Object-Oriented Programming

What is an Object?

An ***object*** is some ***private data*** and a set of ***public operations*** that can access that data.

The external interface to the object is ***only*** through the operations.

An object is requested to perform one of its operations by sending it a ***message*** telling the object what to do.

The ***receiver*** responds to the message by first choosing the operation that implements the message name, executing this operation, and then returning control to the caller.

What is Object-Oriented Programming (OOP)?

A code packaging technique.

A top-down design methodology.

A process of creating computer-based objects which are analogs of the real world.

A modelling approach to programming.

Properties of OOP

1) ***Classes of Objects***

2) ***Messaging***

3) ***Polymorphism***

4) ***Inheritance***

5) ***Dynamic Binding***

These basic properties are language independent. However, in the following sections, specifics are drawn from Objective-C.

1. Classes of Objects

(abstraction and encapsulation)

Abstraction:

An object will be defined as a member of a ***class***.

A class is an ***abstraction*** which describes the properties shared by a group of similar objects.

An ***instance*** of a class is an individual occurrence of an object.

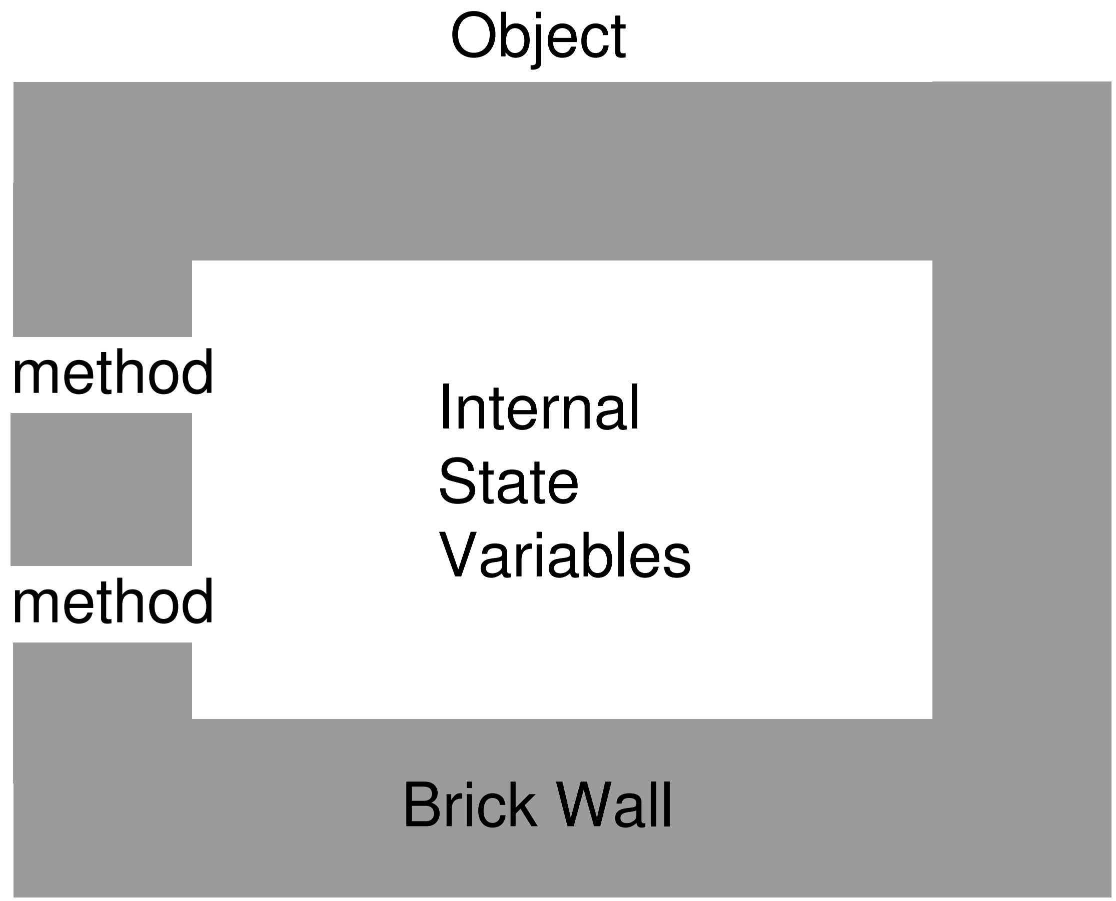
Encapsulation:

A class encapsulates in a single object both data and procedural abstraction, state and behavior.

Encapsulation is done to separate the user of an object from its author.

The object is defined by its ***behavior***, the ***public operations*** it can perform. The ***data remains private*** to the object.

Users of these objects can access the object's data ***only*** using the procedures the object developer provides.



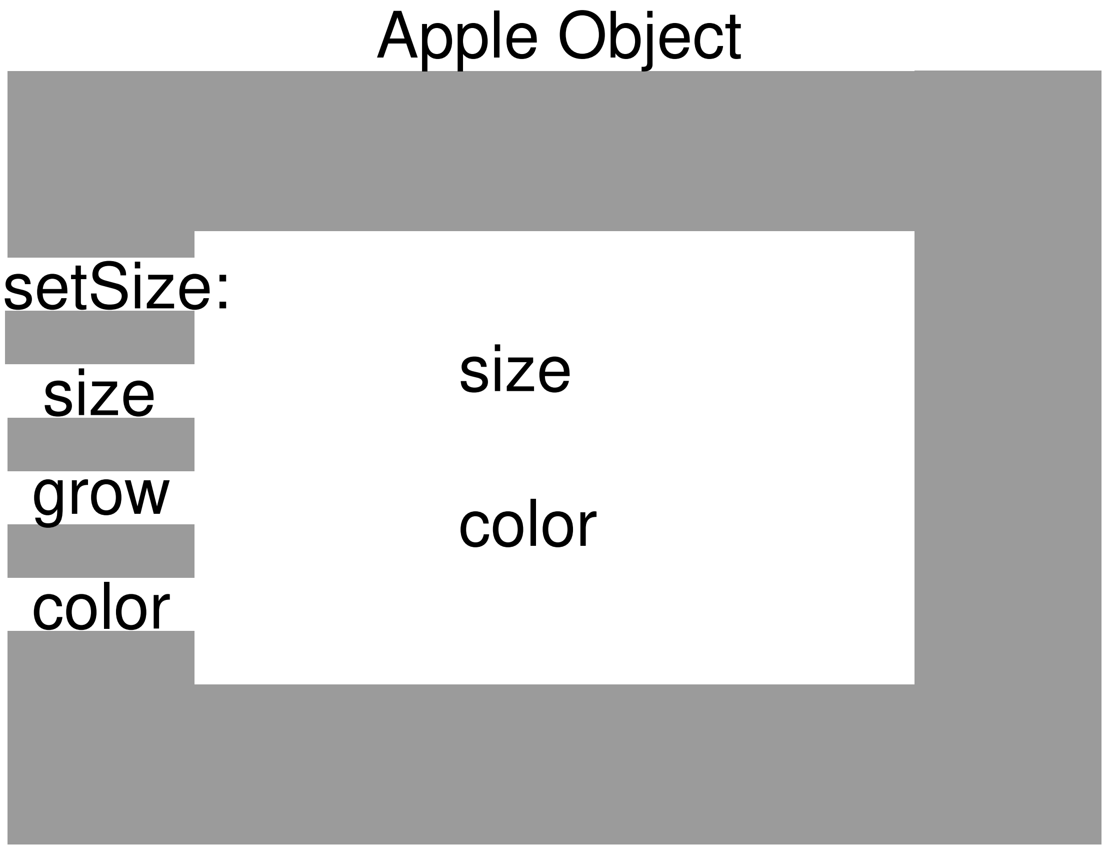
Internal states are called the ***instance variables.***
The functions used to access them are called ***methods***.

Example: Apple object

Instance variables: size, color

Methods:

setSize: - sets size to argument passed
size - returns the value of size
grow - increments size by one
color - returns the value of color



Note: there is no **setColor:** method here, so the color cannot be changed!

Classes and Instances

Actually, there are two types of "objects":

Class Object (or "factory object"):

Contains the definition of the class itself.

Has ***class methods*** that it can execute.

Knows how to build new objects *belonging* to the class.

Class names begin with uppercase letters. *(Example: Apple*)

Classes and Instances *(continued)*

Instance of the Class:

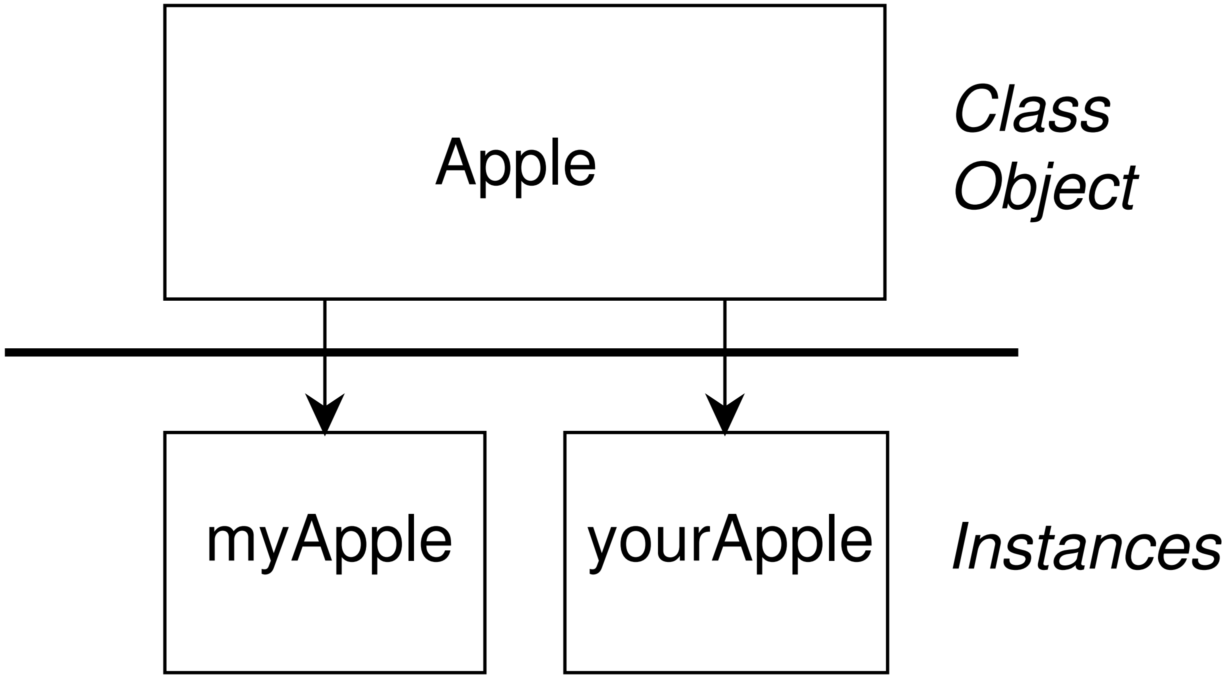
An individual occurrence of an object created by a class method in the class object.

Has ***instance variables*** and ***instance methods***.

These are the ***objects*** that do work in your program.

Instance names begin with lowercase letters.
(Example: myApple)

*Example: An **Apple** class object produces two instances of its class, **myApple** and **yourApple**.*



Benefits of Object Classes

- Insulation and protection of data
  ---> Enhanced reliability

- Transparent changes

- Anthropomorphic behavior: an object "knows how" to do things when "told" to do so.

2. Messaging

An object can ask another object to perform one of its methods or actions via a ***message***.

A message statement contains:

1) A ***reference*** to the object which is to be called (the ***receiver***).

2) The ***name*** of the method to be executed

3) Any ***arguments***, if required.

Example of Messaging

Declare an object to be in the "Apple" class, and send a "setSize:" message to it:

```
Apple *myApple;
.
.
[myApple setSize:5];
```

target object: `myApple`

class of target: `Apple`

message name: `setSize:`

argument: `5`

Messaging vs. Calling

A message is essentially a function call, but the *emphasis* is on the data itself rather than the operation. For example:

Message a stack to push a data value onto itself:

```
[myStack push:value];
```

instead of ...

Call a push routine, with a stack and data value as arguments:

```
push(myStack,value);
```

3. Polymorphism

The term ***polymorphism*** generally refers to the ability to take on more than one form.

Operator overloading is a form of polymorphism used in some conventional programming languages.

In object-oriented languages, a *polymorphic object* is a variable which can, during the course of execution, point to objects of more than one class.

Polymorphic object references allow the programmer to design at a high level of abstraction.

Examples of Polymorphism

Example: "draw"message

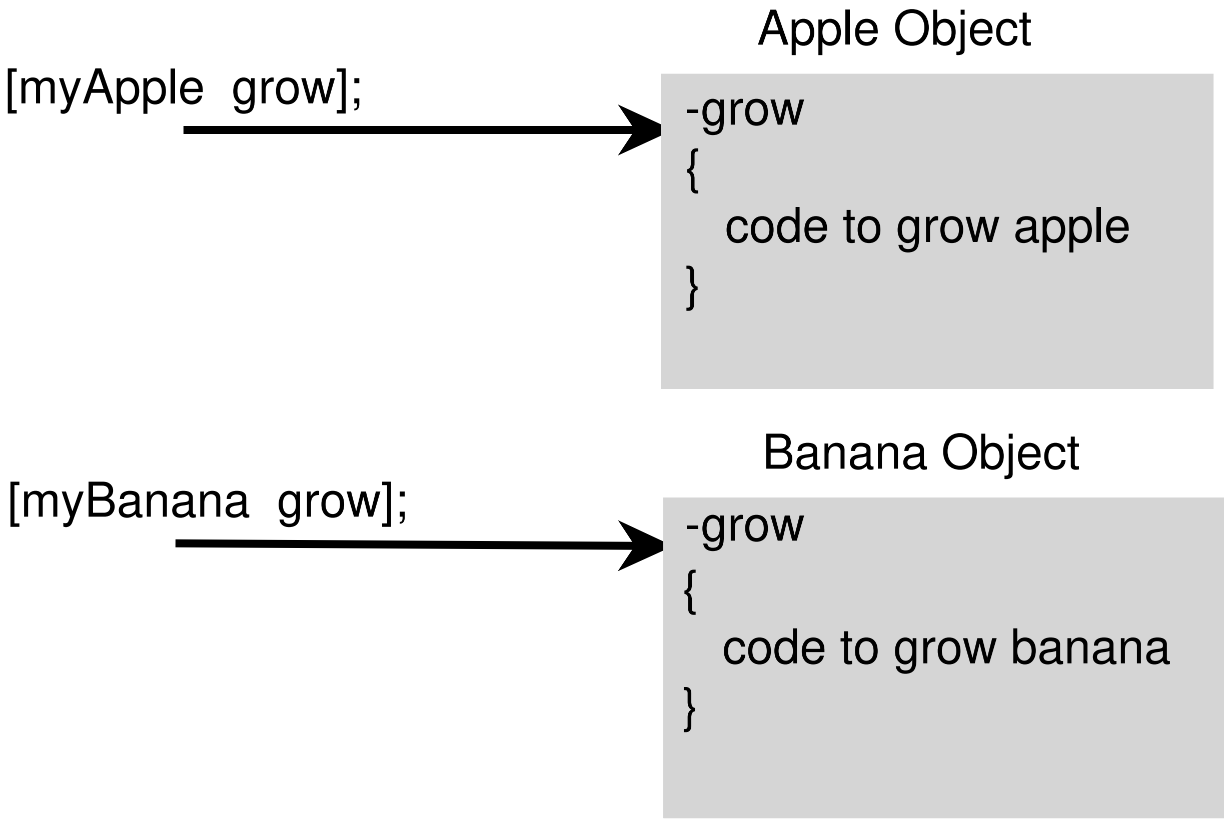
Main program tells all screen objects to "draw" themselves.

"Line" object does one thing.

"Circle" object does another.

Adding a new object is easy: you provide the "draw" method with the new object and you don't change the original program code!

Example: Send same "grow" message to objects in two different classes, Apple and Banana:



4. Inheritance

First, as objects in a class:

Objects are always defined as members of a ***class***.

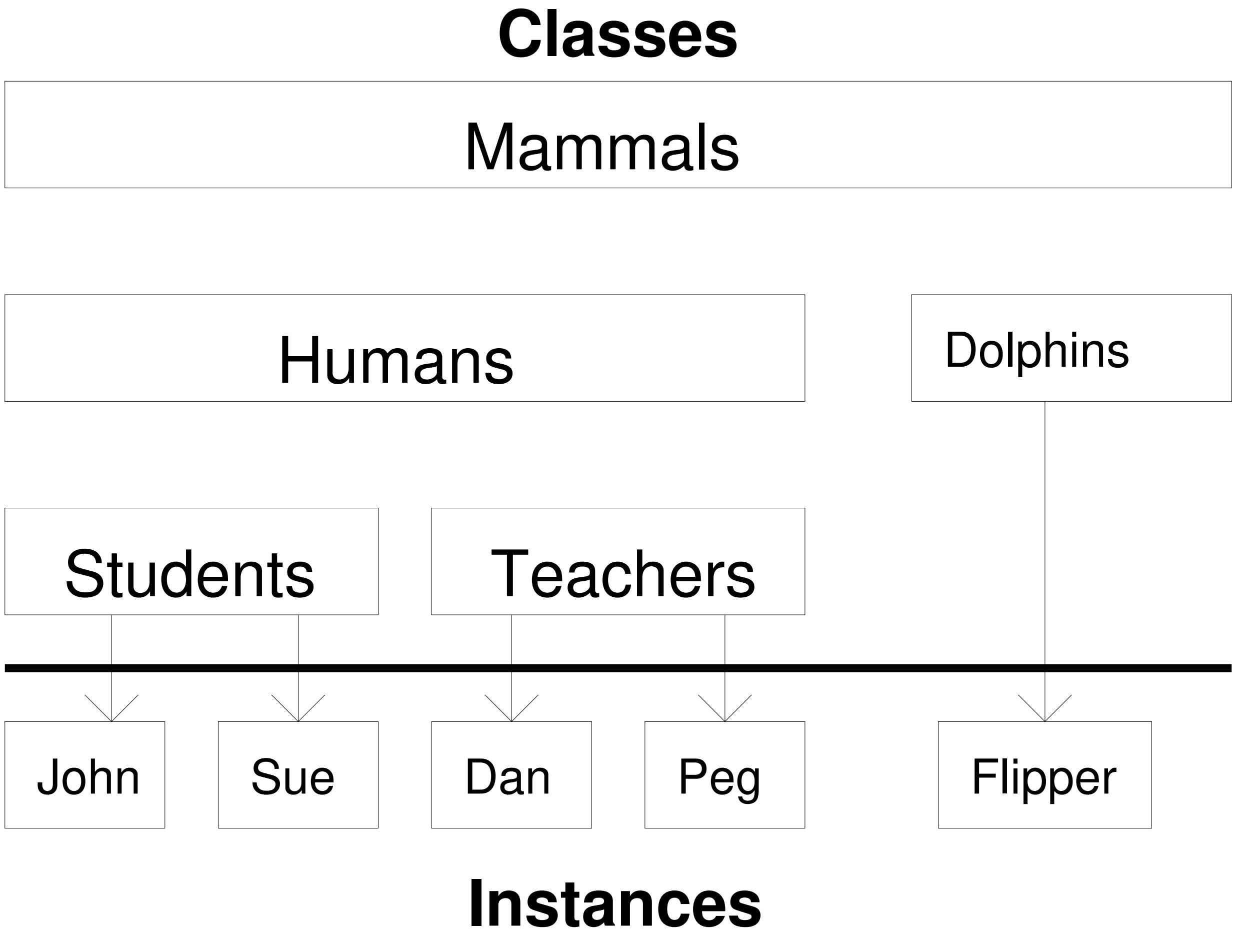
An instance automatically has the instance variables and methods defined for its class. This is basic encapsulation.

Inheritance goes further:

A class is ***also*** defined as a ***subclass*** of some other class, called its ***superclass***.

A subclass also ***inherits*** the instance variables and methods defined for all of its superclasses.

Sample Inheritance Hierarchy



Inheritance (continued)

When a class object creates a new *instance*, the new object has the ***instance methods*** and ***instance variables*** defined for its superclass, its superclass' superclass, ... to the root ***Object*** class (*Objective-C*).

However, a *class object* itself inherits only the ***class methods*** from its ancestors.

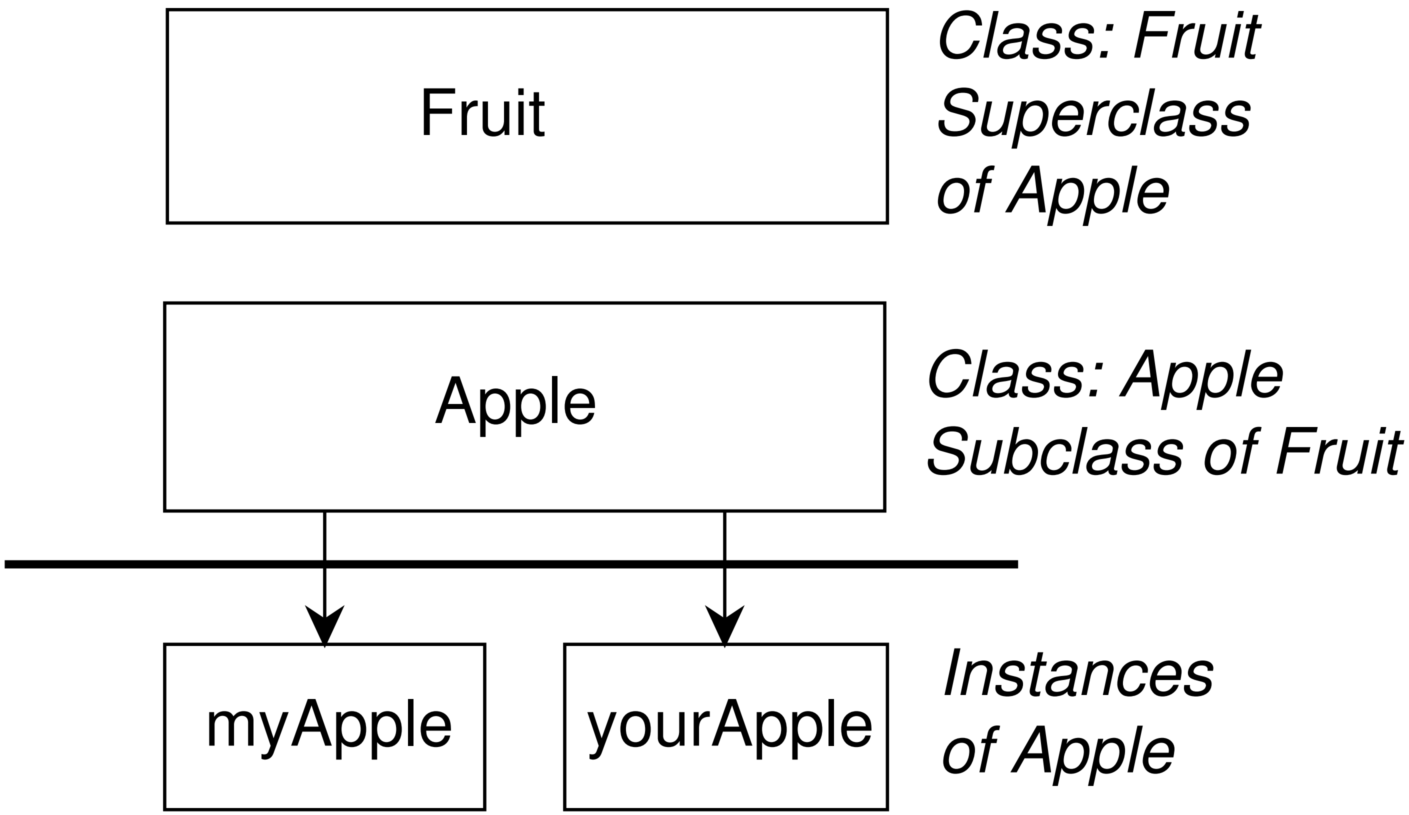
Inheritance (continued)

A new subclass may ***add*** additional instance variables and methods.

A new subclass may ***override*** (*redefine*) methods defined in a superclass. The overriding method may completely replace the superclass' method, or may simply add functionality (code) to it.

Inheritance

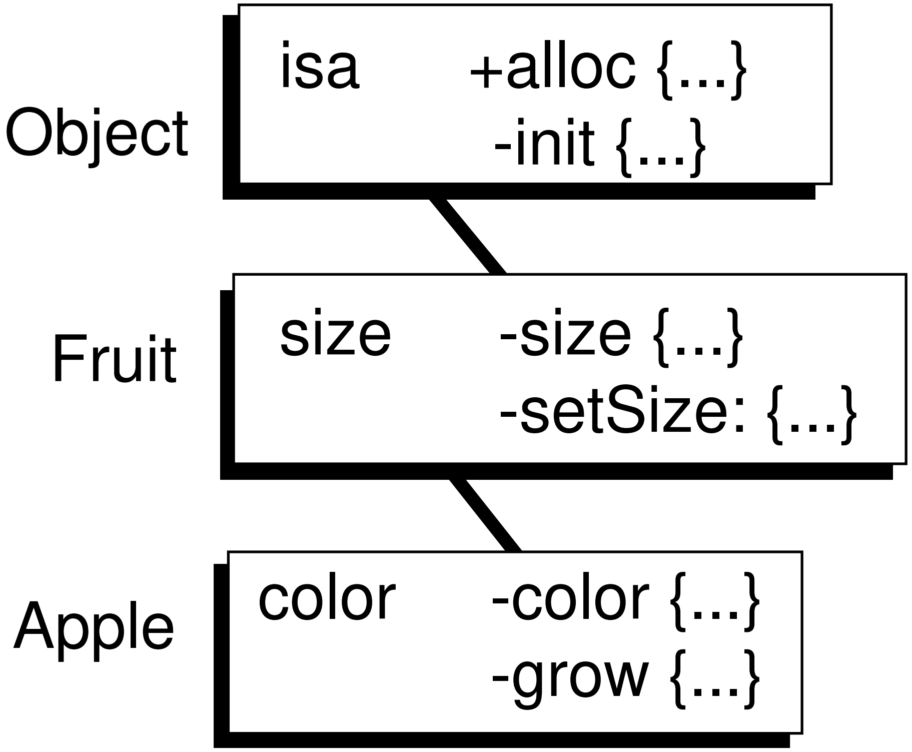
Example: ***Apple*** *is subclass of* ***Fruit****.*



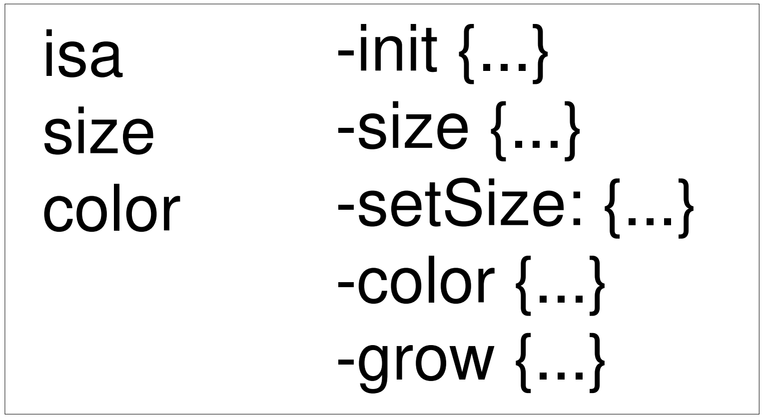
Apple *(in Objective-C)*

INHERITS FROM	Fruit : Object	
INSTANCE VARIABLES		
Inherited from Object	Class	isa;
Inherited from Fruit	int	size;
Declared in Apple	char*	color;
METHODS		
Inherited from Object	+alloc { ... } (class method)	
	-init { ... }	
Inherited from Fruit	-size { ... }	
	-setSize: { ... }	
Declared in Apple	-color { ... }	
	-grow {...}	

myApple Inheritance Hierarchy



--> If myApple is an ***instance*** of Apple:



myApple

Objects in Memory
(*Objective-C*)

For each class used in a program, the following is in memory:

Class Object

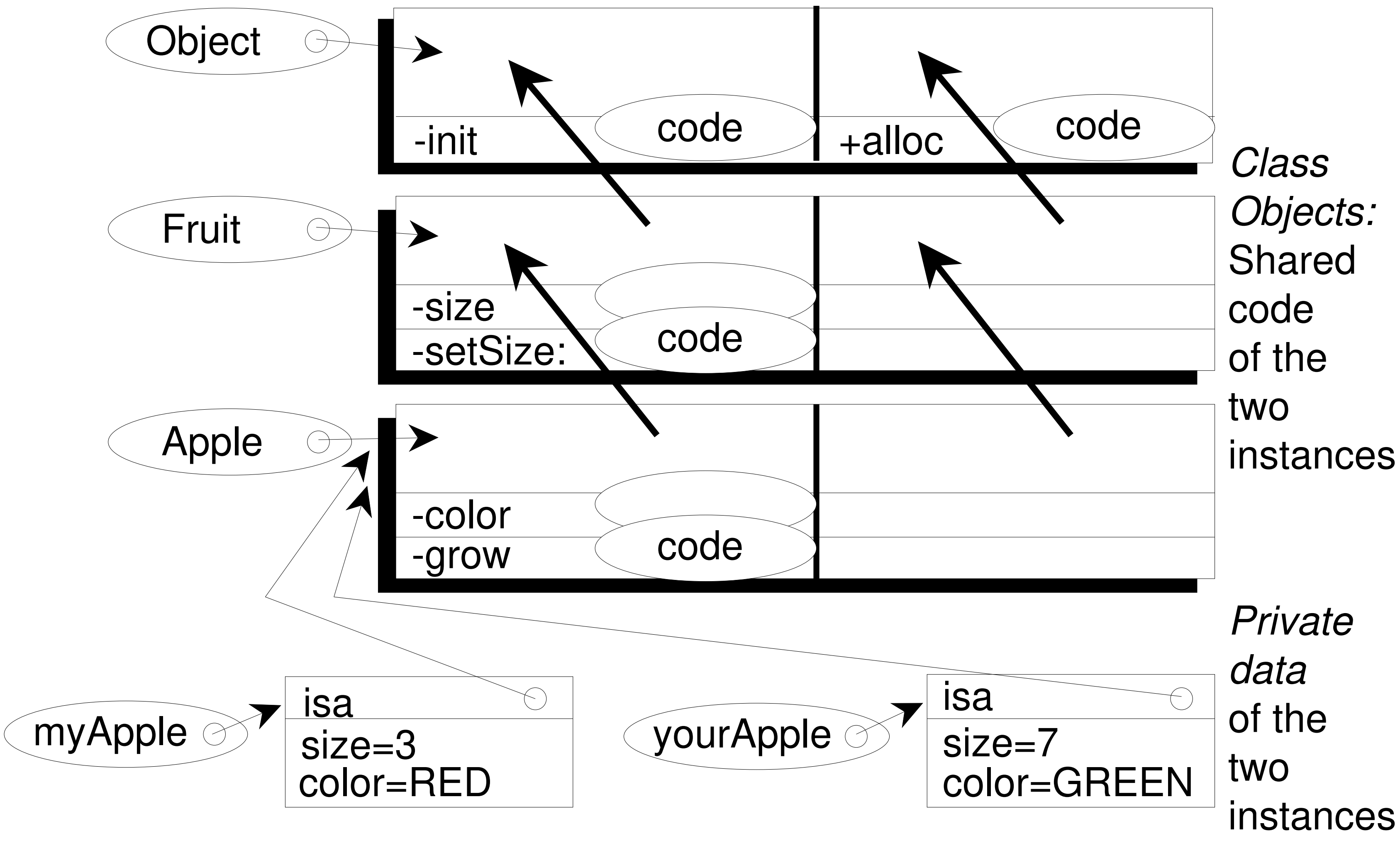
There is one copy of each class object. It contains the ***shareable*** code for the methods, and other information describing the structure of the class. This class object has links up the inheritance chain to superclass class objects.

Objects in Memory
(continued)

Instances

Each instance appears as simply a data structure containing that instance’s ***private*** copies of the ***instance variables***. The ***isa*** variable points to the instance’s Class Object where messages will be directed in search of a method, following the inheritance chain as necessary.

Example: Class objects and instances in memory.



Benefits of Inheritance

Inheritance provides an explicit expression of ***commonality***.

We will specify common attributes and services ***once***, and allow them to be inherited by newly-defined objects.

Almost never start from scratch when building a new object: It is easy to define a new object that is just like an old object except for a few minor differences.

Principal factor in enhancing the ***reusability*** of objects.

Inheritance is unique to OOP. Languages with objects but without inheritance are called *"object-based,"* e.g., ADA.

Control of Procedures

Classes, inheritance, messaging, and polymorphism provide one answer to two important questions:

- *Where* is knowledge about procedures stored?
- What process decides *which* procedures act?

Two approaches can be taken:

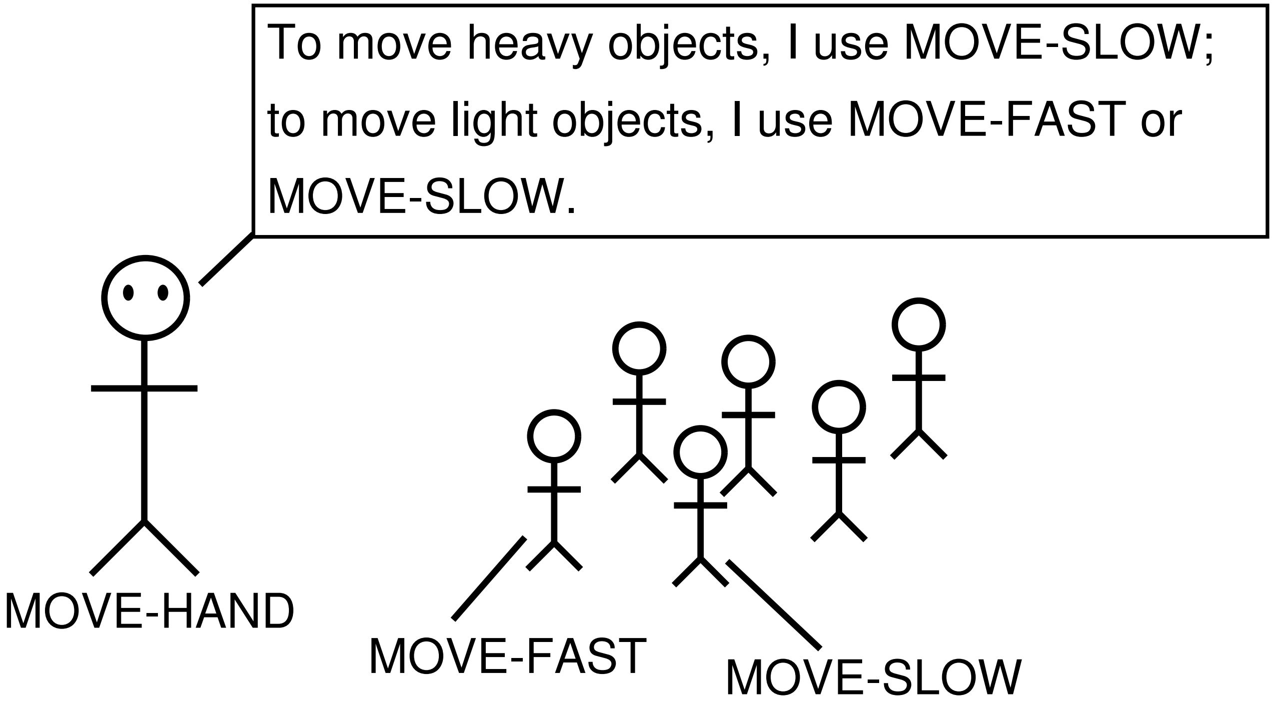
- **Action-centered** control

- **Object-centered** control

Action-Centered Control

A system exhibits ***action-centered*** control when the system's procedures know what subprocedures to use to perform actions.

Action-Centered Control of Procedures

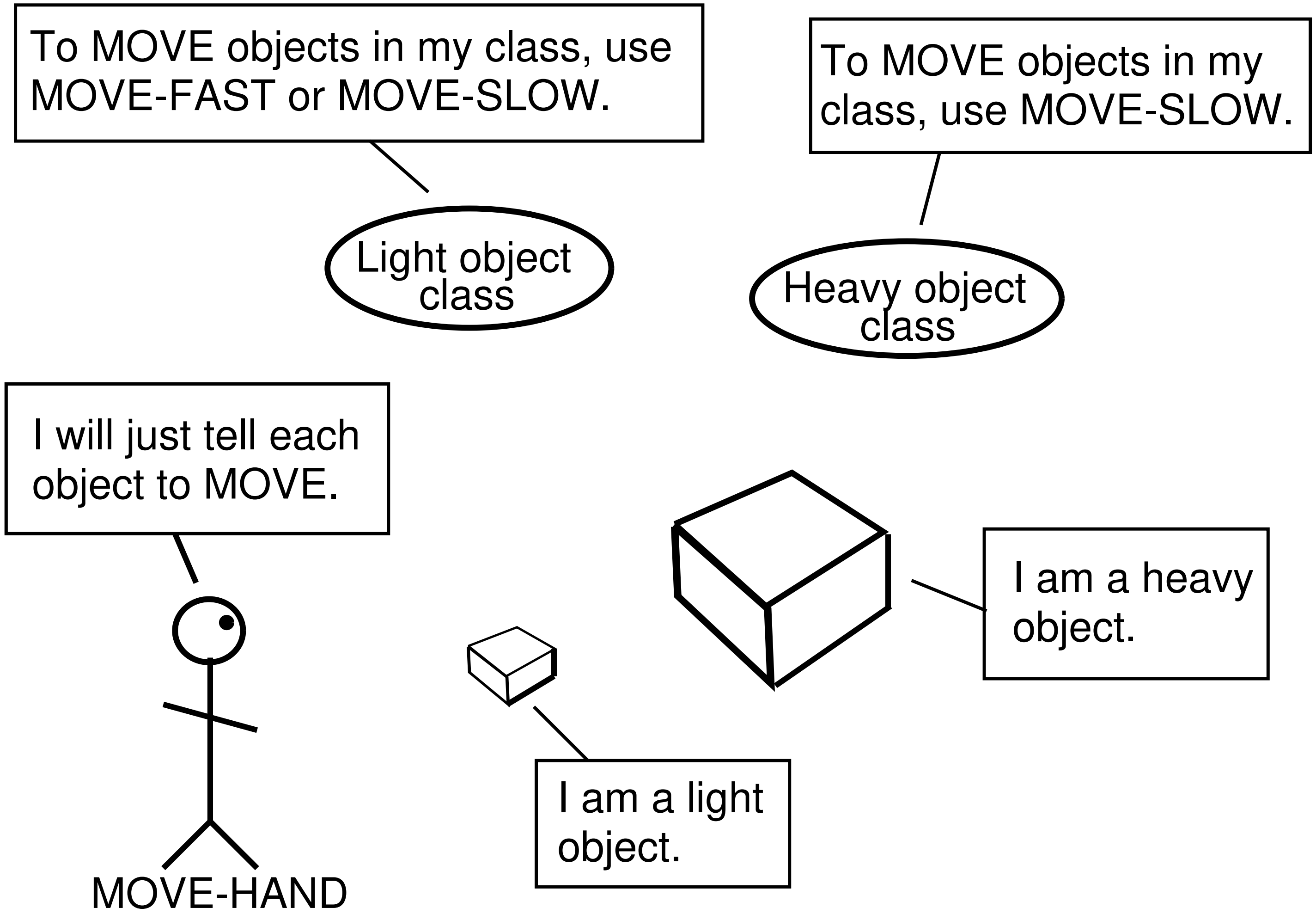


This approach represents simply calling the right routine for the job to be done. This is the old way.

Object-Centered Control

Here the system’s ***class descriptions*** specify how to deal with objects in their own class.

Object-Centered Control of Procedures



5. Dynamic Binding
(late binding, runtime binding)

Two approaches to *method binding* when sending a message to a variable (pointer to an object):

- **static binding:** Matching the message to the method is done by the compiler based on the static declaration of the type of the variable.

- **dynamic binding:** The runtime system discovers the type (class) of the variable based on its value at the time the message is sent.

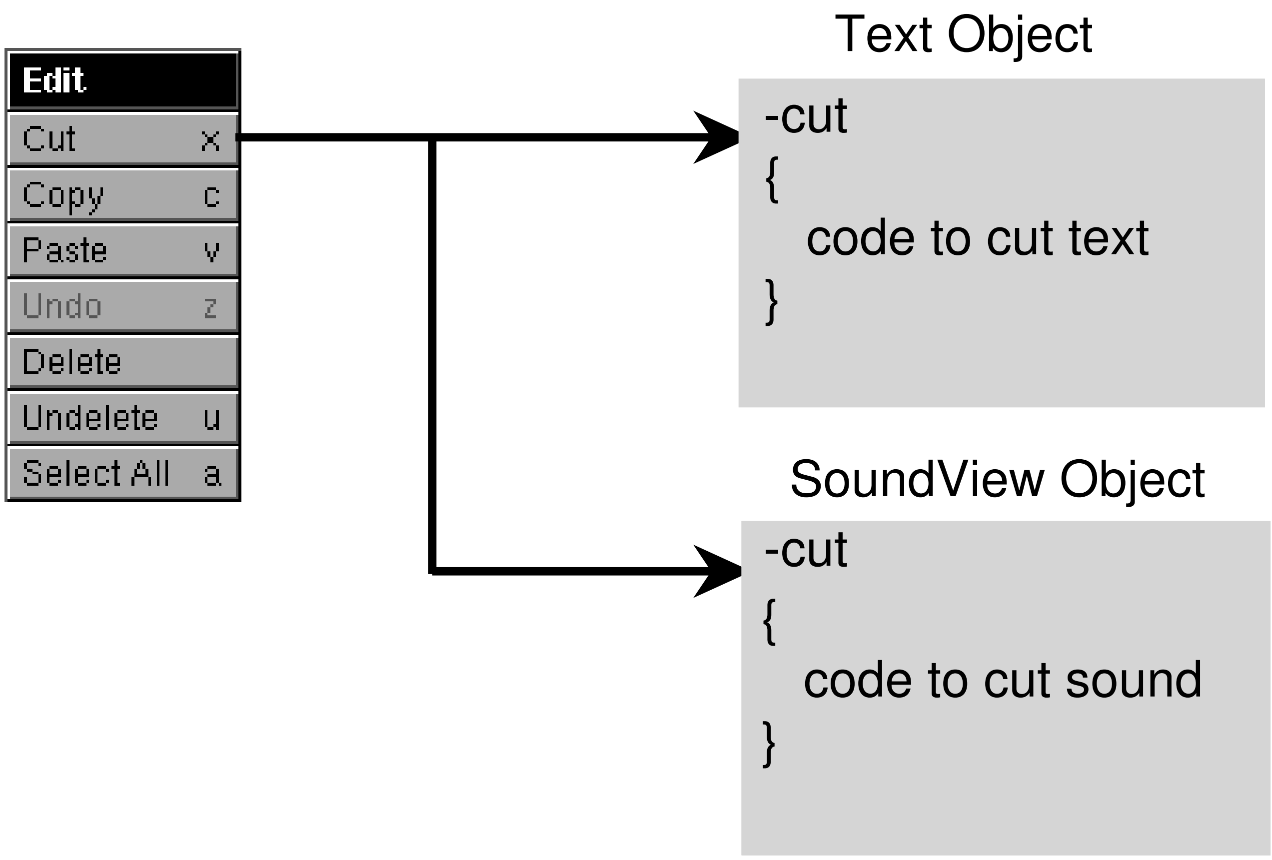
Example of Dynamic Binding

Ask a "basket" object for an item, and send a "grow" message to it, without knowing whether it is "apple" or "banana".

```
Basket *myBasket;
id fruitItem;           //class unknown
fruitItem = [myBasket  item];  //get item
[fruitItem  grow];   //tell it to grow
```

---> Cannot bind (link) to "grow" method of either Apple or Banana class until runtime. Only then is the class of the item finally learned based on the value of the variable **`fruitItem`** at that time.

Example: Edit cannot know the class of the object to be "cut" until runtime.



Benefits of Dynamic Binding

Polymorphism takes on greater meaning and usefulness.

Developers of programs can do rapid prototypes and not have to anticipate every future type of object.

Programs can be extensible, dynamically loading code modules at runtime.

The flexibility provided the programmer outweighs the loss of efficiency in the runtime system.

***Flexibility** is more important at higher levels of abstraction!*

Review of OOP Properties

Classes of Objects - Procedural and data abstraction, encapsulation, data security.

Messaging - Flexible communication between objects, with emphasis on the data rather than the operation.

Polymorphism - Allows programming at a higher level of abstraction.

Inheritance - Makes it easy to extend and reuse objects.

Dynamic Binding - Allows flexibility at runtime.

Summary of OOP

1. Object-oriented programming consists of sending messages to objects.

2. Problem-solving consists of (a) identifying the objects, (b) identifying the messages associated with the objects, (c) developing the sequence of messages to objects that solves the problem.

3. OOP languages generally support classes of objects, messaging, polymorphism, inheritance, and (preferably) dynamic binding.

4. The unit of encapsulation is the object, which includes private data and methods.

Summary of OOP *(continued)*

5. Abstraction is supported by a hierarchy of classes representing different kinds of objects. A class description protocol defines the properties of each abstraction. New abstractions are added by adding new subclasses.

6. Subclasses inherit the properties of their superclasses, including private data and methods.

7. Abstraction is further supported by polymorphic definition of the same message in different classes.

References

An Introduction to Object-Oriented Programming, by Timothy Budd, Addison-Wesley, 1991. An excellent book on OOP, with comparisons between Objective-C, C++, Object Pascal, and Smalltalk.

Object-Oriented Programming: An Evolutionary Approach, by Brad Cox, Addison-Wesley, 1987. This book has Objective-C examples, but is best for background information, not as a reference manual.

Objective-C : Object-Oriented Programming Techniques, by Lewis J. Pinson & Richard S. Wiener, Addison-Wesley, 1991. A good book for OOP with Objective-C examples.

An Introduction to Object-Oriented Programming and Smalltalk, by Lewis J. Pinson & Richard S. Wiener, Addison-Wesley, 1988. Although the book is about Smalltalk, Chapter 1 has some good general OOP information that is worth reading.