Java Inner Classes (Nested Classes)

**Java inner class** or nested class is a class that is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.

Additionally, it can access all the members of the outer class, including private data members and methods.

Syntax of Inner class

**class** Java\_Outer\_class{

 //code

**class** Java\_Inner\_class{

  //code

 }

}

Advantage of Java inner classes

There are three advantages of inner classes in Java. They are as follows:

Nested classes represent a particular type of relationship that is **it can access all the members (data members and methods) of the outer class,** including private.

1. Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.
2. **Code Optimization**: It requires less code to write.

Need of Java Inner class

Sometimes users need to program a class in such a way so that no other class can access it. Therefore, it would be better if you include it within other classes.

If all the class objects are a part of the outer object then it is easier to nest that class inside the outer class. That way all the outer class can access all the objects of the inner class.

Difference between nested class and inner class in Java

An inner class is a part of a nested class. Non-static nested classes are known as inner classes.

Types of Nested classes

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

* Non-static nested class (inner class)
  1. Member inner class
  2. Anonymous inner class
  3. Local inner class
* Static nested class

|  |  |
| --- | --- |
| **Type** | **Description** |
| [Member Inner Class](https://www.javatpoint.com/member-inner-class) | A class created within class and outside method. |
| [Anonymous Inner Class](https://www.javatpoint.com/anonymous-inner-class) | A class created for implementing an interface or extending class. The java compiler decides its name. |
| [Local Inner Class](https://www.javatpoint.com/local-inner-class) | A class was created within the method. |
| [Static Nested Class](https://www.javatpoint.com/static-nested-class) | A static class was created within the class. |
| [Nested Interface](https://www.javatpoint.com/nested-interface) | An interface created within class or interface. |

Java Member Inner class

A non-static class that is created inside a class but outside a method is called **member inner class**. It is also known as a **regular inner class**. It can be declared with access modifiers like public, default, private, and protected.

**Syntax:**

**class** Outer{

 //code

**class** Inner{

  //code

 }

}

Java Member Inner Class Example

In this example, we are creating a msg() method in the member inner class that is accessing the private data member of the outer class.

**TestMemberOuter1.java**

**class** TestMemberOuter1{

**private** **int** data=30;

**class** Inner{

**void** msg(){System.out.println("data is "+data);}

 }

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

  TestMemberOuter1 obj=**new** TestMemberOuter1();

  TestMemberOuter1.Inner in=obj.**new** Inner();

  in.msg();

 }

}

**Output:**

data is 30

Java Anonymous inner class

Java anonymous inner class is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain "extras" such as overloading methods of a class or interface, without having to actually subclass a class.

In simple words, a class that has no name is known as an anonymous inner class in Java. It should be used if you have to override a method of class or interface. Java Anonymous inner class can be created in two ways:

1. Class (may be abstract or concrete).
2. Interface

Java anonymous inner class example using class

**TestAnonymousInner.java**

**abstract** **class** Person{

**abstract** **void** eat();

}

**class** TestAnonymousInner{

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

  Person p=**new** Person(){

**void** eat(){System.out.println("nice fruits");}

  };

  p.eat();

 }

}

Java anonymous inner class example using interface

**interface** Eatable{

**void** eat();

}

**class** TestAnnonymousInner1{

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

 Eatable e=**new** Eatable(){

**public** **void** eat(){System.out.println("nice fruits");}

 };

 e.eat();

 }

}

Java Local inner class

A class i.e., created inside a method, is called local inner class in java. Local Inner Classes are the inner classes that are defined inside a block. Generally, this block is a method body. Sometimes this block can be a for loop, or an if clause. Local Inner classes are not a member of any enclosing classes. They belong to the block they are defined within, due to which local inner classes cannot have any access modifiers associated with them. However, they can be marked as final or abstract. These classes have access to the fields of the class enclosing it.

If you want to invoke the methods of the local inner class, you must instantiate this class inside the method.

Java local inner class example

**LocalInner1.java**

**public** **class** localInner1{

**private** **int** data=30;//instance variable

**void** display(){

**class** Local{

**void** msg(){System.out.println(data);}

  }

  Local l=**new** Local();

  l.msg();

 }

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

  localInner1 obj=**new** localInner1();

  obj.display();

 }

}

Java static nested class

A static class is a class that is created inside a class, is called a static nested class in Java. It cannot access non-static data members and methods. It can be accessed by outer class name.

* It can access static data members of the outer class, including private.
* The static nested class cannot access non-static (instance) data members or

Java static nested class example with instance method

**TestOuter1.java**

**class** TestOuter1{

**static** **int** data=30;

**static** **class** Inner{

**void** msg(){System.out.println("data is "+data);}

  }

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

  TestOuter1.Inner obj=**new** TestOuter1.Inner();

  obj.msg();

  }

}

# Java Nested Interface

An interface, i.e., declared within another interface or class, is known as a nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred to by the outer interface or class. It can't be accessed directly.

### Points to remember for nested interfaces

There are given some points that should be remembered by the java programmer.

* The nested interface must be public if it is declared inside the interface, but it can have any access modifier if declared within the class.
* Nested interfaces are declared static

### Syntax of nested interface which is declared within the interface

**interface** interface\_name{

 ...

**interface** nested\_interface\_name{

  ...

 }

}

### Syntax of nested interface which is declared within the class

**class** class\_name{

 ...

**interface** nested\_interface\_name{

  ...

 }

}

### Example of nested interface which is declared within the interface

In this example, we will learn how to declare the nested interface and how we can access it.

**TestNestedInterface1.java**

**interface** Showable{

**void** show();

**interface** Message{

**void** msg();

  }

}

**class** TestNestedInterface1 **implements** Showable.Message{

**public** **void** msg(){System.out.println("Hello nested interface");}

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

  Showable.Message message=**new** TestNestedInterface1();//upcasting here

  message.msg();

 }

}

**Output:**

hello nested interface

# Java Enums

The **Enum in Java** is a data type which contains a fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc. According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.

Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change). The Java enum constants are static and final implicitly. It is available since JDK 1.5.

Enums are used to create our own data type like classes. The **enum** data type (also known as Enumerated Data Type) is used to define an enum in Java. Unlike C/C++, enum in Java is more powerful. Here, we can define an enum either inside the class or outside the class.

Java Enum internally inherits the Enum class, so it cannot inherit any other class, but it can implement many interfaces. We can have fields, constructors, methods, and main methods in Java enum.

## Points to remember for Java Enum

* Enum improves type safety
* Enum can be easily used in switch
* Enum can be traversed
* Enum can have fields, constructors and methods
* Enum may implement many interfaces but cannot extend any class because it internally extends Enum class

### Simple Example of Java Enum

**class** EnumExample1{

//defining the enum inside the class

**public** **enum** Season { WINTER, SPRING, SUMMER, FALL }

//main method

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

//traversing the enum

**for** (Season s : Season.values())

System.out.println(s);

}}

Output:

WINTER

SPRING

SUMMER

FALL

Let us see another example of Java enum where we are using value(), valueOf(), and ordinal() methods of Java enum.

**class** EnumExample1{

//defining enum within class

**public** **enum** Season { WINTER, SPRING, SUMMER, FALL }

//creating the main method

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

//printing all enum

**for** (Season s : Season.values()){

System.out.println(s);

}

System.out.println("Value of WINTER is: "+Season.valueOf("WINTER"));

System.out.println("Index of WINTER is: "+Season.valueOf("WINTER").ordinal());

System.out.println("Index of SUMMER is: "+Season.valueOf("SUMMER").ordinal());

}}

Output:

WINTER

SPRING

SUMMER

FALL

Value of WINTER is: WINTER

Index of WINTER is: 0

Index of SUMMER is: 2

# Collections in Java

[Java Collection Framework](https://www.javatpoint.com/collections-in-java)

[Hierarchy of Collection Framework](https://www.javatpoint.com/collections-in-java#collectionhierarchy)

[Collection interface](https://www.javatpoint.com/collections-in-java#collectionmethods)

[Iterator interface](https://www.javatpoint.com/collections-in-java#collectioniterator)

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](https://www.javatpoint.com/java-arraylist), Vector, [LinkedList](https://www.javatpoint.com/java-linkedlist), [PriorityQueue](https://www.javatpoint.com/java-priorityqueue), HashSet, LinkedHashSet, TreeSet).

#### What is Collection in Java

A Collection represents a single unit of objects, i.e., a group.

#### What is a framework in Java

It provides readymade architecture.

It represents a set of classes and interfaces.

It is optional.

#### What is Collection framework

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

Interfaces and its implementations, i.e., classes

Algorithm

### Hierarchy of Collection Framework

Let us see the hierarchy of Collection framework. The **java.util** package contains all the [classes](https://www.javatpoint.com/object-and-class-in-java) and [interfaces](https://www.javatpoint.com/interface-in-java) for the Collection framework.



### Methods of Collection interface

There are many methods declared in the Collection interface. They are as follows:

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean add(E e) | It is used to insert an element in this collection. |
| 2 | public boolean addAll(Collection<? extends E> c) | It is used to insert the specified collection elements in the invoking collection. |
| 3 | public boolean remove(Object element) | It is used to delete an element from the collection. |
| 4 | public boolean removeAll(Collection<?> c) | It is used to delete all the elements of the specified collection from the invoking collection. |
| 5 | default boolean removeIf(Predicate<? super E> filter) | It is used to delete all the elements of the collection that satisfy the specified predicate. |
| 6 | public boolean retainAll(Collection<?> c) | It is used to delete all the elements of invoking collection except the specified collection. |
| 7 | public int size() | It returns the total number of elements in the collection. |
| 8 | public void clear() | It removes the total number of elements from the collection. |
| 9 | public boolean contains(Object element) | It is used to search an element. |
| 10 | public boolean containsAll(Collection<?> c) | It is used to search the specified collection in the collection. |
| 11 | public Iterator iterator() | It returns an iterator. |
| 12 | public Object[] toArray() | It converts collection into array. |
| 13 | public <T> T[] toArray(T[] a) | It converts collection into array. Here, the runtime type of the returned array is that of the specified array. |
| 14 | public boolean isEmpty() | It checks if collection is empty. |
| 15 | default Stream<E> parallelStream() | It returns a possibly parallel Stream with the collection as its source. |
| 16 | default Stream<E> stream() | It returns a sequential Stream with the collection as its source. |
| 17 | default Spliterator<E> spliterator() | It generates a Spliterator over the specified elements in the collection. |
| 18 | public boolean equals(Object element) | It matches two collections. |
| 19 | public int hashCode() | It returns the hash code number of the collection. |

### Iterator interface

|  |
| --- |
| Iterator interface provides the facility of iterating the elements in a forward direction only. |

#### Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean hasNext() | It returns true if the iterator has more elements otherwise it returns false. |
| 2 | public Object next() | It returns the element and moves the cursor pointer to the next element. |
| 3 | public void remove() | It removes the last elements returned by the iterator. It is less used. |

## Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

Iterator<T> iterator()

It returns the iterator over the elements of type T.

## Collection Interface

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some of the methods of Collection interface are Boolean add ( Object obj), Boolean addAll ( Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

## List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

List <data-type> list1= **new** ArrayList();

List <data-type> list2 = **new** LinkedList();

List <data-type> list3 = **new** Vector();

List <data-type> list4 = **new** Stack();

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

The classes that implement the List interface are given below.

## ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

**import** java.util.\*;

**class** TestJavaCollection1{

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

ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist

list.add("Ravi");//Adding object in arraylist

list.add("Vijay");

list.add("Ravi");

list.add("Ajay");

//Traversing list through Iterator

Iterator itr=list.iterator();

**while**(itr.hasNext()){

System.out.println(itr.next());

}

}

}

Output:

Ravi

Vijay

Ravi

Ajay

## LinkedList

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Consider the following example.

**import** java.util.\*;

**public** **class** TestJavaCollection2{

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

LinkedList<String> al=**new** LinkedList<String>();

al.add("Ravi");

al.add("Vijay");

al.add("Ravi");

al.add("Ajay");

Iterator<String> itr=al.iterator();

**while**(itr.hasNext()){

System.out.println(itr.next());

}

}

}

Output:

Ravi

Vijay

Ravi

Ajay

## Vector

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Consider the following example.

**import** java.util.\*;

**public** **class** TestJavaCollection3{

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

Vector<String> v=**new** Vector<String>();

v.add("Ayush");

v.add("Amit");

v.add("Ashish");

v.add("Garima");

Iterator<String> itr=v.iterator();

**while**(itr.hasNext()){

System.out.println(itr.next());

}

}

}

Output:

Ayush

Amit

Ashish

Garima

## Stack

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Consider the following example.

**import** java.util.\*;

**public** **class** TestJavaCollection4{

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

Stack<String> stack = **new** Stack<String>();

stack.push("Ayush");

stack.push("Garvit");

stack.push("Amit");

stack.push("Ashish");

stack.push("Garima");

stack.pop();

Iterator<String> itr=stack.iterator();

**while**(itr.hasNext()){

System.out.println(itr.next());

}

}

}

Output:

Ayush

Garvit

Amit

Ashish

## Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

Queue<String> q1 = **new** PriorityQueue();

Queue<String> q2 = **new** ArrayDeque();

There are various classes that implement the Queue interface, some of them are given below.

## PriorityQueue

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example.

**import** java.util.\*;

**public** **class** TestJavaCollection5{

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

PriorityQueue<String> queue=**new** PriorityQueue<String>();

queue.add("Amit Sharma");

queue.add("Vijay Raj");

queue.add("JaiShankar");

queue.add("Raj");

System.out.println("head:"+queue.element());

System.out.println("head:"+queue.peek());

System.out.println("iterating the queue elements:");

Iterator itr=queue.iterator();

**while**(itr.hasNext()){

System.out.println(itr.next());

}

queue.remove();

queue.poll();

System.out.println("after removing two elements:");

Iterator<String> itr2=queue.iterator();

**while**(itr2.hasNext()){

System.out.println(itr2.next());

}

}

}

Output:

head:Amit Sharma

head:Amit Sharma

iterating the queue elements:

Amit Sharma

Raj

JaiShankar

Vijay Raj

after removing two elements:

Raj

Vijay Raj

## Deque Interface

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

Deque d = **new** ArrayDeque();

## ArrayDeque

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Consider the following example.

**import** java.util.\*;

**public** **class** TestJavaCollection6{

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

//Creating Deque and adding elements

Deque<String> deque = **new** ArrayDeque<String>();

deque.add("Gautam");

deque.add("Karan");

deque.add("Ajay");

//Traversing elements

**for** (String str : deque) {

System.out.println(str);

}

}

}

Output:

Gautam

Karan

Ajay

## Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of

elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

Set<data-type> s1 = **new** HashSet<data-type>();

Set<data-type> s2 = **new** LinkedHashSet<data-type>();

Set<data-type> s3 = **new** TreeSet<data-type>();

## HashSet

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

Consider the following example.

**import** java.util.\*;

**public** **class** TestJavaCollection7{

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

//Creating HashSet and adding elements

HashSet<String> set=**new** HashSet<String>();

set.add("Ravi");

set.add("Vijay");

set.add("Ravi");

set.add("Ajay");

//Traversing elements

Iterator<String> itr=set.iterator();

**while**(itr.hasNext()){

System.out.println(itr.next());

}

}

}

Output:

Vijay

Ravi

Ajay

## LinkedHashSet

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

**import** java.util.\*;

**public** **class** TestJavaCollection8{

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

LinkedHashSet<String> set=**new** LinkedHashSet<String>();

set.add("Ravi");

set.add("Vijay");

set.add("Ravi");

set.add("Ajay");

Iterator<String> itr=set.iterator();

**while**(itr.hasNext()){

System.out.println(itr.next());

}

}

}

Output:

Ravi

Vijay

Ajay

## SortedSet Interface

SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

SortedSet<data-type> set = **new** TreeSet();

## TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

**import** java.util.\*;

**public** **class** TestJavaCollection9{

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

//Creating and adding elements

TreeSet<String> set=**new** TreeSet<String>();

set.add("Ravi");

set.add("Vijay");

set.add("Ravi");

set.add("Ajay");

//traversing elements

Iterator<String> itr=set.iterator();

**while**(itr.hasNext()){

System.out.println(itr.next());  }   }   }

Output:

Ajay

Ravi

Vijay

# Java Map Interface

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

A Map is useful if you have to search, update or delete elements on the basis of a key.

## Java Map Hierarchy

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap. The hierarchy of Java Map is given below:

Java Map Hierarchy

A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

A Map can't be traversed, so you need to convert it into Set using keySet() or entrySet() method.

|  |  |
| --- | --- |
| **Class** | **Description** |
| [HashMap](https://www.javatpoint.com/java-hashmap) | HashMap is the implementation of Map, but it doesn't maintain any order. |
| [LinkedHashMap](https://www.javatpoint.com/java-linkedhashmap) | LinkedHashMap is the implementation of Map. It inherits HashMap class. It maintains insertion order. |
| [TreeMap](https://www.javatpoint.com/java-treemap) | TreeMap is the implementation of Map and SortedMap. It maintains ascending order. |

## Map.Entry Interface

Entry is the subinterface of Map. So we will be accessed it by Map.Entry name. It returns a collection-view of the map, whose elements are of this class. It provides methods to get key and value.

### Methods of Map.Entry interface

|  |  |
| --- | --- |
| **Method** | **Description** |
| K getKey() | It is used to obtain a key. |
| V getValue() | It is used to obtain value. |
| int hashCode() | It is used to obtain hashCode. |
| V setValue(V value) | It is used to replace the value corresponding to this entry with the specified value. |
| boolean equals(Object o) | It is used to compare the specified object with the other existing objects. |
| static <K extends Comparable<? super K>,V> Comparator<Map.Entry<K,V>> comparingByKey() | It returns a comparator that compare the objects in natural order on key. |
| static <K,V> Comparator<Map.Entry<K,V>> comparingByKey(Comparator<? super K> cmp) | It returns a comparator that compare the objects by key using the given Comparator. |
| static <K,V extends Comparable<? super V>> Comparator<Map.Entry<K,V>> comparingByValue() | It returns a comparator that compare the objects in natural order on value. |
| static <K,V> Comparator<Map.Entry<K,V>> comparingByValue(Comparator<? super V> cmp) | It returns a comparator that compare the objects by value using the given Comparator. |

### Java Map Example: Non-Generic (Old Style)

//Non-generic

**import** java.util.\*;

**public** **class** MapExample1 {

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

    Map map=**new** HashMap();

    //Adding elements to map

    map.put(1,"Amit");

    map.put(5,"Rahul");

    map.put(2,"Jai");

    map.put(6,"Amit");

    //Traversing Map

    Set set=map.entrySet();//Converting to Set so that we can traverse

    Iterator itr=set.iterator();

**while**(itr.hasNext()){

        //Converting to Map.Entry so that we can get key and value separately

        Map.Entry entry=(Map.Entry)itr.next();

        System.out.println(entry.getKey()+" "+entry.getValue());

    }

}

}

Output:

1 Amit

2 Jai

5 Rahul

6 Amit

### Java Map Example: Generic (New Style)

**import** java.util.\*;

**class** MapExample2{

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

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

  map.put(100,"Amit");

  map.put(101,"Vijay");

  map.put(102,"Rahul");

  //Elements can traverse in any order

**for**(Map.Entry m:map.entrySet()){

   System.out.println(m.getKey()+" "+m.getValue());

  }

 }

}

Output:

102 Rahul

100 Amit

101 Vijay

### Java Map Example: comparingByKey()

**import** java.util.\*;

**class** MapExample3{

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

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

      map.put(100,"Amit");

      map.put(101,"Vijay");

      map.put(102,"Rahul");

      //Returns a Set view of the mappings contained in this map

      map.entrySet()

      //Returns a sequential Stream with this collection as its source

      .stream()

      //Sorted according to the provided Comparator

      .sorted(Map.Entry.comparingByKey())

      //Performs an action for each element of this stream

      .forEach(System.out::println);

 }

}

Output:

100=Amit

101=Vijay

102=Rahul

### Java Map Example: comparingByKey() in Descending Order

**import** java.util.\*;

**class** MapExample4{

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

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

      map.put(100,"Amit");

      map.put(101,"Vijay");

      map.put(102,"Rahul");

      //Returns a Set view of the mappings contained in this map

      map.entrySet()

      //Returns a sequential Stream with this collection as its source

      .stream()

      //Sorted according to the provided Comparator

      .sorted(Map.Entry.comparingByKey(Comparator.reverseOrder()))

      //Performs an action for each element of this stream

      .forEach(System.out::println);

 }

}

Output:

102=Rahul

101=Vijay

100=Amit

### Java Map Example: comparingByValue()

**import** java.util.\*;

**class** MapExample5{

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

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

      map.put(100,"Amit");

      map.put(101,"Vijay");

      map.put(102,"Rahul");

      //Returns a Set view of the mappings contained in this map

      map.entrySet()

      //Returns a sequential Stream with this collection as its source

      .stream()

      //Sorted according to the provided Comparator

      .sorted(Map.Entry.comparingByValue())

      //Performs an action for each element of this stream

      .forEach(System.out::println);

 }

}

Output:

100=Amit

102=Rahul

101=Vijay

# Java Regex

The **Java Regex** or Regular Expression is an API to define a pattern for searching or manipulating strings.

It is widely used to define the constraint on strings such as password and email validation.

Java Regex API provides 1 interface and 3 classes in **java.util.regex** package.

#### java.util.regex package

The Matcher and Pattern classes provide the facility of Java regular expression. The java.util.regex package provides following classes and interfaces for regular expressions.

MatchResult interface

Matcher class

Pattern class

PatternSyntaxException class

## Matcher class

It implements the **MatchResult** interface. It is a regex engine which is used to perform match operations on a character sequence.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | boolean matches() | test whether the regular expression matches the pattern. |
| 2 | boolean find() | finds the next expression that matches the pattern. |
| 3 | boolean find(int start) | finds the next expression that matches the pattern from the given start number. |
| 4 | String group() | returns the matched subsequence. |
| 5 | int start() | returns the starting index of the matched subsequence. |
| 6 | int end() | returns the ending index of the matched subsequence. |
| 7 | int groupCount() | returns the total number of the matched subsequence. |

## Pattern class

It is the compiled version of a regular expression. It is used to define a pattern for the regex engine.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | static Pattern compile(String regex) | compiles the given regex and returns the instance of the Pattern. |
| 2 | Matcher matcher(CharSequence input) | creates a matcher that matches the given input with the pattern. |
| 3 | static boolean matches(String regex, CharSequence input) | It works as the combination of compile and matcher methods. It compiles the regular expression and matches the given input with the pattern. |
| 4 | String[] split(CharSequence input) | splits the given input string around matches of given pattern. |
| 5 | String pattern() | returns the regex pattern. |

### Example of Java Regular Expressions

There are three ways to write the regex example in Java.

**import** java.util.regex.\*;

**public** **class** RegexExample1{

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

//1st way

Pattern p = Pattern.compile(".s");//. represents single character

Matcher m = p.matcher("as");

**boolean** b = m.matches();

//2nd way

**boolean** b2=Pattern.compile(".s").matcher("as").matches();

//3rd way

**boolean** b3 = Pattern.matches(".s", "as");

System.out.println(b+" "+b2+" "+b3);

}}

#### Output

true true true

## Regular Expression . Example

The . (dot) represents a single character.

**import** java.util.regex.\*;

**class** RegexExample2{

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

System.out.println(Pattern.matches(".s", "as"));//true (2nd char is s)

System.out.println(Pattern.matches(".s", "mk"));//false (2nd char is not s)

System.out.println(Pattern.matches(".s", "mst"));//false (has more than 2 char)

System.out.println(Pattern.matches(".s", "amms"));//false (has more than 2 char)

System.out.println(Pattern.matches("..s", "mas"));//true (3rd char is s)

}}

## Regex Character classes

|  |  |  |
| --- | --- | --- |
| **No.** | **Character Class** | **Description** |
| 1 | [abc] | a, b, or c (simple class) |
| 2 | [^abc] | Any character except a, b, or c (negation) |
| 3 | [a-zA-Z] | a through z or A through Z, inclusive (range) |
| 4 | [a-d[m-p]] | a through d, or m through p: [a-dm-p] (union) |
| 5 | [a-z&&[def]] | d, e, or f (intersection) |
| 6 | [a-z&&[^bc]] | a through z, except for b and c: [ad-z] (subtraction) |
| 7 | [a-z&&[^m-p]] | a through z, and not m through p: [a-lq-z](subtraction) |

## Regular Expression Character classes Example

**import** java.util.regex.\*;

**class** RegexExample3{

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

System.out.println(Pattern.matches("[amn]", "abcd"));//false (not a or m or n)

System.out.println(Pattern.matches("[amn]", "a"));//true (among a or m or n)

System.out.println(Pattern.matches("[amn]", "ammmna"));//false (m and a comes more than once)

}}

## Regex Quantifiers

The quantifiers specify the number of occurrences of a character.

|  |  |
| --- | --- |
| **Regex** | **Description** |
| X? | X occurs once or not at all |
| X+ | X occurs once or more times |
| X\* | X occurs zero or more times |
| X{n} | X occurs n times only |
| X{n,} | X occurs n or more times |
| X{y,z} | X occurs at least y times but less than z times |

## Regular Expression Character classes and Quantifiers Example

**import** java.util.regex.\*;

**class** RegexExample4{

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

System.out.println("? quantifier ....");

System.out.println(Pattern.matches("[amn]?", "a"));//true (a or m or n comes one time)

System.out.println(Pattern.matches("[amn]?", "aaa"));//false (a comes more than one time)

System.out.println(Pattern.matches("[amn]?", "aammmnn"));//false (a m and n comes more than one time)

System.out.println(Pattern.matches("[amn]?", "aazzta"));//false (a comes more than one time)

System.out.println(Pattern.matches("[amn]?", "am"));//false (a or m or n must come one time)

System.out.println("+ quantifier ....");

System.out.println(Pattern.matches("[amn]+", "a"));//true (a or m or n once or more times)

System.out.println(Pattern.matches("[amn]+", "aaa"));//true (a comes more than one time)

System.out.println(Pattern.matches("[amn]+", "aammmnn"));//true (a or m or n comes more than once)

System.out.println(Pattern.matches("[amn]+", "aazzta"));//false (z and t are not matching pattern)

System.out.println("\* quantifier ....");

System.out.println(Pattern.matches("[amn]\*", "ammmna"));//true (a or m or n may come zero or more times)

}}

## Regex Metacharacters

The regular expression metacharacters work as shortcodes.

|  |  |
| --- | --- |
| **Regex** | **Description** |
| . | Any character (may or may not match terminator) |
| \d | Any digits, short of [0-9] |
| \D | Any non-digit, short for [^0-9] |
| \s | Any whitespace character, short for [\t\n\x0B\f\r] |
| \S | Any non-whitespace character, short for [^\s] |
| \w | Any word character, short for [a-zA-Z\_0-9] |
| \W | Any non-word character, short for [^\w] |
| \b | A word boundary |
| \B | A non word boundary |

## Regular Expression Metacharacters Example

**import** java.util.regex.\*;

**class** RegexExample5{

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

System.out.println("metacharacters d....");\\d means digit

System.out.println(Pattern.matches("\\d", "abc"));//false (non-digit)

System.out.println(Pattern.matches("\\d", "1"));//true (digit and comes once)

System.out.println(Pattern.matches("\\d", "4443"));//false (digit but comes more than once)

System.out.println(Pattern.matches("\\d", "323abc"));//false (digit and char)

System.out.println("metacharacters D....");\\D means non-digit

System.out.println(Pattern.matches("\\D", "abc"));//false (non-digit but comes more than once)

System.out.println(Pattern.matches("\\D", "1"));//false (digit)

System.out.println(Pattern.matches("\\D", "4443"));//false (digit)

System.out.println(Pattern.matches("\\D", "323abc"));//false (digit and char)

System.out.println(Pattern.matches("\\D", "m"));//true (non-digit and comes once)

System.out.println("metacharacters D with quantifier....");

System.out.println(Pattern.matches("\\D\*", "mak"));//true (non-digit and may come 0 or more times)

}}

## Regular Expression Question 1

/\*Create a regular expression that accepts alphanumeric characters only.  Its length must be six characters long only.\*/

**import** java.util.regex.\*;

**class** RegexExample6{

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

System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun32"));//true

System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "kkvarun32"));//false (more than 6 char)

System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "JA2Uk2"));//true

System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun$2"));//false ($ is not matched)

}}

## Regular Expression Question 2

/\*Create a regular expression that accepts 10 digit numeric characters

 starting with 7, 8 or 9 only.\*/

**import** java.util.regex.\*;

**class** RegexExample7{

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

System.out.println("by character classes and quantifiers ...");

System.out.println(Pattern.matches("[789]{1}[0-9]{9}", "9953038949"));//true

System.out.println(Pattern.matches("[789][0-9]{9}", "9953038949"));//true

System.out.println(Pattern.matches("[789][0-9]{9}", "99530389490"));//false (11 characters)

System.out.println(Pattern.matches("[789][0-9]{9}", "6953038949"));//false (starts from 6)

System.out.println(Pattern.matches("[789][0-9]{9}", "8853038949"));//true

System.out.println("by metacharacters ...");

System.out.println(Pattern.matches("[789]{1}\\d{9}", "8853038949"));//true

System.out.println(Pattern.matches("[789]{1}\\d{9}", "3853038949"));//false (starts from 3)

}}

## Java Regex Finder Example

**import** java.util.regex.Pattern;

**import** java.util.Scanner;

**import** java.util.regex.Matcher;

**public** **class** RegexExample8{

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

        Scanner sc=**new** Scanner(System.in);

**while** (**true**) {

            System.out.println("Enter regex pattern:");

            Pattern pattern = Pattern.compile(sc.nextLine());

            System.out.println("Enter text:");

            Matcher matcher = pattern.matcher(sc.nextLine());

**boolean** found = **false**;

**while** (matcher.find()) {

                System.out.println("I found the text "+matcher.group()+" starting at index "+

                 matcher.start()+" and ending at index "+matcher.end());

                found = **true**;

            }

**if**(!found){

                System.out.println("No match found.");

            }

        }

    }

}

Output:

Enter regex pattern: java

Enter text: this is java, do you know java

I found the text java starting at index 8 and ending at index 12

I found the text java starting at index 26 and ending at index 30

Varargs:

Java supports variable length arguments function means we can pass any no. of args into the function.

Syntax:

Returntype funcname(type... varname)

{

………….

}

-> we can specify only one arg as variable length arg.

-> Only last arg should be variable length arg.