Contents

1. Classes and Objects
2. Inheritance
3. Interfaces
4. Exceptions and Error Handling
5. Intro to Concurrency
6. Concurrency in Java
7. Graphics and Animation
8. Applets
Chapter 1: Classes and Objects

After this chapter you will be able to:

- Write class definitions
- Create and initialize objects
- Define and call methods
- Use keywords public and private and static
Key to Symbols

Crystal ball - indicates a topic that is covered in more detail later in the course

Elephant - indicates an important point not to forget

Bug - indicates an erroneous program

Check - indicates a correct program

Thumb down - indicates a bad practice
OO Terms

• A class definition is a specification describing the data and operations associated with objects

• An object is an instance of some class
Example

Class Person

Name:
Age:

Objects

Name: Ed
Age: 24

Name: Alice
Age: 30
Benefits of OOP

- **Encapsulation**: an object encapsulates data and the associated operations

- **Abstraction**: can think about objects without looking at internal details

- **Well-defined interfaces to data**: OOP prevents attempting to apply operations to the wrong kind of data
Class Components

• A class definition may consist of:
  – instance variables
  – class variables
  – instance methods
  – class methods
  – class constructors

• We’ll look at each in turn
Instance Variables

- Instance variables are variables declared at the top level of a class definition

```java
class Account {
    int balance;
    String owner;
}
```
Initialization

- Instance variables declarations may also initialize

```java
class Account {
    int balance = 0;
    String owner = "nobody";
}
```
Making Objects

• Class definition creates a user-defined datatype
• An object can be bound to an object variable

```java
Account a1 = new Account();
```

• a1 stores the address of an Account object

balance: 0
owner: "nobody"
Accessing Instance Variables

- Syntax: `object.variable`

```java
Account a1 = new Account();
a1.owner = "george";
a1.balance = 200;

System.out.println("owner = " + a1.owner);
System.out.println("balance = " + a1.balance);
```

Output

```
Owner = george
balance = 200
```
Object Storage

• Each object has its own storage for instance variables

```java
Account a1 = new Account();
a1.owner = "george";
a1.balance = 200;

Account a2 = new Account();
a2.owner = "sally";
a2.balance = 300;
```

balance: 200  owner:  "george"

balance: 3  owner:  "sally"
Instance Methods

- A method is a procedure defined inside a class
- An instance method is called via an object

```java
class Account {
    int balance = 0;
    String owner = "nobody";

    void deposit (int d) {
        balance += d;
    }
}
```
Method Parts

• All methods have a return type and name
• Parameters are optional

```java
class Account {
    int balance = 0;
    String owner = "nobody";

    void deposit (int d) {
        balance += d;
    }
}
```
Calling Instance Methods

• Syntax: `object.method (args)`

```java
Account a1 = new Account();
a1.balance = 200;
System.out.println("balance = "+a1.balance);

a1.deposit(20);
System.out.println("balance = "+a1.balance)
```

**Output**

```
balance = 200
balance = 220
```
Complete Example

- File: Main.java

```java
public class Main {
    public static void main (String [] args) {
        Account a1 = new Account();
        a1.owner = "fred";
        a1.balance = 100;
        a1.deposit(20);
        a1.print();
    }
}

class Account .../* continued on next page */
```
Complete Example Cont

• Only one public class per file

```java
class Account {
    int balance = 0;
    String owner = "nobody";

    void deposit (int d) {
        balance += d;
    }
    void print () {
        System.out.print("owner = " + owner + " balance = " + balance);
    }
}
```

Overloading

• Can overload a method name by providing multiple method definitions
• There must be a difference in either the number of arguments or in types of the arguments

```java
class Account {
    int balance = 0;
    String owner = "nobody";

    void deposit (int d) {
        balance += d;
    }
    void deposit (String s) {
        balance += Integer.parseInt(s);
    }
    ...
}
```
Invalid Overloading

- Cannot overload based on only distinguishing the return type

```java
class Foo
    int bar () {
        return 4;
    }
    float bar () {
        return 4.4;
    }
}
```

- Compile-time error: cannot resolve overloading
Controlling Access

- Instance variables and methods may be declared as:
  - private: access only allowed from inside class
  - public: access allowed from anywhere in the program
  - default: (no specifier) access allowed from within the same package as the class
Private Access

- Protect the internal components with `private`

```java
class Account {
    private int balance = 0;
    private String owner = "nobody";

    void deposit (float d) {
        balance += d;
    }

    void print () {
        System.out.print("owner = " + owner +
                        " balance = " + balance);
    }
}
```
Private

- Cannot access balance outside of class Account

```java
Account a1 = new Account();
a1.balance = 200;
```
Public Allows Access

- Make the client interface public

```java
class Account {
    private int balance = 0;
    private String owner = "nobody";

    public void deposit (float d) {
        balance += d;
    }

    public void print () {
        System.out.print("owner = " + owner +
                        " balance = " + balance);
    }
}
```
Bad Practice

• Don’t expose the internals by using public
• This allows random uncontrolled access to the internals of an object

```java
class Account {
    public int balance = 0;
    public String owner = "nobody";
}
```
Problem

• Now we cannot even look at the balance and owner from outside

• balance is private and cannot be directly accessed

• owner is private and cannot be directly accessed
Solution

• Provide accessor methods which are part of the public interface

```java
class Account {
    private int balance = 0;
    private String owner = "nobody";

    public int getBalance () {
        return balance;
    }
    public String getOwner () {
        return owner;
    }
    ...
}
```
Another Problem

- What if we want to initialize an account object with a specified balance and name?

- balance is private and cannot be directly accessed
- owner is private and cannot be directly accessed
Solution

- Incorporate a *class constructor*
- Constructors are special methods which have the same name as the class
- Constructors are invoked via `new` as part of creating an object

```java
Account a1 = new Account(500, "ed");
```
Defining Constructors

• Constructors are methods having the same name as the class and no return type

class Account {
    private int balance = 0;
    private String owner = "nobody";

    public Account (int b, String o) {
        balance = b;
        owner = o;
    }

    ...
}
Default Constructor

• A default constructor has no arguments

```java
class Account {
    private int balance;
    private String owner;

    public Account () {
        balance = 0;
        owner = "nobody";
    }
    ...
}
```

• Note: instance variable initializers are executed before the constructor body
Question

• Earlier we wrote this:

```java
Account a1 = new Account();
```

• before adding a default constructor

• What constructor was being called?

• Answer: A default constructor with an empty body is implicitly provided for free*

  * = except when you define any class constructor yourself
Example

- One (non-default) constructor defined

```java
class Account {
    private int balance;
    private String owner;

    public Account (int b, String o) {
        balance = b;
        owner = o;
    }

    public void deposit (float d) {
        balance += d;
    }

    public void print () {
        System.out.print("owner = " + owner +
                           " balance = " + balance);
    }
}
```
Example continued

• No default constructor available

```java
Account a1 = new Account();
```

• Compile-time error!
End of Part I
It’s Exercise Time
Part II
Calling Other Constructors

- Constructors can call other constructors using keyword this

```java
class Account {
    private int balance;
    private String owner;

    public Account (int b, String o) {
        balance = b;
        owner = o;
    }
    public Account () {
        this(0,"nobody"); /* must be first statement */
    }
    ...
}
```
More About this

• Keyword this can also be used inside a method to refer to the object used to call the method
• Every method has an implicit parameter - the object used to call the method
• The keyword this is how we refer to the implicit parameter
Example 1

• Not necessary here - but we can qualify instance variable references

```java
class Account {
    private int balance;
    private String owner;

    public Account () {
        this.balance = 0;
        this.owner = "nobody";
    }
    public Account () {
        this(0);
    }
    public Account () {
        this(0);
    }
    ...
}
```
Example 2

- It is necessary to use `this` here since the parameters `shadow/hide` the instance variables.

```java
class Account {
    private int balance;
    private String owner;

    public Account (int balance, String owner) {
        this.balance = balance;
        this.owner = owner;
    }
    public Account () {
        this.balance = 0;
        this.owner = "nobody";
    }
    ...
}
```
Example 3

- Can test by comparing the addresses of two objects using this

```java
class Account {
    private int balance;
    private String owner;

    public boolean equal (Account other) {
        if (this == other) /* test addresses */
            return true;
        else if ((owner.equals(other.getOwner())) &&
            (balance == other.getBalance()))
            return true;
        else
            return false;
    }

    ...
}
```
Example 3 continued

• Method equal is passed two objects

```java
Account a1 = new Account(100, "fred");
Account a2 = new Account(200, "sally");
a1.equal(a2);
```

• `a1` is the *implicit* argument (referenced by *this* inside `equal`)

• `a2` is the *explicit* argument (referenced by *other*)
Class Variable

• A class variable lives with the class as opposed to the object

```
Class Person
numPersons: 2
```

Objects

```
Name: Ed
Age: 24
```
```
Name: Alice
Age: 30
```

• Only one copy of numPersons
Declaring Class Variables

• Use keyword `static`

```java
class Account {
    private int balance;
    private String owner;
    private static int numAccounts = 0;

    public Account (int b, String o) {
        numAccounts++;
        balance = b;
        owner = o;
    }

    public int getNumAccounts () {return numAccounts;}
    ...
}
```
Account a1 = new Account(100, "fred");
Account a2 = new Account(200, "sally");

System.out.println("# Accounts =" + a1.getNumAccounts());
Class Methods

- Class methods may be called independently of any object
- Use keyword `static` in method declaration

```java
class Account {
    private int balance;
    private String owner;
    private static int numAccounts = 0;

    public static Account[] make2Accounts() {
        Account[] list = {new Account(), new Account()};
        return list;
    }
    ...
}
```
Calling Class Methods

• Syntax1: `className.method (args)`

```java
Account [] twoAccouts = Account.make2Accounts();
```

• Syntax2: `object.method (args)`

```java
Account [] a = new Account();
Account [] twoAccouts = a.make2Accounts();
```
public class Main {
    public static void main (String [] args) {
        Account a1 = new Account();
        a1.owner = "fred";
        a1.balance = 100;
        a1.deposit(20);
        a1.print();
    }
}

class Account .../* continued on next page */
Final Version continued

• Can you identify the different components of class Account?

class Account {
    private int balance;
    private String owner;
    private static int numAccounts = 0;

    public Account (int b, String o) {
        numAccounts++;
        balance = b;
        owner = o;
    }

    public Account () {
        this(0,"nobody");
    }

    ...
}
void deposit (int d) {
    balance += d;
}
void deposit (String s) {
    balance += Integer.parseInt(s);
}
public int getBalance () {
    return balance;
}
public String getOwner () {
    return owner;
}
public static Account [] make2Accounts () {
    Account [] list = {new Account(), new Account()};
    return list;
}
} /* end class Account */
Review of Packages

- Packages help to prevent name conflicts in large applications
- Ada (1983) was the first language to support packages
- In Java every package corresponds to a directory in the file system
Creating A Package

1. Create a directory for the package

2. Install corresponding .class or .java files using `package` statement at top of source file

3. Add directory path to list of directories in `CLASSPATH` environment variable

• Java Compiler uses `CLASSPATH` to find the contents of packages
Packages

- In the Java API (Application Programmer Interface) classes are grouped into *packages*
Package Visibility

• All classes in package java.lang are visible by default

• Classes in other packages can be made visible using the import statement

```java
import java.util.Vector;    // import only Vector
/* note: file for class Vector is in directory
util which is a subdirectory of
directory java which is in CLASSPATH */
import java.util.*;         // import all classes
// in java.util
```

• import is always at the top of a file
Package account

• Keyword `package` appears at the top to place class `Account` in package `account`

```java
package account;    /* file: Account.java */

public class Account {
    private int balance;
    private String owner;
    private static int numAccounts = 0;

    ...  
```

• `Account.java` should be in directory `account`

• Path for `account` is in `CLASSPATH`
It’s Exercise Time