MC322 - Object Oriented Programming
Lesson 1
Introduction to Object-Oriented Concepts

Prof. Marcos M. Raimundo
Instituto de Computação - UNICAMP

Motivation

Object-Oriented Adoption

- Object-oriented (OO) software development has existed since the early 1960s.
- The Software industry can be slow-moving at times, mainly because if systems work fine, there must be a compelling reason to replace them.
- Many non-OO legacy systems (that is, older systems already in place) are doing the job—so why risk potential disaster by changing them?
- There is nothing inherently wrong with systems written in non–OO code.
- However, brand-new development warrants the consideration of using OO technologies

Impact of Internet on Object-Oriented Development

- Steady and significant growth in Object-Oriented (OO) development in the past 15 years.
- The reliance on the Internet has catapulted OO development further into the mainstream.
- Emergence of day-to-day business transactions on the Internet has opened a new arena for software development.
- Much of the software development in this arena is new and mostly unencumbered by legacy concerns.

Object Wrappers

Object wrappers are object-oriented code that includes other code inside. For example, you can wrap a structured module inside an object to make it look like an object. You can also use object wrappers to wrap functionality such as security features, non-portable hardware features, etc.

Object-Oriented Development

- Today, one of the most interesting areas of software development is the marriage of legacy and Internet-based systems. In many cases, a web-based front-end ultimately connects to data on a Mainframe. Developers who combine mainframe and web development skills are in demand.
- Objects have certainly made their way into our personal and professional information systems (IS) lives—and they cannot be ignored. You probably experience objects in your daily life without even knowing it. These experiences can take place in your car, talking on your cell phone, using your digital TV, and many other situations.
- With the success of Java, Microsoft's .NET technologies, and many others, objects are becoming a significant part of the technology equation. With the explosion of the Internet and countless local networks, the electronic highway has, in essence, become an object-based highway (in the case of wireless, object-based signals).
- As businesses gravitate toward the Web, they are gravitating toward objects because the technologies used for electronic commerce are mostly OO in nature.

Moving from Procedural to Object-Oriented Development

Procedural Development

In procedural programming, for example, code is placed into distinct functions or procedures. Ideally, as shown in Figure, these procedures then become "black boxes," where inputs go in and outputs come out. Data is placed into separate structures and is manipulated by these functions or procedures.

(a) Black-box OO behavior.

(b) Global data in procedural.

Difference Between OO and Procedural

In OO design, the attributes and behaviors are contained within a single object, whereas in procedural or structured design, the attributes and behaviors are normally separated.

Moving from procedural to OO - Why and how?

- Instead of replacing other software development paradigms, objects represent an evolutionary response.
- Structured programs utilize complex data structures, such as arrays.
- In languages like C++, structures share many characteristics with objects (classes).
- However, objects are more than just data structures and primitive data types like integers and strings.
- Objects contain data (attributes) and behavior (methods).
- Methods in objects are used to perform operations on the data and other actions.
- Importantly, access to members of an object (attributes and methods) can be controlled.
- This means that some members can be hidden from other objects.
- For example, an object called Math might contain integers myInt1 and myInt2, with methods to set and retrieve their values and a method like sum() to add them together.

OO Development - General Aspects

- Before delving deeper into the advantages of Object-Oriented (OO) development, let's consider a fundamental question: What exactly is an object?
- This question is both complex and simple.
  - **Complex:** Learning any software development method is not trivial.
  - **Simple:** People already think about objects.
- When you look at a person, you see the person as an object.
  - An object is defined by two terms: attributes and behaviors.
  - A person has attributes such as eye color, age, height, etc.
  - A person also has behaviors, such as walking, talking, breathing, etc.
  - In its basic definition, an object is an entity that contains data and behavior.
  - The key difference between OO programming and other methodologies is that objects contain data and behavior.

Data Hiding

In OO terminology, data is referred to as attributes, and behaviors are referred to as methods. Restricting access to certain attributes and/or methods is called data hiding.

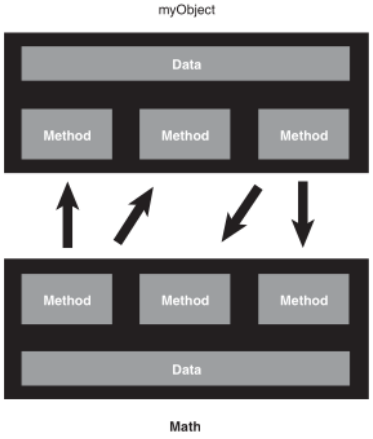
With encapsulation, we can control access to the data in the Math object. By defining these integers as off-limits, another logically unconnected function cannot manipulate the integers myInt1 and myInt2—only the Math object can do that.

Sound Class Design Guidelines

Creating poorly designed OO classes that do not restrict access to class attributes is possible. The bottom line is that **you can design bad code just as efficiently with OO design as with any other programming methodology**.

OO Development - Object Communication and Responsibility

- When another object, such as myObject, wants to gain access to the sum of myInt1 and myInt2, it sends a message (essentially a call) to the Math object: it calculates the sum and returns the value to myObject.
- **The beauty of this design is that myObject doesn't need to know how the sum is calculated.**
- You can change how the Math object calculates the sum without impacting myObject.
- Using a simple calculator example illustrates this concept: when using a calculator, you only interact with its interface—the keypad and LED display—without knowing how the sum is calculated internally.



What Exactly Is an Object?

Understanding Objects in Object-Oriented Programming

Objects are the fundamental building blocks of an Object-Oriented (OO) program with **data** and **behavior**. An OO program being is essentially a collection of objects.

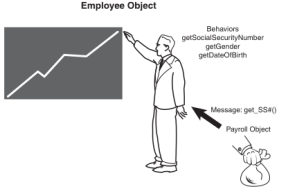
Object Data

- **Attributes** (data) stored within an object represents its state.
- For example, employee objects might have attributes like Social Security numbers, date of birth, gender, phone number, etc.
- Attributes differentiate between objects.

Employee Attributes

Object Behaviors

- In procedural languages, behaviors, procedures, functions, and subroutines define.
- In OO programming, behaviors (what the object can do) are encapsulated in methods.
- You invoke a method by sending a **message** to the object.



Employee behaviors

Object Behaviors: Methods

- In our employee example, one of the behaviors required of an employee object is to set and return the values of its attributes.
- Each attribute would have corresponding methods, such as setGender() and getGender().
- When another object needs information about an employee, it can send a message to the employee object and ask it what its gender is.

Getters and Setters

The concept of getters and setters supports the concept of data hiding. Because other objects should not directly manipulate data within another object, the getters and setters pro vide controlled access to an object's data. Getters and setters are sometimes called accessor methods and mutator methods, respectively.

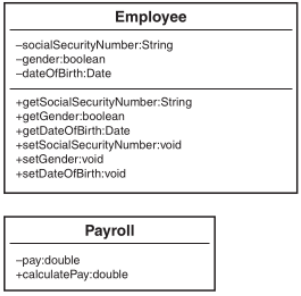
Method Interface and Implementation

```
public class Person {
    // Interfaces
    public String getName();

    public void setName(String n);

    public String getAddress();

    public void setAddress(String adr);
}
```

- When using methods, we typically only show their interface, not their implementation.
- Users need to know:
  - The name of the method
  - The parameters passed to the method
  - The return type of the method

Method Interface and Implementation

- For example, a Payroll object contains a method called CalculatePay() to calculate the pay for an employee.
- To obtain the Social Security number of an employee, the Payroll object sends a message to the Employee object using the getSocialSecurityNumber() method.
- This means that the Payroll object calls the getSocialSecurityNumber() method of the Employee object, which then returns the requested information.



Employee and payroll class diagrams

UML Class Diagrams

Visual modeling tools provide a mechanism to create and manipulate class diagrams using the Unified Modeling Language (UML). Because this is the first class diagram we have seen, it is fundamental and lacks some of the constructs (such as constructors) a proper class should contain.

What Exactly Is a Class?

Understanding Classes

- In short, a class serves as a blueprint for an object.
- When you instantiate an object, you use a class as the basis for how the object is built.
- Explaining classes and objects is akin to a chicken-and-egg dilemma: describing a class without mentioning objects is challenging, and vice versa.
- For instance, consider a specific bike as an object. However, the blueprints (i.e., the class) were necessary to construct the bike.
- In Object-Oriented (OO) software, unlike the chicken-and-egg dilemma, we know what comes first: the class.
- An object cannot be instantiated without a class, making the class fundamental.
- Many concepts discussed earlier, such as attributes and methods, are similar when discussing classes and objects.
- To further illustrate classes and methods, let's consider an example from the relational database world:
  - In a database table, the table's definition (fields, description, data types) acts as a class (metadata).

Understanding Classes

- A class can be conceptualized as a higher-level data type.
- Similar to how you declare variables of basic data types like integer or float:
  - `int x;`
  - `float y;`
- You can create an object by using a predefined class:
  - `myClass myObject;`
- In this example, `myClass` represents the class and `myObject` represents the object.
- Each object has its own attributes (similar to fields) and behaviors (similar to functions or routines).
- A class defines the attributes and behaviors that all objects created with this class will possess.
- Classes are pieces of code, and objects instantiated from classes can be distributed individually or as part of a library.
- Since objects are created from classes, classes must define the basic building blocks of objects (attributes, behavior, and messages).

Understanding Classes: Example of a Person Class

```
public class Person {
    // Attributes
    private String name;
    private String address;

    // Methods
    public String getName() {
        return name;
    }

    public void setName(String n) {
        name = n;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String adr) {
        address = adr;
    }
}
```

Attributes and Access Designations

- **Attributes:**
  - Attributes represent the data of a class.
  - Each class must define attributes to store the state of objects instantiated from that class.
  - For example, in the Person class example, attributes for name and address are defined.

Access Designations

When a data type or method is defined as **public**, other objects can directly access it.

When a data type or method is defined as **private**, only that specific object can access it.

Another access modifier, **protected**, allows access by related objects.

Methods in Object-Oriented Programming

- **Methods:**
  - Methods implement the required behavior of a class.
  - Every object instantiated from a class has the methods as defined by the class.
  - Methods may implement behaviors that are called from other objects (messages) or provide the internal behavior of the class.
  - Internal behaviors are private methods that are not accessible by other objects.
- **In the Person class:**
  - Behaviors include getName(), setName(), getAddress(), and setAddress().
  - These methods allow other objects to inspect and change the values of the object's attributes.
  - Controlling access to attributes within an object should be managed by the object itself—no other object should directly change an attribute of another.

Messages: Communication Between Objects

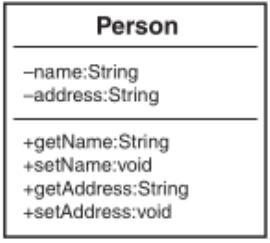
- **Messages:**
  - Messages are the communication mechanism between objects.
  - When Object A invokes a method of Object B, Object A is sending a message to Object B.
  - Object B's response is defined by its return value.
  - Only the public methods, not the private methods, of an object can be invoked by another object.
- **Example Code:**

```
public class Payroll {
    String name;
    Person p = new Person();
    String result = p.setName("Joe");
    ... code
    String name = p.getName();
}
```

- In this example, the Payroll object is sending a message to a Person object to retrieve the name via the getName method.
- Focus on understanding the concepts rather than the specific code details.

Using UML to Model a Class Diagram

- **Unified Modeling Language (UML):**
  - Over the years, many tools and modeling methodologies have been developed to assist in designing software systems.
  - One of the most popular tools in use today is Unified Modeling Language (UML).
  - UML class diagrams are used to illustrate the classes that we build.
- **Attributes and Methods in UML:**
  - Attributes and methods are typically separated in UML class diagrams.
  - Attributes are shown at the top, and methods at the bottom.
- As we delve more deeply into Object-Oriented (OO) design, these class diagrams will become more sophisticated and convey more information on how different classes interact.



The Person class diagram

Encapsulation and Data Hiding

Encapsulation in Object-Oriented Design

- **Advantages of Objects:**
  - One primary advantage of using objects is that they need not reveal all their attributes and behaviors.
  - In good Object-Oriented (OO) design, an object should only reveal the interfaces that other objects must have to interact with it.
  - Details not pertinent to the use of the object should be hidden from all other objects.
- **Encapsulation:**
  - Encapsulation is defined by the fact that objects contain both attributes and behaviors.
  - Data hiding is a major part of encapsulation.
  - For example, an object that calculates the square of a number must provide an interface to obtain the result, but the internal attributes and algorithms used for the calculation need not be made available to other objects.
  - Robust classes are designed with encapsulation in mind.
- Next, we cover the concepts of interface and implementation, which are the basis of encapsulation.

Interfaces in Object-Oriented Design

- Interfaces define the fundamental means of communication between objects in Object-Oriented (OO) design.
- Each class design specifies interfaces for the proper instantiation and operation of objects.
- Any behavior the object provides must be invoked by a message sent using one of the provided interfaces.
- The interface should completely describe how users of the class interact with the class.

Private Data

For data hiding to work, all attributes should be declared as private. Thus, attributes are never part of the interface. Only the public methods are part of the class interface. Declaring an attribute as public breaks the concept of data hiding.

In the example of the square of a number, the Interface must show how to (1) instantiate a Square object and (2) send a value to the object and get the square of that value in return

Interfaces in Object-Oriented Design

- If a user needs access to an attribute, a method is created to return the value of the attribute (a getter).
- If a user wants to obtain the value of an attribute, a method is called to return its value.
- The object that contains the attribute controls access to it.
- Vital importance in security, testing, and maintenance.
- Control over access to attributes simplifies problem tracking and maintenance.
- Uncontrolled code should not change or retrieve sensitive data such as passwords and personal information.

Interfaces Versus Interfaces

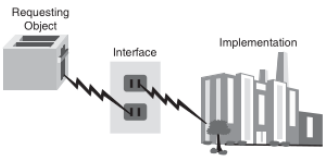
It is important to note that there are interfaces to the classes and the methods—don't confuse the two. The interfaces to the classes are the public methods while the interfaces to the methods relate to how you call (invoke) them. This will be covered in more detail later.

Implementation in Object-Oriented Design

- Only the **public attributes and methods** are considered the interface.
- The user should not see any part of the implementation—interacting with an object solely through class interfaces.
- In the previous example, for instance the **Employee class**, only the attributes were hidden.
- In many cases, there will be methods that also should be hidden and thus not part of the interface.
- Continuing the example of the square root from the previous section, the user does not care how the square root is calculated—as long as it is the correct answer.
- Thus, the implementation can change, and it will not affect the user's code.
- For example, the company that produces the calculator can change the algorithm (perhaps because it is more efficient) without affecting the result.

Interface between Appliance and Power Source

- The toaster requires electricity.
- To get this electricity, the cord from the toaster must be plugged into the electrical outlet, which is the interface.
- All the toaster needs to do to obtain the required electricity is to use a cord that complies with the electrical outlet specifications; this is the interface between the toaster and the power company (actually the power industry).
- The fact that the actual implementation is a coal-powered electric plant is not the concern of the toaster.
- In fact, for all the toaster cares, the implementation could be a nuclear power plant or a local power generator.
- With this model, any appliance can get electricity, as long as it conforms to the interface specification.



Power plant example

A Model of the Interface/Implementation Paradigm

- Let's explore the Square class further.
- Assume that you are writing a class that calculates the squares of integers.
- You must provide a separate **interface and implementation**.
- That is, you must provide a way for the user to invoke and obtain the square value.
- You must also provide the implementation that calculates the square; however, the user should not know anything about the specific implementation.

```
// private attribute
private int squareValue;

// public interface
public int getSquare(int value) {
    squareValue = calculateSquare(value);
    return squareValue;
}

// private implementation
private int calculateSquare(int value) {
    return value * value;
}
```

- public method getSquare is the interface
- implementation of the square algorithm is in the method calculateSquare, which is private
- SquareValue is private because users do not need to know that this attribute exists
- If the implementation were to change, you would not need to change the interface

Inheritance

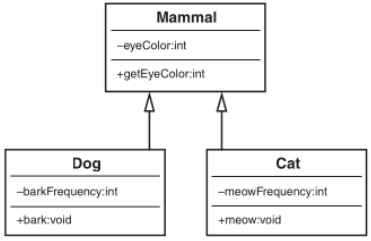
Code Reuse and Inheritance

- One of the most powerful features of OO programming is, perhaps, code reuse.
  - Structured design provides code reuse to a certain extent—you can write a procedure and then use it as many times as you want.
  - However, OO design goes an important step further, allowing you to define relationships between classes that facilitate not only code reuse but also better overall design, by organizing classes and factoring in commonalities of various classes.
- Inheritance is a primary means of providing this functionality.
  - Inheritance allows a class to inherit the attributes and methods of another class.
  - This allows creation of brand new classes by abstracting out common attributes and behaviors.

Factoring Out Commonality and Inheritance

One of the major design issues in OO programming is to factor out commonality of the various classes.

- For example, say you have a Dog class and a Cat class, and each will have an attribute for eye color.
- In a procedural model, the code for Dog and Cat would each contain this attribute.
- In an OO design, the color attribute could be moved up to a class called Mammal—along with any other common attributes and methods.
- In this case, both Dog and Cat inherit from the Mammal class.



Mammal hierarchy

- Dog class has the following attributes: eyeColor (inherited), and barkFrequency;
- and following methods: getEyeColor (inherited), bark;

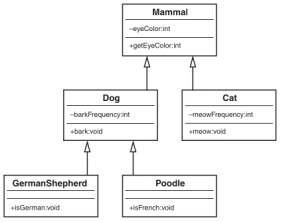
Thus, Dog has all the properties of its class definition, as well as the inherited properties.

Superclasses and Subclasses

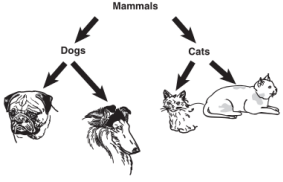
- The superclass, or parent class, contains all the attributes and behaviors that are common to classes that inherit from it.
  - For example, in the case of the Mammal class, all mammals have similar attributes such as eyeColor and hairColor, as well as behaviors such as generateInternalHeat and growHair.
  - All mammals have these attributes and behaviors, so it is not necessary to duplicate them down the inheritance tree for each type of mammal.
  - Duplication requires a lot more work, it can introduce errors and inconsistencies.
  - Dog and Cat classes inherit all common attributes and behaviors from Mammal class.
  - Mammal class is the superclass of the Dog and the Cat subclasses, or child classes.
- Inheritance provides a rich set of design advantages.
  - When designing a Cat class, Mammal class provides the functionality needed.
  - By inheriting from the Mammal object, Cat already has all the attributes and behaviors that make it a true mammal.
  - To make it more specifically a cat type of mammal, the Cat class must include any attributes or behaviors that pertain solely to a cat.

Abstraction

- An inheritance tree can grow quite large.
  - When the Mammal and Cat classes are complete, other mammals, such as dogs (or lions, tigers, and bears), can be added quite easily.
  - The Cat class can also be a superclass to other classes. For example, it might be necessary to abstract the Cat class further, to provide classes for Persian cats, Siamese cats, and so on.
  - Just as with Cat, the Dog class can be the parent for GermanShepherd and Poodle.
- The power of inheritance lies in its abstraction and organization techniques.
- Note that the classes GermanShepherd and Poodle both inherit from Dog. Thus, they also inherit from Mammal.
- Thus, the GermanShepherd and Poodle classes contain all the attributes and methods included in Dog and Mammal.



Mammal UML diagram



Mammal hierarchy

Is-a Relationships

- Consider a Shape example where Circle, Square, and Star all inherit from.
  - This relationship is often referred to as an is-a relationship because a circle is a shape, and Square is a shape.
  - When a subclass inherits from a superclass, it can do anything that the superclass can do.
  - Thus, Circle, Square, and Star are all extensions of Shape.
- When we design this Shape system it would be very helpful to standardize how we use the various shapes.
  - Thus, we could decide that if we want to draw a shape, no matter what shape, we will invoke a method called draw.
  - If we adhere to this decision, whenever we want to draw a shape, only the Draw method needs to be called, regardless of what the shape is.
- Here lies the fundamental concept of polymorphism—it is the individual object's responsibility, be it a Circle, Star, or Square, to draw itself.
- This is a common concept in many current software applications like drawing and word processing applications.

Polymorphism

Polymorphism

- Polymorphism is a Greek word that means many shapes.
- Although polymorphism is tightly coupled to inheritance, it is often cited separately as one of the most powerful advantages of object-oriented technologies.
- When a message is sent to an object, the object must have a method defined to respond to that message.
- In an inheritance hierarchy, all subclasses inherit the interfaces from their superclass. However, each subclass might require a separate response to the same message because each subclass is a separate entity.
- Consider the Shape class and the behavior called Draw. When you tell somebody to draw a shape, the first question is, "What shape?" No one can draw a shape, as it is an abstract concept.
- To specify a concrete shape, you provide the actual implementation in Circle. Even though Shape has a Draw method, Circle overrides with its own Draw() method. Overriding means replacing an implementation of a parent with one from a child.

Polymorphism Example: Shape and Circle Classes

- Consider the Shape example. Even though you treat them all as Shape objects and send a Draw message to each Shape object, the result is different for each because Circle, Square, and Star provide the actual implementations.
- In short, each class can respond differently to the same Draw method and draw itself. This is what is meant by polymorphism.
- The Shape class has an attribute called area that holds the value for the area of the shape. The method getArea() includes an identifier called abstract.
- A subclass must provide the implementation for an abstract method. Thus, Shape requires subclasses to provide a getArea() implementation.

Consider the following Shape class:

```
public abstract class Shape {
    private double area;
    public abstract double getArea();
}
```

Now let's create a class called Circle that inherits from Shape:

```
public class Circle extends Shape {
    double radius;
    public Circle(double r) {
        radius = r;
    }
    public double getArea() {
        area = 3.14 * (radius *
              radius);
        return (area);
    }
}
```

Constructors and Rectangle Class

- We introduce a new concept here called a constructor. The Circle class has a method with the same name, Circle.
- This class is the constructor that is the entry point for the class, where the object is built; the constructor is a good place to perform initializations and start-up tasks.
- The Circle constructor accepts a single parameter representing the radius and assigns it to the radius attribute of the Circle class.
- The Circle class also implements the getArea method, originally defined as abstract in the Shape class.

We can create the class Rectangle:

```
public class Rectangle extends Shape {
    double length;
    double width;
    public Rectangle(double l, double
         w) {
        length = l;
        width = w;
    }
    public double getArea() {
        area = length * width;
        return (area);
    }
}
```

Now we can create any number of rectangles, circles, and so on and invoke their getArea() method.This is because we know that all rectangles and circles inherit from Shape, and all Shape classes have a getArea() method.

Implementing Abstract Methods and Polymorphism

- If a subclass inherits an abstract method from a superclass, it must provide a concrete implementation, or else it will be an abstract class itself.
- This approach also provides the mechanism to create other, new classes quite easily.
- Thus, we can instantiate the Shape classes in this way:

```
Circle circle = new Circle(5);
Rectangle rectangle = new Rectangle(4, 5);
```

- Then, using a construct such as a stack, we can add these Shape classes to the stack:

```
stack.push(circle);
stack.push(rectangle);
```

- Now comes the fun part. We can empty the stack, and we do not have to worry about what kind of Shape classes are in it (we just know they are shapes):

```
while (!stack.empty()) {
    Shape shape = (Shape) stack.pop();
    System.out.println("Area = " + shape.getArea());
}
```

Message Passing and Behavior

- However, the actual behavior that takes place depends on the type of shape. For example, Circle will calculate the area for a circle, and Rectangle will calculate the area of a rectangle. In effect (and here is the key concept), we are sending a message to the Shape classes and experiencing different behavior depending on what subclass of Shape is being used.
- This approach is meant to provide standardization across classes, as well as applications.
- Consider an office suite that includes a word processing and a spreadsheet application. Let's assume that both have a method called Print. This Print method can be part of the Office class as a requirement any class that inherits from it to implement a Print method. The interesting thing here is that although both the word processor and spreadsheet do different things when the Print method is invoked, one prints a processing document and the other a spreadsheet document.

Composition

Objects Containing Other Objects

- It is natural to think of objects as containing other objects.
- A television set contains a tuner and video display. A computer contains video cards, keyboards, and drives.
- Although the computer can be considered an object unto itself, the drive is also considered a valid object.
- In fact, you could open up the computer and remove the drive and hold it in your hand.

Abstraction and Composition

- Just as with inheritance, composition provides a mechanism for building objects. In fact, I would argue that there are only two ways to build classes from other classes: inheritance and composition.
- As we have seen, inheritance allows one class to inherit from another class. We can thus abstract out attributes and behaviors for common classes. For example, dogs and cats are both mammals because a dog is-a mammal and a cat is-a mammal.
- With composition, we can also build classes by embedding classes in other classes.
- Consider the relationship between a car and an engine. The benefits of separating the engine from the car are evident.
- By building the engine separately, we can use the engine in various cars—not to mention other advantages. But we can't say that an engine is-a car.
- This just doesn't sound right when it rolls off the tongue (and because we are modeling real-world systems, this is the effect we want). Rather, we use the term has-a to describe composition relationships. A car has-a(n) engine.

Has-a Relationships

- Although an inheritance relationship is considered an is-a relationship for reasons already discussed, a composition relationship is termed a has-a relationship.
- Using the example in the previous section, a television has-a tuner and has-a video display. A television is obviously not a tuner, so there is no inheritance relationship.
- In the same vein, a computer has-a video card, has-a keyboard, and has-a disk drive.
- The topics of inheritance, composition, and how they relate to each other are covered in great detail in Chapter 7, "Mastering Inheritance and Composition."

Conclusion

Summary of Key Topics

- Encapsulation: Encapsulating the data and behavior into a single object is of primary importance in OO development. A single object contains both its data and behaviors and can hide what it wants from other objects.
- Inheritance: A class can inherit from another class and take advantage of the attributes and methods defined by the superclass.
- Polymorphism: Polymorphism means that similar objects can respond to the same message in different ways. For example, you might have a system with many shapes. However, a circle, a square, and a star are each drawn differently. Using polymorphism, you can send each of these shapes the same message (for example, Draw), and each shape is responsible for drawing itself.
- Composition: Composition means that an object is built from other objects.

This chapter covers the fundamental OO concepts of which by now you should have a good grasp.

MC322 - Object Oriented Programming
Lesson 1
Introduction to Object-Oriented Concepts

Prof. Marcos M. Raimundo
Instituto de Computação - UNICAMP