*Chapter 2*

# Creating Classes

## Thinking like a Class Producer

## A View from the Inside

# Chapter 2 Creating Classes

## Goal

To cover the concepts and code syntax for creating new classes.

## Prerequisites

*Chapter 1—Using Objects.*

## Objectives

At the end of this lesson, you will be able to:

- Define a new Java class
- Implement accessor methods
- List the different access modifiers
- Overload methods
- Override inherited methods
- Define and implement an interface
- Implement one or more constructors
- Define static variables and methods

## To code in Java, you define new classes

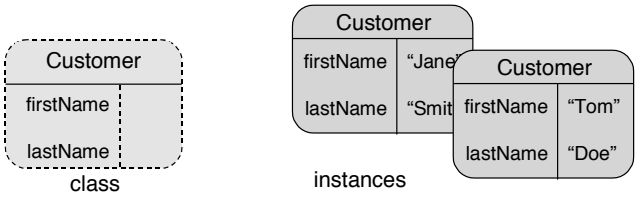
The class is the basic code packaging mechanism in Java

All code must be part of a class definition

To write Java code, you must define new classes

A class is a type, a blueprint for creating objects of that type

When you create a class, you create a new object type




### To code in Java, you define new classes

The class is the basic code packaging unit in Java. If you write any Java code at all, you must create at least one new class to contain the code. Initially, it is useful to think of yourself as merely a class consumer—you use objects. But you must also become an effective class producer—you must design and implement new classes.

A class is a type in Java. A new class defines a new type of object that you can create. Every object is an instance of a specific class. A class essentially provides a blueprint for creating objects. The blueprint defines the attributes and the behavior that belongs to each object, each instance of the class. Sometimes, a class is called a factory to vividly illustrate its function—a class manufactures objects.

## Classes derive from an inheritance hierarchy

Object is the *root class*

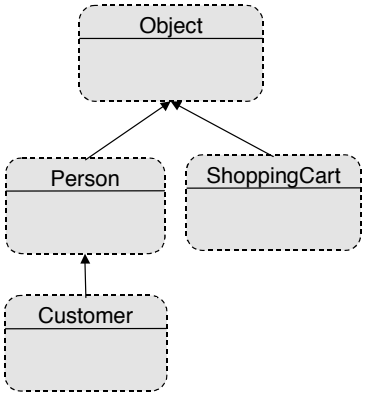
Every other class is a *subclass*

Every subclass has one *super class*

To make a new class, create a subclass

Every subclass inherits

- Instance variables
- Methods
- Interfaces




Until you specialize with code, every subclass is just like its superclass

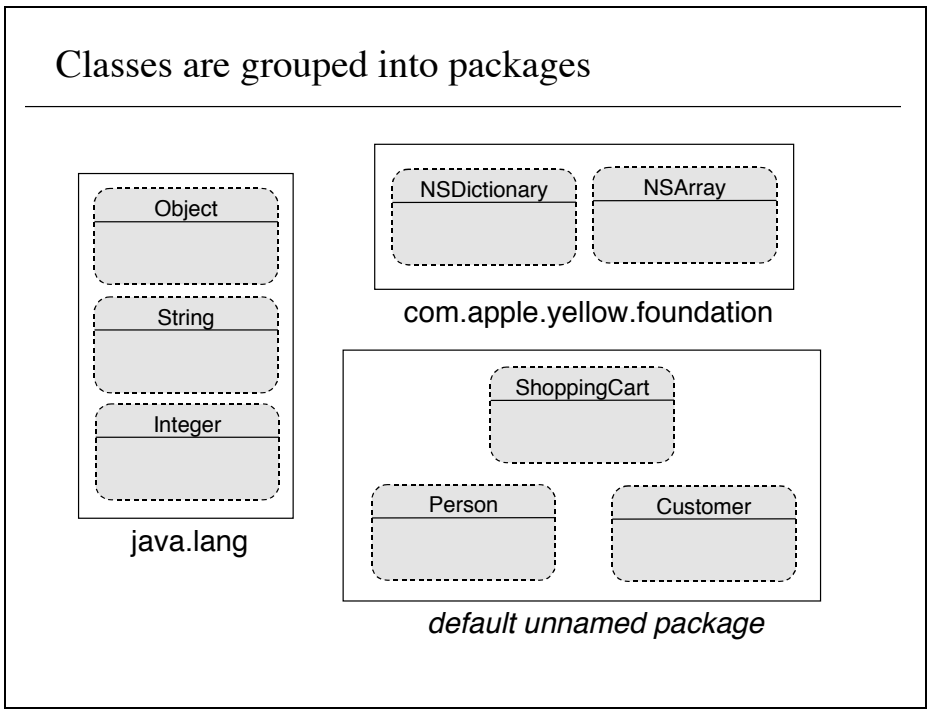
### Classes derive from an inheritance hierarchy

A class is defined within a hierarchy of related classes. This reflects our common tendency to classify something according to how it is similar to as well as different from something else.

The top of the hierarchy is called the *root* or *base* class. This is the most generic class called Object. It defines the basic attributes and behavior common to all classes. All other classes are direct or indirect *subclasses* of Object.

Except for the root class, all classes have one *superclass*. Java does not support multiple inheritance. The diagram shows that Person is a subclass of Object. Object is the superclass of Person. Customer is a subclass of Person; Person is the superclass of Customer.

A subclass is a specialized version of its superclass. A customer is a person, but somehow specialized to be, more specifically, a customer. A customer inherits all the properties and behavior of a person so that a customer can be treated generically as a person. But a customer has more specific properties and behaviors. Because a subclass inherits qualities from its superclass, the classification hierarchy is usually called an inheritance hierarchy.




## Classes are grouped into packages

Java defines another mechanism for organizing code: a package. A package is a collection of one or more related classes. Each package addresses a specific area of functionality. Classes within a package are typically used together to solve a particular design problem. Java provides several packages: basic language features, file I/O, networking, and so on. WebObjects defines additional packages: basic foundation classes, database connectivity, web application classes, etc. Every package has a name.

You can create your own custom packages. The diagram shows a package of classes specific to a shopping cart application: ShoppingCart, Person, and Customer. These classes are related in that they are used together to create e-commerce applications.

Every class must be placed in a package. Your Java code files can explicitly designate which package your class should be added to. When you don't specify a package, your class is placed in the unnamed default package. This is the common case in a WebObjects application unless you are creating classes that will be reused in multiple applications.

## What you can do in a new subclass

Add new methods—extend the superclass's behavior

Override inherited methods—modify the superclass's behavior

Add new instance variables—add new attributes

Implement an interface—add a well-defined set of methods

Implement one or more constructors—control initialization

Add static variables and methods—class rather than instance behavior

### What you can do in a new subclass

When you create a new class, you always create a new subclass of an existing class. You are creating a more specialized version of the superclass. You automatically inherit the qualities of the superclass. From there, you can specialize. Put simply, you can either add new functionality or replace existing functionality.

To extend the functionality of the superclass, you can add new methods and instance variables. You can also add static—class—methods and variables. You can implement an interface by adding the specific set of methods it defines. You can add one or more constructors to control the initial state of objects created from your class.

You can also modify existing functionality inherited from the superclass. You do this by *overriding* inherited methods. By re-implementing methods already defined in the superclass, you can extend or completely replace existing functionality.

Your first decision is which class to use for the superclass. Look for a class of which your new class will be a specialized type. For example, a customer is a kind of person. If you were creating a Customer class, you could base it on an existing Person class. Often, you will not have a related superclass to start with. In these cases, use Object as the superclass.

## Access modifiers enforce encapsulation

| Keyword | Who has access outside the class |
|---|---|
| **public** | Any class |
| **protected** | Only subclasses |
| **private** | No class |
| *package* | Other classes in the same package |

Note: there is no keyword for *package*—it is the default

### Access modifiers enforce encapsulation

As a class designer, you must think about what aspects of your class you wish to *encapsulate*—hide—and which you want to expose—make public. Encapsulation hides private internal details behind a carefully controlled public interface. Class consumers use the class name, a set of public methods, and sometimes a set of public instance or class variables. The rest of the internal class details should be hidden, available only to you, the class producer.

Class consumers are divided into three different groups: unrelated classes, subclasses of your class, and other classes in the same package.

Java defines keywords that you use with class, method, and variable definitions to restrict access to consumers of your class. They are called access or visibility modifiers. There are three keywords and four different settings:

- **public**—available to any other class.
- **protected**—available only to subclasses.
- **private**—available to your class alone.
- package—available to other classes in the same package. There is no keyword for package access—it is the default.

Instances of your class—objects—can always access instance variables and methods regardless of the access modifier used to define them.

## A typical class template

***ClassName*.java**

```
import necessary packages

public ClassName extends SuperClassName {

  instance variables . . .

  methods . . .

}
```

### A typical class template

Here is a typical class template. It shows all the parts needed for a complete class definition.

A class definition is stored in a file with the .java extension. The file name must match the class name. It is possible to have multiple class definitions in the same file but only one public class is allowed and it must match the file name.

For any existing class names you use in your class definition—your superclass, variable, and method types—you must import the packages that define them. Remember that the java.lang package is imported automatically. You can use basic classes like Object and String without an **import** statement.

The formal class definition starts with a statement naming the new class, the **extends** keyword, and the superclass. Most often, your class is public. If you omit the **extends** keyword and the superclass, the superclass defaults to Object.

The rest of the class definition is a code block of statements that define variables and methods belonging to the class. The body is enclosed with braces. Everything about the class appears within its body.

## A simple complete class example

**Person.java**

```
public Person extends Object {
  private String name;

  public String name() {
      return name;
  }
  public void setName(String value) {
      name = value;
  }
}
```

### A simple complete class example

Here is a complete example of a simple Person class.

Person is a public class. Person is a subclass of Object.

A Person has one instance variable called name of type String. The name instance variable is private; it is encapsulated and cannot be accessed directly by any consumer outside of the Person class. To get a person's name, consumers must use an accessor method.

The Person class defines a pair of public accessor methods for the name instance variable. Consumers can get and set a person's name using these methods.

Familiarize yourself with the format of this simple class definition. The details of variable and method definitions are explained in the following pages.

## Adding new instance variables

Each object—an *instance* of the class—has its own set of *variables*

Define new instance variables outside of method blocks

```
protected String name;
private int count;
```

Values are automatically initialized to ‘0’

- Object references are *null*
- Primitive number types are 0

You can include initializer expressions

```
private int count = 1;
NSMutableArray items = new NSMutableArray();
```

### Adding new instance variables

A class is a template for creating instances of that class—objects. From one class, a consumer can create many objects. Each object maintains its own state with a set of variables whose values are unique to that object. A class defines the set of variables that each object instance gets when it is created. For this reason, they are called instance variables. In Java, they are also called fields.

Each variable is defined within the class definition block but outside of any method code blocks. Each definition includes a variable name, a type, and typically an access modifier. Instance variables are most often private or protected.

Instance variables are automatically initialized to 0 when the object is constructed. Object references are null, numbers are 0, and so on. You can include initializer expressions with the variable definitions for non-0 default values.

## Adding new methods

A method has a *name* and usually an access modifier

A method has a *type*—the type of value it returns

- Any Java class, interface or primitive type
- **void** for no return value at all

A method has an *argument list*—zero or more typed variables

A method has a *body*—a code block including local variables

Method code uses the **return** keyword. **return** does two things

- Leaves the method, returning to the caller
- Specifies the return value for non-void methods

### Adding new methods

You can add new methods to your class definition. A method definition includes the following parts usually listed in the following order:

- Access modifier
- Return type
- Name
- Argument list
- Code block or body

Methods are typed: they return a value of a specific type. Methods that return no value at all use the key word **void**.

Arguments passed by the caller become local variables with the method body. The method body is a code block and can define additional local variables that are temporarily valid only while the method is executing.

Method code usually includes the **return** keyword. It does two things. The return keyword stops executing the method code and returns to the caller. It also takes a value, the value to return to the caller. If the method is **void**, use the **return** keyword without specifying a return value.

## A variety of method declarations

No return value, no arguments

```
public void start()
```

Return value

```
protected int count()
```

Return value and argument

```
private String encrypt(String value)
```

Return value and multiple arguments

```
Double average(Integer value, Integer value2)
```

### A variety of method declarations

Here are a variety of different valid method declarations. For clarity, they exclude the method body so they are incomplete.

By convention, the access modifier appears first. Notice that each of the four examples uses a different access modifier. The last example that uses no access keyword at all indicating package-level access.

Every method must declare the type of value it returns. Use **void** if the method returns nothing. Return types can be primitive types—like int—or object types—like String. Methods can also use Interface names for return types.

Method names follow a naming convention: like instance variables, they begin with lowercase letters but capitalize each subsequent word.

The argument list specifies zero or more typed arguments that must be provided by the caller. For zero arguments, the list is empty. Each argument has a type and a name. Multiple arguments are separated by commas.

## A simple complete method example

```
public double balance() {
  // local variable for calculating balance
  double balance = 0;
  // enumerate objects in instance variable items
  Enumeration e = items.objectEnumerator();
  while(e.hasMoreElements()) {
    balance = balance +
           ((Asset)e.nextElement()).balance();
  }
  return balance; // leave method returning value
}
```

### A simple complete method example

A complete method example is shown above. It is a public method named **balance**(). It returns a double value and takes no arguments. Imagine that this is the method for calculating the balance of items in a shopping cart. This method would be defined with the ShoppingCart class.

To calculate the balance, the method defines a local variable called balance and initializes it to 0. In Java, you can use the same name for a variable and a method without conflict. Local variables should generally not use the same names as instance variables.

The balance is calculated by adding the balance of each individual item in the collection named items. Notice that items is neither a method argument nor a local variable. It is most likely that items is an instance variable of this class. Remember that methods can reference instance variables directly.

Once all items are enumerated, the balance it computed. The method finishes execution and returns the final balance using the return statement.

## Each instance can access its own data and behavior

Instance variables are directly accessible from methods

```
private int count;
public void increment() {count++;}
```

One method can invoke another

```
public void increment() {count++;}
public void addOne() {increment());
```

Every object has a reference to itself named **this**

```
public void increment() {
  this.count++;
}
public void addOne() {this.increment());
```

### Each instance can access its own data and behavior

While methods can take arguments and define new local variables for temporary purposes, methods are chiefly concerned with instance variables. Instance variables are automatically accessible to methods defined in the same class. Simply refer to them by name.

A method can call other methods defined in the same class. Remember that methods are normally invoked by sending a message to an object reference:

```
anObject.someMethod();
```

Java provides a shorthand for an object to send a message to itself. If the object reference is excluded from a message expression, the implied reference is reflexive—send the message to the object sending the message:

```
someMethod();
```

Java provides a keyword to make this explicit:

```
this.someMethod();
```

It is optional. You can use it for clarity.

Unlike functions or procedures in traditional languages, methods can only be invoked through an object reference. Whether it is explicit or implicit, there is always an object associated with the currently executing code. The data and behavior are never separated and can never be mismatched. This is one of the hallmarks of object-oriented programming.

## Accessor methods encapsulate instance variables

Encapsulated instance variable is private or protected

```
protected String name;
```

Public *accessor methods* provide and control access

```
public String name() {return name;}
public void setName(String value) {
   name = value;
}
```

Alternative naming convention for get method

```
public String getName() {return name;}
```

### Accessor methods encapsulate instance variables

The essence of encapsulation is barring direct access to instance variables. Encapsulated instance variables are protected or private. Outsiders must access values indirectly through methods rather than directly through instance variables. This provides a hook for arbitrary processing within the object. This maintains a firewall so that you can modify the implementation without compromising the public interface and breaking existing code.

Methods used to access instance variables are called accessor methods. They generally come in pairs—one for getting the value, one for setting the value. The method for setting the value is sometimes called a mutator method. It is possible for read-only or computed values to have get methods not set methods. It is even possible to have accessor methods but no underlying instance variable. This is the case for values that are derived from other data.

There are two different naming conventions for get methods. One form simply uses the variable name as the method name. The other prepends the word "get". WebObjects most often uses the former convention for its framework classes. Java packages vary from class to class. As a class designer, you can choose either or possibly implement both.

## Overloading methods—same name, different types

Overloading—multiple versions of a method with different arguments

Each is distinct due to unique number and/or type of arguments

```
double balance()
double balance(double discount)
double balance(BigDecimal discount)
double balance(double discount, NSArray coupons)
```

You cannot change the return value type

```
double balance()
int    balance()   // will not compile
```

### Overloading methods—same name, different types

Java supports method *overloading*. Overloading means defining multiple versions of a given method, each with a different and distinct argument list. Notice that the method name and the return type must stay the same. Conceptually, each method version should produce analogous behavior.

Overloading is useful when the same action can be performed based on different sets of parameters. Consider the shopping cart example. You can calculate the balance in different ways. You can calculate the simple balance of all items in the cart. You can specify a discount rate as an argument. Sometimes, the discount rate is a primitive type, other times, it is a number object. Occasionally the customer has an additional set of coupons.

As a class designer, you can implement multiple versions of the balance() method, each taking a different set of arguments. This is called overloading the balance() method. It creates a much more flexible class design which is potentially reusable in multiple scenarios.

Remember, overloaded methods differ by the number and type of arguments. The name remains the same. You cannot vary the return type.

## Overriding inherited methods

Overriding—replacing the implementation of an inherited method

Re-implement the method, using the same name, type and arguments

You have two choices

- *Replace* the superclass behavior
- *Extend* the superclass behavior

To extend the behavior, include a call to the superclass method

Invoke the superclass method using the keyword **super**

### Overriding inherited methods

*Overriding* is different from overloading. Overriding a method means changing the behavior of a method inherited from a superclass. When a class inherits a method, it responds to it as though it had implemented the method itself. Often, you need to modify the response to an inherited method. While a subclass cannot make the method away, it can change the implementation. Besides adding new methods, you can change the implementation of existing methods.

Overriding means providing a new implementation of an inherited method without changing the name, the arguments, the return value, or the accessibility. Conceptually, you have two different choices:

- Completely replace the implementation—forget the superclass's version
- Extend the implementation—make use of the superclass's version

To extend the superclass's method, reuse it as the core of your new logic. When your class implements a method for which the superclass also has a version, you need a way to differentiate the two. You need a reference to invoke the superclass's method rather than your own. Java defines the keyword **super** for this purpose.

## To extend when overriding, use the **super** keyword

Overriding to extend the superclass method

```
public double balance() {
  // call the super class implementation
  double balance = super.balance();
  // extend it
  balance = balance + (balance * taxRate);
  return balance;
}
```

Invoking overridden version of the method

`this.balance();` or simply `balance();`

Invoking the superclass's version of the method

```
super.balance();
```

### To extend when overriding, use the super keyword

Suppose that a shopping cart class implements a balance() method. It simply calculates the total of all items. You are implementing a subclass that takes tax into account. You also need to calculate a balance but include the tax as well. Since your method of calculating the balance is different, you must override the superclass version. Make use of the balance logic already correctly implemented in the superclass but merely add some additional processing. In this case, you are extending the superclass logic rather than replacing it.

A method can invoke the superclass method using the keyword **super**. Like **this**, **super** is a pre-defined object reference for sending messages. Normally, when an object sends a message to itself, it wants to find the method implementation in the same class:

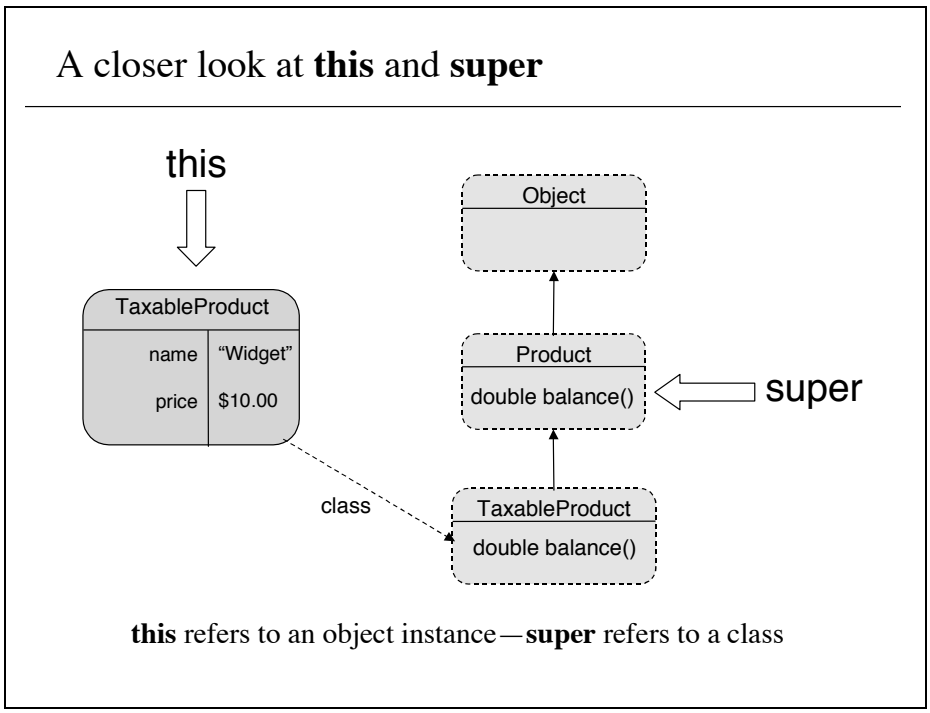
```
someMethod();
```

You can be more explicit with the keyword this:

```
this.someMethod();
```

In both cases, the statement will find the method whether the class implements it or inherits it from a superclass. When overriding, you need to bypass the implementation in the subclass and invoke the implementation in the superclass. To do so, use the keyword **super**:

```
super.someMethod();
```




## A closer look at this and super

You use the keywords **this** and **super** in similar ways: both are special, predefined references used to invoke methods. Take a moment to study how they work and especially how they are different.

**this** is a reference to the current object, the target of a message that caused the invocation of the current method. While executing the method, the current object can send a message to itself using **this**. Fundamentally, **this** is a reference to an object.

When a message is sent to an object—such as balance()—the Java runtime determines the class of the object and starts looking for a method of the same name—balance(). The search starts with the most specific class then continues "upward", visiting each of the superclasses. As soon as an implementation is found, the search stops and the method is executed. This is the essence of a mechanism called *dynamic binding*, coupled with the mechanism of inheritance.

Often, the method is implemented in the most specific class, even if superclasses also have a matching method. In this case, the specific class has overridden the method version it inherited from its superclass.

When you send a message using **super**, the search does not start with the most specific class of the object. By design, it ignores overridden methods in the current class, skipping one level upward in the hierarchy. While **this** is a reference to an object, **super** is conceptually a reference to a class.

## Constructors guarantee proper initialization

Class consumers create objects with a *constructor*

```
ShoppingCart cart = new ShoppingCart();
```

Class producers can implement a constructor to initialize the object

```
public ShoppingCart() {
   items = new NSMutableArray();
}
```

The constructor name is the same as the class name

The constructor has no return type, not even **void**

If you don't provide a constructor, Java generates one by default

### Constructors guarantee proper initialization

As a class producer, you need to control the initial state of a newly created object. You need a reliable mechanism for initializing the object before the consumer gains access to it. Java guarantees the proper initialization of objects with a special type of method called the constructor. By coding a custom constructor, you can control an object's initial state.

A class consumer creates a new object by calling the constructor with the keyword **new**. As a class designer, you can implement the constructor to perform any actions necessary to initialize the state of the object. Typically this means assigning default values—additional objects—to instance variables.

While a constructor looks much like any method, it has some special properties. The constructor name is the same as the class name. Most constructors are public. And a constructor has no return value, not even void.

You don't have to implement a constructor. If you don't, Java provides a default constructor for you. This enables a class consumer to call the default constructor even when you don't write one. The default constructor takes no arguments and leaves the instance variables your class defines in their default state—0 or null values. Since the default constructor takes no arguments, it is often called the "no-arg" constructor.

## Every superclass plays a role during construction

A Customer is a Person and an Object
- Each superclass contributes functionality
- Each superclass gets to initialize

Constructing an object . . .

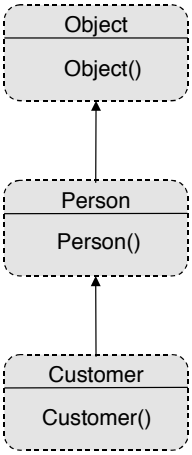
```
new Customer()
```

Involves multiple constructors

```
Object()
Person()
Customer()
```

Java ensures that all constructors are called

Constructors are executed from top down




### Every superclass plays a role during construction

Remember that your class is defined within an inheritance hierarchy. Even the most basic class has at least one superclass—Object. Because of inheritance, the set of instance variables in an object is a combination of variables defined by multiple superclasses in the hierarchy. A Person class defines the first and last name variables. The Customer subclass adds the customer number, a sales person reference, and so on. Proper construction of a new object requires that each participating class gets a chance to initialize the variables it defines.

Java ensures that when a constructor is called for a particular class—Customer for example—the constructor for each of its superclasses is also called—Person and Object. This is called constructor chaining. The constructors are executed from top down, from the most general to the most specific class. In this example, the sequence is Object, then Person, then Customer.

## Calling the superclass constructor

A constructor can call the superclass constructor explicitly

```
public Customer() {
  super();    // must be the first statement
  name = "Jane";
}
```

Otherwise, it is automatically called by the Java runtime implicitly

```
public Customer() {
  name = "Jane";
}
```

### Calling the superclass constructor

The practical implication of constructor chaining is that superclass constructors will be executed before subclass constructors. By the time your constructor begins any custom initialization logic, the superclass portions of the object have already been initialized. In the current example, when initializing the state of a new customer, you can assume the Person part of the object is ready to use.

You can explicitly call the superclass constructor using the **super** keyword. Use it as though it were a method name:

```
super();
```

If you call the superclass constructor explicitly, it must be the first statement of your constructor. If you insert any statements before the superclass constructor call, your code will not compile.

You can omit the explicit call to the superclass constructor: the Java runtime will call the constructor for you.

Why would you call the superclass constructor explicitly if Java does it for you automatically? Classes can define multiple constructors. They differ in the number or type of arguments they take. When Java automatically calls the superclass constructor, it calls the default no-argument constructor. If you want a different constructor, you must call it explicitly, for example:

```
super("jane","Doe");
```

## Multiple constructors with overloading

You can provide multiple constructors by overloading

One constructor can call the other using **this**

```
public ShoppingCart() {
  super();
  items = new NSMutableArray();
}

public ShoppingCart(Customer newShopper) {
  this();
  shopper = newShopper;
}
```

### Multiple constructors with overloading

Your class can define multiple constructors using overloading. Overloading means reusing the same method name but changing the number or type of arguments. A ShoppingCart class might define a default no-argument constructor and a second constructor that accepts the associated customer object. This allows class consumers to create a shopping cart in two different ways. When the customer is not yet known:

```
ShoppingCart cart = new ShoppingCart();
```

or when the customer is already available:

```
ShoppingCart cart = new ShoppingCart(customer);
```

As the class designer, you can have one of your constructors call another using the keyword **this**. To call a constructor, use it at though it were a method name:

```
this();
```

This way, you can reuse the logic in one constructor as part of the logic of another without duplicating code.

The number and type of arguments you provide determines which constructor is called:

```
this();            // calls ShoppingCart()
this(customer);    // calls ShoppingCart(Customer customer)
```

## When you provide your own constructors

If you implement *any* constructors, you must implement *all* of them

If you don't provide any constructors, Java generates a default

```
public Customer() {      // "no-arg" default
  super();
}
```

If you implement *any* constructors, Java does *not* generate a default

Unless you also provide a default, your class doesn't have one

```
person = new Customer(); // will not compile
```

Constructors are not inherited—you must re-implement them

### When you provide your own constructors

If you don't implement any constructors, Java generates the default "no-arg" constructor. If you implement any constructors, Java does not generate the default. If your class consumers expect to use the no-arg constructor in this case, you must implement it explicitly.

Another special aspect of constructors is that, unlike standard methods, they are not inherited. If the Person superclass defines a one argument constructor such as:

```
public Person(String lastName);
```

this does not imply that the Customer subclass automatically responds to:

```
new Customer("Doe");
```

To enable this capability, the Customer subclass must define a matching constructor, even if it does nothing:

```
public Customer(String lastName) { ...
```

## Adding new static variables and methods

Your class can define **static** variables and methods

```
public static double TaxRate;
public static Store headQuarters() { . . .
```

They are available from the class as well as an instance of the class

```
double rate = ShoppingCart.TaxRate;
Store store  = Store.headQuarters();
```

Instance method code can reference them directly

```
double rate = TaxRate;
Store store  = headQuarters();
```

Often called *class*—as opposed to *instance*—variables and methods

### Adding new static variables and methods

You can also define static—also known as class—methods and variables. Use the keyword **static** to do so.

Static methods can be invoked using the class name. A consumer can invoke the method without first creating an instance of the class. This is a logical convenience for methods that provide services for the class as a whole, without reference to a specific instance.

Static methods can also be invoked by code in your instance methods.

Static variables are not stored in each object instance of the class, but in the class itself. This is ideal for data that applies to the class as a whole without reference to a specific instance. Even though there might be several shopping carts, there is only one copy of the tax rate applied to all shopping carts. This is clearly more efficient that storing a redundant copy in each shopping cart instance. This allows you to change the value of the static variable in one place while making it visible to each object instance.

Static methods can only access static variables. Static methods cannot access instance variables. By definition, static methods are independent of instances and have no way of accessing them or the instance variables they possess.

## Variables have a scope—visibility and lifetime

Static (class) variable

- One copy per *class*; good for the application's lifetime
- Visible to class and all objects
- Default value is 0

Instance variable

- One copy per *object*; good for the object's lifetime
- Visible to that object
- Default value is 0

Local variable

- Good while executing within the block that defines it
- Visible only to that block
- Default value undefined: be sure to initialize

### Variables have a scope—visibility and lifetime

As a class designer, you can define three different types of variables. You should understand the implications of each choice. They differ in terms of *scope*—visibility and lifetime. Visibility means what parts of your class code can "see" the variable name—where is the variable accessible. Lifetime means how long does the variable last—when it is created, when it is destroyed, how long will it maintain a particular value. Lifetime has direct implications on the memory requirements of your application.

Static variables—also known as class variables—reside in the class itself. They last as long as the class, which is generally the lifetime of the application. There is only one copy of the variable—in the class. The default value is 0. The variable is visible to the class and to all instances of the class.

Instance variables reside in the object—an instance of the class. Each object has its own variable. The variable is created and destroyed along with the object and is visible only to that object. The default value is 0. Instance variables have the greatest potential impact on your application's memory resources.

Local variables are the most transient type—they exist only while executing the method in which they are defined. They are often called temporary variables. They are visible only to the code within the method. Local variables generally have the least impact on your application's memory resources.

## Definitions of different variable scopes

```
public class ShoppingCart {
  // static variable
  public static cartCount;

  // instance variable
  protected NSMutableArray items;

  public double balance() {
      // local variable
      double balance = 0;
      . . .
  }
}
```

### Definitions of different variable scopes

This code example demonstrates the three different variable scope definitions. Notice that static and instance variables are defined outside of any method code blocks. A copy of an instance variable is allocated for every object instance that is created at runtime. The instance variable is deallocated when the object is garbage collected. There is only one copy of a static variable application-wide—it belongs to the class itself. While it is accessible to each object instance, it is shared by all objects.

Local variables are defined with method code blocks. They are valid only while executing the code block in which they are defined. Method code blocks can contain additional nested code blocks such as those used for **if** or **while** statements. These can define their own local variables as well.

Local variables differ from static and instance variables in that their default values are undefined. Be sure to initialize them before using their values. The Java compiler is smart about warning you when you forget to do so.

## Making something final

*Final* definitions cannot be modified

- Final classes cannot be subclassed
- Final methods cannot be overridden
- Final variables are constants

Typically used to define static constants

Use the keyword **final**

```
public static final double TaxRate = .08;
```

### Making something final

You can use the **final** keyword to indicate that something cannot be changed. The definition is final. You can apply the keyword to a class, a method, or a variable. Final classes cannot be subclassed, final methods cannot be overridden by subclasses and a final variable's value cannot be modified.

The **final** keyword is mostly used to make a static variable constant. Once it is initialized, it cannot be modified.

You should be careful about making classes or methods final. Inheritance allows your implementations to be refined by future subclasses. It is difficult to anticipate the future. You do not want to prematurely preclude a future subclass designer from leveraging your work without the ability to change it.

## Classes are either abstract or concrete

*Abstract* definitions defer implementation to subclasses

- Abstract classes cannot be instantiated
- Abstract methods must be overridden

*Concrete* subclasses provide non-abstract implementations

Abstract classes provide a template for concrete subclasses

Use the keyword **abstract**

```
public abstract class Asset {
   private double price;
   public abstract double balance();
}
```

### Classes are either abstract or concrete

All classes in the hierarchy fall into to different categories: *concrete* or *abstract*. Concrete classes are ready to use by class consumers—they can be instantiated. Abstract classes, as their name implies, are not complete nor directly usable. Abstract classes cannot be instantiated. They provide a template but require a concrete subclass implementation to make them usable.

Java defines the **abstract** keyword for marking a class or a method abstract. An abstract class cannot be instantiated. An abstract method has no implementation—it has no method body. If a class has at least one abstract method, it is automatically considered abstract.

Abstract classes are used to define a prototype for subclass designers. Consider the concept of an Asset. A reasonable design mandates that all assets should have a price attribute and a method for calculating the balance. Different assets may compute their balances differently. The balance() method is marked abstract and the implementation is deferred to subclass designers. Although the balance() method is abstract, it is a formal part of the class—it is required. If subclasses do not provide a complete implementation, they are also considered abstract.

## You can define new interfaces

Interfaces specify a set of methods

- Independent of class
- Independent of implementation

Interfaces can also define constants

**Asset.java**

```
public interface Asset {
  public final static double TaxRate = .08;
  public double balance();
}
```

### You can define new interfaces

You may find it useful to define a new interface. An interface defines a set of methods without providing an implementation. An interface describes how an object behaves without specifying its class or superclass type. Interfaces are not classes and cannot be instantiated. Rather, they are implemented by classes and define a valid way of typing objects from those classes.

An interface has a name and must be defined in a file of the same name followed by the .java extension. Interfaces must be public. Instead of the class keyword, use the **interface** keyword. The body of an interface—enclosed between braces—contains method declarations without the method code blocks. By definition, interfaces do not provide implementations. Interfaces can also define static constants.

Interfaces can extend other interfaces just like subclasses extend superclasses. This implies that the interface inherits methods and constants from the interface it extends. Since there are no implementations to inherit, this is usually referred to as interface inheritance. Class-based inheritance combines interface and implementation inheritances.

Since one class can implement more than one interface, one class can essentially have multiple behaviors or personalities. Java interfaces provide an alternative to traditional multiple inheritance.

## Your class can implement interfaces

Classes implement an interface by implementing its methods

**ShoppingCart.java**

```
public class ShoppingCart

          extends Object implements Asset {

  public double balance(){

      . . .

  }

}
```

### Your class can implement interfaces

Your class can implement one or more interfaces. Your class declares its superclass followed by an optional declaration of one or more interfaces that your class implements:

```
implements Asset
```

Multiple interface names are separated by a comma. Remember that you must import the package that defines the interface name unless the interface is part of the same package as your class.

To implement an interface, your class must provide an implementation for every method defined in the interface. You can inherit the implementation from a superclass as well.

Once you have implemented an interface, a consumer can reference an object of your class using at least three different types. Which is most appropriate depends on how the consumer intends to use the object—how they expect the object to behave:

```
Object cart;     // generic reference; no particular behavior
ShoppingCart cart;  // specific class with specific behavior
Asset cart;      // specific interface regardless of class
```

When possible, typing by interface is the ideal choice.

## Java naming requirements and conventions

Only one public class per file

File name matches the public class name

Only one interface per file

Without a package statement, class goes in the unnamed package

Consult the Java documentation for package naming conventions

**ShoppingCart.java**

```
public class ShoppingCart {
  . . .
```

### Java naming requirements and conventions

Java enforces some simple rules for class providers. Your class code must be stored in a file of the same name, ending with the .java extension. While you can have multiple class definitions in a single file, only one of the classes can be public. The public class name must match the file name.

Interfaces must be public and must match the filename. It follows that a file can only contain one interface. The filename must match the interface name.

You can place your class in an explicitly named package if you choose. Consult the Java documentation for information about the package statement and package naming conventions. If you omit the package statement, you class is added to the default unnamed package. Your class definition likely incorporates a superclass name, class names used for variable and method types, and possibly interface names. Be sure to import necessary packages accordingly. Class and interface names from the same package—the default unnamed package for example—need not be imported.

## Common pitfalls

Missing an import statement required for the superclass or an interface

Misspelling a method name when overriding a method

Changing the access, return value, or argument list when overriding

Missing a call to super in an overridden method

Calling super in an overridden method when you don't need it

Confusing overloading with overriding

Implementing constructors but omitting the default no-arg constructor

Missing the implementation of *all* methods in an interface

### Common pitfalls

Here is a list of common pitfalls encountered by class designers. Most result in either compile or runtime errors. Some result in unexpected behavior. Use this list as a review of topics presented in this chapter.

The more subtle pitfalls invoke overriding inherited methods. First, remember that overriding is different than overloading. Overriding is re-implementing a method you inherit from a superclass. Overloading means implementing multiple versions of the same method with different numbers or types of arguments.

To properly override an inherited method, you must use the same access modifier, return type, name, and argument list. If you change the name, you are, in effect, implementing a new, unrelated method. While it may compile, it will not match the superclass method and will not replace it—it will not be called when you expect. This happens accidentally if you misspell the name.

A properly overridden method can extend the superclass method by including a call to **super**. It can replace the method completely, in which case it should not call **super**. If you mix this logic, you are sure to get incorrect and unexpected behavior.

Another common pitfall occurs when you implement one or more constructors but forget to include the default no-arg constructor. This may be your intention but the effect is that consumers cannot instantiate your class using the default no-arg constructor.