Java: Object-Oriented Programming

Source: Appendix B & C, Textbook (Carrano)
Motivation

• Large software
  • Millions of lines of code, 1000s of programmer
  • Many versions over years

• Object Oriented Programming (OOP) helps
  • Divide System into modules (e.g., classes, packages)
  • Limit inter-module interactions to simplify
    • Debugging, Division of labor
  • Encourage Reuse

• OOP Concepts
  • object, method, class, package, generic data-type
  • inheritance, polymorphism
Outline

• Objects, Classes
  • Objects, References, ALiases
  • Class definition, using methods

• Defining a Java Class

• Packages

• Generic Data Types

• Inheritance, Polymorphism
Objects and Classes

- **Object** in a Java program (Ex. Figure B-1, pp. B-3)
  - A program construct to contain data fields & methods
  - Objects interact to solve a problem

- **Class** (Ex. Figure B-1, pp. B-2)
  - A general description of a family of objects
  - All instance have
    - same kinds of data & methods, but possibly different values
  - May be viewed as
    - An abstract data-type
    - A collection of objects
Using Methods

• Consider a class “Name” with
  • constructor: Name()
  • accessor methods (getters): getFirst(), getLast(), ...
  • mutator methods: (setters) setFirst(), setLast(), getFirst(), getLast(), ...

• Using methods of class Name
  Name joe;  ioe = new Name(); // Fig. B-2, pp. B-3
  joe.setFirst("joseph"); // object-name.method-name( arguments )
  joe.setLast("Brown");
  System.out.println( "Joe’s First Name is “ + joe.getFirst() );

• What values are returned by setFirst() or setLast()?
  • void
  • Other methods may return primitive types or objects!
References & Aliases

• A Java Variable
  • may hold a primitive value, e.g., int
  • may reference an object, Ex. Fig. B-2, pp. B-3

• Two variable may reference an object

  Name jamie = new Name();
  jamie.setFirst(“Jamie”); jamie.setLast(“Jones”);
  Name friend = jamie;  // Fig. B-3, pp. B-5
  jamie.setLast(“Smith”);
  System.out.println( friend.getLast() ); // Q? What will it print?
Self-Test Questions

B-1. Write Java statement to create an object of type Name to represent your name.

B-2. Write java statements to print your names using the object created in last question.

B-3. Suggest likely return type for methods in class Automobile (Figure B-1, pp. B-2).

B-4. Is setFirst an accessor method?

B-5. Should a typical accessor method return void?

B-6. Should a typical modifier method return void?
Outline

• Objects and Classes

• Defining a Java Class
  • Class & Method Definition
  • Constructors
  • Static Fields and Methods
  • Overloading Methods

• Packages

• Generic Data Types

• Inheritance, Polymorphism
Defining Class

// File: Name.java

public class Name {
    private String first;
    private String last;
    <definition of methods setFirst, setLast, getFirst, getLast, ...>
}

• Access/Visibility Modifiers
  • public – can be used anywhere, e.g., inside other classes
  • private – not available inside other classes!
  • protected  (Appendix C: visible in derived classes, or in same package)

• Convention
  • Data members of a class are usually private. Q? Why?
  • Selected methods of a class are public
  • This simplifies debugging!
Defining Method

// access-modifier use-modifier return-type methodname ( parameters) {body}
public String getFirst { return first; }  
public void setFirst ( String FirstName ) { first = firstName ; }

• Access-modifier, e.g., public, private, ...

• Use-modifier
  • abstract - no definition, i.e., to be defined in derived classes
  • final - can not be redefined by a derived class
  • static - shared across objects in this class

• Return-type, e.g., void, int, String, ...

• Parameters – list of <parameter-name, parameter-type> pairs
  • Leading Question? How many parameters are there in getFirst?
  • Note an implicit parameter, i.e., the object on which a method is called!

• Body – a sequence of statements
B.10 The object “this”

```java
public String getFirst { return this.first; }
public void setFirst ( String FirstName ) { this.first = firstName ; }
public  boolean sameFirst ( Name otherName ) {
        return ( this.first.equals( otherName.first ) );
}
```

• A trivial use case
  • “this” may be used to disambiguate “first” across different objects!

• Better Use Case
  • A method needs to refer to two objects from same class
  • Second object is referred to via a parameter name
  • “this” refers to the first object (on which method is called.)

• More on methods
  • A method may define local variables
  • Names of parameter and local variable are visible within a method
Arguments and Parameters

• Terminology
  • Formal parameters: variables named in method header
  • Arguments: variables (or values) passed by caller

• Passing Arguments
  • Primitive types: call by value
    • Only input to method
    • If method modifies the parameter, it does not affect argument
  • Objects: call by reference
    • Input to method
    • Output from method, i.e., modify data member of objects referred to by arguments!
    • However, it can make arguments refer to different objects!
public class Name {
    private String first;    private String last;
    private char initial;
    ...
    public void setMiddleInitial( char middleInitial ) { initial = middleInitial; }
    ...
}
...
// in a client
char joesMI = ’T’ ;
Name joe = new Name();
joe.setMiddleInitial( joesMI ) ; // Q? Can setMiddleInitial() change joesMI ?

// Execution trace shown in Figure B-4, pp. B-10
Arguments: Call by Reference

```java
public class Name {
    private String first;    private String last;
    ...
    public void giveLastNameTo( Name child ) { child.setLast = this.last; }
    ...
    public static void main (String[] args) {
        Name jamie = new Name();
        jamie.setFirst("Jamie"); jamie.setLast("Jones");
        Name jane = new Name();
        jane.setFirst("Jane"); jane.setLast("Doe");
        jamie.giveLastNameTo( jane ); // can giveLastNameTo() change jane?
        ...
    }
}
// Execution trace shown in Figure B-5, pp. B-12
```
Self-Test Questions

B-8. Consider: public void process(int number, Name aName)
    And its invocation: someObject.process(5, jamie);
    What are values of “number” and “aName” within method body?

B-9. Can method “process” change data fields in “jamie”?

B-10. Can method “process” assign a new object to “jamie”?

Exercises: Review complete definition of class Name in Listing B-1 on page B-13 and B-14 to answer the following questions:

A. How many accessor methods does it have?
B. How many mutator methods are there?
C. Suggest a use for toString() method.
D. Which methods do not specify return type (not even void)? Why?
Outline

• Objects and Classes

• Defining a Java Class
  • Method Definition, Arguments, Parameters
  • Constructors, Overloading
  • Static Fields and Methods

• Packages

• Generic Data Types

• Inheritance, Polymorphism
Constructors

• Conventions
  • Method has same name as the class
  • Has no return types, not even void!
  • If you do not define any, Java gives a default constructor with no parameters

• Use case 1: allocate memory for an object
  • Figure B-7 (pp. B-16) for trace of the following:
    Name jill = new Name(“Jill”, “Jones”); // allocate memory
    Jill = new Name(“Jill”, “Smith”);       // allocate different memory.
    // Q? What happens to previous memory referred to by jill?

• Use Case 2: initialize values of data members
  • Leverage Default for primitive types: null for references, zero for numbers, ...
  • Must initialize all reference type data members to avoid run-time errors!

• Leading Q? How many constructor may a class have?
  • many as long as they differ in number of (or type of) parameters!
Overloading Methods

• Overload
  • Several methods in a class have same name!
  • They must differ in number of or data-type of parameters
  • Not sufficient to differ in return type or parameter-names!

• Example 1: constructors in Name class

• Example 2:
  
  ```java
  public void setName(String firstname, String lastName);
  public void setName(Name otherName);
  public void setName(String firstName, Name otherName);
  ```

• Leading Q? It is nice for methods to share names.
  • Can methods or data-members be shared among objects in a common class?
Outline

• Objects and Classes

• Defining a Java Class
  • Method Definition, Arguments, Parameters
  • Constructors, Overloading
  • Static Fields and Methods

• Packages

• Generic Data Types

• Inheritance, Polymorphism
“Static” Fields & Methods

• Static fields
  • class properties, e.g., cardinality, listOfAllInstances, ...
  • constant values, (e.g., pi), which do not need to be replicated across objects
  • Examples: public static int numberOfInstance;
    public static final double YARD_PER_METER = 1.0936;

• Static methods
  • A method that does not belong to individual objects
  • Example: compute class statistics, e.g. average, standardDeviation()
  • Example: main(), testRoutine(), Math.max(), Math.sqrt(), ...

• Restriction on body of static methods
  • Directly access static fields and other static methods
  • Create an local object to invoke non-static methods
  • A constructors can not be a static method
Self-Test Questions

B-11. What is a default constructor?

B-12. How do you invoke a constructor?

B-13. What happens if you do not define a constructor for a class?

B-14. What happens if you define a constructor with parameters but no default constructor?

B-15. What happens when an object no longer has a variable that references it?

B-17. What happens if a constant data field is not declared static?

B-18. Can two methods with same name differ in return types?
Outline

• Objects and Classes
• Defining a Java Class
• Packages
  • Java Class Library
• Generic Data Types
• Inheritance, Polymorphism
Packages

• Purpose
  • Group several related classes together as a unit
  • Example: java.util, java.lang, ...
  • Similar to a library

• Syntax
  package packageName; // place at start of a file
  import packageName.* ; // use public classes of a package within a program

• Package Access
  • If a class, method or field does not specify access level (e.g., private, public)
  • It is available within the package
Outline

• Objects and Classes
• Defining a Java Class
• Packages
• Generic Data Types
• Inheritance, Polymorphism
Generic Data Types

• Generic Data Type
  • A placeholder instead of an actual type in definition of a class or interface
  • Generics are better than “Object” by providing restrictions
  • Examples: public class MyClass<T>; // definition header
    MyClass<String> item = new MyClass<String>(); // use with a reference type
    MyClass<int> item = new MyClass<int>(); // Illegal with a primitive type!

• Example: OrderedPair<S,T>
  • Review the code on page B-27
  • Exercise: Can it create a pair with two distinct types? Why or why not?
  • Example: main(), testRoutine(), Math.max(), Math.sqrt(), ...

• Restriction on <>
  • <T> follows identifier name in the class header
  • <T> does not follow constructor names in their definitions
  • T (not <T>) can be used to type variables, parameters or method return value
Generic Data Types: Example

```java
public class OrderedPair<T> {
    private T first, second;
    public OrderedPair(T fName, T sName) { first = fName; second = sName; }
    public T getFirst() { return first; }
    public T getSecond() { return second; }
    public String toString() { return "(" + first + ", " + second + ")" ; }
    public void changeOrder() { T temp = first; first = second; second = temp; }
}

OrderPair<String> fruit = new OrderedPair<String>("apples", "oranges");
System.out.println(fruit); fruit.changeOrder(); System.out.println(fruit);
...
Name n1 = new Name("Tweedle", "Dee"); Name n2 = new Name("Tweedle", "Dum");
OrderedPair<Name> namePair = new OrderedPair<Name>(n1, n2);
System.out.println(namePair); fruit.changeOrder(); System.out.println(namePair);

// Q? Can one create a OrderedPair with distinct data-type, e.g., String, Name ? Why?
```
public class Pair<S, T> {
    private S first; private T second;
    public OrderedPair(T fItem, S sItem) { first = fItem; second = sItem; }
    public T getFirst() { return first; }
    public S getSecond() { return second; }
    public String toString() { return "(" + first + ", " + second + ")"; }
}

Name n1 = new Name("Joe", "Java");
String joePhone = "(401) 555-1234";
Pair<Name, String> joeEntry = new Pair<Name, String>(joe, joePhone);
System.out.println(joeEntry);
Outline

• Objects and Classes
• Defining a Java Class
• Packages
• Generic Data Types
• Inheritance, Polymorphism
Motivation

• Large software
  • Millions of lines of code, 1000s of programmer
  • Many versions over years

• Object Oriented Programming (OOP) helps
  • Divide System into modules (e.g., classes, packages)
  • Limit inter-module interactions to simplify
    • Debugging, Division of labor
  • Encourage reuse

• OOP Concepts
  • object, method, class, package, generic data-type
  • Composition, inheritance, polymorphism
Reuse by Composition

• A class uses composition if it
  • has objects as data fields, Ex. Figure C-1 (pp. C-2)
  • manipulates those data field objects via their methods
• Listing C-1 (pp. C-3)

```java
public class Student {
    private Name fullName; // Recall Name class from Appendix B (pp. B-13, B-14)
    private String id;
    public Student() { fullName = new Name(); }
    public Student(Name sName, String sId) { fullName = sName; id = sId; }
    public void setStudent(Name sName, String sId) { this.setName(sName); setId(sId); }
    public void setName(Name sName) { fullName = sName; }
    public Name getName() { return fullName; }
    public void setId(String sId) { id = sId; }
    public String getId() { return sId; }
    public String toString() { return id + " " + fullName.toString(); }
}
```
Reuse by Inheritance

• What is inheritance?
  • Biological analogy: child inherits traits (e.g., hair color) from parent
  • Use Case: a classification system, Ex. Figure C-2 (pp. C-6)
  • Common properties and methods defined once in super-class,
    • these are reused by specialized classes via inheritance of
    • data fields and non-constructor methods from its super-class

• Ex. Listing C-2 (pp. C-8, C-9)
  public class CollegeStudent extends Student {
    private int year; // year of graduation
    private String degree; // degree sought
    public CollegeStudent() { super(); year = 0; degree = ""; } // super(); is not needed. Why?
    public CollegeStudent(Name sName, String sId, int y; String d) {
      super(sName, sId ); year = y; degree = d; } // call constructor of super-class
    public void setStudent(Name sName, String sId, int y, String d) {
      this.setName(sName); setId(sId); year = y; degree = d; }
    // … methods setYear, getYear, setDegree, getDegree go here
    public String toString() { return super.toString() + "", " + degree + ", " + year ; }
  }
Reuse by Polymorphism

• Overloading: a method name used in a class and its super-class
  • Ex. Figure C-6, pp. C-22: Three definitions for method display()

• Dynamic Binding: Run-time disambiguation for a method invocation
  public class Student {
    public display() {
    }
    public displayAt(int numberOfLiines) {
    } // Which display() will it call?
  }
  public class CollegeStudent extends Student {
  }
  public class UndergradStudent extends CollegeStudent {
  }
  public class client {
    public static void main(String[] args)
    {
      UndergradStudent ug = new UndergradStudent();
      ug.displayAt(2); // Call 1
      CollegeStudent c = new CollegeStudent();
      c.displayAt(2); // Call 2
      Student s = ug; s.displayAt(2); // Call 3
      Student s = s.displayAt(2); // Call 4
    }
  }

Exercise: For each call of displayAt(), determine which display() method is invoked?

Hint: see Figure C-8, pp. C-24.
Self-Test Questions

1. Which of the following declarations are legal for Figure C-3 (pp. C-7)?
   
   ```java
   Student a = new CollegeStudent();
   Student b = new UndergradStudent();
   CollegeStudent c = new Student();
   UndergradStudent u = new Student();
   ```

C-6. Some vehicles have wheels and some do not. Revise Figure C-2 (pp. C-6) to organize vehicles according to whether they have wheels.

C-8. Are the two definitions of the constructors for the class Student (Listing C-1, pp. C-3) an example of overloading or overriding? Why?

C-9. If you add the method “public void setStudent(Name studentName, String studentId)” to the class CollegeStudent and let it give some default values to the fields year and degree, are you overloading or overriding setStudent? Why?

C-15. Is overloading a method name an example of polymorphism?

C-16. Will two invocation of displayAt() produce same output?

   ```java
   Student s = new UndergradStudent(...); s.displayAt(2);
   s = new GradStudent(...); s.displayAt(2);
   ```