Implementation (Impl) Objects

The Concepts:

Interface

An interface is a named collection of method definitions (without implementations). A class that implements the interface agrees to implement all the methods defined in the interface, thereby agreeing to certain behavior.

You use an interface to define a protocol of behavior that can be implemented by any class anywhere in the class hierarchy. Interfaces are useful for the following:

- Capturing similarities among unrelated classes without artificially forcing a class relationship
- Declaring methods that one or more classes are expected to implement
- Revealing an object's programming interface without revealing its class
- Modeling multiple inheritance, a feature that some object-oriented languages support that allows a class to have more than one super-class

The Concepts:

Delegation

Delegation is a technique where an object outwardly expresses certain behavior, but in reality it delegates (passes off) responsibility of providing that behavior to an associated object. This process is transparent to “client” code.

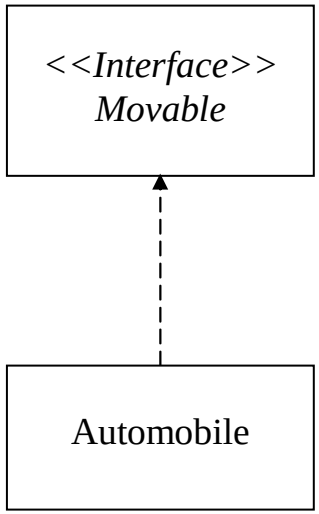
Delegation can be viewed as a relationship between objects where one object forwards certain method calls to another object, called its delegate. Delegation is a powerful design/reuse technique.

The primary advantage of delegation is run-time flexibility – the delegate can easily be changed at run-time.

Interface Implementation

An interface in Java is much like an abstract class, but with no constructors, method bodies, or instance variables. An interface can be used to define a restricted view of a group of objects, or to specify a minimal set of features a group of objects is expected to have for some particular purpose.

To "use" an interface, we need a class that implements the interface. For a class to implement an interface, the class must provide a method body to all of the abstract methods defined in the interface.




```
public interface Movable
{

    public void move(int seconds);

}
```

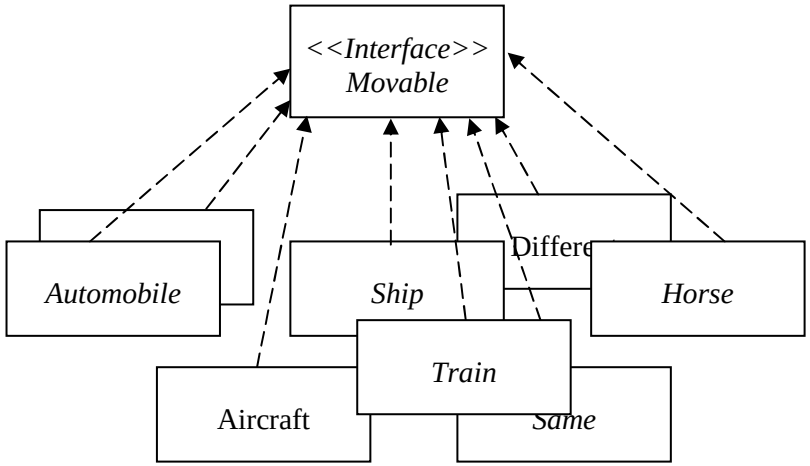
```
public class Automobile implements Movable
{
    private double speed;
    private Point location;

    public Automobile ()
    {…}

    public void move(int seconds)
    {
    // Automobile movement code goes here
    }
}
```

Interface Implementation

This simple interface implementation technique works well when most or all of the implementing classes require unique behavior implementations. However, when several of the implementing classes require identical implementations, this technique leads to code and data attribute duplication – a maintenance headache.




```
public class Automobile implements Movable
{
     private double speed;
     private Point location

     public Automobile ()
     {…}

     public void move(int seconds)
     {
     // Movement code goes here
     }
}
```

```
public class Aircraft implements Movable
{
     private double speed;
     private Point location

     public Aircraft ()
     {…}

     public void move(int seconds)
     {
     // Unique Aircraft-Related Movement
     // code goes here
     }
}
```

```
public class Train implements Movable
{
     private double speed;
     private Point location

     public Train ()
     {…}

     public void move(int seconds)
     {
     // Same Movement code duplicated here
     }
}
```

```
public class Ship implements Movable
{
     private double speed;
     private Point location

     public Ship ()
     {…}

     public void move(int seconds)
     {
     // Same Movement code duplicated here
     }
}
```

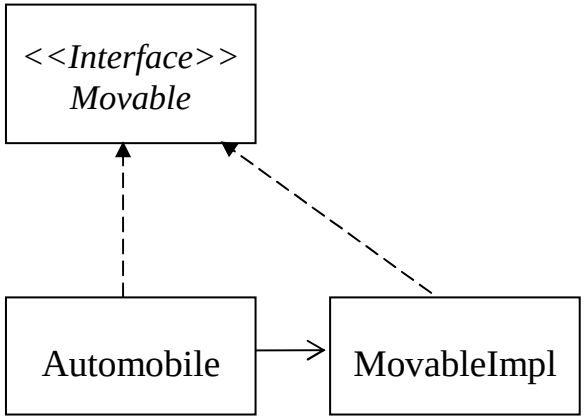
```
public class Horse implements Movable
{
     private double speed;
     private Point location

     public Horse ()
     {…}

     public void move(int seconds)
     {
     // Same Movement duplicated goes here
     }
}
```

Implementation Object

To solve the problem of duplicated code and data attribute definitions, the common behaviors and related data can be encapsulated in a class specifically designed to implement an interface (sometimes referred to as an "implementation" class or "Impl" class).




```
public interface Movable
{

    public void move(int seconds);

}
```

```
public class Automobile
                implements Movable
{
    private Movable movable;

    public Automobile ()
    {…}

    public void move(int seconds)
    {
        movable.move(seconds);
    }
}
```

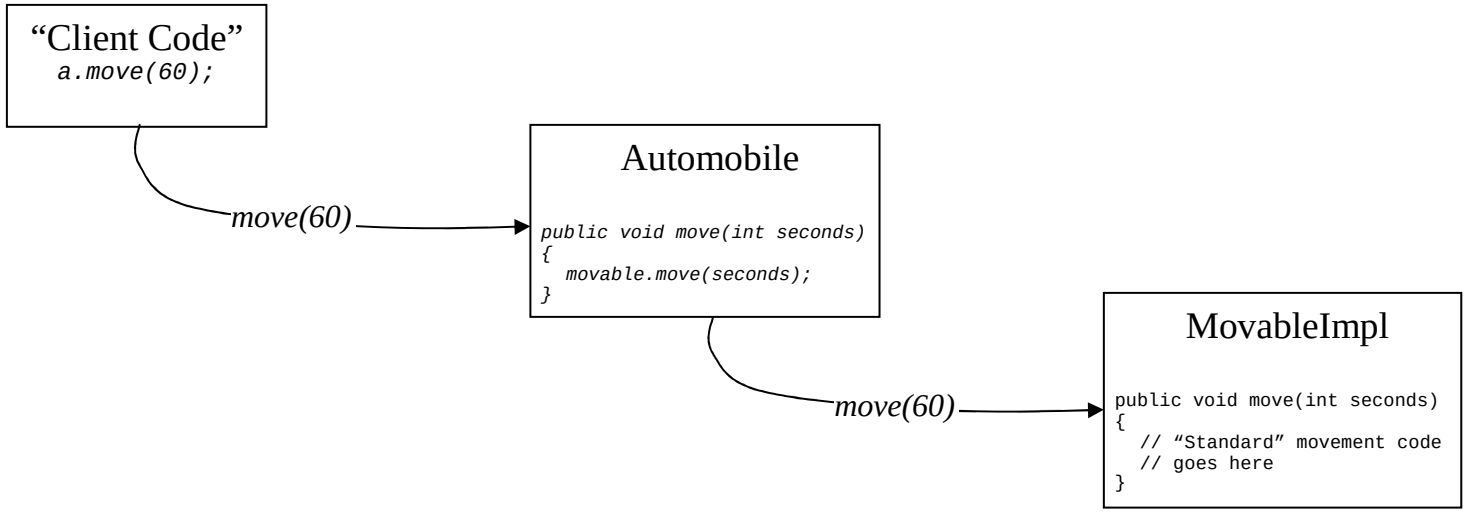
```
public class MovableImpl
                implements Movable
{
    private double speed;
    private Point location;

    public MovableImpl ()
    {…}

    public void move(int seconds)
    {
    // "Standard" movement code goes here
    }
}
```

Implementation Objects

Classes that require the same implementation of some or all interface behaviors can now "delegate" the interface-defined behaviors to this Impl object.

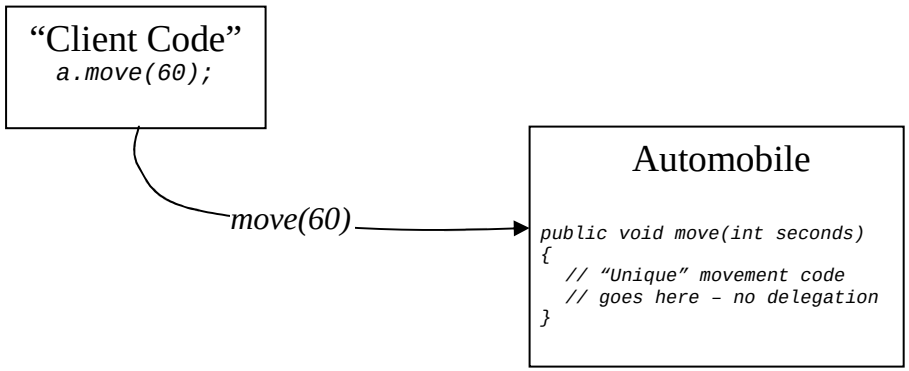


This delegation can be simple, as shown above, or more complex – spanning multiples classes, processes or applications.

Implementation Objects

Classes that require *unique* implementations of some of the interface behaviors need not delegate the interface-defined behaviors to this Impl class. Instead, the implementing class can define a specific implementation of one or more behaviors locally.

Those with unique implementations of *all* interface behaviors need not own an instance of the Impl class at all.



Summary:

- An interface is a named collection of method definitions (without implementations).
- Delegation is a technique where an object outwardly expresses certain behavior, but in reality it delegates (passes off) responsibility of providing that behavior to an associated object.
- Java interfaces allow the specification of behaviors that a group of objects is expected to have for some particular purpose.
- Code and data attribute duplication can result when several classes require identical interface implementations.
- Creating “implementation” classes is a great way to encapsulate interface implementations used by many implementors. Classes can now “delegate” the interface-defined behaviors to the associated implementation object.