Lecture 2: Java & Javadoc

CS 62
Fall 2018
Alexandra Papoutsaki & William Devanny
Instance Variables

• or member variables or fields
• Declared in a class, but outside of any method, constructor or block
• Each object has its own copy of the variable!
• Invoked as: `myObject.variableName`
Static Variables

- or class variables
- static means constant, i.e. it will be constant for all instances of the class
- cannot be defined in method body
- Invoked as: `myClass.variableName`
Local Variables

- Declared in method, constructor or block
- Destroyed after the execution of the method
- **No** access modifier
Methods

• A collection of grouped statements that perform a logical operation and control the behavior of objects

• Syntax:
  • modifier return-type method-name(type parameter-name,...){...}
  • e.g., `public int enrollInClass(int classID){...}

  • Signature: method name and the number, type, and order of its parameters. Not return type

• Can also be `static`, therefore shared by all instances of a class

• Can be `overloaded` (same name, different parameters)
**this**

- Within an instance method or a constructor used to refer to current object
  - can be used to call instance variables, methods, and constructors

```java
public class Car{
    private String color;

    public Car(){
        this("undefined");
    }

    public Car(String color){
        this.color = color;
    }
}
```
Combination of variables and methods

• Instance methods can access instance variables and instance methods directly.
• Instance methods can access static variables and static methods directly.
• Static methods can access static variables and static methods directly.
• Static methods cannot access instance variables or instance methods directly—they must use an object reference.
  • “Cannot make a static reference to the non-static field” in main method
• Static methods cannot use the this keyword as there is no instance for this to refer to.
Exercise: Bicycle.java

• Write the class Bicycle that contains the following fields:
  • cadence
  • gear
  • speed
  • id
  • numberOfBicycles

• Primitive types or objects?
• Instance variables or static? Instantiate?
Exercise: **Bicycle.java**

```java
public class Bicycle {

    private int cadence;
    private int gear;
    private int speed;
    private int id;

    private static int numberOfBicycles = 0;
}
```
Exercise: **Bicycle.java**

- Write the appropriate getters and setters for these variables
public int getID() {
    return id;
}

public static int getNumberOfBicycles() {
    return numberOfBicycles;
}

public int getCadence() {
    return cadence;
}

public void setCadence(int cadence) {
    this.cadence = cadence;
}

public int getGear() {
    return gear;
}

public void setGear(int gear) {
    this.gear = gear;
}

public int getSpeed() {
    return speed;
}
Exercise: Bicycle.java

• Create a non-parameterized constructor that sets the id to the number of bicycles and increases the counter

```java
public Bicycle() {
    id = ++numberOfBicycles;
}
```

• Create a constructor that takes 3 parameters: cadence, gear, speed. How can you use the previous constructor?

```java
public Bicycle(int cadence, int speed, int gear) {
    this();
    this.cadence = cadence;
    this.gear = gear;
    this.speed = speed;
}
```
Exercise: `Bicycle.java`

- Write a `main` method within your class
- Print the total number of bicycles
- Create an object (`unknown`) using the non-parameterized constructor
- Print its gear field
- Create an object (`myBike`) passing the following 3 arguments (2, 3, 5).
- Print its speed
- Print the total number of bicycles
public static void main (String args[]) {
    System.out.println(Bicycle.numberOfBicycles);
    Bicycle unknown = new Bicycle();
    System.out.println(unknown.getGear());
    Bicycle myBike = new Bicycle(2,3,5);
    System.out.println(myBike.getSpeed());
    System.out.println(Bicycle.getNumberOfBicycles());
}
A vocabulary refresher for variables

- **Declaration:** state the type of variable and its identifier. A variable can only be declared once. E.g. `int x;`
- **Initialization:** the first time a variable takes a value. E.g., `x = 3;`
  - Can be combined with declaration, e.g., `int y = 3;`
- **Assignment:** discarding the old value and replacing it with a new. E.g., `x = 2;`
- Static or instance variables are automatically initialized with default values, i.e. `null` for objects, 0 for `int`, `false` for `boolean`, etc.
- Local variables are not automatically initialized and your code won’t compile if you have not initialized them and you are trying to use them. E.g.,
  ```java
  public void foo() {
      int x;
      System.out.println(x);
      //The local variable x might not have been initialized
  }
  ```
Inheritance

• When you want to create a new class and there is already a class that includes some of the code you want your new class to have, you can derive the new class from the existing class → reuse code!

• We say that a class extends or inherits another class

• E.g., public class Car extends Vehicle

• Car is a subclass of Vehicle

• Vehicle is a superclass of Car

• Car IS-A Vehicle
Inheritance in Java

- A subclass inherits all of the **public** and **protected** members of parent
- **Hiding**: same name of variables or of static method between super and subclass
- **Overriding**: same signature of instance methods between super and subclass
- Single inheritance!
  - A class can only extend ONE AND ONLY ONE class
- Multilevel inheritance
  - Class **SUV** extends class **Car** which extends class **Vehicle**
Example: Animal.java

```java
public class Animal {
    public int legs = 2;
    public static String species = "Animal";
    public static void testClassMethod() {
        System.out.println("The static method in Animal");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Animal");
    }
}
```
Example: Cat.java

```java
public class Cat extends Animal {
    public int legs = 4;
    public static String species = "Cat";
    public static void testClassMethod() {
        System.out.println("The static method in Cat");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat");
    }
}
```
Hiding vs Overriding

```java
public static void main(String[] args) {
    Cat myCat = new Cat();
    myCat.testClassMethod(); //invoking a hidden method
    myCat.testInstanceMethod(); //invoking an overridden method
    System.out.println(myCat.legs); //accessing a hidden field
    System.out.println(myCat.species); //accessing a hidden field
}
```

• Output: “The static method in Cat
• The instance method in Cat
• What you were expecting, right?
public static void main(String[] args) {
    Animal yourCat = new Cat();
    yourCat.testClassMethod(); //invoking a hidden method
    yourCat.testInstanceMethod(); //invoking an overridden method
    System.out.println(yourCat.legs); //accessing a hidden field
    System.out.println(yourCat.species); //accessing a hidden field
}

- Output: “The static method in Animal
  The instance method in Cat
  Animal”

- **Hiding**: For fields (instance+static) and methods (static) the class is determined at compile-time. Here, the compiler sees that `yourCat` is declared as `Animal`.

- **Overriding**: For instance methods this is determined at run-time. At this point, we know that `yourCat` is of type `Cat`.

- One form of polymorphism (dynamic)
super keyword

- refers to the direct parent class of the current class
- `super.variable` (for hidden fields ➔ avoid altogether)
- `super.instanceMethod()` (for overridden methods)
- `super(args)` ➔ to call the constructor of the superclass
  - First line in subclass constructor
All classes inherit `Object`

- Directly (if they do not extend any other class) or indirectly
- `Object` class has methods (and more):
  - `public boolean equals (Object other)`
    - Default behavior returns true only if same `object`
  - `public String toString()`
    - Returns string representation of object - default is hexadecimal
    - Does not print the string
    - Typically needs to be overridden to be useful
  - `public int hashCode()`
    - Unique identifier defined so that if `a.equals(b)` then `a`, `b` have same `hashCode`
final

• variable - only assigned once in its declaration or in constructor - its value cannot change initialization
  • Often paired with static, e.g., static final PI = 3.14;

• method - cannot be overridden by subclass

• class - cannot be extended
abstract

• Class – cannot be instantiated but can be extended

• Method – declared without an implementation
  • no braces and body, just semicolon
  • `public abstract int enrollInClass(int classID);`

• If a class has at least one abstract method then it should be declared abstract itself

• If you extend an abstract class either declare subclass as abstract too or implement all abstract methods
Interfaces

• Contracts on how the program should work, abstracting from implementation
  • `public interface Moveable {...}`

• A class can `implement` many interfaces
  • `public class Car extends Vehicle implements Moveable`

• Variables - implicitly `public, static, and final`
• Methods - implicitly `public` (abstract, default, or static)
• Cannot be instantiated
• Can extend any number of interfaces
  • `public interface GroupedInterface extends Interface1, Interface2`
public interface Moveable{
    int turn(Direction direction, double radius, double speed);

    default int stop(){
        speed=0;
    }
}

public class Car extends Vehicle implements Moveable{
    int turn(Direction direction, double radius, double speed){
        //code goes here
    }
}
Abstract Classes vs Interfaces

• Can declare fields that are not static and final

• Can define public, protected, private concrete methods

• Can extend only one class whether or not abstract

• All fields are public, static, final

• All methods are public

• Can implement any number of interfaces
Nested class

• A class defined within a class

```java
class Outer{
    ...
    static class Nested{...
    class Inner{...}
}
```

• Logically groups classes that are only used once in one place
• Increases encapsulation
• Better code
Enum Types

• Example
  • `enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES}`

• Operations:
  • `int compareTo(Suit other)`
  • `String toString()`
  • `int ordinal()` returns position in its enum declaration. *starts with 0*
  • `static Suit.valueOf(String name)`
  • `static Suit[] values()` *returns array of all values*
Documentation

• Important for code maintainability
  • This matters even for 1st week assignments

• Critical when working on a team

• Create documentation first—this is design work!
JavaDoc

• Document generation system
  • Reads JavaDoc comment ➔ HTML pages

• JavaDoc comment = description written in HTML + tags

• Enclosed in /**     */

• Must precede class, variable, constructor or method declaration

• Read the style guide
JavaDoc

• Common tags:
  • for class:
    • @author author name - classes and interfaces
    • @version date - classes and interfaces
  
  • for method:
    • @param param name and description - methods and constructors
    • @return value returned, if any - methods
    • @throws description of any exceptions thrown - methods
Packages

• Use them! E.g., `package assignment1;` ... before everything else

• Package name == folder name

• Helps organize large projects e.g, `java.lang→fundamental`

• Import a package member: `import package.member;`
• Import an entire package: `import package.*;`
public class IdentifyMyParts {
    public static int x = 7;
    public int y = 3;
}

• What is the output from the following code:

    IdentifyMyParts a = new IdentifyMyParts();
    IdentifyMyParts b = new IdentifyMyParts();
    a.y = 5;
    b.y = 6;
    a.x = 1;
    b.x = 2;
    System.out.println("a.y = " + a.y);
    System.out.println("b.y = " + b.y);
    System.out.println("a.x = " + a.x);
    System.out.println("b.x = " + b.x);
    System.out.println("IdentifyMyParts.x = " + IdentifyMyParts.x);