- **Interface:** An interface is a collection of abstract methods. It is used to achieve abstraction and multiple inheritance in Java.

**Ex:**
```
interface Animal {
    void sound();
}
class Dog implements Animal {
    public void sound() {
        System.out.println("Dog barks");
    }
}
```

2018-19 : MID III

2021-22 : Final

II (a) Method override vs Method overloading:

| Overloading | Overriding |
|---|---|
| - Method overloading is a compile time polymorphism | - Method overriding is a run time polymorphism |
| - It is occured within the class | - It is performed in two classes with inheritance |
| - Method Signature is different | - Method Signature + return type is same |
| - May or may not need inheritance | - Must need inheritance |
| - A user can easily overload it | - It is not possible for a user to override it |
| - Checked at compile line | - Checked at run time |
| - It is also known as the early binding | It is also known as the late binding |

**Method overloading:** Method overloading is a compile time polymorphism. A method called "print" that can accept an integer, a double or a string.

Method overriding: Method overriding is a run time polymorphism. For example: A "Vehicle" class may have a method called "drive" that is overridden in a "Car" subclass to provide a different implementation.

(b) In object oriented programming a constructor is a special method or function with a class that is automatically called when an object of that class is created. Its main purpose is to initialize the object's attributes and perform any object necessary to function correctly. The technique of having two (or more) constructors in a class is known as Constructor Overloading. A class can have multiple constructors that differ in the number or type of their parameters. It's not, however possible to have two constructors with the exact same parameters.

(a). Encapsulation: Binding Data and methods into a single unit.
Ex: A class with private variable and public getters/setters.

(d)

Inheritance: Acquiring properties from a parent class.

Polymorphism: One method behaves directly based on the object

Abstraction: Hiding implementation details.

2 (a) The access modifiers in java specifies the accessibility or scope of a field, method, constructor or a class. We can change the access level of field, method, constractor, and class by applying the access modifiers on it. There are four types of java access modifiers:

* private: The access level of private modifier is only within the class. It can not be accessed from outside of the class.

* Default: The access level of default modifier is only within the package. It can not be accessed from outside the package. If we do not specify any access level, it will be the default.

* protected: The access level of protected access modifier is within the package and outside the package through child class. If we do not make the child class, it can not be accessed from outside the package.

* public: the access level of a public access modifier is everywhere. You can access it from within the class, outside the class within the package, outside the package.

(b) corrected code:

```
public abstract Class Bike {
protected string Model {get; set;}
    protected abstract void Display();
}
public sealed class Honda : Bike {
    public int speed {get;} = 160
    protected override void display() {
        Console.WriteLine(Model);
    }
}
class program {
public static void Main() {
```

```
Bike bike = new Bike Honda();
bike.Display();
Console.WriteLine(((Honda)bike).speed);
}
```

(2) Write a java program to create a person class

```
class person {
    private String name;
    private int age;
    private char gender;
public person (String name, int age, char gender) {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }
public void displayDetails () {
    System.out.println ("Name: " + name);
    "      "      ("age: " + age);
    "      "      ("Gender: " + gender);
}
public static void main (String[] args) {
    person person = new person("John", 25, 'M');
    person.displayDetails();
}
}
```

3 [a] Correct the code to eliminate the compile time error:

→

```
class palindrome {
    public static void main(String[] args){
    String palindrome = "Rod saw I was Dor";
    int len = palindrome.length();
  StringBuilder dest = new StringBuilder();
  for (int i = len - 1 ; i >= 0; i--){
        dest.append(palindrome.charAt(i));
    }
  System.out.println(dest.toString());
}
```

[b] Define an interface and implement it:

→

```
interface Animal {
    void eat();
    void sleep();
}
class dog implements Animal {
    public void eat(){
  System.out.println("Dog eats bones");
}
public void sleep(){
    System.out.println("Dog sleeps");
}
}
```

```
class Cat implements Animal {
    public void eat() {
        System.out.println("Cat eats fish");
    }
    public void sleep() {
        System.out.println("Cat sleeps");
    }
}

Public class main {
    public static void main(String[] args) {
        Animal dog = new Dog();
        Animal cat = ney Cat();
        dog.eat();
        dog.sleep();
        Cat.eat();
        Cat.sleep();
    }
}
```

[e] Design a Book class with multiple constructors.

→

```
class Book {
    private String title;
    "       "   author;
    "       double price;

public Book (String title) {
        this.title = title;
    }
```

```
public Book (String title, String author) {
    this.title = title;
    this.author = author;
}
public Book (String title, String author, Double price) {
    this.title = title;
    this.author = author;
    this.price = price;
}

public void displayDetails () {
    System.out.println ("Title : " + title);
    "          ("author" + (author != null ?
                                author : "N/A"));
    "          "          ("price : " + (price > 0 ?
                                price : "N/A"));
}

public static void main (String [] args) {

    Book book1 = new Book ("Java Basics");
    Book book2 = new Book ("java", "Mim");
    Book book3 = new Book ("java", "mim", 1000);

    book1.DisplayDetails ();
    book2.DisplayDetails ();
    book3.DisplayDetails ();

}
}
```

④ [a] Difference between OOP and procedural programming:

→ OOP:

- organizes code into objects and classes.
- encapsulates data and behaviour.
- promotes reusability through inheritance and polymorphism.
- Example: Java, C++;

Code:

```
class car {
    String model;
    void drive() {
        System.out.println("Driving" + model);
    }
}
```

procedural programming:

- focuses on functions that operate on data.
- sequential execution of tasks.
- example: C, fortran

Code:

```
void drive(char * model){
    printf("Driving %.s", model);
}
```

[b] create a bank account class with encapsulation

→

```
class BankAccount {
    private string accountnumber;
    private double Balance;
    private string AccountHolder;
// getters
public String getAccountNumber(){
    return accountNumber;
}
```

```
public double getBalance() {
    return balance;
}
public String getAccountHolder() {
    return accountHolder;
}
//setters with validation
public void setAccountNumber(String accountNumber) {
    this.accountNumber = accountNumber;
}
public void setBalance(double balance) {
    this.balance = balance;
    if (balance >= 0) {
        this.balance = balance;
    } else {
        System.out.println("Balance cannot be negative");
    }
}
public void setAccountHolder(String accountHolder) {
    this.accountHolder = accountHolder;
}
}
```

[C] Friendly Function:

- A friendly function is a concept in c++ called a friend function that can access private and protected members of a class. but declare the outside of class.

Example:

```
class A {
    private: int x;
```

```
public:
    A(int value) { x(value) }
        friend void display (A obj);
};
void display (A obj) {
        cout << " value of obj =" << obj.x ;
}
int main () {
        A boj (10);
        display (obj); // x: 10
        return 0;
}
```

- Allows non-meber functions to access private and protected data.
- useful in operator overloading when the left operand is not an object of the class.
- Helps in situations where two classes need to closely collaborate by accessing each others private data.

(5) Role of JVM:

- the JVM is an integral part of the (JRE). It is responsible for executing Java programs by converting java bytecode into machine-specific instructions. JVM ensures that java program can run on any platform as long as the JVM is present.

- Class loader:
  - Responsible for loading java class files into memory during runtime.

- **Execution engine:**
  - converts bytecode into machine code.
- **Garbage collector:**
  - Automitacally manages memory by reclaiming unused objects.
- **platform independence:**
  - jvm abstracts platform-specific details.
  - converts bytecode into machine-specific instructions.

[b] Static variable and method:

→

- **Static variable:** shared across all object of a class.
- **Static method:** called without creating an instance.

**Example:**

```
class example{
    static int counter = 0;
    static void increment() {
        counter++;
    }
}
```

[c] Wrapper classes:

**purpose:**

- Allow handling of null values.
- Add additional methods to primitive type.

- Scenario:

```
Integer Sales = null;
Sales = 50;
System.out.println(Sales); // 50
```

(6) [a] Java package:

- A package is groups related classes and interfaces.
- Example:

```
package mypackage;
public class example {
    public static void main (String[], args){
        System.out.println("This is a package");
    }
}
```

[b] calculator with divide method:

```
class Calculator {
    public int divide (int a, int b){
        try {
            return a/b;
        } catch (ArithmeticException e){
            System.out.println("can not divide by zero");
            return 0;
        }
    }
```

```
public static void main (String[], args) {
    Calculator calc = new Calculator();
System.out.println (calc.divide (10,2)); // 5
    ;,
                    (calc.divide (10,0));
                    // can not divide by zero
}
```

## Random Access file:

- A Random access file is a type of file that allows you to read from or write to any position in the file directly, without reading through the file order. (Reads or write data at any position).

- difference between Random-access file and Sequential file:

| Feature | Random | Sequential |
|---|---|---|
| Access type | Any position directly | From beginning to end |
| speed | faster when specific record is needed | slower for large files |
| Flexibility | more flexible | less flexible |
| Use case | Databases, record system | Logs, reports, simple data storage |
| Example | Jump to 100th record in a file | Read 1st → 2nd → 3rd. - - - 100th. |

- Why need a random access file:
  - fast lookup:
  - modify specific Data.

- efficient for large files.
- supports fixed-size records.

(efficient updates, large data handling).

(7) [a] Scenario using threads:
- threads can perform tasks simultaneously.

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running");
    }
public static void main(String[] args) {
    MyThread t = new MyThread();
    t.start(); // calls run() method;
}}
```

[b] Base and derived classes:

```
class Vehicle {
    String make, model;
    int year;
    void displayInfo() {
        System.out.println("Make: "+make+", model
            +model+", Year: "+year);
    }
}
```

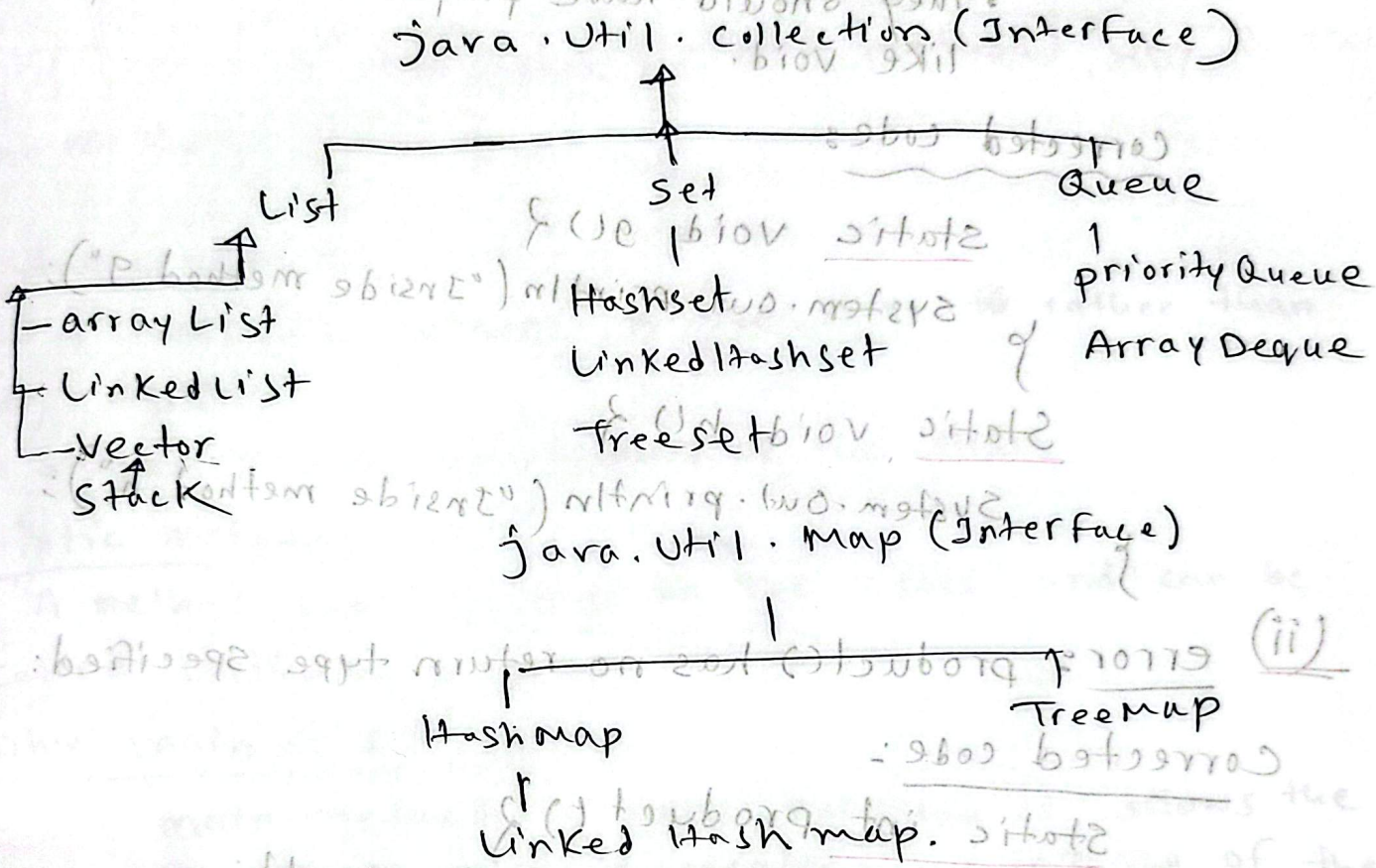
```
class Car extends Vehicle {
    int numberOfDoors;
    void displayInfo() {
        super.displayInfo();
        System.out.println("Number of Doors: " +
                           numberOfDoors);
    }
    public static void main(String[] args) {
        Car car = new Car();
        car.make = "Toyota";
        car.model = "Corolla";
        car.year = 2020;
        car.numberOfDoors = 4;
        car.displayInfo();
    }
}
```

(8) Ans: @. Collections class:

- Collections is a utility class in java.util package that provides static methods to work with collections like List, Set etc.
- Condition for using these methods:
  1. elements must be comparable:
     otherwise Java won't know how to compare them.
  2. No null elements allowed

(b) Collection framework:

- A java collection framework is a set of classes and interface that helps you store, manipulate and access data efficiently in the form of collections.




(c) Static Binding VS Dynamic Binding.

| Static | Dynamic |
|---|---|
| - Actual object is not used | - Actual object is used |
| - also known as early binding | - Also known as late binding |
| - Speed is high | - Speed is low |
| - Ex: method overloading | - Ex: method overriding |