Chapter 6: Introduction to Classes and Objects

面向对象的

所有的东西都是对象

TAO Yida

taoyd@sustech.edu.cn

Rust → function

Java → Object

Object-Oriented Programming

- Object-oriented programming (OOP) involves programming using objects (对象).
- An object represents an entity (实体) in the real world that can be distinctly identified, e.g., a student, a desk, a cat, a button, a book, etc.

Object-Oriented Programming

- An object has a unique identity, states, and behaviors.
  - States (properties or attributes): e.g., a cat has its age, weight, color, breed, etc.
  - Behaviors (actions): e.g., a cat can eat, sleep, and jump.
- Objects can interact with each other for computing tasks.
  - E.g., a student feeds a cat, so that the cat's weight increases.

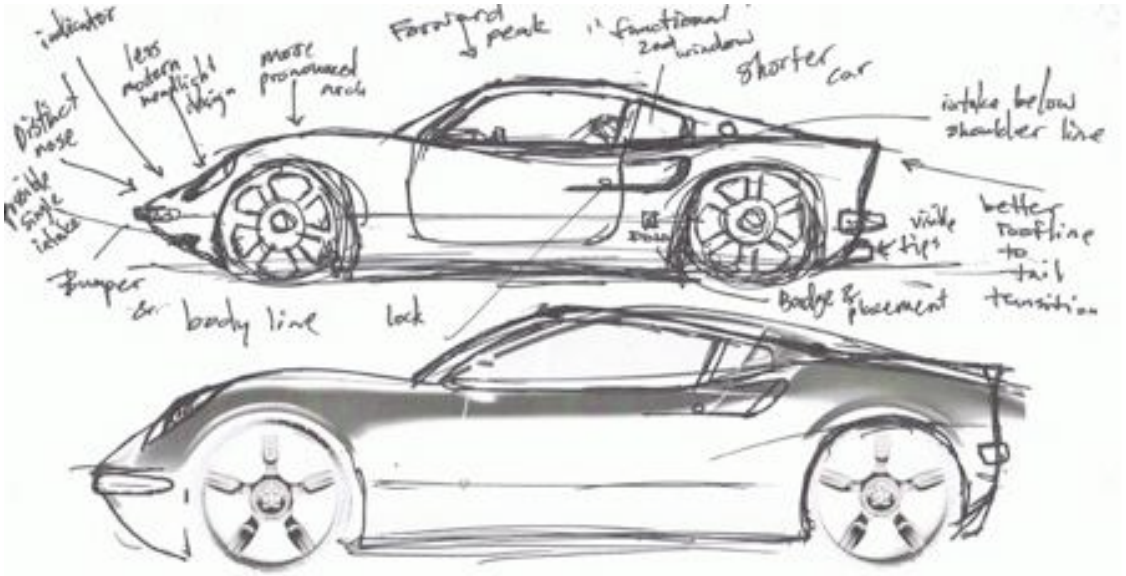
The Car Driving Analogy

- Suppose that our computational task is to drive a car and accelerate it by pressing down on its accelerator pedal (油门)

How to make it happen?

The Car Driving Analogy

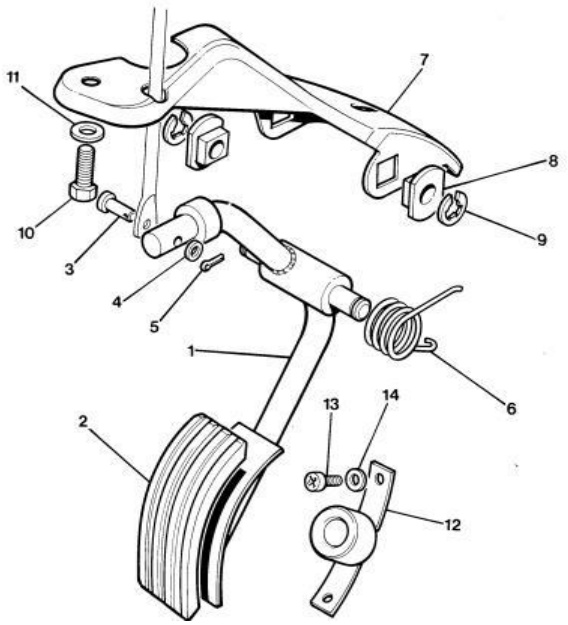
- **The very first step:** Before you can drive a car, someone has to design it (engineering drawings / blueprints).



The Car Driving Analogy

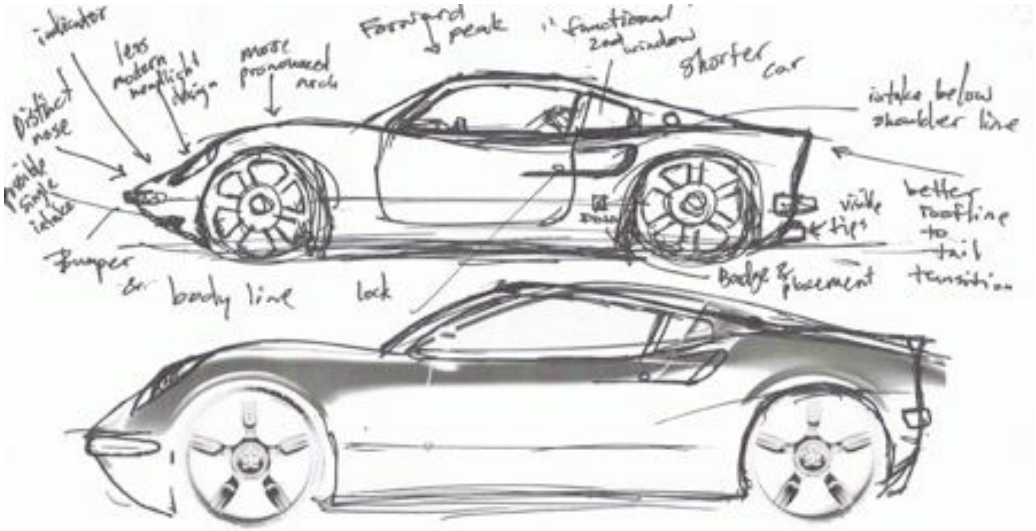
部件→属性

- The car's blueprints should also include the design for an accelerator pedal (油门) , the brake pedal（刹车）, the steering wheel（方向盘）
- The behaviors of these components should also be specified, e.g., the car's speed slows down once the brake is pressed.



The Car Driving Analogy

- We cannot drive a car's engineering drawings



The Car Driving Analogy

- We cannot drive a car's engineering drawings
- Before we drive a car, 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 driving the car

The Car Driving Analogy

- We cannot drive a car's engineering drawings **Designing classes**
- Before we drive a car, it must be built from the engineering drawings **Creating concrete instances of a class** 类 → 创建实例
- Even building a car is not enough, the driver must press the accelerator pedal to perform the task of driving the car

Invoking methods for computation 调用方式 usage → 使用实例

Programming the Car Driving Scenario --Design--

- **Class**: when programming in Java, we begin by creating a program unit (template) called **the `Car` class**, just like we begin with engineering draws in the driving example.
- **Method**: In the Car class, we provide one or more *methods* to define a Car's behaviors/actions.
  - `speedup()`
  - `slowdown()`
  - ……

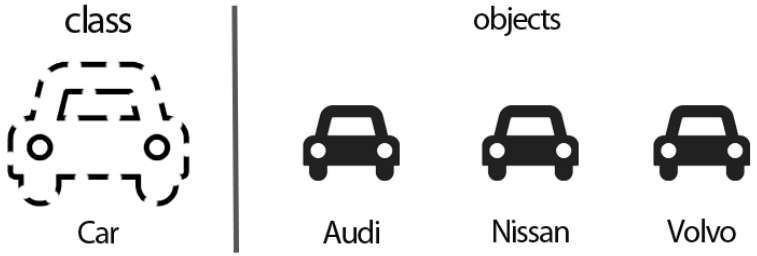
Programming the Car Driving Scenario --Design--

- Variables: Java uses *variables* to define the attributes of a class
  - A car can have many **attributes**: its color, 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



Programming the Car Driving Scenario --Instantiation--

- 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 **instance (object)** of a class before getting the program to perform tasks.



Programming the Car Driving Scenario --Usage--

- 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 by a method call to tell the method of the object to perform its task.
  - A driver instance invokes the speedup() method of a car instance
  - After method invocation, the states (e.g., speed) of the car are updated

The Whole Picture

- Class – a car's engineering drawings (a blueprint/template)
  - Variable – to specify the attributes (e.g., color, speed)
  - Method – designed to perform tasks (e.g., making a car move)
- Instance / Object – the real car that we drive
- Method call – perform the task (pressing the accelerator pedal)
  方法调用

Objectives

- Understand basic concepts of OOP, classes and objects
- Learn to declare a class and use it to create an object
- Learn to declare instance methods to implement class behavior
- Learn to declare instance variables to implement class attributes
- Learn to use a constructor to initialize an object when it is created

Declaring a Class

Every class declaration contains the keyword `class` + the class' name

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

在其他文件夹中也可以使用

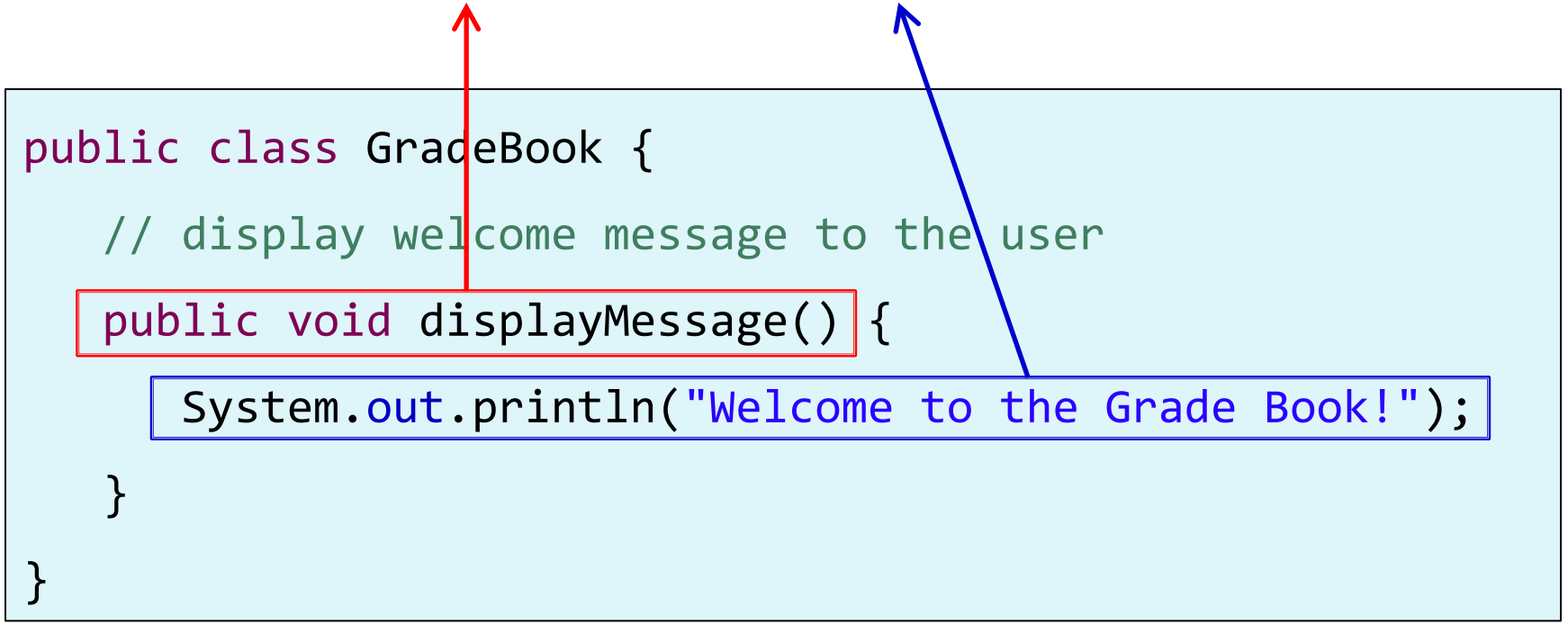
The **access modifier** `public` indicates that the declared class is visible to all classes everywhere.

公开的

Declaring a Method

A class usually consists of one or more methods.

Method = Method header + Method body (enclosed by {})



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

Declaring a Method

The **return type** specifies the type of data the method returns after performing its task, `void` means returning nothing to its calling method.

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

The **access modifier** `public` indicates that the method is "available to public", that is, can be directly called from the methods of other classes.

Declaring a Method

By convention, **method names** are in **Lower Camel Case** : the initial letter is in lower case, subsequent words begin with a capital letter.

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

一个Java之体大能

有一个public days

Tips: try to use meaningful names when declaring a method to make your programs understandable.

Declaring a Method

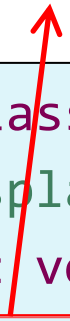
By convention, **method names** are in **Lower Camel Case** : the initial letter is in lower case, subsequent words begin with a capital letter.

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

The parentheses enclose the information that the method requires to perform its task. Empty parentheses indicate no information needs.

Declaring a Method

Like class, the method body is also enclosed in {}. The method body contains **statements** that perform the method's task.



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

Can We Run this Program?

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

Instantiation and Usage of a Class

```
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();
    }
}
```

Define a variable of the type GradeBook. Note that each new class you create becomes a new data type. Java is an **extensible language**.

Instantiation and Usage of a Class

```
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();
    }
}
```

含main方法的类为主类/client(使用GradeBook类)

实例

调用方法

Class instantiation expression. The keyword **new** is used to create a new instance / object of the specified class. Class name + () represent a call to a **constructor** (构造方法, a special method used to initialize the object's data).

Instantiation and Usage of a Class

```
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();
    }
}
```

Variable `myGradeBook` is a reference type that refers to the created object

We can use the reference variable `myGradeBook` to call the method `displayMessage()` using the **member operator** "`.`".

Instantiation and Usage of a Class

- Sometimes a method needs additional information (messages) to perform its task. **Parameters** are for this purpose.

```
public class GradeBook {
   // display welcome message to the user
   public void displayMessage( String courseName ) {
      System.out.printf("Welcome to the Grade Book for
      the course%s!\n", courseName);
   }
}
```

Instantiation and Usage of a Class

何时写static

```
public class GradeBookTest {
    public static void main(String[] args) {
        GradeBook myGradeBook = new GradeBook();
        myGradeBook.displayMessage("Java Programming");
    }
}
```

- Here when calling the method `displayMessage`, we supply a value for the parameter `courseName`. We call such values **arguments**.
- **(Parameter vs. Argument. 形式参数与实际参数)** A parameter is the variable that is part of the method's declaration. An argument is the actual values passed to the method.

Objectives

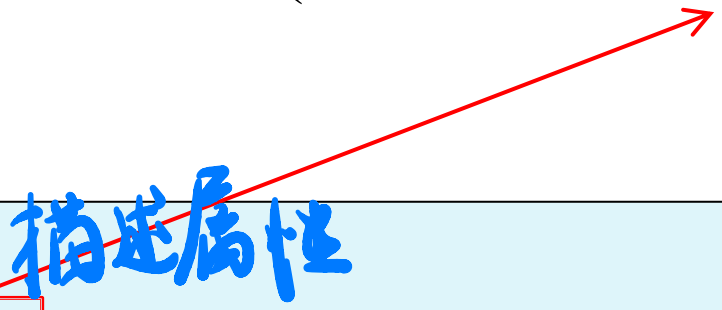
- Understand basic concepts of OOP, classes and objects
- Learn to declare a class and use it to create an object
- Learn to declare instance methods to implement class behavior
- Learn to declare instance variables to implement class attributes
- Learn to use a constructor to initialize an object when it is created

Class Attributes

- An object has attributes (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 methods that manipulate (read or write) the attributes of a particular object of the class.

Class Attributes

Attributes are represented as variables (also called **fields, 字段**) in a class declaration.




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

Instance Variables

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**.

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

作用域不同

Instance Variables

```
public class GradeBook {
   private String courseName;

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

Variables declared in the body of a particular method are known as **local variables** and can be only used in that method.

Instance Variables

```
public class GradeBook {
   private String courseName;

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

因为变量作用域重合
→ 用最内层的

Instance variables are declared inside a class declaration, but outside the bodies of the class' method declarations.

Instance Variables

```
public class GradeBook {
   private String courseName;

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

只能在本类使用

私有属性 方法

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

Access Control

- Suppose that there is a `score` attribute in a `GradeBook` class
- If `score` can be accessed by anyone, a student can change his score from 60 to 100, an attacker may change everyone's scores to 0
- To avoid this, Java provides the **access control** mechanism to control who can access which attributes or methods, using keywords such as `private` and `public`
- Most instance variables are declared to be `private`, which can only be accessed by `public` methods

Public Getter and Setter Methods

```
public class GradeBook {
   private String courseName;

   public void setCourseName(String name) {
      courseName = name;
   }

}
```

Update (write) the data field

Public Getter and Setter Methods

```
public class GradeBook {
   private String courseName;

   public void setCourseName(String name) {
      courseName = name;
   }

   public String getCourseName() {
      return courseName;
   }
}
```

检索

Retrieve (read) the data field

可操作去除敏感信息

get/set + 变量名

Public Getter and Setter Methods

```
public class GradeBook {
   private String courseName;

   public void setCourseName(String name) {
      courseName = name;
   }

   public String getCourseName() {
      return courseName;
   }
   public void displayMessage() {
      System.out.printf("Welcome to the grade book
      for\n%s!\n", courseName);
   }
}
```

同一类可访问该字段

private courseName can be accessed by methods within its own class

Public Getter and Setter Methods

```
public class GradeBook {
   private String courseName;

   public void setCourseName(String name) {
      courseName = name;
   }

   public String getCourseName() {
      return courseName;
   }
   public void displayMessage() {
      System.out.printf("Welcome to the grade book
      for\n%s!\n", getCourseName());
   }
}
```

Calling public getter method is also okay (recommended)

Public Getter and Setter Methods

```
import java.util.Scanner;
public class GradeBookTest {
    public static void main(String[] args) {
        GradeBook myGradeBook = new GradeBook();

        Scanner input = new Scanner(System.in);
        myGradeBook.setCourseName(input.nextLine());
        System.out.println(myGradeBook.getCourseName());

        myGradeBook.setCourseName("Java A");
        myGradeBook.displayMessage();
    }
}
```

Objectives

- Understand basic concepts of OOP, classes and objects
- Learn to declare a class and use it to create an object
- Learn to declare instance methods to implement class behavior
- Learn to declare instance variables to implement class attributes
- Learn to use a constructor to initialize an object when it is created

Initializing Objects with Constructors

- Each class can provide a special method called a **constructor** to 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` instantiates a class (creates an object) by allocating memory for a new object and returning a reference to that memory, then calls the corresponding class's constructor to initialize the object.
  - `GradeBook myGradeBook = new GradeBook();`

Declaration **Instantiation & Initialization**

Initializing Objects with Constructors

- The empty parentheses after "`new GradeBook`" indicate a call to the class's constructor without arguments

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

- Like methods, constructors can be overloaded to provide custom initialization for objects of your class

初始化

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

GradeBook myGradeBook = new GradeBook("Alice", "Java");

GradeBook myGradeBook = new GradeBook(2022, "Spring", "Java");
```

Example

```
public class GradeBook {
    public String courseName; // course name for this grade book
    // constructor initializes courseName with String argument
    public GradeBook(String name) {
        courseName = name; // initializes courseName
    }
    // method to set the course name
    public void setCourseName(String name) {
        courseName = name;
    }
    // ...
}
```

Initializing Objects with Constructors

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

- A constructor must have the same name with its class
- Normally, constructors are declared `public`.
- An important difference between constructors and methods is that constructors cannot return values, so they cannot specify a return type (not even `void`).

Initializing Objects with Constructors

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

- Like a method, a constructor's parameter list specifies the data it requires to perform its task.
  - When creating a new object, the data is placed in the parentheses after the class name: `GradeBook book = new GradeBook("CS102A");`
- A class instance creation expression returns a **reference** to the new object (the address to its variables and methods in memory).

Initializing Objects with Constructors

```
public class GradeBookTest {
   public static void main(String[] args) {
      // create GradeBook objects
      GradeBook gradeBook1 = new GradeBook(
         "CS101 Introduction to Java Programming");
      GradeBook gradeBook2 = new GradeBook(
         "CS102 Data Structures in Java");
      // display initial value of CourseName for each GradeBook
      System.out.printf("gradeBook1 course name is: %s\n",
         gradeBook1.getCourseName());
      System.out.printf("gradeBook2 course name is: %s\n",
         gradeBook2.getCourseName());
   }
}
```

```
gradeBook1 course name is: CS101 Introduction to Java Programming
gradeBook2 course name is: CS102 Data Structures in Java
```

Default Constructors

- A class may not have any constructor.
- In this case, the compiler provides a default constructor with 0 parameters
  - 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)
- If you explicitly declare any constructors for a class, the Java compiler will not create a default constructor for the class.

Default Constructors

```
public class GradeBook { // no constructor provided by the programmer
   private String courseName;
   public void setCourseName(String name) {
      courseName = name;
   }
   public String getCourseName() {
      return courseName;
   }
   public void displayMessage() {
      System.out.printf("Welcome to the grade book for\n%s!\n", getCourseName());
   }
}
```

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

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

Yes. Compiler will provide a default constructor with no parameters.

`courseName` is initialized to `null` (default for reference types)

Default Constructors

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

构造方法

再修改

设置了构造～则无默认～

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

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

No. Compiler will not provide a default constructor this time. The statement will cause a **compilation error**.

Case Study Time!

Case Study Ⅰ: Pet Show

- A happy family has two pets: a poodle (贵宾犬) named “Fluffy”, a hound (猎犬) named “Alfred”.
- Suppose we want to write a Java program for a pet show: each dog makes a self introduction.

“Hello, my name is Fluffy. I am a poodle.”

“Hello, my name is Alfred. I am a hound.”

Program Design

- **Observation 1:** The two pets are both dogs. So we can design a Dog class to represent them.

```
public class Dog {

}
```


Program Design

- **Observation 2:** The two pets have their own names and belong to different breeds (品种). We can define two instance variables to represent such information.

```
public class Dog {
    private String name;
    private String breed;
}
```

Program Design

- In order to create instances of the Dog class, we need to define a constructor. Since each dog has two attributes, we can define a constructor that takes two arguments.

```
public class Dog {
    private String name;
    private String breed;
    public Dog(String name, String breed) {
        this.name = name;
        this.breed = breed;
    }
}
```


Program Design

- In order to create instances of the `Dog` class, we need to define a constructor. Since each dog has two attributes, we can define a constructor that takes two arguments.

```
public class Dog {
    private String name;
    private String breed;
    public Dog(String name, String breed) {
        this.name = name;
        this.breed = breed;
    }
}
```

The passed arguments will be used to initialize the attributes.

Program Design

- In order to create instances of the `Dog` class, we need to define a constructor. Since each dog has two attributes, we can define a constructor that takes two arguments.

```
public class Dog {
    private String name;
    private String breed;
    public Dog(String name, String breed) {
        this.name = name;
        this.breed = breed;
    }
}
```

The keyword `this` points to the current object. Helps differentiate the method parameters (local variables) and the instance variables with the same name.

Program Design

- In order to create instances of the `Dog` class, we need to define a constructor. Since each dog has two attributes, we can define a constructor that takes two arguments.

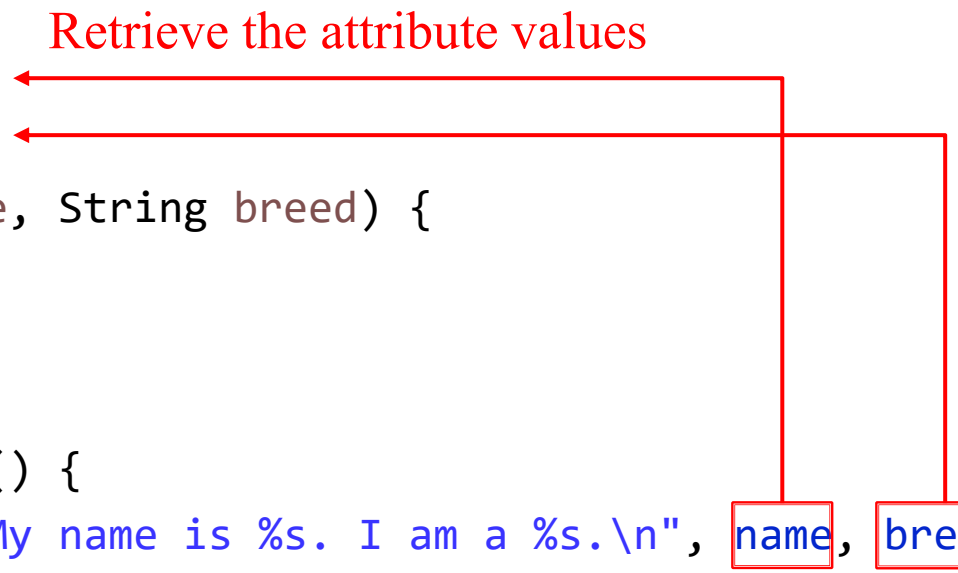
```
public class Dog {
    private String name;
    private String breed;
    public Dog(String dogName, String dogBreed) {
        name = dogName;
        breed = dogBreed;
    }
}
```

"this" is not compulsory if the parameters and instance variables have different names (no ambiguity).

Program Design

- The dogs have the ability of making self introductions.

```
public class Dog {
  private String name;
  private String breed;
  public Dog(String name, String breed) {
    this.name = name;
    this.breed = breed;
  }
  public void selfIntro() {
    System.out.printf("My name is %s. I am a %s.\n", name, breed);
  }
}
```



Program Design

- Finally, we implement the `PetShow` program with a `main` method.

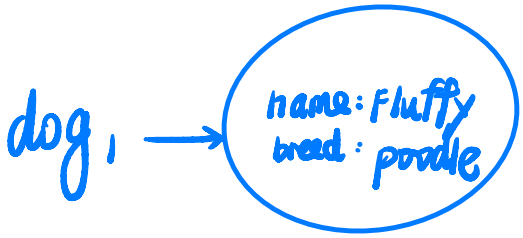
```
public class PetShow {
  public static void main(String[] args) {
    Dog dog1 = new Dog("Fluffy", "poodle");
    Dog dog2 = new Dog("Alfred", "hound");
    dog1.selfIntro();
    dog2.selfIntro();
  }
}
```

Instantiation:
Invoke the constructor to create two dog objects

Program Design

- Finally, we implement the `PetShow` program with a `main` method.

```
public class PetShow {
  public static void main(String[] args) {
    Dog dog1 = new Dog("Fluffy", "poodle");
    Dog dog2 = new Dog("Alfred", "hound");
    dog1.selfIntro();
    dog2.selfIntro();
  }
}
```



Invoke methods on the two objects.

Object references (or names) are needed to invoke instance methods.

Case Study II: Account Balances

- Suppose we are asked to design a Java program for managing bank accounts.
- For simplicity, we assume that the bank only provides two types of services:
  - Adding money to an account (存款)
  - Checking the balance of an account (查询余额)

The key task is to define an Account class

```
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
```

```java
// Account class with a constructor to validate and
// initialize instance variable balance of type double

public class Account {

   // instance variable that stores the balance
   private double balance;

   // constructor
   public Account(double initialBalance) {
      // if initialBalance is not greater than 0.0
      // balance is initialized to the default value 0.0
      if(initialBalance > 0.0) balance = initialBalance;
   }

   // add an amount to the account
   public void deposit(double amount) {
      balance += amount;
   }

   // return the account balance
   public double getBalance() {
      return balance;
   }
}
```

属性

Validating Constructor Arguments

- It's common for users to open an account to deposit money immediately, so the constructor receives a parameter `initialBalance` of type `double` that represents the initial balance.
  - The constructor ensures that `initialBalance` is greater than `0.0`
  - If so, `initialBalance`'s value is assigned to instance variable `balance`.
  - Otherwise, `balance` remains to be `0.0` (its default initial value).

```
// constructor
public Account(double initialBalance) {
   // if initialBalance is not greater than 0.0
   // balance is initialized to the default value 0.0
   if(initialBalance > 0.0) balance = initialBalance;
}
```

Case Study II: Account Balances

- We further define a class `AccountTest` that creates and manipulates two `Account` objects.
- The two classes `Account` and `AccountTest` can be placed in the same or different directories (packages) from the same project. Method invocation is slightly different for these two cases.
- For now, let's assume that they are in the same directory.

```
import java.util.Scanner;
public class AccountTest {
  public static void main(String[] args) {
    Account account1 = new Account(50.00);
    Account account2 = new Account(-7.53);
    // display initial balance of each object
    System.out.printf("account1 balance: $%.2f\n",
            account1.getBalance());
    System.out.printf("account2 balance: $%.2f\n\n",
            account2.getBalance());
    Scanner input = new Scanner(System.in);
    double depositAmount; // deposit amount read from user
```



```
System.out.print("Enter deposit amount for account1: ");
depositAmount = input.nextDouble();
System.out.printf("\nadding %.2f to account1 balance\n\n",
        depositAmount);
account1.deposit(depositAmount); // add to account1 balance

// display balances
System.out.printf("account1 balance: $%.2f\n",
        account1.getBalance());
System.out.printf("account2 balance: $%.2f\n\n",
        account2.getBalance());
```



```
      System.out.print("Enter deposit amount for account2: ");
      depositAmount = input.nextDouble();
      System.out.printf("\nadding %.2f to account2 balance\n\n",
         depositAmount);
      account2.deposit(depositAmount); // add to account2 balance

      //display balances
      System.out.printf("account1 balance: $%.2f\n",
         account1.getBalance());
      System.out.printf("account2 balance: $%.2f\n\n",
         account2.getBalance());
      input.close();
   }
}
```



```
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
```

Primitive Types vs. Reference Types

- Java types are divided into two categories: primitive types and reference types.
- Primitive types are the basic types of data
  - byte, short, int, long, float, double, boolean, char
  - A primitive-type variable can store one value of its declared type

Type	Description	Default value	Size	Example code
boolean	Truth value	false	1 bit	`boolean b = false;`
char	Unicode character	\u0000	16 bits	`char c = 'z';`

Primitive Types vs. Reference Types

- All non-primitive types are reference types, including instantiable classes and arrays (an array is a container object that holds a fixed number of values of a single type)
  - Java Built-in: `Scanner, Random, String, String[], int[]`
  - User defined: `Dog, Account, etc.`
- Reference-type variables store the memory locations of objects
  - `Dog dog1 = new Dog("Fluffy", "Poodle");`
  - Such a variable is said to refer to an object in the program. Objects that are referenced may each contain instance variables of primitive or reference types.

引用数据类型可调自己的方法

array.length → 是属性不是方法

Primitive Types vs. Reference Types

- Reference-type variables, if not explicitly initialized, are initialized by default to the value `null` (reference to nothing)
  变量不指向任何东西
  NullPointerException
- To call methods of an object, you need to use the reference (must be non-null) to the object: `dog1.selfIntro();`
- Primitive-type variables (e.g., `int` variables) do not refer to objects, so such variables cannot be used to call methods