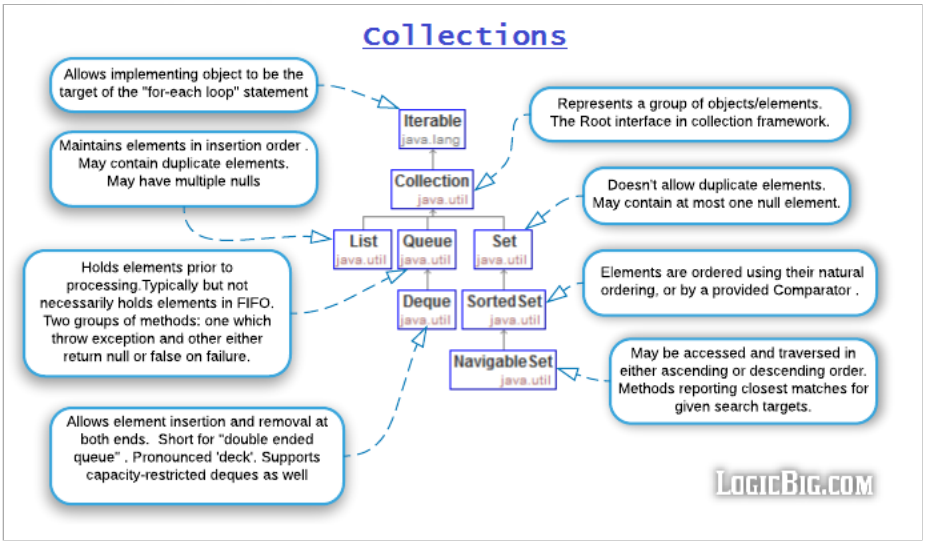
Java - Collection Interfaces and Implementations

[Last Updated: Nov 25, 2018]

Previous Page Next

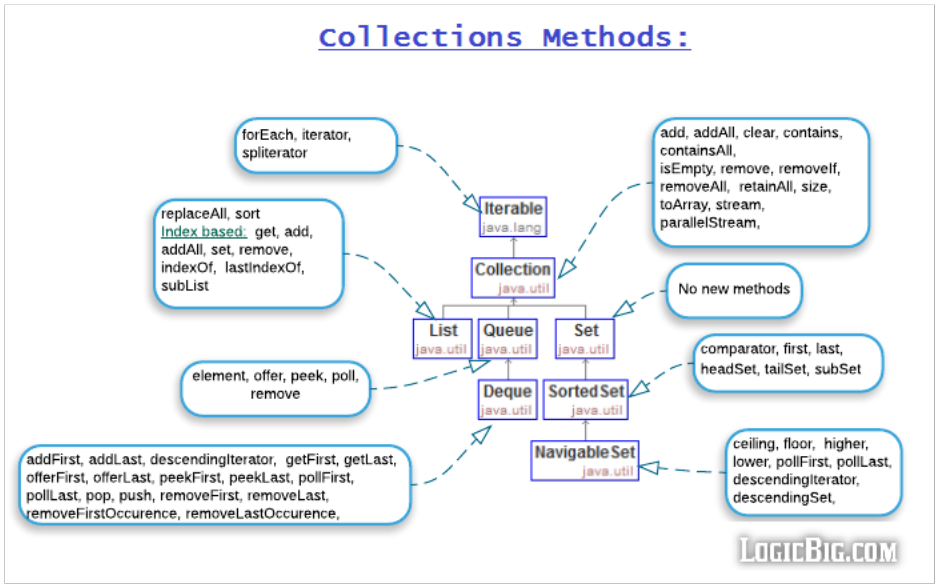
This is a quick walk-through tutorial of Java Collections interfaces and their implementations.

Collection Interfaces



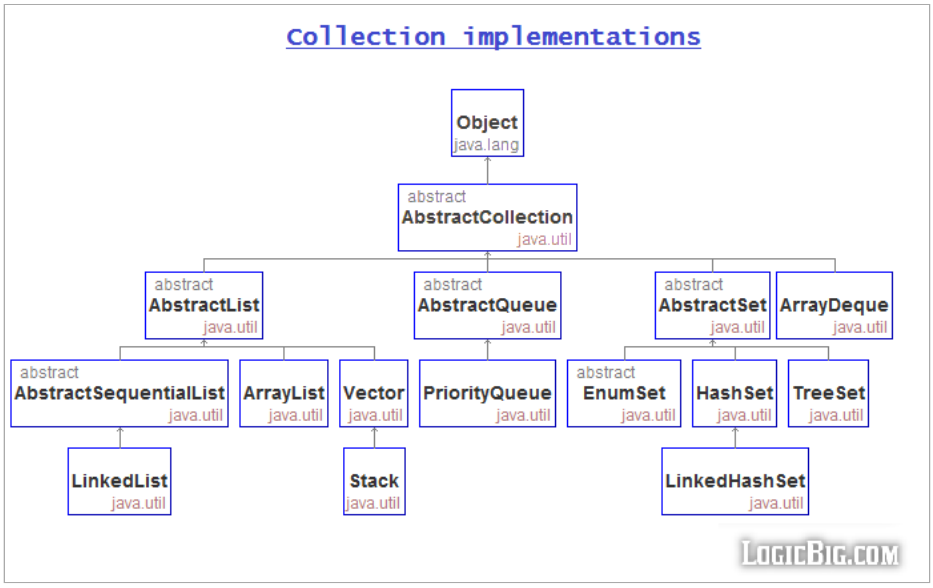
Collection Operations

It's good to know what methods each interface offer:



Note that Set doesn't have get(int index) method because no order is maintained with Set so elements don't have fixed index.

Collection Implementations

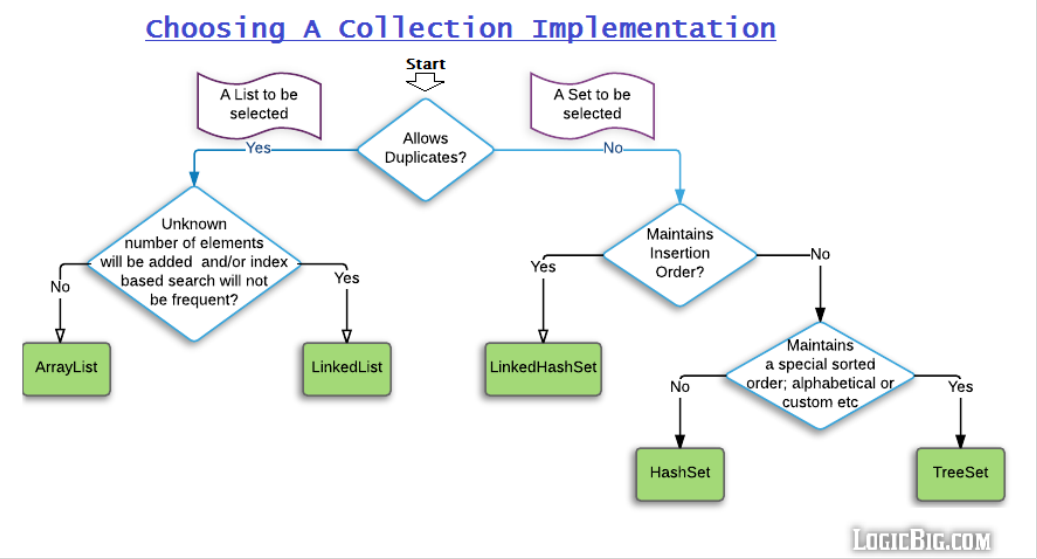


Impl	ADT	Data Structure	Performance (Big O notation)
ArrayList (sync ×)	List	Array of objects. A new array is created and populated whenever elements are added beyond the current length (capacity) of the underlying array.	**add(E element) method:** O(1) amortized . That is, adding n elements within capacity: constant time O(1). Adding an element beyond capacity: O(n) times. It's better to specify initial capacity at construction if known. **remove(int index)**: O(n - index), removing last is O(1). **All other operations including get(int index)** run in linear time O(1) . The constant factor of O(1) is low compared to that for the LinkedList implementation.
LinkedList (sync ×)	List, Deque	Doubly-linked list. Each element has memory addresses of the previous and next item used internally.	**get(int index), remove(int index)**: O(n) **add(E element) and others**: Constant time O(1).
Vector (sync ✓) (Legacy ✓)	List	Array of objects. Similar to ArrayList	Similar to ArrayList but slower because of synchronization.
Stack extends Vector (sync ✓) (Legacy ✓)	List	Array of objects. LIFO (Last in first out). It provides addition methods **empty(), peek(), pop(), push(E e) and search(Object o)**	Similar to Vector/ArrayList but slower because of synchronisation.
HashSet (sync ×)	Set	Backed by HashMap (a Hash table data structure). Elements of the set are populated as key of the HashMap. Allows at most one null.	**add, remove, contains, size**: O(1) **Iteration**: O(n + capacity). Better don't set initial capacity (size of backing hasMap) too high or load factor too low if iteration is frequently used.
LinkedHashSet (sync ×)	Set	Backed by LinkedHashMap where elements of this	**add, remove, contains, size**: O(1) **Iteration**: O(n), slightly slow that of HashSet, due to maintaining the linked list.

Core Java Tutorials

		LinkedHashSet are populated as key of the Map. Maintains elements in insertion order. Allows at most one null.	
TreeSet (sync ×)	NavigableSet	Backed by TreeMap (a red-black tree data structure). The elements of this set are populated as key of the Map. Doesn't permit null.	**add, remove, contains**: O(log n) **Iteration**: O(n) slower than HashSet.
EnumSet (sync ×)	Set	Bit vectors All of the elements must come from a single enum type.	**All methods**: O(1). Very efficient
PriorityQueue (sync ×)	Queue	Binary Heap Unbounded Elements are ordered to their natural ordering or by a provided Comparator.	**offer, poll, remove() and add**: O(log n) **remove(Object), contains(Object)** O(n) **peek, element, and size**: O(1)
ArrayDeque (sync ×)	Dequeue	Resizable-array (similar to ArrayList). Unbounded Nulls not permitted.	**remove, removeFirstOccurrence, removeLastOccurrence, contains, iterator.remove(), and the bulk operations**: O(n) **All other operations** O(1) amortized

Making a Decision in choosing a Collection



Making a decision regarding choosing a Queue implementation is quite straight forward and is not relevant in above diagram. Whenever we need to hold elements prior to processing and just need the method remove() to get and remove elements at the same time we are going to use queues. Also there's no index based operation to be performed when using queues. If we want to order queue elements in some order then use **PriorityQueue**, otherwise use **ArrayDeque**.

EnumSet is also very straight forward i.e. whenever we want to hold certain elements of an enum we will use it. We can also use other collections for the same purpose but remember EnumSet is very very efficient comparatively.

Java 10 Features

Java 9 Module System

Java 9 Misc Features

Java 9 JShell

Recent Tutorials

Spring - DefaultFormattingConversionSe with DataBinder

Spring - DefaultFormattingConversionSe with ApplicationContext

Spring - Built-in Formatter List

Spring - Custom Formatter

Spring - Built-in converter List

Spring - DataBinder Using ConversionSe

Spring - ConditionalGenericConverter

Spring - Generic Converter

Spring - Creating Custom ConverterFact

Spring - Converter Factory

Spring - Creating a Custom Converter

Spring - Using ConversionService With ApplicationContext

Python - Tuples

Python - Lists

Python - Naming Conventions

Python - Built In Data Structures

Python None Type

Python Boolean

Python Keywords

Python String

Python Numbers

Python Variables

Python Comments

Python Basic Syntax

Python with Intellij

Python language getting started

Installing Python 3.10.x on windows

Spring - Mixed validation using JSR 349 Annotations and Spring validator

Spring Annotation Based Validation

Spring Core Validation

Java Bean Validation - Creating multiple validators for the same constraint

Spring - The Use Of PropertyEditors Wit Annotation

Spring - The Use Of PropertyEditors Wit Configuration

Spring - Use of PropertyEditors via Bean

Spring - Using BeanUtils for resolving st based method signatures

Spring - Copying properties using BeanU

Spring - Obtaining BeanInfo And PropertyDescriptors

Synchronization

It's not recommended to use legacy collections. One advantage of using them might seem to be that they are synchronized. java.util.Collections provides methods to wrap any collection around a new collection which is synchronized. These methods are:

```
Collection<T> synchronizedCollection(Collection<T> c)
```

```
Set<T> synchronizedSet(Set<T> s)
```

```
SortedSet<T> synchronizedSortedSet(SortedSet<T> s)
```

```
NavigableSet<T> synchronizedNavigableSet(NavigableSet<T> s)
```

```
List<T> synchronizedList(List<T> list)
```

Read Only Collections

java.util.Collections provides methods to wrap any collection and return a new Collection which are read only. Attempts to modify (calling mutative methods), directly or via iterators will cause `UnsupportedOperationException`. Followings are the methods:

```
Collection<T> unmodifiableCollection(Collection<T> c)
```

```
Set<T> unmodifiableSet(Set<T> s)
```

```
SortedSet<T> unmodifiableSortedSet(SortedSet<T> s)
```

```
NavigableSet<T> unmodifiableNavigableSet(NavigableSet<T> s)
```

```
List<T> unmodifiableList(List<T> list)
```

In next topics we will explore concurrent collections and then maps.

←

Spring - Directly setting fields via DirectFieldAccess
Spring - BeanWrapper, setting nested be
Spring - Using BeanWrapper
Spring - Transforming events
Spring - Conditional Event Handling via @EventListener annotation
Spring - Using @EventListener annotatic multiple event types
Spring - Publishing and Consuming Cust Events
Java - Introduction to ResourceBundle
Spring - Injecting Resource using @Valu annotation
Spring - Injecting ResourceLoader
Spring - Resource Loading
Spring - Using Spring Expression Langua @Value Annotation
Spring - Using @Value Annotation
Spring - Injecting Environment to access properties in beans
Spring - Adding New Property Source to Environment
Spring - Adding user properties by using @PropertySource
Spring - Accessing Environment Properti
Spring - Profiles

See Also

Big O notation
Abstract Data Type (ADT)
Data Structure
Algorithm
What is Collections Framework?
Concurrent Collections
Map Interfaces and Implementations
Concurrent Maps
Java HashMap - Understanding equals() and hashCode() methods
Iterator vs Spliterator
Dynamically typesafe view of collections and maps
Understanding WeakHashMap
Java Collections quick examples

Core Java Tutorials

Java 16 Features
Java 15 Features
Java 14 Features
Java 13 Features
Java 12 Features
Java 11 Features
Java 10 Features
Java 9 Module System
Java 9 Misc Features
Java 9 JShell

Recent Tutorials

Spring - DefaultFormattingConversionService with DataBinder
Spring - DefaultFormattingConversionService with ApplicationContext
Spring - Built-in Formatter List
Spring - Custom Formatter
Spring - Built-in converter List
Spring - DataBinder Using ConversionService
Spring - ConditionalGenericConverter
Spring - Generic Converter
Spring - Creating Custom ConverterFactory
Spring - Converter Factory
Spring - Creating a Custom Converter
Spring - Using ConversionService With ApplicationContext

Python - Tuples
Python - Built In Data Structures
Python - Lists
Python None Type
Python - Naming Conventions

Share

Previous Page Next

THE JAVA LANGUAGE CHEAT SHEET

Primitive Types:

INTEGER: byte(8bit),short(16bit),int(32bit),
long(64bit),**DECIM:**float(32bit),double(64bit)
,**OTHER:** boolean(1bit), char (Unicode)
HEX:0x1AF,**BINARY:**0b00101,**LONG:**8888888888888**L**
CHAR EXAMPLES: 'a','\n','\t','\'','\\','\"'

Primitive Operators

Assignment Operator: = (ex: int a=5,b=3;)
Binary Operators (two arguments): + - * / %
Unary Operators: + - ++ --
Boolean Not Operator (Unary): !
Boolean Binary: == != > >= < <=
Boolean Binary Only: && ||
Bitwise Operators: ~ & ^ | << >> >>>
Ternary Operator: bool?valtrue:valfalse;

Casting, Conversion

```
int x = (int)5.5; //works for numeric types
int x = Integer.parseInt("123");
float y = Float.parseFloat("1.5");
int x = Integer.parseInt("7A",16); //fromHex
String hex = Integer.toString(99,16);//toHex
//Previous lines work w/ binary, other bases
```

java.util.Scanner, input, output

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt(); //stops at whitespace
String line = sc.nextLine(); //whole line
System.out.println("bla"); //stdout
System.err.print("bla"); //stderr,no newline
```

java.lang.Number types

```
Integer x = 5; double y = x.doubleValue();
double y = (double)x.intValue();
//Many other methods for Long, Double, etc
```

java.lang.String Methods

```
//Operator +, e.g. "fat"+"cat" -> "fatcat"
boolean equals(String other);
int length();
char charAt(int i);
String substring(int i, int j); //j not incl
boolean contains(String sub);
boolean startsWith(String pre);
boolean endsWith(String post);
int indexOf(String p); //-1 if not found
int indexOf(String p, int i); //start at i
int compareTo(String t);
//"a".compareTo("b") -> -1
String replaceAll(String str, String find);
String[] split(String delim);
```

StringBuffer, StringBuilder

StringBuffer is synchronized StringBuilder
(Use StringBuilder unless multithreaded)
Use the .apend(xyz) methods to concat
toString() converts back to String

java.lang.Math

```
Math.abs(NUM),Math.ceil(NUM),Math.floor(NUM)
,Math.log(NUM),Math.max(A,B),Math.min(C,D),
Math.pow(A,B),Math.round(A),Math.random()
```

IF STATEMENTS:

```
if( boolean_value ) { STATEMENTS }
else if( bool )     { STATEMENTS }
else if( ..etc )    { STATEMENTS }
else                { STATEMENTS }
//curly brackets optional if one line
```

LOOPS:

```
while( bool )         { STATEMENTS }
for(INIT;BOOL;UPDATE) { STATEMENTS }
//1INIT 2BOOL 3STATEMENTS 4UPDATE 5->Step2
do{ STATEMENTS }while( bool );
//do loops run at least once before checking
break;    //ends enclosing loop (exit loop)
continue; //jumps to bottom of loop
```

ARRAYS:

```
int[] x = new int[10]; //ten zeros
int[][] x = new int[5][5]; //5 by 5 matrix
int[] x = {1,2,3,4};
x.length; //int expression length of array
int[][] x = {{1,2},{3,4,5}}; //ragged array
String[] y = new String[10]; //10 nulls
//Note that object types are null by default

//loop through array:
for(int i=0;i<arrayname.length;i++) {
  //use arrayname[i];
}

//for-each loop through array
int[] x = {10,20,30,40};
for(int v : x) {
  //v cycles between 10,20,30,40
}

//Loop through ragged arrays:
for(int i=0;i<x.length;i++)
  for(int j=0;j<x[i].length;j++) {
     //CODE HERE
  }

//Note, multi-dim arrays can have nulls
//in many places, especially object arrays:
Integer[][] x = {{1,2},{3,null},null};
```

FUNCTIONS / METHODS:

```
Static Declarations:
public static int functionname( … )
private static double functionname( … )
static void functionname( … )
Instance Declarations:
public void functionname( … )
private int functionname( … )
Arguments, Return Statement:
int myfunc(int arg0, String arg1) {
  return 5; //type matches int myfunc
}
//Non-void methods must return before ending
//Recursive functions should have an if
//statement base-case that returns at once
```

CLASS/OBJECT TYPES:

```
INSTANTIATION:
public class Ball {//only 1 public per file
  //STATIC FIELDS/METHODS
  private static int numBalls = 0;
  public static int getNumBalls() {
    return numBalls;
  }
  public static final int BALLRADIUS = 5;

  //INSTANCE FIELDS
  private int x, y, vx, vy;
  public boolean randomPos = false;

  //CONSTRUCTORS
  public Ball(int x, int y, int vx, int vy)
  {
    this.x = x;
    this.y = y;
    this.vx = vx;
    this.vy = vy;
    numBalls++;
  }
  Ball() {
    x = Math.random()*100;
    y = Math.random()*200;
    randomPos = true;
  }

  //INSTANCE METHODS
  public int getX(){ return x; }
  public int getY(){ return y; }
  public int getVX(){ return vx; }
  public int getVY(){ return vy; }
  public void move(){ x+=vx; y+=vy; }
  public boolean touching(Ball other) {
    float dx = x-other.x;
    float dy = y-other.y;
    float rr = BALLRADIUS;
    return Math.sqrt(dx*dx+dy*dy)<rr;
  }

}

//Example Usage:
public static void main(String[] args) {
  Ball x = new Ball(5,10,2,2);
  Ball y = new Ball();
  List<Ball> balls = new ArrayList<Ball>();
  balls.add(x); balls.add(y);
  for(Ball b : balls) {
    for(Ball o : balls) {
      if(b != o) { //compares references
        boolean touch = b.touching(o);
      }
    }
  }
}
```

POLYMORPHISM:

Single Inheritance with "extends"

```
class A{ }
class B extends A{ }
abstract class C { }
class D extends C { }
class E extends D
```

Abstract methods

```
abstract class F {
  abstract int bla();
}
class G extends F {
  int bla() {  //required method
   return 5;
  }
}
```

Multiple Inheritance of interfaces with "implements" (fields not inherited)

```
interface H {
  void methodA();
  boolean methodB(int arg);
}
interface I extends H{
  void methodC();
}
interface K {}
class J extends F implements I, K {
  int bla() { return 5; } //required from F
  void methodA(){} //required from H
  boolean methodB(int a) { //req from A
    return 1;
  }
  void methodC(){} //required from I
}
```

Type inference:

```
A x = new B(); //OK
B y = new A(); //Not OK
C z = new C(); //Cannot instantiate abstract
//Method calls care about right hand type
(the instantiated object)
//Compiler checks depend on left hand type
```

GENERICS:

```
class MyClass<T> {
  T value;
  T getValue() { return value; }
}
class ExampleTwo<A,B> {
  A x;
  B y;
}
class ExampleThree<A extends List<B>,B> {
  A list;
  B head;
}
//Note the extends keyword here applies as
well to interfaces, so A can be an interface
that extends List<B>
```

JAVA COLLECTIONS:

List<T>: Similar to arrays

```
ArrayList<T>: Slow insert into middle
//ArrayList has fast random access
LinkedList<T>: slow random access
//LinkedList fast as queue/stack
Stack: Removes and adds from end
```

List Usage:

```
boolean add(T e);
void clear(); //empties
boolean contains(Object o);
T get(int index);
T remove(int index);
boolean remove(Object o);
//remove uses comparator
T set(int index, E val);
Int size();
```

List Traversal:

```
for(int i=0i<x.size();i++) {
       //use x.get(i);
}

//Assuming List<T>:
for(T e : x) {
       //use e
}
```

Queue<T>: Remove end, Insert beginning

```
LinkedList implements Queue
```

Queue Usage:

```
T element(); // does not remove
boolean offer(T o); //adds
T peek(); //pike element
T poll(); //removes
T remove(); //like poll
Traversal: for(T e : x) {}
```

Set<T>: uses Comparable<T> for uniqueness

```
TreeSet<T>, items are sorted
HashSet<T>, not sorted, no order
LinkedHashSet<T>, ordered by insert
Usage like list: add, remove, size
Traversal: for(T e : x) {}
```

Map<K,V>: Pairs where keys are unique

```
HashMap<K,V>, no order
LinkedHashMap<K,V> ordered by insert
TreeMap<K,V> sorted by keys

V get(K key);
Set<K> keySet(); //set of keys
V put(K key, V value);
V remove(K key);
Int size();
Collection<V> values(); //all values

Traversal: for-each w/ keyset/values
```

java.util.PriorityQueue<T>

A queue that is always automatically sorted using the comparable function of an object

```
public static void main(String[] args) {
  Comparator<String> cmp= new LenCmp();
  PriorityQueue<String> queue =
     new PriorityQueue<String>(10, cmp);
  queue.add("short");
  queue.add("very long indeed");
  queue.add("medium");
  while (queue.size() != 0)
    System.out.println(queue.remove());
}
class LenCmp implements Comparator<String> {
  public int compare(String x, String y){
    return x.length() - y.length();
  }
}
```

java.util.Collections algorithms

Sort Example:

```
//Assuming List<T> x
Collections.sort(x); //sorts with comparator
```

Sort Using Comparator:

```
Collections.sort(x, new Comparator<T>{
  public int compareTo(T a, T b) {
    //calculate which is first
    //return -1, 0, or 1 for order:
    return someint;
  }
}
```

Example of two dimensional array sort:

```
public static void main(final String[] a){
   final String[][] data = new String[][] {
     new String[] { "20090725", "A" },
     new String[] { "20090726", "B" },
     new String[] { "20090727", "C" },
     new String[] { "20090728", "D" } };
     Arrays.sort(data,
       new Comparator<String[]>() {
        public int compare(final String[]
entry1, final String[] entry2) {
          final String time1 = entry1[0];
          final String time2 = entry2[0];
          return time1.compareTo(time2);
        }
     });

     for (final String[] s : data) {
       System.out.println(s[0]+" "+s[1]);
     }
   }
}
```

More collections static methods:

```
Collections.max( … ); //returns maximum
Collections.min( … ); //returns maximum
Collections.copy( A, B); //A list into B
Collections.reverse( A ); //if A is list
```

Java Collections Cheat Sheet

Basics

What is Java Collection Framework?

Java Collection Framework is a framework which provides some predefined classes and interfaces to store and manipulate the group of objects. Using Java collection framework, you can store the objects as a List or as a Set or as a Queue or as a Map and perform basic operations like adding, removing, updating, sorting, searching etc... with ease.

Why Java Collection Framework?

Earlier, arrays are used to store the group of objects. But, arrays are of fixed size. You can't change the size of an array once it is defined. It causes lots of difficulties while handling the group of objects. To overcome this drawback of arrays, Java Collection Framework is introduced from JDK 1.2.

Java Collections Hierarchy :

All the classes and interfaces related to Java collections are kept in java.util package. List, Set, Queue and Map are four top level interfaces of Java collection framework. All these interfaces (except Map) inherit from java.util.Collection interface which is the root interface in the Java collection framework.

List

Intro :

- List is a sequential collection of objects.
- Elements are positioned using zero-based index.
- Elements can be inserted or removed or retrieved from any arbitrary position using an integer index.

Popular Implementations :

- ArrayList, Vector And LinkedList

Internal Structure :

- **ArrayList :** Internally uses re-sizable array which grows or shrinks as we add or delete elements.
- **Vector :** Same as ArrayList but it is synchronized.
- **LinkedList :** Elements are stored as Nodes where each node consists of three parts – Reference To Previous Element, Value Of The Element and Reference To Next Element.

Null Elements :

- **ArrayList :** Yes
- **Vector :** Yes
- **LinkedList :** Yes

Duplicate Elements :

- **ArrayList :** Yes
- **Vector :** Yes
- **LinkedList :** Yes

Order Of Elements :

- **ArrayList :** Insertion Order
- **Vector :** Insertion Order
- **LinkedList :** Insertion Order

Synchronization :

- **ArrayList :** Not synchronized
- **Vector :** Synchronized
- **LinkedList :** Not synchronized

Performance :

- **ArrayList :** Insertion -> O(1) (if insertion causes restructuring of internal array, it will be O(n)), Removal -> O(1) (if removal causes restructuring of internal array, it will be O(n)), Retrieval -> O(1)
- **Vector :** Similar to ArrayList but little slower because of synchronization.
- **LinkedList :** Insertion -> O(1), Removal -> O(1), Retrieval -> O(n)

When to use?

- **ArrayList :** Use it when more search operations are needed then insertion and removal.
- **Vector :** Use it when you need synchronized list.
- **LinkedList :** Use it when insertion and removal are needed frequently.

Queue

Intro :

- Queue is a data structure where elements are added from one end called tail of the queue and elements are removed from another end called head of the queue.
- Queue is typically FIFO (First-In-First-Out) type of data structure.

Popular Implementations :

- PriorityQueue, ArrayDeque and LinkedList (implements List also)

Internal Structure :

- **PriorityQueue :** It internally uses re-sizable array to store the elements and a Comparator to place the elements in some specific order.
- **ArrayDeque :** It internally uses re-sizable array to store the elements.

Null Elements :

- **PriorityQueue :** Not allowed
- **ArrayDeque :** Not allowed

Duplicate Elements :

- **PriorityQueue :** Yes
- **ArrayDeque :** Yes

Order Of Elements :

- **PriorityQueue :** Elements are placed according to supplied Comparator or in natural order if no Comparator is supplied.
- **ArrayDeque :** Supports both LIFO and FIFO

Synchronization :

- **PriorityQueue :** Not synchronized
- **ArrayDeque :** Not synchronized

Performance :

- **PriorityQueue :** Insertion -> O(log(n)), Removal -> O(log(n)), Retrieval -> O(1)
- **ArrayDeque :** Insertion -> O(1) , Removal -> O(n), Retrieval -> O(1)

When to use?

- **PriorityQueue :** Use it when you want a queue of elements placed in some specific order.
- **ArrayDeque :** You can use it as a queue OR as a stack.

Set

Intro :

- Set is a linear collection of objects with no duplicates.
- Set interface does not have its own methods. All its methods are inherited from Collection interface. It just applies restriction on methods so that duplicate elements are always avoided.

Popular Implementations :

- HashSet, LinkedHashSet and TreeSet

Internal Structure :

- **HashSet :** Internally uses HashMap to store the elements.
- **LinkedHashSet :** Internally uses LinkedHashMap to store the elements.
- **TreeSet :** Internally uses TreeMap to store the elements.

Null Elements :

- **HashSet :** Maximum one null element
- **LinkedHashSet :** Maximum one null element.
- **TreeSet :** Doesn't allow even a single null element

Duplicate Elements :

- **HashSet :** Not allowed
- **LinkedHashSet :** Not allowed
- **TreeSet :** Not allowed

Order Of Elements :

- **HashSet :** No order
- **LinkedHashSet :** Insertion order
- **TreeSet :** Elements are placed according to supplied Comparator or in natural order if no Comparator is supplied.

Synchronization :

- **HashSet :** Not synchronized
- **LinkedHashSet :** Not synchronized
- **TreeSet :** Not synchronized

Performance :

- **HashSet :** Insertion -> O(1), Removal -> O(1), Retrieval -> O(1)
- **LinkedHashSet :** Insertion -> O(1), Removal -> O(1), Retrieval -> O(1)
- **TreeSet :** Insertion -> O(log(n)), Removal -> O(log(n)), Retrieval -> O(log(n))

When to use?

- **HashSet :** Use it when you want only unique elements without any order.
- **LinkedHashSet :** Use it when you want only unique elements in insertion order.
- **TreeSet :** Use it when you want only unique elements in some specific order.

Map

Intro :

- Map stores the data in the form of key-value pairs where each key is associated with a value.
- Map interface is part of Java collection framework but it doesn't inherit Collection interface.

Popular Implementations :

- HashMap, LinkedHashMap And TreeMap

Internal Structure :

- **HashMap :** It internally uses an array of buckets where each bucket internally uses linked list to hold the elements.
- **LinkedHashMap :** Same as HashMap but it additionally uses a doubly linked list to maintain insertion order of elements.
- **TreeMap :** It internally uses Red-Black tree.

Null Elements :

- HashMap : Only one null key and can have multiple null values
- LinkedHashMap : Only one null key and can have multiple null values.
- TreeMap : Doesn't allow even a single null key but can have multiple null values.

Duplicate Elements :

- **HashMap :** Doesn't allow duplicate keys but can have duplicate values.
- **LinkedHashMap :** Doesn't allow duplicate keys but can have duplicate values.
- **TreeMap :** Doesn't allow duplicate keys but can have duplicate values.

Order Of Elements :

- **HashMap :** No Order
- **LinkedHashMap :** Insertion Order
- **TreeMap :** Elements are placed according to supplied Comparator or in natural order of keys if no Comparator is supplied.

Synchronization :

- **HashMap :** Not synchronized
- **LinkedHashMap :** Not Synchronized
- **TreeMap :** Not Synchronized

Performance :

- **HashMap :** Insertion -> O(1), Removal -> O(1), Retrieval -> O(1)
- **LinkedHashMap :** Insertion -> O(1), Removal -> O(1), Retrieval -> O(1)
- **TreeMap :** Insertion -> O(log(n)), Removal -> O(log(n)), Retrieval -> O(log(n))

When to use?

- **HashMap :** Use it if you want only key-value pairs without any order.
- **LinkedHashMap :** Use it if you want key-value pairs in insertion order.
- **TreeMap :** Use it when you want key-value pairs sorted in some specific order.

Java Exception Handling Cheat Sheet

(javaconceptoftheday.com)

Basics

What is exception?

Exception is an abnormal condition which occurs during execution of a program and disrupts the normal flow of a program.

Ex : NumberFormatException, ArithmaticException, ArrayIndexOutOfBoundsException, ClassCastException, NullPointerException, StackOverflowError, OutOfMemoryError etc...

Exception Handling In Java :

Exceptions in Java are handled using try, catch and finally blocks.

```
try
{
   This block contains statements which may
   throw exceptions during run time.
}
catch(Exception e)
{
   This block handles the exceptions thrown by the
   try block.
}
finally
{
   This block is always executed whether an
   exception is thrown or not and thrown exception
   is caught or not.
}
```

Rules To Follow While Writing try-catch-finally Blocks :

- ✓ try, catch and finally blocks form one unit. There must be one try block and one or more catch blocks. finally block is optional.
- ✓ There should not be any statements in between the blocks.
- ✓ If there are multiple catch blocks, the order of catch blocks must be from most specific to general ones. i.e. lower classes in the hierarchy of exceptions must come first and higher classes later.

If try-catch-finally blocks are supposed to return a value :

- ✓ If finally block returns a value then try and catch blocks may or may not return a value.
- ✓ If finally block does not return a value then both try and catch blocks must return a value.
- ✓ finally block overrides return values from try and catch blocks.
- ✓ finally block will be always executed even though try and catch blocks are returning the control.

Frequently Occurring Exceptions

1) NullPointerException occurs when your application tries to access null object.

2) ArrayIndexOutOfBoundsException occurs when you try to access an array element with an invalid index i.e index greater than the array length or with a negative index.

3) NumberFormatException is thrown when you are trying to convert a string to numeric value like integer, float, double etc..., but input string is not a valid number.

4) ClassNotFoundException is thrown when an application tries to load a class at run time but the class with specified name is not found in the classpath.

5) ArithmeticException is thrown when an abnormal arithmetic condition arises in an application.

6) SQLException is thrown when an application encounters with an error while interacting with the database.

7) ClassCastException occurs when an object of one type can not be casted to another type.

8) IOException occurs when an IO operation fails in your application.

9) NoClassDefFoundError is thrown when Java Runtime System tries to load the definition of a class which is no longer available.

10) StackOverflowError is a run time error which occurs when stack overflows. This happens when you keep calling the methods recursively.

Types Of Exceptions

There are two types of exceptions in Java.

1. **Checked Exceptions** are the exceptions which are checked during compilation itself.
2. **Unchecked Exceptions** are the exceptions which are not checked during compilation. They occur only at run time.

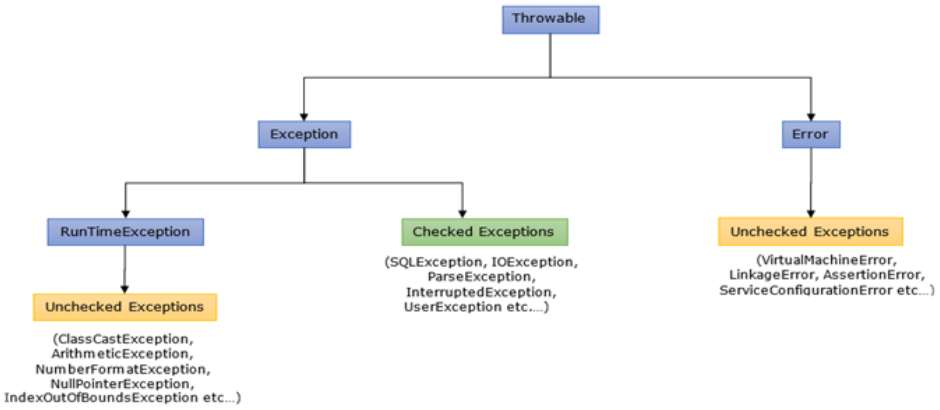
Checked Exceptions	Unchecked Exceptions
They are checked at compile time.	They are not checked at compile time.
They are compile time exceptions.	They are run time exceptions.
These exceptions must be handled properly either using try-catch blocks or using throws clause, otherwise compiler will throw error.	If these exceptions are not handled properly, compiler will not throw any error. But, you may get error at run time.
All the sub classes of java.lang.Exception (except sub classes of java.lang.RunTimeException) are checked exceptions.	All the sub classes of java.lang.RunTimeException and all the sub classes of java.lang.Error are unchecked exceptions.
Ex : FileNotFoundException, IOException, SQLException, ClassNotFoundException	Ex : NullPointerException, ArithmeticException, ClassCastException, ArrayIndexOutOfBoundsException

Hierarchy Of Exceptions

java.lang.Throwable is the super class for all type of errors and exceptions in Java.

It has two sub classes.

1. **java.lang.Error :** It is the super class for all types of errors in Java.
2. **java.lang.Exception :** It is the super class for all types of exceptions in Java.



throw Keyword

throw keyword is used to throw an exception explicitly.

```
try
{
      throw InstanceOfThrowableType;
}
catch(InstanceOfThrowableType)
{

}
```

where, InstanceOfThrowableType must be an object of type Throwable or subclass of Throwable.

throws Keyword

throws keyword is used to specify the exceptions that may be thrown by the method.

```
return_type method_name(parameter_list) throws
exception_list
{
   //some statements
}
```

where, exception_list is the list of exceptions that method may throw. Exceptions must be separated by commas.

Try-with Resources

Try with resources blocks are introduced from Java 7. In these blocks, resources used in try blocks are auto-closed. No need to close the resources explicitly. But, Java 7 try with resources has one drawback. It requires resources to be declared locally within try block. It doesn't recognize resources declared outside the try block. That issue has been resolved in Java 9.

Before Java 7

```
//Declare resources here

try
{
      //Use resources here
}
catch (Exception e)
{
      //Catch exceptions here if any
}
finally
{
      //Close resources here
}
```

After Java 7

```
try (Declare resources here OR ELSE use local
variable referring to a declared resource)
{
      //Use resources here
}
catch (Exception e)
{
      //Catch exceptions here if any
}

//Resources are auto-closed
//No need to close resources explicitly
```

After Java 9

```
//Declare resources here

try (Pass reference of declared resources
here)
{
      //Use resources here
}
catch (Exception e)
{
      //Catch exceptions here if any
}

//Resources are auto-closed
//No need to close resources explicitly
```

Java I/O Cheat Sheet

javaconceptoftheday.com

What is stream?

A stream is a sequence of data generated by the input source and consumed by the output destination.

There are two types of I/O streams in Java.

1. Byte Streams
2. Character Streams

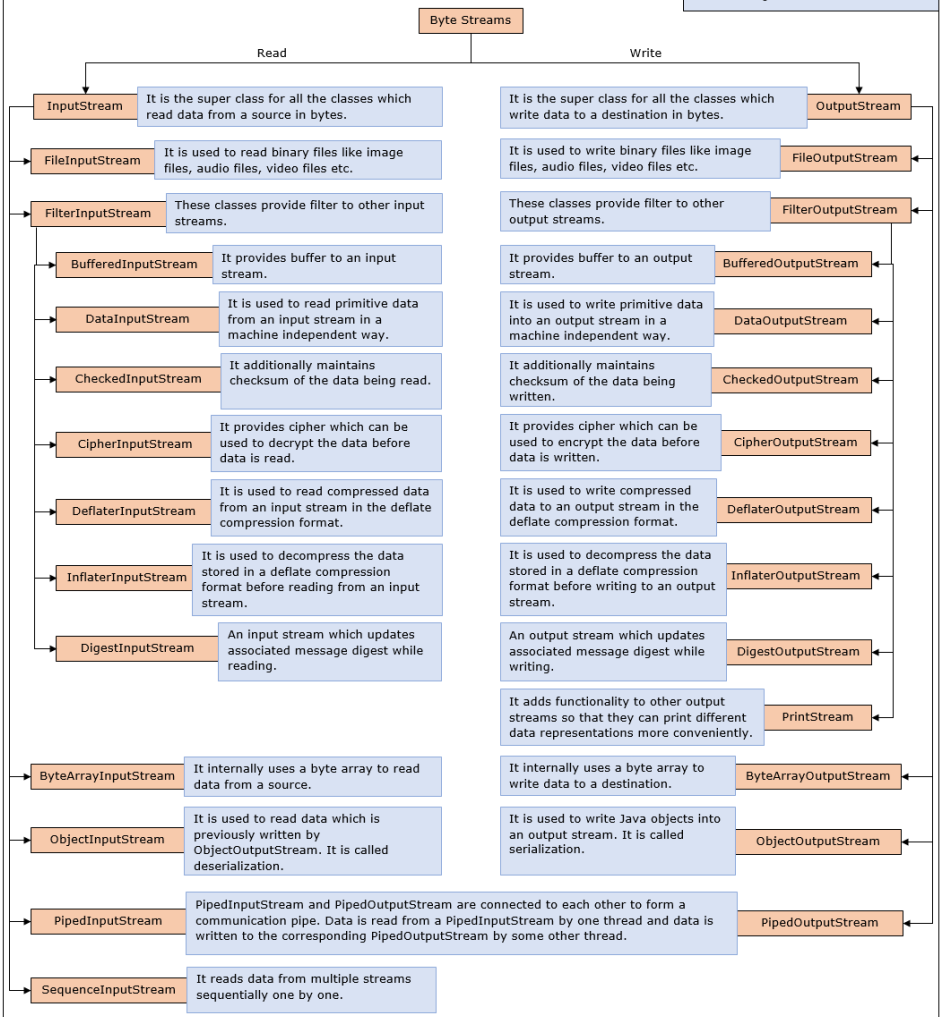
1) Byte Streams

- ✓ Byte streams read and write data byte by byte i.e. 8 bits maximum at a time.
- ✓ These streams are most suitable to process the binary files like image files, audio files, video files, executable files etc.
- ✓ All byte stream classes in Java are of type InputStream and OutputStream.
- ✓ InputStream classes are used to read data from a source and OutputStream classes are used to write data to a destination.

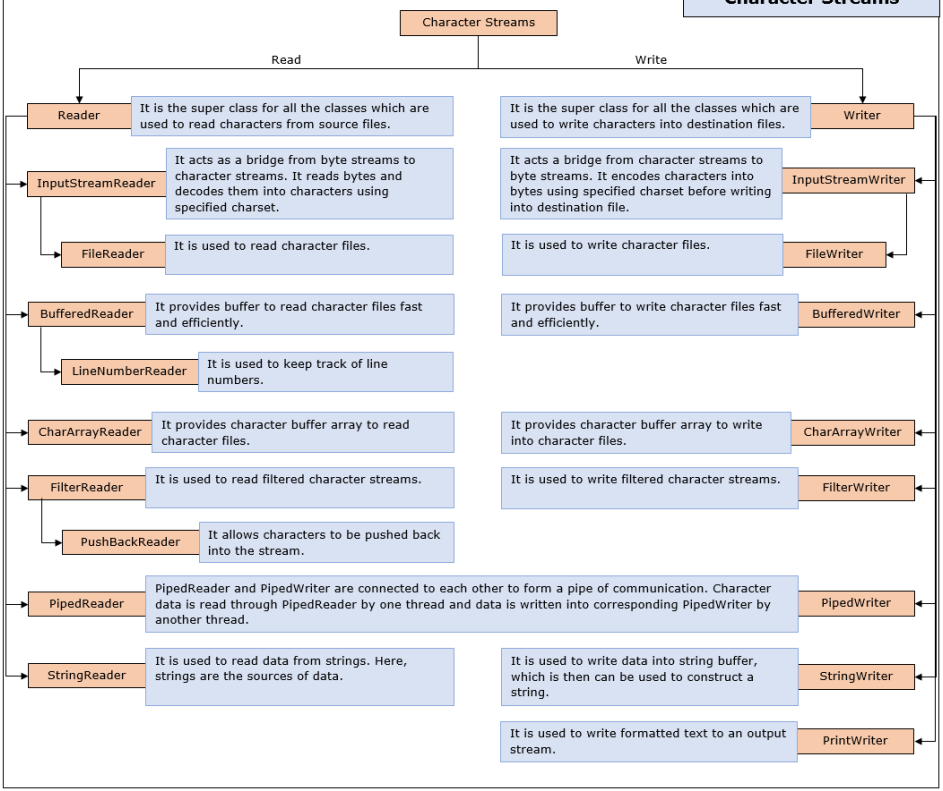
2) Character Streams

- ✓ Character streams read and write data character by character.
- ✓ Character streams are mainly used to process text files.
- ✓ All character stream classes in Java are of type Reader and Writer.
- ✓ Reader classes are used to read source files and Writer classes are used to write destination files.

Byte Streams



Character Streams



Java JDBC Cheat Sheet

Java Concept Of The Day

Basics

What is JDBC?

JDBC - Java Database Connectivity - is an API which is used by the Java applications to interact with the database management systems.

It consists of several classes and interfaces - written entirely in Java - which can be used to establish connection with the database, send the queries to the database and process the results returned by the database.

What are JDBC Drivers?

JDBC API doesn't directly interact with the database. It uses JDBC driver of that particular database with which it wants to interact.

JDBC drivers are nothing but the implementations of classes and interfaces provided in the JDBC API. These implementations are provided by a particular database vendor and supplied along with the database. These implementations are used by the JDBC API to interact with that database.

Types Of JDBC Drivers

There are four types of JDBC drivers.

1) Type 1 JDBC Drivers / JDBC-ODBC Bridge Drivers

This type of drivers translates all JDBC calls into ODBC calls and sends them to ODBC driver which interact with the database.

These drivers just acts as a bridge between JDBC and ODBC API and hence the name JDBC-ODBC bridge drivers.

They are partly written in Java.

2) Type 2 JDBC Drivers / Native API Drivers

This type of drivers translates all JDBC calls into database specific calls using native API of the database.

They are also not entirely written in Java.

3) Type 3 JDBC Drivers / Network Protocol Drivers

This type of drivers make use of application server or middle-tier server which translates all JDBC calls into database specific network protocol and then sent to the database.

They are purely written in Java.

4) Type 4 JDBC Drivers / Native Protocol Drivers

This type of JDBC drivers directly translate all JDBC calls into database specific network protocols without a middle tier.

They are most popular of all 4 type of drivers. They are also called thin drivers. They are entirely written in Java.

JDBC API

JDBC API is comprised of two packages - java.sql and javax.sql. Below are the some important classes and interfaces of JDBC API.

java.sql.DriverManager (Class) :

It acts as a primary mediator between your Java application and the driver of the database you want to connect with. Driver class of every database you want to connect with first has to get registered with this class before you start interacting with the database.

java.sql.Connection (Interface) :

It represents a session between Java application and a database. All SQL statements are executed and results are returned within the context of a Connection object. It is mainly used to create Statement, PreparedStatement and CallableStatement objects. You can also use it to retrieve the metadata of a database like name of the database product, name of the JDBC driver, major and minor version of the database etc...

java.sql.Statement (Interface) :

It is used to execute static SQL queries.

java.sql.PreparedStatement (Interface) :

It is used to execute parameterized or dynamic SQL queries.

java.sql.CallableStatement (Interface) :

It is used to execute SQL stored procedures.

java.sql.ResultSet (Interface) :

It contains the data returned from the database.

java.sql.ResultSetMetaData (Interface) :

This interface provides quick overview about a ResultSet object like number of columns, column name, data type of a column etc...

java.sql.DatabaseMetaData (Interface) :

It provides comprehensive information about a database.

java.sql.Date (Class) :

It represents a SQL date value.

java.sql.Time (Class) :

It represents a SQL time value.

java.sql.Blob (Interface) :

It represents a SQL BLOB (Binary Large Object) value. It is used to store/retrieve image files.

java.sql.Clob (Interface) :

It represents a SQL CLOB (Character Large Object) value. It is used to store/retrieve character files.

Database Connection Using JDBC API

Step 1 : Updating the class path with JDBC Driver

Add JDBC driver of a database with which you want to interact in the class path. JDBC driver is the jar file provided by the database vendors along with the database. It contains the implementations for all classes and interfaces of JDBC API with specific to that database.

Step 2 : Registering the driver class

```
Class.forName("Pass_Driver_Class_Here");
```

Step 3 : Creating the Connection object.

```
Connection con =
DriverManager.getConnection(URL,
username, password);
```

Step 4 : Creating the Statement Object

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

Step 5 : Execute the queries.

```
ResultSet rs = stmt.executeQuery("select *
from AnyTable");
```

Step 6 : Close the resources.

Close ResultSet, Statement and Connection objects.

Transaction Management

A transaction is a group of operations used to perform a particular task.

A transaction is said to be successful only if all the operations in a transaction are successful. If any one operation fails, the whole transaction will be cancelled.

In JDBC, transactions are managed using three methods of a Connection interface.

setAutoCommit() : It sets the auto commit mode of this connection object. By default it is true. It is set to false to manually manage the transactions.

commit() : It is called only when all the operations in a transaction are successful.

rollback() : It is called if any one operation in a transaction fails.

Batch Processing

Batch processing allows us to group similar queries into one unit and submit them all at once for execution. It reduces the communication overhead significantly and increases the performance.

Three methods of Statement interface are used for batch processing.

addBatch() : It is used to add SQL statement to the batch.

executeBatch() : It executes all SQL statements of a batch and returns an array of integers where each integer represents the status of a respective SQL statement.

clearBatch() : It removes all SQL statements added in a batch.

executeQuery() Vs executeUpdate() Vs execute()

executeQuery()	executeUpdate()	execute()
This method is used to execute the SQL statements which retrieve some data from the database.	This method is used to execute the SQL statements which update or modify the database.	This method can be used for any kind of SQL statements.
This method returns a ResultSet object which contains the results returned by the query.	This method returns an int value which represents the number of rows affected by the query. This value will be the 0 for the statements which return nothing.	This method returns a boolean value. TRUE indicates that query returned a ResultSet object and FALSE indicates that query returned an int value or returned nothing.
This method is used to execute only select queries.	This method is used to execute only non-select queries.	This method can be used for both select and non-select queries.
Ex: SELECT	Ex: DML → INSERT, UPDATE and DELETE DDL → CREATE, ALTER	This method can be used for any type of SQL statements.

Statement Vs PreparedStatement Vs CallableStatement

Statement	PreparedStatement	CallableStatement
It is used to execute normal SQL queries.	It is used to execute parameterized or dynamic SQL queries.	It is used to call the stored procedures.
It is preferred when a particular SQL query is to be executed only once.	It is preferred when a particular query is to be executed multiple times.	It is preferred when the stored procedures are to be executed.
You cannot pass the parameters to SQL query using this interface.	You can pass the parameters to SQL query at run time using this interface.	You can pass 3 types of parameters using this interface. They are – IN, OUT and IN OUT.
This interface is mainly used for DDL statements like CREATE, ALTER, DROP etc.	It is used for any kind of SQL queries which are to be executed multiple times.	It is used to execute stored procedures and functions.
The performance of this interface is very low.	The performance of this interface is better than the Statement interface (when used for multiple execution of same query).	The performance of this interface is high.

Java Strings Cheat Sheet

Java Concept Of The Day

Strings Basics

What are strings?

Strings are nothing but the sequence of characters enclosed within double quotes. For example, "ABC", "xyz", "123" etc.

How strings are represented in Java?

In some other languages, strings are represented as array of characters. But in Java, strings are represented as objects of java.lang.String class.

How do you create string objects in Java?

There are two ways to create string objects in Java.

1) Using String Literals

```
String s1 = "ABC";
String s2 = "123";
```

2) Using new Operator

```
String s1 = new String("ABC");
String s2 = new String("123");
```

How string objects are stored in the memory?

Whenever you create string objects using string literals, those objects will be stored in string constant pool and whenever you create string objects using new operator, such objects will be stored in normal heap memory.

String constant pool is a part of heap memory which is especially dedicated to store string objects. JVM allocates pool space to an object depending upon its content. There will be no two objects in the string constant pool with same content.

Whenever you create a string object using string literal, JVM first checks content of an object to be created. If there exist an object in the pool with same content, then it returns reference of that object. It doesn't create new object. If the content is different from the existing objects then only it creates new object.

java.lang.String objects are immutable :

java.lang.String objects, either created using string literals or using new operator, are immutable in nature. That means, once you create a string object, you can't modify the contents of that object. If you try to modify the contents of a string object, a new string object will be created with modified content.

java.lang.String Methods

charAt()
compareTo()
concat()
contains()
contentEquals()
copyValueOf()
endsWith()
startsWith()
equals()
equalsIgnoreCase()
format()
indexOf()
lastIndexOf()
intern()
isEmpty()
length()
matches()

replace()
replaceAll()
replaceFirst()
split()
subsequence()
substring()
toCharArray()
toLowerCase()
toUpperCase()
trim()
valueOf()

Java 8 :
join()

Java 9 :
chars()
codePoints()

Java 11 :
isBlank()
lines()
repeat()
strip()
stripLeading()
stripTrailing()

Java 12 :
indent()
transform()
describeConstable()
resolveConstantDesc()

Java 15 :
formatted()
stripIndent()
translateEscapes()

java.lang.StringBuffer Class

java.lang.StringBuffer class is used to create mutable and thread-safe string objects. In other terms, this class is same as java.lang.String class except its objects are mutable.

It is not possible to create StringBuffer objects using string literals. You have to use new operator to create StringBuffer objects.

Important Methods : append(), insert(), replace(), delete(), reverse(), length(), charAt() and substring().

java.lang.StringBuilder Class

java.lang.StringBuilder class is used to create mutable and non thread-safe string objects. In other terms, this class is same as java.lang.StringBuffer class except its objects are not thread-safe.

It is also not possible to create StringBuilder objects using string literals. You have to use new operator to create StringBuilder objects.

Important Methods : append(), insert(), replace(), delete(), reverse(), length(), charAt() and substring().

String Vs StringBuffer Vs StringBuilder

String	StringBuffer	StringBuilder
Immutable	Mutable	Mutable
Thread-safe	Thread-safe	Not thread-safe
Objects can be created either through string literal or through new operator.	Objects can be created only through new operator.	Objects can be created only through new operator.
Objects are stored in string constant pool as well as heap memory.	Objects are stored in heap memory only.	Objects are stored in heap memory only.
Slower	Slower	Faster

String Intern

String intern refers to string object in the string constant pool.

Interning is the process of creating a string object in String Constant Pool which will be exact copy of string object in heap memory.

intern() method of java.lang.String class is used to perform interning i.e. creating an exact copy of heap string object in string constant pool.

Java Thread Cheat Sheet

javaconceptoftheday.com

Basic Definitions

What is thread?

Thread is a smallest executable unit of a process. Thread has its own path of execution in a process. A process can have multiple threads.

What is process?

Process is an executing instance of an application. For example, when you double click MS Word icon in your computer, you start a process that will run MS word application.

What is application?

Application is a program which is designed to perform a specific task. For example : MS Word, Google Chrome, a video or audio player etc.

What is multithreaded programming?

In a program or in an application, when two or more threads execute their task simultaneously then it is called multithreaded programming. Java supports multithreaded programming.

Types Of Threads

There are two types of threads in Java.

1) User Threads :

User threads are threads which are created by the application or user. They are high priority threads. JVM will not exit until all user threads finish their execution. JVM wait for user threads to finish their task. These threads are foreground threads.

2) Daemon Threads :

Daemon threads are threads which are mostly created by the JVM. These threads always run in background. These threads are used to perform some background tasks like garbage collection. These threads are less priority threads. JVM will not wait for these threads to finish their execution. JVM will exit as soon as all user threads finish their execution.

Thread Priority

MIN_PRIORITY :

It defines the lowest priority that a thread can have and it's value is 1.

NORM_PRIORITY :

It defines the normal priority that a thread can have and it's value is 5.

MAX_PRIORITY :

It defines the highest priority that a thread can have and it's value is 10.

The default priority of a thread is same as that of it's parent. We can change the priority of a thread at any time using setPriority() method.

Thread States

There are six thread states - NEW, RUNNABLE, BLOCKED, WAITING, TIMED _WAITING and TERMINATED. At any point of time, a thread will be in any one of these states.

NEW : A thread will be in this state before calling start() method.

RUNNABLE : A thread will be in this state after calling the start() method.

BLOCKED : A thread will be in this state when a thread is waiting for object lock to enter into synchronized method/block or a thread will be in this state if deadlock occurs.

WAITING : A thread will be in this state when wait() or join() method is called.

TIMED_WAITING : A thread will be in this state when sleep() or wait() with timeOut or join() with timeOut is called.

TERMINATED : A thread will be in this state once it finishes it's execution.

How do you create threads in Java?

There are two ways to create threads in Java.

1) By extending java.lang.Thread class

```
class MyThread extends Thread
{
   @Override
   public void run()
   {
      //Keep the task to be performed here
   }
}

//Creating and starting MyThread
MyThread myThread = new MyThread();
myThread.start();
```

2) By implementing java.lang.Runnable interface

```
class MyRunnable implements Runnable
{
   @Override
   public void run()
   {
      //Keep the task to be performed here
   }
}

//Creating and starting MyRunnable
Thread t = new Thread(new MyRunnable());
t.start();
```

Thread Synchronization

Through synchronization, we can make the threads to execute a particular method or block in sync not simultaneously. Synchronization in Java is achieved using **synchronized** keyword.

When a method or block is declared as synchronized, only one thread can enter into that method or block.

The synchronization in Java is built around an entity called object lock or monitor.

Any thread wants to enter into synchronized methods or blocks of any object, they must acquire object lock associated with that object and release the lock after they are done with the execution.

```
synchronized void synchronizedMethod()
{
        //Synchronized Method
}
```

Deadlock

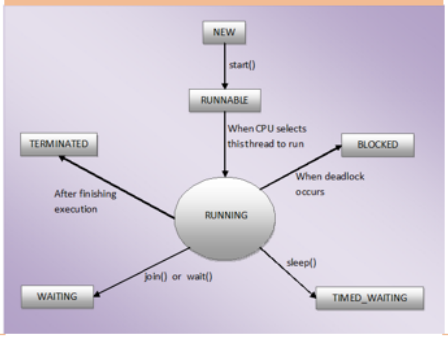
Deadlock in Java is a condition which occurs when two or more threads get blocked waiting for each other for an infinite period of time to release the resources (Locks) they hold.

Lock ordering and lock timeout are two methods which are used to avoid the deadlock in Java.

Lock Ordering : In this method of avoiding the deadlock, some predefined order is applied for threads to acquire the locks they need.

Lock Timeout : It is another deadlock preventive method in which we specify the time for a thread to acquire the lock. If it fails to acquire the specified lock in the given time, then it should give up trying for a lock and retry after some time.

Thread Life Cycle



Java.lang.Thread Methods

start() :

It starts execution of a thread.

run() :

It contains main task to be performed by the thread.

sleep() :

It makes the currently executing thread to pause it's execution for a specified period of time. When the thread is going for sleep, it does not release the locks it holds.

join() :

Using this method, you can make the currently executing thread to wait for some other threads to finish their task.

yield() :

It causes the currently executing thread to temporarily pause its execution and allow other threads to execute.

wait() :

It makes the currently executing thread to release the lock of this object and wait until some other thread notifies it.

notify() :

It wakes up one thread randomly which is waiting for this object's lock.

notifyAll() :

It wakes up all thread which are waiting for this object's lock. But, only one thread will acquire lock of this object depending upon the priority.

isAlive() :

It checks whether a thread is alive or not.

isDaemon() :

It checks whether a thread is daemon thread or user thread.

setDaemon() :

It sets daemon status of a thread.

currentThread() :

It returns a reference to currently executing thread.

interrupt() :

It is used to interrupt a thread.

isInterrupted() :

It checks whether a thread is interrupted or not.

getId() :

It returns ID of a thread.

getState() :

It returns current state of a thread.

getName() and setName() :

Getter and setter for name of a thread.

getPriority() and setPriority() :

Getter and setter for priority of a thread.

getThreadGroup() :

It returns a thread group to which this thread belongs to.

Inter Thread Communication

Threads in Java communicate with each other using wait(), notify() and notifyAll() methods.

wait() : This method tells the currently executing thread to release the lock of this object and wait until some other thread acquires the same lock and notify it using either notify() or notifyAll() methods.

notify() : This method wakes up one thread randomly that called wait() method on this object.

notifyAll() : This method wakes up all the threads that called wait() method on this object. But, only one thread will acquire lock of this object depending upon the priority.

Inheritance

Access Specifiers

	public	protected	internal	protected internal	private	private protected
Entire program	Yes	No	No	No	No	No
Containing class	Yes	Yes	Yes	Yes	Yes	Yes
Current assembly	Yes	No	Yes	Yes	No	No
Derived types	Yes	Yes	No	Yes	No	No
Derived types within current assembly	Yes	Yes	Yes	Yes	No	Yes