
in
Java

Java is a general-purpose, class-based, object-oriented programming language. Java is platform-independent, object-oriented and secure. It is a high-level language which is first compiled into a binary byte-code and this byte-code is run on the Java Virtual Machine, a software-based interpreter. **Source code >> Java Compiler >> Byte Code >> Java Interpreter >> Native Code** To convert the java file to Java byte code using the Java Compiler, run

```
javac <filename.java>
```

This will produce a `.class` file which contains machine-readable instructions that can be executed by JVM. To execute the compiled file,

```
java <filename>
```

Contents	Contents	Contents
1. JDK	2. APIs	3. Data Types in Java
4. Identifiers	5. Scanner	6. Math Methods
7. Random Numbers	8. Conditionals: `if...else`	9. Switch Statements
10. Logical Operators	11. Loops: `for` and `while`	12. Arrays
13. String Methods	14. Wrapper Classes	15. ArrayLists
16. for-each loop	17. Methods	18. The `main()` method
19. Method Overloading	20. `printf` method	21. The `final` keyword
22. Objects	23. Constructors	24. Scope
25. Constructor Overloading	26. toString	27. Array of Objects
28. Static	29. Inheritance	30. Method Overriding
31. Super	32. Abstract Keyword	33. Access Modifiers
34. Encapsulation	35. Copying Objects	36. Interface
37. Polymorphism	38. Runtime Polymorphism	39. Exception Handling
40. File Class	41. FileWriter	42. FileReader
43. Nested Classes	44. Enumerations	45. Generics
46. Collections	47. Stack	48. Queue
49. Deque (Double-Ended Queue)	50. LinkedList	51. HashMap
52. Set	53. PriorityQueue	54. Abstract Window Toolkit (AWT)
55. Multithreading		

JDK

Java Development Kit (JDK) is a software development kit for Java. It contains the basic tools and libraries required in Java Programming. There are 7 main programs in JDK:

Programs	Description
javac	Java Compiler
java	Java Interpreter
javadoc	Creates Documentation in HTML
appletviewer	Java Interpreter to execute Java applets
jdb	Java Debugger for java programs
javap	Java Disassembler. Provides internal information about .class files
javah	Create interface between Java and C

APIs

Application Programming Interface in JDK enables developers to make various applets and applications. It contains 9 packages:

Packages	Description
java.applet	Applet programming (obsolete)
java.awt	Abstract Windowing Toolkit: GUI like Buttons, Menu etc.
java.io	File input/output handling
java.lang	Imported by default. Provides useful classes.
java.net	Network Programming classes
java.util	Built-In Data Structures like Vector, Stack, Dictionary, Hash etc.
javax.swing	Extension of java.awt used for designing GUI
java.sql	Database Connectivity

Data Types in Java

Primitive - 8 types

Type	Size	Type	Size
boolean	1 bit	char	16 bits
byte	8 bits	short	16 bits
int	32 bits	long	64 bits

float	32 bits	double	64 bits

Reference

String - a sequence of variable number of characters

- Primitive datatypes store a value, meanwhile reference datatypes store addresses.

Identifiers

Identifiers are names given to various program elements like variables, classes, constants, methods etc. It may contain letters, numbers and '_'. The first character cannot be a number. Identifiers are case sensitive.

```
public class Main {
    public static void main(String[] args) {
        byte x = 100;
        long y = 12345678901234L;
        int z = 123;
        float f = 12.345f;
        double d = 1.9001;
        boolean b = true;
        char symbol = '@';
    }
}
```

Scanner

To receive input from the user.

```
import java.util.Scanner;
---
Scanner sc = new Scanner(System.in);
String name = sc.next();
```

- If we call `nextInt()` and then `nextLine()`, the `\n` we entered after we entered the integer will be read by the scanner object and the nextLine() will exit at that '\n'. So no input will be read by the nextLine. Hence, we should use a `nextLine()` to clear this '\n' in between.

Math Methods

Function	Use
Math.max(a, b), Math.min(a, b)	Maximum and minimum of two numbers
Math.abs(a)	Absolute value
Math.sqrt(a)	Square root
Math.round(x), Math.floor(x), Math.ceil(x)	Rounding numbers
Math.pow(number, base)	number ^ base

Random Numbers

To use random numbers, we need to import the random class

```
import java.util.Random;
```

To generate (pseudo) random values, create an object of the class and do

```
Random random = new Random();
int x = Random.nextInt(<upperbound>);
```

Conditionals: `if...else`

```
if (condition1) {
    statements;
    ...
} else if (condition2) {
    statements;
    ...
} else {
    statements;
    ...
}
```

Switch Statements

Switch is a statement that allows a variable to be tested for equality against a list of values.

```
switch (condition) {
    case value1:
        statements;
        ...
        break;
    case value2:
        statements;
        ...
        break;
    ...
    default:
        statements;
        break;
}
```

Logical Operators

Logical operators are used to connect two or more expressions.

Operator	Name	Meaning
&&	AND	True if all conditions are true.
||	OR	True if atleast one of the conditions are true.
!	NOT	True if the condition is false.

Loops: `for` and `while`

1. `while` : executes a block of code as long as it's condition remains true

```
while (condition) {
    statements;
    ...
}
```

   - `do` : a variation of while loop which executes a block of code atleast once

```
 do {
     statements;
     ...
 } while (condition);
```

2. `for` : executes a block of code for a limited number of times

```
for (<from>; <condition>; <to>) {
    statements;
    ...
}
```

Arrays

Array is a finite collection of homogeneous data elements.

1. Declaration

```
<type> <arrayName>[];
        or
<type>[] <arrayName>;
```

2. Allocation of memory

```
<arrayName> = new <type> [<size>];
```

   Or we can do both by

```
<type> <name>[] = new <type> [<size>];
```

3. Load values in the array

```
<arrayName>[index] = value;
```

Or just do

```
    int x[] = {1, 2, 3, 4};
```

We can initialise 2D arrays like

```
int A[2][3] = {1, 2, 3, 4, 5, 6};
                or
int A[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

String Methods

Function	Use
str1.equals(str2), str1.equalsIgnoreCase(str2)	Compare two strings
str.length()	Length of the string - number of characters
str.charAt(index)	Character at a particular index.
str.indexOf(character)	Index of the the first occurence of a character in the string
str.isEmpty()	Checks if the string is empty
str.toUpperCase(), str.toLowerCase()	Changes the case
str.trim()	Removes the empty space at both ends
str.replace(oldChar, newChar)	Replace all occurences of one character with another

Wrapper Classes

Wrapper Classes provide a way to convert primitive data types into an object (reference datatype).

Primitive	Wrapper
int	Integer
boolean	Boolean
char	Character
double	Double

Strings are reference datatypes. Reference datatypes are naturally slower than primitive ones, but there are several useful methods associated with reference datatypes.

- **Autoboxing**: Automatic conversion of primitive datatype into the corresponding wrapper object by the compiler.

  ```
  Boolean b = true;
  ```

- **Unboxing**: Automatic conversion of wrapper object to its primitive datatype.

  ```
  // b is treated as a 'boolean' primitive datatype here
  if (b == true)
      return;
  ```

ArrayLists

ArrayList is **resizable array**. ArrayLists can only store reference datatypes. To declare an ArrayList,

```
import java.util.ArrayList;

ArrayList<datatype> name = new ArrayList<datatype>();
```

To add elements into the arraylist, do

```
name.add(item);
```

To retrieve elements, do

```
name.get(index);
```

Some important methods of ArrayLists are

- `name.set(index, element)` --- Insertion at a position
- `name.size()` --- Returns the length of the Arraylist
- `name.remove(index)` --- Deletion from a position
- `name.clear()` --- Clear the list

To declare a 2 Dimensional ArrayList,

```
ArrayList<ArrayList<datatype>> name = new ArrayList();
```

We can display the ArrayList by just doing

```
System.out.println(arraylist_name);
```

`for-each` loop

Also known as an advanced for loop, for-each loop is used to iterate through all the elements in a collection. for-each loops can be written in less steps and are more readable, but less flexible.

```
for (datatype item : iterable) {
    statements_using_item;
    ...
}
```

Methods

A method is a collection of statements grouped together to perform a certain task.

```
access_specifier return_type method_name(parameters...) {
    method_body;
}
```

- **Method Signature**: Part of the method declaration including **method name** and **parameter list**.

- **Access Specifier**: Specifies the access type/ visibility of the method.
    1. Public: The method is accessible by all classes
    2. Private: Accessible only in the classes in which the method is defined.
    3. Protected: The method is accessible within the same package or sub-classes in a different package
    4. Default: Visible only within the same package it was defined
- **Return Type**: The datatype of the value that the method returns.
- **Method Name**: A unique name used to identify a method.
- **Parameter List**: A list of parameters separated by commas, enclosed in a set of parentheses, containing the datatype and variable name of each parameter.
- **Method Body**: Actions to be performed on method call.

The `main()` method

The main method is the starting point for the JVM to execute the program. The syntax of the main() method is

```
         Keyword    Method Name
            ↓           ↓
    public static void main(String[] args)
      ↑             ↑               ↑
Access Specifier  Return type   Array of Strings
```

We use the `public` keyword so that the method is identifiable to the JVM. **Static Methods**, made using the keyword `static`, are methods which invokes without creating an object. The main method also accepts data from the user. It accepts a *string array*, which is used to hold the command line arguments. The `String[] args` parameter is used for this purpose.

Parameters	Arguments
Variables used in a function	Values passed to a function
Defined in the function declaration	Supplied during function call
Placeholders for Arguments	Actual values that are processed in the function

Method Overloading

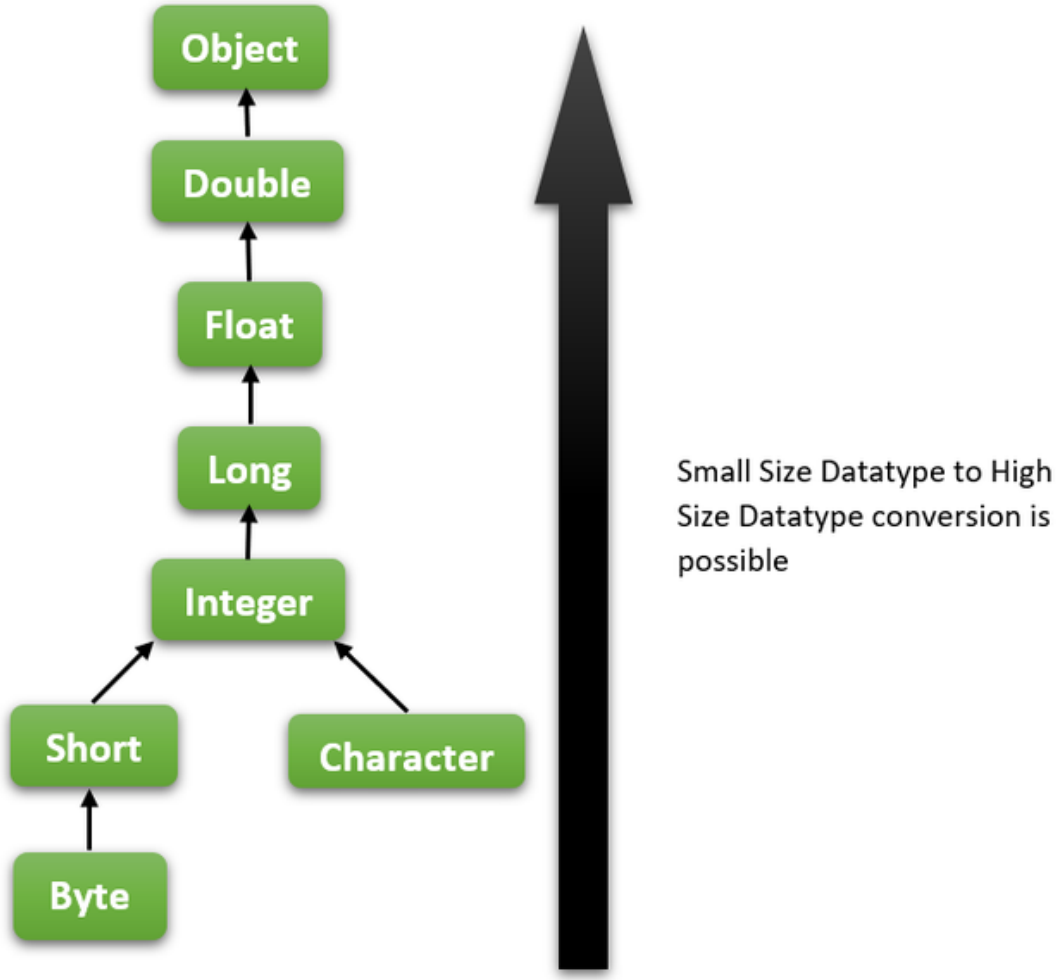
Overloaded Methods are methods with the same name, having different parameters. The methods must have different *Method Signatures*.

```
static int add(int x, int y) {
    return x + y;
}
static double add(int x, int y, int z) {
    return x + y + z;
}
```

In the functions

```
void f(int n) {
    ...
}
void f(float f) {
    ...
}
```

If we call the function `f` using an argument that is neither `int` nor `float`, `automatic type promotion` will be applied. Thus, if we pass a `character` as an argument, a the function with `int` argument will be called.



`printf` method

`printf()` is an optional method to control, format and display text. It takes two arguments:

- A format string
- An object, variable or value

The format string contains a **format specifier**, starting with a `%` which tells where the value should appear.

```
System.out.printf("This is a format string %[format-specifier]", value);
```

The format specifier formats the value and inserts it into the format string. The structure of the format specifier is

```
% [flags] [precision] [width] [conversion-character]
```

- The conversion character corresponds to the datatype of the value we are inserting into the string. e.g. d for integer, b for boolean, etc.

  ```
  System.out.printf("Boolean: %b, Character: %c, true, 'c');
  ```

- The width field field sets the **minimum** number of characters to be written as output.

  ```
  System.out.printf("Hello%10s", "world");
                            ↑
          Inserts atleast 10 characters at this position
  ```

  - If we insert a negative number as width, the text will become left justified.

- Precision sets the number of digits of precision when outputing floating point values.

  ```
  "%.2f" // Limit to two digits after the decimal point
  ```

- Flags add a particular effect to the output.

  1. -: left-justify
  2. +: include sign in the output for numerical values
  3. 0: add zero padding to numerics
  4. ,: comma grouping separator

The `final` keyword

Anything that is declared as **final** cannot be changed in the program.

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

Common practice is to make the variable name uppercase.

Objects

An `object` is an instance of a class. Any entity with a state or behavior could be an object. A class may contain attributes and methods. Objects can be physical or logical. A collection of objects / a blueprint from which we can create an object, is called a `class`. It is a logical entity

Constructors

Constructor is a special method that is called when a constructor is instantiated. A constructor can be created by making a function with the same name as that of the class.

```
class Human {

    String name;
    int age;

    Human(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

The current object can be referred inside the class using the `this` keyword.

Scope

A variable is said to be

- `local` if it declared inside a method. The variable is visible only to that method.
- `global` if it is declared outside a method, but within a class visible to all parts of a class.

Constructor Overloading

Constructor overloading is done by creating different constructors with the same name within a class, but with different parameters. Each constructor must have their own unique instances.

```
class Addition {
    Addition(int a, int b) {
        System.out.println(a + b);
    }
    Addition(int a, int b, int c) {
        System.out.println(a + b + c);
    }
}
```

toString

`toString()` is a special method that returns a string that textually represents an object. We use it implicitly while printing an object, or we can use it explicitly by calling `object.toString()`. By default, it returns the memory location in which the object is stored. We can `override` this method and display an object appropriately, by returning a string.

```
class Time {
    int hour;
    int minute;
    int second;
```

```
    public String toString() {
        return hour + ":" + minute + ":" + second;
    }

}
```

Array of Objects

Normally arrays are created as

```
int[] nums = new int[5];
```

To create an array of objects, just do

```
myObject[] name = new myObject[size];
```

Static

A static method / variable is a method / variable that **belongs to a class**, rather than an object.

```
class Planet {
    static String myPlanet;
    ...
}
```

We can call static methods/variables **without creating an object**.

```
System.out.println(Planet.myPlanet);
```

An example of a static method is

```
Math.round()
```

Inheritance

Inheritance is a mechanism in which one class acquires the properites and behaviours of a parent object. The class which inherits the other class is called a Sub Class or `Child Class`. The Super Class / `Parent Class` is the class from which a subclass inherits the features.

```
class Car extends Vehicle {
    ...
}
```

Method Overriding

Method overriding is the **specific implementation** of a method by the child class which has been declared by its parent class.

```
class Organism {
    void move() {
        System.out.println("Move");
    }
```

```
}
class Fish extends Organism {
    @Override
    void move() {
        System.out.println("Swim");
    }
}
```

The method in the child class is called the overriding method, and the one in the parent class is called the overriden method.

As a good practice, overriding method should have the `@Override` annotation.

Super

The `super` keyword refers to the superclass / parent class of an object. This is similar to `this`, which refers to the same class.

```
class Organism {
    String name;

    Organism(String name) {
        this.name = name;

    }
}

class Fish extends Organism {
    double weight;

    Fish(String name, double weight) {
        super(name);
        this.weight = weight;
    }
}
```

Abstract Keyword

The `abstract` keyword can be applied to both classes and methods. An `abstract class` cannot be instantiated, but its sub-classes can.

```
abstract class Vehicle {
    ...
}
class Car extends Vehicle {
    ...
}
```

An `abstract method` is declared without an implementation.

```
abstract void go();
```

This forces us to implement the method in one of its child classes.

Access Modifiers

Access Levels Modifier | Class | Package | Subclass | World --- | --- | --- | --- | --- public | Y | Y | Y | Y protected | Y | Y | Y | N no modifier | Y | Y | N | N private | Y | N | N | N

Without using a modifier, we cannot use the method in a different package by importing it.

A class defined without `public` is available only to classes within the same package.

A `protected method` is accessible to a different class in a different package as long as it is a sub-class of the class containing the protected member.

A `private method` is only visible to the class that it contains.

Encapsulation

Encapsulation is the process of wrapping code and data together into a single unit. Encapsulation means hiding the attributes of a class, by making it private or protected. These attributes can be accessed only by special methods called `getters` and `setters`.

```
class Person {
    private String name;

    // Getter
    public String getName() {
        return name;
    }
    // Setter
    public void setName(String name) {
        this.name = name;
    }
}
```

We can use setters in the constructor by

```
Person(String name) {
    this.setName(name);
}
```

Copying Objects

If we have two objects

```
Car car1;
Car car2;
```

and we want to copy one to another, we may do

```
car1 = car2;
```

which is wrong, since now both the names `car1` and `car2` are refering to the same object. Instead, we should create a copy method within our class.

```
public void copy(Car car) {

    this.setMake(car.getMake());
    this.setModel(car.getModel());

}
```

Now we can copy car2 to car1 by

```
car1.copy(car2);
```

Copy Constructors

We can create `copy constructors` to copy one object to another during the creation of an object, like

```
car2 = new Car(car1);
```

To do this, add another constructor (overloading) which does the copying at the creation.

```
Car(Car car) {
    this.copy(car);
}
```

Interface

An interface is a template that can be applied to a class. The template specifies what a class has / must do. This is similar to inheritance, but inheritance is limited to one super-class, meanwhile classes can apply more than one interface.

```
interface Prey {
    void flee();
}

public class Rabbit implements Prey {
    @Override
    public void flee() {
        System.out.println("I'm fleeing!");
    }
}
```

We can implement more than one interfaces to a class

```
interface Predator {
    void hunt();
}

# Fish can be prey or predator
public class Fish implements Predator, Prey {
    ...
}
```

Polymorphism

Polymorphism is a concept by which we can perform a single action in different ways. It can be seen as the ability of an object to be identified as more than type.

```
Car car = new Car();
Bicycle bicycle = new Bicycle();

Vehicle[] arr = {car, bicycle};

for (Vehicle v : arr) {
    v.drive();
}
```

- If one class inherits from another, an object of the child class can be seen as the same datatype as the parent class's object.
- All objects are children-classes of the `Object` class, hence they also identify as an Object.

  ```
  Object[] arr = {car, bicycle}
  ```

Runtime Polymorphism

`Runtime` / `Dynamic Polymorphism` is a process in which a call to an overridden method is resolved at runtime rather than compile time.

An example of doing this is by first creating a general object,

```
Language myLang;
```

and then changing its datatype on the runtime

```
if (choice == 1)
    myLang = new English();
else if (choice == 2)
    myLang = new Malayalam();
```

and each of these objects may have overriden methods, which can be invoked according to its datatype

```
myLang.greet();
```

Exception Handling

An `exception` is an unexpected event that occurs during the execution of a program which disrupts the normal flow of instructions. Example of exceptions include ArithmeticException, InputMismatchException etc. We need to gracefully handle these exceptions, using the `try` and `catch` blocks.

```
try {

    risky_statements;
```

```
} catch (ArithmeticException e) {

    statements;

} catch (Exception e) {

    statements;

} finally {

    statements_to_always_execute;

}
```

The `finally` block is executed whether or not an exception is catched.

- The `Exception` exception catches all exceptions.

File Class

A `File` is an abstract representation of a file and directory path names. We can import it by

```
File file = new File("path/to/filename");
```

- To check if a file exists, do `file.exists()`
- Do `file.getPath()` to get the path entered
- For the complete path in the system, run `file.getAbsolutePath()`
- `file.isFile()` checks if the selected object is indeed a file and not a folder
- To delete a file, do `file.delete()`

FileWriter

```
FileWriter writer = new FileWriter("path/to/file");
```

To write to a file, do

```
writer.write("Something to write");
```

Or to append, do

```
writer.append("Something to append");
```

FileReader

The FileReader object lets us read the contents of a file as a stream of characters.

```
FileReader reader = new FileReader("path/to/file");
```

`read()` returns an 'int' value which contains the byte value of the character currently being read. When read() returns `-1`, the file has no more content to read.

```
int data = reader.read();
while (data != -1) {
    System.out.print((char) data);
    data = reader.read();
}
reader.close();
```

Make sure to handle necessary exceptions like `FileNotFoundException` and `IOException`.

Nested Classes

A class which is declared inside another class is called a `nested class`. Nested classes are used to group classes that belong together, increase encapsulation and readability. There are four types of nested classes:

1. **Static nested classes**: A static nested class is a nested class that is declared with the static keyword. It is associated with the outer class, **but it does not have access to the instance variables and methods of the outer class**. It can be accessed using the outer class name followed by the nested class name, like `OuterClass.NestedClass`.
2. **Inner classes** (Non-static nested classes): A non-static nested class, also known as an inner class, is a nested class that is declared without the static keyword. It is associated with an instance of the outer class and has access to the instance variables and methods of the outer class. It can be accessed using an instance of the outer class.
3. Local classes: A local class is a nested class that is defined inside a block, such as a method or a loop. It is only accessible within the block where it is defined. Local classes can access the variables and parameters of the enclosing block, but they must be declared as final or effectively final.
4. Anonymous classes: An anonymous class is a nested class that does not have a name. It is defined and instantiated at the same time. Anonymous classes are often used to implement interfaces or extend classes on the fly without creating a separate named class.

```
class ParentClass {
    ...
    class InnerClass {
        ...
    }
    static class StaticNestedClass {
        ...
    }
}
```

Enumerations

An `enum` is a special datatype that allows a variable to be a set of predefined constants. The enum has a list of constants, and we can use these constants to create variables.

```
enum Level {
    LOW,
    MEDIUM,
    HIGH
}

Level myLevel = Level.MEDIUM;
```

The `values()` method returns an array containing all of the values of the enum.

```
for (Level l : Level.values()) {
    System.out.println(l);
}
```

Generics

`Generics allow types to be passed as parameters at compile time.`

We can create a **Generic Class** by

```
class Box<T> {
    private T t;
    void set(T t) {
        this.t = t;
    }
    T get() {
        return t;
    }
}

Box<Integer> myBox = new Box<Integer>();
myBox.set(12);
System.out.println(myBox.get());
```

Generics can also be applied to interfaces and methods.

Example of a **Generic Method** in Java:

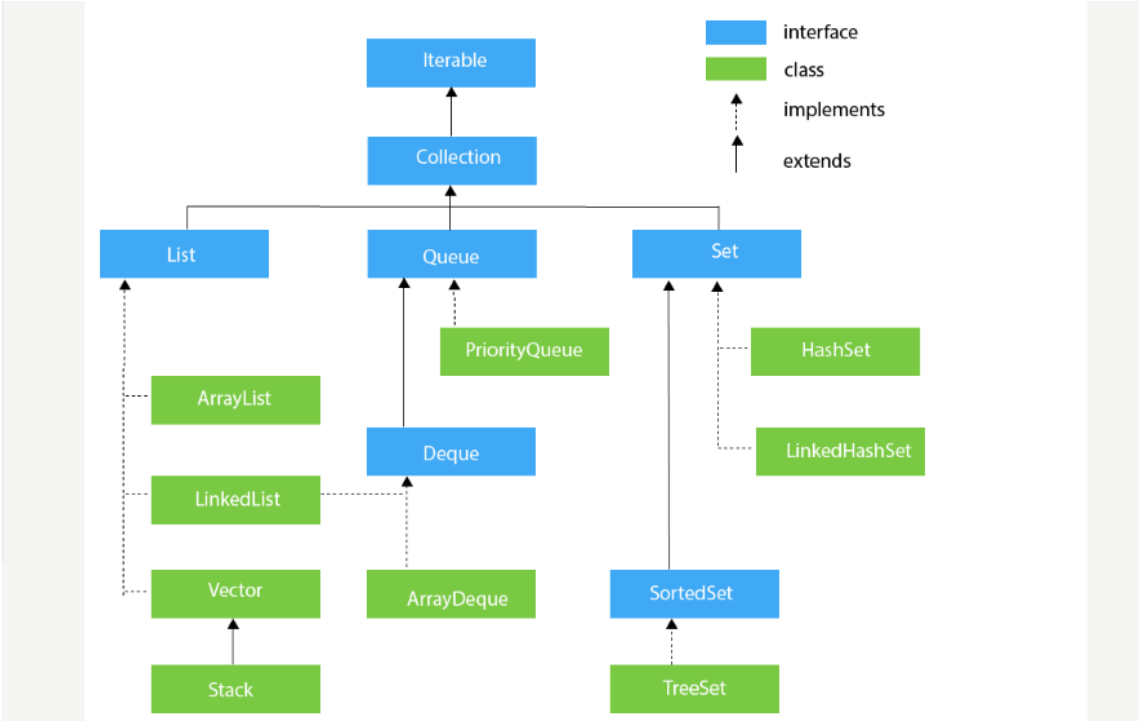
```
public static <E> void printArray(E[] inputArray) {
    for (E element : inputArray) {
        System.out.printf("%s ", element);
    }
    System.out.println();
}
```

- Generic methods can only be used with Reference Types (Integer, Character etc.).

Collections

The `Collections` class consists of static methods that operate on collections. Collections are used to store, retrieve, manipulate and communicate aggregate data. It contains polymorphic algorithms that operate on collections.

The hierarchy of **Collection framework** (readymade architecture) is as follows:



Method	Description
`sort(List<T> list)`	Sorts the specified list into ascending order, according to the natural ordering of its elements.
`reverse(List<?> list)`	Reverses the order of the elements in the specified list.
`shuffle(List<?> list)`	Randomly permutes the elements in the specified list.
`swap(List<?> list, int i, int j)`	Swaps the elements at the specified positions in the specified list.
`min(Collection<? extends T> coll)`	Returns the minimum element of the given collection, according to the natural ordering of its elements.
`max(Collection<? extends T> coll)`	Returns the maximum element of the given collection.
`rotate(List<?> list, int distance)`	Rotates the elements in the specified list by the specified distance.
`replaceAll(List<T> list, T oldVal, T newVal)`	Replaces all occurrences of one specified value in a list with another.
`frequency(Collection<?> c, Object o)`	Returns the number of elements in the specified collection equal to the specified object.
`fill(List<? super T> list, T obj)`	Replaces all of the elements of the specified list with the specified element.

```
Collections.sort(list);
```

Stack

A `Stack` is a collection of elements, with two main operations: `push` and `pop`. It follows the Last In First Out (LIFO) principle.

```
Stack<Integer> stack = new Stack<Integer>();
stack.push(1);  // Insertion
stack.pop();    // Deletion
stack.peek();   // Returns the top element
stack.empty();  // Returns true if the stack is empty
stack.search(1);// Returns the position of an element
```

Queue

A `Queue` is a collection of elements, with two main operations: `enqueue` and `dequeue`. It follows the First In First Out (FIFO) principle.

```
Queue<Integer> queue = new LinkedList<>();
queue.add(1);   // Enqueue
queue.remove(); // Dequeue
queue.poll();   // Dequeue
queue.peek();   // Front
queue.element();// Front
```

Deque (Double-Ended Queue)

A `Dequeue` is a double-ended queue, which allows insertion and deletion from both ends. It can be used as a stack or a queue.

```
Deque<Integer> deque = new ArrayDeque<>();
deque.push(0);          // Insert to the beginning
deque.addFirst(1);
deque.offerFirst(2);    // Returns true if success
deque.addLast(3);       // Insert to the end
deque.offerLast(4);     // Returns true if success
pop();                  // Removes the first element and returns it
removeFirst();          // Throws exception if empty
removeLast();
pollFirst();            // Returns null if deque empty
pollLast();
getFirst();             // Throws exception if empty
getLast();
peekFirst();            // Returns exception if empty
peekLast();
```

LinkedList

A `LinkedList` is a collection of elements, with each element having a reference to the next element in the list.

```
LinkedList<String> list = new LinkedList<String>();
list.add("A");          // Insertion at the end
list.add("B", 2);       // Insertion at a position
```

```
list.addFirst("Z");
list.addLast("Y");
list.get(2);            // Returns element at a position
list.getFirst();
list.getLast();
list.remove();          // Removes the head
list.remove(3);         // Deletion at a position
list.removeFirst();
list.removeLast();
list.size();
```

HashMap

The `HashMap` class is an implementation of `Map` interface, which is a collection of key-value pairs.

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("A", 1);
map.put("B", 2);
map.put("C", 3);
map.get("B");
map.remove("C");
map.containsKey("D");       // Returns true if a key is present
map.containsValue(4);       // Returns true if a value is present
map.keySet();               // Returns a Set of keys
map.entrySet();             // Returns a Set of entries
map.size();
map.isEmpty();
```

- `Hashtable` is a class in java which also stores key-value pairs.
- Hashtable provides thread safety and does not allow null keys / values
- HashMap is generally faster.

Set

A `Set` is a collection of unique elements.

```
Set<Integer> set = new HashSet<Integer>();
set.add(1);
set.contains(2);
set.remove(3);
set.size();
set.isEmpty();
set.clear();
set.iterator();     // Returns an iterator over the set
```

PriorityQueue

A `PriorityQueue` is a collection of elements, with the element with the highest priority being removed first.

```
PriorityQueue<Integer> pq = new PriorityQueue<Integer>();
pq.add(1);
pq.offer(1);    // Insertion
```

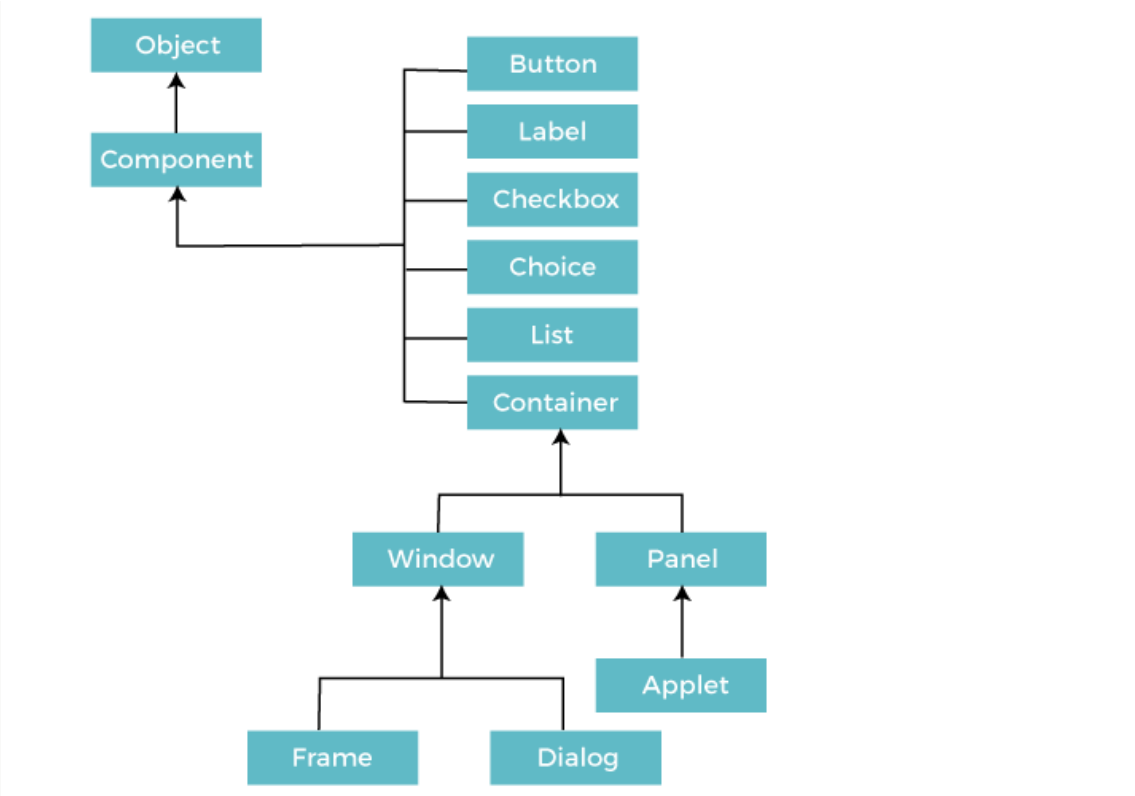
```
pq.peek();
pq.poll();      // Deletion
pq.remove();
pq.size();
pq.isEmpty();
```

Abstract Window Toolkit (AWT)

AWT is a platform-dependent API used to develop window-based applications in Java. It's part of the Java Foundation Classes, which is the standard API for providing a GUI for a Java program. AWT includes a set of native user interface components, a robust event-handling model, and graphics and imaging tools.

AWT components are platform-dependent, so they're shown by the operating system's view. AWT is also considered heavyweight, meaning its components use the operating system's resources. The Java.awt package contains classes for the AWT API, including **TextField, CheckBox, Choice, Label, TextArea, Radio Button, and List**.

The hierarchy of classes in AWT are:



Component	Description
Components	All elements like button, text fields, scroll bars etc.
Container	Elements that can contain other elements. Containers extend the `Container` class (eg. Frame, Dialog, Panel etc.).
Window	A top-level container with no borders and menu bar. Windows extend

	the `Window` class.
Panel	A generic container for holding other containers.
Frame	A top-level window with a title and border (optionally, menu bars). Frames extend the `Frame` class.
Dialog	A top-level window with a title and border, used for taking user input. Dialogs extend the `Dialog` class.
Button	A clickable button that can trigger an event. Buttons extend the `Button` class.
Label	A non-interactive text element. Labels extend the `Label` class.
TextField	A single-line text input field. TextFields extend the `TextField` class.
TextArea	A multi-line text input field. TextAreas extend the `TextArea` class.
CheckBox	A box that can be checked or unchecked. CheckBoxes extend the `Checkbox` class.
RadioButton	A circular button that can be selected or deselected, usually used in groups where only one can be selected at a time. RadioButtons extend the `Checkbox` class but are used with a `CheckboxGroup`.
Choice	A drop-down list of options. Choices extend the `Choice` class.
List	A list of items that the user can select. Lists extend the `List` class.
Scrollbar	A bar that can be moved up and down or left and right to navigate through content. Scrollbars extend the `Scrollbar` class.
MenuBar, Menu, and MenuItem	Components for creating a menu system. These extend the `MenuBar`, `Menu`, and `MenuItem` classes respectively.

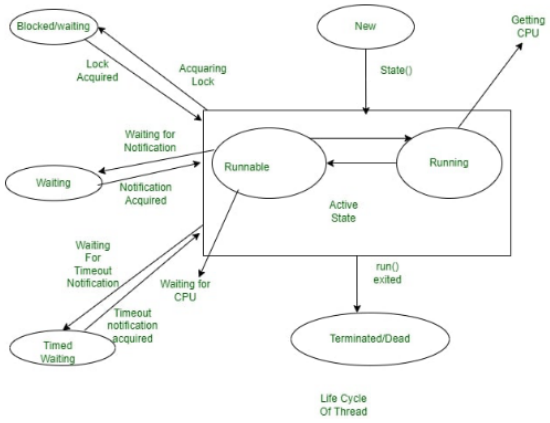
Useful methods in the Component Class include

```
public void add(Component c) // Inserts a component on this component
public void setSize(int width, int height) // Sets the size of the component
public void setLayout(LayoutManager m) // Defines the layout manager for the component
public void setVisible(boolean status) // Changes the visibility of a component (false by default)
```

Multithreading

Multithreading is a feature of modern programming languages that allows a single program to perform multiple tasks concurrently. A "thread" is a single sequence of execution in a program.

Every java program has one *Main Thread* provided by the JVM.



We can use multithreading by either extending the `Thread` class or implementing the `Runnable` interface.

```
// Define a class that extends the Thread class
class MyThread extends Thread {
    // Override the run() method
    public void run() {
        ...
    }
}
public class Main {
    public static void main(String[] args) {
        // Create an instance of the class
        MyThread t = new MyThread();
        // Call the start() method to start the thread.
        t.start();
    }
}
```

```
class MyRunnable implements Runnable {
    public void run() {
        ...
    }
}
public class Main {
    public static void main(String[] args) {
        MyRunnable r = new MyRunnable();
        Thread t = new Thread(r);
        t.start();
    }
}
```

If multiple threads are accessing and modifying the same data, you need to ensure that they do so in a way that doesn't cause inconsistencies or other issues. This is known as "thread-safety".

We can use the `isAlive()` method to check whether a thread has finished running.