Principles of Object-Orientation III

- Interface
- Interface Polymorphism
- Delegation

8) Interface

What is Interface?

A Java programming language keyword used to define a collection of method definitions and constant values. An interface can later be implemented by classes that define this interface with the "implements" keyword.

- *Sun (java.sun.com)*

In object technology, the ability of one class of objects to inherit properties from a higher class.

- *Computing Dictionary*

Inheritance is the incremental construction of a new definition in terms of existing definitions without disturbing the original definitions and their clients.

- *Dictionary of Object Technology*

Interface (cont.)

Interface

In general, an interface is a device or a system that unrelated entities use to interact. According to this definition, a remote control is an interface between you and a television set, the English language is an interface between two people, and the protocol of behavior enforced in the military is the interface between people of different ranks.

Within the Java programming language, an interface is a type, just as a class is a type. Like a class, an interface defines methods. Unlike a class, an interface never implements methods; instead, classes that implement the interface must implement the methods defined by the interface. A class can implement *multiple* interfaces.

You use an interface to define a protocol of behavior (i.e., a set of behaviors) 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 superclass

Interface (cont.)

Interface

Remember - an interface defines a protocol of behavior that can be implemented by any class anywhere in the class hierarchy. An interface defines a set of methods but does not implement them. A class that implements the interface agrees to implement all the methods defined in the interface, thereby agreeing to certain behavior.

An interface is a named collection of method definitions (without implementations).

Because an interface is simply a list of unimplemented, and therefore abstract, methods, you might wonder how an interface differs from an abstract class. The differences are significant.

- An interface cannot implement any methods, whereas an abstract class can.
- A class can implement many interfaces but can have only one superclass.
- An interface is not part of the class hierarchy. Unrelated classes can implement the same interface.

Interface (cont.)

Interface - Example

Suppose that you have written a class that can watch stock prices coming over a data feed. This class allows other classes to register to be notified when the value of a particular stock changes. First, the class (called StockMonitor) would implement a method called “watchStock(…)” that lets other objects register for notification:

```
public class StockMonitor
{
     public void watchStock(StockWatcher watcher,
                            TickerSymbol tickerSymbol,
                            BigDecimal delta)
      {
            . . .
      }
}
```

The first argument to the “watchStock(…)” method is a StockWatcher object. StockWatcher is the name of an *interface* whose code you will see in the next section. Any object that implements the StockWatcher interface can be passed to the “watchStock(…)” method.

The StockWatcher interface declares one method: “valueChanged(…)”. An object that wants to be notified of stock changes must be an instance of a class that *implements* this interface and thus implements the “valueChanged(…)”method.

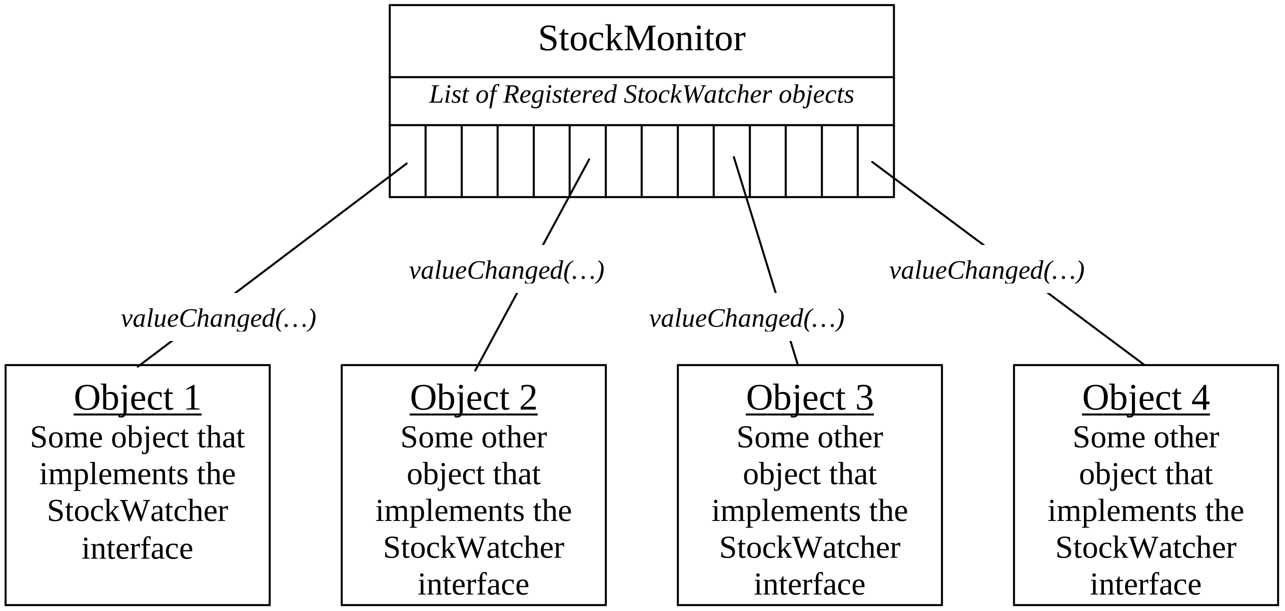
The other two arguments provide the symbol of the stock to watch (IBM, DELL, AMZN, etc.) and the amount of change that the watcher considers interesting enough to be notified of. When the StockMonitor class detects an interesting change, it calls the “valueChanged(…)” method of the watcher (the StockWatcher object passed in).

Interface (cont.)

Interface - Example

When a stock value changes, those StockWatcher objects (those that have implemented the StockWatcher interface) that have registered with the StockMonitor object receive a "valueChanged(…)" message.

The registered StockWatcher objects can be of *any* class type – the only requirement is that they all implement the StockWatcher interface so that the StockMonitor is guaranteed that all the registered objects will understand the "valueChanged(…)" message.


StockMonitor
List of Registered StockWatcher objects
valueChanged(…)
valueChanged(…)
valueChanged(…)
valueChanged(…)
Object 1
Some object that implements the StockWatcher interface
Object 2
Some other object that implements the StockWatcher interface
Object 3
Some other object that implements the StockWatcher interface
Object 4
Some other object that implements the StockWatcher interface

Interface (cont.)

Interface - Example

The “watchStock” method ensures, through the data type of its first argument, that all registered objects implement the “valueChanged(…)” method. Since they all implement the StockWatcher interface they are guaranteed to have an implementation of the “valueChanged(…)” method.

It makes sense to use an interface data type here because it matters only that registrants implement a particular method. If StockMonitor had used a class name as the data type, that would artificially force a class relationship on its users.

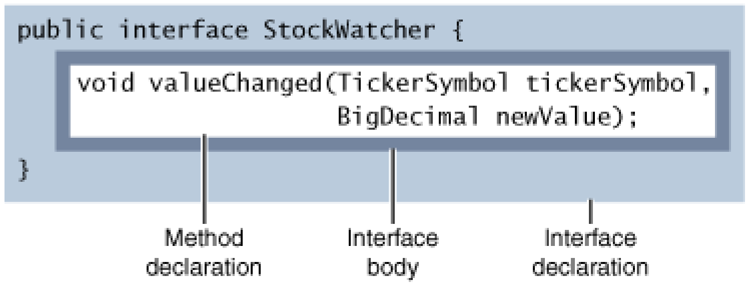
Because a class can have only one superclass, it would also limit what type of objects can use this service. By using an interface, the registered objects class could be anything, thus allowing any class anywhere in the class hierarchy to use this service.

The figure below shows that an interface definition has two components: the interface declaration and the interface body. The interface declaration declares various attributes about the interface, such as its name and whether it extends other interfaces. The interface body contains the constant and the method declarations for that interface.

Interface (cont.)

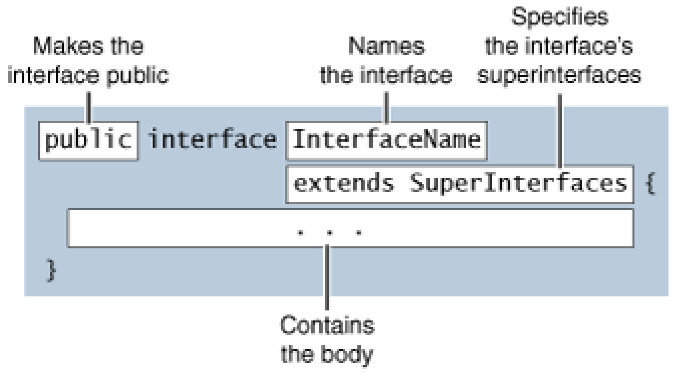
Interface - Example

As you can see, the StockWatcher interface shown in the figure below declares, but does not implement, the "valueChanged(…)" method. Classes that *implement* this interface provide the implementation (the code) for that method.



```
public interface StockWatcher {
    void valueChanged(TickerSymbol tickerSymbol,
                      BigDecimal newValue);
}
```

The following figure shows all possible components of an interface declaration:



```
public interface InterfaceName
                 extends SuperInterfaces {
    . . .
}
```

Makes the interface public
Names the interface
Specifies the interface's superinterfaces
Contains the body

Two elements are required in an interface declaration - the *interface* keyword and the *name* of the interface. The public access specifier indicates that the interface can be used by any class in any package. If you do not specify that your interface is public, your interface will be accessible only to classes that are defined in the same package as the interface.

Interface (cont.)

Interface – Example

An interface declaration can have one other component: a list of superinterfaces. An interface can extend other interfaces, just as a class can extend or subclass another class. However, whereas a class can extend only one other class, an interface can extend any number of interfaces. The list of superinterfaces is a comma-separated list of all the interfaces extended by the new interface.

The interface body contains method declarations for all the methods included in the interface. A method declaration within an interface is followed by a semicolon “;” because an interface does not provide implementations for the methods declared within it. All methods declared in an interface are implicitly public and abstract.

An interface can contain *constant declarations* in addition to method declarations. All constant values defined in an interface are implicitly *public, static,* and *final*.

Member declarations in an interface disallow the use of some declaration modifiers; you cannot use transient, volatile, or synchronized in a member declaration in an interface. *Also, you may not use the private and protected specifiers when declaring members of an interface.*

Interface (cont.)

Interface - Example

An interface defines a protocol of behavior. A class that implements an interface adheres to the protocol defined by that interface. To declare a class that implements an interface, include an “implements” clause in the class declaration. Your class can implement more than one interface, so the implements keyword is followed by a comma-separated list of the interfaces implemented by the class.

Here's a partial example of a class that implements the StockWatcher interface:

```
public class StockPgm implements StockWatcher
{
    public void valueChanged(TickerSymbol tickerSymbol,
                             BigDecimal newValue)
    {
        switch (tickerSymbol)
        {
            case SUNW:
                ...
                break;
            case ORCL:
                ...
                break;
            default:
                // handle unknown stocks
                ...
                break;
        }
    }
}
```

Interface (cont.)

Interface – Example

When a class implements an interface, it is essentially signing a contract.

Either the class must implement all the methods declared in the interface and its superinterfaces, or the class must be declared abstract. The method signature (the name and the number and type of arguments) in the class must match the method signature as it appears in the interface. The StockPgm class chown earlier implements the StockWatcher interface, so the class provides an implementation for the “valueChanged(…)” method.

When you define a new interface, you are actually defining a new reference data type. You can use interface names anywhere you can use any other data type name. Recall that the data type for the first argument to the “watchStock” method in the StockMonitor class is a StockWatcher:

```
public class StockMonitor
{
    public void watchStock(StockWatcher watcher,
                           TickerSymbol tickerSymbol,
                           BigDecimal delta)
     {
         ...
     }
}
```

Only an instance of a class that implements the interface can be assigned to a reference variable whose type is an interface name. So only instances of a class that implements the StockWatcher interface can register to be notified of stock value changes. StockWatcher objects are guaranteed to have a “valueChanged(…)” method.

Interface (cont.)

Interface Extension

Suppose that you want to add some functionality to the StockWatcher interface. For instance, suppose that you want to add a method that reports the current stock price, regardless of whether the value changed:

```
public interface StockWatcher
{
    void valueChanged(TickerSymbol tickerSymbol,
                      BigDecimal newValue);

    void currentValue(TickerSymbol tickerSymbol,
                      BigDecimal newValue);
}
```

However, if you make this change, *all* classes that implement the StockWatcher interface will break because they don't fully implement the interface anymore! Programmers relying on this interface will protest loudly.

Try to anticipate all uses for your interface up front and specify it completely from the beginning. Given that this is often impossible, you may need either to create more interfaces later or to break your customer's code. For example, you could create a StockWatcher *subinterface* called StockTracker that declared the new method:

```
public interface StockTracker extends StockWatcher
{
     void currentValue(TickerSymbol tickerSymbol,
                           BigDecimal newValue);
}
```

Now users of your code can choose to upgrade to the new interface or to stick with the old interface.

9) Interface Polymorphism

Interface Polymorphism

Interface polymorphism - Multiple classes may implement the same interface, and a single class may implement one or more interfaces.

Interfaces are essentially definitions of how a class needs to respond. An interface describes the methods, properties, and events that a class needs to implement, and the type of parameters each member needs to receive and return, but leaves the specific implementation of these members up to the implementing class.

A powerful technique in component programming is being able to implement multiple interfaces on an object. Each interface is composed of a small group of closely related methods, properties, and events.

By implementing interfaces, your component can provide functionality to any other component requiring that interface, without regard to the particular functionality contained within. This allows successive versions of components to incorporate different capability without disturbing core functionality.

Interface Polymorphism (cont.)

Interface Polymorphism

A group of heterogeneous objects can be grouped together and treated as the same “type” provided they all implement the same interface. Then, a *single* method call be made to those heterogeneous objects which will result in an infinite variety of resultant behaviors.

Another benefit of defining features in terms of interfaces is that you can add features to your component incrementally by defining and implementing additional interfaces. Advantages include:

- Design effort is simplified, because components can begin small, with minimal functionality, and continue to provide that functionality while acquiring additional features over time, as it becomes clear from actual use what those features should be.
- Maintaining compatibility is simplified, because new versions of a component can continue to provide existing interfaces, while adding new interfaces. Subsequent versions of client applications can take advantage of these when it makes sense for them to do so.

Interface (cont.)

Abstract Classes vs. Interfaces

The choice of whether to design your functionality as an interface or an abstract class can sometimes be a difficult one. An abstract class is a class that cannot be instantiated, but must be inherited from. An abstract class may be fully implemented, but is more usually partially implemented or not implemented at all, thereby encapsulating common functionality for inherited classes.

An interface, by contrast, is a totally abstract set of members that can be thought of as defining a contract for conduct. The implementation of an interface is left completely to the developer.

Both interfaces and abstract classes are useful for component interaction. If a method requires an interface as an argument, then any object that implements that interface can be used in the argument.

Interface (cont.)

Abstract Classes vs. Interfaces

The following are guidelines to help you to decide whether to use an interface or an abstract class to provide polymorphism for your components.

- If you anticipate creating multiple versions of your component, create an abstract class. Abstract classes provide a simple and easy way to version your components. By updating the base class, all inheriting classes are automatically updated with the change. Interfaces, on the other hand, cannot be changed once created. If a new version of an interface is required, you must create a whole new interface.

- If the functionality you are creating will be useful across a wide range of disparate objects, use an interface. Abstract classes should be used primarily for objects that are closely related, whereas interfaces are best suited for providing common functionality to unrelated classes.

- If you are designing small, concise bits of functionality, use interfaces. If you are designing large functional units, use an abstract class.

- If you want to provide common, implemented functionality among all implementations of your component, use an abstract class. Abstract classes allow you to partially implement your class, whereas interfaces contain no implementation for any members.

10) Delegation

What is Delegation

Delegation is handing a task over to a subordinate. It is the assignment of authority and responsibility to another person to carry out specific activities. It allows a subordinate to make decisions, i.e. it is a shift of decision-making authority from one organizational level to another lower one. Delegation, if properly done, is not abdication. Ultimate responsibility cannot be delegated.

- *Wikipedia*

Delegation involves the transfer to others of responsibility for carrying out certain tasks, functions or decisions.

- *Business 2000*

Delegation is a way of extending and reusing a class by writing another class with additional functionality that uses instances of the original class to provide the original functionality.

- *Grand*

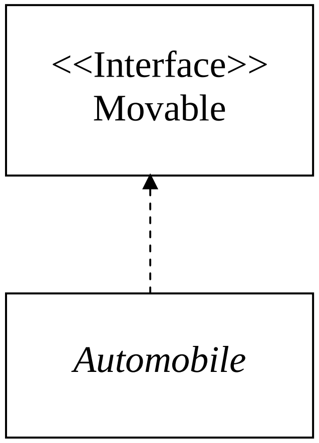
Delegation

Delegation

Delegation can be viewed as a relationship between objects where one object forwards certain method calls to another object, called its “delegate”. The primary advantage of delegation is run-time flexibility – the delegate can easily be changed at run-time.

Recall that 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.

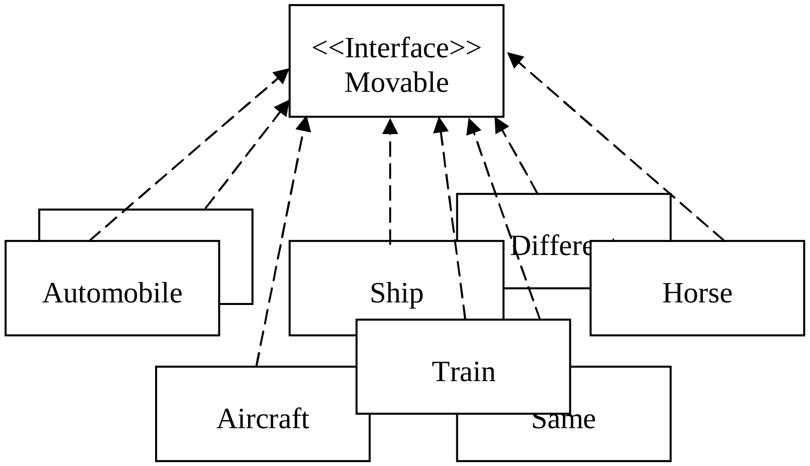


```
public interface Movable
{
   public void move(int seconds);
}

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

   public Automobile ()
   {…}

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

Delegation (cont.)

Delegation

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
     }
}
```

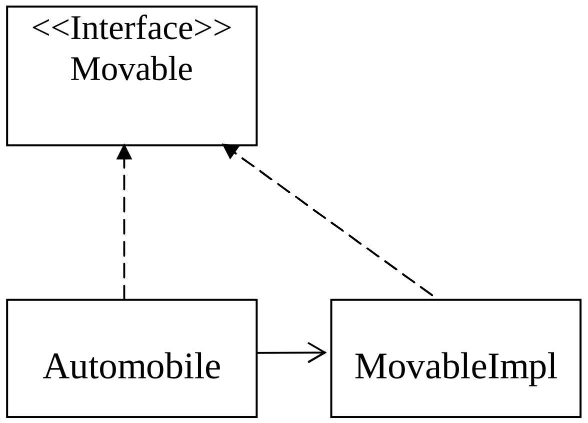
```
public class Ship implements Movable
{
     private double speed;
     private Point location

     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
     }
}
```

Delegation (cont.)

Delegation

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);

}
```

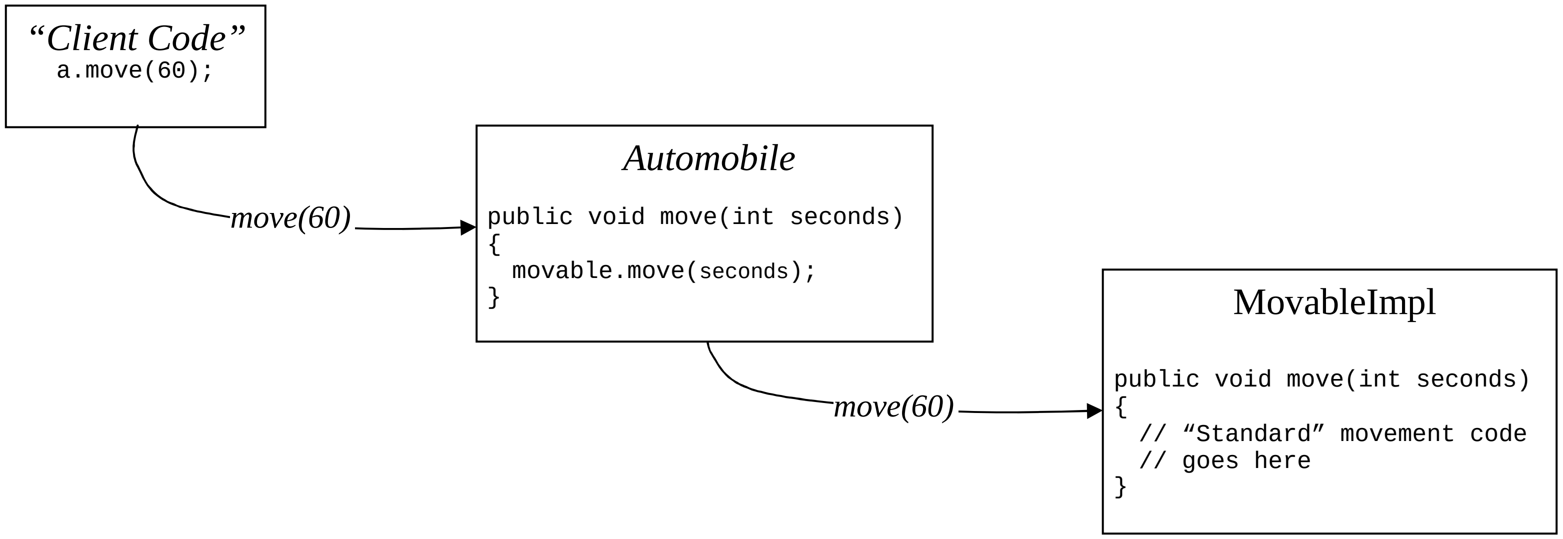
```
public class Automobile
                implements Movable
{
    private Movable movable;

    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
    }
}
```

Delegation (cont.)

Delegation

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



```
a.move(60);
```

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

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

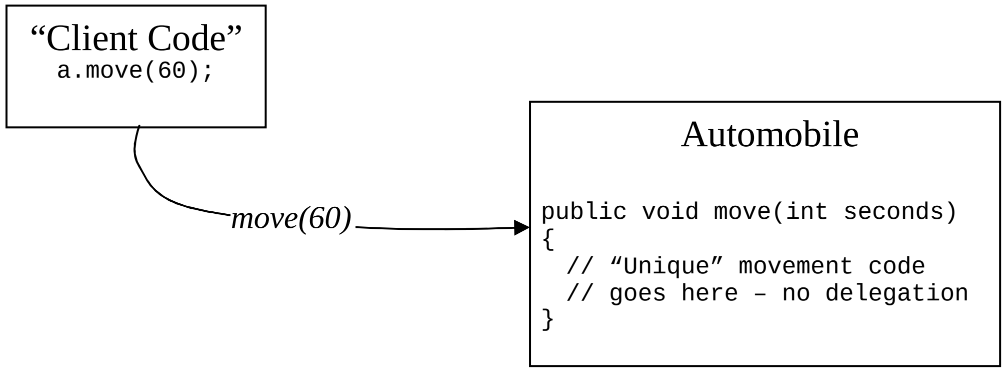
This delegation can be simple, as shown above, or more complex – spanning multiples classes, processes or applications.

Delegation (cont.)

Delegation

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.



```
public void move(int seconds)
{
  // "Unique" movement code
  // goes here – no delegation
}
```

Summary:

- An interface defines a protocol of communication between two objects.
- An interface definition is comprised of a declaration and a body. The interface body contains declarations, but no implementations, for a set of methods. An interface might also contain constant definitions.
- A class that implements an interface must implement all the methods declared in the interface. An interface name can be used anywhere a type can be used.
- Delegation can be viewed as a relationship between objects where one object forwards certain method calls to another object, called its “delegate”. The primary advantage of delegation is run-time flexibility – the delegate can easily be changed at run-time.
- Delegation is a way of extending and reusing a class by writing another class with additional functionality that uses instances of the original class to provide the original functionality.