1. SOLID:

* **Single responsibility principle**: Every class in Java should have a single job to do.
* **Open-closed principle**: classes should be open for extension, but closed for modification.
* **Liskov substitution principle**: Simply put, if class A is a subtype of class B, then we should be able to replace B with A without disrupting the behavior of our program.
* **Interface Segregation**: larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.
* **Dependency Inversion**: This way, instead of high-level modules depending on low-level modules, both will depend on abstractions.

STUPID: Keep in mind that these are principles, not laws. However, considering them as laws would be good for those who want to improve themselves.

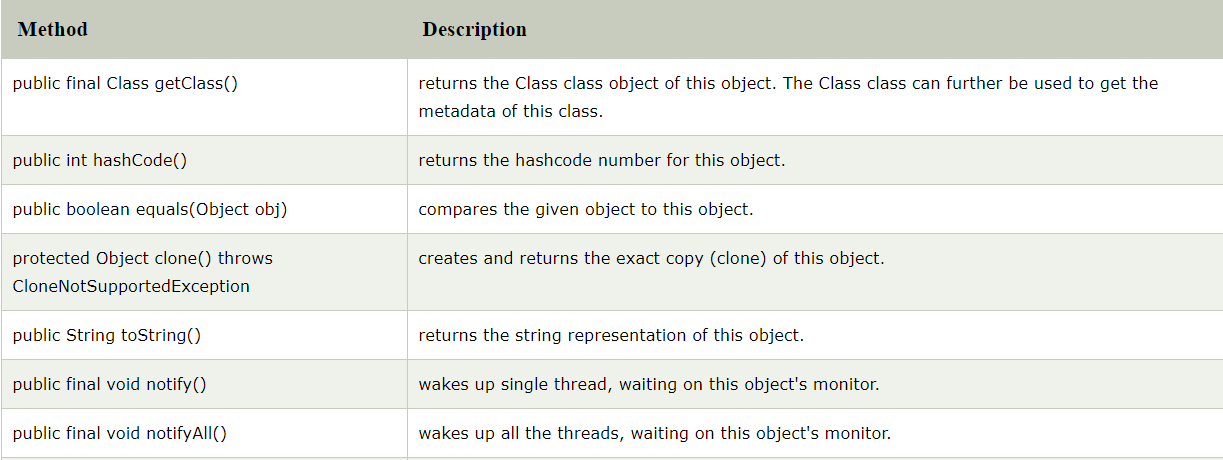
* **Singleton**: Singletons ensure that **a single instance** of a class is **created,** which seems to make sense. For example, if you have to play sounds in your system, you probably have a single SoundManager class. You call SoundManager.playSound(), and it sets the appropriate volume, loads the sound file, and launches the platform’s media player. But the real problem is when this becomes your way of viewing most functionality. It becomes tempting to have a singleton for everything. Then you’re stuck creating those pesky manager classes that handle connections, video, databases, and more.
* **Tight Coupling:** That class **can’t be extracted and used elsewhere.**
* **Untestability:** There are many reasons why a class is **difficult or impossible to test.** Testing a component can also be tricky when it **violates single responsibility** and **does too many things.**
* **Premature Optimization:**
* **Indescriptive Naming**
* **Duplication**
* STUPID approaches lead to difficult-to-maintain and hard-to-test code designs.
* It’s easy to fall into STUPID traps, so stay vigilant and ask yourself the right

1. Class Object:

**The Object class is the parent class of all the classes in java by default.**

The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, known as upcasting.

***Java****provides a****class****with name****Class in java****. lang package. Instances of the****class Class****represent****classes****and interfaces in a running****Java****application. The primitive****Java****types (boolean, byte, char, short, int, long, float, and double), and the keyword void are also represented as****Class****objects.*



1. **Different Types of Inheritance**

**Single inheritance**>(*A class inherits from another class*)   
**Multiple inheritance**>(*Class C inherits from parents A and B*),   
Multi-level inheritance>(*Class C inherits B which inherits A)*  
 Multipath inheritance  
 Hierarchical Inheritance>(*Classes B, C and D inherits A*)  
 Hybrid Inheritance>(*Class D, inherits B and C, where B and C inherits A*)

1. equals()

Main difference between “.equals()” method and “==” operator is that one is method and other is operator.

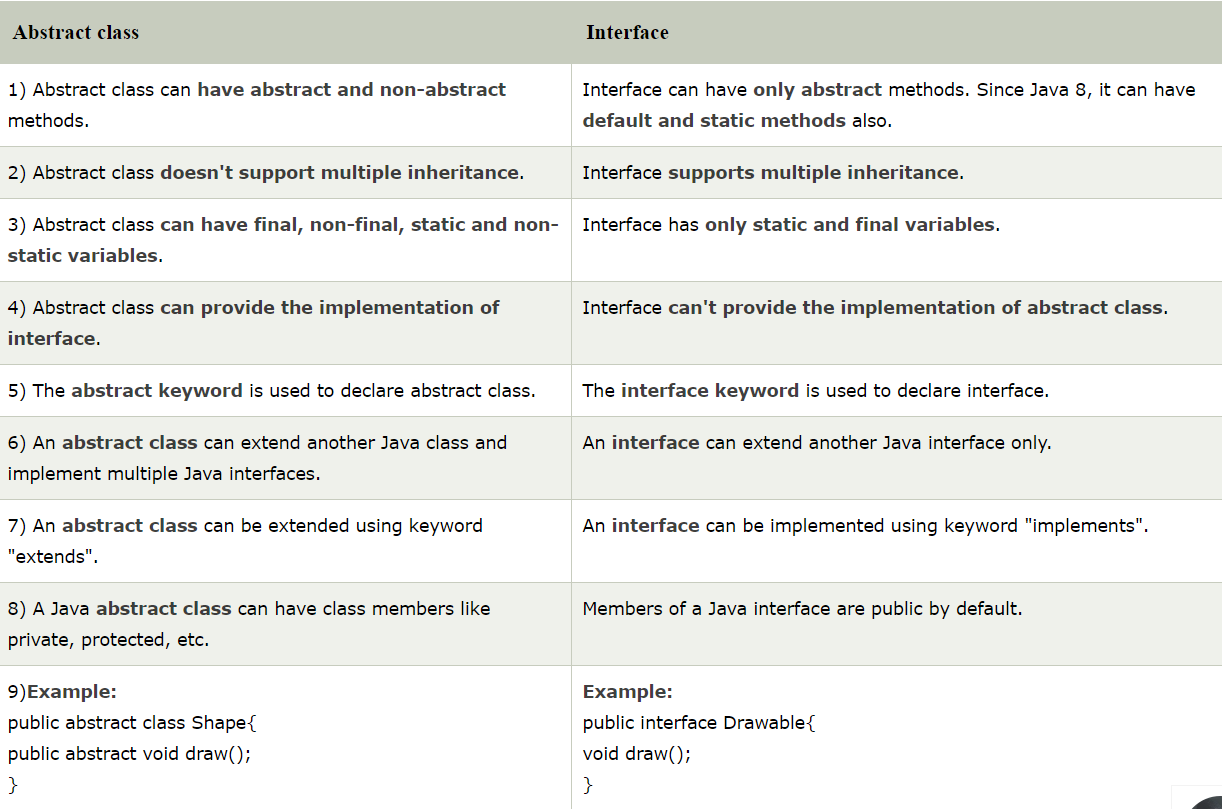
We can use == operators for reference comparison (**address comparison**) and .equals() method for **content comparison**.

1. toString() -The **method** is used to get a String object representing the value of the Number Object.  
   *The String class represents character strings. The class String includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase.*
2. hashCode() - **returns** an integer value, generated by a hashing algorithm.  
   A **hash code** is an integer value that is associated with each object in **Java**. Its main purpose is to facilitate hashing in hash tables, which are used by data structures like HashMap.  
     
   **The purpose of the hashCode() method is to provide a numeric representation of an object's contents so as to provide an alternate mechanism to loosely identify it. *By default the hashCode() returns an integer that represents the internal memory address of the object.***

The hashcode method in the program: This method is useful for implementing Object.hashCode() on objects containing multiple fields.

1. Abstract An **abstract class** is a **class** that is declared **abstract** —it may or may not include **abstract** methods. **Abstract classes** cannot be instantiated, but they can be subclassed. An **abstract method** is a **method** that is declared without an implementation
2. The interface in Java is a mechanism to achieve [*abstraction*](https://www.javatpoint.com/abstract-class-in-java). There can be only abstract methods in the Java interface, not method body.

Default methods enable you to add new functionality to existing interfaces and ensure binary compatibility with code written for older versions of those interfaces. In particular, default methods enable you to add methods that accept lambda expressions as **parameters** to existing interfaces.



1. Enum

An *enum type* is a special data type that enables for a variable to be a set of predefined constants.

1. static vs final

final: final methods cannot be overridden by subclasses  
 A **final field** cannot have its value changed

static: fields belongs to the class, rather than an object  
static methods are used to create **methods** that will exist independently of any instances created for the class. **Static methods** do not use any instance variables of any object of the class they are defined in.

abstract: Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example **abstract void run();**. The body is provided by the subclass (inherited from).

1. **Covariant return** type refers to **return** type of an overriding method. It allows to narrow down **return** type of an overridden method without any need to cast the type or check the **return** type. **Covariant return** type works only for non-primitive **return** types.
2. Lambda expression

A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method

WHY USE LAMBDA EXPRESSIONS  
1. To provide the implementation of Functional interface.

1. Less coding.

Java lambda expression is consisted of three components.

**1) Argument-list:** It can be empty or non-empty as well.

**2) Arrow-token:** It is used to link arguments-list and body of expression.

**3) Body:** It contains expressions and statements for lambda expression.

1. Method reference - Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression.