# Adapter Design Pattern

In this lecture, we describe the `Adapter` design pattern. We start by reviewing the concept of structural design pattern. Next, we shall enumerate some structural patterns, then, we completely define the functionality of the `Adapter`. Afterwards, we will outline its main features and we will conclude with an example of a concrete implementation of an `Adapter` pattern.

## 1 Structural design pattern

Structural patterns define compositional arrangements of classes or objects to create larger structures. They show how to glue together classes or objects to create flexible and extensible configurations. Hence, new functionality is obtained by combining classes or objects following well-defined rules. For example, some of the important structural design patterns are:

1. *Adapter* - adapts the interface of a class into another interface that the client expects.
2. *Bridge* - decouples interface and implementation into two different class hierarchies, so that they can be developed independently.
3. *Composite* - treats a group of objects in a similar way, as a single object. Client treats individual objects and compositions of objects similarly.
4. *Decorator* - adds new functionality to a class at runtime.
5. *Facade* - creates a new interface to simplify the use of an complex interface.
6. *Proxy* - uses a class to behave as a substitute for another one at the interface level.

## 2 Adapter intent

The `Adapter` pattern acts as a glue between two unrelated classes. It converts the interface of one given class to another interface that clients expect. Thus, the adapter object provides the functionality requested by clients, without having to know the implementation details of the class providing it.

The `Adapter` is sometimes called `Wrapper` because it wraps the existing class interface with a new interface that the client demands.

## 3 Occurring problem

The problem arises when one wants to use the functionality of an existing class with an interface that is different from the interface of the existing class.

For example, let us suppose there is a class that simulates the effects of climate change for a given temperature. However, the client only provided the temperatures in Fahrenheit, while class implementation requires the temperature in Celsius. Solution: Let the client invoke methods using Fahrenheit, implement a method that maps Fahrenheit to Celsius, then invoke methods on the class implementation using Celsius. This is the `Adapter` pattern intent. To follow the principle of such a design, one must understand the structure and the components of the adapter pattern.

## 4 Adapter pattern elements

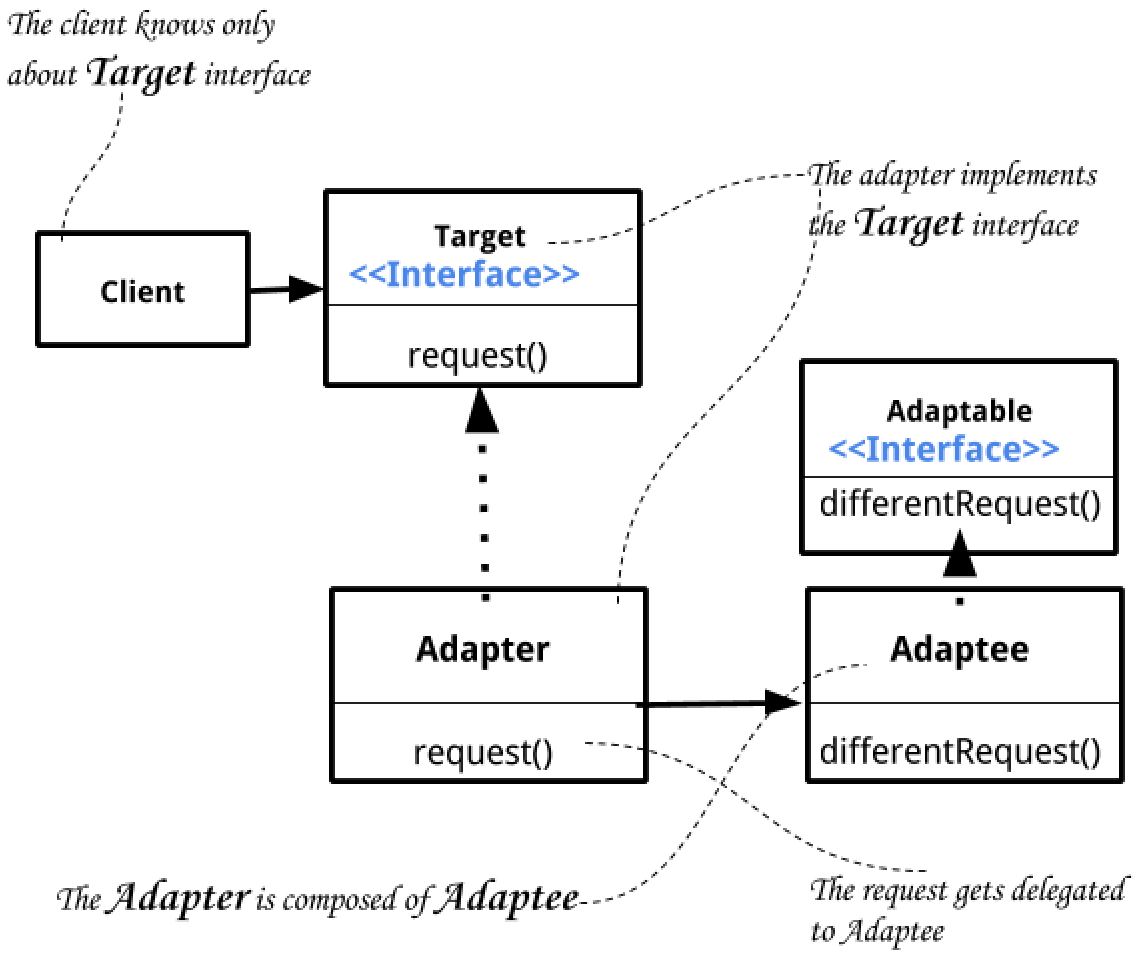
To properly design an adapter, one needs to define and name its elements. Here are the elements as UML specifications for the adapter pattern :

1. `Target interface`: The interface requested by client.

2. `Adaptee class`: The class that implements the functionality needed. The implementation is based on the `Adaptable` interface that is different from the `Target` interface the client wants.

3. `Adapter class`: This class is the core of the adapter pattern. It implements the `Target` interface. It modifies the client request and uses the `Adaptee` class to invoke its functionality. It obtains the results from the `Adaptee` class, but with a request specified by the `Target` interface.

4. `Client class`: The client interacts with a `Target` type object, implemented by an `Adapter` type object.


The client knows only about Target interface
Client
Target
<<Interface>>
request()
The adapter implements the Target interface
Adaptable
<<Interface>>
differentRequest()
Adapter
request()
Adaptee
differentRequest()
The Adapter is composed of Adaptee
The request gets delegated to Adaptee

## 5 Object adapter vs. class adapter

The adapter pattern described above is called `Object Adapter Pattern` since the adapter holds an object of the adaptee type. There is also another adapter category called `Class Adapter Pattern`. It uses inheritance instead of composition. However, it requires multiple inheritance to implement it, and therefore, we cannot use it in Java. In any case, the adapter pattern is needed insofar as there are fundamental incompatibilities between the target interface and the adaptee interface.

## 6 Object adapter usage

When a client makes a request by calling the method defined by the target interface, the object implementor of that interface is the adapter. The adapter translates the request from the client to the adaptee.

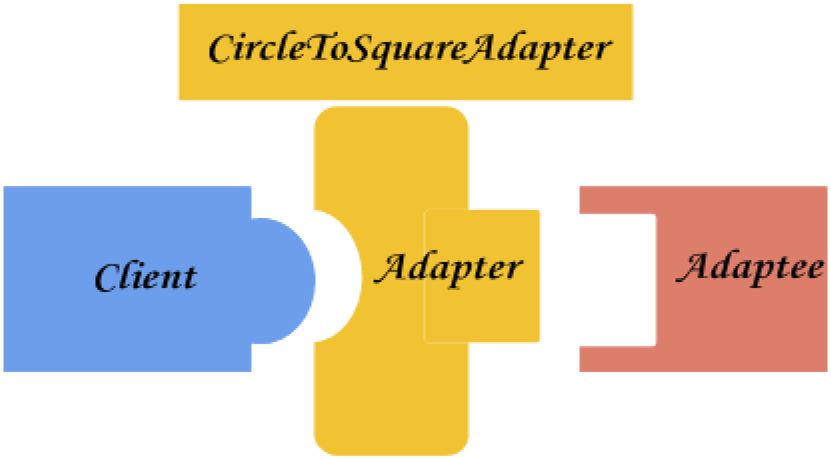
Eventually, the client receives the result of the request. The client is unaware of the existence of an adaptee object, since the adapter wraps the client request and forwards it to the adaptee. To demonstrate the usage of such an adapter pattern and to illustrate the process of building one, a simple problem is defined.

## 7 Adapter pattern example

Let us suppose that we want to calculate the area of a circle inscribed into a square. Therefore, for a given a square we want to develop a program that allows us to calculate the area of a circle inscribed into the square.

However, we know that there is a class defining a square, where a method to calculate the area of a square is implemented.

Consequently, given a square object, what we need is an adapter that is able to map the request (i.e., calculate the area of a circle) to another request – calculate the area of a square –, where the circle is inscribed into the given square. We will be calling this adapter `SquareToCircleAdapter.`



## 8 Example implementation

Our implementation defines an interface called **`Squareable`** and implements it in the class **`Square`**. It is simply a contract of the **`Square`** class emphasizing that a **`Squareable`** object knows how to calculate the area of a square.

```
/*************************************************************************
 *  Compilation:  javac Squareable.java
 *
 *  Interface to provide the area of a shape of type  square
 *  This is the contract of the Adaptee for the Adapter Design Pattern
 *************************************************************************/

public interface Squareable {
    /**
     * Returns the area of a square
     * @return The area of the square
     */
    double squareArea();
}
```

In our example, the **Square** class has a minimal implementation only to illustrate the role of an adaptee. It implements the **Squareable** interface. The only purpose of such a class is to act as an **Adaptee** for the adapter design pattern.

```
/*************************************************************************
 *  Compilation:  javac Square.java
 *
 *  This class describes a Square.  The Square class is defined by its
 *  dimension and it is used as an example for Adapter design pattern
 *
 *  This is the ADAPTEE for the Adapter Design Pattern
 *************************************************************************/
public class Square implements Squareable {

    private double width;

    /**
     * Construct a new Square with the specified width
     * @param width the width of the new Square
     */
    public Square(double width) {
        this.width = width;
    }

    /**
     * Get the area of this Square.
     * @return the area of this Square.
     */
    public double squareArea() {
        return width * width;
    }

    /**
     * @return the Square printable format as a string
     */
    @Override
    public String toString() {
        return "Square width=" + width;
    }
}
```

The **Target** interface in our example is the **Inscribable** interface. The client having a square shape wants to know the area of a circle inscribed into that given square. Since the **Square** class does not have such a method, we need to implement an adapter to map the client request to the **Square** class (i.e., the **Adaptee** class).

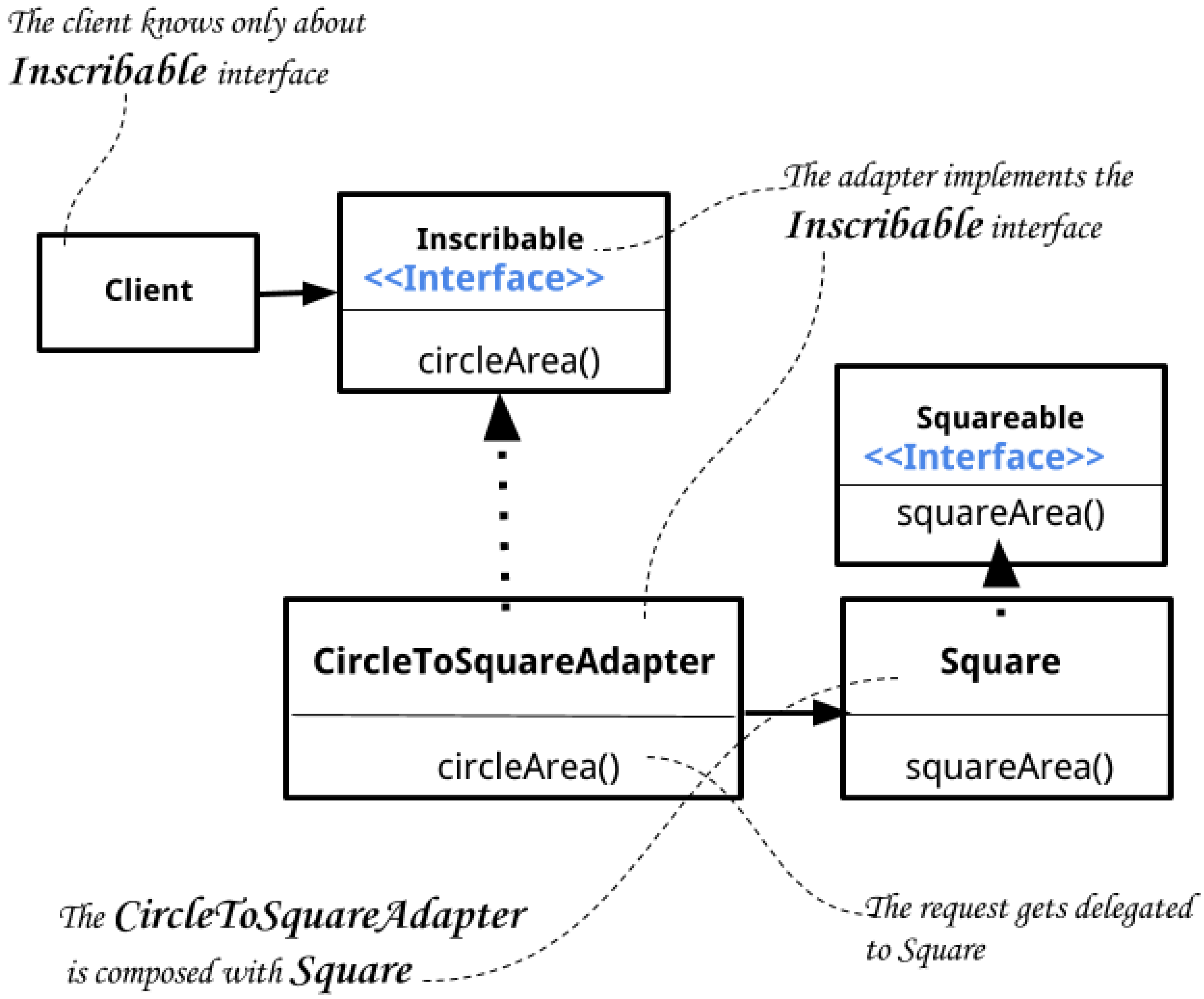
```
/***********************************************************************
 *  Compilation:  javac Inscribabel.java
 *
 *  Interface to calculate the area of a shape of type circle
 *  where the circle is inscribed in a given square
 *
 *  This is the TARGET interface for the Adapter Design Pattern
 ***********************************************************************/

public interface Inscribable {
    /**
     * Calculates the area of a circle inscribed in a square
     *
     * @param  shape The object of type Squareable
     * @return The area of the circle inscribed in the Square
     */
    double circleArea(Squareable shape);
}
```

Here is the `Adapter` pattern diagram transposed to the current example. The client wants to use the **Inscribable** interface, but the **Adaptee** class (the **Square** class in our example) implements only the **Squareable** interface.

To reconcile the client request with the existing implementation of the **Adaptee** class, we need to develop an adapter. The adapter must implement the **Inscribable** interface and embed through composition a **Squareable** object.

Thus, when the request comes from the client through an **`Inscribable`** type object, the adapter can map the request to the **`Squareable`** object (the **`Adaptee`** object), using the **`CircleToSquareAdapter`** object.



The **`CircleToSquareAdapter`** is the adapter class. It implements the **`Inscribable`** interface - the **`Target`** interface. It uses an **`Adaptee`** object to map the request from the target interface to the adaptee interface – the **`Squareable`** interface. The most important element of the **`CircleToSquareAdapter`** class is the field `shape`:

```
private final Squareable shape;
```

and the method

```
    public double circleArea(Squareable shape) {

        double area = shape.squareArea();
        double width = sqrt(area);

        return Math.pow(width / 2.0, 2.0) * PI;
    }
```

that implements the method defined by **`Inscribable`** interface, so that the client can invoke it.

Here is where the transformation – the mapping from **`Squareable`** to **`Inscribable`** – takes place. This is the core of the adapter class. It uses an object of type **`Squareable`** to calculate its area and implements the **`circleArea`** method to answer the client request.

```
import static java.lang.Math.PI;
import static java.lang.Math.sqrt;

/*************************************************************************
 *  Compilation:  javac CircleToSquareAdapter.java
 *
 *  This class is the ADAPTER class for the Adapter Design Pattern
 *
 *  It knows how to calculate the area of a circle inscribed in a square
 *  based ONLY on the Squareable object
 *
 *  It implements the Inscribable interface (the Target interface)
 *
 *  It is composed with the Squareable type (the Adaptee component)
 *  All the requests get delegated to the Adaptee (the Squareable)
 *
 *************************************************************************/
public class CircleToSquareAdapter implements Inscribable {

    /**
     * the immutable adaptee object of type Squareable
     **/
    private final Squareable shape;

    /**
     * Constructor - it builds an object of type adapter
     *
     * @param shape a Squareable object
     */
    public CircleToSquareAdapter(Squareable shape) {
        this.shape = shape;
    }
```

```java
    /**
     * Calculate the area of a circle inscribed in a square
     *
     * @param shape the object of type Square
     * @return the area of the circle.
     */

    @Override
    public double circleArea(Squareable shape) {

        double area = shape.squareArea();
        double width = sqrt(area);

        return Math.pow(width / 2.0, 2.0) * PI;
    }
}
```

The `CircleToSquareAdapter` is the adapter class. It implements the **`Inscribable`** interface - the **`Target`** interface. It uses an **`Adaptee`** object to map the request of the target interface to the adaptee interface – the **`Squareable`** interface.

When the client wants to calculate the area of a circle inscribed into a square, they need to:

1. Create the adaptee (**`Squareable`** object using the **`Square`** class)

2. Create the adapter, the **`CircleToSquareAdapter`** object, and expose it as the **`Target`** interface, namely the **`Inscribable`** object.

3. Invoke the method of the **`Inscribable`** interface and print the result.

```java
/*********************************************************************
 *  Compilation:  javac Client.java
 *  Execution:    java Client
 *
 *  This class uses an Adapter object
 *  to calculate the area of a circle inscribed in an square.
 *  1. Create the adaptee (Square object)
 *  2. Create the adapter (CircleToSquareAdapter object)
 *  3. Invoke the adapter method
 *
 *  Output:
 *  The Area of the circle inscribed in the [Square width=2.0] is 3.14159
 *********************************************************************/
```

```
public class Client {

    public static void main(String[] args) {

        /* create an object of type Squareable */
        Squareable square = new Square(2.0);

        /**
         * create an object of type Inscribable
         * as an adapter to calculate the circle area
         */
        Inscribable adapter = new CircleToSquareAdapter(square);


        //use the adapter and calculate the area of the circle inscribed
        double area = adapter.circleArea(square);

        //print the result
        System.out.println("The Area of the circle inscribed in the" +
                " [" + square + "] is " + area );
    }
}
```

## 9 Chapter resources

The basic online documentation for the design patterns is available at: https://en.wikipedia.org/wiki/Design_Patterns

More about the design patterns in Java, including examples and implementation, can be found at: http://www.journaldev.com/1487/adapter-design-pattern-java

## 10 Exercises

1. In the adapter example provided, change the **Inscribable** interface, so that the method is defined by:

```
public interface Inscribable {
    /**
     * Calculates the area of a circle inscribed in a square
     *
     * @param  width The dimension of the square
     * @return The area of the circle inscribed in the Square
     */
    double circleArea(double width);
}
```

What else needs to be changed? Refactor all the adapter components: the **Adaptee** class and its interface, the **Adapter** class and the **Client** class.

2. Develop an Adapter pattern implementation for solving the following problem:

The client wants to add two numbers in the `Binary` format, where `Binary` is a class that you have to implement. However, there exists an implementation that adds two numbers as integers.

For example, if your client invokes `add (Binary x, Binary y)`, you have an implementation of `add (Integer x, Integer y)`

## 11 A challenging exercise

1. Design and implement a solution for the following problem statement:

Let us suppose there is a web commerce application that uses a gateway payment system. The current gateway uses a representation of the credit card date in the format `year-month-day.`

For some internal reasons, management has decided to replace the gateway with another one that uses a different representation format, such as: `day/month/year`

Hint:
Apply the adapter design pattern for mapping the old gateway format to the new one.