



Prosoft Courseware

Prosoft courseware provides a solid foundation for your training experience. Along with instructional text, coursebooks offer hands-on lab exercises and a glossary of course-specific terms. Many coursebooks also list additional reference works for continued learning outside the classroom. Experienced developers, frequently in consultation with our instructors, write all Prosoft courses. And because our courses are developed by our own staff, we can update or add new offerings as soon as new technologies enter the market.

CIW Enterprise Developer Sample Lesson

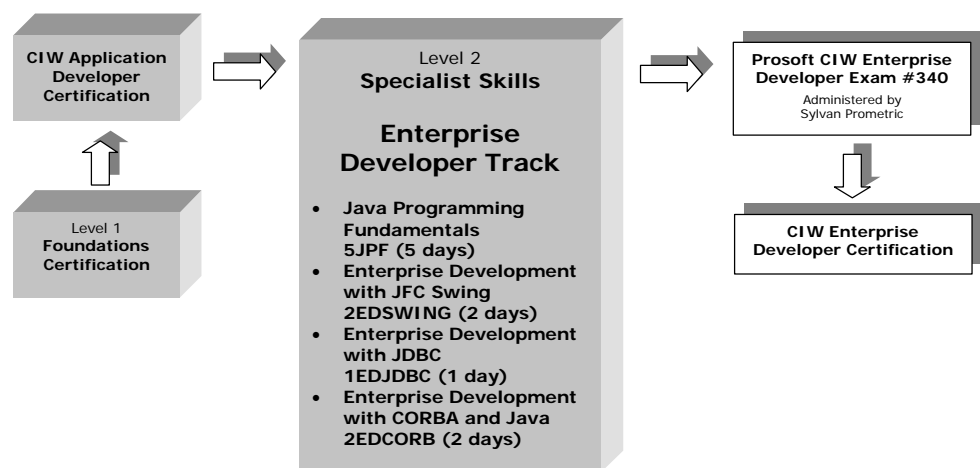
The structure of the CIW Enterprise Developer track is detailed on the next page. The following Prosoft lesson is extracted from the *Java Programming Fundamentals* course in the CIW Enterprise Developer track. This lesson demonstrates the exercise-driven content, lesson structure and straightforward, accessible narrative that is developed for every Prosoft CIW coursebook.

CIW Enterprise Developer

The CIW Enterprise Developer builds n-tier database and legacy connectivity solutions for Web applications, using Java, Java APIs, Java Database Connectivity (JDBC) solutions, middleware tools, and distributed object models such as CORBA/ORB and IIOP.

Target Audience Database developers, Internet application developers, database architects, middleware programmers, database administrators, Java developers, and client/server developers.

Job Responsibilities Develop n-tier database and legacy connectivity solutions for Web applications using Java, Java APIs, Java Database Connectivity solutions, middleware tools, and distributed object models such as CORBA/ORB and IIOP.



Certification Requirements

Prerequisites Students must have:

- CIW Foundations (exam #310) certification (or equivalent experience for students not seeking CIW certification).
- CIW Application Developer (exam #330) certification (or equivalent experience for students not seeking CIW certification).
- Familiarity with Windows NT and Web server administration concepts.
- A working knowledge of a programming language such as C, Pascal, or C++.

Training/Experience Students should take the following 10 days of Prosoft courses, or have equivalent experience, before taking the CIW Enterprise Developer (#340) exam:

- Java Programming Fundamentals (5 days)
- Enterprise Development with JFC Swing (2 days)
- Enterprise Development with JDBC (1 day)
- Enterprise Development with CORBA and Java (2 days)

Certification Award In order to become a CIW Enterprise Developer and receive the CIW Enterprise Developer certificate, students must pass the **CIW Foundations (#310)**, **CIW Application Developer (#330)**, and **CIW Enterprise Developer (#340)** exams administered by Sylvan Prometric.

Java Programming Fundamentals

Book 1

Developers

Ken Cohen and Gregg Frosti

Copyright Information

This training manual is copyrighted and all rights are reserved by ProsoftTraining.com. No part of this publication may be reproduced, transmitted, stored in a retrieval system, modified, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise without written permission of ProsoftTraining.com, 3001 Bee Caves Road, Austin, TX 78746.

Copyright © 1999 by ProsoftTraining.com
All Rights Reserved.

Notices and Disclaimer

Prosoft is a trademark of ProsoftTraining.com. All product names, Web sites and services identified throughout this book are, or contain, trademarks or registered trademarks of their respective companies. They are used throughout this book in editorial fashion only. No such references to any trademarks, service marks, trade names, domain names, Web sites, URLs or other designation is intended to convey endorsement or other affiliation with such company, product or service. Data used in examples and exercises is intended to be fictional even if actual data is used or accessed. Any resemblance to, or use of real persons or organizations should be treated as entirely coincidental. Prosoft makes a genuine attempt to ensure the accuracy and quality of the educational content described herein; however, Prosoft makes no warranty whatsoever, either express or implied, with respect to the quality, reliability, accuracy, or freedom from error of this document or the products it describes. Prosoft disclaims all liability for any direct, indirect, incidental or consequential, special or exemplary damages resulting from the use of the information in this document or from the use of any products described in this document.

Course Description

Welcome to *Java Programming Fundamentals*. This five-day course teaches you how to write Java applications and applets. You will learn the Java language mechanics found in other programming languages, such as variables, iterations, control statements, methods and arrays. You will also discuss object-oriented theory as it relates to Java. You will create Graphical User Interfaces (GUIs) for both applications and applets, emphasizing components, layouts and graphics. The course concludes with an in-depth study and implementation of the JDK 1.1 event delegation model, an essential element in further Java studies. You will complete a course-long project to create an operational client/server messaging system. This course will also prepare you for the Sun Certified Programmer Examination by providing reviews and examples relative to the exam.

Sections I and II present language essentials. Section I focuses on language constructs found in traditional procedural languages, while Section II focuses on object-oriented (OO) theory and Java's implementation of the object model. These two sections comprise an essential part of the course, and considerable time is spent mastering these topics to build a strong and necessary foundation.

This coursebook will assist you during the class and serve as a valuable resource when you return to your office or home. We will also frequently refer you to another excellent Java resource, David Flanagan's *Java in a Nutshell*. You will be experienced in writing Java applications and applets upon completion of this course.

Table of Contents

Course Description

Course Objectives

System Requirements

Conventions Used in This Book

Graphics Used in This Book

Section I Java is Just Like C

Lesson 1: Java Runtime Environment

The Java Virtual Machine

Java Comments

Sun Certification

Lesson 2: Data Types, Variables and Operators

Data Types

Variables

Operators

Sun Certification

Lesson 3: Control Statements

Code Blocks

Conditional Statements

Iteration (Loop) Statements

Sun Certification

Lesson 4: Methods

Methods

Return Statement

Calling a Method

Parameters

Pass by Value

Overloading

Sun Certification

Lesson 5: Arrays

What Is an Array?

Initializing an Array

Objects

Using an Array

Passing an Array to a Method

Garbage Collection

Command Line Parameters

Sun Certification

Table of Contents (cont'd)

Section II Java is Nothing Like C

Lesson 6: Classes and Objects

- Object-Oriented Programming
- What Is a Class?
- What Is an Object?
- Instance and Class Members
- Abstraction
- Object References
- Sun Certification

Lesson 7: Inheritance

- What Is Inheritance?
- Overriding Methods
- Sun Certification

Lesson 8: Constructors

- What Is a Constructor?
- Using Constructors
- this*
- Constructor Process
- Constructors and Callbacks
- Strings* and *StringBuffer*
- Sun Certification

Lesson 9: Interfaces and Abstract Classes

- What Is an Interface?
- Polymorphism
- What Is an Abstract Class?
- Sun Certification

Lesson 10: Packages and Access Modifiers

- Packages and Access Modifiers
- The Java API
- Information Hiding
- Encapsulation
- Sun Certification

Table of Contents (cont'd)

Section III Abstract Windowing Toolkit

Lesson 11: Swing Components

What Is the AWT?

What Is Swing?

Basic Swing Components

Sun Certification

Lesson 12: Layout Managers

What Is a Layout Manager?

Sun Certification

Lesson 13: Graphics

What Are Graphics in Java?

Graphics Class

Sun Certification

Java Programming Fundamentals - Book 1

Section II

Java Is Nothing Like C

In Section I, you learned that the syntax of Java is virtually identical to that of C and C++ as well as other languages. You also observed that much of Java can be treated as any procedural language with control loops, conditional statements, and methods. However, the similarities end there. Java is a truly object-oriented language.

"Object oriented" is a term used frequently at this point in technological history, and for good reason. Endless debates have favored procedural programming over object-oriented programming and vice versa. However, once the learning curve is overcome, most will probably agree that object-oriented (OO) programming is superior.

To fully appreciate OO, it is necessary to develop its concepts along two avenues: the theoretical and the practical. This section of the course will address both.

The theoretical discussion adheres to the key concepts of OO programming, specifically abstraction, inheritance, encapsulation and polymorphism. You will explore the practical aspects of OO by applying and practicing the theoretical aspects.

Java Technology in the Real World

The Ontario College Application Service

The Ontario College Application Service (OCAS) provides administrative and registration processing for more than 25 colleges in Canada. Its current setup uses numerous legacy systems and treats each information process as a unique operation, despite great similarity from school to school. This approach is inefficient and difficult to manage. In addition, much of the information resides on legacy systems that would be expensive and time-consuming to replace.

According to Sun Microsystems, Java is helping the OCAS resolve these problems.

The Java advantage

- A Java middle layer will add much-needed processing logic.
- Cross-platform functionality will use established legacy systems, saving time and money.
- Cross-platform functionality will also provide cross-platform database connectivity.

By implementing Java as a middle layer between the clients and the information contained on the legacy systems, a layer of logic will be added, unifying the previously incompatible information processes. As a cross-platform language, Java can also use JDBC to interface the varying database systems.

Lesson 6:

Classes and Objects

Objectives

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

- ↗ Identify the parts of an object.
- ↗ Create object references.
- ↗ Create and use instance members.
- ↗ Identify the differences between instance and class members.

Object-Oriented Programming

Object-oriented programming refers to the creation of a number of objects that communicate with each other. Each object contains data elements (variables), and the communication occurs through method calls from one object to another. In this section, we will discuss some of the principles of object-oriented programming and how Java implements those principles. You will learn about inheritance, abstraction, encapsulation and polymorphism.

What Is a Class?

Java code is always contained in either a class or an interface. Although you have already used classes, we will now discuss them in more depth.

class
The template or blueprint for an object.

Classes have many other names associated with them, such as blueprint, cookie cutter or template. This blueprint is defined by creating the necessary variables and methods within your class. Variables and methods will be explained in more detail shortly, but first you need to consider how the classes you write are used. They function as definitions for creating objects.

What Is an Object?

object
An instance of a class, with unique data and operations.

If a class is a blueprint, then an **object** is the implementation or realization of this blueprint. Imagine that you have the blueprint for a television set (your class). You cannot watch a blueprint of a television. Therefore, you need to implement this blueprint in order to create the working television set (you use the blueprint to create the object). In object-oriented terminology, you **instantiate** the television class to create a television object. In other words, an *instance* of a television object is created by instantiating the television class.

instantiation
The use of a class definition to create an object.

To transform a class into an instance of that class (an object) with Java, use the **new** keyword. For example, given that you have created a class called **Tel e vi si on. cl ass**, you can instantiate it as follows:

```
Tel e vi si on myTel e vi si on = new Tel e vi si on();
```

This code is more complex than it appears. For now, it is important to understand that you must instantiate a class using the **new** keyword to create a functional object from the class. Once you have an instance of the class, you can work with its instance members.

Instance and Class Members

instance member

A variable or method unique for each object (instance of the class).

class member

A variable or method that exists only once for the class regardless of how many objects are created; designated by the keyword `static`.

The **members** of a **class** or **instance** are the variables and methods defined in that class or instance. Although members of a class are virtually synonymous with members of an instance, their differences must be considered.

In short, class members will always have the modifier `static` in front of them. For example:

```
static int myAge = 30; // a class variable.
```

The same variable defined as an instance variable would be as follows:

```
int myAge = 30; // an instance variable.
```

The following paragraphs discuss instance and class members in detail.

Instance members

Although you already worked with class members in Section I, we will discuss instance members first because they are more commonly used. Using class members is the exception in Java; using instances of classes is more the rule.

For example, create a class structure that may represent the structure of a hospital or other medical organization. You will begin with a class called **Employee**. An **Employee** has several characteristics that you can represent as variables such as **name**, **height**, **age** and **weight**. You will also create a method that will print all the information about each **Employee**. This method consists of four **println** statements that print the instance variables. Your **Employee** class is as follows:

```
class Employee
{
    // Instance Variables
    String name; // person's name
    double height; // height in inches
    int weight; // weight in pounds
    int age; // age in years
    double salary; // salary in dollars
    int sickDays; // allotted sick days

    // Instance Method
    void printAll()
    {
        System.out.println("Name: " + name);
        System.out.println("Height: " + height);
        System.out.println("Weight: " + weight);
        System.out.println("Age: " + age);
        System.out.println("Salary: " + salary);
    }
}
```

type
Identifies the kind of information that a variable can store.

new
A keyword that signals the JVM to allocate space for an object.

You will create another class that contains your `main()` method and creates objects (instances of `Employee` class). Create two variables: `sam` and `rachael`. Each of these will be an object of **type** `Employee`. After creating the `Employee` type variables, create an `Employee` object with the **new** keyword. Like primitive variables, the value can be assigned either when the variable is declared or later in the code.

```
class Main
{
    public static void main(String[] args)
    {
        Employee sam = new Employee();
        Employee rachael;           // Instantiating a class
        rachael = new Employee();  // may be done on two
                                   // lines.
    }
}
```



You have come across an essential point in distinguishing instances from classes. In the above code, you instantiated the `Person` class twice in order to obtain `rachael` and `sam` objects. Although `rachael` and `sam` are both instances of `Employee`, each has its own copy in memory. Therefore, the `age` instance variable in `rachael` is completely independent of the `age` instance variable in `sam`. We will later show that this is not the case with class members.

Accessing instance members

dot notation

A reference used to access a member of an object.

You can assign or reference the instance members by using **dot notation** (i.e., stating the name of the object, followed by a dot, followed by the member name). You will add names and ages for **sam** and **rachael**, then print the results. You will print the instance variables from the **main()** method first, then use the **printAll()** method.

```
class Main
{
    public static void main(String[] args)
    {
        Employee sam = new Employee();
        Employee rachael;
        rachael      = new Employee();
        sam.name     = "Sam";
        sam.age      = 5;
        rachael.name = "Rachael";
        rachael.age  = 6;
        rachael.sickDays = 5;

        // Print out sam's instance variables first.
        System.out.println(sam.name + " is " +
                           sam.age + " years old");

        // Now print out rachael's information using
        // printAll()
        rachael.printAll();
    }
}
```

Notice the values for the uninitialized instance variables in **rachael**. All instance variables initialize to **0** or **0.0** for numeric types, **false** for Boolean types, **'\u0000'** for char types, and **null** for all non-primitive types. Each object (**rachael** and **sam**) has its own set of values for the instance variables. The object reference determines which instance variables are referenced.

When we discuss class members, you will see more similarities than differences. The greatest differences between class and instance members will be apparent when you run an example.

Class members

Class members function quite differently than instance members. First of all, some syntactical concerns must be noted:

- Class members (variables and methods) are modified with the keyword **static**.
- Class methods can only access class members, but instance methods can access either class or instance variables. Therefore, if you are in a **static** method, you can only have access to other **static** methods or variables.
- If you want to access a class member from another class, use either the class name or the object name.

The main difference between instance and class members is summarized in the following tech note.



*Class members maintain only one copy in memory. Therefore, even though you may have instantiated many instances of a class, each member that is declared as **static** will be shared across all instances of that class. This is similar to the concept of global variables, but the word "global" is not acceptable in OO concepts.*

Accessing class members

An excellent example for working with class members is the **Math** class in the Java API. If you want to take the sine of an angle, it is pointless to have multiple instances of the **sin()** method because there is only one sine. Thus, it is **static**. According to the syntactical points discussed above, you can access the **sin()** method (a class method) of the **Math** class in the following way:

```
double mySin = Math.sin(3.14); // By accessing the
                               // Math class
                               // directly.
```



*You do not have to instantiate the **Math** class to use a member. You can reference it directly using the name of the class.*

An effective example is to use class variables to count the number of times a class was instantiated. For example, if you want to add an `employeeNumber` to the instances of `Employee`, create a class variable.

```
class Employee
{
    // class variable
    static int numberOfEmployees = 0;

    // instance variables
    int employeeNumber;

    String name;
    double height;
    int weight;
    int age;
    double salary;
    int sickDays;

    void printAll()
    {
        System.out.println("ID:      " + employeeNumber);
        System.out.println("Name:    " + name);
        System.out.println("Height: " + height);
        System.out.println("Weight: " + weight);
        System.out.println("Age:    " + age);
        System.out.println("Salary: " + salary);
        System.out.println("Sick days: " + sickDays);
    }
}

class Main
{
    public static void main(String[] args)
    {
        Employee di = new Employee();
        di.employeeNumber = ++Employee.numberOfEmployees;

        Employee ken = new Employee();
        ken.employeeNumber = ++Employee.numberOfEmployees;

        System.out.println(di.employeeNumber);
        System.out.println(Employee.numberOfEmployees);
    }
}
```

Study the preceding code carefully. The class `Employee` maintains a class-wide copy of `numberOfEmployees`.

Abstraction

abstraction

Considering an object in terms of its functionality and not its implementing details.

Abstraction is an important concept to master when learning Java. Abstraction is a way of looking at your objects in terms of what you want them to do, rather than how they do it.

When you design classes (and ultimately complete systems), the level of sophistication involved makes it nearly impossible to consider all the details necessary to create your class or system. Therefore, you must consider your project at the level of abstraction.

Abstraction directs you to look at an object in terms of what you want it to do, instead of in terms of the methods used to implement it. Abstraction helps you put things in perspective. For example, you will write an Internet-based chat system for your project at the end of this section. Although this may seem daunting at first, abstraction helps you put the entire task into perspective. You know that there must be a client, and it needs to establish a socket connection to a server to exchange information. You do not yet know how you will accomplish this.

Abstraction can guide you in deciding that the server should probably have a method that receives clients to establish the connection (perhaps an `establishConnection()` method).

Abstraction also helps you predict the nature of Java's API. You know that you want to create a `Button` for a GUI. It seems that you should be able to specify a label for that `Button`. This process is abstraction.

Object References

Suppose you create a new `Employee` variable, `tmpEmp`, and set it equal to `rachael`. The result is shown in Figure 6-1.

```
Employee tmpEmp = rachael;
```

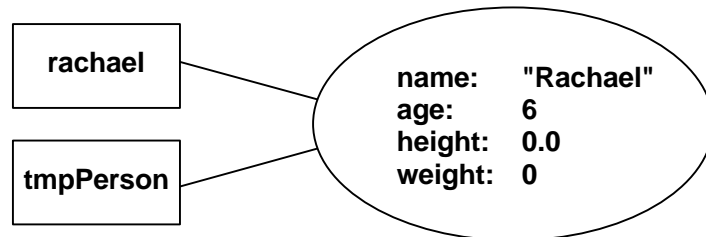


Figure 6-1: `tmpEmployee` variable

Once the object has no more `Employee` type variables pointing to it, the object is eligible for garbage collection. Garbage collection will be performed by the JVM and is a low-priority process. The programmer need not manually invoke the garbage collector.



The programmer should make no assumptions about when the garbage collector will run. It is only guaranteed to run sometime after the object no longer has a valid reference to it.



Exercise 6-1: Creating your own classes

In this exercise, you will create your own classes, instantiate them, and practice accessing their members using dot notation.

1. Create an **Employee** class with the following (minimal) instance variables (you may add more).

```
String name;  
double height;  
int weight;  
int age;  
double salary;  
int sickDays;
```

2. Add a **printAll()** method that will print the above instance variables to the command line.
3. Create the starting class and call it **SectionTwo**. Add a **main()** method. Then instantiate two instances of the **Employee** class, set their variables and print them using the **printAll()** method that you implemented.
4. (Optional) Create a two-element array that can store elements of type **Employee**. Populate the array with the two **Employee** instances created in Step 3, then attempt to invoke the **printAll()** method of each instance.
5. (Optional) Use a **for** loop to loop through the elements of the array created in Step 4, then invoke the **printAll()** method.

It is often helpful to keep the different types of members together in your class (class methods, class variables, instance variables and methods). Comments are very useful to make these separations.



*Remember that all members (class and instance, variables and methods) begin with a lower-case letter. You can distinguish between variables and methods by the presence (or absence) of parentheses. The exception to this is **final** variables (constants), which are typically all capitalized.*

Sun Certification

Static members (*Math*)

Before taking the Sun certification exam, you should memorize several `static` methods from the class `java.lang.Math`. These methods are:

```
ceil()    // Returns the next higher integer.
floor()   // Returns the next lower integer.
random()  // Returns a random double between 0 and 1.
abs()     // Returns the absolute value.
min()     // Returns the smallest of two values.
max()     // Returns the largest of two values.
round()   // Finds the closest integer to a floating-point
           // number.
sqrt()    // Returns the square root of a number.
sin()     // Returns the sine of an angle in radians.
cos()     // Returns the cosine in radians.
tan()     // Returns the tangent in radians.
```

We will discuss packages later in this section. You first need to know how to use these methods. These methods are all used in the same way, so we will focus on their use. The class `Math` has no instance methods or variables. Therefore, you will never create an instance of `Math`. For example, if you wanted to use the following method:

```
public static int max(int x, int y);
```

you would need to reference the class name (`Math`) followed by the dot and the method name. The statement would be:

```
int biggerNumber = Math.max(15, 24);
```

Garbage collection

Review the discussion of garbage collection. Automatic garbage collection is one of the key features of the Java language, and you should be thoroughly familiar with how it operates.

Initial values

Remember that the initial values for variables are only applicable for instance and class variables. Local variables (those declared inside a method) are not automatically initialized. This question could be on the exam.

Lesson Summary

In this lesson, you learned about the roles of classes and objects in object-oriented programming. You learned the differences between instance and class members, and how to access them. You also learned the value of abstraction when designing systems with Java. With this introduction to classes and objects, you have begun your journey into the world of object-oriented programming. You will use classes and objects in more complex ways in later lessons.
