Coding Bootcamp

Core Java

TECHNOLOGY

Object Oriented Programming in Java

simplilearn

Learning Objectives

By the end of this lesson, you will be able to:

- Analyze the principles and roles of encapsulation, inheritance, polymorphism, and abstraction in Java
- Apply classes and objects using attributes and methods to encapsulate data and behavior
- Evaluate constructors' effectiveness in maintaining object integrity and smooth execution
- Create and implement Java class hierarchies and abstract structures for maximum code reuse and adaptability

TECHNOLOGY

OOPS in Java: Concept of Object Orientation

simplilearn

OOPs in Java

Object-Oriented Programming (OOP) empowers developers to model real-world entities by organizing their attributes and behaviors into classes.

```
public class Main {
    int x = 5;

    public static void main(String[] args) {
        Main myObj1 = new Main(); // Chair Object
        Main bottle = new Main(); // Bottle
Object
        Main cat = new Main();    // Cat Object
        System.out.println(bottle.x);
        System.out.println(myObj1.x); //
Corrected variable name
        System.out.println(cat.x);
    }
```

OOPs in Java

OOPs have made the art of programming effortless with the help of the following concepts:

- Class
- Object
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Designing the User and Product Objects in OOPs

Problem Statement:

You have been asked to implement the design of the user and product object in OOPs.

Outcome:

By implementing the design of the user and product objects in OOP, you will master the principles of object-oriented programming, including encapsulation, inheritance, and polymorphism. This will empower you to build more modular, scalable, and maintainable software systems.

Note: Refer to the demo document for detailed steps: 01_Designing_the_User_and_Product_Object_in_OOPs

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Open the IDE and use a one-dimensional array with suitable examples
2. Create a new class called OOPS
3. Use the sample e-store application to add objects
4. Identify the data in the attributes
5. Create a real object in the memory
6. Assign hash codes to the objects
7. Implement a read operation on the user object
8. Perform operations on the object
9. Access the attribute and print the details
10. Run the code and populate the data
11. Select the default constructor from the source, then generate the constructor with fields
12. Create an array with a specific size
13. Implement a two-dimensional array with suitable examples

Comparing Static and Non-Static Methods in OOPs

Problem Statement:

You have been asked to compare the static and non-static methods in OOPs.

Outcome:

By comparing static and non-static methods in object-oriented programming (OOP), you will gain a clear understanding of when and why to use each type of method. This knowledge will help you make informed decisions about method accessibility and behavior in relation to the class itself, enhancing your ability to design more effective and efficient object-oriented applications.

Note: Refer to the demo document for detailed steps: 02_Comparing_Static_and_Non-Static_Methods_in_OOPs

ASSISTED PRACTICE

Assisted Practice: Guidelines

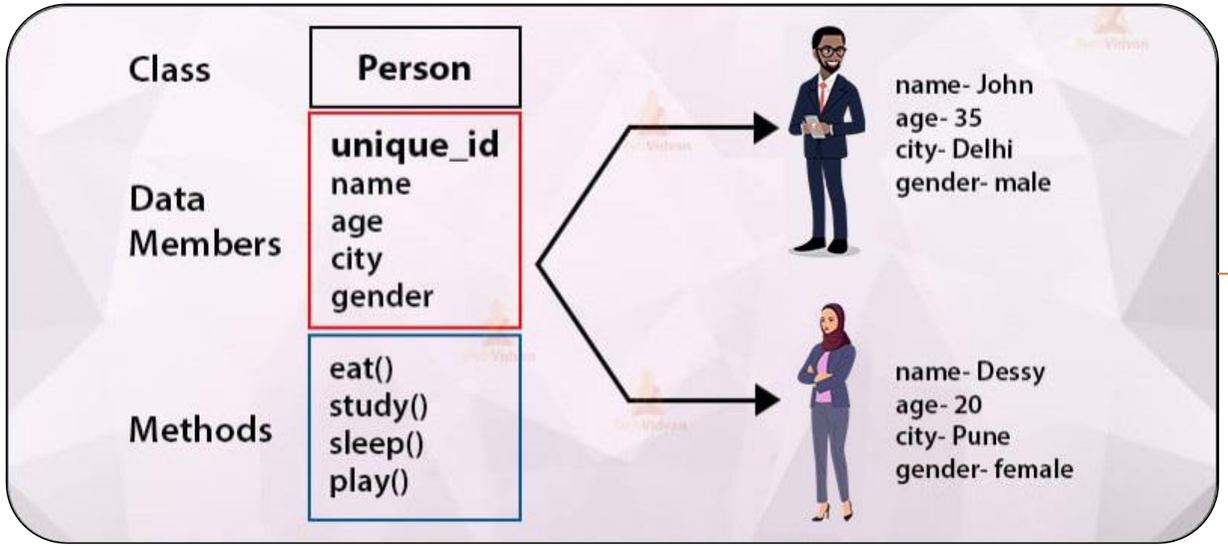
Steps to be followed are:

1. Open the Eclipse IDE and open a new Java project and a class
2. Create a default constructor
3. Write a function to update the data in the object
4. Create real objects in memory and execute the code
5. Create an attribute and differentiate between static and non-static attributes
6. Create more objects, use a reference copy, and execute the code
7. Create methods for increment and decrement
8. Create a static variable and execute the code

Classes

Class

Classes are the basic units of object-oriented programming in Java. They define objects with similar qualities and behaviors.



The class consists of data members contributing to the Java program's fully functional logic.

Class

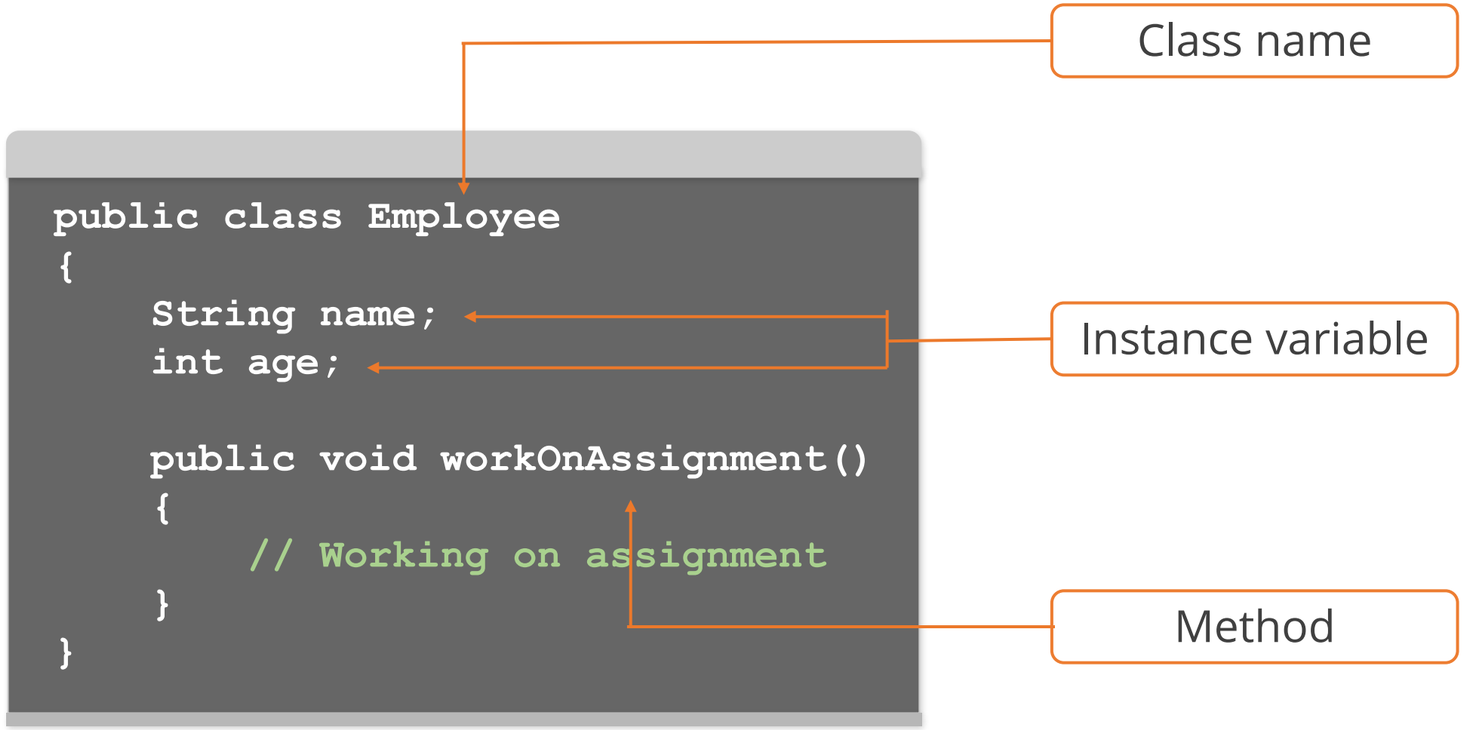
Below is an illustration of the general syntax of the Java class:

```
class class_name {
  type instance_variable_1;
  type instance_variable_2;
  …..
  ......
  type method_name_1(arguments){
    method.....
  }

  type method_name_2(arguments){
    method.....
  }}
```

Class

Instance variables can be initialized inside a class. Usually, the class structure comprises methods and instance variables, collectively forming functional code.



https://java2blog.com/object-class-java/

Class

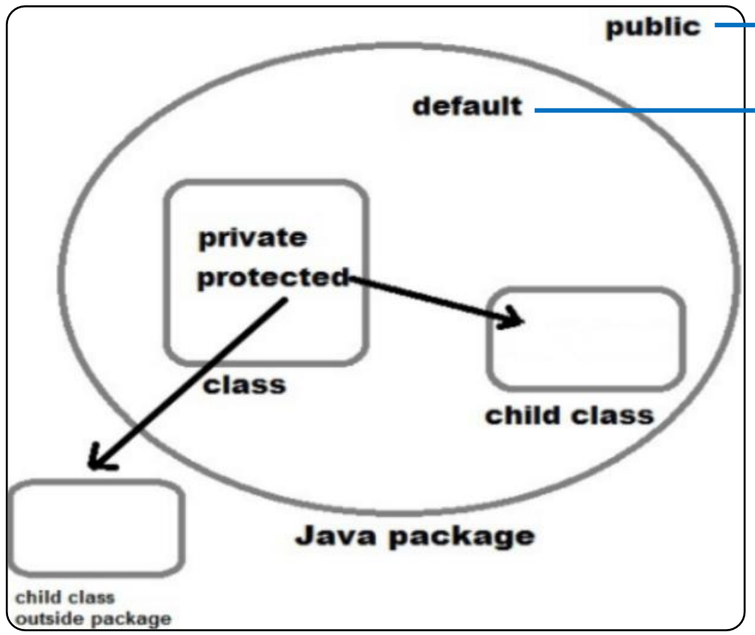
Below is an illustration of Java class:

```
class Student{
  String name;
  Int rollNo;
}
class StudentDemo {
  public static void main(String[] args) {
    Student student = new Student();
    student.name = “John”;
    student.rollNo = 5;
   System.out.println("Roll number of the student is " +
student.rollNo);
  }   }
```


The modifiers already defined in the code would impact the runtime behavior.

Access Modifiers

The first type of modifier, access modifiers, assigns public and default privileges to the classes.



Secondary or outside classes can have access to public classes.

Only internal variables and methods have access to default classes.

https://dotnettutorials.net/lesson/access-modifiers-in-java/

Access Modifiers

A non-access modifier is a special modifier that distinguishes the behavior of methods or variables in a class.

Keywords

Send notification

JVM

https://www.toptal.com/scala/scala-bytecode-and-the-jvm
http://imarket4u.com/selecting-your-seo-keywords/

Objects

Object

It is an instance of a class that encapsulates state (data) and behavior (methods) into a single entity.

Users consider nouns to identify objects. If the requirements document is for an e-commerce website, nouns such as **customer**, **product**, **order**, **payment**, and **shipping** can be considered objects.

Object

```
Vehicle bike = new Vehicle();
Item bottle = new Item();
Vehicle car = new Car();
```


Here, the **Car** class inherits properties and behaviors from the **Vehicle** class, which is a superclass. This allows the **Car** object to have all the same properties and behaviors as the **Vehicle** object but with additional functionality specific to a car.

Object

A reference created by the student object creates an instance for it and then initializes it for further execution in code.

01 Declaration

02 Instantiation

03 Initialization

```
class Student{
  char name;
  int rollno;
  //methods
  ……
  ……
}
```

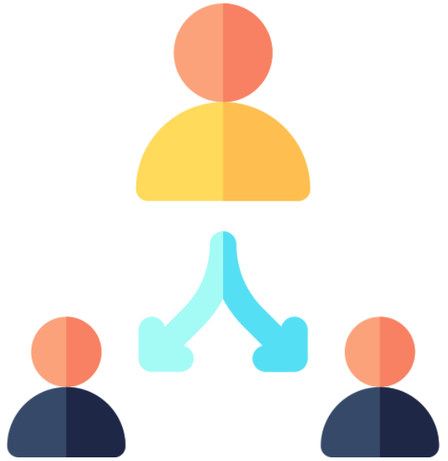
Student objects hold a reference to the object created by the Student class.

```
Student roll no;
student = new Student();
```

Inheritance

Inheritance

It is a mechanism that allows a class to inherit another class's properties (fields and methods).



Its core usage is code reusability. Inheritance allows the derived class to inherit the properties and behaviors of the base class, enabling code reuse.

Inheritance

Inheritance allows an object to inherit the properties and behaviors of its parent object, promoting code reusability and enabling runtime polymorphism.

An inherited class is referred to as a superclass or base class.

A class that inherits the superclass members is called a subclass or derived class.

The relationship between a superclass and a subclass is an **IS-A** relationship, meaning a subclass is a specialized version of its superclass.

Note

Inheritance is achieved using the **extends** keyword.

Inheritance

The extension keyword attaches to the parent-child relationship between the base and derived classes.

Syntax:

```
class Sub_class extends Super_class{
      ...
      ...
}
```

Inheritance

A derived class does not inherit a base class's private fields and methods and cannot be accessed directly by the derived class.

The derived class does not inherit the base class's constructors, but they can be invoked using the super keyword in the derived class constructor.

Note

The derived class inherits all other members of the base class (public, protected, and package-private fields and methods).

Inheritance

It is a concept wherein the properties of one class are acquired by other classes.

```
//Base class
class A{
 public int a = 9;
 private int b = 14;
 public void partOfA(){
  System.out.println("Value of a " + a);
 }
}
 // sub class
class B extends A{
 int c = 19;
 public void partOfB(){
  System.out.println("Value of field a " + a);
  // This line will give compiler error as b
  // is private and not visible
  System.out.println("Value of field b " + b);
  // Calling inherited method directly as if
  // it is a method of this class, because it is inherited
  // from super class
  partOfA();
  // ok, field of this class
  System.out.println("Value of field c " + c);
 } }
```

Inheritance

There are many types of inheritance. The most prominent among them are listed below:

Single Inheritance	A derived class is derived from a single base class.
Multi-level Inheritance	A derived class is created from another derived class, creating hierarchy levels.
Multiple Inheritance	A derived class is created with the help of more than one base class.
Hierarchical Inheritance	More than one derived class is created from the same base class.
Hybrid Inheritance	A combination of more than one type of inheritance.

Using Inheritance in Java

Problem Statement:

You have been asked to explain the need for and importance of inheritance in Java.

Outcome:

By exploring the need for and importance of inheritance in Java, you will understand how it simplifies code management and reduces redundancy. This knowledge will help you build flexible and maintainable software systems, enhancing code reuse and functionality.

Note: Refer to the demo document for detailed steps: 03_Using_Inheritance_in_Java

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Open Eclipse IDE and open a new Java project and a class
2. Create a class with three attributes
3. Create an object using the new operator and execute the code
4. Create an object with the default constructor
5. Implement the concepts of sub-class and superclass
6. Access the attributes of classes
7. Execute the code with example data
8. Understand inheritance rules and their importance, and execute sample code
9. Create a default constructor and a parameterized constructor
10. Initialize the arrays

Implementing Types of Inheritance

Problem Statement:

You have been asked to explore the types of inheritance in Java.

Outcome:

By exploring the types of inheritance in Java, you will learn about the different ways classes can inherit properties and behaviors from other classes. This understanding will enable you to design more robust and scalable applications by effectively using single, multilevel, and hierarchical inheritance.

Note: Refer to the demo document for detailed steps: 04_Implementing_types_of_Inheritance

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Open Eclipse IDE and open a new Java project and a class
2. Implement single-level inheritance
3. Implement multi-level inheritance
4. Implement the concept of hierarchy with example data
5. Implement multiple inheritances with example data
6. Use the concept of hybrid

Method Overloading and Polymorphism

Method Overloading in Java

This feature allows a class to have multiple methods with the same name but different parameters.

It can provide different methods with different parameters to cater to the various needs of users.

Polymorphism

It refers to an object's ability to take on multiple forms, allowing a single task to be performed in different ways.

Polymorphism can be done by:

Method overloading

Each class has different methods with the same name and differs in parameters.

Method overriding

Child class and parent class have the same method.

Polymorphism

Polymorphism is of two types:

- Compile time polymorphism (static polymorphism)
- Runtime polymorphism (dynamic polymorphism)

Polymorphism

Example of compile-time polymorphism:

```
public class OverloadingExample {
 // overloaded Method
 void overloadedMethod(int i){
  System.out.println("In overloaded Method
with int parameter- " + i);
}
// overloaded Method
 void overloadedMethod(int integer, String
string){
  System.out.println("In overloaded Method
with int and string parameters- " + integer
+ " " + string);
}
public static void main(String args[]){
  OverloadingExample obj = new
OverloadingExample();
  obj.overloadedMethod(8);
  obj.overloadedMethod(8, "Hello");
 } }
```

Output:

```
In overloaded Method with int
parameter- 8
In overloaded Method with int and
string parameters- 8 Hello
```

Here, there are two overloaded methods: one has a single int parameter, and the other has one int and one string parameter.

Polymorphism

Example of runtime polymorphism:

```
// Super Class
class Shape{
 protected double length;
 Shape(double length){
  this.length = length;
 }
 void area(){
 }
}

// Child class
class Square extends Shape{
 //constructor to initialize length
 Square(double side){
  super(side); // calling the superclass
constructor
 }
 //Overriding the area() method
 void area(){
  System.out.println("In area method of square");
  System.out.println("Area of square - " +
length*length);
 }
}
```

Polymorphism

Example of runtime polymorphism:

```
// Child class
class Circle extends Shape{
 //constructor to initialize length
 Circle(double radius){
  super(radius); // calling the superclass
constructor
 }
 //Overriding the area() method
 void area(){
  System.out.println("In area method of circle");
  System.out.println("Area of circle - " +
22/7*length*length);;
 }
}
public class PolymorphicTest {
 public static void main(String[] args){
  Shape Polymorphic Test;
  Square Polymorphic Test = new Square(5.0);
  Circle Polymorphic Test = new Circle(5.0);
  // shape dynamically bound to the Square object
referenced by square
  shape = square;
```

Polymorphism

Example of runtime polymorphism:

```
// area method of the square called
  shape.area();
  // shape dynamically bound to the Circle
object referenced by circle
  shape = circle;
  // area method of the circle called
  shape.area();
 }
}
```

Output:

```
In area method of square
Area of square - 25.0
In area method of circle
Area of circle - 75.0
```

The square class overrides the area() method to calculate a square's area, while the circle class overrides the area() method to calculate a circle's area.

The shape object initially references a square object, so the square's area method is called. Then, the shape object references a circle object, so the circle's area method is called.

Implementing Overloading and Overriding

Problem Statement:

You have been asked to differentiate and implement the Java overloading and overriding concepts.

Outcome:

By differentiating and implementing the concepts of overloading and overriding in Java, you will master the techniques to enhance the functionality of methods within your applications. This knowledge will allow you to create more flexible and powerful Java programs by effectively utilizing polymorphism.

Note: Refer to the demo document for detailed steps: 05_Implementing_Overloading_and_Overriding

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Create a class called overloading vs. overriding, followed by selecting the main method
2. Use the class called authentication with a login method
3. Overload the method with unique inputs
4. Use example scenarios for overloading and overriding

Implementing Run Time Polymorphism

Problem Statement:

You have been asked to depict the concept of polymorphism in Java.

Outcome:

By depicting the concept of polymorphism in Java, you will understand how this powerful feature allows objects to be treated as instances of their parent class, enabling multiple forms of behavior through a common interface. This knowledge is crucial for designing flexible and easily extendable code.

Note: Refer to the demo document for detailed steps: 06_Implementing_Run_Time_Polymorphism

ASSISTED PRACTICE

Assisted Practice: Guidelines

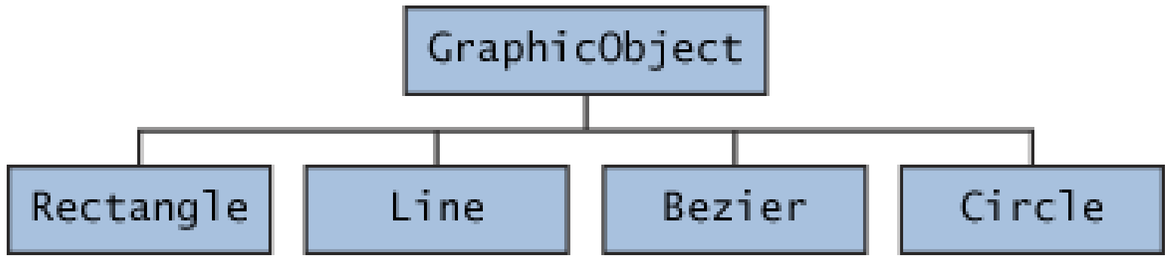
Steps to be followed are:

1. Create a class called Polymorphic statement, followed by selecting the main method
2. Write a class called CA with a method named show and a subclass called CB with its show method
3. Create a reference variable for the parent to hold the hash code of the object of the child
4. Execute the show method
5. Implement the concept of downcasting

Abstraction

Abstraction

It shows the outer functionalities and does not show the self-properties.



Classes Rectangle, Line, Bezier, and Circle Inherit from GraphicObject

https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

Abstraction

syntax

```
public interface MyInterface {
 void display(String msg);
 String getValue(String str);
}
```

The display method has only one parameter of type string, and no value is returned.

The getValue() method contains one string parameter.

Abstraction

The following classes can be implemented with the help of the interface shown below:

```
MyClass1
public class MyClass1 implements
MyInterface {
 @Override
 public void display(String msg) {
  System.out.println("Message is " + msg);
 }
 @Override
 public String getValue(String str) {
  // TODO Auto-generated method stub
  return "Hello " + str;
 }
```


Here, **display()** displays the passed parameter and **getValue()** prefixes.

Abstraction

The following classes can be implemented with the help of the interface shown below:

```
MyClass2
public class MyClass2 implements
MyInterface {
 @Override
 public void display(String msg) {
  System.out.println("Message in Uppercase
" + msg.toUpperCase());
 }
 @Override
 public String getValue(String str) {
  return str.toUpperCase();
 }
```


Here, **display()** displays the passed attribute in uppercase, and **getValue()** returns the passed attribute after converting it to uppercase.

Abstraction: Example

```
public class Test {
 public static void main(String[] args) {
  MyInterface obj;
  // Holding reference of MyClass1
  obj = new MyClass1();
  callClass(obj);
  // Holding reference of MyClass2
  obj = new MyClass2();
  callClass(obj);
 }
  private static void callClass(MyInterface obj){
  obj.display("Calling class");
  String str = obj.getValue("abstraction test");
  System.out.println("Value - " + str);
 }
}
```

Abstraction: Example

Test class along with the console output:

```
Message is Calling class
Value - Hello abstraction test
Message in Uppercase CALLING
CLASS
Value - ABSTRACTION TEST
```

An object's behavior depends on the class it references.

This abstraction hides implementation details from users.

Abstraction

Abstract methods are bodyless and are a unique way to define classes without the need for implementation.

They are methods without implementation.

The class must be abstract if it has an abstract method.

The abstract keyword defines an abstract method.

Syntax:

```
abstract void method_name();
```

https://www.javatpoint.com/abstract-class-in-java

Abstraction

Below are the snippets for syntax and a basic example of an abstract class:

```
abstract class Example //abstract class
{
//abstract method declaration
abstract void display();
}
public class MyClass extends Example
{
//method implementation
void display()
{
System.out.println("Abstract method?");
}
public static void main(String args[])
{
//creating object of abstract class
Example obj = new MyClass();
//invoking abstract method
obj.display();
}  }
```

Implementing an Abstract Class

Problem Statement:

You have been asked to use abstract classes in Java.

Outcome:

By using abstract classes in Java, you will learn how to define a class template that cannot be instantiated on its own but can form a base for other classes. This skill will enable you to create a more structured and robust hierarchy in your Java applications, promoting reusability and scalability.

Note: Refer to the demo document for detailed steps: 07_Implementing_an_Abstract_Class

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Create a class called AbstractDemo, followed by selecting the main method
2. Write a class called a cab and create a constructor for the cab class
3. Create the object of a regular class
4. Implement the concept of abstract class with examples
5. Execute the code along with the use of regular methods

Creating a Payment Gateway with Abstraction and Inheritance

Problem Statement:

You have been asked to use the concepts of abstraction and inheritance in Java.

Outcome:

By using the concepts of abstraction and inheritance in Java, you will learn to design systems that hide complex implementation details while exposing essential features. This approach, combined with the ability to extend base classes, will allow you to build more maintainable and scalable applications.

Note: Refer to the demo document for detailed steps: 08_Creating_a_Payment_Gateway_with_Abstraction_and_Inheritance

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Create a class called PaymentsApp, followed by selecting the main method
2. Create an abstract class PayTmPaymentgateway with a method bay and a private Boolean variable
3. Create a message for PayTmPaymentgateway using a conditional loop
4. Integrate the payment gateway
5. Define rules and methods for success and failure and create a constructor to execute the code
6. Implement the rule of inheritance and the limitation of abstraction

Encapsulation

Encapsulation

Encapsulation is when the code and data are wrapped into a single unit.

- The method must define how member variables can be used.
- Access control over variables and methods is achieved through access modifiers.

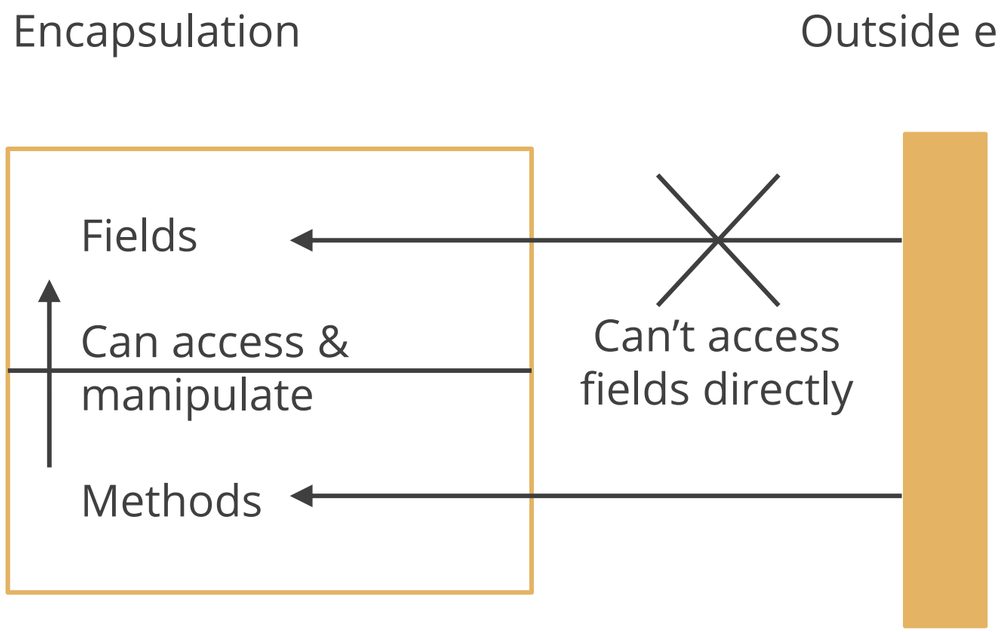
Encapsulation

Every variable or method in the class should be marked public or private to guarantee access control.

- Complete encapsulation in Java is achieved by making variables of a class private and outside of it.
- Any method that contains logic and is specific to that class should typically be marked as private.
- It prevents outside interference with an object's internal state by restricting direct access to its variables, often through private access modifiers.

Encapsulation

The methods and fields are encapsulated within the class.



Encapsulation: Example

```
public class StudentBean {
          private String firstName;
          private String lastName;
          private int rollNumber;
          private int age;
          public String getLastName() {
                    return lastName;
          }
          public void setLastName(String lastName) {
                    this.lastName = lastName;
          }
          public String getFirstName() {
                    return firstName;
          }
          public void setFirstName(StringfirstName) {
                    this.firstName = firstName;
          }
          public String getRollNumber() {
                    return rollNumber;
          }
public void setRollNumber(int rollNumber) {
                    this.rollNumber = rollNumber;
          }
          public int getAge() {
                    return age;
          }
          public void setAge(int age) {
                    if(age <= 20) {
                              this.age = age;
                    }
 // throw exception if condition fails }
```

Encapsulation: Example

```
@Override
public String toString() {

      return getFirstName() + " " +
getLastName() + " " + getAge() + " " +
getRollNumber();
      }}
 public static void main(String[] args) {
    StudentBean studentBean = new StudentBean();
    studentBean.setRollNumber(23)
    studentBean.setLastName("John");
    studentBean.setFirstName("Mathew");
    // not possible to do this as field is not
public
    studentBean.age(9);
    studentBean.setAge(14);
    System.out.println("Added Student- " +
studentBean);
  } }
```

The class employee has private fields that cannot be accessed directly from outside the class.

The values of these fields can only be modified and accessed through class methods.

Setter methods modify the values of the fields.

Getter methods retrieve the values of the fields.

Encapsulation

Output:

```
Added Student- John Mathew 14 23
```

Note

Accessing the age field with studentBean.age(7) is not possible because the field is private. Instead, using the setAge(int age) method would be best.

TECHNOLOGY

Attributes and Methods

Attributes and Methods

Attributes can be referred to as the means for storing values. The operator (.) can access the attributes. You can see the illustrations below:

```
public class Test() {
    int a = 3;   // 3 will be stored in variable 'a'.
    Int b = 1;   // 1 will be stored in variable 'b'
}
```

```
public class Example {
  int a = 9;

  public static void main(String[] args) {
    Example obj = new Example();
    System.out.println(obj.a);
  }
}
```

Command Prompt Shape – Power Point

Attributes and Methods

The code mentioned below shows how the (.) operator changes the attribute values.

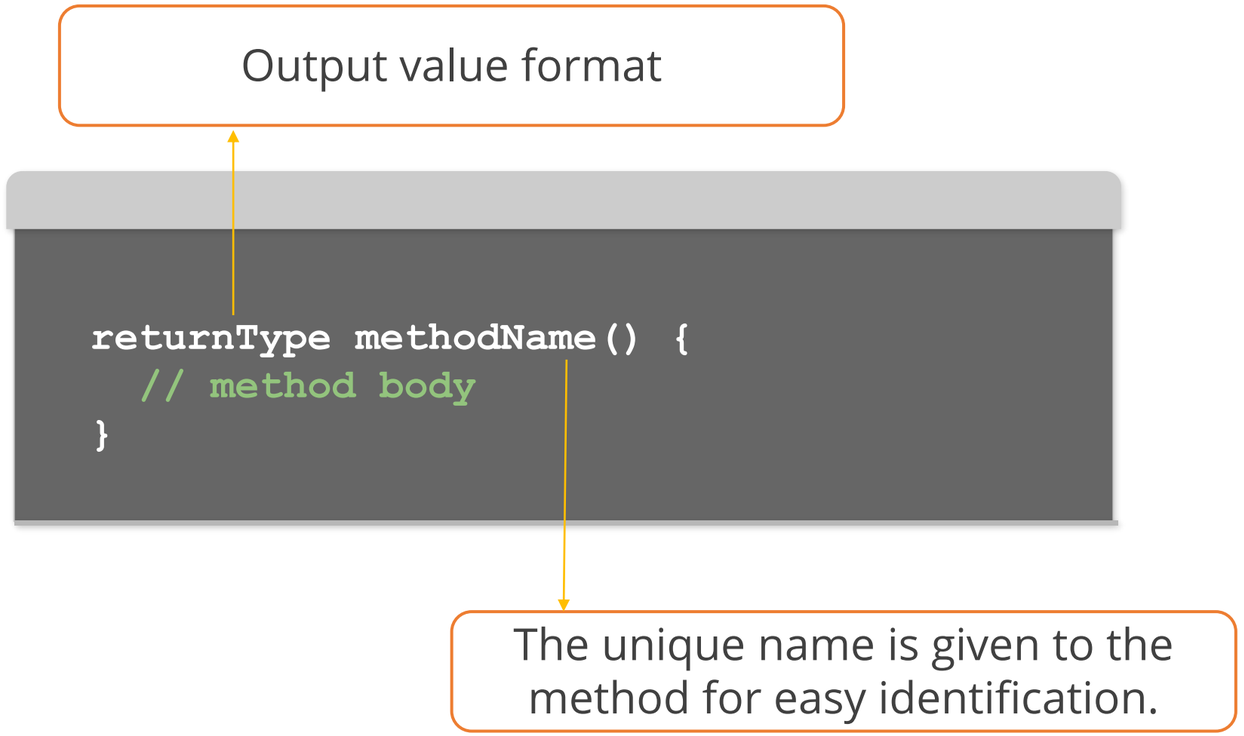
```
public class Example {
  int a = 9;

  public static void main(String[] args) {
    Example obj = new Example();
    obj.a = 11;   // Value is changed here to 11.
    System.out.println(obj.a);
  } }
```


Command Prompt Shape – Power Point

Attributes and Methods

Efficiently execute day-to-day code operations with the help of methods.
The curly brackets encompass all the variables.



Constructors

Constructors

A constructor is a special method used to initialize objects automatically during creation. It helps simplify the initialization process.

- The constructor shares the same name as the class and is defined as a method.
- It is executed automatically when an object is created using the new operator.
- Constructors do not have a return type, including void.

Constructors

The example and output is shown below:

Example:

```
public class Constructor_Example {
 int i;
 String name;
 // Constructor
 public Constructor_Example() {
  System.out.println("Creating an object");
  System.out.println("i - " + i + " name - " + name);
 }
 public static void main(String[] args) {
  Constructor_Example c1 = new Constructor_Example();
 }

//Here Constructor is this part
public Constructor_Example(String str) {
 System.out.println("Creating an object");
 System.out.println("i - " + i + " name - " + name);
}
}
```

Output:

```
Creating an object
i - 0 name - null
```

https://www.techbeamers.com/java-constructor/
Powerpoint Shapes

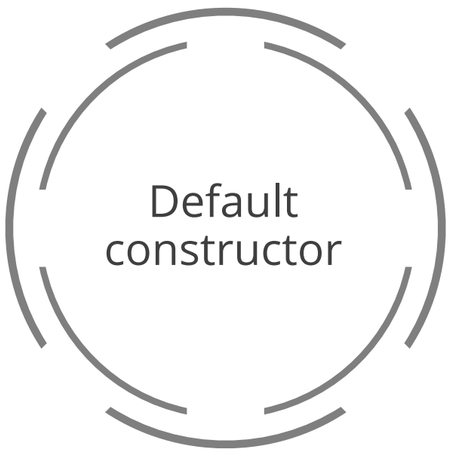
Constructors

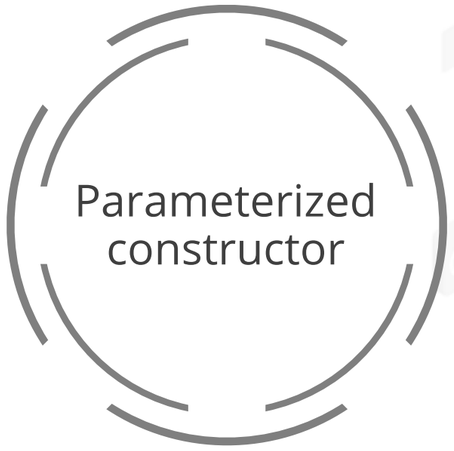
Constructor chaining refers to the order in which constructors are executed.

```
class A{
 A(){
  System.out.println("class A's constructor");
 }
}
class B extends A{
 B(){
  System.out.println(" class B's constructor");
 }
}
class C extends B{
 C(){
  System.out.println("class C's constructor");
 }
}
public class Constructor_Chaining {
public static void main(String[] args) {
  C c = new C();
 }
}
```

Constructors

There are three types of constructors:





Default Constructor

When no constructor is defined inside a class, Java inserts a default no-arg constructor for a class.

Here, Java automatically adds a default constructor.

Example:

```
public class Constructor_Example {
 int i;
 String name;

 public static void main(String[] args) {
  Constructor_Example Constructor_Example
= new Constructor_Example();
 }
}
```

Example:

```
public class .Constructor_Example {
  int i;
  Java.lang.String name;
  // Constructor
  public
org.netjs.examples.Constructor_Example();
  public static void
main(Java.lang.String[]);
}
```

No-arg Constructor

A no-arg constructor is a constructor with no parameters and must be explicitly written in a class.

Example:

```
public class Constructor_Example {
 int i;
 String name;
 public Constructor_Example() {
  System.out.println("Creating an object");
  this.name = "JOhn";
  this.i = 29;
 }
   public static void main(String[] args) {
  Constructor_Example Constructor_Example = new
Constructor_Example();
  System.out.println("i = " + Constructor_Example.i);
  System.out.println("name = " + Constructor_Example.name);
 }
}
```

Output:

```
Creating an object
i = 29
name = John
```

If a class does not explicitly define a constructor, the Java compiler automatically provides a no-argument constructor. This default constructor allows for the creation of objects without passing any arguments.

Parameterized Constructor

Constructors set initial values for an object's fields using parameters provided during object creation. A class can have multiple constructors.

```
public class Constructor_Example {
 int i;
 String year;
 // Parameterized Constructor
 Constructor_Example(int i, String year) {
  System.out.println("Creating a parameterized object");
  this.i = i;
  this.year = year;
  System.out.println("i - " + i + " year - " + year);
 }
 //no-arg constructor
 Constructor_Example() {
  System.out.println("Creating a object");
  System.out.println("i - " + i + " year - " + year);
 }
   public static void main(String[] args) {
  Constructor_Example Constructor_Example1 = new Constructor_Example(15,
"2021");
  Constructor_Example Constructor_Example2 = new Constructor_Example();
 } }
```

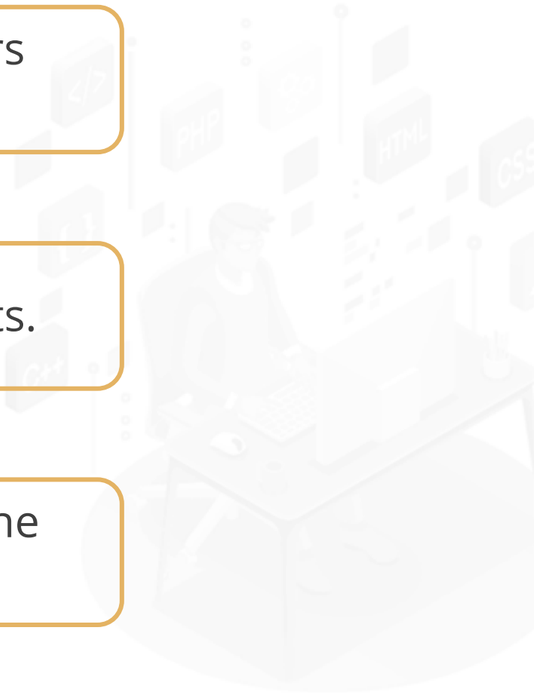
Output:

```
Creating a
parameterized object
i - 15 year - 2021
Creating an object
i - 0 year - null
```

Constructor Overloading

Constructor overloading refers to the concept where a class can have multiple constructors that differ in their types or numbers of parameters.

- It allows objects to be initialized differently based on the parameters provided.
- It means creating multiple constructors with different parameter lists.
- Each constructor can initialize an object in a specific way based on the parameters passed to it.



Initializer Block

An initializer block in Java is used to initialize instance variables, providing an alternative to constructors. Its appearance is similar to that of a static initializer block.

General form of initializer block:

```
{
Code for initialization
}
```


In each constructor, the initializer block is duplicated by Java. This code block can be utilized across multiple constructors to facilitate code sharing.

Initializer Block

Example:

```
public class InitBlockDemo {
 // no-arg constructor
 InitBlockDemo(){
  System.out.println("no-arg constructor invoked");
 }
 // constructor with one param
 InitBlockDemo(int i){
  System.out.println("constructor with one param
invoked");
 }
 // initializer block
 {
  System.out.println("This will be invoked for all
constructors");
 }
 public static void main(String[] args) {
  InitBlockDemo ibDemo = new InitBlockDemo();
  InitBlockDemo ibDemo1 = new InitBlockDemo(10);
 }
}
```

Final Keyword

Final Keyword

The final keyword restricts the modification of variables, methods, and classes.

- A blank final variable is uninitialized and can only be assigned a value within the constructor.
- Static final variables can only be initialized in a static block.
- The value of a final variable cannot be changed.

Final Keyword

The value of the final variable cannot be changed.

```
class Bike{
 final int speedLimit=80;//final
variable
 void run(){
  speedLimit=300;
 }
 public static void main(String args[]){
 Bike obj=new  Bike();
 obj.run();
 }  }
```

The method cannot be overridden if it is marked as final.

```
class Bike{
  final void
run(){System.out.println("running");}
}
class Honda extends Bike{
   void
run(){System.out.println("running safely
with 100kmph");}
        public static void main(String
args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

Implementing Final Variables and Methods in Java

Problem Statement:

You have been asked to use the concept of the final keyword in Java.

Outcome:

By using the concept of the final keyword in Java, you will learn how to prevent further modification of variables, methods, or classes. This knowledge will help you ensure that your critical code components remain unchanged and secure throughout the application lifecycle, enhancing stability and predictability.

Note: Refer to the demo document for detailed steps: 09_Implementing_Final_Variables_and_Methods_in _Java

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Create a class called FinalKeyword, followed by selecting the main method
2. Define a normal variable and a final variable with examples
3. Create a class with a method pay and inherit this class
4. Override and customize the pay method
5. Mark the method as final to limit redefining methods

Using This and Super in Java

Problem Statement:

You have been asked to differentiate the use of This and Super in Java.

Outcome:

By differentiating the use of 'this' and 'super' in Java, you will learn how 'this' refers to the current object instance, while 'super' connects to the superclass of the current object. This understanding will enable you to manage class hierarchies more effectively and enhance object interaction within your applications.

Note: Refer to the demo document for detailed steps: 10_Using_this_and_super_in_Java

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Create a class called ThisDemo, followed by selecting the main method
2. Create a class called User and a default constructor
3. Create two user objects, the first one to hold the hash code of the user object and the second one for a new user
4. Execute the code with examples
5. Implement constructor chaining and create a user object
6. Create a class called SuperDemo with two classes: parent and child
7. Create multiple constructors for the parent constructor and use parameterized constructors
8. Create the object of the child with the default constructor while reusing the code
9. Create an inheritance relationship
10. Implement a super execution call and override the default behavior

Static Keyword

Static

Properties of static keyword are:

- Static keywords aid in memory management in Java.
- It can be used with methods, variables, blocks, and nested classes.
- Static variables belong to the class, not an instance of the class.
- They refer to a common property shared by all objects, not unique to each.
- Static variables are allocated memory once during class loading.

Static

Example

```
class Employe{
   int employeId; //instance variable
   String name;
   static String companyName ="ABC"; //static variable
   //constructor
   Employe(int id, String n){
   employeId = id;
   name = n;
   }
   //method to display the values
   void display (){System.out.println(employeId+"
"+name+" "+companyName);}
}
//Example class to show the values of objects
public class Example{
 public static void main(String args[]){
 Employe e1 = new Employe (121,"Manoj");
 Employe  e2 = new Employe (322,"Brij");

 e1.display();
 e2.display();
 }
}
```

Static: Example

```
public class example
{
public static void main(String[]
args)
{
show();
}
static void show()
{
System.out.println("Example for
static method.");
}
}
```

A static method belongs to a class, not an instance of the class.

It can be created using the **static** keyword before the method name.

Static methods do not require an object to be called.

They can access and manipulate static data members and be called using the class name.

Static

The restrictions for using the static method are:

Static methods cannot directly access static data members or invoke non-static methods.

The **super** and **this** keywords cannot be used within the context of a static method.

Static

A static block is used to initialize the static data member.

```
class A2{

static{System.out.println("sta
tic block is called");}
  public static void
main(String args[]){
   System.out.println("Hello
World");  //Error
  }
}
```

Implementing Object Class in Java

Problem Statement:

You have been asked to implement the use of object class in Java.

Outcome:

By implementing the use of the object class in Java, you will learn how this root class of the Java class hierarchy is integral in allowing all classes to inherit common methods. This knowledge will enhance your ability to manipulate and manage objects efficiently across your Java applications.

Note: Refer to the demo document for detailed steps: 11_Implementing_Object_Class_in_Java

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed are:

1. Create a class called Object Demo, followed by selecting the main method
2. Create a class named Product with three attributes
3. Create a parameterized constructor and create the object of product
4. Execute the code with examples and overriding

Key Takeaways

- OOPs in Java is to improve code readability and reusability by defining a Java program efficiently.
- Objects can be generated from classes, which act like a blueprint.
- An object has a behavior and a state. It can be created in three steps.
- Inheritance helps with code reusability and runtime polymorphism.
- Polymorphism can be done through method overloading or overriding.

Key Takeaways

- Encapsulation provides maintainability and flexibility.
- Final keyword restricts the user and is used for variables, method, or classes.
- Static keywords can be used with methods, variables, or blocks and aid in memory management.
- Abstraction hides the internal details by only displaying the functionalities.