Java Concepts for InfyTQ Qualifying Round for 2022 Batch by

TalentBattle

One Stop Platform for Placement Preparation

www.talentbattle.in

For Off-campus updates & Placement Preparation Join our Whatsapp group & Telegram channel here: https://bit.ly/3jyTDkg

InfyTQ DMBS Concepts PDF Notes

Complete Placement Preparatory LIVE MasterClass

Complete Placement Preparatory LIVE Masterclass

Preparing for the upcoming placement season? Battle your way through the crowd with the best placement preparation course online.

- This course is a complete end-to-end solution to your hurdles in placement preparation with a perfect blend of **LIVE** & Recorded Classes.
- Aptitude(Quantitative Aptitude, Logical Reasoning, Verbal Ability), Programming concepts, Coding Statement Preparation, Personal and Technical Interview preparation is completely covered in this course
- In this course, our experts are well known for their teaching methods which will help to understand Aptitude, Coding, Programming concepts in the easiest and quickest possible way from basic to advanced level.
- Company Specific Training Modules required for TCS, Wipro, Infosys, Accenture, Cognizant, Capgemini, Mindtree, Deloitte, Tech Mahindra, LTI, IBM, Atos-Syntel, Persistent, Hexaware, and many more companies are covered in this course according to the latest pattern.

More Details and Registration Link: https://bit.ly/39sgQSM

INDEX

Exam Pattern: 20 Questions

Topics:

Class Abstraction, Exception Handling, Flow Control, Queue, Deque, Array, Inheritance, Iterators, Access Specifiers.

Java is a powerful general-purpose programming language. It is used to develop desktop and mobile applications, big data processing, embedded systems, and so on. According to Oracle, the company that owns Java, Java runs on 3 billion devices worldwide, which makes Java one of the most popular programming languages.

About Java Programming

•**Platform independent** - We can write Java code in one platform (operating system) and run on another platform without any modification.

•**Object-oriented** - Java is an object-oriented language. This helps to make our Java code more flexible and reusable.

•**Speed** - Well optimized Java code is nearly as fast as lower-level languages like C++ and much faster than Python, PHP, etc.

Why Learn Java?

•Java is a platform-independent language. We can write Java code in one platform and run it in another platform

•Java is a general-purpose language with a wide range of applications. It's used for developing mobile and desktop applications, big data processing, embedded systems, and so on.

•Java is an object-oriented programming language. It helps in code reusability.

Main Features of JAVA

Java is a platform independent language

Compiler(javac) converts source code (.java file) to the byte code(.class file). As mentioned above, JVM executes the bytecode produced by compiler. This byte code can run on any platform such as Windows, Linux, Mac OS etc. Which means a program that is compiled on windows can run on Linux and vice-versa. Each operating system has different JVM, however the output they produce after execution of bytecode is same across all operating systems. That is why we call java as platform independent language.

Java is an Object Oriented language

Object oriented programming is a way of organizing programs as collection of objects, each of which represents an instance of a class.

4 main concepts of Object Oriented programming are:

1. **Abstraction**
2. **Encapsulation**
3. **Inheritance**
4. **Polymorphism**

Simple

Java is considered as one of simple language because it does not have complex features like Operator overloading, **Multiple inheritance**, pointers and Explicit memory allocation.

Robust Language

Robust means reliable. Java programming language is developed in a way that puts a lot of emphasis on early checking for possible errors, that's why java compiler is able to detect errors that are not easy to detect in other programming languages. The main features of java that makes it robust are garbage collection, Exception Handling and memory allocation.

Secure

We don't have pointers and we cannot access out of bound arrays (you get ArrayIndexOutOfBoundsException if you try to do so) in java. That's why several security flaws like stack corruption or buffer overflow is impossible to exploit in Java.

Java is distributed

Using java programming language we can create distributed applications. RMI(Remote Method Invocation) and EJB(Enterprise Java Beans) are used for creating distributed applications in java. In simple words: The java programs can be distributed on more than one systems that are connected to each other using internet connection. Objects on one JVM (java virtual machine) can execute procedures on a remote JVM.

Multithreading

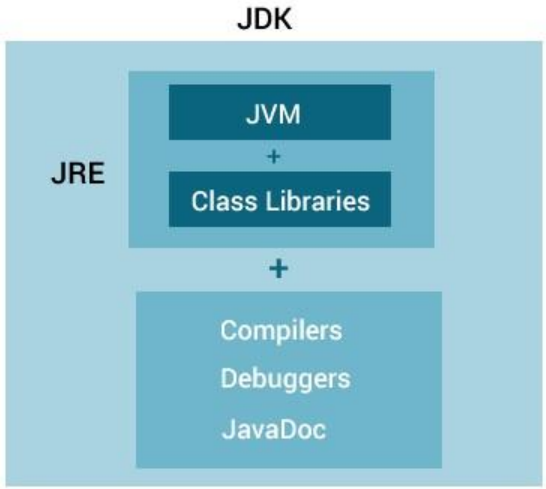
Java supports **multithreading**. Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU.

Portable

As discussed above, java code that is written on one machine can run on another machine. The platform independent byte code can be carried to any platform for execution that makes java code portable.

Java JDK, JRE and JVM

Working of Java Program



Relationship between JVM, JRE, and JDK

Java Flow Control

In computer programming, we use the if statement to control the flow of the program. For example, if a certain condition is met, then run a specific block of code. Otherwise, run another code.

For example assigning grades (A, B, C) based on percentage obtained by a student.

if the percentage is above 90, assign grade A
if the percentage is above 75, assign grade B
if the percentage is above 65, assign grade C

There are below mentioned forms of if...else statements in Java.

1. if statement
2. if...else statement
3. if...else if...else statement
4. Nested if...else statement

1. Java if (if-then) Statement

Condition is true

```
int number = 10;

if (number > 0) {
  // code
}

// code after if
```

Condition is false

```
int number = 10;

if (number < 0) {
  // code
}

// code after if
```

2. Java if...else (if-then-else) Statement

Condition is true

```
int number = 5;

if (number > 0) {
  // code
}
else {
  // code
}

// code after if...else
```

Condition is false

```
int number = 5;

if (number < 0) {
  // code
}
else {
  // code
}

// code after if...else
```

3. Java if...else...if Statement

1st Condition is true

```
int number = 2;
if (number > 0) {
  // code
}
else if (number == 0){
  // code
}
else {
  //code
}

//code after if
```

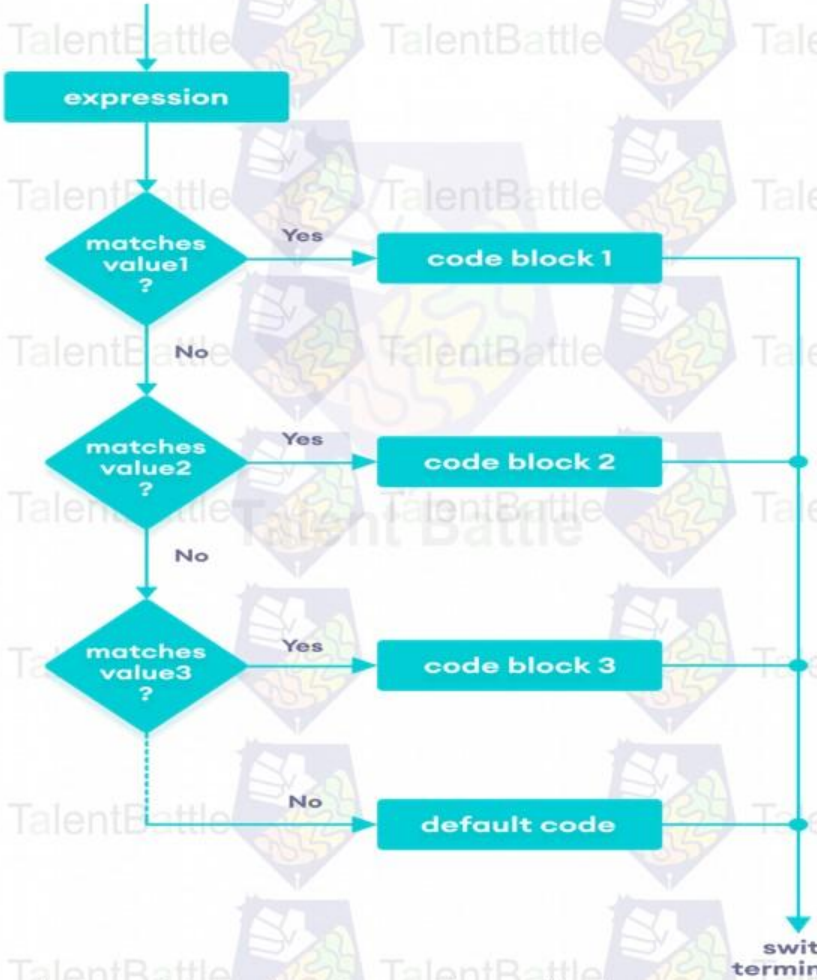
2nd Condition is true

```
int number = 0;
if (number > 0) {
  // code
}
else if (number == 0){
  // code
}
else {
  //code
}

//code after if
```

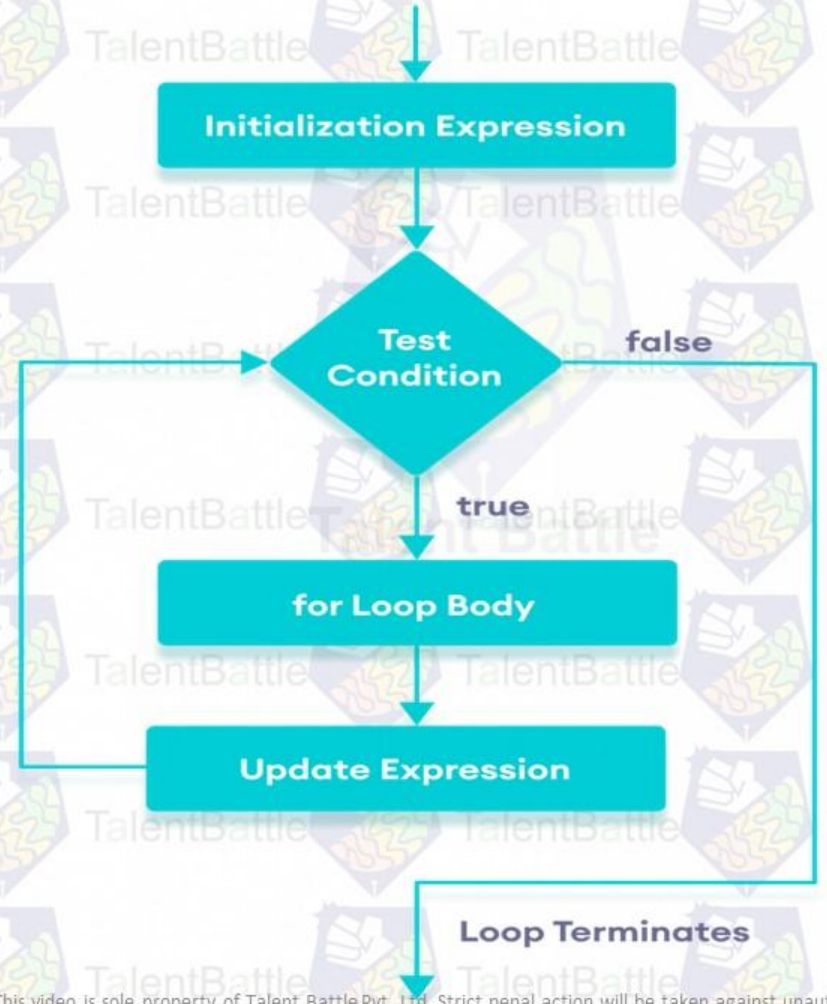
All Conditions are false

```
int number = -2;
if (number > 0) {
  // code
}
else if (number == 0){
  // code
}
else {
  //code
}

//code after if
```

Java switch Statement



Java for Loop



Java for-each Loop

In Java, the for-each loop is used to iterate through elements of arrays and collections (like ArrayList). It is also known as the enhanced for loop.

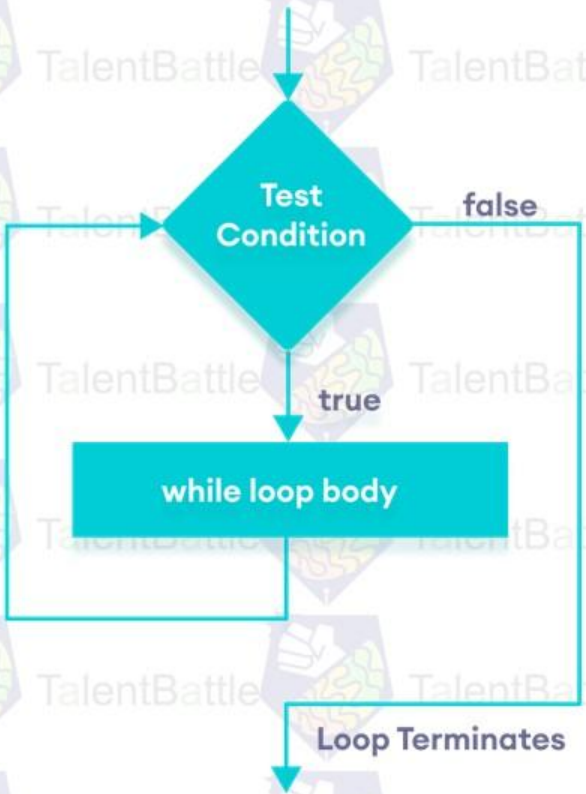
for-each Loop Syntax
The syntax of the Java for-each loop is:

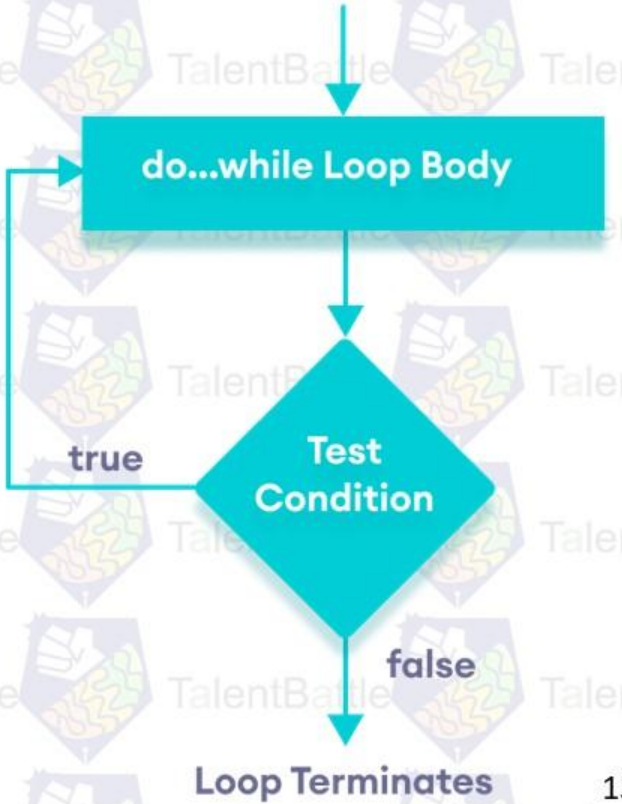
```
for(dataType item : array) {
  ...
}
```

Here,

array - an array or a collection
item - each item of array/collection is assigned to this variable
dataType - the data type of the array/collection

Java while and do...while Loop





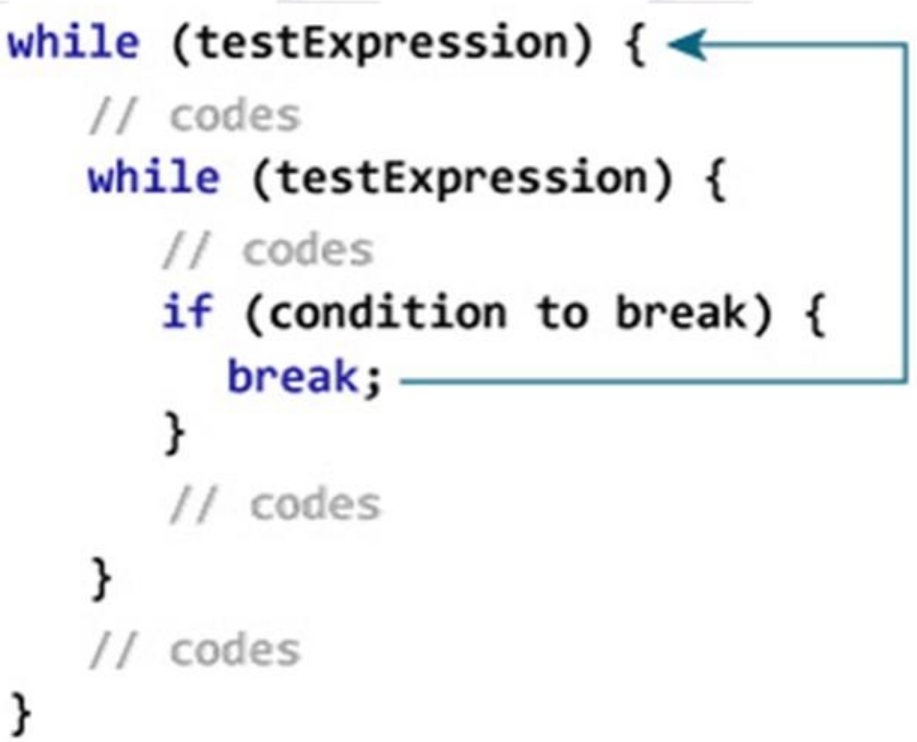
Java break Statement

```
while (testExpression) {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
```

```
do {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
while (testExpression);
```

```
for (init; testExpression; update) {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
```

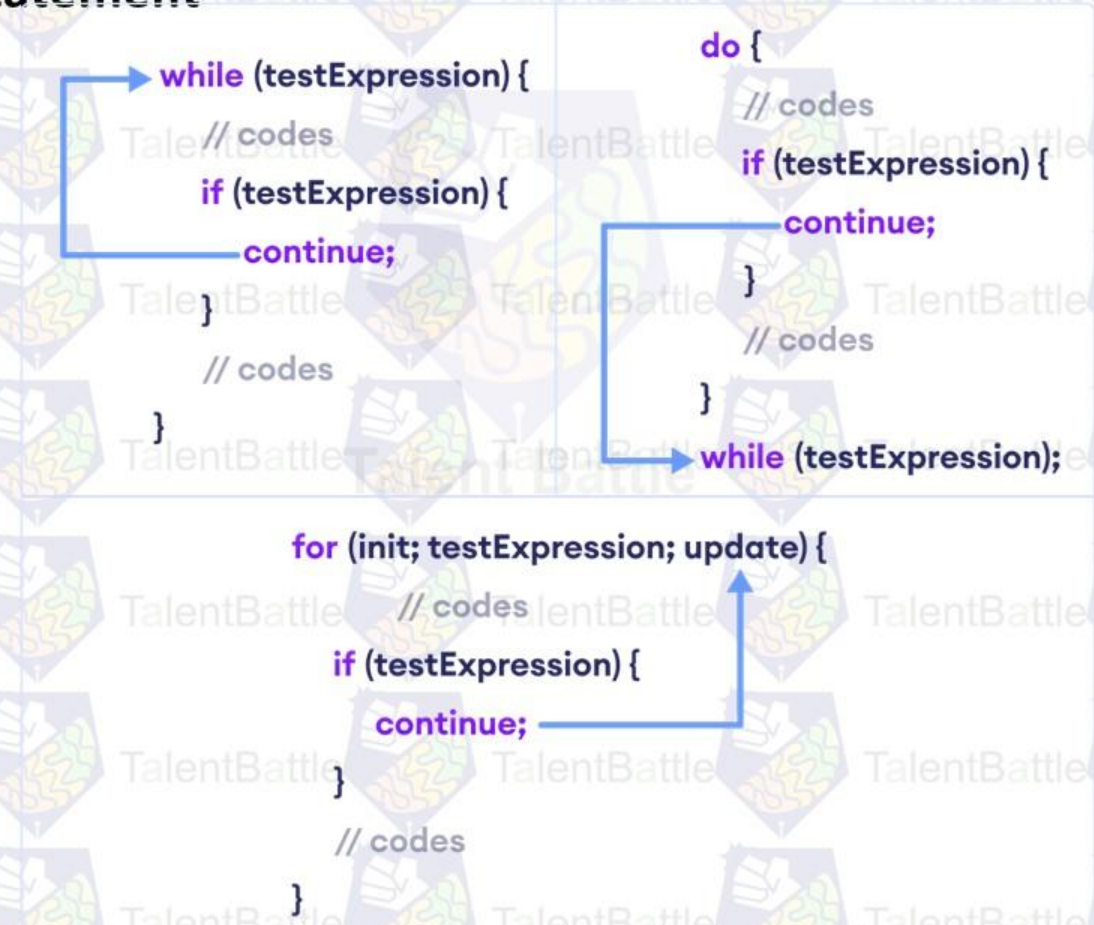
Java break and Nested Loop



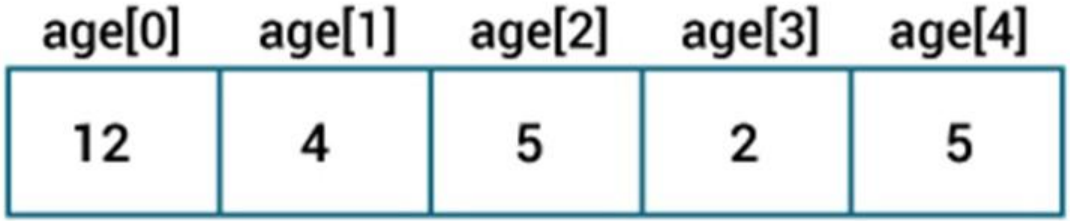
Labeled break Statement

```
label:
for (int; testExpresison, update) {
    // codes
    for (int; testExpression; update) {
        // codes
        if (condition to break) {
            break label;
        }
        // codes
    }
    // codes
}
```

Java continue Statement



Java Arrays



Note:

Array indices always start from 0. That is, the first element of an array is at index 0.
If the size of an array is n, then the last element of the array will be at index n-1.

An array is a collection of similar types of data.

For example, if we want to store the names of 100 people then we can create an array of the string type that can store 100 names.

String[] array = new String[100];
Here, the above array cannot store more than 100 names. The number of values in a Java array is always fixed.

Java Multidimensional Arrays

A multidimensional array is an array of arrays.
Each element of a multidimensional array is an array itself.

For example,

```
int[][] a = new int[3][4];
```

Here, we have created a multidimensional array named a. It is a 2-dimensional array, that can hold a maximum of 12 elements,

	Column 1	Column 2	Column 3	Column 4
Row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]

As we can see, each element of the multidimensional array is an array itself. And also, unlike C/C++, each row of the multidimensional array in Java can be of different lengths.

	Column 1	Column 2	Column 3	Column 4
Row 1	1 a[0][0]	2 a[0][1]	3 a[0][2]	
Row 2	4 a[1][0]	5 a[1][1]	6 a[1][2]	9 a[1][3]
Row 3	7 a[2][0]			

Java Class and Objects

Java is an object-oriented programming language. The core concept of the object-oriented approach is to break complex problems into smaller objects.

An object is any entity that has a state and behavior. For example, a bicycle is an object. It has

States: idle, first gear, etc

Behaviors: braking, accelerating, etc.

Java Class

A class is a blueprint for the object. Before we create an object, we first need to define the class.

We can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.

Since many houses can be made from the same description, we can create many objects from a class.

Create a class in Java

We can create a class in Java using the class keyword. For example,

```
class ClassName {
  // fields
  // methods
}
```

Here, fields (variables) and methods represent the state and behavior of the object respectively.

- fields are used to store data
- methods are used to perform some operations

Java Objects

An object is called an instance of a class. For example, suppose Bicycle is a class then MountainBicycle, SportsBicycle, TouringBicycle, etc can be considered as objects of the class.

Creating an Object in Java

Here is how we can create an object of a class.

```
className object = new className();

// for Bicycle class
Bicycle sportsBicycle = new Bicycle();

Bicycle touringBicycle = new Bicycle();
```

We have used the new keyword along with the constructor of the class to create an object. Constructors are similar to methods and have the same name as the class.

Java Methods

A method is a block of code that performs a specific task.

In Java, there are two types of methods:

User-defined Methods: We can create our own method based on our requirements.
Standard Library Methods: These are built-in methods in Java that are available to use.

The syntax to declare a method is:

```
returnType methodName() {
  // method body
}
```

Here,

returnType - It specifies what type of value a method returns For example if a method has an int return type then it returns an integer value.

If the method does not return a value, its return type is void.
methodName - It is an identifier that is used to refer to the particular method in a program.
method body - It includes the programming statements that are used to perform some tasks. The method body is enclosed inside the curly braces { }.

Java Access Modifiers

What are Access Modifiers?

In Java, access modifiers are used to set the accessibility (visibility) of classes, interfaces, variables, methods, constructors, data members, and the setter methods. For example,

```
class Animal {
   public void method1() {...}

  private void method2() {...}
}
```

In the above example, we have declared 2 methods: method1() and method2(). Here,

method1 is public - This means it can be accessed by other classes.
method2 is private - This means it can not be accessed by other classes.
Note the keyword public and private. These are access modifiers in Java. They are also known as visibility modifiers.

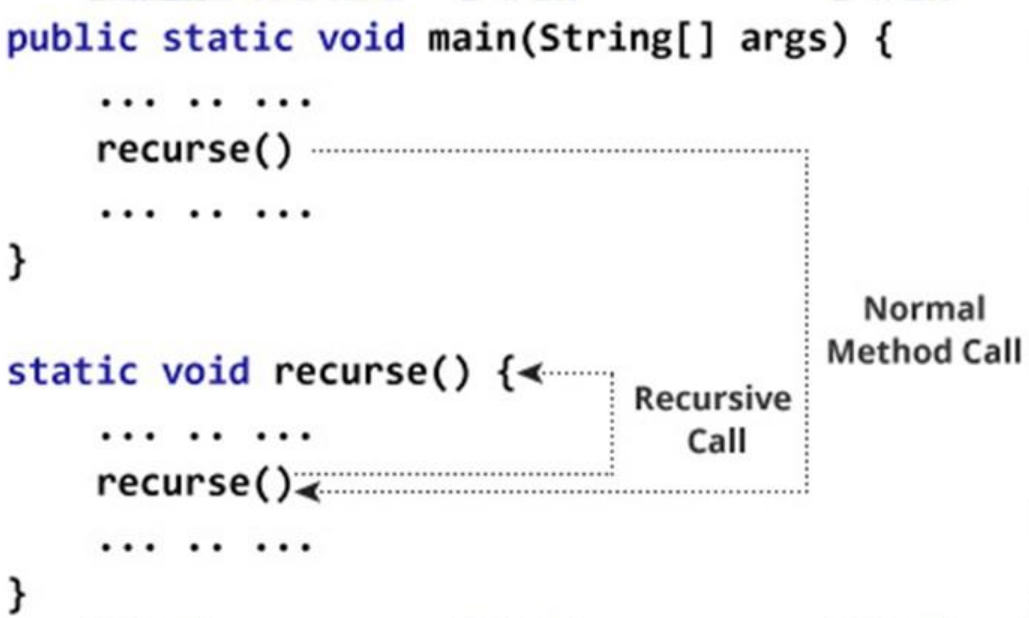
Note: You cannot set the access modifier of getters methods.

Types of Access Modifier

Modifier	Description
Default	declarations are visible only within the package (package private)
Private	declarations are visible within the class only
Protected	declarations are visible within the package or all subclasses
Public	declarations are visible everywhere

Java Recursion

In Java, a method that calls itself is known as a recursive method. And, this process is known as recursion.



Java Inheritance

The process by which one class acquires the properties(data members) and functionalities(methods) of another class is called inheritance. The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.

Child Class:

The class that extends the features of another class is known as child class, sub class or derived class.

Parent Class:

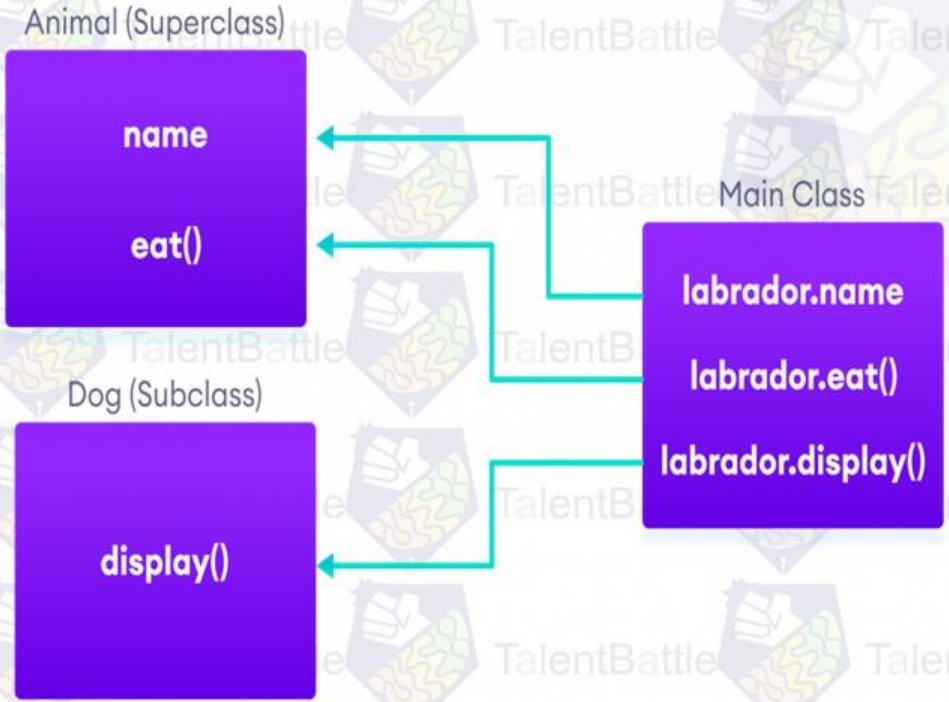
The class whose properties and functionalities are used(inherited) by another class is known as parent class, super class or Base class.

Inheritance is a process of defining a new class based on an existing class by extending its common data members and methods.

Inheritance allows us to reuse of code, it improves reusability in your java application.

Note: The biggest advantage of Inheritance is that the code that is already present in base class need not be rewritten in the child class.

This means that the data members(instance variables) and methods of the parent class can be used in the child class as.



is-a relationship

In Java, inheritance is an **is-a** relationship. That is, we use inheritance only if there exists an is-a relationship between two classes.

For example,

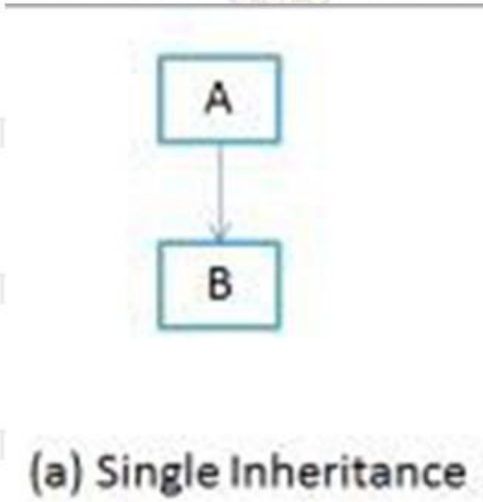
- **Car** is a **Vehicle**
- **Orange** is a **Fruit**
- **Surgeon** is a **Doctor**
- **Dog** is an **Animal**

Here, **Car** can inherit from **Vehicle**, **Orange** can inherit from **Fruit**, and so on.

Types of inheritance in Java: Single, Multiple, Multilevel, Hierarchical & Hybrid

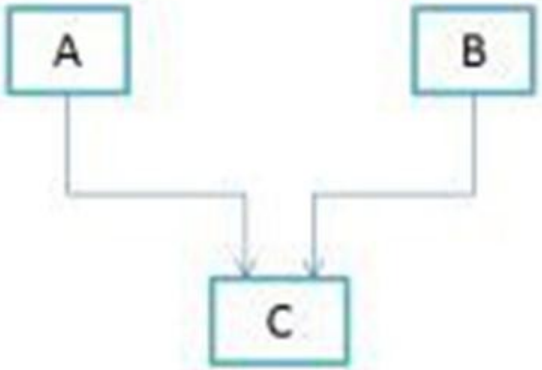
1) Single Inheritance

When a class extends another one class only then we call it a single inheritance.
The below flow diagram shows that class B extends only one class which is A.
Here A is a parent class of B and B would be a child class of A.



(a) Single Inheritance

2) Multiple Inheritance



(b) Multiple Inheritance

"**Multiple Inheritance**" refers to the concept of one class extending (Or inherits) more than one base class.

The inheritance we learnt earlier had the concept of one base class or parent.

The problem with "multiple inheritance" is that the derived class will have to manage the dependency on two base classes.

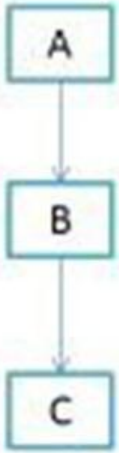
Note 1: Multiple Inheritance is very rarely used in software projects. Using Multiple inheritance often leads to problems in the hierarchy. This results in unwanted complexity when further extending the class.

Note 2: Most of the new OO languages like **Small Talk, Java, C# do not support Multiple inheritance**. Multiple Inheritance is supported in C++.

3) Multilevel Inheritance

Multilevel inheritance refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new class.

As you can see in below flow diagram C is subclass or child class of B and B is a child class of A.

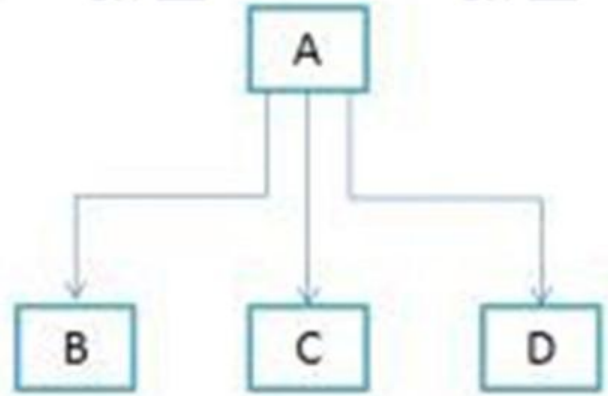


(d) Multilevel Inheritance

4) Hierarchical Inheritance

In such kind of inheritance one class is inherited by many **sub classes**.

In below example class B,C and D **inherits** the same class A. A is **parent class (or base class)** of B,C & D.

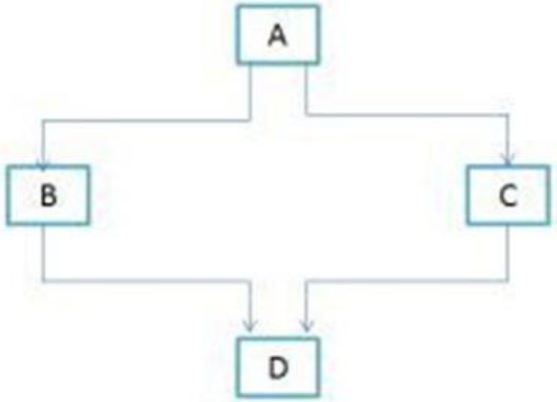


(c) Hierarchical Inheritance

5) Hybrid Inheritance

In simple terms you can say that Hybrid inheritance is a combination of **Single** and **Multiple inheritance.**

A hybrid inheritance can be achieved in the java in a same way as multiple inheritance can be!! Using interfaces. By using **interfaces** you can have multiple as well as **hybrid inheritance** in Java.



(e) Hybrid Inheritance

Java Abstract Class and Abstract Methods

Java Abstract Class

The abstract class in Java cannot be instantiated (we cannot create objects of abstract classes). We use the abstract keyword to declare an abstract class. For example,

```
// create an abstract class
abstract class Language {
  // fields and methods
}
```

An abstract class can have both the regular methods and abstract methods.
For example,

```
abstract class Language {

  // abstract method
  abstract void method1();

  // regular method
  void method2() {
    System.out.println("This is regular method");
  }
}
```

Java Abstract Method

A method that doesn't have its body is known as an abstract method. We use the same abstract keyword to create abstract methods.

For example,

abstract void display();

Here, display() is an abstract method. The body of display() is replaced by ; .

If a class contains an abstract method, then the class should be declared abstract. Otherwise, it will generate an error.

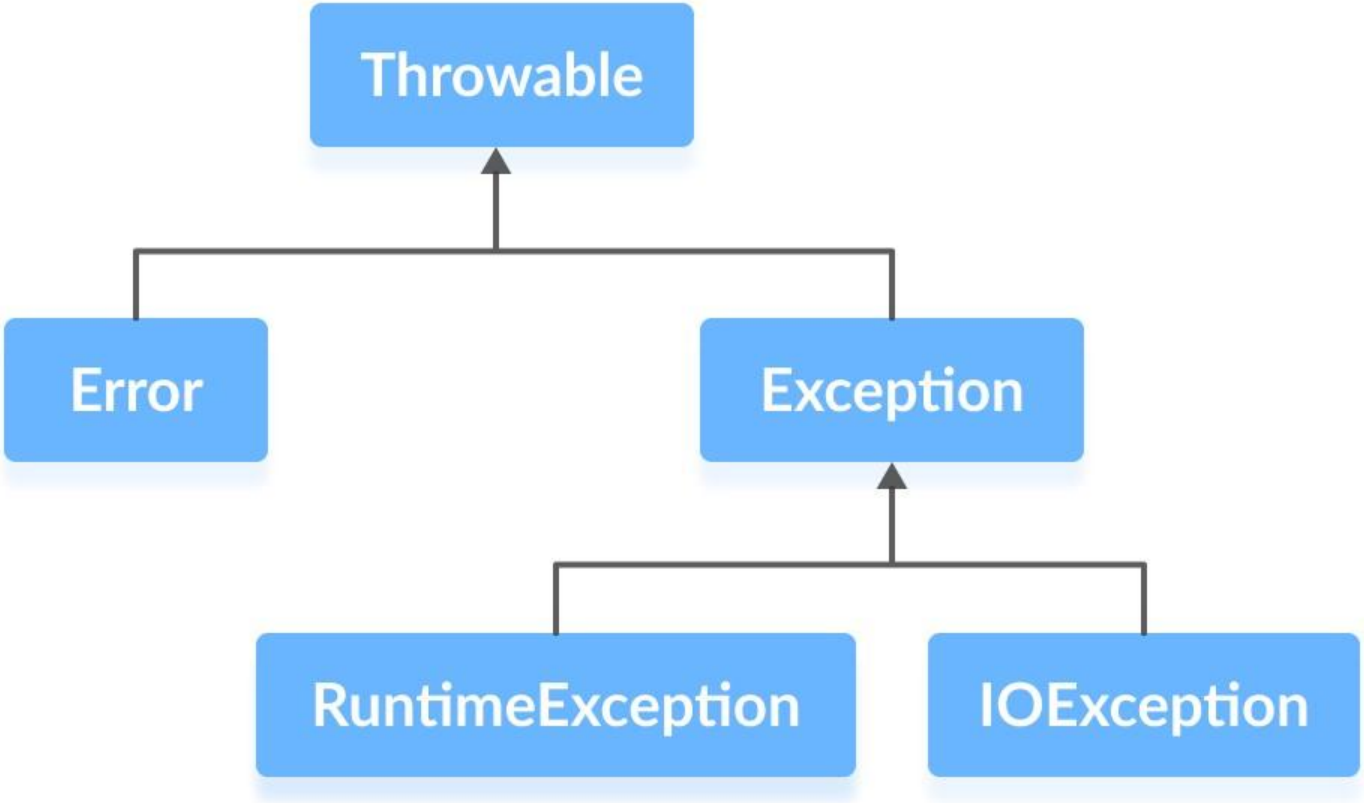
Java Exceptions

An exception is an unexpected event that occurs during program execution. It affects the flow of the program instructions which can cause the program to terminate abnormally.

An exception can occur for many reasons. Some of them are:

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out of disk memory)
- Code errors
- Opening an unavailable file

Java Exception hierarchy



Java Exception Types

The exception hierarchy also has two branches: RuntimeException and IOException.

1. RuntimeException

A runtime exception happens due to a programming error. They are also known as unchecked exceptions.
These exceptions are not checked at compile-time but run-time. Some of the common runtime exceptions are:

- Improper use of an API - IllegalArgumentException
- Null pointer access (missing the initialization of a variable) - NullPointerException
- Out-of-bounds array access - ArrayIndexOutOfBoundsException
- Dividing a number by 0 - ArithmeticException

2. IOException

An IOException is also known as a checked exception. They are checked by the compiler at the compile-time and the programmer is prompted to handle these exceptions.

Some of the examples of checked exceptions are:

- Trying to open a file that doesn't exist results in FileNotFoundException
- Trying to read past the end of a file

Java Exception Handling

Here's a list of different approaches to handle exceptions in Java.

- try...catch block
- finally block
- throw and throws keyword

1. Java try...catch block

The try-catch block is used to handle exceptions in Java. Here's the syntax of try...catch block:

```
try {
  // code
}
catch(Exception e) {
  // code
}
```

Here, we have placed the code that might generate an exception inside the try block. Every try block is followed by a catch block.

When an exception occurs, it is caught by the catch block. The catch block cannot be used without the try block.

2. Java finally block

In Java, the finally block is always executed no matter whether there is an exception or not.

The finally block is optional. And, for each try block, there can be only one finally block.

The basic syntax of finally block is:

```
try {
  //code
}
catch (ExceptionType1 e1) {
  // catch block
}
finally {
  // finally block always executes
}
```

If an exception occurs, the finally block is executed after the try...catch block. Otherwise, it is executed after the try block. For each try block, there can be only one finally block.

3. Java throw and throws keyword

The Java throw keyword is used to explicitly throw a single exception.

When we throw an exception, the flow of the program moves from the try block to the catch block.

Example: Exception handling using Java throw

```
class Main {
  public static void divideByZero() {

    // throw an exception
    throw new ArithmeticException("Trying to divide by 0");
  }

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

Output

```
Exception in thread "main" java.lang.ArithmeticException: Trying to divide by 0
        at Main.divideByZero(Main.java:5)
        at Main.main(Main.java:9)
```

Java try...catch

The try...catch block in Java is used to handle exceptions and prevents the abnormal termination of the program.

Here's the syntax of a try...catch block in Java.

```
try{
  // code
}
catch(exception) {
  // code
}
```

The try block includes the code that might generate an exception.

The catch block includes the code that is executed when there occurs an exception inside the try block.

Java throw and throws

In Java, exceptions can be categorized into two types:

Unchecked Exceptions: They are not checked at compile-time but at run-time.
For example: ArithmeticException, NullPointerException,
ArrayIndexOutOfBoundsException, exceptions under Error class, etc.

Checked Exceptions: They are checked at compile-time.
For example, IOException, InterruptedException, etc.

Usually, we don't need to handle unchecked exceptions. It's because unchecked exceptions occur due to programming errors. And, it is a good practice to correct them instead of handling them.

Java throws keyword

We use the throws keyword in the method declaration to declare the type of exceptions that might occur within it.

Its syntax is:

```
accessModifier returnType methodName() throws ExceptionType1, ExceptionType2 …
{
  // code
}
```

As you can see from the above syntax, we can use throws to declare multiple exceptions.

Java throw keyword

The throw keyword is used to explicitly throw a single exception.

When an exception is thrown, the flow of program execution transfers from the try block to the catch block. We use the throw keyword within a method.

Its syntax is:

throw throwableObject;

A throwable object is an instance of class Throwable or subclass of the Throwable class.

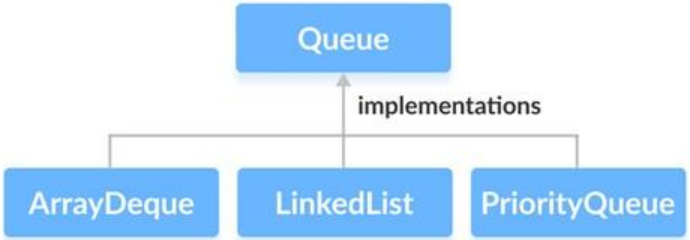
Java Queue Interface

The Queue interface of the Java collections framework provides the functionality of the queue data structure.
It extends the Collection interface.

Classes that Implement Queue
Since the Queue is an interface, we cannot provide the direct implementation of it.

In order to use the functionalities of Queue, we need to use classes that implement it:

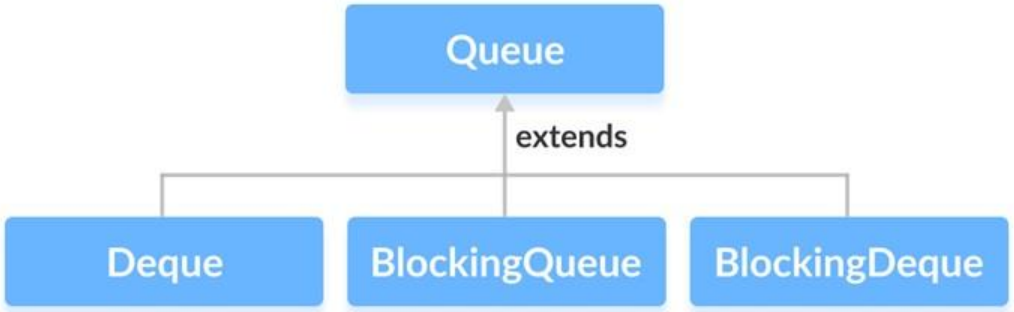
- ArrayDeque
- LinkedList
- PriorityQueue



Interfaces that extend Queue

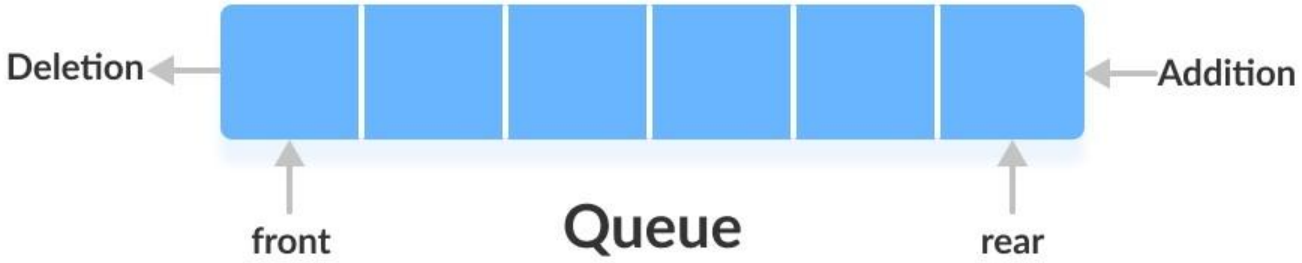
The Queue interface is also extended by various subinterfaces:

- Deque
- BlockingQueue
- BlockingDeque



Working of Queue Data Structure

In queues, elements are stored and accessed in **First In, First Out** manner.
That is, elements are **added from the behind** and **removed from the front**.



How to use Queue?

In Java, we must import java.util.Queue package in order to use Queue.

```
// LinkedList implementation of Queue
Queue<String> animal1 = new LinkedList<>();

// Array implementation of Queue
Queue<String> animal2 = new ArrayDeque<>();

// Priority Queue implementation of Queue
Queue<String> animal 3 = new PriorityQueue<>();
```

Here, we have created objects animal1, animal2 and animal3 of classes LinkedList, ArrayDeque and PriorityQueue respectively.
These objects can use the functionalities of the Queue interface.

Methods of Queue

The Queue interface includes all the methods of the Collection interface. It is because Collection is the super interface of Queue.

Some of the commonly used methods of the Queue interface are:

add() - Inserts the specified element into the queue. If the task is successful, add() returns true, if not it throws an exception.

offer() - Inserts the specified element into the queue. If the task is successful, offer() returns true, if not it returns false.

element() - Returns the head of the queue. Throws an exception if the queue is empty.

peek() - Returns the head of the queue. Returns null if the queue is empty.

remove() - Returns and removes the head of the queue. Throws an exception if the queue is empty.

poll() - Returns and removes the head of the queue. Returns null if the queue is empty.

Java Deque Interface

The Deque interface of the Java collections framework provides the functionality of a double-ended queue. It extends the Queue interface.

Working of Deque

In a regular queue, elements are added from the rear and removed from the front. However, in a deque, we can insert and remove elements from both front and rear.

How to use Deque?

In Java, we must import the java.util.Deque package to use Deque.

```
// Array implementation of Deque
Deque<String> animal1 = new ArrayDeque<>();

// LinkedList implementation of Deque
Deque<String> animal2 = new LinkedList<>();
```

Here, we have created objects animal1 and animal2 of classes ArrayDeque and LinkedList, respectively.
These objects can use the functionalities of the Deque interface.

Methods of Deque

Since Deque extends the Queue interface, it inherits all the methods of the Queue interface.

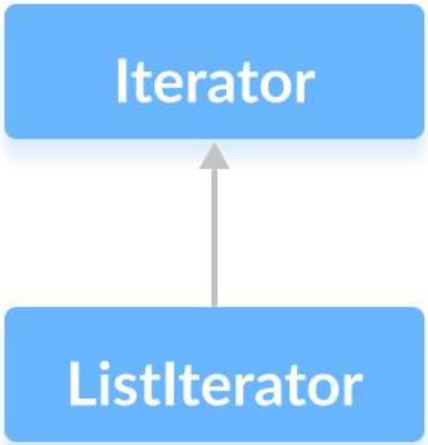
Besides methods available in the Queue interface, the Deque interface also includes the following methods:

addFirst() - Adds the specified element at the beginning of the deque. Throws an exception if the deque is full.
addLast() - Adds the specified element at the end of the deque. Throws an exception if the deque is full.
offerFirst() - Adds the specified element at the beginning of the deque. Returns false if the deque is full.
offerLast() - Adds the specified element at the end of the deque. Returns false if the deque is full.
getFirst() - Returns the first element of the deque. Throws an exception if the deque is empty.
getLast() - Returns the last element of the deque. Throws an exception if the deque is empty.
peekFirst() - Returns the first element of the deque. Returns null if the deque is empty.
peekLast() - Returns the last element of the deque. Returns null if the deque is empty.
removeFirst() - Returns and removes the first element of the deque. Throws an exception if the deque is empty.
removeLast() - Returns and removes the last element of the deque. Throws an exception if the deque is empty.
pollFirst() - Returns and removes the first element of the deque. Returns null if the deque is empty.
pollLast() - Returns and removes the last element of the deque. Returns null if the deque is empty.

Java Iterator Interface

The Iterator interface of the Java collections framework allows us to access elements of a collection.

It has a subinterface ListIterator.



All the Java collections include an iterator() method. This method returns an instance of iterator used to iterate over elements of collections.

Methods of Iterator

The Iterator interface provides 4 methods that can be used to perform various operations on elements of collections.

hasNext() - returns true if there exists an element in the collection

next() - returns the next element of the collection

remove() - removes the last element returned by the next()

forEachRemaining() - performs the specified action for each remaining element of the collection

THANK YOU!!!