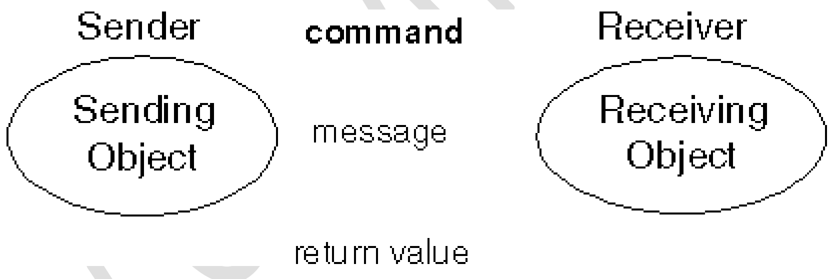
What is Object-Oriented Programming

Object-Oriented Programming (OOP) is different from procedural programming languages (C, Pascal etc.) in several ways. Everything in OOP is grouped as "objects". OOP, defined in the purest sense, is implemented by sending *messages* to *objects*. To understand this concept, we first need to know what an object is.

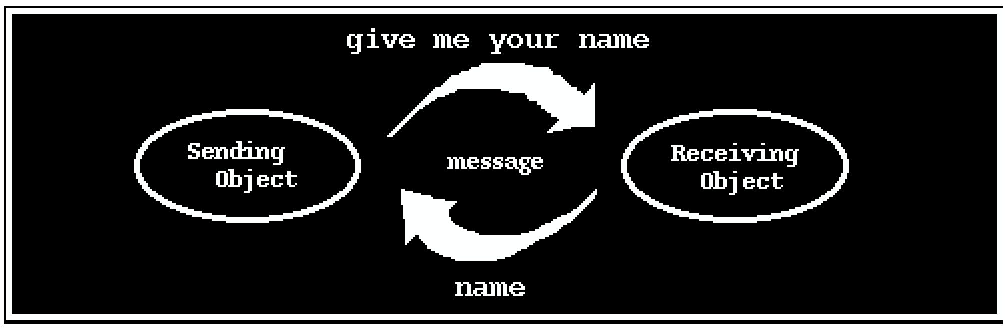
What is an Object?

An object can be considered a "thing" that can perform a set of activities. The set of activities that the object performs defines the object's behavior. For example, a "**StudentStatus**" object can tell you its grade point average, year in school, or can add a list of courses taken. A "**Student**" object can tell you its name or its address.

The object's interface consists of a set of commands, each command performing a specific action. An object asks another object to perform an action by sending it a *message*. The requesting (sending) object is referred to as *sender* and the receiving object is referred to as *receiver*.

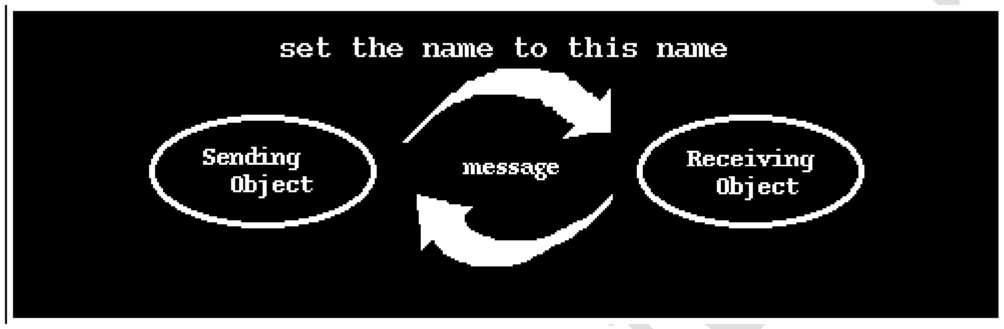


Control is given to the receiving object until it completes the command; control then returns to the sending object. For example, a **School** object asks the **Student** object for its *name* by sending it a *message* asking for its name. The receiving **Student** object returns the *name* back to the sending object.



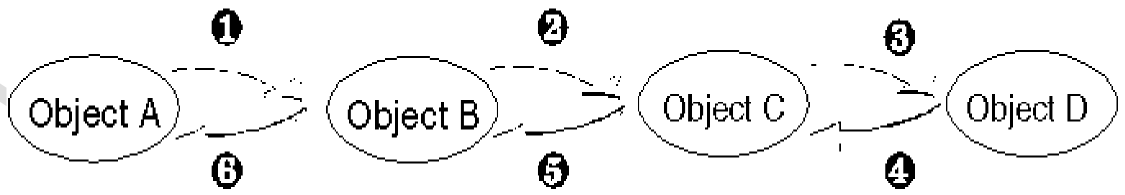
A message can also contain information the sending objects needs to pass to the receiving object, called the *argument* in the message. A receiving object always returns a *value* back to the sending object. This returned value may or may not be useful to the sending object.

For example, the **School** object now wants to change the student's name. It does this by sending the **Student** object a message to set its name to a new name. The new address is passed as an argument in the message. In this case, the **School** object does not care about the return value from the message.



Sequential Operation

It is very common that a message will cause other messages to be sent, either to itself or to other objects, in order to complete its task. This is called *sequential operation*. Control will not return to the original sending object until all other messages have been completed. For example, in the following diagram, Object A sends a message to Object B. For Object B to process that message it sends a message to Object C. Likewise, Object C sends a message to Object D. Object D returns to Object C who then returns to Object B who returns to Object A. Control does not return to Object A until all the other messages have completed.

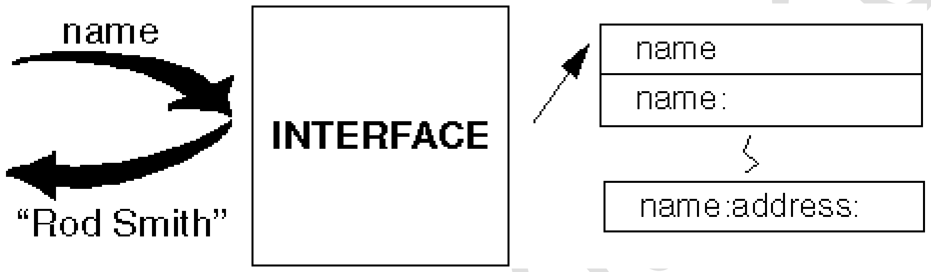


Method

How do receiving objects interpret messages from the senders? How are the messages processed? Each message has code associated with it. When an object receives a message, code is executed. In other words, these messages determine an object's behavior and the code determines how the object carries out each message. The code that is associated with each message is called a *method*. The message name is also called the *method name* due to its close association with the method.

When an object receives a message, it determines what method is being requested and passes control to the *method*. An object has as many methods as it takes to perform its designed actions.

Refer to the following diagram, **name**, **name:**, **address** and **name:address** are method names for the **Student** object. When the **Student** object receives the **name** message, the **name** message passes control to the *name* method defined in **Student**.



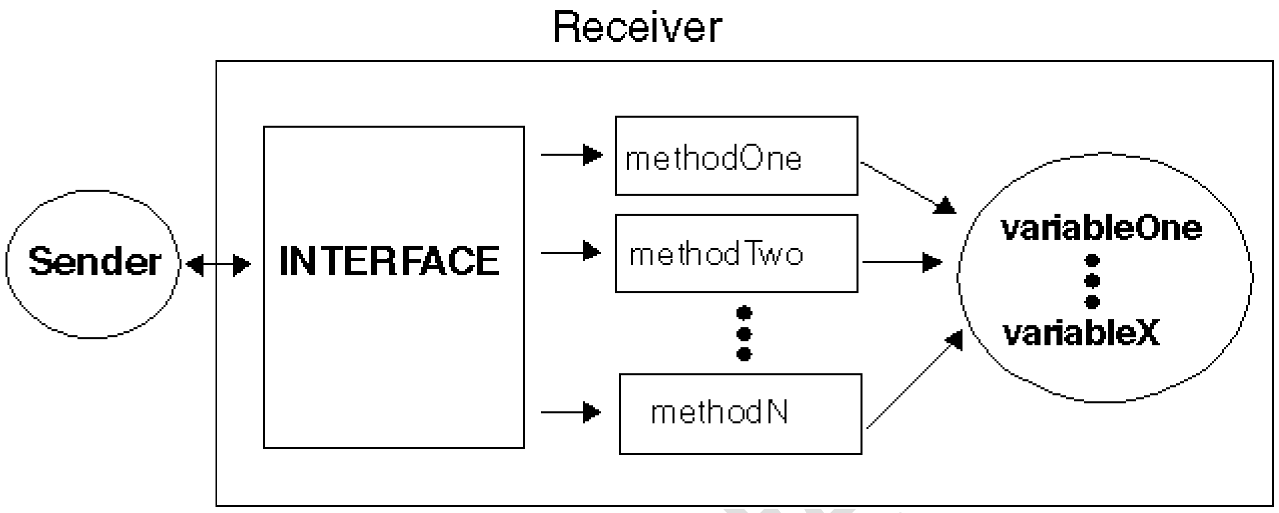
Methods that operate on specific objects are *instance methods* and messages that invoke instance methods are called *instance message*. Methods that operate on specific classes are *class methods*.

Methods are similar to subroutines, procedures, or functions found in procedural languages (C, Pascal). For example, a method name equates to a subroutine name, and the code for the method equates to the code found in a subroutine. Sending a message to an object is similar to calling a subroutine.

Object's Data

Each object need to keep the information on how to perform its defined behavior. Some objects also contain variables that support their behavior. These variables are called *instance variables*. Only the *instance method* for an object can refer to and change the values stored in the instance variables. The instance methods for other objects cannot refer to this object's data. An object may only access another object's data by sending it messages. This is called *encapsulation* and assures that there is a secure process for getting to an object's data.

Refer to the following diagram, the instance variables *variableOne* through *variableX* can only be accessed by the sender via instance methods *methodOne* through *methodX*. The sender can not refer to the variables directly by itself.



Unlike procedural programming where common data areas are often used for sharing information, object-oriented programming discourages direct access to common data (other than the use of *global variables*) by other programs. Only the object that "owns" the data can change its content. Other objects can view or change this data by sending message to the "owner."

The instance variable names can be identical to the method names that associate with them. For example, the **Student** object has methods of **name**, **address**, and **major** as well as instance variables of *name*, *address*, and *major*. Smalltalk distinguishes variable identifier and a method identifier by the identifier's position in the expression.

Object-Oriented Problem Solving Approach

Object-oriented problem solving approach is very similar to the way a human solves daily problems. It consists of identifying *objects* and how to use these *objects* in the correct sequence to solve the problem. In other words, object-oriented problem solving can consist of designing objects whose behavior solves a specific problem. A message to an object causes it to perform its operations and solve its part of the problem.

The object-oriented problem solving approach, in general, can be divided into four steps. They are:

1. Identify the problem
2. Identify the objects needed for the solution
3. Identify messages to be sent to the objects
4. Create a sequence of messages to the objects that solve the problem.

Following is an example of solution to a simple problem using the four steps.

Example 1: Object-oriented Problem Solving Approach

Step 1: *Problem Identification* - Compute the sum of two numbers and print out the result.

Step 2: *Object Identification* - Identify objects needed to solve the problem.

```
Num1 - first number (object).
Num2 - second number (object).
Sum - result of the addition of the two numbers (object).
```

Step 3: *Message Identification* - Messages needed to send to objects.

```
+ aNum - This is the message sent to the receiver
         object with an parameter aNum. The result of
         this message is the value (a numeric object) of the
         total sum of the receiver object and aNum.
print - a message that displays the value of the receiver
        object.
```

Step 4: *Object-message sequences* - Following is the object-message sequence to solve the problem.

```
(Num1 + Num2)print
```

The *message* **+** with a parameter **Num2** (an object), is sent to the object **Num1**. The result of this is an object (Num1 + Num2), which is sent the message **print**. The parentheses are included to avoid any ambiguity as to which message should be activated first.

Note: The sequence of Smalltalk evaluation is from left to right unless is specified by parentheses ( ). For example:

```
                 2 + 3 * 4        is 20
  while          2 +(3 * 4)       is 14
```

Object-Oriented Paradigm

A computer language is object-oriented if they support the four specific object properties called *abstraction*, *polymorphism*, *inheritance,* and *encapsulation*.

Inheritance??!!

Data Abstraction??? **Encapsulation?!!?**

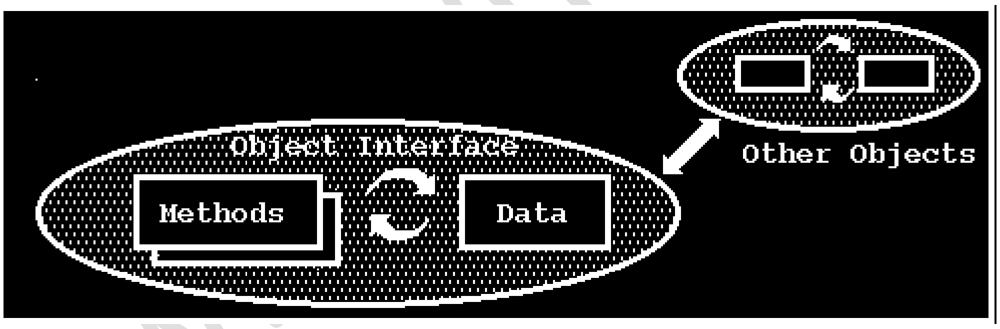
Reuse??? **Polymorphism??**

Smalltalk??!!

The objective of this section is to provide a thorough understanding of the principles of object-oriented paradigm.

Data Abstraction

Encapsulation



In object-oriented programming, objects interact with each other by messages. The only thing that an object knows about another object is the object's interface. Each object's data and logic is hidden from other objects. In other words, the interface *encapsulates* the object's code and data.

This allows the developer to separate an object's implementation from its behavior. This separation creates a "black-box" affect where the user is isolated from implementation changes. As long as the interface remains the same, any changes to the internal implementation are transparent to the user.

For example, if the **name** message is sent to the **Student** object, it does not matter to the user how the developer implemented the code to handle this message. All the sending object needs is the correct protocol for interacting with the **Student** object. The developer can change the implementation at any time, but the **name** message would still work because the interface is the same.

Polymorphism

Another benefit of separating implementation from behavior is *polymorphism*. Polymorphism allows two or more objects respond to the same message. A method called **name** could also be implemented for an object of the class **Course**. Even though the implementation of this name message may return a course number and a course title, its protocol is the same as the **name** message to the **Student** object. Polymorphism allows a sending object to communicate with different objects in a consistent manner without worrying about how many different implementations of a message.

An analogy of polymorphism to daily life is how students response to a school bell. Every student knows the significant of the bell. When the bell (message) rings, however, it has its own meaning to different students (objects). Some students go home, some go to the library, and some go to other classes. Every student responds to the bell, but how they response to it might be different.

Another example of polymorphism is the function of printing. Every printable object must know how to print itself. The message is the same to all the different objects: **print**, but the actual implementation of what they must do to print themselves varies.

The sending object does not have to know how the receiving object implements the message. Only the receiving objects worries about that. Assume that there is a **printPage** method in a **Document** object that has the responsibility of printing a page. To print the page, the **printPage** method sends the **print** message to each object on the page. The **Document** does not need to know what types of objects are on the page, only that each object supports the behavior of printing.

New objects can be added to the page without affecting the **printPage** method. This method still sends the **print** message and the new object provides its own **print** method in response to that message.

Polymorphism allows the sending object to communicate with receiving objects without having to understand what type of object it is, as long as the receiving objects support the messages.

Inheritance

Another important concept of object-oriented programming is *inheritance*. Inheritance allows a class to have the same behavior as another class and extend or tailor that behavior to provide special action for specific needs.

Let's use the following application as an example. Both **Graduate** class and **Undergraduate** class have similar behavior such as managing a name, an address, a major, and a GPA. Rather than put this behavior in both of these classes, the behavior is placed in a new class called **Student**. Both **Graduate** and **Undergraduate** become *subclass* of the **Student** class, and both *inherit* the **Student** behavior.

Both **Graduate** and **Undergraduate** classes can then add additional behavior that is unique to them. For example, **Graduate** can be either Master's program or PhD program. On the other hand, **Undergraduate** class might want to keep track of either the student is freshman, sophomore, junior or senior.

Classes that inherit from a class are called *subclasses*. The class a subclass inherits from is called *superclass*. In the example, **Student** is a superclass for **Graduate** and **Undergraduate**. **Graduate** and **Undergraduate** are subclasses of **Student**.

The Power of Reuse

One of the most important features of object-oriented programming is the ability to modify existing solution to solve new problems. If a particular kind of problem has been solved using the OOP approach, a similar but slightly different problem can usually be solved by making some changes in the object-message protocol that already exist. Most of the time, this requires adding new *messages*. Other cases may require adding new *objects* and new *messages* to which those objects respond.

Reusability is probably the most important and the strongest feature of Smalltalk. Although *procedural languages* can be reuse too, Smalltalk makes programming for reuse much easier.

The above notes are submitted by *Nikhil Tibdewal*.