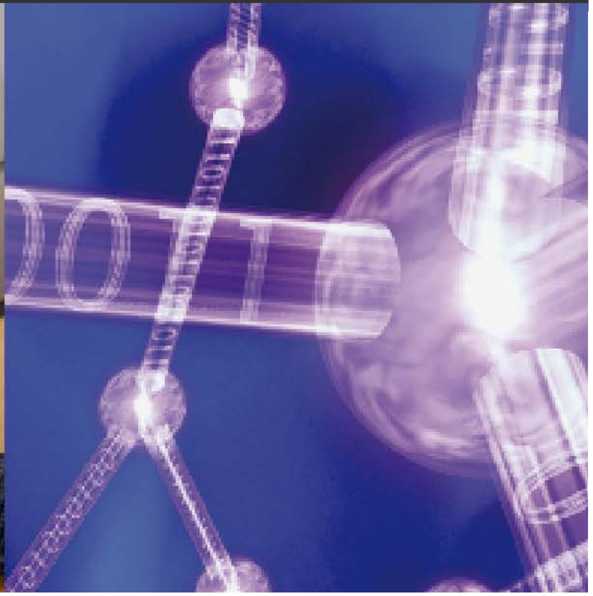
MONASH University

Information Technology

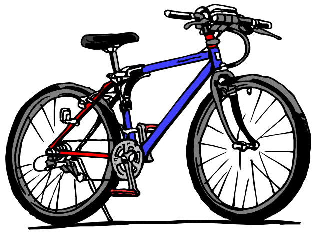
Module 8: Interfaces, Abstract Classes and Callbacks

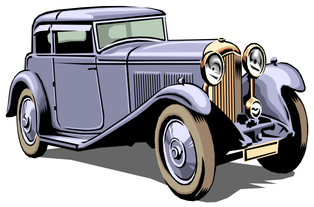
FIT2034 Computer Programming 2
Faculty of Information Technology



Generalisation - Revision

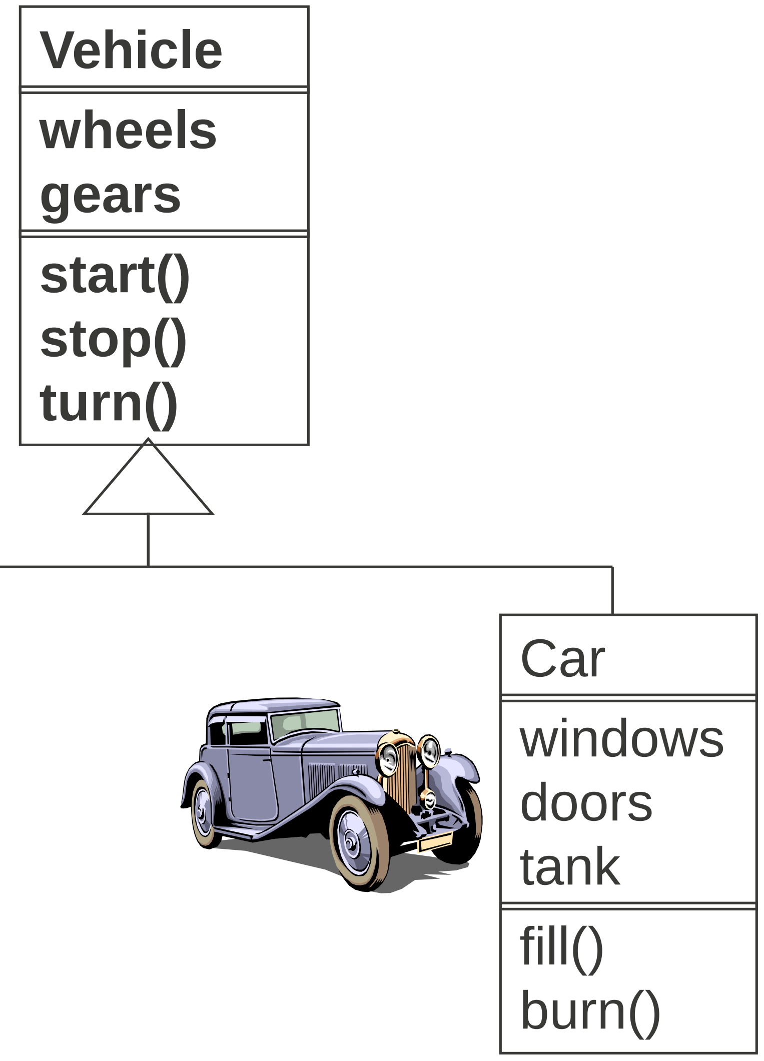
- Sometimes a group of objects share some common attributes, relationships, and/or methods
  - Bicycles and cars both are types of vehicles, both have wheels and both can stop, start and turn
  - Cars have doors, windows and fuel tank, but bicycles don't
  - Bicycles have pedals and handlebars but cars don't
  - Cars are powered by fuel burning whereas bicycles are powered by pedaling

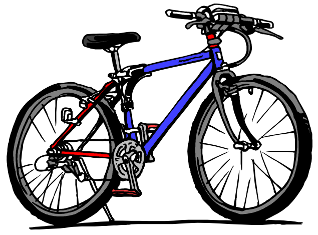


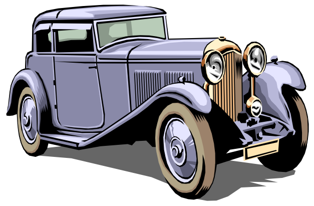


Generalisation – Revision (2)

- Best to define common members in a **general**, *parent* or *super* class
- *Child* or *sub* classes can ***inherit*** common members from parent class
- Child classes can have additional **specific** members







Generalisation vs Specialisation – Revision

- **Generalisation** is the process of moving common members from distinct classes into a more-general class which becomes a common parent.
- **Specialisation** is the process of creating a new subclass for an existing class (i.e. other direction)

Reference Variables & Instantiating Objects

- Variable can refer to any object instance that is of the same class type, *or* of a derived class of that type

- Examples:

```
Car myCar = new Car(…);
Bicycle yourBike = new Bicycle(…);

Vehicle transportDevice = myCar;
Vehicle cheapRide = yourBike;
```

Polymorphism - Revision

- Polymorphism arises when a reference variable could refer to a range of types of objects
  - Exact method performed not known at compile time
  - Different behavior
  - Late binding

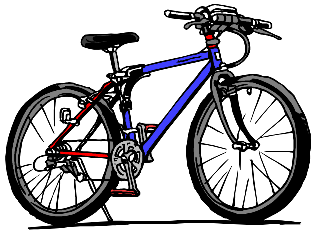
Polymorphism - Revision

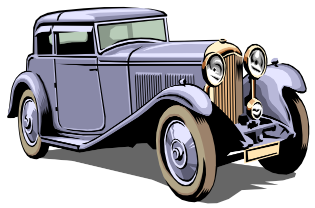
```
public class Color {
  public String test() {
        return "I'm a generic color";
  }
}

public class Blue extends Color {
  public String test() {
        return "I'm blue";
  }
}

public class Red extends Color {
  public String test() {
        return "I'm red";
  }
}
```

```
public class Test
{
  public String test(int which) {
    Color c = null;

    if (which==0) c = new Blue();
    else c = new Red();

    return c.test();
  }
}
```

Instantiating Objects (2)

- Some classes should not be able to be instantiated.
- Consider the previous examples:
  - Can you instantiate a generic ‘Color’? A generic Vehicle?
  - Do you need to?
  - What would a ‘vehicle’ look like in the real world? How would ‘vehicles’ behave?
- Vehicle is an abstract concept. The more-specific derived classes such as Car and Bicycle are more concrete.





Proposed Vehicle class

```
public class Vehicle {
   int wheels;
   int gears;
   Vehicle (int wheels, int gears) {
      this.wheels = wheels;
      this.gears = gears;
   }
   public void start () {…}
   public void stop() {…}
   public void turn() {…}
    public String toString() {.. }
}
```

What code would we write for the start(), stop() and turn() methods?

Abstract methods

- Methods for which you do not wish to provide code for can be left unimplemented
- Such methods are called ***abstract methods***
- You must mark these methods explicitly, using the Java keyword **abstract**
- Instead of using the open and close braces to give the method a body, you just use a semi-colon
- Example:

```
public abstract void start();
public abstract void turn();
```

Abstract Classes

- If at least one method in a class is declared as abstract, then that class becomes an ***Abstract class***
- Abstract classes are classes which are never to be instantiated because it is an incomplete class
  - A subclass must be instantiated instead of the abstract class
- You must mark these classes using the **abstract** keyword

- You still specify all other details of the class, such as attributes and methods for which you know how to implement in this general class

Vehicle class – made abstract

```
public abstract class Vehicle {
   int wheels;
   int gears;
   Vehicle (int wheels, int gears) {
      this.wheels = wheels;
      this.gears = gears;
   }
   public abstract void start () ;
   public abstract void stop() ;
   public abstract void turn() ;
    public String toString() {.. }
}
```

Concrete Child Classes

- Since a subclass inherits everything from its superclass, it would normally inherit the abstract definitions of methods, and automatically be an abstract class.
- The opposite of an abstract class is a ***concrete class***
- You cannot instantiate a class which is abstract.
- You can only instantiate a class which is concrete.

- Child classes must therefore provide a ***concrete implementation*** (bodies) for all inherited abstract methods, in order for the class to become a concrete class
  - The method signatures must match *except* for the 'abstract' keyword

Bicycle Class – a concrete Vehicle subclass

```
public class Bicycle extends Vehicle {
    String handles;
    double pedals;
    Bicycle (int wheels, int gears, String handles) {
        super(wheels, gears);
        this.handles = handles;
        pedals = 0.0;
     }
    public double pedal () {.. }
    public String toString() {   return (super.toString() + " " +
       handles + " " + pedals);  }
    public void start () { … }
    public void stop () { … }
    public void turn () { … }
}
```

The constructor in Vehicle must still be implemented concretely

Bodies will need to be provided in this class

Color example using abstract class

Abstract

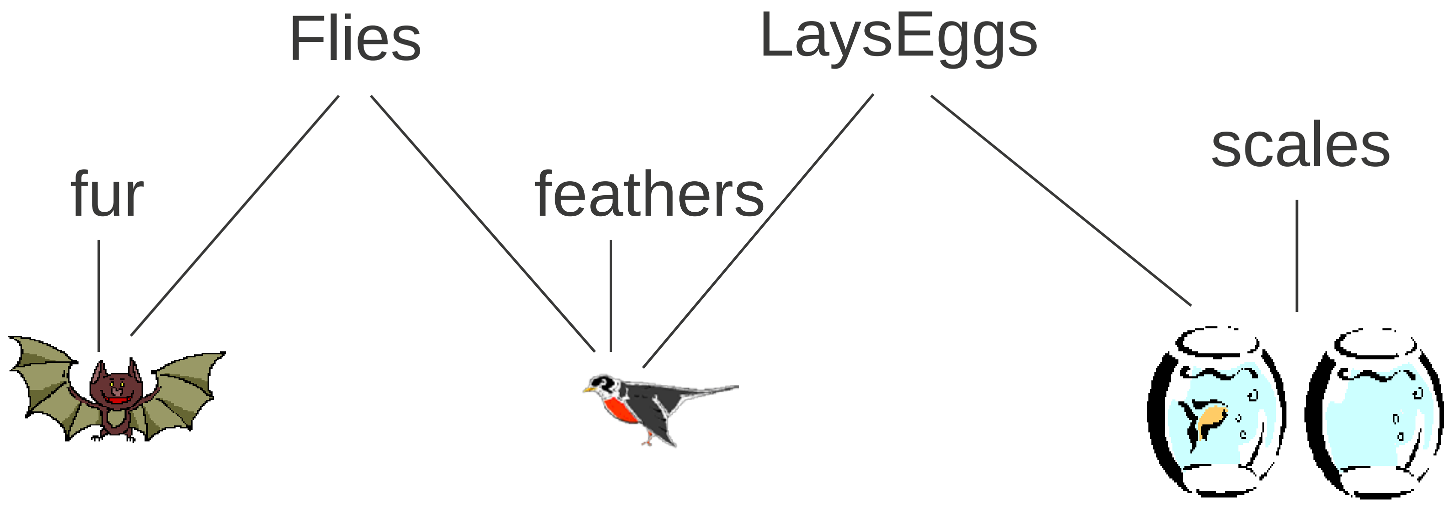
```
public abstract class Color {
   public abstract String test();
}

public class Blue extends Color {
   public String test() {              No abstract
      return "I'm blue";
   }
}

public class Red extends Color {
   public String test() {              No abstract
      return "I'm red";
   }
}
```

Generalization hierarchies – Revision

- A class with no significant parents is a base class (e.g. Vehicle)
- A child class is a refined-kind of some parent class (e.g. Car is a child of Vehicle)
- A class with no children is a leaf class
- Objects of the child class can be substituted for the parent, but not the reverse
- A generalisation relationship is structural, not behavioural
  - Messages cannot generally be passed between objects in a generalisation relationship: the identity of the object is one.
  - However, you can invoke a method inherited from an ancestor, but it impacts on the object itself
- In Java a class can have only one direct superclass

Multiple inheritance

- Sometimes classes should inherit members from more than one class.



- How could the bird inherit behaviors defined by ‘Flies’ and also ‘LaysEggs’?

Multiple inheritance in Java

- An ***interface*** is a device in Java for marking a class as conforming to a defined protocol or format
- Consists of a list of method signatures, but with no bodies
  - like an abstract method, but without the abstract keyword
- It supports the technique of ***programming by contract***
- It allows role-based programming in Java
  - (More discussion later)
- Most importantly, it allows a type of multiple-inheritance. i.e. the resultant class inherits features from several sources

Multiple inheritance in Java (2)

- Any class wishing to accept the responsibilities specified by an interface must *implement* the interface
- This means that the class must provide a concrete implementation for every method of the interface
- This ensures that:
  - the additional responsibilities/functionality is implemented, while
  - the inheriting classes can remain in their structural classification hierarchy, retaining their flexibility

Multiple inheritance in Java - Example

```
public interface Flies {
  public static final int LEFT_WING = 1;
  public static final int RIGHT_WING = 2;

  // method signatures which classes must implement…
  public void flap (int whichWing, double flapRate, int
    duration);
}
class Bird extends FeatheredAnimal implements Flies, LaysEggs
  {
  public void flap(int whichWing, double rate, int
    duration)
  {            // we need to provide a body here in
               // the Bird class
      ...
  }
}
```

Interface Type

- Object reference variables can be declared to be of an interface type:
  - The variable may refer to any object which implements the interface
  - Note: the objects could be from classes with completely different parent classes
- Example:

```
Flies aBat;
Flies aBird;
aBat = new Bat();
aBird = new Bird();
// Assuming there is also a Bird concrete class implementing Flies
// Recall the concept of polymorphism
```

Example: A general interface for shapes

- Example taken from Reges & Stepp (2nd ed.) – Page 600.

```
public interface Shape {
   public double getArea();
   public double getPerimeter();
}
```

- All shapes have methods to compute their areas and their perimeters.
- We don't specify how they will be implemented yet.

Implementing the Shape Interface (1)

- When Rectangle class implements Shape, we are promising that the class will contain implementations of the getArea and the getPerimeter methods.

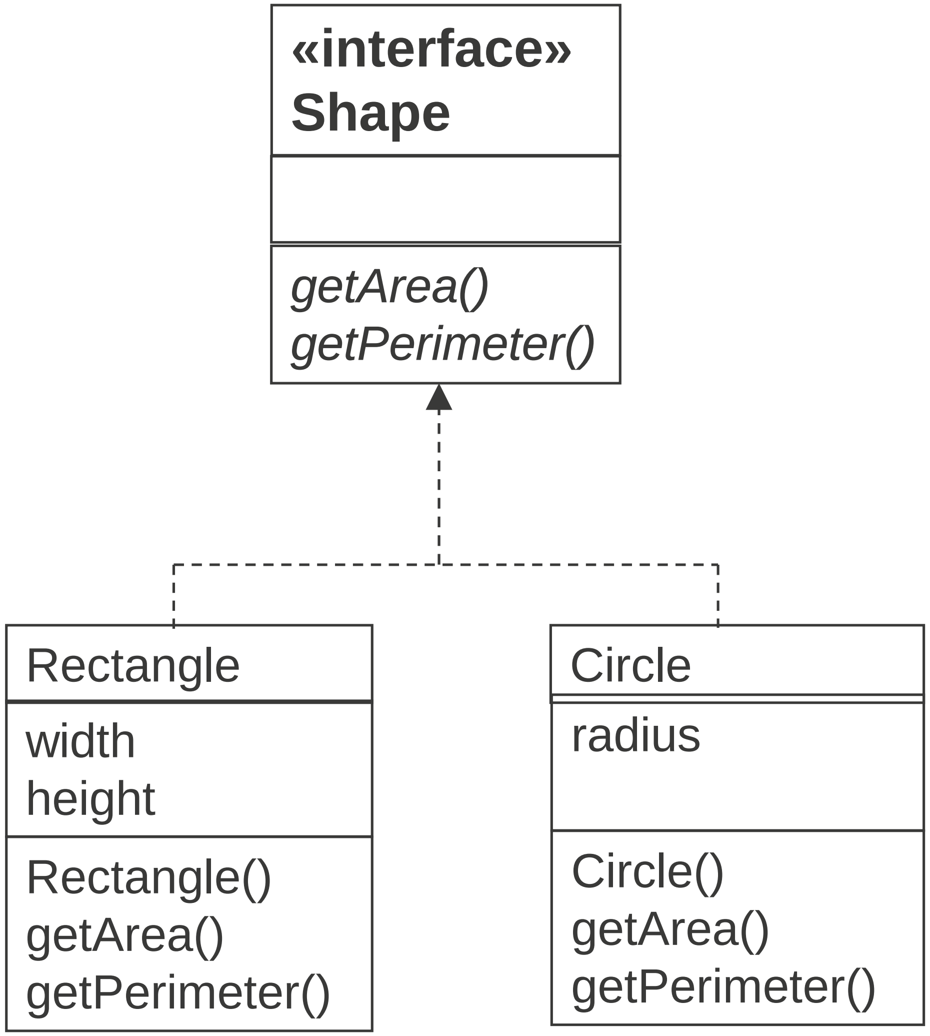
```
public class Rectangle implements Shape {
  private double width;
  private double height;
  public Rectangle(double width, double height) {
       this.width = width;
       this.height = height;
  }
  public double getArea() {
       return width * height;
  }
  public double getPerimeter() {
       return 2.0 * (width + height);
  }
} // end of Rectangle class
```

Implementing the Shape Interface (2)

```
public class Circle implements Shape {
  private double radius;

  public Circle(double radius) {
       this.radius = radius;
  }

  public double getArea() {
       return Math.PI * radius * radius;
       // PI is a constant in the Math class
  }

  public double getPerimeter() {
       return 2.0 * Math.PI * radius;
  }
}
```

Benefits of Interfaces

- Classes that implement a common interface form a type hierarchy similar to those created by inheritance.
- interfaces can be used to achieve polymorphism as well as normal class inheritence



Polymorphism using Shape classes

```
...
Shape[] shapes = new Shape[3];
Shapes[0] = new Rectangle(10, 10);
Shapes[1] = new Circle(30);
Shapes[2] = new Rectangle(10, 15);
for (int i = 0; i < shapes.length; i++)
  System.out.println(shapes[i].getArea() + " " +
                     shapes[i].getPerimeter());
...
```

`List` Interface

- The Java API defines the **List** interface (in java.util).
  - It is actually List<E>
- The ArrayList class implements List
  - Remember, we must provide element type as parameter
- Therefore, the following is entirely valid due to polymorphism:

```
List<LineItem> orderlines;
orderlines = new ArrayList<OrderLine>();
```

  - The **List** interface, and the advantage of declaring an object reference variable as an interface type will be discussed further in a later week (week 11 – Java’s Collection API)

Empty interfaces

- Sometimes interfaces contain no methods but indicate that objects of the class implement a particular functionality or role
- implements Serializable
  - indicates that object input and object output is permitted for instances of that class (and all subclasses)
- Implements Cloneable
  - indicates to the **Object.clone()** method that it is legal for that method to make a field-for-field copy of instances of objects of this class.
- The name of interfaces is typically '…-able' to indicate that implementing classes have certain abilities:
  - Examples from Java API: Clone**able**, Serializ**able**, Compar**able**

Comparing abstract classes and interfaces

	Abstract Class	Interface
Declaration	public abstract class	public interface
Derived class declaration clause	extends	implements
Abstract methods	At least one with abstract modifier	All implicit – do not need the modifier
Instance variables	Normal	Not allowed
Constants (final static variables)	Explicit public static final	Implicit – only type modifier required
Derived classes can	Extend only one base class	Implement any number of interfaces

Zoo Demo

- The **zoo-demo** program on Moodle contains the code for a demonstration of many concepts from today
- The program instantiates a variety of animals at the Zoo.
- Purpose of demo is to show:
  - Inheritance
  - Abstract classes and methods
  - Interfaces
  - Consolidate previous weeks' concepts with new concepts from this week

Inner Classes

- Just as you can define methods and attributes inside the definition of a class, you can also define other classes inside an enclosing class:

```
public class Customer
{
     public class Address
     {
            public int number;
            public String roadName;
            public String town;
            public int postCode;
     }
     public String name;
     public Address address;
}
```

Inner Classes (2)

- Inner classes may be declared as static or non-static.
  - Default is non-static
- For non-static inner classes:
  - The instance of the inner class ‘belongs’ to the object which instantiates it.
  - Both the object of the inner class type and the enclosing object which has the instance of the inner type have full access to all private (and public) elements of each other – more privileges.
- For static inner classes:
  - The instance of the inner class is not attached to the enclosing object
  - Therefore, no access to the private methods of the enclosing object

Anonymous Objects

- **Anonymous objects** are objects which are instantiated but not assigned to a variable
- They may be passed as an argument
- Example:

$$\texttt{LibraryCatalogue.addLibraryItem(}\underbrace{\texttt{new Book("Title", "Author", ...)}}_{\textit{Anonymous Object}}\texttt{)}$$

Anonymous Classes

- **Anonymous classes** are *classes* which have no name for their type
- Example use: there is a method which expects an object of some interface; the interface only has 1 method; rather than write a class in its own file, you provide the class definition *as an argument* when invoking the method, i.e. with the (…)
- Usually used in GUI programs – we will wait until then for an example code

Callbacks – an application of interfaces

- Messages between objects are one of two types:
- **Synchronous** – the sender waits for the receiver to complete (and possibly returns a result).
- **Asynchronous** – the sender does not wait for the receiver to return its result, but rather progresses with its own work.
  - The result will come later, at an unexpected time in the future

- All programming you have done so far in this unit is synchronous.
- Asynchronous messages are needed for ***event-driven programming*** such as Graphical User Interface systems

Callback Mechanism

- Requires two classes which have specific roles to play:
- **Publisher class** – announces events that have occurred
- **Subscriber class** – wants to be informed that events have occurred so that it may do something in response.

- Example:

  You arrive at the Bank and take a number, then wait for the number to be called. The tellers announce the number that is next in line. Eventually one of the tellers calls out your number. You respond to this ***notification*** because it is your number, and you ignored the other numbers. You are a ***subscriber***. The tellers are a ***publisher*** – they publish the fact of whose turn it is to be served.

Events

- An *event* is something that is observable, that occurs, and about which there may be additional information
- Examples:
  - The door was closed
  - The door was opened
  - The power turned off
- Software Examples
  - The mouse's left-button was clicked
  - The keyboard's F5 key was pressed
  - The customer completed ordering their 100th Order with our company
  - The connection to the database server was lost
  - Someone has now posted on your Facebook wall.

Callback Mechanism (2)

The process for implementing callbacks is:

1. Subscriber object must inform the publisher object that it wants to receive notification of a future event:
   - called ***subscribing***.
2. At some later time, when the Publisher has some interesting fact, it will notify all subscribers by raising an event:
   - called ***publishing***
3. Subscriber can inform the publisher to no longer send notifications of future events:
   - called ***unsubscribing***

Employing a callback mechanism

1. Identify a Publisher class which will generate events
2. Design an interface for the Subscriber classes
3. Implement specific Subscriber classes
    - They may respond different to each other for same events
4. Design the subscription and unsubscription mechanism (in the Publisher)
5. Design the publishing mechanism
6. Write code which creates the publisher and the subscriber objects, and which registers the subscribers with the publisher

Example: People looking for a house to rent

- Individual people will register their interest with one or more estate agencies. When a new property becomes available at an agency, the agency will immediately tell all the people who have registered, so the people can decide if they want to inspect the house.

- Subscribers: People who want to find a home
- Publishers: The real estate agencies
- Events the subscribers are interested in:
  - A new house has just been listed

Example 2: Monitoring levels of stock

- A Supermarket has some products on display for purchase. They have additional stock warehoused out the back. The quantity on shelves and quantity out the back are tracked. As a customer purchases an item at the cash register, the shelf-counts are decreased. When it is determined that the amount of stock remaining on the shelf has got too low, the staff are alerted so that more stock can be taken out of the warehouse to the shelf.
- Subscribers: Staff
- Publishers: Individual ProductTypes (that represent types of products that are sold in the store; not the individual items on the shelf)
- Events the subscribers are interested in:
  - Quantity on shelf has gone below the level deemed acceptable.

Designing an Interface for subscribers

- Decide on the notifications which the Subscribers want to be informed about (Publisher will announce).
- Write an interface to specify methods for each possible notification
- Examples:

  public void quantityOnHandTooLow(ProductType what, int quantityOnHand) – notification that there is not enough stock on-hand of the specified kind of product

  - A staff member will go to address the problem

  public void productRestocked(ProductType what, int quantityOnHand) – notification that the stock level has now increased due to re-stocking

  - May mean other staff that were advised of it being low, now stop worrying.

Designing a publisher class

- Needs a method to allow subscribers to subscribe
- Needs a method to allow unsubscription
- Needs an array to remember all subscribers
  - The elements must all be of the same type. The type must be the interface described on the previous slide.
- Needs a method to generate notifications of interesting events
  - This method must send the same notification to all subscribers.

- StoreInventory class would play the role of being the publisher.
  - Individual ProductType objects would inform the StoreInventory to cause the notifications to be sent to subscribers.

Provide concrete subscriber classes

- Write classes which are interested in receiving notifications
- These classes must be declared as implementing the subscriber interface
  - So that the publisher can invoke the notification methods of these classes
  - Needs to be an interface (or an abstract class) because the publisher won't necessarily know what specific subclass the subscriber is.
- Implement the methods of the interface
  - Provides the specific response to the messages

Register the subscribers with the publisher

- Code in a driver class should first create the instance of the Publisher, and the instance(s) of the Subscriber(s)
- The subscriber instances should be given to the publisher:
- Example:

```
// publisher:
StoreInventory inventoryMonitor = new StoreInventory ();
// subscriber:
ShelfStackerStaff ssStaff = new ShelfStackerStaff();
// register the subscriber with the publisher
inventoryMonitor.addWatcher(ssStaff);
```

- *When the stock level later goes below a threshold level, the **ssStaff** object will have its **quantityOnHandTooLow** method invoked by inventoryMonitor*