# Design Principles

B.Tech. (IT), Sem-6,
Applied Design Patterns and Application Frameworks (ADPAF)

Dharmsinh Desai University
Prof. H B Prajapati

## Topics

- Interfaces
- Abstract classes
- Design choice: Interfaces vs Abstract class
- Composition over inheritance

## Interfaces

- Interface a common boundary or interconnection between two entities (could be human beings, natural language, systems, concepts, machines, etc.)
  - A natural language such as English is an interface between two humans that allows them to exchange their views.
- In Java, an interface is a set of abstract methods that defines a protocol
  - It is a contract for conduct between interface user and interface implementer
- Classes that implement an interface must implement the methods specified in the interface.

## ?

## Example of Interface in real world

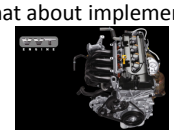
- Want to learn driving a manual gear car
  - We learn driving only one (say Alto 800)
- While learning driving, we learn driving using interface
  - How to use steering
  - How to apply break in a car
  - How to change gear using Clutch in a car
  - How to accelerate a car
  - etc

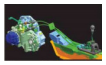
## Example of Interface in real world

- Then, can drive any other similar manual gear car

## Example of Interface in real world

- What about implementation?





- We do not worry. We do not need to learn how implementation is done

## Example: java.lang.Comparable interface

```
public interface Comparable{
    public int compareTo(Object o); //It has no implementation
    // Intent is to compare this object with the specified object
    // The return type is integer. Returns negative,
    // zero or a positive value when this object is less than,
    // equal to, or greater than the specified object
}
```

- The algorithms (i.e., clients of the interface) are completely ignorant about how the compareTo()method is implemented.
- Advantage: when a method takes an interface as an argument, you can pass any object that implements that interface (due to runtime polymorphism).

## Declaring and using interfaces

```
interface Rollable {
    void roll(float degree);
}

class Circle implements Rollable {
     public void roll(float degree) {
     /* implement rolling functionality here */
     }
}
```

- The Circle class implements all the methods of Rollable interface

## Declaring and using interfaces

- If you are implementing an interface in an abstract class, the abstract class does not need to define the method.

```
interface Rollable {
    void roll(float degree);
}
abstract class CircularShape implements Rollable extends Shape { }
```

## Interfaces can have multiple methods

```
public interface Iterator<E> {
     boolean hasNext();
     E next();
     void remove();
}
```

- This interface is meant for traversing a collection.

- A class can implement multiple interfaces at the same time

```
class Circle extends CircularShape implements Cloneable,
    Serializable {
/* definition of methods such as clone here */
}
```

## Important points about interfaces (Prior Java 8)

- An interface cannot be instantiated.
- An interface can extend another interface.
- Interfaces cannot contain instance variables.
  - If you declare a data member in an interface, it should be initialized, and
  - all such data members are implicitly treated as "public static final" members.
- An interface cannot declare static methods. It can only declare instance methods.
- You cannot declare members as protected or private. Only public access is allowed for members of an interface.
- All methods declared in an interface are implicitly considered to be abstract. If you want, you can explicitly use the abstract qualifier for the method.
- You can only declare (and not define) methods in an interface.

## Important points about interfaces (Prior Java 8)

- An interface can be declared with empty body (i.e., an interface without any members.
  - Such interfaces are known as tagging interfaces (or marker interfaces).
  - Such interfaces are useful for defining a common parent, so that runtime polymorphism can be used.
  - For example, java.util defines the interface EventListener without a body.

  public interface Serializable{}
- An interface can be declared within another interface or class; such interfaces are known as nested interfaces.
- Unlike top-level interfaces that can have only public or default access, a nested interface can be declared as public, protected, or private.

## Choosing Between an Abstract Class and an Interface

- If you are identifying a base class that abstracts common functionality from a set of related classes, you should use an abstract class.
- If you are providing common method(s) or protocol(s) that can be implemented even by unrelated classes, this is best done with an interface.
- If you want to capture the similarities among the classes (even unrelated) without forcing a class relationship, you should use interfaces.
- On the other hand, if there exists an is-a relationship between the classes and the new entity, you should declare the new entity as an abstract class.

## Example: Choosing Between an Abstract Class and an Interface

- You can have Shape as an abstract base class for all shapes (like Circle, Square, etc.); this is an example of an is-a relationship.
- For example, a few shapes can be rotated, and a few can be rolled.
  - A shape like Square can be rotated and a shape like Circle can be rolled.
  - So, it does not make sense to have rotate() or roll() in the Shape abstract class.
- The implementation of rotate() or roll() differs with the specific shape, so default implementation could not be provided.
- It is best to use interfaces rather than an abstract class. You can create Rotatable and Rollable interfaces that specify the protocol for rotate() and roll() individually

## Example: Choosing Between an Abstract Class and an Interface

```
public abstract class Shape {
    abstract double area();
    private Shape parentShape;
    public void setParentShape(Shape shape) {
          parentShape = shape;
    }
    public Shape getParentShape() {
       return parentShape;
    }
}
```

## Example: Choosing Between an Abstract Class and an Interface

```
// Rollable.java
// Rollable interface can be implemented by circular shapes such as
    Circle and Ellipse
public interface Rollable {
    void roll(float degree);
}
// Rotatable.java
// Rotable interface can be implemented by shapes such as Square,
    Rectangle, and Rhombus
public interface Rotatable {
    void rotate(float degree);
}
```

## Example: Choosing Between an Abstract Class and an Interface

```
// Circle.java
// Circle is a concrete class that is-a subtype of Shape; you can roll it and hence implements Rollable
public class Circle extends Shape implements Rollable {
      private int xPos, yPos, radius;
      public Circle(int x, int y, int r) {
            xPos = x;
            yPos = y;
            radius = r;
      }
    public double area() { return Math.PI* radius * radius; }

      @Override
      public void roll(float degree) {
          // implement rolling functionality here
      }
      public static void main(String[] s) {
            Circle circle = new Circle(10,10,20);
            circle.roll(45);
      }
}
```

## Example: Choosing Between an Abstract Class and an Interface

```
// Rectangle.java
// Rectangle is a concrete class and is-a Shape; it can be rotated and hence
    implements Rotatable
public class Rectangle extends Shape implements Rotatable {
      private int length, height;
      public Rectangle(int l, int h) {
            length = l;
            height = h;
      }
      public double area() { return length * height; }
      @Override
      public void rotate(float degree) {
            // implement rotating functionality here
      }
}
```

## Object Composition

- A composite object that is made up of other smaller objects.
- The composite object shares has-a relationships with the containing objects, and the underlying concept is referred to as object composition.
- Many real world objects having composition
  - E.g., A car contains Gear box system.
  - Advantage: non-functional gear box can be replaced.


## Example: Circle class

```
public class Circle {
      private int xPos;
      private int yPos;
      private int radius;
      public Circle(int x, int y, int r) {
            xPos = x;
            yPos = y;
            radius = r;
      }
      public String toString() {
          return "mid point = (" + xPos + "," + yPos + ") and radius = " + radius;
      }
}
```

- In this simple implementation, you use xPos and yPos to define the center of a Circle.
- Instead of defining these variables as members of class Circle, let's define a class Point, which can be used to define Circle's center.

## Example: Circle class with composition of Point

```
// Point is an independent class and here we are using it with Circle class
class Point {
      private int xPos;
      private int yPos;
      public Point(int x, int y) {
            xPos = x;
            yPos = y;
      }
      public String toString() {
          return "(" + xPos + "," + yPos + ")";
      }
}
```

## Example: Circle class with composition of Point

```
// Circle.java
public class Circle {
      private Point center; // Circle "contains" a Point object
      private int radius;
      public Circle(int x, int y, int r) {
            center = new Point(x, y);
            radius = r;
      }
      public String toString() {
          return "center = " + center + " and radius = " + radius;
      }
      public static void main(String []s) {
          System.out.println(new Circle(10, 10, 20));
      }
}
```

## Better solution using composition

- This is a better solution than having independent integer members xPos and yPos. Why?
  - You can reuse the functionality provided by the Point class. Note the rewriting of the toString() method in the Circle class by simplifying it:

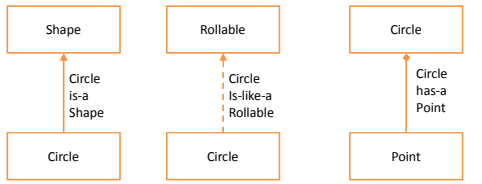
```
public String toString() {
    return "center = " + center + " and radius = " + radius;
}
```

- Here, the use of the variable center expands to center.toString().
- In this example, Circle has a Point object.
  - In other words, Circle and Point share a has-a relationship;
  - In other words, Circle is a composite object containing a Point object.

## Composition vs. Inheritance

- A rule of thumb is to use has-a and is-a phrases for composition and inheritance, respectively.
- For instance,
  - A computer has-a CPU.
  - A circle is-a shape.
  - A circle has-a point.
  - A laptop is-a computer.
  - A vector is-a list.
- This rule can be useful for identifying wrong relationships.
- Class inheritance implies an is-a relationship,
- Interface inheritance implies an is-like-a relationship,
- Composition implies a has-a relationship.

## Diagrams: Composition vs. Inheritance




Class inheritance (Generalization relation)

Interface inheritance (Realization relation)

Composition (Composition relation)

## Composition vs. Inheritance: Example

- Take a set of classes—say, DynamicDataSet and SnapShotData Set—which require a common functionality—say, sorting.
- Now, one could derive these data set classes from a sorting implementation.

```
public class Sorting {
      public List sort(List list) {
          // sort implementation
          return list;
      }
}
class DynamicDataSet extends Sorting {
    // DynamicDataSet implementation
}
class SnapshotDataSet extends Sorting {
    // SnapshotDataSet implementation
}
```

## Composition vs. Inheritance: Example

- It's not a good solution for the following reasons:
  1. DynamicDataSet is not a Sorting type.
  2. What if these two types of data set classes have a genuine base class, DataSet?
     - In that case, either Sorting will be the base class of DataSet or
     - One could put the class Sorting in between DataSet and two types of data sets. Both solutions would be wrong.

## Composition vs. Inheritance: Example

- There is another challenging issue:
  - What if one data set class wants to use one sorting algorithm (say, MergeSort) and
  - Another data set class wants to use a different sorting algorithm (say, QuickSort)?
- Will you inherit from two classes implementing two different sorting algorithms?
  - First, you cannot directly inherit from multiple classes, since Java does not support multiple class inheritance.
  - Second, even if you were able to somehow inherit from two different sorting classes (MergeSort extends QuickSort, QuickSort extends DataSet), that would be an even worse design.

## use a has-a relationship instead of an is-a relationship

```
interface Sorting {
    List sort(List list);
}
class MergeSort implements Sorting {
      public List sort(List list) {
          // sort implementation
          return list;
      }
}
class QuickSort implements Sorting {
      public List sort(List list) {
      // sort implementation
      return list;
      }
}
```

## use a has-a relationship instead of an is-a relationship

```
class DynamicDataSet {
      Sorting sorting;
      public DynamicDataSet() {
          sorting = new MergeSort();
      }
     // DynamicDataSet implementation
}
class SnapshotDataSet {
      Sorting sorting;
      public SnapshotDataSet() {
          sorting = new QuickSort();
      }
      // SnapshotDataSet implementation
}
```

## Design Choice

- Adhere to the OO design principle of "favor composition over inheritance."
- Composition encourages you to follow another useful OO design principle: "program to an interface, not to an implementation."
  - A class should depend only on the interface of another abstraction and not on the specific implementation details of that abstraction.
- Implementation of a class should not depend on the internal implementation aspects of the other class.

## Design Choice

- There are many terms related to composition, such as association and aggregation.
- Association is the most general form of a relationship between two objects, whereas
- Composition and aggregation are special forms of association.
  - In composition, the lifetime of the contained object and the container object is the same,
  - Whereas that is not the case with aggregation.

## Motivations for design patterns and frameworks

- Developing software is hard
- Developing reusable software is even harder
- Established solutions include patterns and framework
- Design pattern supports reuse of software architecture and solution design
- Framework supports reuse of detailed design and skeleton code.
- Use of design patterns and frameworks can reduce software development time and improve the quality of software

## References


- Oracle Certified Professional Java SE 7 Programmer Exams 1Z0-804 and 1Z0-805, A Comprehensive OCP JP 7 Certification Guide, by S G Ganesh and Tushar Sharma, Apress,