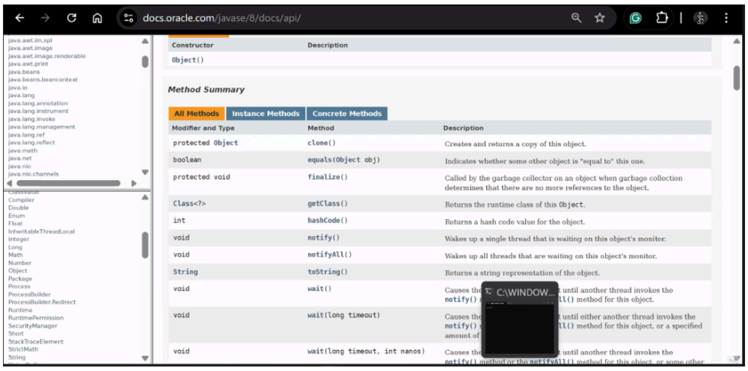
Day_12_OOPJ_Sanket_Shalukar

Wednesday, September 10, 2025 10:26 AM

Topics are in the Day_12

1. Object Class
2. Packages
3. Access Modifiers
4. Collections



Constructor	Description
Object()	

Method Summary

Modifier and Type	Method	Description
protected Object	clone()	Creates and returns a copy of this object.
boolean	equals(Object obj)	Indicates whether some other object is "equal to" this one.
protected void	finalize()	Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	getClass()	Returns the runtime class of this Object.
int	hashCode()	Returns a hash code value for the object.
void	notify()	Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll()	Wakes up all threads that are waiting on this object's monitor.
String	toString()	Returns a string representation of the object.
void	wait()	Causes the [illegible] until another thread invokes the notify() [illegible] ll() method for this object.
void	wait(long timeout)	Causes the [illegible] until either another thread invokes the notify() [illegible] ll() method for this object, or a specified amount of [illegible]
void	wait(long timeout, int nanos)	Causes the [illegible] until another thread invokes the notify() method or the notifyAll() method for this object, or some other

Object Class :

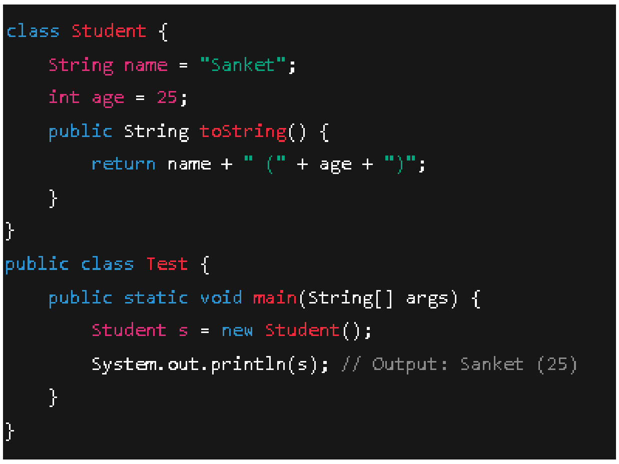
1. Object class is from "Java.lang" packge!
2. Root in hierarchy : Every class directly or indirectly is accessing this object class.
3. If a class does not explicitly extends another class, it is considered as Object class.

Object class methods :

1. ToString () - Return string representation of the object
2. Equals () - Compare two objects for equality
3. Hashcode () - Returns hash code of the objects
4. Clone () - Create clone of object
5. GetClass () - returns the runtime name of a class of the object
6. Finalize () - Called by GC before object destruction
7. Wait () - Causes the current thread to wait
8. Notify () - wakes up the thread on the object's monitor
9. Notifyall () - Wakes up all the thread waiting on the objects monitor

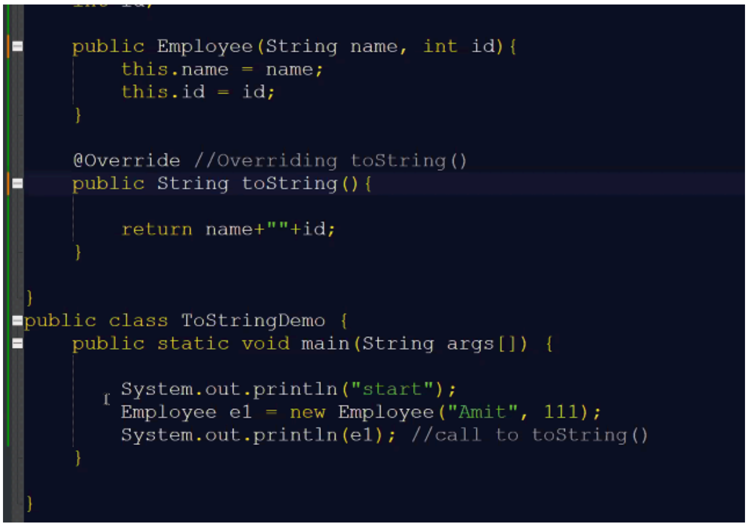
Examples of Object class method.

1. ToString ()



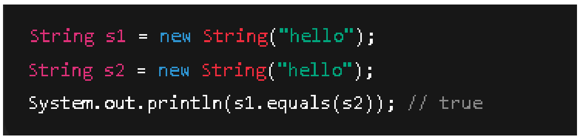
```java
class Student {
    String name = "Sanket";
    int age = 25;
    public String toString() {
        return name + " (" + age + ")";
    }
}
public class Test {
    public static void main(String[] args) {
        Student s = new Student();
        System.out.println(s); // Output: Sanket (25)
    }
}
```

```java
class Employee{
    String name;
    int id;
```



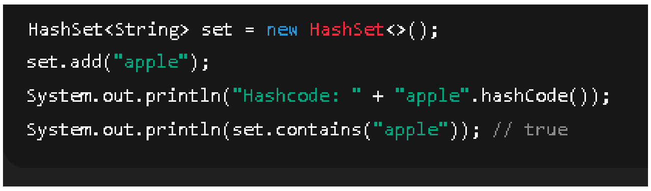
```
    int id;

    public Employee(String name, int id){
        this.name = name;
        this.id = id;
    }

    @Override //Overriding toString()
    public String toString(){

        return name+""+id;
    }

}
public class ToStringDemo {
    public static void main(String args[]) {

        System.out.println("start");
        Employee e1 = new Employee("Amit", 111);
        System.out.println(e1); //call to toString()
    }

}
```

2. Equals ()



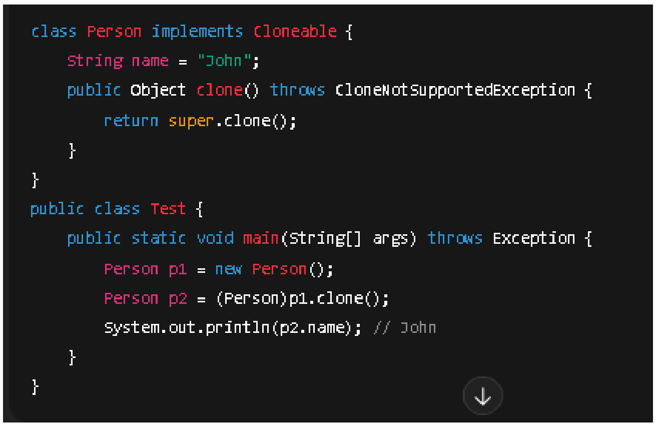
```
String s1 = new String("hello");
String s2 = new String("hello");
System.out.println(s1.equals(s2)); // true
```

3. Hashcode ()



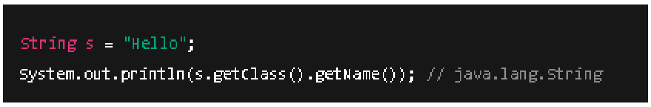
```
HashSet<String> set = new HashSet<>();
set.add("apple");
System.out.println("Hashcode: " + "apple".hashCode());
System.out.println(set.contains("apple")); // true
```

4. Clone () -



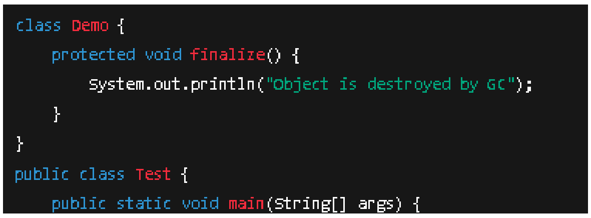
```
class Person implements Cloneable {
    String name = "John";
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
public class Test {
    public static void main(String[] args) throws Exception {
        Person p1 = new Person();
        Person p2 = (Person)p1.clone();
        System.out.println(p2.name); // John
    }
}
```

1. GetClass () -

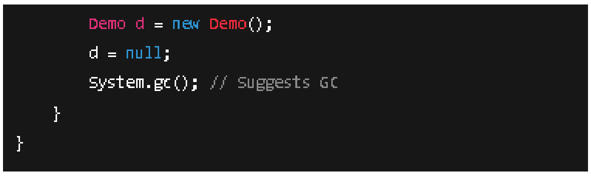


```
String s = "Hello";
System.out.println(s.getClass().getName()); // java.lang.String
```

1. Finalize () -

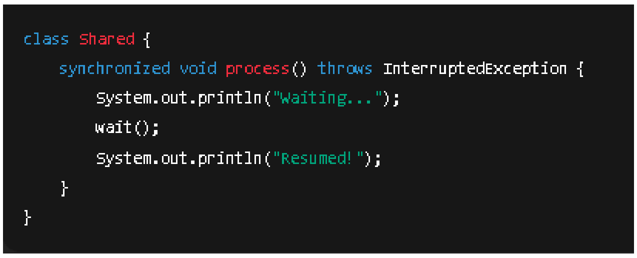


```
class Demo {
    protected void finalize() {
        System.out.println("Object is destroyed by GC");
    }
}
public class Test {
    public static void main(String[] args) {
```



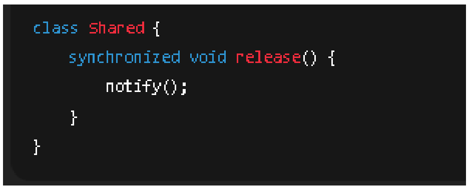
```
        Demo d = new Demo();
        d = null;
        System.gc(); // Suggests GC
    }
}
```

1. Wait () -



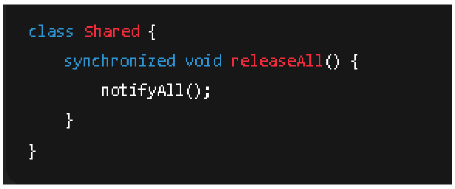
```
class Shared {
    synchronized void process() throws InterruptedException {
        System.out.println("Waiting...");
        wait();
        System.out.println("Resumed!");
    }
}
```

1. Notify () -



```
class Shared {
    synchronized void release() {
        notify();
    }
}
```

1. Notifyall () -



```
class Shared {
    synchronized void releaseAll() {
        notifyAll();
    }
}
```

Packages :

Definition

A **package** in Java is a way to group related **classes, interfaces, and sub-packages** together.
It works like a **folder/directory** structure for organizing code.

Info

- Packages prevent **name conflicts** (two classes with the same name can exist in different packages).
- They provide **modularity** and **reusability** of code.
- They help in **access protection** by combining with access modifiers.
- They make code **easier to maintain** and organize.

Types of Packages

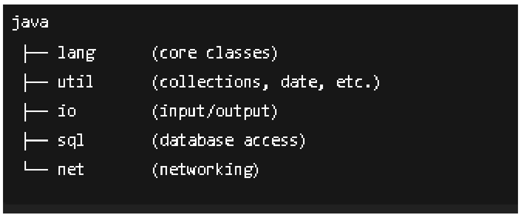
1. **Built-in Packages**
- Provided by Java API.
- Examples:
- `java.lang` → Core classes (String, Math, Object, etc.)
- `java.util` → Data structures (ArrayList, HashMap, etc.)
- `java.io` → Input/Output classes
- `java.sql` → Database connectivity
- `javax.swing` → GUI components

2. **User-defined Packages**
- Created by programmers to organize their own classes and projects.

- Useful in large projects to avoid messy code.

Uses of Packages

- **Code organization**: Keep related classes together.
- **Namespace management**: Avoid class name conflicts.
- **Reusability**: Classes in one package can be reused in other programs.
- **Access control**: Work with access modifiers for controlled visibility.
- **Modularity**: Divide a big project into smaller, manageable modules.



```
java
 ├── lang      (core classes)
 ├── util      (collections, date, etc.)
 ├── io        (input/output)
 ├── sql       (database access)
 └── net       (networking)
```

Use of eclipse and installation :

Eclipse IDE

What is Eclipse?

Eclipse is a free, open-source Integrated Development Environment (IDE). It is most commonly used for Java development, but supports many other languages with plugins.

Uses of Eclipse

- Java Development – writing, compiling, debugging, and running applications.
- Project Management – managing multiple projects within a workspace.
- Code Assistance – automatic suggestions, error highlighting, and quick fixes.
- Debugging – step-by-step execution, breakpoints, and variable inspection.
- Integration – works with Git, Maven, Gradle, and other developer tools.
- Extensibility – plugins available for frameworks such as Spring, Hibernate, and JUnit.
- Cross-platform – runs on Windows, macOS, and Linux.

Installation Steps

- Download Eclipse from the official website: https://www.eclipse.org/downloads/
- Install the latest Java Development Kit (JDK) before using Eclipse.
- Run the Eclipse installer and select "Eclipse IDE for Java Developers".
- Choose the installation path and complete installation.
- Open Eclipse and select a workspace folder (this is where your projects will be saved).

How to Start Eclipse

- Locate the Eclipse application icon (on desktop or start menu) and open it.
- When prompted, choose a workspace (a directory where projects will be stored).
- The Welcome screen will appear with tutorials and sample options. Close it to access the main IDE.
- Create a new Java Project by selecting:
- File > New > Java Project
- Add a class:
- Right click on src > New > Class
- Enter a class name and select "public static void main(String[] args)" if you want a main method.
- Write and run code using the Run button (green play icon) or press Ctrl + F11.

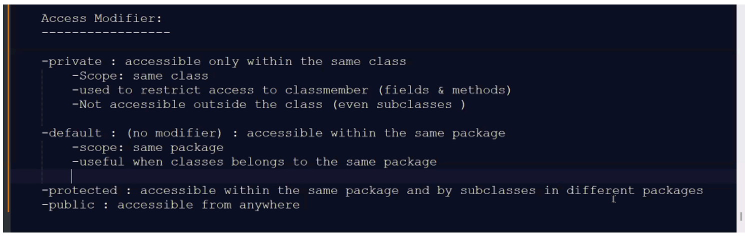
How to Generate Constructor, Setter, and Getter Automatically

Eclipse can generate these methods for you:

- Open your Java class file.
- Place the cursor inside the class body.
- From the menu bar, select:
- Source > Generate Constructor using Fields → creates constructor.

- Source > Generate Getters and Setters → creates getters and setters.
- Select the fields for which you want the methods to be generated.
- Click OK, and Eclipse will insert the methods automatically.

Access Modifiers :



```
Access Modifier:
-----------------

-private : accessible only within the same class
    -Scope: same class
    -used to restrict access to classmember (fields & methods)
    -Not accessible outside the class (even subclasses )

-default : (no modifier) : accessible within the same package
    -scope: same package
    -useful when classes belongs to the same package

-protected : accessible within the same package and by subclasses in different packages
-public : accessible from anywhere
```

1. **public**

- **Definition**: The member/class is accessible **everywhere** in the program.
- **Info**: Highest visibility. Can be accessed from same class, same package, subclass, or different package.
- **Use**: For methods, classes, or variables that should be used globally.

2. **protected**

- **Definition**: The member is accessible within the same package and also in subclasses (even in different packages).
- **Info**: More restrictive than `public`, but more open than `default`.
- **Use**: Common for methods/variables that subclasses may need to inherit but should not be exposed to the world.

3. **default** (package-private)

- **Definition**: If no modifier is specified, access is limited to the **same package**.
- **Info**: Neither `public` nor `private`. Accessible only within package scope.
- **Use**: For package-level functionality where classes and methods should not be exposed outside.

4. **private**

- **Definition**: The member is accessible only within the **same class**.
- **Info**: Most restrictive access level. Not visible in subclasses or other packages.
- **Use**: For internal details (like helper methods, sensitive data) that should not be exposed outside the class.

• Collections :

Legacy Classes (before Java 2, later adapted into JCF)

- These existed before the Collection Framework was introduced (Java 2, JDK 1.2) and were later re-engineered to fit into it.
- Vector
- A growable array (like ArrayList)
- Thread-safe (synchronized)
- Considered legacy, but still used in multi-threaded environments.
- Subclass: Stack (LIFO order).
- Stack
- A subclass of Vector.
- Implements LIFO (Last In First Out).
- Methods: push(), pop(), peek().
- Hashtable

- Map implementation that is synchronized.
- Legacy version of HashMap.
- Does not allow null key or null values.
- Enumeration (interface)
- Used to iterate legacy classes like Vector and Hashtable.
- Predecessor of Iterator.
- Methods: hasMoreElements(), nextElement().

2. Utility Classes

- Collections (with "s") → Provides static utility methods such as sort(), shuffle(), reverse(), max(), min().
- Arrays (java.util) → Utility class for working with arrays (sorting, searching, conversion to List).

3. Other Interfaces in Collection Framework

- Deque (subinterface of Queue)
- Double-ended queue (insertion/deletion from both ends).
- Implementations: ArrayDeque, LinkedList.
- SortedSet (subinterface of Set)
- Maintains elements in ascending order.
- Implementation: TreeSet.
- NavigableSet (subinterface of SortedSet)
- Adds navigation methods (lower, higher, ceiling, floor).
- SortedMap (subinterface of Map)
- Orders keys in natural order.
- Implementation: TreeMap.
- NavigableMap (subinterface of SortedMap)
- Adds navigation methods to Map (firstEntry, lastEntry, ceilingEntry).

---

1. Iterable

- Iterable is the root interface of the Collection framework.
- It is in java.lang package.
- Any class that implements Iterable can be iterated using a for-each loop or Iterator.

2. Collection

- Collection is the root interface of the Collection Framework (in java.util).
- It extends Iterable.
- It represents a group of objects.
- Subinterfaces include List, Set, and Queue.

3. List

- Ordered collection.
- Allows duplicates.
- Indexed access to elements.
- Implementations:
- ArrayList
- LinkedList
- Vector → Stack

4. Queue

- Represents a collection used to hold elements before processing.
- Follows FIFO (First In First Out) order (some queues can be priority-based).
- Implementations:
- PriorityQueue
- LinkedList (can act as Queue)
- ArrayDeque

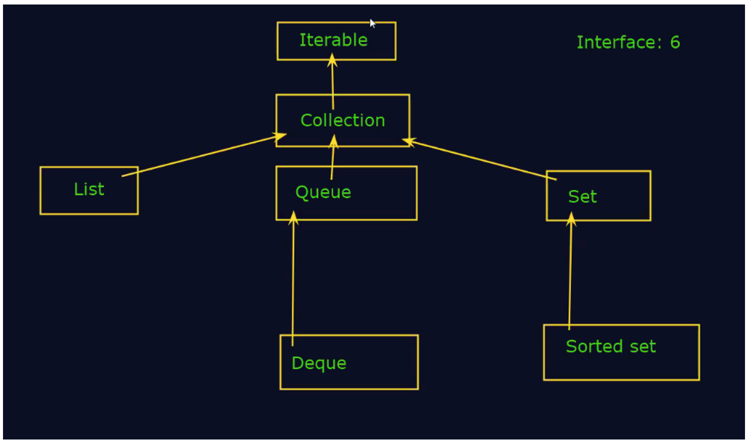
5. **Set**

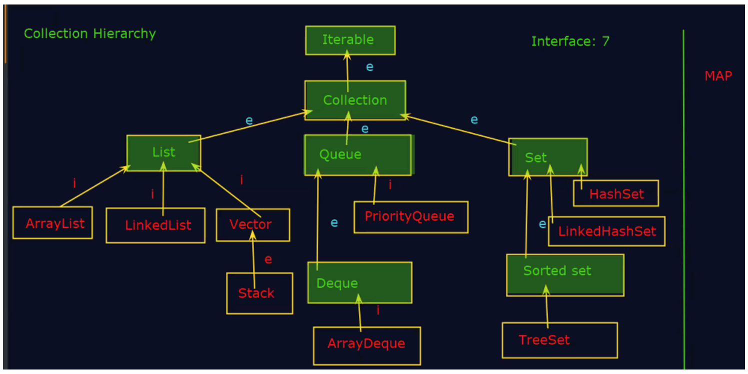
- Represents an unordered collection.
- Does not allow duplicates.
- Implementations:
- HashSet
- LinkedHashSet
- TreeSet

6. **Map**

- Map is not a child of Collection, but part of the framework.
- Stores data in key-value pairs.
- Keys are unique, values can be duplicate.
- Implementations:
- HashMap
- LinkedHashMap
- TreeMap
- Hashtable

1. **Root Interface**



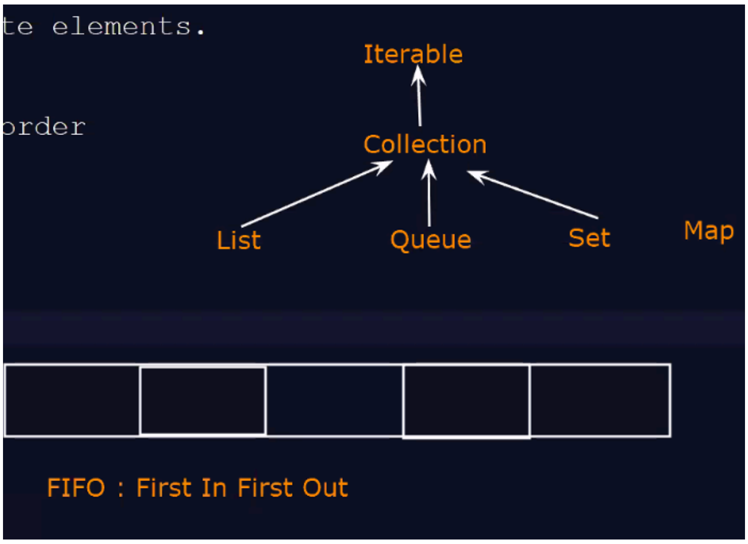


2. **List ordered :**
collection, allow duplicates

3. **Set :**
Unordered, does not allow duplicate elements

4. **Queue :**
Follows FIFO (First in First Out) order

5. **Map :**
Not a part of collection interface
Stroes key-value pairs



List Interface:

- **Ordered collection :**
- **Allows duplicates elements :**
- **Maintains insertion order :**
- **Implementation :**

Array list : Dynamic array, fast read access
Linkedlist : Doulby Linked list, , fast insert /delete
Vector : Thread-safe, legacy classes

ArrayList a1 = new Array List () :
Arraylist<String> a1 = new ArrayList <>();
ArrayList<Interger> a1 = new ArrayList<>();

Collection c = new ArrayList();

Collection c = new arrayList ();
Collection<Strinng> c = new ArrayList <>();
LinkedList l1 = new LinkedList();

Differentiate between Collection vs Collections.

- **Collection** → It is an **interface** in `java.util` package.
- **Collections** → It is a **utility class** in `java.util` package.
- **Collection** → Represents a group of objects (like List, Set, Queue).
- **Collections** → Provides **static methods** to operate on collections (like sort, reverse, shuffle).
- **Collection** → Root of the collection hierarchy.
- **Collections** → A helper class with algorithms for working on collections.

Differentiate between ArrayList vs LinkedList

- **ArrayList →** Uses dynamic array internally.
- **LinkedList →** Uses doubly linked list internally.
- **ArrayList →** Provides fast random access (get by index).
- **LinkedList →** Provides fast insertion and deletion in the middle.
- **ArrayList →** Slower in insertion/deletion because elements need shifting.
- **LinkedList →** Slower in access because it must traverse nodes.
- **ArrayList →** Better when you do more searching/reading.
- **LinkedList →** Better when you do more add/remove operations.

Differentiate between ArrayList vs Array

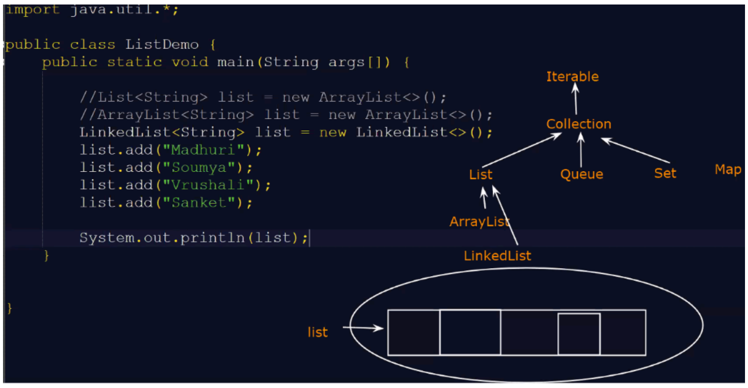
- **ArrayList** → Size is **dynamic**, can grow or shrink.
- **Array** → Size is **fixed**, must be defined at creation.
- **ArrayList** → Can store only **objects** (cannot store primitives directly).
- **Array** → Can store **primitives and objects**.
- **ArrayList** → Provides many inbuilt methods (add, remove, contains, etc.).
- **Array** → Does not provide utility methods.
- **ArrayList** → Type-safe with generics.
- **Array** → Type-safe but limited, no generics.

Differentiate between Static Array vs Array

- **Static Array** → Usually refers to an **array with fixed size** (like in C or low-level Java usage).
- **Array** → General term for an indexed collection of elements.
- **Static Array** → Size cannot be changed once defined.
- **Array** → In Java, always fixed size, but can also mean dynamic structure in higher-level languages.
- **Static Array** → Memory is allocated at **compile time** (in some languages like C).
- **Array (Java)** → Memory allocated at **runtime** but still fixed length.
- **Static Array** → More common in C/C++.
- **Array (Java)** → Object in heap with fixed length property.

If you want to use collections in the code you need to

import java.util.*;



```
import java.util.*;

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

        //List<String> list = new ArrayList<>();
        //ArrayList<String> list = new ArrayList<>();
        LinkedList<String> list = new LinkedList<>();
        list.add("Madhuri");
        list.add("Soumya");
        list.add("Vrushali");
        list.add("Sanket");

        System.out.println(list);
    }


}
```

LinkedList in Java

Reasons why we use LinkedList:

- **Efficient Insertion and Deletion**
  In a LinkedList, adding or removing elements in the middle or beginning is fast because only references (pointers) need to be changed.
  Unlike ArrayList, no shifting of elements is required.
- **Implements Both List and Deque**
  LinkedList can work as a **List** (like ArrayList) and also as a **Deque/Queue**.
  Supports operations like addFirst(), addLast(), removeFirst(), removeLast().
- **Dynamic Size**
  No need to worry about resizing (like arrays).
  Memory is allocated as nodes are added.
- **Good for Frequent Insert/Remove Operations**
  Best suited when you perform more insertions and deletions than searching.
- **Doubly Linked List Implementation**
  Each node points to both previous and next node.
  Makes traversal in both directions possible.

When NOT to use LinkedList:

- If your program requires **fast random access** (get by index), ArrayList is better.
- LinkedList traversal is slower because it has to move node by node.

List in Java

Definition

- `List` is an **interface** in the **java.util package**.
- It is a **child interface of Collection**.
- It represents an **ordered collection** of elements where **duplicates are allowed**.

Key Features of List

- **Ordered collection** – Elements are stored in the same sequence as they are inserted.
- **Indexed access** – Each element can be accessed using its index (like arrays).
- **Allows duplicates** – Multiple elements with the same value are permitted.
- **Null values** – List can store multiple `null` values.

Important Implementations of List

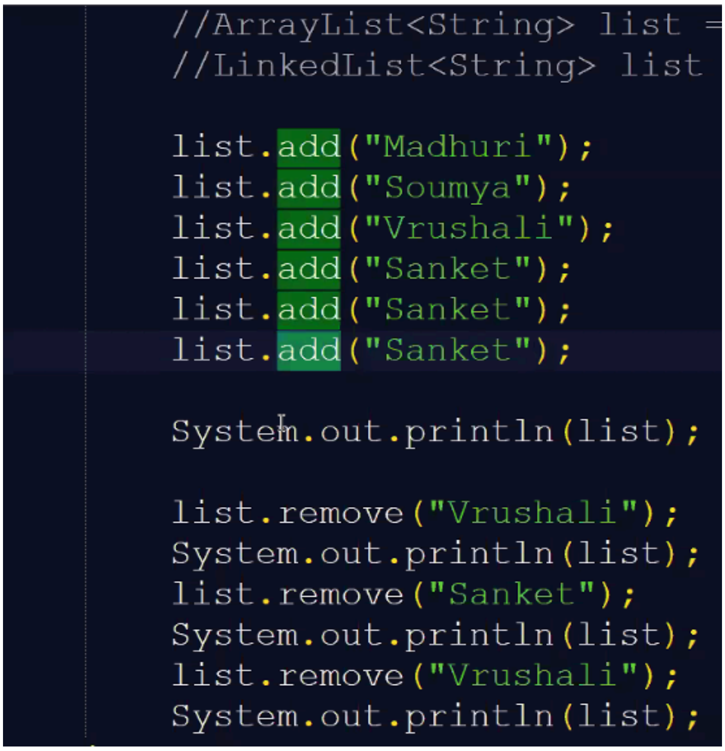
- **ArrayList** – Uses dynamic arrays, fast random access, slow insertion/deletion.
- **LinkedList** – Uses doubly linked list, fast insertion/deletion, slow random access.
- **Vector** – Legacy class, synchronized version of ArrayList.
- **Stack** – Subclass of Vector, follows LIFO (Last In First Out).

Commonly Used Methods in List

- `add(E e)` → adds element
- `add(int index, E e)` → adds at specific index
- `get(int index)` → retrieves element at index
- `set(int index, E e)` → replaces element at index
- `remove(int index)` → removes element at index
- `size()` → returns number of elements
- `contains(Object o)` → checks if element exists
- `clear()` → removes all elements

When to Use List

- When order of elements matters.
- When duplicate values are required.
- When index-based access is needed.



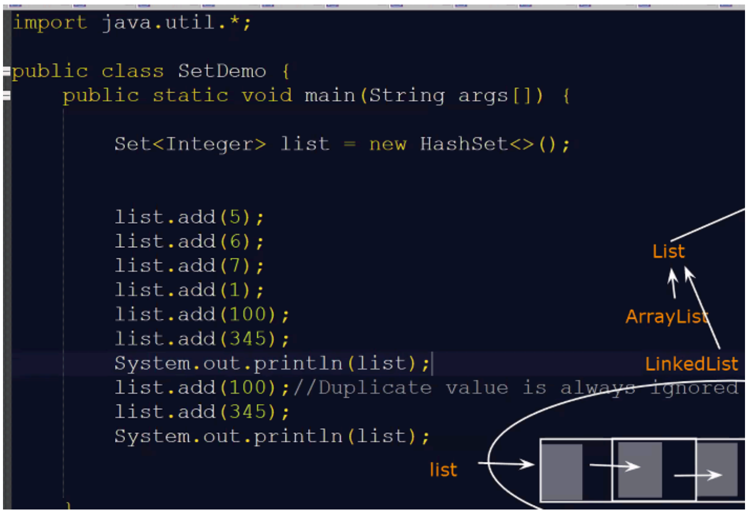
```
        //ArrayList<String> list =
        //LinkedList<String> list

        list.add("Madhuri");
        list.add("Soumya");
        list.add("Vrushali");
        list.add("Sanket");
        list.add("Sanket");
        list.add("Sanket");

        System.out.println(list);

        list.remove("Vrushali");
        System.out.println(list);
        list.remove("Sanket");
        System.out.println(list);
        list.remove("Vrushali");
        System.out.println(list);
```

Set Interface :

- Unordered
- Does not allow duplicate values
- Implementation

1. Hashset : Unordered, fast operations
2. LinkedHashset : Maintains insertion order
3. TreeSet : Sorted order

HashSet h1 = new HashSet();
LinkedHashSet<Float> l1 = new LinkedHashset<>();
TreeSet<String> t1 = new TreeSet();

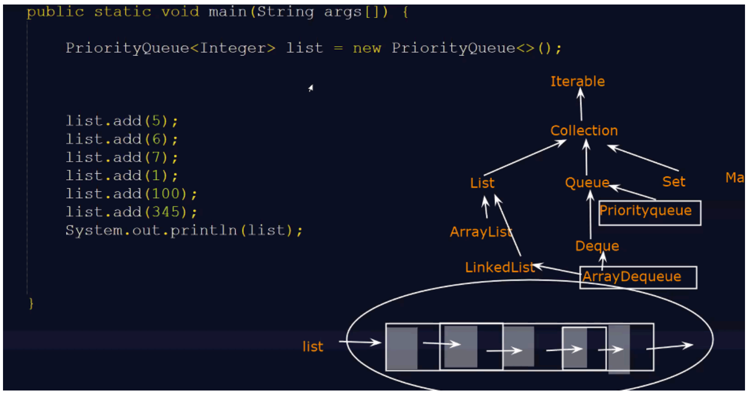


```
import java.util.*;

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

        Set<Integer> list = new HashSet<>();


        list.add(5);
        list.add(6);
        list.add(7);
        list.add(1);
        list.add(100);
        list.add(345);
        System.out.println(list);
        list.add(100);//Duplicate value is always ignored
        list.add(345);
        System.out.println(list);
```

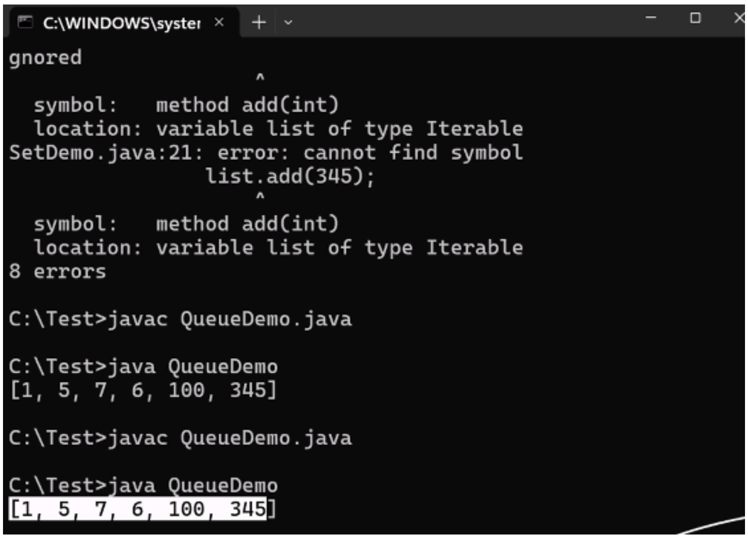
Queue :

- FIFO
- Implimentation
- PriorityQueue : Ordered elements based on priority
- Linkedlist : Deque Implimentation

Queue q1 = new Queue



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

    PriorityQueue<Integer> list = new PriorityQueue<>();


    list.add(5);
    list.add(6);
    list.add(7);
    list.add(1);
    list.add(100);
    list.add(345);
    System.out.println(list);

}
```

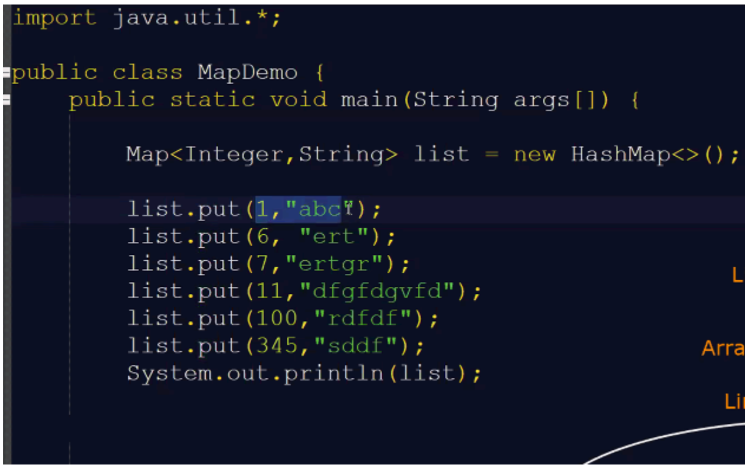


```
gnored
                  ^
  symbol:   method add(int)
  location: variable list of type Iterable
SetDemo.java:21: error: cannot find symbol
                list.add(345);
                  ^
  symbol:   method add(int)
  location: variable list of type Iterable
8 errors

C:\Test>javac QueueDemo.java

C:\Test>java QueueDemo
[1, 5, 7, 6, 100, 345]

C:\Test>javac QueueDemo.java

C:\Test>java QueueDemo
[1, 5, 7, 6, 100, 345]
```

```
C:\Test>
```

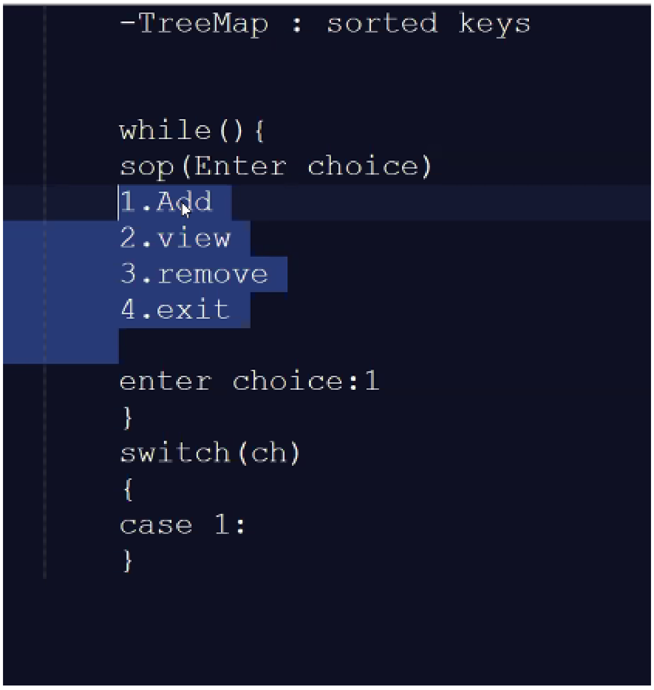
Map Interface : Key-Value Pair :

- It is not a part of collection interface
- It does not allow duplicate keys, but allows duplicate values
- Implementations
  1. HashMap
  2. LinkedMap
  3. TreeMap



```
import java.util.*;

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

        Map<Integer,String> list = new HashMap<>();

        list.put(1,"abc");
        list.put(6, "ert");
        list.put(7,"ertgr");
        list.put(11,"dfgfdgvfd");
        list.put(100,"rdfdf");
        list.put(345,"sddf");
        System.out.println(list);
```

Make Todo App



```
-TreeMap : sorted keys


while(){
sop(Enter choice)
1.Add
2.view
3.remove
4.exit

enter choice:1
}
switch(ch)
{
case 1:
}
```