Introduction to Classes and Objects

CS102A Lecture 7

James YU
yujq3@sustech.edu.cn

Department of Computer Science and Engineering
Southern University of Science and Technology

Oct. 26, 2020

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Objectives

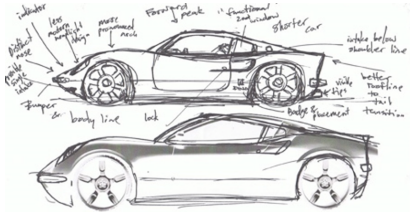
- Understand *classes*, *objects*, *instance variables*.
- Learn to declare a class and use it to create an object.
- Learn to declare non-static methods to implement the class's behavior.
- Learn to declare instance variables to implement the class's attributes.
- Learn to use a constructor to ensure that an object's data is initialized when the object is created.

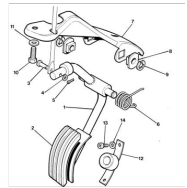
Introduction

- Typically, Java applications consist of one or more classes, each containing one or more methods.
  - E.g., `Welcome1`, `Welcome2`, `Addition`, `Comparison` ...
- In this chapter, we present a simple framework for organizing object-oriented applications in Java.
- We begin with an analogy to introduce classes and their contents.

Classes, objects, methods

- To drive a car and accelerate it by pressing down on its accelerator pedal.
  - Before you can drive a car, someone has to design it (engineering drawings/blueprints).
  - Including the design for an accelerator pedal.
  - A lot more designs, e.g., the brake pedal, the steering wheel.
  - **We don't need to know the complex mechanisms behind the design to drive the car.**





Classes, objects, methods

- We cannot drive a car's engineering drawings.
  - Before we drive, it must be **built from the engineering drawings**.
  - Even building a car is not enough, the driver must **press the accelerator pedal** to perform the task of drive the car.
- Three key concepts in Java:
  - *class* – a car's engineering drawings (blueprint),
  - *method* – designed to perform tasks (make a car move),
  - *object* – the car we drive.

Classes, objects, methods

- When programming Java, we begin by creating a program unit called `class`, just like we begin with engineering draws in the driving example.
- In a `class`, we provide one or more *methods* that are designed to perform the class' tasks. The method hides from its user the complex tasks that it performs, just like the accelerator pedal of a car hides from the driver the complex mechanisms that make the car move faster.

Classes, objects, methods

- We cannot drive a car's engineering drawings
- Similarly, we cannot "drive" a `class` to perform a task.
- Just as we have to build a car from its engineering drawings before driving it, we must *build an object* of a `class` before getting a program to perform tasks.
- This is one reason why Java is called an "object-oriented" programming language.

Classes, objects, methods

- When driving a car, pressing the accelerator pedal **sends a message to the car** to perform a task –- make the car go faster.
- Similarly, we **send a message to an object** –- implemented as a method call that tells a method of the object to perform its task.

Instance variables

- A car can have many *attributes*, such as its color, the number of doors, the amount of gas in its tank, its current speed, and the total miles driven.
  - These attributes are represented as part of a car's design in its engineering diagrams.
  - As you drive a car, **these attributes are always associated with the car** (not other cars of the same model).
  - Every car maintains its own attributes (e.g., knowing how much gas is left in its tank, but do not know about other cars).
- Similarly, an object has *attributes* that are carried with the *object* as it's used in a program.
  - These attributes are specified as the class' instance variables. E.g., a bank account object has a balance attribute (implemented as an instance variable) that represents the amount of money in that account.

The whole picture

- *Class* –a car's engineering drawings (a blueprint);
- *Method* –designed to perform tasks (e.g., making a car move);
- *Object* –the car we drive;
- *Method* call –perform the task (e.g., pressing the accelerator pedal)
- *Instance variable* –to specify the attributes (e.g., the amount of gas).

Declaring a `class`

- Every class declaration contains the keyword `class` + the class' name.
- The access modifier `public` indicates that the declared `class` is visible to all `class`es everywhere.

```
1 public class GradeBook {
2   // every class' body is enclosed in a pair of
3   // left and right curly braces
4 }
```

Declaring a *method*

- A class usually consists of one or more *method*s.
- The access modifier `public` indicates that the method is “available to public”, that is, can be called from the methods of other classes.

```java
public class GradeBook {
  // display welcome message to the user
  public void displayMessage() {
    System.out.println("Welcome to the Grade Book!");
  }
}
```

Object creation and method calling

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

```java
public class GradeBookTest {
  public static void main(String[] args) {
    // create a GradeBook object - assign it to myGradeBook
    GradeBook myGradeBook = new GradeBook();

    // call myGradeBook's displayMessage method
    myGradeBook.displayMessage();
  }
}
```

- `GradeBook myGradeBook`: define a variable of the type `GradeBook`. Note that each new class you create becomes a new type, this is why Java is an extensible language.

Object creation and method calling

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

```java
public class GradeBookTest {
  public static void main(String[] args) {
    // create a GradeBook object - assign it to myGradeBook
    GradeBook myGradeBook = new GradeBook();

    // call myGradeBook's displayMessage method
    myGradeBook.displayMessage();
  }
}
```

- `new GradeBook()`: class instance creation expression. The keyword `new` is used to create a new object of the specified class. Class name + `()` represent a call to a *constructor* (a special method used to initialize the object's data).

Object creation and method calling

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

```java
public class GradeBookTest {
  public static void main(String[] args) {
    // create a GradeBook object - assign it to myGradeBook
    GradeBook myGradeBook = new GradeBook();

    // call myGradeBook's displayMessage method
    myGradeBook.displayMessage();
  }
}
```

- We can use the variable `myGradeBook` to refer to the created object and that we call the method `displayMessage()`. The empty parentheses indicate "**provide no additional data (arguments) to the called method**".

More on instance variables

- An object has *attribute*s (e.g., the amount of gas of a car) that are carried with the object as it is used in a program.
- Such attributes exist before a method is called on an object and after the method completes execution.
- A class typically consists of one or more *method*s that manipulate the attributes that belong to a particular object of the class.
- Attributes are represented as variables in a class declaration.

More on instance variables

```java
public class GradeBook {
  private String courseName;
  public void displayMessage(String courseName) {
    System.out.println("Welcome to the Grade Book for the course%s!\n",
        courseName);
  }
}
```

- Object attributes are represented as *variable*s (called *field*s) in a class declaration.
- Each *object* (*instance*) of the class has its own copy of an attribute in memory, the *field* that represents the attribute is also know as an instance variable.

Don’t confuse with local variables

```java
public class GradeBookTest {
  public static void main(String[] args) {
    // create a GradeBook object
    // assign it to myGradeBook
    GradeBook myGradeBook = new GradeBook();

    // call myGradeBook's displayMessage method
    myGradeBook.displayMessage();
  }
}
```

- Variables declared in the body of a particular method are known as *local variable*s and can be only used in that method.
- Instance variables are declared inside a class declaration, but outside the bodies of the class’ method declarations.

Manipulating instance variables

```java
public class GradeBook {
  private String courseName;
  // method to set the course name
  public void setCourseName(String name) {
    courseName = name;
  }
  // method to retrieve the course name
  public String getCourseName() {
    return courseName;
  }
}
```

- Most instance variables are declared to be `private` (*data hiding*). Variables (or methods) declared to be `private` are accessible only to methods of the class in which they are declared.

Using *getter* and *setter*

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

```
1  import java.util.Scanner;
2  public class GradeBookTest {
3    public static void main(String[] args) {
4      GradeBook myGradeBook = new GradeBook();
5      Scanner input = new Scanner(System.in);
6      System.out.printf("Enter course name: ");
7      String theName = input.nextLine();
8      myGradeBook.setCourseName(theName);
9      myGradeBook.displayMessage();
10   }
11 }
12 public class GradeBook {
13   ...
14   public void displayMessage() {
15     System.out.printf("Welcome to the grade book for\n%s!\n",
           getCourseName());
16   }
17 }
```

Initializing objects with constructors

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

```
1 GradeBook myGradeBook = new GradeBook();
```

- Each class you declare can provide a special method called a *constructor* that can be used to initialize an object of a class when the object is created.
- Java requires a *constructor* call for every object that is created.
- Keyword new requests memory from the system to store an object, then calls the corresponding class's constructor to initialize the object.

Initializing objects with constructors

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

```
1 GradeBook myGradeBook = new GradeBook();
```

- The empty parentheses after `new GradeBook` indicate a call to the class' constructor without arguments.
- The compiler provides a *default constructor* with no parameters in any class that does not explicitly include a constructor.
  - When a class has only the default constructor, its instance variables are initialized with default values (e.g., an `int` variable gets the value `0`).
- When you declare a class, you can provide your own constructor to specify custom initialization for objects of your class.

Initializing objects with constructors

```java
public class GradeBook {
  private String courseName; // course name of this Gradebook

  // constructor initialize
  public GradeBook(String name) {
    courseName = name;
  } // end constructor
}
```

- The modified `GradeBook` class contain a constructor that receives an argument.
- Like a method, a constructor's parameter list specifies the data it requires to perform its task.

Initializing objects with constructors

```java
public class GradeBookTest {
  public static void main(String[] args) {
    GradeBook gradeBook1 = new GradeBook("CS101");
    GradeBook gradeBook2 = new GradeBook("CS102");

    System.out.printf("gradeBook1 course name is %s%n", gradeBook1.
        getCourseName());
    System.out.printf("gradeBook2 course name is %s%n", gradeBook2.
        getCourseName());
  }
}
```

Initializing objects with constructors

- An important difference between constructors and methods is that constructors **cannot return values**, so they cannot specify a return type (not even `void`).
- Normally, constructors are declared `public`.
- If you declare any constructors for a class, the Java compiler will not create a default constructor for the class.

More on *default constructor*

- Can we write the following statement to create a GradeBook object?

```
GradeBook myGradeBook = new GradeBook();
```

```
public class GradeBook {
  // no constructor provided
  private String courseName;
  public void setCourseName(String
      name) {
    courseName = name;
  }
  public String getCourseName() {
    return courseName;
  }
  ...
}
```

```
public class GradeBook {
  // this version has a constructor
  private String courseName;
  public GradeBook(String name) {
    courseName = name;
  }
  public void setCourseName(String
      name) {
    courseName = name;
  }
  ...
}
```

Case study: Account balances

- We define a class named `Account` to maintain the balance of a bank account.

```java
// Account class with a double instance variable balance and a
// constructor and deposit method that perform validation.
public class Account {
  private String name; // instance variable
  private double balance; // instance variable

  // Account constructor that receives two parameters
  public Account(String name, double balance)     {
      this.name = name; // assign name to instance variable name

      // validate that the balance is greater than 0.0; if it's not,
      // instance variable balance keeps its default initial value of 0.0
      if (balance > 0.0) // if the balance is valid
          this.balance = balance; // assign it to instance variable
  }
```

Case study: Account balances

```
16   // method that deposits (adds) only a valid amount to the balance
17   public void deposit(double depositAmount) {
18     if (depositAmount > 0.0) // if the depositAmount is valid
19       this.balance += depositAmount; // add it to the balance
20   }
21
22   // method returns the account balance
23   public double getBalance() {
24     return this.balance;
25   }
```

Case study: Account balances

```
26    // method that sets the name
27    public void setName(String name) {
28      this.name = name;
29    }
30
31    // method that returns the name
32    public String getName() {
33        return this.name; // give value of name back to caller
34    } // end method getName
35  } // end class Account
```

Case study: Account balances

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

```java
// Inputting and outputting floating-point numbers with Account objects.
import java.util.Scanner;
public class AccountTest {
  public static void main(String[] args) {
    Account account1 = new Account("Jane Green", 50.00);
    Account account2 = new Account("John Blue", -7.53);

    // display initial balance of each object
    System.out.printf("%s balance: $%.2f\n",
      account1.getName(), account1.getBalance());
    System.out.printf("%s balance: $%.2f\n\n",
      account2.getName(), account2.getBalance());

    // create a Scanner to obtain input from the command window
    Scanner input = new Scanner(System.in);
```

Case study: Account balances

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

```
16      System.out.print("Enter deposit amount for account1: "); // prompt
17      double depositAmount = input.nextDouble(); // obtain user input
18
19      System.out.printf("\nadding %.2f to account1 balance\n\n",
            depositAmount);
20      account1.deposit(depositAmount); // add to account1's balance
21
22      // display balances
23      System.out.printf("%s balance: $%.2f\n", account1.getName(), account1
            .getBalance());
24      System.out.printf("%s balance: $%.2f\n\n", account2.getName(),
            account2.getBalance());
25
26      System.out.print("Enter deposit amount for account2: "); // prompt
27      depositAmount = input.nextDouble(); // obtain user input
```

Case study: Account balances

```
30      System.out.printf("%nadding to account2 balance%n%n", depositAmount);
31      account2.deposit(depositAmount); // add to account2 balance
32
33      // display balances
34      System.out.printf("%s balance: $%.2f\n", account1.getName(), account1
            .getBalance());
35      System.out.printf("%s balance: $%.2f\n\n", account2.getName(),
            account2.getBalance());
36    } // end main
37  } // end class AccountTest
```

Case study: Account balances

```
account1 balance: $50.00
account2 balance: $0.00

Enter deposit amount for account1: 25.53

adding 25.53 to account1 balance

account1 balance: $75.53
account2 balance: $0.00

Enter deposit amount for account2: 123.45

adding 123.45 to account2 balance

account1 balance: $75.53
account2 balance: $123.45
```

Introduction to *collections* and class `ArrayList`

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

- *Collections* provide efficient methods that organize, store and retrieve your data without requiring knowledge of how the data is being stored.
- The collection class `ArrayList<T>` (from package `java.util`) can dynamically change its size to accommodate more elements.
- The `T` is a placeholder – when declaring a new `ArrayList`, replace it with the type of elements that you want the `ArrayList` to hold.
- This is similar to specifying the type when declaring an array, except that only non-primitive types can be used with these collection classes.
- Classes with this kind of placeholder that can be used with any type are called *generic class*es.

```
1 ArrayList<String> items = new ArrayList<String>();
```

Introduction to *collections* and class `ArrayList`

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Method	Description
`add`	Adds an element to the end of the `ArrayList`
`clear`	Removes all the elements from the `ArrayList`
`contains`	Returns `true` if the `ArrayList` contains the specified element; otherwise, returns `false`
`get`	Returns the element at the specified index
`indexOf`	Returns the index of the first occurrence of the specified element in the `ArrayList`
`remove`	Removes the first occurrence of the specified value
`remove`	Removes the element at the specified index
`size`	Returns he number of elements stored in the `ArrayList`
`trimToSize`	Trims the capacity of the `ArrayList` to current number of elements

Introduction to *collections* and class `ArrayList`

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

```java
// Generic ArrayList<T> collection demonstration.
import java.util.ArrayList;
public class ArrayListCollection {
  public static void main(String[] args) {
    // create a new ArrayList of Strings with an initial capacity of 10
    ArrayList<String> items = new ArrayList<String> ();

    items.add("red"); // append an item to the list
    items.add(0, "yellow"); // insert "yellow" at index 0

    // header
    System.out.print("Display list contents with counter-controlled loop:
        ");

    // display the colors in the list
    for (int i = 0; i < items.size(); i++)
      System.out.printf(" %s", items.get(i));
```

Introduction to *collections* and class `ArrayList`

```
18      // display colors using enhanced for in the display method
19      display(items, "%nDisplay list contents with enhanced for statement:"
            );
20
21      items.add("green"); // add "green" to the end of the list
22      items.add("yellow"); // add "yellow" to the end of the list
23      display(items, "List with two new elements:");
24
25      items.remove("yellow"); // remove the first "yellow"
26      display(items, "Remove first instance of yellow:");
27
28      items.remove(1); // remove item at index 1
29      display(items, "Remove second list element (green):");
30
31      // check if a value is in the List
32      System.out.printf("\"red\" is %sin the list%n",
33        items.contains("red") ? "" : "not ");
```

Introduction to *collections* and class `ArrayList`


```
34      // display number of elements in the List
35      System.out.printf("Size: %s%n", items.size());
36    }
37
38    // display the ArrayList's elements on the console
39    public static void display(ArrayList<String> items, String header) {
40      System.out.printf(header); // display header
41
42      // display each element in items
43      for (String item: items)
44        System.out.printf(" %s", item);
45      System.out.println();
46    }
47  } // end class ArrayListCollection
```

Introduction to *collections* and class `ArrayList`

```
Display list contents with counter-controlled loop: yellow red
Display list contents with enhanced for statement: yellow red
List with two new elements: yellow red green yellow
Remove first instance of yellow: red green yellow
Remove second list element (green): red yellow
"red" is in the list
Size: 2
```