UNIVERSITY OF TISSEMSILT
FACULTY OF SCIENCE & TECHNOLOGY
DEPARTEMENT OF MATH AND COMPUTER SCIENCE

OBJECT-ORIENTED PROGRAMMING
Introduction to Java

23 février 2024

Lecturer

Dr. HAMDANI M

Speciality : Computer Science (ISIL)
Semester : S4

Plan

- Object-Oriented Programming
- Java
- Control Structures
- Array in Java

Introduction

Object-oriented programing (OOP) is a programming paradigm that structures code around the concept of objects.

Object-oriented programs are often easier to understand, correct and modify.

Structured Programming

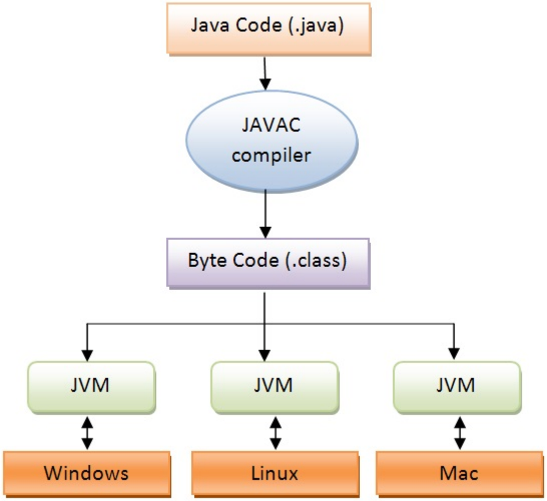
Wirth's equation :

Programs = Algorithms + Data Structures

The choice of algorithms and the use of suitable data structures are the fundamental building blocks for writing software.

Write once, run anywhere

- Develloped by Sun Microsystems in 1991 (*James Gosling*).
- A key goal of Java is to be able to write programs that will run on a great variety of computer systems and computer-controlled devices.



Java Interpreter

IDE

There are many popular Java IDEs, including :

- Eclipse (www.eclipse.org)
- NetBeans(www.netbeans.org)
- IntelliJ IDEA (www.jetbrains.com)

First Program in Java

```java
public class Welcome
{
   /* main method begins execution of Java application
         */
   public static void main(String[] args)
   {
     System.out.println("Hello World !");
   } // end method main
} // end class Welcome
```

***System.out.printf** :(f means "formatted") displays formatted data*

Input values

```
import java.util.Scanner;
public class Demo {
  public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);
    System.out.print("Enter any number: ");

    int num = scan.nextInt();  // reads the number
    scan.close();  // Closing Scanner after the use
    System.out.println("The number entered : " + num);
  }
}
```

string : nextLine()   float : nextFloat

Good Programming Practices

- ⋆ *Declare each variable in its own declaration. This format allows a descriptive comment to be inserted next to each variable being declared.*
- ⋆ *Choosing meaningful variable names helps a program to be self-documenting.*

Entering Text in a Dialog

```
import javax.swing.JOptionPane;

public class EnteringText_InDialog {

  public static void main(String[] args) {

  String name = JOptionPane.showInputDialog("Your
        name:");

   // display the message to welcome the user by name
  JOptionPane.showMessageDialog(null, " "+ name);

  }
}
```

Good Programming Practices

Format a Code

In order to format a selected region of code or an entire file :

- Click menu : Source > Format, or
- Eclipse : CTRL + SHIFT + F
- Netbeans : ALT + SHIFT + F

Data Types in Java

Category	**Data Types**	**Example**
Primitive Data Types	byte, short, int, long, float, double, char, boolean	`int age = 30;`
Reference Data Types	String, Classes, Arrays, Interfaces, Enums, custom objects	`String s = "John";`
Derived Data Types	Arrays, Classes (created using primitive/reference types)	`int[] a={1, 2, 3};`
User-Defined Data Types	Custom classes and interfaces	`class MyClss {...}`

Data Types in Java

Data Type	Size (bits)	Range	Example
byte	8	-128 to 127	`byte b = 42;`
short	16	-32,768 to 32,767	`short s = 1000;`
int	32	-2^{31} to $2^{31} - 1$	`int i = 123456;`
long	64	-2^{63} to $2^{63} - 1$	`long l = 9876543210L;`
float	32	IEEE 754 single-precision	`float f = 3.14f;`
double	64	IEEE 754 double-precision	`double d = 2.71828;`
char	16	0 to 65,535 (Unicode characters)	`char c = 'X';`
boolean	-	true or false	`boolean b = true;`

Java Primitive Data Types

String

```java
// Using a string literal
String str1 = "Hello, World!";

// Using the String constructor
String str2 = new String("Java");
```

Constants

A variable's value can not be changed after it has been assi-gned.

```
final double PI = 3.14159;
final int MAX_VALUE = 100;
```

Using the *static final* modifier combination :

```
public class Constants {
    public static final double PI = 3.14159;
    public static final int MAX_VALUE = 100;
  }
```

static : means that this variable belongs to the class itself, not to instances of the class.

Data Type Conversion

- Implicit Type Conversion (Widening) : automatic type conversion
- Explicit Type Conversion (Narrowing) : Also known as casting. converting a data value from one data type to another

Implicit conversion

A value of one data type is automatically and safely converted to another data type

1. Widening of Numeric Types
2. Promotion of Numeric Types
3. Boolean to Numeric Conversion
4. String to Numeric Conversion

Widening of Numeric Types

A smaller numeric data type is assigned to a larger numeric data type.

```java
int smallerValue = 42;
long largerValue = smallerValue;
// Implicit conversion from int to long
```

Promotion of Numeric Types

Different numeric data types are mixed in an expression, the Java compiler promotes them to a common, larger data type before performing the operation.

```
int num = 5;
double result = num + 3.5;
//Implicit conversion of int to double for addition
```

Boolean to Numeric Conversion

In some cases, boolean values can be implicitly converted to numeric values (1 for true and 0 for false).

```java
boolean flag = true;
int num = flag ? 1 : 0;
// Implicit conversion from boolean to int
```

Numeric to String Conversion

When you use the **+** operator to concatenate a String with a numeric value, the numeric value is implicitly converted to a String and then concatenated.

```java
String str = "The answer is: ";
int answer = 42;
String result = str + answer;
// Implicit conversion of int to String
```

Explicit Type Conversion

- Converting a value from one data type to another.
- Specifying the target data type explicitly using casting operators. Explicit type conversion is used when you need to convert a larger data type to a smalle

```java
double doubleValue = 1000.75;
int intValue = (int) doubleValue;
// Explicit casting from double to int
```

Conversion should be used judiciously, and programmers should be aware of the potential consequences and limitations when performing such conversions (Data Loss, Range Limitation, ...)

Parsing

Parsing refers to the process of extracting meaningful information or values from a textual representation, such as a string.

```
String strNumber = "42";
int number = Integer.parseInt(strNumber);

String strValue = "3.14159";
double value = Double.parseDouble(strValue);

String dateString = "2024-02-03";
SimpleDateFormat dateFormat = new
      SimpleDateFormat("yyyy-MM-dd");
Date date = dateFormat.parse(dateString);
```

if-else statement

The ***if-else*** statement is the most basic way to control program flow. The ***else*** is optional, so you can use ***if*** in two forms :

```
    if(Boolean-expression)
      statement
```

or

```
    if(Boolean-expression)
        statement
    else
      statement
```

Executes one block of code if a specified condition is true and another block of code if the condition is false

Nested ifs

- A ***nested if*** is an ***if statement*** that is the target of another ***if*** or ***else***
- An ***else statement*** always refers to the nearest ***if statement*** that is within the same block

```
if(i == 10) {
  if(j < 20) a = b;
        if(k > 100) c = d; // this if is
              else a = c; // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

if-else-if Ladder

A series of if statements followed by an ***else*** block, allowing for the evaluation of multiple conditions in sequence.

```
if(condition)
  statement;
  else if(condition)
      statement;
    else if(condition)
            statement;
        .
        .
        else
            statement;
```

Slection Statements Example

```
int num = 10;
int grade = 85;
if (num % 2 == 0)    // if-else statement
  System.out.println("Number is even");
else
  System.out.println("Number is odd");

      // else-if ladder
if (grade >= 90)
  System.out.println("Excellent!");
 else if (grade >= 80)
         System.out.println("Very good!");
      else if (grade >= 70)
              System.out.println("Good!");
            else
             System.out.println("Needs
                   improvement!");
```

switch statement

Multiway branch statement, execute one block of code from multiple options based on the value of an expression

```
switch (expression) {
  case value1: // code, if expression == value1
          break;
  case value2: // code, if expression ==
        value2
          break;
   // ... more cases
  default: // code to be executed if no match
        found
}
```

For versions of Java prior to JDK 7, expression must be of type byte, short, int, char, or an enumeration. Beginning with JDK 7, expression can also be of type String.

```
int day = 3;
String dayName;
switch (day) {
  case 1:  dayName = "Sunday";
           break;
  case 2: dayName = "Monday";
           break;
  case 3:  dayName = "Tuesday";
           break;
  case 4:  dayName = "Wednesday";
           break;
  case 5:  dayName = "Thursday";
           break;
  case 6:  dayName = "Friday";
         break;
  case 7:  dayName = "Saturday";
           break;
  default: dayName = "Invalid day";
}
```

Iteration Statements

- ***for*** loop : Executes a block of code a specified number of times.
- ***while*** loop : Executes a block of code as long as a specified condition is true.
- ***do-while*** loop : Executes a block of code at least once and then repeatedly executes the block as long as a specified condition is true.

"for-each" loop

```
public class ForEachLoopExample {

    public static void main(String[] args) {

        int[] numbers = {1, 2, 3, 4, 5};

        /* Using a for-each loop to print each element
              of the array*/
        for (int num : numbers) {
          System.out.println(num);
        }
    }
}
```

break statement

- Terminates the loop or switch statement and transfers control to the statement immediately following the loop or switch.
- continue statement : Skips the current iteration of a loop and proceeds to the next iteration.
- return statement : Exits the current method and optionally returns a value.

"break - example

Example in a for loop :

```
  for (int i = 0; i < 10; i++) {
    if (i == 5)
      break; // Terminates the loop when i is
    System.out.println(i);
  }
```

Example in a while loop :

```
int i = 0;
while (i < 10) {
  if (i == 5)
     break; // Terminates the loop when i is 5
  System.out.println(i);
  i++;
}
```

continue statement

continue

- Used within looping constructs (**for**, **while**, and **do-while** loops) to skip the current iteration of the loop and proceed to the next iteration.
- Unlike the break statement, which exits the loop entirely, continue merely skips the remaining code in the loop for the current iteration and then continues with the next iteration of the loop.

continue - example

Example in a for loop :

```
for (int i = 1; i <= 10; i++) {
  if (i == 5)
    continue; // Skip the rest of the loop body for i
          == 5
  System.out.println(i);
}
```

Example in a while Loop :

```
int i = 0;
while (i < 10) {
  i++; // Increment i at the beginning to avoid
          infinite loop
  if (i == 5) continue; // Skip printing 5
    System.out.println(i);
```

Array in Java

- In Java, an array is a container object that holds a fixed number of values of a single type.
- The length of an array is established when the array is created and cannot be changed after creation.
- Each item in an array is called an element, and each element is accessed by its numerical index, with the first element's index being **0**.

Creating Arrays

Creating Arrays :

```
int[]  myIntArray = new int[10]; // An array of 10
      integers;
String[] myStringArray = new String[5]; // An array of
      5 Strings
```

You can also initialize the array at the time of creation by enclosing the initial values in curly braces .

```
int[] myIntArray = {1, 2, 3, 4, 5};
String[] myStringArray = {"Hello", "World"};
```

Creating Arrays

Creating Arrays :

```
int[]  myIntArray = new int[10]; // An array of 10
      integers;
String[] myStringArray = new String[5]; // An array
      of 5 Strings
```

You can also initialize the array at the time of creation by enclo-sing the initial values in curly braces .

```
int[] myIntArray = {1, 2, 3, 4, 5};
String[] myStringArray = {"Hello", "World"};
```

Accessing Array Elements

Accessing Array Elements : Array indexes start at 0. So, the first element of an array is at index 0, the second is at index 1, and so on.

```
int firstElement = myIntArray[0]; // Access the first
      element
myIntArray[4] = 100; // Assign a value to the fifth
      element
```

Array Length : The length property of an array is used to find out the size of an array.

```
int arraySize = myIntArray.length;
```

Looping Through Arrays

You can loop through an array using a for loop or an enhanced for loop (also known as the "for-each" loop).

```java
// Using a for loop
for (int i = 0; i < myIntArray.length; i++) {
  System.out.println(myIntArray[i]);
}
```

```java
// Using an enhanced for loop
for (int element : myIntArray) {
  System.out.println(element);
}
```

Multidimensional Arrays

Java supports multidimensional arrays, which are arrays of arrays. The most common type is the two-dimensional array.

```
int[][] my2DArray = new int[10][20]; // A 2D array
      of size 10x20

my2DArray[0][0] = 1; // Assign a value to the
      first element

int[][] my2DArrayInitialized = {{1, 2}, {3, 4}};
      // Initialization
```

Array in Java

- Arrays have a fixed size and cannot grow or shrink once created.
- They can hold only one type of data.
- For more flexible operations like inserting, deleting, or re-sizing, consider using Java Collections Framework classes such as **ArrayList**.

```
public class BubbleSort {
  public static void bubbleSort(int[] arr) {
    int n = arr.length;
    boolean swapped;
    for (int i = 0; i < n - 1; i++) {
      swapped = false;
      for (int j = 0; j < n - i - 1; j++) {
        if (arr[j] > arr[j + 1]) {
          // swap arr[j] and arr[j+1]
          int temp = arr[j];
          arr[j] = arr[j + 1];
          arr[j + 1] = temp;
          swapped = true;
        }
      }
      if (!swapped)  // IF no two elements
        break;       //were swapped, then break
    }
  }
```

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

      int[] numbers = {64, 34, 25, 12, 22, 11, 90};

        bubbleSort(numbers);
      System.out.println("Sorted array: ");

      for (int number : numbers) {
       System.out.print(number + " ");
      }
    }
}
```

Questions ?

University of Tissemsilt
Faculty of Science & Technology
Departement of Math and Computer Science

Object-Oriented Programming
Encapsulation

3 mars 2024

Lecturer

Dr. HAMDANI M

Speciality : Computer Science (ISIL)
Semester : S4

Plan

Definition

- A class is a blueprint or template from which objects are created.
- It defines the properties (attributes/fields) and behaviors (methods/functions) that the objects created from the class will have.
- It defines a group of objects with similar characteristics (properties) and behaviors (methods).

Benefits of Using Classes

- **Encapsulation** : Classes encapsulate data and behavior, promoting modularity and code organization.
- **Reusability** : Classes can be reused in different parts of the program, reducing code duplication.
- **Abstraction** : Classes hide implementation details, allowing users to interact with objects at a higher level of abstraction.

Components of a Class

- **Fields (Attributes)** :Represent the properties or characteristics of an object (e.g., color, name, age).
- **Methods (Behaviors)** : Functions that define what an object can do. These methods can access and modify the fields of the class and perform other operations.
- **Constructors :** A special method used to initialize an object when it's created..

```java
public class Person {
  private String name;  // Fields representing
  private int age;      // Person data
  // Constructor to initialize the student
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  // Methods to access Person information
  public String getName() {
    return name;
  }
  public int getAge() {
    return age;
  }
  public void printInfo() {
    System.out.println("Name: " + name);
    System.out.println("Age: " + age);
  }
}
```

Constructors

- Have the same name as the class they belong to.
- Are used to initialize the state of an object.
- Do not specify a return type, not even ***void***.
- Can be overloaded (a class can have multiple constructors with different parameters).
- If no constructor is explicitly defined in a class, Java provides a default constructor automatically.
- The default constructor initializes the instance variables to their default values (e.g., ***0*** for numeric types, ***null*** for object references, ***false*** for boolean primitives)

Constructor - Example

```java
public class Person {
  private String name;
  private int age;
```

```java
  // Default constructor is created here
  public Person() {
    /* name and age will be initialized to their
    default values (null and 0, respectively) */
  }
```

```java
  // Another constructor with parameters
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
}
```

Some important points to note about **default constructors** :

- **Not created if other constructors exist** : If you define even one constructor explicitly, the compiler will not create a default constructor.
- **Can not be called explicitly** : You can not call the default constructor explicitly since it's not defined in your code.
- **Subclass implications** : if a subclass doesn't have any constructors explicitly defined, it will inherit the default constructor from its superclass.
  However, if the superclass doesn't have a no-argument (*default*) constructor and only has parameterized constructors, the subclass cannot have a default constructor.

Destructor

- Java does not have destructors.
- There is a mechanism called **"garbage collection"** that handles memory deallocation when objects are no longer needed
- Java Virtual Machine (JVM) automatically handles memory management and garbage collection.
- The ***finalize()*** method provides a mechanism for performing cleanup or finalization tasks before an object is garbage collected."
- ***finalize()*** is not recommended, due to its non-deterministic nature and limitations (utilize : *try-with-resources, close(), closeConnection(), AutoCloseable*,...)

What is Encapsulation ?

Combines data and methods into a class.

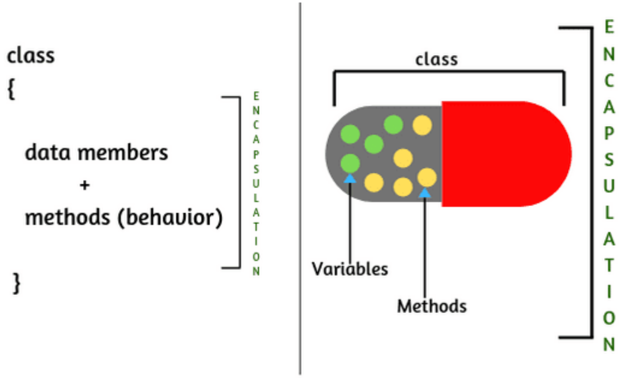


Fig: Encapsulation

Principles of Encapsulation

- Data Hiding : Encapsulation hides the internal state of objects from direct access by external code, preventing unauthorized access and manipulation.
- Access Control : Encapsulation allows you to control the visibility and accessibility of class members using access modifiers (public, private, protected, and default).
- Information Hiding : Encapsulation hides the implementation details of a class from its users, exposing only the necessary interfaces (public methods) for interacting with the class.

Benefits of Encapsulation

- Modularity : Encapsulation promotes modular design by encapsulating related data and behaviors within a single class .
- Data Integrity : Encapsulation helps maintain data integrity by providing controlled access to the internal state of objects, preventing invalid or inconsistent data states.
- Code Reusability : Encapsulation enables you to encapsulate reusable components (classes) that can be easily reused in different parts of the codebase.
- Security : Encapsulation enhances security by restricting access to sensitive data and preventing unauthorized modification of object state, protecting the integrity and confidentiality of the data.

Encapsulation in Java

- In Java, encapsulation is achieved through the use of access modifiers (public, private, protected, and default) to control the visibility and accessibility of class members.
- Getter and setter methods are used to provide controlled access to private attributes, allowing for safe and controlled manipulation of object state.

Access Modifiers

Access modifiers control method and data visibility

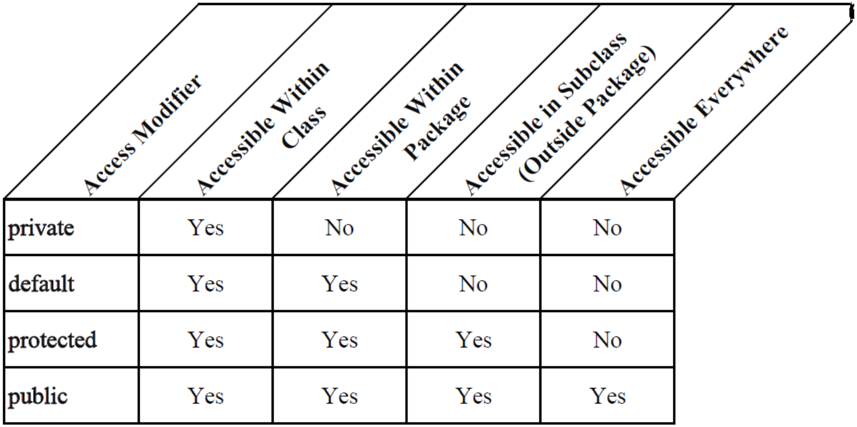
public : Accessible from anywhere in the program (like a master key)

private : Accessible only within the class itself (like a key for a specific room)

protected : Accessible within the class and its subclasses (like a key for a specific building)

default (no modifier) : Accessible only within the same package

Access Modifiers in Java



Access Modifier	Accessible Within Class	Accessible Within Package	Accessible in Subclass (Outside Package)	Accessible Everywhere
private	Yes	No	No	No
default	Yes	Yes	No	No
protected	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes

Java Access Modifiers and Their Accessibility

Best Practices

- Use the most restrictive access level that makes sense for a particular member. Start with ***private*** and only increase accessibility as needed.
- ***Fields*** are typically made ***private*** to enforce encapsulation.
- ***Constructors*** can be any access level, depending on whether you want to restrict instantiation of your class.
- ***Methods*** that are intended for use outside of the class should be ***public***, but internal utility methods should be ***private*** or ***default*** to limit their scope.
- ***Constant*** values (static final variables) are usually made public since they don't change and are meant to be accessible wherever needed.

Remember

By carefully choosing the appropriate access modifier for each class member, you can ensure that your Java classes expose a well-defined interface to the rest of your program, while keeping their internal implementation details hidden and protected.

Accessors (getters)

- An Accessor method is commonly known as a get method or simply a ***getter***.
- used to retrieve or access the value of an object's state (its fields or properties) from outside the class.
- Provide controlled access to internal data while maintaining encapsulation.

Getters - Accessiblity

Getter access in Java :

- **Public** (*Most Common Use*) : Widely accessible, but weakens encapsulation.
- **Protected** : Accessible within package and subclasses, good for inheritance.
- **Package-private** : Accessible within the same package, useful for internal collaboration.
- **Private** : Only accessible within the class, hides data but needs additional access methods.

Always use private fields

Getter - Example

```
public class Person {
    private String name;
    private int age;
    private boolean employed;

    public String getName() {
      return name;
    }
    public int getAge() {
      return age;
    }
    public boolean isEmployed() {
      return employed;
    }
}
```

Mutator (setters)

- Sets or updates the value (mutators) : ***setter*** is a method in java used to update or set the value of the data members or variables.
- The ***setter*** It takes a parameter and assigns it to the attribute..
- Importance : ensure data encapsulation, validate inputs, and enhance code maintainability.

Setters - Accessiblity (1)

Choose the access modifier based on context, sensitivity, and future needs.

- Public : Most common ; used when you want to allow external classes to modify the state of an object. Suitable for fields that can be safely changed by any class.
- Protected : Restricts the setting of a field to the defining class, its subclasses, and classes within the same package. Useful when you want to limit modifications to a more controlled group of classes, typically within a hierarchy.

Setters - Accessiblity (2)

- Default (Package-Private) : Allows only classes within the same package to modify the field. Good for when you're working with a set of closely related classes that need to interact more freely with each other but are not exposed to the outside.
- Private : Very rare, as setters are meant to modify the state of an object from outside. However, they can be used internally to encapsulate the setting logic within the class itself, not intended for use by any external or subclass.

Setter - Example

```java
public class Person {
        private String name;
        private int age;
      public void setName(String name)
      {
        this.name = name;
      }
      public void setAge(int age) {
        if (age >= 0) {
          this.age = age;
        } else {
          throw new IllegalArgumentException("Age cannot
                be negative");
        }
      }
}
```

Naming Getters and Setters

- Use the same prefix for all getters and setters : **get**, **set**, **is**, **has**
- Use meaningful names : The name should clearly communicate the purpose of the method. Avoid generic names like *getValue*, *setValue*
- Follow the Java Naming Conventions : Use ***camelCase*** for method names
- Keep it concise : While clarity is important, avoid overly verbose names
- Consider the mutability of the property : For boolean properties, is or has prefixes can be used for getters, while set is used for setters.
- For complex properties : Consider using descriptive names : *getEmployeeInformation()*

Examples : Naming Getters and Setters

Getter names :

- getFirstName()
- getLastName()
- getSalary()
- getEmployeeInformation()
- isManager()
- isEnabled()

Setter names :

- setFirstName(String firstName)
- setLastName(String lastName)
- setSalary(double salary)
- setManager(boolean manager)

Automatically Inserting Getters and Setters

in *Netbens/Eclipse*, you can automatically generate getters and setters for your Java class fields :

- Position your cursor within the class where you want to insert the getters and setters.
- Select **Source** > **Generate Getters and Setters....**
- In the dialog that appears, Select the fields you want to generate accessors for.
- Click OK to generate the methods.
- Or, select : **Refactor** > **Encapsulate Fields...**

What is "this" ?**

- ***this*** is a reference variable that points to the current object instance.
- It is implicitly available within all instance methods and constructors.
- You can use ***this*** to access instance variables, methods, and even call other constructors within the same class.
- can be used only inside a non-static method

Uses of "this"

- Accessing instance variables : Use this to differentiate between local variables and instance variables with the same name.
- Calling other methods : Use this to call other methods on the same object, promoting modularity and code reuse.
- Calling constructors : Use this to call other constructors from within a constructor, enabling initialization with different parameters.
- Passing as an argument : You can pass this as an argument to other methods, allowing them to interact with the current object.

Example : Accessing Instance Variables

```
public class Person {
  private String name;

  public void setName(String name) {
    this.name = name;
    // this.name refers to the instance variable
  }
}
```

this.name is used within the *setName* method to differentiate the local variable name from the instance variable name

Example : Calling Other Methods

```
public class Person {
  public void greet(Person other) {
    System.out.println("Hello " + other.name + "!");
  }
  public void sayHelloTo(Person other) {
    this.greet(other); // Call greet() on the current
          object
  }
}
```

the *sayHelloTo* method uses *this.greet* to call the *greet* method on the current object instance (this). *This* allows the *sayHelloTo* method to reuse the functionality of *greet* without code duplication.

Example : Calling Constructors

```
public class Person {
  private String name;
  public Person(String name) {
    this.name = name;
  }
  public Person() {
    this("Dennis Ritchie");/* this() calls the
          constructor with the String argument */
  }
}
```

the no-argument constructor (*Person()*) uses *this("Dennis Ritchie")* to call the constructor that takes a name argument. *This* allows you to create objects with different initial names using different constructors.

Example : Passing "this" as an Argument

```
public class Person {
  private String name;

  public void compare(Person other) {
    if (this.age > other.age) {
      System.out.println(this.name + " is older than " +
             other.name);
    } else {
      System.out.println(other.name + " is older than " +
             this.name);
    }
  }
}
```

In this example, the *compare* method takes another *Person* object as an argument and uses *this* to access the current object's *age* and *name*.

Static Variables

- Class variables, also known as static variables, are variables declared with the ***static*** keyword within a class.
- Belong to the class rather than instances of the class.
- Shared by all instances of the class, meaning only one copy exists (global variables).
- Accessed directly using the class name (e.g., *ClassName.variableNam*
- Can also be accessed through an object instance followed by the class name (e.g., *objectInstance.className.variableName*).
- Initialized only once at the start of the program's execution.

Accessing Class Variables

- Use the class name directly (e.g., *MyClass.count*).
- Access through an object instance (e.g., *object.MyClass.count*).
- Remember, accessing class variables through an object instance might not always reflect the latest value due to potential changes made by other objects.

Examples of Class Variables

Counter for Objects :

```
static int objectCount = 0; // to count the number of objects
```

Constants :

```
static final double PI = 3.14159;
```

Configuration Settings :

```
static String defaultLanguage = "English";
```

Database Connection Information :

```
static String databaseURL =
       "jdbc:mysql://localhost:3306/mydatabase";
 static String username = "myuser";
 static String password = "mypassword";
```

Example - Constants

```java
public class Constants {
  public static final double PI = 3.14159;
  public static final int MAX_CONNECTIONS = 10;
  public static final String DEFAULT_USERNAME = "admin";
  public static final String DEFAULT_PASSWORD = "password123";
}

public class Application {
  public static void main(String[] args) {
    System.out.println("PI: " + Constants.PI);
    System.out.println("Max Connections:" +
          Constants.MAX_CONNECTIONS);
    System.out.println("Default Username: " +
          Constants.DEFAULT_USERNAME);
    System.out.println("Default Password: " +
          Constants.DEFAULT_PASSWORD);
  }
}
```

Accessing Class Variables

- Static methods are methods that belong to the class itself, not to individual objects of the class
- They are declared with the static keyword
- They can be accessed using the class name, not an object reference

```
Math.sqrt(900.0);
```

Key Characteristics of Static Methods

- Static methods cannot directly access instance variables or other non-static methods.
- They can access static variables and other static methods of the same class.
- They can be passed instance variables or other methods as arguments.
- Often used for operations that don't require any data from an instance of the class.

Static methods are like guests in a house. They can see and use the things that are publicly available in the house (static members), but they can't go into individual rooms (instance members) unless they are invited (passed as arguments)

Common Use Cases for Static Methods

- Utility methods : Perform general-purpose tasks like calculations, string manipulation, or I/O :

```
MathUtil.add(a, b); //Math operations
StringUtil.capitalizeFirstLetter(str) //String manipulation
FileUtil.readFile(fileName) //Input/Output operations
Validator.isEmailValid(email)    //Validation
```

- Constants : Define class-wide constants using : *public static final*
- Factory methods : Create and return new instances of the class :

```
Integer intValue = Integer.valueOf("123");
System.out.println(intValue); // Output: 123
```

- State-independent Methods : When a method's behavior is not dependent on the state of an object

```
public static double inchesToCentimeters(double inches)
{  return inches * 2.54;   }
```

Example 01 : Static Method

```
public class TemperatureConverter {

  // Static method to convert Fahrenheit to Celsius
  public static double fahrenheitToCelsius(double fahrenheit) {
    return (fahrenheit - 32) * 5 / 9;
  }

  public static void main(String[] args) {
  /* Call the static method without creating an instance of the
        class*/
    double celsius =
          TemperatureConverter.fahrenheitToCelsius(100);
    System.out.println("100 degrees Fahrenheit is " + celsius
          + " degrees Celsius.");
  }
}
```

Example 02 : Static Method

```
public class MathUtil {

  public static double add(double x, double y) {
    return x + y;
  }

  public static double square(double x) {
    return x * x;
  }
}

// Usage:
double result = MathUtil.add(5, 3);
double area = MathUtil.square(4);
```

Why Is Method main Declared static ? (1)

- When you execute the Java Virtual Machine (JVM) with the java command, the JVM attempts to invoke the main method of the class you specify. Declaring main as static allows the JVM to invoke main without creating an object of the class. When you execute your application, you specify its class name as an argument to the java command, as in :

  ```
  java ClassName argument1 argument2 . . .
  ```

  The JVM loads the class specified by *ClassName* and uses that class name to invoke method **main**.

Why Is Method main Declared static ? (2)

1. **Program Entry Point :** The main method serves as the entry point for your Java program. This means it's the first method that the Java Virtual Machine (JVM) executes when you run the program. Since the main method is the starting point, it's executed directly without needing an object instance of the class containing it.
2. **No Object Required** : If the main method wasn't static, you would need to create an object of the class to call the main method. However, creating an object before the program can even start wouldn't make sense. By declaring it static, the main method becomes independent of any object instances, allowing the JVM to execute it directly

Why Is Method main Declared static ? (3)

3. **Class Loading and Memory Management** : When you run a Java program, the JVM first loads the class containing the main method into memory. Since the main method is static, it's allocated in the class memory rather than in the memory of any object instance. This simplifies memory management for the JVM.
4. **Simplicity and Efficiency** : Declaring main as static keeps the code simple and efficient. You don't need to worry about creating objects or managing their lifecycle just to run the program.
5. **Consistency with Other Languages** : Many other programming languages that use object-oriented features also follow the convention of having a static main method as the program's entry point. This consistency can help programmers familiar with other languages understand how Java programs execute.

Static main execution

```java
public class Greeting {

  public static void main(String[] args) {

    String firstName = args[0]; // First command-line argument
    String lastName = args[1];  // Second command-line argument

    System.out.println("Hello, " + firstName + " " + lastName
          + "!");
  }
}
```

Execution : java Greeting James Gosling

Result : Hello, James Gosling !

What are Instances ?

- Instances are objects created from a class.
- They represent specific entities with their own state and behavior.
- Think of a class as a blueprint, and an instance as a physical object built from that blueprint
- Each instance has its own set of attributes and methods, independent of other instances of the same class.

Syntax for Creating Instances

- Instances are created using the ***new*** keyword followed by a ***constructor***.
- Constructors are special methods within a class responsible for initializing newly created objects.

```
Person prs = new Person("Kebas", 20, "Algeria");
```

Accessing Instance Variables

Dot notation (.) is used to access instance variables / methods

```
prs.greet();
prs.introduce();

System.out.println(prs.name);
  //depends on the type of access mofifier
```

Example of an instance (object)

```java
class Person {
  String name;
  int age;
  String nationality;

  Person(String name, int age, String nationality) {
    this.name = name;
    this.age = age;
    this.nationality = nationality;
  }
  void greet() {
    System.out.println("Hello, my name is " + name + ".");
  }
  void introduce() {
    System.out.println("I am " + age + " years old and from "
          + nationality + ".");
  }
}
```

```java
public class Main {
  public static void main(String[] args) {
    // Create a Person instance (instantiation)
    Person prs = new Person("Kebas", 20, "Algeria");

    // Call object methods
    prs.greet();
    prs.introduce();
  }
}
```

Questions ?

University of Tissemsilt
Faculty of Science & Technology
Departement of Math and Computer Science

Object-Oriented Programming
Inheritance

10 avril 2024

Lecturer

Dr. HAMDANI M

Speciality : Computer Science (ISIL)
Semester : S4

Plan

Inheritance

A mechanism where a class acquires properties and behaviors from another class

- A new class of objects can be created conveniently by inheritance
- the new class (called the ***subclass***) starts with the characteristics of an existing class (called the ***superclass***), possibly customizing them and adding unique characteristics of its own.

Benefits of inheritance

- Code Reusability
- Improved Code Organization
- Flexibility and Polymorphism
- Reduced Development and Maintenance Costs
- Promotes Extensibility

Superclass

Superclass (parent class / base class) :

- The original class from which a subclass inherits.
- Defines common attributes and methods that can be used by subclasses.
- Serves as a foundation for building more specialized classes

Subclass

Subclass (child class) :

- A new class that inherits from a superclass
- Inherits all public and protected members (attributes and methods) from the superclass
- Can add its own attributes and methods to specialize its behavior
- Can override inherited methods to provide different implementations

Extends : The keyword used to declare that a subclass inherits from a superclass

Syntax of Inheritance

```
class Superclass {
  // Superclass body
}

class Subclass extends Superclass {
  // Subclass body
}
```

Example - Superclass

```java
public class Person {
  private  String firstName;
  private String lastName;
  private int age;

  public Person(String firstName, String lastName, int
        age) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
  }
   public  String getFirstName() {
    return firstName;
  }
  // Methods ...
}
```

Example - Subclass

```
public class Student extends Person {
  protected int id;
  private String speciality;

  public Student(String firstName, String lastName, int
        age, int id, String speciality) {
    super(firstName, lastName, age);
    this.id = id;
    this.speciality = speciality;
  }
  public void setSpeciality(String speciality) {
    this.speciality = speciality;
  }
    // Methods...
}
```

Things to Consider

- **Visibility** : Subclasses only inherit members declared as public, or protected in the superclass (*and **default** in the same package*). Private members are not accessible.
- **Overriding** : Subclasses can override inherited methods to provide different behavior. This is useful for customization.
- **Final Keyword** : You can prevent a class from being inherited using the ***final*** keyword.

Constructors and Inheritance

- **Constructors are not inherited** : Unlike fields and methods, constructors are not directly inherited by subclasses in Java. This is because constructors are specific to the object creation process of a particular class and its needs.
- **Calling the superclass constructor** : Subclasses can explicitly call the constructor of their superclass using the "**super**" keyword..
- **Default constructor** : If a subclass doesn't have an explicitly defined constructor, the Java compiler will implicitly call the superclass's no-argument (default) constructor, if it exists. Otherwise, a compilation error will occur :

  (If the superclass doesn't have a no-argument (*default*) constructor and only has parameterized constructors, the subclass cannot have a default constructor)

The super Keyword

- Used with inheritance to interact with the superclass (parent class) from a subclass
- Access and interact with **fields** (variables) and **methods** defined in the superclass, even if they are *hidden* or *overridden* in the subclass
- Used to explicitly call the constructor of the superclass

super is a keyword that provides a way to access members of the superclass in the context of a subclass

Example - super

```
class Person {

  private String name;

  public Person(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }
}
```

```java
class Student extends Person {
  private int studentId;

  public Student(String name, int studentId) {
    super(name);
    this.studentId = studentId;
  }

  public void introduce() {
    System.out.println("Hello, my name is " + super.getName()
          + " and my student ID is " + studentId + ".");
  }
}

public class Main {
  public static void main(String[] args) {
    Student student = new Student("Alice", 12345);
    student.introduce();
  }
}
```

Types of Inheritance

- **Single inheritance** – one superclass, one subclass
- **Multilevel inheritance** – chained subclasses and superclasses
- **Hierarchical inheritance** – multiple subclasses extend one superclass
- **Multiple inheritance** : not supported in Java
- **Hybrid Inheritance** : a mix of two or more of the above types of inheritance It can involve any combination of single, multilevel, and hierarchical Inheritance

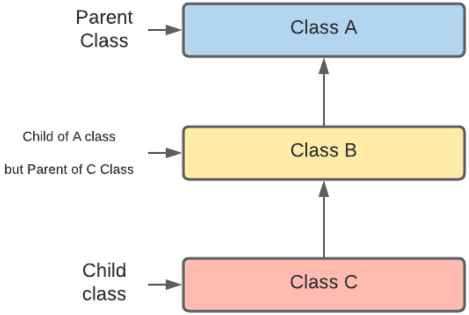
Single inheritance

A class can have only one direct parent class :

```
class Parent {

  // Parent class methods and fields
}

class Child extends Parent {

  // Child class can access methods and fields of
        Parent
}
```

Multilevel Inheritance

A class is derived from a class which is also derived from another class :

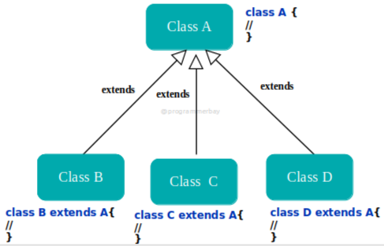


```
class Grandparent {
  // Grandparent class methods and fields
}
class Parent extends Grandparent {
  // Inherits from Grandparent
}
class Child extends Parent {
  // Inherits from Parent (and indirectly from Grandparent)
}
```

```
public class Person {
  String name;  int age;
  public Person(String name, int age)
  {  this.name = name;    this.age = age;  }
}
public class Employee extends Person {
  String employeeID;
  public Employee(String name, int age, String employeeID) {
    super(name, age);
    this.employeeID = employeeID;
  }
}
public class Professor extends Employee {
  String department;
  public Professor(String name, int age, String employeeID,
         String department) {
    super(name, age, employeeID);
    this.department = department;
  }
}
```

Hierarchical Inheritance

multiple classes inherit from a single parent class. This means a single superclass can have multiple subclasses :



```
class Parent {
  // Parent class methods and fields
}
class Child1 extends Parent {
  // Inherits from Parent
}
class Child2 extends Parent {
  // Also inherits from Parent
}
```

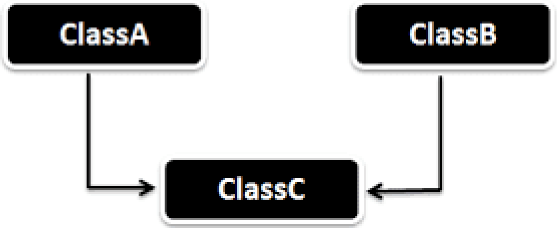
```java
public abstract class Shape {
  public abstract double area();
  public abstract double perimeter();
}

public class Circle extends Shape {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  @Override
  public double area() {
    return Math.PI * radius * radius;
  }

  @Override
  public double perimeter() {
    return 2 * Math.PI * radius;
  }
```

```java
public class Rectangle extends Shape {
  private double width;
  private double height;

  public Rectangle(double width, double height) {
    this.width = width;
    this.height = height;
  }

  @Override
  public double area() {
    return width * height;
  }

  @Override
  public double perimeter() {
    return 2 * (width + height);
  }
}
```

```java
public class Main {
  public static void main(String[] args) {
  Shape circle = new Circle(5.0);
  Shape rectangle = new Rectangle(4.0, 6.0);

  System.out.println("Circle Area: " + circle.area());
  System.out.println("Circle Perimeter: " +
         circle.perimeter());

  System.out.println("Rectangle Area: " + rectangle.area());
  System.out.println("Rectangle Perimeter: " +
         rectangle.perimeter());
}
}
```

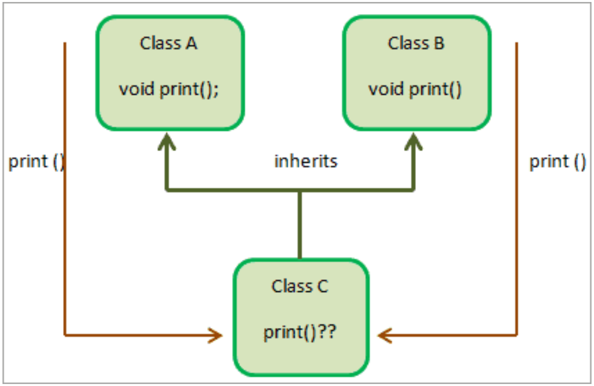
Multiple inheritance

A class inherits behaviours and attributes from more than one parent class



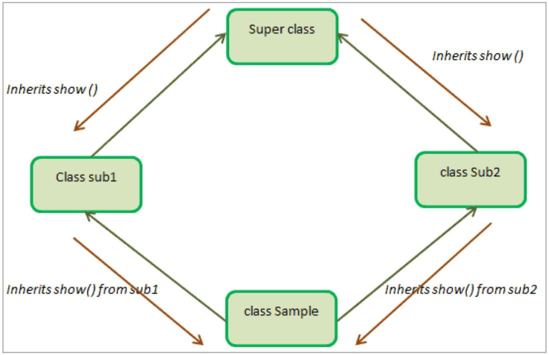
Multiple inheritance : Ambiguity

- **Complexity and Ambiguity** : when the parent classes have methods or attributes with the same names but different implementations. This ambiguity can make the code harder to read, maintain, and debug.



Multiple inheritance : Diamond Problem

- This problem arises when a class inherits from two parent classes, which themselves inherit from a common ancestor, forming a ***diamond*** shape in the inheritance hierarchy.
- The subclass inherits two copies of the methods and fields from the common ancestor, leading to ambiguity about which implementation to use



Exercice 01

Create a set of Java classes representing individuals within an academic institution, utilizing inheritance :

- **Person** : This base class should hold basic information common to all individuals (e.g., name, age, contact information).
- **Employee** : This class should inherit from Person and include additional attributes and methods specific to employees (e.g., job title, salary).
- **Student** : This class should inherit from Person and include attributes and methods related to students (e.g., student ID, Speciality).
- **Professor** : This class should inherit from Employee and include attributes and methods specific to professors (grade, specialization, departement).

https://github.com/hamdani2023/javaPOO_ISIL_S04

Declaring a class as final (1)

- This prevents other classes from inheriting from the final class.
- This is useful when you want to ensure a class cannot be extended and its behavior remains consistent.
- For example, a MathUtils class containing static utility methods might be declared final to prevent subclasses from overriding these methods and potentially introducing unexpected behavior.

Declaring a class as final (2)

When a class is declared with the final keyword, it cannot be subclassed.

```
public final class FinalClass {

  // class body
}

// This would cause a compile-time error
  class ExtendedClass extends FinalClass {

  }
```

Declaring a method as final (1)

- This prevents subclasses from overriding the method.
- This is useful when you want to ensure the specific implementation of a method remains unchanged in subclasses.
- For example, a calculateArea() method in a Shape class might be declared *final* to ensure all shapes (e.g., Circle, Rectangle) use the same logic for calculating area..

Declaring a ***field*** as ***final*** simply means its value cannot be changed after initialization, but it does not prevent inheritance −> **Constant**

Declaring a method as final (2)

Declaring a method as final means it cannot be overridden by sub-classes

```
public class SuperClass {
  public final void showFinalMethod() {
    System.out.println("This method is final and cannot be
          overridden.");
  }
}

public class SubClass extends SuperClass {
  // This would cause a compile-time error
   @Override
   public void showFinalMethod() {
         System.out.println("Attempting to override a final
                method.");
     }
}
```

Questions ?

University of Tissemsilt
Faculty of Science & Technology
Departement of Math and Computer Science

Object-Oriented Programming
Polymorphism, Abstract Classes and Interfaces

11 avril 2024

Lecturer

Dr. HAMDANI M

Speciality : Computer Science (ISIL)
Semester : S4

Plan

Introduction

Polymorphism is a fundamental concept in Java and other object-oriented programming languages, allowing for actions to behave differently based on the actual object that is performing the action

What is Polymorphism ?

- The term "polymorphism" originates from the Greek words "poly" (many) and "morph" (form).
- In Java, it refers to the ability of a single interface to control access to a general class of actions.
- You can specify a general set of stack routines that all share the same names
- The concept of polymorphism is often expressed by the phrase “one interface, multiple methods

Types of Polymorphism in Java

1. Compile-time Polymorphism (Static Polymorphism)

- Achieved through method overloading.
- Java does not support operator overloading.
- Example :
  - `void display(int a)`
  - `void display(int a, int b)`

2. Runtime Polymorphism (Dynamic Polymorphism)

- Achieved through method overriding.
- Requires inheritance.
- Example :
  - In a superclass `Animal`, a method `makeSound()` is defined.
  - The subclass `Dog` overrides `makeSound()` to provide a specific implementation.

Example : Compile-time Polymorphism

```java
public class Calculator {

  public int add(int a, int b) {
    return a + b;
  }

  public double add(double a, double b) {
    return a + b;
  }
}
```

Example : Runtime Polymorphism

```
class Animal {
  void sound() {
    System.out.println("Some sound");
  }
}
class Lion extends Animal {
  @Override
  void sound() {
    System.out.println("Roar");
  }
}
class Snake extends Animal {
  @Override
  void sound() {
    System.out.println("Hiss");
  }
}
```

```java
  public class TestPolymorphism {
   public static void main(String[] args) {
     Animal myAnimal = new Animal();
     Animal myLion = new Lion();
     Animal mySnake = new Snake();

     myAnimal.sound(); // Outputs: Some sound
     myLion.sound();   // Outputs: Roar
     mySnake.sound();  // Outputs: Hiss
   }
 }
```

What are Abstract Classes ?

- Abstract classes are classes that cannot be instantiated on their own.
- They are used to provide a base for subclasses to build upon.
- Abstract classes can include abstract methods, which are method declarations without an implementation.

Characteristics of Abstract Classes

Key Features

- **Instantiation :** Cannot create instances directly.
- **Subclassing :** Must be subclassed by concrete classes.
- **Abstract Methods :** Can contain abstract methods that *must* be implemented by subclasses.

Purpose

- Provides a template for future specific classes.
- Helps to avoid redundancy and enhance reusability.

Rules for Abstract Classes

- An abstract class may contain both abstract and non-abstract methods.
- Abstract methods do not specify a body and only provide a method signature.
- If a class includes even one abstract method, the class must be declared abstract.

Using Abstract Classes

- Abstract classes are crucial for situations where a general framework needs to be established, and specific behaviors need to be enforced.
- Subclasses of an abstract class must implement all abstract methods, but they can also override other methods.

Example

```java
// Abstract class defining common functionality for shapes
public abstract class Shape {

  // Abstract method - subclasses must provide implementation
  public abstract double calculateArea();

  // Non-abstract method with default implementation (can be
        overridden)
  public void printDetails() {
    System.out.println("This is a shape.");
  }
}
```

```java
public class Circle extends Shape {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }
  // Implementation for calculateArea() specific to circles
  @Override
  public double calculateArea() {
    return Math.PI * radius * radius;
  }
  // Overriding printDetails() to provide specific information
        for circles
  @Override
  public void printDetails() {
    System.out.println("This is a circle with radius: " +
          radius);
  }
}
```

```
public class Square extends Shape {
  private double sideLength;

  public Square(double sideLength) {
    this.sideLength = sideLength;
  }

    // Implementation for calculateArea() specific to squares
  @Override
  public double calculateArea() {
    return sideLength * sideLength;
  }
  }
```

```java
public class Rectangle extends Shape {
  private double width;
  private double height;

  public Rectangle(double width, double height) {
    this.width = width;
    this.height = height;
  }

  @Override
  public double calculateArea() {
    return width * height;
  }

  @Override
  public void printDetails() {
    System.out.println("This is a rectangle with width: " +
          width + " and height: " + height);
  }
}
```

```java
public class Main {
  public static void main(String[] args) {
/*You cannot directly create an object of the abstract class
      Shape*/
    Circle circle = new Circle(5);
    Square square = new Square(4);
    Rectangle rectangle = new Rectangle(6, 3);

    System.out.println("Circle Area: " +
          circle.calculateArea());
    System.out.println("Square Area: " +
          square.calculateArea());
    System.out.println("Rectangle Area: " +
          rectangle.calculateArea());

    circle.printDetails();
    square.printDetails();
    rectangle.printDetails();
  }
}
```

What are Interfaces ?

- Interfaces in Java are a blueprint of a class. They have static constants and abstract methods.
- Java interfaces specify what a class must do but not how it does it.
- They are implemented by classes which then define the methods' behavior.

Characteristics of Interfaces

Key Features

- **Abstract Methods :** All methods in interfaces are implicitly abstract and public.
- **Constants :** All fields are public, static, and final (constant values).
- **Implementation :** A class can implement multiple interfaces.

Why Use Interfaces ?

- To achieve abstraction.
- To support the functionality of multiple inheritance.
- To separate the method definition from the method implementation.

Defining an Interface

- Syntax to define an interface is similar to class.
- Example :

Simple Interface

```
    public interface Vehicle  {
        void cleanVehicle();
        int getNumberOfWheels();
    }
```

- This 'Vehicle' interface can be implemented by any class that pertains to a mode of transport that needs cleaning and uses wheels.

Implementing an Interface

- A class implements an interface using the '**implements**' keyword.
- It must provide a body for all abstract methods from the interface.

```
public class Car implements Vehicle {
    public void cleanVehicle() {
      System.out.println("Cleaning the vehicle");
    }
    public int getNumberOfWheels() {
      return 4;
    }
  }
```

- 'Car' class implements the 'Vehicle' interface and provides implementation for the cleaning and wheel count methods.

Example

```java
public interface Drawable {
  double PI = 3.14159; // implicitly public, static, and final

  void draw(); // implicitly public and abstract
  default void printMessage() {
    System.out.println("This is a drawable object.");
  }
}

public interface Resizable {
  void resize(int newSize);
}

public abstract class Shape {
  public abstract double getArea();
  public abstract double getPerimeter();
}
```

```java
public class Circle extends Shape implements Drawable,
      Resizable {
  private double radius;
  public Circle(double radius)
  {  this.radius = radius; }

  @Override
  public void draw()
  {   System.out.println("Draw a circle, radius:" + radius);}
  @Override
  public double getArea()
  {  return Math.PI * radius * radius;  }
  @Override
  public double getPerimeter()
  {  return 2 * Math.PI * radius;}
  public double getRadius()
  {  return radius; }
  @Override
  public void printMessage()
  {  System.out.println("This is a circle.");}
}
```

```java
Circle c = new Circle(5.0);

c.draw();

System.out.println("Area: " + c.getArea());

System.out.println("Perimeter: " + c.getPerimeter());

System.out.println("Radius: " + c.getRadius());

c.printMessage();
```

Questions ?