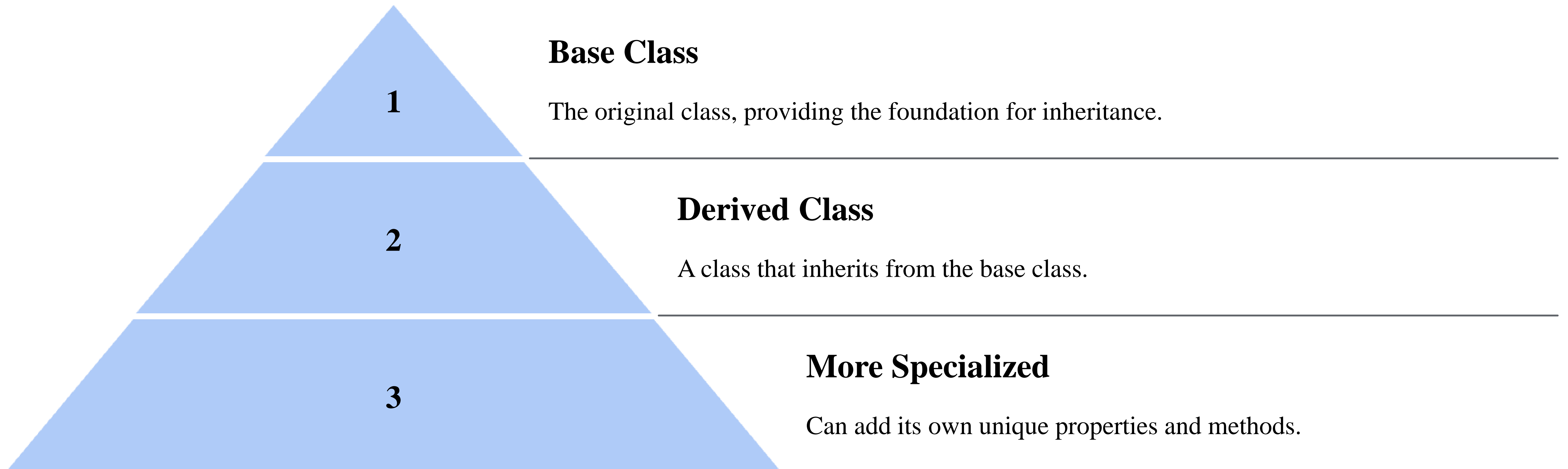
Abstraction

Practical Use Cases:

- Creating frameworks and libraries
- Designing APIs
- Building plug-in architectures
- Implementing layered architectures (MVC, etc.)

Inheritance: Building on Existing Code

Inheritance allows creating new classes that inherit properties and methods from existing classes.



Base Class

The original class, providing the foundation for inheritance.

Derived Class

A class that inherits from the base class.

More Specialized

Can add its own unique properties and methods.

Polymorphism: Objects in Multiple Forms

Polymorphism allows objects to behave differently in different situations.

1

Method Overriding

Derived classes can provide their own implementations of methods inherited from base classes.

2

Dynamic Binding

The specific method called is determined at runtime based on the object's type.

OOP in Action: Real-World Applications

OOP powers diverse applications, including websites, games, and desktop software.

Web Development

OOP builds dynamic, interactive web apps.

Game Development

OOP creates complex game elements and interactions.

Desktop Applications

OOP builds user-friendly interfaces and manages data efficiently.

4. Introduction to Java language

Java is a general-purpose programming language created in 1995 by Sun Microsystems. It combines features from languages like SmallTalk (object-oriented programming), ADA (modularity), and C (syntax). Java is object-oriented, modular (enabling reusable code), rigorous (errors are mostly caught at compile-time), and portable (compiled programs run on different environments). However, Java applications are generally slower in execution compared to languages like C.

Java is an interpreted language, meaning its compiled programs require an interpreter to run rather than being executed directly by the operating system.

Java Programming Language Work

1 Developer
2 Writes
3 Java file say HelloWorld.java

```
public class HelloWorld {

}
```

4 Is compiled By
5 Java Compiler javac
6 Generates

compiler

7 bytes codes say HelloWorld.class

0101010101010101010101010
0101010101010101010101010
0101010101010101010101010
0101010101010101010101010
0101010101010101010101010

8 Is Executed By
9 JVM
10 On
11 Operating System

Hello World

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.

Java source code (.java) is compiled into bytecode (.class) by javac and interpreted by the JVM, ensuring portability. Oracle provides Java SE tools in the JDK.

2-Compilation

Java programs are compiled using javac, producing a .class file. The -classpath option specifies dependencies, and -d sets the output directory. Keeping source and bytecode files separate is recommended.

3 Interpretation

Bytecode is executed using the java command, specifying the class name without .class. External classes may require the -classpath option.

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

ava has primitive data types, with void indicating no return or parameters in methods. Each type has a corresponding wrapper class (e.g., Integer for int). Java enforces strict type safety, requiring explicit conversion between different types.

```
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;
```

Primitive data types in Java

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

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 combines variables, constants, and operators to produce a value. Java provides various operators, including arithmetic, assignment, unary, comparison operators, Bit-wise operators, Logical operators and **Conditional operators. Operators** act on operands, which can be single variables or expressions, and are used for computations, comparisons, and condition testing.

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	a +=b	Adds the of b to a The expression could also be left operand 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 stores the result in the left operand	a*=b	Multiplies the values a and b by the right operand and and stores the result in a Equivalent to a = a*b
/=	Divides the left operand stores the result in the left operand	a/=b	Divides a by b and stores the by the right operand and result in a Equivalent to a = a/b
*=	Divides the left operand stores the remainder in the left operand	a%=b	Divides a by b and stores the by the right operand and 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) and -- (decrement) operators can be used as prefixes (++var) or postfixes (var++).
Prefix (++var): The variable is incremented first, then used in the expression.

var1 = 20;var2 = ++var1; // var1 becomes 21, var2 is assigned 21

Postfix (var++): The variable is used first, then incremented.var1 = 20;var2 = var1++; // var2 is assigned 20, then var1

becomes 21The same logic applies to the decrement (--) operator.

**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

Shift operators move bits left or right within a byte, operating only on integer data types (not on char, bool, float, or double). Data is stored in binary format, with each byte consisting of 8 bits.

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)		Example given below. all 0 bits to 1s

7.Logical Operators

Use logical operators to combine the results of Boolean expressions.

<table>
<tr><th>Operator</th><th>Description</th><th>Example</th><th>Explanation</th></tr>
<tr><td>&&</td><td>Evaluates to true, if both the conditions evaluate to true, false otherwise</td><td>a>6&&y<20</td><td>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.</td></tr>
<tr><td>||</td><td>Evaluate to true, if at least one of the conditions evaluates to true and false if none of the conditions evaluate to true.</td><td>a>6 || y < 20</td><td>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.</td></tr>
</table>

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-Control Structures

Control structures allow executing instructions repeatedly (iterative) or based on an expression's value (conditional/multiple-choice). These instructions are either a single statement or a sequence enclosed in braces { }.

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...

Syntax: **while(<condition>) <block>** **Example:** while(a != b) a++;

a.3.2.2 Do...While...

Syntax: **do <block> while(<condition>);** **Example:** do a++ while (a != b);

a.3.2.3 For…

Syntax: **for(<init>; <condition>; <post_iteration_instr>) <block>**

Example : 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.

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

If the called constructor requires input parameters, they must be specified within these parentheses (just like in a normal method call).

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

c. Access to Variables and Methods

To access a variable associated with an object,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:

b. Classes and Instantiation

Remarks

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. 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. Input/Output

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.

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.

1. The constructor is executed when the object is created
2. Useful for setting up initial values for the object's fields when it is created
3. 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
        }
    }
}
```

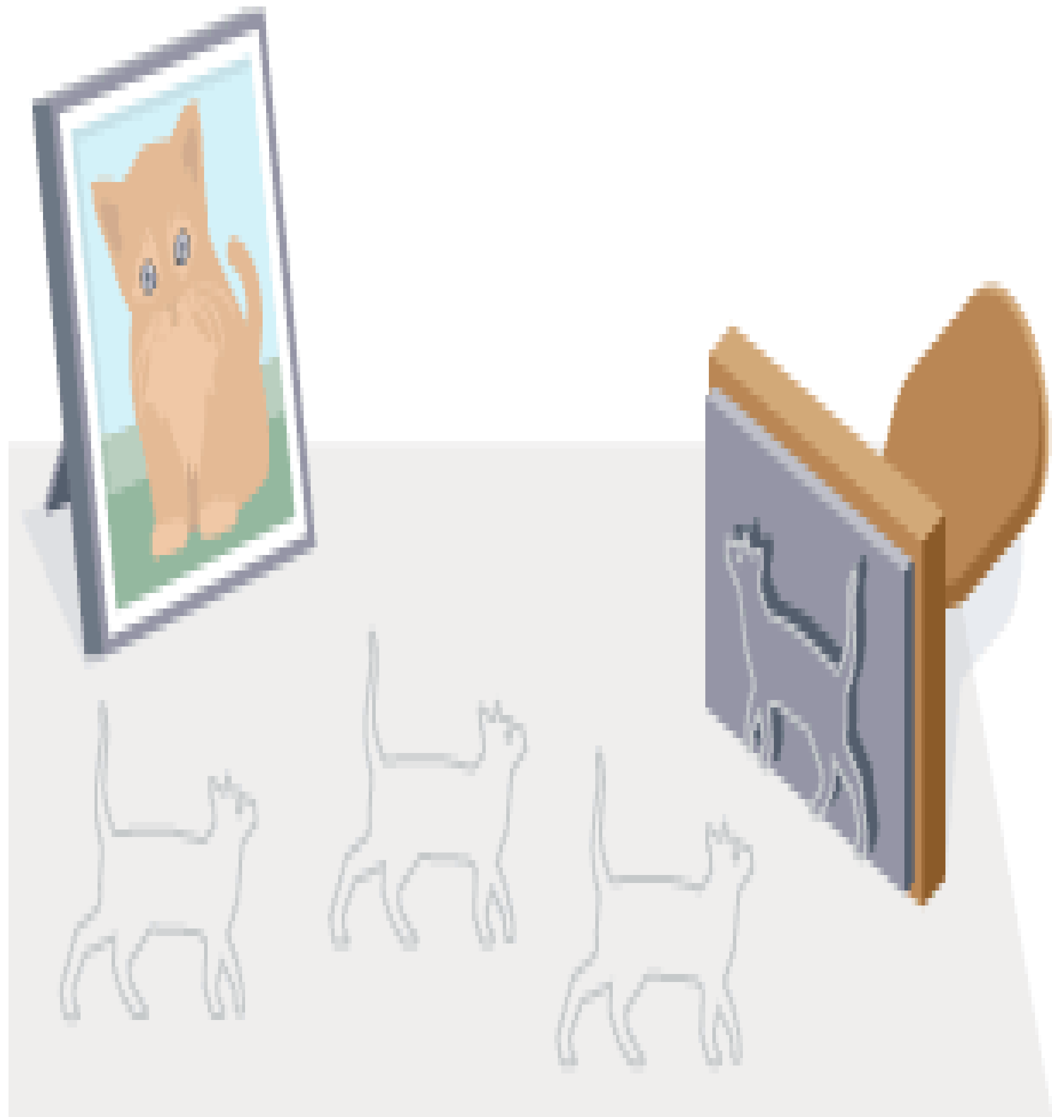
However, modern Java best practices suggest using try-with-resources or explicit resource management (like closing files, streams, or database connections) instead of relying on finalize().

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.

Thanks



Abstraction

What are specific details or characteristics?

We noted that all cats have general characteristics, which are common to all cats, eg eyes, a tail, a liking for fish and the ability to make meowing sounds. In addition, each cat has specific characteristics, such long tail, green eyes, a love of salmon, and a loud meow. These details are known as specifics.

In order to draw a basic cat, we do need to know that it has a tail and eyes. These characteristics are relevant. We don't need to know what sound a cat makes or that it likes fish. These characteristics are irrelevant and can be filtered out. We do need to know that a cat has a tail and eyes, but we don't need to know what size and colour these are. These specifics can be filtered out.

From the general characteristics we have (tail, eyes) we can build a basic idea of a cat, ie what a cat basically looks like. Once we know what a cat looks like we can describe how to draw a basic cat.