



## **M23DE0202 – Object Oriented Programming using JAVA**

**II Semester MCA Academic Year : 2024 - 2025**

**School of Computer Science and Applications**

**Pinaka Pani.R**  
**Assitant Professor**





# Unit II

Object Oriented Programming using JAVA

Lecture – 1

Unit Lecture – 2.1

# Unit-II

## Introduction to Polymorphism, Inheritance, String Handling, Collections

Method Overloading, Inheritance, Method Overriding, String Handling, Wrapper Class, Input/Output Java Streams

**Collections:** Collections Overview, The Collection Interfaces, The Collection Classes. Lambda Expressions, Java Memory Management.



# Method Overloading in Java

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- If we have to perform only one operation, having same name of the methods increases the readability of the program.

## Advantage of method overloading

- Method overloading *increases the readability of the program*.



# Different ways to overload the method

There are two ways to overload the method in java

- By changing number of arguments.
- By changing the data type.

**In java, Method Overloading is not possible by changing the return type of the method only.**



## Method overloading

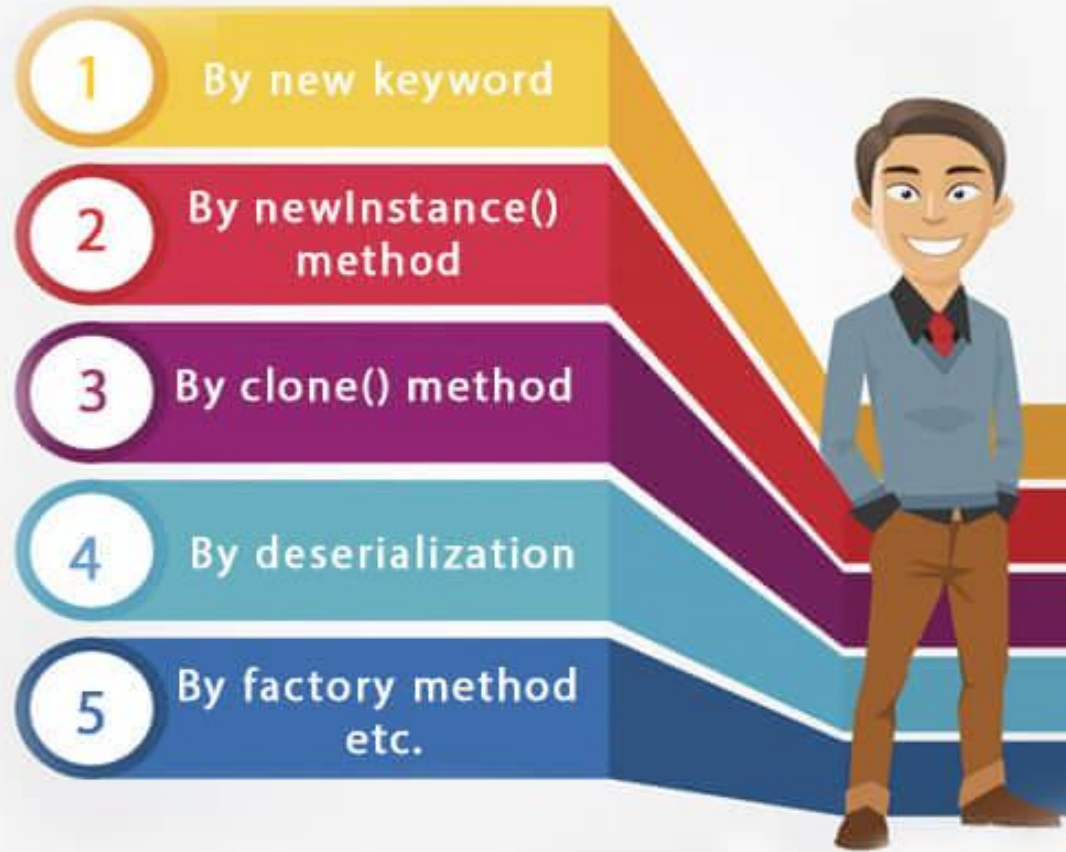
- Why Method Overloading is not possible by changing the return type of method only?
  - method overloading is not possible by changing the return type of the method only because of **ambiguity**.
  - It considered **method name + parameter list** only but not return type.

```
int show() {  
    return 100;  
}  
  
double show() {  
    return 10.5;  
}
```

```
int show(int x) {  
    return x;  
}  
  
double show(double x) {  
    return x;  
}
```

# Creation of object in JAVA

Different ways to create an object in Java



# Object Creation in Java:

- **Using new keyword**

```
Student s1 = new Student();
```

- **Using Reflection (Class.forName)**

```
Student s2 = (Student) Class.forName("Student").newInstance();
```

- **Using Cloning (clone() method)**

```
Student s3 = (Student) s1.clone();
```



## Using Deserialization:

- Object created from byte stream.
- Deserialization is the process of converting a byte stream back into a Java object.
- It's commonly used in saving/loading objects from files, sockets, or networks.
- **Steps to Create Object Using Deserialization:**
  - Reading objects from a **file or network**.
  - Restoring objects that were **saved earlier**.
  - Transferring objects between systems (e.g., **Java RMI, Web Services**, etc.).



# Quiz for this class

What is the return type of a method that does not return any value?

- A. int
- B. float
- C. void
- D. double

void

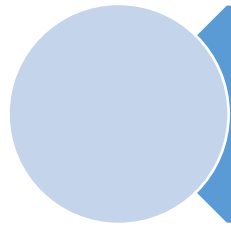
2. What is the process of defining more than one method in a class differentiated by method signature?

- A. Function overriding
- B. Function overloading
- C. Function doubling
- D. None of the mentioned

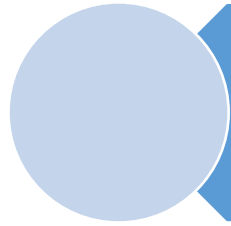
Function overloading



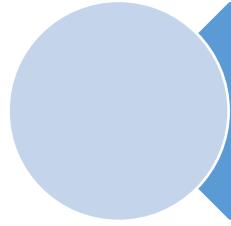
# Topics to be learned from this class



Inheritance in JAVA



Inheritance (IS-A)



Aggregation (HAS – A)



# Inheritance

- **Inheritance** is one of the key features of OOP that allows us to **create a new class from an existing class**.
- The new class that is created is known as **subclass** (child or derived class) and the existing class from where the child class is derived is known as **superclass** (parent or base class).
- The **extends** keyword is used to perform inheritance in Java.



# Inheritance in java

- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object..
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

## Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability



---

Inheritance is an **is-a** relationship.

For example,

**Car is a Vehicle**

**Orange is a Fruit**

**Surgeon is a Doctor**

**Dog is an Animal**

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



## Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.



```
class Animal {  
    // methods and fields  
}  
// use of extends keyword  
// to perform inheritance  
class Dog extends Animal {  
    // methods and fields of Animal  
    // methods and fields of Dog  
}
```



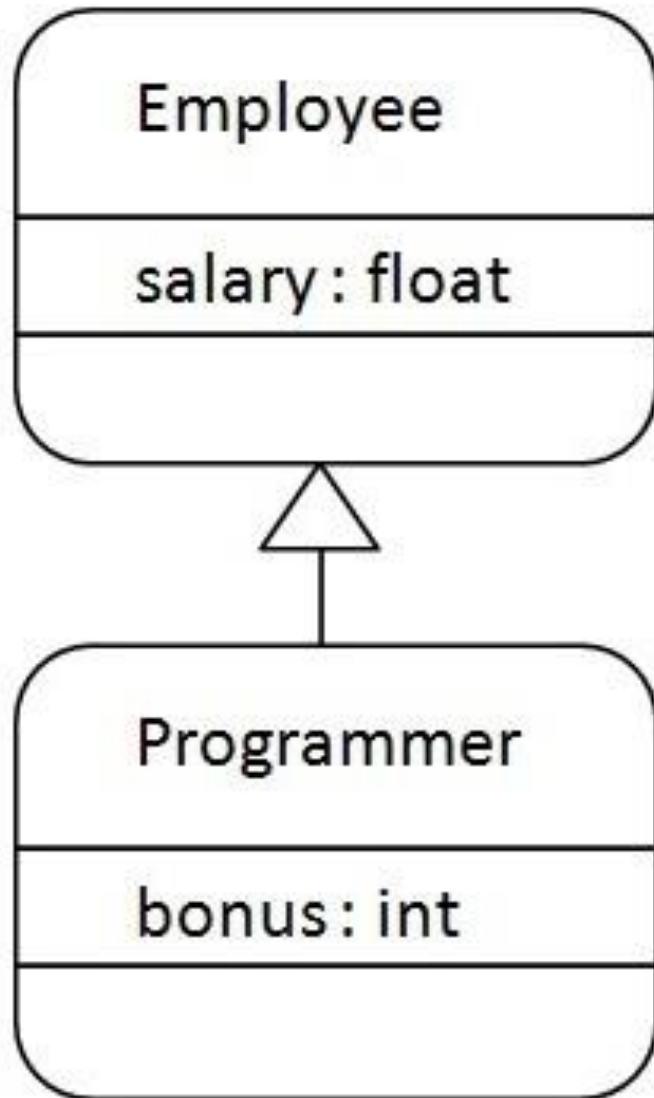
## Inheritance:

The Dog class is created by inheriting the methods and fields from the Animal class.

Dog is the subclass and Animal is the superclass.



# Java Inheritance Example

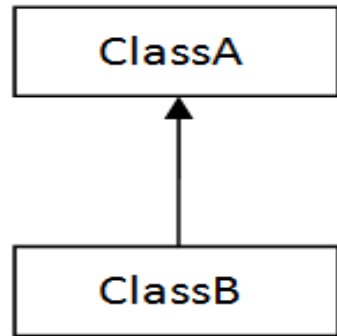


```
class Employee{
    float salary=40000;
}
```

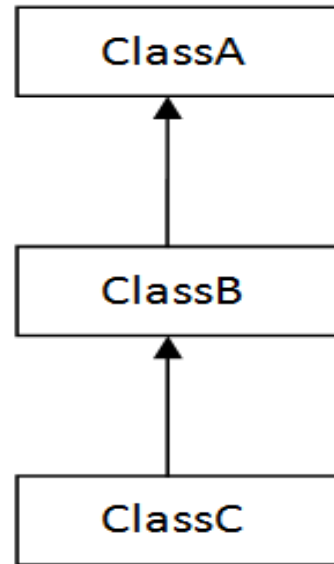
```
class Programmer extends Employee{
    int bonus=10000;
    public static void main(String args[]){
        Programmer p=new Programmer();
        System.out.println("Programmer salary is:"+p.salary);
        System.out.println("Bonus of Programmer is:"+p.bonus);
    }
}
```



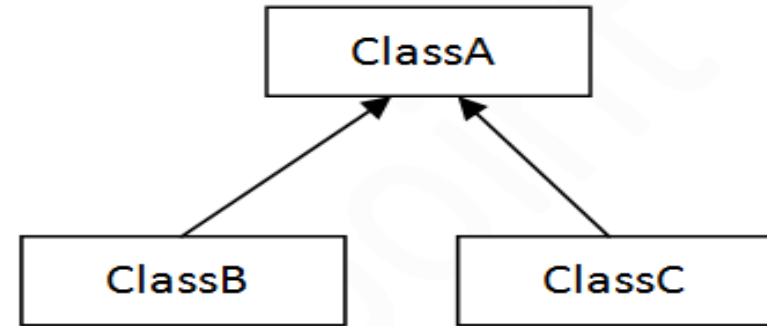
# Types of inheritance in java



1) Single



2) Multilevel



3) Hierarchical

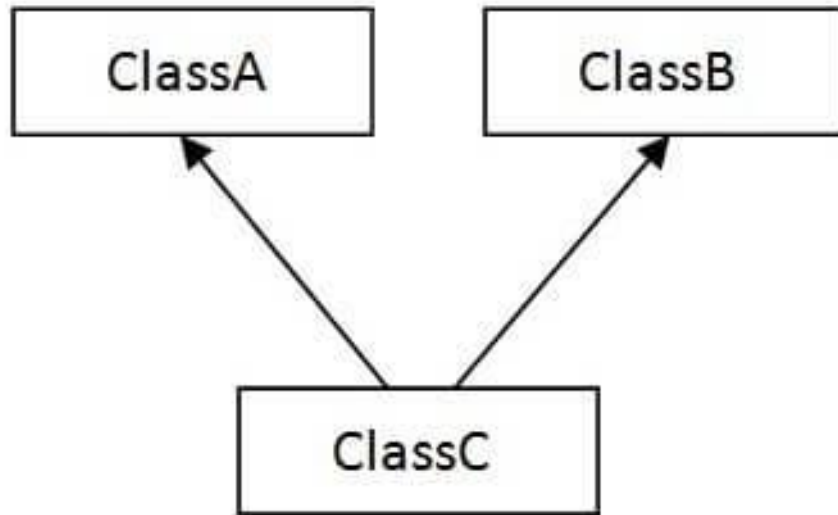
[Single Inheritance](#)

[Multilevel Inheritance](#)

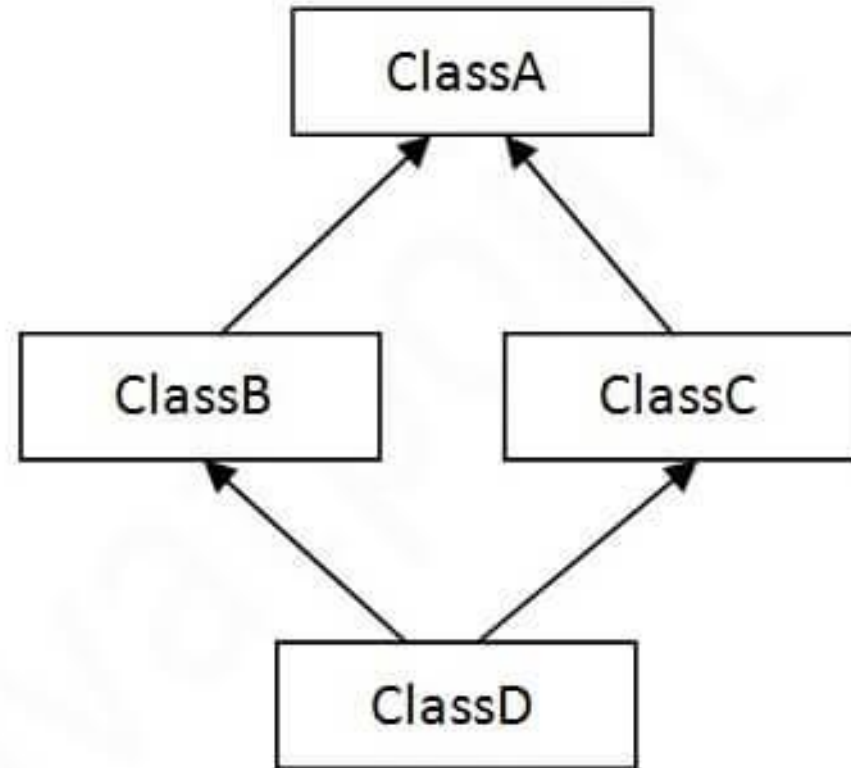
[Hierarchical](#)



# Types of inheritance in java



4) Multiple



5) Hybrid

# Method overriding in Java Inheritance

**If the same method is present in both the superclass and subclass, what will happen?**

- The method in the subclass overrides the method in the superclass. This concept is known as method overriding in Java.

## **The annotation - @Override**

- Annotation specifies that the method that has been marked with this annotation overrides the method of the superclass with the same method name, return type, and parameter list.
- It is not mandatory to use @Override when overriding a method.
- If we use it, the compiler gives an error if something is wrong (such as wrong parameter type) while overriding the method.



# Overriding Methods

- If sub class has the same method as declared in the super class then it is known as **method overriding**.
- The method in a subclass should have
  - **same name as a method** in its super-class,
  - **same parameters or signature as** a method in its super-class
  - **same return type** as a method in its super-class,
- Method overriding is possible only through **inheritance**
- Method overriding is used for **runtime polymorphism**



# RULES for Overriding Methods

- Both the super class and the subclass must have the **same method name, the same return type and the same parameter list.**
- We cannot override the method declared as **final and static.**
- **Private** methods can not be overridden
- We should always **override abstract methods** of the superclass



# Tips on Method Overriding

- Can we override static method?
  - **No, a static method cannot be overridden.**
- Why we cannot override static method?
  - **It is because the static method is bound with class whereas instance method is bound with an object.**
  - **Static belongs to the class area, and an instance belongs to the heap area.**
- Can we override java main method?
  - **No, because the main is a static method.**

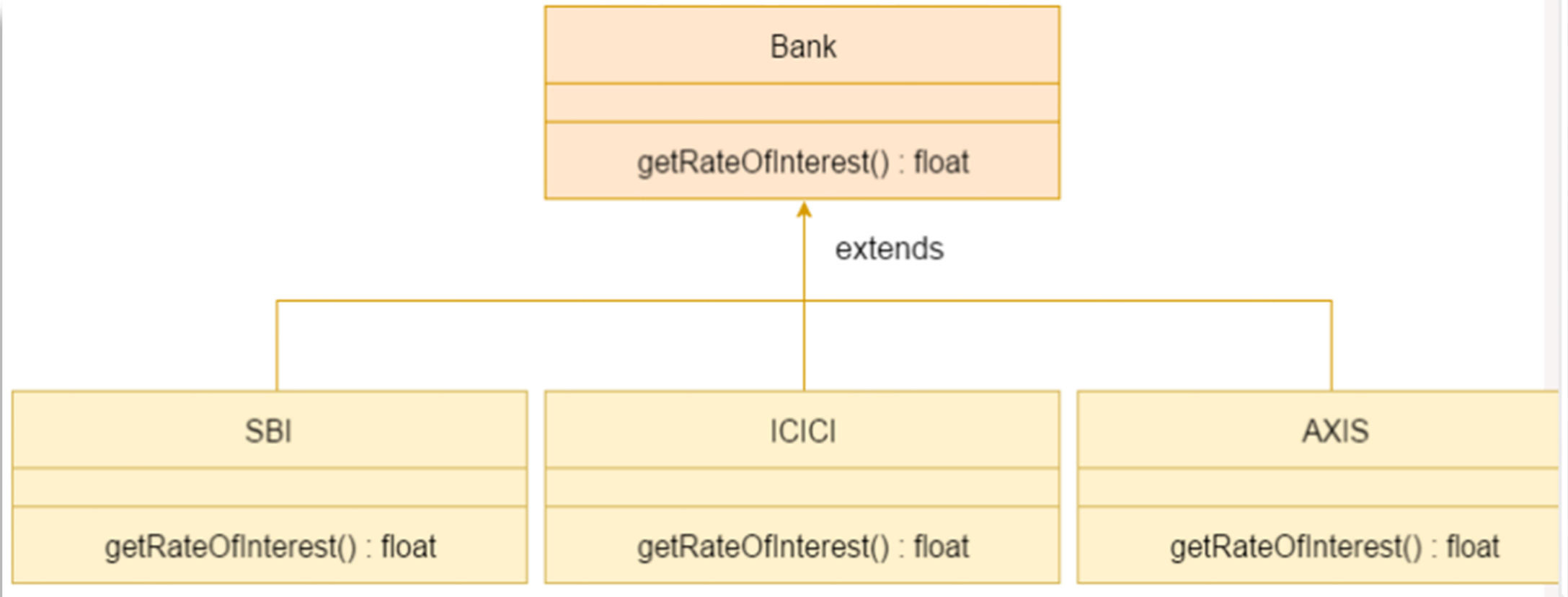


# Method Overriding

```
class Vehicle{
    void run(){
        System.out.println("Vehicle is running");
    }
}
class Bike extends Vehicle{
    void run(){
        System.out.println("Bike is running safely");
    }
    public static void main(String args[]){
        Bike b = new Bike();
        b.run();
    }
}
```



# Method Overriding



- **Why multiple inheritance is not supported in java?**
  - **To reduce the complexity and simplify the language, multiple inheritance is not supported in java.**
  - **Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.**



# super Keyword in Java Inheritance

The **super** keyword is used to call the method of the parent class from the method of the child class.

## Why use inheritance?

- The most important use of inheritance in Java is code reusability.
- The code that is present in the parent class can be directly used by the child class.
- Method overriding is also known as runtime polymorphism.

Can we achieve Polymorphism in Java with the help of inheritance?



- super class default constructor is available to sub class by default.
- First, super class default constructor is executed then subclass default constructor is executed.
- super class parameterized constructor is not automatically available to sub class.
- super is the keyword that refers to super class

**super is used to refer:**

- super class variable as: `super.variable_name;`
- `super.method()`
- `super(values)`



# Aggregation in Java

- If a class have an **entity reference**, it is known as Aggregation.
- Aggregation represents **HAS-A relationship**.
- Consider a situation, Employee object contains many information such as id, name, email ID etc.
- It contains one more object named **address**, which contains its own information's such as city, state, country, zipcode etc. as given below.
  - class Employee{
  - int id;
  - String name;
  - Address address;//Address is a class
  - ...
  - }



# Quiz for this class

1. Which of this keyword must be used to inherit a class?

A. super

B. this

C. extent

D. extends

extends

2. A class member declared protected becomes a member of subclass of which type?

A. public member

B. private member

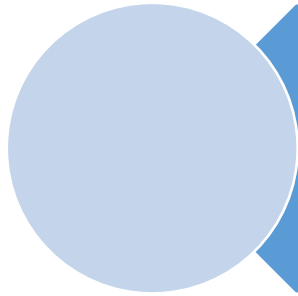
C. protected member

D. static member

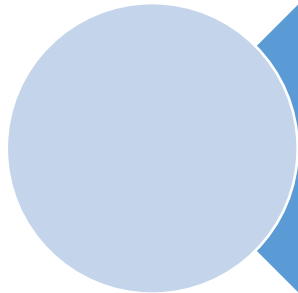
private member



# Topics to be learned from this class



**Final Variables, Methods and  
Classes**



**Finalize Method**



# Finalize Method

- **finalize()** is the method of **Object** class.
- This method is called before an **object** is **garbage collected**.
- Garbage means **unreferenced objects**.
- Garbage Collection is a process of **destroying the unused objects**.
- **finalize()** method is used to perform **clean-up** activities and **minimize memory leaks**.
- Java provides **automatic garbage collection**.
- **Syntax :**

**protected void finalize() throws Throwable**



- **Java finalize() method of Object class is a method that the Garbage Collector always calls just before the deletion/destroying the object.**
- **used to perform cleanup activity before destroying any object.**
- **The method is protected and therefore is accessible only through this class or through a derived class**



# Finalize Method

```
public class FinalizeDemo {  
    public static void main(String[] args){  
        FinalizeDemo fd = new FinalizeDemo();  
        System.out.println(fd.hashCode());  
        fd= null;  
        System.gc();  
        System.out.println("end of garbage collection");  
    }  
    @Override  
    protected void finalize(){  
        System.out.println("finalize method called");  
    }  
}
```

## Output

```
2018699554  
end of garbage collection  
finalize method called
```



# TOPICS TO BE LEARNED FROM THIS CLASS

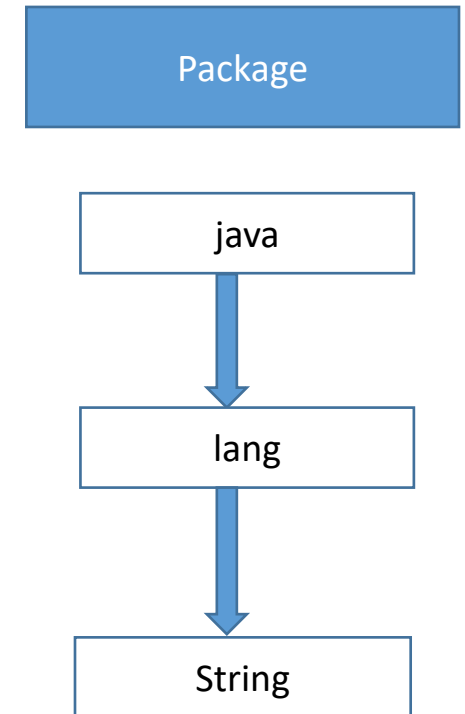


Strings in Java



# STRINGS IN JAVA

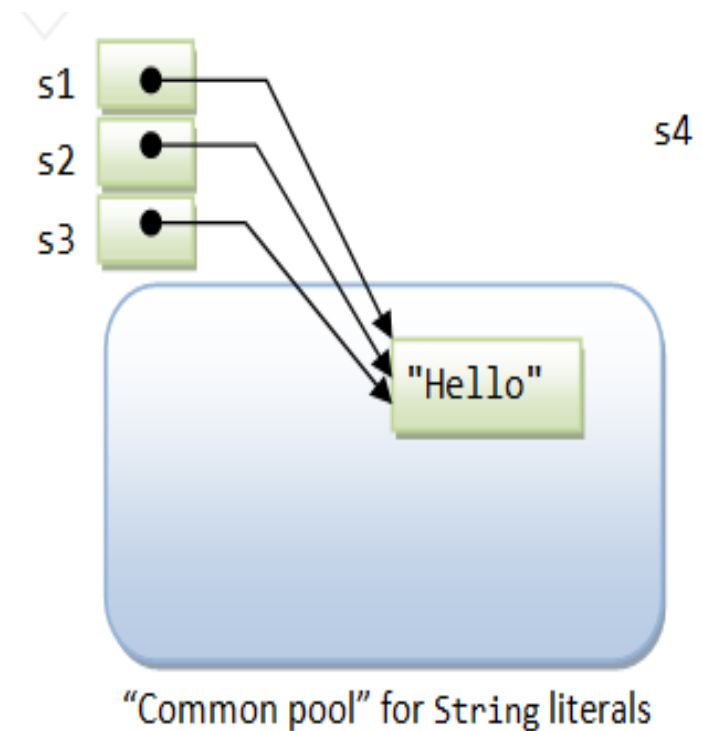
- String is a sequence of characters.
- String is an **object** that represents a sequence of characters.
- The `java.lang.String` class is used to create a String object.
- The String is a in-built class which is available in `java.lang` package.
- There are two ways to create string object:
  1. By String literal
  2. By new keyword



# STRING LITERAL

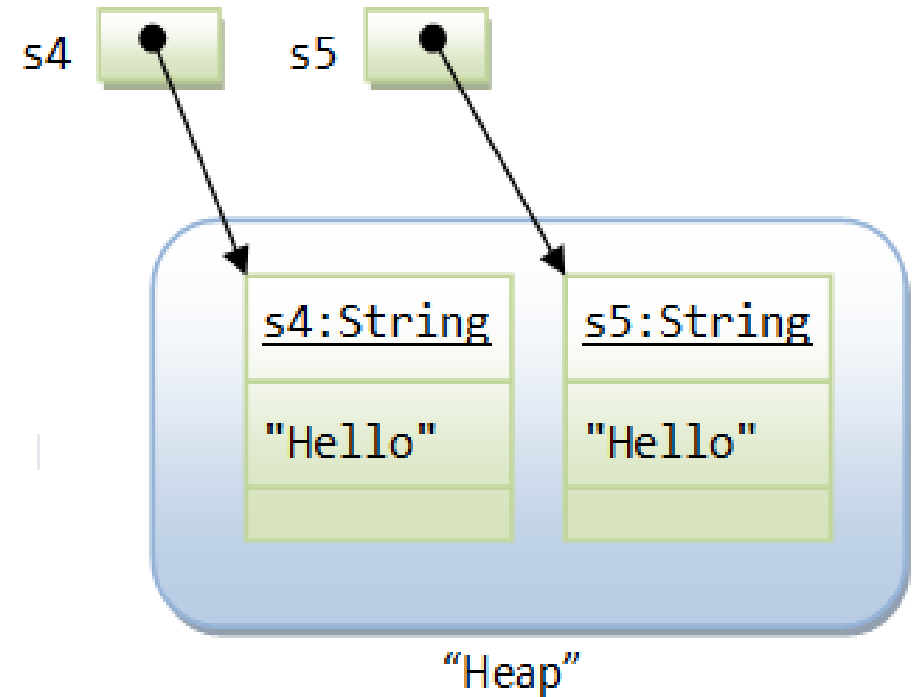
- Java String literal is created by using double quotes.
- Each time we create a string literal, the JVM checks the "string constant pool".
- If the string already exists in the pool, a reference to the pooled instance is returned.
- If the string doesn't exist in the pool, a new string instance is created and placed in the pool.
- Example:

```
String s1="Hello";  
String s2="Hello";  
String s3="Hello";
```



# USING NEW KEYWORD

- String objects can be created by using new keyword.
- JVM will create a new string object in normal heap memory.
- Example:-  
`String s4=new String("Hello");`  
`String s5=new String("Hello");`



# STRING CLASS METHODS

No	Method	Description
1	<a href="#"><u>char charAt(int index)</u></a>	returns char value for the particular index
2	<a href="#"><u>int length()</u></a>	returns string length
3	<a href="#"><u>String substring(int beginIndex)</u></a>	returns substring for given begin index.
4	<a href="#"><u>String substring(int beginIndex, int endIndex)</u></a>	returns substring for given begin index and end index.
5	<a href="#"><u>boolean contains(CharSequence s)</u></a>	returns true or false after matching the sequence of char value.
6	<a href="#"><u>boolean equals(Object another)</u></a>	checks the equality of string with the given object.



# STRING CLASS METHODS

No	Method	Description
7	<a href="#"><u>String concat(String str)</u></a>	concatenates the specified string.
8	<a href="#"><u>String replace(char old, char new)</u></a>	replaces all occurrences of the specified char value.
9	<a href="#"><u>static String equalsIgnoreCase(String another)</u></a>	compares another string. It doesn't check case.
10	<a href="#"><u>String[] split(String regex)</u></a>	returns a split string matching regex.
11	<a href="#"><u>int indexOf(int ch)</u></a>	returns the specified char value index.
12	<a href="#"><u>String toLowerCase()</u></a>	returns a string in lowercase.
13	<a href="#"><u>String toUpperCase()</u></a>	returns a string in uppercase.
14	<a href="#"><u>String trim()</u></a>	removes beginning and ending spaces of this string.



## EXAMPLE

```
class StringMethods{  
    public static void main(String[] args){  
        String str1="Hello";  
        String str2="Welcome";  
        System.out.println("Str1 length is :"+str1.length());  
        System.out.println("Str2 length is :"+str2.length());  
        System.out.println("The concatenation is :"+str1.concat(" "+str2));  
        System.out.println("The first character of " +str1+" is: "+str1.charAt(0));  
        System.out.println("The uppercase of " +str1+" is: "+str1.toUpperCase());  
        System.out.println("The lowercase of " +str1+" is: "+str1.toLowerCase());  
        System.out.println("The 'c' occurs at position in str2 : "+str2.indexOf('c'));  
        System.out.println("Replacing 'H' with 'T' in str1 : "+str1.replace('H','T'));  
    }  
}
```



# STRING PRACTICE PROGRAMS

1. Write a java program to find reverse of the string.
2. Write a java program to convert first letter of every word to uppercase from the given statement.

(Example : this is a simple program = TIASP).

3. Write a java program to find the largest word in the string.



# SUMMARY



Creating Strings



Demonstration of String  
Methods



## QUIZ FOR THIS CLASS

1. The method used to remove beginning and ending spaces of the string?

A. split()

B. trim()

C. replace()

D. concat()

trim()

2. String class is available in this package?

A. java.util

B. java.awt

C. java.io

D. java.lang

java.lang



# Topics to be learned from this class



Wrapper Classes



# Wrapper classes

- Wrapper classes are used to convert **primitive** into **object** and **object** into **primitive** (to wrap primitive data types into objects).
  - Java provides a wrapper class for each of the primitive types.
  - The java.lang package provides **wrapper classes**.
- The primitive data types in Java (such as int, char, boolean, etc.) do not have the ability to act as objects, and they don't offer methods to perform actions.
- Wrapper classes bridge this gap by converting primitives into objects and providing utility methods for operations.



- **Collections Framework:** Java collections like ArrayList, HashMap, etc., can only store objects, not primitives. Wrapper classes allow you to store primitives in collections.
- **Utility Methods:** Wrapper classes provide several useful methods, such as parseInt(), valueOf(), etc.
- **Null Values:** Objects can hold a null value, while primitives cannot. This is useful when representing missing or optional values.
- **Autoboxing and Unboxing:** Java automatically converts between primitive types and their corresponding wrapper classes in certain situations. This is called autoboxing (primitive to wrapper) and unboxing (wrapper to primitive)



# Wrapper Classes

➤ The different scenarios, where we need to use the wrapper classes are .

## 1. Change the value in Method

- Java supports **only call by value**.
- If we pass a **primitive** value, it will not change the **original value**.
- If we **convert** the **primitive** value in an **object**, it will **change the original value**.

## 2. Serialization

- We need to **convert** the **objects** into **streams** to perform the **serialization**.
- If we have a **primitive value**, we can **convert** it in **objects** through the **wrapper classes**.

## 3. Synchronization

- Java synchronization works with objects in **Multithreading**.



# Wrapper Classes

- There are totally eight wrapper classes present in the java.lang package

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double



# Example : Wrapper to Primitive

- The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing.
- Java 5 version onwards the `intValue()` method of wrapper classes is not needed to convert the wrapper into primitives.



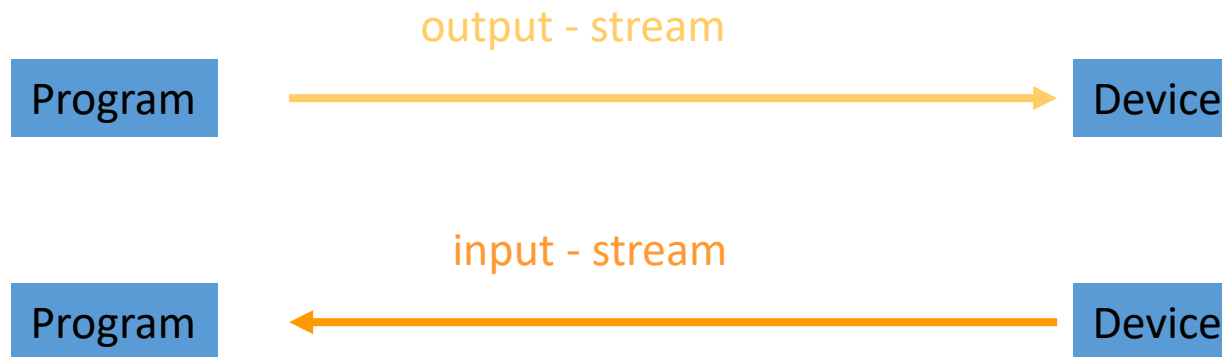
# Wrapper classes Example

- Java Wrapper classes:
- Custom Wrapper class in Java:

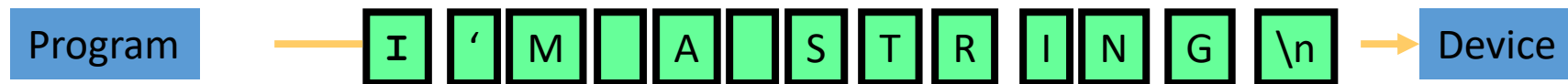


# Streams I/O

- Usual Purpose: storing data to 'nonvolatile' devices, e.g. harddisk
  - Classes provided by package java.io
  - Data is transferred to devices by 'streams'

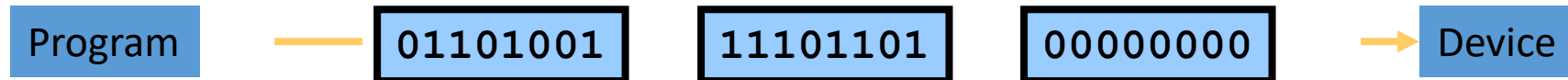


# Streams



JAVA distinguishes between 2 types of streams:

- Text – streams, containing ‘characters’
- Binary Streams, containing 8 – bit information



# Streams

Streams in JAVA are Objects.

- 2 types of streams (text / binary) and
- 2 directions (input / output)

results in 4 base-classes dealing with I/O:

1. Reader: text-input
2. Writer: text-output
3. InputStream: byte-input
4. OutputStream: byte-output



# Streams

- InputStream, OutputStream, Reader, Writer are abstract classes
- Subclasses can be classified by 2 different characteristics of sources / destinations:

For final device (data sink stream)

purpose: serve as the source/destination of the stream  
(these streams 'really' write or read !)

for intermediate process (processing stream)

Purpose: alters or manages information in the stream  
(these streams are 'luxury' additions, offering methods for convenient or more efficient stream-handling)



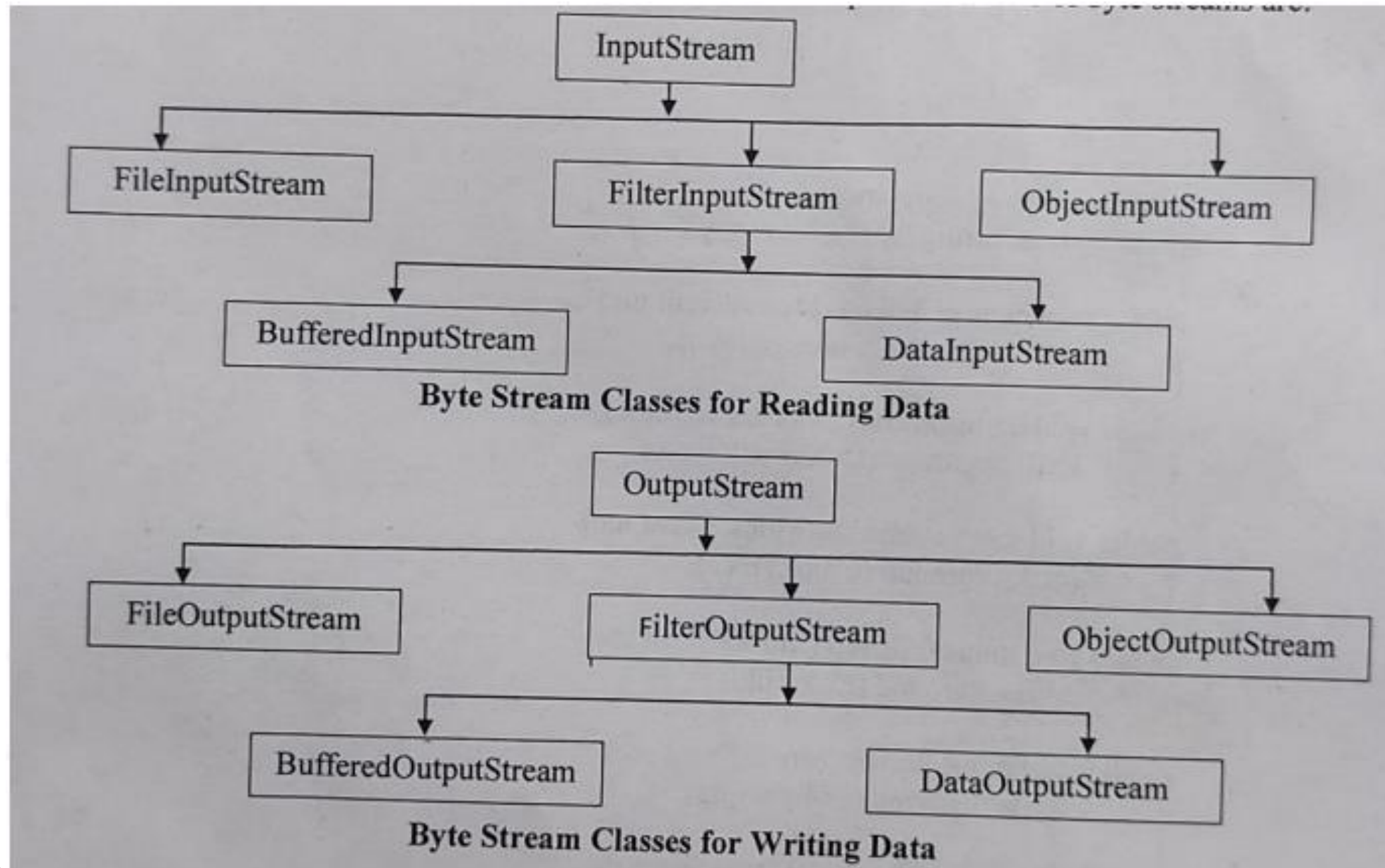
# I/O: General Scheme

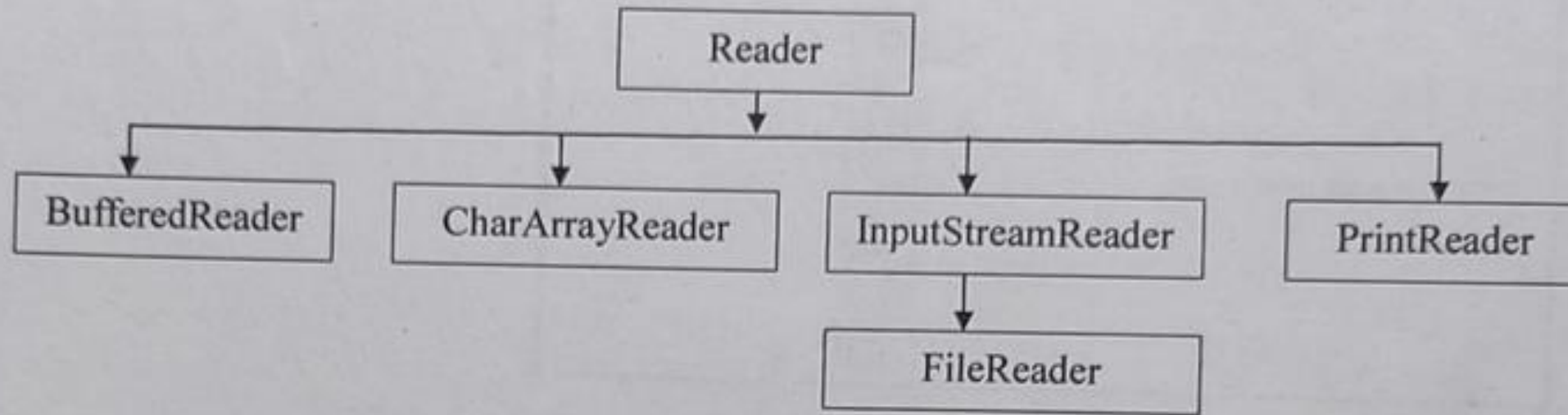
Reading (writing):

- open an input (output) stream.
- while there is more information read(write) next data from the stream.
- close the stream.

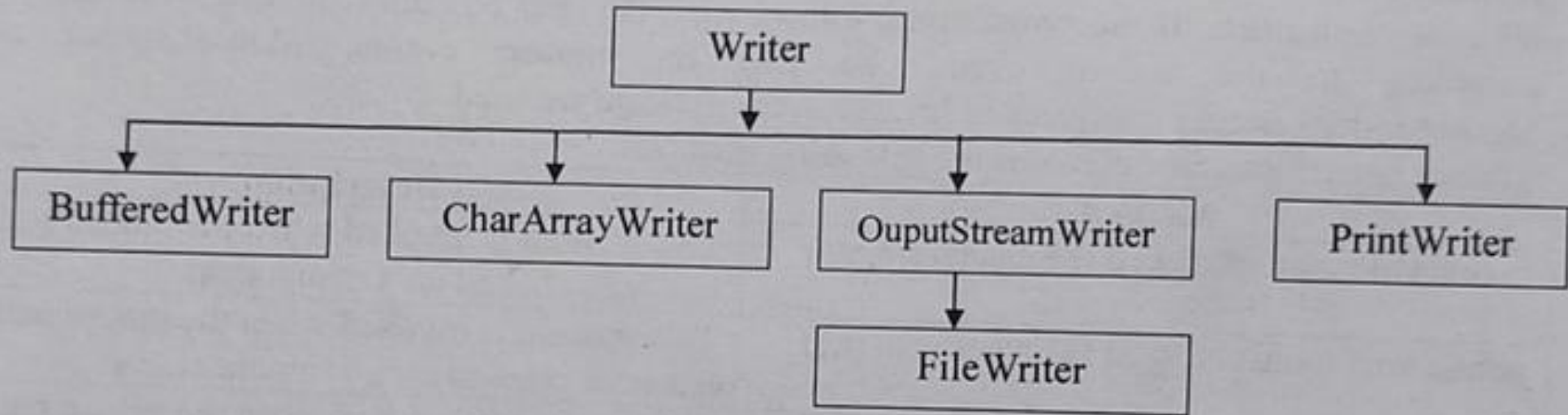
In JAVA:

- Create a stream object and associate it with a disk-file
  - Give the stream object the desired functionality
- while there is more information read(write) next data from(to) the stream
- close the stream.





### Text Stream Classes for Reading Data



### Text Stream Classes for Writing Data



# Example 1

## Writing a Text File

```
import java.io.*;

public class IOTest
{
    public static void main(String[] args)
    {
        try{

            FileWriter out = new FileWriter("test.txt");
            BufferedWriter b = new BufferedWriter(out);
            PrintWriter p = new PrintWriter(b);

            p.println("I'm a sentence in a text-file");

            p.close();
        } catch (Exception e) {}
    }
}
```

- Create a stream object and associate it with a disk-file
- Give the stream object the desired functionality
- write data to the stream
- close the stream.

# Writing Textfiles

## Class: FileWriter

### Frequently used methods:

#### Method Summary

abstract void	<u><a href="#">close</a></u> () Close the stream, flushing it first.
abstract void	<u><a href="#">flush</a></u> () Flush the stream.
void	<u><a href="#">write</a></u> (char[] cbuf) Write an array of characters.
abstract void	<u><a href="#">write</a></u> (char[] cbuf, int off, int len) Write a portion of an array of characters.
void	<u><a href="#">write</a></u> (int c) Write a single character.
void	<u><a href="#">write</a></u> ( <a href="#">String</a> str) Write a string.
void	<u><a href="#">write</a></u> ( <a href="#">String</a> str, int off, int len) Write a portion of a string.



## Writing Textfiles

### Using FileWriter for

- is not very convenient (only String-output possible)
- Is not efficient (every character is written in a single step, invoking a huge overhead)

Better: wrap FileWriter with processing streams

- BufferedWriter
- PrintWriter

## Wrapping Textfiles

### BufferedWriter:

- Buffers output of FileWriter, i.e. multiple characters are processed together, enhancing efficiency

### PrintWriter

- provides methods for convenient handling, e.g. `println()`

( remark: the `System.out.println()` – method is a method of the PrintWriter-instance `System.out` ! )



# Wrapping a Writer

A typical codesegment for opening a convenient, efficient textfile:

```
FileWriter out = new FileWriter("test.txt");  
BufferedWriter b = new BufferedWriter(out);  
PrintWriter p = new PrintWriter(b);
```

Or with anonymous ('unnamed') objects:

```
PrintWriter p = new PrintWriter(  
    new BufferedWriter(  
        new FileWriter("test.txt")));
```

# Binary Files

- Stores binary images of information identical to the binary images stored in main memory
- Binary files are more efficient in terms of processing time and space utilization
- drawback: not 'human readable', i.e. you can't use a texteditor (or any standard-tool) to read and understand binary files

# Binary Files

Example: writing of the integer '42'

- TextFile: '4' '2' (internally translated to 2 16-bit representations of the characters '4' and '2')
  - Binary-File: 00101010, one byte  
(= 42 decimal)

# Package java.io

- Provides for system input and output through data streams, serialization and the file system.

## Interface Summary

- [.DataInput](#) The DataInput interface provides for reading bytes from a binary stream and reconstructing from them data in any of the Java primitive types.
- [DataOutput](#) The DataOutput interface provides for converting data from any of the Java primitive types to a series of bytes and writing these bytes to a binary stream
- [.Externalizable](#) Only the identity of the class of an Externalizable instance is written in the serialization stream and it is the responsibility of the class to save and restore the contents of its instances.
- [Serializable](#) Serializability of a class is enabled by the class implementing the java.io.Serializable interface.



- **BufferedInputStream**: A BufferedInputStream adds functionality to another input stream-namely, the ability to buffer the input and to support the mark and reset methods.
- **BufferedOutputStream**: The class implements a buffered output stream.
- **BufferedReader**: Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
- **BufferedWriter**: Writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings
- **ByteArrayInputStream**: A ByteArrayInputStream contains an internal buffer that contains bytes that may be read from the stream.
- **ByteArrayOutputStream**: This class implements an output stream in which the data is written into a byte array.



- [CharArrayReader](#): This class implements a character buffer that can be used as a character-input stream
- [.CharArrayWriter](#): This class implements a character buffer that can be used as an Writer
- [Console](#): Methods to access the character-based console device, if any, associated with the current Java virtual machine.
- [DataInputStream](#): A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way.
- [DataOutputStream](#): A data output stream lets an application write primitive Java data types to an output stream in a portable way.



- **File**: An abstract representation of file and directory pathnames.
- **FileInputStream**: A FileInputStream obtains input bytes from a file in a file system.
- **FileOutputStream**: A file output stream is an output stream for writing data to a File or to a FileDescriptor.
- **FileReader**: Convenience class for reading character files.
- **FileWriter**: Convenience class for writing character files.
- **FilterInputStream**: A FilterInputStream contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality.
- **FilterOutputStream**: This class is the superclass of all classes that filter output streams
- **.FilterReader**: Abstract class for reading filtered character streams
- **.FilterWriter**: Abstract class for writing filtered character streams
- **.InputStream**: This abstract class is the superclass of all classes representing an input stream of bytes.
- **InputStreamReader**: An InputStreamReader is a bridge from byte streams to character streams; It reads bytes and decodes them into characters using a specified **charset**.



- **ObjectInputStream**: An ObjectInputStream deserializes primitive data and objects previously written using an ObjectOutputStream
- **ObjectOutputStream**: An ObjectOutputStream writes primitive data types and graphs of Java objects to an OutputStream.
- **OutputStream**: This abstract class is the superclass of all classes representing an output stream of bytes.
- **OutputStreamWriter**: An OutputStreamWriter is a bridge from character streams to byte streams: Characters written to it are encoded into bytes using a specified [charset](#).
- **PrintWriter**: Prints formatted representations of objects to a text-output stream.
- **RandomAccessFile**: Instances of this class support both reading and writing to a random access file.
- **StreamTokenizer**: The StreamTokenizer class takes an input stream and parses it into "tokens", allowing the tokens to be read one at a time.



# Collections

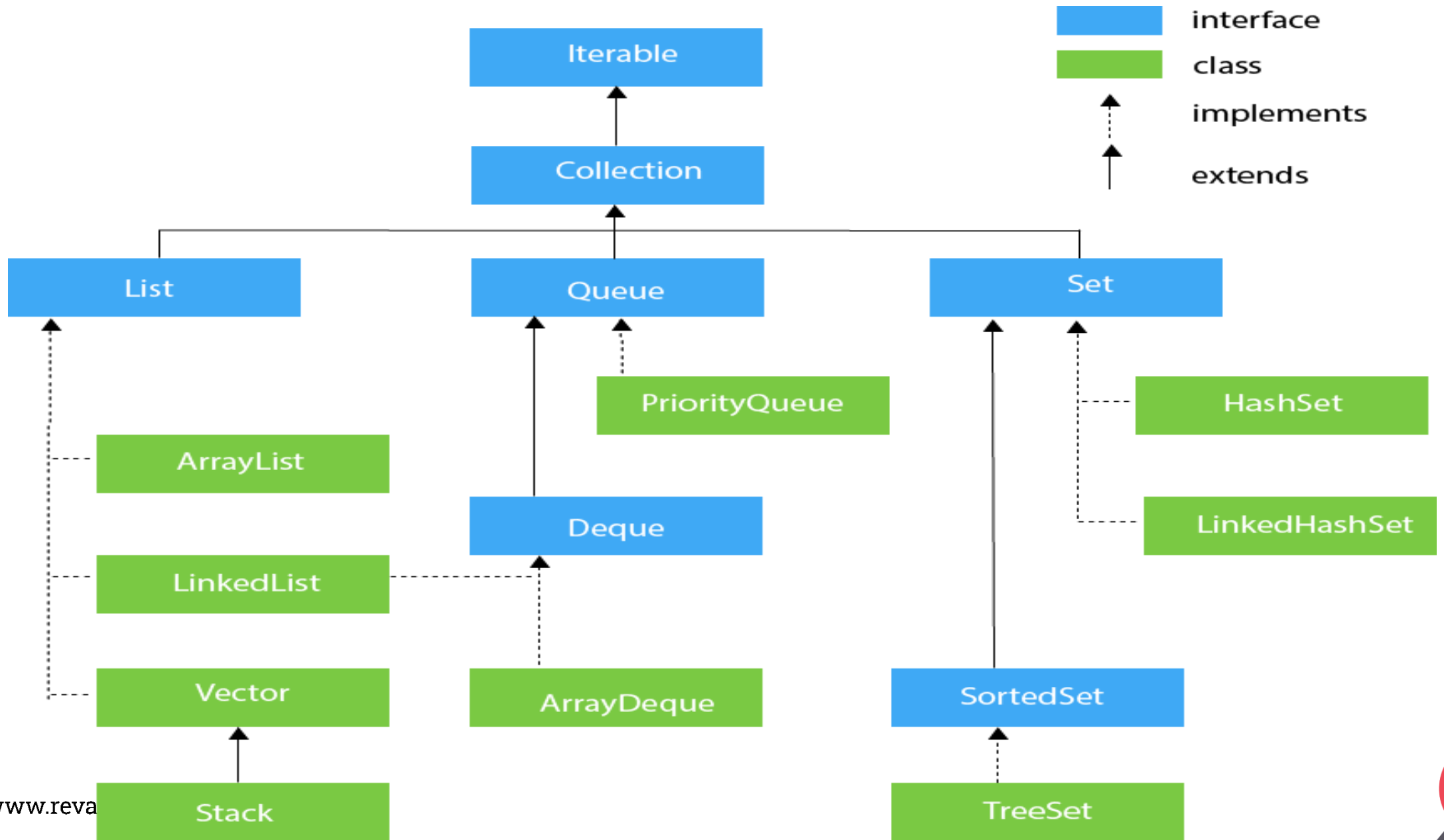
- A Collection represents a single unit of objects. i.e group
- The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.
- Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
- Java Collection means a single unit of objects.
- Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).



# Collections

- It is part of the `java.util` package and provides a set of classes and interfaces to help developers work with collections of data.
- Collections enable developers to manage and organize data more efficiently and improve code readability, flexibility, and performance.





- **Class:** A class is a blueprint from which individual objects are created. It encapsulates data for objects through fields (attributes) and defines behavior via methods. Classes support inheritance, allowing one class to inherit the properties and methods of another, facilitating code reuse and polymorphism.
- **Interface:** An interface is a reference type in Java that can contain constants and abstract methods (methods without a body). Interfaces specify what a class must do but not how it does it, enforcing a set of methods that the class must implement. Interfaces are used to achieve abstraction and multiple inheritance in Java.



# Classes in collection framework

- **ArrayList**: Resizable array implementation of the List interface, ideal for dynamic array storage.
- **LinkedList**: Doubly-linked list implementation, used for both List and Queue operations.
- **HashSet**: Implements the Set interface using a hash table, with no guarantee of element order.
- **HashMap**: Hash table-based implementation of the Map interface, providing key-value mapping with constant-time performance for operations.
- **TreeSet**: A Set that maintains elements in a sorted (ascending) order.



# Interfaces in the Java Collections Framework

- **Collection:** The root interface of the framework that represents a group of objects, called elements. Collection has subinterfaces like List, Set, and Queue.
- **List:** An ordered collection that allows duplicate elements. Examples: ArrayList, LinkedList, Vector.
- **Set:** A collection that does not allow duplicate elements and is typically unordered. Examples: HashSet, LinkedHashSet, TreeSet.
- **Queue:** A collection used to hold multiple elements before processing. Supports FIFO (First-In-First-Out) processing. Examples: LinkedList, PriorityQueue.
- **Map:** An interface that represents a collection of key-value pairs, where each key is unique. Examples: HashMap, TreeMap, Hashtable.



# Benefits of Using the Collections Framework

- Efficiency:** Collections are optimized for performance and allow developers to use ready-made data structures.
- Flexibility:** Different types of collections serve different purposes, allowing developers to choose the best structure for their needs.
- Code Readability:** Simplified syntax and predefined classes reduce the need for complex custom data structures, improving code readability.
- Interoperability:** Collections can easily be transferred or passed between different parts of a program, supporting modular programming.



# ARRAYLIST

**class ArrayList<E>**

- **ArrayList<String> arl=new ArrayList<String>();**

**Methods ArrayList are:**

- **boolean add(element obj)**
- **void add(int position, element obj)**
- **element remove(int position)**
- **int size();**
- **int indexOf(object obj)**
- **int lastIndexOf(Object obj)**



# Retrieving elements from collections

- **Using Iterator interface:**

- Boolean hasNext()
- Element next()
- Void remove()

- **Using ListIterator interface.**

boolean hasNext()                      element next()  
boolean hasPrevious()  
void remove()                      element previous()

- **Using Enumeration interface.**

boolean hasMoreElements()  
Element nextElement()



# what is lambda expression in java?

- Lambda Expressions were added in Java 8.
- A lambda expression is a short block of code which takes in parameters and returns a value.
- Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.
- A lambda expression is a concise way to implement a functional interface.
- A functional interface is an interface that has only one abstract method, and it can be implemented using a lambda expression.
- In general, a class or inner class to implement the functional interface is required. But using lambda expression we do not need of new class and inner class to implement.



# Properties of the lambda expression

- Lambda expressions provide a concise and expressive syntax for defining functional interfaces.
- They allow you to define the behavior of a functional interface in a single line of code.
- Lambda expressions are a key component of functional programming in Java.
- They allow you to write code that is more declarative and expressive, and less verbose than traditional imperative code.
- **No need for anonymous inner classes:** Lambda expressions provide an alternative to anonymous inner classes, which are often used to implement functional interfaces in Java. Lambda expressions are more concise and easier to read than anonymous inner classes.



# syntax

(int arg1, String arg2) -> {System.out.println("Two arguments "+arg1+" and "+arg2);}

Argument List      Arrow token      Body of lambda expression

() -> System.out.println("Zero parameter lambda"); // Zero Parameter

(p) -> System.out.println("One parameter: " + p); // Single Parameter

(parameter1, parameter2) -> expression; // two Parameters

(parameter1, parameter2) -> { code block }; // multiple line in the body of lambda expression

