Rules of Interfaces

By Syed Zabi Ulla Sir,
Java & DSA Mentor,
PW Skills

#1. An interface helps to achieve Standardization.

#2. An interface provides a contract for the implementing classes which promise to provide the implementation for the abstract methods defined in the contract.

#3. A class is a blueprint of an object and an interface is a blueprint of a class.

#4. Interfaces helps to achieve 100% abstraction as it contains only abstract methods (methods without implementation).

#5. An interface can have any number of implementing classes.

INTERFACES

$$\begin{tikzcd} \textbf{PL} & \text{JDBC API} & \textbf{DBMS} \\ \text{Java} \arrow[r, leftrightarrow] & \text{Driver} \arrow[r, leftrightarrow] & \text{Oracle} \\ & & \text{Sybase} \\ & & \text{Informix} \end{tikzcd}$$

```
interface Connection
{
    getConnection();
    executeQuery();
    close();
}
```

interface -> Blueprint of a class

class -> Blueprint of an object

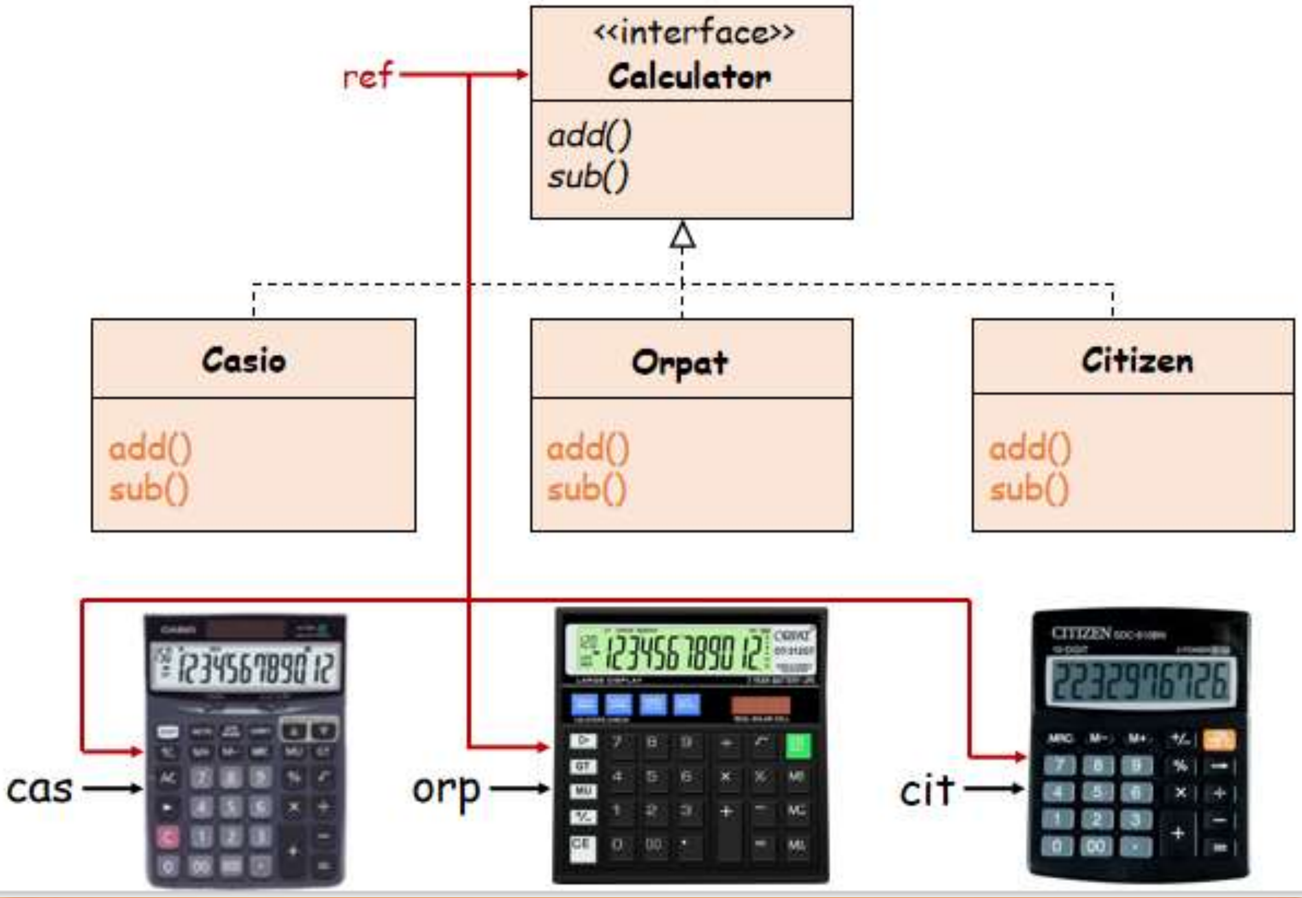
object -> Real-life Entity

```
class OracleDB  implements Connection
{
    getConnection()
    ~~buildConnection()~~
    executeQuery()
    ~~runQuery()~~
    close()
    ~~exitConnection()~~
}

class SybaseDB  implements Connection
{
    ~~openConnection()~~
    ~~processQuery()~~
    ~~quitConnection()~~
}

class InformixDB  implements Connection
{
    ~~acceptConnection()~~
    ~~operateQuery()~~
    ~~terminateConnection()~~
}
```

"STANDARDIZATION"

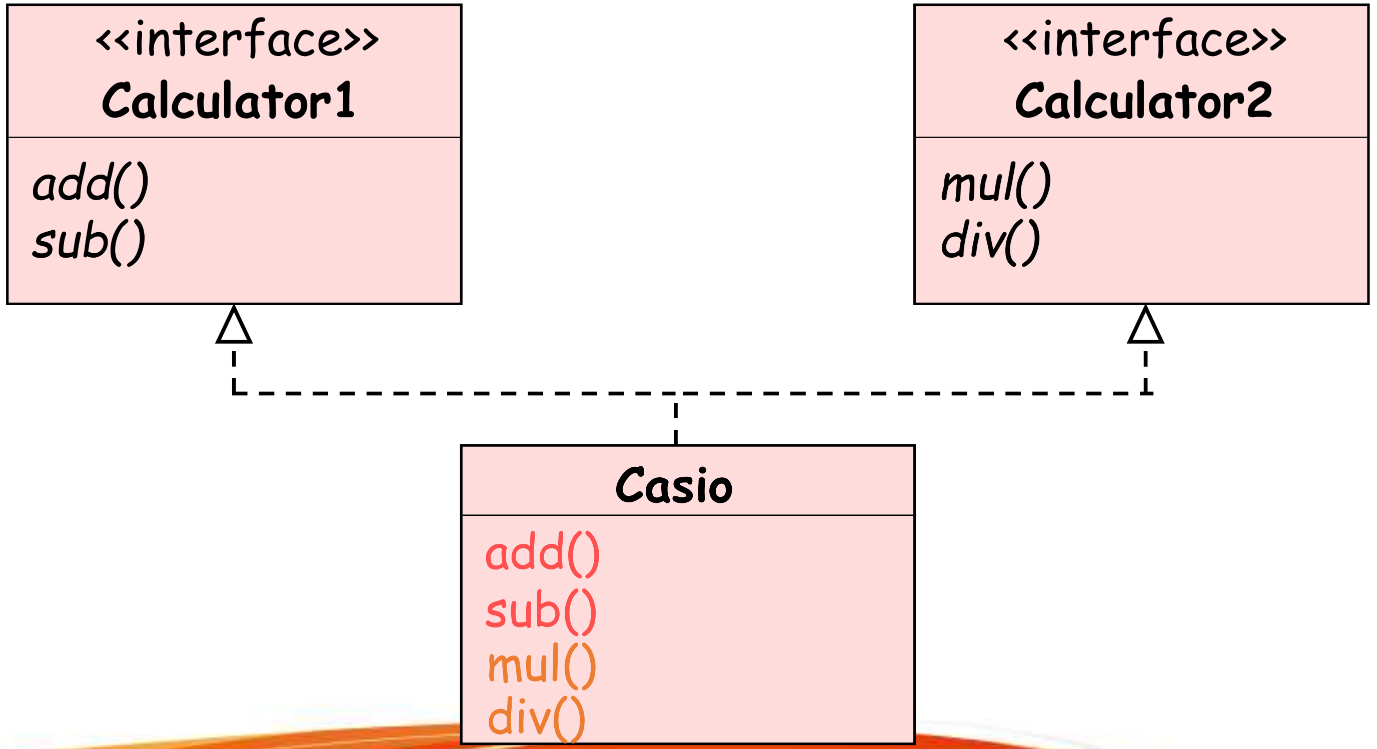
#6. An interface cannot be instantiated. But the reference of an interface can be created. Using interface type reference, we can access the overridden methods of the implementing classes. Interfaces promotes loose-coupling & hence Polymorphism can be achieved.

Note: Using interface type reference, we can access only the overridden methods of the implementing classes. But we cannot directly access the specialized methods of the implementing classes. However, if specialized methods muct be accessed then we must perform "down-casting".

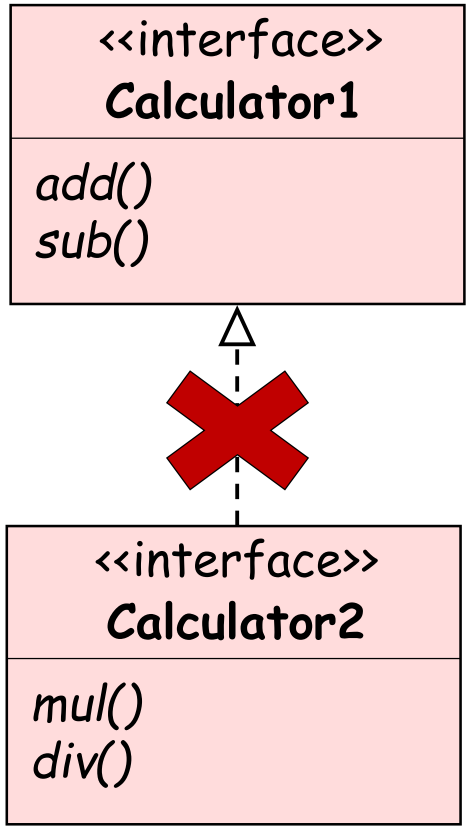


#7. A class can implement any number of interfaces. Hence, multiple inheritance in Java is indirectly achieved using interfaces. However, the same is not possible incase of extending multiple classes as it would lead to ambiguity (Diamond-Shape Problem).

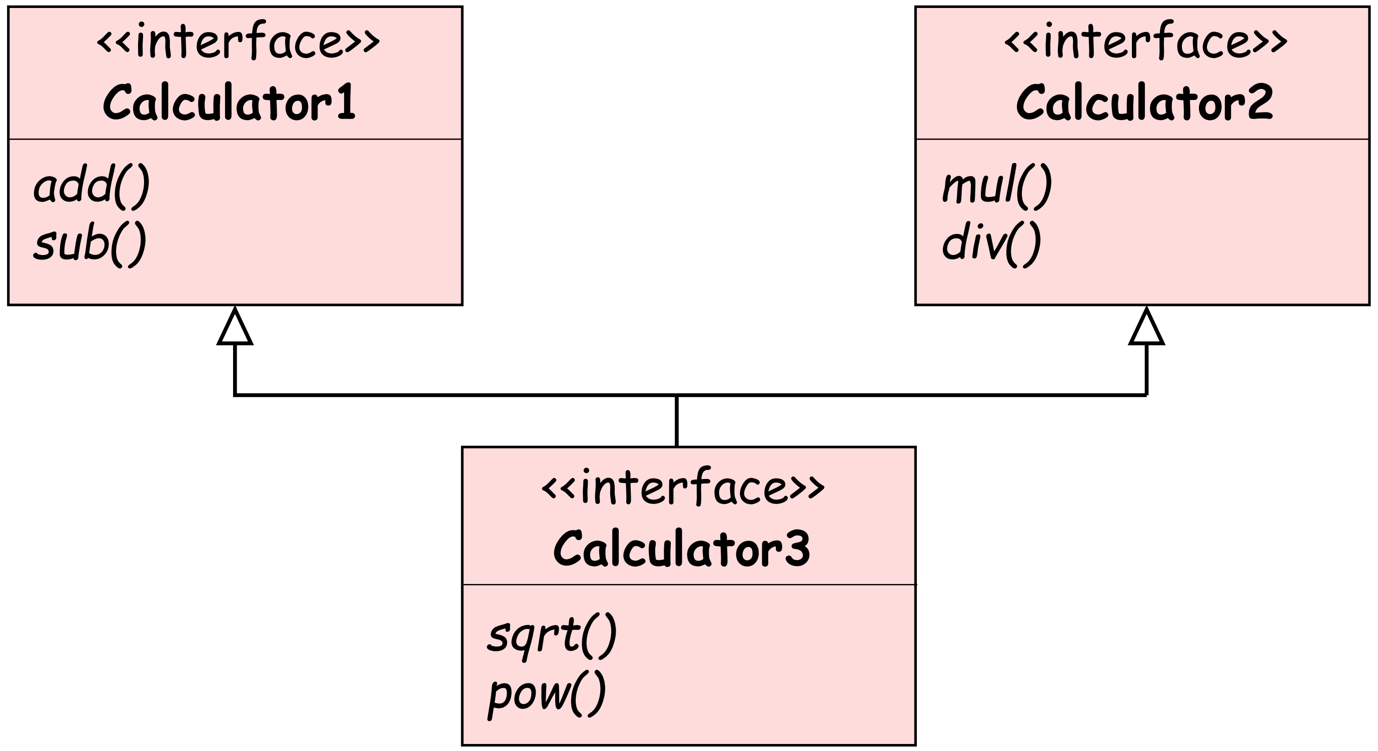
But incase of interfaces, there won't be any ambiguity even if same method is defined in multiple interfaces, because its implementation will be provided by the implementing classes & only the method signature will be provided in the interfaces.



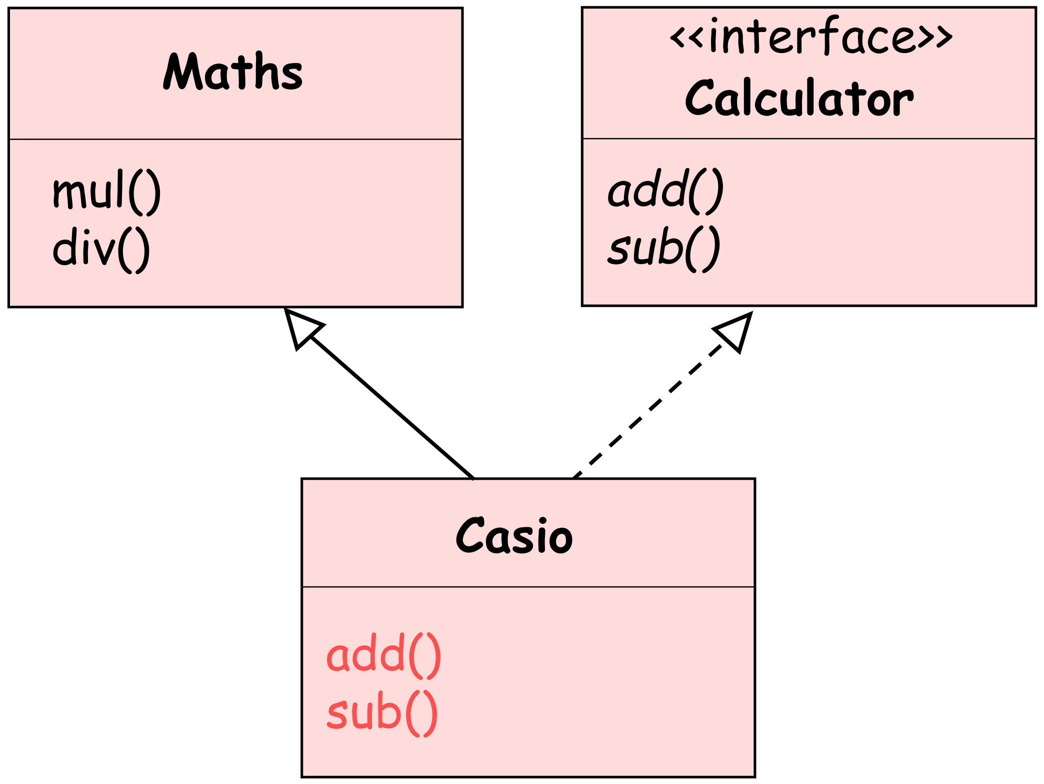
#8. An interface cannot cannot implement another interface. This is because an interface can contain only method signatures & not method implementations.



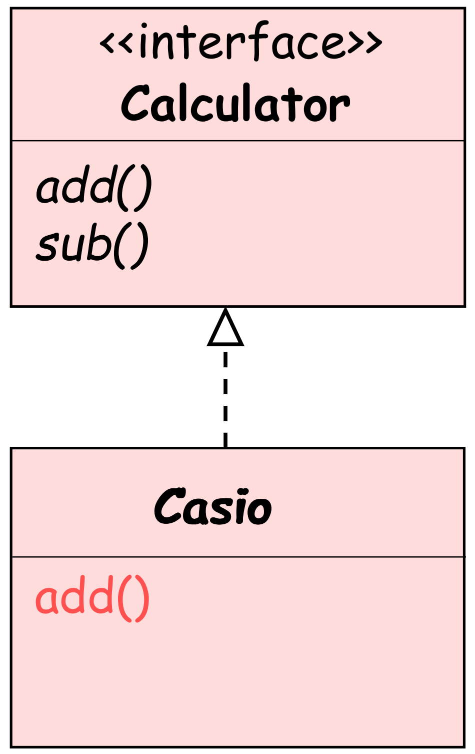
#9. An interface can extend any number of interfaces. Hence, through this also multiple inheritance can be achieved in Java.



#10. If a class is both extending from another class and implementing an interface then it must first extend the class and then implement the interface.



#11. A class can partially implement an interface by declaring itself as abstract.



#12. An interface cannot only have method signatures but it can also have variables (constants).

*** Methods within an interface are by default public & abstract.**

*** Variables within an interface are by default public, static & final.
In otherwords, they are constants.**

```
interface ATM
{
    int MAX_ATTEMPTS = 3;

    void withDraw();
    void checkBalance();
}
```

After Compilation

```
interface ATM
{
       public static finalint MAX_ATTEMPTS = 3;

       public abstractvoid withDraw();
       public abstractvoid checkBalance();
}
```

#13. It is possible to declare an empty interface in Java. It is also called as Tagged interface or Marker interface.

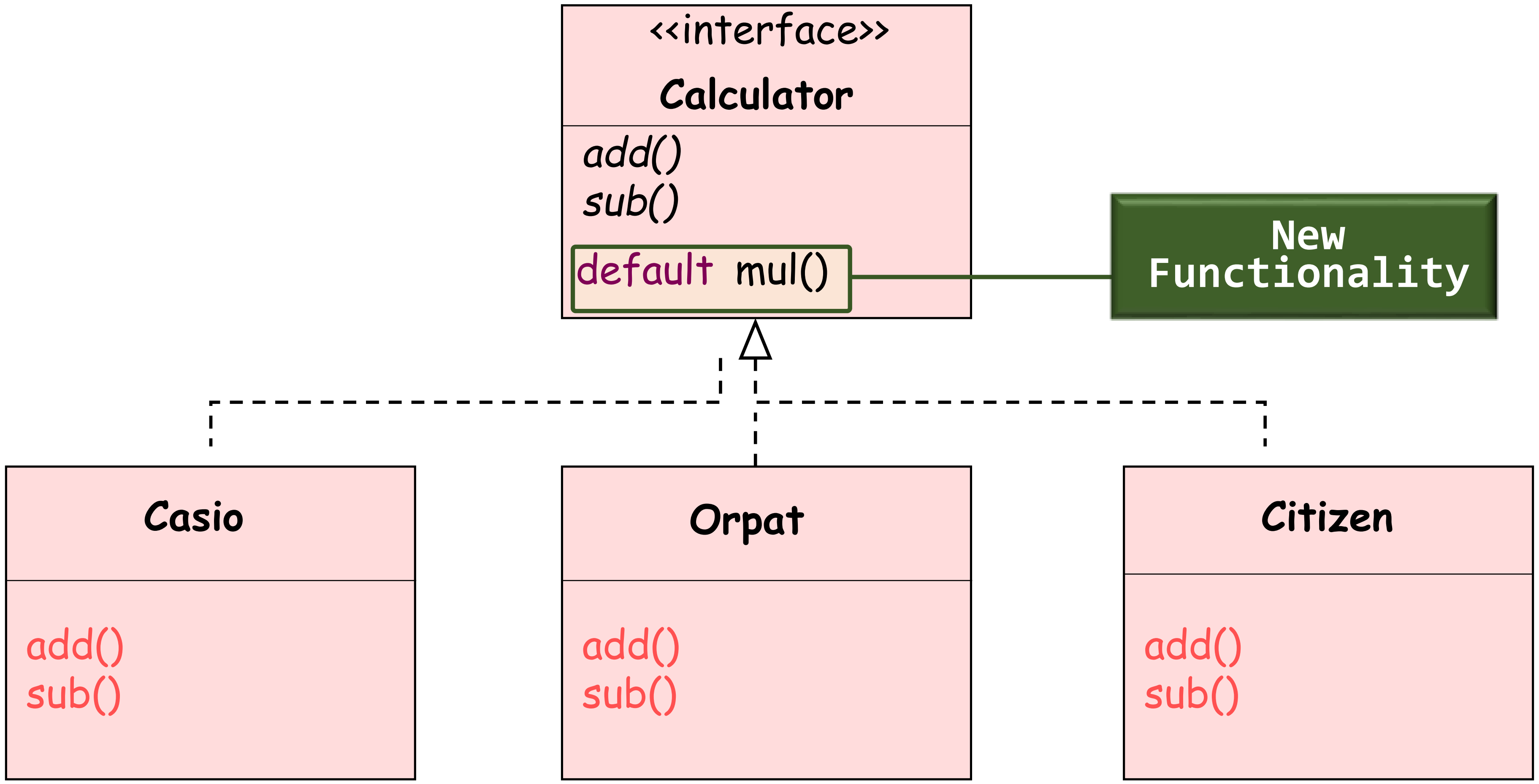
```
interface Calculator
{

}
```

Tagged Interface
or
Marker Interface

#14. An interface can have multiple implementations. However, if a new functionality (abstract method) is added to an existing interface then all the implementing classes will be forced to provide the implementation for the new abstract method added in the interface or their code would break.
From Java 8, default methods (concrete methods) allow us to add new methods to an interface that are automatically available in the implementing classes. Hence, we don't need to modify the implementing classes. This ensures backward compatibility.

Features of default methods:
* They are automatically inherited to the implementing classes.
* They are implicitly public.
* They can be overridden.
* They can be invoked only with the instance of the implementing class.

```
        Casio cas = new Casio();
        cas.mul();
```



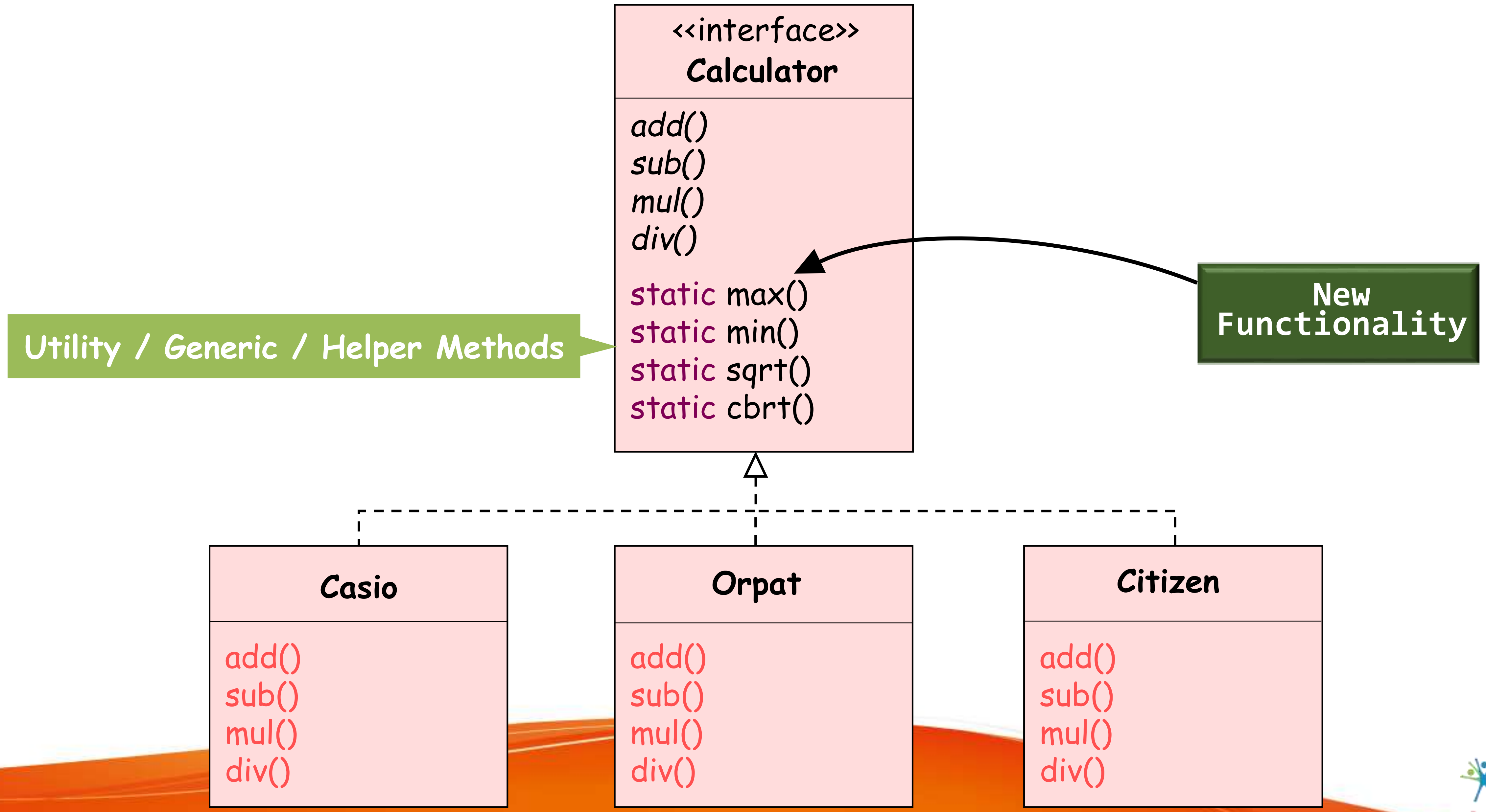
<<interface>>
Calculator
add()
sub()
default mul()
New Functionality
Casio
add()
sub()
Orpat
add()
sub()
Citizen
add()
sub()

#15. From Java 8, in addition to default methods, we can also define static methods in an interface. These static methods cannot be inherited & overridden in the implementing classes. In other words, it must be invoked only by using the interface name. This feature enables us to provide utility methods in the interface itself instead of creating a separate utility class.

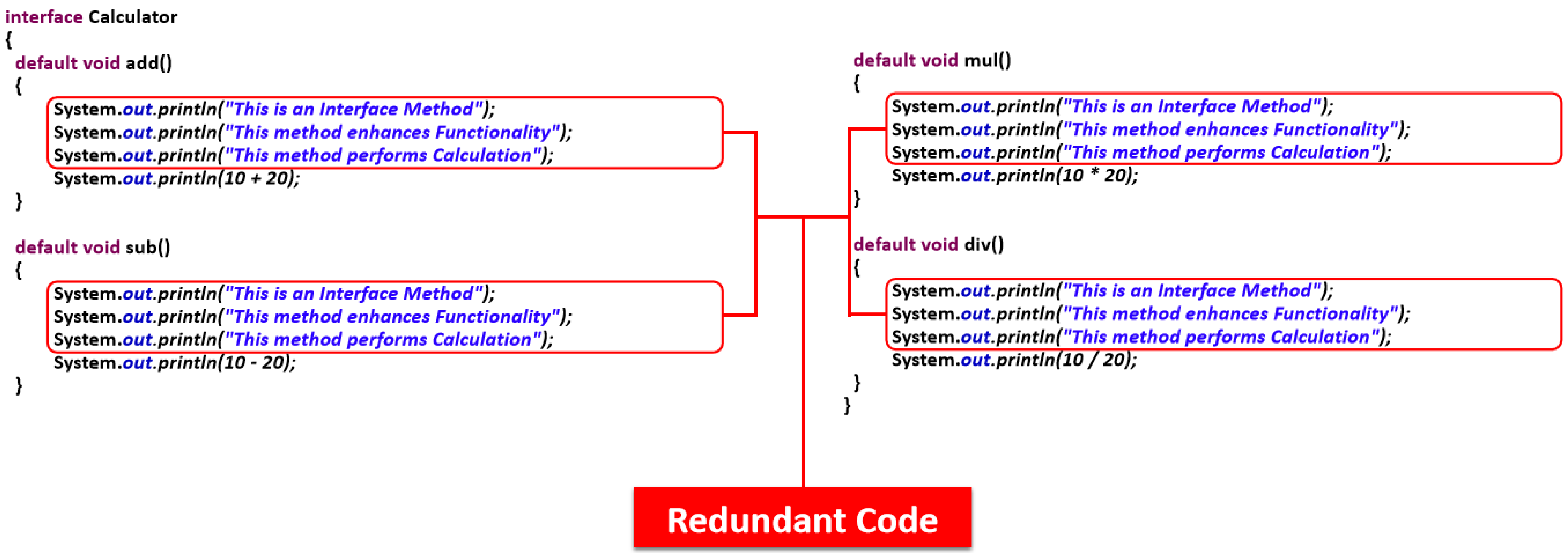
Features of static methods:

* They are implicitly public.
* Static methods in interfaces are never inherited (static methods in classes are inherited).
* They can be invoked only by using the interface name.

Interface Static Methods



#16. From Java 9, private methods can be added to interfaces. They can be implemented as static or non-static. These private methods will improve code re-usability inside interfaces & will provide choice to expose only our intended methods' implementations to users. These methods are only accessible within the interface only & cannot be accessed or inherited from an interface to another interface or class.



```
interface Calculator
{
 default void add()
 {
     System.out.println("This is an Interface Method");
     System.out.println("This method enhances Functionality");
     System.out.println("This method performs Calculation");
     System.out.println(10 + 20);
 }

 default void sub()
 {
     System.out.println("This is an Interface Method");
     System.out.println("This method enhances Functionality");
     System.out.println("This method performs Calculation");
     System.out.println(10 - 20);
 }

 default void mul()
 {
     System.out.println("This is an Interface Method");
     System.out.println("This method enhances Functionality");
     System.out.println("This method performs Calculation");
     System.out.println(10 * 20);
 }

 default void div()
 {
     System.out.println("This is an Interface Method");
     System.out.println("This method enhances Functionality");
     System.out.println("This method performs Calculation");
     System.out.println(10 / 20);
 }
}
```

Private Interface Methods

```
interface Calculator
{
 default void add()
 {

       printInfo();

       System.out.println(10 + 20);
 }

 default void sub()
 {

       printInfo();

       System.out.println(10 - 20);
 }
```

```
default void mul()
{

      printInfo();

      System.out.println(10 * 20);
}

default void div()
{

      printInfo();

      System.out.println(10 / 20);
}
```

Java 9

```
  private   void printInfo()
 {
  System.out.println("This is an Interface Method");
  System.out.println("This method enhances Functionality");
  System.out.println("This method performs Calculation");
 }

}
```

Code Reusability

Encapsulation

Expose Only Intended Methods

Private Static Interface Methods

```
interface Calculator
{
  static void add()
  {

    printInfo(); ✓

    System.out.println(10 + 20);
  }

  static void sub()
  {

    printInfo(); ✓

    System.out.println(10 - 20);
  }

  static void mul()
  {

    printInfo(); ✓

    System.out.println(10 * 20);
  }

  static void div()
  {

    printInfo(); ✓

    System.out.println(10 / 20);
  }

    private static void printInfo()
    {
      System.out.println("This is an Interface Method");
      System.out.println("This method enhances Functionality");
      System.out.println("This method performs Calculation");
    }
  }
```

Code Reusability

Encapsulation

Expose Only Intended Methods

#17. An interface that contains exactly one abstract method is known as a Functional interface. However, it can have any number of default, static and private methods but can contain only one abstract method.
Functional interfaces are also known as Single Abstract Method (SAM) interfaces.
It is a new feature in Java which helps to achieve functional programming approach from Java 8.

```java
interface Calculator
{
   void max();
}
```

Functional Interface

#18. .class file will be generated for every interface by the compiler.
From Java 8, it is possible to define main() method (static method) within an interface and execute the program. In other words, without declaring a class, it is possible to execute a Java program.

```
interface Launch
{
  public static void main(String[] args)
  {
    System.out.println("main() inside an interface");
  }
}
```

Output:

```
main() inside an interface
```

THANK YOU

By Syed Zabi Ulla Sir,
Java & DSA Mentor,
PW Skills