OO Programming with Java

Part I

Objectives

At the end of this module you should be able to

- Understand the benefits & concepts of Object Oriented Programming (OOP)
- Appreciate the basic structure of an OO program
- Understand the relationship between objects, types and class definitions
- Work with the DI Banking Application

What is Object Oriented Programming (OOP)?

- Implementation of OO Analysis & Design
  - OOA&D is natural way of solving a complex problem
  - OOA&D is based on OO Methodology
- OO Methodology has been around since the 1960s
  - Abstraction
  - Encapsulation - also known as “data hiding”
  - Inheritance
  - Polymorphism - multiple behaviors
- OOP was used more as the complexity of problems increased
- Typical areas that could benefit from OOP
  - Air traffic control systems,
  - Smart missiles and battlefield robots,
  - Artificial intelligence applications A collection of APIs

Agented Systems

- Object oriented programs are sometimes classified as “agented systems”
- An agented system is comprised of
  - Collections of autonomous and self-contained processing agents - Objects
  - Collaborations between agents that accomplish the system's objectives - Associations and compositions
  - Coordinated actions through exchanging messages - Method invocations

Example of Agented Systems

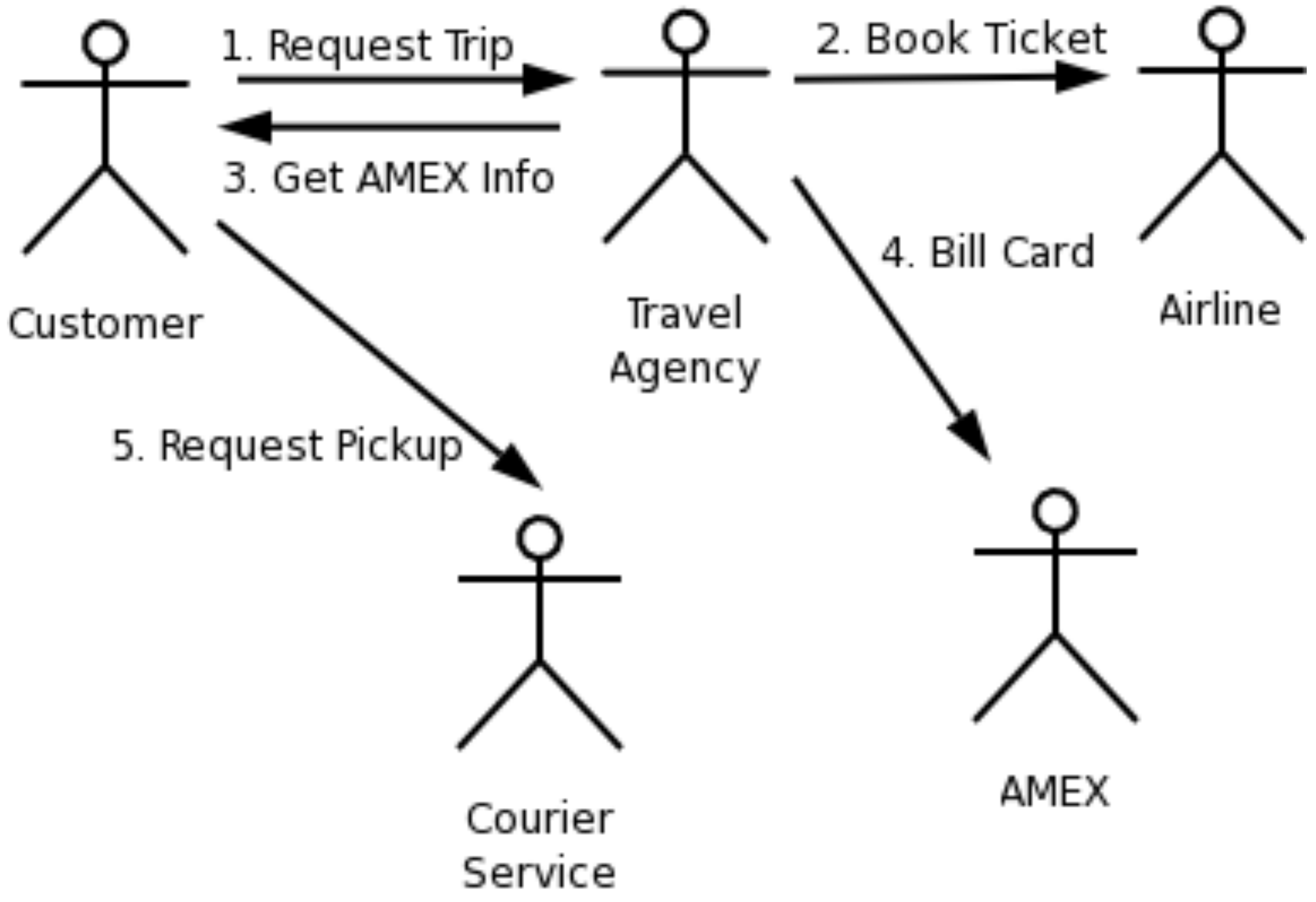


Fig. 2-1: **An Agented System – getting a ticket to Paris**

Agented Systems => OOP

- The challenge in OOP is to write programs that automates agented activity
- OOP does this by creating software *objects*, each object
  - Represents an agent
  - Possibly runs on its own processor
  - Possibly operates autonomously from the other objects
  - Possibly operates concurrently with the other objects

Agented Systems => OOP

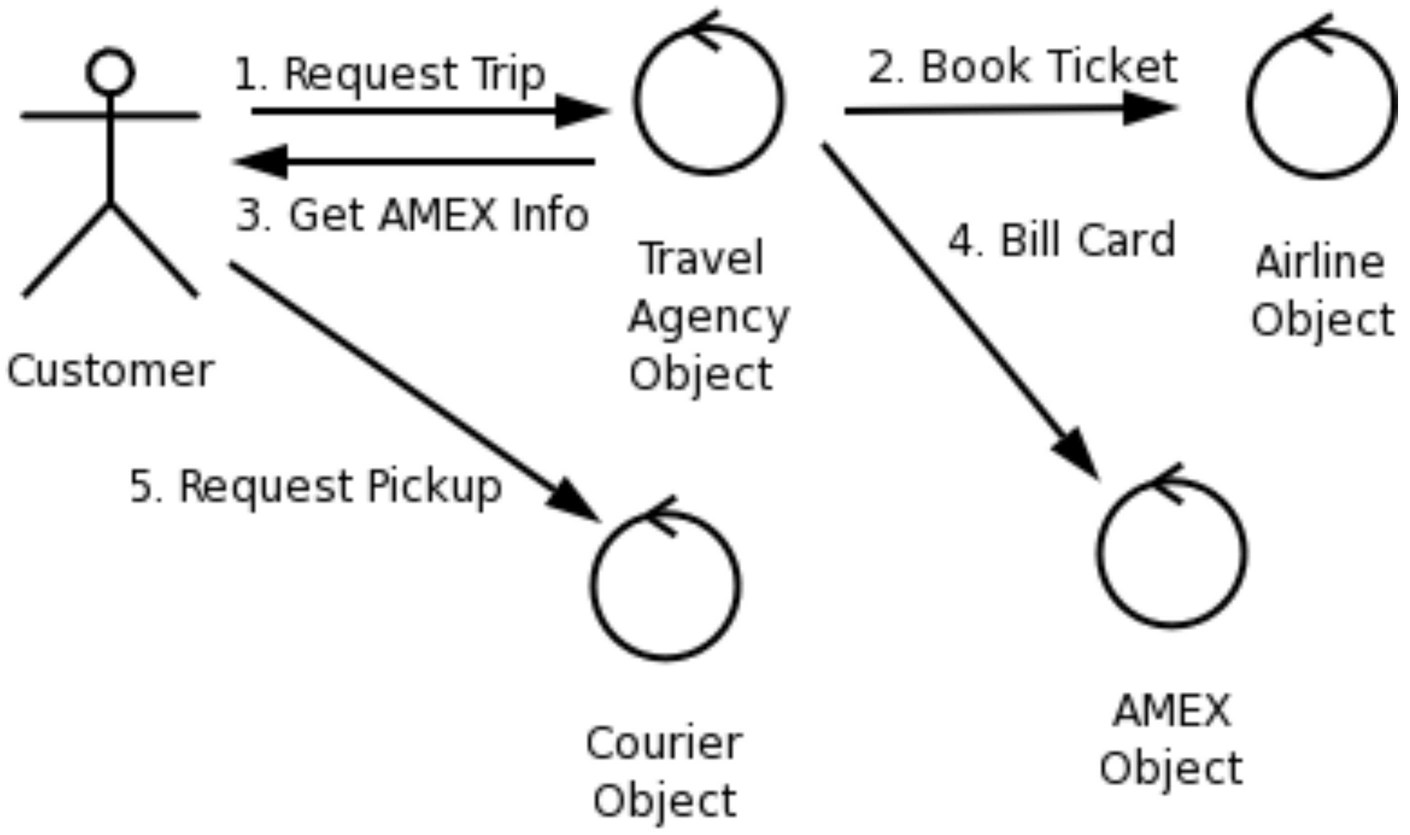


Fig. 2-2: **The Agented System – Now Automated**

Fundamental OOP principles

- *The Principle of Recursive Design*
  - *The structure of the part **mirrors** the structure of the whole*
    - Dividing a system (or computer) up into a collection of small processing engines, called objects
    - Each object will have similar computational power as the whole
    - Processing happens when the objects work together
  - However, the recursive part is
    - Each object in turn is a collection of sub-objects
    - Each with similar computational power to the containing object
    - And so on . . .

Recursive Design

- Recursive design is at work in the ticket purchase system
- The system is made up of a collection of objects, one of which is an "Airline" object
- An Airline object is made up of a collection of objects
  - Each delivers part of the overall functionality of the Airline object
  - Ticketing object is responsible for doing ticketing
  - Ticketing object is made up of objects, each of which specializes in some aspect of the Ticketing object's functionality

Recursive Design

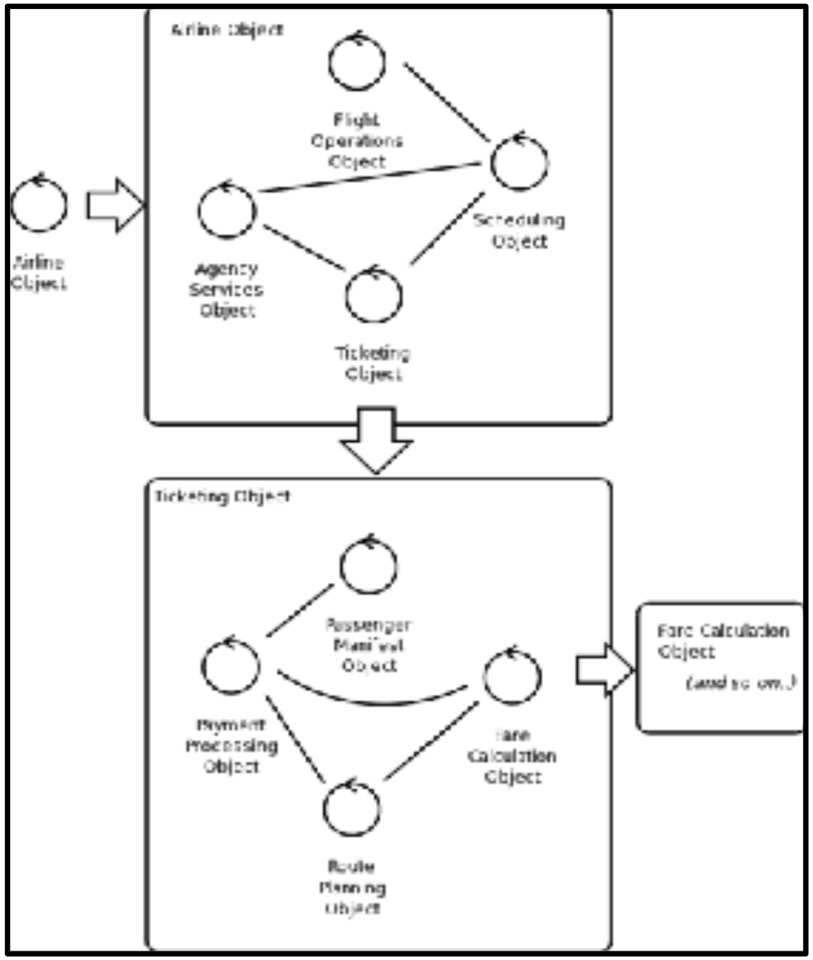


Fig. 2-3: **Principle of Recursive Resign – Airline Object**

The OOP Principles

- Everything can be represented as an object
  - Objects are what we talk or think about
  - When we want to talk about something, we make it into an object
  - For example AMEX views the billing of the AMEX card for the flight to Paris as a credit card transaction - an object
- Systems are collections of objects collaborating for a purpose
- An object can have an internal structure composed of hierarchies of sub-objects

The OOP Principles (cont.)

- An object presents an interface that
  - Specifies which requests can be made of it and what the results of those requests are
  - Is often referred to as the object's "contract”
- An object is of one or more types
  - Each type defines an interface
  - Each type may be derived from a hierarchy of types

Where do Objects come from?

- OO programs are made up of objects
- Programmers develop code that serves as the templates for those objects
  - Source code is compiled into some “executable”
  - These templates are called *classes*
- Object behavior (or functionality) is defined by its *type*
  - A type is described by a class definition
  - A class definition can be thought of as an object’s job description
- Each object in an OO program is of at least one type
  - An object is completely described by the class definitions associated with its types
  - Objects can have multiple types, therefore, multiple job descriptions

Where do Objects come from? (cont.)

- To write an OO program
  - Create a class description for each type of object in our system
  - Define the interactions between the various types
- To run a program
  - We create a collection of objects using the class descriptions as templates
  - Objects created this way are called *instances* of the class
  - We say we have *instantiated* the objects from the class definitions

Where do Objects come from? (cont.)

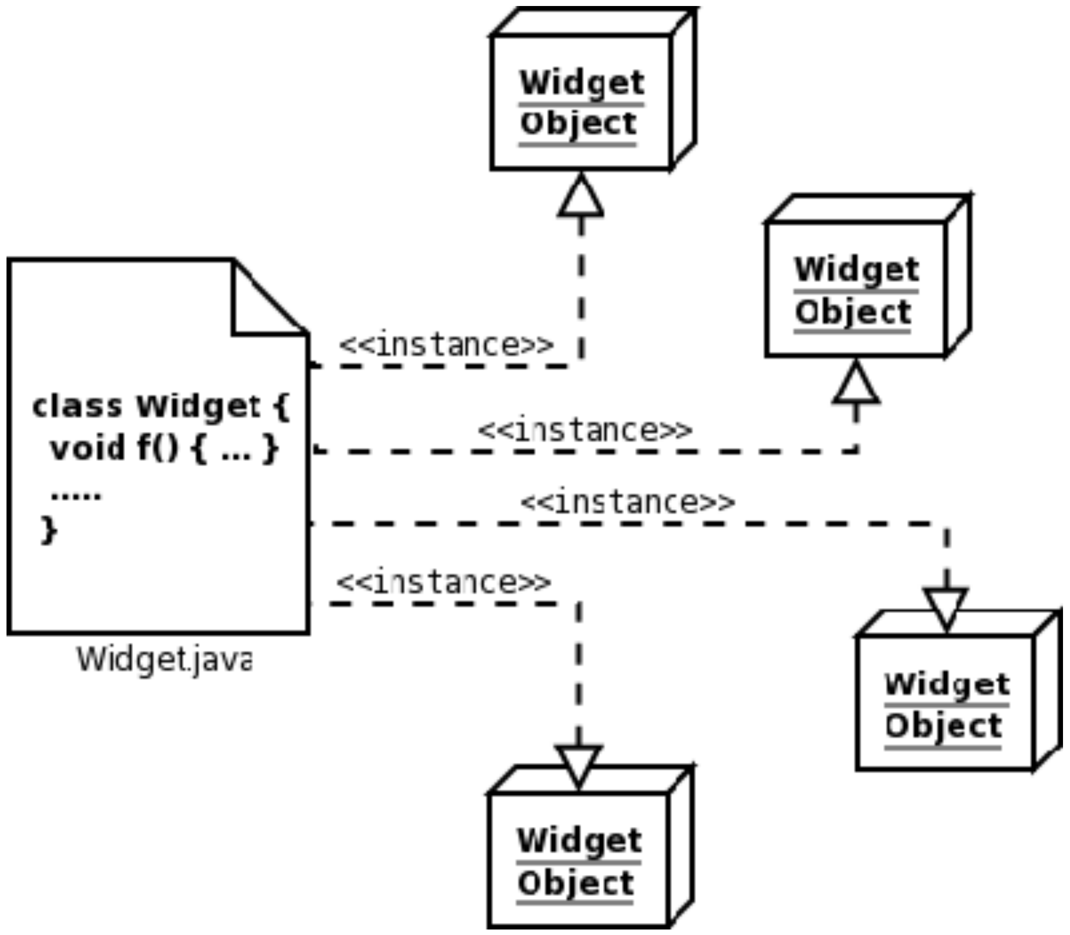


Fig. 2-4: **Instantiating Objects from Class Definition**

Writing Java Applications

- A Java application is a collection of class descriptions
- A Java application consists only of objects
  - All behavior required in the application must be provided by the objects
  - Objects respond to methods
  - Methods are units of executable code (a.k.a, functions)
  - The logic of a method is described in the appropriate class definition
- In other words, when we write a Java application, we write a set of class definitions that collaborate with one another

Writing Java Applications (cont.)

- Class definitions are often grouped into *packages*
  - Packages will be covered later in the course
  - For now, think of packages as groups of related classes
- Each class definition starts as source code
- Source code files have to follow certain rules
  - Must end with the extension `.java`
  - Can be multiple class definitions per file
  - Only one `public` class definition per file
  - `public` class definitions are stored in files with the same name as the `public` class name - including case
    - Java, as a language, is case sensitive
    - Java, as a platform, is case sensitive including the
      - compiler
      - class loader

Deciding on Classes

- How does the programmer decide what classes should be in the program?
- In a perfect world, the programmer does not make those decisions
- The classes and their responsibilities should have been determined in the design of the application before the programming begins

The Banking Application

- Customers have accounts at the DI Bank
  - There are two kinds of accounts
    - Checking
    - Savings
  - There are two kinds of customers
    - Business
    - Personal
- Customers can access their accounts through bank ATMs
  - All ATMs support withdrawals and account queries
  - Only the full service ATMs support deposits
- Customers can also access their accounts at the Teller window, they can
  - Open new accounts
  - Close existing accounts

The Banking Application

- BankAccount
  - Like your bank account, it should have an account number and a balance associated with it
  - You should also be able to query the balance, do a deposit and a withdrawal
  - Bank accounts can be opened and closed
  - They can be either checking or savings
- Customer
  - Customers have names and customer id numbers
  - Customers are either of type business or type personal
  - Customers and bank accounts are many to many relationship

The Banking Application

- ATM
  - Customers talk to ATMs and ATMs access accounts
  - ATMs have an identification number
  - ATMs receive messages from customers and send them on to accounts
  - ATMs should also know how much cash they currently hold
  - Can only dispense multiples of $20
- Teller
  - Tellers have an employee id
  - They can do everything that ATMs can do

The Banking Application

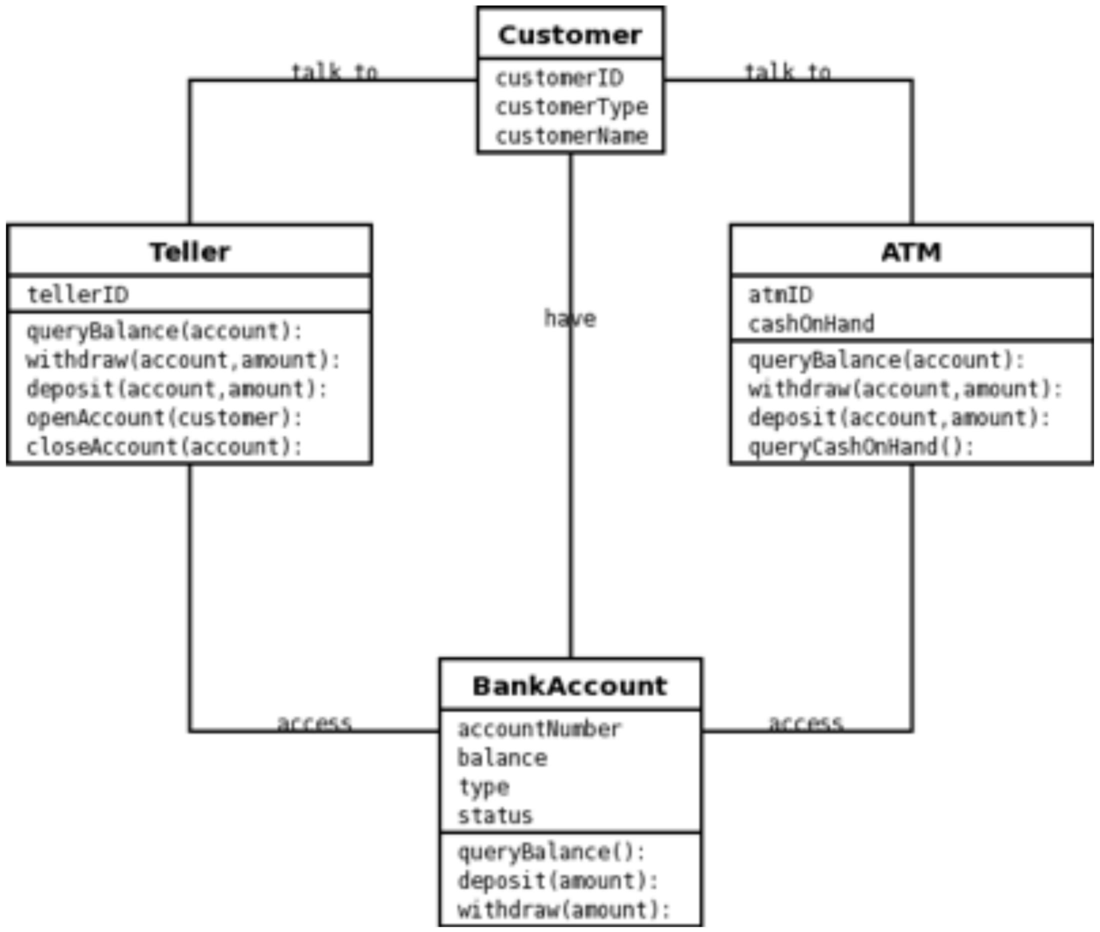


Fig. 2-5: **UML for the banking application classes**

First implementation of BankAccount

```
class BankAccount {

    // Data part of the class definition
    String  accountNumber;
    int accountBalance;
    String  accountStatus;   // open or closed
    String  accountType;     // savings or checking

    // Methods or Message part of the class definition
    int queryBalance() { return 0; }
    int deposit (int amount) { return 0; }
    int withdraw(int amount) { return 0;  }

} // end of class definition
```

Example 2-1: First implementation of BankAccount

Coding the Bank Application

- There are only two types of things that go into a class definition
  - Variable declarations
  - Method definitions
- All variables and methods have to be defined inside a class definition
- The `BankAccount` class has four variables
  - Store the data associated with a `BankAccount` object
  - These are also called *instance variables*
- Objects of type `BankAccount` can respond to three kinds of messages:
  - `deposit(amount)`
  - `withdraw(amount)`
  - `queryBalance()`

Exercise: Class Open Discussion

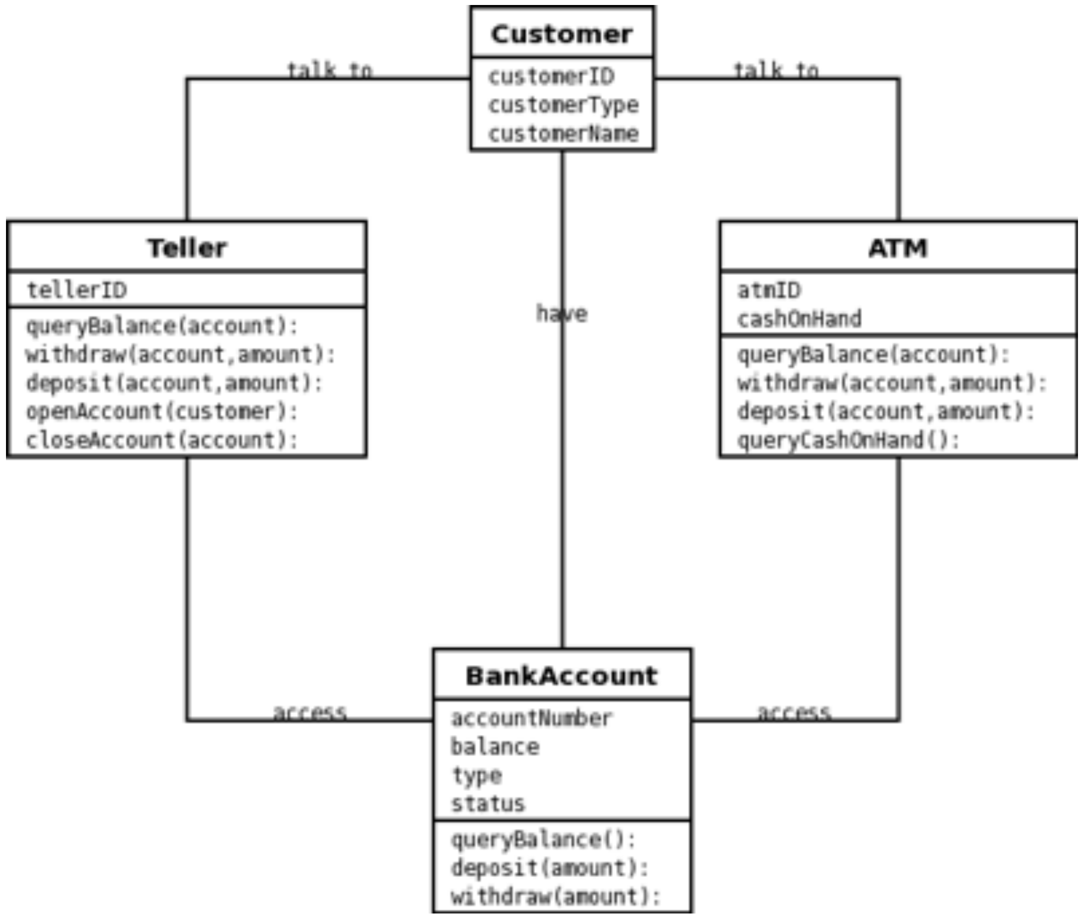


Fig. 2-5: **UML for the banking application classes**

Summary

We covered the

- Benefits & concepts of Object Oriented Programming (OOP)
- Basic structure of an OO program
- Relationship between objects, types and class definitions
- DI Banking Application

OO Programming with Java

Objectives

At the end of this section you should be able to

- Discuss how a Java class definition is organized
- Write and compile Java class definitions
- Discuss what `init` and `main` methods do in Java
- Understand the purpose of an application class and a run-time container
- Use `javadoc` to create documentation

About Class Definitions

- The class definition starts with the class name
- Can be preceded by the optional keyword public
- We can name our classes whatever we want
  - Subject to a small set of rules about names
  - Class name cannot be a keyword
  - A keyword is a word reserved by Java, like `public` or `class`
- Normally we try to pick a meaningful name in the context of the application
- Typically follow Java conventions
  - Not required by Java, but good style
  - Capitalize the first letter of each word in the class name
  - e.g. `BankAccount`, `FullServiceATM`

About Class Definitions (cont.)

- The class definition contains a list of
  - Variable definitions
  - Method definitions
- The variable definitions and the method definitions can be placed in any order in the class definition
  - Order is not important like in structure programs
  - Java has no rules about order of definitions
- Java convention is to capitalize the first letter of each word in
  - A variable
    `accountNumber, ssn, firstName`
  - A method name except for the first one
    `getAccountNumber(), setFirstName(String s)`

About Class Definitions (cont.)

- The whole class definition is
  - Defined in a block of code “labeled” class
  - Whose body is contained in braces `{ }`
- Each method has a body
  - Enclosed in braces `{ }`
  - Called the method definition
  - Provides the execution logic for the method
- Statements end with a semi-colon
  - Variable declarations
  - Variable assignment
- Java is not white space sensitive, except when dealing with `String`

Class Definition Example

```
class A {
  private int i;
  public void setI(int j) {
    i = j;
  }
}
```

```
class B
{
  private int i;
  public void setI(int j)
  {
    i = j;
  }
}
```

Comments in Java

- Java, like all programming languages, uses comments to document source code
- Java uses the same two types of comments that appear in C++
  - `// single line comment`

  -
    ```
    /*  multi
        line
        comments */
    ```
- Also adds a third kind - the `javadoc` comment - which is unique to Java

  -
    ```
    /** multi
        line
        javadoc
        comment */
    ```

Comment Example

Example 2-2: Comments in Java Code

```
/**
* This is the start of a Javadoc comment
* This is the second line of a Javadoc comment
*/

class HelloWorld { // This is a single line comment
     //Another single line comment
     public static void main (String [] args) {
     /*Here is where we have used a multi-line
      comment to temporarily comment-out a piece of
      code
      System.out.println("I’m commented out"); */
    System.out.println("I’m not commented out");
  }
}
```

Using javadoc

- `javadoc` is one of the utilities that is provide in the SDK
  - Designed so that a programmer can document classes, methods, fields and packages in an application
  - Formats, organizes and cross-references the documentation
  - Generates HTML documentation
- There is a syntax that `javadoc` comments have to follow
  - Allows for complex formatting and cross-referencing
  - For example, the whole set of API documentation for the J2SE and the other Java 2 editions are generated using `javadoc`
  - There are built-in “tags” for things like
    - Version
    - Author
    - Date
    - Etc.

javadoc Comment Structure

The basic structure of a `javadoc` comment is:

```
/**
* first line.
* <p>
* comment body
* <p>
* more comment body
*
* @directive
* @directive
*/
```

javadoc Comment Structure

Some of the basic rules for `javadoc` are

- First line in each `javadoc` comment is used as an index entry
- Line ends at the first period
- Utilize HTML to make the `javadoc` generated documentation easier to read
- Paragraph tags <p> are used to start new paragraphs in the comment body
- Utilize directives
- The comment body ends when a directive is encountered
- Place `javadoc` comments before the class and each method in the class

javadoc Comment Structure

Directives are lines that

- Start with a `@`
- Are immediately followed by a keyword like
  - `param`
  - `return`
  - `author`
  - Etc.

javadoc Comment Structure

- For method comments
  - A @param directive is required for each parameter
  - Looks like

```
@param nameofparamater  description which
does on until another directive  or the
end of the comment is encountered
```

- A @return directive is also required for methods, which describes what is returned and looks like

```
@return a description of the return value
of the method
```

Starting up the Application

- The `main(String [] args)` method has a special role in Java applications
- The `main` method contains the start-up code that is used to bootstrap the Java application
- The `main` method is the same as the mainline program in structured programming
  - The Java application runs as a process on the host operating system
  - `main` is the entry point to that process

Starting up the Application

- The `main` method works like this
  - `java HelloWorld` is executed at the command line
  - The class loader finds and loads the file `HelloWorld.class`
  - The JVM looks through the class file for a method that looks like

```
public static void main(String [] args) {
    /* main method body */
}
```

  - The JVM executes the body of the `main` method
  - When the `main` method finishes executing, the Java application finishes and the JVM exits

Starting up the Application

Example 2-4: Adding a main method to BankAccount

```
public class BankAccount {
  String accountNumber;
  int accountBalance;
  String accountType;
  String accountStatus;
  int queryBalance() {
    return 0;
  }
  int deposit (int amount) {
    return 0;
  }
  int withdraw(int amount) {
    return 0;
  }

  public static void main(String [] args) {
     System.out.println("Bank Account main method executing..");
  }
}
```

Use of main

- We create a special class which is responsible for starting up and shutting down the application
  - The main method then goes into this "application" class
  - Commonly called Main.class
- For example, we if we had a banking application we could
  - Define a new class called BankApp
  - Responsible for overall management of our banking application
- The BankApp class can start off like this

Command Line Arguments

- Command line arguments are passed to an application following an the applications name
  `java HelloWorld arg1 arg2 "this is argument 3"`
- The command line arguments are passed to the `main` method as an array of `String`s

```
public static void main(String [] args)
```

  - `args` is a variable name
    - It references an array of `String` objects
    - You can name it whatever you want
  - Has a zero length if no arguments were specified
  - The following example demonstrates this

Command Line Arguments Example

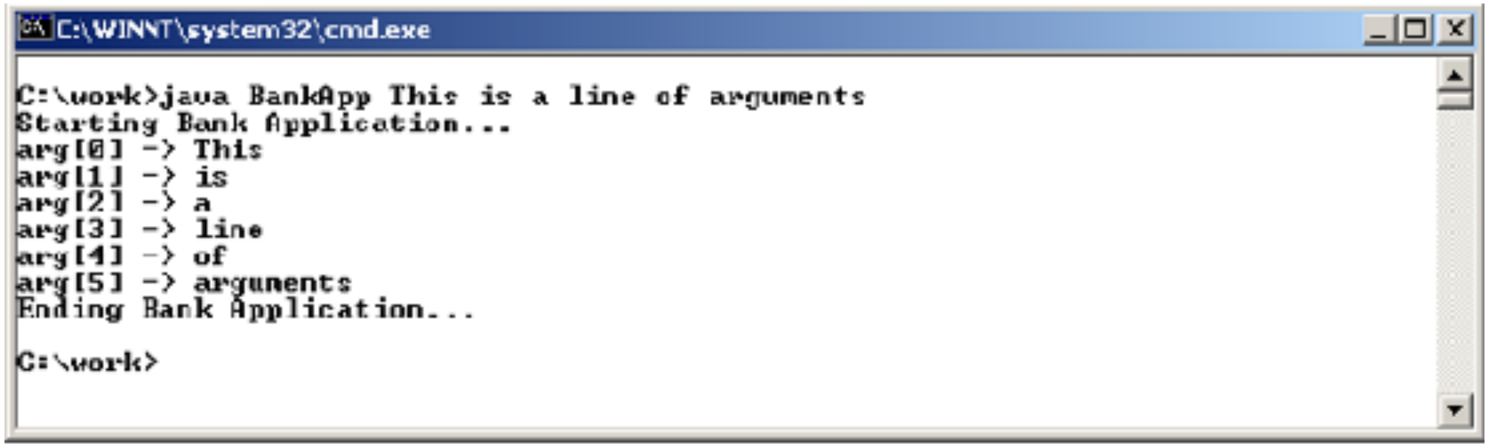
Example 2-6: Using command line arguments with main

```
public class BankApp {
  public static void main(String [] args) {
    System.out.println("Starting Bank Application");

    // print out the command line arguments
    for (int argcount = 0; argcount < args.length; argcount ++)
      System.out.println("arg["+argcount+"] -> "+args[argcount]);

    System.out.println("Ending Bank Application");
  }
}
```

Fig. 2-12: **Output of example 2-6**



C:\WINNT\system32\cmd.exe

```
C:\work>java BankApp This is a line of arguments
Starting Bank Application...
arg[0] -> This
arg[1] -> is
arg[2] -> a
arg[3] -> line
arg[4] -> of
arg[5] -> arguments
Ending Bank Application...

C:\work>
```

The init method

- Applets do not have a `main` method
  - So how do they get bootstrapped?
  - This is the responsibility of the runtime container
- When a browser launches an `Applet` for the first time
  - Class definition for the `Applet` loaded into the local JVM
  - Browser calls the `public void init()` to initialize the `Applet`
  - Browser calls other *life-cycle* methods to manage the `Applet`
    - `public void start()`
    - `public void stop()`
    - `public void destroy()`

The Applet Lifecycle

Every programmer needs to implement the code for the four lifecycle callback

- The `init` method - initializes the applet when it is first loaded by the browser
- The `start` method - sent from the browser to the applet to every time the browser returns to the page containing the applet
- The `stop` method - sent when the browser moves off the page containing the applet
- The `destroy` method - Sent when the browser shuts down

The Bank Application Class

- The `Applet` model is an example of good OO program design
- A runtime environment that
  - Creates the program objects
  - Initializes them
  - Kicks the application off
  - Shuts down the application gracefully
  - Disposes of the program objects
- In our banking application the application class will act like a runtime container

The Bank Application Class

We are going to use a `BankApp` class to fill three roles:

1. Manage our bank application's lifecycle
2. Act as a container for all of the objects that need to work together in our bank application
3. Provide services analogous to a runtime container for the objects that make up the bank application

Summary

We covered

- How a Java class definition is organized
- What `init` and `main` methods do in Java
- The purpose of an application class and a run-time container
- Using `javadoc` to create documentation

Exercises

- *Exercise 2-1: Bank Application Starter Code*
  - *In this exercise, you are going to create the Bank Application classes that you will continue to modify and improve as the course progresses. After you have some starter classes, you are going to document them using* `javadoc`.
- *Exercise 2-2: Adding an Application Class*
  - *In this exercise you will add an additional class to your application.*