1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Lecture 7: Objects and classes

Object oriented programming (OOP)

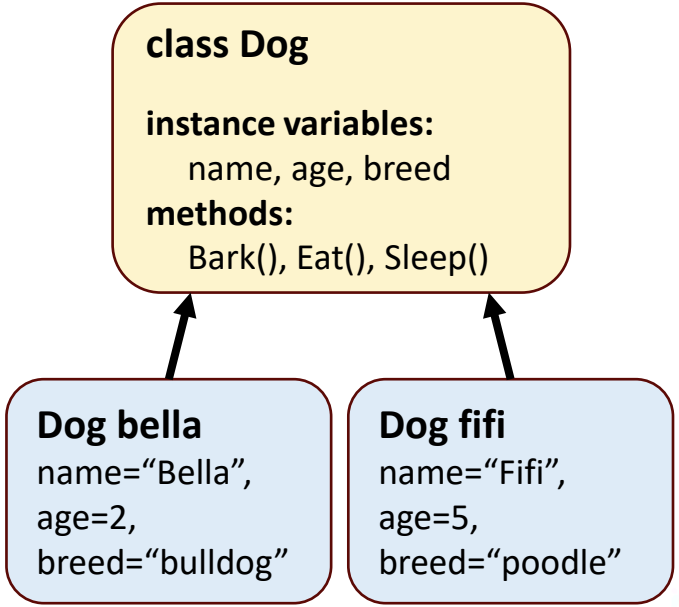
- Today, we will cover the fundamentals of object oriented programming (OOP) in Java
  - Basic concepts of classes and objects
  - Instance variables, set and get methods
  - Scope and access modifiers
  - Enum types
  - Inheritance, composition, and polymorphism
- Much of the material is based on slides from ***Java: How to Program***, chapter 7, available via MyAberdeen

Learning objectives

- After the theory sessions today, you should be able to:
  - Explain the basic concepts of classes and objects
  - Declare classes and instantiate objects in your Java programs
  - Select appropriate access modifiers for your classes
  - Use inheritance and composition in your Java programs

Concepts of classes and objects

- ***Class*** is a data structure that represents a category of objects with some shared characteristics
  - Class can include *instance variables* defining its state, as well as *methods* implementing its behavior
- ***Object*** is an instance of a class
  - For example, class “Person” represents human beings, and object “John” is an instance of class “Person”, representing a specific person



Classes and objects in Java

- In Java, you can declare new classes as needed; this is one reason Java is known as an *extensible* language
- Each class you create becomes a new type that can be used to declare variables and create objects
  - By convention, class names, method names and variable names are all identifiers and all use the camel-case naming scheme
  - Also, by convention, class names begin with an initial uppercase letter, and method names and variable names begin with an initial lowercase letter
  - Note that these conventions are *not* forced by Java syntax; however, it is highly recommended to follow them

Instance variables

- An object has attributes that are implemented as instance variables and carried with it throughout its lifetime
- Each object (instance) of the class has its own copy of each of the class's instance variables
- Instance variables are declared inside a class declaration but outside the bodies of the class's method declarations
- A class normally contains one or more methods that manipulate the instance variables that belong to particular objects of the class

Getter and setter methods

- By convention, we use *set* and *get* methods to store / obtain instance variable values (i.e., attributes) in an object
    - If variable is defined as private, it is not possible to access directly
    - If variable is defined as public, it can be accessed directly, but even then, it is best to use set and get methods to modify the variable
- *Set* methods are commonly called *mutator methods*
- *Get* methods are commonly called *accessor methods* or *query methods*

Get and Set example

Account.java

```
1   public class Account {
2     private String name; // instance variable
3     // method to set the name
4     public void setName(String name) {
5       this.name = name;
6     }
7     // method to retrieve the name
8     public String getName() {
9        return name; // return name value
10    }
11  }
```

AccountTest.java

```
1   import java.util.Scanner;
2   public class AccountTest {
3     public static void main(String[] args) {
4       // create a Scanner object for input
5       Scanner input = new Scanner(System.in);
6       // create an Account object myAccount
7       Account myAccount = new Account();
8       // display initial value of name (null)
9       System.out.printf("Initial name is:
10        %s%n%n", myAccount.getName());
11      // prompt for and read name
12      System.out.println("Please enter the name:");
13      String theName = input.nextLine();
15      myAccount.setName(theName);
16      System.out.println(); // outputs a blank line
17      // display the name stored in object myAccount
18      System.out.printf("Name in object myAccount
19        is:%n%s%n", myAccount.getName());
20    }
21  }
```

```
Initial name is: null

Please enter the name:
Jane Green

Name in object myAccount is:
Jane Green
```

Get and Set example

Account.java

```
1   public class Account {
2     private String name; // instance varia
3     // method to set the name
4     public void setName(String name) {
5       this.name = name;
6     }
7     // method to retrieve the name
8     public String getName() {
9        return name; // return name value
10    }
11  }
```

Setter takes one parameter, return is void

Getter takes no parameter, return is String

AccountTest.java

```
1   import java.util.Scanner;
2   public class AccountTest {
3     public static void main(String[] args) {
4       // create a Scanner object for input
5       Scanner input = new Scanner(System.in);
6       // create an Account object myAccount
7       Account myAccount = new Account();
```

```
ue of name (null)
tial name is:
etName());
l name
ease enter the name:");
13      String theName = input.nextLine();
ame);
// outputs a blank line
tored in object myAccount
e in object myAccount
t.getName());
20    }
21  }
```

```
Initial name is: null

Please enter the name:
Jane Green

Name in object myAccount is:
Jane Green
```

Access modifiers (public and private)

- Most instance-variable declarations are preceded with the keyword `private`, which is an access modifier
- Variables or methods declared with access modifier `private` are accessible only to methods of the class in which they're declared
- Declaring instance variables with access modifier private is known as *information hiding*
    - When a program creates (instantiates) an object of class *Account*, variable *name* is encapsulated (hidden) in the object and can be accessed only by methods of the object's class

Method's local variables

- Parameters of a method are *local variables* of the method
  - Local variables declared in the body of a particular method can be used *only* in that method
  - When a method terminates, the values of its local variables are lost
  - Local variables are not automatically initialized
- If a method contains a local variable and instance variable with the same name, the method's body will refer to the local variable rather than the instance variable
  - Local variable *shadows* the instance variable in the method's body.
  - Keyword **`this`** can be used to refer to the shadowed instance variable explicitly

Using keyword *this*

```
1   public class Account {
2       private String name; // instance variable
3
4       // method to set the name in the obj
5       public void setName(String name) {
6           this.name = name; // store the
7       }
8
9       // method to retrieve the name from
10      public String getName() {
11          return name; // return value of name to caller
12      }
13  }
```

We could have avoided the need for keyword **`this`** by choosing different parameter name on line 5, but using **`this`** keyword is a widely accepted practice.

More about keyword *this*

- Every object can access a reference to itself with keyword **`this`** (sometimes called the **`this`** reference)
- When an instance method is called for a particular object, the method's body implicitly uses keyword **`this`** to refer to the object's instance variables and other methods
  - Therefore, the class's code knows which object should be manipulated
- There is only one copy of each method per class; every object of the same class shares the method's code
- On the other hand, each object has its own copy of the class's instance variables, and the non-static methods implicitly use **`this`** to determine the specific object to manipulate

Instantiating an object

- A class instance (object) is created using keyword **`new`**
- A *constructor* is similar to a method, but it is called implicitly by the **`new`** operator to initialize an object's instance variables when the object is created
  - If a class does not define a constructor, the compiler provides a default constructor with no parameters, and the class's instance variables are initialized to their default values
  - Every instance variable has a default initial value (a value provided by Java) if you do not specify the initial value
  - The default value for an instance variable of type `String` is `null`

Constructor example

- In this example, instance variable `name` is set using the constructor, so we do not need to call `setName` after creating the object

Account.java

```
1   public class Account {
2     private String name; // instance variable
3     // constructor initializes name
4     public Account(String name) {
5       this.name = name;
6     }
7     // method to set the name
8     public void setName(String name) {
9       this.name = name;
10    }
11    // method to retrieve the name
12    public String getName() {
13       return name; // return name value
14    }
15  }
```

AccountTest.java

```
...
9     System.out.println("Please enter the name:");
10    String theName = input.nextLine();
11    Account myAccount = new Account(theName);
...
```

The constructor is a method with the same name as the class. It is invoked when an object is instantiated using the keyword new.

Constructor overloading example

- *Overloaded constructors* allow different ways to initialise objects
  - Only the parameters for the constructors are different

Account.java

```
1   public class Account {
2     private String name; // instance variable
3     // constructor with full name as input
4     public Account(String name) {
5       this.name = name;
6     }
7     // constructor with first and last name
8     // as input
9     public Account(String first, String last) {
10      this.name = first + " " + name;
11    }
12  }
```

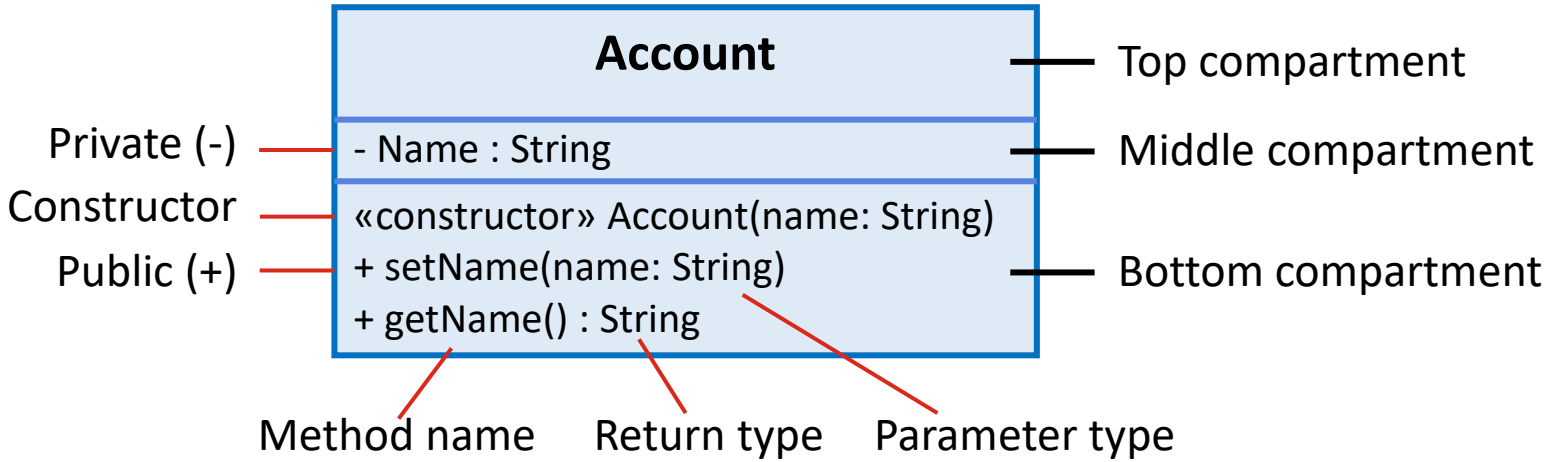
AccountTest.java

```
...
9     Account lisasAccount = new Account("Lisa Brown");
10    Account bobsAccount = new Account("Bob", "Blue");
...
```

The constructor with one parameter is invoked when `lisasAccount` is created, and the constructor with two parameters is invoked when `bobsAccount` is created.

UML class diagram

- UML class diagrams are often used to illustrate classes



Questions, comments?