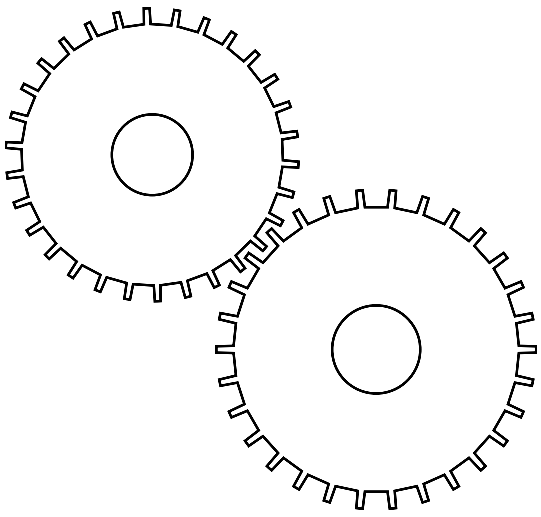


Java for WebObjects Developers

Provides a quick-start guide for developers learning to use Java with WebObjects

Produced by Apple Technical Publications. Original authored by Kai Christiansen. Updated for WebObjects 5 by Malcolm Crawford. Publication management by Katherine Wenc.

1 Introduction

Mandatory Reading—Start Here

Java for WebObjects developers—Java in 21 minutes

If you plan on building WebObjects applications, you need to become a Java programmer. Java is a popular programming language available in many diverse contexts for implementing real software solutions. But Java is more than just a programming language—it is a set of tools, a runtime with a virtual machine, a broad landscape of packages full of reusable classes. Java is an environment.

Learning Java "the environment" seems overwhelming the first time you approach it. There are reams of on-line materials, and bookstores are brimming with all kinds of Java books. It may be difficult to decide where to start, especially if your primary goal is to learn how to develop applications with WebObjects.

Aside from the glamour of applets with sophisticated graphical user interfaces, and the rigors of multi-threaded or networked programming, Java turns out to be a rather simple language for humble, general-purpose jobs running on a plain old computer. With some basic ideas, and some familiarity with the most commonly used language constructs, you can go a long way. This is especially true when your Java code is part of a larger system that handles a lot of the details already. To get started, all you need is the Java relevant for a WebObjects developer.

But Java "the language" is still something to learn. It is a general-purpose programming language, and it is *object-oriented*. The term "object-oriented" is vague and spooky to some, fresh and intuitive to others, even old hat to a few. Mostly, it means just good, modern software technology. Object-oriented programming is based on a small set of powerful concepts and somewhat specialized terminology. From this perspective, it should be clear that Java is a way of thinking.

Your job, however, is to communicate your thinking to others—a computer, or another programmer. From this perspective, Java is a way of speaking.

The basic goal—getting you to think and speak Java

While it may take a bit longer than 21 minutes to digest this guide, it will likely take a lot less of your time and energy than other approaches, while achieving similar results. The content is based on the fact that, to begin developing WebObjects applications you don't need to know everything in the Java environment, nor even everything about the Java language itself. This guide presents the Java you absolutely must know before you start.

The approach is designed to be simple and direct. Java and the world of object-oriented programming is in some ways so simple and direct that it is paradoxically confusing. The obvious meaning is somehow elusive. This is not to say that there isn't great sophistication and complexity in object-oriented design and implementation. But the basic way of thinking and the general style of coding—in Java—is clear and straightforward.

This guide follows a straightforward narrative. To see the forest for the trees—and to be convinced that the forest is a nice place to live—you need to hear a simple, useful, Java story. Without exhaustive detail, exceptions, or a survey of many different clever ways to do the same thing, the story describes typical Java usage—real and useful Java usage.

On the other hand, this guide is not a detailed or comprehensive text on either Java or object-oriented programming, and it does not claim to make them unnecessary for your success. It offers enough so that you discover what is required to develop WebObjects applications, and get started with WebObjects development. As soon as you are engaged in real work, your own experience will tell you what more you need to know. You should consult the additional resources at the end of this guide.

Prerequisites and assumptions—where are you coming from?

From the perspective of its richest and most powerful capabilities, WebObjects is a programming environment. *The chief assumption this guide makes about your background is that your are a programmer.* From a multitude of languages, you have used at least one, ideally two or more. You should be familiar with the following terms and the concepts they convey: data type, variable, operator, expression, statement, conditional, loop, procedure or function, argument or parameter, and return value. This much is required.

You don't necessarily have to know much about object-oriented programming, but it certainly helps if you have been exposed to the vocabulary and the concepts. For many, the hard part of Java programming is learning how to think like an object-oriented programmer. Terms are essential, but it is the ideas that the terms convey—the way of thinking—that is both simple and elusive. This guide does not present object-oriented programming in rich detail. It takes the opposite view—learn by example and gain your own understanding simply by using it.

There are a number of good resources to strengthen your skills in both programming and object-oriented thinking listed at the end of the guide.

You're a Java hacker—do you even need this guide?

The best way to determine if you are already Java-savvy enough to work with WebObjects is to see if you understand some Java code typically found in a WebObjects application. The guide includes a small self-evaluation. It is realistic, and intentionally uses just about everything presented in this guide. The ultimate goal of this guide is to enable you to understand that particular bit of code. Go over the code carefully—very carefully. Read the guide, then go over the code again. If you are already Java-savvy, consider it a sanity check, a refresher, a bit of stretching before coding. For you, this may truly be Java in 21 minutes.

Java in two one-hour chapters

Java programming focuses on useful objects and they way they are classified. An effective Java programmer must cultivate two different perspectives about objects: using the object from the outside and implementing the object from the inside. More properly put, you must think like a consumer of objects on one hand, and like a producer of the classes that define them, on the other. This division is the very spirit of encapsulation, one of the chief concepts in object-oriented programming.

This guide has two core chapters to support these two perspectives and they move from outside to inside:

- Using Objects—Thinking like a class consumer
- Creating Classes—Thinking like a class producer

The problem is that to write any code in Java, you must create a new class. It is a bit of a chicken and the egg problem. As such, you cannot do anything real in Java until you have covered the second chapter. Although the first chapter presents real and useful Java code examples, they are incomplete outside of a class definition. While reading the first chapter, you might wonder where and how these code samples are used. Who calls them? Where do I place them in order to compile and run? The answers will become clear by the time you are finished with the guide. Some answers may not become clear until you begin building a WebObjects application. In the meantime, relax and absorb what's at hand.

An optional third chapter is included to help you understand how your Java code fits in with the rest of the WebObjects infrastructure to form a complete application. Here is where you learn a bit about compilers, class files, and the big bang that launches the application and activates your code. But these are incidental details; they are not part of the core spirit of thinking—and speaking—like a Java programmer. The fourth chapter provides an overview of Java's exception-handling architecture, which allow you to deal with error conditions within your application. Exceptions are used pervasively in WebObjects, and in Java in general, and a basic understanding of their role is essential for effective Java development.

What's not covered and why

There are a number of Java features typically covered in Java books that are not covered here. Some aspects of the Java environment are not used in HTML-based WebObjects applications. A good example is Swing, a package for building graphical interfaces. Although you may eventually use Java applets, you don't have to. Java programming in WebObjects is fundamentally server-side Java. Learning the core Java language is different from learning any number of packages that you can use with Java. As a WebObjects developer, your job is, first, to learn the Java language. Next, you need to learn the packages that are specific to WebObjects. You may not necessarily ever have to learn any of the "standard" Java packages, at least for developing WebObjects applications.

There are also aspects of the Java language itself that are not included in the guide—arrays, bitwise operators, and initialization blocks, among others. For topics that are included, the guide does not say everything. The goal of Java For WebObjects Developers is to present the most practical and commonly-used features of the language without excessive detail, nuance or caveat. The more advanced your code becomes, the more likely you will need some of these additional features. Eventually, you will need a comprehensive reference. Again, see the suggestions at the end.

What about different versions of Java? Changes were made to the Java environment as it matured from version 1 into version 2. This guide, however, is mostly about the core Java language itself, which has remained fairly consistent over time. One area of relevance where this is less true, though, is Java's support for collections of objects (arrays, sets, and so on). Depending on which resources you read, you may see references to the new collection features, the old ones, or both. WebObjects' support for collections is currently loosely associated with the legacy features, however this guide notes both new and old.

A bit about WebObjects

WebObjects is an award-winning cross-platform Web-based application server. With frameworks that define a coherent, rich, and mature object model, WebObjects gives Java developers a first class object-oriented environment. With a complete runtime support infrastructure, WebObjects provides everything for packaging and serving components that focus on your application-specific logic.

WebObjects is a complete development and deployment environment. The integrated graphical tools encompass the full open-ended life cycle of production Web applications—prototyping, development, documentation, debugging, performance analysis and stress testing, deployment, monitoring, reusing, and evolving.

The WebObjects framework handles Web-based transactions. It features a flexible component-based design for dynamic HTML generation, request processing and navigation. The framework defines application and session abstractions for state management, and a multi-threaded service infrastructure for robust and scalable designs.

Enterprise Objects Framework, a second framework bundled with WebObjects, defines a sophisticated model for integrating persistent data stores such as relational databases. It implements session-based change tracking, object faulting, caching, and dynamic SQL generation. It is ultimately driven by your enterprise-specific model definition which is language and schema independent.

Both frameworks use an adaptor pattern to transparently run on multiple servers—HTTP and database—without compromising the object model or the portability of your implementation. You can deploy to virtually any J2EE-capable server, or use the included WebObjects J2SE application server. Furthermore, WebObjects provides support for J2EE technologies, including Servlet integration, an Object Request Broker (ORB), and an Enterprise Java Beans (EJB) container, allowing you to mix and match technologies.

The architecture maintains a multi-tier modularity that cleanly separates the user interface, the business logic, the persistent object store, and the application server infrastructure. WebObjects supports—and even enforces—the modular focus of enterprise developers. In addition to letting you develop HTML-based applications, WebObjects also allows you to create web services and three-tier Java server applications; its modularity therefore greatly increases opportunities for code reuse.

WebObjects, now in its fifth iteration, has been on the market for eight years. Its core technology derives from over ten years of iterative development and deployment experience. There are now thousands of commercial Web sites from an impressive list of enterprise customers, all powered by WebObjects. WebObjects continues to define the highest standard for inspired developers and intelligent online success stories.

For more information, visit http://www.apple.com/webobjects/.

Evaluating Your Java Skill

Do you already speak Java?
Can you think in Java?

Goal

Test your current Java skills.

Prerequisites

None.

Objectives

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

- I have great Java skills—I'm ready for WebObjects—or—
- I had better read through this guide then try the evaluation again—or—
- I have great Java skills but I'm going to read through this guide anyway

Evaluating your Java skill

The Java code example on the following page defines an interface and a class that implements it. In the spirit of WebObjects and a typical e-commerce application, it defines a simple shopping cart class and a related interface. The design is simple—perhaps not entirely real world—but it incorporates just about all the important concepts and code constructs you need to program a WebObjects application in Java.

Although the code is brief, it is powerful. Read it carefully. Be sure you understand every byte of it. It compiles and runs fine. Comments are omitted on purpose.

The remainder of the guide explains everything necessary to understand it.

The shopping cart specification

A shopping cart is a collection of items associated with a customer. An item is something you can purchase. As a collection of items, a shopping cart represents an aggregate purchase. Since they both represent a type of purchase, both the shopping cart and its items have similar behavior. You can ask a purchase for its subtotal. You can ask a purchase for its total—the subtotal plus tax. All purchases are subject to the same tax rate, at least in this simple model.

A few special details

The code uses one class specific to WebObjects—NSMutableArray. It is a collection class much like Java's Vector or ArrayList class, or an array construct featured in most programming languages. With that it mind, it should be obvious how—and why—it is used.

The code features a customer object but does not show the Customer class. As an object-oriented developer, it's good for you to be comfortable with black boxes.

The shopping cart example

Purchase.java

```
public interface Purchase {
    public final static double TaxRate = 0.085;
    public double subtotal();
    public double total();
}
```

ShoppingCart.java

```
import java.util.Enumeration;
import com.webobjects.foundation.*;

public class ShoppingCart extends Object implements Purchase
{
    protected NSMutableArray items;
    protected Customer shopper;

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

    public ShoppingCart(Customer customer) {
        this();
        setShopper(customer);
    }
    public Customer getShopper() { return shopper; }

    public void setShopper(Customer newShopper) {
        shopper = newShopper;
    }

    public void addItem(Purchase item) {
        items.addObject(item);
    }

    public int getItemCount() { return items.count(); }

    public double subtotal() {

        double subtotal = 0;
        Enumeration e = items.objectEnumerator();
        while(e.hasMoreElements())
            subtotal += ((Purchase)
                e.nextElement()).subtotal();
        return subtotal;
    }

    public double total() {
        double subtotal = this.subtotal();
        return subtotal + subtotal * Purchase.TaxRate;
    }

    public String toString() {
        return shopper + " " + items;
    }
}
```

Evaluate your Java skills in your own words

Write a Java statement to create a new shopping cart.

Write the code to associate a shopping cart with a customer.

Write the code to get the shopping cart's current total.

ShoppingCart is missing a method to remove an item. Write it.

Write the code for the Item class. It has a name and it implements the Purchase interface.

Draw a graphical representation of a shopping cart with a customer and two items.

2 Using Objects

Thinking like a Class Consumer
A View from the Outside

Goal

To cover the basic concepts and Java code syntax for using objects.

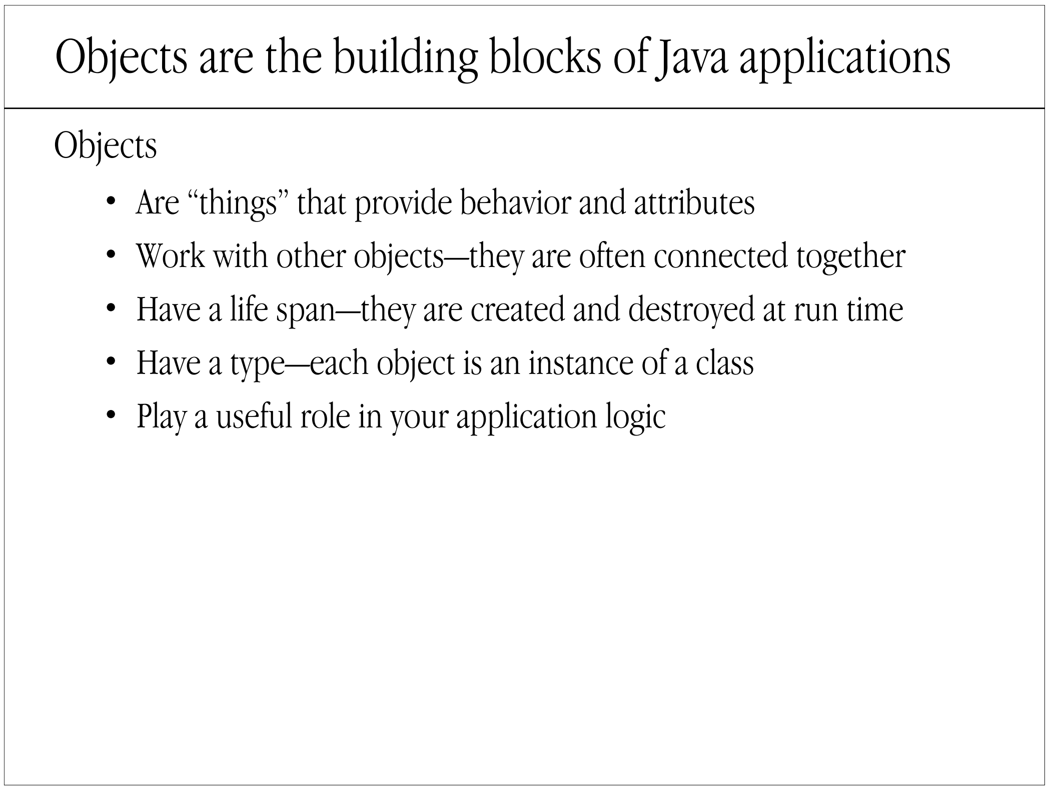
Prerequisites

Basic programming skills.

Objectives

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

- Create objects
- Send messages to objects
- Access object attributes
- Collect objects in arrays and dictionaries
- Use casts and determine the class type of an object
- Explain Java naming conventions
- List common coding pitfalls



Objects are the building blocks of Java applications

Objects

- Are "things" that provide behavior and attributes
- Work with other objects—they are often connected together
- Have a life span—they are created and destroyed at run time
- Have a type—each object is an instance of a class
- Play a useful role in your application logic

Objects are the building blocks of Java applications

Java is an object-oriented language. This means that Java programmers think of problems, designs, and code in terms of objects. An object is something that plays a meaningful role in your application. It can be as simple as a number or as complex as a shopping mall.

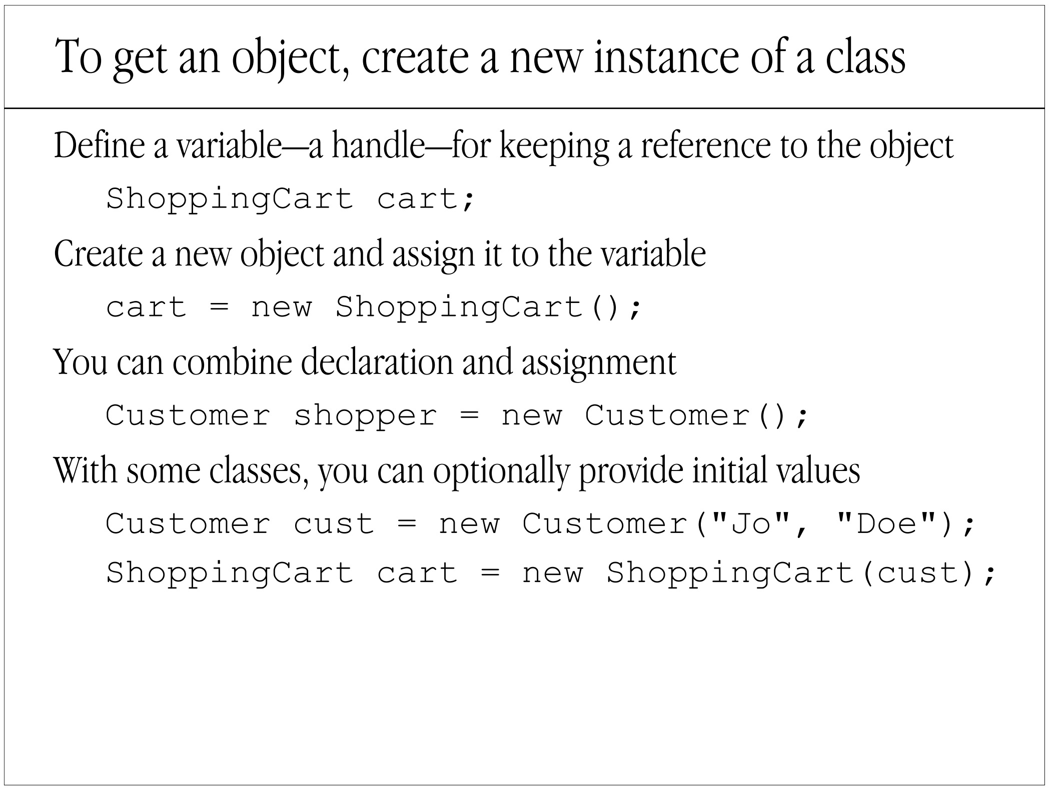
An object provides useful behavior—it can do things. A shopping cart object might collect items, report which items it currently has, and compute the outstanding balance.

An object has attributes—it has properties or characteristics. A customer object might have a name, a credit card number, maybe some coupons.

An object has a life span. You have to create a new shopping cart object when you need it. When you are finished using it, you can throw it away.

Although it is obvious, it is important to understand that each object is a specific type of object. An object is clearly a shopping cart, or a customer, or a number, or whatever. By naming its type, you are classifying the object—relating it to other objects that share the same characteristics and behavior. Every object belongs to a specific class of objects.

Objects are connected together to create systems of objects. A customer has a shopping cart that contains several items. In many ways, object-oriented programming is about creating relevant objects and connecting them into meaningful patterns that model a real-world situation.



To get an object, create a new instance of a class

Define a variable—a handle—for keeping a reference to the object

```
ShoppingCart cart;
```

Create a new object and assign it to the variable

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

You can combine declaration and assignment

```
Customer shopper = new Customer();
```

With some classes, you can optionally provide initial values

```
Customer cust = new Customer("Jo", "Doe");
ShoppingCart cart = new ShoppingCart(cust);
```

To get an object, create a new instance of a class

Before you make an object do something useful, you have to get a hold of it. Unless it already exists, you have to create the object. As long as you need to use it, you need to hang on to it, keeping track of it for future use. You keep track of an object by creating a *variable* to reference the object. The variable creates a symbolic name by which you can refer to an object. It is usually called a *reference*. Sometimes this is called a handle or a pointer.

You can assign an object to the reference variable. The Java assignment operator is the equal sign.

An object is a real instance of a specific class of objects. Since an object is often called an instance, the act of creating a new object is often called *instantiation*.

You can combine the reference variable definition, the object creation, and assignment to the variable in one Java statement. All Java statements end with a semicolon.

For some classes, you can provide initial data arguments when constructing a new object.

The reference and the object are two different things

Every variable has a type, the class of object it references

```
ShoppingCart cart;
```

The class name is pre-defined—the variable name is up to you

The variable is not the object but a reference to a potential object

Until you assign an object, the reference is `null`

To create the object, use the `new` keyword and the class name

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

The class name is used like a function and is called a *constructor*

The reference and the object are two different things

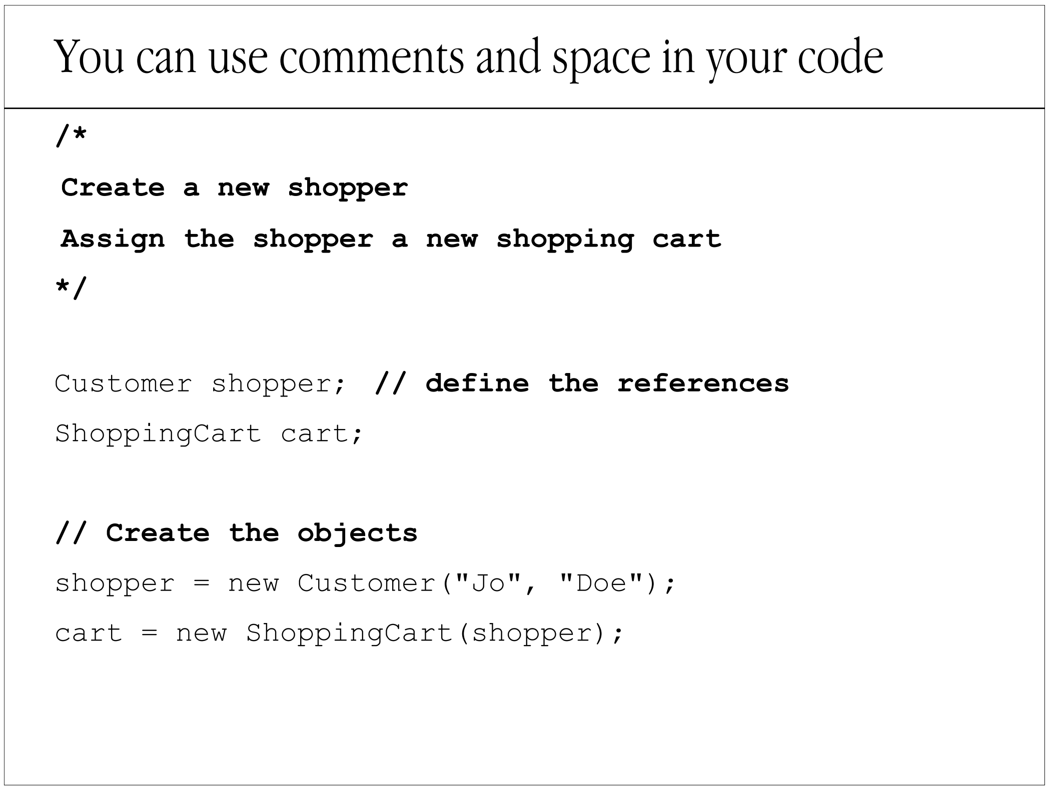
A variable declaration has two parts: the type and the name. The type is the class of object the variable will reference. Class names are pre-defined. The variable name is up to you.

How do you know which classes are available? Java defines many standard classes, and development environments like WebObjects add many more. Learning about available classes is a large part of your learning curve. Read your chosen reference material, or consult online documentation—for example Sun Microsystems' training materials at http://developer.java.sun.com/developer/onlineTraining/ or the documentation installed as part of the Mac OS X developer tools at file:///Developer/Documentation/Java/java.html. You will learn about several useful Java classes in this guide.

Although you have defined a variable, you don't yet have an object. The variable is a reference to an object, it is not an object itself. Some languages call the reference variable a *pointer*. It is not an object, but something that points to an object. The variable is the name of the object you use when referring to the object in your code. But you still need to create or retrieve the object itself and assign it to its name.

Until you assign an actual object, a reference points to nothing. There is no object. In Java, you say that the reference is `null`. `null` means the non-object.

To create the object, use the `new` keyword followed by the name of class and a pair of parentheses. The syntax looks much like a function call. In Java, this is called invoking a *constructor* because it is a function that constructs a new object, an instance of the specified class.



You can use comments and space in your code

```
/*
 Create a new shopper
 Assign the shopper a new shopping cart
*/

Customer shopper; // define the references
ShoppingCart cart;

// Create the objects
shopper = new Customer("Jo", "Doe");
cart = new ShoppingCart(shopper);
```

You can use comments and space in your code

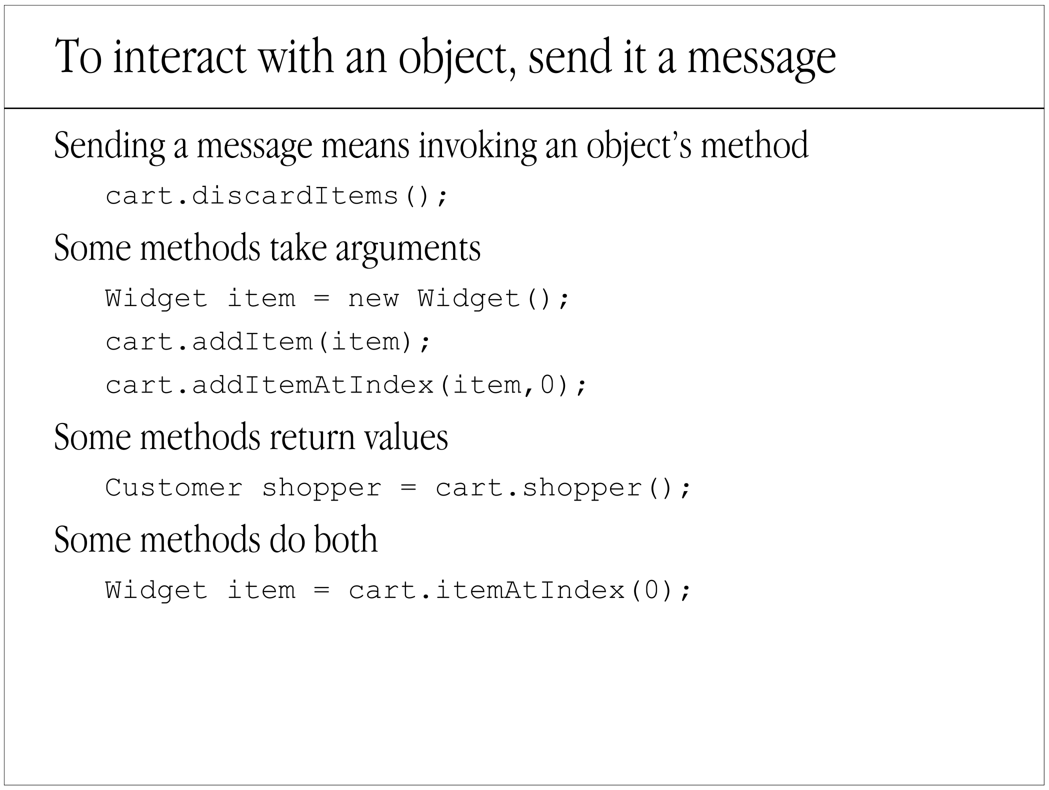
Java is a free-form language. You can format your code as you wish. You can include blank lines, indentation, break a statement into multiple lines and so on. You may quickly notice common formatting conventions that you should adopt. Conventions allow you to easily understand someone else's code as well as ensuring that they will understand yours. One of the best ways to learn Java is to read and imitate real Java code.

You can include comments in your code. Unlike the strict syntax required for the Java compiler, comments are free-form and intended for human readers. Java provides two different comment styles:

Arbitrary multi-line comments: begin with `/*` and end with `*/`

Single line comments: begin with `//`. The remainder of the line is considered a comment.

Javadoc is a tool from Sun Microsystems for extracting comments from source code to generate API documentation in HTML format. To use it, you have to follow certain conventions and guidelines for writing "doc" comments. For more information about Javadoc, see http://java.sun.com/j2se/javadoc/.



To interact with an object, send it a message

Sending a message means invoking an object's method

```
cart.discardItems();
```

Some methods take arguments

```
Widget item = new Widget();
cart.addItem(item);
cart.addItemAtIndex(item,0);
```

Some methods return values

```
Customer shopper = cart.shopper();
```

Some methods do both

```
Widget item = cart.itemAtIndex(0);
```

To interact with an object, send it a message

Once you have a reference to an object, you can interact with it. Objects serve useful roles and your interest is getting them to act. To make an object do something, send it a message.

Sending a message is just as simple as it sounds: tell the object what you want it to do. In Java, you send a message with a statement that combines the object name—the reference variable—and the name of the message. They are connected by a period:

```
object.message()
```

This looks much like a function or procedure invocation in traditional languages. In Java, this is called a *method invocation*. You send a message, and the object has a method for servicing it. Sending a message is often simply referred to as *invoking a method*. The difference between message and method reflects the fundamental difference between you and the object. You make a request—the message—and the object has a response—the method.

Some methods take arguments, some return a value, some do both. Many methods neither take arguments nor return values.

What messages can you send to an object? The answer is specific to each class. Just as important as knowing the classes you can use is knowing what those classes do. This is the largest part of your Java learning curve. To understand an object's capabilities, consult the documentation for its class, if it is available. Documentation for all the core Java classes can be found at http://java.sun.com/api/index.html.

Access object attributes using methods

Most attributes are private—you must use an accessor method

```
Customer shopper = cart.shopper();   // get
cart.setShopper(shopper);            // set
```

There are alternate naming conventions for get accessor methods

```
Customer shopper = cart.shopper();
Customer shopper = cart.getShopper();
```

In some cases attributes are public—you can access them directly

```
Customer shopper = cart.shopper;
cart.shopper = shopper;
```

Access object attributes using methods

Central to the role an object plays is its state—its attributes or properties. A customer has a name, a product has a part number, a shopping cart is related to a customer. In Java, these are often called *fields*. In general object-oriented terminology, they are called *instance variables* because each instance of the class has its own set. Objects usually provide methods to get at the attributes they want to expose publicly. To get an attribute from an object, send a message.

Methods that access object attributes are common and basic. They have a special name—*accessor* methods. Usually, they come in pairs: one method to get a value, another to set the value. They are often called the *get accessor method* and the *set accessor method* respectively. Sometimes, the get method is called the *accessor* or *getter* method while the set method is called the *mutator* or *setter* method.

Different classes use different naming conventions for the get accessor method. Some methods start with the word "get", others simply use the attribute name itself, without the word "get". To use an object properly, consult the class documentation to determine which naming convention it adopts.

In a few cases, attributes are directly accessible without sending a message. Java syntax for accessing public attributes is much like accessing fields of a record or a structure in traditional languages. Notice that you do not use parentheses as you would when invoking a method.

You can nest code to avoid creating variables

Create a new object for a method argument

```
cart.addItem(new Widget());
cart.setShopper(new Customer("Jo", "Doe"));
```

Invoke a method in the return value of another method

```
String name = cart.shopper().lastName();
```

Nest method invocations

```
cart1.addItem(cart2.itemAtIndex(4));
```

You can nest code to avoid creating variables

Often, you need to create an object merely for the purpose of giving it to another object. You don't need your own reference variable in the meantime. Java syntax allows you to nest the code for creating an object within the list of arguments you are sending to another object's method. Create it, pass it, and forget about it—all in one statement.

In a similar spirit, you often need to send a message to get an object then immediately send a message to that object to get what you're really after. For example, assume you need the name of the customer that owns the shopping cart. You could create a temporary variable to hold the intermediate object:

```
Customer customer = cart.shopper();
String name = customer.lastName();
```

But in this case, you are interested in the name, not the customer. Java syntax allows you to connect multiple messages into a single expression where each subsequent message is sent to the return value of the previous:

```
String name = cart.shopper().lastName();
```

To generalize, wherever you need to supply an object reference, you can supply any expression that returns an object reference.

Character strings are objects

Character strings are instances of the class *String*

```
String name = shopper.lastName();
```

You can use literal strings

```
String banner = "All widgets on sale";
shopper.setFirstName("John");
```

You can concatenate strings with the "+" operator

```
String fullName = "John " + "Doe";
Customer s = cart.shopper();
fullName = s.firstName() + " " + s.lastName();
```

Strings are immutable—once created, you cannot change their value

Character strings are objects

Java represents character strings as objects. They are instances of the class named String. Strings are simple values and are used to represent basic object attributes. Because they are so common, convenient handling of strings is built into the Java language itself.

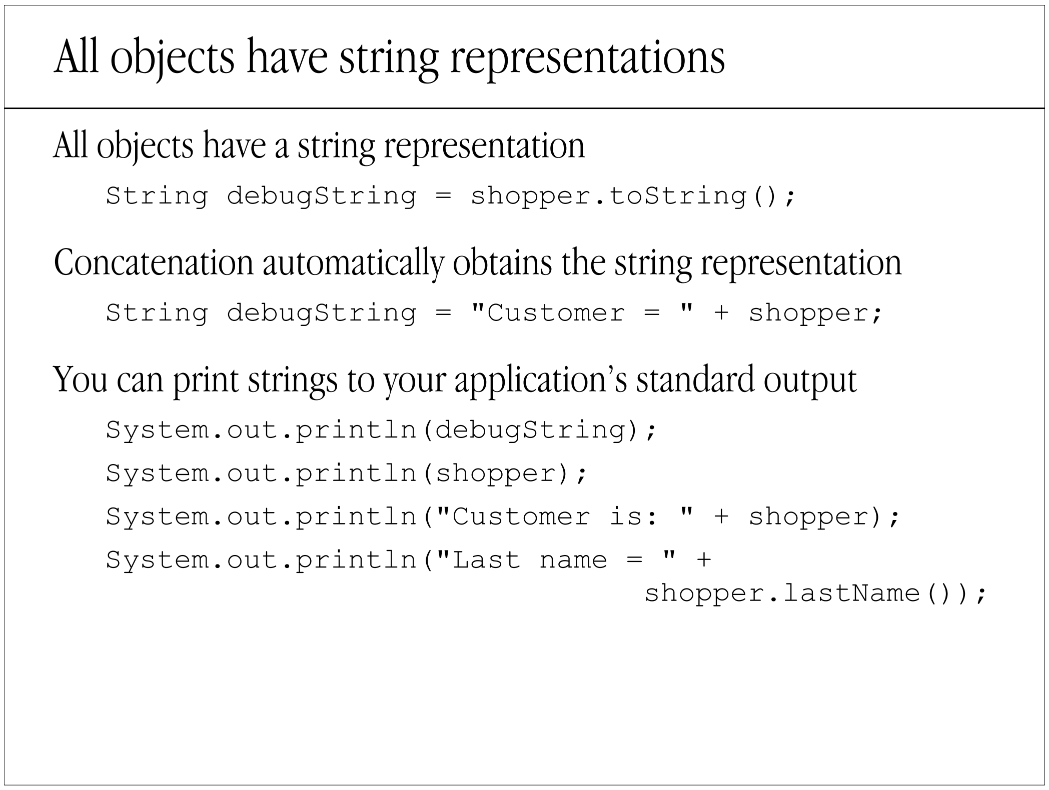
You can use a literal string value in quotes as an alternative to constructing a string object literally. Notice that the following two lines are equivalent:

```
message = "hello";
message = new String("hello");
```

You can combine strings—concatenate them—using the plus operator. This creates a new string instance that combines the values of operands. The String class provides additional methods for manipulating strings—consult the Java documentation to learn more about them.

Strings are immutable: once they are created, you cannot change their value. If you wish to modify a string without creating a separate result string, you can use the StringBuffer class.

The fact that many objects represent attributes as strings reveals something fundamental about objects: objects are typically composed of other objects. A customer is an object. A customer has a name—a string—which is itself another object.



All objects have string representations

All objects have a string representation

```
String debugString = shopper.toString();
```

Concatenation automatically obtains the string representation

```
String debugString = "Customer = " + shopper;
```

You can print strings to your application’s standard output

```
System.out.println(debugString);
System.out.println(shopper);
System.out.println("Customer is: " + shopper);
System.out.println("Last name = " +
                               shopper.lastName());
```

All objects have string representations

Regardless of its class, any object can generate a string representation of itself. This is useful for debugging and for displaying a value in a user interface such as a Web page. Every object in Java responds to the message `toString()`.

You can print a string to the standard output unit of your application using the `println()` message. Consider the following code:

```
System.out.println("hello");
```

This statement is saying, “send the `println()` message to the `out` object which is available as a public attribute of the `System` object.” What happens when you pass an argument that is not a string? Consider the following:

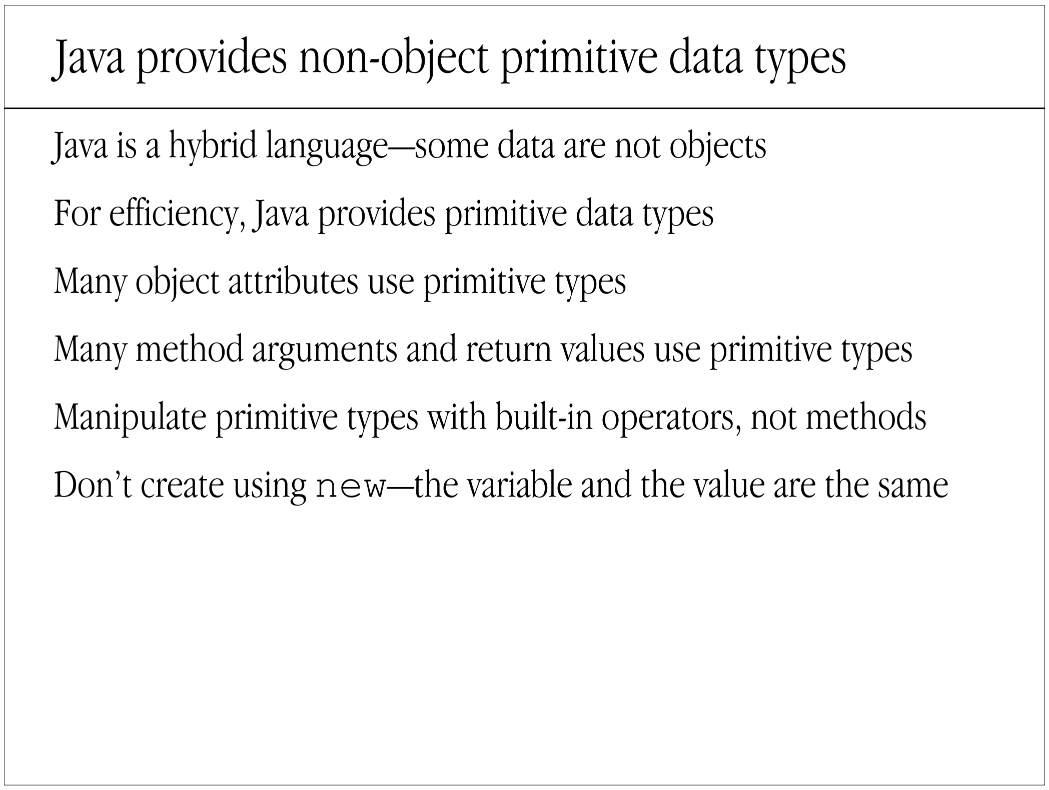
```
System.out.println(shoppingCart);
```

The `println()` method automatically accesses the string representation of the shopping cart object by sending it the `toString()` message. The object generates a string suitable for printing. This also takes place when you use the plus operator to concatenate strings.

```
message = "Customer = " + customer;
```

The customer object is not a String object but the statement automatically obtains a string representation by sending `toString()` to the customer, equivalent to:

```
message = "Customer = " + customer.toString();
```



Java provides non-object primitive data types

Java is a hybrid language—some data are not objects

For efficiency, Java provides primitive data types

Many object attributes use primitive types

Many method arguments and return values use primitive types

Manipulate primitive types with built-in operators, not methods

Don't create using `new`—the variable and the value are the same

Java provides non-object primitive data types

Java is a *hybrid* language—not everything in Java is an object. For efficiency and convenience, Java provides primitive types for simple things like numbers, characters, and boolean values.

Even when working with objects, you will need to handle primitive types. They are used to represent many object attributes—the number of items in a shopping cart, for example. They are also used for many method arguments and return values.

Primitive types are much like basic data types in traditional non-object-oriented languages. You handle primitive types differently than objects in two fundamental ways:

- Manipulate primitive types with operators not methods.
- Don't instantiate primitive types—there is no difference between a value and a reference to the value, they are the same.

Useful subset of primitive data types

Type	Contains	Examples
`byte`	8-bit signed value	Any arbitrary bit pattern
`char`	16-bit unicode character	`'a'`,`'0'`, `\u00F1`
`int`	32-bit signed integer	`10`, `-5`
`double`	64-bit IEEE floating point	`10.5`, `-5.2`
`boolean`	1-bit true or false value	`true`, `false`

Useful subset of primitive data types

Java defines several primitive types. Here is a useful subset: `byte`, `char`, `int`, `double`, and `boolean`.

`char` represents 16-bit Unicode characters, not the traditional 8-bit ASCII characters used in languages like C and C++.

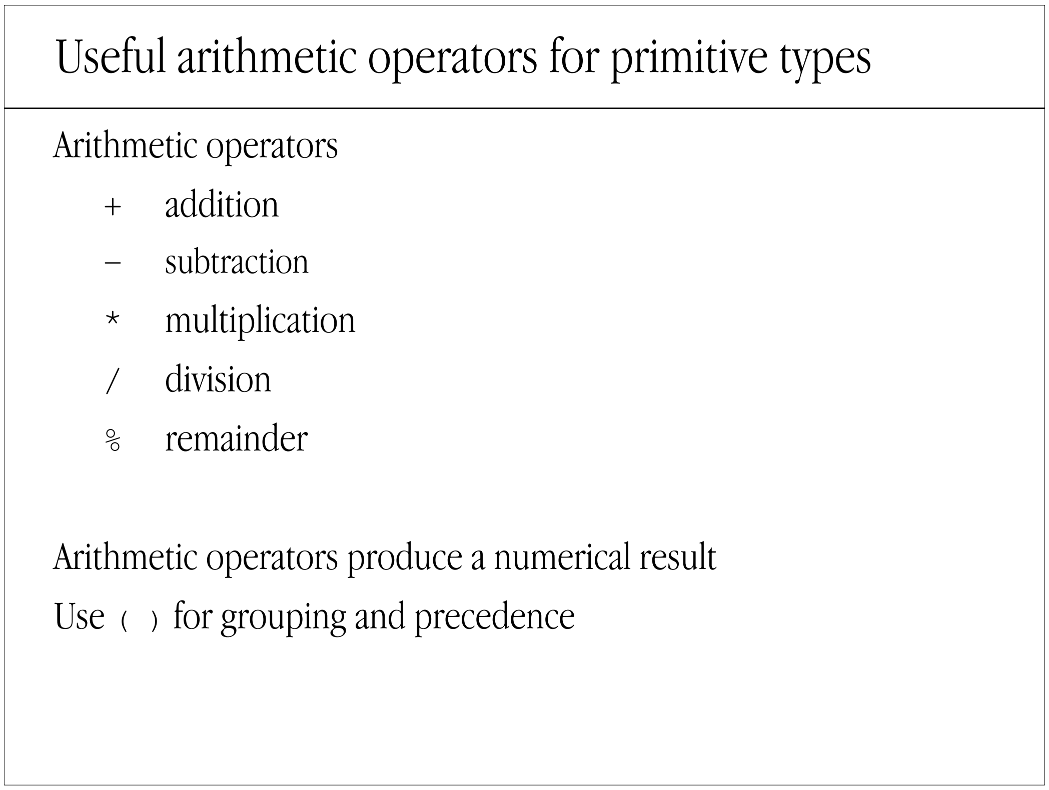
`int` is always 32 bits regardless of the underlying hardware platform. This fixes a number of portability issues inherent in C and C++ due to different word sizes on different machines.

Additional types not shown above offer different possibilities for number values: `short`, `long` and `float`. They differ in size and magnitude, and reflect the C and C++ origins of Java.

Java does not provide any unsigned types.

`boolean` values use 1 bit and have only two possible values—`true` and `false`. These are Java keywords. Unlike C and C++, Java does not allow numbers or references to be used directly as boolean values. For example, `0` is always the number `0`, not the boolean value `false`.

The table above shows that literal values are allowed for all primitive types. Wherever you need to supply a primitive type, you can supply a variable, a literal, or an expression that results in a primitive type value.



Useful arithmetic operators for primitive types

Arithmetic operators

`+`	addition
`–`	subtraction
`*`	multiplication
`/`	division
`%`	remainder

Arithmetic operators produce a numerical result

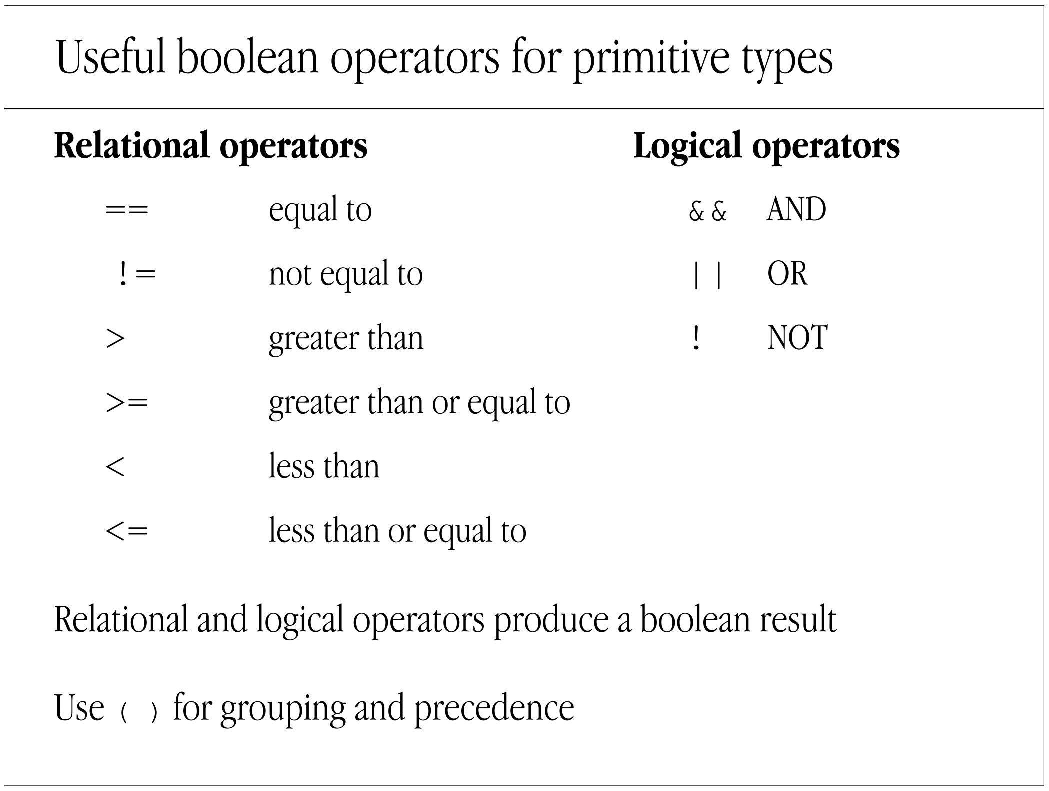
Use `( )` for grouping and precedence

Useful arithmetic operators for primitive types

When working with primitive number types, you use *operators* not messages. Java provides the standard arithmetic operators: `+`, `-`, `*`, `/`, and `%`. Java provides several additional operators not shown above such as `++` for increment or `+=` for a combination of addition and assignment. Java also provides bitwise operators.

Arithmetic operators expect primitive number operands and produce primitive number results—likewise for any arithmetic expression of arbitrary complexity. Do not confuse primitive number values with either boolean or object values. There is one exception: the + operator is also valid for concatenating string objects. The result of concatenation is a string object. This is the only case in Java where an operator is overloaded to support object rather than primitive types. This is built into the language. Java does not support operator overloading for custom classes.

You can use parentheses for grouping and readability. Because of the precedence rules in Java, you many need to use parentheses to enforce the meaning of an expression when the default precedence produces unexpected results.



Useful boolean operators for primitive types

Relational operators

`==`	equal to
`!=`	not equal to
`>`	greater than
`>=`	greater than or equal to
`<`	less than
`<=`	less than or equal to

Logical operators

`&&`	AND
`||`	OR
`!`	NOT

Relational and logical operators produce a boolean result

Use `( )` for grouping and precedence

Useful boolean operators for primitive types

You can perform a variety of boolean tests using the relational operators: ==, `!=`, >, <, >=, and <=. The relational operators are valid only for primitive types with the exception of == and `!=` which you can also use for comparing object references (more on this later—see "You can perform basic object tests"). The result of an expression using relational operators is a boolean value—`true` or `false`.

You can join multiple boolean expressions using the logical operators: `&&`, `||`, and `!`. The logical operators work only with boolean operands. The result of any boolean expression is a boolean type value.

Code examples using primitive types

Variable definitions

```
int count;
double price = 10.75, discount = 0.15;
double total = price * count * (1 - discount);
boolean orderConfirmed = false;
```

Statements

```
count = count + 1;
orderConfirmed = true;
total = cart.total() * (1 - discount);
shopper.setCreditLimit(500.00);
shopper.setCreditLimit(500.00 * 0.75);
shopper.setCreditLimit(cart.total());
```

Code examples using primitive types

You can define variables of primitive types including an initial value. The initial value can be the result of an expression using literals, other primitive variables, and even messages to objects that return primitive values. You can define multiple variables of the same type in one statement. Use the comma to separate the names. If you attempt to use a variable that has not been initialized or assigned, the Java compiler will generate an error.

Java's syntax permits great flexibility in building expressions using a mixture of literals, variables, messages to objects, and nested expressions. The key is to make sure the resulting type of each component in the expression matches the overall type, in this case, a primitive non-object type.

You can make decisions for conditional logic

Simple conditional statement with `if` and a `boolean` expression

```
if (orderConfirmed)
   cart.checkOut();
```

Either-or logic using `else`

```
if (cart.getItemCount() >= 10)
   discount = 0.25;
else
   discount = 0.10;
```

Complex boolean expression

```
if (cart.getItemCount() > 0 && !orderConfirmed)
   askForConfirmation = true;
```

You can make decisions for conditional logic

Boolean expressions enable you to make decisions. Often, your code is conditional—you only want to execute it under certain circumstances. For example: if the customer is ready, then send the shopping cart through check out.

Java provides the `if` and `if-else` statements for these occasions. `if` uses a parenthesized boolean expression to determine whether or not the subsequent code should be executed. If the expression is `true`, the code is executed. If `false`, it is not. The `else` keyword is optional and provides the alternative choice. If the expression is `true` do one thing, else, do the other.

The simplicity of the `if-else` statement is deceptive. Its flexibility permits multi-way decisions of arbitrary complexity:

```
if (subtotal > 1000)
    discount = 0.20;
else if (subtotal > 500)
    discount = 0.10;
else
    discount = 0;
```

Java also provides the `switch` statement and a conditional operator for making decisions. These are not shown here.

Multiple statements require a block

Multiple statements within an `if` or `else` clause require a *block*

A block is a group of statements delimited by braces `{ }`

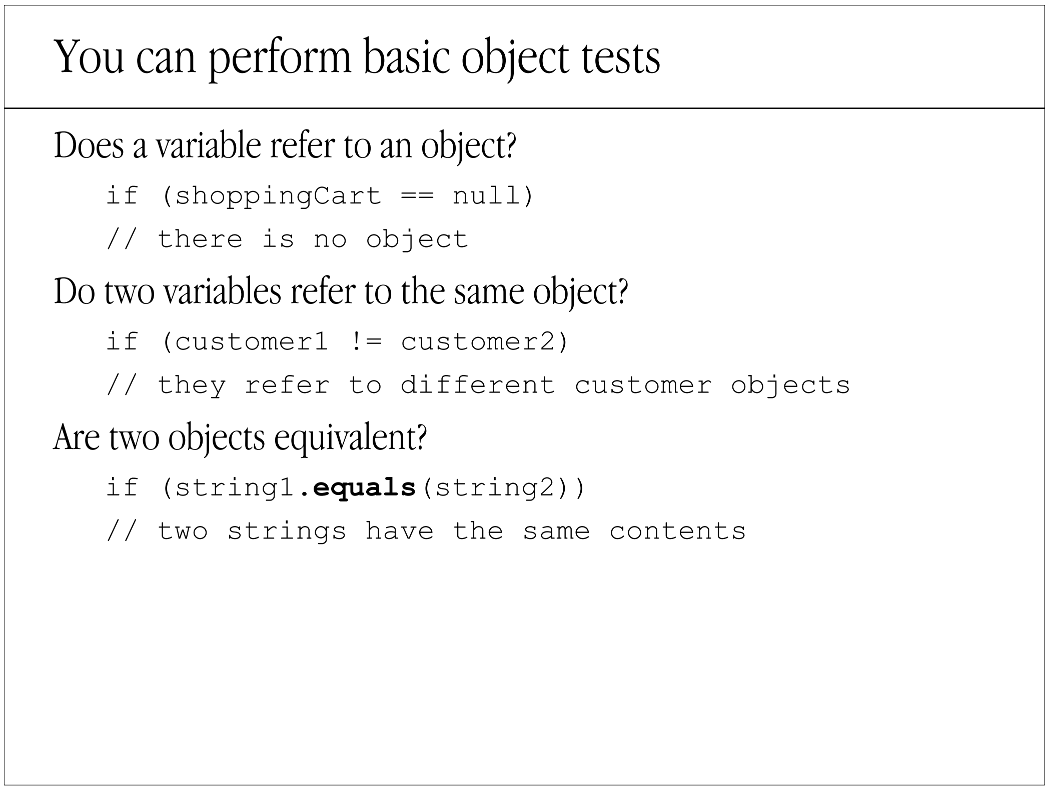
You can create temporary local variables inside a block

```
double total = cart.total();
if (cart.itemCount() > 0) {
 double discount = 0; // temporary variable
   if (cart.getItemCount() >= 10)
      discount = 0.25;
   else
      discount = 0.10;
   total = total * discount;
}
```

Multiple statements require a block

Frequently, you need to do several things based on a certain decision. To group multiple statements within an `if` or `else` clause, you must enclose them in a *block*. A block is a group of statements enclosed in braces. Blocks are used in several different places in the Java language. A block is sometimes called a *scope*.

Within a block, you can define new variables. These are *local variables*, visible only within the block that defines them. They are *temporary variables* in that they come into existence only when you enter the block, and they are destroyed when you leave the block.



You can perform basic object tests

Does a variable refer to an object?

```
if (shoppingCart == null)
// there is no object
```

Do two variables refer to the same object?

```
if (customer1 != customer2)
// they refer to different customer objects
```

Are two objects equivalent?

```
if (string1.equals(string2))
// two strings have the same contents
```

You can perform basic object tests

You often make decisions based on simple object tests. The most fundamental test is whether or not a variable refers to an object. Remember that the reference variable is one thing, the object it refers to is another. Java provides the `null` keyword which means "no object". You can use `null` to determine if a variable actually refers to an object. It is illegal to send a message to a variable whose value is `null`. This makes sense: the variable does not reference a valid object. Notice that you use an operator to make this test, not a message.

In a similar vein, you may have two variables of the same type and wish to know if they refer to the same object. This is called an *identity test* because you are asking if two objects are identical—the same object under different names. Use the comparison operators == or `!=` to test the reference values.

A third test asks not whether two objects are the same, but whether they are equivalent. They might be different instances of the same class, but have the same contents. Imagine two physical copies of the same credit card. They represent the same account and should be treated equally with respect to making a charge. This is an *equivalence test* rather than an identity test. It requires that you ask the objects themselves using a message rather than an operator. All objects respond to the `equals()` message though how they test for equivalence is specific to each class.

It is a common mistake to confuse identity tests with equivalence tests.

Classes often provide attributes and special objects

Classes often provide attributes, independent of any specific instance

Use a *class* method rather than an instance method

```
int count = ShoppingCart.activeCartCount();
```

Often, class attributes are public—access them directly

```
String version = Customer.Version;
```

To access some objects, use a class method rather than a constructor

```
Store mainStore = Store.headquarters();
```

Note—*Class* methods and attributes are called *static* in Java

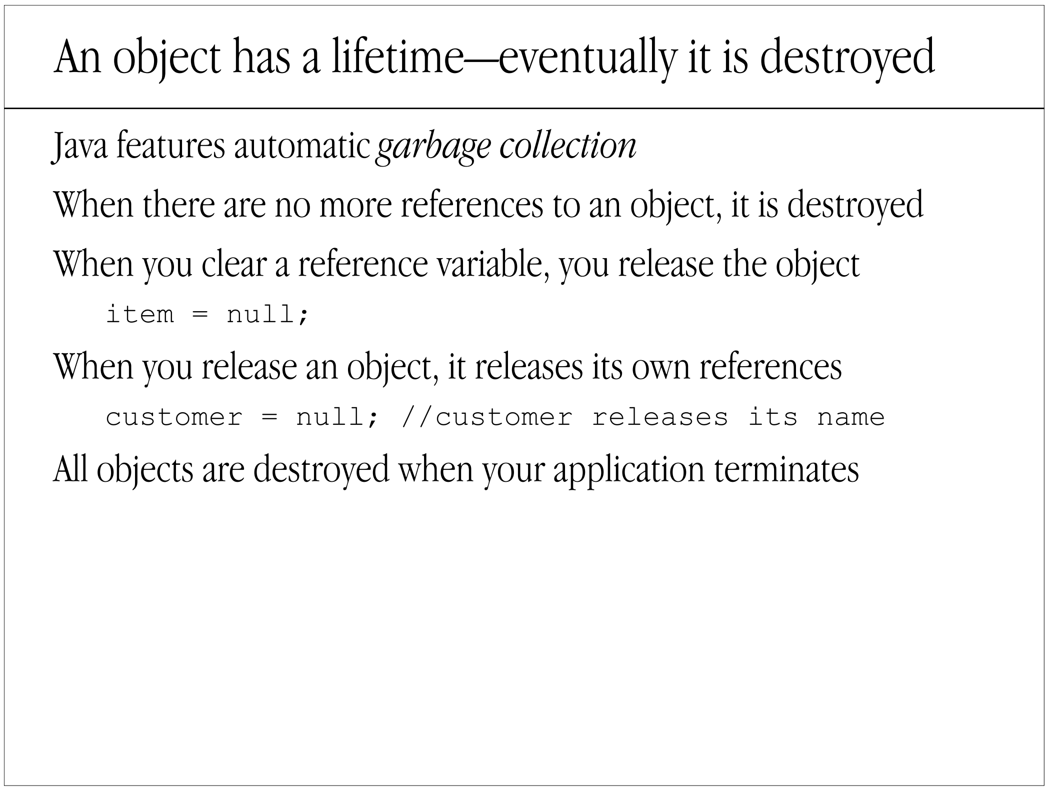
Classes often provide constants and special objects

Imagine that you want to know how many active shopping carts there are. How many customers are currently shopping? It doesn't make sense that any particular shopping cart instance would know this information. You may not even have a reference to any particular shopping cart object. You need to ask the manager of all shopping carts—the factory that creates them. And you do know the name of the shopping cart class.

Many classes provide methods and attributes that you can access directly from the class itself. You don't need an object instance. Java calls these *static*—static methods and static variables. Since they are available directly from the class, general object-oriented terminology calls them *class* methods and variables. Because a class is like a factory—it is used to manufacture instances—these are also called *factory methods*.

You can send a message to a class. Use the class name where you would normally use a reference to an object instance. For class attributes, there is one value for all instances, even if there are no instances at all. In Java, you can also send class methods to an object instance. You can get class attributes from the object as well.

A common use of a class method is to access an object instance by a means other than creating a new one directly with the `new` keyword. For example, instead of creating a new store, ask the Store class for a specific instance—in the example above, headquarters is accessed using the `headquarters()` static method.



An object has a lifetime—eventually it is destroyed

Java features automatic *garbage collection*

When there are no more references to an object, it is destroyed

When you clear a reference variable, you release the object

```
item = null;
```

When you release an object, it releases its own references

```
customer = null; //customer releases its name
```

All objects are destroyed when your application terminates

An object has a lifetime—eventually it is destroyed

To use an object, you must first create it. What happens when you are done with it? How do you destroy it? This is an important design question since you need to control the amount of resources your application uses. If you create new objects but never destroy them, you are wasting memory and may encounter performance problems.

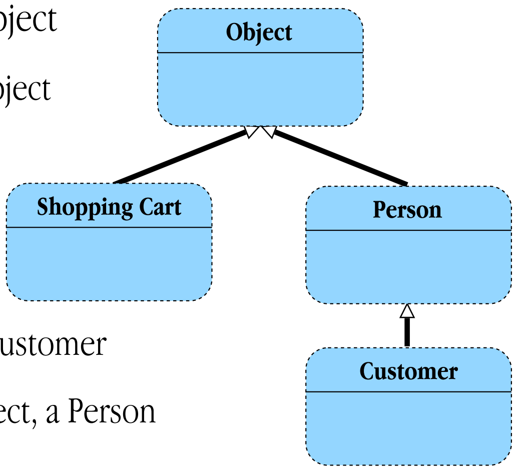
Java features automatic *garbage collection*. When an object is no longer used, it is automatically treated as garbage and eventually picked up and thrown out—destroyed. How does Java know when you are no longer using an object?

As long as you maintain a reference to an object, the object will not be considered garbage. You can forget about an object—thereby throwing it in the trash—by clearing any and all references you have to that object. The direct way to do this is by assigning `null` to your reference variable.

Many objects are composed of other objects—a customer object consists of at least a string for the first name and another for the last name. When you give up your reference to the customer, you can assume the same for the objects it references—the strings for the name attributes. Although the strings are referenced by the customer, no one has a reference to the customer itself. You can trust that it will all be considered garbage.

Class types are hierarchical

- Object is the root class
- All classes are subclasses of Object
    - ShoppingCart is a kind of Object
    - Customer is a kind of Object
- Customer is also a kind of Person
    - Person is the superclass of Customer
    - A Customer object is an Object, a Person and, specifically, a Customer
- Every object is, most generically, an instance of the Object class



Class types are hierarchical

Class types form a hierarchy. It is often called a classification hierarchy because it reflects the way we naturally classify objects in the real world: a class is defined as both similar to yet different from another class. It is also referred to as an *inheritance hierarchy*.

The root class is named Object. It is the most generic class. Other classes are defined relative to the root class—either directly or indirectly, they are subclasses of the Object class. Object is the superclass of ShoppingCart. ShoppingCart is a subclass of Object and inherits its attributes and behaviors. A shopping cart is a kind of Object but it is more, it has additional behavior and attributes.

The diagram shows a Person class and a subclass called Customer. Generally speaking, a customer is a kind of person. A customer can be treated as a person and used in any situation where a generic person can be used. But at the same time, a customer is different than a person. A customer is more specifically a customer—it is appropriate in some situations where a generic person is not.

As an object consumer, it is important to see that an object is an instance of a specific class but can be treated more generically as an instance of any of its superclasses. That you can regard an object from multiple perspectives—according to how you classify it—suggests that you can use that object in multiple contexts. How you treat an object—generically or specifically—has impact on how your write your code.

Object references can be generic or specific

This code is correct, though the variable type is not specific

```
Object customer = new Customer();
String string = customer.toString();
```

- `toString()` is a method in the Object class

This code is not correct—the variable type is not specific enough

```
Object customer = new Customer();
String string = customer.firstName(); //error!
```

- `firstName()` is a method in the Customer class, not the Object class

Generic references are useful when specific details are unimportant

- A mailing list references all Persons, not just Customers
- A shopping cart references objects of many different classes

Object references can be generic or specific

Sometimes you can treat an object generically. A customer object, like any object, can produce a string representation of itself. Any subclass of Object responds to the `toString()` message. It is not important to know that the customer is specifically an instance of the Customer class. You can refer to it as a generic object. You can declare a reference variable using a generic type—like Object—and use it with any subclass of that type—like Customer. It is valid to assign an object of a specific class to a reference variable of a more general class type.

The class type you declare for a reference variable is a promise you make regarding how you will use the object. You promise to send only those messages defined by the generic class, even though the specific object may have additional capabilities defined in its subclass. With a reference of type Object, you can only send messages defined in the Object class. Even though you assign a customer to the reference, you promise to treat it like any other Object. You cannot use the reference to ask for a customer's first name. Your code will not compile.

There are many cases where it is useful to treat different specific types of objects as a single more generic type. A shopping cart can hold many different types of product. But for the purpose of calculating the balance, the shopping cart need only treat each object as a generic product with a price. A company mailing list needs only the name and address of a recipient, without making finer distinctions about which are customers, employees, friends, and so on.

Casting allows you to be more specific

To refer to an object using a more specific class, use a cast

```
Object anObject = lostAndFound.firstItem();
Customer shopper = (Customer)anObject;
String name = shopper.firstName();
```

You can use a temporary cast to avoid creating a new variable

```
String name = ((Customer)anObject).firstName();
```

A shopping cart holds many classes of objects

```
Widget item = cart.firstItem();    // incorrect
Object item = cart.firstItem();    // correct
// If it is a Widget, use a cast to treat it so
Widget item = (Widget)cart.firstItem();
```

Casting allows you to be more specific

Sometimes, you need to be more specific about an object that is otherwise treated generically. This is often the case when you don't create the object but you get it from another object.

Imagine a lost-and-found object. It collects objects of various types: car keys, wallets, even customers themselves. It is not important to know the specific class of each object, merely that each is simply an object. You can query the lost-and-found using simple messages like `firstItem()` or `nextItem()`. Each message returns a generic object reference valid for any specific class of object.

Once you get the object from the lost and found, you want to make distinctions according to the specific class of object you have. If it is a customer, you can ask for its name. If it is a wallet, you need to check it for identification. To treat an object as a specific type, you must create a reference of that specific type. To assign a generic object to a more specific reference, you explicitly declare your intentions. In Java this is called a *cast*.

A cast must reflect reality: the object's class must be the same as or any subclass of the class named in the cast. If the object's actual class does not match the class named in the cast, you will get a runtime error.

You can use a cast to send a message without assigning the object in a reference variable. You can temporarily treat the object more specifically. Because of precedence rules in Java, a nested cast requires an extra set of parentheses: cast first, then send the message.

You can determine an object's class

Testing if an object belongs to a class or any of its subclasses

```
if (anObject instanceof Person)
   // anObject is a kind of Person
   // Could be, more specifically, a Customer
```

Using an object with a more specific class type

```
if (anObject instanceof Customer) {
   Customer customer = (Customer)anObject;
   String customerCode = customer.code();
}
```

Checking for the exact class, excluding potential subclasses

```
if (anObject.getClass() == Customer.class)
```

You can determine an object's class

Since an incorrect cast generates a runtime error, you may need to check the actual type of an object before applying the cast. Sometimes, you don't know what kind of object you have—you need to ask it. In Java, this is called *runtime type identification* or RTI.

Java defines the `instanceof` keyword to check if an object is an instance of a specified class. Notice that it is an operator, not a message. Also notice that the 'o' is not capitalized. `instanceof` is `true` if the object is an instance of the specified class or any of its subclasses.

If you need to check for the exact class type of an object, you can ask the object for its class then compare it—using the comparison operator—against a specific class. Notice the Java construct for getting the class from a class name:

```
Customer.class
```

It is not a message; there are no parentheses. You cannot simply use the class name alone. Using the class name, you can obtain the class itself. This reveals that, in Java, classes are also objects that you can access at runtime.

You can gather objects in a collection

You need a way to group objects together in a collection

Java provides many different collection classes, for example

- Vector
  - Superseded by ArrayList in Java 2
- Hashtable
  - Superseded by HashMap in Java 2

With WebObjects, you use analogous Apple foundation classes

- NSArray
- NSDictionary

These are fundamental for connecting objects into larger structures

You can gather objects in a collection

Until now, you have created explicit references for every object. Sometimes, instead of a single reference to a specific object, you need a single reference to a set of objects. For example, a shopping cart can contain an arbitrary number of items from zero, to one, to many. At runtime, you need to dynamically add and remove objects yet refer to the entire collection with one reference—the shopping cart reference.

For this purpose, Java provides utility classes called collection classes. There are different classes for different purposes. Moreover, Java 2 (with subsequent enhancements in version 1.4) provides a suite of collection classes, grouped in a framework, that supersede classes in the original language.

The Vector class (superseded by ArrayList in Java 2) works much like a traditional array: it collects objects and assigns them numbered positions like 0, 1, or 25. A Hashtable (superseded by HashMap in Java 2) also collects objects but stores them using symbolic lookup keys like strings such as "name", "email", and "phone number".

WebObjects also provides collection classes. They are part of the WebObjects foundation framework. NSArray is much like a Vector/ArrayList, NSDictionary is much like Hashtable/HashMap. In WebObjects programs, you most often use the WebObjects collection classes.

Collection classes are essential for connecting objects into larger structures. They allow you to group an arbitrary number of objects determined dynamically at runtime. With collections, you treat many objects as one: you reference the collection object rather than the objects within the collection.

NSArray maintains an ordered collection

NSArray maintains an ordered collection of objects

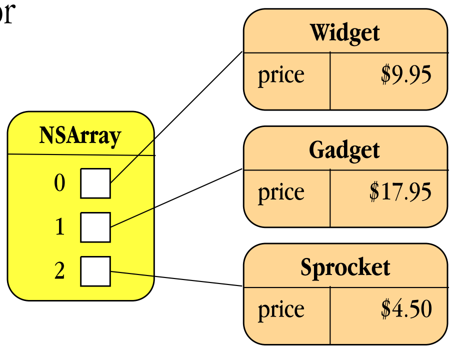
- Objects do not have to be of the same type

You access objects by their integer index—starting with 0

- NSArray provides runtime bounds checking

NSArray is similar to Java's Vector

- ArrayList in Java 2



NSArray maintains an ordered collection

NSArray objects maintain an ordered collection of objects (you cannot put values of primitive types such as int or float in an NSArray). Much like a traditional array in many different languages, NSArray stores objects using numbered positions. You access an object in the array using an index of type int. Index values start at 0. If the array holds 10 objects, they are stored with index values ranging from 0 to 9.

Array objects provide bounds checking at runtime. If you try to access an object using an index that is out of bounds—less than 0 or greater than the highest valid index, the array generates a runtime error by throwing an exception.

An NSArray object does not really contain the objects themselves, it contains references to the objects. The objects are properly "outside" but accessed by references inside the array.

NSArray is similar to Java's Vector (ArrayList in Java 2). Note that Java also defines native arrays which are different from either an NSArray or a Vector. There are at least two important differences between native arrays and collection classes:

- Native arrays gather values of the same type; collections can gather a heterogeneous set of object types.
- Native arrays can reference primitive types; collections reference only objects.

NSArray and NSMutableArray

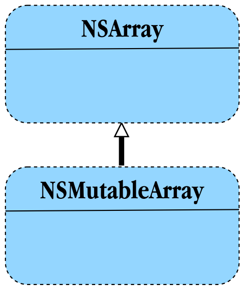
NSArray is constant—you cannot add or remove objects

NSMutableArray can be modified—you can add and remove objects

- NSMutableArray has a dynamic size—it grows automatically

NSMutableArray is a subclass of NSArray

- An NSMutableArray is a kind of NSArray



NSArray and NSMutableArray

The WebObjects Foundation framework defines two different array classes: one that is constant, one that is mutable. An NSArray is constant: once it has been created with an initial set of objects, you cannot add or remove objects. This is useful and efficient for sharing an array with other clients with the confidence that it cannot be modified.

NSMutableArray is a subclass of NSArray. It is a kind of NSArray and can be treated generically like any NSArray. But it is more specifically a mutable array: you can remove elements and add new ones. NSMutableArray responds to every message that NSArray does. It also responds to additional messages for adding and removing objects. When researching NSMutableArray, be sure to consult the NSArray documentation as well.

NSArray—useful methods

You often get an array from another object

```
NSArray items = shoppingCart.allItems();
```

Getting the current count of objects in the array

```
int count = items.count();
```

Getting an object at a specific index

```
Object anObject = items.objectAtIndex(i);
Widget widget = (Widget)items.objectAtIndex(i);
```

Searching for an object

```
if (items.containsObject(widget)) {
    int i = items.indexOfObject(widget);
    . . .
}
```

NSArray—useful methods

You often get a pre-constructed NSArray from other objects. For example, a shopping cart might define a method to return all items in an NSArray. Instances of NSArray are constant: you cannot add or remove objects, but you can access the existing objects.

You can find out how many objects are in an NSArray using the `count()` method. You retrieve an object using an index value—an integer—as the argument to `objectAtIndex()`. Recall that if you attempt to retrieve an object using an invalid index, the NSArray will generate an out of bounds exception. You can also ask an NSArray if it contains a specific object, and if so, retrieve its index value with `indexOfObject()`.

NSArray access methods are defined to return a generic object reference (Object). If you need to treat an object from an array more specifically, you must use a cast. Unlike many arrays in traditional languages, NSArray can store objects of any type, they do not all have to be the same class of object. NSArray can only store objects, not primitive types like int or double. You cannot store a `null`.

The NSArray class defines many additional methods. The methods shown here comprise a useful subset. Consult the WebObjects foundation documentation for details. Note in particular the various—frequently overlooked—constructor methods which you can use to create an NSArray containing, for example, a single object, or a collection of objects in a Java native array.

NSMutableArray—useful methods

Constructing a new mutable array

```
NSMutableArray items = new NSMutableArray();
```

Adding an object

```
items.addObject(widget);
```

Removing an object

```
items.removeObject(widget);
```

Creating a new mutable array from an existing immutable array

```
NSArray items = shoppingCart.allItems();
NSMutableArray items2 = new NSMutableArray(items);
```

NSMutableArray—useful methods

An NSMutableArray responds to the same messages as an NSArray. You can determine the count of objects in the array, get an object at a specific index, and search for an object to determine its index.

NSMutableArray also defines the methods `addObject()` and `removeObject()` for adding and removing objects respectively. When you add an object, it is placed at the end of the array at the next available index. When you remove an object, the array adjusts the indices of all objects that follow, essentially shifting them down to fill in the gap. There are additional messages for inserting and removing an object at a specific index.

You construct a new mutable array like you construct any Java object. The new array is initially empty. Its count is 0. There are no valid indices since there are no objects in the array.

What if you need to add or remove the objects in an immutable NSArray object? You can construct a new NSMutableArray and initialize it with objects from the existing NSArray. Now you can add and remove objects using the mutable array. Although you now have two different arrays you do not have multiple copies of the objects they reference. Arrays contain references not objects. You merely have multiple references to the shared, underlying objects.

NSDictionary maintains a set of key-value pairs

NSDictionary maintains a collection for efficient lookup

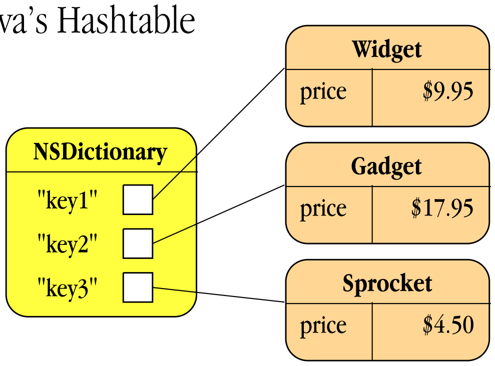
- Values in the collection must be objects

You access an object using a key

- The key can be any type of object, but it is usually a String

NSDictionary is similar to Java's Hashtable

- HashMap in Java 2



NSDictionary maintains a set of key-value pairs

Like an NSArray, an NSDictionary maintains a collection of objects (like NSArray, you cannot put values of primitive types such as int or float in an NSDictionary). But objects are not stored using numerical indices. There is no implied ordering in an NSDictionary. Rather, objects are associated with keys. You access an object using its key—another object. Usually the key is a String object where the string value is a meaningful symbol like "name", "email", or "phone number". Dictionaries in other languages are often called associative arrays or hashtables. They are said to store objects as *key-value pairs*.

An NSDictionary is useful for collecting objects that need to be efficiently accessed using a symbolic lookup key. In this sense, it is like real-world language dictionaries: you supply the word—a lookup key—and the dictionary returns the definition—the object value associated with the key. Dictionaries are implemented for efficient lookup operations. Given a key, they can quickly locate the corresponding value. To do this, dictionaries use a hashing mechanism making them similar to Java's Hashtable class (HashMap in Java 2).

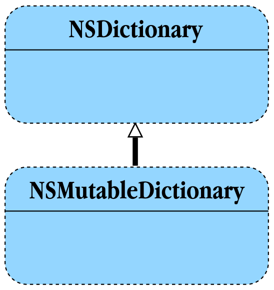
NSDictionary and NSMutableDictionary

NSDictionary is constant—you cannot add or remove objects

- NSMutableDictionary can be modified—you can add/remove keys

NSMutableDictionary is a subclass of NSDictionary

- NSMutableDictionary is a kind of NSDictionary



NSDictionary and NSMutableDictionary

Like arrays in WebObjects, dictionaries are implemented in two different classes, one constant, the other mutable. NSDictionary is constant: once the dictionary is created, you cannot add or remove objects.

NSMutableDictionary is a subclass of NSDictionary. It is a kind of NSDictionary and can be treated generically like any NSDictionary. More specifically, NSMutableDictionary extends the NSDictionary superclass with additional methods for adding and removing objects. When researching NSMutableDictionary, be sure to consult the NSDictionary documentation as well. As with NSArray, do not overlook the various constructor methods.

An NSMutableDictionary does not have a fixed size. You can add new objects and the dictionary automatically grows in size to accommodate them. An NSDictionary does not have a concept of bounds checking either. If you ask for an object using a key that is not in the dictionary, the dictionary simply returns `null` to indicate that there is no corresponding object.

NSDictionary—useful methods

You often get a dictionary from another object

```
NSDictionary props = customer.properties();
```

Getting the current count of objects in the dictionary

```
int count = props.count();
```

Getting an object value using a key

```
Object name = props.objectForKey("name");

String phoneNumber =
    (String)props.objectForKey("phone number");
```

If there is no object for that key, the dictionary returns `null`

```
String email = (String)props.objectForKey("email");

if (email != null)
// dictionary contains value for "email"
```

NSDictionary—useful methods

You often get a pre-constructed dictionary from another object. Imagine that a shopping cart can report a set of properties about itself—model name, serial number, the vendor that built it, or that a customer can provide a dictionary containing name, phone number, and email address.

You can determine the number of objects in a dictionary using the `count()` method. You can get individual objects—values—from the dictionary if you know the correct key using `objectForKey()`.

Like array methods, dictionary methods are declared to return generic object references. Dictionaries can hold any kind of object. A single dictionary often collects object values of several different class types: strings, dates, numbers, and custom classes like customers and shopping carts. A generic object reference is valid for any class type. When you get an object from a dictionary, you typically use a cast to treat it as a more specific class type.

Recall that if the dictionary does not contain a value for the requested key, `objectforKey()` returns `null`. When in doubt, check the return value. Because of this convention, you cannot store a `null` in a dictionary.

NSMutableDictionary—useful methods

Defining and constructing a new mutable dictionary

```
NSMutableDictionary items =
    new NSMutableDictionary();
```

Adding an object for a key

```
items.addObjectForKey(widget, "product");
```

Removing a value

```
items.removeObjectForKey("product");
```

Creating a new mutable dictionary from an existing dictionary

```
NSDictionary props = shoppingCart.properties();
NSMutableDictionary props2 =
    new NSMutableDictionary(props1);
```

NSMutableDictionary—useful methods

You can construct a new NSMutableDictionary. Initially, it contains no key-value pairs. Its count is 0. Any attempts to retrieve a value will return `null`.

You can add a new object to the NSMutableDictionary, associating it with the specified key, using the method `addObjectForKey()`. If an object is already stored in the dictionary for that key, it will be replaced by the new object. You can explicitly remove an object associated with a specific key using the method `removeObjectForKey()`. Subsequent attempts to get an object for that key return `null`.

In some cases, you will want to create a mutable version of an immutable dictionary. You can construct a new mutable dictionary, providing the existing dictionary as an argument to the constructor.

Like NSArray, instances of NSDictionary do not really contain objects, only references to objects.

To use a Java class, import its package

Classes are grouped into libraries or packages of related functionality

There are many classes in many different packages

- Packages that are part of the Java runtime
- Custom packages from 3rd parties or your organization

To use any class in your code, you must import its package

NSArray and NSDictionary are in the WebObjects Foundation package

```
import com.webobjects.foundation.*;
```

The `java.lang` package is automatically imported

- `java.lang` includes basic classes like Object and String

To use a Java class, import its package

The Java runtime environment defines a large number of standard classes that you can use to build your applications. Products like WebObjects define even more. Your own organization may define its own set of reusable classes.

In Java, classes are organized into *packages*. A package typically groups related classes that address a specific set of problems. One package might provide advanced math operations and extended value classes. Another package provides classes for performing file I/O. Yet another package deals with networking. Packages make classes easier to find and to use. They also make it easier to avoid naming conflicts, and to control access.

In general, to use a class in your code, you must explicitly *import* the package that defines it. The `import` statement specifies a class name including its package name. You can use an asterisk to import all classes in a package. If you use a class name without also providing an appropriate import statement, the Java compiler generates an error specifying that it does not recognize the class.

There is one package that is automatically imported for you: `java.lang`. This is the most basic of all packages since it defines fundamental classes like Object and String. You do not have to explicitly import a package when using just these basic classes.

Iterating over the items in a collection

Use an Enumeration object with a `while` loop

```
import java.util.*; // package with Enumeration
import com.webobjects.foundation.*;  // NSArray

double total = 0;
NSArray items = shoppingCart.items();
Enumeration e = items.objectEnumerator();
while (e.hasMoreElements()) {
   Product item = (Product)e.nextElement();
   total = total + item.price();
}
```

Don't modify the collection while enumerating

Java 2 provides Iterator and ListIterator

Iterating over the items in a collection

When using a collection, you will often need to process every object it contains. This is called *iterating* over the collection or *enumerating* the elements of a collection. Consider the `balance()` method of a shopping cart object: it must iterate over each of its items, get its price, and add it to the total. Java provides two tools for getting the job done—an Enumeration object and a `while` loop. Java 2 also provides Iterator and ListIterator classes, which supersede Enumeration.

You get the enumeration object from the collection. It returns an object capable of enumerating all objects currently stored in that specific collection. Objects of type Enumeration respond to the following messages:

- `hasMoreElements()`—returns `true` if there are more objects to visit.
- `nextElement()`—returns the next object in the collection.

In Java 2, the Iterator class implements the corresponding methods `hasNext()` and `next()`.

Use the enumeration object with a `while` loop to process each object in the collection. The while loop is a code block which is repeatedly executed as long as the conditional test—a boolean expression—evaluates to `true`:

```
while (condition) {
      loop body ...
}
```

Java also provides `for` and `do-until` loop statements not shown here.

While using an enumeration, you should not add and remove objects from the collection.

Wrapper classes turn primitives into objects

Collections only store non-null object references

- Can't store `null` as a value in a collection
- Can't store primitive types—`int`, `float`, `boolean`, etc.

Java defines wrapper classes for treating primitives like objects

```
Integer     Long        Float       Double
Short       Character   Byte        Boolean
```

Required for some method arguments and return values in other classes

Wrapper classes are automatically imported—part of `java.lang`

Wrapper classes turn primitives into objects

Remember that Java is a hybrid language—not all data types are objects. Your code often makes use of simple values typed as int, double or boolean. In some cases, though, you need to treat these primitive values as objects. Collection classes like NSArray and NSDictionary cannot store primitive types. They can only store objects. Many other classes define method arguments and return values as object types and similarly, will not handle primitive types.

Java defines a special set of classes called wrapper classes. Their purpose is to wrap an object container around a primitive type. Wrapper classes enable you to turn primitive types into objects suitable for storing in a collection or passing to any method that requires a true object type. There is a specific wrapper class for each underlying primitive type—Integer for int, Double for double, and so on.

From a wrapper object, you can extract the original primitive type value. You can convert the type in both directions—from primitive to object and object back to primitive.

The wrapper classes are fundamental classes in the Java language. They are defined in the `java.lang` package which is automatically imported for you.

Conversions between primitive and object types

From primitive to object

```
int i = 10;
Integer number = new Integer(i);
```

From object to primitive

```
i = number.intValue();
```

Wrapper objects are immutable—you cannot modify the value

Conversions between primitive and object types

Here is a simple illustration. Assume you have a primitive type value, an int:

```
int i = 10;
```

You can create an instance of the Integer wrapper class that contains the int, thereby turning a primitive value into an object:

```
Integer number = new Integer(i);
```

You can now store this value in a collection such as an array:

```
array.addObject(number);
```

Later, you can retrieve the object from the collection and extract the original primitive value again:

```
Integer number = array.objectAtIndex(x);
int i = number.intValue();
```

The wrapper classes provide many additional capabilities for converting between different types, parsing values from strings, and generating values as formatted strings. Consult the Java documentation for additional details.

Additional foundation classes used with WebObjects

BigDecimal—arbitrary precision fixed point floating point number

```
import java.math.*;
```

NSTimestamp—calendar date, time, and time zone

```
import com.webobjects.foundation.*;
```

NSData—buffer of arbitrary binary data

```
import com.webobjects.foundation.*;
```

Additional foundation classes used with WebObjects

WebObjects applications commonly make use of additional foundation classes. You should familiarize yourself with each of these classes.

The Java math package defines the BigDecimal class useful for representing large decimal numbers with specific rules for rounding and formatting. BigDecimal is ideal for storing monetary values. When you incorporate database connectivity into your WebObjects applications, you usually fetch number values as instances of BigDecimal.

The WebObjects foundation package defines the NSTimestamp class. NSTimestamp objects represent time and date values. NSTimestamp includes a simple way to ask for the current time and, through NSTimestampFormatter, rich formatting capabilities. NSTimestamp's superclass is java.sql. Timestamp, which in turn inherits from java.util.Date. These define methods for comparing and calculating dates. To extract pieces of the time and date like the month, the year, the minute, and the second, however, you need to use java.util.GregorianCalendar, as in the following example.

```
NSTimestamp aTimestamp = new NSTimestamp();
GregorianCalendar aCalendar = new GregorianCalendar();
aCalendar.setTime(aTimestamp);
int year = aCalendar.get(GregorianCalendar.YEAR);
```

The NSData class also comes from the WebObjects foundation package. It is used to represent an arbitrary buffer of binary data. Dynamically generated images or the contents of an uploaded file are good examples. Think of an NSData object as a collection of bytes that can be conveniently handled with a single object reference.

Behavior versus type—methods versus class

Objects can be of many different types—unrelated classes

```
ShoppingCart cart;
BankAccount account;
Inventory inventory;
```

But they can have analogous behavior—respond to the same messages

```
double balance = cart.balance();
double balance = card.balance();
double balance = inventory.balance();
```

Often, you need to type by behavior, not class

```
? thing = (?)items.objectAtIndex(0);
double balance = thing.balance();
```

Behavior versus type—methods versus class

In real-world Java programs, you use many objects of many diverse class types. They are often not related to each other in terms of the inheritance hierarchy. They don't share common superclasses except that they are all subclasses of the most generic class, Object. Consider how fundamentally different the following classes are from each other: ShoppingCart, BankAccount, Inventory.

Though unrelated in the class hierarchy, they have analogous behavior—they implement the same methods. What ShoppingCart, BankAccount, and Inventory might have in common is some aspect of their behavior: they each respond to the `balance()` method.

In many cases, you need to write code that works with a diverse set of objects that have common behavior, regardless of their dissimilar class types. You might write some code that takes any object and displays its balance to a user interface. You are depending on the fact that the object implements a `balance()` method. You specifically want to avoid making any assumptions about the class type of object. You don't really care about the class type at all.

This poses a simple coding problem: what type should the reference variable be? In this case, you want to type by behavior, not by class. A generic reference of type Object is not sufficient. The Object class does not define a `balance()` method.

Interfaces provide another kind of type

An interface defines a name for a group of related methods

An interface defines a type of behavior, independent from class type

A class is an implementation, an interface is only a specification

Often, you use an interface name rather than a class name

```
// objects with balance() behave like Assets
Asset thing = (Asset)items.objectAtIndex(0);
double balance = thing.balance();
```

Interfaces can also provide constants

```
double taxRate = Asset.TaxRate;
```

Packages define interfaces and classes implement them

Interfaces provide another kind of type

Java defines an alternate to classes for typing objects called *interfaces*. An interface provides a list of methods that define a type of behavior—a role. An interface has a name. An interface defines a formal type in Java.

Compare an interface with a class. A class defines all the attributes and all the methods valid for an object of that class. A class is a blueprint that describes the structure and origin an object. A class is an implementation. An interface is simply a list of methods. Interfaces typically capture only a subset of what an object can do. An interface is merely a specification that one or more classes may adhere to. A particular class may implement multiple interfaces.

Returning to the example, ShoppingCart, BankAccount, and Inventory are all class names. Each class definition provides details particular and specific to objects of that class. ShoppingCarts can add and remove products. BankAccounts have numbers and statements, credits and debits. An Inventory has a depreciation rate.

Yet ShoppingCart, BankAccount and Inventory objects all have a subset of behavior in common—they all respond to the `balance()` message. It is possible to define an interface called Asset that lists the messages you can send to any object that behaves like an Asset. In this case, the Asset interface defines one message—`balance()`.

Use interfaces to type an object according to its behavior, regardless of its class.

Java naming conventions

Java is case-sensitive

- `ShoppingCart` is different than `shoppingcart`

Class names are capitalized; inner words are capitalized

- `Asset`
- `ShoppingCart`

Method and variable names start with lowercase letters

- `index`
- `checkOut()`

Java defines many reserved words—use them only as intended

- `null`, `if`, `int`, `boolean . . .`

Java naming conventions

Naming conventions are often just that: they are conventions rather than strictly enforced rules. But you should take them seriously. Faithfully adopting these rules will help you in several ways:

- You will be to understand another programmer's code more efficiently
- Another programmer will be able to understand your code more efficiently
- You can understand documentation more efficiently
- You are more likely to write correct code.

Java is case-sensitive. "ShoppingCart" is different than "shoppingcart". Many compiler errors encountered by beginning programmers are merely misspellings that use the wrong case.

Class names are always capitalized. Multiple word names capitalize the first letter of each subsequent word. Method and variable names start with lowercase letters but also capitalize the first letter of subsequent words in multi-word names. Static constants vary in their naming, but usually begin with a capital letter. Often, the entire name is in uppercase.

Remember that there are many reserved names in Java—`null`, `while`, `if`, `int`, `boolean`, `void`, to list but a few. You should use them only as intended. Do not accidentally reuse them for your own variable names.

Common pitfalls

Compile-time errors

- Incorrect class or interface name—misspelled or missing import
- Incorrect method or attribute name—invalid or misspelled
- Missing a cast—method name is not valid for generic type

Runtime errors

- Sending a message to null—forgot to create and assign an object
- Applying an incorrect cast—the object is not the type you expect
- Exceeding the bounds of an array—index is negative or too high
- Adding a primitive or null to a collection—only non-null objects

Common pitfalls

If you are a new Java programmer, you may likely run into a few common coding pitfalls. They show up as errors during compilation or at runtime. Often, the problem is simple: you forgot to import a class or you are missing a cast. One of the most common pitfalls shows up as a null pointer exception at runtime. The cause is typically that you forgot to create a new object and assign it to a reference variable. Without an object, you end up sending a message to null.

The Java compiler is quite good about providing meaningful error messages. Be sure to read the message carefully. The compiler typically lists the exact line number where the problem occurred. Review your code patiently until you locate the problem.

In many cases, the problem is not due to a fundamental lack of understanding but a peccadillo of coding syntax. With practice, you will learn to identify and remedy these common errors quickly. The list above provides some of the most common errors as a convenience to help you on your way.

3

Creating Classes

Thinking like a Class Producer
A View from the Inside

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

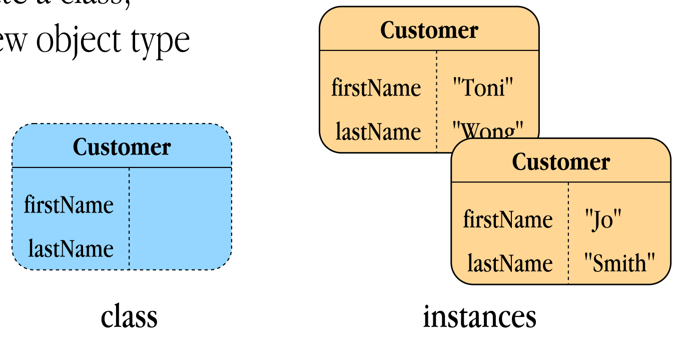
The class is the basic code packaging mechanism in Java

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 belong to each object, each instance of the class. Sometimes, a class is called a factory to illustrate its function—a class manufactures objects.

Classes derive from an inheritance hierarchy

`Object` is the root class

Every other class is a subclass

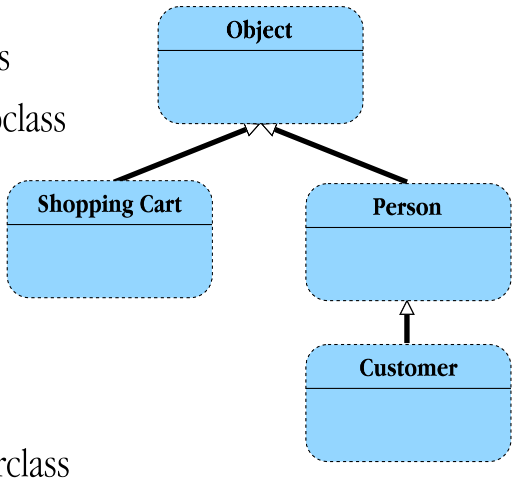
Every subclass has one super class

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 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*.

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

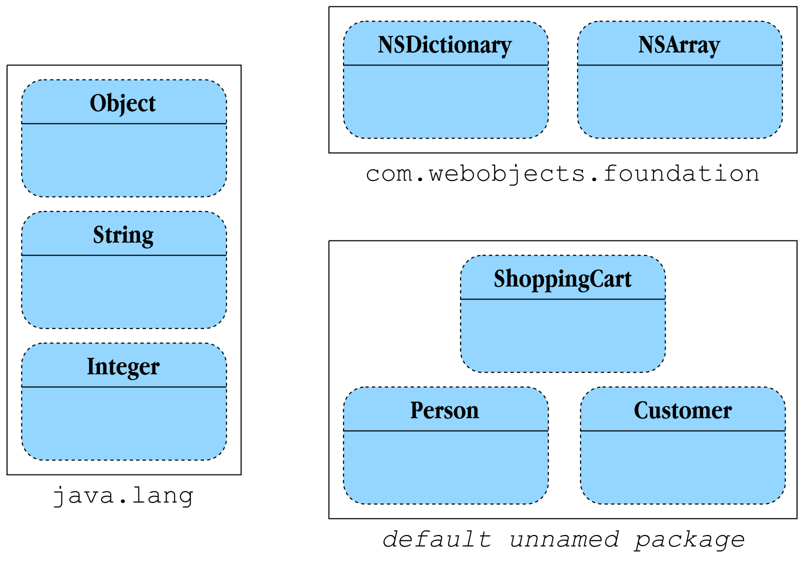
When you create a new class, you always create a new subclass of an existing class. You are creating a more specialized version of the superclass. You automatically inherit the qualities of the superclass. From there, you can specialize, by replacing existing functionality, and adding new functionality.

You can 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.

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.

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. If you do not have a related superclass to start with, use Object as the superclass.

Classes are grouped into packages



Classes are grouped into packages

In addition to the class hierarchy, Java defines another mechanism for organizing code: a *package*. A package is a collection of one or more related classes. Every package has a name, such as `java.lang`, `java.util`, or `com.webobjects.foundation`. 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 adds a package for building web-based applications, a package of useful foundation classes like arrays and dictionaries, and a third package for database connectivity—Enterprise Objects Framework.

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.

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 a class—objects—can always access their own 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, with its superclass, 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 can also take 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 value1, 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();
  }
  // leave method returning value
  return balance;
}
```

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 the class's 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(int i) { count+=i; }
```

One method can invoke another

```
public void addOne() { increment(1) };
```

Every object has a reference to itself named `this`

```
public void increment(int i) {
    this.count+=1;
}
public void addOne() {
    this.increment(1)
};
```

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—`this`—to make it 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;
}
```

Alternate 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. It also 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" (and capitalizes the variable name). 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, though this is not common practice).

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 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 that is potentially reusable in multiple scenarios.

Remember, overloaded methods have the same name, but differ by the number and type of arguments. 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

When a class inherits a method, by default it responds to the corresponding message as though it had implemented the method itself. Often, however, you need to modify the response to an inherited method. While a subclass cannot take the method away, it can change the implementation. This is called *overriding*. Overriding is different from overloading. 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:

1. Completely replace the implementation—forget the superclass's version.
2. 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 use in this circumstance.

To extend when overriding, use the `super` keyword

Overriding to extend the superclass method

```
public double balance() {
    // call the superclass 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. You can make use of the balance logic already correctly implemented in the superclass, but 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();
```

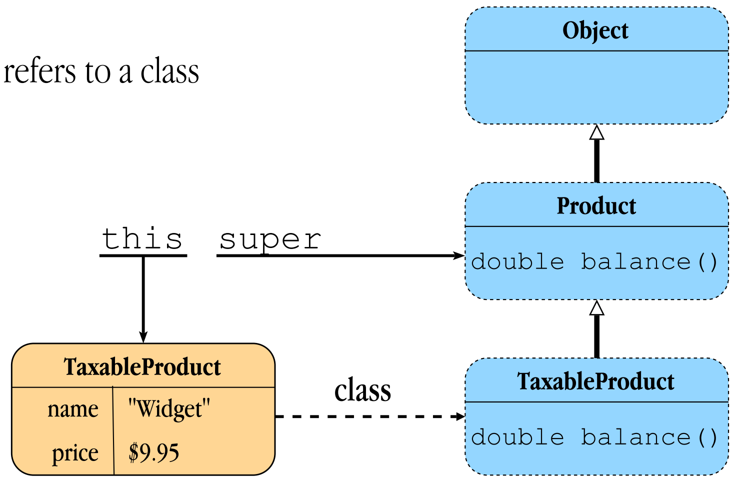
In both cases, the statement will find the method, whether the class implements it itself, or inherits it from a superclass. When overriding, you need to bypass the implementation in the current class and invoke the implementation in the superclass. To do so, use the keyword `super`:

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

A closer look at `this` and `super`

`this` refers to an object instance

`super` refers to a class



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`.

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 to instance variables.

While a constructor looks much like any other 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 . . .

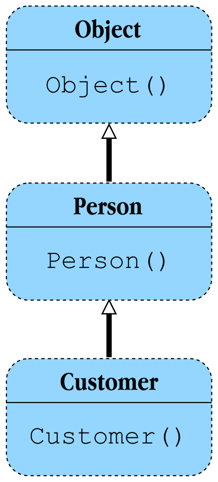
```
new Customer()
```

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 = "Jo";
}
```

Otherwise, it is automatically called by the Java runtime implicitly

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

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.

If you 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("Jo","Doe");
```

Multiple constructors with overloading

You can provide multiple constructors by overloading

One constructor can call another 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. Recall that 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" (no-argument) 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 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 tax rate applied to all shopping carts. It is clearly more efficient to store a single value for the tax rate in the class, than storing a redundant copy in each shopping cart instance. Furthermore, this design 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 variable. 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 again 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

The code example in the slide 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, application-wide, copy of a static variable —it belongs to the class itself. While it is accessible to each object instance, it is shared by all objects.

Local variables are defined within 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 warns you if 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 = .085;
```

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. Indeed, abstract classes cannot be instantiated. They provide a template but require a concrete subclass implementation to make them usable.

You can also define methods to be abstract. 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.

Java defines the `abstract` keyword for marking a class or a method 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 = .085;
  public double balance();
}
```

You can define new interfaces

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.

You can define your own interfaces. 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 therefore provide an alternative to 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 properly implement an interface, your class must provide an implementation for every method declared in the interface. The compiler will give an error if you do not. You can inherit a method 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, since it offers the flexibility to be free from the constraints of a particular class hierarchy.

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 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 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 from 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 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.

Getting a Bigger Picture

Optional Reading
Useful Even for Java Hackers

Goal

Consider the larger context of a complete Java application.

Prerequisites

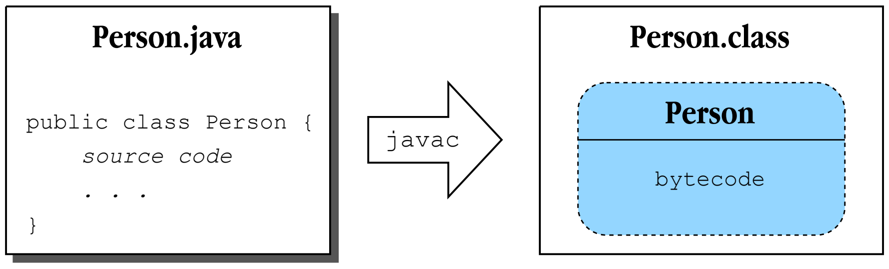
Familiarity with Java programming, and the concept of an event-driven server application.

Objectives

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

- Identify key components of a typical Java application, such as bytecode, class files, packages, and the Java virtual machine
- List the general steps a WebObjects application takes between startup and activating your custom class code
- Describe how frameworks differ from traditional libraries

Java source code is compiled into bytecode

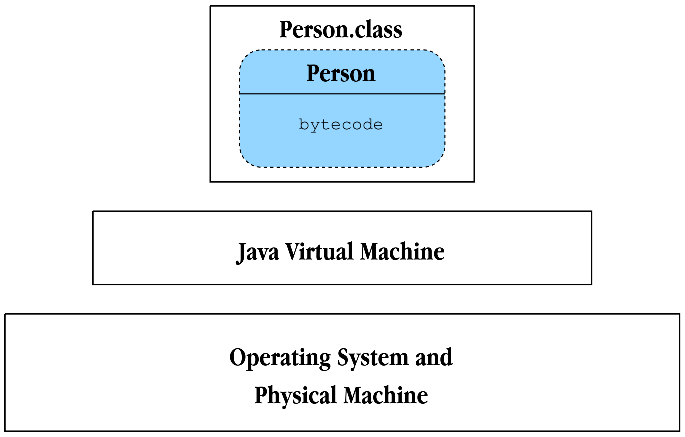


Java source code is compiled into bytecode

Like most programming languages, Java is compiled. Before you can use your custom classes, you must compile their source code, using a compiler such as `javac`. Unlike most programming languages, Java does not compile into binary machine language. Java uses an intermediate form for compiled code called *bytecode*. Since it is not machine code, it cannot run directly on the CPU of a physical machine. To execute, bytecode requires translation software called the Java virtual machine (JVM). In this sense, Java is more like an interpreted language with a runtime component. The JVM reads and executes bytecode much like any interpreter.

Your Java source is stored in files ending with the .java extension. The compiler generates bytecode and stores it in a file with the .class extension. ShoppingCart.java compiles into ShoppingCart.class. Bytecode is not human readable, editable, nor directly executable. Bytecode is intended only for the JVM.

Bytecode is executed by the virtual machine

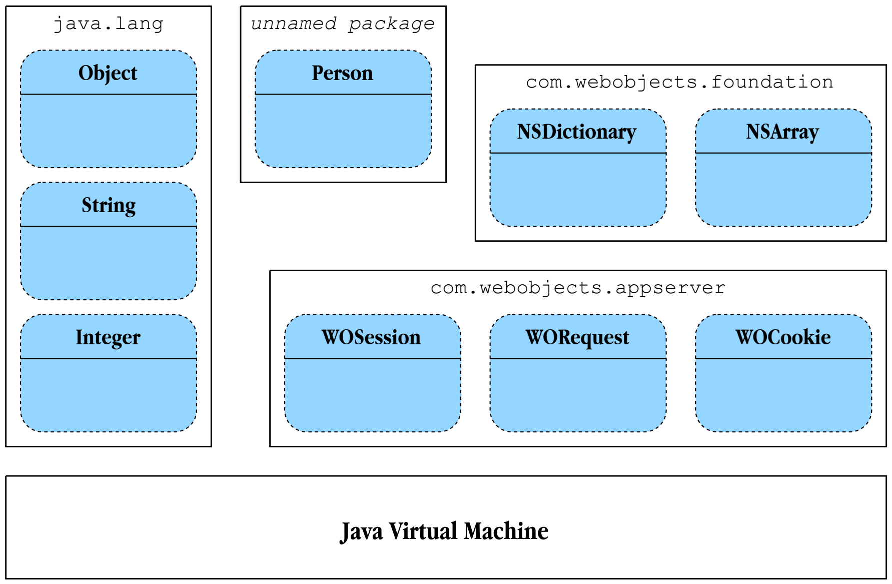


Bytecode is executed by the virtual machine

Bytecode is ready to execute, but requires the JVM. The JVM is an executable program itself. It is like an interpreter: it reads bytecode and translates it into machine code, which is executed by the underlying operating system and hardware platform. The JVM allows the bytecode to run on the physical machine.

Because of the JVM, bytecode is operating system and hardware independent. As long as a server machine has a suitable JVM, it can run any Java bytecode. Even in binary form, compiled Java code is highly portable. This makes Java ideal for serving binary applications over the Web to a wide range of diverse client machines.

Your application incorporates many packages



You application incorporates many packages

Your Java application uses many classes in addition to the custom classes you create. A Java application incorporates a collection of packages, each of which contains bytecode for many related classes, typically from multiple vendors.

Java defines several basic packages that are available on any Java runtime implementation. WebObjects applications commonly make use of classes in the `java.lang` and `java.util` packages.

WebObjects applications make use of pre-defined classes from additional packages specific to the WebObjects runtime environment. WebObjects supplements the Java foundation with classes like NSArray, NSDictionary, and NSSet. Built on these foundations, the WebObjects package defines several classes that handle the infrastructure of Web-based applications such as WOApplication, WOSession, and WOComponent.

Your custom classes complete the final application by adding business-specific logic, and controlling its overall look and feel. Your classes may be grouped in custom packages you define. Most often, your classes are simply collected into the unnamed default package.

Launching your application

Java applications are launched in different ways

- A Java applet class is downloaded to a Web browser
- A Java class's main method is invoked from the java tool
- An application wrapper—executable program—is launched

WebObjects applications are server-side application wrapper programs

- Launched like any daemon process on a server
- Service incoming requests from Web browsers
- Gain control from the HTTP server through CGI
- Pass control to your Java code during request processing

Launching your application

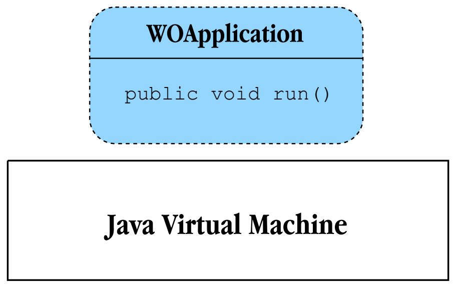
There are different ways to launch a Java application. Each technique leads to the same basic result: executing bytecode in the JVM. What differs between the techniques is where and how the Java machine is started and which class is used as the initial startup class.

Applets are essentially small Java applications that are downloaded from a Web browser to a client machine and executed in a Java machine embedded within the browser application. Although you can use Java applets within a WebObjects application, WebObjects applications are not client-side applets—they are server-side applications.

You can launch full-fledged independent Java applications from the Java command line tool—java. This is essentially the Java machine as an independent program. With a command-line parameter, you specify the class that contains the main entry point—a static method named `main()`.

WebObjects applications use a third technique: they are complete command-line applications in and of themselves. A WebObjects application completely encapsulates the details of the JVM. A WebObjects application is launched like any executable native to your server machine. It is a server application from which clients download a (typically HTML-based) user interface.

The WOApplication class provides the first object



When a WebObjects application launches, it

- Starts the Java virtual machine
- Creates an instance of a custom WOApplication subclass
- Transfers control to it by invoking the `run()` method
- Waits for incoming Web client requests

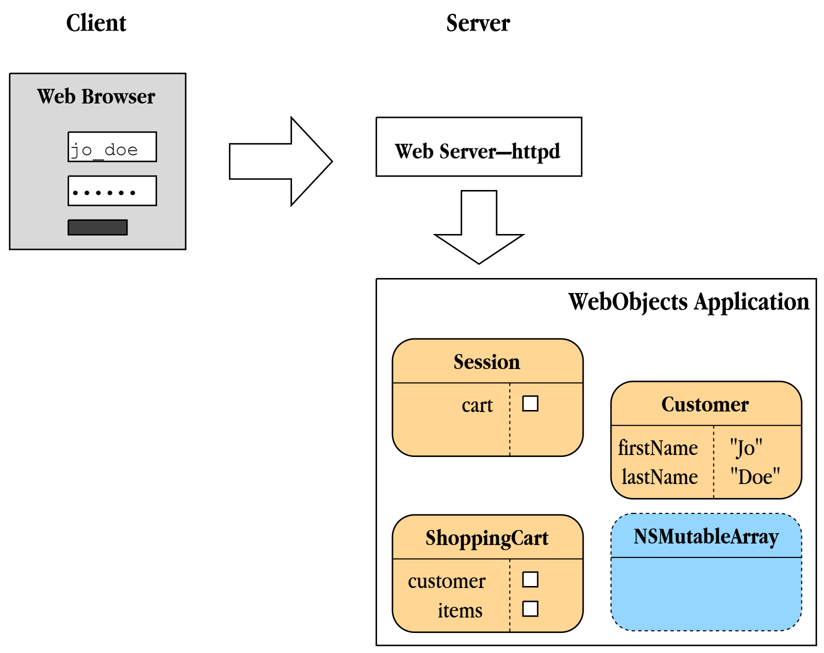
The WOApplication class provides the first object

When a WebObjects application is launched, it automatically starts the JVM and passes control to a specific startup class named WOApplication. To be more precise, the application creates an instance of a custom WOApplication subclass named Application. You implement the Application subclass and thereby gain early control in the lifetime of a WebObjects application.

You inherit many attributes and much useful behavior from the WOApplication subclass. It is possible that you needn't customize WOApplication at all. WOApplication implements the `run()` method that puts the WebObjects application in state where it is ready to service incoming requests from web clients. Once the application is listening for incoming HTTP requests, its flow of control is driven by events generated from Web clients.

In this sense, your application is much like any CGI program implemented as a server daemon. The HTTP server receives an incoming request which it passes to your WebObjects application for servicing.

Your code is activated from a client's request



Your code is activated from a client's request

Ultimately, your custom code is activated in the context of servicing a web client's request. Control passes from the Web server daemon to your WebObjects applications where the Application object takes control.

Although there is additional infrastructure omitted from the diagram for clarity, control eventually passes to your custom classes such as the ShoppingCart and the Customer. Imagine that a Web user logs into the application: a Session object is created to represent the user, and provide state management. While browsing your catalog, the user chooses to purchase an item: a ShoppingCart object is created to hold the object and so on. When the user checks out, a Customer object is created, either from details the user enters in a from, or perhaps retrieved from a database based on login information.

It is important to see that your code is typically activated indirectly, as a result of a client request. In this sense, WebObjects applications are said to be event-driven. The core of the application's logic is a repetitive action: wait for request, process the request, wait for another request, process the request, and so on. This is called an event loop which, in the case of a WebObjects application, is called the *request-response loop*.

Libraries, frameworks, and packages—reusable code

Library

- Provides low-level interfaces for specific tasks
- Generally affects your design on a small scale

Framework

- Provides high-level structures for general design problems
- Generally affects your design on a large scale

Package

- Java collection of reusable classes
- Can be a library or a framework

Libraries, frameworks, and packages—reusable code

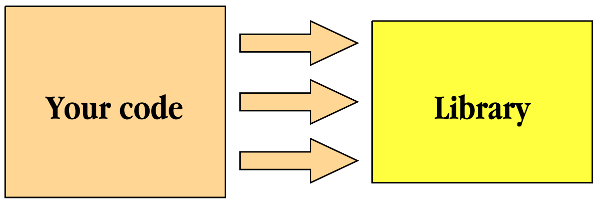
When looking at the bigger picture, it is useful to clarify the terms library, framework, and package. All three refer to software constructs for sharing reusable code.

A library is a traditional software construct for gathering reusable programming interfaces. Libraries contain a set of functions or procedures useful as low level convenience routines for specific tasks. One library might define functions for file I/O. Another might support a wide range of advanced math functions. Libraries can be object-oriented, in which case these conveniences are represented as reusable classes. Libraries typically have a small impact on the overall design of your application. You use the library functions when you need them within the larger application structure of your own design.

A framework is also a collection of reusable interfaces or classes, but it addresses a larger design problem on a broader scale. As its name implies, a framework provides more comprehensive structures for designing entire applications within a specific problem domain. There are frameworks for graphics editors, financial modeling, database, and Web applications. A framework has a major impact on the overall design of your larger application structure.

A package is simply a Java mechanism for collecting related reusable classes. A Java package may be as simple as a library or as elaborate as a framework. Conceptually, the java.util package is a library. The WebObjects package, on the other hand, is a framework.

Traditional programming model using libraries



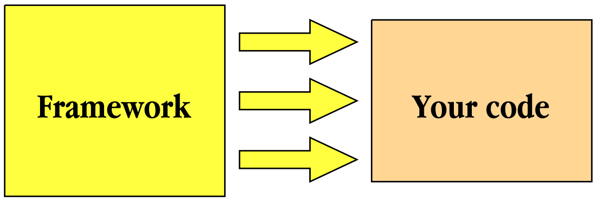
- You have the initial control—typically through a main entry point
- You sequence the main flow of control
- You call the library
- You define the application structure
- You make most of the decisions

Traditional programming model using libraries

From a big and somewhat oversimplified perspective, consider the traditional programming model using libraries. Your custom code takes initial control through a well-defined entry point. From there, your application logic sequences the main flow of control, calling library routines for low-level conveniences. You open and read a file. You perform some specific and elaborate math computations. You return to your own code to continue the main flow of logic.

You call the library within a larger application structure that you design. You make most of the large-scale design decisions.

Event-driven model with frameworks



- Framework has the initial control—typically through an event
- Framework sequences the main flow of control
- Framework calls you
- Framework defines the application structure
- You integrate with framework design patterns
- Framework makes most of the decisions

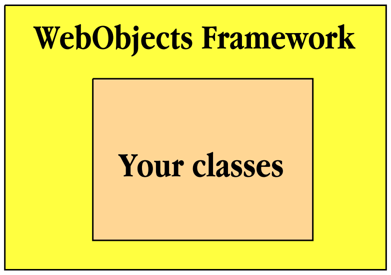
Event-driven model with frameworks

Because frameworks address larger designs problems on the scale of complete applications rather than low level convenience functions, the framework programming model is significantly different than the traditional library model. This is especially true for event-driven applications like Web applications. Compared to the library model, the framework model is practically the reverse.

The framework typically takes initial control, both at startup and when servicing an event. The framework sequences the main flow of control, not you. Most often, the framework calls you, rather than the other way around.

The framework defines the overall structure of the application and makes most of the large-scale design decisions. You integrate your code within the framework using well-defined design patterns.

WebObjects is a framework



- Web applications are driven by events—client requests
- WebObjects is a framework for servicing them
- Your classes fit within the WebObjects application framework
- You play a specific role within in a bigger picture
- WebObjects handles the big picture

WebObjects is a framework

WebObjects is a framework that supplies much of the structure of your custom WebObjects application. The model is event-driven so that your code is usually activated indirectly by callbacks. Framework classes gain initial control and call you when the time is right. Your classes fit within a larger, general application structure defined by WebObjects. You play a specific role within a bigger picture.

This can be confusing to new WebObjects developers, especially when they are also new to object-oriented and event-driven programming. The flow of control seems indirect, even magic. Who is calling your code? How and when do you gain control? What is the larger and seemingly hidden design implied by the WebObjects framework?

Keeping the event-driven framework model in mind will help. WebObjects presents a slightly steeper learning curve since you have to understand the larger picture. To some extent, you have to yield control, accept and trust a more elaborate infrastructure.

But this is not giving up flexibility or power. Indeed, it is the opposite. WebObjects provides a coherent and highly generalized framework for an arbitrary number of sophisticated custom applications. In many ways, the tedious and general infrastructure problems are solved once and for all. You don’t have to reinvent the wheel. You are free to focus on the interesting problems, the meaningful content of a new application—the fun stuff!

Project Builder manages the details

Project Builder is the IDE tool in WebObjects

From a single graphical application you can

- Create new application projects
- Create and edit new Java classes
- Launch additional graphical modeling and layout tools
- Compile and install
- Launch and debug your WebObjects application

No need to handle the details of Java environment

```
javac, CLASSPATH, java, etc. . .
```

Project Builder manages the details

While the WebObjects framework handles much of the big picture, the WebObjects developer tools handle many of the small, mechanical details. WebObjects developer tools include Project Builder, an integrated development environment (IDE) for managing WebObjects application projects.

From Project Builder, you can perform all the common tasks for developing, compiling, testing and debugging your evolving application. Project Builder manages the details of invoking the Java compiler, collecting the Java class and package files, linking frameworks, setting the `CLASSPATH` environment variable, and launching your application.

While you may wish to use other Java tools, you generally don't have to. You can focus mostly on the Java language and worry less about the Java runtime environment.

5 Handling Exceptions

When bad things happen to good objects
The essentials of Java error-handling

Goal

To effectively catch and handle runtime exceptions.

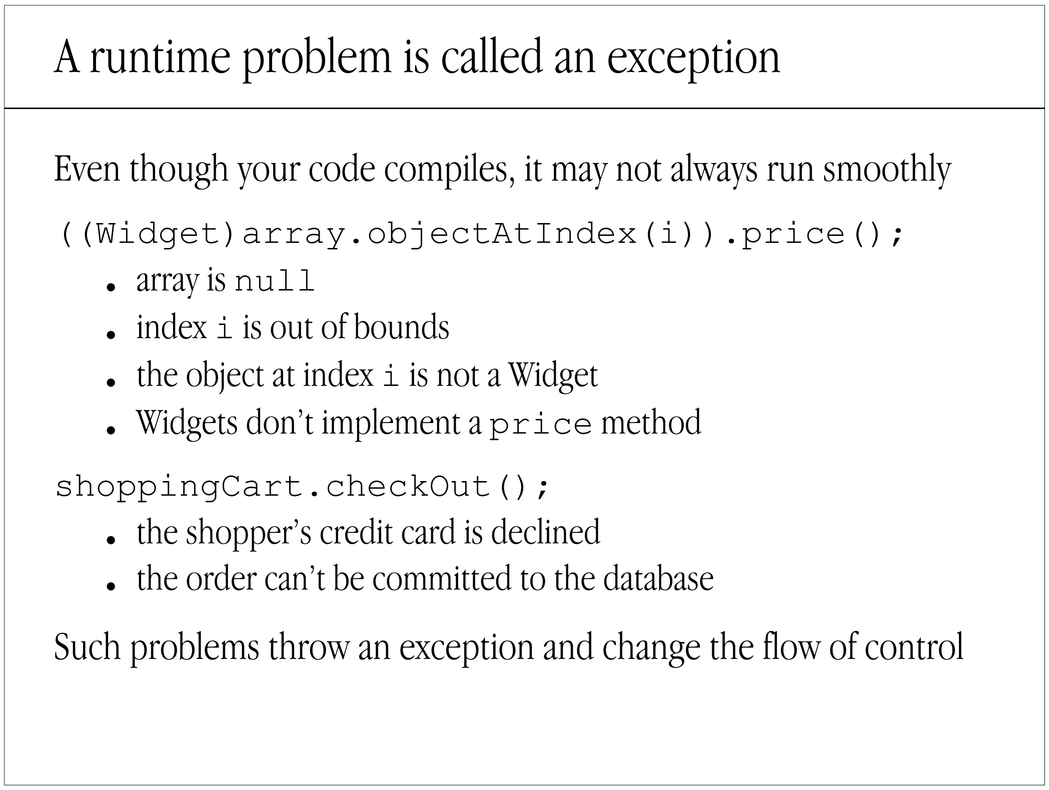
Prerequisites

Basic Java programming skills.

Objectives

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

- List several common runtime problems
- Catch an exception
- Distinguish among multiple exceptions types
- Define a new exception class
- Throw an exception



A runtime problem is called an exception

Even though your code compiles, it may not always run smoothly

```
((Widget)array.objectAtIndex(i)).price();
```

- array is `null`
- index `i` is out of bounds
- the object at index `i` is not a Widget
- Widgets don't implement a `price` method

```
shoppingCart.checkOut();
```

- the shopper's credit card is declined
- the order can't be committed to the database

Such problems throw an exception and change the flow of control

A runtime problem is called an exception

Once your Java code compiles successfully, you can launch your application and run the code. Depending on what happens during the life of your application, though, the code may encounter problems. If your code is well constructed, a runtime problem is not the normal case, but an exceptional one. The Java language and runtime environment define a formal and practical way to deal with exceptions, and to create robust production-ready code, you will have to make use of a few important features.

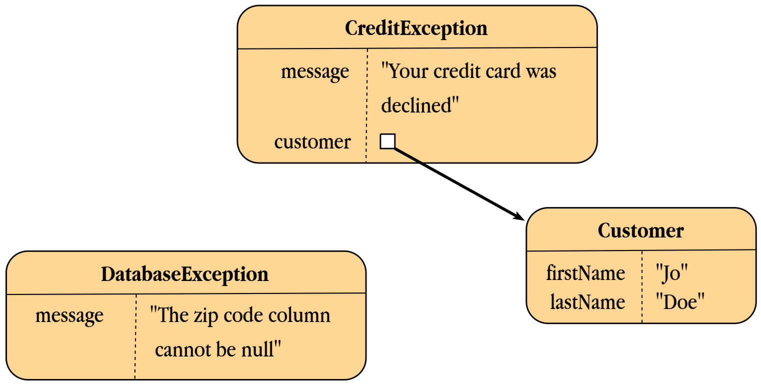
Notice, a runtime problem might be a language issue: an object reference is null, an array index is out of bounds, a cast turns out to be false, and so on. These exceptions usually indicate a fundamental logic or code problem in your implementation. In other cases, your code may be well written, but includes true exceptional cases beyond your control: a customer's credit card is rejected or the database encounters an error when committing a transaction.

In all of these cases, the Java runtime brings the problem to your attention by throwing an exception. By throwing an exception, Java instantly changes the normal flow of control of your code. In the end, the exception requires your attention which you provide through a bit of custom code.

An exception object identifies the problem

An exception is an object representing the error

- Specific types of exceptions are represented by specific Java classes
- An exception may contain data that further identifies the problem
- All exceptions contain an error message (and a stack trace)

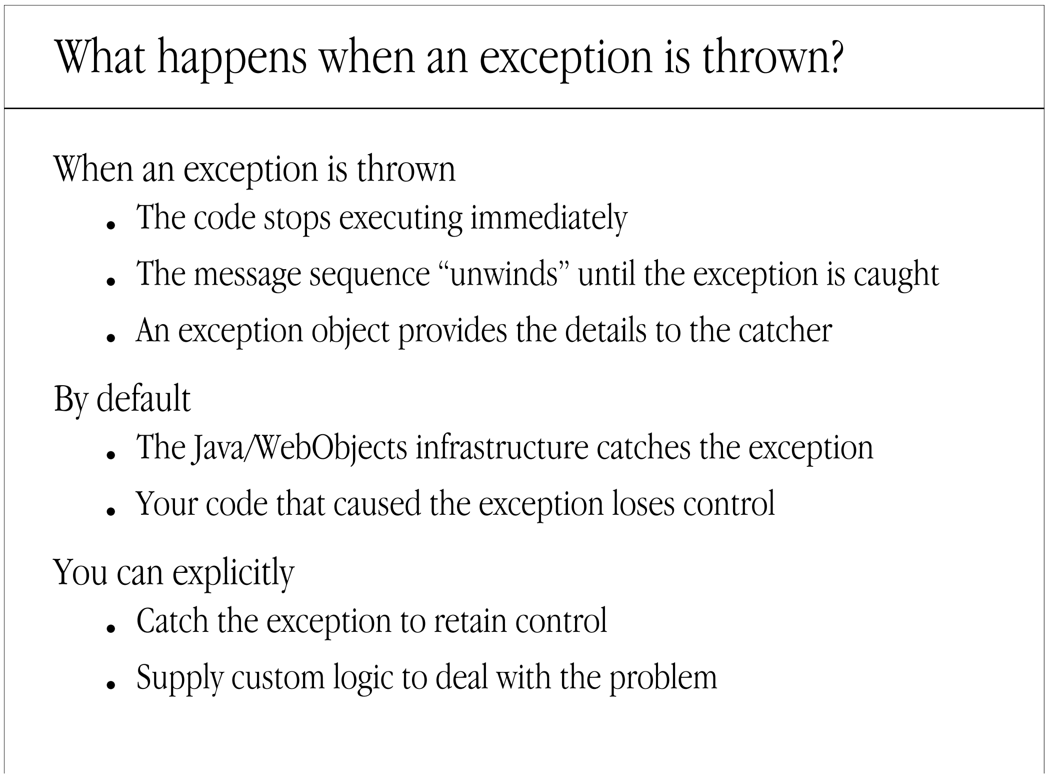


An exception object identifies the problem

The runtime problem will be represented by an object, an instance of a special exception class. Different exceptions—a null pointer, a database failure—are usually represented by instances of different Java classes. The type of exception object begins to explain exactly what happened.

Since each exception type is implemented by a specific, separate Java class, different exceptions can provide different data and behavior specific to a particular runtime problem. A credit exception may include a reference to the customer or credit card that is the focus of the problem.

All exception objects have two things in common: a message string suitable for presenting in your application's user interface, and a Java stack trace, useful for debugging the problem.



What happens when an exception is thrown?

When an exception is thrown

- The code stops executing immediately
- The message sequence “unwinds” until the exception is caught
- An exception object provides the details to the catcher

By default

- The Java/WebObjects infrastructure catches the exception
- Your code that caused the exception loses control

You can explicitly

- Catch the exception to retain control
- Supply custom logic to deal with the problem

What happens when an exception is thrown?

What really happens when an exception is thrown during the life your of application? First, your code stops executing, dead in its tracks. The exceptional condition means that the normal flow of control is somehow impossible. The code that throws the exception manufactures an exception object to record the details of the problem. Finally, the Java runtime “throws” the exception object so that it is “up for grabs” by a special part of your code that “catches” the exception to deal with it. In essence, the Java runtime now unwinds backwards through your code until it reaches the first available exception handler.

By default, the Java or WebObjects infrastructure provides the exception handling code. The default behavior typically displays a detailed stack trace showing you where exactly the code was when the exception occurred, and more importantly, how the code got there in the first place. The key fact, however, is that you lose control of the situation. Your code stops and someone else’s code takes over. Control never returns to you, at least, in the same place that you were when the problem occurred.

You can write code to explicitly catch the exception yourself. This way, you retain control of the situation, continuing to execute where you have the most information about the context in which the problem has occurred. But now you take on an important burden: you must effectively deal with the problem.

Catching an exception in your own code

To catch an exception, use the `try/catch` keywords

```
try {
  // the code that may throw an exception
   shoppingCart.checkOut();
}
catch (Exception e) {
  // your code to process the actual exception
   System.err.println("Could not check out");
   System.err.println(e.getMessage());
   return; // abort processing
}
// no exception, continue processing
```

Catching an exception in your own code

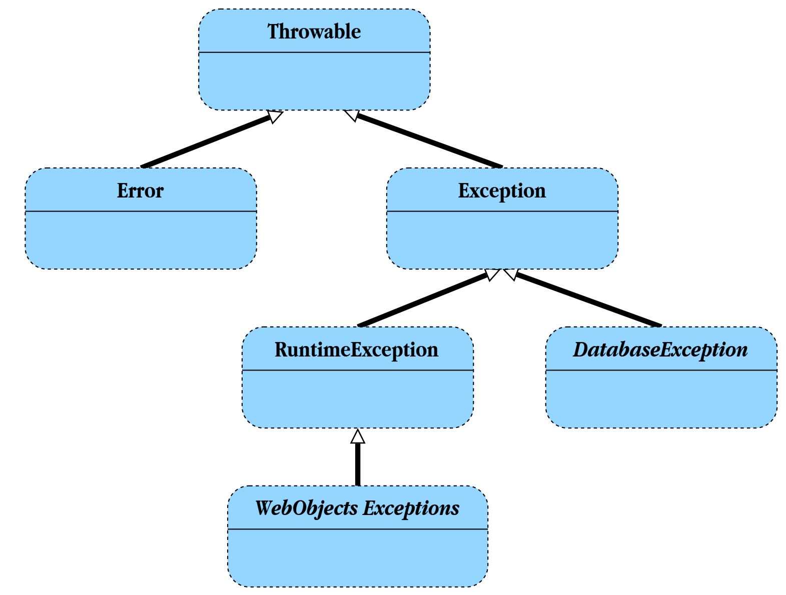
To catch an exception, use the Java keywords `try` and `catch`. First, identify the line or lines of code that may throw an exception. Nest the code inside a `try` block indicating that you wish to try to catch something that may go awry.

Next, supply a block of special code that will be activated only if the code you are trying actually throws an exception. The `catch` keyword defines the block and takes a single local argument: an exception object. Notice that the exception argument is typed to the specific class of exception you are prepared to handle. To catch just about any exception, you can use the more generic type, java.lang.Exception. The exception argument is in scope only for the duration of the `catch` block that defines it. In this sense, the `catch` block is much like a local "in line" method.

What you do inside the catch block depends on the logic of your application. You know the class of exception object from the `catch` argument. The exception object at minimum contains a message string (and a stack trace). Your `catch` code usually takes an alternate path such as returning from the method or avoiding additional processing that is carried out in the normal case.

If the code in the `try` block executes successfully without an exception, the `catch` block is ignored and the flow of control resumes with the next Java statement following the exception handler.

There are multiple exception classes



There are multiple exception classes

Each exceptional case in the Java runtime or your custom code can be represented by a specific exception class. Java defines several basic exception types in an inheritance hierarchy rooted at the abstract class java.lang.Throwable. Serious errors are subclasses of java.lang.Error and often cannot be caught at all. Catchable exceptions are subclasses of Exception. Exceptions defined by the WebObjects frameworks are usually subclasses of java.lang.RuntimeException. These have a special property, as you will see.

Most other exception classes—defined by 3rd-party packages or new custom classes that you might define—are subclasses of Exception. Our fictional DatabaseException is a good example.

The key point is that specific exception classes can be defined to represent specific runtime problems. You can implement different code for different classes of exception. Multiple exception types can be classified into sub-groups that inherit common behavior from a superclass. This enables you to be specific—to deal with is a DatabaseException—or more general—to deal with some kind of generic runtime problem.

You can handle different exceptions differently

```
try {
  shoppingCart.checkOut();
}
catch (CreditException e) {
  System.err.println("Your credit is bad");
  return;
}
catch (DatabaseException e) {
   System.err.println("Database problem");
   return;
}
System.out.println("Thanks for shopping!");
```

You can handle different exceptions differently

To handle different classes of exceptions differently, you can supply multiple `catch` blocks in your exception handler. Each `catch` block declares the specific class of exception it is prepared to deal with. When the `try` block throws an exception, Java will select the most specific `catch` block it can find. It will execute at most only one `try` block. If the exception is not covered by any of your `try` blocks—the actual exception class is not within any of the sub-groups your declared—it will be handled by the default exception handler from Java or WebObjects.

This makes sense: provide the code for those specific exception cases you can handle and ignore the others.

Exception or not, some code is always necessary

Often, some code is necessary regardless of what happens

Use the `finally` clause for code that always executes

```
try {
  shoppingCart.checkOut();
}
catch (CreditException e) {
  System.err.println("Your credit is bad");
  return;
}
finally {
  shoppingCart.setProcessingComplete(true);
}
```

Exception or not, some code is always necessary

The code within the `try` block might well succeed without throwing an exception. On the other hand, only one of many different `catch` blocks will be activated if there is an exception. Often, your logic requires some final processing in all cases—regardless of whether or not there was an exception, or regardless of the actual exception type that was thrown.

To implement this requirement, you can provide a `finally` block. The `finally` block takes no arguments (it is independent of any exception), and will be activated no matter what happens in the `try` or `catch` blocks. It is always the last piece of code to execute before the flow of control resumes.

Exception classes are often inner classes

Exceptions are often specific to a class or interface

For example, credit exceptions might only occur with shopping carts

CreditException could be defined within the ShoppingCart class

You refer to inner classes using a dot separated path name, like

```
ShoppingCart.CreditException e = new
          ShoppingCart.CreditException();
```

or

```
catch (ShoppingCart.CreditException e) {
  . . .
}
```

Exception classes are often inner classes

Java includes a feature for defining inner classes, that is, a class definition inside of another class or interface. This provides a useful encapsulation feature. Consider that a credit exception might only be thrown when interacting with a shopping cart object. In this case, it may make sense to define the CreditException class within the ShoppingCart class. It is up to the class designer.

To use an inner class, you must include the outer class (or interface) at part of the formal type name. Much like a package name, the inner class name is specified by a dot-separated path that moves from outer to inner class.

You can define your own new exception classes

Custom exceptions typically subclass `java.lang.Exception`

You can add new instance data and behavior

```
class CreditException extends Exception {

  private Customer customer;

  public Customer getCustomer() {
    return customer;
  }

  public setCustomer(Customer customer) {
    this.customer = customer;
  }

}
```

You can define your own new exception classes

What if your own custom code detects a runtime problem, an exception to the normal, successful flow? You may wish to define your own custom exception classes that you can throw.

Most custom exception classes are subclasses of java.lang.Exception. You can define a new subclass and, like any Java subclass, you can add optional data and behavior. Notice, however, that often just the new class type alone may be sufficient for a `catch` block to be specific about your particular exception.

A simple example of custom behavior might be the ability of a credit exception to reference the customer to which it applies. In the CreditException, you might add an instance variable and a pair of accessor methods.

Note that java.lang.Exception defines a one-argument constructor for creating a new exception object along with the message string. To provide the same one-argument constructor in your CreditException subclass, you must include a one-argument constructor such as:

```
public CreditException(String message) {
    super(message);
}
```

You can explicitly throw an exception

You can signal a runtime error by throwing your own exception

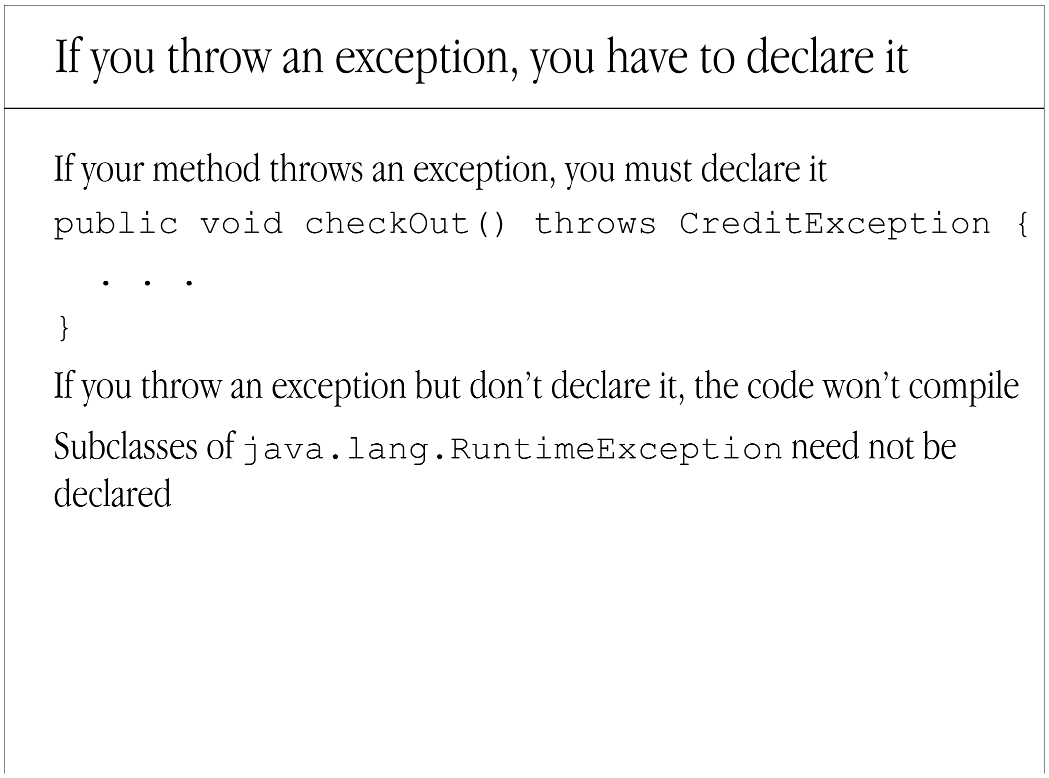
Create an exception instance, then throw it with the `throw` keyword

```
if (badCredit) {
  CreditException e =
     new CreditException("credit denied");
  e.setCustomer(currentCustomer);
  throw e;
}
```

You can explicitly throw an exception

When your custom code detects that a specific runtime problem has occurred, it can throw an exception. This aborts the flow of control and makes an exception object available to whichever `catch` block the Java runtime finds to handle it.

To throw an exception, create a new instance of the specific exception class that is most appropriate, typically a custom exception class you have defined. Initialize the exception object in any additional way the situation requires—in our example, setting the customer reference in the exception. Finally, throw the exception using the Java keyword `throw`.



If you throw an exception, you have to declare it

If your method throws an exception, you must declare it

```
public void checkOut() throws CreditException {
  . . .
}
```

If you throw an exception but don't declare it, the code won't compile

Subclasses of `java.lang.RuntimeException` need not be declared

If you throw an exception, you have to declare it

Java features checked exceptions. That is, if a Java method throws an exception, it must formally declare it in the method signature. This way, the Java complier can warn you if you call a method but do not make provisions for catching the exceptions that might result.

Use the keyword `throws` in your method declaration. If you don't, but the body of the method contains a `throw` statement, your code will not compile. If you throw multiple types of exception, specify each one in a comma-separated list.

Subclasses of java.lang.RuntimeException need not be declared—you can throw them without advertising it in your method declaration. This is usually considered bad design because it withholds important information from the client of your code. You are discouraged from using RuntimeException classes for this reason.