CMPS 251

Please read Chapter 8

Composition

Dr. Abdelkarim Erradi

CSE@QU

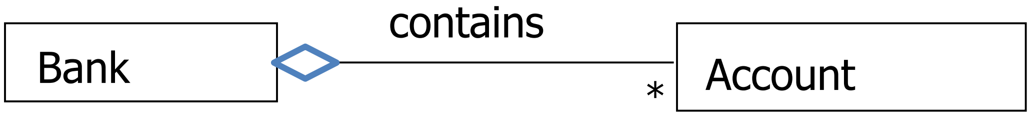
Outline

- Classes Composition
- Introduction to Arrays and Lists
- Enumeration
- Exceptions

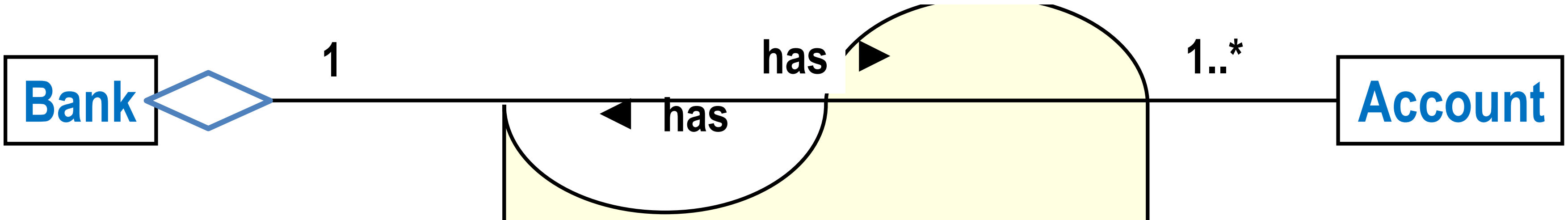
Classes Composition

Composition

- Composition is a capability that allows a class to have object attributes (i.e., instances of other classes)
  - This is also referred to as a has-a relationship.
- Example:
  - An account has an Owner (1 to 1 association)
  - A Bank has many accounts (1 to many association)
    - We say that Bank **aggregates** many Accounts

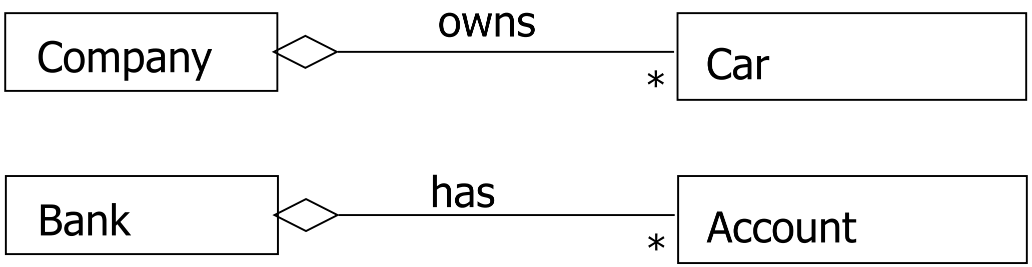


Implementation of bidirectional association



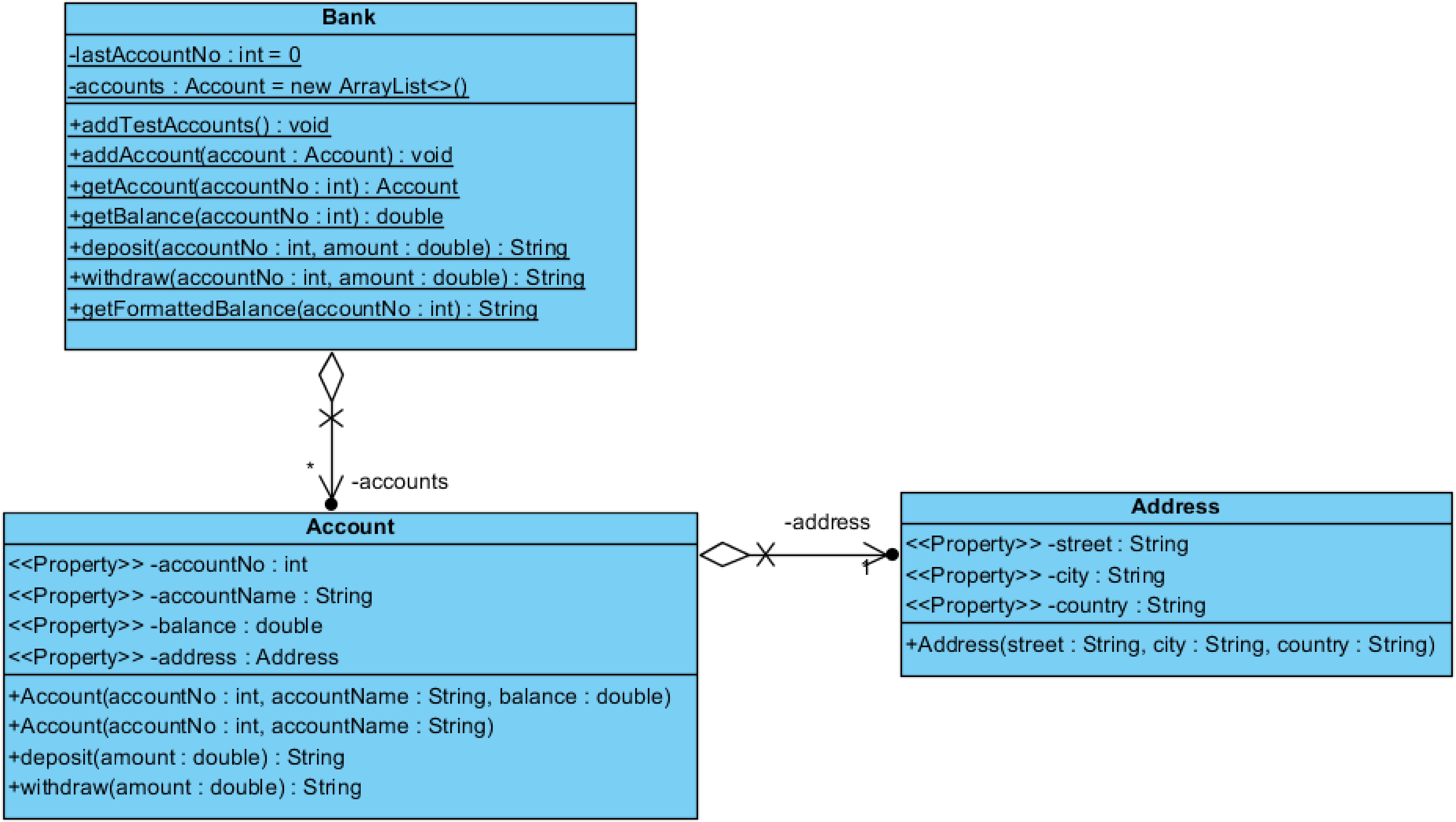
```
class Bank
{
    List<Account> accounts;
    . . . . .
}

class Account
{
    Bank bank;
    . . . . .
}
```

Aggregation



- Aggregation represents “whole-part” / “contains” relationship
  - – Part instances can be added to and removed from the aggregate
- The aggregate is represented in the Whole class using an **array** or an **ArrayList**

Composition Examples



Bank

-lastAccountNo : int = 0
-accounts : Account = new ArrayList<>()

+addTestAccounts() : void
+addAccount(account : Account) : void
+getAccount(accountNo : int) : Account
+getBalance(accountNo : int) : double
+deposit(accountNo : int, amount : double) : String
+withdraw(accountNo : int, amount : double) : String
+getFormattedBalance(accountNo : int) : String

Account

<<Property>> -accountNo : int
<<Property>> -accountName : String
<<Property>> -balance : double
<<Property>> -address : Address

+Account(accountNo : int, accountName : String, balance : double)
+Account(accountNo : int, accountName : String)
+deposit(amount : double) : String
+withdraw(amount : double) : String

Address

<<Property>> -street : String
<<Property>> -city : String
<<Property>> -country : String

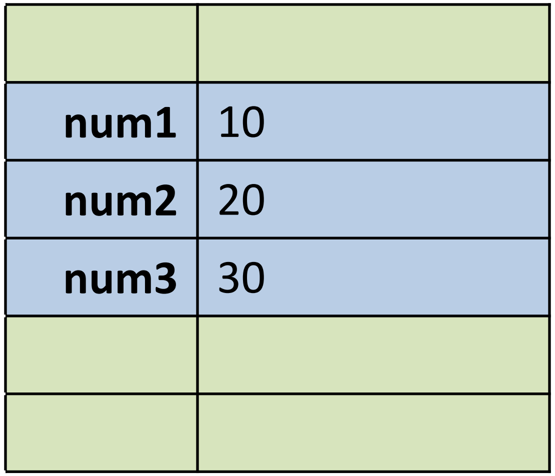
+Address(street : String, city : String, country : String)

Arrays and Lists

A simple variable stores a single value

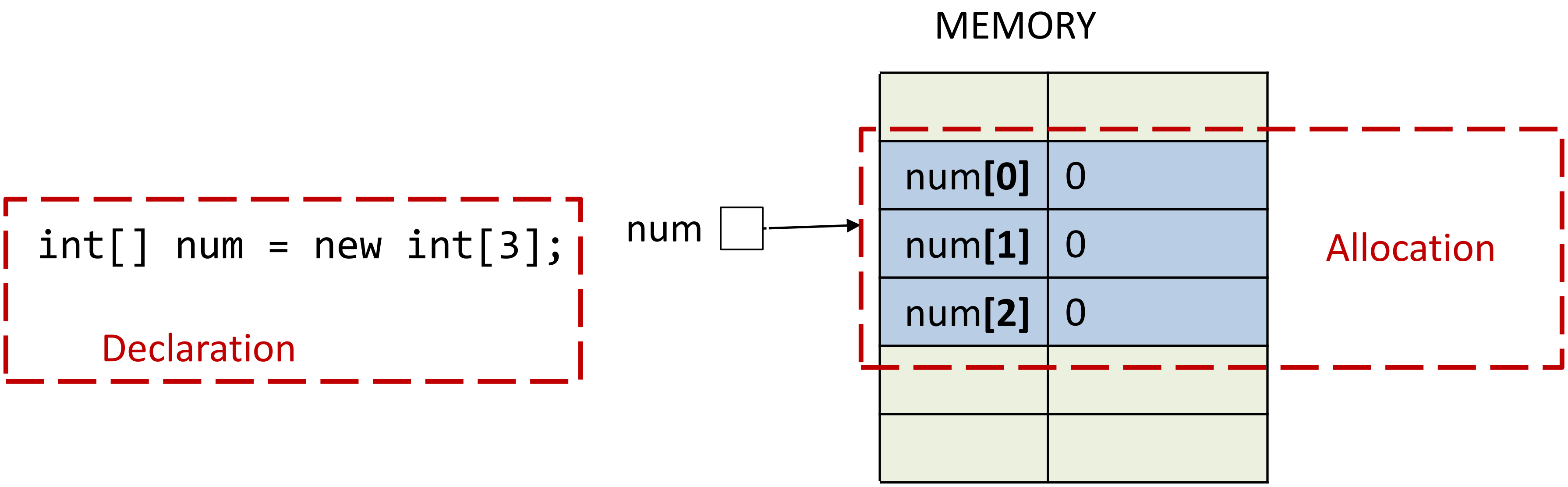
```
int num1 = 10;
int num2 = 20;
int num3 = 30;
```

MEMORY



num1	10
num2	20
num3	30

An array object stores multiple values



```
int[] num = new int[3];
```

Declaration

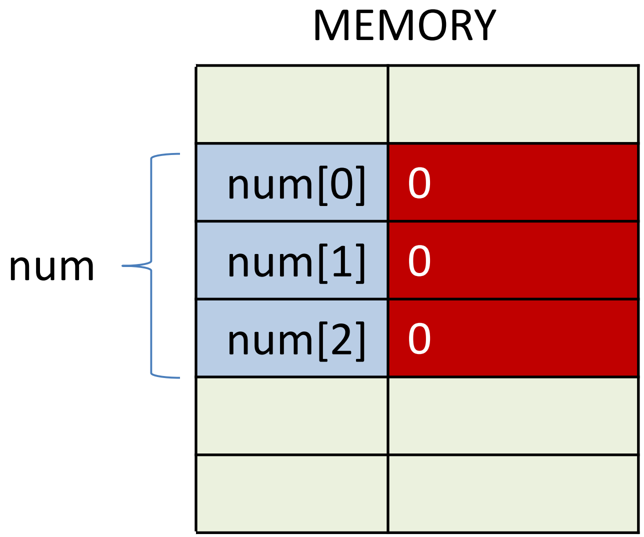
Array objects can hold any type of object

```
int[] number = new int[100];         // stores 100 integers

double[] salesTax = new double[10];  // stores 10 doubles

char[] alphabet = new char[26];      // stores 26 characters

Student[] students = new Student[40];    // stores 40 students
```

The size of the array determines the number of elements in the array.

Array elements are initialized to the type's default value

```
int[] = new num[3];
```



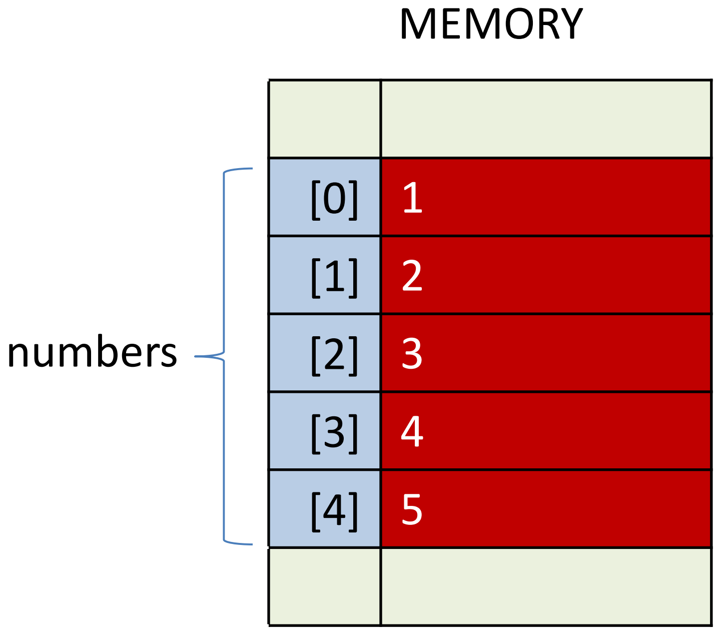
num

MEMORY

num[0]	0
num[1]	0
num[2]	0

You may initialize an array explicitly

```
int[] numbers = {1, 2, 3, 4, 5};
```



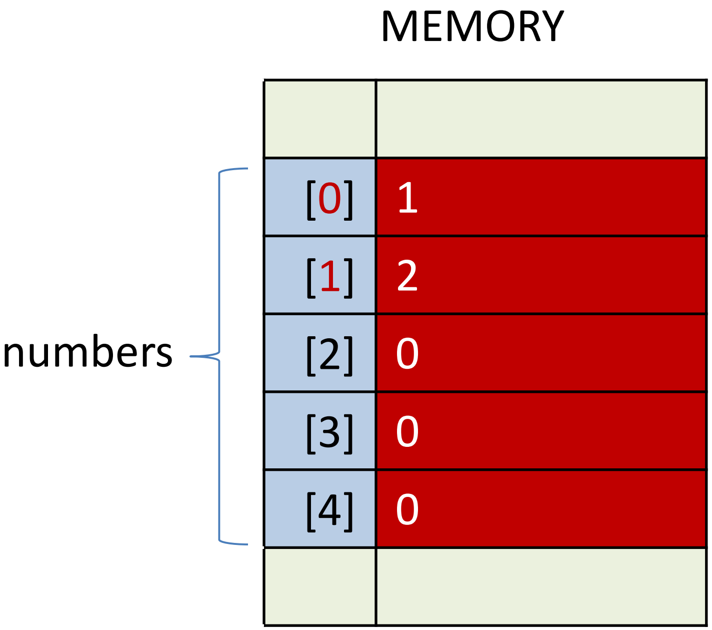
MEMORY

numbers

[0]	1
[1]	2
[2]	3
[3]	4
[4]	5

Array elements are indexed

```
int[] numbers = new int[5];
```



```
numbers[0] = 1;
numbers[1] = 2;
```

Arrays can be instance variables

```
public class Department {
    private Employee[] employee;
    ...
}
```

Arrays can be local variables

```
public void getHourlyEmployees() {
    Employee[] hourlyEmployee;
    ...
}
```

Arrays can be parameters

```
public static void main(String[] args) {
    ...
}
```

Arrays can be return values

```
public Employee[] getEmployees() {
    ...
}
```

Example - Method that returns an array

```
public int[] initArray(int size, int initValue) {
   int[] array = new int[size];

   for (int i = 0; i < array.length; i++) {
      array[i] = initValue;
   }

   return array;
}
```

Arrays are objects, thus

```
int[] a = {1, 2, 3};
int[] b;

b = a;                  // makes b and a refer to the same
                        // memory location
```

Example method that copies an array

```
public void copyArray(int[] source, int[] target) {
   // Both arrays must be the same size.
   target = new int[source.length];
   for (int i = 0; i < source.length; i++) {
      target[i] = source[i];
   }

}
```

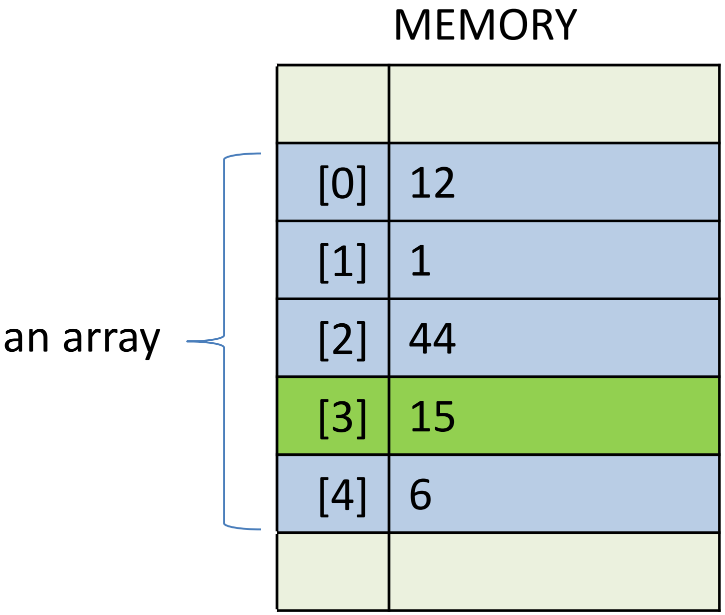
Arrays are objects, thus

```
int[] a = {1, 2, 3};
int[] b = {1, 2, 3};

if (a == b) {...}          // evaluates to false
                           // since a and b refer to two
                           // different memory locations
```

Example - Method that tests for array equality

```
public boolean areEqual(int[] array1, int[] array2) {
   if (array1.length != array2.length) {
      return false;
   } else {
      for(int i = 0; i < array1.length; i++) {
         if(array1[i] != array2[i])
             return false;
      }// end for
   }// end if
   return true;
}
```

Use linear(sequential) search to locate values



MEMORY

[0]	12
[1]	1
[2]	44
[3]	15
[4]	6

an array

Q: is this the value? A: No

Q: is this the value? A: No

Q: is this the value? A: No

Q: is this the value? A: Yes

Q: is this the value 15 in the array?

Linear Search

```
// Returns true if array contains item, false otherwise.
private boolean contains(String[] items, String item) {
  for(int i = 0; i < items.length; i++) {
         if (items[i].equalsIgnoreCase(item)) {
            return true;
        }
    }// end for
    return false;
}
```

Lists

- Problem
  - You must know the array size when you create the array
    - Although Java arrays are better than C++ arrays since the size does not need to be a compile-time constant
  - Array size cannot change once created.
- Solution:
  - Use **ArrayList**: they stretch as you add elements to them

Syntax: ArrayList

- Summary of operations
  - Create empty list
    - **`new ArrayList<>()`**
  - Add entry to end
    - **`add(value)`** (adds to end)
  - Retrieve *n*th element
    - **`get(index)`**
  - Check if element exists in list
    - **`contains(element)`**
  - Remove element
    - **`remove(index)`** or **`remove(element)`**
  - Find the number of elements
    - **`size()`**

ArrayList Example

```
import java.util.*;  // Don't forget this import

public class ListTest2 {
  public static void main(String[] args) {
    List<String> entries = new ArrayList<>();
    double d;
    while((d = Math.random()) > 0.1) {
      entries.add("Value: " + d);
    }
    for(String entry: entries) {
      System.out.println(entry);
    }
  }
}
```

This tells Java your list will contain only strings.

ArrayList Example: Output

```
> java ListTest2
Value: 0.6374760850618444
Value: 0.9159907384916878
Value: 0.8093728146584014
Value: 0.7177611068808302
Value: 0.9751541794430284
Value: 0.2655587762679209
Value: 0.3135791999033012
Value: 0.44624152771013836
Value: 0.7585420756498766
```

Enumeration

```
enum LightState {...}
```

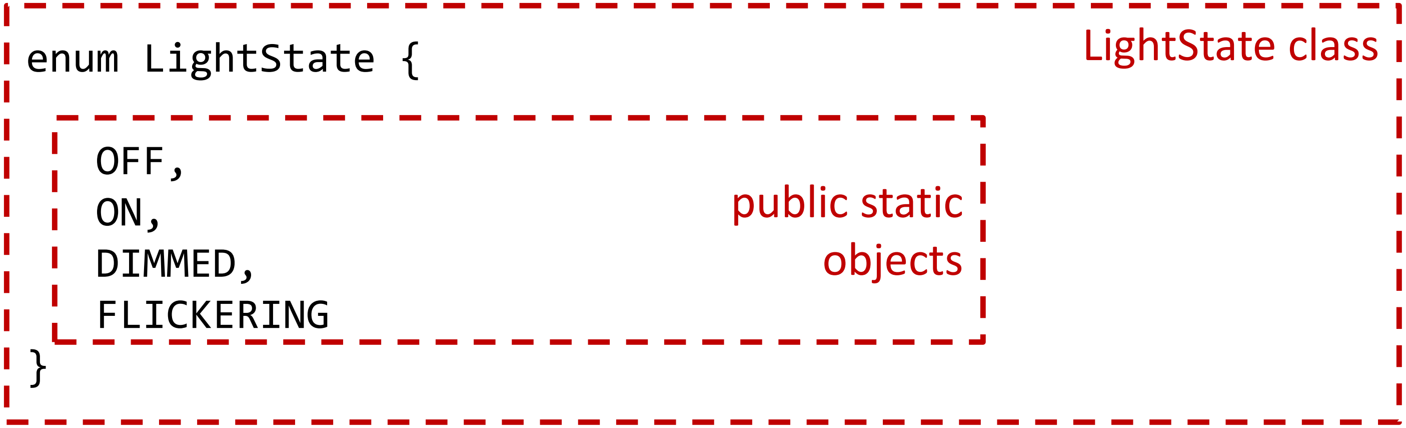
Enumerations

- The basic `enum` type defines a set of constants represented as unique identifiers
- An `enum` type is declared with an enum declaration, which is a comma-separated list of `enum` constants
- The declaration may optionally include other components of traditional classes, such as constructors, fields and methods

Enumerations (Cont.)

- Each `enum` declaration declares an `enum` class with the following restrictions:
  - `enum` constants are implicitly `final`, because they declare constants that shouldn’t be modified.
  - `enum` constants are implicitly `static`.
  - Any attempt to create an object of an enum type with operator `new` results in a compilation error.
  - `enum` constants can be used anywhere constants can be used, such as in the `case` labels of `switch` statements and the condition of an if statement.
- For every `enum`, the compiler generates the `static` method values that returns an array of the `enum`’s constants.
- When an `enum` constant is converted to a `String`, the constant’s identifier is used as the `String` representation.

`enum` is actually a class



```
enum LightState {
    OFF,
    ON,
    DIMMED,
    FLICKERING
}
```

LightState class

public static objects

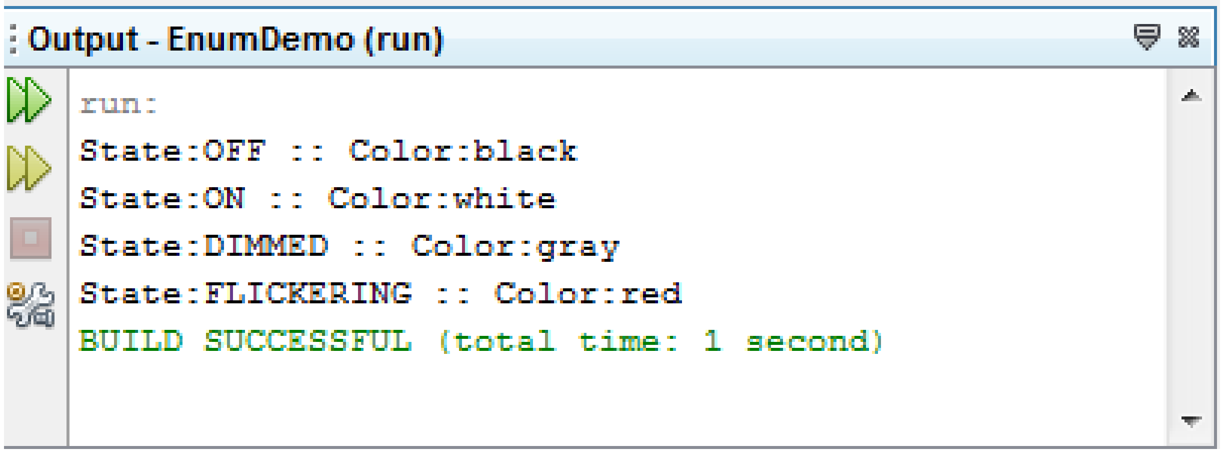
<EnumDemo.java>

```
public class EnumDemo {
    enum LightState {
        // Each object is initialized to a color.
        OFF("black"),
        ON("white"),
        DIMMED("gray"),
        FLICKERING("red");

        private final String colorField;

        // Private constructor to set the color.
        private LightState(String color) {
            colorField = color;
        }

        // Public accessor to get color.
        public String getColor() {
            return colorField;
        }
}

    public static void main(String[] args) {
        LightState off = LightState.OFF;
        LightState on = LightState.ON;
        LightState dimmed = LightState.DIMMED;
        LightState flickering = LightState.FLICKERING;
```

You can enhance the enum class with instance variables and methods

<EnumDemo.java>

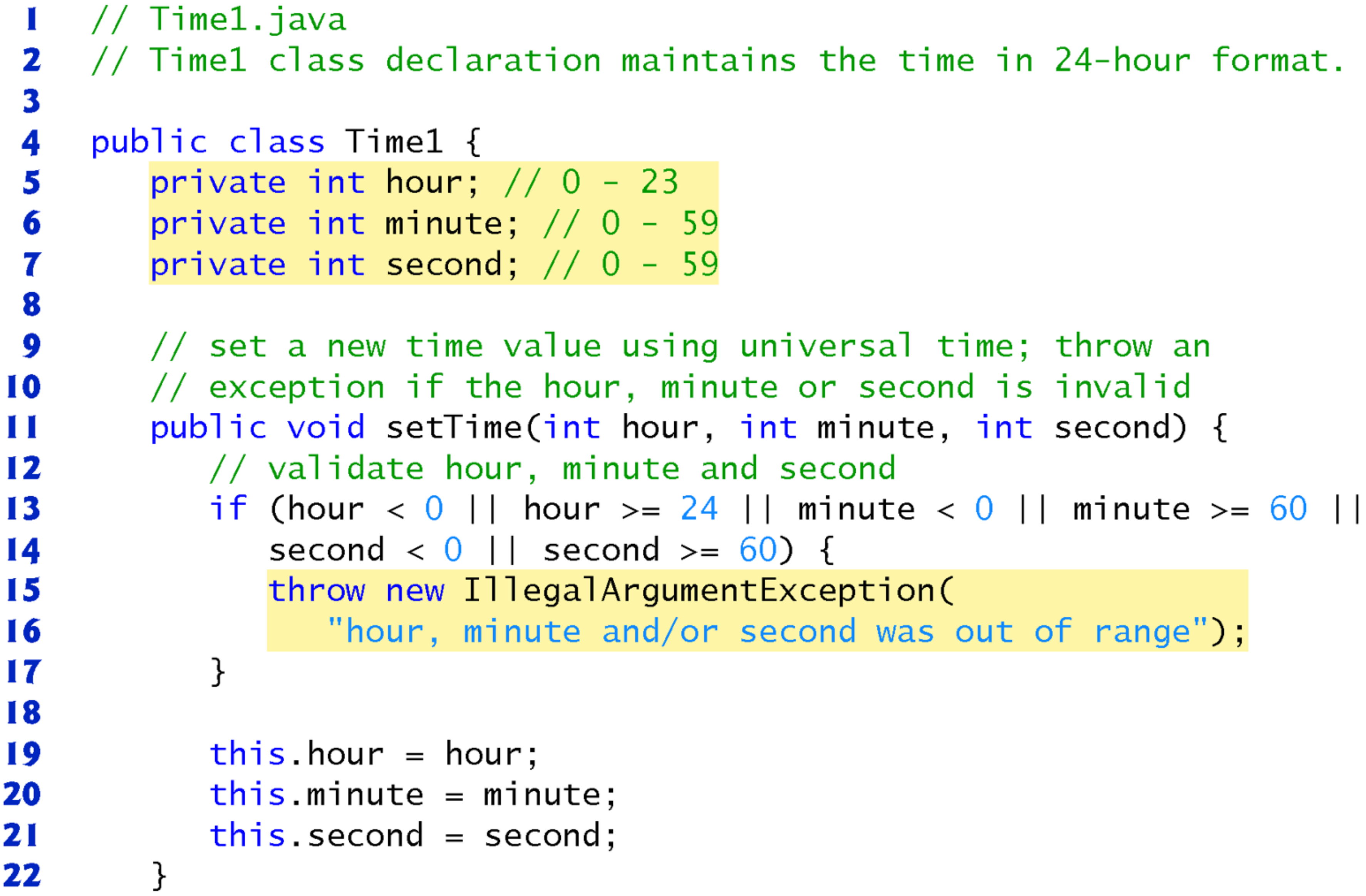
```
        System.out.println("State:" + off.toString() +
                           " :: Color:" + off.getColor());
        System.out.println("State:" + on.toString() +
                           " :: Color:" + on.getColor());
        System.out.println("State:" + dimmed.toString() +
                           " :: Color:" + dimmed.getColor());
        System.out.println("State:" + flickering.toString() +
                           " :: Color:" + flickering.getColor());
    }// end main()
}// end EnumDemo
```



```
run:
State:OFF :: Color:black
State:ON :: Color:white
State:DIMMED :: Color:gray
State:FLICKERING :: Color:red
BUILD SUCCESSFUL (total time: 1 second)
```

Exceptions

Throwing Exceptions



```
// Time1.java
// Time1 class declaration maintains the time in 24-hour format.

public class Time1 {
   private int hour; // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59

   // set a new time value using universal time; throw an
   // exception if the hour, minute or second is invalid
   public void setTime(int hour, int minute, int second) {
      // validate hour, minute and second
      if (hour < 0 || hour >= 24 || minute < 0 || minute >= 60 ||
         second < 0 || second >= 60) {
         throw new IllegalArgumentException(
            "hour, minute and/or second was out of range");
      }

      this.hour = hour;
      this.minute = minute;
      this.second = second;
   }
```

Throwing Exceptions

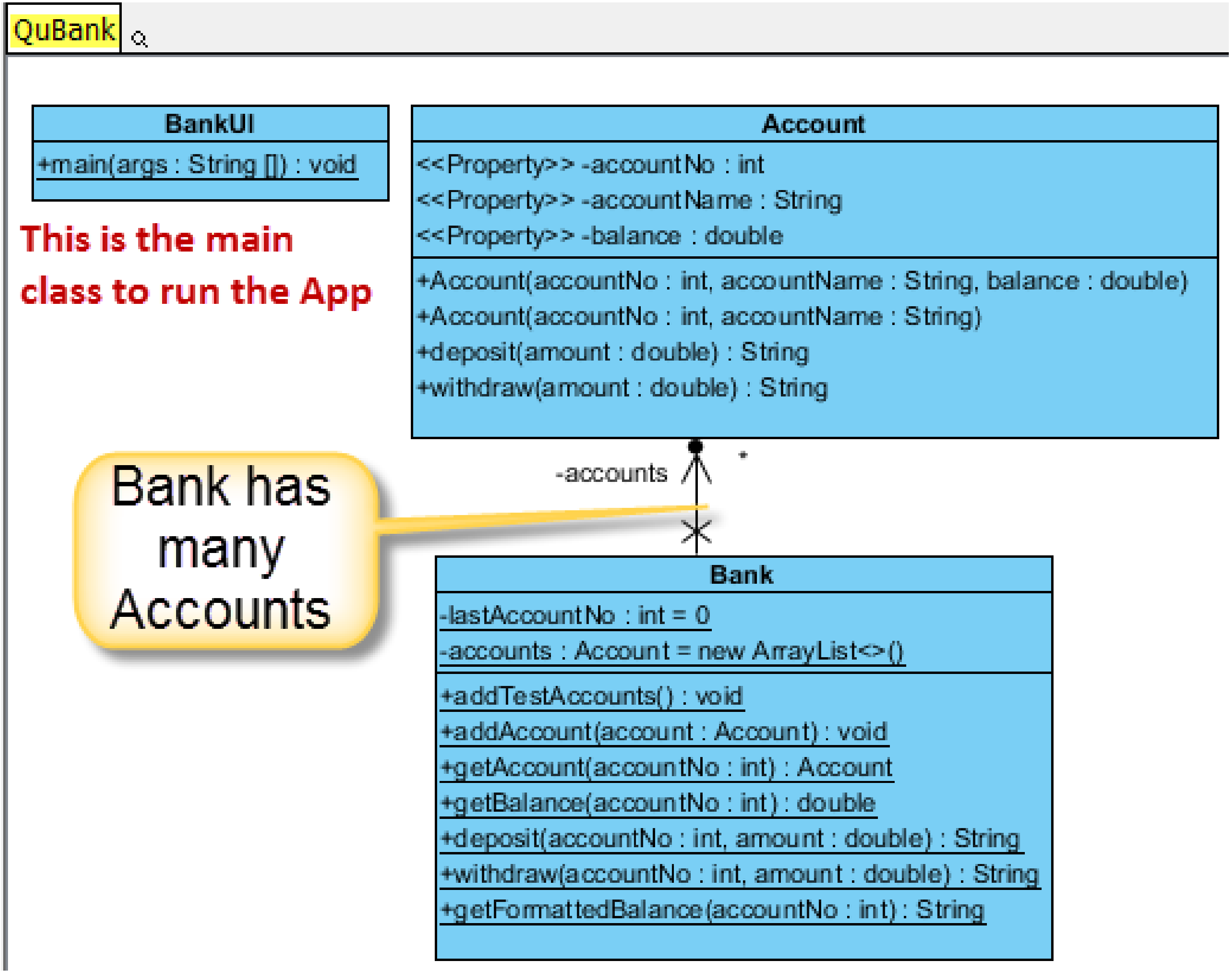
- Method `setTime` declares three `int` parameters and uses them to set the time.
- Lines 13–14 test each argument to determine whether the value is outside the proper range.
- For incorrect values, `setTime` throws an exception of type `IllegalArgumentException`
  - Notifies the client code that an invalid argument was passed to the method.
  - The `throw` statement creates a new object of type **`IllegalArgumentException`** and specifies a custom error message.
  - `throw` statement immediately terminates method `setTime` and the exception is returned to the calling method that attempted to set the time.

try and catch

```
18         // attempt to set time with invalid values
19         try {
20            time.setTime(99, 99, 99); // all values out of range
21         }
22         catch (IllegalArgumentException e) {
23            System.out.printf("Exception: %s%n%n", e.getMessage());
24         }
25
26         // display time after attempt to set invalid values
27         displayTime("After calling setTime with invalid values", time);
28      }
29
30      // displays a Time1 object in 24-hour and 12-hour formats
31      private static void displayTime(String header, Time1 t) {
32         System.out.printf("%s%nUniversal time: %s%nStandard time: %s%n",
33            header, t.toUniversalString(), t.toString());
34      }
35   }
```

Lines 19 to 24 use **try…catch** to catch and handle the exception (e.g., display the error message to the user)

Banking System Example



QuBank

BankUI

+main(args : String []) : void

Account

<<Property>> -accountNo : int
<<Property>> -accountName : String
<<Property>> -balance : double

+Account(accountNo : int, accountName : String, balance : double)
+Account(accountNo : int, accountName : String)
+deposit(amount : double) : String
+withdraw(amount : double) : String

Bank

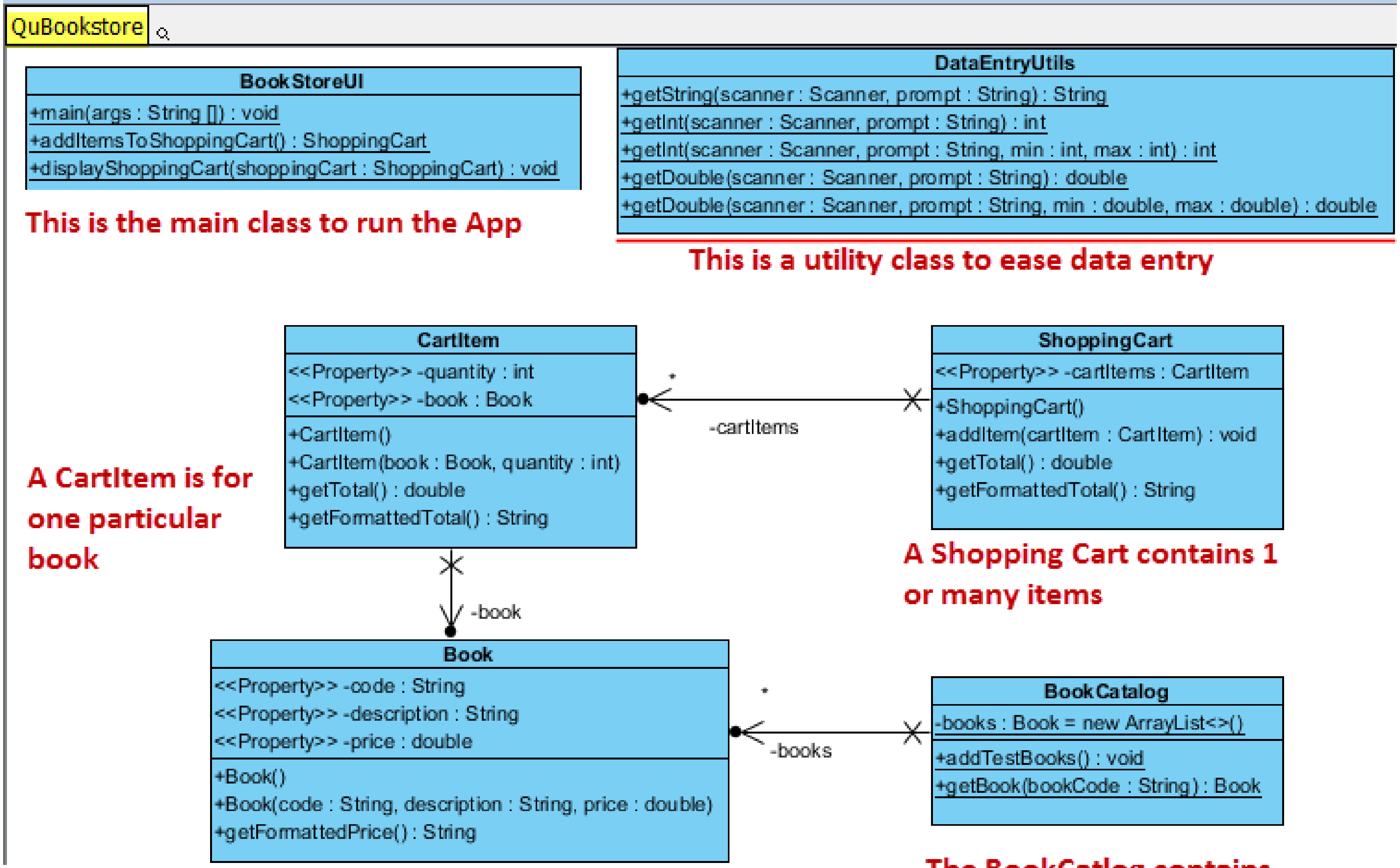
-lastAccountNo : int = 0
-accounts : Account = new ArrayList<>()

+addTestAccounts() : void
+addAccount(account : Account) : void
+getAccount(accountNo : int) : Account
+getBalance(accountNo : int) : double
+deposit(accountNo : int, amount : double) : String
+withdraw(accountNo : int, amount : double) : String
+getFormattedBalance(accountNo : int) : String

This is the main class to run the App

Bank has many Accounts

Bookstore System example



QuBookstore

BookStoreUI

+main(args : String []) : void
+addItemsToShoppingCart() : ShoppingCart
+displayShoppingCart(shoppingCart : ShoppingCart) : void

This is the main class to run the App

DataEntryUtils

+getString(scanner : Scanner, prompt : String) : String
+getInt(scanner : Scanner, prompt : String) : int
+getInt(scanner : Scanner, prompt : String, min : int, max : int) : int
+getDouble(scanner : Scanner, prompt : String) : double
+getDouble(scanner : Scanner, prompt : String, min : double, max : double) : double

This is a utility class to ease data entry

CartItem

<<Property>> -quantity : int
<<Property>> -book : Book

+CartItem()
+CartItem(book : Book, quantity : int)
+getTotal() : double
+getFormattedTotal() : String

ShoppingCart

<<Property>> -cartItems : CartItem

+ShoppingCart()
+addItem(cartItem : CartItem) : void
+getTotal() : double
+getFormattedTotal() : String

Book

<<Property>> -code : String
<<Property>> -description : String
<<Property>> -price : double

+Book()
+Book(code : String, description : String, price : double)
+getFormattedPrice() : String

BookCatalog

-books : Book = new ArrayList<>()

+addTestBooks() : void
+getBook(bookCode : String) : Book

A CartItem is for one particular book

A Shopping Cart contains 1 or many items

The BookCatlog contains many books