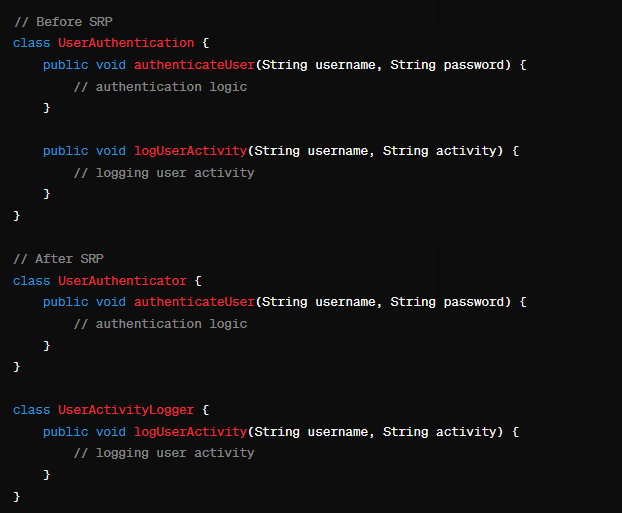
**SOLID PRINCIPLE**

SOLID is an acronym that represents a set of principles in object-oriented programming designed to make software more scalable, flexible, and maintainable.

1. **Single Responsibility Principle (SRP):**
   * This principle suggests that a class should have only one reason to change, meaning it should have only one responsibility or job.
   * Example with Netflix: In Netflix, you could have a class responsible for handling user authentication and another class for managing the user's watch history. Each class has a single responsibility.



1. **Open/Closed Principle (OCP):**
   * This principle states that a class should be open for extension but closed for modification. It encourages adding new features by extending existing code rather than changing it.
   * Example with Netflix: If Netflix wants to introduce a new genre of movies, they should create a new class or module for that genre without modifying the existing movie classes.

A screen shot of a computer program

Description automatically generated

1. **Liskov Substitution Principle (LSP):**
   * This principle emphasizes that objects of a superclass should be able to be replaced with objects of a subclass without affecting the correctness of the program.
   * Example with Netflix: If you have a class representing a generic "Movie," you should be able to replace it with a subclass like "Documentary" without breaking the functionality of the program.

A screen shot of a computer

Description automatically generated A screen shot of a computer program

Description automatically generated

1. **Interface Segregation Principle (ISP):**
   * This principle suggests that a class should not be forced to implement interfaces it does not use. It's better to have multiple small, specific interfaces rather than a large, generic one.
   * Example with Netflix: Instead of having a single interface for all streaming services, you could have separate interfaces for features like streaming, user ratings, and search functionalities. This way, a class can implement only the interfaces relevant to its functionality.

A screen shot of a computer program

Description automatically generated A screen shot of a computer screen

Description automatically generated

1. **Dependency Inversion Principle (DIP):**
   * This principle encourages high-level modules to depend on abstractions rather than on low-level modules. It also promotes that abstractions should not depend on details; details should depend on abstractions.
   * Example with Netflix: Instead of having a class directly depend on a specific payment gateway implementation for processing subscription payments, it could depend on an interface (abstraction) for payment processing. This allows for flexibility, as you can easily switch between different payment gateway implementations without changing the high-level class. The high-level module (e.g., SubscriptionManager) depends on the abstraction (e.g., PaymentProcessor interface), and the low-level details (specific payment gateway implementations) depend on the same abstraction. This way, changes in the payment processing details won't directly impact the higher-level subscription management logic.

A screen shot of a computer program

Description automatically generatedA screen shot of a computer program

Description automatically generated

**DESIGN PATTERNS**

Design patterns are reusable solutions to common problems that arise in software design. They provide a way to structure code in a flexible and modular manner. There are several categories of design patterns.

## Creational Design Pattern

1. Factory Pattern
2. Abstract Factory Pattern
3. Singleton Pattern
4. Prototype Pattern
5. Builder Pattern.

## Structural Design Pattern

1. Adapter Pattern
2. Bridge Pattern
3. Composite Pattern
4. Decorator Pattern
5. Facade Pattern
6. Flyweight Pattern
7. Proxy Pattern

## Behavioural Design Pattern

1. Chain Of Responsibility Pattern
2. Command Pattern
3. Interpreter Pattern
4. Iterator Pattern
5. Mediator Pattern
6. Memento Pattern
7. Observer Pattern
8. State Pattern
9. Strategy Pattern
10. Template Pattern
11. Visitor Pattern

**1. Creational Design Patterns:**

* These patterns deal with the process of object creation, providing mechanisms to instantiate objects in a manner suitable to the situation.
* Examples include:
  + **Singleton Pattern:** Ensures a class has only one instance and provides a global point of access to that instance.

There are two forms of singleton design pattern.

* **Early Instantiation:** creation of instance at load time.
* **Lazy Instantiation:** creation of instance when required.

#### Advantage of Singleton design pattern

* Saves memory because object is not created at each request. Only single instance is reused again and again.

#### How to create Singleton design pattern?

To create the singleton class, we need to have static members of class, private constructor, and static factory method.

* **Static member:** It gets memory only once because of static, it contains the instance of the Singleton class.
* **Private constructor:** It will prevent to instantiate the Singleton class from outside the class.
* **Static factory method:** This provides the global point of access to the Singleton object and returns the instance to the caller.
  + **Factory Method Pattern:** Defines an interface for creating an object but let’s subclasses alter the type of objects that will be created.
  + **Builder Pattern:** Separates the construction of a complex object from its representation, allowing the same construction process to create various representations.

**2. Structural Design Patterns:**

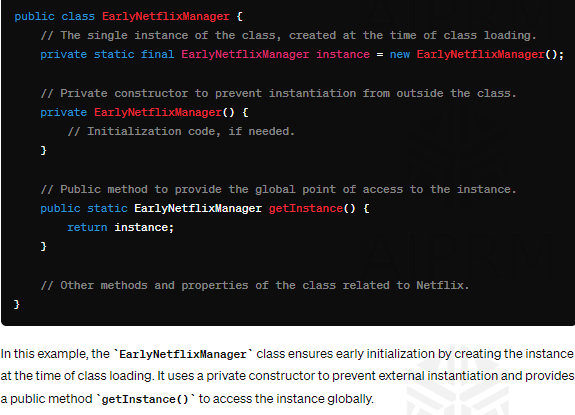
* These patterns are concerned with the composition of classes and objects, creating larger structures from individual parts.
* Examples include:
  + **Adapter Pattern:** Allows the interface of an existing class to be used as another interface.
  + **Decorator Pattern:** Attaches additional responsibilities to an object dynamically.
  + **Composite Pattern:** Composes objects into tree structures to represent part-whole hierarchies.

**3. Behavioural Design Patterns:**

* These patterns define the ways in which objects interact, encapsulating how they communicate and collaborate.
* Examples include:
  + **Observer Pattern:** Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
  + **Strategy Pattern:** Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
  + **Command Pattern:** Encapsulates a request as an object, thereby allowing for parameterization of clients with different requests, queuing of requests, and logging of the parameters for requests.

#Examples (Creational Design Pattern):

1. **Singleton Design Pattern:**



A screen shot of a computer

Description automatically generated

The second example with lazy initialization (**LazyNetflixManager**) is not inherently thread safe. In a multi-threaded environment, there is a possibility that multiple threads may simultaneously check if **instance** is null and try to create a new instance, leading to the creation of multiple instances of the singleton.

To address this issue and make the lazy initialization thread-safe, you can use synchronization or the double-checked locking idiom. Here's an updated version using synchronization:

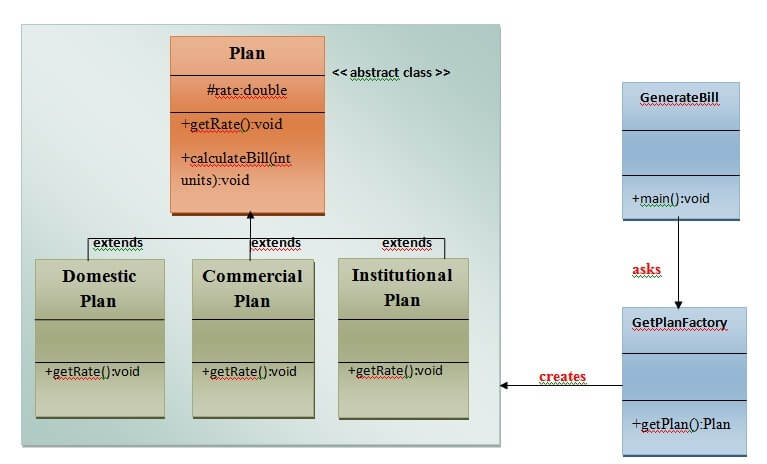
A screen shot of a computer program

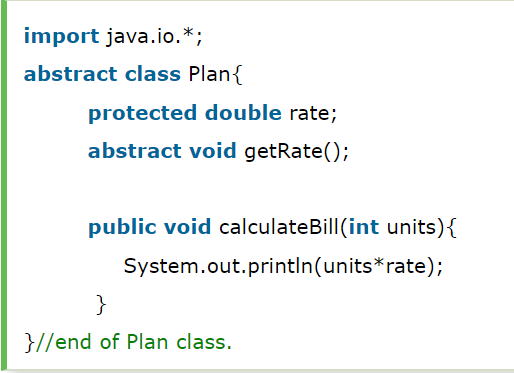
Description automatically generated

In this modified version, the **synchronized** keyword is used to ensure that only one thread can access the critical section at a time. The **volatile** keyword is also used to ensure that changes to the instance variable are visible to all threads.

Note: While this approach addresses the thread-safety issue, it introduces some performance overhead due to synchronization.

1. Factory Design Pattern:



A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated

**#Examples (Structural Design Pattern):**

Structural design patterns are concerned with how classes and objects can be composed, to form larger structures.

The structural design patterns simplify the structure by identifying the relationships.

These patterns focus on how the classes inherit from each other and how they are composed from other classes.

* **Adapted Design Pattern**

The Adapter Pattern is a structural design pattern that allows the interface of an existing class to be used as another interface. It enables incompatible interfaces to work together, acting as a bridge between them.

**A screenshot of a computer program

Description automatically generated**

In this scenario, the VideoPlayerAdapter acts as a bridge, allowing the existing LegacyVideoPlayer to be used seamlessly as a MediaPlayer. The client code interacts with the MediaPlayer interface, and the adapter ensures the correct communication with the legacy code.

**#Examples (Behavioural Design Pattern):**

Behavioral design patterns are concerned with **the interaction and responsibility of objects.**

In these design patterns,**the interaction between the objects should be in such a way that they can easily talk to each other and still should be loosely coupled.**

That means the implementation and the client should be loosely coupled in order to avoid hard coding and dependencies.

* **Command Design Pattern:** The Command Pattern is a behavioural design pattern that turns a request into a stand-alone object, containing all information about the request. This transformation allows for parameterization of clients with different requests, queuing of requests, and logging of the parameters.

**Demonstration Without Command Pattern:**

Let's imagine a simple scenario where a remote control is used to control various electronic devices. Without the Command Pattern, the remote control directly calls methods on each device.

**A screenshot of a computer program

Description automatically generated**

**Demonstration With Command Pattern:**

**A screen shot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

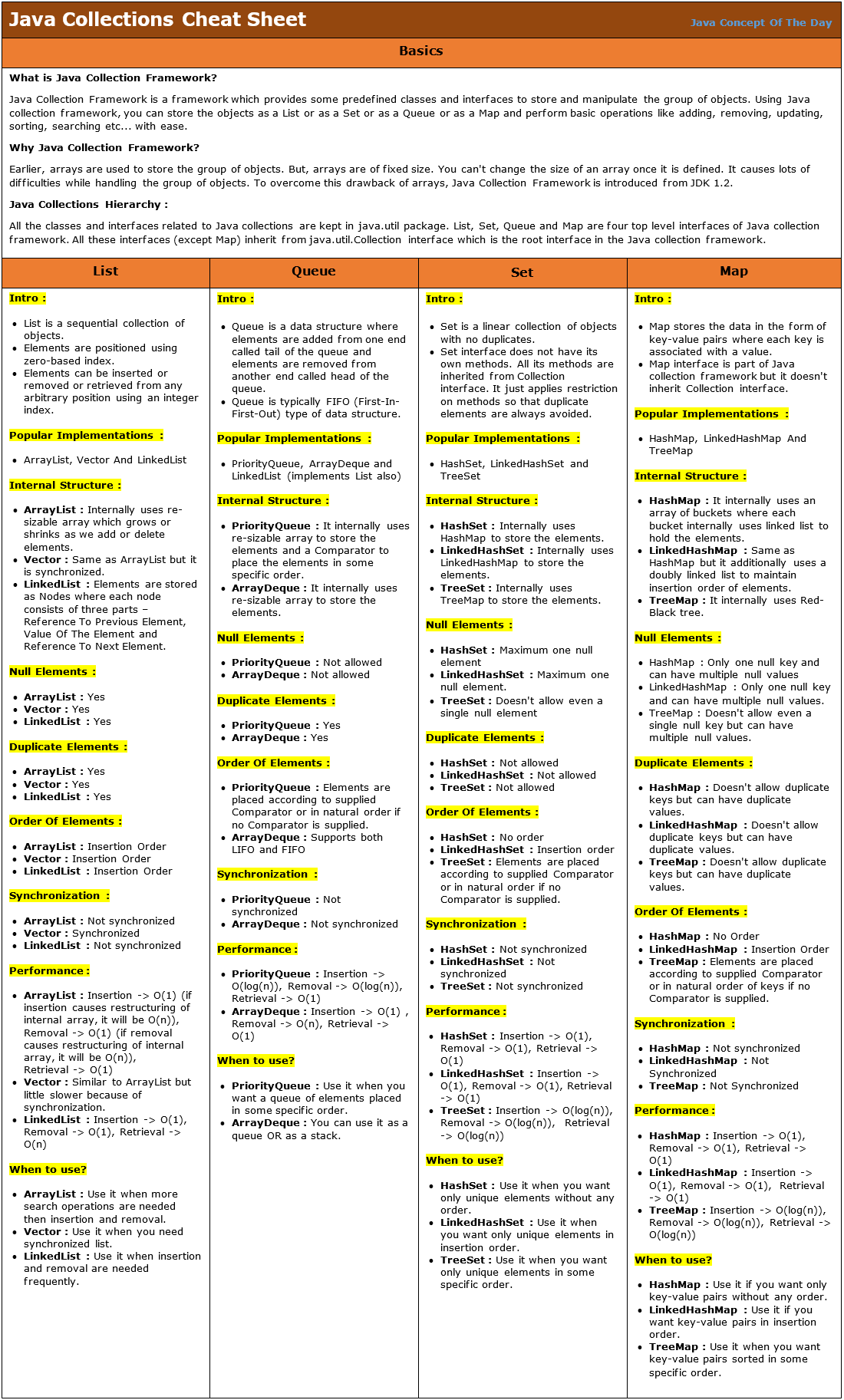
**JAVA Questions**

* **Why wait notify and notify all methods are in Object class?**
* **What are the important features of Java 8 release?**
* **What is abstraction and how you achieved it?**
* **What is polymorphism and how you achieved it?**
* **What is encapsulation and how you achieved it?**
* **What is Inheritance and how you achieved it?**
* **What is garbage collector and how to call it**
* **What is overloading and overriding in java?**
* **In try catch if we return and we also have finally. what will happen?**
* **Can we declare a class as static?**
* **What is try-with-resources in java?**
* **What is multi-catch block in java? Can we have higher exception first?**
* **What is the difference between abstract class and interface?**
* **What is Marker interface?**
* **Why String is immutable or final in Java?**
* **Why is Char array preferred over String for storing password?**
* **Is String thread-safe in Java?**
* **What is the difference between String, StringBuilder, and StringBuffer in Java?**
* **What is the impact of declaring a method as final?**
* **I don't want my class to be inherited by any other class. What should I do?**
* **How is final different from finally and finalize ()?**
* **Can a class be declared as static?**
* **I want to print "Hello" even before main () is executed. How will you achieve that?**
* **Can we declare a static variable inside a method? - No**
* **Can a abstract class be declared final? - No**
* **Can you create an object of an abstract class? - No**
* **Can a method inside a Interface be declared as final? - No**
* **What does it mean that a method or field is "static"?**
* **Name the eight primitive Java types.**
* **What is switch case and can we use string in it?**
* **What is the difference between a break statement and a continue statement?**
* **How are this() and super() used with constructors?**
* **What is a transient variable?**
* **Volatile**
* **Explain different ways of creating a thread?**
* **Difference between Array list And Vector.**
* **Differentiate between Iterator and Enumeration?**
* **Difference between the Inner Class and Sub Class.**
* **What’s the base class of all exception classes?**
* **Write a code to make Collections readOnly?**
* **Difference between Collections and Collection**
* **Difference between Array and Array List.**
* **What is string pool?**
* **Difference between HashMap and HashTable.**
* **Difference between HashSet and TreeSet.**
* **What are the types of Exceptions?**
* **What are the different ways to handle exceptions?**
* **Difference between throw and throws**
* **What is SerialVersionUID?**
* **What are the different ways to create arrays in Java?**
* **Can you change the size of an array once you define it?**
* **Write a program to print all permutations of String?**
* **How to sort Employee class in Java?**
* **Write code to check a String is palindrome or not?**
* **Write a method which will remove any given character from a String?**
* **How to count the occurrence of a given character in a String?**
* **How to check if two String are Anagram?**
* **How do you find the second highest number in an integer array?**
* **How do you count the number of occurrences of each character in a string?**
* **How do you remove all white spaces from a string in Java?**
* **Write a Java program to reverse a string?**
* **Write a Java program to reverse a given string with preserving the position of spaces?**

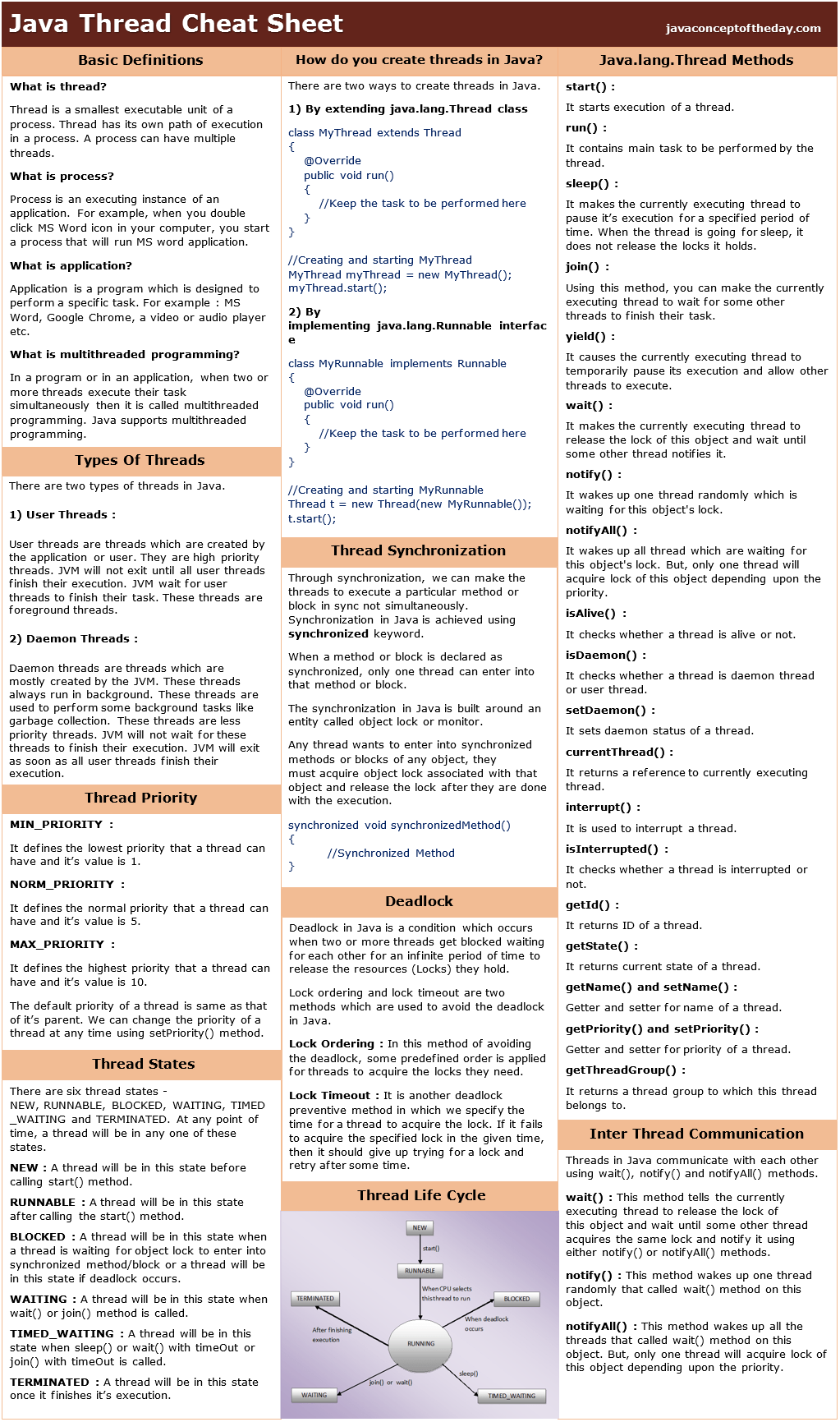
For example, if **“I Am Not String”** is the given string then the reverse of this string with preserving the position of spaces is **“g ni rtS toNmAI”**.

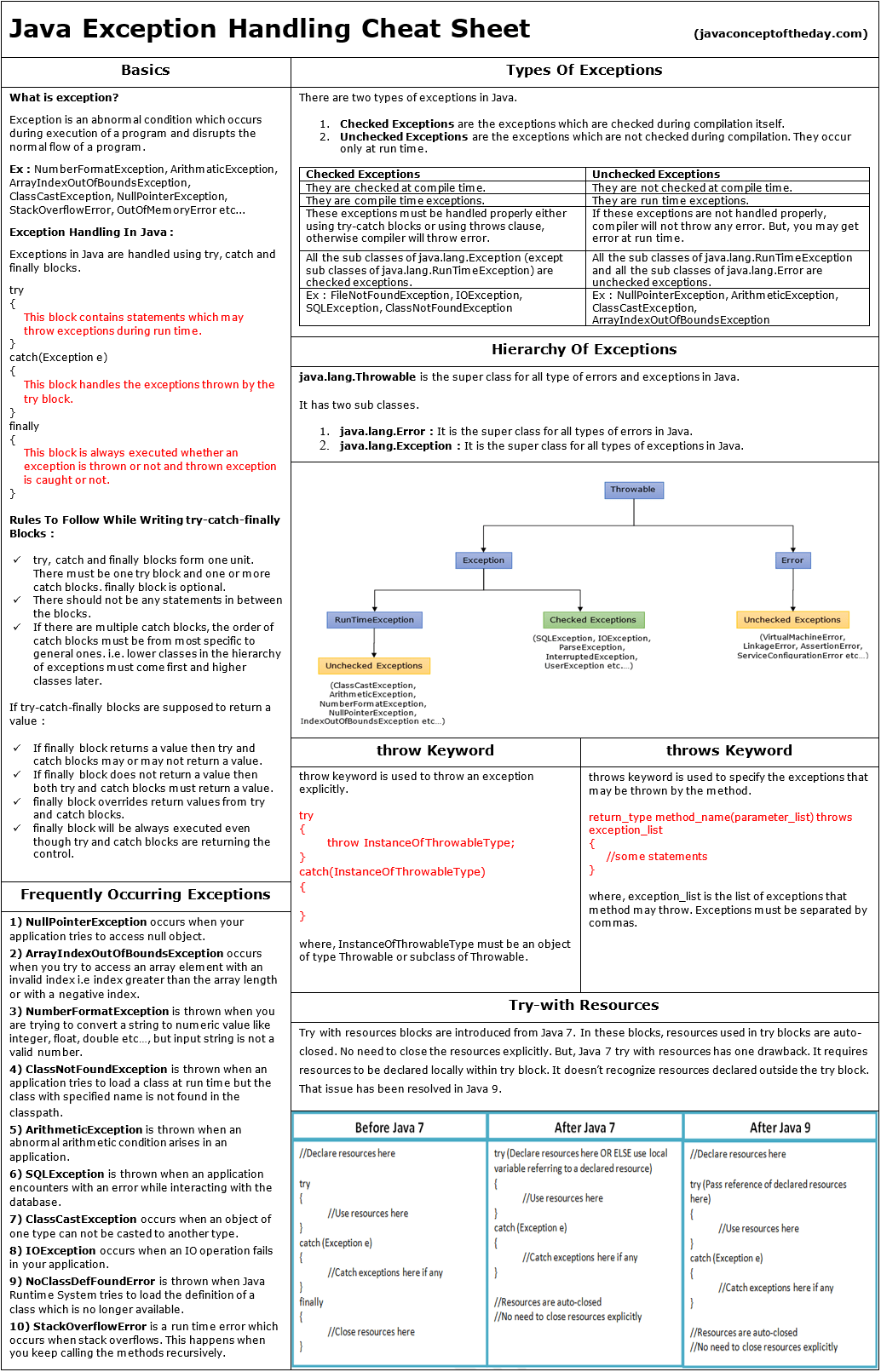
* **How do you convert string to integer and integer to string in Java?**
* **Write a code to prove that strings are immutable in Java?**
* **Write a code to check whether one string is a rotation of another?**
* **How do you find longest substring without repeating characters in the given string?**
* **How to remove all vowels from a string in Java?**
* **How do you swap two string variables without using third or temp variable in Java?**

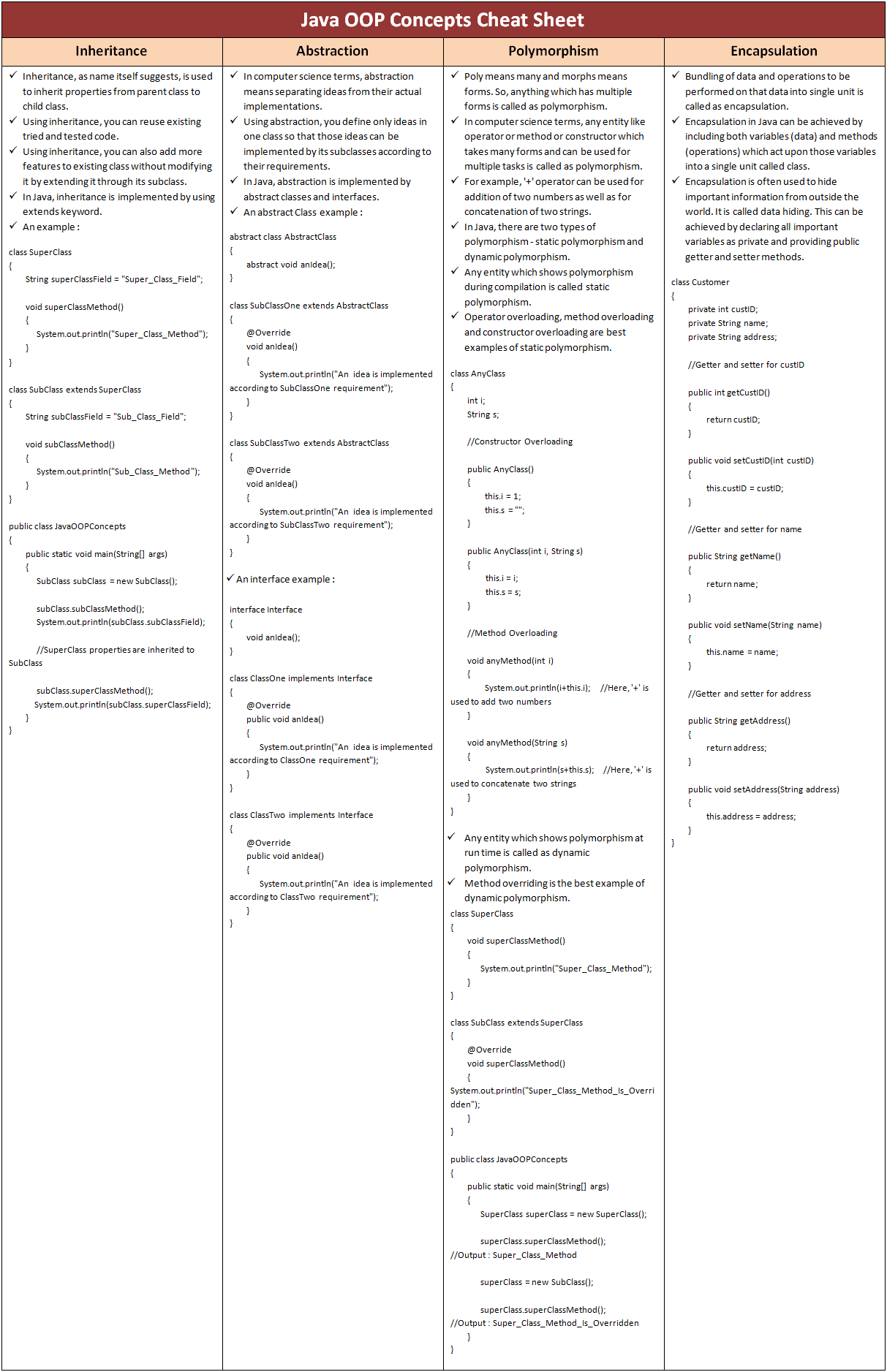
**Collection cheat sheet:**

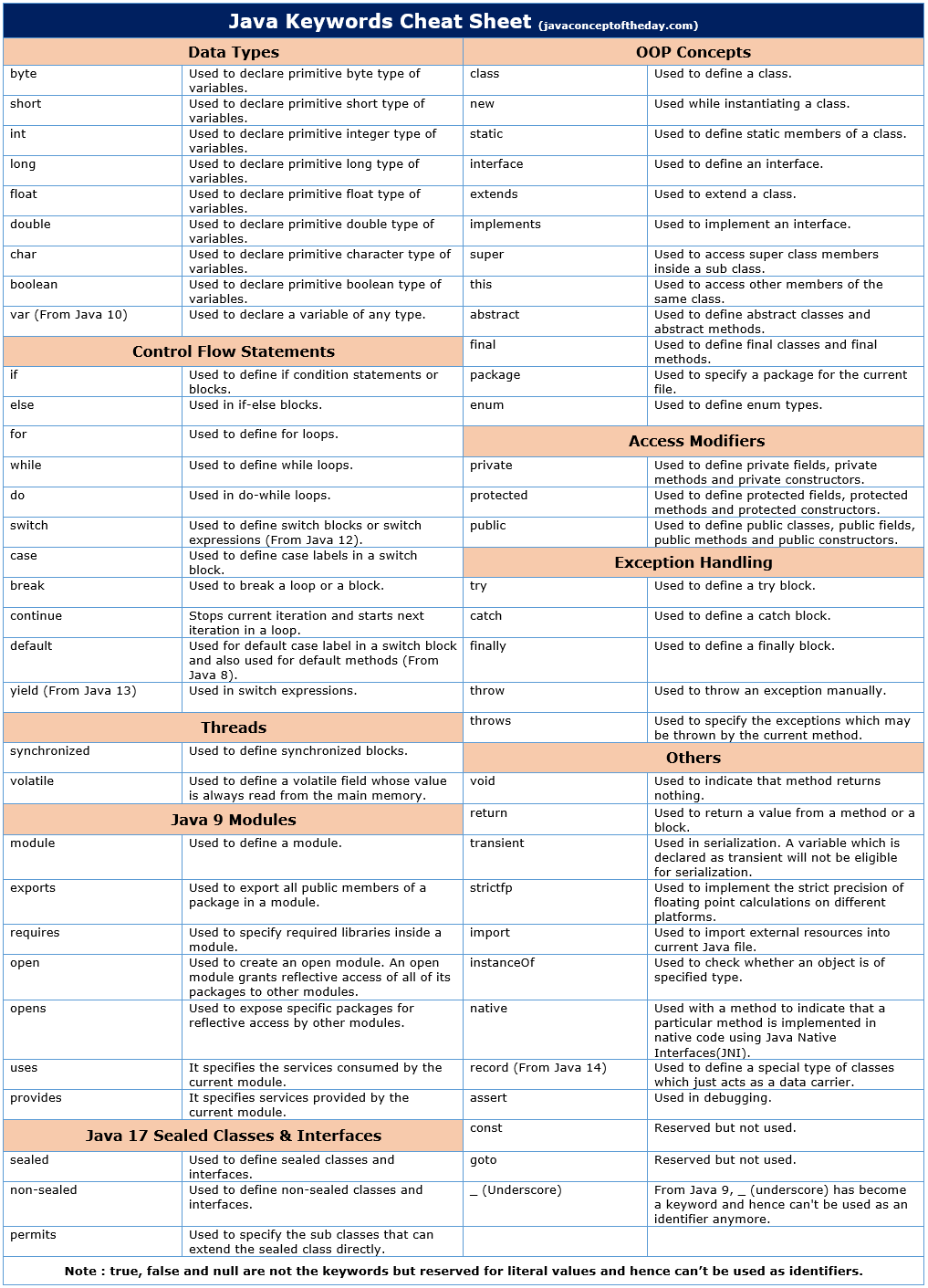


**Thread Cheat Sheet**

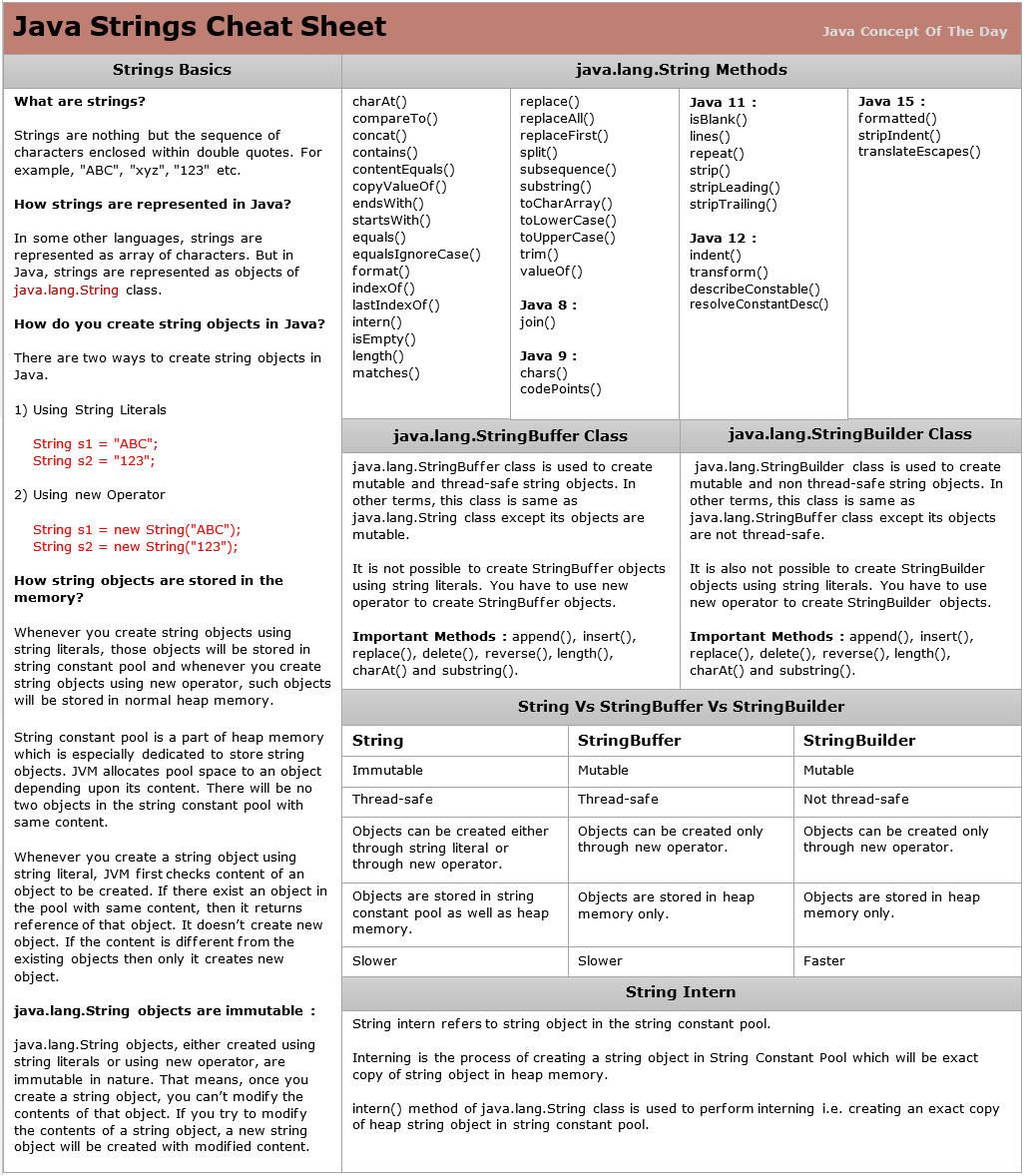


**Exception handling cheet sheet**

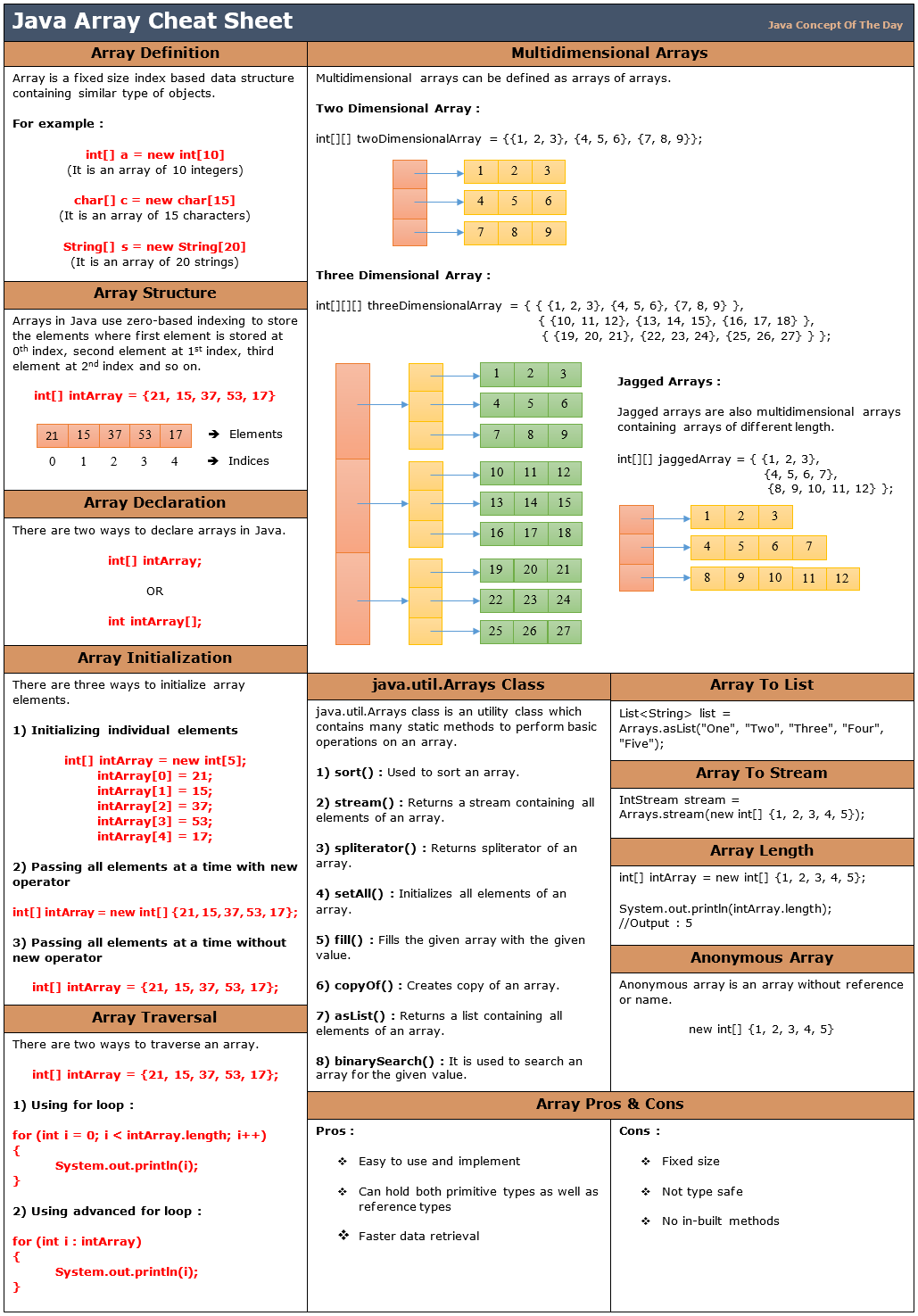
**OOPS cheat sheet**

**Java Keywords Cheat Sheet** 

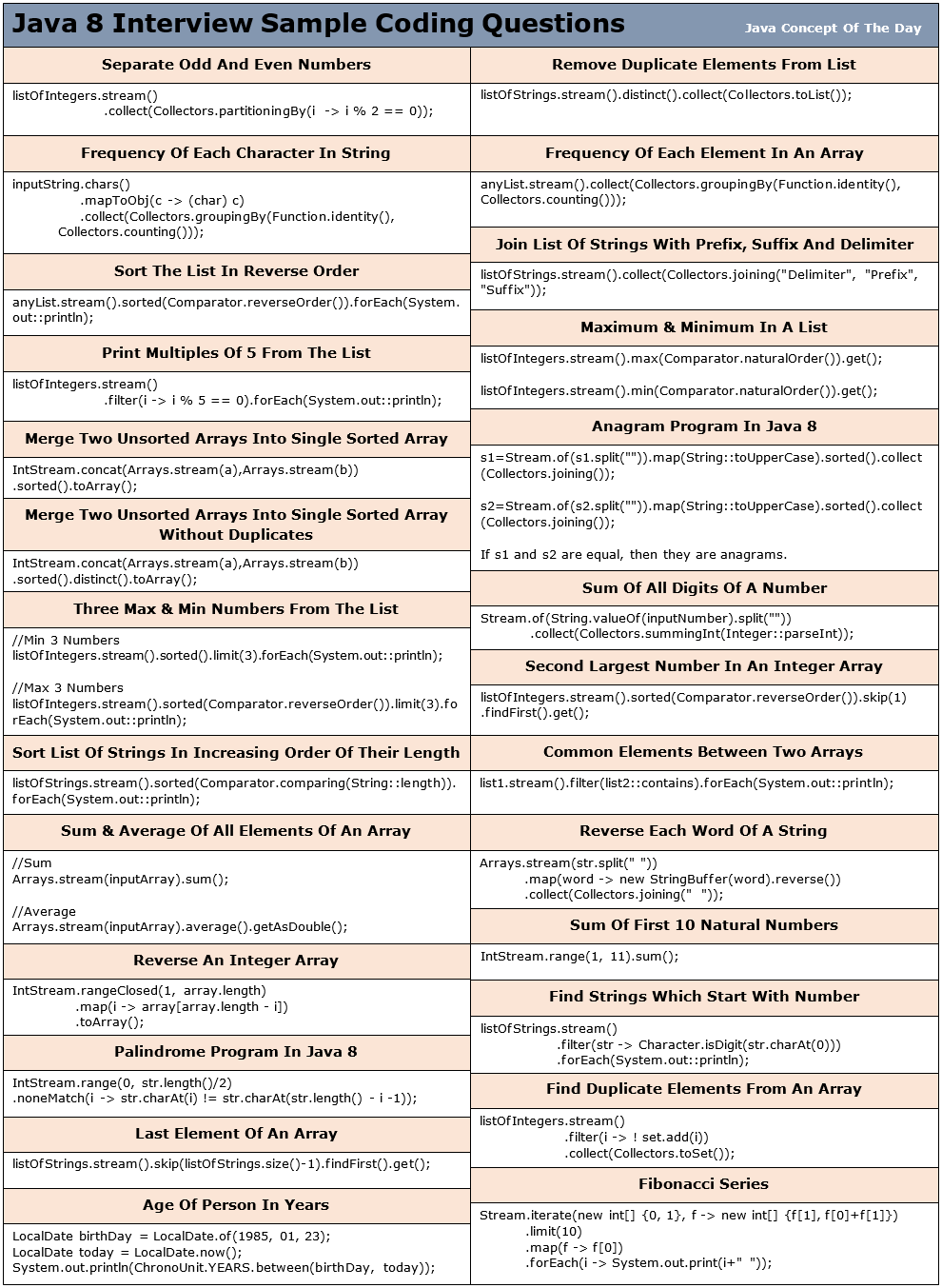
**String Cheat Sheet**



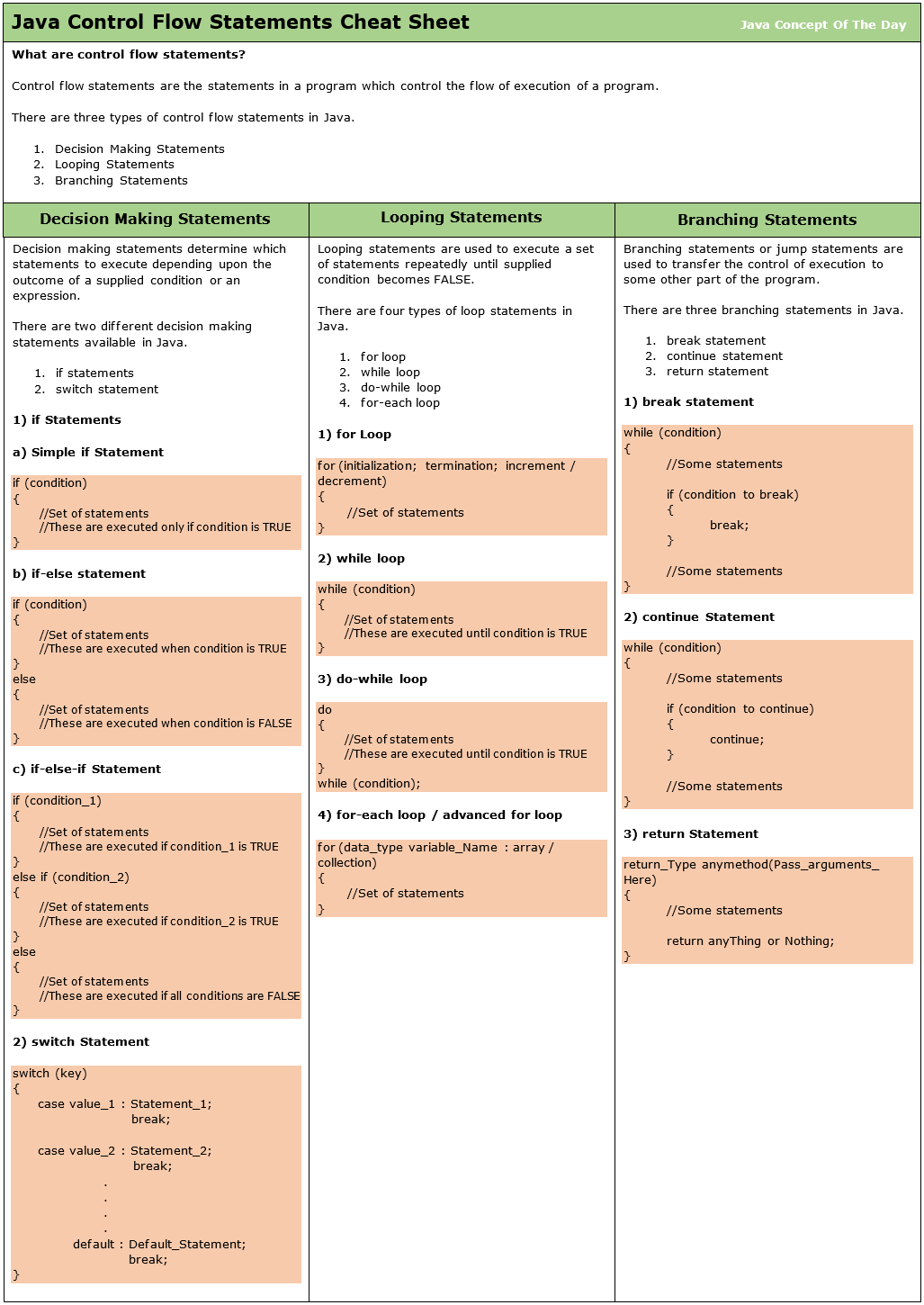
**Array Cheat Sheet**



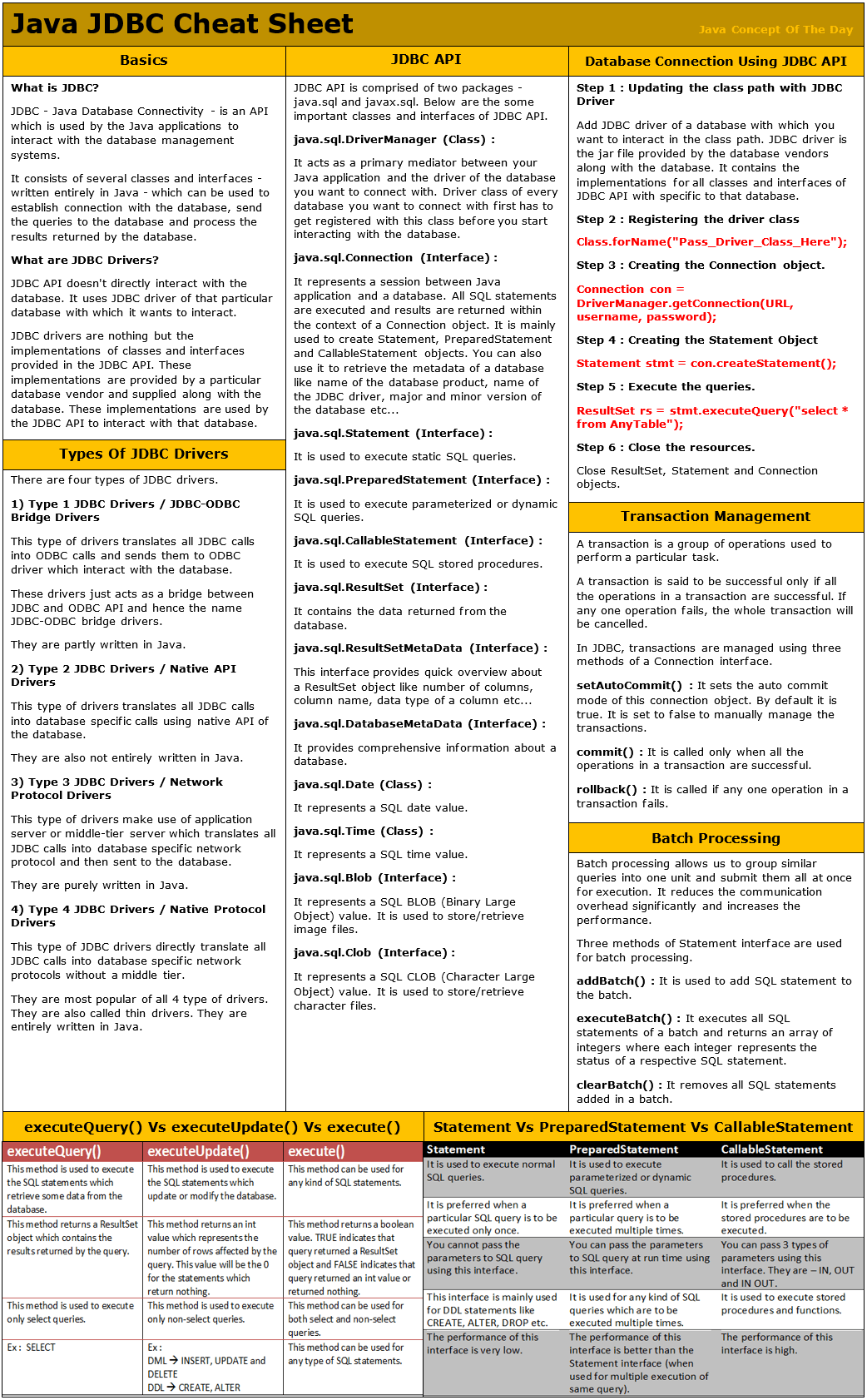
**Java 8 Code cheat sheet**



**Control Flow Cheat Sheet**



**JDBC Cheat Sheet**



**Key Differences**

**A white sheet with black text

Description automatically generated**

|  |  |
| --- | --- |
| **Array** | **ArrayList** |
| Arrays are static in nature. Arrays are fixed length data structures. You can’t change their size once they are created. | ArrayList is dynamic in nature. Its size is automatically increased if you add elements beyond its capacity. |
| Arrays can hold both primitives as well as objects. | ArrayList can hold only objects. |
| Arrays can be iterated only through *for* loop or *for-each* loop. | ArrayList provides iterators to iterate through their elements. |
| The size of an array is checked using *length*attribute. | The size of an ArrayList can be checked using *size()* method. |
| Array gives constant time performance for both add and get operations. | ArrayList also gives constant time performance for both add and get operations provided adding an element doesn’t trigger resize. |
| Arrays don’t support generics. | ArrayList supports generics. |
| Arrays are not type safe. | ArrayList are type safe. |
| Arrays can be multi-dimensional. | ArrayList can’t be multi-dimensional. |
| Elements are added using assignment operator. | Elements are added using add() method. |

|  |  |
| --- | --- |
| **StackOverflowError** | **OutOfMemoryError** |
| It is related to Stack memory. | It is related to heap memory. |
| It occurs when Stack is full. | It occurs when heap is full. |
| It is thrown when you call a method and there is no space left in the stack. | It is thrown when you create a new object and there is no space left in the heap. |
| It occurs when you are calling a method recursively without proper terminating condition. | It occurs when you are creating lots of objects in the heap memory. |
| How to avoid? Make sure that methods are finishing their execution and leaving the stack memory. | How to avoid? Try to remove references to objects which you don’t need anymore. |

|  |  |
| --- | --- |
| **Shallow Copy** | **Deep Copy** |
| Cloned Object and original object are not 100% disjoint. | Cloned Object and original object are 100% disjoint. |
| Any changes made to cloned object will be reflected in original object or vice versa. | Any changes made to cloned object will not be reflected in original object or vice versa. |
| Default version of clone method creates the shallow copy of an object. | To create the deep copy of an object, you have to override clone method. |
| Shallow copy is preferred if an object has only primitive fields. | Deep copy is preferred if an object has references to other objects as fields. |
| Shallow copy is fast and also less expensive. | Deep copy is slow and very expensive. |

|  |  |
| --- | --- |
| **“==” Operator** | **equals() Method** |
| It is a binary operator in Java. | It is a public method of java.lang.Object class. |
| It compares the two objects based on their location in the memory. | The default version of equals method also does the comparison of two objects based on their location in the memory. But, you can override the equals method so that it performs the comparison of two objects on some condition. |
| It can be used on both primitive types as well as on derived types. | It can be used only on derived types. |
| It is best suitable for primitive types. | It is best suitable for derived types. |
| You can’t override the “==” operator. It behaves same for all objects. | You can override the equals method according to your business requirements. |

|  |  |
| --- | --- |
| **Errors** | **Exceptions** |
| Errors in Java are of type java.lang.Error. | Exceptions in Java are of type java.lang.Exception. |
| All errors in Java are unchecked type. | Exceptions include both checked as well as unchecked type. |
| Errors happen at run time. They will not be known to compiler. | Checked exceptions are known to compiler where as unchecked exceptions are not known to compiler because they occur at run time. |
| It is impossible to recover from errors. | You can recover from exceptions by handling them through try-catch blocks. |
| Errors are mostly caused by the environment in which application is running. | Exceptions are mainly caused by the application itself. |
| Examples : java.lang.StackOverflowError, java.lang.OutOfMemoryError | Examples : Checked Exceptions : SQLException, IOException Unchecked Exceptions : ArrayIndexOutOfBoundException, ClassCastException, NullPointerException |
| Class Variables | Instance Variables |
| Class variables are declared with keyword static. | Instance variables are declared without static keyword. |
| Class variables are common to all instances of a class. These variables are shared between the objects of a class. | Instance variables are not shared between the objects of a class. Each instance will have their own copy of instance variables. |
| As class variables are common to all objects of a class, changes made to these variables through one object will reflect in another. | As each object will have its own copy of instance variables, changes made to these variables through one object will not reflect in another object. |
| Class variables can be accessed using either class name or object reference. | Instance variables can be accessed only through object reference. |

|  |  |
| --- | --- |
| **Fail-Fast Iterators** | **Fail-Safe Iterators** |
| Fail-Fast iterators doesn’t allow modifications of a collection while iterating over it. | Fail-Safe iterators allow modifications of a collection while iterating over it. |
| These iterators throw ConcurrentModificationException if a collection is modified while iterating over it. | These iterators don’t throw any exceptions if a collection is modified while iterating over it. |
| They use original collection to traverse over the elements of the collection. | They use copy of the original collection to traverse over the elements of the collection. |
| These iterators don’t require extra memory. | These iterators require extra memory to clone the collection. |
| Ex : Iterators returned by *ArrayList*, *Vector*, *HashMap*. | Ex : Iterator returned by *ConcurrentHashMap.* |

|  |  |  |
| --- | --- | --- |
| final | finally | finalize() |
| final is a keyword in Java which is used to make a variable or a method or a class as unchangeable. | finally is a block in Java which is used for exception handling along with try and catch blocks. | finalize() method is a protected method of java.lang.Object class which is used to perform some clean up operations on an object before it is removed from the memory. |
| The value of a variable which is declared as final can’t be changed once it is initialized. | finally block is always executed whether an exception is occurred or not and occurred exception is handled or not. | This method is called by garbage collector thread before an object is removed from the memory. |
| A method declared as final can’t be overridden or modified in the sub class and a class declared as final can’t be extended. | Most of time, this block is used to close the resources like database connection, I/O resources etc soon after their use. | This method is inherited to every class you create in Java. |

|  |  |
| --- | --- |
| **ClassNotFoundException** | **NoClassDefFoundError** |
| It is an exception. It is of type java.lang.Exception. | It is an error. It is of type java.lang.Error. |
| It occurs when an application tries to load a class at run time which is not updated in the classpath. | It occurs when Java runtime system doesn’t find a class definition, which is present at compile time, but missing at run time. |
| It is thrown by the application itself. It is thrown by the methods like Class.forName(), loadClass() and findSystemClass(). | It is thrown by the Java Runtime System. |
| It occurs when classpath is not updated with required JAR files. | It occurs when required class definition is missing at run time. |

|  |  |
| --- | --- |
| **start()** | **run()** |
| New thread is created. | No new thread is created. |
| Newly created thread executes task kept in run() method. | Calling thread itself executes task kept in run() method. |
| It is a member of *java.lang.Thread* class. | It is a member of *java.lang.Runnable* interface. |
| You can’t call start() method more than once. | You can call run() method multiple times. |
| Use of multi-threaded programming concept. | No use of multi-threaded programming concept. |

|  |  |  |
| --- | --- | --- |
| **throw** | **throws** | **Throwable** |
| throw is a keyword in Java which is used to throw an exception manually. | throws is also a keyword in java which is used in the method signature to indicate that this method may throw mentioned exceptions. | Throwable is a super class for all types of errors and exceptions in Java. This class is a member of java.lang package. |
| Using throw keyword, you can throw an exception from any method or block. But, that exception must be of type **java.lang.Throwable** class or it’s sub classes. | The caller to such methods must handle the mentioned exceptions either using try-catch blocks or using throws keyword. | Only instances of this class or it’s sub classes are thrown by the java virtual machine or by the throw statement. |

|  |  |
| --- | --- |
| **User Threads** | **Daemon Threads** |
| JVM waits for user threads to finish their work. It will not exit until all user threads finish their work. | JVM will not wait for daemon threads to finish their work. It will exit as soon as all user threads finish their work. |
| User threads are foreground threads. | Daemon threads are background threads. |
| User threads are high priority threads. | Daemon threads are low priority threads. |
| User threads are created by the application. | Daemon threads, in most of time, are created by the JVM. |
| User threads are mainly designed to do some specific task. | Daemon threads are designed to support the user threads. |
| JVM will not force the user threads to terminate. It will wait for user threads to terminate themselves. | JVM will force the daemon threads to terminate if all user threads have finished their work. |

|  |  |
| --- | --- |
| **notify()** | **notifyAll()** |
| When a thread calls *notify()* method on a particular object, only one thread will be notified which is waiting for the lock or monitor of that object. | When a thread calls *notifyAll()* method on a particular object, all threads which are waiting for the lock of that object are notified. |
| The thread chosen to notify is random i.e randomly one thread will be selected for notification. | All notified threads will get the lock of the object on a priority basis. |
| Notified thread doesn’t get the lock of the object immediately. It gets once the calling thread releases the lock of that object. Until that it will be in BLOCKED state. It will move from BLOCKED state to RUNNING state once it gets the lock. | All notified threads will move from WAITING state to BLOCKED state. The thread which gets the lock of the object moves to RUNNING state. The remaining threads will remain in BLOCKED state until they get the object lock. |

|  |  |
| --- | --- |
| **WAITING** | **BLOCKED** |
| The thread will be in this state when it calls wait() or join() method. The thread will remain in WAITING state until any other thread calls notify() or notifyAll(). | The thread will be in this state when it is notified by other thread but has not got the object lock yet. |
| The WAITING thread is waiting for notification from other threads. | The BLOCKED thread is waiting for other thread to release the lock. |
| The WAITING thread can be interrupted. | The BLOCKED thread can’t be interrupted. |

|  |  |
| --- | --- |
| **Implements Runnable** | **Extends Thread** |
| You can extend any other class. | You can’t extend any other class. |
| No overhead of additional methods . | Overhead of additional methods from Thread class. |
| Separates the task from the runner. | Doesn’t separate the task from the runner. |
| Best object oriented programming practice. | Not a good object oriented programming practice. |
| Loosely coupled. | Tightly coupled. |
| Improves the reusability of the code. | Doesn’t improve the reusability of the code. |
| More generalized task. | Thread specific task. |
| Maintenance  of the code will be easy. | Maintenance of the code will be time consuming. |

|  |  |
| --- | --- |
| **Collection** | **Collections** |
| Collection is a root level interface of the Java Collection Framework. Most of the classes in Java Collection Framework inherit from this interface. | Collections is an utility class in java.util package. It consists of only static methods which are used to operate on objects of type Collection. |
| List, Set and Queue are main sub interfaces of this interface. | Collections.max(), Collections.min(), Collections.sort() are some methods of Collections class. |

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| ArrayList is an index based data structure where each element is associated with an index. | Elements in the LinkedList are called as nodes, where each node consists of three things – Reference to previous element, Actual value of the element and Reference to next element. |
| Insertions and Removals in the middle of the ArrayList are very slow. Because after each insertion and removal, elements need to be shifted. | Insertions and Removals from any position in the LinkedList are faster than the ArrayList. Because there is no need to shift the elements after every insertion and removal. Only references of previous and next elements are to be changed. |
| Insertion and removal operations in ArrayList are of order O(n). | Insertion and removal in LinkedList are of order O(1). |
| Retrieval of elements in the ArrayList is faster than the LinkedList . Because all elements in ArrayList are index based. | Retrieval of elements in LinkedList is very slow compared to ArrayList. Because to retrieve an element, you have to traverse from beginning or end (Whichever is closer to that element) to reach that element. |
| Retrieval operation in ArrayList is of order of O(1). | Retrieval operation in LinkedList is of order of O(n). |
| ArrayList is of type Random Access. i.e elements can be accessed randomly. | LinkedList is not of type Random Access. i.e elements can not be accessed randomly. you have to traverse from beginning or end to reach a particular element. |
| ArrayList can not be used as a Stack or Queue. | LinkedList, once defined, can be used as ArrayList, Stack, Queue, Singly Linked List and Doubly Linked List. |
| ArrayList requires less memory compared to LinkedList. Because ArrayList holds only actual data and it’s index. | LinkedList requires more memory compared to ArrayList. Because, each node in LinkedList holds data and reference to next and previous elements. |
| If your application does more retrieval than the insertions and deletions, then use ArrayList. | If your application does more insertions and deletions than the retrieval, then use LinkedList. |

|  |  |
| --- | --- |
| **HashSet** | **HashMap** |
| HashSet implements Set interface. | HashMap implements Map interface. |
| HashSet stores the data as objects. | HashMap stores the data as key-value pairs. |
| HashSet internally uses HashMap. | HashMap internally uses an array of Entry<K, V> objects. |
| HashSet doesn’t allow duplicate elements. | HashMap doesn’t allow duplicate keys, but allows duplicate values. |
| HashSet allows only one null element. | HashMap allows one null key and multiple null values. |
| Insertion operation requires only one object. | Insertion operation requires two objects, key and value. |
| HashSet is slightly slower than HashMap. | HashMap is slightly faster than HashSet. |

|  |  |
| --- | --- |
| **HashMap** | **HashTable** |
| HashMap is not synchronized and therefore it is not thread safe. | HashTable is internally synchronized and therefore it is thread safe. |
| HashMap allows maximum one null key and any number of null values. | HashTable doesn’t allow null keys and null values. |
| Iterators returned by the HashMap are fail-fast in nature. | Enumeration returned by the HashTable are fail-safe in nature. |
| HashMap extends AbstractMap class. | HashTable extends Dictionary class. |
| HashMap returns only iterators to traverse. | HashTable returns both Iterator as well as Enumeration for traversal. |
| HashMap is fast. | HashTable is slow. |
| HashMap is not a legacy class. | HashTable is a legacy class. |
| HashMap is preferred in single threaded applications. If you want to use HashMap in multi threaded application, wrap it using Collections.synchronizedMap() method. | Although HashTable is there to use in multi threaded applications, now a days it is not at all preferred. Because, ConcurrentHashMap is better option than HashTable. |

|  |  |
| --- | --- |
| **Iterator** | **ListIterator** |
| Using Iterator, you can traverse List, Set and Queue type of objects. | But using ListIterator, you can traverse only List objects. |
| Using Iterator, we can traverse the elements only in forward direction. | But, using ListIterator you can traverse the elements in both the directions – forward and backward. |
| Using Iterator you can only remove the elements from the collection. | But using ListIterator, you can perform modifications (insert, replace, remove) on the list. |
| You can’t iterate a list from the specified index using Iterator. | But using ListIterator, you can iterate a list from the specified index. |
| Methods : hasNext(), next() and remove() | Methods : hasNext(), hasPrevious(), next(), previous(), nextIndex(), previousIndex(), remove(), set(), add() |

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| ArrayList is not thread safe. | Vector is thread safe. |
| As ArrayList is not synchronized, it gives better performance than Vector. | As Vector is synchronized, it is slightly slower than ArrayList. |
| ArrayList is not a legacy code. | Vector class is considered as legacy, due for deprecation. |

|  |  |  |
| --- | --- | --- |
| **HashSet** | **LinkedHashSet** | **TreeSet** |
| HashSet uses HashMap internally to store it’s elements. | LinkedHashSet uses  LinkedHashMap internally to store it’s elements. | TreeSet uses TreeMap internally to store it’s elements. |
| HashSet doesn’t maintain any order of elements. | LinkedHashSet maintains insertion order of elements. i.e elements are placed as they are inserted. | TreeSet orders the elements according to supplied Comparator. If no comparator is supplied, elements will be placed in their natural ascending order. |
| HashSet gives better performance than the LinkedHashSet and TreeSet. | The performance of LinkedHashSet is between HashSet and TreeSet. It’s performance is almost similar to HashSet. But slightly in the slower side as it also maintains LinkedList internally to maintain the insertion order of elements. | TreeSet gives less performance than the HashSet and LinkedHashSet as it has to sort the elements after each insertion and removal operations. |
| HashSet gives performance of order O(1) for insertion, removal and retrieval operations. | LinkedHashSet also gives performance of order O(1) for insertion, removal and retrieval operations. | TreeSet gives performance of order O(log(n)) for insertion, removal and retrieval operations. |
| HashSet uses equals() and hashCode() methods to compare the elements and thus removing the possible duplicate elements. | LinkedHashSet also uses equals() and hashCode() methods to compare the elements. | TreeSet uses compare() or compareTo() methods to compare the elements and thus removing the possible duplicate elements. It doesn’t use equals() and hashCode() methods for comparision of elements. |
| HashSet allows maximum one null element. | LinkedHashSet also allows maximum one null element. | TreeSet doesn’t allow even a single null element. If you try to insert null element into TreeSet, it throws NullPointerException. |
| HashSet requires less memory than LinkedHashSet and TreeSet as it uses only HashMap internally to store its elements. | LinkedHashSet requires more memory than HashSet as it also maintains LinkedList along with HashMap to store its elements. | TreeSet also requires more memory than HashSet as it also maintains Comparator to sort the elements along with the TreeMap. |
| Use HashSet if you don’t want to maintain any order of elements. | Use LinkedHashSet if you want to maintain insertion order of elements. | Use TreeSet if you want to sort the elements according to some Comparator. |

|  |  |
| --- | --- |
| **Map()** | **flatMap()** |
| It processes stream of values. | It processes stream of stream of values. |
| It does only mapping. | It performs mapping as well as flattening. |
| It’s mapper function produces single value for each input value. | It’s mapper function produces multiple values for each input value. |
| It is a One-To-One mapping. | It is a One-To-Many mapping. |
| Data Transformation : From Stream<T> to Stream<R> | Data Transformation : From Stream<Stream<T> to Stream<R> |
| Use this method when the mapper function is producing a single value for each input value. | Use this method when the mapper function is producing multiple values for each input value. |

|  |  |
| --- | --- |
| **Intermediate Operations** | **Terminal Operations** |
| They return stream. | They return non-stream values. |
| They can be chained together to form a pipeline of operations. | They can’t be chained together. |
| Pipeline of operations may contain any number of intermediate operations. | Pipeline of operations can have maximum one terminal operation, that too at the end. |
| Intermediate operations are lazily loaded. | Terminal operations are eagerly loaded. |
| They don’t produce end result. | They produce end result. |
| Examples : filter(), map(), distinct(), sorted(), limit(), skip() | Examples : forEach(), toArray(), reduce(), collect(), min(), max(), count(), anyMatch(), allMatch(), noneMatch(), findFirst(), findAny() |

|  |  |
| --- | --- |
| **Static Binding** | **Dynamic Binding** |
| It is a binding that happens at compile time. | It is a binding that happens at run time. |
| Actual object is not used for binding. | Actual object is used for binding. |
| It is also called early binding because binding happens during compilation. | It is also called late binding because binding happens at run time. |
| Method overloading is the best example of static binding. | Method overriding is the best example of dynamic binding. |
| Private, static and final methods show static binding. Because, they can not be overridden. | Other than private, static and final methods show dynamic binding. Because, they can be overridden. |

|  |  |
| --- | --- |
| **Method Overloading** | **Method Overriding** |
| When a class has more than one method with same name but with different arguments, then we call it as method overloading. | When a super class method is modified in the sub class, then we call this as method overriding. |
| Overloaded methods must have different method signatures.  That means they should differ at least in any one of these three things – Number of arguments, Types of arguments and order of arguments. But, they must have same name. | Overridden methods must have same method signature. I.e. you must not change the method name, types of arguments, number of arguments and order of arguments while overriding a super class method. |
| Overloaded methods can have same or different return types. | The return type of the overridden method must be compatible with that of super class method. That means if super class method has primitive type as its return type, then it must be overridden with same return type. If super class method has derived type as its return type then it must be overridden with same type or its sub class type. |
| Overloaded methods can have same visibility or different visibility. | While overriding a super class method either you can keep the same visibility or you can increase the visibility. But you can’t reduce it. |
| Overloaded methods can be static or not static. It does not affect the method overloading. | You can’t override a static method. |
| Binding between method call and method definition happens at compile time (Static Binding). | Binding between method call and method definition happens at run time (Dynamic Binding). |
| It shows static polymorphism. | It shows dynamic polymorphism. |
| Private methods can be overloaded. | Private methods can’t be overridden. |
| Final methods can be overloaded. | Final methods can’t be overridden. |
| For method overloading, only one class is required. I.e. Method overloading happens within a class. | For method overriding, two classes are required – super class and sub class. That means method overriding happens between two classes. |

|  |  |  |
| --- | --- | --- |
| **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 | 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. |

|  |  |
| --- | --- |
| **Process** | **Thread** |
| Processes are heavy weight operations. | Threads are light weight operations. |
| Every process has its own memory space. | Threads use the memory of the process they belong to. |
| Inter process communication is slow as processes have different memory address. | Inter thread communication is fast as threads of the same process share the same memory address of the process they belong to. |
| Context switching between the process is more expensive. | Context switching between threads of the same process is less expensive. |
| Processes don’t share the memory with other processes. | Threads share the memory with other threads of the same process. |

|  |  |
| --- | --- |
| **Checked Exceptions** | **Unchecked Exceptions** |
| They are known at compile time. | They are known at run time. |
| They are checked at compile time. | They are not checked at compile time. Because they occur only at run time. |
| These are compile time exceptions. | These are run time exceptions. |
| If  these exceptions are not handled properly in the application, they give compile time error. | If these exceptions are not handled properly, they don’t give compile time error. But application will be terminated prematurely at run time. |
| All sub classes of java.lang.Exception Class except sub classes of RunTimeException are checked exceptions. | All sub classes of RunTimeException and sub classes of java.lang.Error are unchecked exceptions. |

|  |  |
| --- | --- |
| **HashMap** | **ConcurrentHashMap** |
| HashMap is not synchronized internally and hence it is not thread safe. | ConcurrentHashMap is internally synchronized and hence it is thread safe. |
| HashMap is the part of Java collection framework since JDK 1.2. | ConcurrentHashMap is introduced in JDK 1.5 as an alternative to HashTable. |
| HashMap allows maximum one null key and any number of null values. | ConcurrentHashMap doesn’t allow even a single null key and null value. |
| Iterators returned by HashMap are fail-fast in nature. | Iterators returned by ConcurrentHashMap are fail-safe in nature. |
| HashMap is faster. | ConcurrentHashMap is slower. |
| Most suitable for single threaded applications. | Most suitable for multi threaded applications. |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Synchronized HashMap** | **HashTable** | **ConcurrentHashMap** |
| Locking Level | Object Level | Object Level | Segment Level |
| Synchronized operations | All operations are synchronized. | All operations are synchronized. | Only update operations are synchronized. |
| How many threads can enter into a map at a time? | Only one thread | Only one thread | By default, 16 threads can perform update operations and any number of threads can perform read operations at a time. |
| Null Keys And Null Values | Allows one null key and any number of null values. | Doesn’t allow null keys and null values. | Doesn’t allow null keys and null values. |
| Nature Of Iterators | Fail-Fast | Fail-Safe | Fail-Safe |
| Introduced In? | JDK 1.2 | JDK 1.1 | JDK 1.5 |
| When To Use? | Use only when high level of data consistency is required in multi threaded environment. | Don’t Use. Not recommended as it is a legacy class. | Use in all multi threaded environment except where high level of data consistency is required. |

**SPRING Questions**

**HTTP Status code cheat sheet**

A screenshot of a computer

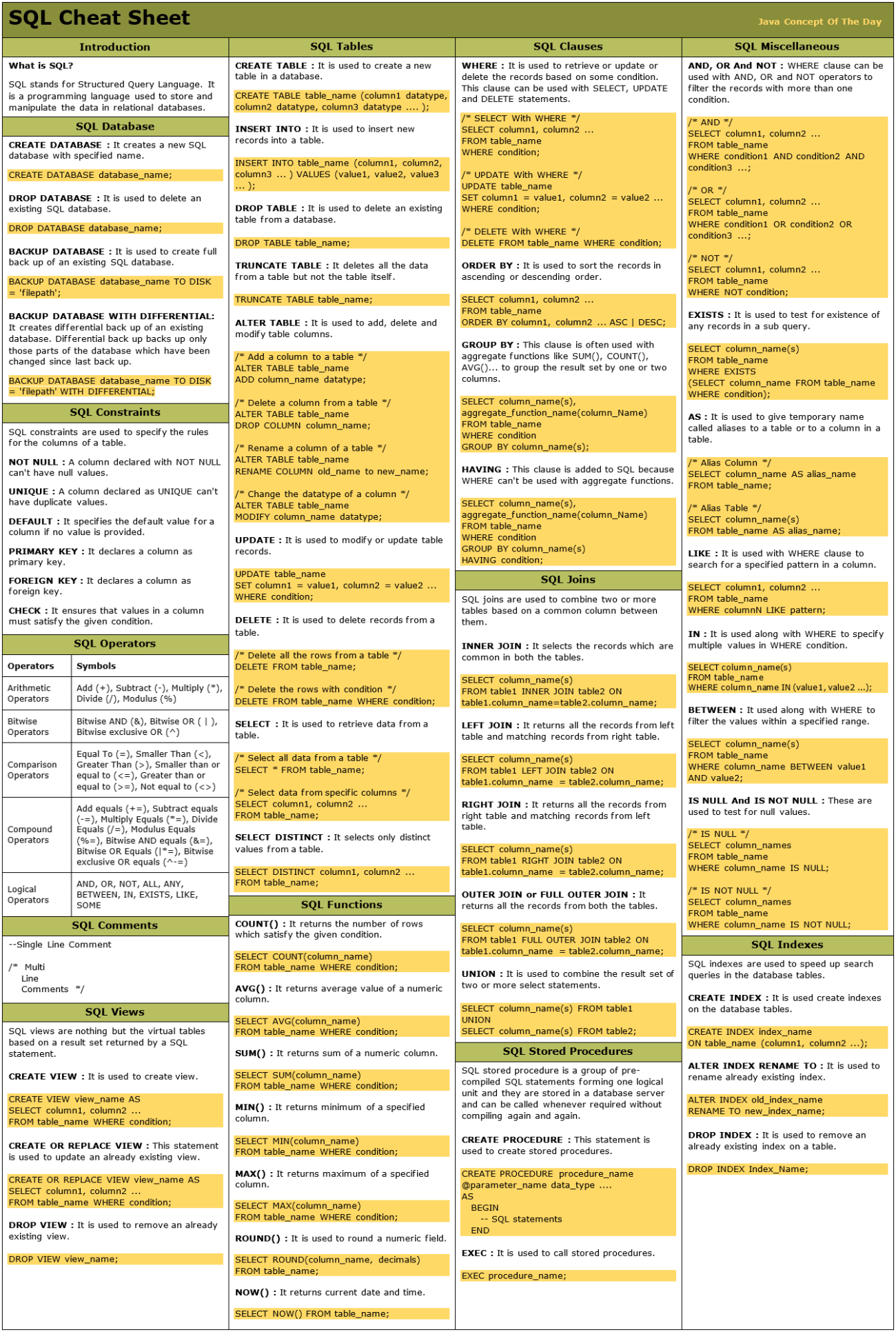
Description automatically generated

**JSP/SERVLET Questions**

**MAVEN Questions**

**Hibernate Questions**

**Database Questions**

**SQL Cheat Sheet**

**PROJECT Questions**