Chapter 1

Introduction To Objects

CHAPTER 1 INTRODUCTION TO OBJECTS

Programming languages have traditionally divided the world into two parts—data and operations on data. Data is static and immutable, except as the operations may change it. The procedures and functions that operate on data have no lasting state of their own; they're useful only in their ability to affect data.

Object-oriented programming doesn't so much dispute this view of the world as restructure it at a higher level. It groups operations and data into modular units called objects and lets you combine objects into structured networks to form a complete program. In an object-oriented programming language, objects and object interactions are the basic elements of design. — Object-Oriented Programming and the Objective-C Language, page 1.

Goal

To understand the role of objects and classes in object-oriented programming.

Prerequisites

Experience using the C programming language, including a basic understanding of pointers.

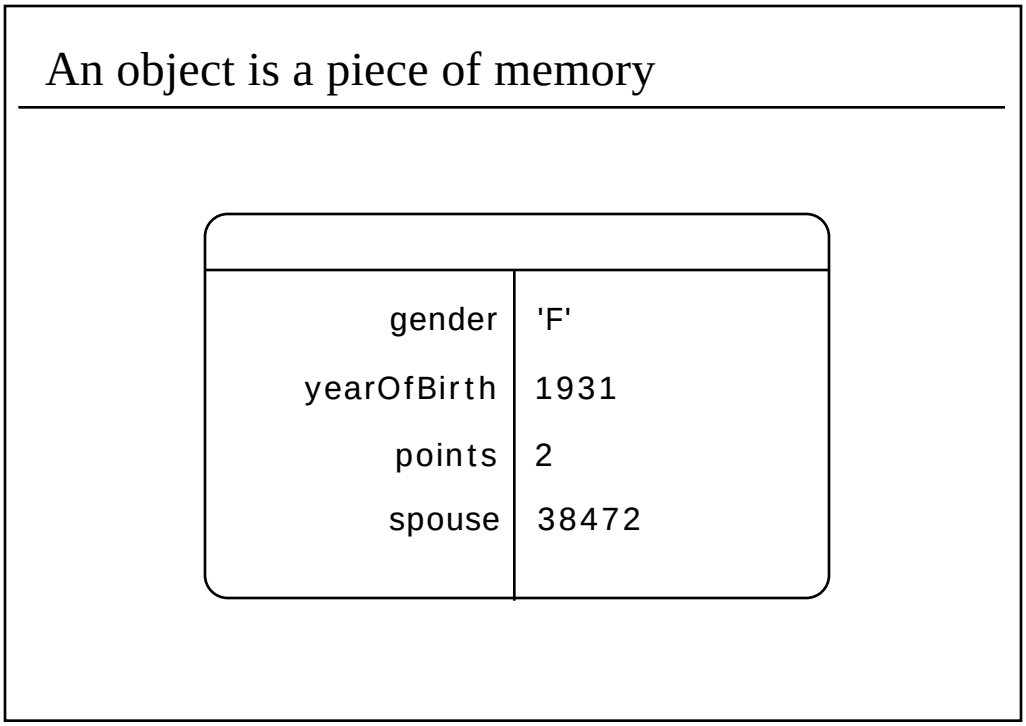
Objectives

After completing this chapter, you'll be able to:

- Explain objects, instance variables, and classes
- Define polymorphism, encapsulation, and identity

Reading

You can find more information about objects and object-oriented programming in **/System/Documentation/Developer/TaskAndConcepts/ ObjectiveC**

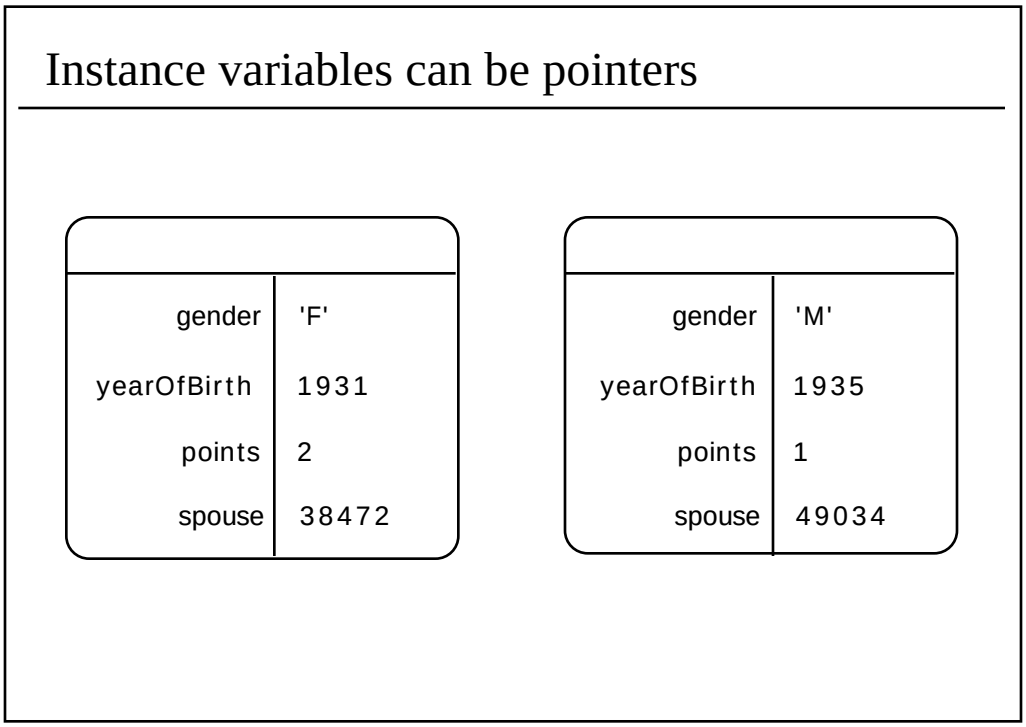


An object is a piece of memory

Computers treat nearly everything as a piece of memory. Programs, numbers, characters, images, and documents are all pieces of memory to a computer. Objects are no exception.

Typically, an object is a piece of memory that represents something in the real world. For example, imagine a program that holds data on people for a car insurance company. While the program is running, there are many pieces of memory, each containing data for one person. In a traditional C program, these pieces of memory would be structs. In an object-oriented program, these pieces of memory are objects.

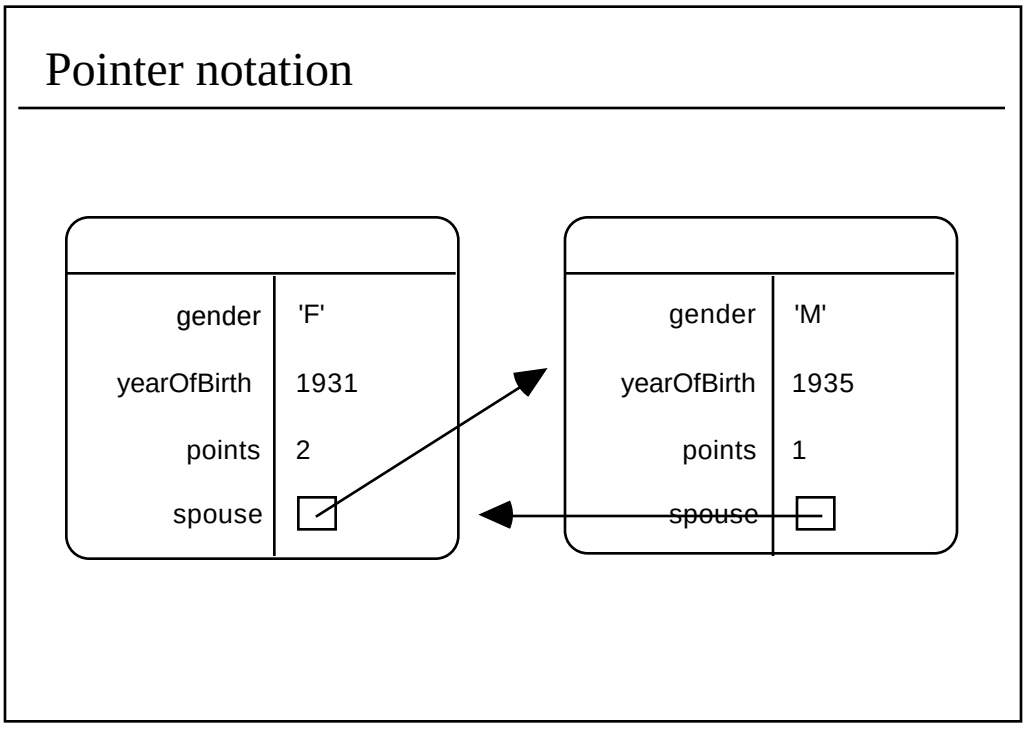
Inside each object, there are several smaller pieces of memory: a character that represents the gender of the person, an integer representing the year of birth, another integer representing the number of points on the person's drivers license. These smaller pieces inside an object are called **instance variables**. Objects use instance variables to store information about themselves.



Instance variables can be pointers

Recall that a pointer is a variable that holds a memory address. In C it's possible to follow a pointer in order to access the information that resides at that memory address.

Like other pieces of memory, every object lives at some address. Because instance variables can be any type, an instance variable can contain the address of another object. Therefore, an instance variable can point to another object. For example, an object representing a person could have an instance variable called spouse. This instance variable would contain the address of the object representing the person's **spouse**. In this way, one object can "know" another.

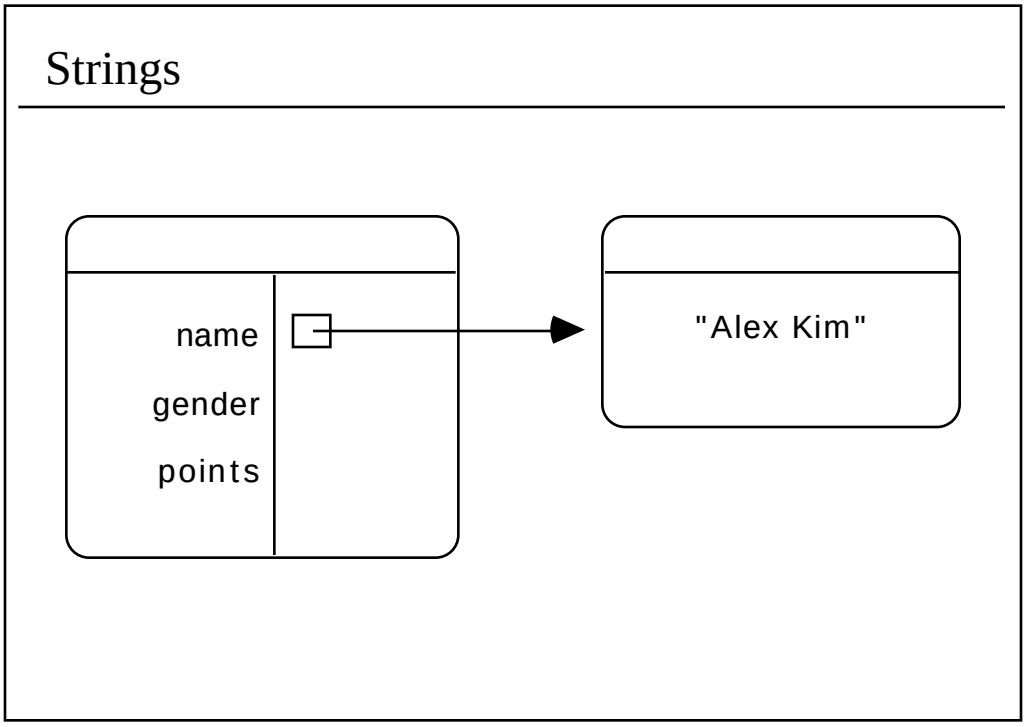


Pointer notation

Objects often have instance variables that are pointers to other objects. To make the diagrams clearer, this is represented by an arrow that points from the object with the pointer to the referenced object. It is easy to imagine that objects can be arranged in a complex network of pointers.

Recognizing object connections is a very important part of object-oriented development. Object-oriented analysis can be thought of as having two parts:

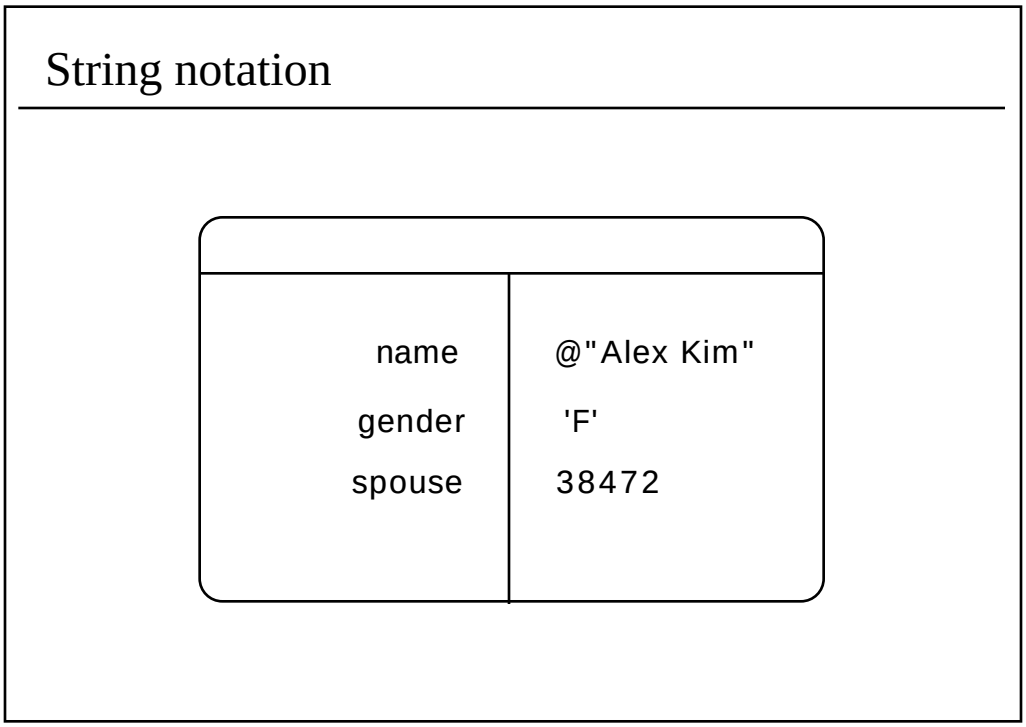
- Finding the objects that represent your enterprise
- Understanding their connections to other objects



Strings

Character strings are represented as objects. When your object has an instance variable that refers to a string, the instance variable is actually a pointer to a string object. String objects are used instead of char *'s in order to make string manipulation easier. This has many benefits, including transparent support for international character sets.

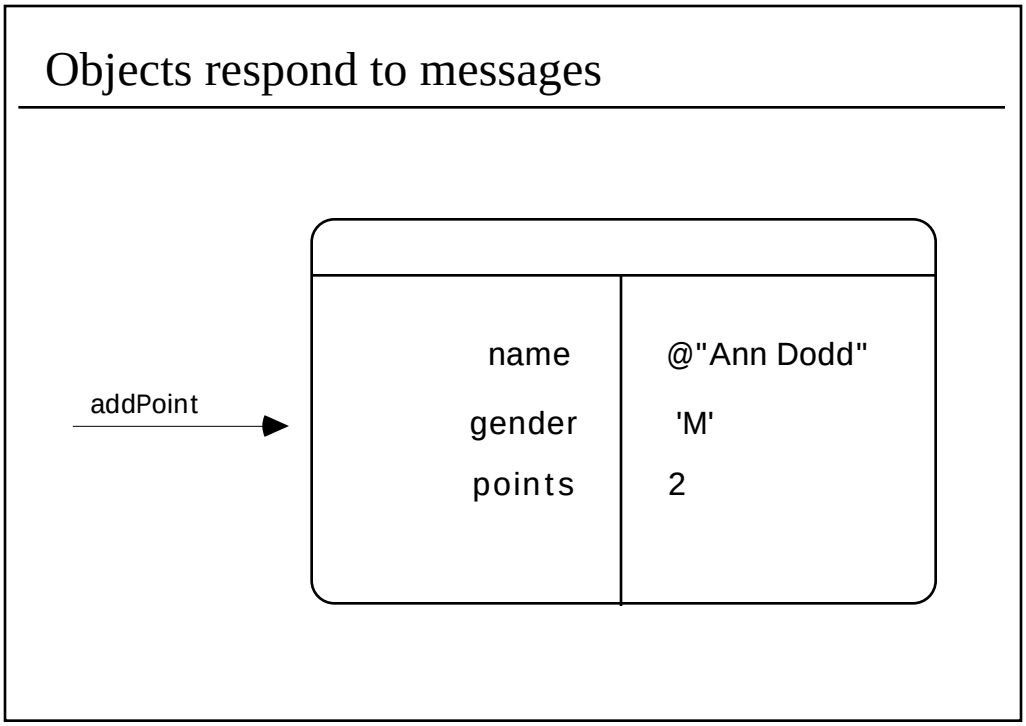
Many other types of commonly encountered data can be represented as objects. For example, dates can be represented as objects.



String notation

Strings are very common. Diagrams would be very complex if they used a separate box for every string. Therefore, the diagrams simply show the string inside the object. It's important to remember that the string is actually a separate object.

The symbol "@" is used in many contexts. It often means, "This is an object." The @ symbol that precedes the character string in the diagram is a reminder that the string is really an object.



Objects respond to messages

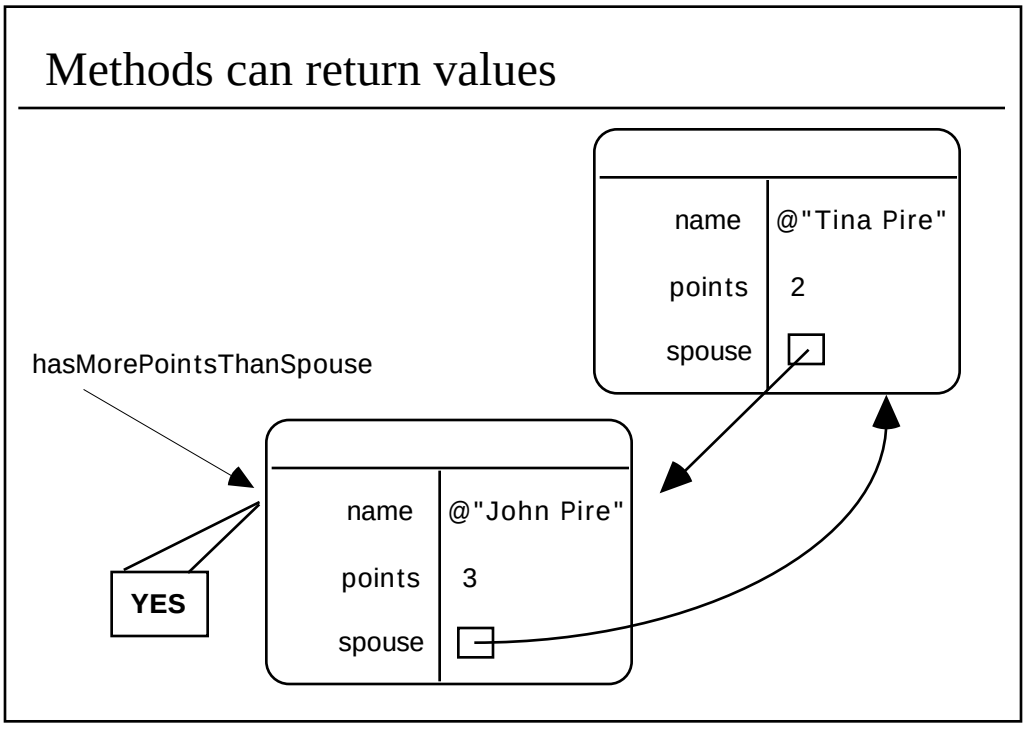
Objects are different from most other pieces of memory.They have behavior. In addition to storing data, they understand how to perform operations on that data. You send **messages** to objects to invoke their behavior. Messages are a lot like function calls—they're a way to tell an object what to do.

When you send a message to an object, the object invokes a corresponding **method**. Methods are like functions, the logic that provides the desired behavior. Most methods read or change the values of instance variables, because the values of its instance variables are a large part of what makes an object unique.

For example, a person object might understand the message **addPoint**. The response to that message would be to increase the number of points on their driver's license by one. This response is the method.

Programmers who are new to objects often confuse methods and messages. Messages are stimuli that cause a method to be executed. A method is a set of instructions to be executed. Messages are transient. A method is an unchanging property of the object.

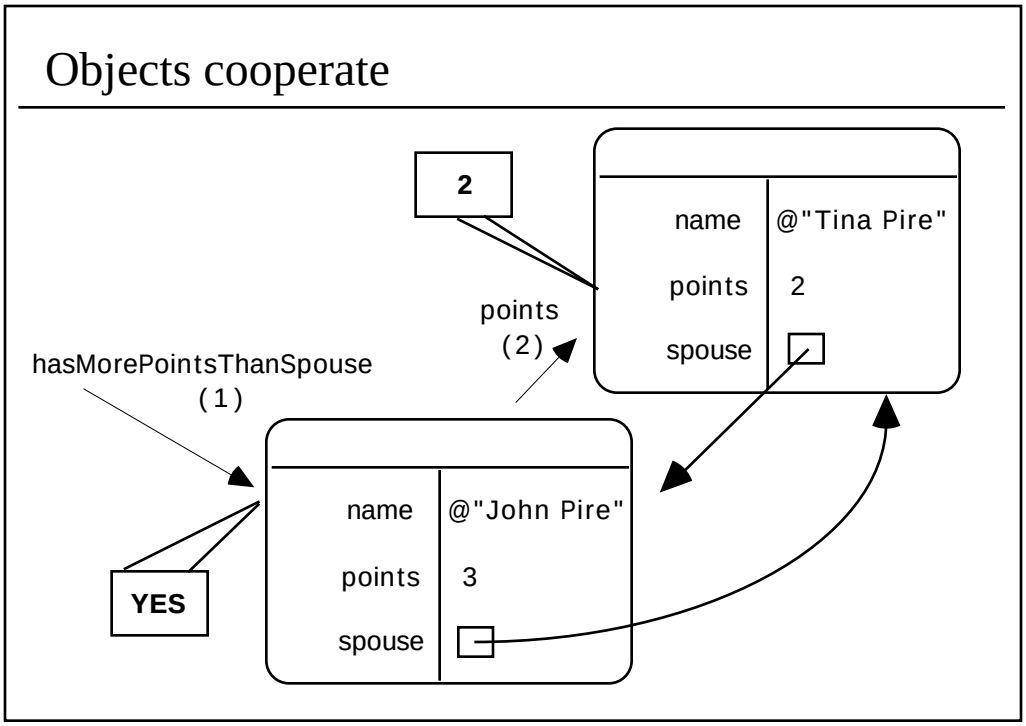
Each object has a set of messages it understands. Two different types of objects might not understand the same set of messages. Car objects, for example, would not understand the message **addPoint**.



Methods can return values

Some messages are orders to the objects. When you send **addPoint** to a person object, you're telling that object to do something. Sometimes you want to ask the object something about itself, not make it do something. For example, you might want to ask a person object if it had more points than its spouse object. You'd send a message like **hasMorePointsThanSpouse**. In order for this message to do any good, the corresponding method must be able to return a yes or no. To support this, methods can have return values.

Different methods provide different types of return values. Some return a yes or no, some return a number, some return a pointer to another object. Return values can be of any C type, including a pointer to an object.



Objects cooperate

In order to implement the **hasMorePointsThanSpouse** method, the person object needs to know how many points its spouse has. To find this out, the person object simply sends a points message to its spouse asking how many points the spouse has. The person can then compare this value with the number of points it has. In this way, the person and spouse object cooperate to provide the correct answer to a **hasMorePointsThanSpouse** message.

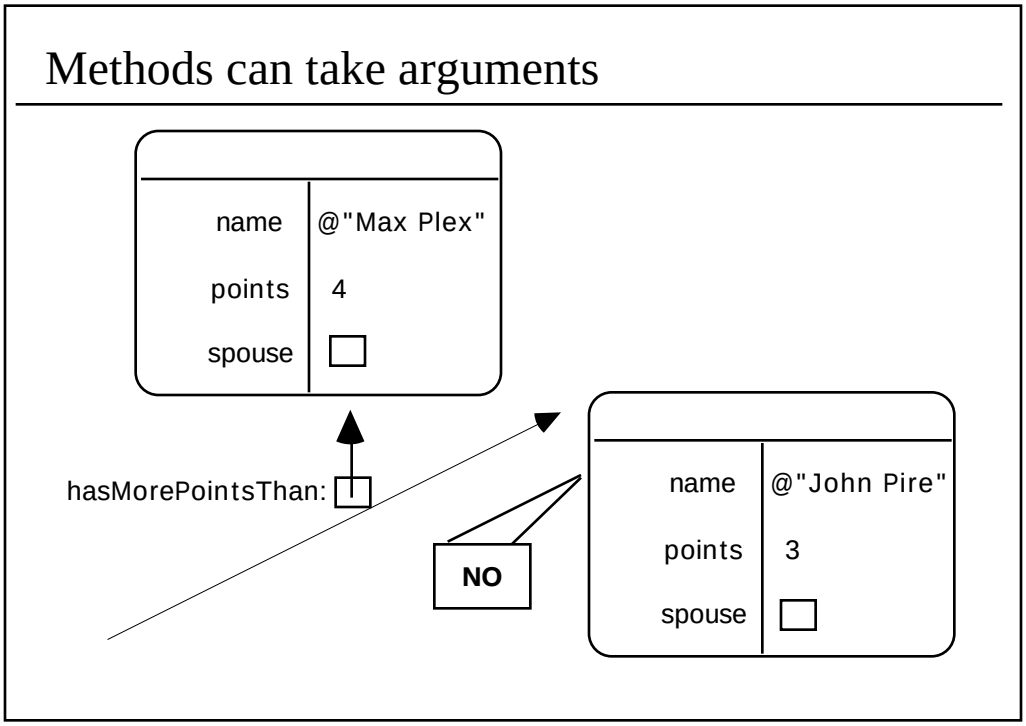
In the diagram, this operation takes place in three steps:

1. Some client sends **hasMorePointsThanSpouse** to the person object whose name is "John Pire." This is message 1 in the diagram.

2. To respond to this message, the person object needs to know how many points its spouse has. The person object sends a **points** message to the object pointed to by its **spouse** pointer. This is message 2 in the diagram.

3. The person object compares the value returned by its spouse with its own number of points, and returns YES.

Most activity in an object-oriented program is this sort of cooperation. Objects know about other objects through pointers. Objects send messages to the objects they know about. It's useful to think of objects as actors. To find out how many points its spouse has, the person object asks—it sends a **points** message to its spouse.

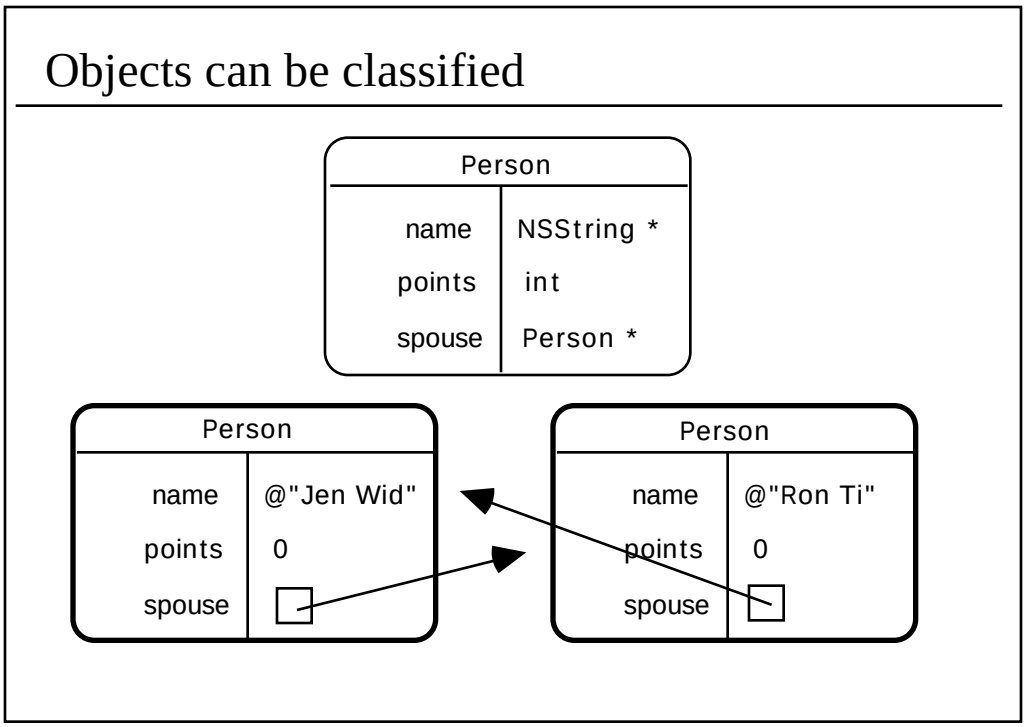


Methods can take Arguments

Messages like **points** and **hasMorePointsThanSpouse** are fairly static. They assume there's an existing object network, and clients are only interested in querying that network. This is very limiting.

To allow for increased flexibility, methods can be written that take arguments. For example, the Person class could have a method **hasMorePointsThan**: that takes another person as an argument. When you send the **hasMorePointsThan**: message, you include a pointer to another person. The method has access to this pointer and can query the other person about how many points it has.

As with return values, arguments can be of any C type, including a pointer to another object. Methods can take multiple arguments. For each argument expected by a method, there is a ':' in the method name. Again, methods are very similar to C functions.



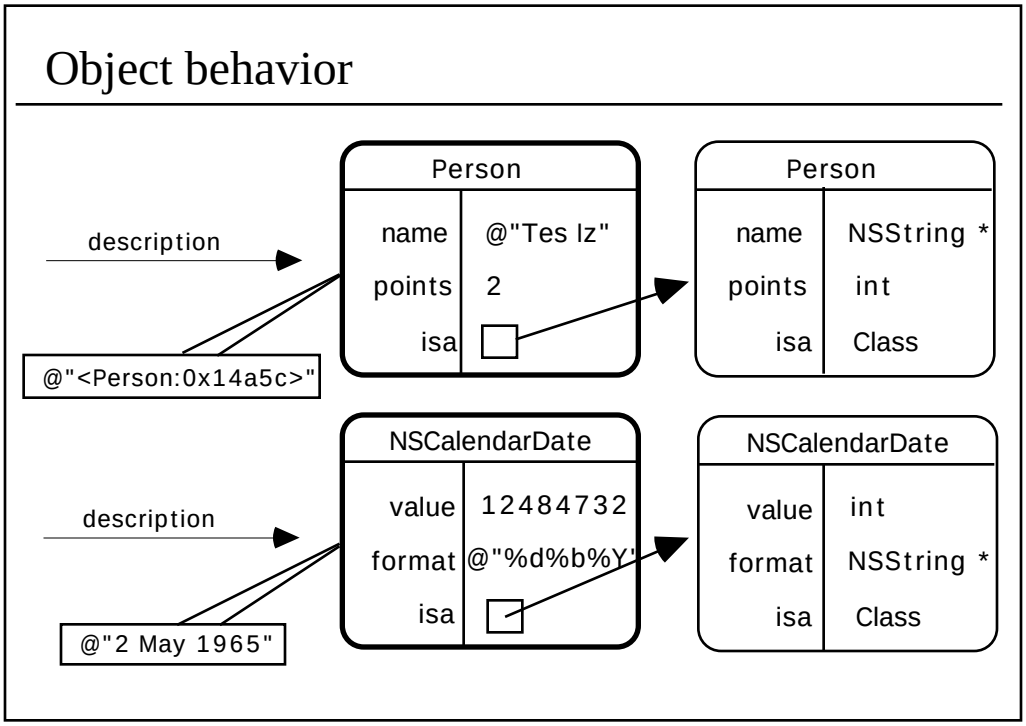
Objects can be classified

When you think about objects in the real world, you tend to classify them. For example, human beings can be classified as "people." Similarly, objects can be classified. In the previous example, you saw two different objects that were both classified as "person objects." Objects of the same type are associated with the same class.

A **class** is like a factory that can create objects of the same type. For example, take the Person class. The Person class can create objects representing people. These objects are known as **instances** of the Person class.

Every object is an instance of some class. Different types of objects are instances of different classes. If a program contains information about people and cars, it probably has a Person and Car class. When you run the program, the Person and Car classes each create several instances of themselves.

The information a class contains can be thought of as a blueprint for an object. The class has a list of instance variables. It also defines methods for its instances. Each object the class creates has all the instance variables on the list. Each object the class creates responds to messages according to the methods defined in the class. To create a new type of object, a programmer must write a new class.



Object behavior

The behavior of an object depends upon the class that created it. Objects of different classes can respond to the same message in different ways.

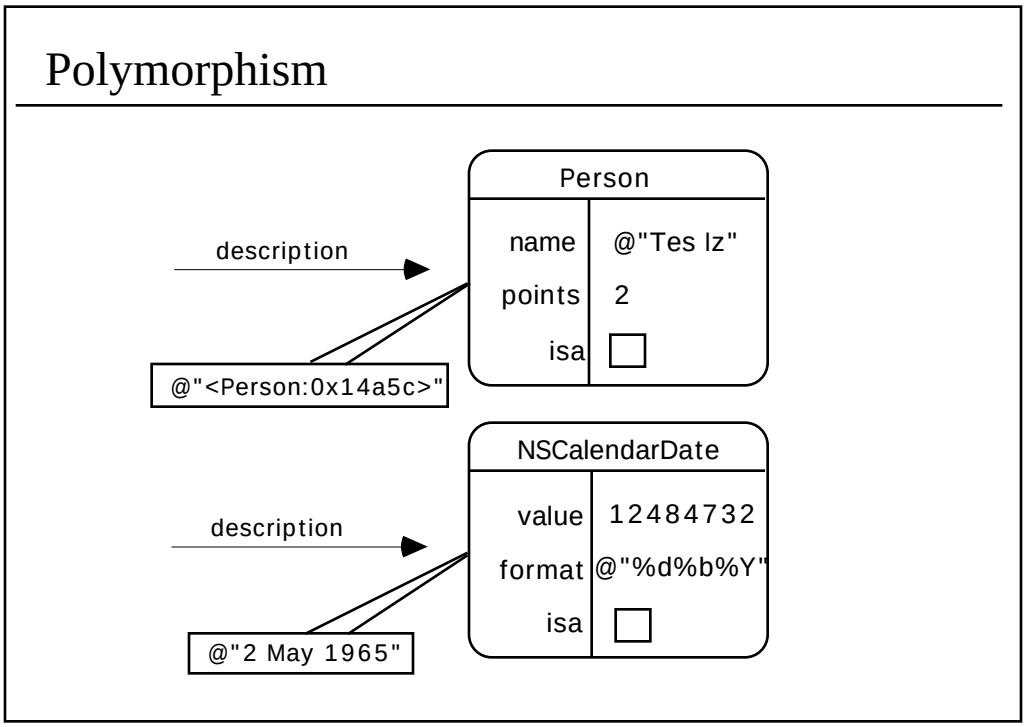
For example, all objects respond to the message **description**. When an object receives a **description** message, it returns a string describing itself. Most objects return a string containing the name of their class and the address in memory where the object instance resides. For example, an object of the Person class located at memory address 0x270e8 would return the following description:

```
<Person: 0x270e8>
```

While this type of description is accurate, it's not always the most useful thing. For example, when you ask a string for its description, you'd expect to get some sort of representation of its contents. Therefore, a string simply returns a copy of itself. A date returns a neatly formatted representation of itself, such as this:

```
22 November 1990
```

When an object receives a message, it looks up the corresponding method in its class. How does an object remember which class created it? Every object has an instance variable **isa**. The **isa** pointer contains the address of the object's class.



Polymorphism

Note that with **description**, objects in different classes respond to the same message in different ways. However, the result is the same: a good text representation of the object. This concept is known as polymorphism. **Polymorphism** is the ability of two objects to respond to the same message in different ways with analogous results.

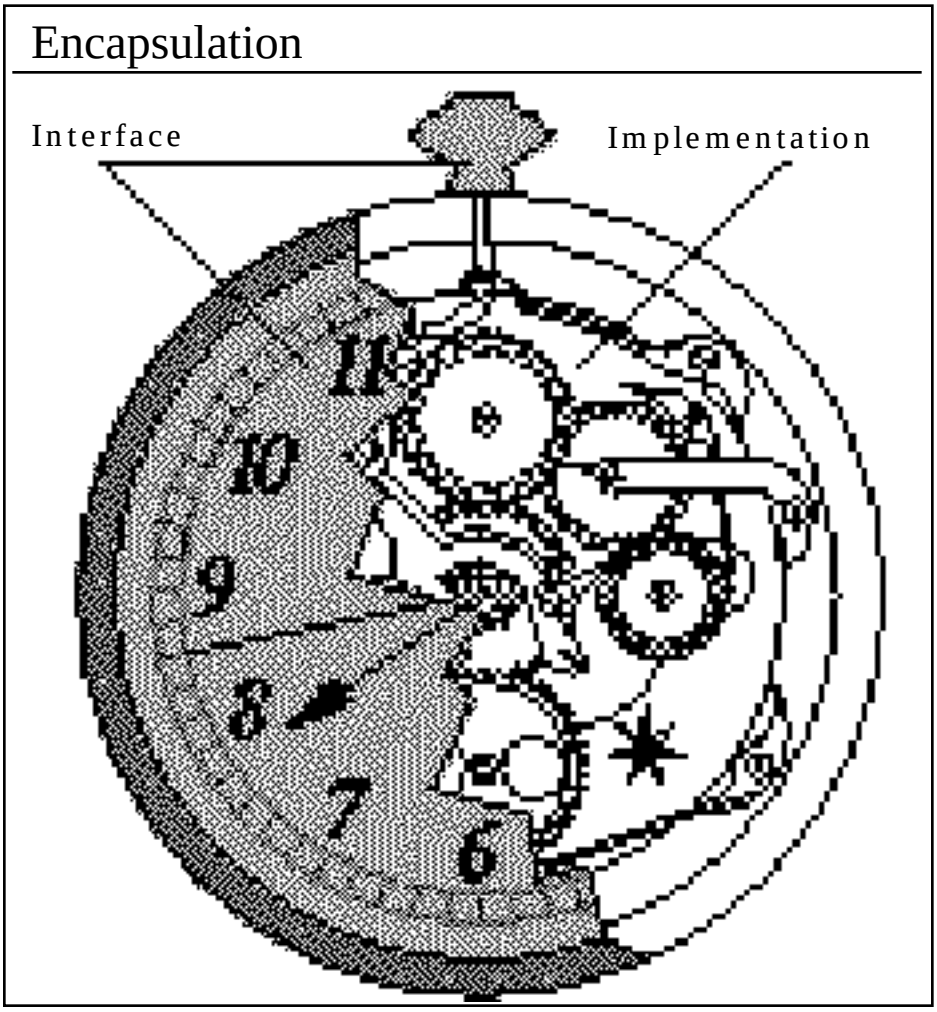
The way polymorphism is implemented is by providing a separate namespace for method names in each class. This allows two classes to have methods with the same name, but different implementations. Without polymorphism, method implementations would be unmaintainable rat's nests of conditional statements. For example:

```
if (theObject's class is Date)
   theDescription = [theObject describeAsDate];
else
   theDescription = [theObject describeAsPerson];
```

With polymorphism, this code becomes:

```
theDescription = [theObject description];
```

This code is reusable with many different classes of objects. It also means fewer method names for a programmer to learn and remember.



Encapsulation

The only way to manipulate an object is to send it messages. Only the methods of an object are allowed to read or change the values of its instance variables. You cannot directly access an object's instance variables from outside the object. They are "encapsulated" in the object.

The **interface** of a class is the list of methods provided by objects of that class. People using instances of your class only have access to the interface. They don't know how your class does something, they just know how to access it. For example, the interface to a watch is the watch face, minute hand, and hour hand. Using this interface, someone using a watch can find out what time it is, even if they don't know how a watch works.

The **implementation** of a class is its instance variables and methods. These are the details of how the work gets done. In the clock example, the gears, springs, and fly wheels are the implementation. The way the watch tells time is through its implementation. The way the watch makes the time available to others is through its interface.

When you use a class, you only need to know the interface. When you write a class, you need to create an implementation that makes the interface work. **Encapsulation** is the separation of interface from implementation.

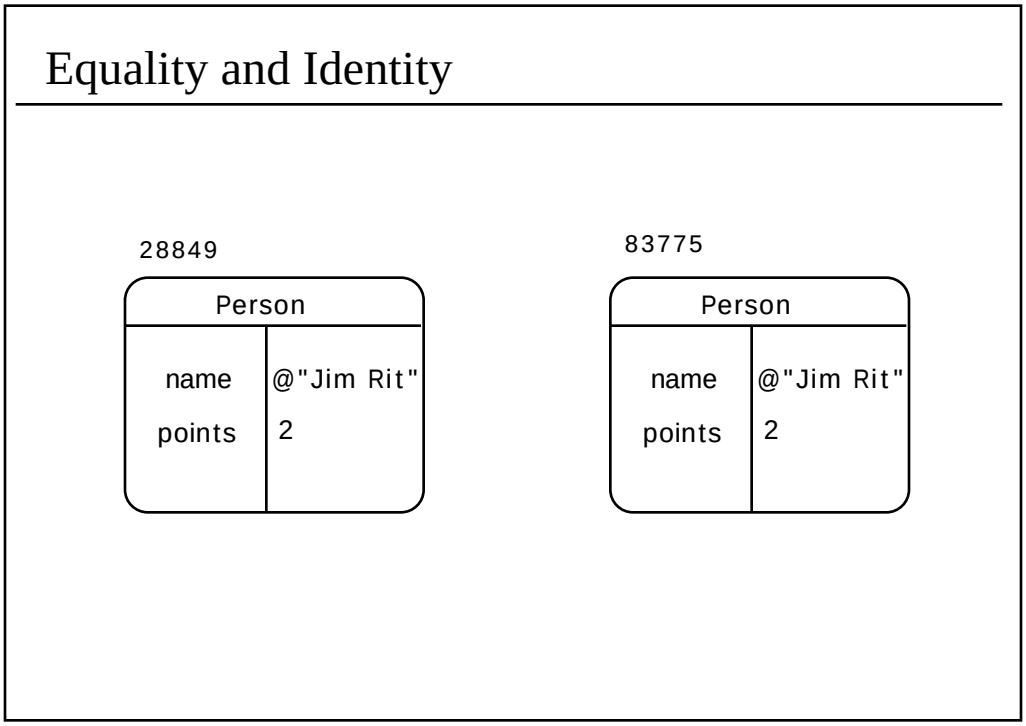
Benefits of Encapsulation

Encapsulation has two main benefits:

1. To use a class, you need to have its interface, but not the implementation. Encapsulation makes it easier to use a large framework of classes by providing a layer of abstraction that hides the implementation. When you use a class you don't need to know **how** an instance does something, you just need to know **what** the instance does. Without thinking about the detail of a class, you can think at the level of the object itself. This shifts the problem solving to a higher level, namely, that of real world objects.

2. Encapsulation makes it possible to change implementations. As long as the interface doesn't change, clients of the class won't know or care that the implementation changed.

   For example, imagine you wrote a Person class with the instance variable **yearOfBirth**, an integer. Later on, you decide it would make more sense to use **dateOfBirth**, a pointer to a date object. You could rewrite the class to use the **dateOfBirth** date object without changing the interface. Because other parts of the program only use the interface to the Person class, changing the implementation won't break them.



Equality and identity

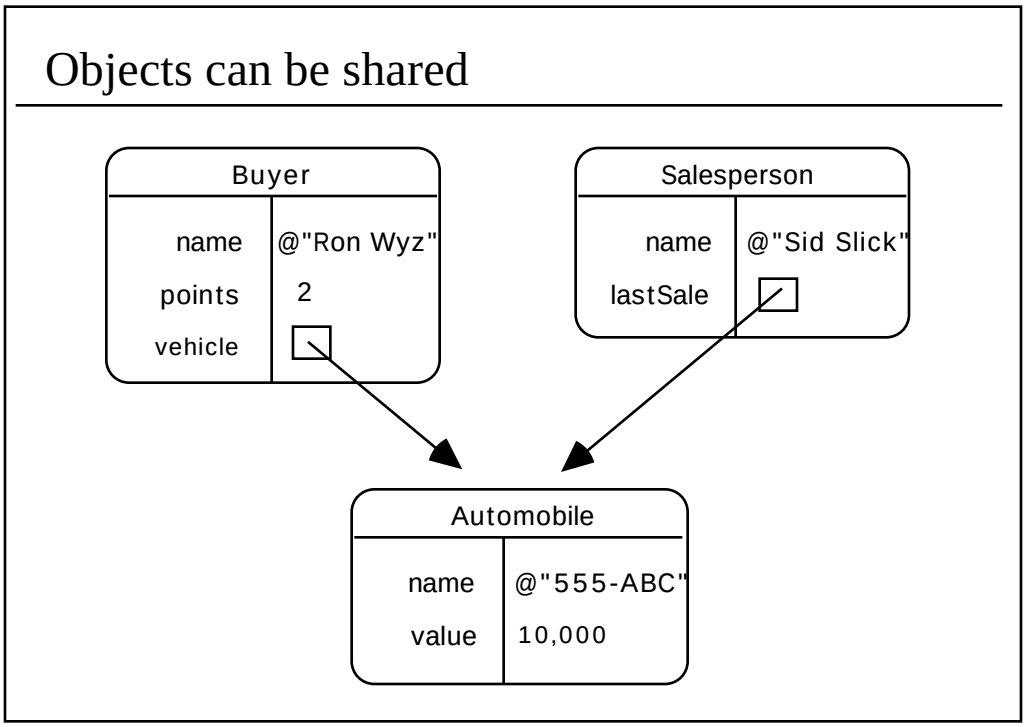
It's possible for two different objects to be of the same class and contain the same data. These objects are distinguishable only because they live at different addresses in memory. However, in many ways the two objects are equivalent. After all, they're of the same class and have the same data.

In order to be able to talk about this type of situation precisely, it's useful to define some terms for use when comparing pointers to objects.

- Two object pointers are **identical** if they point to the same address in memory. They point to the exact same object.
- Two object pointers are **equal** if they point to equivalent objects. Generally this means the objects have the same class and the same data. In real terms, it means that if you send the **isEqual:** message to one object with the other object as the argument, the return value will be YES.

For example, imagine two string objects. Each contains the string "dog". However, the two objects are located at different addresses in memory. These two objects are equal, but not identical.

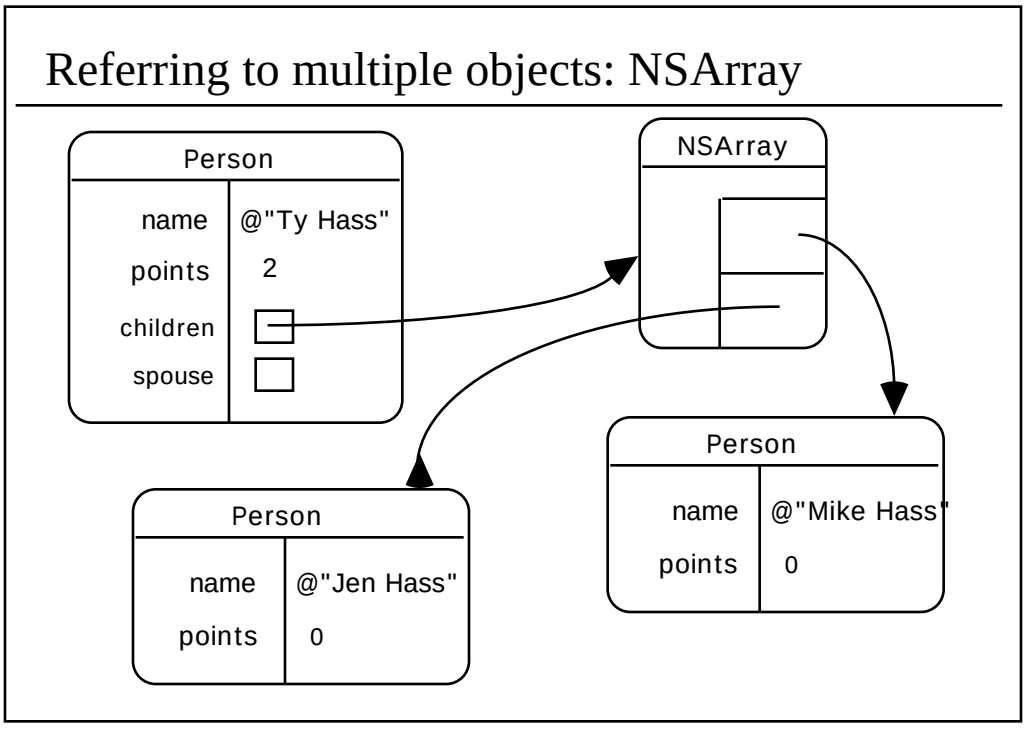
Identity is the property of an object that distinguishes it from all other objects. In nearly all object-oriented languages, identity is determined by the memory address of the object.



Objects can be shared

When constructing a network of objects, two different objects might both want to reference some third object. For example, both the buyer and the salesperson would be interested in the same automobile. The buyer wants to purchase the automobile, the salesperson wants to sell it. Both need to know how to query the automobile for information about its price, color, features, or other attributes.

This type of object sharing works very easily. Two different objects can both have instance variables that point to the same third object. Because neither the buyer nor the salesperson have their own private copy of the automobile, any changes made to the automobile are visible to both. For example, if the factory installs an air conditioning system, both the buyer and the salesperson could find this out by querying the automobile.



Referring to multiple objects: NSArray

Sometimes it's useful for one object to refer to many other objects, all of them conceptually the same type of thing. For example, imagine that you want the Person class to include a reference to any children the person might have. A person might have zero, one, or more children. How can we implement this?

One option would be to create a fixed number of instance variables named **child0**, **child1**, **child2**, on up to the maximum number of children you think is reasonable. This suffers from a number of flaws. For example, what if someone has a different idea of how many children are "reasonable?"

A much better solution is to have a single instance variable that can refer to a variable number of children. To get this sort of behavior we use **collection classes**. The most common collection class is NSArray. An instance of NSArray can hold references to any number of objects. Using NSArray, you would create a single instance variable, **children**, that pointed to an NSArray. The NSArray would in turn point to however many children the person had.

Arrays are represented in diagrams using a special symbol. While an array is an object like any other, it is very important to later sections of this course. Arrays have a special symbol in diagrams so they're easy to recognize at a glance.

Important ideas from this section

- Objects are pieces of memory
- Objects store information about themselves in instance variables
- Instance variables can be pointers to other objects
- Objects have behavior as well as data
- You invoke an object's behavior by sending it messages
- Classes are blueprints for objects
- Classes create objects
- An object's behavior depends on its class
- Objects of different classes can respond to the same message in different, analogous ways
- Encapsulation is the separation of interface from implementation
- Objects can be shared

REVIEW INTRODUCTION TO OBJECTS

1. What is an object?

2. What is a class?

3. How do you get an object to do something?

4. Describe polymorphism:

5. The following is a quotation from Object-Oriented Programming: An Evolutionary Approach by Brad Cox:

> "Before the industrial revolution, the firearms industry was hardly an industry at all but a loose coalition of individual craftsmen. Each firearm was crafted by an individual gunsmith who built each part from raw materials. Firearms produced in this way were expensive and each was the distinctive product of a gunsmith's personal inspiration.
>
> The revolution was sparked when Eli Whitney received a large manufacturing contract to build muskets for the government. Whitney's innovation was to divide the work so that each part was produced by a specialist to meet a specified standard. Each gunsmith focused on a single part, using sophisticated tools to optimize that task. This produced economies of scale that drove down manufacturing costs, and best of all, Whitney's customer, the government, quickly realized that the standards would allow parts to be interchanged, greatly simplifying their firearm repair problems."

What does this have to do with encapsulation?