UNIT-5

Prepared By:
Deepak Kumar Sharma
Asst. Professor
SoCS UPES Dehradun

UNIT-V

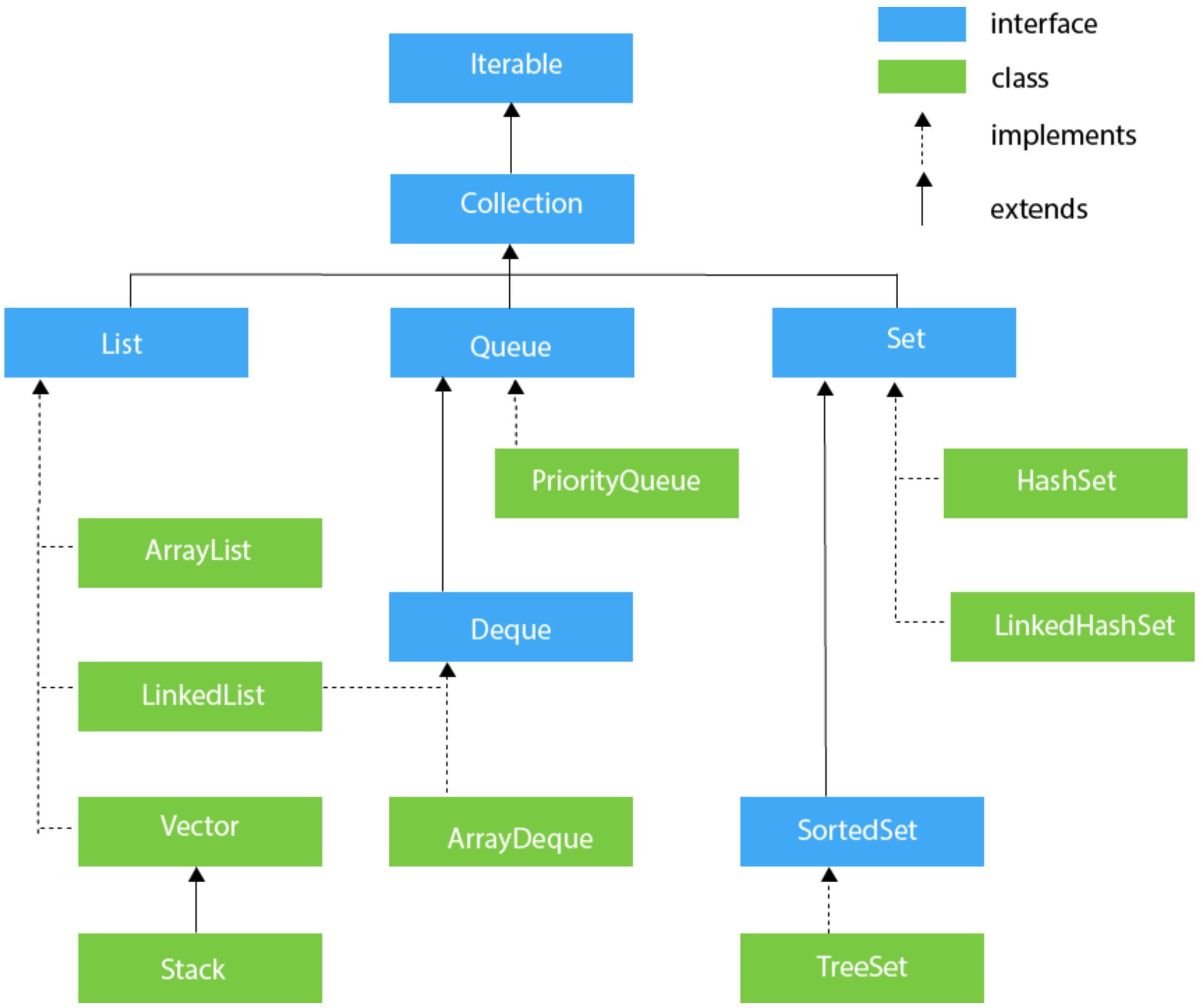
Collections, Iteration, collection interface, Set and SortedSet
List, Map and SortedMap
Wrapped Collections and Collections Class, Arrays, Legacy Collection, Properties
Structure of JDBC program, JDBC: Types of drivers, driver manager class, characteristic, components
Database connectivity, JDBC statement: prepared, callable
Types of result set, Inserting and updating records

Collections

- 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).

Hierarchy of Collection Framework

- The **java.util** package contains all the classes and interfaces for the Collection framework.



Methods of Collection interface

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.
- There are only three methods in the Iterator interface.

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

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();
```

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.

```java
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.

```
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());
}
}
}
```

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.

```java
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.

```
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();
```

PriorityQueue

- 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.

```
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();//returns and removes the element at the front end
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.
- 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.

```java
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.

```
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.

```java
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.

```
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 List

Create a List

- You create a List instance by creating an instance of one of the classes that implements the List interface.
- Here are a few examples of how to create a `List` instance:

```
List listA = new ArrayList();
List listB = new LinkedList();
List listC = new Vector();
List listD = new Stack();
```

- Remember, most often you will use the ArrayList class, but there can be cases where using one of the other implementations might make sense.

Java ArrayList

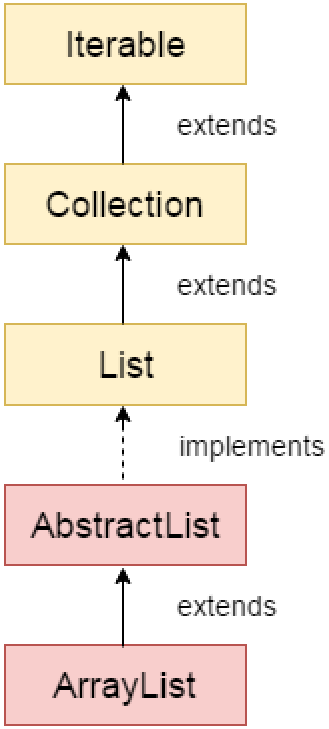
- Java **ArrayList** class uses a *dynamic array* for storing the elements.
- It is like an array, but there is *no size limit*.
- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because the array works on an index basis.
- In ArrayList, manipulation is a little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.

Java ArrayList

- We can not create an array list of the primitive types, such as int, float, char, etc.
- It is required to use the required wrapper class in such cases.

```
ArrayList<int> al = ArrayList<int>(); // does not work

ArrayList<Integer> al = new ArrayList<Integer>(); // works fine
```



Non-generic Vs. Generic Collection

- Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.
- Java new generic collection allows you to have only one type of object in a collection. Now it is type-safe, so typecasting is not required at runtime.

Non-generic example of creating a Java collection:

```
ArrayList list=new ArrayList();//creating old non-generic arraylist
```

generic example of creating java collection:

```
ArrayList<String> list=new ArrayList<String>();//creating new generic arraylist
```

- In a generic collection, we specify the type in angular braces.
- Now ArrayList is forced to have the only specified type of object in it.

ArrayList Example

```
import java.util.*;
 public class ArrayListExample1{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();//Creating arraylist
      list.add("Mango");//Adding object in arraylist
      list.add("Apple");
      list.add("Banana");
      list.add("Grapes");
      //Printing the arraylist object
      System.out.println(list);
 }
}
```

Output:

```
[Mango, Apple, Banana, Grapes]
```

Insert Elements in a Java List

```
List<String> listA = new ArrayList<>();

listA.add("element 1");
listA.add("element 2");
listA.add("element 3");
```

- The first three **add()** calls add a String instance to the end of the list.

Insert null Values

```
Object element = null;

List<Object> list = new ArrayList<>();

list.add(element);
```

Insert Elements at Specific Index

- inserting an element at index 0 into a Java List

```
list.add(0, "element 4");
```

Insert All Elements From One List Into Another

- List addAll() method
- example adds all elements from listSource into listDest

```
List<String> listSource = new ArrayList<>();
listSource.add("123");
listSource.add("456");
List<String> listDest   = new ArrayList<>();
listDest.addAll(listSource);
```

Note: can add all elements from a List or Set into a List with **addAll()**

Get Elements From a Java List

- get(int index) method

```
List<String> listA = new
ArrayList<>();

listA.add("element 0");
listA.add("element 1");
listA.add("element 2");

//access via index
String element0 = listA.get(0);
String element1 = listA.get(1);
String element3 = listA.get(2);
```

Find Elements in a List

- indexOf()
- lastIndexOf()

```
List<String> list = new ArrayList<>();

String element1 = "element 1";
String element2 = "element 2";

list.add(element1);
list.add(element2);

int index1 = list.indexOf(element1);
int index2 = list.indexOf(element2);

System.out.println("index1 = " + index1);
System.out.println("index2 = " + index2);
```

output:

```
index1 = 0
index2 = 1
```

Find Last Occurrence of Element in a List

- lastIndexOf() method

```
List<String> list = new ArrayList<>();


String element1 = "element 1";
String element2 = "element 2";


list.add(element1);
list.add(element2);
list.add(element1);


int lastIndex = list.lastIndexOf(element1);
System.out.println("lastIndex = " + lastIndex);
```

O/P:

```
lastIndex = 2
```

Checking if List Contains Element

- contains() method

```
List<String> list = new ArrayList<>();

String element1 = "element 1";

list.add(element1);
boolean containsElement = list.contains("element 1");

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

```
O/P:
true
```

Remove Elements From a Java List

2 Methods:

  - remove(Object element)
  - remove(int index)

- Removes that element in the list, if it is present.
- All subsequent elements in the list are then moved up in the list.

```
List<String> list = new ArrayList<>();
String element = "first element";
list.add(element);
list.remove(element);
```

```
List<String> list = new ArrayList<>();
list.add("element 0");
list.add("element 1");
list.add("element 2");
list.remove(0);//remove(int index)
```

Remove All Elements From a Java List

- clear() method which removes all elements from the list.

```
List<String> list = new ArrayList<>();
list.add("object 1");
list.add("object 2");
//etc.

list.clear(); //will make list empty
```

Retain All Elements From One List in Another

retainAll()

- removes all the elements from the target List which are not found in the other List.
- The resulting List is the ***intersection*** of the two lists

```
List<String> list      = new ArrayList<>();
List<String> otherList = new ArrayList<>();
String element1 = "element 1";
String element2 = "element 2";
String element3 = "element 3";
String element4 = "element 4";
list.add(element1);
list.add(element2);
list.add(element3);
otherList.add(element1);
otherList.add(element3);
otherList.add(element4);
list.retainAll(otherList);
```

O/P:
List will have element1 and element3

List Size

- **size() method:** number of elements

```
List<String> list = new ArrayList<>();
list.add("object 1");
list.add("object 2");

int size = list.size();
```

Convert List to Set

- Convert list to set using addall() method.
- Will remove all duplicate elements

```
List<String> list      = new ArrayList<>();

list.add("element 1");
list.add("element 2");
list.add("element 3");
list.add("element 3");

Set<String> set = new HashSet<>();
set.addAll(list); //
```

Convert List to Array

- toArray() method

```
List<String> list       = new ArrayList<>();

list.add("element 1");
list.add("element 2");
list.add("element 3");
list.add("element 3");

Object[] objects = list.toArray();
```

Convert Array to List

- **Arrays.asList()** method : converts the array to a List

```
String[] values = new String[]{ "one", "two", "three" };

List<String> list = (List<String>) Arrays.asList(values);
```

Sort List

- Collections sort() method

```
List<String> list = new ArrayList<>();

list.add("c");
list.add("b");
list.add("a");

Collections.sort(list);
```

Iterate List

- Using an Iterator
- Using a for-each loop
- Using a for loop
- Using the Java Stream API

Iterate List Using Iterator

```java
List<String> list = new ArrayList<>();

list.add("first");
list.add("second");
list.add("third");

Iterator<String> iterator = list.iterator();
while(iterator.hasNext()) {
    String next = iterator.next();
}
```

Iterate List Using For-Each Loop

```
List<String> list = new ArrayList<String>();

//add elements to list

for(String element : list) {
    System.out.println(element);
}
```

Iterate List Using For Loop

```
List list = new ArrayList();

list.add("first");
list.add("second");
list.add("third");

for(int i=0; i < list.size(); i++) {
    Object element = list.get(i);
}
```

```
List<String> list = new ArrayList<String>();

list.add("first");
list.add("second");
list.add("third");

for(int i=0; i < list.size(); i++) {
    String element = list.get(i);
}
```

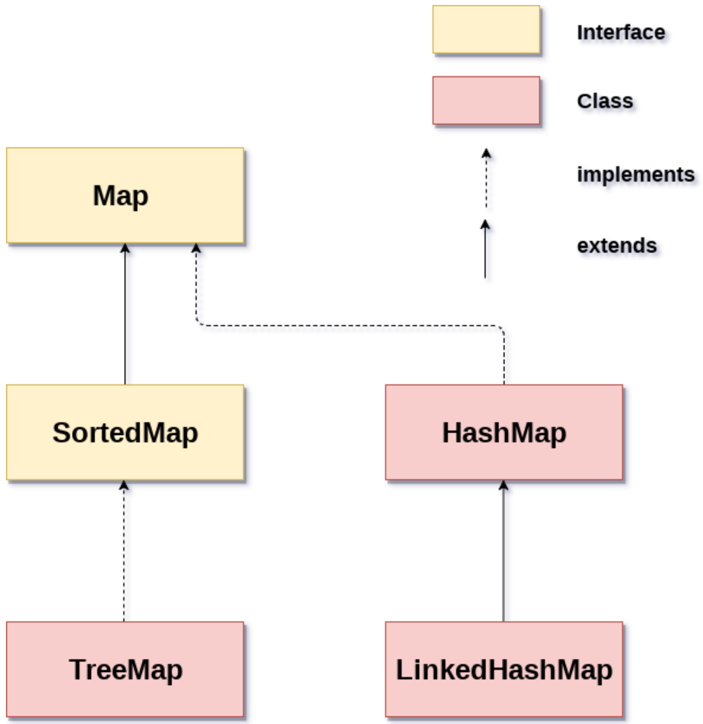
Java Map

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

- 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 .



Useful methods of Map interface

Method	Description
V put(Object key, Object value)	It is used to insert an entry in the map.
void putAll(Map map)	It is used to insert the specified map in the map.
V putIfAbsent(K key, V value)	It inserts the specified value with the specified key in the map only if it is not already specified.
V remove(Object key)	It is used to delete an entry for the specified key.
boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the map.
Set keySet()	It returns the Set view containing all the keys.
Set<Map.Entry<K,V>> entrySet()	It returns the Set view containing all the keys and values.
void clear()	It is used to reset the map.

Useful methods of Map interface

boolean containsValue(Object value)	This method returns true if some value equal to the value exists within the map, else return false.
boolean containsKey(Object key)	This method returns true if some key equal to the key exists within the map, else return false.
boolean equals(Object o)	It is used to compare the specified Object with the Map.
void forEach(BiConsumer<? super K,? super V> action)	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
V get(Object key)	This method returns the object that contains the value associated with the key.
V getOrDefault(Object key, V defaultValue)	It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.
int hashCode()	It returns the hash code value for the Map
boolean isEmpty()	This method returns true if the map is empty; returns false if it contains at least one key.

Map.Entry Interface

- Entry is the subinterface of Map.
- So we will access 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: Generic

```
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 HashMap

- Java HashMap contains values based on the key.
- Java HashMap contains only unique keys.
- Java HashMap may have one null key and multiple null values.
- Java HashMap is non synchronized.
- Java HashMap maintains no order.
- The initial default capacity of Java HashMap class is 16

HashMap class Parameters

Let's see the Parameters for java.util.HashMap class.

- **K**: It is the type of keys maintained by this map.
- **V**: It is the type of mapped values.

Constructor	Description
HashMap()	It is used to construct a default HashMap.
HashMap(Map<? extends K,? extends V> m)	It is used to initialize the hash map by using the elements of the given Map object m.
HashMap(int capacity)	It is used to initializes the capacity of the hash map to the given integer value, capacity.
HashMap(int capacity, float loadFactor)	It is used to initialize both the capacity and load factor of the hash map by using its arguments.

Methods of Java HashMap class

Method	Description
void clear()	It is used to remove all of the mappings from this map.
boolean isEmpty()	It is used to return true if this map contains no key-value mappings.
Object clone()	It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
Set entrySet()	It is used to return a collection view of the mappings contained in this map.
Set keySet()	It is used to return a set view of the keys contained in this map.
V put(Object key, Object value)	It is used to insert an entry in the map.
void putAll(Map map)	It is used to insert the specified map in the map.
V putIfAbsent(K key, V value)	It inserts the specified value with the specified key in the map only if it is not already specified.
V remove(Object key)	It is used to delete an entry for the specified key.
boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the map.
V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)	It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null.
V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null.

boolean containsValue(Object value)	This method returns true if some value equal to the value exists within the map, else return false.
boolean containsKey(Object key)	This method returns true if some key equal to the key exists within the map, else return false.
boolean equals(Object o)	It is used to compare the specified Object with the Map.
void forEach(BiConsumer<? super K,? super V> action)	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
V get(Object key)	This method returns the object that contains the value associated with the key.
V getOrDefault(Object key, V defaultValue)	It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.
boolean isEmpty()	This method returns true if the map is empty; returns false if it contains at least one key.
V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
V replace(K key, V value)	It replaces the specified value for a specified key.
boolean replace(K key, V oldValue, V newValue)	It replaces the old value with the new value for a specified key.
void replaceAll(BiFunction<? super K,? super V,? extends V> function)	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
Collection<V> values()	It returns a collection view of the values contained in the map.
int size()	This method returns the number of entries in the map.

Java HashMap Example

```java
import java.util.*;
public class HashMapExample1{
 public static void main(String args[]){
   HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating HashMap
   map.put(1,"Mango");  //Put elements in Map
   map.put(2,"Apple");
   map.put(3,"Banana");
   map.put(4,"Grapes");

   System.out.println("Iterating Hashmap...");
   for(Map.Entry m : map.entrySet()){
    System.out.println(m.getKey()+" "+m.getValue());
   }
}
}
```

```
Iterating Hashmap...
1 Mango
2 Apple
3 Banana
4 Grapes
```

Note: You cannot store duplicate keys in HashMap.
However, if you try to store duplicate key with another value, it will replace the value.

Java HashMap example to add() elements

```
import java.util.*;
class HashMap1{
 public static void main(String args[]){
   HashMap<Integer,String> hm=new HashMap<Integer,String>();
    System.out.println("Initial list of elements: "+hm);
     hm.put(100,"Amit");
     hm.put(101,"Vijay");
     hm.put(102,"Rahul");
     System.out.println("After invoking put() method ");
     for(Map.Entry m:hm.entrySet()){
      System.out.println(m.getKey()+" "+m.getValue());
     }

     hm.putIfAbsent(103, "Gaurav");
     System.out.println("After invoking putIfAbsent() method ");
     for(Map.Entry m:hm.entrySet()){
         System.out.println(m.getKey()+" "+m.getValue());
        }
     HashMap<Integer,String> map=new HashMap<Integer,String>();
     map.put(104,"Ravi");
     map.putAll(hm);
     System.out.println("After invoking putAll() method ");
     for(Map.Entry m:map.entrySet()){
         System.out.println(m.getKey()+" "+m.getValue());
        }
 }
}
```

```
Initial list of elements: {}
After invoking put() method
100 Amit
101 Vijay
102 Rahul
After invoking putIfAbsent() method
100 Amit
101 Vijay
102 Rahul
103 Gaurav
After invoking putAll() method
100 Amit
101 Vijay
102 Rahul
103 Gaurav
104 Ravi
```

Java HashMap example to remove() elements

```
import java.util.*;
public class HashMap2 {
   public static void main(String args[]) {
    HashMap<Integer,String> map=new HashMap<Integer,String>();
      map.put(100,"Amit");
      map.put(101,"Vijay");
      map.put(102,"Rahul");
      map.put(103, "Gaurav");
    System.out.println("Initial list of elements: "+map);
    //key-based removal
    map.remove(100);
    System.out.println("Updated list of elements: "+map);

  //key-value pair based removal
    map.remove(102, "Rahul");
    System.out.println("Updated list of elements: "+map);
   }
}
```

Difference between HashSet and HashMap

- HashSet contains only values whereas HashMap contains an entry(key and value).

HashSet

- Java HashSet class is used to create a collection that uses a hash table for storage.
- HashSet stores the elements by using a mechanism called **hashing.**
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.
- HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
- HashSet is the best approach for search operations.
- The initial default capacity of HashSet is 16

Constructors of Java HashSet class

SN	Constructor	Description
1)	HashSet()	It is used to construct a default HashSet.
2)	HashSet(int capacity)	It is used to initialize the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.
3)	HashSet(int capacity, float loadFactor)	It is used to initialize the capacity of the hash set to the given integer value capacity and the specified load factor.
4)	HashSet(Collection<? extends E> c)	It is used to initialize the hash set by using the elements of the collection c.

Methods of Java HashSet class

SN	Modifier & Type	Method	Description
1)	boolean	add(E e)	It is used to add the specified element to this set if it is not already present.
2)	void	clear()	It is used to remove all of the elements from the set.
3)	object	clone()	It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned.
4)	boolean	contains(Object o)	It is used to return true if this set contains the specified element.
5)	boolean	isEmpty()	It is used to return true if this set contains no elements.
6)	Iterator<E>	iterator()	It is used to return an iterator over the elements in this set.
7)	boolean	remove(Object o)	It is used to remove the specified element from this set if it is present.
8)	int	size()	It is used to return the number of elements in the set.
9)	Spliterator<E>	spliterator()	It is used to create a late-binding and fail-fast Spliterator over the elements in the set.

Java HashSet Example

```
import java.util.*;
class HashSet1{
 public static void main(String args[]){
  //Creating HashSet and adding elements
    HashSet<String> set=new HashSet();
           set.add("One");
           set.add("Two");
           set.add("Three");
           set.add("Four");
           set.add("Five");
           Iterator<String> i=set.iterator();
           while(i.hasNext())
           {
           System.out.println(i.next());
           }
 }
}
```

```
Five
One
Four
Two
Three
```

Java HashSet example ignoring duplicate elements

```
import java.util.*;
class HashSet2{
 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());
  }
 }
```

```
Ajay
Vijay
Ravi
```

Java HashSet from another Collection

```
import java.util.*;
class HashSet4{
 public static void main(String args[]){
   ArrayList<String> list=new ArrayList<String>();
          list.add("Ravi");
          list.add("Vijay");
          list.add("Ajay");

          HashSet<String> set=new HashSet(list);
          set.add("Gaurav");
          Iterator<String> i=set.iterator();
          while(i.hasNext())
          {
          System.out.println(i.next());
          }
 }
}
```

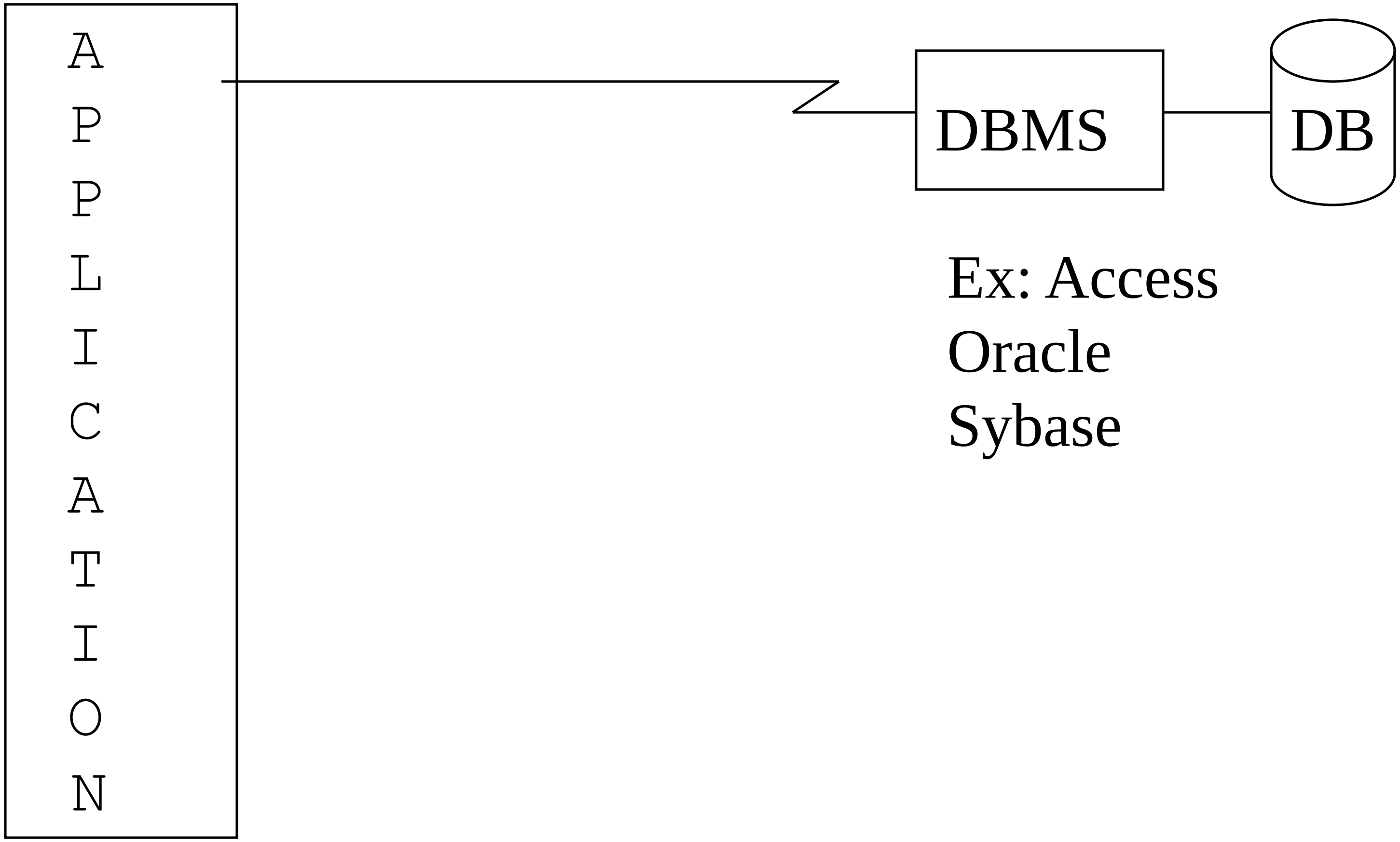
```
Vijay
Ravi
Gaurav
Ajay
```

JDBC

Introduction

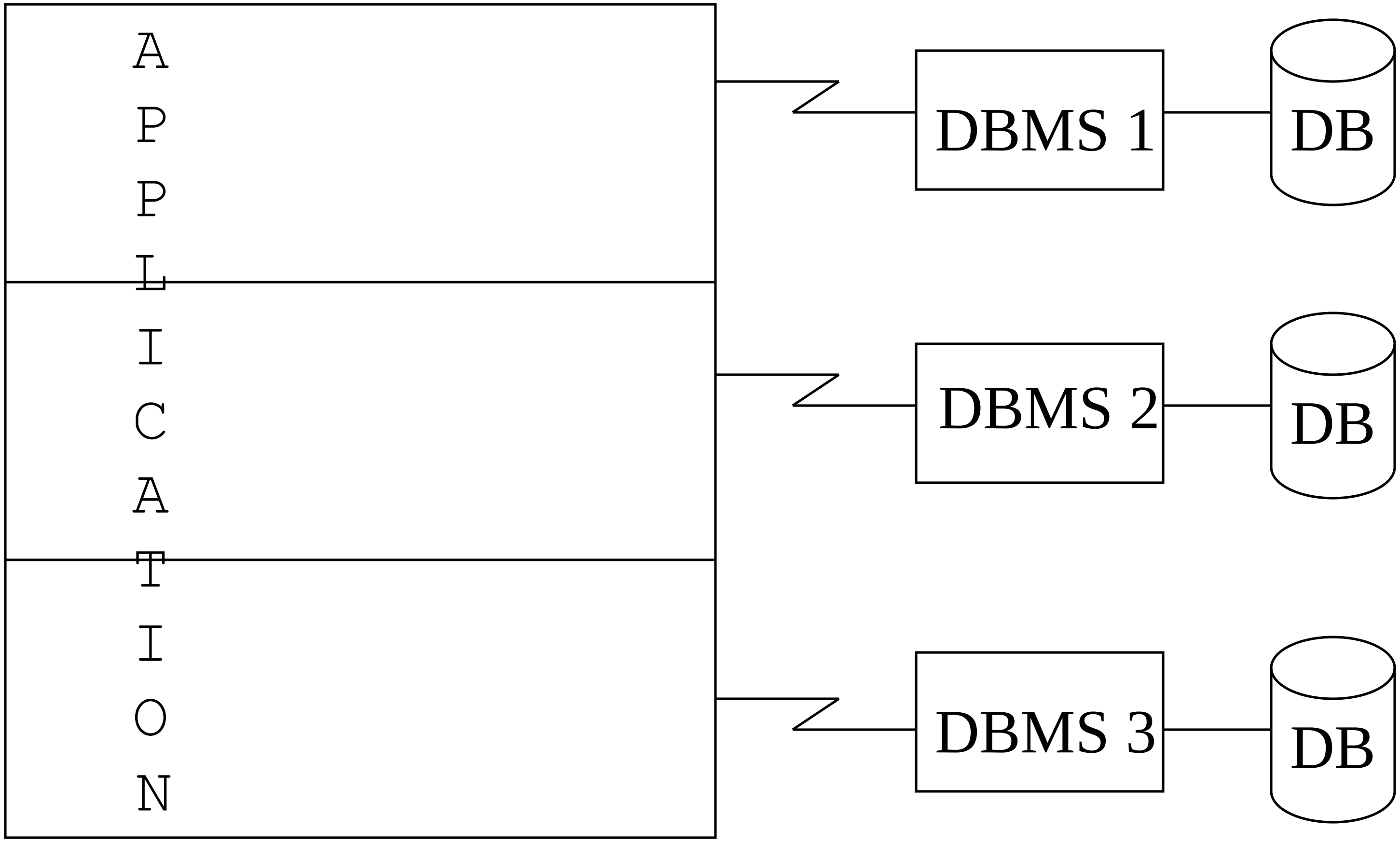
- Most popular form of database system is the relational database system.
- Examples: MS Access, Sybase, Oracle, MySQL.
- Structured Query Language (SQL) is used among relational databases to construct queries.
- These queries can be stand-alone or embedded within applications. This form of SQL is known as embedded SQL.

Simple Database Application

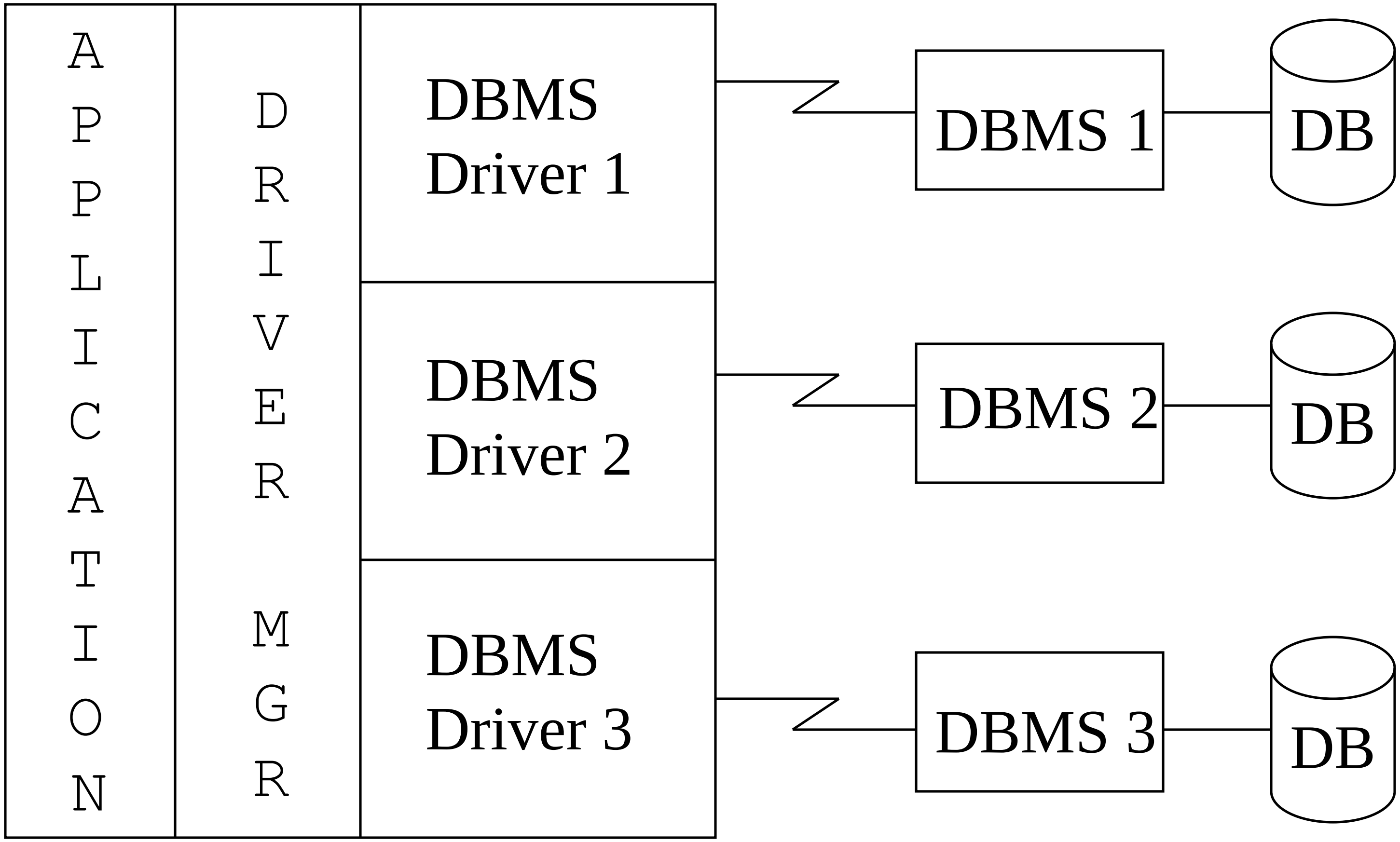


Ex: Access
Oracle
Sybase

Multi-Databases



Standard Access to DB



Design History

History:

- 1995 Sun starts working on standard Java library for SQL access
- Initially the idea was to **extend Java in such a way that pure Java can access any database**
- Fundamental Problem: **too many** databases with corresponding number of **proprietary protocols**.
- All database vendors agreed on a common protocol as long as it was **their** own protocol.

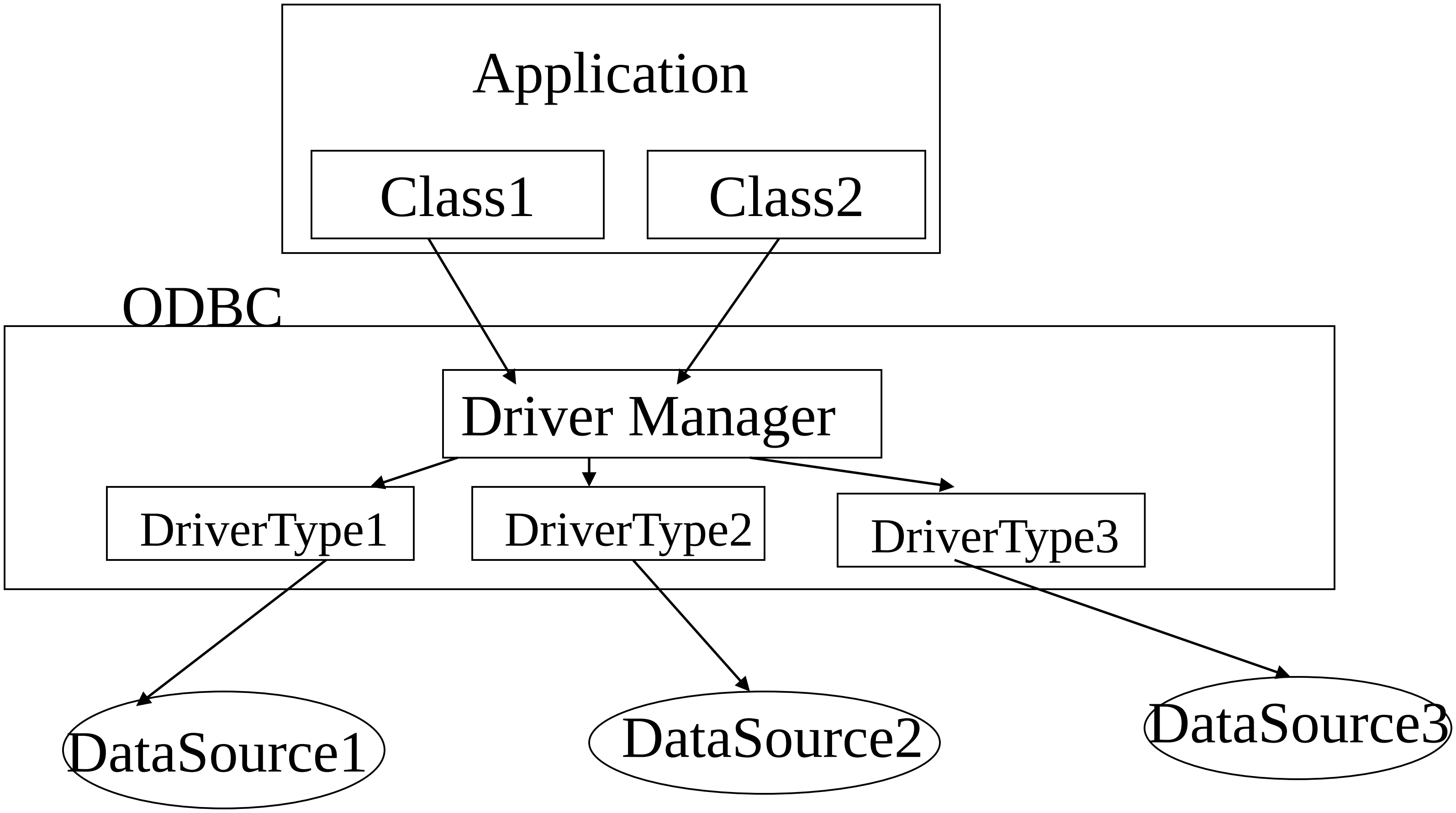
Microsoft's ODBC model:

- C programming language interface for database access;
- Programs talk to the ODBC driver manager that in turn talks to drivers that are plugged into it, and these drivers talk to the database.

Open Database Connectivity (ODBC) Standard

- ODBC standard is an interface by which application programs can access and process SQL databases in a DBMS-independent manner. It contains:
- A **Data Source** that is the database, its associated DBMS, operating system and network platform
- A **DBMS Driver** that is supplied by the DBMS vendor or independent software companies
- A **Driver Manager** that is supplied by the vendor of the O/S platform where the application is running

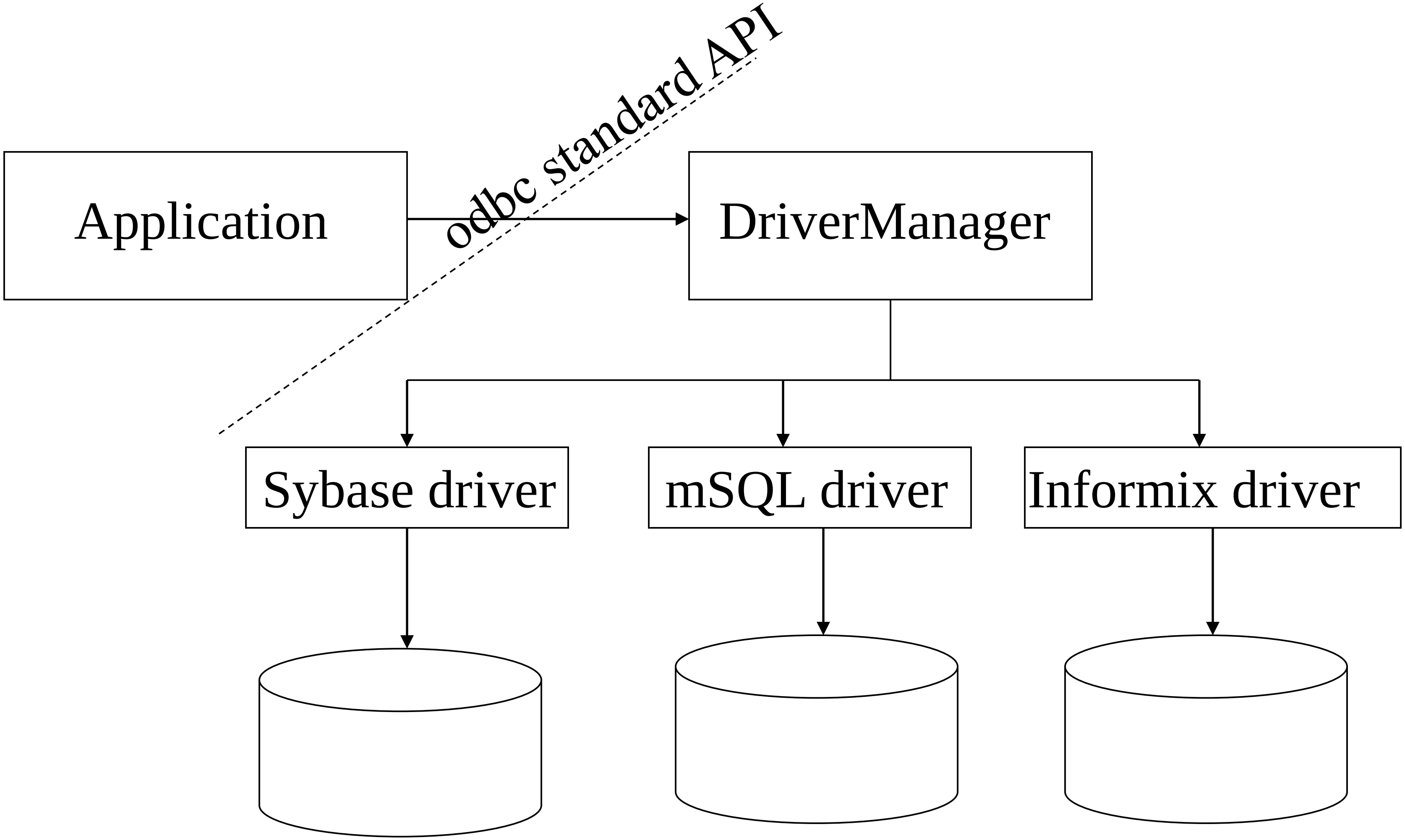
ODBC Architecture



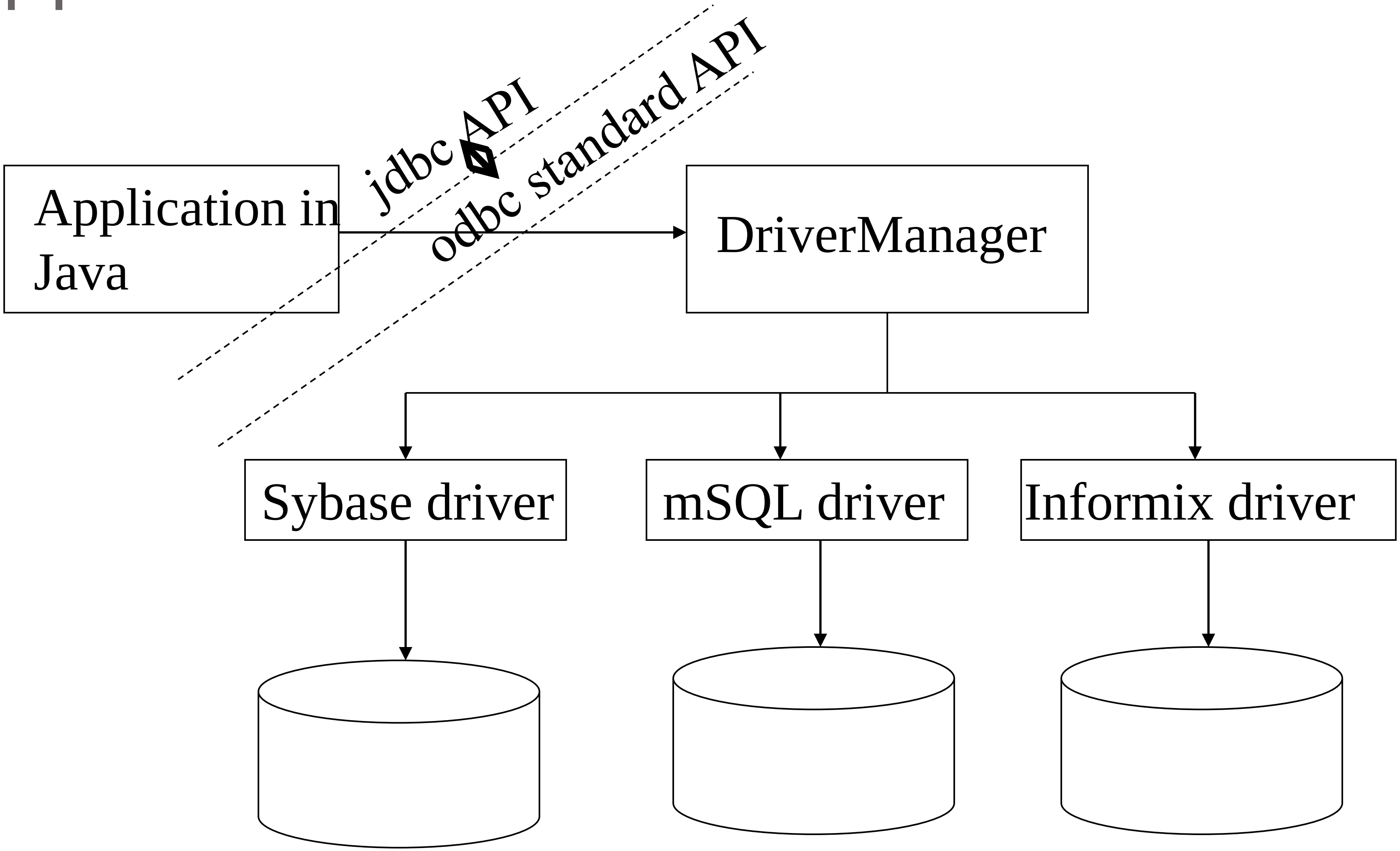
ODBC Interface

- It is a system independent interface to database environment that requires an ODBC driver to be provided for each database system from which you want to manipulate data.
- The database driver bridges the differences between your underlying system calls and the ODBC interface functionality.

An Example


odbc standard API
Application
DriverManager
Sybase driver
mSQL driver
Informix driver

Application in Java



Java Support for ODBC : JDBC

- When applications written in Java want to access data sources, they use classes and associated methods provided by Java DBC (JDBC) API.
- JDBC is specified as an "interface".
- An interface in Java can have many "implementations".
- So it provides a convenient way to realize many "drivers"

JDBC

Drivers

- JDBC uses drivers to translate generalized JDBC calls into vendor-specific database calls
  - Drivers exist for most popular databases
  - Four Classes of JDBC drivers exist

    Type I

    Type II

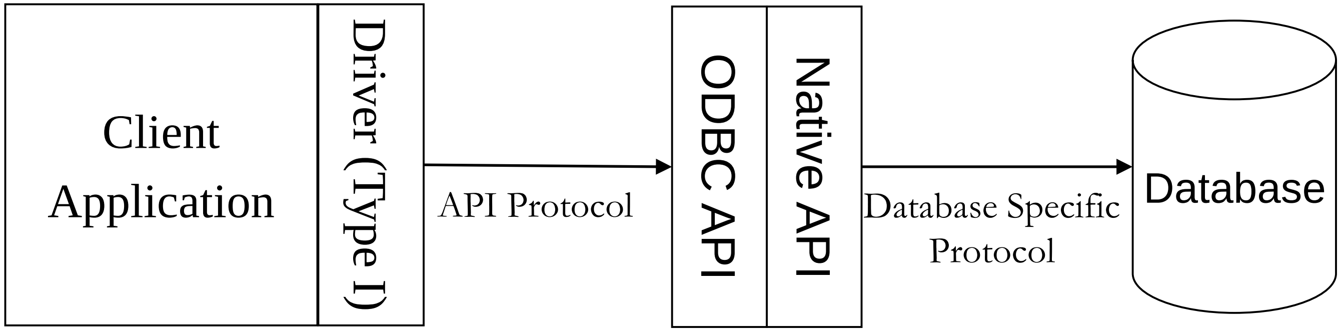
    Type III

    Type IV

JDBC

Drivers (Type I)

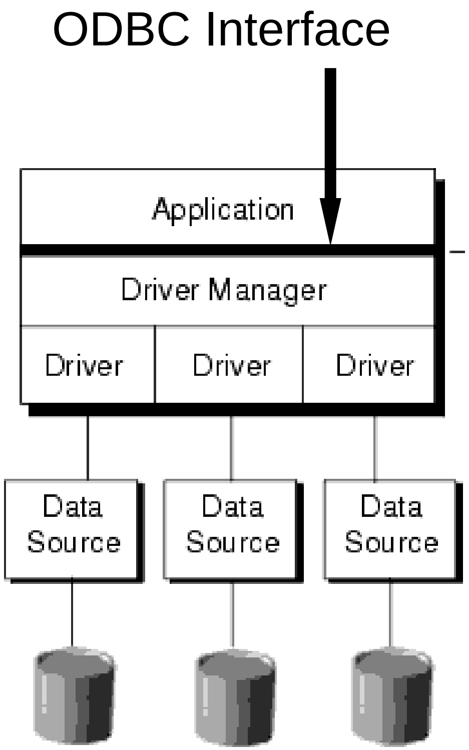
- Type I driver provides mapping between JDBC and access API of a database
  - The access API calls the native API of the database to establish communication
- A common Type I driver defines a JDBC to ODBC bridge
  - ODBC is the database connectivity for databases
  - JDBC driver translates JDBC calls to corresponding ODBC calls
  - Thus if ODBC driver exists for a database this bridge can be used to communicate with the database from a Java application
- Inefficient and narrow solution
  - Inefficient, because it goes through multiple layers
  - Narrow, since functionality of JDBC code limited to whatever ODBC supports



JDBC

Open Database Connectivity (ODBC)

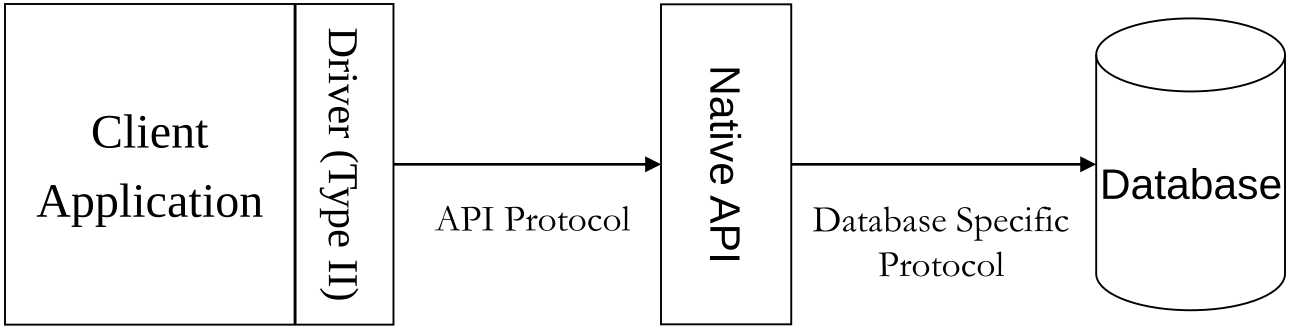
- A standard database access method developed by the SQL Access group in 1992.
  - The goal of ODBC is to make it possible to access any data from any application, regardless of which database management system (DBMS) is handling the data.
  - ODBC manages this by inserting a middle layer, called a database *driver* , between an application and the DBMS.
  - The purpose of this layer is to translate the application's data queries into commands that the DBMS understands.
  - For this to work, both the application and the DBMS must be *ODBC-compliant*, that is, the application must be capable of issuing ODBC commands and the DBMS must be capable of responding to them.



JDBC

Drivers (Type II)

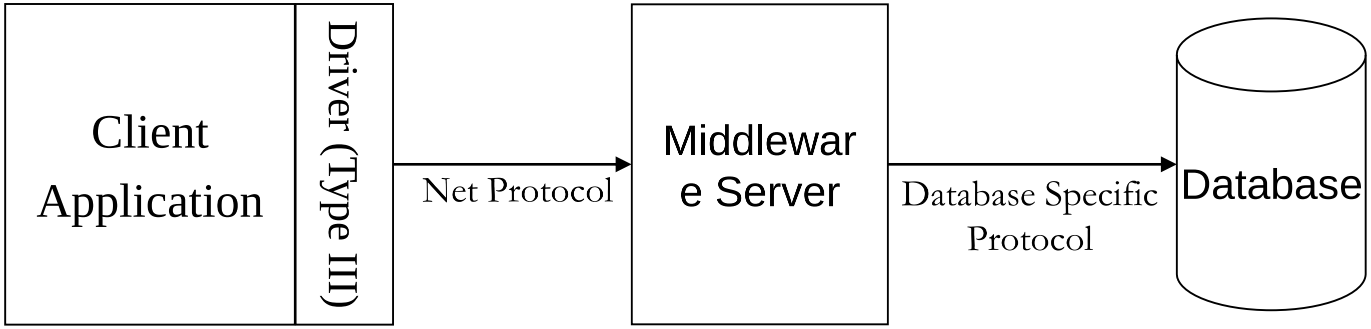
- Type II driver communicates directly with native API
  - Type II makes calls directly to the native API calls
  - More efficient since there is one less layer to contend with (i.e. no ODBC)
  - It is dependent on the existence of a native API for a database



JDBC

Drivers (Type III)

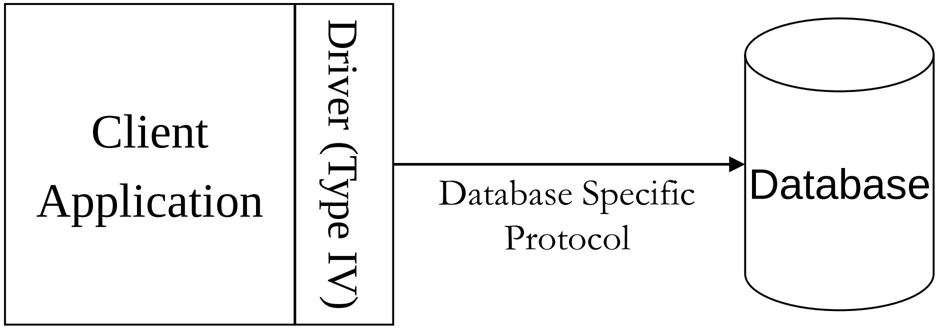
- Type III driver make calls to a middleware component running on another server
  - This communication uses a database independent net protocol
  - Middleware server then makes calls to the database using database-specific protocol
  - The program sends JDBC call through the JDBC driver to the middle tier
  - Middle-tier may use Type I or II JDBC driver to communicate with the database.



JDBC

Drivers (Type IV)

- Type IV driver is an all-Java driver that is also called a thin driver
  - It issues requests directly to the database using its native protocol
  - It can be used directly on platform with a JVM
  - Most efficient since requests only go through one layer
  - Simplest to deploy since no additional libraries or middle-ware



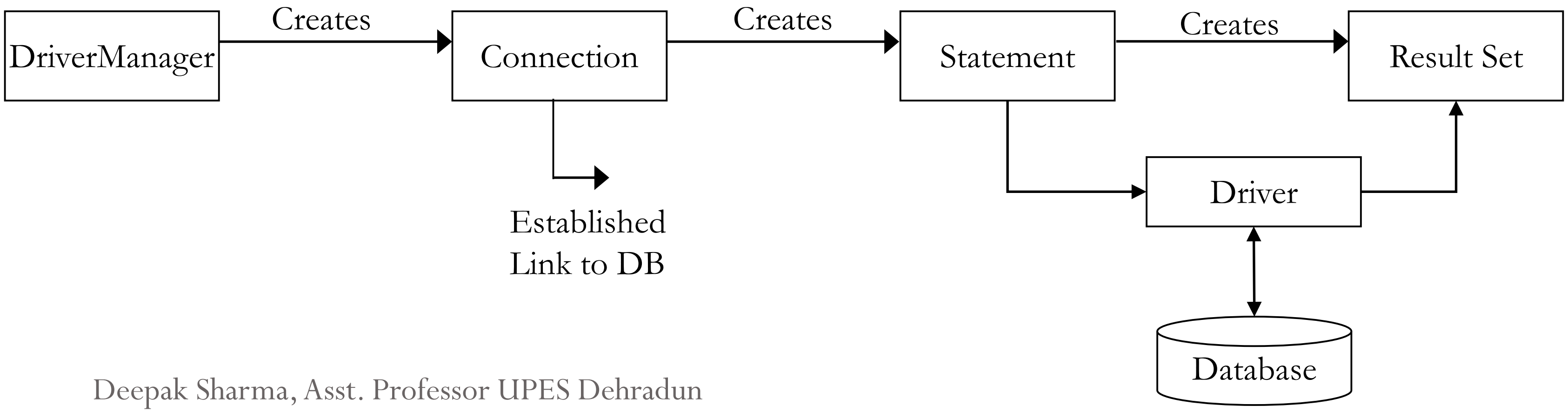
Data Source and Driver

- Data source is the data base created using any of the common database applications available.
- Your system should have the driver for the database you will be using.
- For example your Windows system should have the MS Access Driver.
- There are a number of JDBC drivers available.
  - Information on installing them is available at :
    - http://industry.java.sun.com/products/jdbc/drivers

JDBC

Conceptual Components

- **Driver Manager:** Loads database drivers and manages connections between the application and the driver
- **Driver:** Translates API calls into operations for specific database
- **Connection:** Session between application and data source
- **Statement:** SQL statement to perform query or update
- **Metadata:** Information about returned data, database, & driver
- **Result Set:** Logical set of columns and rows of data returned by executing a statement



JDBC Classes

Java supports DB facilities by providing classes and interfaces for its components

- **DriverManager**
- **Connection**
- **Statement**
- **ResultSet**

Driver Manager Class

- Provides static, “factory” methods for creating objects implementing the **connection** interface.
  - Factory methods create objects on demand
- It contains all the appropriate methods to register and deregister the database driver class and to create a connection between a Java application and the database.

Connection interface

- Connection class represents a session with a specific data source.
- Connection object establishes connection to a data source, allocates statement objects, which define and execute SQL statements.
- Connection can also get info (metadata) about the data source.

```
Example:
Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/my
db","root","root");
```

Statement interface

- Statement interface is implemented by the connection object.
- Statement object provides the workspace for SQL query, executing it, and retrieving returned data.
- SELECT {what} FROM {table name} WHERE {criteria} ORDER BY {field}
- Queries are embedded as strings in a Statement object.
- Types: Statement, PreparedStatement, CallableStatement

```
Example:
Statement stmt=con.createStatement();

String q="delete from student where name='raj'";
stmt.execute(q);
```

ResultSet interface

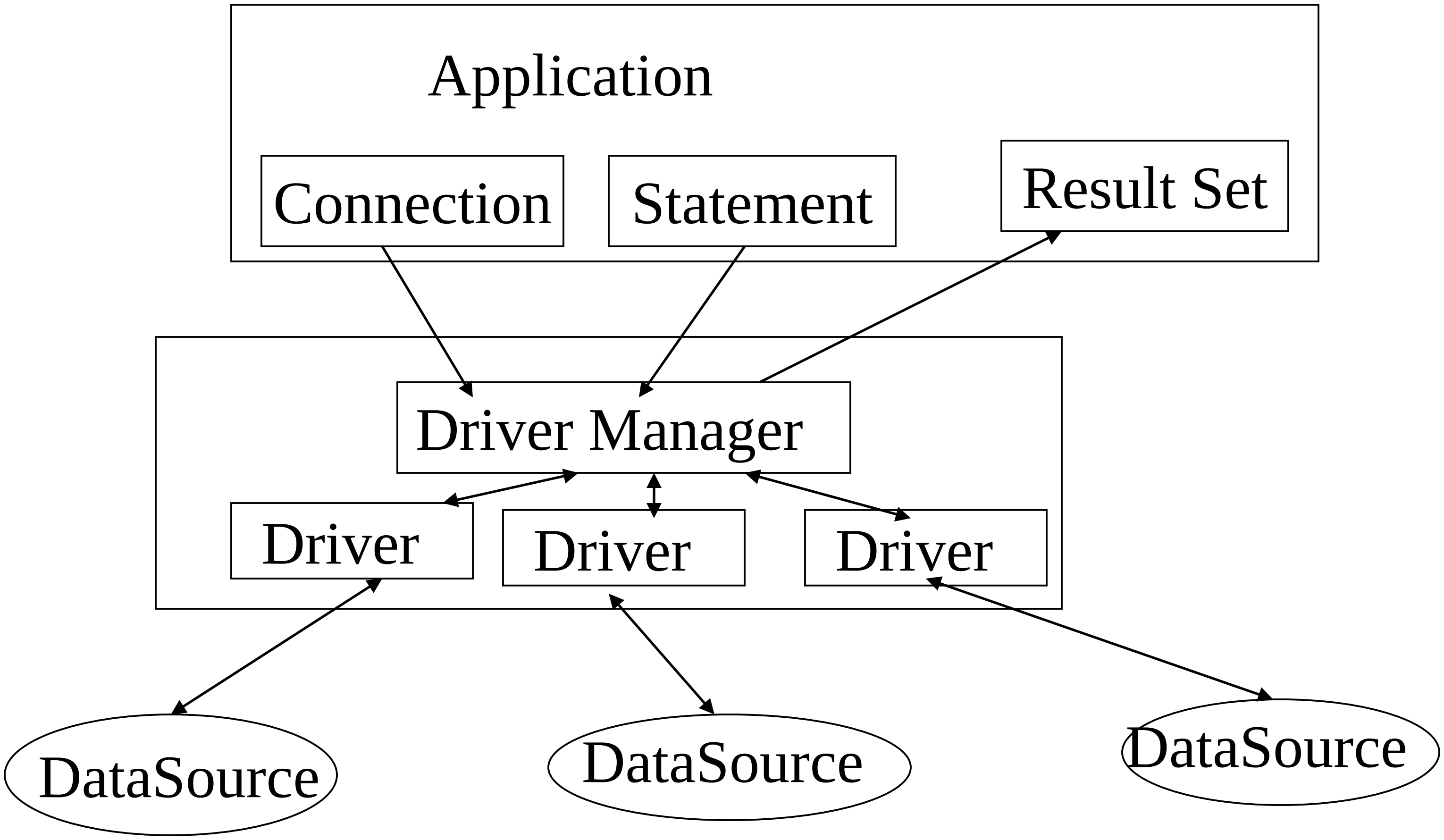
- Results are returned in the form of an object implementing the ResultSet interface.
- You may extract individual columns, rows or cell from the ResultSet using the metadata.

```
Example:

String q="Select * from student";

ResultSet rs=stmt.executeQuery(q);
```

JDBC Application Architecture



JDBC Programming Steps

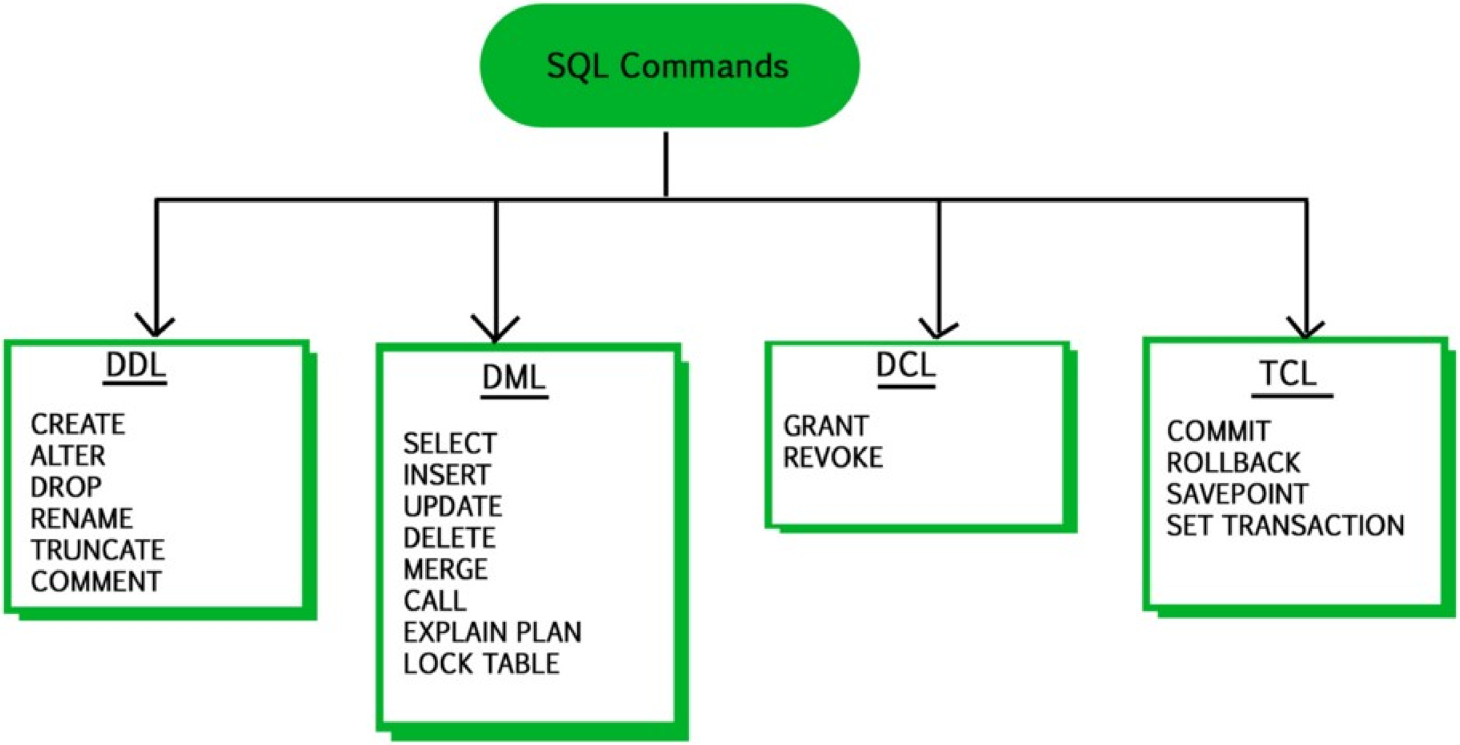
- Import necessary packages; Ex: import java.sql.*;
- Load JDBC driver(driver should have been installed)
  - For IDE: Can be added directly by adding connector (.jar) in java build path.
- Data source and its location should have been registered.
- Allocate Connection object, Statement object and ResultSet object
- Execute query using Statement object
- Retrieve data from ResultSet object
- Close Connection object.

Executing Queries

Methods

- Two primary methods in statement interface used for executing Queries
  - executeQuery Used to retrieve data from a database
  - executeUpdate: Used for creating, updating & deleting data
- executeQuery used to retrieve data from database
  - Primarily uses Select commands
- executeUpdate used for creating, updating & deleting data
  - SQL should contain Update, Insert or Delete commands
- Uset setQueryTimeout to specify a maximum delay to wait for results

SQL Commands Overview



SQL Commands

DDL

CREATE
ALTER
DROP
RENAME
TRUNCATE
COMMENT

DML

SELECT
INSERT
UPDATE
DELETE
MERGE
CALL
EXPLAIN PLAN
LOCK TABLE

DCL

GRANT
REVOKE

TCL

COMMIT
ROLLBACK
SAVEPOINT
SET TRANSACTION

Executing Queries

Data Definition Language (DDL)

- Data definition language queries use executeUpdate
- Syntax: int executeUpdate(String sqlString) throws SQLException
  - It returns an integer which is the number of rows updated
  - sqlString should be a valid String else an exception is thrown
- Example 1: Create a new table

```
Statement statement = connection.createStatement();
String sqlString =
“Create Table Catalog”
+ “(Title Varchar(256) Primary Key Not Null,”+
+ “LeadActor Varchar(256) Not Null, LeadActress Varchar(256) Not Null,”
+ “Type Varchar(20) Not Null, ReleaseDate Date Not NULL )”;
Statement.executeUpdate(sqlString);
```

  - executeUpdate returns a zero since no row is updated

Executing Queries

DDL (Example)

- Example 2: Update table

```
Statement statement = connection.createStatement();
String sqlString =
 “Insert into Catalog”
+ “(Title, LeadActor, LeadActress, Type, ReleaseDate)”
+ “Values(‘Gone With The Wind’, ‘Clark Gable’, ‘Vivien Liegh’,”
+ “’Romantic’, ‘02/18/2003’ ”
Statement.executeUpdate(sqlString);
```

  - executeUpdate returns a 1 since one row is added

Executing Queries

Data Manipulation Language (DML)

- Data definition language queries use executeQuery
- Syntax

  ResultSet executeQuery(String sqlString) throws SQLException

  - It returns a ResultSet object which contains the results of the Query
- Example 1: Query a table

```
Statement statement = connection.createStatement();
String sqlString = “Select Catalog.Title, Catalog.LeadActor, Catalog.LeadActress,” +
                   “Catalog.Type, Catalog.ReleaseDate From Catalog”;
ResultSet rs = statement.executeQuery(sqlString);
```

ResultSet

Definition

- ResultSet contains the results of the database query that are returned
- Allows the program to scroll through each row and read all columns of data
- ResultSet provides various access methods that take a column index or column name and returns the data
  - All methods may not be applicable to all resultsets depending on the method of creation of the statement.
- When the executeQuery method returns the ResultSet the cursor is placed before the first row of the data
  - Cursor refers to the set of rows returned by a query and is positioned on the row that is being accessed
  - To move the cursor to the first row of data next() method is invoked on the resultset
  - If the next row has a data the next() results true else it returns false and the cursor moves beyond the end of the data
- First column has index 1, not 0

Example: JDBC MySQL Connectivity

Steps to follow:

```
//add mysql connector (See Next slide)
```

1. create connection
2. create statement/Query
3. Execute statement/Query
4. Store the results in resultset (optional)
5. close connection

Install MySQL and create database and a table

```
//create database mydb
//use mydb
//create table Student( sapid int(10), name varchar(30), cgpa
decimal(5,2))
```

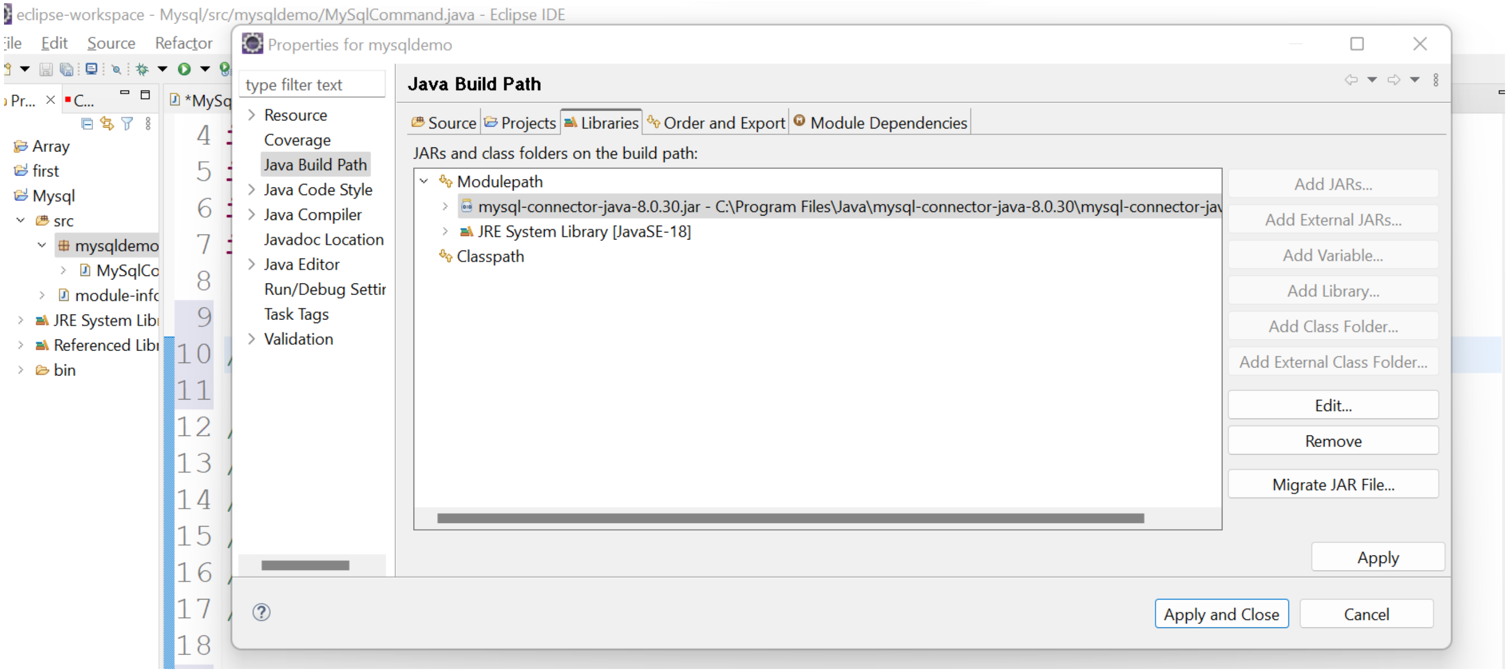
Load Java DriverManager class

Class.forName(" com.mysql.jdbc.driver ");

OR

Steps in Eclipse:

- Add MySql connector Jar file (mysql-connector-java-8.0.30)
  - Right click in your project->properties
  - Select Built Path
  - Then select Add External Jar



eclipse-workspace - Mysql/src/mysqldemo/MySqlCommand.java - Eclipse IDE

Properties for mysqldemo

Java Build Path

JARs and class folders on the build path:

Modulepath
- mysql-connector-java-8.0.30.jar - C:\Program Files\Java\mysql-connector-java-8.0.30\mysql-connector-jav
- JRE System Library [JavaSE-18]

Classpath

Example-1

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
public class MySqlCommand {
public static void main(String[] args) throws SQLException
{

//1. create connection
Connection
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb","root","root");

//2. create statement/Query
Statement stmt=con.createStatement();
String q="insert into student values(50002561,'Ranjit',7.5)";

//String q="update student set cgpa=8.9 where name=‘Ranjit'";
//String q="delete from student where name='raj'";

//3. Execute statement/Query
stmt.execute(q);
con.close();
System.out.println("Query Executed");

}
}
```

O/P:
Query Executed

Example-2

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class MySqlCommand {
public static void main(String[] args) throws SQLException
{
//1. create connection
Connection
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb","root","root");
//2. create statement/Query
Statement stmt=con.createStatement();
//String q="insert into student values(50002561,'Ranjit',7.5)";

//4. store the results in ResultSet
String q="Select * from student";
ResultSet rs=stmt.executeQuery(q);
while(rs.next())
{
/*int sapid=rs.getInt("sapid");
String name=rs.getString("name");
float cgpa= rs.getFloat("cgpa");*/
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getFloat(3));
}
con.close();
System.out.println("Query Executed");
}
}
```

```
50001364     Aman       8.0
50001366     Rahul      9.0
50002561     Ranjit     7.5
```

Summary of important SQL Commands

- SELECT - extracts data from a database
- UPDATE - updates data in a database
- DELETE - deletes data from a database
- INSERT INTO - inserts new data into a database
- CREATE DATABASE - creates a new database
- ALTER DATABASE - modifies a database
- CREATE TABLE - creates a new table
- ALTER TABLE - modifies a table

Prepared Statement

- The PreparedStatement interface is a subinterface of Statement.
- It is used to execute parameterized query.

Example: `String sql="insert into emp values(?,?,?)";`

- we are passing parameter (?) for the values.
- Its value will be set by calling the setter methods of PreparedStatement.

Why?

Improves performance: The performance of the application will be faster if you use PreparedStatement interface because query is compiled only once.

PreparedStatement

To create instance:

public Prepared Statement **prepare**Statement(String query)**throws** SQLException{}

Methods of PreparedStatement interface

Method	Description
public void setInt(int paramIndex, int value)	sets the integer value to the given parameter index.
public void setString(int paramIndex, String value)	sets the String value to the given parameter index.
public void setFloat(int paramIndex, float value)	sets the float value to the given parameter index.
public void setDouble(int paramIndex, double value)	sets the double value to the given parameter index.
public int executeUpdate()	executes the query. It is used for create, drop, insert, update, delete etc.
public ResultSet executeQuery()	executes the select query. It returns an instance of ResultSet.

PreparedStatement

How to execute the query using PreparedStatement

We need to follow 4 steps.

- Load driver & create an instance of Connection.
- Create PreparedStatement instance & SQL Query.
- Replace parameters with dynamic values.
- Execute query.

Example of PreparedStatement interface that inserts the record

Assume Table: create table emp(id number(10),name varchar2(50));

```
......
Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb","root","root");

PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");

stmt.setInt(1,101);//1 specifies the first parameter in the query
stmt.setString(2,"Ratan");

int i=stmt.executeUpdate();

System.out.println(i+" records inserted");

con.close();
}
}
```

Example- UPDATE using PreparedStatemt

```java
package PreparedStmt;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class PreparedStmt {
public static void main(String[] args) throws SQLException {
Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb","root","root");

//Create PreparedStatement instance & SQL Query
String sql = "UPDATE STUDENT SET NAME=?,sapid=? where cgpa>=?";
PreparedStatement pstmt = con.prepareStatement(sql);

//Replace paramenters with dynamic values
pstmt.setString(1, "TARUN");
pstmt.setInt(2, 50001234);
pstmt.setDouble(3, 8.0);

//Execute query
pstmt.executeUpdate();

PreparedStatement pstmt1 = con.prepareStatement("Select * from student");
ResultSet rs=pstmt1.executeQuery();
while(rs.next())
{
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getFloat(3));
}
con.close();
System.out.println("Query Executed");
}}
```

```
50001234  TARUN  8.0
50001234  TARUN  9.0
50002561  Ranjit  7.5
Query Executed
```

Example of PreparedStatement interface that deletes the record

```
………………….
…..
PreparedStatement stmt=con.prepareStatement("delete from student where sapid=?");

stmt.setInt(1, 50001234);

int i=stmt.executeUpdate();

System.out.println(i+" records deleted");
…..
…..
```

Example of PreparedStatement interface that retrieve the records of a table

```
………………. . .
PreparedStatement pstmt1 = con.prepareStatement("Select * from student");


ResultSet rs=pstmt1.executeQuery();

while(rs.next())
{
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  “ +
rs.getFloat(3));
  }
con.close();
….
```

Example of PreparedStatement to insert records until user press n

```
…………….
Connection
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb","root","root");

PreparedStatement ps=con.prepareStatement("insert into student values(?,?,?)");

BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

do{
System.out.println("enter SAP id:");
int sapid=Integer.parseInt(br.readLine());
System.out.println("enter name:");
String name=br.readLine();
System.out.println("enter cgpa:");
float cgpa=Float.parseFloat(br.readLine());

ps.setInt(1,sapid);
ps.setString(2,name);
ps.setFloat(3,cgpa);
int i=ps.executeUpdate();
System.out.println(i+" records affected");
System.out.println("Do you want to continue: y/n");
String s=br.readLine();
if(s.startsWith("n")){
break;
}
}while(true);
…………….
```

Java CallableStatement Interface

- CallableStatement interface is used to call the **stored procedures and functions.**
- will make the performance better because these are precompiled.

What is the difference between stored procedures and functions.

Stored Procedure	Function
is used to perform business logic.	is used to perform calculation.
must not have the return type.	must have the return type.
may return 0 or more values.	may return only one values.
We can call functions from the procedure.	Procedure cannot be called from function.
Procedure supports input and output parameters.	Function supports only input parameter.
Exception handling using try/catch block can be used in stored procedures.	Exception handling using try/catch can't be used in user defined functions.

How to get the instance of CallableStatement?

- **Syntax:**

public CallableStatement **prepareCall**("{ call procedurename(?,?...?)}");

- Example:

CallableStatement stmt=con.prepareCall("{call myprocedure(?,?)}");

Example

```
…
CallableStatement stmt=con.prepareCall("{call insertR(?,?)}");
stmt.setInt(1,1011);
stmt.setString(2,"Amit");
stmt.execute();

System.out.println("success");
…..
```

simple program to call the function

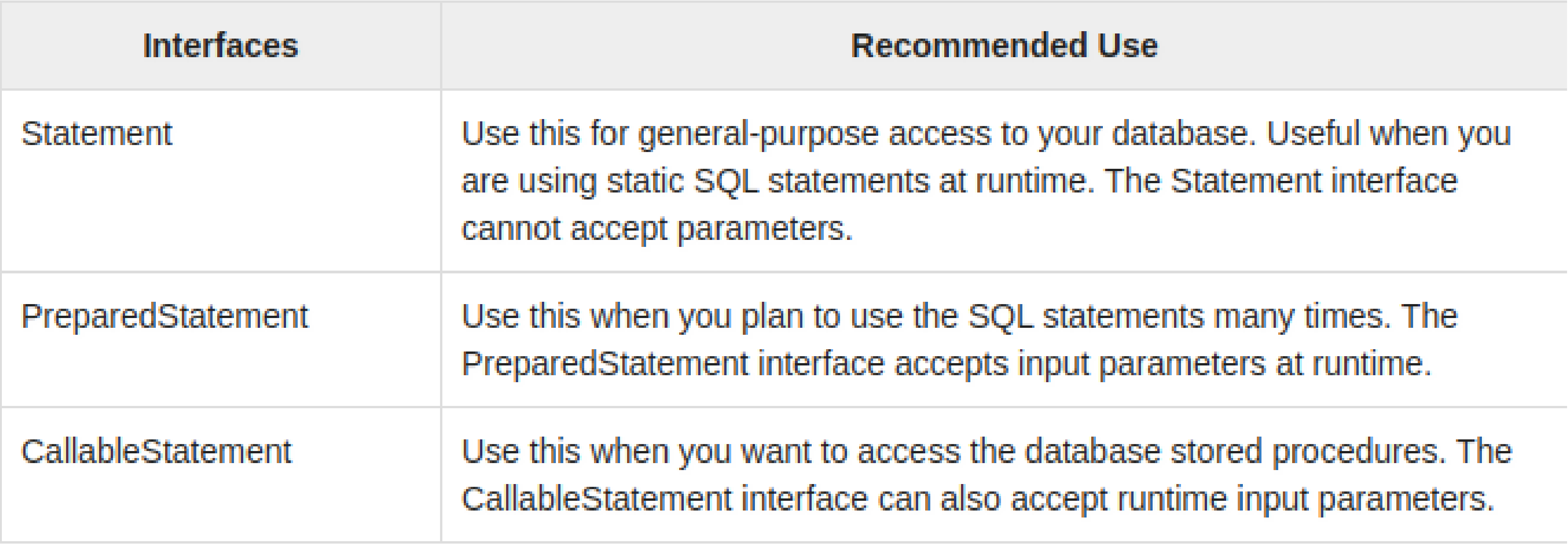
```
CallableStatement stmt=con.prepareCall("{?= call dosum(?,?)}");  //dosum is function

stmt.setInt(2,10);
stmt.setInt(3,43);

stmt.registerOutParameter(1,Types.INTEGER);

stmt.execute();
```

Statement vs Prepared statement vs Callable



Interfaces	Recommended Use
Statement	Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use this when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use this when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

ResultSet Types & Concurrency

statement.createStatement(int resultSetType, int resultSetConcurrency)

1) Forward Only (ResultSet.*TYPE_FORWARD_ONLY)*

- This type of ResultSet instance can move only in the forward direction from the first row to the last row.
- ResultSet can be moved forward one row by calling the next() method.
- We can obtain this type of ResultSet while creating Instance of Statement, PreparedStatement or CallableStatement.

```
Statement stmt = connection.createStatement(ResultSet.TYPE_FORWARD_ONLY,
ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery("select * from tbluser");
```

2) Scroll Insensitive (ResultSet.*TYPE_SCROLL_INSENSITIVE*)

- Scroll Insensitive ResultSet can scroll in both forward and backward directions.
- It can also be scrolled to an absolute position by calling the absolute() method.
- But it is not sensitive to data changes.
- It will only have data when the query was executed and ResultSet was obtained.
- It will not reflect the changes made to data after it was obtained.

```
Statement stmt =
connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                           ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery("select * from tbluser");
```

3) Scroll Sensitive (ResultSet.*TYPE_SCROLL_SENSITIVE)*

- Scroll Sensitive ResultSet can scroll in both forward and backward directions.
- It can also be scrolled to an absolute position by calling the absolute() method.
- But it is sensitive to data changes.
- It will reflect the changes made to data while it is open.

```
Statement stmt =
connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
           ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery("select * from tbluser");
```

ResultSet Concurrency

1) Read Only (ResultSet.*CONCUR_READ_ONLY)*

It is the default concurrency model. We can only perform Read-Only operations on ResultSet Instance. No update Operations are allowed.

2) Updatable (ResultSet.*CONCUR_UPDATABLE)*
In this case, we can perform update operations on ResultSet instance.

Resultset methods

1) public boolean next():	is used to move the cursor to the one row next from the current position.
2) public boolean previous():	is used to move the cursor to the one row previous from the current position.
3) public boolean first():	is used to move the cursor to the first row in result set object.
4) public boolean last():	is used to move the cursor to the last row in result set object.
5) public boolean absolute(int row):	is used to move the cursor to the specified row number in the ResultSet object.
6) public boolean relative(int row):	is used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative.
7) public int getInt(int columnIndex):	is used to return the data of specified column index of the current row as int.
8) public int getInt(String columnName):	is used to return the data of specified column name of the current row as int.
9) public String getString(int columnIndex):	is used to return the data of specified column index of the current row as String.
10) public String getString(String columnName):	is used to return the data of specified column name of the current row as String.