HEXAWARE

Abstract Classes & Interfaces

Course Objectives

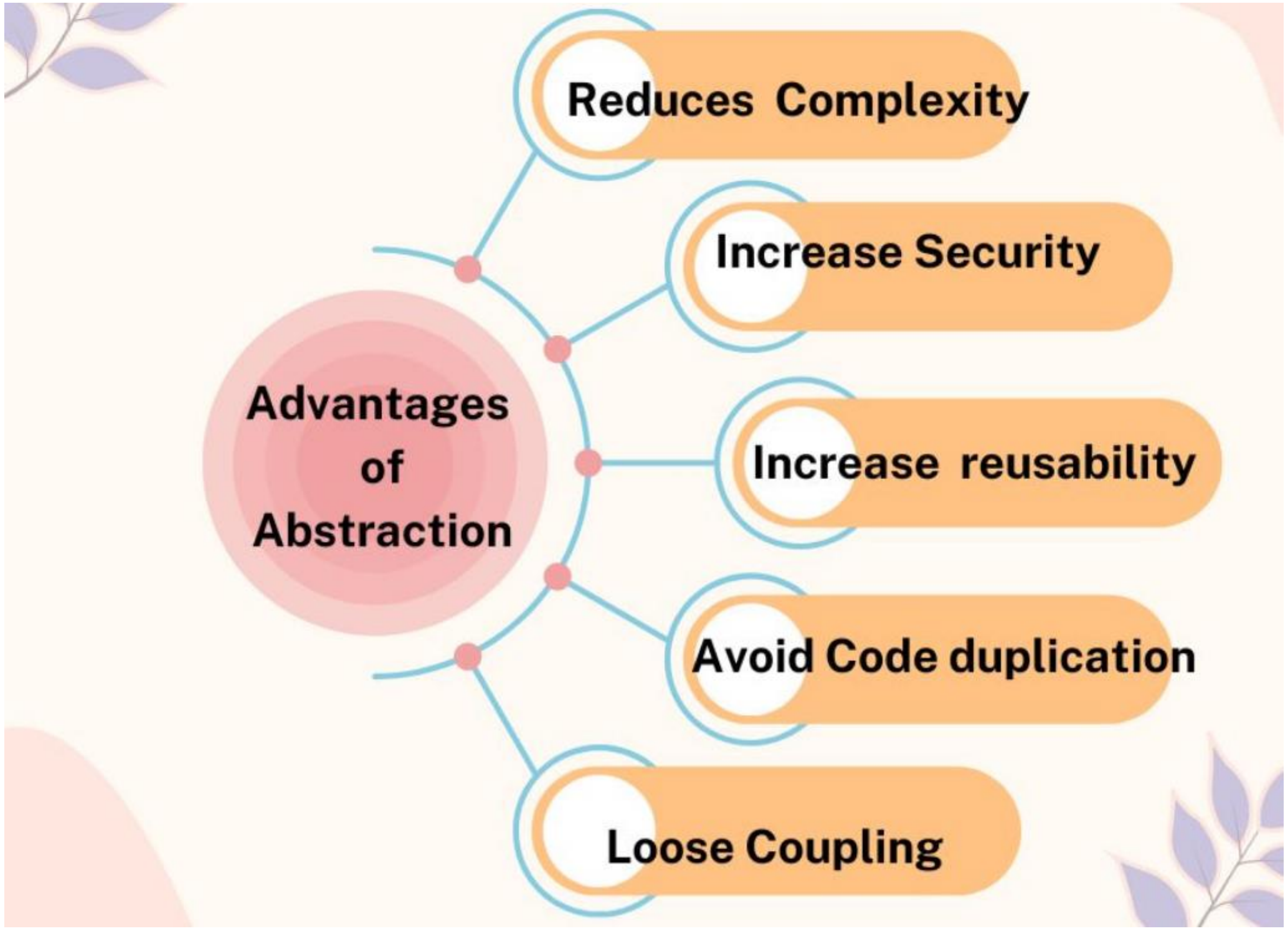
- Abstraction
- Real life example for abstraction
- Abstract Class
- Interfaces

Abstraction

Abstraction

- An **Abstraction** is a process of exposing all the necessary details and hiding the rest.
- In **Java**, **Data Abstraction** is defined as the process of reducing the object to its essence so that only the necessary characteristics are exposed to the users.
- **Abstraction** defines an object in terms of its **properties** (**attributes**), **behavior** (**methods**), and **interfaces** (means of communicating with other objects).

Advantages of Abstraction



Types of Abstraction

Types of Abstraction

1.Data Abstraction:

When the object data is not visible to the outer world, it creates data abstraction. If needed, access to the Objects' data is provided through some methods.

2. Process Abstraction:

We don't have to offer information about all of an object's functions.

Data Abstraction

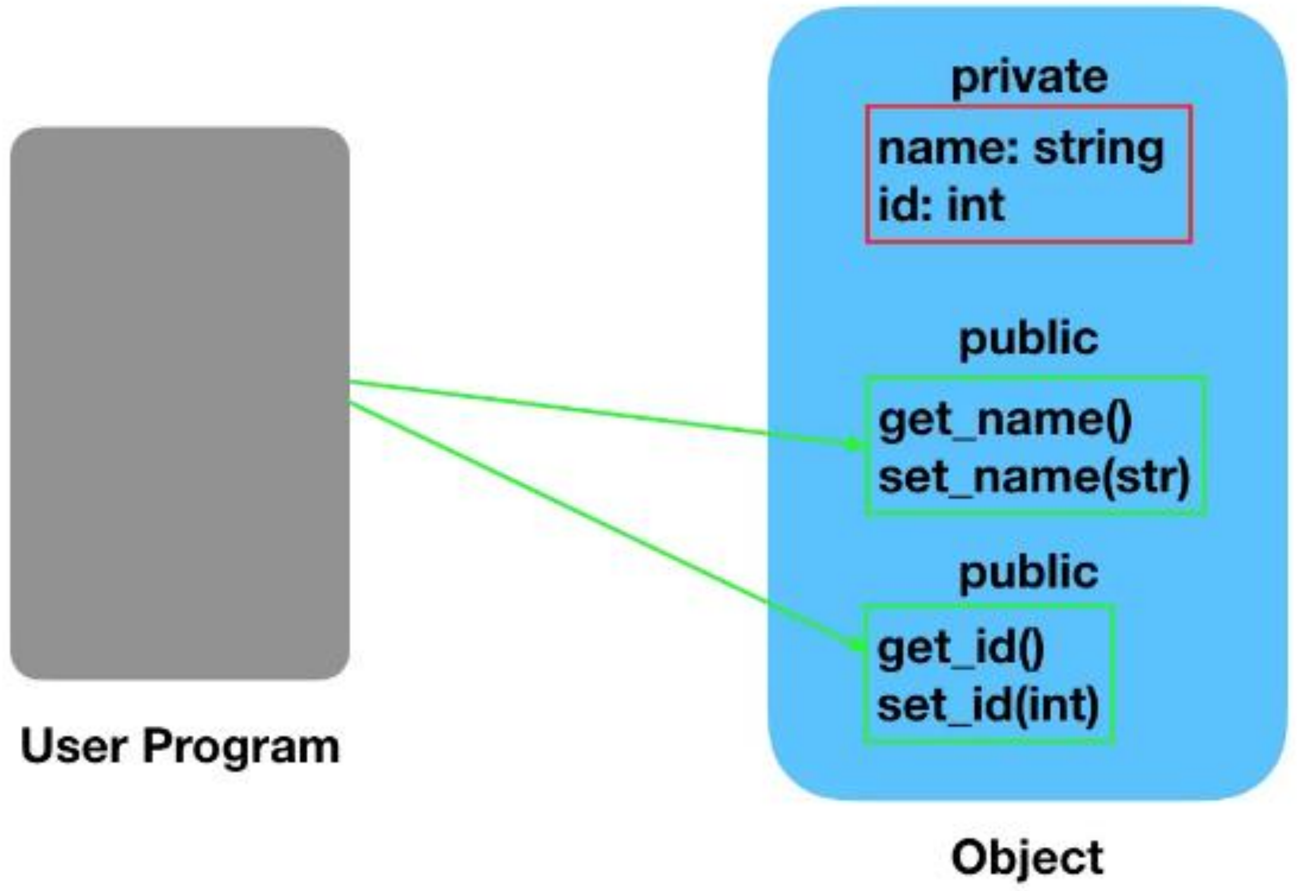
- **Data abstraction** is a technique for creating complicated data types and exposing only the actions that are necessary to interact with the data type, while keeping the implementation details hidden from outside activities.

- The benefit of this approach involves capability of improving the implementation over time e.g. solving performance issues is any.

- The idea is that such changes are not supposed to have any impact on client code since they involve no difference in the abstract behavior.

.

Data Abstraction



Process Abstraction

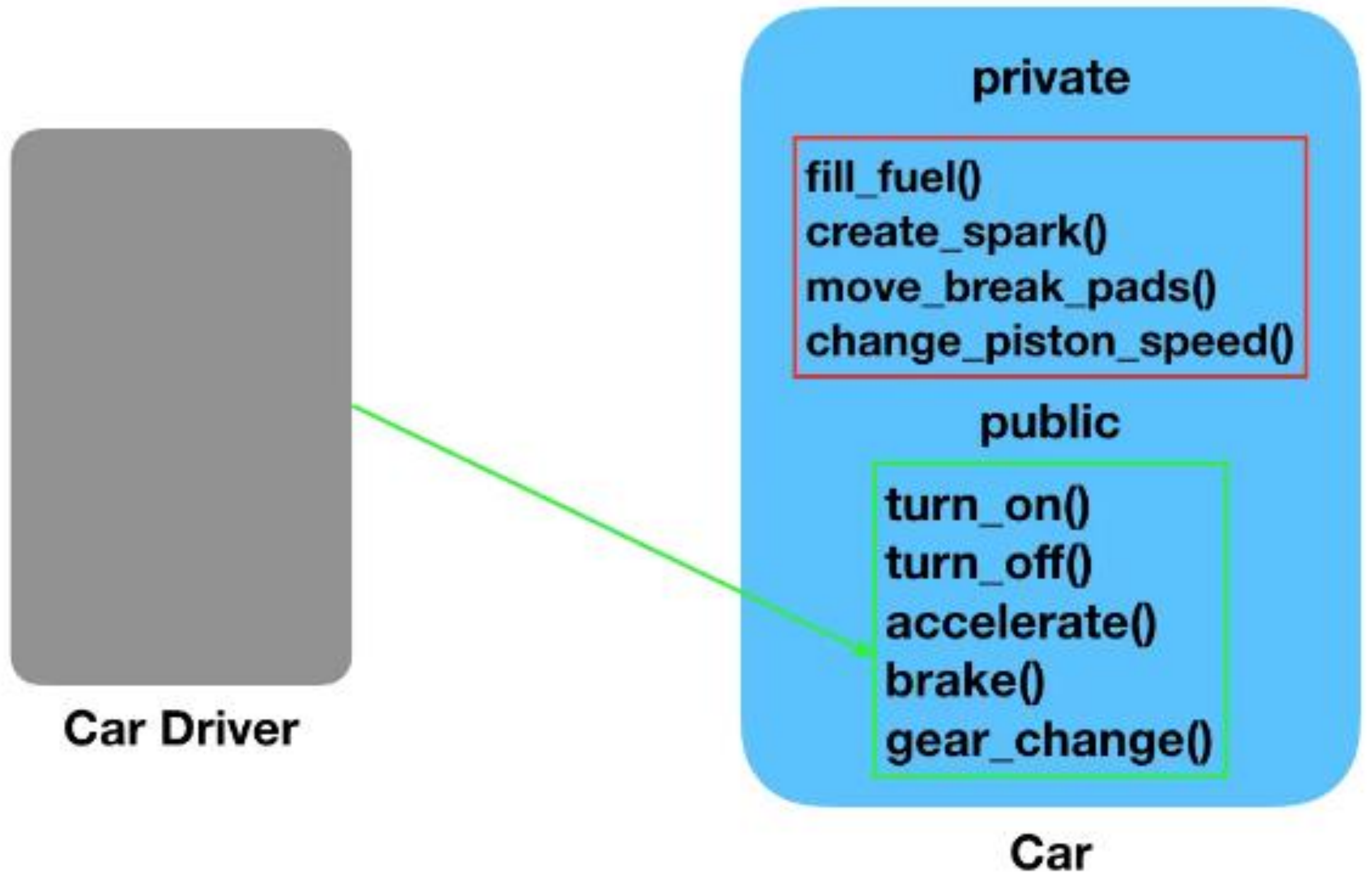
- Process abstraction is achieved by hiding the internal implementation of the many functions involved in a user operation.

- A piece of software is essentially a set of statements written in any programming language. The majority of the time, statements are similar and are repeated in several locations.

- The process of finding all such assertions and exposing them as a unit of work is known as process abstraction. When we build a function to execute any task, we usually use this feature.

.

Process Abstraction

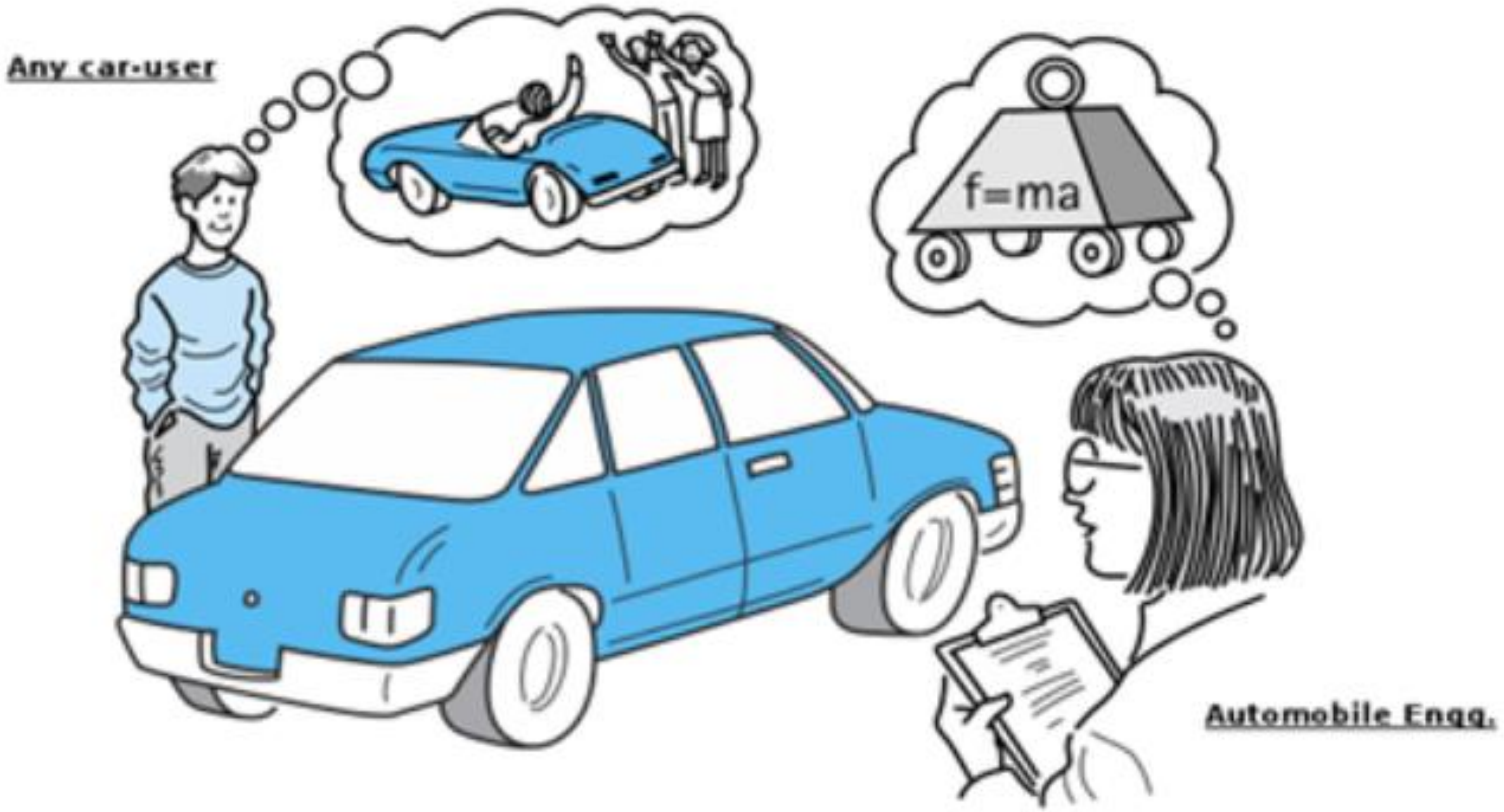


Ways to Implement Abstraction in Java

Real World Example of Abstraction

Using Abstraction
Without Abstraction

Real World Example of Abstraction



Ways to Implement Abstraction in Java

Ways to Implement Abstraction in Java

Implementing Abstraction in Java

1. **Using abstract class:**

   Achieve partial abstraction as concrete methods can also be defined in them**.**

2. Using Interfaces:

Achieve complete abstraction since it dos not consists of any method implementations**.**

Abstract Class

Abstract class is used when you know something and rely on others for what you don't know.

(here it is partial abstraction as some of the things you know and some you don't know.)

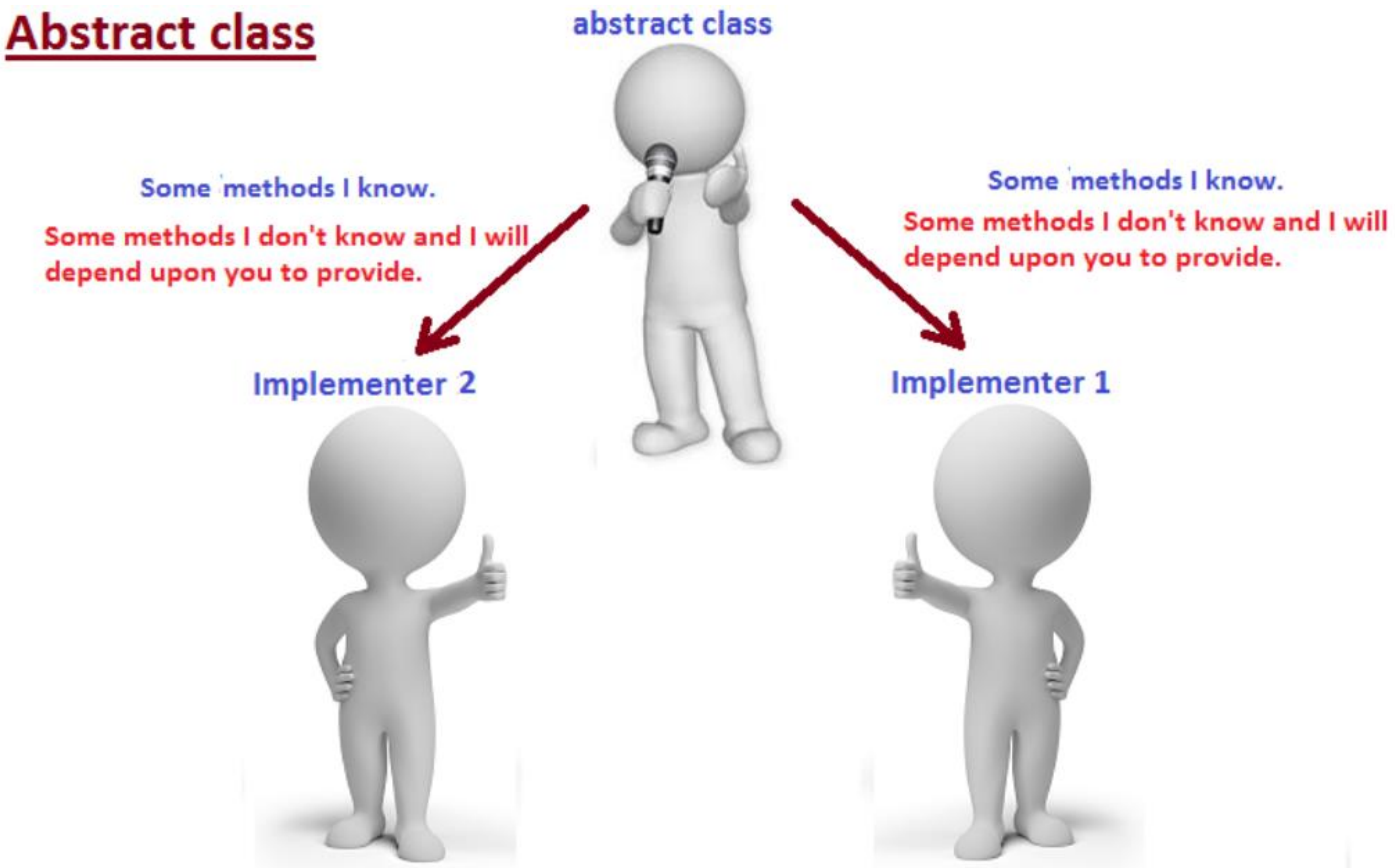
Abstract Class

- An **Abstract** class is a class whose objects can’t be created. An Abstract class is created through the use of the abstract keyword. It is used to represent a concept.
- An abstract class can have abstract methods (methods without body) as well as non-abstract methods or concrete methods (methods with the body). A non-abstract class cannot have abstract methods.
- The class has to be declared as abstract if it contains at least one abstract method.

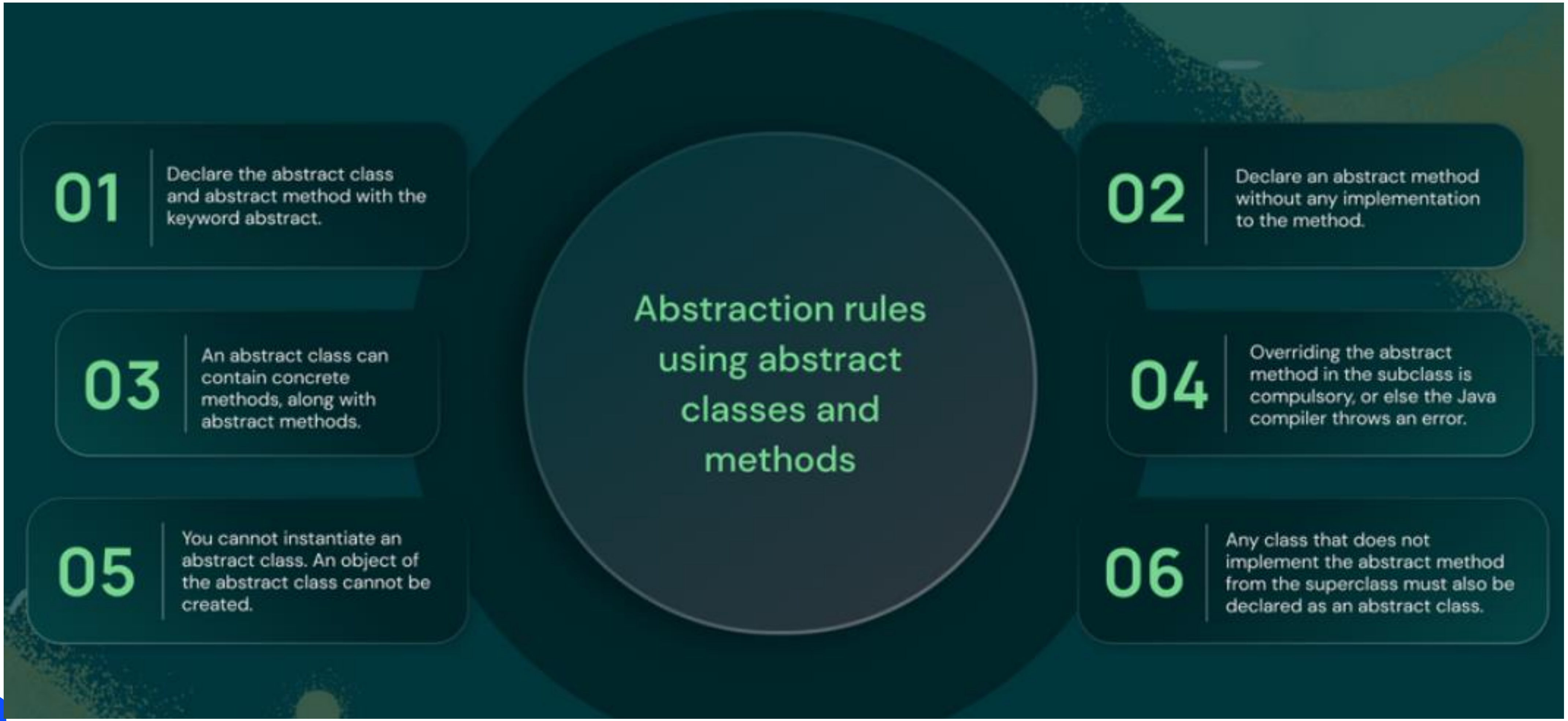
Abstract Class

- An abstract class does not allow you to create objects of its type. In this case, we can only use the objects of its subclass.
- Using an abstract class, we can achieve 0 to 100% abstraction.
- There is always a default constructor in an abstract class, it can also have a parameterized constructor.
- The abstract class can also contain final and static methods.

Abstract Class


Abstract class
abstract class
Some methods I know.
Some methods I don't know and I will depend upon you to provide.
Some methods I know.
Some methods I don't know and I will depend upon you to provide.
Implementer 2
Implementer 1

Rules of Abstract Class and Methods



Abstraction rules using abstract classes and methods

01 Declare the abstract class and abstract method with the keyword abstract.

02 Declare an abstract method without any implementation to the method.

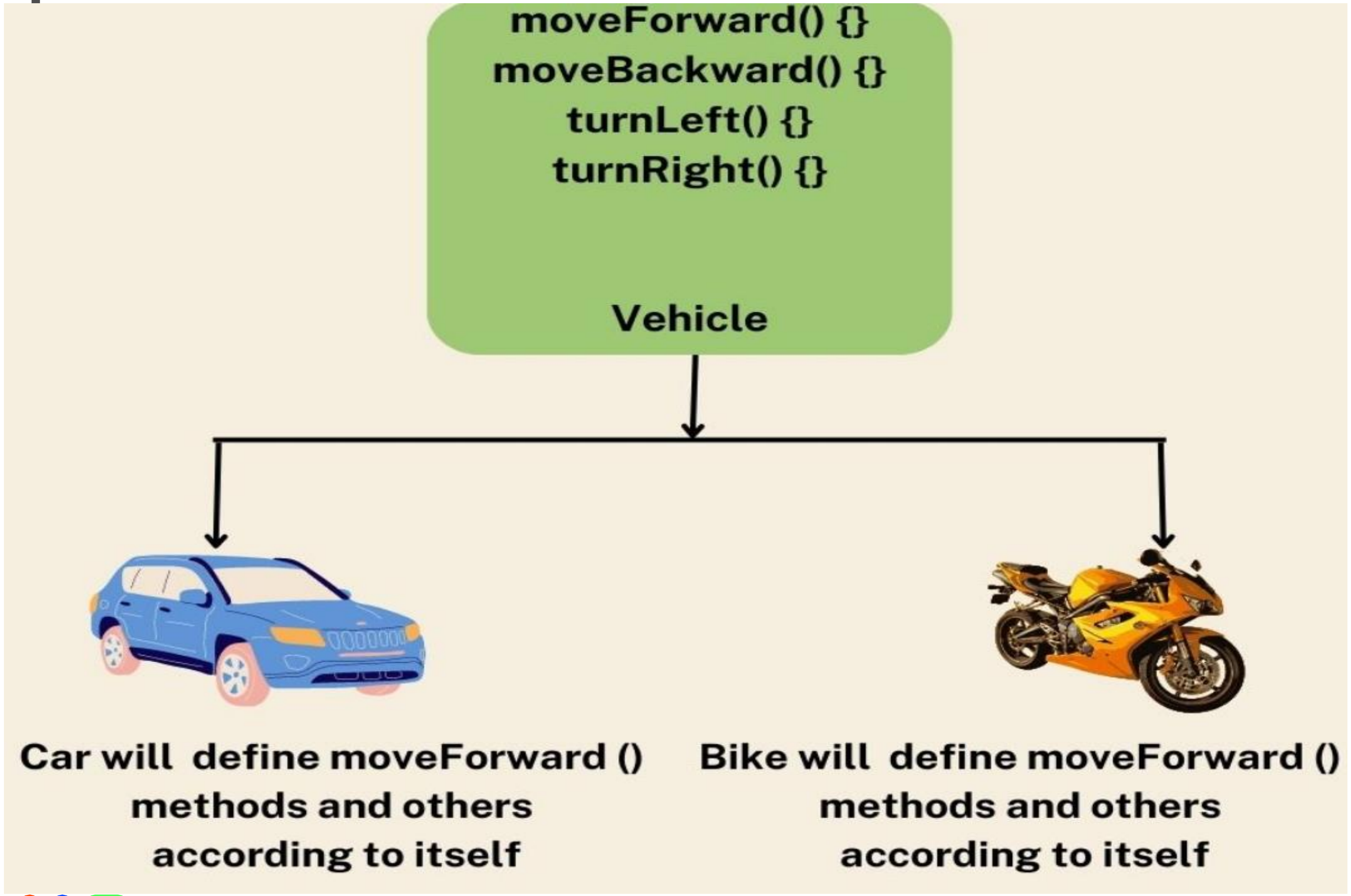
03 An abstract class can contain concrete methods, along with abstract methods.

04 Overriding the abstract method in the subclass is compulsory, or else the Java compiler throws an error.

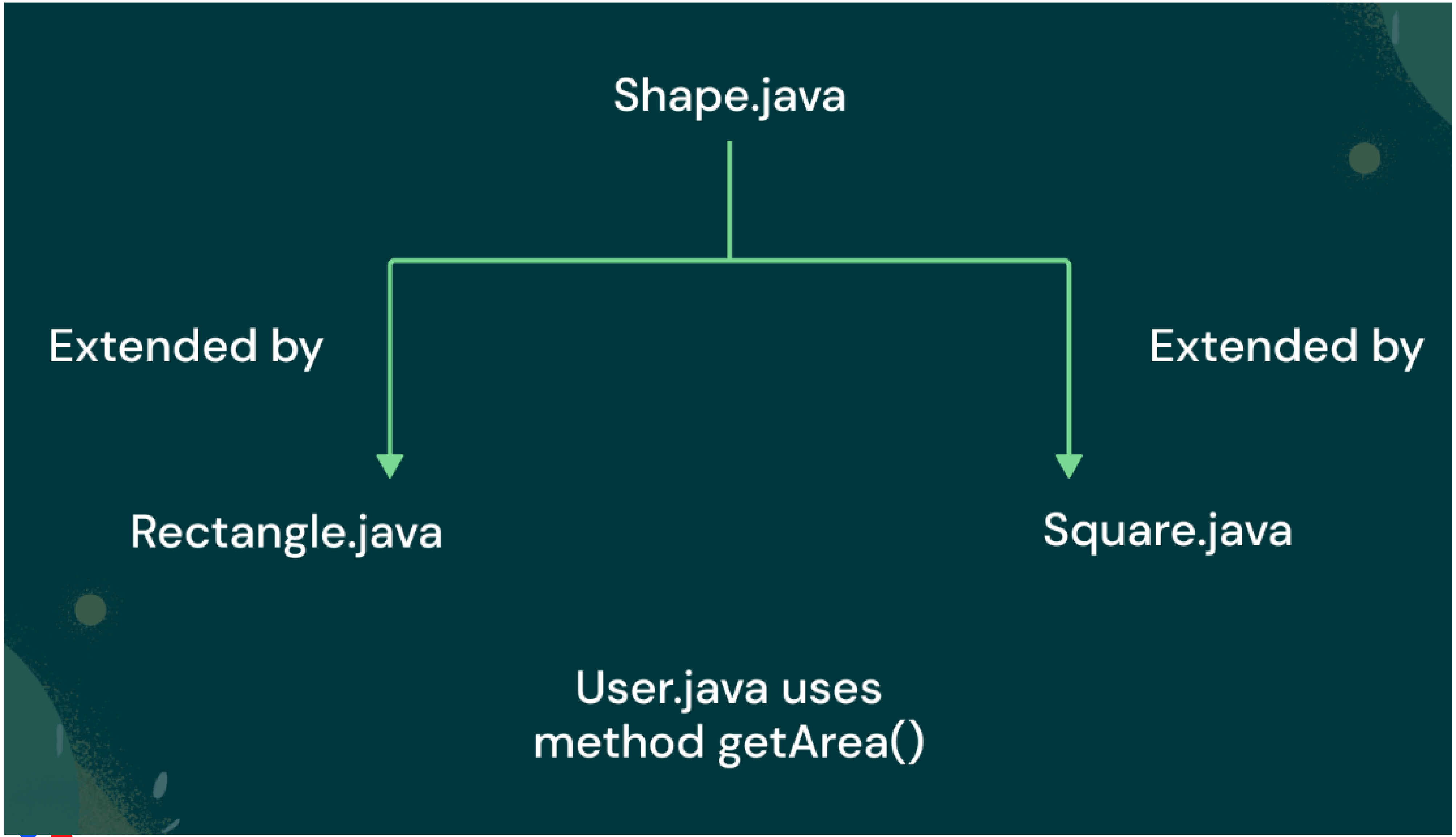
05 You cannot instantiate an abstract class. An object of the abstract class cannot be created.

06 Any class that does not implement the abstract method from the superclass must also be declared as an abstract class.

Example of abstract class-1


moveForward() {}
moveBackward() {}
turnLeft() {}
turnRight() {}
Vehicle
Car will define moveForward () methods and others according to itself
Bike will define moveForward () methods and others according to itself

Example of abstract class-2



Scenario of using abstract class

- Consider we want to start a service like **Bulk SMS sender**, where we take orders from various telecom vendors **like Airtel, France Telecom, Vodafone** etc.
- For this, we don't have to setup our own infrastructure for sending SMS like Mobile towers but we need to take care of government rules like after 9PM, we should not send promotional **SMS**, we should also not send SMS to users registered under **Do Not Disturb(DND)** service etc.
- Remember, we need to take care of government rules for all the countries where we are sending **SMS**

Scenario of using abstract class

- For infrastructure like towers, we will be relying on vendor who is going to give us order.
  Example, In case of,
  Vodafone request us for bulk messaging, in that case we will use **Vodafone** towers to send SMS.
  Airtel request us for bulk messaging, in that case we will use **Airtel** towers to send SMS.
  What our job is to manage **Telecom Regulations** for different countries where we are sending **SMS**.

Scenario of using abstract class

- For infrastructure like towers, we will be relying on vendor who is going to give us order.
  Example, In case of,
  Vodafone request us for bulk messaging, in that case we will use **Vodafone** towers to send SMS.
  Airtel request us for bulk messaging, in that case we will use **Airtel** towers to send SMS.
  What our job is to manage **Telecom Regulations** for different countries where we are sending **SMS**.

Scenario of using abstract class

```
public void eastablishConnectionWithYourTower(){
    //connect using vendor way.
    //we don't know how, candidate for abstract method
}

public void sendSMS(){
    eastablishConnectionWithYourTower();
    checkForDND();
    checkForTelecomRules();
    //sending SMS to numbers...numbers.
    destroyConnectionWithYourTower()
}

public void destroyConnectionWithYourTower(){
    //disconnect using vendor way.
    //we don't know how, candidate for abstract method
}

public void checkForDND(){
    //check for number present in DND.
}

public void checkForTelecomRules(){
    //Check for telecom rules.
}

```

Scenario of using abstract class

1. Out of above 5 methods,
   Methods we know is "**sendSMS**()", "**checkForDND**()", "**checkForTelecomRules**()".
2. **Methods we don't know** is "**eastablishConnectionWithYourTower**()", "**destroyConnectionWithYourTower**()".

- We know how to check government rules for sending SMS .

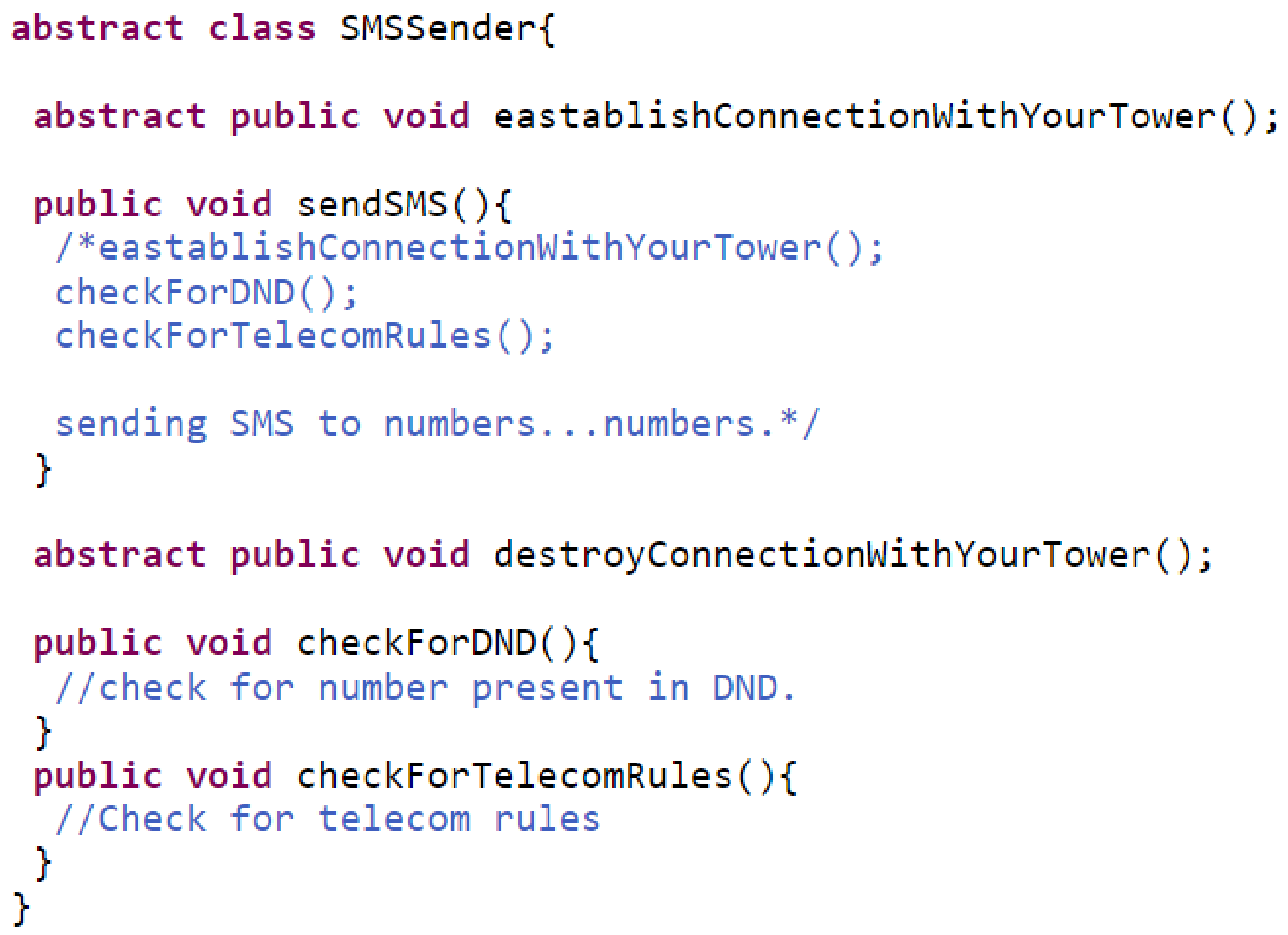
- We don't how to establish connection with tower and how to destroy connection with tower because this is purely customer specific, airtel has its own way, **vodafone** has its own way etc.

Abstract Class or Interface?

In this case, Abstract class will be helpful, because we know partial things like "checkForDND()", "checkForTelecomRules()" for sending sms to users.

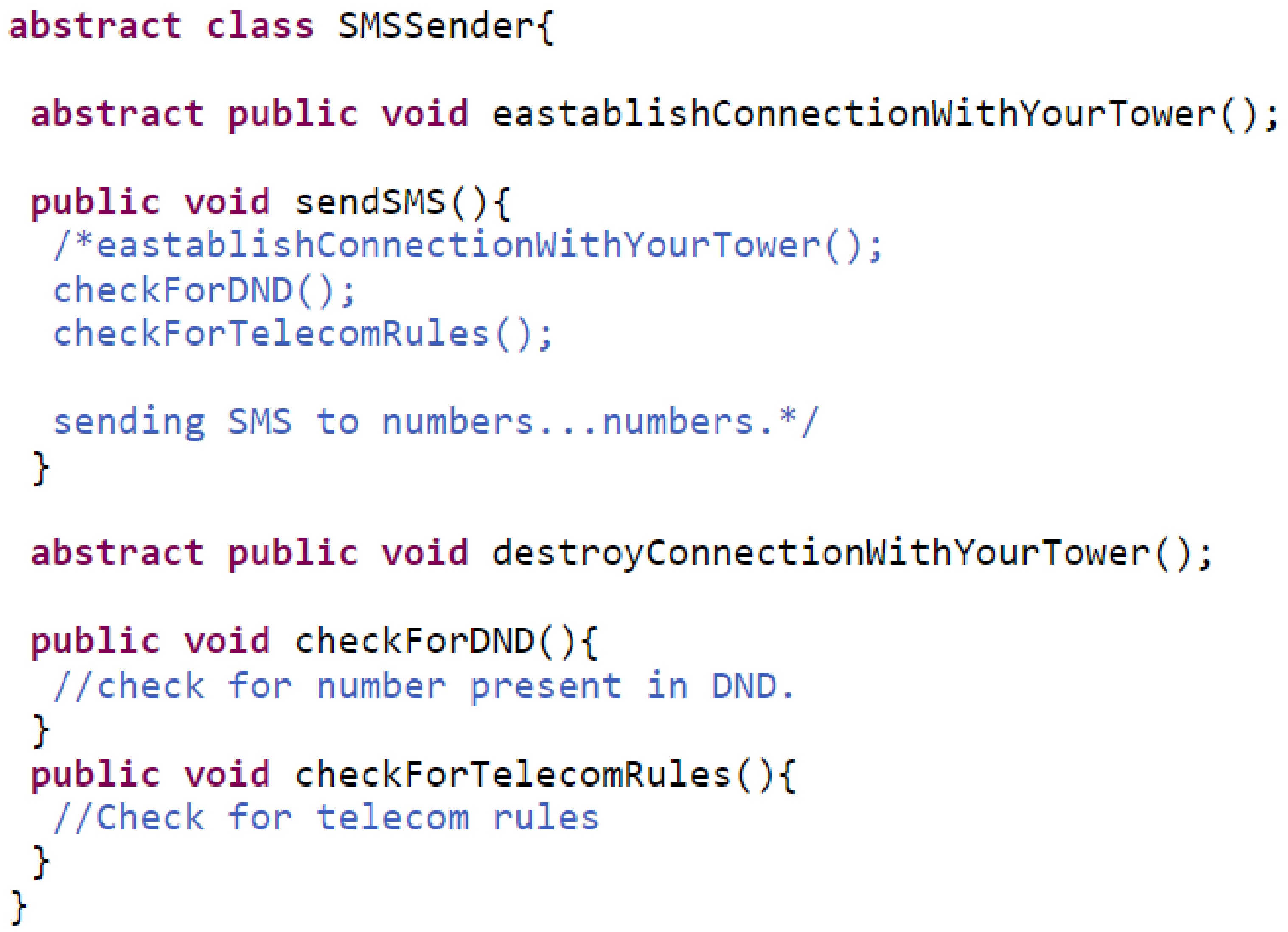
But we don't know how to eastablishConnectionWithTower() and destroyConnectionWithTower() and need to depend on vendor specific way to connect and destroy connection from their towers.

Implementation



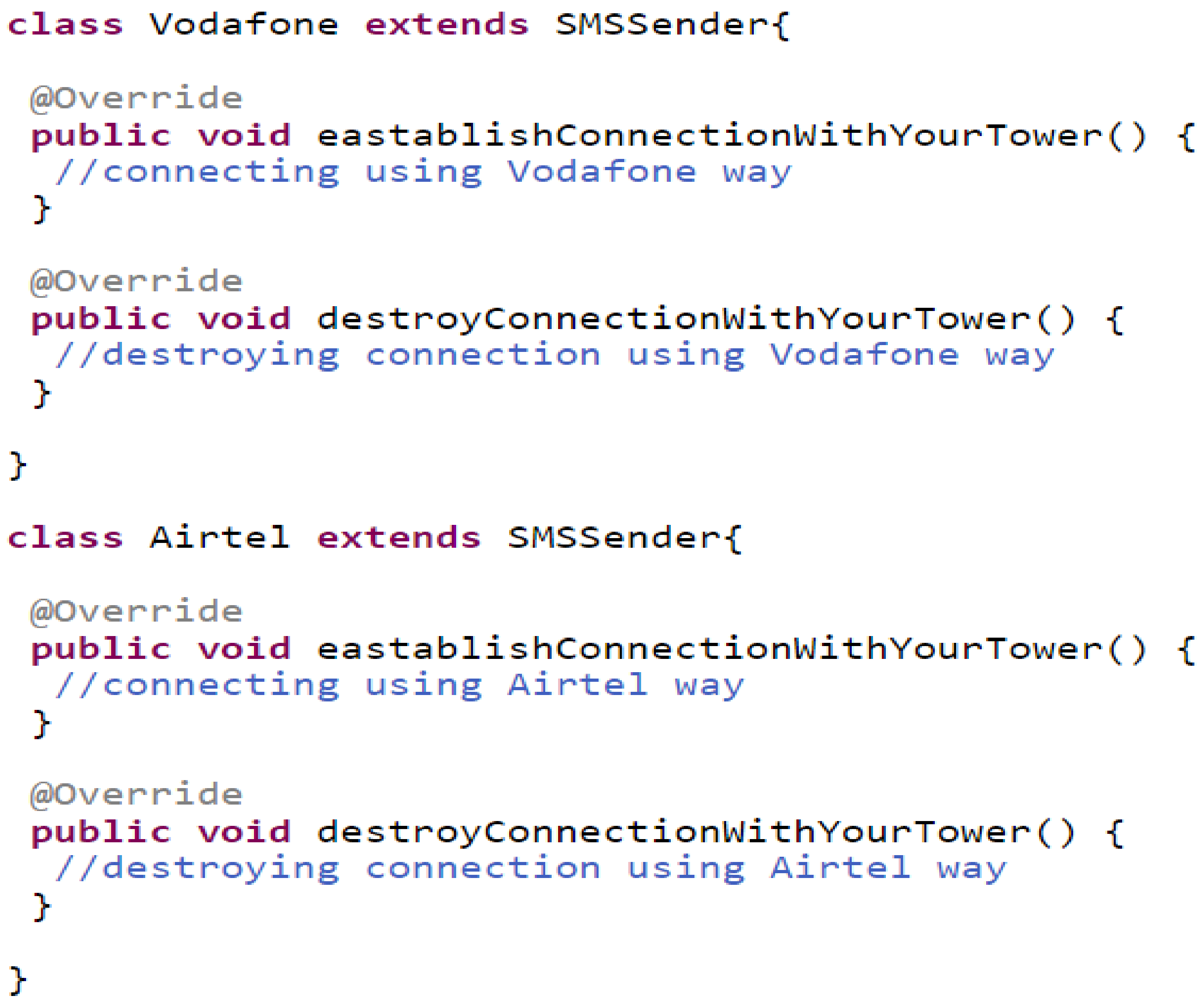
```
abstract class SMSSender{

 abstract public void eastablishConnectionWithYourTower();

 public void sendSMS(){
  /*eastablishConnectionWithYourTower();
  checkForDND();
  checkForTelecomRules();

  sending SMS to numbers...numbers.*/
 }

 abstract public void destroyConnectionWithYourTower();

 public void checkForDND(){
  //check for number present in DND.
 }
 public void checkForTelecomRules(){
  //Check for telecom rules
 }
}
```

Implementation



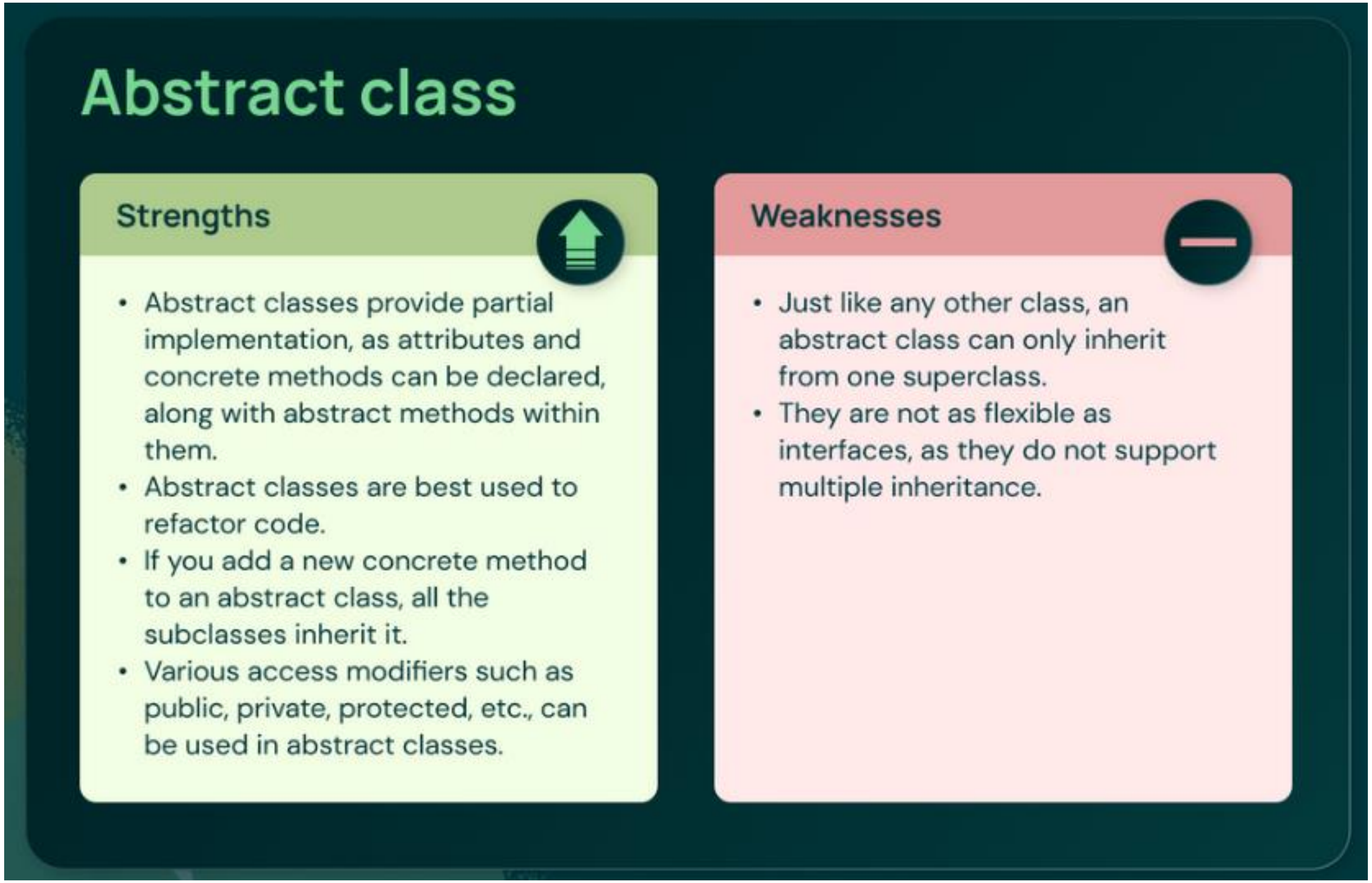
```
abstract class SMSSender{

 abstract public void eastablishConnectionWithYourTower();

 public void sendSMS(){
  /*eastablishConnectionWithYourTower();
  checkForDND();
  checkForTelecomRules();

  sending SMS to numbers...numbers.*/
 }

 abstract public void destroyConnectionWithYourTower();

 public void checkForDND(){
  //check for number present in DND.
 }
 public void checkForTelecomRules(){
  //Check for telecom rules
 }
}
```

Implementation



```
class Vodafone extends SMSSender{

 @Override
 public void eastablishConnectionWithYourTower() {
  //connecting using Vodafone way
 }

 @Override
 public void destroyConnectionWithYourTower() {
  //destroying connection using Vodafone way
 }

}

class Airtel extends SMSSender{

 @Override
 public void eastablishConnectionWithYourTower() {
  //connecting using Airtel way
 }

 @Override
 public void destroyConnectionWithYourTower() {
  //destroying connection using Airtel way
 }

}
```

Advantages/Disadvantages of Abstract Class



Abstract class

Strengths

- Abstract classes provide partial implementation, as attributes and concrete methods can be declared, along with abstract methods within them.
- Abstract classes are best used to refactor code.
- If you add a new concrete method to an abstract class, all the subclasses inherit it.
- Various access modifiers such as public, private, protected, etc., can be used in abstract classes.

Weaknesses

- Just like any other class, an abstract class can only inherit from one superclass.
- They are not as flexible as interfaces, as they do not support multiple inheritance.

Demo on Abstract Class

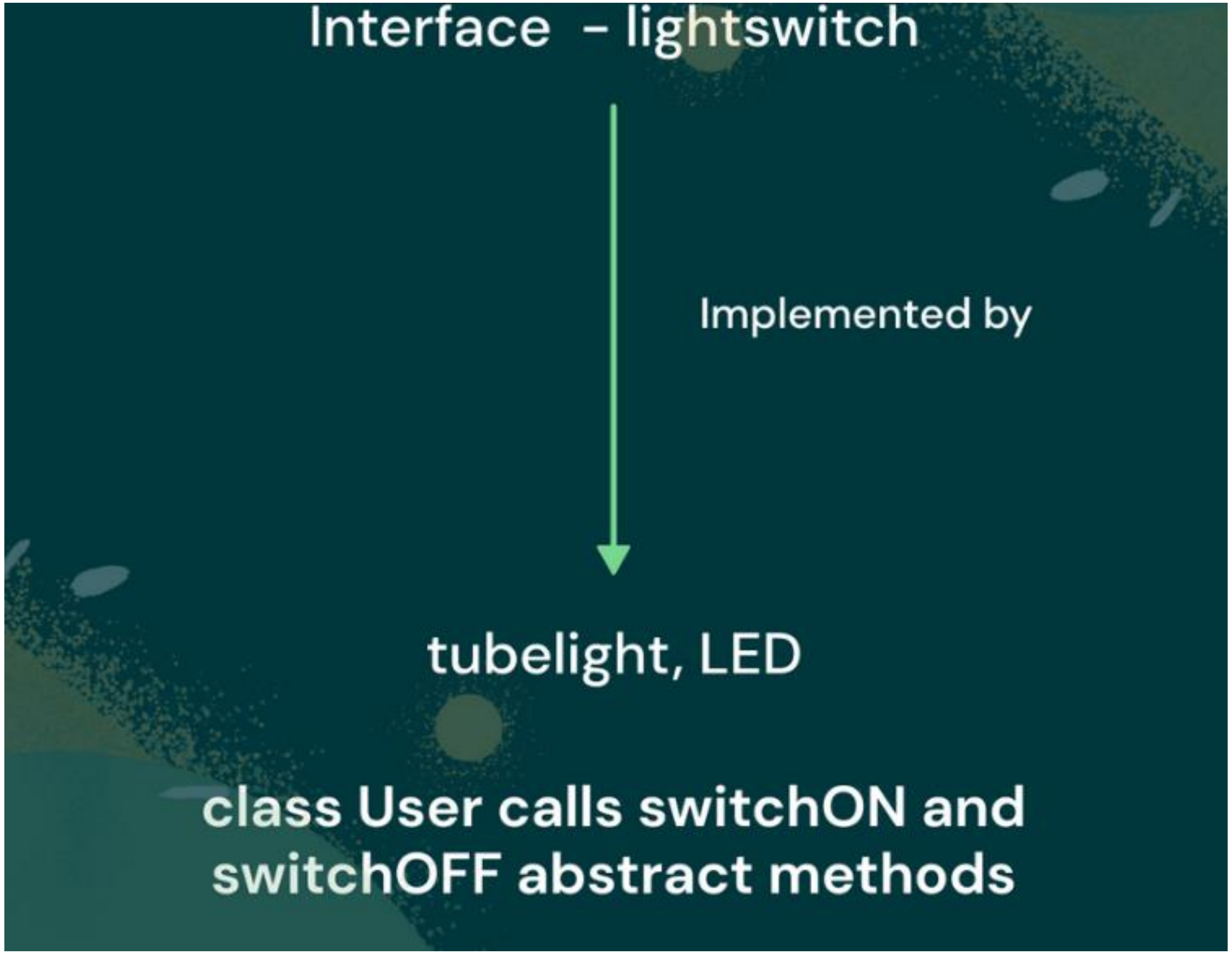
Interfaces

Interface

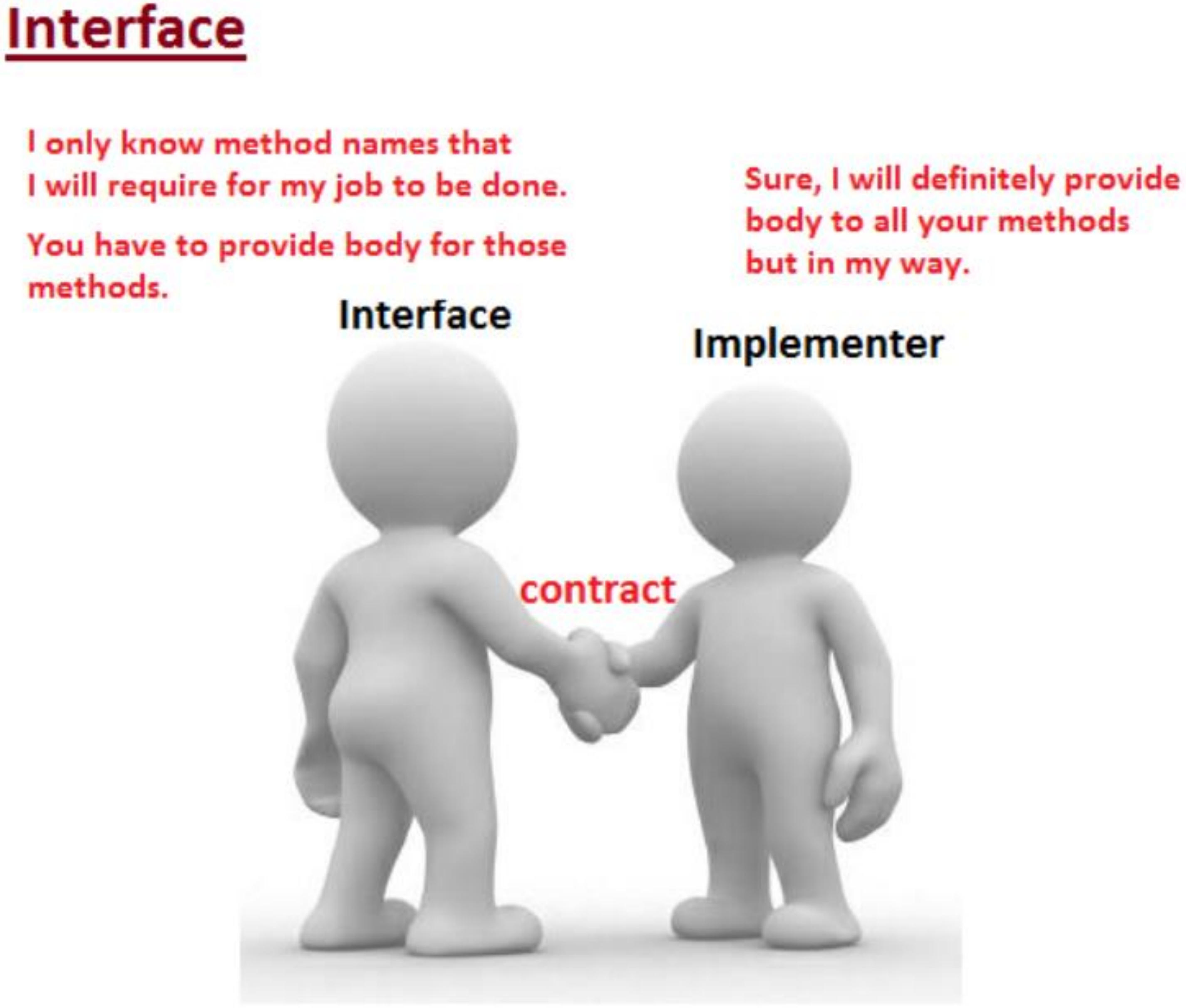
Interface is used when you want to define a contract and you don't know anything about implementation.

(here it is total abstraction as you don't know anything.)

Interface


Interface – lightswitch
Implemented by
tubelight, LED
class User calls switchON and switchOFF abstract methods

Interface


Interface
I only know method names that I will require for my job to be done.
You have to provide body for those methods.
Sure, I will definitely provide body to all your methods but in my way.
Interface
Implementer
contract

When to Use Interface

1. Consider we want to start a service like "**makemytrip.com**" or "**expedia.com**", where we are responsible for **displaying the flights** from **various flight service company** and place an order from customer.
   Lets keep our service as simple as,
   Displaying flights available from **vendors** like "**Airasia**", "**British Airways**" and "Emirates".
2. Place and order for seat to respective vendor.

Interface or Abstract Class

In this application, **we don't own any flight**. we are just a **middle man/aggregator** and our task is to first enquire “**Airasia"**, then enquire “**British Airways**" and “**Emirates**" about the list of flights available and later if customer opts for booking then inform the respective flight vendor to do booking.

For this, first we need to tell “**Airasia**", “**British Airways"** and “**Emirates**" to give us list of flights, internally how they are giving the list that we are not concerned.

Interface or Abstract Class

In this application, **we don't own any flight**. we are just a **middle man/aggregator** and our task is to first enquire “**Airasia**", then enquire “**British Airways**" and “**Emirates**" about the list of flights available and later if customer opts for booking then inform the respective flight vendor to do booking.

For this, first we need to tell “**Airasia**", “**British Airways**" and “**Emirates**" to give us list of flights, internally how they are giving the list that we are not concerned.

Interface or Abstract Class

This means we only care about the method "**getAllAvailableFlights**()"

"**getAllAvailableFlights**()" from “**Airasia**" may have used **SOAP** service to return list of flights.
"**getAllAvailableFlights**()" from “**British Airways**" may have used **REST** service to return list of flights.
"**getAllAvailableFlights**()" from “**Emirates**" may have used **CORBA** service to return list of flights.

We don't care how it is internally implemented and what we care is the **contract** method "*getAllAvailableFlights*" that all the flight vendor should provide and **return list of flights.**

Interface or Abstract Class

Similarly, for booking we only care for method "**booking**()" that all vendors should have, internally how this vendors are doing booking that we are not concerned.

So we can say that we know the contract that irrespective of who the Flight vendor is, we need "**getAllAvailableFlights**()" and "**booking**()" method from them to run our aggregator service.

Interface or Abstract Class

In this situation, Interface is useful because we are not aware of the implementation of all the 2 methods required, and what we know is the contract methods that vendor(implementer) should provide. so due to this total abstraction and for defining the contract, interface is useful in this place.

Coding Interface

FlightOpeartions.java(Contract)

```java
interface FlightOpeartions{
 void getAllAvailableFlights();
 void booking(BookingObject bookingObj);
}
```

BookingObject.java

```java
class BookingObject{}
```

Coding Implementation class1

BritishAirways.java (Vendor 1)

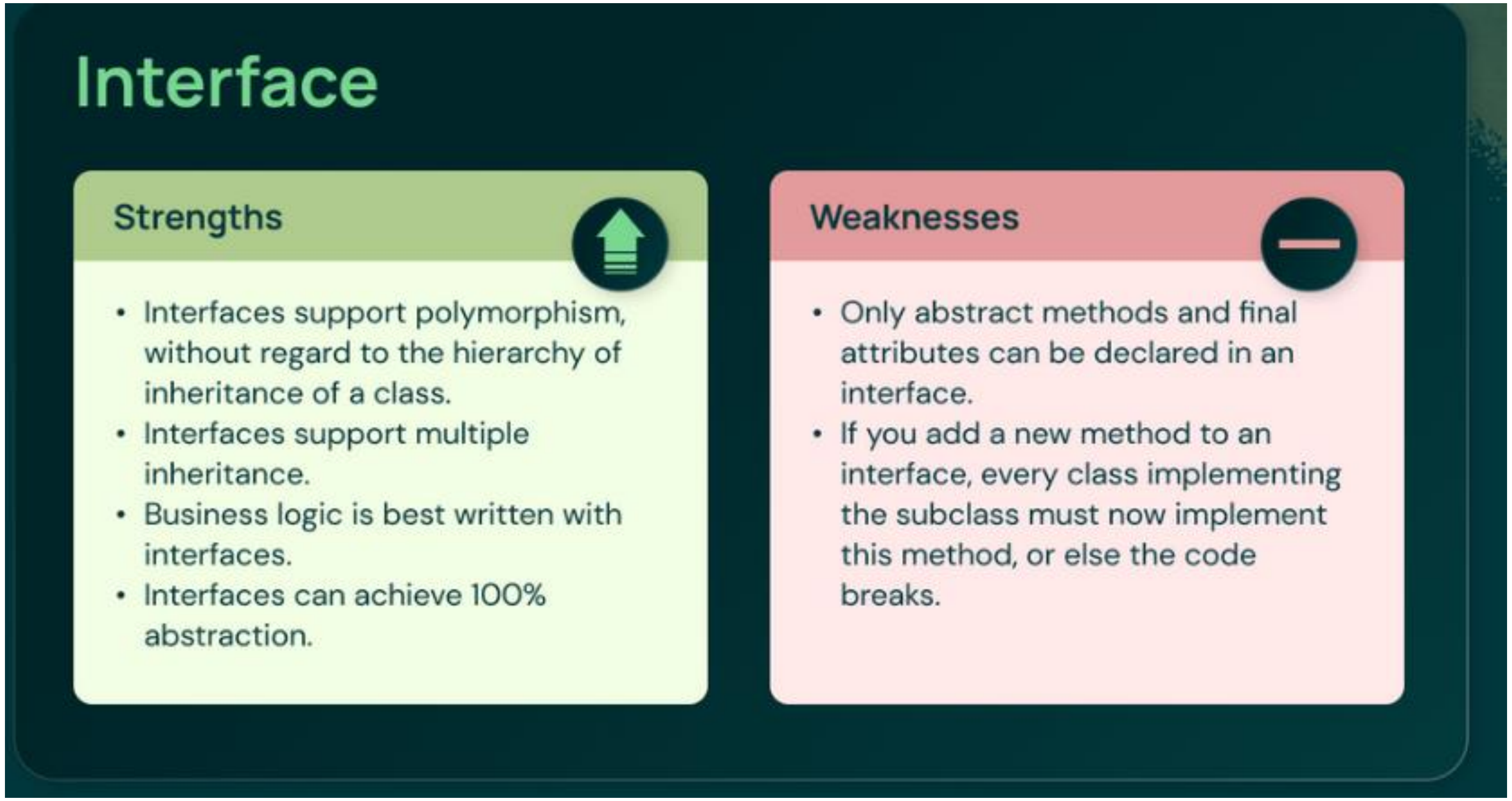
```java
class BritishAirways implements FlightOpeartions{

 public void getAllAvailableFlights(){
           //get british airways flights in the way
           //they told us to fetch flight details.
 }

 public void booking(BookingObject flightDetails){
          //place booking order in a way British airways
          //told us to place order for seat.
 }

}
```

Coding Implementation class2

Emirates.java (Vendor 2)

```java
class Emirates implements FlightOpeartions{

 public void getAllAvailableFlights(){
        //get Emirates flights in the way
        //they told us to fetch flight details.
 }

 public void booking(BookingObject flightDetails){
        //place booking order in a way Emirates airways
        //told us to place order for seat.
 }
}
```

Advantages/Disadvantages of Interfaces



Interface

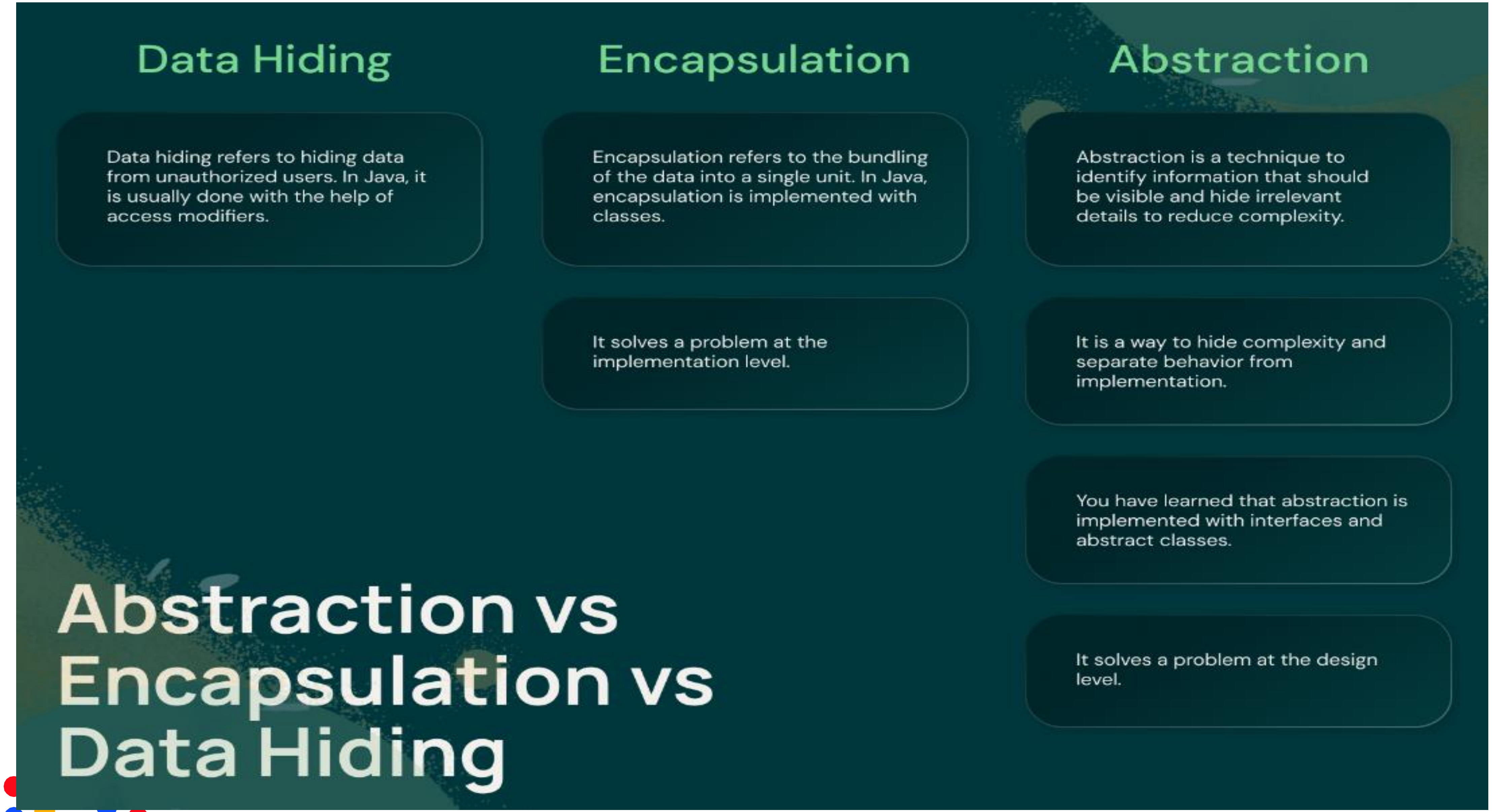
Strengths

- Interfaces support polymorphism, without regard to the hierarchy of inheritance of a class.
- Interfaces support multiple inheritance.
- Business logic is best written with interfaces.
- Interfaces can achieve 100% abstraction.

Weaknesses

- Only abstract methods and final attributes can be declared in an interface.
- If you add a new method to an interface, every class implementing the subclass must now implement this method, or else the code breaks.

Summary



Data Hiding

Data hiding refers to hiding data from unauthorized users. In Java, it is usually done with the help of access modifiers.

Encapsulation

Encapsulation refers to the bundling of the data into a single unit. In Java, encapsulation is implemented with classes.

It solves a problem at the implementation level.

Abstraction

Abstraction is a technique to identify information that should be visible and hide irrelevant details to reduce complexity.

It is a way to hide complexity and separate behavior from implementation.

You have learned that abstraction is implemented with interfaces and abstract classes.

It solves a problem at the design level.

Abstraction vs Encapsulation vs Data Hiding

Questions

Learning material references

Website

Interface in Java - Javatpoint
Interfaces in Java - GeeksforGeeks
What Is an Interface? (The Java™ Tutorials > Learning the Java Language > Object-Oriented Programming Concepts) (oracle.com)
Abstract Class in Java – Javatpoint
Java Abstraction (w3schools.com)
Abstract Methods and Classes (The Java™ Tutorials > Learning the Java Language > Interfaces and Inheritance) (oracle.com)

HEXAWARE

Passionate Employees

Delighted Customers

Thank you

www.hexaware.com