CCBP TECH 4.0 INTENSIVE

With Industry-Ready Certification (IRC)

Java Summary Cheat Sheet - 4

https://www.ccbp.in/

Classes

Classes: `Class` can be used to bundle related properties and actions.

```
class ClassName {
    // attributes
    // methods
}
```

this keyword: Java consists of `this` keyword to access the instance attributes and methods. In Java, however, we can access them without using `this` keyword.

Attributes & Methods: In Java, instance or class attributes/methods are distinguished with the `static` keyword.

Instance Attributes: In Java, instance attributes are also called non-static attributes.

```
class Mobile {
    String model;
    String camera;
    // methods
}
```

Instance Methods: In Java, instance methods are also called non-static methods.

```
class Mobile {
    // attributes
    void makeCall() {
        System.out.println("calling...");
    }
}
```

Constructor: Unlike Java methods, a constructor should be named the same as the class and should not return a value.

```
class Mobile {
    String model;
    String camera;
    Mobile(String modelSpecs, String cameraSpecs) {
        model = modelSpecs;
        camera = cameraSpecs;
    }
}
```

Instance of Class: An instance of a class is called an `object`. An `object` is simply a collection of attributes and methods that act on those data.

```
class Mobile {
    String model;
    String camera;
    Mobile(String modelSpecs, String cameraSpecs) {
        model = modelSpecs;
        camera = cameraSpecs;
    }
}
class Base {
    public static void main(String[] args) {
        Mobile mobile1 = new Mobile("Samsung Galaxy S22", "108 MP");
    }
}
```

Accessing Attributes & Methods with Objects: we can use the dot (`.`) notation to access the attributes and methods with `objects` of a `class`.

```
class Mobile {
    String model;
    String camera;
    Mobile(String modelSpecs, String cameraSpecs) {
        model = modelSpecs;
        camera = cameraSpecs;
    }
    void makeCall(long phnNum) {
        System.out.printf("calling...%d", phnNum);
    }
}
class Base {
    public static void main(String[] args) {
        Mobile mobile1 = new Mobile("Samsung Galaxy S22", "108 MP");
        System.out.println(mobile1.model);
        System.out.println(mobile1.camera);
        mobile1.makeCall(9876543210L);
    }
}
```

```
// Output is:
iPhone 12 Pro
12 MP
calling...9876543210
```

Updating Attributes: We can update the attributes of the `objects`.

```
class Mobile {
    String model;
    String camera;
    Mobile(String modelSpecs, String cameraSpecs) {
        model = modelSpecs;
        camera = cameraSpecs;
    }
}
class Base {
    public static void main(String[] args) {
        Mobile mobile = new Mobile("iPhone 12 Pro", "12 MP");
        mobile.model = "Samsung Galaxy S22";
        mobile.camera = "108 MP";
        System.out.println(mobile.model);
        System.out.println(mobile.camera);
    }
}
```

```
// Output is:
Samsung Galaxy S22
```

```
108 MP
```

Class Attributes: Attributes whose values stay common for all the `objects` are modelled as class Attributes. The `static` keyword is used to create the class attributes.

```
class Cart {
    static int flatDiscount = 0;
    static int minBill = 100;
}
```

Accessing Class Attributes: The class attributes can also be accessed using the dot (`.`) notation. We can access the class attributes directly using the class name.

```
class Cart {
    static int flatDiscount = 0;
    static int minBill = 100;
}
class Base {
    public static void main(String[] args) {
        System.out.println(Cart.flatDiscount);
        System.out.println(Cart.minBill);
    }
}
```

```
// Output is:
0
100
```

Class Methods: In Java, class methods are also called static methods. The `static` keyword is used to create the class methods.

```
class Cart {
    static int flatDiscount = 0;
    static int minBill = 100;
    static void updateFlatDiscount(int newFlatDiscount) {
        flatDiscount = newFlatDiscount;
    }
}
```

Accessing Class Methods: The class methods can also be accessed using the dot (`.`) notation. We can access the class methods directly using the class name.

```
class Cart {
    static int flatDiscount = 0;
    static int minBill = 100;
    static void updateFlatDiscount(int newFlatDiscount) {
        flatDiscount = newFlatDiscount;
    }
}
class Base {
    public static void main(String[] args) {
        Cart.updateFlatDiscount(50);
        System.out.println(Cart.flatDiscount); // 50
    }
}
```

OOPS

OOPS: Object-Oriented Programming System (OOPS) is a way of approaching, designing, developing software that is easy to change.

Encapsulation: It is a process of wrapping related code and data together into a single unit.

```
class Student {
    private int age;
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
class Main {
    public static void main(String[] args) {
        Student student = new Student();
        student.setAge(20);
        System.out.println(student.getAge()); // 20
    }
}
```

Inheritance: It is the mechanism by which one class is allowed to inherit the features(fields and methods) of another class.

```
class Mammal {
  // attributes and methods
}
class Horse extends Mammal {
  // attributes and methods of Horse
}
```

The class from which the subclass is derived is called a `superclass`.

```
class Mammal {
    String name;
    Mammal(String name) {
        this.name = name;
    }
    void eat() {
        System.out.println("I am eating");
    }
}
```

The class that is derived from another class is called a `subclass`.

```
class Horse extends Mammal {
    void displayName() {
        System.out.println("My name is " + name);
    }
}
```

Method Overriding: It allows a subclass to provide a specific implementation of a method that its superclass already provides.

```
class Mammal {
    void eat() {
        System.out.println("Mammal is eating");
    }
}
class Horse extends Mammal {
    void eat() {
        System.out.println("Horse is eating");
    }
}
class Base {
    public static void main(String[] args) {
        Horse horse = new Horse();
        horse.eat();
    }
}
```

```
// Output is:
Horse is eating
```

Rules of Method Overriding

Constructors cannot be overridden
The overriding method should have the same return type and parameters
A final method cannot be overridden
The private methods cannot be overridden
The access level cannot be more restrictive than the overridden method's access level

Composition: It describes a class whose non-static attributes refer to one or more objects of other classes.

```
import java.util.*;
class Book {
    String title;
    String author;
    Book(String title, String author) {
        this.title = title;
        this.author = author;
    }
}
class Library {
    ArrayList<Book> books;
    Library(ArrayList<Book> books) {
        this.books = books;
    }
    void displayBooks() {
        for (Book book : books) {
            System.out.printf("Title: %s, Author: %s\n", book.title, book.author);
        }
    }
}
class Base {
    public static void main(String[] args) {
        Book book1 = new Book("Head First Design Patterns", "Eric Freeman");
        Book book2 = new Book("Clean Code", "Robert C. Martin");
        ArrayList<Book> booksList = new ArrayList<>();
        booksList.add(book1);
        booksList.add(book2);
        Library library = new Library(booksList);
        library.displayBooks();
    }
}
```

```
// Output is:
Title: Head First Design Patterns, Author: Eric Freeman
Title: Clean Code, Author: Robert C. Martin
```

When to use Inheritance & Composition?
Inheritance: Prefer modeling with inheritance when the classes have an IS-A relationship.
Composition: Prefer modeling with composition when the classes have the HAS-A relationship.

Polymorphism: It refers to an object's capacity to take several forms. Polymorphism allows us to perform the same action in multiple ways in Java. Polymorphism is of two types:

Compile-time polymorphism
Runtime polymorphism

Compile-time Polymorphism: A polymorphism that occurs during the compilation stage is known as a Compile-time polymorphism. Method overloading is an example of compile-time polymorphism.

```
class Shapes {
    public void area(double base, double height) {
        System.out.println("Area of Triangle = " + 0.5 * base * height);
    }
    public void area(int length, int breadth) {
        System.out.println("Area of Rectangle = " + length * breadth);
    }
}
class Base {
    public static void main(String[] args) {
        Shapes triangle = new Shapes();
        Shapes rectangle = new Shapes();
        triangle.area(8.5, 10.23);
        rectangle.area(5, 3);
  }
}
```

```
// Output is:
Area of Triangle = 43.4775
Area of Rectangle = 15
```

Runtime Polymorphism: A polymorphism that occurs during the execution stage is called Runtime polymorphism. Method overriding is an example of Runtime polymorphism.

```
class Mammal {
    void eat() {
        System.out.println("Mammal is eating");
    }
}
class Dog extends Mammal {
    void eat() {
        System.out.println("Dog is eating");
    }
}
class Base {
    public static void main(String args[]) {
        Mammal mammal = new Dog();
        mammal.eat();
    }
}
```

```
// Output is:
Dog is eating
```

Abstraction: It is the process of hiding certain details and showing only essential information to the user.
Abstraction can be achieved with,

Abstract classes
Interfaces

Abstract Classes and Methods
Abstract Classes: The `abstract` keyword is a non-access modifier, applied to classes and methods in Java. A class which is declared with the `abstract` keyword is known as an abstract class.

```
abstract class ClassName {
    // attributes and methods
}
```

Abstract Methods: A method which is declared with the `abstract` keyword is known as an abstract method.

```
abstract returnType methodName();
```

Interface: It is similar to an abstract class. It cannot be instantiated and consists of abstract methods.

```
interface InterfaceName {
    // body of the interface
}
```

The `implements` keyword is used to inherit interfaces from a class.

```
interface CricketPlayer {
    void run();
}
class Person implements CricketPlayer {
    public void run() {
        System.out.println("Running");
    };
}
class Base {
    public static void main(String[] args) {
        Person person = new Person();
        person.run();
    }
}
```

```
// Output is:
Running
```

Default Methods in Interfaces: We can write the implementation of a method inside the interface. These methods are called `default methods`.
The `default` keyword is used to declare a method in the interface as `default method`.

```
accessModifier default returnType methodName() {
    // block of code
}
```

Inheritance among Interfaces: An interface can inherit another interface. We use the `extends` keyword to inherit an interface from another interface.

```
interface CricketPlayer {
    void run();
}
```

```
interface Wicketkeeper extends CricketPlayer {
    void wicketkeeping();
}
class Person implements Wicketkeeper {
    public void wicketkeeping() {
        System.out.println("Wicketkeeping");
    }
    public void run() {
        System.out.println("Running");
    };
}
class Base {
    public static void main(String[] args) {
        Person person = new Person();
        person.wicketkeeping();
        person.run();
    }
}
```

```
// Output is:
Wicketkeeping
Running
```

Errors & Exceptions

Errors & Exceptions
Errors: In Java occur due to syntactical errors, infinite recursion, and many other reasons. The most common are syntactical errors that occur when a programmer violates the rules of Java programming language.
Exceptions: Even when our code is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called Exceptions.

Handling Exceptions: In Java, we have `try...catch` block to handle the exceptions.
we can specify multiple `catch` blocks to handle different types of exceptions.
`finally` block which is always executed whether an exception occurs inside the try block or not.

```
try {
    int result = 5 / 0;
    System.out.println(result);
    } catch (ArithmeticException e) {
        System.out.println("Denominator can't be 0");
    } catch (ArithmeticException e) {
        System.out.println("Invalid value");
    } finally {
        System.out.println("Execution completed");
    }
}
```

```
// Output is:
Denominator can't be 0
Execution completed
```

Raising Exceptions: In Java, we can throw an exception explicitly using `throw` keyword.

```
try {
    throw new ArithmeticException("Denominator can't be 0");
} catch (ArithmeticException e) {
    System.out.println(e.getMessage());
}
```

```
// Output is:
Denominator can`t be 0
```

throws Keyword: In Java, `throws` keyword is used to specify the type of exception that might be thrown by a method.

```
class Main {
    static void divideByZero() throws ArithmeticException {
        throw new ArithmeticException("Division with zero");
    }
    public static void main(String[] args) {
        try {
            divideByZero();
        } catch (ArithmeticException e) {
            System.out.println(e);
        }
    }
}
```

```
// Output is:
java.lang.ArithmeticException: Division with zero
```

Working With Dates & Times

Date and Time

Java has a built-in java.time package which provides various classes to work with date and time.

Working with LocalDate class

Java `LocalDate` class allows us to create a `date object` and represent a valid date (year, month and day).
The `of()` method of `LocalDate` class is used to create an instance of `LocalDate` from the given year, month and day.

```
LocalDate dateObj = LocalDate.of(2019, 4, 13);
System.out.println(dateObj); // 2019-04-13
```

Today's Date: The `LocalDate` class provides `now()` method which returns the date object with today's date.

```
LocalDate dateObj = LocalDate.now();
System.out.println(dateObj); //2023-2-22
```

Working with LocalTime class

Java `LocalTime` class allows us to create a `time object` and represent a valid time (hours, minutes and seconds).

```
LocalTime timeObj = LocalTime.of(11, 34, 56);
System.out.println(timeObj); // 11:34:56
```

Working with LocalDateTime class

Java `LocalDateTime` class allows us to create a `date-time object` and represent a valid date and time together.

```
LocalDateTime dateTimeObj = LocalDateTime.of(2018, 11, 28, 10, 15, 26);
System.out.println(dateTimeObj.getYear()); // 2018
System.out.println(dateTimeObj.getMonthValue()); // 11
System.out.println(dateTimeObj.getHour()); // 10
System.out.println(dateTimeObj.getMinute()); // 15
```

Working with DateTimeFormatter class

The `DateTimeFormatter` class have a `ofPattern()` method that creates an instance of the `DateTimeFormatter` for the given pattern of the date, time and date-time like,

mm/dd/yyyy
hh/mm/ss

```
LocalDate now = LocalDate.now();
DateTimeFormatter format1 = DateTimeFormatter.ofPattern("dd MMMM yyyy");
String formattedDate = now.format(format1);
System.out.println(formattedDate); // 13 September 2022
```

Parsing Date and Time
The `DateTimeFormatter` class have a `parse()` method which creates the respective object (date, time, or date-time) from the give string.

```
String dateStr = "28 November 2018";
DateTimeFormatter format1 = DateTimeFormatter.ofPattern("dd MMMM yyyy");
LocalDate date = LocalDate.parse(dateStr, format1);
System.out.println(date); // 2018-11-28
```

Difference Between Dates & Times:
In Java, we have a `Period` class to find the difference between two dates in terms of years, months and days.

```
LocalDate startDate = LocalDate.of(2020, 2, 20);
LocalDate endDate = LocalDate.of(2021, 10, 21);
Period period = Period.between(startDate, endDate);
System.out.println(period.getYears()); // 1
System.out.println(period.getMonths()); // 8
System.out.println(period.getDays()); // 1
```

In Java, we have a `Duration` class to find the difference between two times in seconds.

```
LocalTime startTime = LocalTime.of(10, 30, 30);
LocalTime endTime = LocalTime.of(10, 31, 30);
Duration duration = Duration.between(startTime, endTime);
System.out.println(duration.getSeconds()); // 60
```

Types of Inheritance

Single Inheritance: Single inheritance involves extending a single superclass from a single subclass.

```
class Mammal {
    String type;
}
class Horse extends Mammal {
    String breed;
}
```

Multilevel Inheritance: In multilevel inheritance, a subclass extends from a superclass and then the same subclass acts as a superclass for another class.

```
class Mammal {
    String type;
}
class Horse extends Mammal {
    String breed;
}
class MustangHorse extends Horse {
    String name;
}
```

Hierarchical Inheritance: In hierarchical inheritance, multiple subclasses extend from a single superclass.

```
class Mammal {
    String type;
}
class Horse extends Mammal {
    String breed;
}
class Dog extends Mammal {
    String breed;
}
```

Multiple Inheritance: In multiple inheritance, a single class/interface can inherit multiple interfaces.

```
interface InswingBowler {
    void inswing();
}
interface OutswingBowler {
    void outswing();
}
class BowlerA implements InswingBowler, OutswingBowler {
    public void inswing() {
        System.out.println("Inswing bowling");
    }
    public void outswing() {
        System.out.println("Outswing bowling");
    }
}
```

Final

Final keyword: The `final` keyword is used for variables, classes and methods, which makes them non-changeable (impossible to inherit or override).

Final variable: When the `final` keyword is used with a variable, it indicates that the variable is constant and the value of it cannot be reassigned.

```
final int SPEED_LIMIT = 60;
SPEED_LIMIT = 90;
System.out.println(SPEED_LIMIT);
```

```
// Output is:
file.java:4: error: cannot assign a value to final variable SPEED_LIMIT
        SPEED_LIMIT = 90;
        ^
```

Final method: A method declared with `final` is called a `final method`, it restrict the unwanted and improper use of method definition while overriding the method.

```
class Mammal {
    final void eat() {
        System.out.println("Mammal is eating");
    }
}
class Horse extends Mammal {
    void eat() {
        System.out.println("Horse is eating");
    }
}
class Base {
    public static void main(String[] args) {
        Horse horse = new Horse();
        horse.eat();
    }
}
```

```
// Output is:
file.java:10: error: eat() in Horse cannot override eat() in Mammal
    void eat() {
         ^
overridden method is final
```

Final class: A class declared with `final` is called a `final class`, it cannot be inherited. All the wrapper classes are `final classes`. Hence, we cannot inherit the wrapper classes.

```
final class Product {
    String name;
    int price;
}
class ElectronicItem extends Product {
    int warrantyInMonths;
}
class Base {
    public static void main(String[] args) {
        ElectronicItem obj = new ElectronicItem();
    }
}
```

```
// Output is:
Main.java:5: error: cannot inherit from final Product
class ElectronicItem extends Product {
                             ^
```

Access Modifiers

Access Modifiers: `Access modifiers` are the keywords that set access levels when used with the classes, methods, constructors, attributes, etc.
In Java, we have four access modifiers to set the accessibility. They are,

Private: The access level of a private modifier is only within the declared class. It is not accessible outside of the class.
Default: The access level of a default modifier is up to the class/subclass of the same package. It cannot be accessed from outside the package.
Protected: The access level of a protected modifier is up to the class/subclass of the same package and also to a different package through the subclass. A subclass is required to access it from a different package.
Public: The access level of a public modifier is everywhere in the program. It means that it is accessible from the class/subclass of the same/different package.

Accessibility of Access Modifiers

Access Modifier	Same Class	Same package subclass	Same package other classes	Different package subclass	Different package other classes
private	Yes	No	No	No	No
default	Yes	Yes	Yes	No	No
protected	Yes	Yes	Yes	Yes	No

Access Modifier	Same Class	Same package subclass	Same package other classes	Different package subclass	Different package other classes
public	Yes	Yes	Yes	Yes	Yes

All the `access modifiers` work the same with the variables, methods and constructors.

Access Modifiers with Classes: `Classes` in Java can only have `Public` or `Default` as access modifiers.

Public: When a class is declared `public`, it is accessible from the class/subclass of the same/different package.
Default: When no access modifier is specified to the classes, then they can be called `default classes`, it is accessible only to the classes or subclasses of the same package.

Upcasting

Upcasting: In Java, a superclass reference variable can be used to refer to its subclass object i.e., we can specify a superclass as a type while creating an object of its subclass.

Invoking Methods: While upcasting, we can access all the members of the superclass but can only access a few members like overriding methods of the subclass.

```java
class Mammal {
    void eat() {
        System.out.println("Mammal is eating");
    }
}
class Horse extends Mammal {
    void eat() {
        System.out.println("Horse is eating");
    }
}
class Base {
    public static void main(String[] args) {
        Mammal animal = new Horse();
        animal.eat();
    }
}
```

```
// Output is:
Horse is eating
```

Invoking a method specific to subclass,

```java
class Mammal { }
class Horse extends Mammal {
    void eat() {
        System.out.println("Horse is eating");
    }
}
class Base {
    public static void main(String[] args) {
        Mammal animal = new Horse();
        animal.eat();
    }
}
```

```
// Output is:
Main.java:10: error: cannot find symbol
        animal.eat();
```

Invoking Methods: We cannot access the attributes of the subclass.

```
class Mammal { }
class Horse extends Mammal {
    String breed = "Shire";
}
class Base {
    public static void main(String[] args) {
        Mammal animal = new Horse();
        System.out.println(animal.breed);
    }
}
```

```
// Output is:
file.java:8: error: cannot find symbol
        System.out.println(animal.breed);
```

Super Keyword

super: super is the keyword is used to access the superclass members like attributes, methods and also the constructors inside the subclass.

Accessing Attributes and Methods using super Keyword

```
class Mammal {
    String type = "animal";
    void eat() {
        System.out.println("Eating");
    }
}
class Horse extends Mammal {
    String type = "mammal";
    void display() {
        System.out.println("Horse is an " + super.type);
    }
    void eat() {
        super.eat();
    }
}
class Base {
    public static void main(String[] args) {
        Horse horse = new Horse();
        horse.display();
        horse.eat();
    }
}
```

```
// Output is:
Horse is an animal
Eating
```

Invoking Constructors using super()

Invoking non-parameterized superclass constructor
Invoking parameterized superclass constructor

Invoking non-parameterized superclass constructor

```
class Mammal {
    String name;
    Mammal() {
        System.out.println("Superclass constructor called");
    }
}
class Horse extends Mammal {
    String breed;
    Horse(String breed) {
        this.breed = breed;
    }
}
class Base {
    public static void main(String[] args) {
        Horse horse = new Horse("Shire");
    }
}
```

```
// Output is:
Superclass constructor called
```

Invoking parameterized superclass constructor

```
class Mammal {
    String name;
    Mammal(String name) {
        this.name = name;
    }
}
class Horse extends Mammal {
    String breed;
    Horse(String name, String breed) {
        super(name);
        this.breed = breed;
    }
}
class Base {
    public static void main(String[] args) {
        Horse horse = new Horse("Alex", "Shire");
        System.out.println(horse.name);
        System.out.println(horse.breed);
    }
}
```

```
// Output is:
Alex
Shire
```