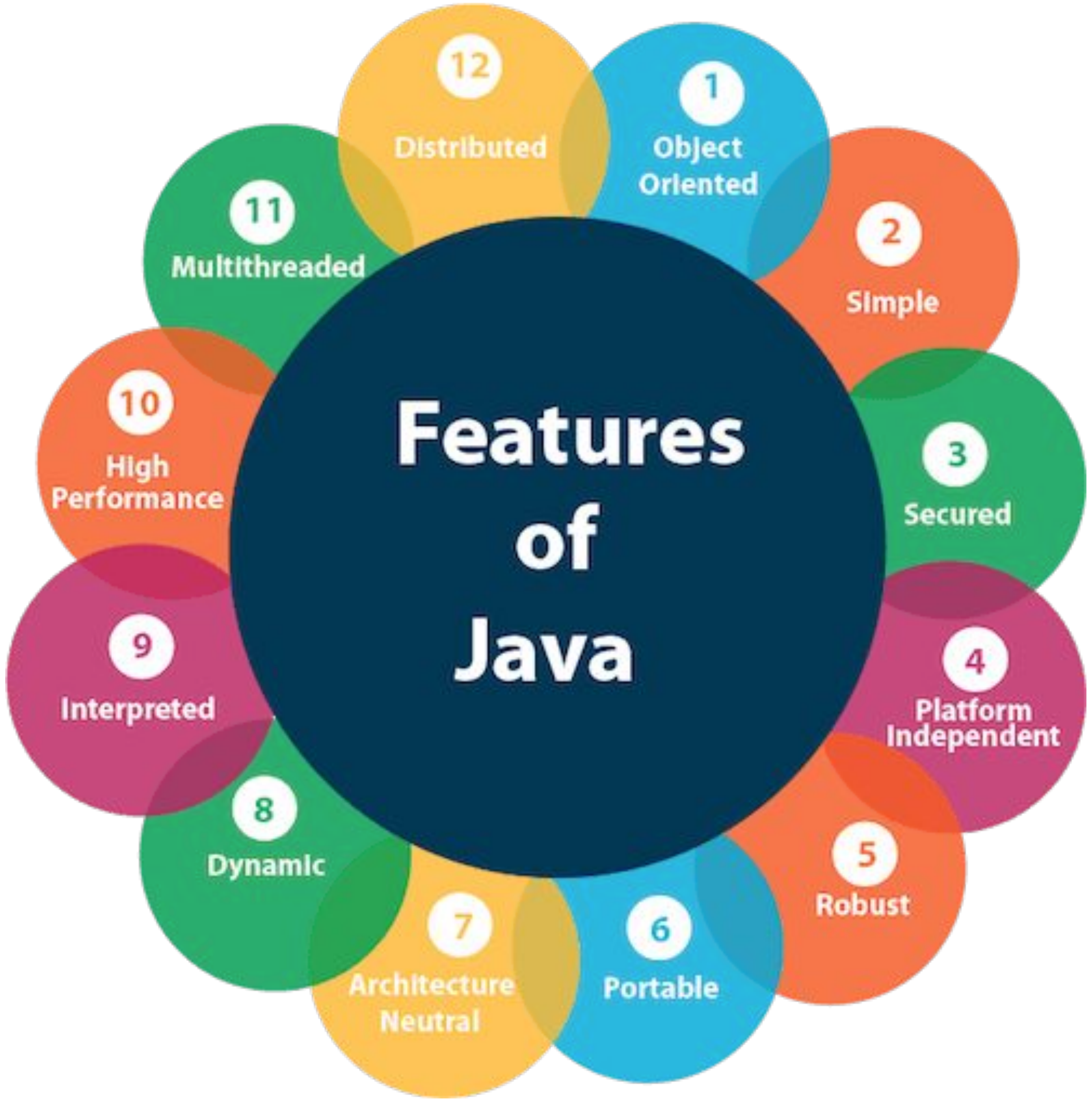
History of Java

- James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called Green Team.
- They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell, and fun to say. According to James Gosling, "Java was one of the top choices along with **Silk**". Since Java was so unique, most of the team members preferred Java than other names.
- Java is an island in Indonesia where the first coffee was produced (called Java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having a cup of coffee nearby his office.
- JDK 1.0 was released on January 23, 1996. After the first release of Java, there have been many additional features added to the language. Now Java is being used in Windows applications, Web applications, enterprise applications, mobile applications, cards, etc. Each new version adds new features in Java.

Java

- Java is a high level, robust, object-oriented and secure programming language.
- Java was developed by *Sun Microsystems* (which is now the subsidiary of Oracle) in the year 1995. *James Gosling* is known as the father of Java
- Since Java has a runtime environment (JRE) and API, it is called a platform.
- Java is used in all kinds of applications like Mobile Applications (Android is Java-based), desktop applications, web applications, client-server applications, enterprise applications, Gaming applications, Cloud based applications ,and many more in all domain.
- Eg- Spotify, twitter, Opera Mini Browser, Acrobat Reader, Netflix, Uber, MatLab, Simcards, Fintech Domain(Barclays, Hsbc, Citi group, goldman Sachs), Eclipse, Hadoop, Google Docs,etc


Features of Java
1
Object Oriented
2
Simple
3
Secured
4
Platform Independent
5
Robust
6
Portable
7
Architecture Neutral
8
Dynamic
9
Interpreted
10
High Performance
11
Multithreaded
12
Distributed

Features of Java

- **Simple**: Java is very easy to learn, and its syntax is simple, clean and easy to understand. It does not have complex features like pointers, operator overloading, multiple inheritances, and Explicit memory allocation. There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.
- **Object-oriented**: Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporate both data and behavior. Main concepts include Abstraction, Encapsulation, Inheritance & Polymorphism.
- **Platform Independent and Portable**: Compiler converts source code to bytecode and then the JVM executes the bytecode generated by the compiler. This bytecode can run on any platform be it Windows, Linux, or macOS which means if we compile a program on Windows, then we can run it on Linux and vice versa. Each operating system has a different JVM, but the output produced by all the OS is the same after the execution of bytecode. it is a software-based platform that runs on top of other hardware-based platforms which has two components (Runtime Environment & API(Application Programming Interface))

Features of Java

- **Secured**: In java, we don’t have pointers, so we cannot access out-of-bound arrays. That’s why several security flaws like stack corruption or buffer overflow are impossible to exploit in Java. Also, java programs run in an environment(JVM) that is independent of the os(operating system) environment which makes java programs more secure. Classloader adds security by separating the package for the classes of the local file system from those that are imported from network sources. Bytecode Verifier checks the code fragments for illegal code that can violate access rights to objects. Security Manager determines what resources a class can access such as reading and writing to the local disk.
- **Robust**: Java language is robust which means reliable. It is developed in such a way that it puts a lot of effort into checking errors as early as possible, that is why the java compiler can detect even those errors that are not easy to detect by another programming language. The main features of java that make it robust are garbage collection, Exception Handling, and memory allocation.
- **Architecture-neutral**: Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

Features of Java

- **High-performance**: Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. The Java architecture is designed in such a way that it reduces the overhead when we run an application, and at some times, it uses the JIT(Java In Time) compiler where the compiler easily compiles the code on-demand basics where it only compiles the methods that we call which makes the applications execute faster.
- **Distributed**: We can create distributed applications using the java programming language. Remote Method Invocation and Enterprise Java Beans are used for creating distributed applications in java. The java programs can be easily distributed on one or more systems that are connected to each other through an internet connection. This feature of Java makes us able to access files by calling the methods from any machine on the internet.
- **Multi-threaded**: We can write Java programs that deal with many tasks at once by defining multiple threads for maximum utilization of the CPU. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

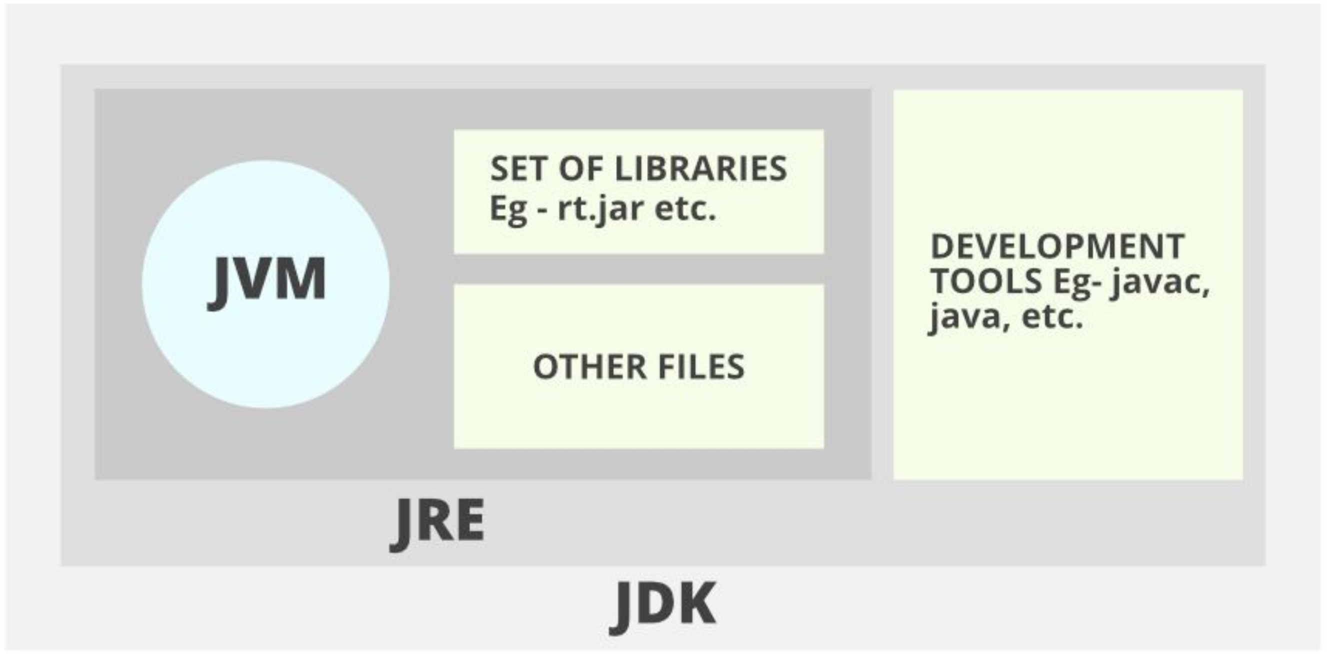
Features of Java

- **Dynamic flexibility**: Java being completely object-oriented gives us the flexibility to add classes, new methods to existing classes and even create new classes through sub-classes. Java even supports functions written in other languages such as C, C++ which are referred to as native methods.
- **Sandbox Execution**: Java programs run in a separate space that allows user to execute their applications without affecting the underlying system with help of a bytecode verifier.
- **Power of compilation and interpretation**: Most languages are designed with purpose either they are compiled language or they are interpreted language. But java integrates arising enormous power as Java compiler compiles the source code to bytecode and JVM executes this bytecode to machine OS-dependent executable code.

JRE, JDK

JDK: (Java Development Kit) is a Kit that provides the environment to develop and execute(run) the Java program. JDK is a kit(or package) that includes two things: 1)Development Tools(to provide an environment to develop your java programs) & 2) JRE (to execute your java program).

JRE: (Java Runtime Environment) is an installation package that provides an environment to only run(not develop) the java program(or application)onto your machine. JRE is only used by those who only want to run Java programs that are end-users of your system.


JVM
SET OF LIBRARIES Eg - rt.jar etc.
OTHER FILES
DEVELOPMENT TOOLS Eg- javac, java, etc.
JRE
JDK

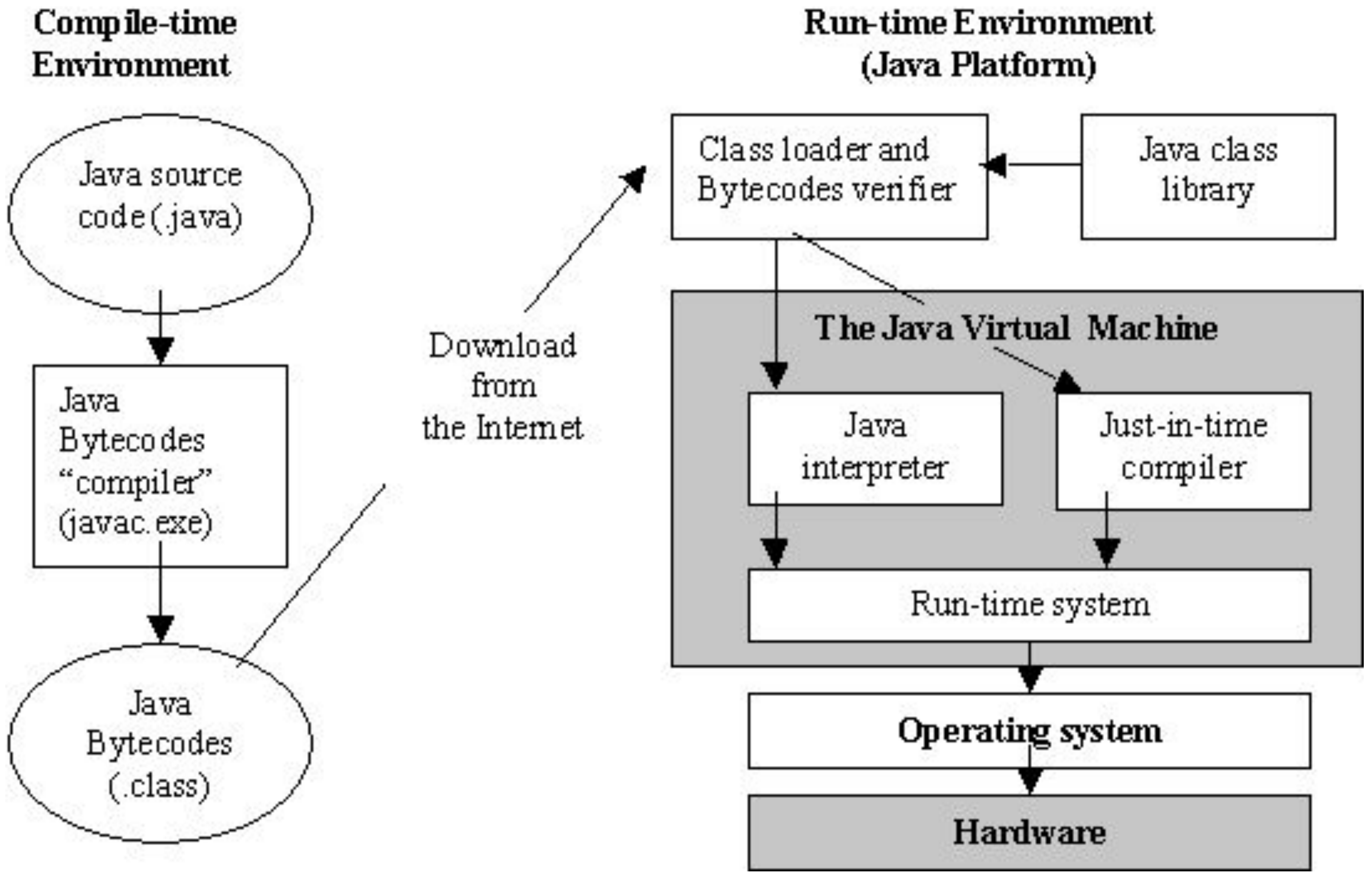
JVM

- **JVM**: (Java Virtual Machine) is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java program line by line, hence it is also known as an interpreter.
- JVM becomes an instance of JRE at the runtime of a Java program. It is widely known as a runtime interpreter.JVM largely helps in the abstraction of inner implementation from the programmers who make use of libraries for their programs from JDK.

- It is mainly responsible for three activities.
- Loading, Linking & Initialization.

- JVM(Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the main method present in a java code. JVM is a part of JRE(Java Runtime Environment).
- Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java-enabled system without any adjustments. This is all possible because of JVM.
- When we compile a .java file, .class files(contains byte-code) with the same class names present in .java file are generated by the Java compiler. This .class file goes into various steps when we run it. These steps together describe the whole JVM.

Running Java Program



- **Eclipse IDE for Enterprise Java and Web Developers**:

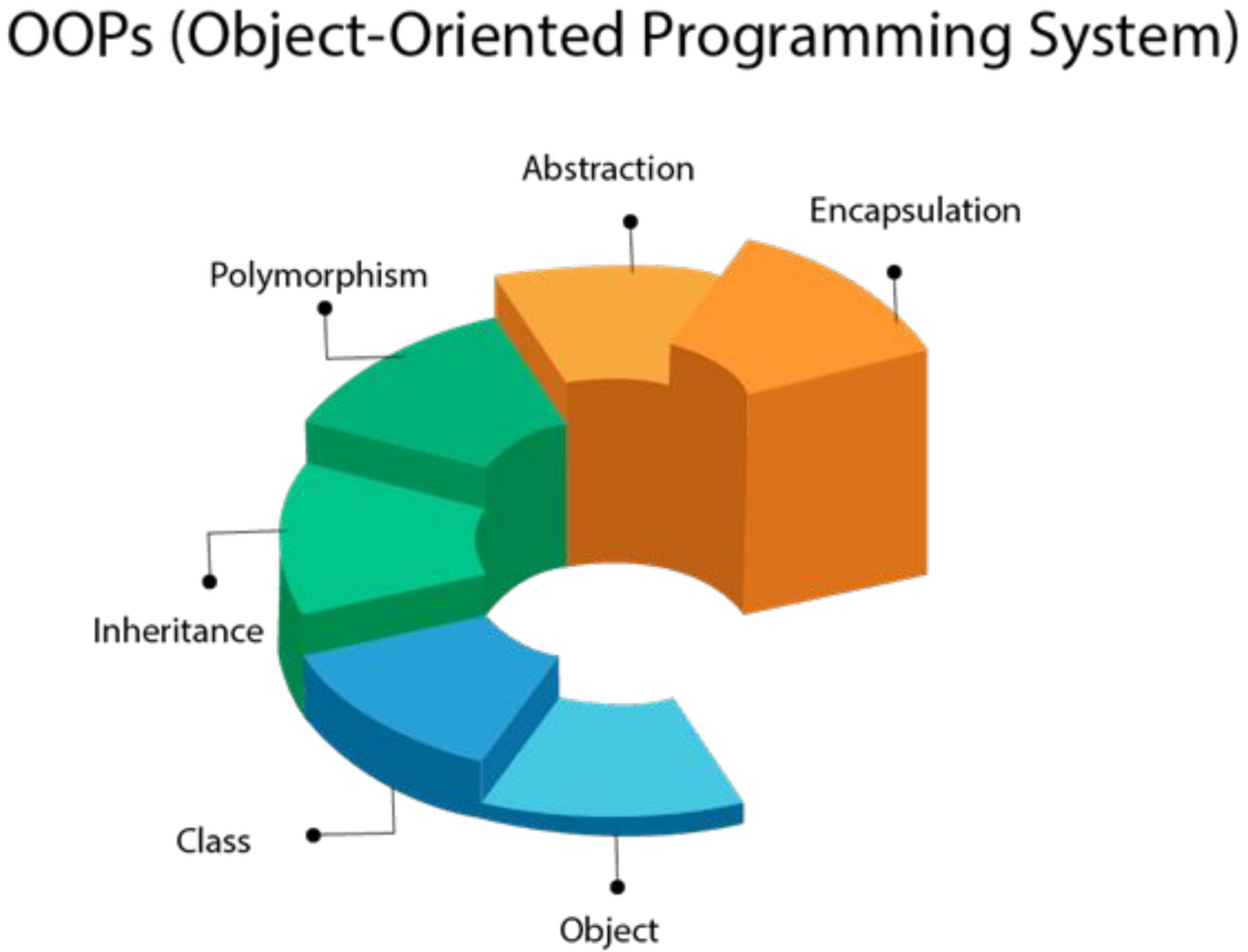
  https://mirror.kakao.com/eclipse/oomph/epp/2022-03/R/eclipse-inst-jre-win64.exe

- **Download OpenJDK11**:

  https://download.visualstudio.microsoft.com/download/pr/8309c7e7-a5f7-4f39-9684-7a1dd4994dc6/26
  81211073d4697580fbafaa04c0c353/microsoft-jdk-11.0.15-windows-x64.zip

OOPS:

- Aims to implement real-world entities like inheritance, hiding, polymorphism etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

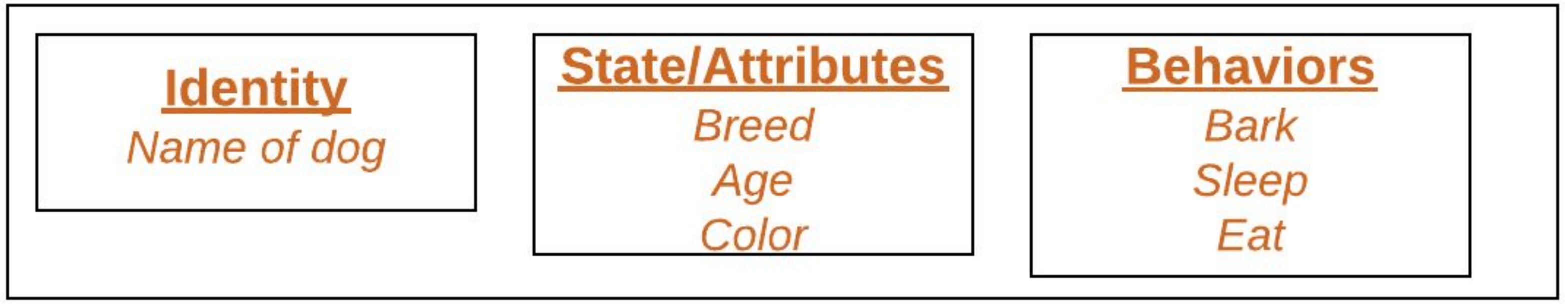


Object

- Physical as well as a logical entity, that has state, behavior and Identity.
- Eg- Mobile, Chair, Laptop, Mouse, Dog, Customer, Account, etc

1. **State**: It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behavior**: It is represented by methods of an object. It also reflects the response of an object with other objects, functionalities.
3. **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

Eg : Dog



Ways to create an object of a class

- **Using new keyword**: Test t = new Test();
- **Using Class.forName(String className) method**: There is a pre-defined class in java.lang package with name Class. The forName(String className) method returns the Class object associated with the class with the given string name. We have to give a fully qualified name for a class. On calling new Instance() method on this Class object returns new instance of the class with the given string name. Eg - Test obj = (Test)Class.forName("com.p1.Test").newInstance();
- **Using clone() method**: clone() method is present in Object class. It creates and returns a copy of the object. Eg - Test t1 = new Test();

  Test t2 = (Test)t1.clone();
- **Deserialization**: De-serialization is a technique of reading an object from the saved state in a file

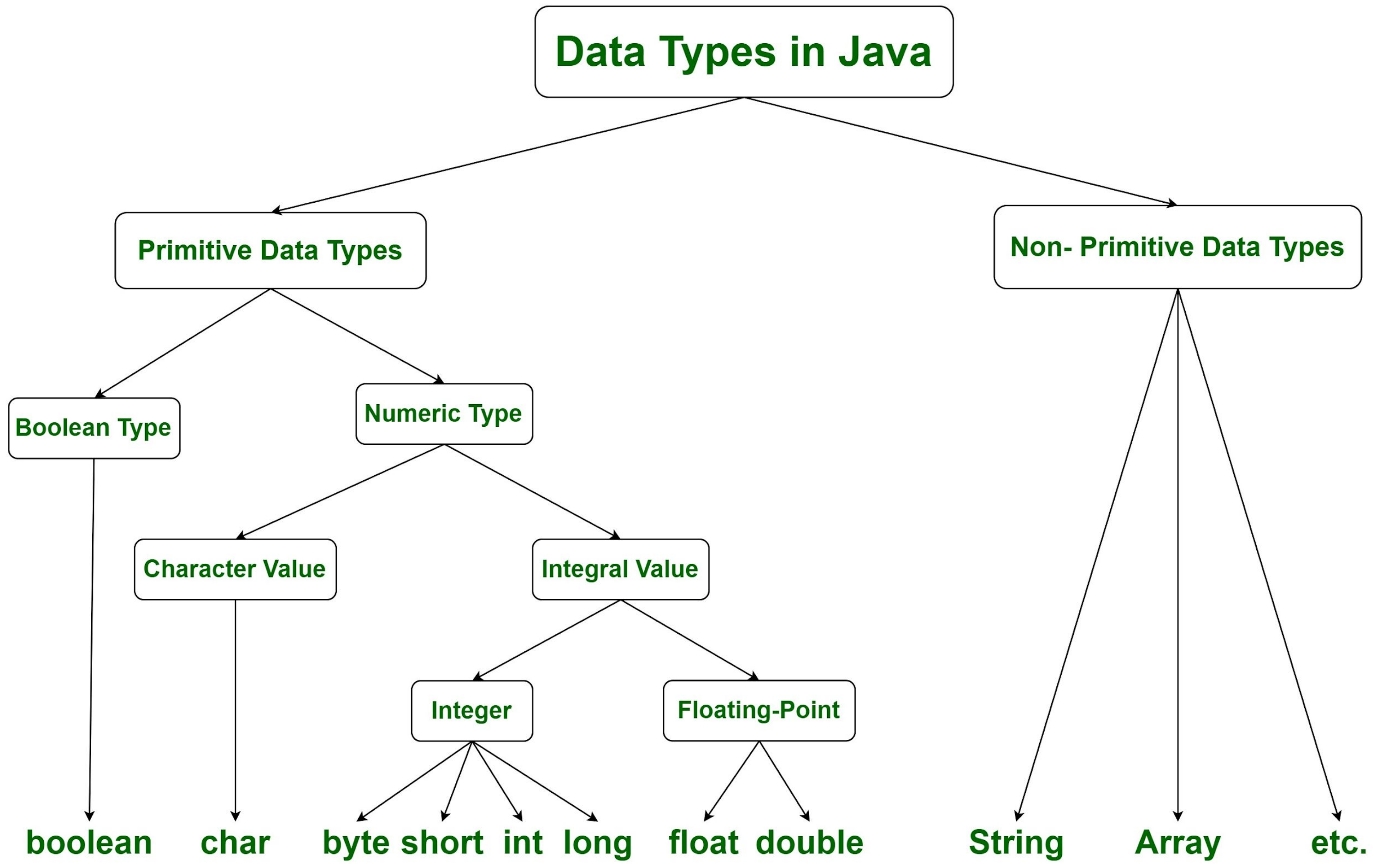
```
FileInputStream file = new FileInputStream(filename);
ObjectInputStream in = new ObjectInputStream(file);
Object obj = in.readObject();
```

Class

- Group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.
- Class Contains Fields, Methods, Constructors, Blocks, Nested class and interface
- **Modifiers**: Private < Default < Protected < Public
- **Constructors**: special method that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes. Same Name, No return type, Called Only Once.

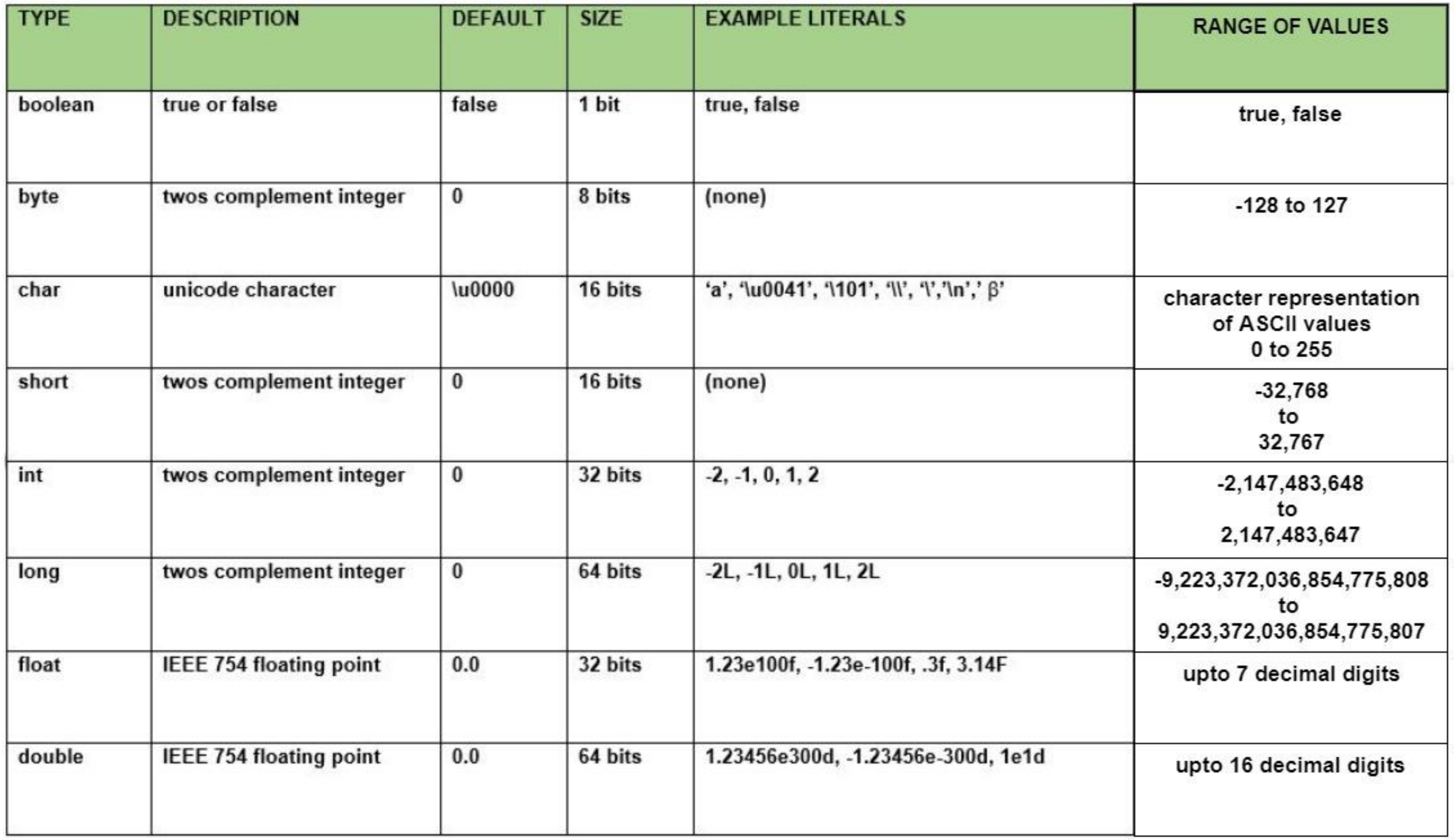
Data Types

Data types specify the different sizes and values that can be stored in the variable.



Primitive Data Types

Primitive data are only single values and have no special capabilities. There are 8 primitive data types.



TYPE	DESCRIPTION	DEFAULT	SIZE	EXAMPLE LITERALS	RANGE OF VALUES
boolean	true or false	false	1 bit	true, false	true, false
byte	twos complement integer	0	8 bits	(none)	-128 to 127
char	unicode character	\u0000	16 bits	‘a’, ‘\u0041’, ‘\101’, ‘\\’, ‘\’,’\n’,’ β’	character representation of ASCII values 0 to 255
short	twos complement integer	0	16 bits	(none)	-32,768 to 32,767
int	twos complement integer	0	32 bits	-2, -1, 0, 1, 2	-2,147,483,648 to 2,147,483,647
long	twos complement integer	0	64 bits	-2L, -1L, 0L, 1L, 2L	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	IEEE 754 floating point	0.0	32 bits	1.23e100f, -1.23e-100f, .3f, 3.14F	upto 7 decimal digits
double	IEEE 754 floating point	0.0	64 bits	1.23456e300d, -1.23456e-300d, 1e1d	upto 16 decimal digits

Non-Primitive/Reference Data Types

The Reference Data Types will contain a memory address of variable values because the reference types won't store the variable value directly in memory. They are strings, objects, arrays, etc.

Feature	Primitive Data Types	Non-Primitive Data Types
Definition	Basic built-in types like `int`, `char`, etc.	Reference types like `String`, `Array`, `Class`
Memory Location	Stored directly in **stack memory**	Reference in stack, actual object in **heap**
Size	Fixed size (e.g., `int` = 4 bytes)	Variable size depending on fields and metadata
Overhead	No overhead	Includes object metadata and method tables
Garbage Collection	Not applicable	Managed by Java's **Garbage Collector**
Nullability	Cannot be `null`	Can be `null`
Object Nature	Not objects	Are objects with methods and properties
Performance	Faster due to direct access	Slower due to reference and heap access
Pass-by Behavior	Passed by value (actual data)	Passed by value (reference to object)
Caching / Pooling	Small values cached (e.g., `Integer` -128 to 127)	Strings may be **interned** for memory efficiency
Method Availability	No methods (raw data only)	Have methods (e.g., `length()`, `toString()`)
Use in Collections	Must use wrapper classes (e.g., `Integer`)	Can be used directly

Variables

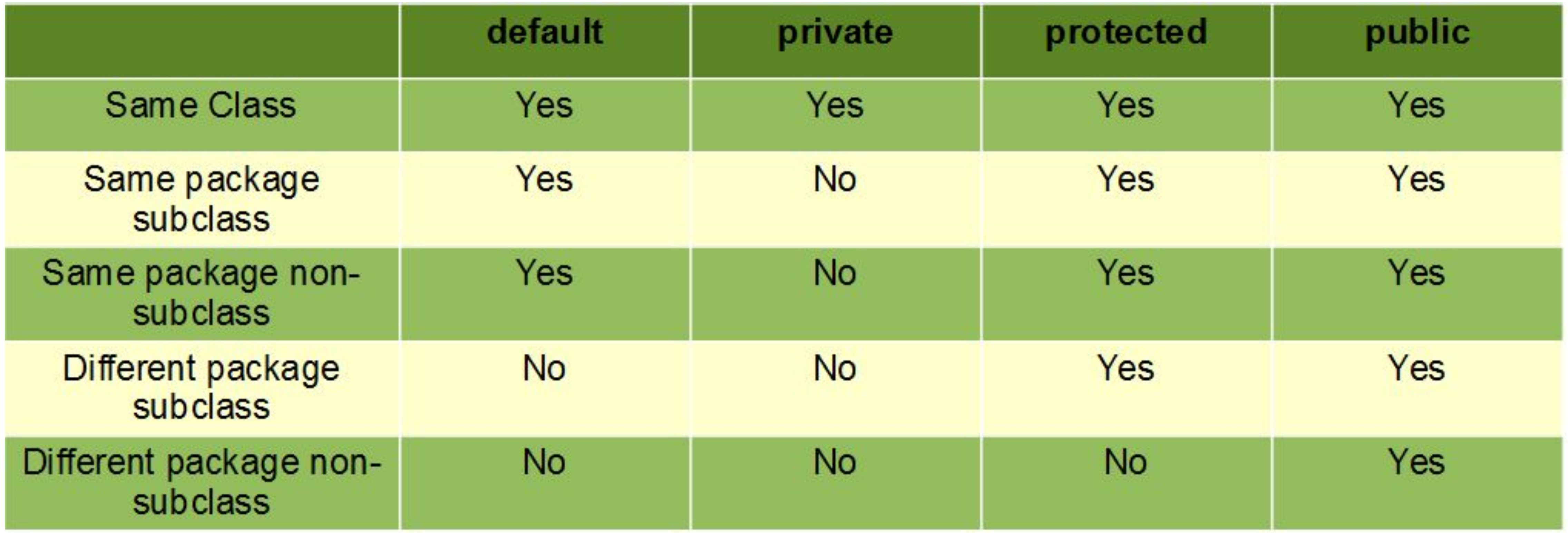
- Variable in Java is a data container that saves the data values during Java program execution. Every variable is assigned a data type that designates the type and quantity of value it can hold. A variable is a memory location name for the data.
- The value stored in a variable can be changed during program execution.
- In Java, all variables must be declared before use.
- **Local Variables:** Defined within a block or method or constructor. The scope of these variables exists only within the block in which the variables are declared.
- **Instance Variables:** Non-static variables and are declared in a class outside of any method, constructor, or block. They are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed. Initialization is not mandatory(default - 0). Can be accessed only by creating Objects.
- **Static Variables:** Declared using the static keyword within a class outside of any method, constructor or block. Static variables are created at the start of program execution and destroyed automatically when execution ends. we can only have one copy of a static variable per class. Can be accessed without Object Creation.

Operators

- Symbol that is used to perform operations, be it logical, arithmetic, relational, etc.
- **Arithmetic Operators:** +, - , *, /, %
- **Unary Operators:** -, !, ++, --, ~
- **Assignment Operators:** =, +=, -=, *=, /=, %=
- **Relational Operators:** ==, !=, >, <, >=, <=
- **Logical Operators:** &&, ||, & (Function similar to AND & OR gate in digital electronics)
- **Ternary Operator:** variable = Expression ? num1: num2
- **Bitwise Operators:** |, &, ^, ~,
- **Shift Operators:** <<, >>, >>>

Access Modifiers

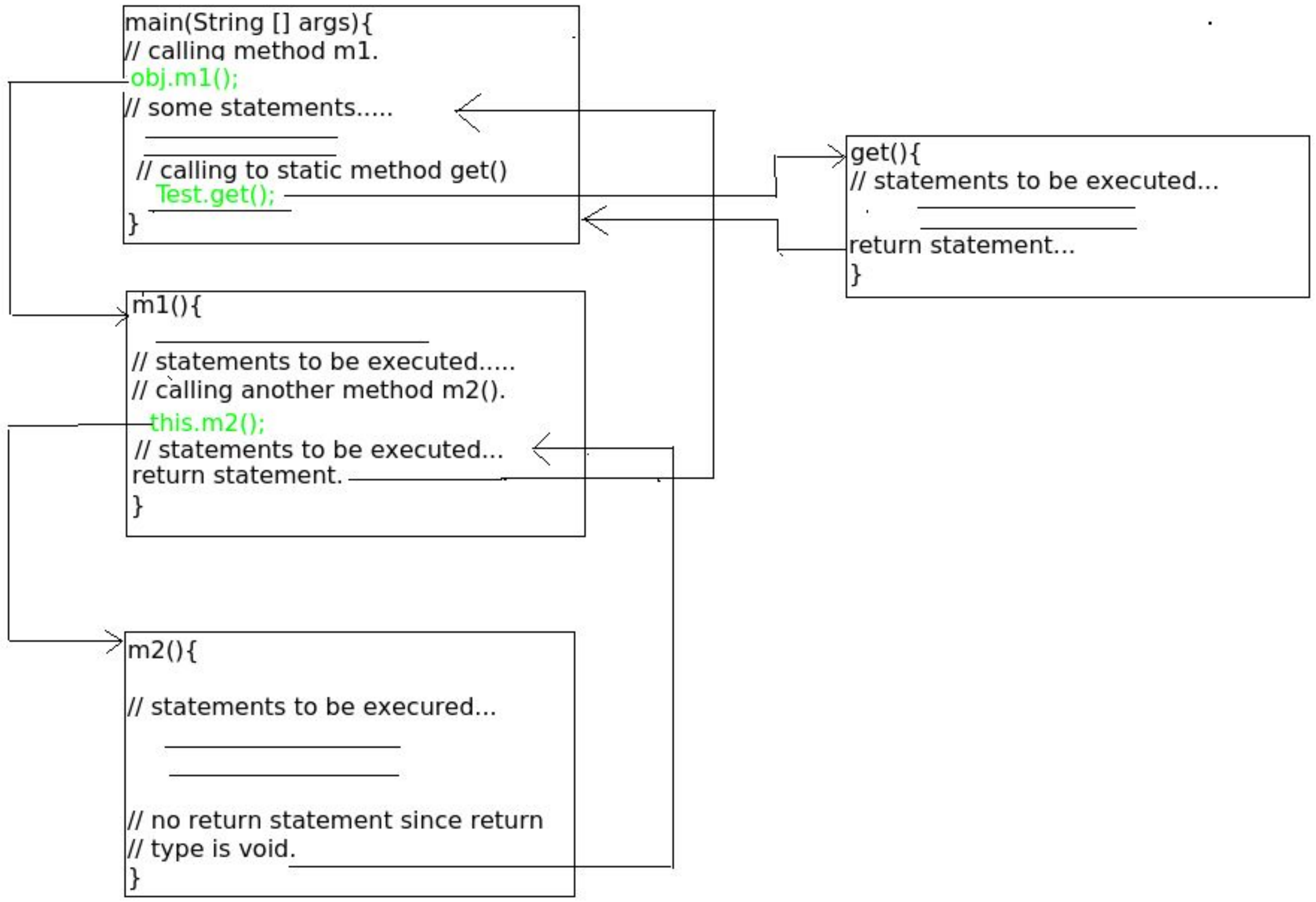
- Access modifiers in Java helps to restrict the scope of a class, constructor, variable, method, or data member. There are four types of access modifiers available in java:
- Private < Default < Protected < Public
- Protected variable is accessible through child class object and not through parent class object.



	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Methods

- Block of code grouped together to perform a certain task or operation. Achieve Reusability. Write Once, Use Many Times. Easy modification and readability. Eg- main().

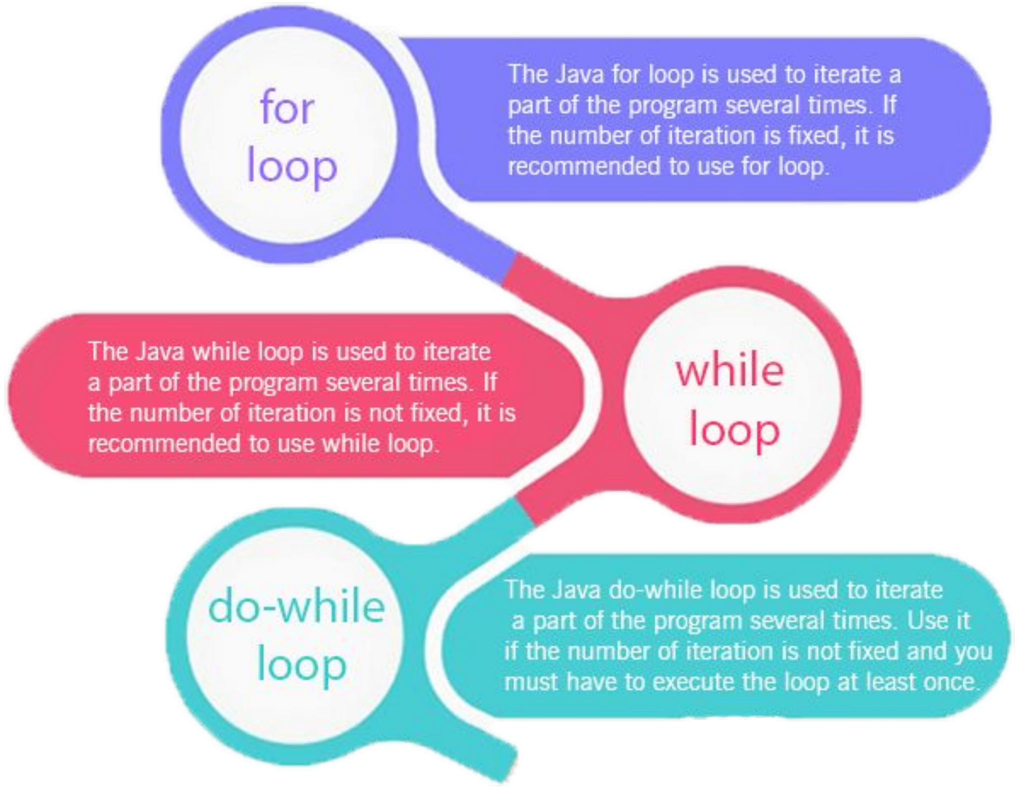

main(String [] args){
// calling method m1.
obj.m1();
// some statements.....
// calling to static method get()
Test.get();
}
m1(){
// statements to be executed.....
// calling another method m2().
this.m2();
// statements to be executed...
return statement.
}
m2(){
// statements to be execured...
// no return statement since return
// type is void.
}
get(){
// statements to be executed...
return statement...
}

Constructor

- Special Type of method, Called when instance of the class is created and memory of object is allocated. Java Compiler provides default constructor, if there is no constructor available. Used to assign values to the class variables at the time of object creation, either explicitly done by the programmer or by Java itself (default constructor).
- Name Same as class. No Return Type. Can't be abstract, Static, Final and Synchronized.
- Java provides a Constructor class which can be used to get the internal information of a constructor in the class. It is found in the java.lang.reflect package.
- **Constructor chaining** is the process of calling one constructor from another constructor with respect to current object. Within same class: It can be done using this() keyword . From base class: by using super() keyword.
- **Destructor**: There is no concept of destructor in Java. In place of the destructor, Java provides the garbage collector that works the same as the destructor. It automatically deletes the unused objects (objects that are no longer used) and free-up the memory. The programmer has no need to manage memory manually. It can be error-prone, vulnerable, and may lead to a memory leak.

Decision Making and Loops

- we want a certain block of code to be executed when some condition is fulfilled.
- **If**: if a certain condition is true then a block of statement is executed otherwise not. We can have if-else, if-else-if, nested if.
- **switch-case**: The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.
- **Jump Statements**: break, continue and return.
- **Loops:** For, While, Do While.

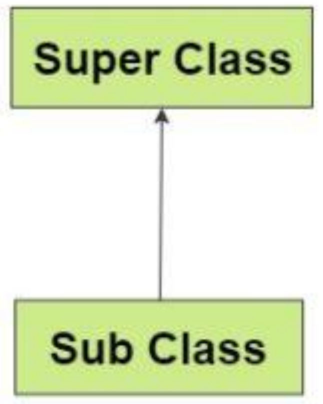

for loop
The Java for loop is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop.
while loop
The Java while loop is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.
do-while loop
The Java do-while loop is used to iterate a part of the program several times. Use it if the number of iteration is not fixed and you must have to execute the loop at least once.

Fenil Shah

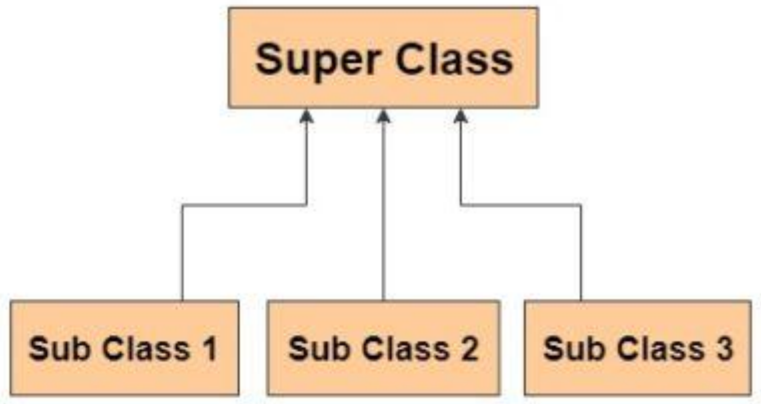
Inheritance

- Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
- Inheritance represents the IS-A relationship which is also known as a parent-child relationship. Eg- relationship between father and son.
- Code Reusability, Method Overriding, Abstraction(we do not have to provide all details is achieved through inheritance)
- Any Class can have only one Super class. Constructors and Private Members are not inherited.
- **Superclass**: The class whose features are inherited is known as superclass (also known as base or parent class)
- **Subclass**: The class that inherits the other class is known as subclass (also known as derived or extended or child class).

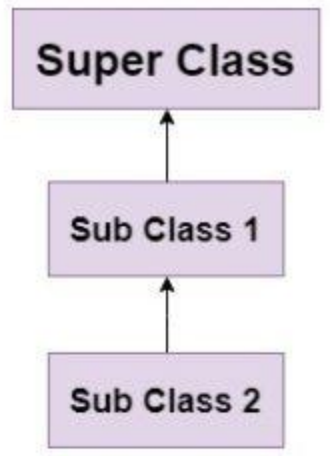
Single Inheritance



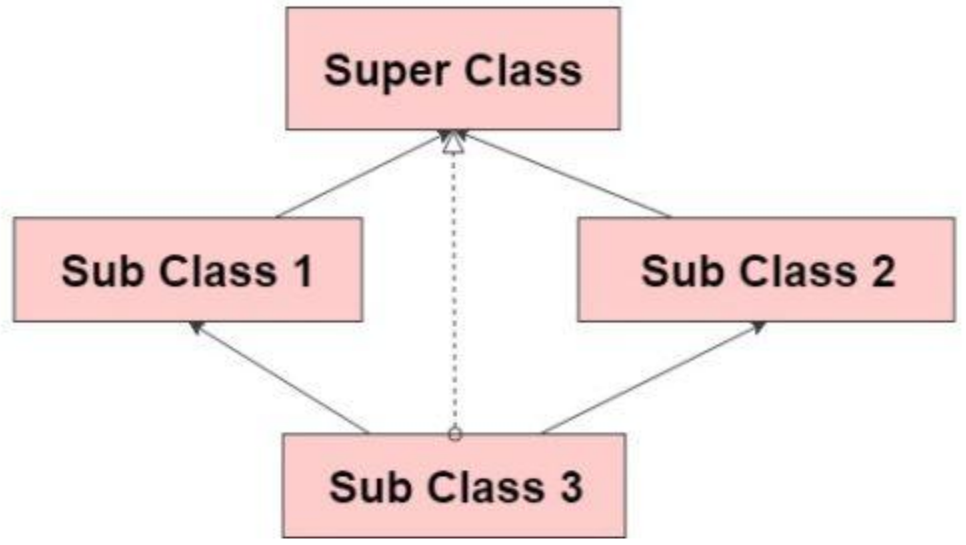
Hierarchial Inheritance



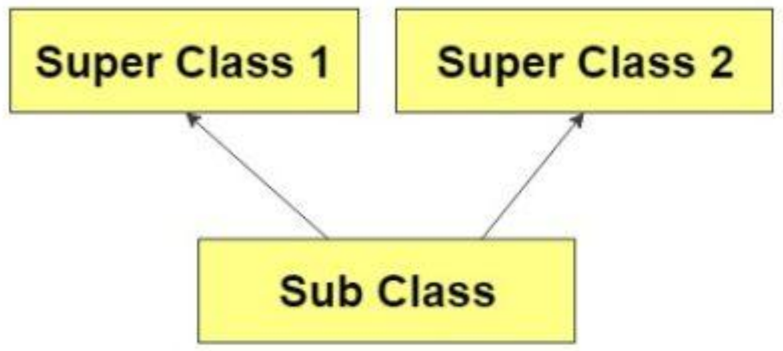
MultiLevel Inheritance



Hybrid Inheritance



Multiple Inhertance



Encapsulation

- Process of wrapping code and data together into a single unit.
- We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it. Eg- Java Bean.
- It helps in getting complete control over data to make it read-only or write-only, helps in achieving data hiding, better for unit Testing.
- Encapsulation prevents access to data members and data methods by any external classes. The encapsulation process improves the security of the encapsulated data.
- Changes made to one part of the code can be successfully implemented without affecting any other part of the code.

Abstraction

- Process of hiding the implementation details and showing only functionality to the user.
- Declared with an abstract keyword, can have abstract(atleast one) and non-abstract methods, cannot be instantiated, can have constructors, static methods & final methods. Bank Example.
- Abstract Method is a method that has just the method definition but does not contain implementation(method body). A method-defined abstract must always be redefined in the subclass, thus making overriding compulsory or making the subclass itself abstract.
- Abstract methods are mostly declared where two or more subclasses are also doing the same thing in different ways through different implementations. It also extends the same Abstract class and offers different implementations of the abstract methods.
- Reduces the complexity of viewing things. Avoids code duplication and increases reusability. Helps to increase the security of an application or program as only essential details are provided to the user.
- It improves the maintainability and modularity of the application. Enhancement will become very easy.
- Eg- Making Phone Call, Sending SMS, brake(), etc.

Abstract Class

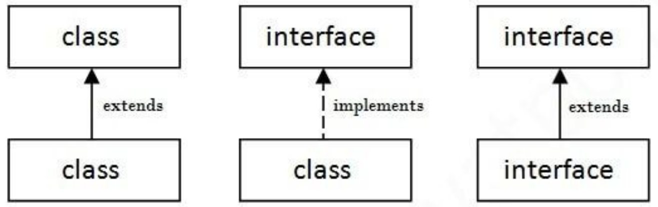
- **Static Methods:** An abstract class can provide common utility functions that are relevant to all its subclasses, even if the abstract class cannot be instantiated. These static methods can be called directly using the class name.
  Eg: An AbstractPaymentGateway class could have a static method isValidCreditCardNumber(String cardNumber) to perform a Luhn algorithm check, which is a common validation needed across all payment gateway implementations.
- **Instance Variables:** Abstract classes can define common state or attributes that all their concrete subclasses will share. This promotes code reuse and ensures consistency in data representation.
  Eg: An AbstractTransaction class could have instance variables like transactionId, amount, and timestamp, which are common to all types of financial transactions (e.g., CreditTransaction, DebitTransaction).
- **Final Methods:** An abstract class can define core logic or essential operations that should not be altered by subclasses, even while allowing other parts of the implementation to be abstract and customized.
  Eg: An AbstractRiskAssessment class might have a final method logRiskScore(double score) to ensure that the logging of risk scores follows a standardized, unchangeable procedure, even if the calculateRiskScore() method is abstract and implemented differently by various risk assessment models.

Abstraction vs Encapsulation

Abstraction	Encapsulation
Abstraction is the process or method of gaining the information.	While encapsulation is the process or method to contain the information.
In abstraction, problems are solved at the design or interface level.	While in encapsulation, problems are solved at the implementation level.
Abstraction is the method of hiding the unwanted information.	Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside.
We can implement abstraction using abstract class and interfaces.	Whereas encapsulation can be implemented using by access modifier i.e. private, protected and public.
In abstraction, implementation complexities are hidden using abstract classes and interfaces.	While in encapsulation, the data is hidden using methods of getters and setters.
The objects that help to perform abstraction are encapsulated.	Whereas the objects that result in encapsulation need not be abstracted.
Abstraction provides access to specific part of data.	Encapsulation hides data and the user can not access same directly (data hiding).
Abstraction focus is on “what” should be done.	Encapsulation focus is on “How” it should be done.

Interface

- Interfaces specify what a class must do and not how. Blueprint of a class. Mechanism to achieve complete abstraction and it specifies the behavior of a class. Represents the IS-A relationship and used to achieve loose coupling. To achieve total(100%) abstraction,
- Coupling describes the dependency of one class for the other. So, while using an interface, we define the method separately and the signature separately. This way, all the methods, and classes are entirely independent and achieves Loose Coupling.
- There can be only abstract methods in the Java interface, not the method body. All the methods are public and abstract. And all the fields are public, static, and final.
- We can use interface to achieve multiple Inheritance. class can implement more than one interface. Eg- ATM Machine Class
- From Java8, Interface can have default and Static Methods with implementation. Static methods allow you to place these utility methods directly within the interface itself instead of separate class, making them more discoverable and logically grouped with the interface they serve. Defines logic that is universally applicable to the interface and should remain consistent across all implementations.
- An interface which has no member is known as a marker/tagged interface,
- Eg- Serializable, Cloneable, Remote, etc. They are used to provide some
- essential information to the JVM so that JVM may perform some useful operation.



Private Methods in Interface - Java 9

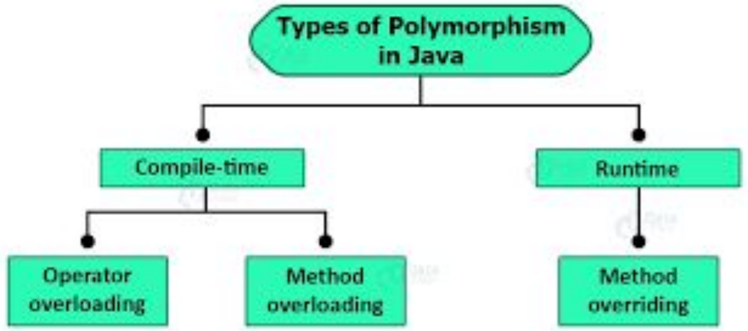
- **Private methods** will improve code re-usability inside interfaces and will provide choice to expose only our intended methods implementations to users.These methods are only accessible within that interface only and cannot be accessed or inherited from an interface to another interface or class. Remove the redundancy by sharing the common code of multiple default methods through private methods.
- **Private Static Methods:** Since java 8 we can have static methods in interfaces along with default methods. We can not share the common code of static methods using the non-static private method, we must have to use the private static method to do that.
- Private interface method cannot be abstract and no private and abstract modifiers together.
- Private method can be used only inside interface and other static and non-static interface methods.
- Private non-static methods cannot be used inside private static methods.
- We should use private modifier to define these methods and no lesser accessibility than private modifier.

Abstract Class vs Interface

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

Polymorphism

- If one task is performed in different ways, it is known as polymorphism. Polymorphism allows us to create consistent code. For example: to convince the customer differently, to draw something - for e.g. shape, triangle, rectangle, etc.
- Eg - A person at the same time can have different characteristics. Man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations.
- **Compile-time Polymorphism**: static polymorphism, achieved by function overloading.
- **Method Overloading**: When there are multiple functions with the same name but different parameters then these functions are said to be overloaded. Increases Readability. Functions can be overloaded by change in the number of arguments or/and a change in the type of arguments. Cannot overload by return type. Type Conversion but to higher type if exact prototype doesn't match. We can overload static methods. We can overload main method too.



Run Time Polymorphism

- Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature, and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class. Also Called **Dynamic Method Dispatch.** function call to the overridden method is resolved at Runtime.
- It is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.
- We can’t Override static methods, its data Hiding.
- **Upcasting**: If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. Eg- A a=new B();//upcasting
- The access modifier for an overriding method can allow more, but not less, access than the overridden method. Private methods can not be overridden. We can’t Override Constructor.
- Invoking overridden method from sub-class method using super() keyword.

Static

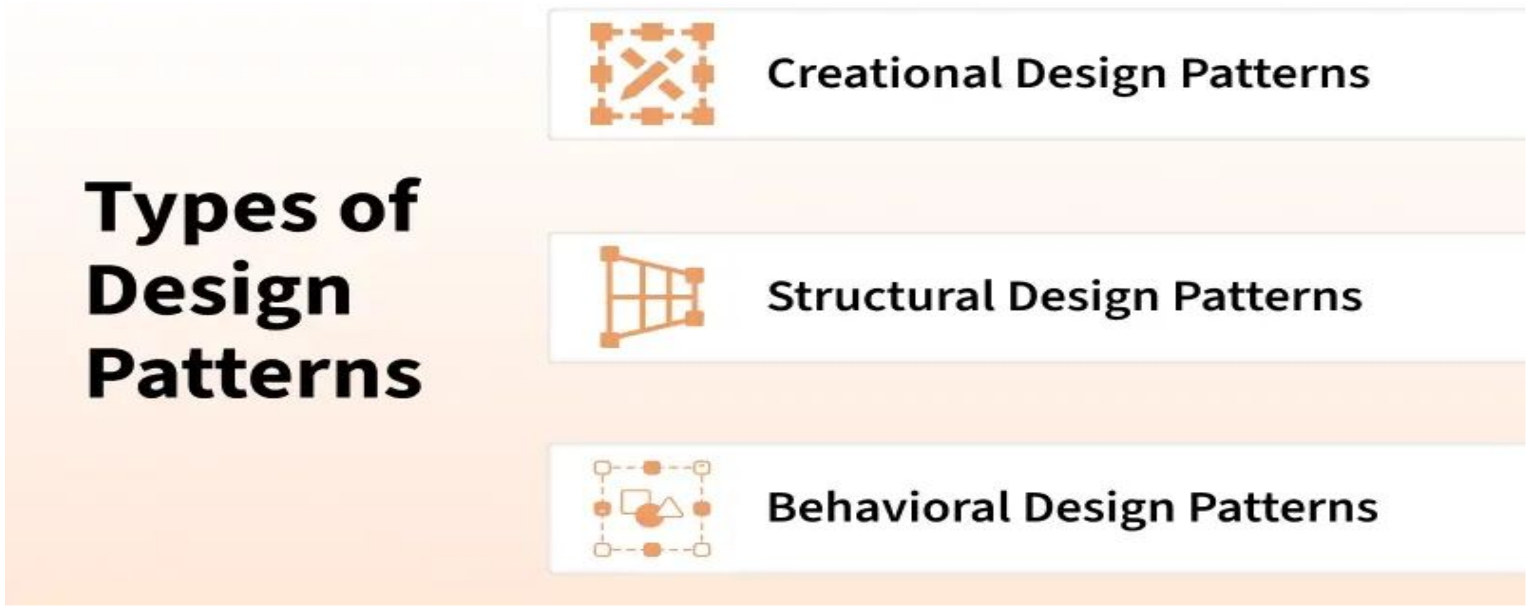
- Mainly used for memory management.
- **Static Variables**: Single copy of the variable is created and shared among all objects at the class level. Static variables are, essentially, global variables. All instances of the class share the same static variable. Gets memory only once in the class area at the time of class loading. Eg- Counter
- **Static method**: belongs to the class rather than the object of a class, can be invoked without the need for creating an instance of a class, can access static data member and can change the value of it, can not use non static data member or call non-static method directly. Eg – main() called by JVM
- **Static Class**: class can be made static only if it is a nested class. We cannot declare a top-level class with a static modifier but can declare nested class as static. Nested static class doesn't need a reference of Outer class. In this case, a static class cannot access non-static members of the Outer class.
- **Static Block**: In order to initialize your static variables, a static block that gets executed exactly once, when the class is first loaded before the main method.
- Eg: Student Class With Static College Name and Static Roll no.

Final

- Used to restrict the user.
- **Final Variables**: Its value can’t be modified, essentially, a constant. Final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. If the final variable is a reference, this means that the variable cannot be re-bound to reference another object. represent final variables in all uppercase, using underscore to separate words. blank final static variable can be initialized inside a static block. Final with foreach loop: final with for-each statement is a legal statement.
- **Final method**: Cannot be overridden
- **Final Class**: Prevent inheritance, as final classes cannot be extended. Eg- all Wrapper Classes like Integer, Float, etc. To create an immutable class like the predefined String class. One can not make a class immutable without making it final.

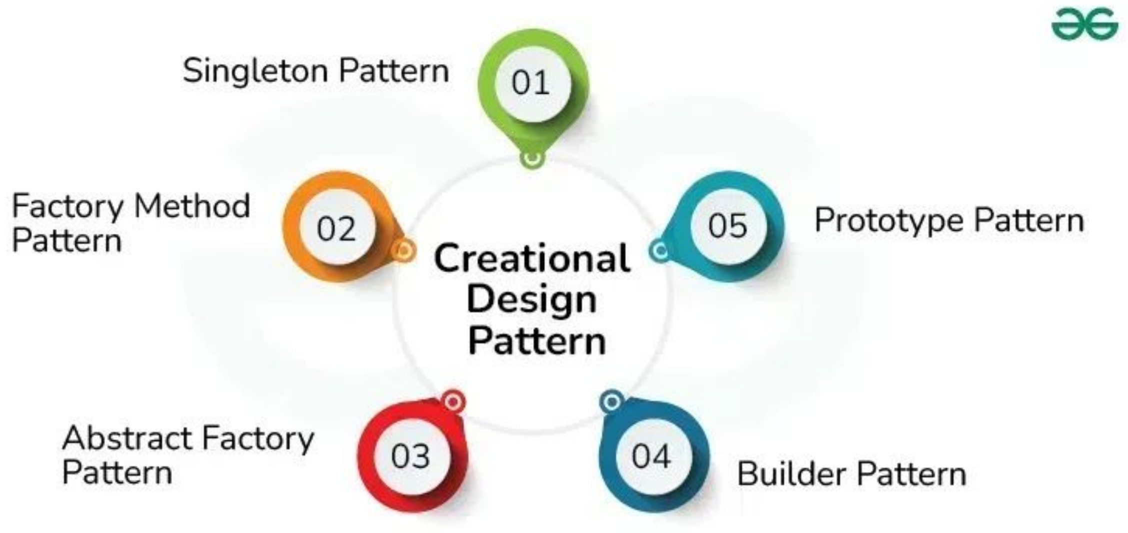
Design Patterns

- Design patterns in Java are reusable solutions to common problems encountered during software design and development. They are not specific pieces of code that can be directly copied and pasted, but rather templates or blueprints that describe how to solve a particular design issue in an object-oriented manner.
- **Reusable Solutions:** They offer proven, generalized solutions to recurring design problems, saving developers time and effort.
- **Established Best Practices:** They represent best practices identified by experienced software developers over time through trial and error.
- **Improved Communication:** They provide a common vocabulary for developers to discuss design challenges and solutions, fostering better collaboration.
- **Enhanced Maintainability and Scalability:** Using design patterns often leads to more organized, flexible, and easily maintainable codebases.


Types of Design Patterns
Creational Design Patterns
Structural Design Patterns
Behavioral Design Patterns

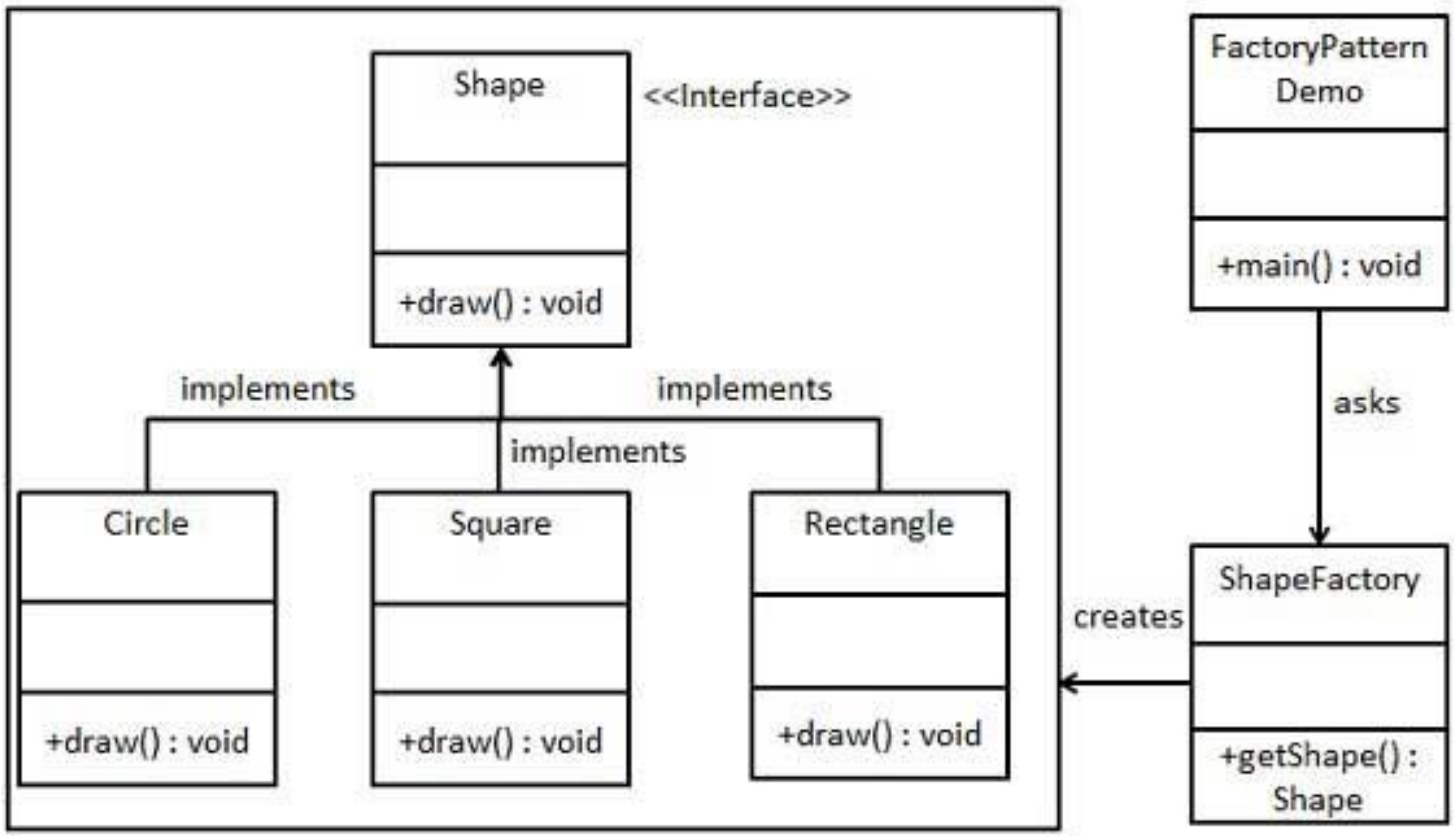
Creational Design Pattern

- Provides various object creation mechanisms, which increases flexibility and reuse of existing code.
- **Builder** is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.
- **Factory method** pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor.
- The **abstract factory** pattern is a creational pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes.



Factory Design Pattern

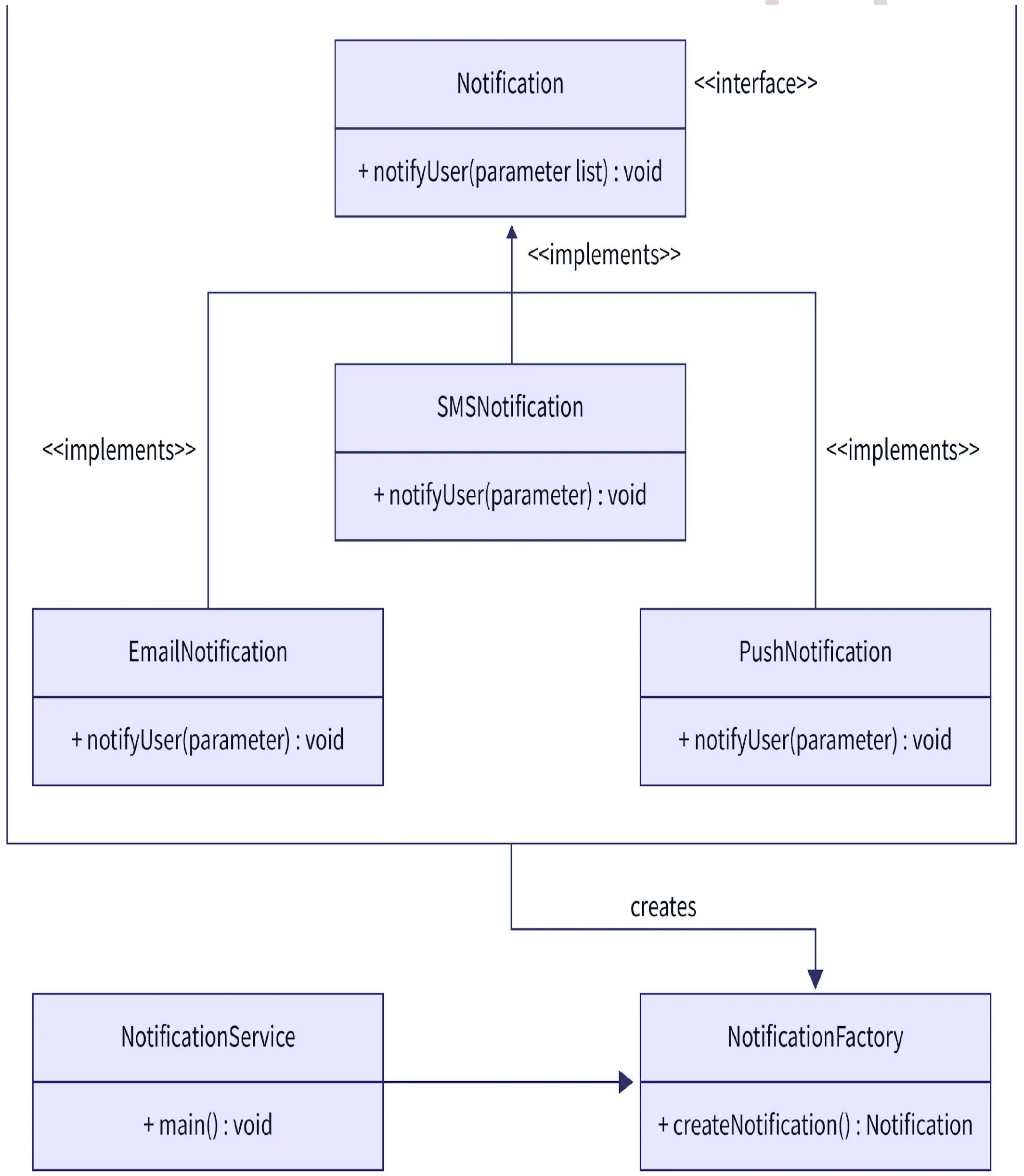
- Takes out the responsibility of the instantiation of a class from the client program to the factory class



Fenil Shah

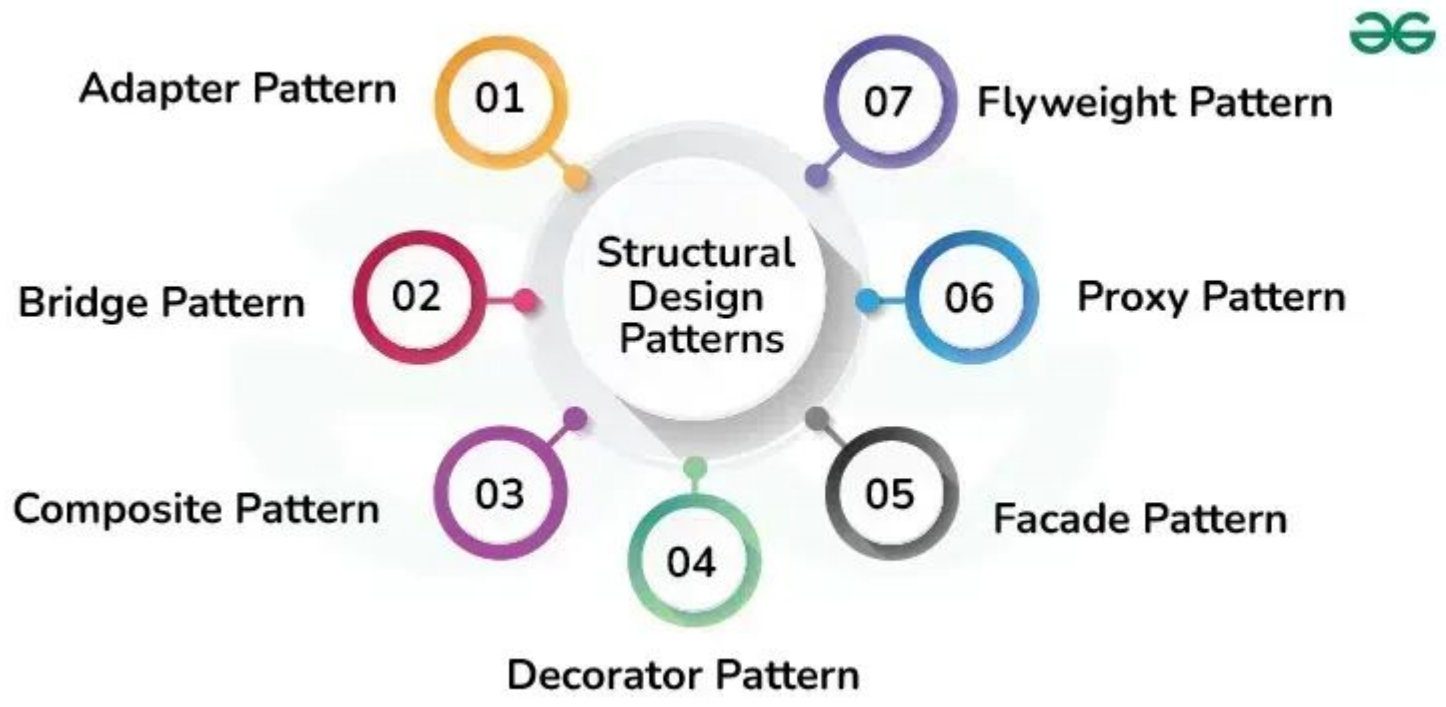
Factory Method Pattern / Virtual Constructor

- The factory design pattern is used in cases when there are multiple subclasses of a parent class or superclass. Using the input provided and the factory design pattern, we can return one of the subclasses of the superclass.
- In a factory design pattern, we define an interface or abstract class and the subclasses decide which object to instantiate.
- Factory pattern removes the instantiation of actual implementation classes from client code. Factory pattern makes our code more robust, less coupled and easy to extend. For example, we can easily change Sub class or factory method implementation because client program is unaware of this.
- Factory pattern provides abstraction between implementation and client classes through inheritance. It promotes loose coupling which helps eliminate the need to bind application-specific classes into the code and supports Open/Closed Principle.
- When object creation involves multiple steps, dependencies, or conditional logic, the factory pattern can encapsulate this complexity within the factory method, simplifying the client code.



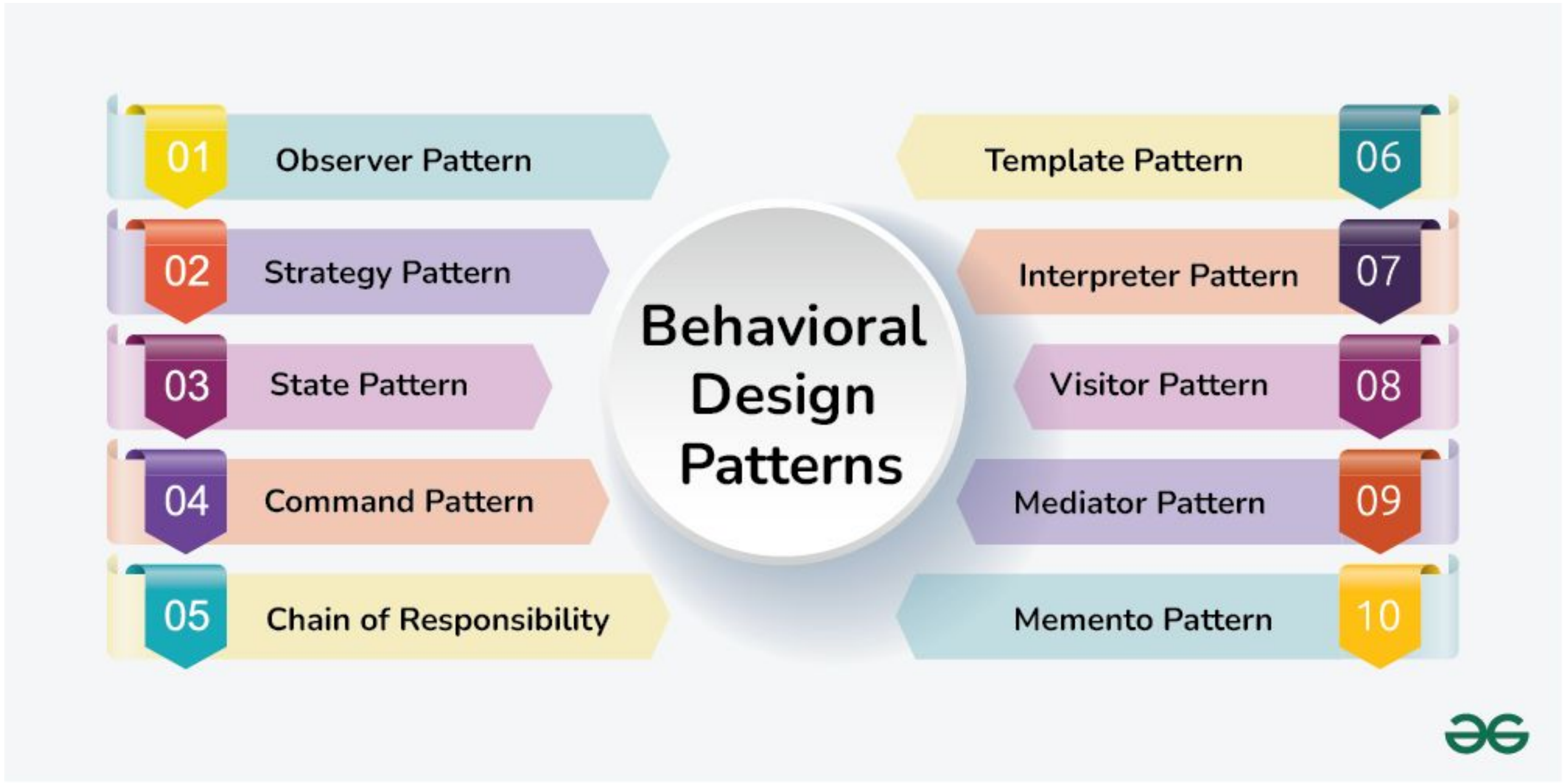
Structural Design Pattern

- Structural design patterns define how classes and objects are combined to form larger, flexible structures. They simplify object relationships, improve reusability and make systems easier to understand and maintain.



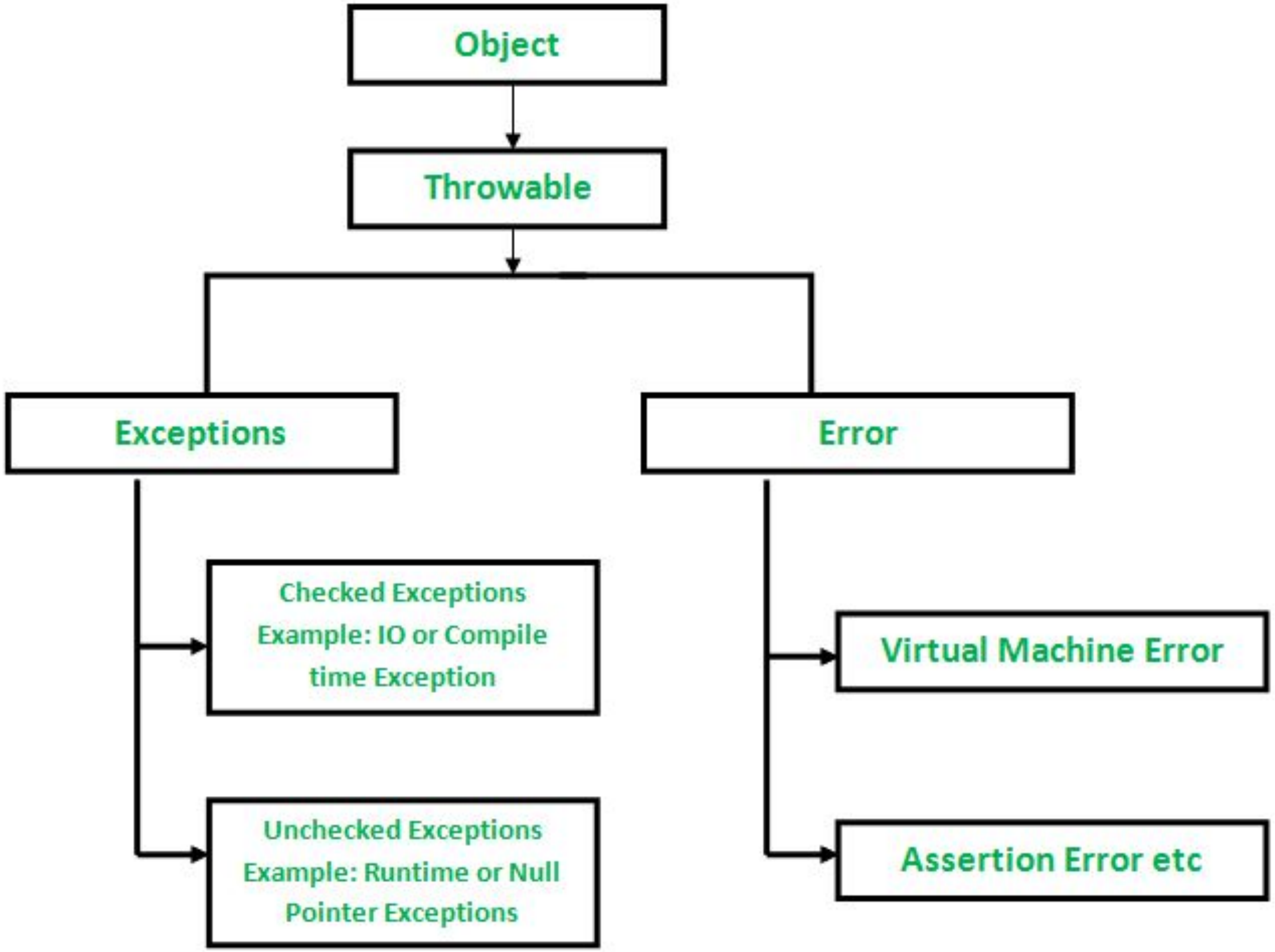
Behavioural Design Pattern

- Behavioral design patterns are a group of design patterns that focus on how objects and classes interact and communicate in software development. They emphasize the collaboration between objects to effectively accomplish tasks and responsibilities, making the system more manageable and adaptable



Exceptions

- Unwanted/Unexpected Event Occurring during execution of Program(runtime)
- Disrupts the normal Flow
- Creates and object called Exception Object (contains name, description, state, etc)
- Why? – Invalid User input, Code Errors, Opening Unavailable files, etc
- Eg- Access an array using an index that is out of bounds,


Object
Throwable
Exceptions
Error
Checked Exceptions Example: IO or Compile time Exception
Unchecked Exceptions Example: Runtime or Null Pointer Exceptions
Virtual Machine Error
Assertion Error etc

Exception Hierarchy

Checked Exception

- Checked Exceptions: Compile Time Exceptions, exceptions that are checked by the java compiler itself at compilation time and are not under runtime exception class hierarchy. Propagated to the caller method. Exception which are not Runtime.
- ClassNotFoundException - thrown when we attempt to use a class that does not exist.
- InstantiationException: Thrown when we try to create an object of abstract class or interface.
- IllegalAccessException: The IllegalAccessException is a checked exception and it is thrown when a method is called in another method or class but the calling method or class does not have permission to access that method.
- NoSuchFieldException: This is a checked exception that is thrown when an unknown variable is used in a program.

Unchecked(Runtime) Exception

- Not Checked at Compile time.Exceptions that are checked by JVM, not by java compiler. They occur during the runtime of a program.
- ArithmeticException - arithmetic problems, such as a number is divided by zero, is occurred. That is, it is caused by maths error.
- NullPointerException - It is thrown when the reference is null
- ArrayIndexOutOfBoundsException - when an array element is accessed out of the index.
- IllegalMonitorStateException: This exception is thrown when a thread does not have the right to monitor an object and tries to access wait(), notify(), and notifyAll() methods of the object.

Error

- Serious problem that a reasonable application should not try to catch.
- Beyond the control of the programmer.
- Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.

Exception Handling

- Handle Runtime Errors to maintain the normal program flow
- Method Creates Exception Object and gives to Run time System(JVM).
- Run time system searches the call stack to find the code that matches the type of exception thrown, else default exception handler – prints and terminates program abnormally

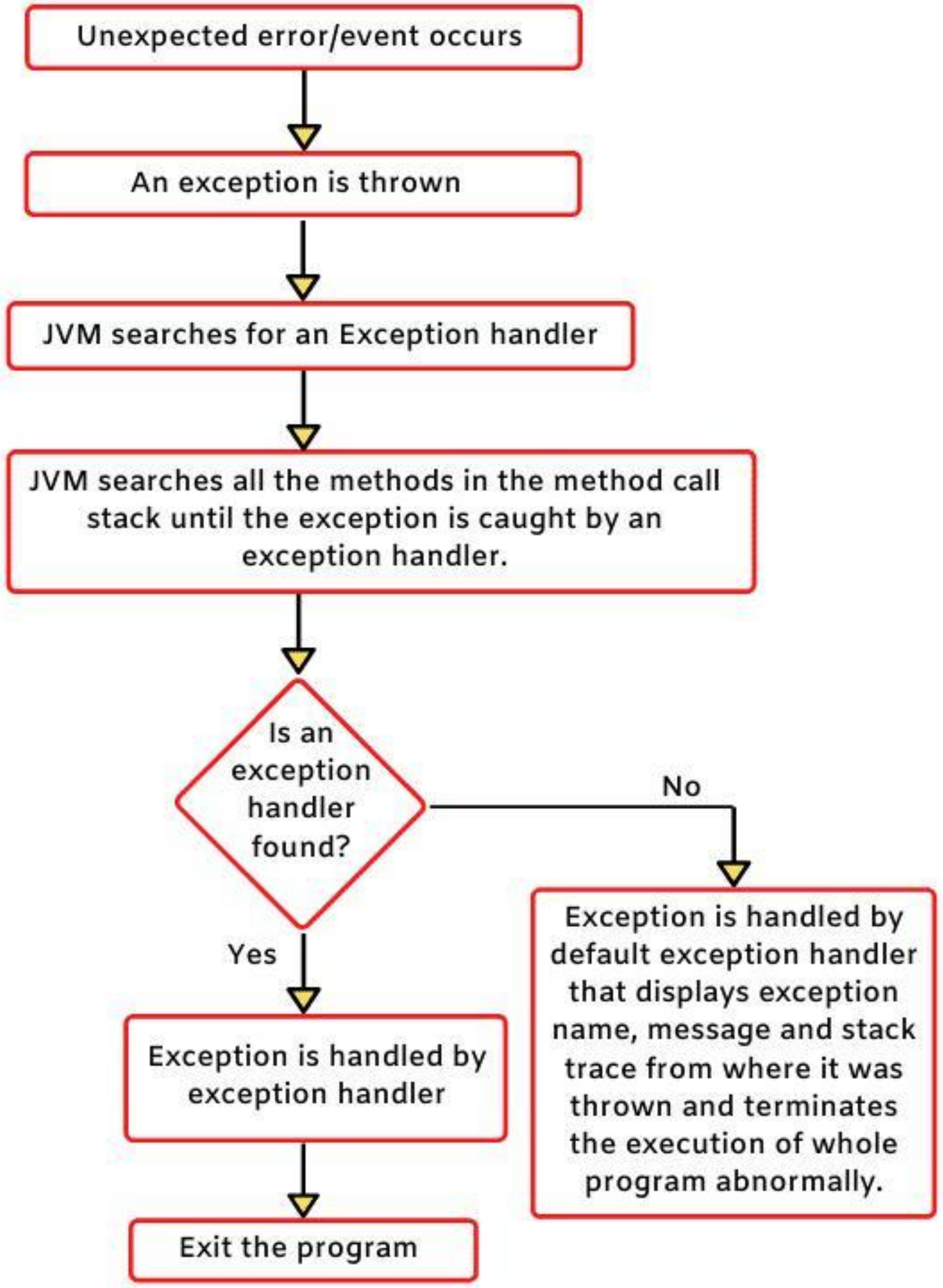
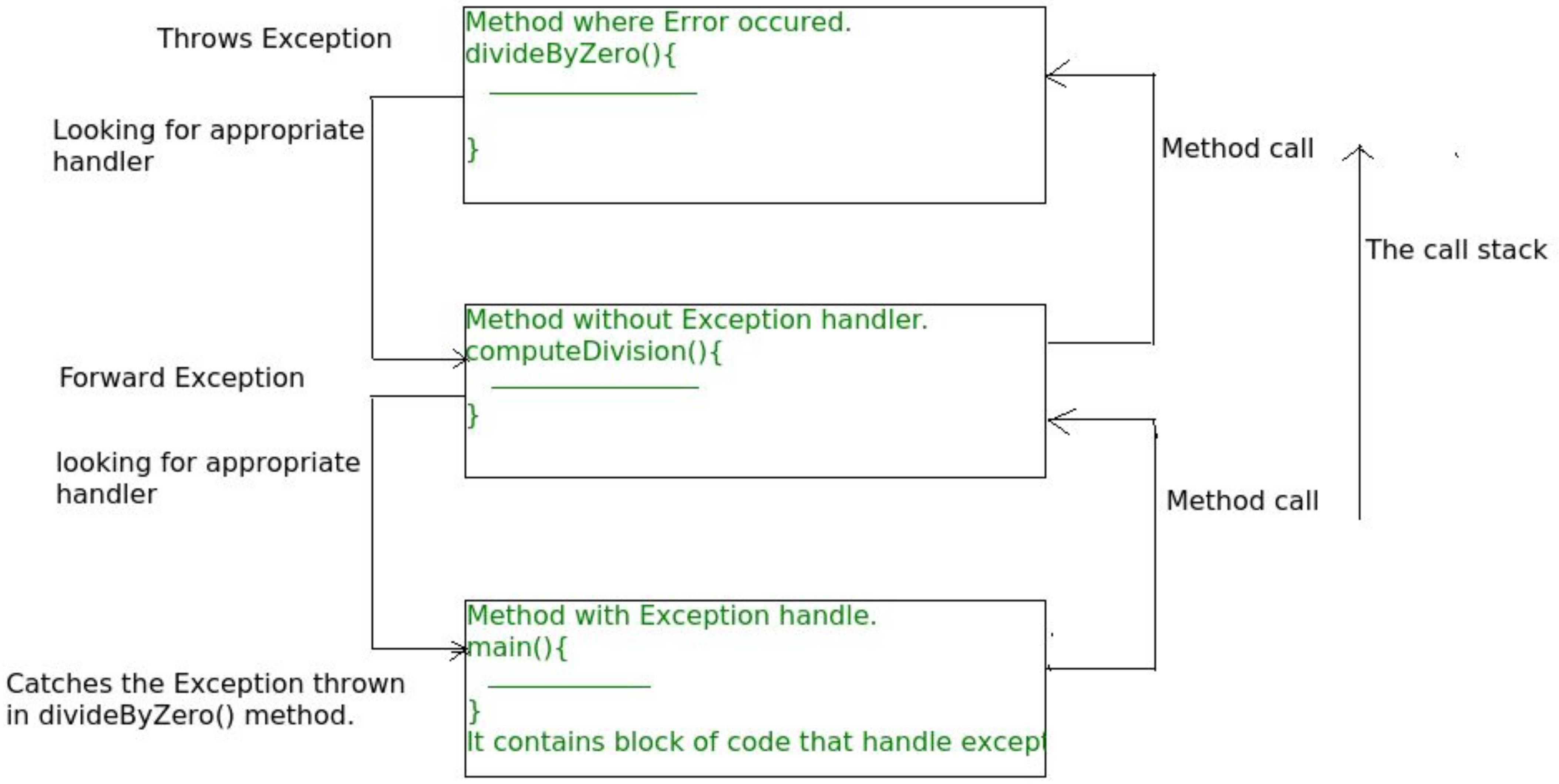

Unexpected error/event occurs
An exception is thrown
JVM searches for an Exception handler
JVM searches all the methods in the method call stack until the exception is caught by an exception handler.
Is an exception handler found?
No
Yes
Exception is handled by exception handler
Exit the program
Exception is handled by default exception handler that displays exception name, message and stack trace from where it was thrown and terminates the execution of whole program abnormally.

Fig: Exception handling mechanism



Throws Exception
Looking for appropriate handler
Forward Exception
looking for appropriate handler
Catches the Exception thrown in divideByZero() method.
Method where Error occured.
divideByZero(){
}
Method without Exception handler.
computeDivision(){
}
Method with Exception handle.
main(){
}
It contains block of code that handle except
Method call
Method call
The call stack

The call stack and searching the call stack for exception handler.

Exception Handling

- Try
- Catch: At a time only one exception occurs and at a time only one catch block is executed. All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.
- Finally: Important codes such as clean up code e.g. closing the file or closing the connection. The finally block executes whether exception rise or not and whether exception handled or not. For each try block there can be zero or more catch blocks, but only one finally block.

Java – Try with Resources

- Allows us to declare resources to be used in a try block with the assurance that the resources will be closed after the execution of that block.
- The resources declared need to implement the *AutoCloseable* interface.
- We can declare multiple resources just fine in a *try-with-resources* block by separating them with a semicolon:

-
  ```
  try (Scanner scanner = new Scanner(new File("testRead.txt"));
   PrintWriter writer = new PrintWriter(new File("testWrite.txt"))) {
    while (scanner.hasNext()) { writer.print(scanner.nextLine()); } }
  ```

Exception Handling with Method Overriding

- If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception. Why? - If checked exception is thrown, it will start giving compilation error and fails Liskov Substitution principle.
- Ref: https://www.geeksforgeeks.org/java/exception-handling-with-method-overriding-in-java/
- If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

THROW

- used throw an exception explicitly in the code, inside the function or the block of code.
- We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.
- Instance must be of type Throwable or a subclass of Throwable

THROWS

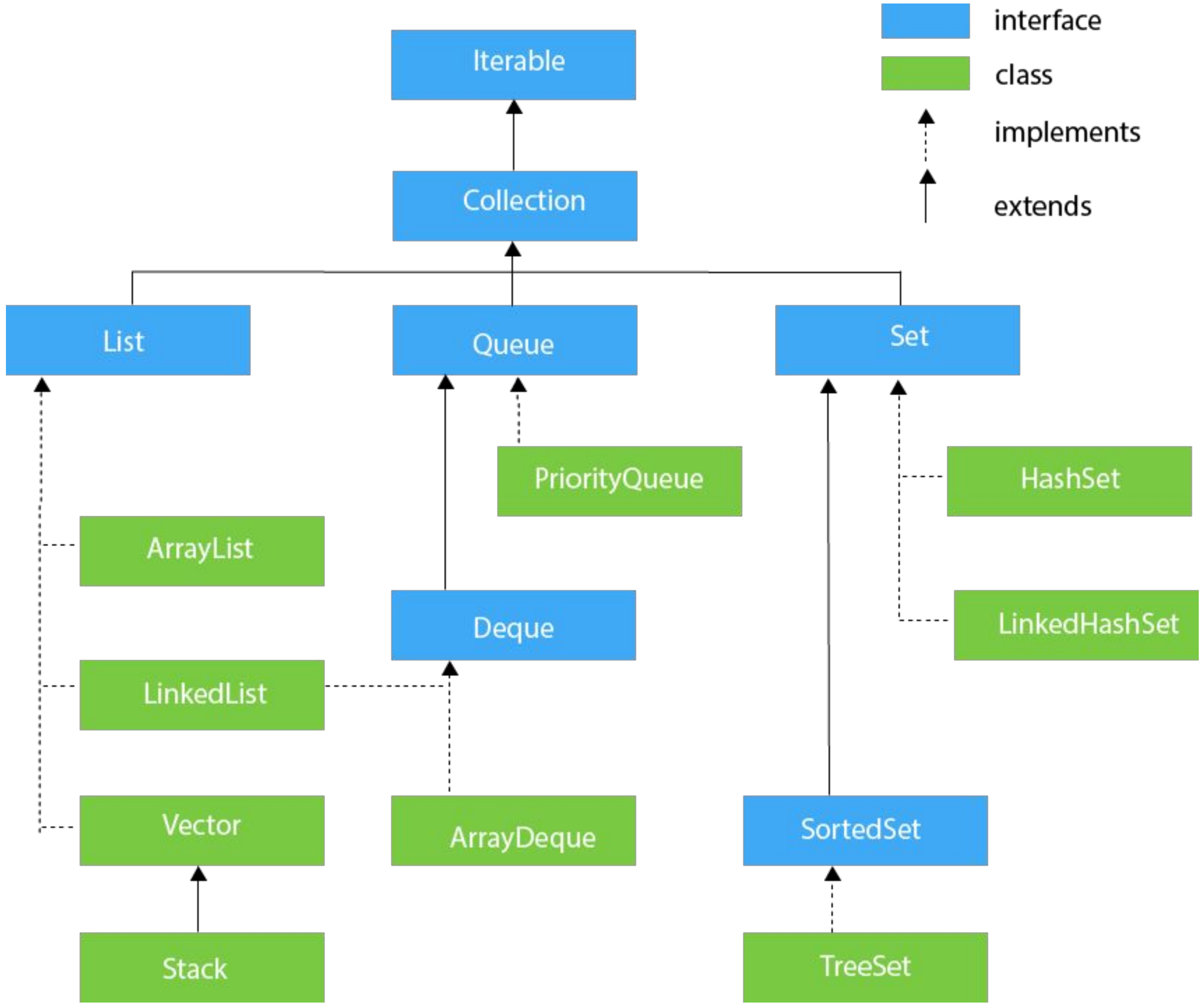
- used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
- We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.
- throws keyword is required only for checked exception. required only to convince compiler and usage of throws keyword does not prevent abnormal termination of program.

Custom Exceptions

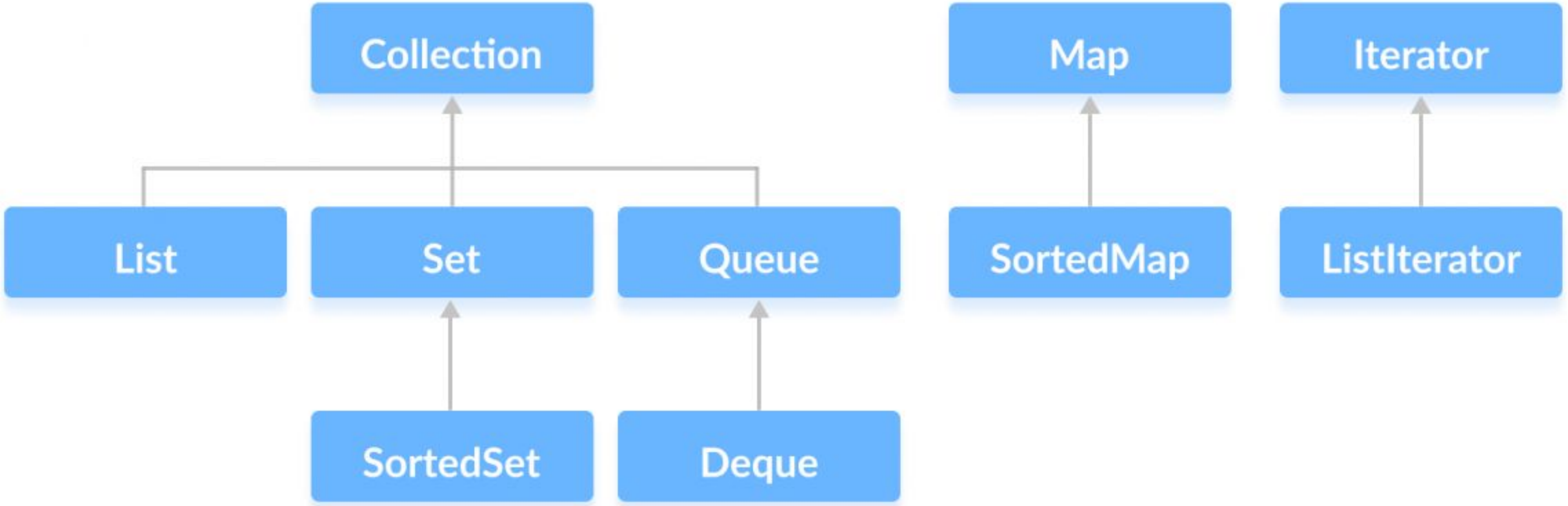
- Creating our own Exception, which are basically derived classes of Exception is known as a custom exception or user-defined exception.
- Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem. Eg - UserNotFoundException

Collections

- Any group of individual objects which are represented as a single unit.
- Provides a set of interfaces and classes to implement various data structures and algorithms.
- The Collection interface is the root interface of the Java collections framework.
- There is no direct implementation of this interface. However, it is implemented through its subinterfaces like List, Set, and Queue.
- Collection Framework: Set of classes and interfaces which provide a ready-made architecture. Reduces programming effort, A programmer doesn't have to worry about the design of the Collection (achieving abstraction). Increases performance by providing high-performance implementations of useful data structures and algorithms


interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet

Java Collections Framework



- **Iterable Interface**: Root interface for the entire collection framework. The collection interface extends the iterable interface. Therefore, inherently, all the interfaces and classes implement this interface. The main functionality of this interface is to provide an iterator for the collections. Therefore, this interface contains only one abstract method which is the iterator. It returns the Iterator iterator(); It has other default methods from java 8.

- **Collection Interface**: Root interface of the collections framework hierarchy. Contains various methods which can be directly used by all the collections which implement this interface.

- **add**(Object) used to add an object to the collection.
- **addAll**(Collection c) adds all the elements in the given collection to this collection
- **clear**() removes all of the elements from this collection.
- **contains**(Object o) returns true if the collection contains the specified element.
- **equals**(Object o) compares the specified object with this collection for equality.
- **parallelStream**() returns a parallel Stream with this collection as its source.
- **remove**(Object o) Used to remove the given object from the collection. If there are duplicate values, then this method removes the first occurrence of the object.
- **toArray**() Used to return an array containing all of the elements in this collection.
- **size**() Used to return the number of elements in the collection.

List Interface

- List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.
- **ArrayList**:
- In Java, we need to declare the size of an array before we can use it. Once the size of an array is declared, it's hard to change it.
- To handle this issue, we can use the ArrayList class. It allows us to create resizable arrays.
- Unlike arrays, arraylists can automatically adjust their capacity when we add or remove elements from them. Hence, arraylists are also known as dynamic arrays. Eg- Getting List of records from db, getting transaction details of user

- **Vector:** synchronizes each individual operation. continuous use of lock for each operation makes vectors less efficient. helpful in programs where lots of manipulation in the array is needed. Vector methods are synchronized, making it inherently thread safe and used in Multi Threaded environment.   Eg- Vector<Integer> vector= new Vector<>();
- **LinkedList**: Implementation of the LinkedList data structure which is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node. Provides the doubly-linked list implementation. To access an element, we need to iterate from the beginning to the element. Eg- Image viewer – Previous and next images are linked, hence can be accessed by next and previous button, Music Player, etc
- **Stack**: Extends the vector class models and implements the Stack data structure. In stack, elements are stored and accessed in Last In First Out manner. That is, elements are added to the top of the stack and removed from the top of the stack. Methods – push(), pop(), peek(), search(), empty().

Eg – Undo/Redo Operation, Recursion, etc

- **Queue Interface**: a queue interface maintains the FIFO(First In First Out) order similar to a real-world queue line. Eg - Person whose request arrives first into the queue gets the ticket.
- **ArrayDeque**: Way to apply resizable-array. This is a special kind of array that grows and allows users to add or remove an element from both sides of the queue. Array deques have no capacity restrictions and they grow as necessary to support usage. Faster than ArrayList and Stack

Iterator Interface

- Iterator interface provides the facility of iterating the elements in a forward direction only.
- All the Java collections include an iterator() method. This method returns an instance of iterator used to iterate over elements of collections.
- Methods: hasNext(), next() – returns next element, remove() – removes last element returned by next and forEachRemaining() – performs action for each remaining element.

Set Interface

- represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.
- **Hashset:** Commonly used if we have to access elements randomly. It is because elements in a hash table are accessed using hash codes. Cannot contain duplicate elements. Hence, each hash set element has a unique hashcode. HashSet is not synchronized. Best approach for search operations.
- All the classes of Set interface are internally backed up by Map. HashSet uses HashMap for storing its object internally. Storage in HashMap: Actually the value we insert in HashSet acts as a key to the map Object and for its value, java uses a constant variable. So in the key-value pair, all the values will be the same.

Map Interface

- supports the key-value pair mapping for the data. This interface doesn’t support duplicate keys because the same key cannot have multiple mappings. A map is useful if there is data and we wish to perform operations on the basis of the key
- **HashMap:** It stores the data in (Key, Value) pairs. To access a value in a HashMap, we must know its key. HashMap uses a technique called Hashing. Hashing is a technique of converting a large String to a small String that represents the same String so that the indexing and search operations are faster. HashSet also uses HashMap internally. Default capacity will be 16 and the default load factor will be 0.75.
- Eg- To Implement Phonebook (Name - Number)