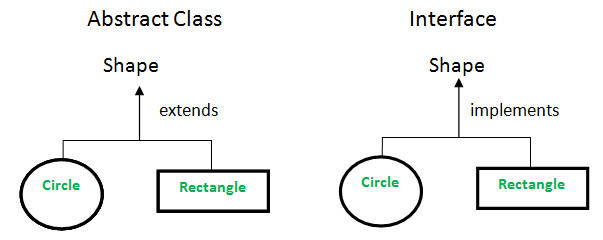
**Abstract vs interfaces**

Differences:

| Abstract | Interfaces |
| --- | --- |
| Has abstract and non abstract methods | Only abstract methods |
| No multiple inheritance | Supports multiple inheritance |
| Can provide implementation of interface | Cannot provide implementation of inheritance |
| Can extends another class and implement multiple interfaces | Can extend another interface only |
| Can have class members like private, protected | Members are public by default |
| Example:  public abstract class Shape{  public abstract void draw();  } | Example:  public interface Drawable{  void draw();  } |
|  |  |
|  |  |



When to use interfaces?

1. Useful for decoupling and polymorphism
2. It is better to use interface when various implementations share only method signature. Polymorphic hierarchy of value types.
3. If the problem needs to be solved using multiple inheritances and is composed of different class hierarchies
4. When application functionalities have to be defined as a contract, but not concerned about who implements the behavior. i.e., third-party vendors need to implement it fully

When to use abstract classes?

1. To avoid independence
2. It should be used when various implementations of the same kind share a common behavior.
3. While classes that extend abstract classes have several common fields or methods
4. When trying to use the inheritance concept in code (share code among many related classes), by providing common base class methods that the subclasses override

**SOLID Principles**

Facts:

* Class design patterns.
* Purpose: "To create understandable, readable, and testable code that many developers can collaboratively work on."
* maintainable, understandable, and flexible software.
* Principles:
  1. Single Responsibility Principle:
     + A class should have one responsibility. A class should have only one reason to change.
     + When there are multiple functions for multiple responsibilities, we need to divide the class into multiple classes. (Printer example)

**Bad example:**

public class Vehicle {

public void printDetails() {}

public double calculateValue() {}

public void addVehicleToDB() {}

}

**How to solve? Separate methods to classes**

public class VehiclePrinter {

public void printDetailse(){}

}



public class VehicleDB {

public void addVehicleToDB(){}

}



* 1. Open-Closed Principle:
     + Classes should be open to extension but closed to modification.
     + Purpose is to not break the existing and working code.
     + We should be able to add new functionality without touching the existing code.
     + Usually done with interfaces and abstract classes.

**Example**:

What is already done?

public class VehicleCalculations {

public double calculateValue(Vehicle v) {

if (v instanceof Car) {

return v.getValue() \* 0.8;

if (v instanceof Bike) {

return v.getValue() \* 0.5;

}

}



Now we want to add truck. Instead of adding it here as a new instance the following approach should be implemented:

public class Vehicle {

public double calculateValue() {...}

}

public class Car extends Vehicle {

public double calculateValue() {

return this.getValue() \* 0.8;

}

public class Truck extends Vehicle{

public double calculateValue() {

return this.getValue() \* 0.9;

}



* 1. Liskov Substitution Principle:
     + If class *A* is a subtype of class *B*, we should be able to replace *B* with *A* without disrupting the behavior of our program.
     + Objects of a superclass should be replaceable with objects of its subclasses without breaking the system.
     + The child class extends the behavior but never narrows it down.

**Bad Example**:

public class Rectangle {

private double height;

private double width;

public void setHeight(double h) { height = h; }

public void setWidth(double w) { width = w; }

...

}

public class Square extends Rectangle {

public void setHeight(double h) {

super.setHeight(h);

super.setWidth(h);

}

public void setWidth(double w) {

super.setHeight(w);

super.setWidth(w);

}

}



What is wrong with the code?

Behaviour of height and width is change in the child class.

* 1. Interface Segregation Principle:
     + Do not force any client to implement an interface that is irrelevant to them.
     + To fix this, ISP proposes that the interfaces be broken down into multiple, small cohesive interfaces so that no class is forced to implement any interface, and therefore methods, that it does not need.

Example:

public interface BearKeeper {

void washTheBear();

void feedTheBear();

void petTheBear();

}



If a bear carer only washes and feeds the bear we cannot implement petTheBear method if we implement BearKeeper. So the following approach should be followed:

**Divide it to:**

**public interface BearCleaner {**

**void washTheBear();**

**}**

****

****publicinterface **BearFeeder {**

void **feedTheBear();**

**}**

****

****publicinterface **BearPetter {**

void **petTheBear();**

**}**

****

**And use the necessary one only**

public class BearCarer implements BearCleaner, BearFeeder {

public void washTheBear() {

//I think we missed a spot...

}

public void feedTheBear() {

//Tuna Tuesdays...

}

}



* 1. Dependency Inversion Principle:
     + High-level modules should not depend on low-level modules, both should depend on abstractions.

Example:

public class Car {

private Engine engine; //Engine should be an interface so any class that implements engine can be used here.

public Car(Engine e) {

engine = e;

}

public void start() {

engine.start();

}

}





public interface Engine {

public void start();

}

