15-440 Distributed Systems Recitation 1

Laila Elbeheiry
Slides by: Tamim Jabban

Please open pollev.com/lailaelbeheiry982

Office Hours

Office 1004, Zoom

Sunday, Tuesday: 10:00 – 12:00 PM

Appointment: send an e-mail

Open door policy

Logistics

- PS1 is out on the course website (due on Jan 23) submit on Gradescope

Java Introduction

- A class-based, object-oriented programming language
- Platform-independent *write once run anywhere (*Compiler converts source code to bytecode and then the JVM executes the bytecode generated by the compiler)
- Java applications are compiled to byte code that can run on any Java Virtual Machine
- The syntax of Java is similar to C/C++
- Eliminates complex features like pointers and explicit memory allocation and deallocation (garbage collection)

Java Language Constructs

- Variables
- Datatypes
  - Primitive
    - boolean, char, byte, short, int, long, float, double
  - Non-primitive
    - String, Array, Classes
- Operators
- Flow Control
  - If, switch-case, break, continue
- Loops
  - For, while, for-each loop
- Arrays
  - Dynamically allocated
  - Immutable (cannot grow)
  - type var-name[]; OR type[] var-name;
  - var-name = new type [size];
  - All elements set to their default value (0 or null)
- Strings
- Other classes
- Naming conventions

Java OOPs: Class

- A user defined blueprint or prototype from which objects are created
- Represents the set of *properties* or *methods* that are common to all objects of one type

```
public class Dog
{
    // Instance Variables
    String name;
    String breed;
    int age;
    String color;

    // method 1
    public String getName()
    {
        return name;
    }
    // method 2
    public String getBreed()
    {
        return breed;
    }
    // method 3
    public int getAge()
    {
        return age;
    }
}
```

attributes

methods

Java OOPS: Object

- An **Object** consists of
  - State: represented by attributes of an object
  - Behavior: represented by methods of an object.
- When an object of a class is created, the class is said to be **instantiated**.
- All the instances (objects of a class) share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object.

Java OOPS: Object

- The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory.
- To create an **Dog Object**:

```
Dog tuffy = new Dog("tuffy","papillon",5, "white");
```


Java OOPS: Constructors

- A Java constructor is special method that is **called when an object is instantiated**
- Constructors take in **zero or more** variables to create an **Object**
- Constructors have the same name as the class and have no return type
- Constructor overloading is their most useful functionality
- All classes have at least **one** constructor. If a class does not explicitly declare any, the Java compiler automatically provides a no-argument constructor, also called the default constructor.

```
public class Dog
{
    // Instance Variables
    String name;
    String breed;
    int age;
    String color;

    // Constructor Declaration of Class
    public Dog(String name, String breed,
               int age, String color)
    {
        this.name = name;
        this.breed = breed;
        this.age = age;
        this.color = color;
    }

    // method 1
```

What is this?

Constructor


```
public class Dog
{
    // Instance Variables
    String name;
    String breed;
    int age;
    String color;
    // Constructor 1
    public Dog(String name, String breed,
               int age, String color)
    {
        this.name = name;
        this.breed = breed;
        this.age = age;
        this.color = color;
    }
    // Constructor 2
    public Dog(String name, String breed)
    {
        this.name = name;
        this.breed = breed;
        this.age = 0;
        this.color = "Black";
    }
```


Java OOPS: Inheritance

- Enables one class to inherit ***methods*** (*behavior)* and ***attributes*** from another class.

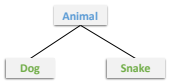

```
class Animal{
    void eat(){ System.out.println("eating..."); }
}

class Dog extends Animal{
    void bark(){ System.out.println("barking..."); }
}

class TestInheritance{
    public static void main(String args[]){
    Dog d = new Dog();
    d.bark();
    d.eat();
}
```

Superclass

Subclass

Java OOPS: Inheritance

- This introduces **subclasses** and **superclasses**.
- A class that *inherits* from another class is called a subclass:
  - **Dog** *inherits* from **Animal**, and therefore **Dog** is a **subclass**.
- The class that is *inherited* is called a superclass:
  - **Animal** is *inherited*, and is the **superclass**.



Java OOPS: Inheritance

- Organizes related classes in a hierarchy:
  - This allows reusability and extensibility of common code
- Subclasses extend the functionality of a superclass
- Subclasses inherit all the methods of the superclass (***excluding constructors and privates***)
- Subclasses can **override** methods from the superclass (*more on this later*)

What's an example use case of class inheritance?

Java OOPS: Access Control

Access modifiers describe the accessibility (*scope*) of data like:

- Attributes:

```
public String name;
```

- Methods:

```
public String getName() { … }
```

- Constructors:

```
private Student(String name, int sAge) { … }
```


Java OOPS: Access Control

- Access modifiers include:
  - Default
  - Public
  - Protected
  - Private

Java OOPS: Access Control

- Access modifiers include:
  - **Default**
  - Public
  - Protected
  - Private

Java OOPS: Access Control

```
package p1;

class Rec
{
    void display()
    {
        System.out.println("Hi!");
    }
}
```

```
package p2;
import p1.*;

class RecNew
{
    public static void main(String args[])
    {
        // Accessing Rec from package p1
        Rec obj = new Rec();

        obj.display();
    }
}
```

Error

Java OOPS: Access Control

- Access modifiers include:
  - Default
  - **Public**
  - Protected
  - Private

Java OOPS: Access Control

```
package p1;

class Rec
{
    public void display()
    {
        System.out.println("Hi!");
    }
}
```

```
package p2;
import p1.*;

class RecNew
{
    public static void main(String args[])
    {
        // Accessing Rec from package p1
        Rec obj = new Rec();

        obj.display();
    }
}
```

Prints "Hi!"

Java OOPS: Access Control

- Access modifiers include:
  - Default
  - Public
  - **Protected**
  - Private

Java OOPS: Access Control

- Access modifiers include:
  - **Protected:**
    - You can use this only in the following
      - Same class as the variable,
      - Any subclasses of that class,
      - Or classes in the same **package**.
    - A **package** is a group of related classes that serve a common purpose.

Java OOPS: Access Control

```
package p1;

class Rec
{
    protected void display()
    {
        System.out.println("Hi!");
    }
}
```

```
package p2;
import p1.*;

class RecNew extends Rec
{
    public static void main(String args[])
    {
        // Accessing Rec from package p1
        RecNew obj = new RecNew();

        obj.display();
    }
}
```

Prints "Hi!"

Java OOPS: Access Control

- Access modifiers include:
  - Default
  - Public
  - Protected
  - **Private**

Java OOPS: Access Control

```
package p1;

class Rec
{
    private void display()
    {
        System.out.println("Hi!");
    }
}
```

```
package p2;
import p1.*;

class RecNew extends Rec
{
    public static void main(String args[])
    {
        // Accessing Rec from package p1
        RecNew obj = new RecNew();
        obj.display();
    }
}
```

Error!

Java OOPS: Object & Class Variables

- Each `Animal` object has its own `name`, `age`, etc…
  - `name` and `age` are examples of **Object Variables**.
- When an attribute should describe an **entire class** of objects instead of a specific object, we use **Class Variables** (or **Static Variables**).
- There's only one copy of class variables for the entire class, regardless of how many objects are created from it.

Java OOPS: Object & Class Variables

```
public class Animal {
   public static final String currentPlanet = "EARTH";
}

   public class Test() {
        public static void main(String[] args) {
              Animal foobar = new Animal();
              String planet = foobar.currentPlanet;
        }
}
```


Java OOPS: Object & Class Variables

```
public class Animal {
   public static final String currentPlanet = "EARTH";
}

   public class Test() {
        public static void main(String[] args) {
              Animal foobar = new Animal();
              String planet = Animal.currentPlanet;
        }
}
```

Java OOPS: Encapsulation

- Encapsulation is **restricting access to an object's components**.
- How can we change or access `name` now?:

```
public class Animal {
   private String name;
   private int age;


}
```


Java OOPS: Encapsulation

- Encapsulation is **restricting access to an object's components**.
- Using **getters** and **setters**:

```
public class Animal {
   private String name;
   private int age;

   public void setName(String newName) {
         this.name = newName;
   }
   public String getName() {
         return name;
   }
}
Animal foobar = new Animal();
foobar.setName("Foo Bar");
```

Why would we do that?

Java OOPS: Overloading Methods

- Methods overload one another when they have the same method name but:
  - The **number of parameters** is different for the methods
  - The parameter **types** are different (i..e. different signatures)
- **Example:**

```
public void changeDate(int year) {
   // do cool stuff here
}

public void changeDate(int year, int month) {
   // do cool stuff here
}
```

Why would we do that?

Java OOPS: Overloading Methods

- Methods overload one another when they have the same method name but:
  - The **number of parameters** is different for the methods
  - The parameter **types** are different (i..e. different signatures)
- **Another Example:**

```
public void addSemesterGPA(float newGPA) {
   // process newGPA
}

public void addSemesterGPA(double newGPA) {
   // process newGPA
}
```

Java OOPS: Overloading Methods

- Methods overload one another when they have the same method name but:
  - The **number of parameters** is different for the methods
  - The parameter **types** are different (i..e. different signatures)
- **Another Example:**

```
public void changeDate(int year) {
    // do cool stuff here
}

public void changeDate(int month) {
    // do cool stuff here
}
```


Java OOPS: Overloading Methods

- Methods overload one another when they have the same method name but:
  - The **number of parameters** is different for the methods
  - The parameter **types** are different
- **Another Example:**

```
public void changeDate(int year) {
    // do cool stuff here
}

public void changeDate(int month) {
    // do cool stuff here
}
```

We can't overload methods by just changing the parameter name!

Java OOPS: Overriding Methods

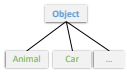
- Example:

```
public class ClassA {
    public Integer someMethod() {
            return 3;
    }
}

public class ClassB extends ClassA {

    // this is method overriding:
    public Integer someMethod() {
          return 4;
    }
}
```

Example use case?

Java OOPS: Overriding Methods

- Any class extends the **Java** superclass "**`Object`**".
- The Java "**`Object`**" class has 3 important methods:
  - `public boolean equals(Object obj);`
  - `public int hashCode();`
  - `public String toString();`
- The hashCode is just a number that is generated by any object:
  - It **shouldn't** be used to compare two objects!
  - Instead, **override** the equals, hashCode, and toString methods.



Java OOPS: Overriding Methods

- Example: **Overriding** the `toString` and `equals` methods in our `Dog` class:

```
public class Dog {
   …
   public String toString() {
      return this.name;
   }
}
```


Java OOPS: Overriding Methods

- Example: **Overriding** the `toString` and `equals` methods in our `Dog` class:

```
public class Dog {
   …
   public boolean equals(Object obj) {
        if (obj.getClass() != this.getClass())
            return false;
        else {
            Dog s = (Dog) obj;
            return (s.getName().equals(this.name));
        }
   }
}
```


Java OOPS: Abstract Classes

- A class that is **not completely implemented**.
- Contains one or more *abstract* methods (methods with no bodies; *only signatures*) that subclasses must implement
- Cannot be used to instantiate objects
- Abstract class header:

```
accessModifier abstract class className
public   abstract class Car
```

- Abstract method signature:

```
accessModifier abstract returnType methodName ( args );
public         abstract int        max_speed  ();
```

- Subclass signature:

```
accessModifier class subclassName extends  className
public         class Mercedes     extends  Car
```


Java OOPS: Interfaces

- A **special abstract class** in which *all the methods are abstract*
- Contains only abstract methods that subclasses **must implement**
- Interface header:

```
accessModifier interface interfaceName
public         interface Car
```

- Abstract method signature:

```
accessModifier abstract returnType methodName ( args );
public         abstract String     CarType    ( args );
```

- Subclass signature:

```
accessModifier class subclassName   implements someInterface
public         class BMW            implements Car
```

Java OOPS: Generic Methods

- *Generic* or *parameterized* classes/methods receive the data-type of elements as a parameter
- E.g.: a generic method for sorting elements in an array (be it `Integers`, `Doubles`, `Objects` etc.)

A Simple Box Class

What's the problem?

- Consider this non-generic Box class:

```
public class Box {
    private Object object;
    public void set(Object object) {
        this.object = object;
    }
    public Object get() {
        return object;
    }
}
```


A Simple Box Class

- A *generic class* is defined with the following format:

```
class my_generic_class <T1, T2, ..., Tn> {
        /* ... */
}
```

Type parameters

A Simple Box Class

- Now to make our Box class *generic*:

```
public class Box<T> {
        // T stands for "Type"
        private T t;
        public void set(T t) {
                this.t = t;
        }
        public T get() {
                return t;
        }
}
```

To create, for example, an `Integer "Box"`:

```
Box<Integer> integerBox;
```

Java Generic Collections

- Classes that represent data-structures
- *Generic* or *parameterized* since the elements' data-type is given as a parameter
- E.g.: LinkedList, Queue, ArrayList, HashMap, Tree
- They provide methods for:
  - Iteration
  - Bulk operations
  - Conversion to/from arrays


Class LinkedList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.AbstractSequentialList<E>
        java.util.LinkedList<E>

Type Parameters:
E - the type of elements held in this collection

All Implemented Interfaces:
Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>

public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable