Objective-C

Objective-C Features

All OOP properties are supported, with these qualifications:

Every class is a subclass of some other class, inheriting ultimately from the root **Object** class.

Objective-C does not support multiple inheritance.

Class methods, instance methods, and instance variables are inherited by subclasses. Static variables can be used as "class variables" but are not inherited by subclasses.

All method binding is dynamic.

Syntax of Objective-C

The syntax of Objective-C is the same as standard C, with the following additions:

+	indicates a class method.
–	indicates an instance method.
[ ]	indicates sending a message.
:	A message argument follows a colon. The colon is part of the message name.

Syntax of Objective-C
(continued)

`id` — a new variable *type* meaning: ***a pointer to an object*** *(class unspecified, dynamic typing)***.**

`self` — a pointer to the object receiving a message, set by the runtime system on entry to the method.

`super` — a reference to the superclass of the object sending a message to `super`.

Syntax of Objective-C
(continued)

`#import`	*#Import* is just like C *#include,* but will not duplicate previously included files.
`< >`	indicates a system file.
`" "`	indicates a local file.
@keyword	defines start and end of code sections.

Class Definitions

The ***class definition*** is the prototype for a kind of object. It declares the instance variables and defines the set of methods that all objects in the class can use.

Classes are defined in two ***separate*** *parts (information hiding):*

Interface file. *The public declarations of the class.*

Implementation file. *The private definitions of the class.*

Interface File

- ***Declares*** the interface (the instance variables and methods) to the new class.

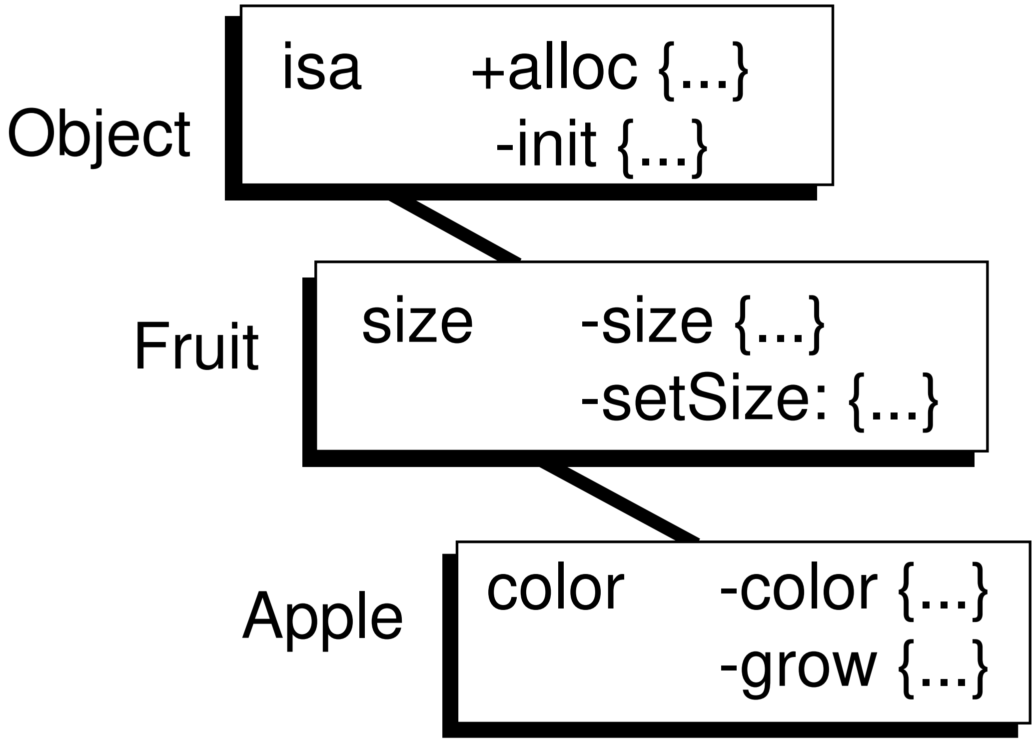
- Has a "**.h**" extension by convention.

- Specifies the name of the ***superclass*** and ***imports*** the interface file of the superclass.

- The interface is the ***public declaration*** of the class and is all that a ***user*** will see.

Example: The Apple class.

Recall the ***Apple*** *class inheritance hierarchy:*



Interface (Header) File (Apple.h)

```
#import "Fruit.h"

@interface Apple:Fruit

/* Instance Variables */
{
    char*  color;
}

 /* Methods */
- (char *)color;
- grow;

@end
```

Implementation File

- ***Defines*** the class and contains the ***code*** that implements it.

- Has a "**.m**" extension by convention.

- Must ***import*** the interface file of its own class.

- Must also ***import*** the interface files of any objects it will send messages to.

- The implementation is ***private*** to the ***developer*** of the class.

Implementation File (Apple.m)

```
#import "Apple.h"
@implementation Apple
/* Return the present color */
-(char *)color{
    return color;
}
/* Grow the apple */
-grow{
    size = size + 1;
    return self;
}
@end
```

Note: Since **`size`** is an *inherited* instance variable, it might be wiser to write:

```
[self setSize:([self size]+1)];
```

Objects in Memory

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

Class Object

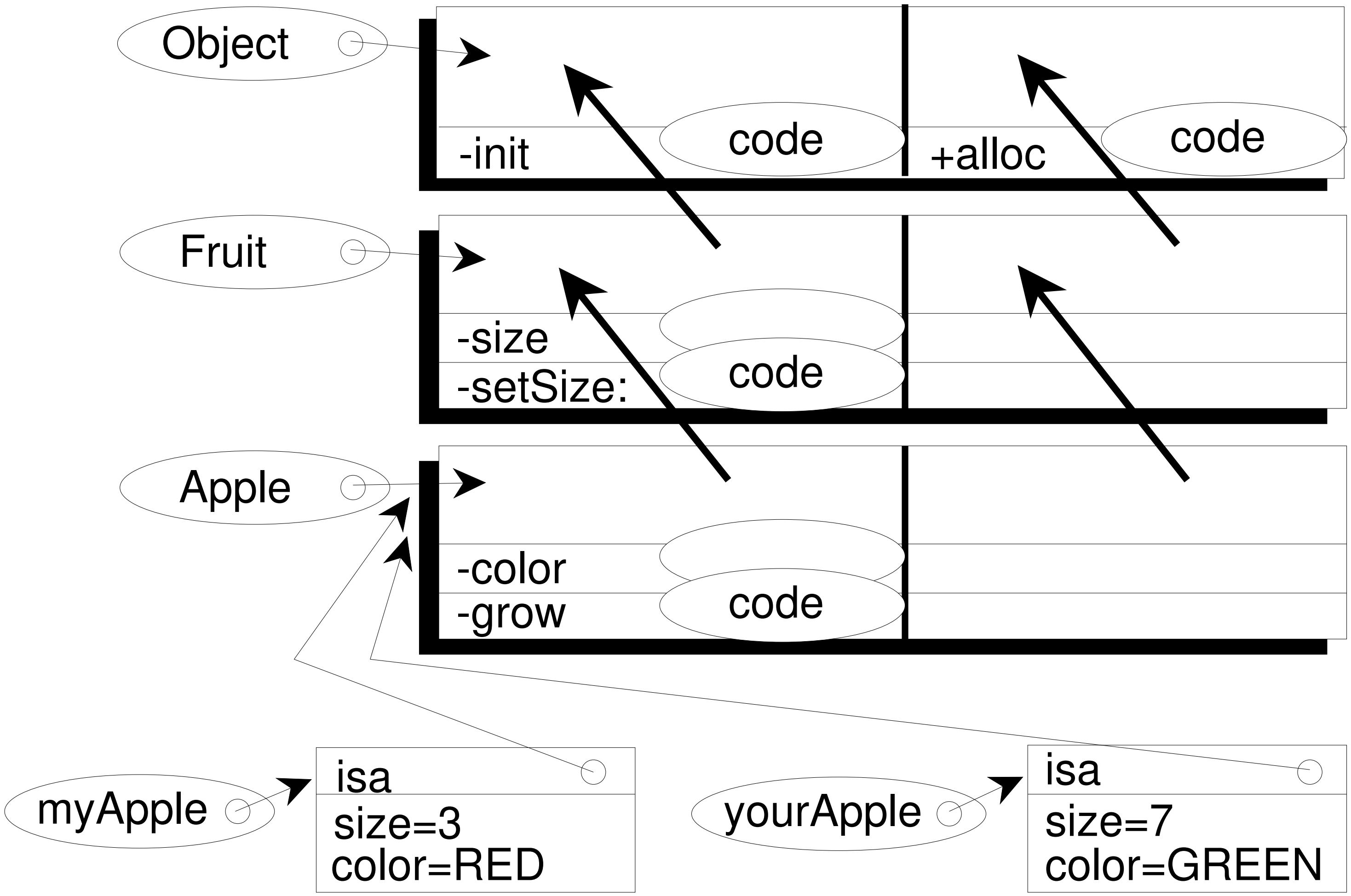
There is one copy of each class object. It contains the ***shareable*** code for the methods, and other information describing the structure of the class. This class object has links up the inheritance chain to superclass class objects.

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.



Class Objects: Shared code of the two instances

Private data of the two instances

Categories

A ***category*** specifies *additions* to an existing class, which are then available to *all* instances of the class or its subclasses.

Only methods can be added, not instance variables.

Methods inherited from the superclass can be overriden, but not methods defined elsewhere in the class itself.

Categories are defined similar to subclasses: They require an *interface file* and an *implementation file*.

Example: Adding to the Fruit class

Interface File (RealFruit.h)

```
#import "Fruit.h"
@interface Fruit(RealFruit)
- rot;          //new method
@end
```

Implementation File (RealFruit.m)

```
#import "RealFruit.h"
@implementation Fruit(RealFruit)
- rot{
    <code to turn color darker>
    return self;
}
@end
```

Categories *(cont'd)*

In the example above, ***all*** instances of the Fruit class, or of subclasses of Fruit, will now have the **`rot`** method.

This is useful when the programmer does not have access to the original class to modify it (because it was defined by someone else).

Category vs. Subclass: Subclassing Fruit to get a RealFruit class would allow creating instances of RealFruit with all of the desired behavior. However, instances of Fruit or its subclasses would not have the new behavior. With a category they would.

Naming Conventions

- Case

Use uppercase letters instead of the underscore "_" character.

Ex: `nameValue`

- Class Names

Begin with upper case.

Ex: `Apple`

Naming Conventions

(continued)

- Variables

Begin with lower case.

Ex: `myApple`

- Methods

Begin with lower case.

Ex: `grow`

Methods and Messages

- ***Method*** refers to the *definition* in the implementation file.

- ***Message*** refers to the *invocation* of a method at runtime.

- Method: like a C-function definition.

- Message: like a C-function call.

- Message Sending --> Method Invocation

Message Expression

`[receiver message];`

Sends **`message`** *to* **`receiver`**. *The name of the message is called the* ***selector***.

`myColor=[myApple color];`

Sends **`color`** *message to* **`myApple`**. *A value is returned and assigned to* **`myColor`**.

`[banana color:"yellow"];`

Sends **`color:`** *message (pronounced "color colon") to* **`banana`**. *Argument passed is* **`"yellow"`**. *(Not the same message as* **`color`**.*)*

Message Expression *(continued)*

`[box setSide:1 toColor:"red"];`

Sends **`setSide:toColor:`** *message to* **`box`**. *Two arguments are passed,* **`1`** *and* **`"red"`**.

`[myView moveTo:x :y];`

Sends **`moveTo::`** *message to* **`myView`**. *Two arguments are passed,* **`x`** *and* **`y`**.

Class and Instance Methods

Messages to invoke ***class methods*** must be sent to the class object. The most common class method is **`alloc`** to allocate memory for (to create) a new instance of a class.

Messages to invoke ***instance methods*** are sent only to instances.

Creating a New Instance

When creating a new instance, first message the class object to ***allocate*** the memory, then ***initialize*** the new instance:

```
myApple = [Apple alloc];
[myApple init];
```

Since **`alloc`** returns the **`id`** of the new instance, and init returns **`self`**, this is best done (and *should* be done) in combination:

```
myApple = [[Apple alloc] init];
```

*Note: For **View** objects, use* **`initFrame:`** *instead of* **`init`**.

Returning Self

In the previous example, **`init`** returned **`self`** as the return value of the method. ***All*** methods should return **`self`** by default ***unless*** they must explicitly return some other value. This allows nested calls. For example,

```
[myBanana setSize:2];
[myBanana color:"yellow"];
```

If **`setSize:`** returns **`self`**, these can be nested as follows:

```
[[myBanana setSize:2] color:"yellow"]
```

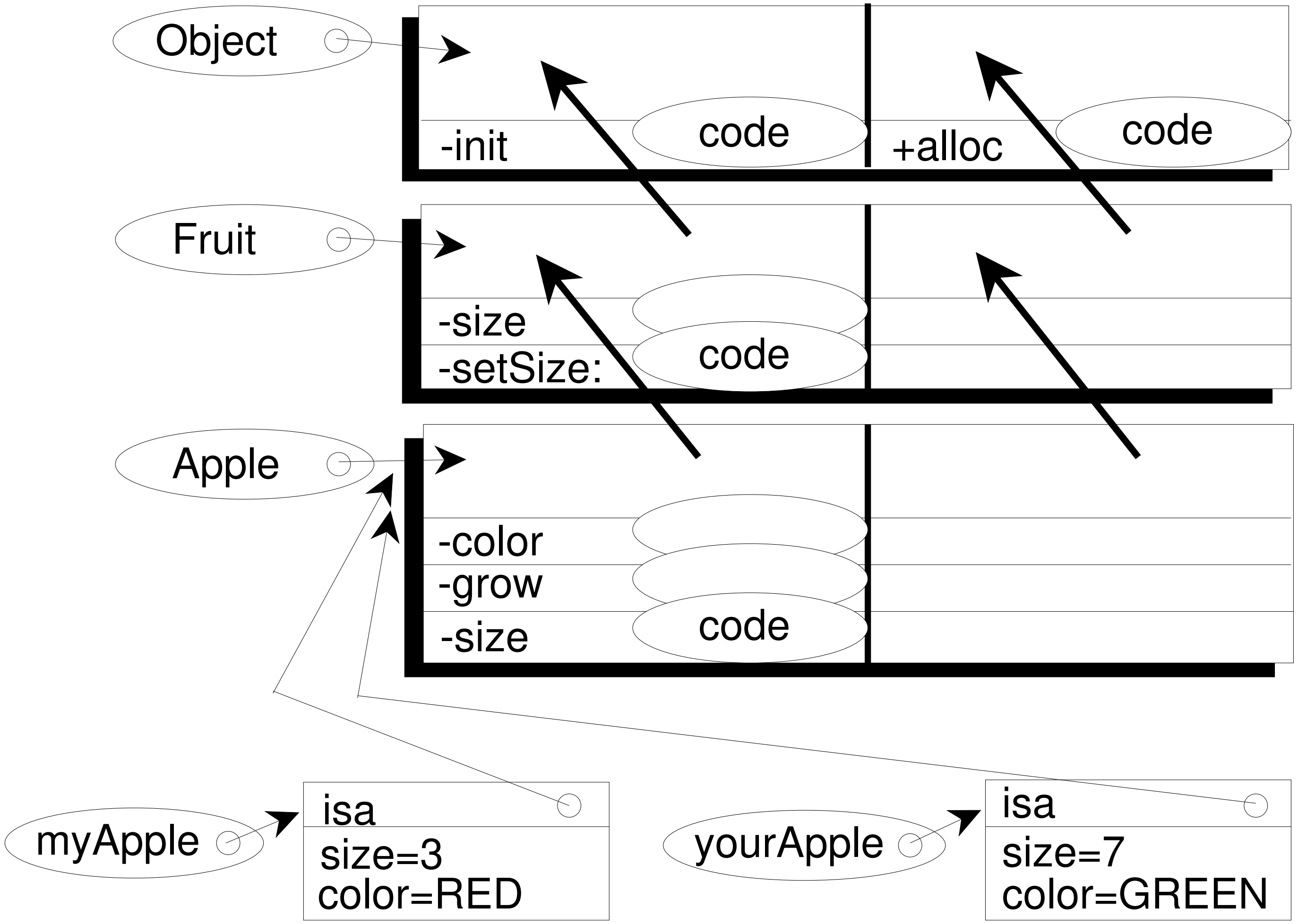
Method Overriding

In general, a subclass inherits all of the methods of its superclass. For example, a **`size`** *method was defined in the* ***Fruit*** *class and was inherited by* ***Apple****. A message to* **`myApple`** *invokes* ***Fruit****'s implementation:*

```
[myApple size]  //inherited size method
```

However, if ***Apple*** *were to define its* ***own*** *version of* **`size`** *in its implementation file, the message to* **`myApple`** *would go to* ***Apple'****s implementation instead, thus* ***overriding Fruit'****s implementation.*

Example: Overriding the `size` method in Apple.



Class Objects: Shared code of the two instances

Private data of the two instances

Method Overriding using `super`

Frequently the goal in overriding the superclass' method is to ***enhance,*** *not replace,* its behavior. Below, all the functionality of the superclass' method is kept, while ***adding*** functionality (**`<extra code>`**) in the subclass. Use **`super`** to get to the superclass implementation.

```
–methodName  /* method defined in subclass */
{
   [super methodName]  //super's method first
   <extra code>
   return self;
 }
```

Method Overriding using `super`

(continued)

The root class ***Object*** defines a default `init` method. Classes wishing to add their own initialization code when instances are created must first invoke all inherited initialization code.

For example, if all new instances of Apple must be initialized with a size of 1:

```
-init{/* Init method for Apple class*/
   [super init]   //first, inherited init
   [self setSize:1]   //<extra code>
   return self;
}
```

Method Overriding using `super`

(continued)

*Another example: Suppose **Macintosh** is a subclass of **Apple**, and every time it grows, its color gets darker. The **grow** method for **Macintosh** might look like:*

```
-grow  /* method defined in Macintosh */
{
   [super grow]  //Apple's method first
   <extra code to set to a darker color>
   return self;
  }
```

References

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.

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.

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.

Documentation manuals from NeXT Computer, Inc.