ATM Case Study Part 2

Visibility

- Access modifiers determine the visibility or accessibility of an object’s attributes and methods to other objects.
  - □ Before we can begin implementing our design, we must consider which attributes and methods of our classes should be `public` and which should be `private`.
  - □ Attributes normally should be `private` and that methods invoked by clients of a given class should be `public`.
  - □ Methods that are called as “utility methods” only by other methods of the class normally should be `private`.
- The UML employs visibility markers for modeling the visibility of attributes and operations.
  - □ Public is indicated by placing a plus sign (+) before an operation or an attribute; a minus sign (–) indicates private.

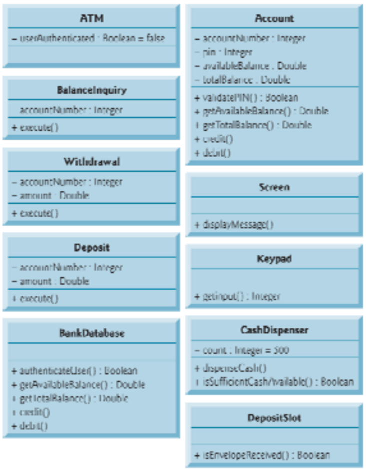

ATM
– userAuthenticated : Boolean = false
BalanceInquiry
– accountNumber : Integer
+ execute()
Withdrawal
– accountNumber : Integer
– amount : Double
+ execute()
Deposit
– accountNumber : Integer
– amount : Double
+ execute()
BankDatabase
+ authenticateUser() : Boolean
+ getAvailableBalance() : Double
+ getTotalBalance() : Double
+ credit()
+ debit()
Account
– accountNumber : Integer
– pin : Integer
– availableBalance : Double
– totalBalance : Double
+ validatePIN() : Boolean
+ getAvailableBalance() : Double
+ getTotalBalance() : Double
+ credit()
+ debit()
Screen
+ displayMessage()
Keypad
+ getInput() : Integer
CashDispenser
– count : Integer = 500
+ dispenseCash()
+ isSufficientCashAvailable() : Boolean
DepositSlot
+ isEnvelopeReceived() : Boolean

Fig. 13.1 | Class diagram with visibility markers.

Navigability

- The class diagram in Fig. 13.2 further refines the relationships among classes in the ATM system by adding navigability arrows to the association lines.
- Navigability arrows
  - □ represented as arrows with stick () arrowhead in the class diagram
  - □ indicate in the direction which an association can be traversed.
- Programmers use navigability arrows to determine which objects need references to other objects.
- Associations that have navigability arrows at both ends or have none at all indicate bidirectional navigability—navigation can proceed in either direction across the association.

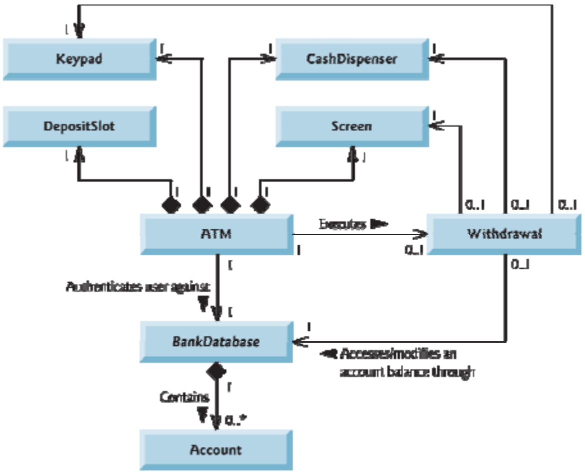
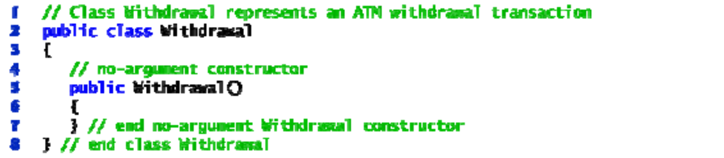


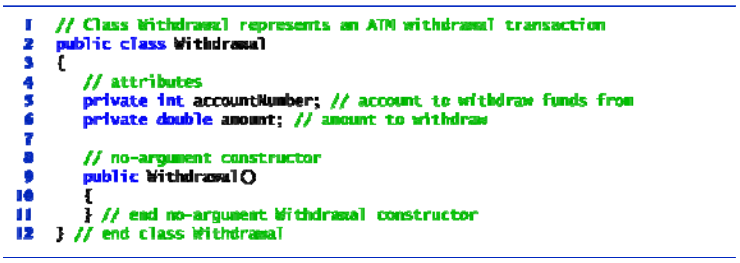
Fig. 13.2 | Class diagram with navigability arrows.



```
// Class Withdrawal represents an ATM withdrawal transaction
public class Withdrawal
{
   // no-argument constructor
   public Withdrawal()
   {
   } // end no-argument Withdrawal constructor
} // end class Withdrawal
```

Fig. 13.3 | Java code for class Withdrawal based on Figs. 13.1–13.2.

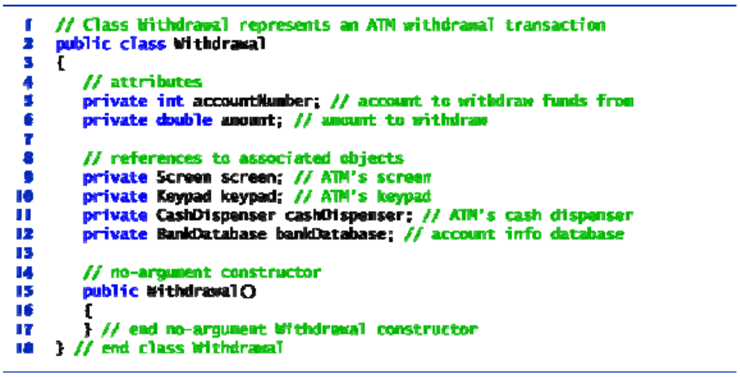
- Four guidelines for each class:
  - 1. Use the name located in the first compartment to declare the class as a `public` class with an empty no-argument constructor (Fig. 13.3).



```java
// Class Withdrawal represents an ATM withdrawal transaction
public class Withdrawal
{
   // attributes
   private int accountNumber; // account to withdraw funds from
   private double amount; // amount to withdraw

   // no-argument constructor
   public Withdrawal()
   {
   } // end no-argument Withdrawal constructor
} // end class Withdrawal
```

Fig. 13.4 | Java code for class `Withdrawal` based on Figs. 13.1–13.2.

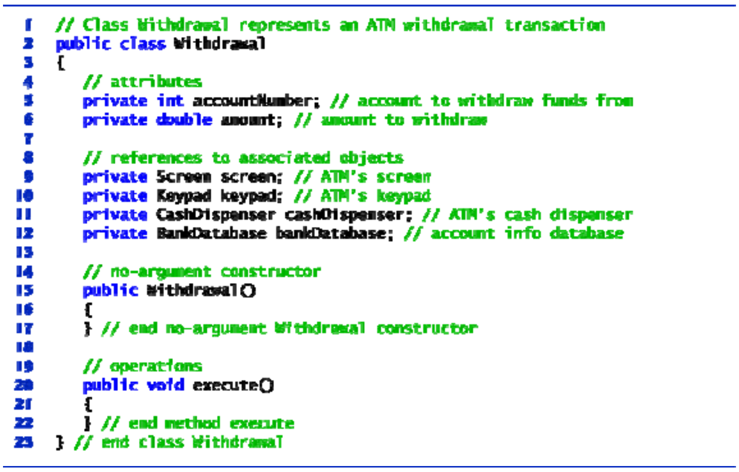
□2.Use the attributes located in the second compartment to declare the instance variables (Fig. 13.4).



```java
// Class Withdrawal represents an ATM withdrawal transaction
public class Withdrawal
{
   // attributes
   private int accountNumber; // account to withdraw funds from
   private double amount; // amount to withdraw

   // references to associated objects
   private Screen screen; // ATM's screen
   private Keypad keypad; // ATM's keypad
   private CashDispenser cashDispenser; // ATM's cash dispenser
   private BankDatabase bankDatabase; // account info database

   // no-argument constructor
   public Withdrawal()
   {
   } // end no-argument Withdrawal constructor
} // end class Withdrawal
```

Fig. 13.5 | Java code for class `Withdrawal` based on Figs. 13.1–13.2.

□3.Use the associations described in the class diagram to declare the references to other objects (Fig. 13.5).



```java
// Class Withdrawal represents an ATM withdrawal transaction
public class Withdrawal
{
   // attributes
   private int accountNumber; // account to withdraw funds from
   private double amount; // amount to withdraw

   // references to associated objects
   private Screen screen; // ATM's screen
   private Keypad keypad; // ATM's keypad
   private CashDispenser cashDispenser; // ATM's cash dispenser
   private BankDatabase bankDatabase; // account info database

   // no-argument constructor
   public Withdrawal()
   {
   } // end no-argument Withdrawal constructor

   // operations
   public void execute()
   {
   } // end method execute
} // end class Withdrawal
```

Fig. 13.6 | Java code for class `Withdrawal` based on Figs. 13.1–13.2.

- □ 4.Use the operations located in the third compartment of Fig. 13.1 to declare the shells of the methods (Fig. 13.6). If we have not yet specified a return type for an operation, we declare the method with return type `void`.

- To apply inheritance, look for commonality among classes in the system.
- Create an inheritance hierarchy to model similar (yet not identical) classes in a more elegant and efficient manner.
- Modify class diagram to incorporate the new inheritance relationships.
- Translate updated design into Java code.

Problem of representing a financial transaction in the system.

- Created three individual transaction classes—`BalanceInquiry`, `Withdrawal` and `Deposit`—to represent the transactions that the ATM system can perform.
- For the classes `BalanceInquiry`, `Withdrawal` and `Deposit`.
  - □ Each has one attribute (`accountNumber`) and one operation (`execute`) in common.
  - □ Each class requires attribute `accountNumber` to specify the account to which the transaction applies.
  - □ Each class contains operation `execute`, which the `ATM` invokes to perform the transaction.
- `BalanceInquiry`, `Withdrawal-` and `Deposit` represent *types of transactions.*

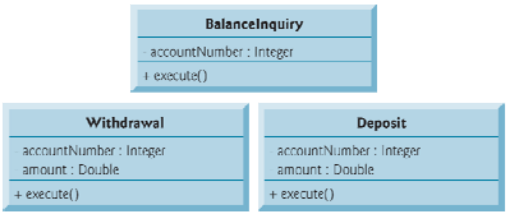


Fig. 13.7 | Attributes and operations of `BalanceInquiry`, `Withdrawal` and `Deposit`.

- The UML specifies a relationship called a generalization to model inheritance.
- Arrows with triangular hollow arrowheads indicate that classes `BalanceInquiry`, `Withdrawal` and `Deposit` extend class `Transaction`.
- Class `Transaction` is said to be a generalization of classes `BalanceInquiry`, `Withdrawal` and `Deposit`.
- Class `BalanceInquiry`, `Withdrawal` and `Deposit` are said to be specializations of class `Transaction`.

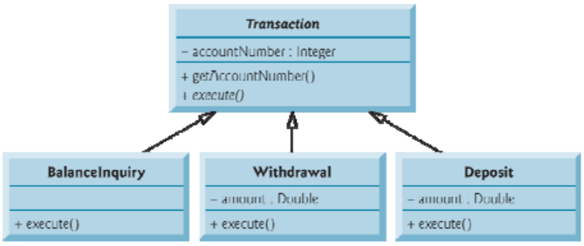


Fig. 13.8 | Class diagram modeling generalization of superclass `Transaction` and subclasses `BalanceInquiry`, `Withdrawal` and `Deposit`. Note that abstract class names (e.g., `Transaction`) and method names (e.g., `execute` in class `Transaction`) appear in italics.

- Polymorphism provides the `ATM` with an elegant way to execute all transactions “in the general.”
- The polymorphic approach also makes the system easily extensible.
- To create a new transaction type, just create an additional `Transaction` subclass that overrides the `execute` method with a version of the method appropriate for executing the new transaction type.

- We model an association between class `ATM` and class `Transaction` to show that the `ATM`, at any given moment, either is executing a transaction or is not (i.e., zero or one objects of type `Transaction` exist in the system at a time).
- Because a `Withdrawal` is a type of `Transaction`, we no longer draw an association line directly between class `ATM` and class `Withdrawal`.
- Subclass `Withdrawal` inherits superclass `Transaction`’s association with class `ATM`.
- Subclasses `BalanceInquiry` and `Deposit` inherit this association, too, so the previously omitted associations between `ATM` and classes `BalanceInquiry` and `Deposit` no longer exist either.

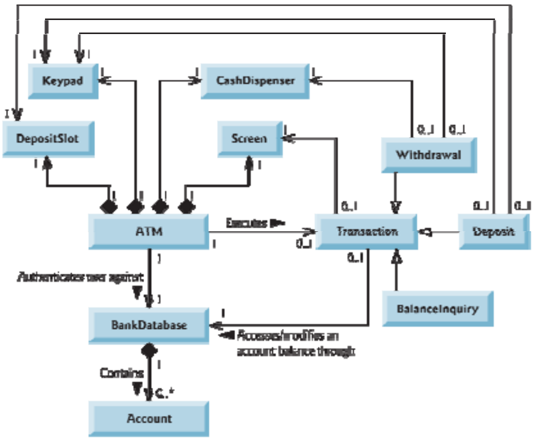
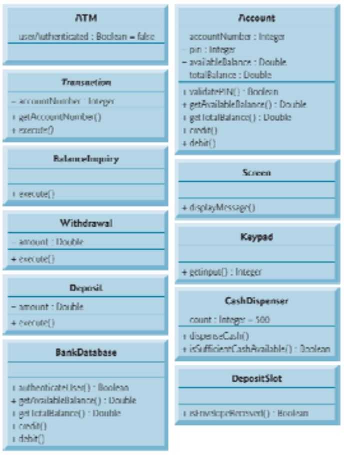


Fig. 13.9 | Class diagram of the ATM system (incorporating inheritance). Note that the abstract class name `Transaction` appears in italics.

- We also add an association between class `Transaction` and the `BankDatabase`
  - □ All `Transaction`s require a reference to the `BankDatabase` so they can access and modify account information.
- We show an association between class `Transaction` and the `Screen`.
  - □ All `Transaction`s display output to the user via the `Screen`.



ATM

- userAuthenticated : Boolean = false

Transaction

- accountNumber : Integer
+ getAccountNumber()
+ execute()

BalanceInquiry

+ execute()

Withdrawal

- amount : Double
+ execute()

Deposit

- amount : Double
+ execute()

BankDatabase

+ authenticateUser() : Boolean
+ getAvailableBalance() : Double
+ getTotalBalance() : Double
+ credit()
+ debit()

Account

- accountNumber : Integer
- pin : Integer
- availableBalance : Double
- totalBalance : Double
+ validatePIN() : Boolean
+ getAvailableBalance() : Double
+ getTotalBalance() : Double
+ credit()
+ debit()

Screen

+ displayMessage()

Keypad

+ getInput() : Integer

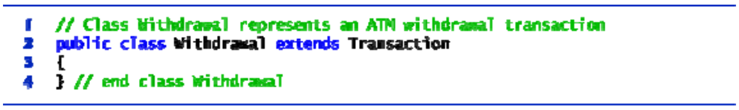
CashDispenser

- count : Integer = 500
+ dispenseCash()
+ isSufficientCashAvailable() : Boolean

DepositSlot

+ isEnvelopeReceived() : Boolean

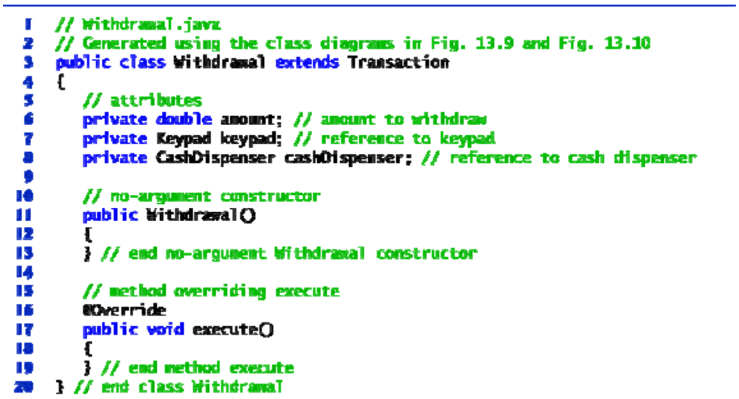
Fig. 13.10 | Class diagram with attributes and operations (incorporating inheritance). Note that the abstract class name `Transaction` and the abstract



```java
// Class Withdrawal represents an ATM withdrawal transaction
public class Withdrawal extends Transaction
{
} // end class Withdrawal
```

Fig. 13.11 | Java code for shell of class `Withdrawal`.

- Figure 13.11 shows the declaration of class `Withdrawal`.



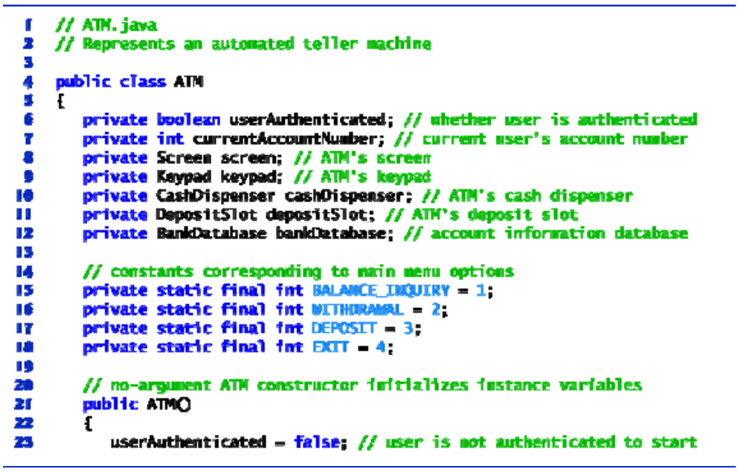
```java
// Withdrawal.java
// Generated using the class diagrams in Fig. 13.9 and Fig. 13.10
public class Withdrawal extends Transaction
{
   // attributes
   private double amount; // amount to withdraw
   private Keypad keypad; // reference to keypad
   private CashDispenser cashDispenser; // reference to cash dispenser

   // no-argument constructor
   public Withdrawal()
   {
   } // end no-argument Withdrawal constructor

   // method overriding execute
   @Override
   public void execute()
   {
   } // end method execute
} // end class Withdrawal
```

Fig. 13.12 | Java code for class Withdrawal based on Figs. 13.9 and 13.10.

- Figure 13.12 is the Java code for class `Withdrawal` from Fig. 13.9 and Fig. 13.10.

- Complete working 673-line implementation of the ATM system.
- Consider the classes in the order in which we identified them - `ATM`, `Screen`, `Keypad`, `CashDispenser`, `Deposit-Slot`, `Account`, `BankDatabase`, `Transaction`, `BalanceInquiry`, `Withdrawal` and `Deposit`.
- Our ATM design does not specify all the program logic and may not specify all the attributes and operations required to complete the ATM implementation.
  - This is a normal part of the object-oriented design process.
- The Java application (`ATMCaseStudy`) starts the ATM and puts the other classes in the system in use.

- Class `ATM` (Fig. 13.13) represents the ATM as a whole.
- Line 7 declares an attribute not found in our UML design—an `int` attribute `currentAccountNumber` that keeps track of the account number of the current authenticated user.
- Lines 8–12 declare reference-type attributes corresponding to the `ATM` class's associations modeled in Fig. 13.9.
  - □ These attributes allow the ATM to access its parts (i.e., its `Screen`, `Keypad`, `CashDispenser` and `DepositSlot`) and interact with the bank's account-information database (i.e., a `BankDatabase` object).



```
 1  // ATM.java
 2  // Represents an automated teller machine
 3
 4  public class ATM
 5  {
 6     private boolean userAuthenticated; // whether user is authenticated
 7     private int currentAccountNumber; // current user's account number
 8     private Screen screen; // ATM's screen
 9     private Keypad keypad; // ATM's keypad
10     private CashDispenser cashDispenser; // ATM's cash dispenser
11     private DepositSlot depositSlot; // ATM's deposit slot
12     private BankDatabase bankDatabase; // account information database
13
14     // constants corresponding to main menu options
15     private static final int BALANCE_INQUIRY = 1;
16     private static final int WITHDRAWAL = 2;
17     private static final int DEPOSIT = 3;
18     private static final int EXIT = 4;
19
20     // no-argument ATM constructor initializes instance variables
21     public ATM()
22     {
23        userAuthenticated = false; // user is not authenticated to start
```

Fig. 13.13 | Class ATM represents the ATM. (Part 1 of 7.)

```java
      currentAccountNumber = 0; // no current account number to start
      screen = new Screen(); // create screen
      keypad = new Keypad(); // create keypad
      cashDispenser = new CashDispenser(); // create cash dispenser
      depositSlot = new DepositSlot(); // create deposit slot
      bankDatabase = new BankDatabase(); // create acct info database
   } // end no-argument ATM constructor

   // start ATM
   public void run()
   {
      // welcome and authenticate user; perform transactions
      while ( true )
      {
         // loop while user is not yet authenticated
         while ( !userAuthenticated )
         {
            screen.displayMessageLine( "\nWelcome!" );
            authenticateUser(); // authenticate user
         } // end while

         performTransactions(); // user is now authenticated
         userAuthenticated = false; // reset before next ATM session
```

Fig. 13.13 | Class ATM represents the ATM. (Part 2 of 7.)

```java
         currentAccountNumber = 0; // reset before next ATM session
         screen.displayMessageLine( "\nThank you! Goodbye!" );
      } // end while
   } // end method run

   // attempts to authenticate user against database
   private void authenticateUser()
   {
      screen.displayMessage( "\nPlease enter your account number: " );
      int accountNumber = keypad.getInput(); // input account number
      screen.displayMessage( "\nEnter your PIN: " ); // prompt for PIN
      int pin = keypad.getInput(); // input PIN

      // set userAuthenticated to boolean value returned by database
      userAuthenticated =
         bankDatabase.authenticateUser( accountNumber, pin );

      // check whether authentication succeeded
      if ( userAuthenticated )
      {
         currentAccountNumber = accountNumber; // save user's account #
      } // end if
```

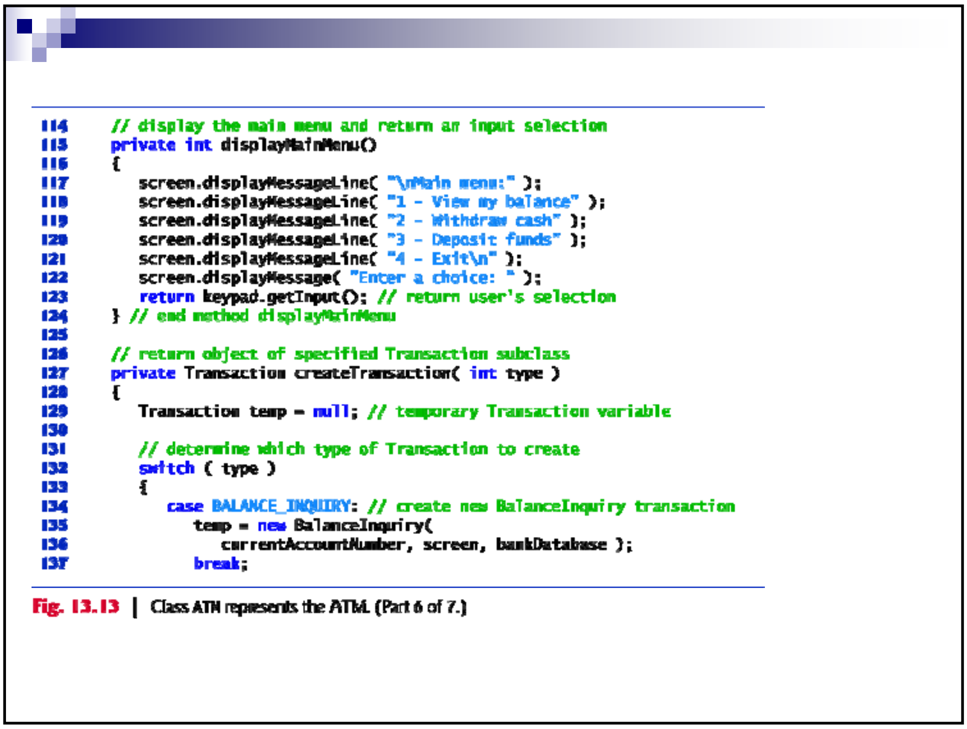
Fig. 13.13 | Class ATM represents the ATM. (Part 3 of 7.)

```java
      else
         screen.displayMessageLine(
            "Invalid account number or PIN. Please try again." );
   } // end method authenticateUser

   // display the main menu and perform transactions
   private void performTransactions()
   {
      // local variable to store transaction currently being processed
      Transaction currentTransaction = null;

      boolean userExited = false; // user has not chosen to exit

      // loop while user has not chosen option to exit system
      while ( !userExited )
      {
         // show main menu and get user selection
         int mainMenuSelection = displayMainMenu();

         // decide how to proceed based on user's menu selection
         switch ( mainMenuSelection )
         {
```

Fig. 13.13 | Class ATM represents the ATM. (Part 4 of 7.)

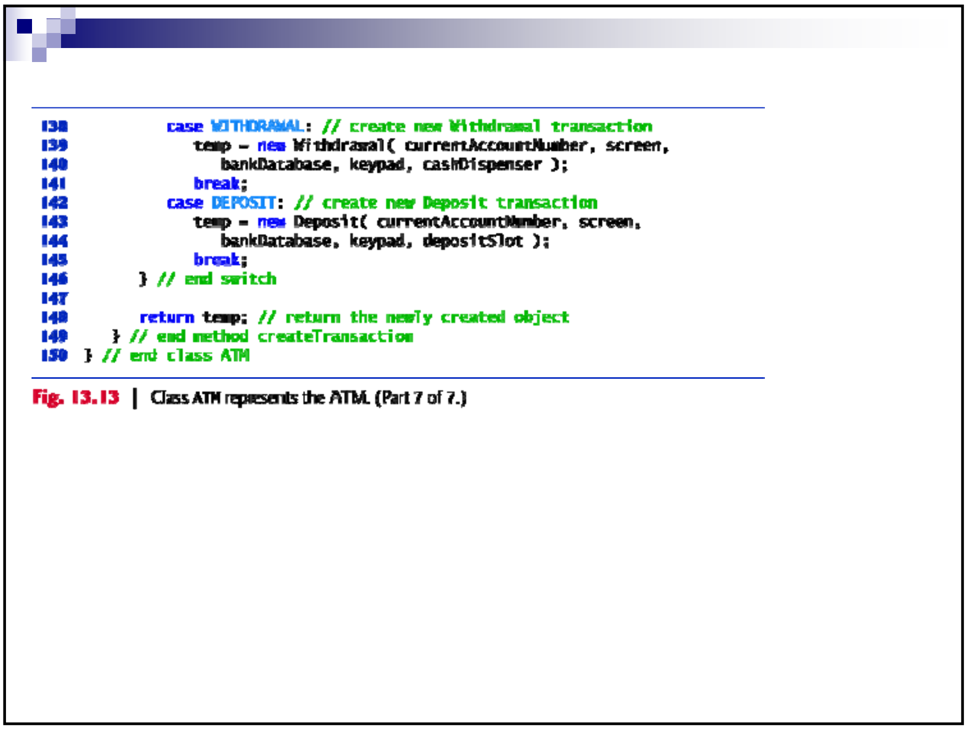
```java
            // user chose to perform one of three transaction types
            case BALANCE_INQUIRY:
            case WITHDRAWAL:
            case DEPOSIT:

               // initialize as new object of chosen type
               currentTransaction =
                  createTransaction( mainMenuSelection );

               currentTransaction.execute(); // execute transaction
               break;
            case EXIT: // user chose to terminate session
               screen.displayMessageLine( "\nExiting the system..." );
               userExited = true; // this ATM session should end
               break;
            default: // user did not enter an integer from 1-4
               screen.displayMessageLine(
                  "\nYou did not enter a valid selection. Try again." );
               break;
         } // end switch
      } // end while
   } // end method performTransactions

```

Fig. 13.13 | Class ATM represents the ATM. (Part 5 of 7.)



```
   // display the main menu and return an input selection
   private int displayMainMenu()
   {
      screen.displayMessageLine( "\nMain menu:" );
      screen.displayMessageLine( "1 - View my balance" );
      screen.displayMessageLine( "2 - Withdraw cash" );
      screen.displayMessageLine( "3 - Deposit funds" );
      screen.displayMessageLine( "4 - Exit\n" );
      screen.displayMessage( "Enter a choice: " );
      return keypad.getInput(); // return user's selection
   } // end method displayMainMenu

   // return object of specified Transaction subclass
   private Transaction createTransaction( int type )
   {
      Transaction temp = null; // temporary Transaction variable

      // determine which type of Transaction to create
      switch ( type )
      {
         case BALANCE_INQUIRY: // create new BalanceInquiry transaction
            temp = new BalanceInquiry(
               currentAccountNumber, screen, bankDatabase );
            break;
```

Fig. 13.13 | Class ATM represents the ATM. (Part 6 of 7.)



```
         case WITHDRAWAL: // create new Withdrawal transaction
            temp = new Withdrawal( currentAccountNumber, screen,
               bankDatabase, keypad, cashDispenser );
            break;
         case DEPOSIT: // create new Deposit transaction
            temp = new Deposit( currentAccountNumber, screen,
               bankDatabase, keypad, depositSlot );
            break;
      } // end switch

      return temp; // return the newly created object
   } // end method createTransaction
} // end class ATM
```

Fig. 13.13 | Class ATM represents the ATM. (Part 7 of 7.)

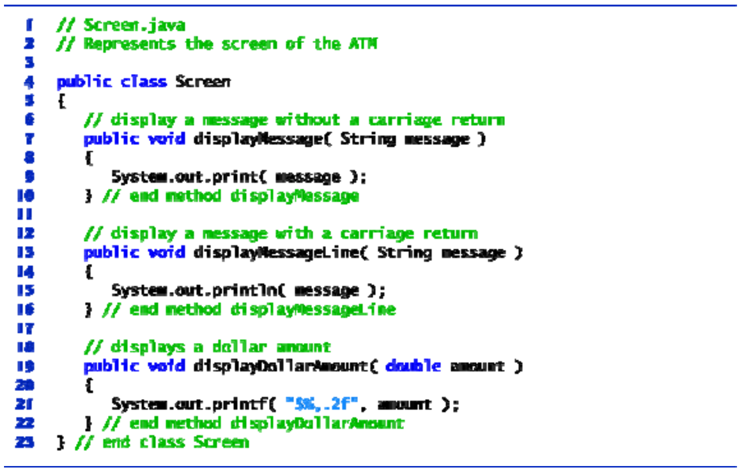
- The class diagram of Fig. 13.10 does not list any operations for class `ATM`.
- We implement one operation in class `ATM` that allows an external client of the class (i.e., class `ATMCaseStudy`) to tell the `ATM` to run.
- `ATM` method `run` (lines 33–50) uses an infinite loop (lines 36–49) to repeatedly welcome a user, attempt to authenticate the user and, if authentication succeeds, allow the user to perform transactions.
  - Simulates the fact that an ATM appears to run continuously until the bank turns it off (an action beyond the user's control).
  - An ATM user has the option to exit the system but not the ability to turn off the ATM completely.

- Lines 39–43 cause the ATM to repeatedly welcome and attempt to authenticate the user as long as the user has not been authenticated (i.e., `!userAuthenticated` is `true`).
- We refer to the requirements document to determine the steps necessary to authenticate the user before allowing transactions to occur.

- Method `performTransactions` (lines 75–112) carries out an ATM session for an authenticated user.
- We use a `Transaction` variable here to allow us to take advantage of polymorphism.
  - □ We name this variable after the role name included in the class diagram of Fig. 12.7—`currentTransaction`.
- Method `createTransaction` (lines 127–149) uses a `switch` statement (lines 132–146) to instantiate a new `Transaction` subclass object of the type indicated by the parameter `type`.

- However, if a user does not perform a transaction and instead selects the main menu option to exit, line 104 sets `userExited` to `true`, causing the condition of the `while` loop (`!userExited`) to become `false`.
- If the user enters an invalid main menu selection (i.e., not an integer from 1–4), lines 107–108 display an appropriate error message, `userExited` remains `false` and the user returns to the main menu to try again.

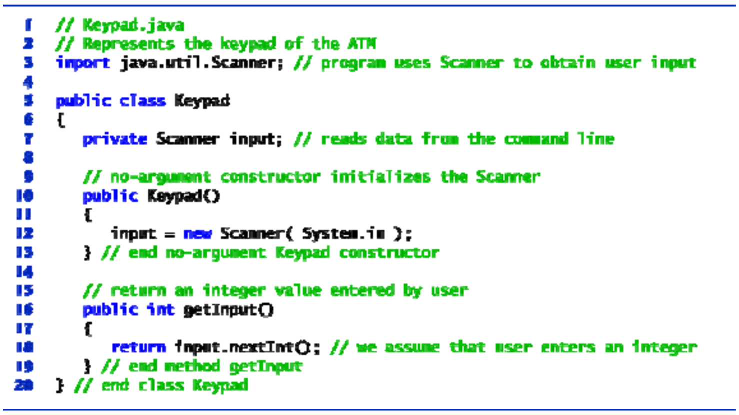
- Class `Screen` (Fig. 13.14) represents the screen of the ATM and encapsulates all aspects of displaying output to the user.
- We designed class `Screen` to have one operation—`displayMessage`.
  - For greater flexibility in displaying messages to the `Screen`, we now declare three `Screen` methods—`displayMessage`, `displayMessageLine` and `displayDollar-Amount`.



```java
// Screen.java
// Represents the screen of the ATM

public class Screen
{
   // display a message without a carriage return
   public void displayMessage( String message )
   {
      System.out.print( message );
   } // end method displayMessage

   // display a message with a carriage return
   public void displayMessageLine( String message )
   {
      System.out.println( message );
   } // end method displayMessageLine

   // displays a dollar amount
   public void displayDollarAmount( double amount )
   {
      System.out.printf( "$%,.2f", amount );
   } // end method displayDollarAmount
} // end class Screen
```

Fig. 13.14 | Class Screen represents the screen of the ATM.

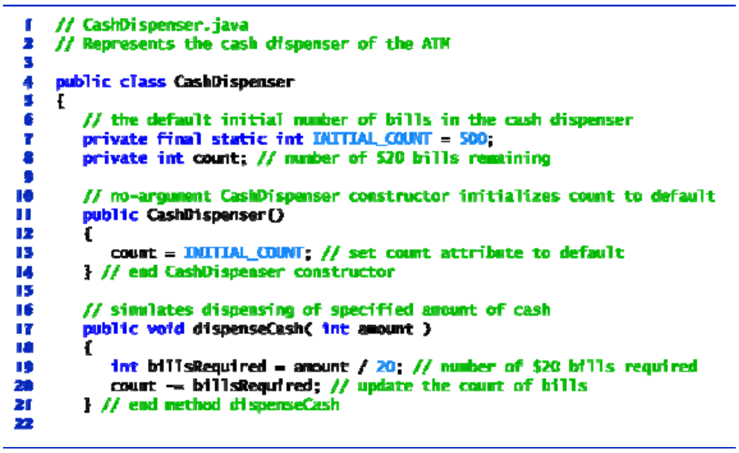
- Class `Keypad` (Fig. 13.15) represents the keypad of the ATM and is responsible for receiving all user input.
- We assume that the user presses only the keys on the computer keyboard that also appear on the keypad—the keys numbered 0–9 and the *Enter* key.



```java
// Keypad.java
// Represents the keypad of the ATM
import java.util.Scanner; // program uses Scanner to obtain user input

public class Keypad
{
   private Scanner input; // reads data from the command line

   // no-argument constructor initializes the Scanner
   public Keypad()
   {
      input = new Scanner( System.in );
   } // end no-argument Keypad constructor

   // return an integer value entered by user
   public int getInput()
   {
      return input.nextInt(); // we assume that user enters an integer
   } // end method getInput
} // end class Keypad
```

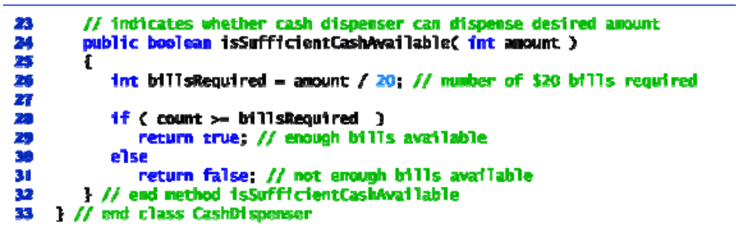
Fig. 13.15 | Class `Keypad` represents the ATM's keypad.

- Class `CashDispenser` (Fig. 13.16) represents the cash dispenser of the ATM.
- Constant `INITIAL_COUNT` indicates the initial count of bills in the cash dispenser when the ATM starts (i.e., `500`).
- The class trusts that a client (i.e., `Withdrawal`) calls `dispenseCash` only after establishing that sufficient cash is available by calling `isSufficientCashAvailable`.
- Thus, `dispenseCash` simply simulates dispensing the requested amount without checking whether sufficient cash is available.



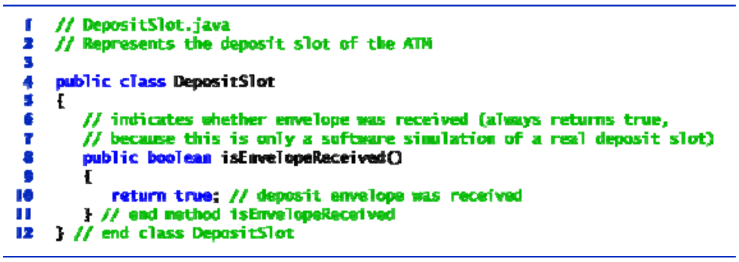
```java
// CashDispenser.java
// Represents the cash dispenser of the ATM

public class CashDispenser
{
   // the default initial number of bills in the cash dispenser
   private final static int INITIAL_COUNT = 500;
   private int count; // number of $20 bills remaining

   // no-argument CashDispenser constructor initializes count to default
   public CashDispenser()
   {
      count = INITIAL_COUNT; // set count attribute to default
   } // end CashDispenser constructor

   // simulates dispensing of specified amount of cash
   public void dispenseCash( int amount )
   {
      int billsRequired = amount / 20; // number of $20 bills required
      count -= billsRequired; // update the count of bills
   } // end method dispenseCash

```

Fig. 13.16 | Class `CashDispenser` represents the ATM's cash dispenser. (Part 1 of 2.)



```
   // indicates whether cash dispenser can dispense desired amount
   public boolean isSufficientCashAvailable( int amount )
   {
      int billsRequired = amount / 20; // number of $20 bills required

      if ( count >= billsRequired  )
         return true; // enough bills available
      else
         return false; // not enough bills available
   } // end method isSufficientCashAvailable
} // end class CashDispenser
```

Fig. 13.16 | Class `CashDispenser` represents the ATM's cash dispenser. (Part 2 of 2.)

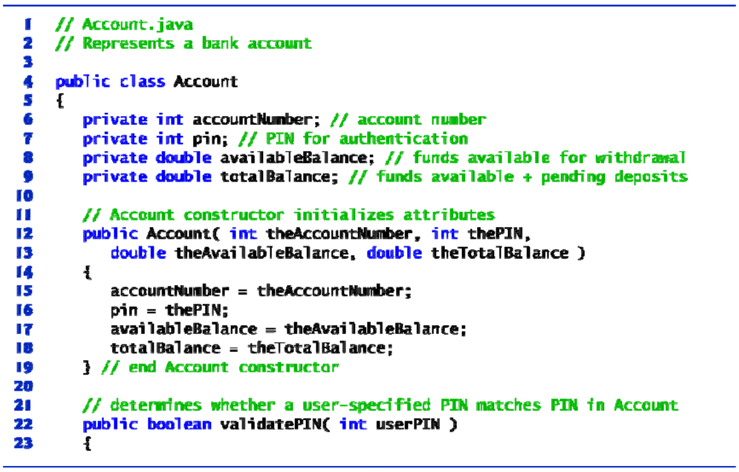


```
 1  // DepositSlot.java
 2  // Represents the deposit slot of the ATM
 3
 4  public class DepositSlot
 5  {
 6     // indicates whether envelope was received (always returns true,
 7     // because this is only a software simulation of a real deposit slot)
 8     public boolean isEnvelopeReceived()
 9     {
10        return true; // deposit envelope was received
11     } // end method isEnvelopeReceived
12  } // end class DepositSlot
```

Fig. 13.17 | Class `DepositSlot` represents the ATM's deposit slot.

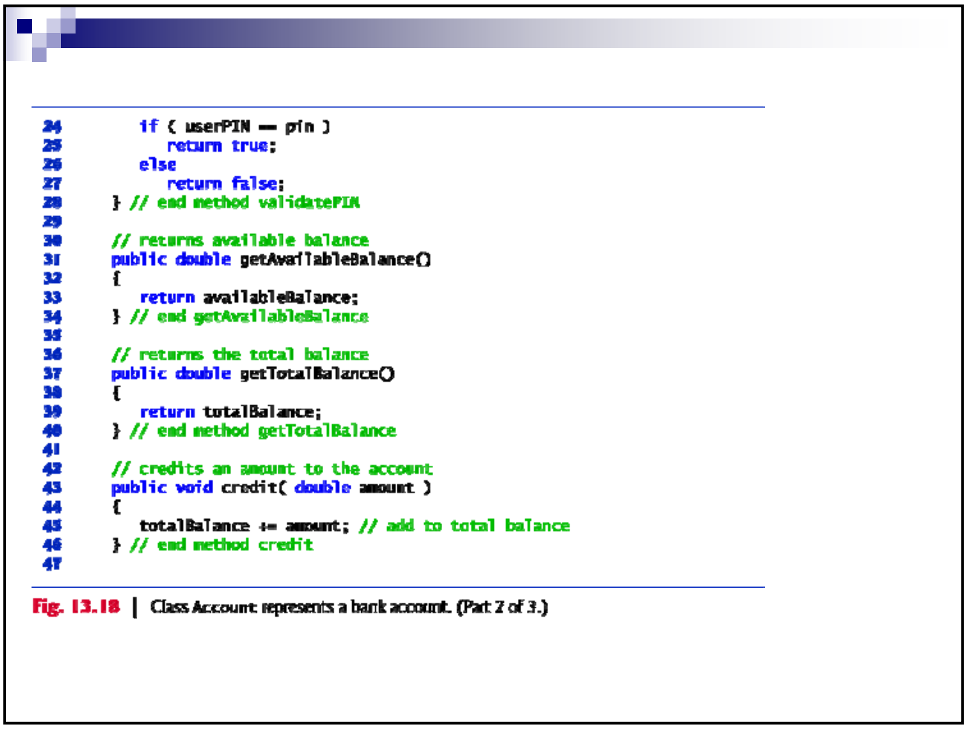
- Class `DepositSlot` (Fig. 13.17) represents the ATM's deposit slot.
- `DepositSlot` has no attributes and only one method—`isEnvelopeReceived` (lines 8–11)—which indicates whether a deposit envelope was received.

- Class `Account` (Fig. 13.18) represents a bank account.
- Each `Account` has four attributes (modeled in Fig. 13.10)—`accountNumber`, `pin`, `availableBalance` and `totalBalance`.
- Variable `availableBalance` represents the amount of funds available for withdrawal.
- Variable `totalBalance` represents the amount of funds available, plus the amount of deposited funds still pending confirmation or clearance.



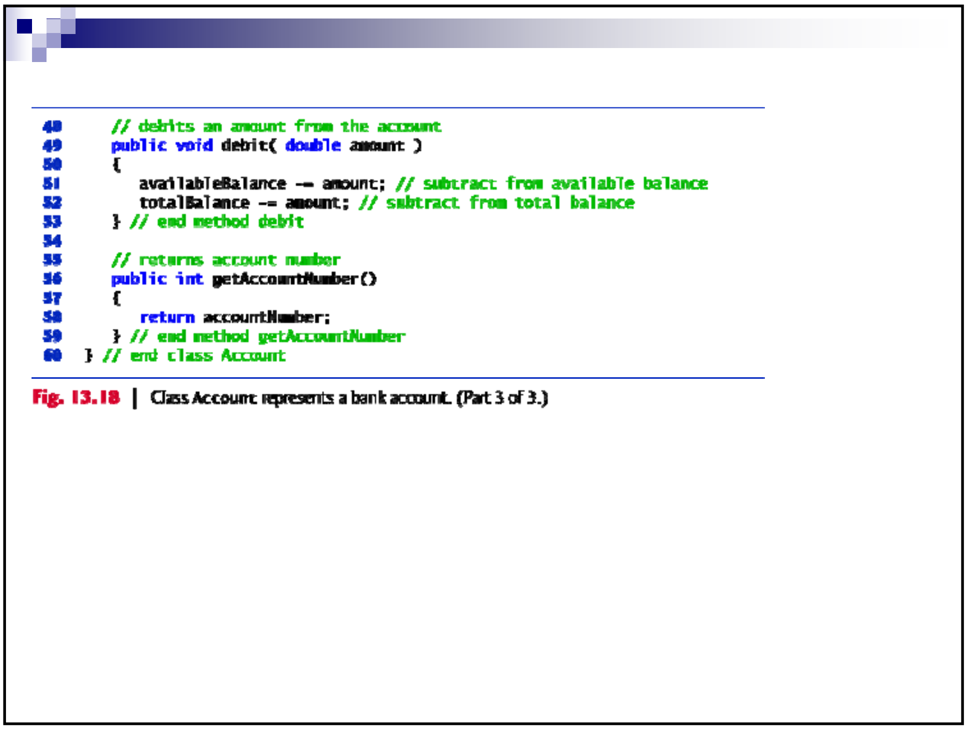
```java
// Account.java
// Represents a bank account

public class Account
{
   private int accountNumber; // account number
   private int pin; // PIN for authentication
   private double availableBalance; // funds available for withdrawal
   private double totalBalance; // funds available + pending deposits

   // Account constructor initializes attributes
   public Account( int theAccountNumber, int thePIN,
      double theAvailableBalance, double theTotalBalance )
   {
      accountNumber = theAccountNumber;
      pin = thePIN;
      availableBalance = theAvailableBalance;
      totalBalance = theTotalBalance;
   } // end Account constructor

   // determines whether a user-specified PIN matches PIN in Account
   public boolean validatePIN( int userPIN )
   {
```

Fig. 13.18 | Class Account represents a bank account. (Part 1 of 3.)



```
      if ( userPIN == pin )
         return true;
      else
         return false;
   } // end method validatePIN

   // returns available balance
   public double getAvailableBalance()
   {
      return availableBalance;
   } // end getAvailableBalance

   // returns the total balance
   public double getTotalBalance()
   {
      return totalBalance;
   } // end method getTotalBalance

   // credits an amount to the account
   public void credit( double amount )
   {
      totalBalance += amount; // add to total balance
   } // end method credit

```

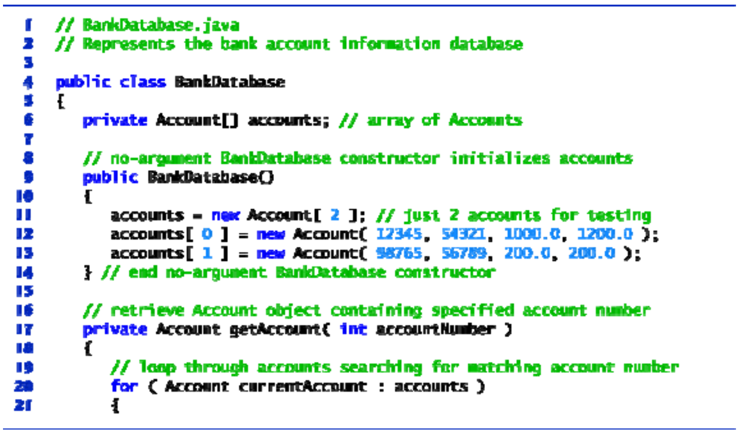
Fig. 13.18 | Class Account represents a bank account. (Part 2 of 3.)



```
   // debits an amount from the account
   public void debit( double amount )
   {
      availableBalance -= amount; // subtract from available balance
      totalBalance -= amount; // subtract from total balance
   } // end method debit

   // returns account number
   public int getAccountNumber()
   {
      return accountNumber;
   } // end method getAccountNumber
} // end class Account
```

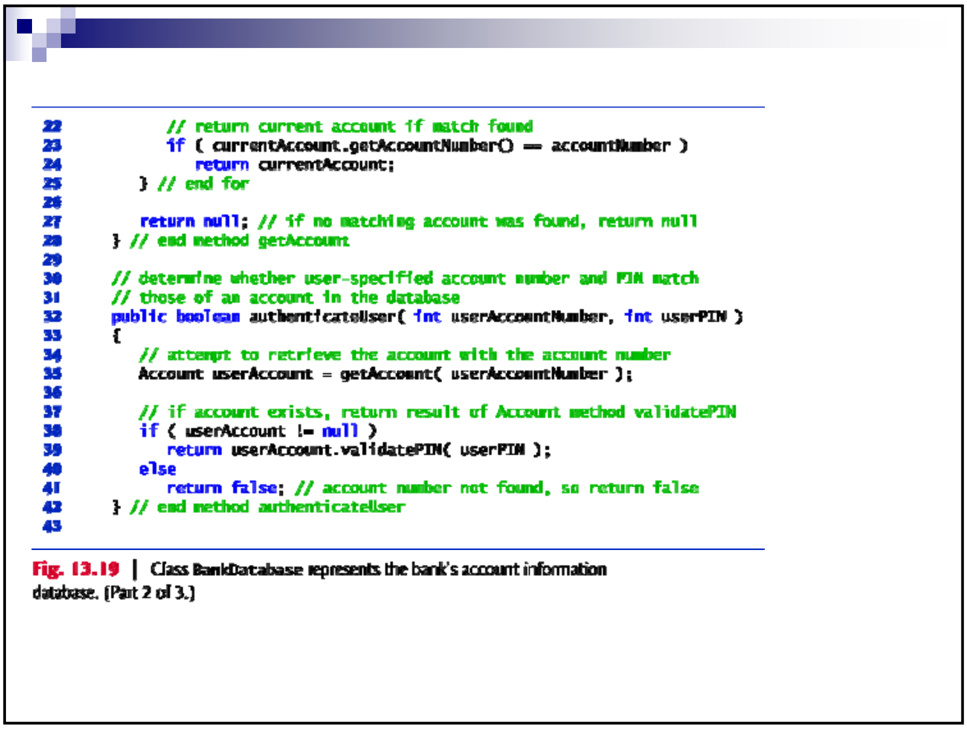
Fig. 13.18 | Class Account represents a bank account. (Part 3 of 3.)

- Class `BankDatabase` (Fig. 13.19) models the bank’s database with which the ATM interacts to access and modify a user’s account information.
- We determine one reference-type attribute for class `BankDatabase` based on its composition relationship with class `Account`.



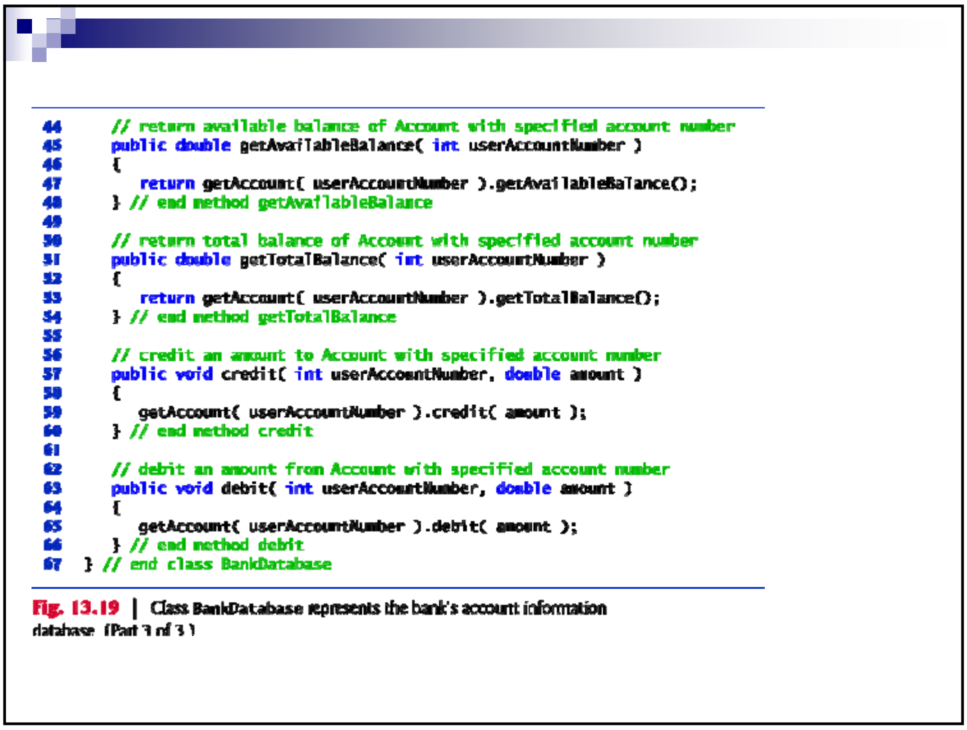
```java
// BankDatabase.java
// Represents the bank account information database

public class BankDatabase
{
   private Account[] accounts; // array of Accounts

   // no-argument BankDatabase constructor initializes accounts
   public BankDatabase()
   {
      accounts = new Account[ 2 ]; // just 2 accounts for testing
      accounts[ 0 ] = new Account( 12345, 54321, 1000.0, 1200.0 );
      accounts[ 1 ] = new Account( 98765, 56789, 200.0, 200.0 );
   } // end no-argument BankDatabase constructor

   // retrieve Account object containing specified account number
   private Account getAccount( int accountNumber )
   {
      // loop through accounts searching for matching account number
      for ( Account currentAccount : accounts )
      {
```

Fig. 13.19 | Class `BankDatabase` represents the bank’s account information database. (Part 1 of 3.)



```
         // return current account if match found
         if ( currentAccount.getAccountNumber() == accountNumber )
            return currentAccount;
      } // end for

      return null; // if no matching account was found, return null
   } // end method getAccount

   // determine whether user-specified account number and PIN match
   // those of an account in the database
   public boolean authenticateUser( int userAccountNumber, int userPIN )
   {
      // attempt to retrieve the account with the account number
      Account userAccount = getAccount( userAccountNumber );

      // if account exists, return result of Account method validatePIN
      if ( userAccount != null )
         return userAccount.validatePIN( userPIN );
      else
         return false; // account number not found, so return false
   } // end method authenticateUser

```

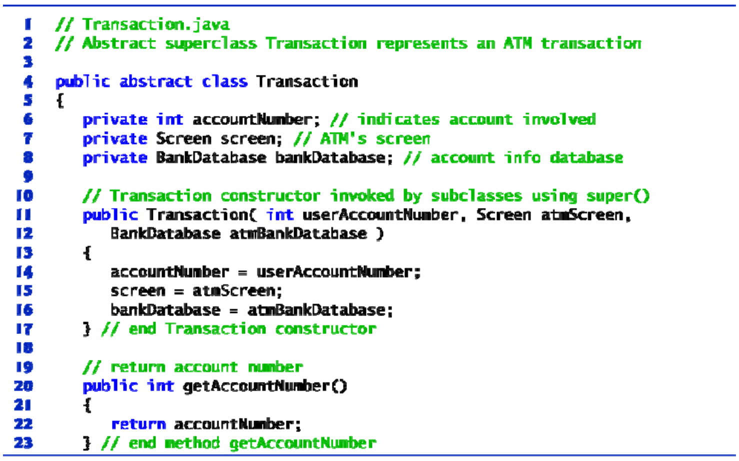
Fig. 13.19 | Class BankDatabase represents the bank's account information database. (Part 2 of 3.)



```
   // return available balance of Account with specified account number
   public double getAvailableBalance( int userAccountNumber )
   {
      return getAccount( userAccountNumber ).getAvailableBalance();
   } // end method getAvailableBalance

   // return total balance of Account with specified account number
   public double getTotalBalance( int userAccountNumber )
   {
      return getAccount( userAccountNumber ).getTotalBalance();
   } // end method getTotalBalance

   // credit an amount to Account with specified account number
   public void credit( int userAccountNumber, double amount )
   {
      getAccount( userAccountNumber ).credit( amount );
   } // end method credit

   // debit an amount from Account with specified account number
   public void debit( int userAccountNumber, double amount )
   {
      getAccount( userAccountNumber ).debit( amount );
   } // end method debit
} // end class BankDatabase
```

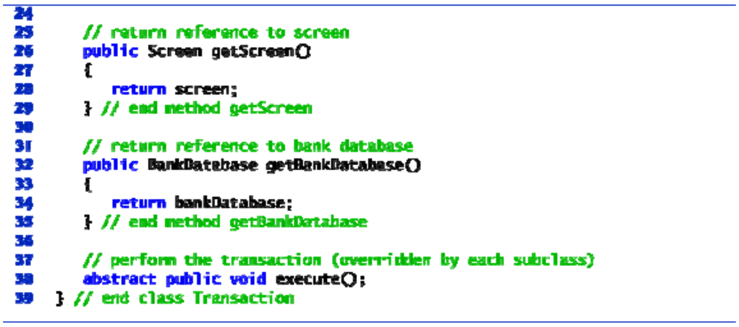
Fig. 13.19 | Class BankDatabase represents the bank's account information database. (Part 3 of 3.)

- Class `Transaction` (Fig. 13.20) is an abstract superclass that represents the notion of an ATM transaction.
- It contains the common features of subclasses `BalanceInquiry`, `Withdrawal` and `Deposit`.
- The class has three `public` *get* methods—`getAccountNumber` (lines 20–23), `getScreen` (lines 26–29) and `getBankDatabase` (lines 32–35).
  - These are inherited by `Transaction` subclasses and used to gain access to class `Transaction`'s `private` attributes.



```
 1  // Transaction.java
 2  // Abstract superclass Transaction represents an ATM transaction
 3
 4  public abstract class Transaction
 5  {
 6     private int accountNumber; // indicates account involved
 7     private Screen screen; // ATM's screen
 8     private BankDatabase bankDatabase; // account info database
 9
10     // Transaction constructor invoked by subclasses using super()
11     public Transaction( int userAccountNumber, Screen atmScreen,
12        BankDatabase atmBankDatabase )
13     {
14        accountNumber = userAccountNumber;
15        screen = atmScreen;
16        bankDatabase = atmBankDatabase;
17     } // end Transaction constructor
18
19     // return account number
20     public int getAccountNumber()
21     {
22        return accountNumber;
23     } // end method getAccountNumber
```

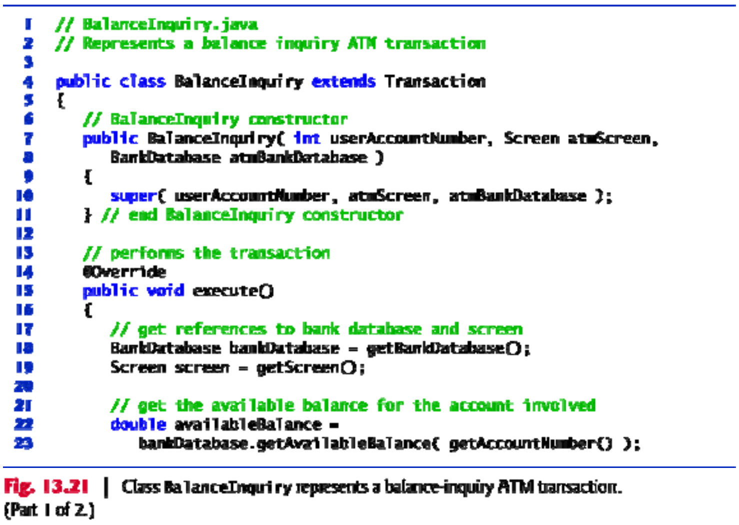
Fig. 13.20 | Abstract superclass `Transaction` represents an ATM transaction. (Part 1 of 2.)



```
    // return reference to screen
    public Screen getScreen()
    {
       return screen;
    } // end method getScreen

    // return reference to bank database
    public BankDatabase getBankDatabase()
    {
       return bankDatabase;
    } // end method getBankDatabase

    // perform the transaction (overridden by each subclass)
    abstract public void execute();
 } // end class Transaction
```

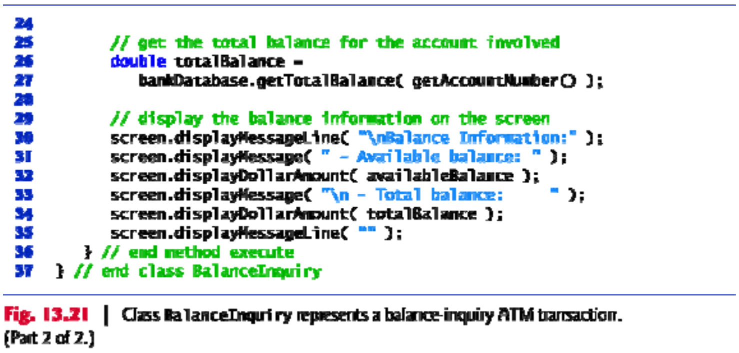
Fig. 13.20 | Abstract superclass `Transaction` represents an ATM transaction. (Part 2 of 2.)

- Class `BalanceInquiry` (Fig. 13.21) extends `Transaction` and represents a balance-inquiry ATM transaction.
- `BalanceInquiry` does not have any attributes of its own, but it inherits `Transaction` attributes `accountNumber`, `screen` and `bankDatabase`, which are accessible through `Transaction`’s `public` *get* methods.



```java
// BalanceInquiry.java
// Represents a balance inquiry ATM transaction

public class BalanceInquiry extends Transaction
{
   // BalanceInquiry constructor
   public BalanceInquiry( int userAccountNumber, Screen atmScreen,
      BankDatabase atmBankDatabase )
   {
      super( userAccountNumber, atmScreen, atmBankDatabase );
   } // end BalanceInquiry constructor

   // performs the transaction
   @Override
   public void execute()
   {
      // get references to bank database and screen
      BankDatabase bankDatabase = getBankDatabase();
      Screen screen = getScreen();

      // get the available balance for the account involved
      double availableBalance =
         bankDatabase.getAvailableBalance( getAccountNumber() );
```

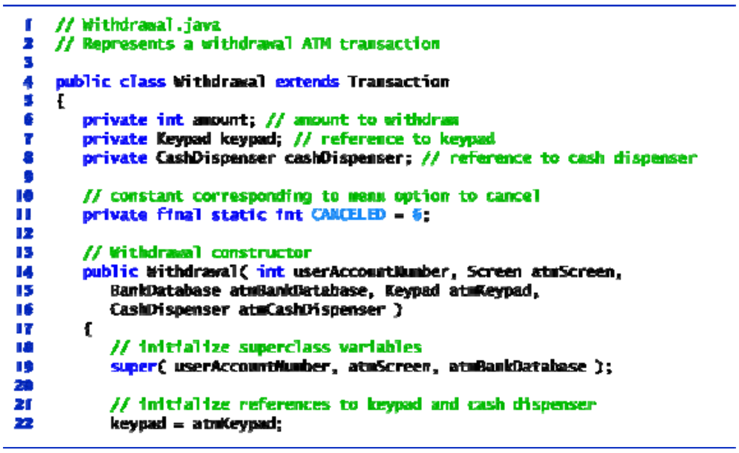
Fig. 13.21 | Class `BalanceInquiry` represents a balance-inquiry ATM transaction. (Part 1 of 2.)



```java

      // get the total balance for the account involved
      double totalBalance =
         bankDatabase.getTotalBalance( getAccountNumber() );

      // display the balance information on the screen
      screen.displayMessageLine( "\nBalance Information:" );
      screen.displayMessage( " - Available balance: " );
      screen.displayDollarAmount( availableBalance );
      screen.displayMessage( "\n - Total balance:     " );
      screen.displayDollarAmount( totalBalance );
      screen.displayMessageLine( "" );
   } // end method execute
} // end class BalanceInquiry
```

Fig. 13.21 | Class `BalanceInquiry` represents a balance-inquiry ATM transaction. (Part 2 of 2.)

- Class `Withdrawal` (Fig. 13.22) extends `Transaction` and represents a withdrawal ATM transaction.
- Figure 13.9 models associations between class `Withdrawal` and classes `Keypad` and `CashDispenser`, for which lines 7–8 implement reference-type attributes `keypad` and `cashDispenser`, respectively.



```
 1  // Withdrawal.java
 2  // Represents a withdrawal ATM transaction
 3
 4  public class Withdrawal extends Transaction
 5  {
 6     private int amount; // amount to withdraw
 7     private Keypad keypad; // reference to keypad
 8     private CashDispenser cashDispenser; // reference to cash dispenser
 9
10     // constant corresponding to menu option to cancel
11     private final static int CANCELED = 6;
12
13     // Withdrawal constructor
14     public Withdrawal( int userAccountNumber, Screen atmScreen,
15        BankDatabase atmBankDatabase, Keypad atmKeypad,
16        CashDispenser atmCashDispenser )
17     {
18        // initialize superclass variables
19        super( userAccountNumber, atmScreen, atmBankDatabase );
20
21        // initialize references to keypad and cash dispenser
22        keypad = atmKeypad;
```

Fig. 13.22 | Class `Withdrawal` represents a withdrawal ATM transaction. (Part 1 of 7.)

```java
      cashDispenser = atmCashDispenser;
   } // end Withdrawal constructor

   // perform transaction
   @Override
   public void execute()
   {
      boolean cashDispensed = false; // cash was not dispensed yet
      double availableBalance; // amount available for withdrawal

      // get references to bank database and screen
      BankDatabase bankDatabase = getBankDatabase();
      Screen screen = getScreen();

      // loop until cash is dispensed or the user cancels
      do
      {
         // obtain a chosen withdrawal amount from the user
         amount = displayMenuOfAmounts();

         // check whether user chose a withdrawal amount or canceled
         if ( amount != CANCELED )
         {
```

Fig. 13.22 | Class `Withdrawal` represents a withdrawal ATM transaction. (Part 2 of 7.)

```java
            // get available balance of account involved
            availableBalance =
               bankDatabase.getAvailableBalance( getAccountNumber() );

            // check whether the user has enough money in the account
            if ( amount <= availableBalance )
            {
               // check whether the cash dispenser has enough money
               if ( cashDispenser.isSufficientCashAvailable( amount ) )
               {
                  // update the account involved to reflect the withdrawal
                  bankDatabase.debit( getAccountNumber(), amount );

                  cashDispenser.dispenseCash( amount ); // dispense cash
                  cashDispensed = true; // cash was dispensed

                  // instruct user to take cash
                  screen.displayMessageLine( "\nYour cash has been" +
                     " dispensed. Please take your cash now." );
               } // end if
```

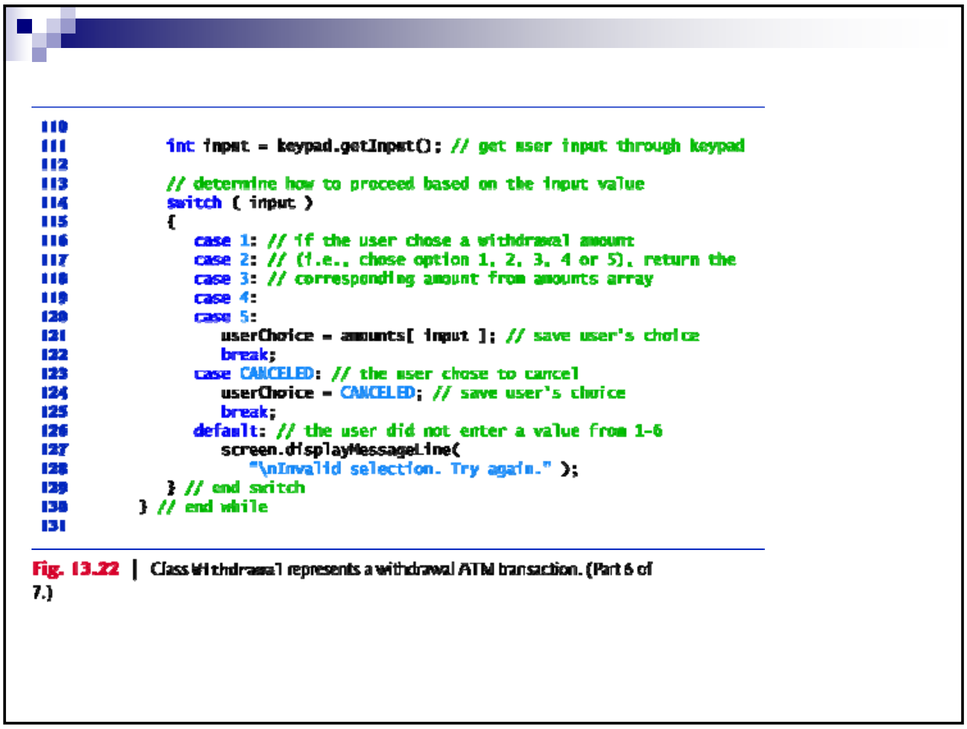
Fig. 13.22 | Class `Withdrawal` represents a withdrawal ATM transaction. (Part 3 of 7.)

```
               else // cash dispenser does not have enough cash
                  screen.displayMessageLine(
                     "\nInsufficient cash available in the ATM." +
                     "\n\nPlease choose a smaller amount." );
            } // end if
            else // not enough money available in user's account
            {
               screen.displayMessageLine(
                  "\nInsufficient funds in your account." +
                  "\n\nPlease choose a smaller amount." );
            } // end else
         } // end if
         else // user chose cancel menu option
         {
            screen.displayMessageLine( "\nCanceling transaction..." );
            return; // return to main menu because user canceled
         } // end else
      } while ( !cashDispensed );

   } // end method execute

```

Fig. 13.22 | Class Withdrawal represents a withdrawal ATM transaction. (Part 4 of 7.)

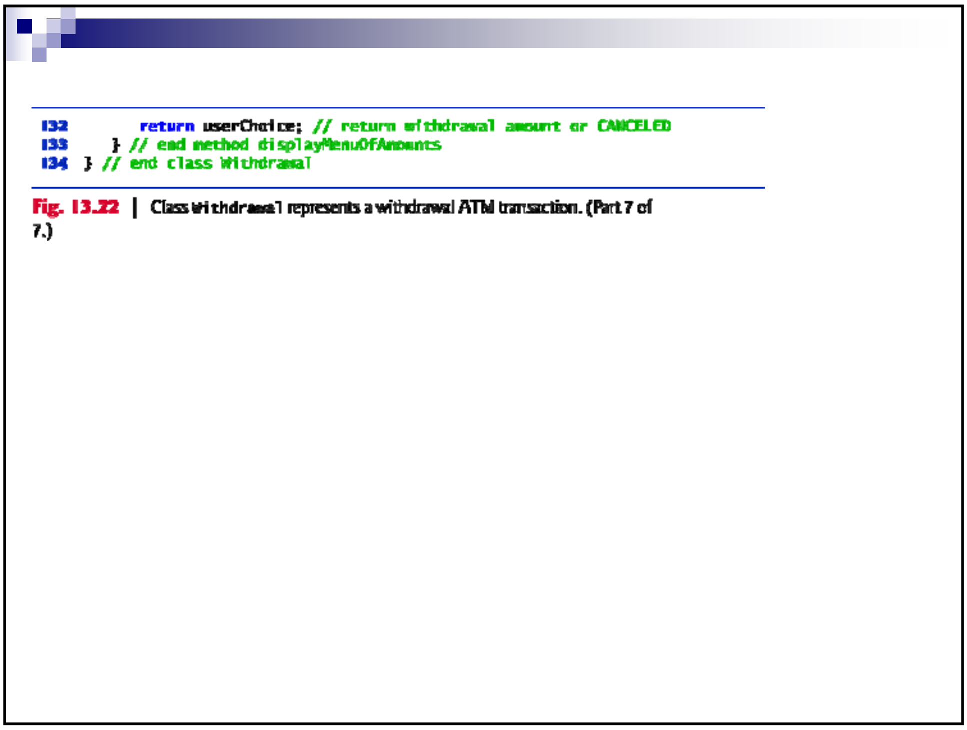
```
   // display a menu of withdrawal amounts and the option to cancel;
   // return the chosen amount or 0 if the user chooses to cancel
   private int displayMenuOfAmounts()
   {
      int userChoice = 0; // local variable to store return value

      Screen screen = getScreen(); // get screen reference

      // array of amounts to correspond to menu numbers
      int[] amounts = { 0, 20, 40, 60, 100, 200 };

      // loop while no valid choice has been made
      while ( userChoice == 0 )
      {
         // display the menu
         screen.displayMessageLine( "\nWithdrawal Menu:" );
         screen.displayMessageLine( "1 - $20" );
         screen.displayMessageLine( "2 - $40" );
         screen.displayMessageLine( "3 - $60" );
         screen.displayMessageLine( "4 - $100" );
         screen.displayMessageLine( "5 - $200" );
         screen.displayMessageLine( "6 - Cancel transaction" );
         screen.displayMessage( "\nChoose a withdrawal amount: " );
```

Fig. 13.22 | Class Withdrawal represents a withdrawal ATM transaction. (Part 5 of 7.)



```
         int input = keypad.getInput(); // get user input through keypad

         // determine how to proceed based on the input value
         switch ( input )
         {
            case 1: // if the user chose a withdrawal amount
            case 2: // (i.e., chose option 1, 2, 3, 4 or 5), return the
            case 3: // corresponding amount from amounts array
            case 4:
            case 5:
               userChoice = amounts[ input ]; // save user's choice
               break;
            case CANCELED: // the user chose to cancel
               userChoice = CANCELED; // save user's choice
               break;
            default: // the user did not enter a value from 1-6
               screen.displayMessageLine(
                  "\nInvalid selection. Try again." );
         } // end switch
      } // end while

```

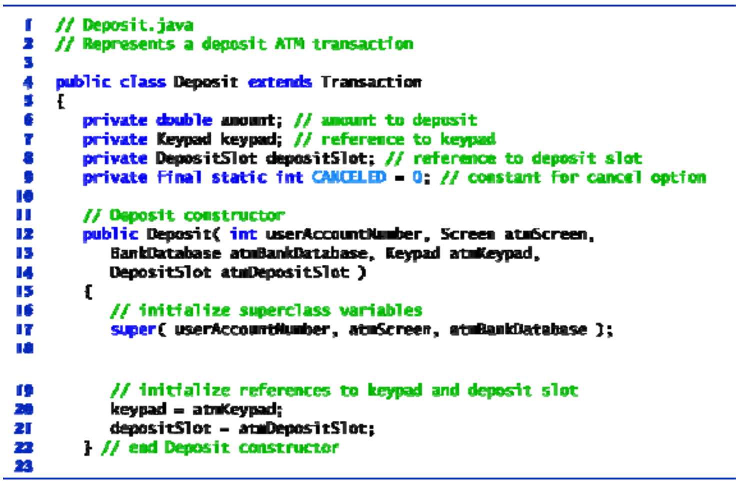
Fig. 13.22 | Class Withdrawal represents a withdrawal ATM transaction. (Part 6 of 7.)



```
      return userChoice; // return withdrawal amount or CANCELED
   } // end method displayMenuOfAmounts
} // end class Withdrawal
```

Fig. 13.22 | Class Withdrawal represents a withdrawal ATM transaction. (Part 7 of 7.)

- Class `Deposit` (Fig. 13.23) extends `Transaction` and represents a deposit transaction.
- Lines 7–8 create reference-type attributes `keypad` and `depositSlot` that implement the associations between class `Deposit` and classes `Keypad` and `DepositSlot` modeled in Fig. 13.9.
- Line 9 declares a constant `CANCELED` that corresponds to the value a user enters to cancel.



```
1  // Deposit.java
2  // Represents a deposit ATM transaction
3
4  public class Deposit extends Transaction
5  {
6     private double amount; // amount to deposit
7     private Keypad keypad; // reference to keypad
8     private DepositSlot depositSlot; // reference to deposit slot
9     private final static int CANCELED = 0; // constant for cancel option
10
11    // Deposit constructor
12    public Deposit( int userAccountNumber, Screen atmScreen,
13       BankDatabase atmBankDatabase, Keypad atmKeypad,
14       DepositSlot atmDepositSlot )
15    {
16       // initialize superclass variables
17       super( userAccountNumber, atmScreen, atmBankDatabase );
18
19       // initialize references to keypad and deposit slot
20       keypad = atmKeypad;
21       depositSlot = atmDepositSlot;
22    } // end Deposit constructor
23
```

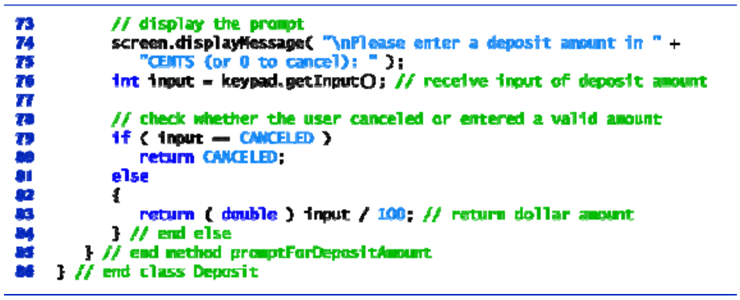
Fig. 13.23 | Class `Deposit` represents a deposit ATM transaction. (Part 1 of 4.)

```
   // perform transaction
   @Override
   public void execute()
   {
      BankDatabase bankDatabase = getBankDatabase(); // get reference
      Screen screen = getScreen(); // get reference

      amount = promptForDepositAmount(); // get deposit amount from user

      // check whether user entered a deposit amount or canceled
      if ( amount != CANCELED )
      {
         // request deposit envelope containing specified amount
         screen.displayMessage(
            "\nPlease insert a deposit envelope containing " );
         screen.displayDollarAmount( amount );
         screen.displayMessageLine( "." );

         // receive deposit envelope
         boolean envelopeReceived = depositSlot.isEnvelopeReceived();

         // check whether deposit envelope was received
         if ( envelopeReceived )
         {
            screen.displayMessageLine( "\nYour envelope has been " +
```

Fig. 13.23 | Class `Deposit` represents a deposit ATM transaction. (Part 2 of 4.)

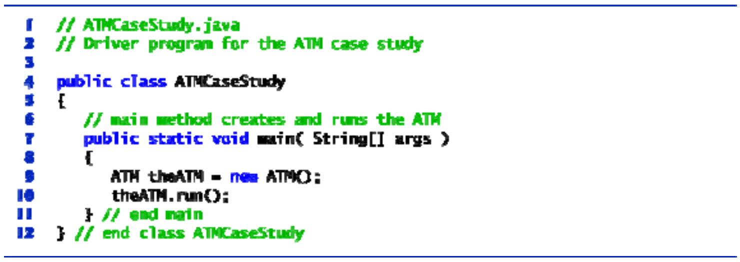
```
               "received.\nNOTE: The money just deposited will not " +
               "be available until we verify the amount of any " +
               "enclosed cash and your checks clear." );

            // credit account to reflect the deposit
            bankDatabase.credit( getAccountNumber(), amount );
         } // end if
         else // deposit envelope not received
         {
            screen.displayMessageLine( "\nYou did not insert an " +
               "envelope, so the ATM has canceled your transaction." );
         } // end else
      } // end if
      else // user canceled instead of entering amount
      {
         screen.displayMessageLine( "\nCanceling transaction..." );
      } // end else
   } // end method execute

   // prompt user to enter a deposit amount in cents
   private double promptForDepositAmount()
   {
      Screen screen = getScreen(); // get reference to screen

```

Fig. 13.23 | Class `Deposit` represents a deposit ATM transaction. (Part 3 of 4.)



```
      // display the prompt
      screen.displayMessage( "\nPlease enter a deposit amount in " +
         "CENTS (or 0 to cancel): " );
      int input = keypad.getInput(); // receive input of deposit amount

      // check whether the user canceled or entered a valid amount
      if ( input == CANCELED )
         return CANCELED;
      else
      {
         return ( double ) input / 100; // return dollar amount
      } // end else
   } // end method promptForDepositAmount
} // end class Deposit
```

Fig. 13.23 | Class `Deposit` represents a deposit ATM transaction. (Part 4 of 4.)



```
// ATMCaseStudy.java
// Driver program for the ATM case study

public class ATMCaseStudy
{
   // main method creates and runs the ATM
   public static void main( String[] args )
   {
      ATM theATM = new ATM();
      theATM.run();
   } // end main
} // end class ATMCaseStudy
```

Fig. 13.24 | `ATMCaseStudy.java` starts the ATM.

- Class `ATMCaseStudy` (Fig. 13.24) is a simple class that allows us to start, or “turn on,” the ATM and test the implementation of our ATM system model.