# Principles of Object-Orientation I

- Abstraction
- Separation
- Encapsulation
- Information Hiding

# 1) Abstraction

**What is Abstraction?**

Any model that includes the most important, essential, or distinguishing aspects of something while suppressing or ignoring less important, immaterial, or diversionary details.

- *Dictionary of Object Technology*

Abstraction is a mechanism and practice to reduce and factor out details so that one can focus on few concepts at a time.

- *Wikipedia*

Abstraction denotes a model, a view, or some other focused representation for an actual item. It's the development of a software object to represent an object we can find in the real world.

- *Data Abstraction Defined*

An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.

- *Object-Oriented Design With Applications*

# Abstraction (cont.)

**Abstraction Specifics**

Abstraction is a design technique that focuses on the essential aspects of an entity and ignores or conceals less important or non-essential aspects. Abstraction is an important tool for simplifying a complex situation to a level where analysis, experimentation, or understanding can take place.

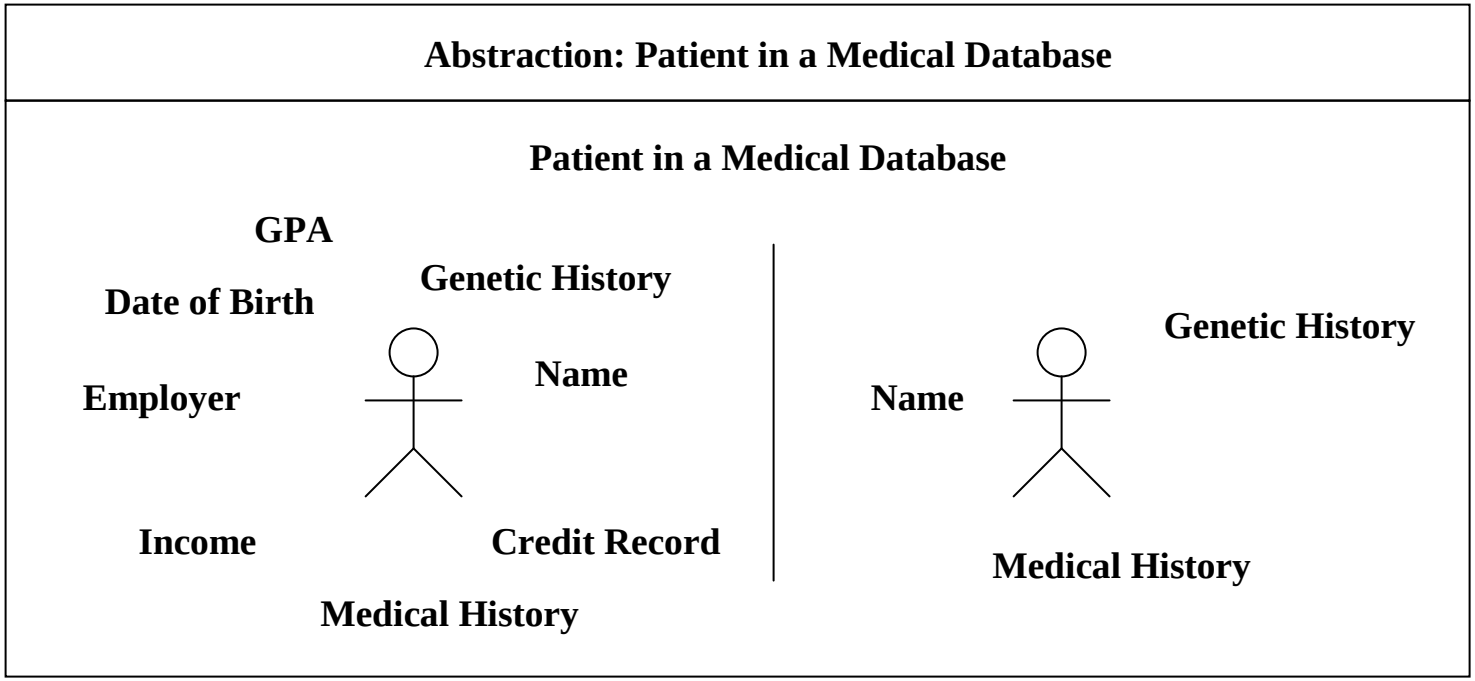
The objects in an object-oriented system are often intended to correspond directly to entities in the "real world". Objects such "salesperson" and "automobiles" that might occur in an automobile dealership tracking system correspond to the actual people on the staff of the dealership and the actual cars owned and sold by the dealership.

The correspondence between the software objects and the real-world entity that they represent is also expressed as the "program" being a "simulation" or " model" of the real world - changes in one being reflected in the other. A "good" program is one which models or simulates accurately what is happening in the real world.

## Abstraction (cont.)

A single entity may have many valid abstractions. While genetic history is not relevant to a sales tracking system, it may be relevant to a medical database system.

Correspondingly, the medical database system developer would not consider the GPA or Credit Record to be a relevant aspect. The name of the abstraction is useful to distinguish among different abstractions for the same entity and among abstractions for different entities.



# Abstraction (cont.)

### Properties of a Good Abstraction

While there may be many abstractions of the same entity, each abstraction should have certain properties that distinguish it as a "good" abstraction. These desirable properties are:

**Well-named**

The nature of an abstraction is conveyed first by the name given to that abstraction. An abstraction is well named if the meanings, intuitions, impressions, and expectations implied by a name accurately reflect the nature of the abstraction.

Whether a name is meaningful depends on the community of people who will use the abstraction. In some cases the name might be a technical term in an application domain that communicates perfectly an abstraction to the group of people in that application area but may mean little to a non-technical group. (i.e., InternalizationOrder, TagSheet, etc.). In other cases, abstraction for widely known entities (i.e., Automobile, Student, etc.) may have a name recognizable to a general population.

# Abstraction (cont.)

**Properties of a Good Abstraction**

**Coherent**

The abstraction should contain a related set of attributes and behavior that make sense from the viewpoint of the modeler. The attributes and behavior have to be what is needed and expected in a given setting.

For example, defining a SalesPerson abstraction that consists of the attributes "commisionRate", "family", and "talents" is not a coherent abstraction, it does not make sense from the viewpoint of a designer building a sales tracking system.

# Abstraction (cont.)

## Properties of a Good Abstraction (cont.)

### Minimal

The abstraction should not contain extraneous attributes or behavior inappropriate for the purpose for which it is defined. For example, adding a mailAddress or telephoneNumber attribute to the SalesPerson abstraction would be extraneous if these additional attributes were not required for the sales tracking system.

### Complete

The abstraction should contain all of the attributes and behavior necessary to manipulate the abstraction for its intended purpose. Assuming that the sales tracking system needed to know the “commisionRate” for each SalesPerson, then an abstraction that did not include this attribute would not be complete.

These properties are clearly ones that required a substantial amount of judgment. This fact implies that the ability to form good abstractions also requires good judgment gained by practice and experience.

## 2) Separation

**Separation in Software Design is the Separation of an Interface from an Implementation.**

- The interface is viewed as the visible, external aspect of the software that must be understood to use the software.
- The implementation is viewed as the hidden, internal aspect of the software that is important only to the implementer.
- An implementation satisfies an interface if the behavior defined in the interface is provided by the implementation.

**The Interface-Implementation Separation Appears at Many Different Levels.**

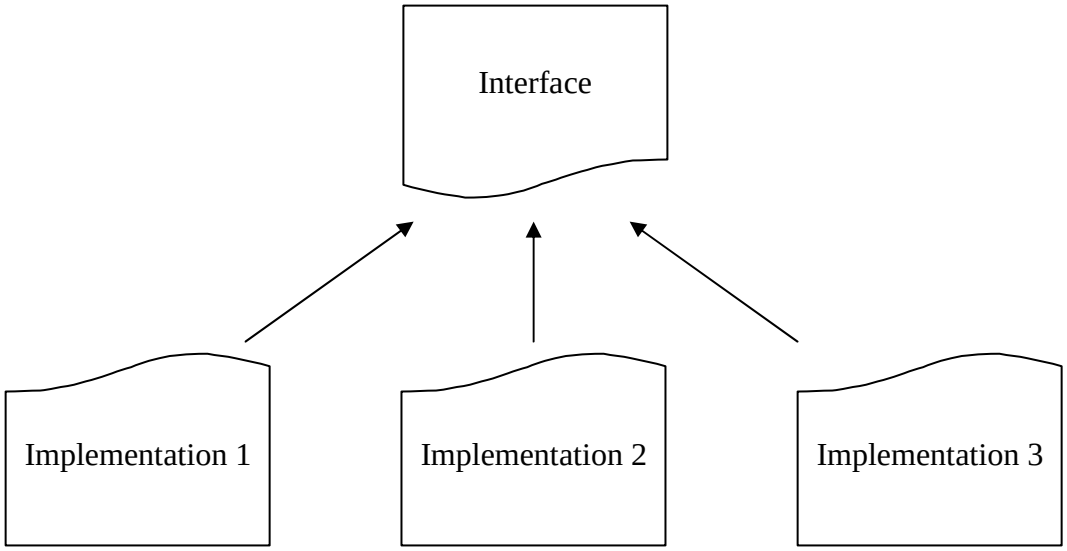
- Manual pages for libraries describe only the interface properties of individual operations *without* describing how any of the operations is implemented.

- A more complex layer of software (e.g., a windowing system, or a networking environment) is an Application Programmer's Interface (API). The API defines what data structures and facilities are available for use by the application programmer *without* defining how the structure and facilities are implemented.

## Separation (cont):

In addition to its simplifying advantages, separation provides flexibility to "implementers" because different implementations may satisfy the same interface. If separation is fully observed, one implementation for a given interface can be replaced by a different implementation of that interface without altering the overall operation of the larger system of which it is a part.

The ability to associate different implementations with the same interface is shown in the following figure.




*The mechanics of creating and using alternate interface implementations will be discussed later in the course*

# 3) Encapsulation

**What is Encapsulation?**

Encapsulation is the ability to provide users with a well-defined interface to a set of functions in a way that hides their internal workings. In object-oriented programming, it is the technique of keeping together data structures and the methods (procedures) that act on them.
*- Free On-line Dictionary of Computing*

Encapsulation is used as a generic term for techniques that realize data abstraction. Encapsulation therefore implies the provision of mechanisms to support both modularity and information hiding. There is a one to one correspondence in this case between the technique of encapsulation and the principle of data abstraction.
*- Blair et al*

Encapsulation is a simple yet effective system-building tool. It allows suppliers to present cleanly specified interfaces around the services they provide. A consumer has full visibility to the procedures offered by an object, and no visibility to its data. From a consumer's point of view, and object is a seamless capsule that offers a number of services, with no visibility as to how these services are implemented ... The technical term for this is encapsulation.
*- Cox*

Encapsulation refers to the practice of including within an object everything it needs, and furthermore doing this in such a way that no other object need ever be aware of this internal structure.
*- Graham*

# Encapsulation

**Encapsulation refers to the bundling of data with the methods that operate on that data.**

Encapsulation supports the principle that users of a software component need to know only the essential details of how to initialize and access the component, and do not need to know the details of the implementation. Encapsulated objects act as a "black box" for other parts of the program that interact with it. They provide a service, but the calling objects do not need to know the details how the service is accomplished.

Examine the following 2 classes. These classes represent the geographical location of a point on the earth's surface and methods for calculating the distance and direction between specified Position objects:

```
public class Position
{
  public double latitude;
  public double longitude;
}
```

```
public class PositionUtility
{
  public static double distance(Position p1, Position p2)
  {
    // Calculate and return the distance between the specified
    // positions. I omit the actual implementation code for the
    // distance calculation to keep the example small.

  }

  public static double direction(Position p1, Position p2)
  {
    // Calculate and return the heading from position1 to
    // position2. I omit the actual implementation code for the
    // direction calculation to keep the example small.
  }
}
```

## Encapsulation (cont.)

The following code represents a typical use of “Position” and “PositionUtility” in a client piece of code

```
[…]

// Create a Position representing a house
      Position aHouse= new Position();
      aHouse.latitude =    36.538611;
      aHouse.longitude = -121.797500;

// Create a Position representing a local coffee shop
      Position coffeeShop = new Position();
      coffeeShop.latitude =    36.539722;
      coffeeShop.longitude = -121.907222;

// Use a PositionUtility to calculate distance and heading from the house
// to the local coffee shop.
      double distance = PositionUtility.distance( aHouse, coffeeShop );
      double heading  = PositionUtility.heading(  aHouse, coffeeShop );

// Print results
      System.out.println( "From the house at (" + aHouse.latitude + ", " + aHouse.longitude +
    ") to the coffee shop at (" + coffeeShop.latitude + ", " + coffeeShop.longitude +
    ") is a distance of " + distance + " at a heading of " + heading + " degrees.");

[…]
```

This code would generate the following output:

```
From my house at (36.538611, -121.7975) to the coffee shop at (36.539722, -121.907222) is
distance of 6.0873776351893385 at a heading of 270.7547022304523 degrees.
```

## Encapsulation (cont.)

However - the “Position” and “PositionUtility” classes are not encapsulated.

Remember our earlier definition of encapsulation:

*Encapsulation is the technique of bundling together data structures and the methods that act on them.*

**Problem:** A programmer must know that to properly work with the associated the geographical data, they must use both classes in order to get the expected functionality. This is poor encapsulation.

The **Position** class contains the *data* needed to represent the geographical location of a point on the earth's surface (latitude and longitude), but on it’s own. the class cannot perform any actions. It has no mechanisms to operate on that data. The methods associated with the data attributes of the Position class are not part of the class. The methods that act on those data attributes are located elsewhere.

The **PositionUtility** class contains the *methods* designed to operate on the latitude and longitude, but it does not actually possess the data attributes that it is designed to work with. All data needed by its methods must be provided from external sources.

## Encapsulation (cont.)

Refactored encapsulated versions of the “Position” and “PositionUtility” classes:
*(Better, but far from perfect)*

```
public class Position
{
    public double latitude;
    public double longitude;

    public double distance( Position position )
    {
           // Calculate and return the distance from this object to the specified
           // position. I omit the actual implementation code for the
           // distance calculation to keep the example small.
    }

    public double direction( Position position )
    {
           // Calculate and return the direction from this object to the specified
           // position. I omit the actual implementation code for the
           // direction calculation to keep the example small.
    }

}
```

In this version of “Position”, the data and related methods are bundled together in one class.

Putting the position data attributes and the methods for calculating distance and direction in the *same* class removes the need for a separate PositionUtility class.

Now the Position class begins to resemble a true object-oriented class.

## Encapsulation (cont.)

The following code *uses* this new version that bundles the data and methods together:

```
[…]
Position myHouse = new Position();
myHouse.latitude =    36.538611;
myHouse.longitude = -121.797500;

Position coffeeShop = new Position();
coffeeShop.latitude =    36.539722;
coffeeShop.longitude = -121.907222;

double distance = myHouse.distance( coffeeShop );
double heading = myHouse.direction( coffeeShop );

System.out.println("From a house at (" + aHouse.latitude + ", " + aHouse.longitude +
    ") to the coffee shop at (" + coffeeShop.latitude + ", " + coffeeShop.longitude +
    ") is a distance of " + distance + " at a heading of " + heading + " degrees.");
[…]
```

By comparison, the above code uses the statement `aHouse.heading(coffeeShop)` to calculate the same heading. The call's semantics clearly indicate that the direction proceeds from a house to the coffee shop.

Converting the two-argument function `heading(Position, Position)` to a one-argument function `position.heading(Position)` is known as *currying* the function.
Currying effectively specializes the function on its first argument, resulting in clearer semantics.

# Encapsulation (cont.)

**Encapsulation Issues**

Note that encapsulation guarantees neither data protection nor information hiding. Nor does encapsulation ensure a cohesive class design. To achieve those quality design attributes requires techniques beyond the encapsulation provided by the language.

As currently implemented, class Position doesn't contain superfluous or non-related data and methods, but Position does expose both latitude and longitude in raw form. That allows any client of class Position to directly change either internal data item without any intervention by Position.

Clearly, encapsulation is not enough.

# 4) Information Hiding

**What is Information Hiding?**

In programming, the process of hiding the details of an object or function. Information hiding is a powerful programming technique because it reduces complexity. One of the chief mechanisms for hiding information is encapsulation -- combining elements to create a larger entity. The programmer can then focus on the new object without worrying about the hidden details.
- *Webopedia*

In computer science, the principle of information hiding is the hiding of design decisions in a computer program that are most likely to change, thus protecting other parts of the program from change if the design decision is changed. Protecting a design decision involves providing a stable interface that shields the remainder of the program from the implementation (the details that are most likely to change). In modern programming languages, the principle of information hiding manifests itself in a number of ways, including encapsulation and polymorphism.
- *Wikipedia*

The principle of information hiding is central. It says that modules are used via their specifications, not their implementations. All information about a module, whether concerning data or function, is encapsulated with it and, unless specifically declared public, hidden from other modules.
- *Graham*

## Information Hiding

Information hiding involves "hiding" the details of the design, algorithms, data structures, etc. of a class that might possibly change over time. This is done by publicly exposing only those methods whose existence is not likely to change, and then "hiding" (making private) all the remaining implementation details.

This isolates clients from requiring intimate knowledge of implementation details to use a class, and isolates clients from the effects of changing implementation details.

*Information hiding promotes object development as real-world concepts, not code.*

Example – Viewing classes as "code" and not a real world object:

```
public class Vehicle
{
      public double getSpeed() {…}
      public void setSpeed(double s) {…}

      public String getVIN() {…}
      public void setVIN(String s) {…}

      public Engine getEngine() {…}
      public void setEngine(Engine e) {…}

      […]
}
```

```
public class Engine
{
      public double getTemperature() {…}
      public void setTemperature(double s) {…}

      public double getRPM() {…}
      public void setRPM(double e) {…}

      public WaterPump getWaterPump() {…}
      public void setWaterPump() {…}

      […]
}
```

***Why does this represent bad design?***

## Information Hiding (cont.)

**Why does this represent bad design?**

*This is not how the corresponding real-world entities work.*

- Most vehicles do not have a speed dial that is used to set an arbitrary speed, so why allow client code to directly set the vehicle speed with the “setSpeed” method?
- In the “real-world”, you can “access” a vehicle’s VIN number but you cannot change it - so why supply a method that allows client code to set it?
- You cannot arbitrarily access and modify engine parameters – why create an Engine accessor method that would allow just that?
- You cannot simply set the temperature of a vehicle’s engine, nor can you set the vehicle’s RPM directly, so why provide “set” methods for those attributes?

## Information Hiding (cont.)

Example – Viewing classes as “real-world" objects:

```
public class Vehicle
{
      public void setAcceleration(double d) {…}
      public void setBraking(double d) {…}
      public void setDirection(double d) {…}

      public void turnOn() {…}
      public void turnOff() {…}

      […]
}
```

**This Example Reflects a Better Design.**

*This class/interface more closely matches how the corresponding real-world entity works.*

- I can set the vehicle acceleration via the accelerator pedal. Likewise the brakes can be set via the brake pedal.
- The direction the vehicle moves can be set - via the steering wheel.
- The vehicle can be turned on and off via the key & ignition.
- Notice that the engine is no longer accessible. It is private. This is appropriate since there is no way to directly modify engine parameters in a normal vehicle.

## Information Hiding (cont.)

Note that the *complete* Vehicle class might look more like this, containing many "hidden" data attributes and methods that client code cannot see:

The internal details of the structure, behavior and design of the Vehicle class (the implementation) are "hidden" – that is, they are "private" and inaccessible to client code.

```
public class Vehicle
{
	// attributes
	private Point3D location;
	private Point3D destination;
	private double maxSpeed;
	private Engine engine;
	[…]

	// methods
	private Point3D getLocation() {…}
	private void setLocation(Point3D p) {…}

	private Point3D getDestination(){…}
	private void setDestination(Point3D p) {…}

	private double getSpeed() {…}
	private void setSpeed(double d) {…}

	private Engine getEngine() {…}
	private void setEngine(Engine e) {…}

	private void updateStatus() {…}

	public void setAcceleration(double d) {…}
	public void setBraking(double d) {…}
	public void setDirection(double d) {…}

	public void turnOn() {…}
	public void turnOff() {…}

	[…]
}
```

The publicly exposed methods - the "interface" - represents the "real-world" behavior that client code works with.

Everything else is unavailable and should not be known to client code.

# Information Hiding (cont.)

## Related Information

### Attribute Access

**Accessors:** A method that accesses ("gets") and return the value of a data attribute.

**Modifiers:** A method that modifies ("sets") the value of a data attribute. Modifiers commonly perform error checking to insure that bad incoming values are not accepted.

### Visibility/Protection

**Public:** The attribute or method is accessible anywhere the containing class is accessible. This is the least restrictive type of access control.

**Private:** The attribute or method is never accessible, except within the class itself. This is the most restrictive type of access control.

**Protected:** The attribute or method is accessible within the body of any subclass of the class, regardless of the package in which that subclass is defined. This is more restrictive than public access, but less restrictive than package access.

*If an attribute or method is not declared with any of these modifiers, it has the default package access: it is accessible to code within all classes that are defined in the same package, but inaccessible outside of the package.*

# Information Hiding (cont.)

**Hiding Information - Myth**

Amateur designers assume that because they make their data attributes private and create public accessor and modifiers – their classes are encapsulated and their "information" is "hidden".

This approach can occasionally be adequate for native (non-reference) data attributes, but it is completely unacceptable for reference-type data attributes. Allowing client code access to an object's data attributes gives client code access to the data and objects that those attributes refer to.

It is *not* a good idea to blindly create public accessor and modifier methods for an object's data attributes. This can inadvertently expose the implementation details of your objects. The creating of public accessor and modifier methods should only be done if the functionality of these methods reflects the true behavior of the real-world entity. You should strive to keep your method visibility as tightly controlled as possible.

- Allowing client code to access the "internals" of some class (i.e., the other objects it may refer to) allows the client code to invoke methods on those objects.

- Allowing client code to manipulate objects in this fashion can result in inconsistencies within those objects and the object that owns them.

## Information Hiding (cont.)

**Example of Poor Information Hiding:**

```
   public class Truck
   {
      private ArrayList cargoContents;

      […]

      // Accessor
      public ArrayList getCargoContents() { return cargoContents; }

      // Modifier
      public void setCargoContents(ArrayList a) throws Exception
      {
         if (a == null) throws new Exception(…);
         cargoContents = a;
      }
      […]
}
```

The Truck class as designed above would allow client code to do the following:

```
myTruck.getCargoContents().clear();
```

-and-

```
myTruck.setCargoContents(new ArrayList());
```

## Information Hiding (cont.)

```
myTruck.getCargoContents().clear();
```

-and-

```
myTruck.setCargoContents(new ArrayList());
```

This is dangerous, and does not make sense in real-world terms. Though encapsulated, the information is *not* hidden

Why would some non-Truck piece of code be clearing (emptying out) the cargo contents of the Truck class?

That would appear to be behavior particular to the Truck itself. This code allows client code to modify or empty the truck's contents without the truck ever knowing about it.

Worse, the entire "cargoContents" ArrayList (representing the Truck's contents) could be over-written by calling the "setCargoContents" method. Invoking this method once the "cargoContents" ArrayList has been set (and then used) would completely wipe out the previous ArrayList.

This reflects poor design – the Truck should be responsible for maintaining it's own contents.

# Information Hiding (cont.)

**Encapsulation is Not Information Hiding**

*If encapsulation was "the same thing as information hiding," then one might make the argument that "everything that was encapsulated was also hidden." This is obviously not true. For example, even though information may be encapsulated within record structures and arrays, this information is usually not hidden (unless hidden via some other mechanism).*
*- Berard*

The term encapsulation is often considered to be interchangeable with information hiding. However, not differentiating between these two important concepts deprives Java developers of a full appreciation of either.

Encapsulation is a language facility, whereas information hiding is a design principle.

*Recall that encapsulation refers to the bundling of data with the methods that operate on that data.*

Often that definition is misconstrued to mean that the data is somehow hidden. You can in fact have encapsulated data that is not hidden at all.

# Information Hiding (cont.)

**Information Hiding is More Than Hiding Your Data Attributes**

Hiding data is not the full extent of information hiding. Hiding design, algorithms, data structures, etc. which might possibly change isolates clients from requiring intimate knowledge of the design to use a module, and from the effects of changing those decisions. This helps decouple client code from your classes.

Information hiding conceals how an object implements its functionality behind the abstraction of the object's interface.

The Java language encapsulation facility isn't enough to ensure a solid class design. The principle of information hiding stipulates that you shield an object's clients from the internal design decisions made for that class of objects.

When possible, don't even reveal whether an attribute is stored or derived. Client objects don't need to know the difference. An attribute is a quality or characteristic inherent in a class of objects. From the client's perspective, object attributes correspond to responsibilities, not internal data structure.

## Information Hiding (cont.)

Rather than publicly exposing the internal data structures of the Truck via public accessor & modifiers, the class should be changed to reflect a more encapsulated, hidden design:

The internal details of the structure, behavior and design of the Truck class (the implementation) are "hidden" – that is, they are "private" and inaccessible to client code.

```
public class Truck
{
   private ArrayList cargoContents = new ArrayList();

   […]

   // Add some cargo – the Truck decides if that's ok to do
   public void addCargo(Cargo c) throws Exception
   {…}

   // Remove some cargo – the Truck decides if that's ok to do
   public void removeCargo(String id) throws Exception
   {…}

   // Access an individual cargo element – the Truck decides
   // if that's ok to do. This is a questionable thing to do
   // but I'll leave it here as an example.
   public Cargo getCargo(String id) throws Exception
   {…}

   public double getCargoSize() throws Exception
   {…}

   […]
}
```

The publicly exposed methods - the "interface" - represents the "real-world" behavior that client code works with.

Everything else is unavailable and should not be known to client code.

## Information Hiding (cont.)

**Now, this Truck class design *is* encapsulated, and the "information" *is* hidden.**

- Client code can only manipulate the Truck via the publicly exposed interface.
- All internal data structures are completely inaccessible.
- The available methods more truly reflect the behavior of the real-world entity.
- Modifying the internal structure of the Truck class should not affect client classes.

## Summary:

- Abstraction - Abstraction is a design technique that focuses on the essential aspects of an entity and ignores or conceals less important or non-essential aspects. Abstraction is an important tool for simplifying a complex situation to a level where analysis, experimentation, or understanding can take place.

- Separation - Separation refers to distinguishing between a goal or effect and the means or mechanism by which the goal or effect is achieved. This is often stated as separating "what" is to be done from "how" it is to be done.

- Encapsulation - Encapsulation supports the principle that users of a software component need to know only the essential details of how to initialize and access the component, and do not need to know the details of the implementation.

- Information Hiding - Information hiding involves “hiding” the details of the design, algorithms, data structures, etc. of a class that might possibly change over time.