Java & OO basics

Tuesday, October 2

Oregon State University

Announcements

Sprint 0 grades have been posted

Contact your team members

Java

Object oriented, statically typed, imperative language

- With a few functional constructs

Syntax influenced heavily by C++

Compiles to **bytecode**, that is then executed by a virtual machine (JVM)

- Platform independent (as long as you have a JVM)

Decomposing programs

In many languages (e.g. C), programs are decomposed into functions, that operate on common data structures.

This is called ***functional decomposition***

Functional decomposition

Pros:

- Easy to add new functions or features

Cons:

- Modern systems perform **more than one function**
- Systems evolve, their functions change

Object Oriented Decomposition

A system is decomposed according to the **objects** a system is supposed to manipulate.

Objects communicate through well defined interfaces.

OO Concepts

There are 3 core concept at the heart of OO:

1. Encapsulation

2. Inheritance

3. Polymorphism

Encapsulation

Group together data (variables) and methods (functions) in one unit.

Also, all variables should be **hidden** (private) and only accessible by the methods in the class.

Classes

A class is a template for creating objects.

Example: a car

it has two attributes: brand name and fuel level

and two methods: drive and refuel

```
public class Car {

    private String brandName;
    private double fuelLevel;

    public Car(String brandName) {
        this.brandName = brandName;
        fuelLevel = 10;
    }

    public void drive() {
        fuelLevel = fuelLevel - 1;
    }

    public void refuel() {
        fuelLevel = 10;
    }
}
```

```
public class Car {

    private String brandName;
    private double fuelLevel;

    public Car(String brandName) {
        t
        f
    }

    publi
        f
    }

    public void refuel() {
        fuelLevel = 10;
    }

}
```

These an attribute of the class.

In Java, attributes are known as **fields.**

The **private** keyword specifies that the attribute is only accessible by the method of that class.

```
public class Car {

    private String brandName;
    private double fuelLevel;

    public Car(String brandName) {
        this.brandName = brandName;
        fuelLevel = 10;
    }

    public void drive() {
        fuelLevel = fuelLevel – 1;
    }

    public void refuel() {
        fuelLevel = 10;
    }
}
```

```
public class Car {

    private String brandName;
    private double fuelLevel;

    public Car(String brandName) {
        this.brandName = brandName;
        fuelLevel = 10;
    }

    public vo
        fuelL
    }

    public vo
        fuelL
    }
}
```

This is the **constructor**.

It is used for creating objects, with the *new* keyword

The **this** keyword disambiguates between the field and parameter.

```java
public class Car {

    private String brandName;
    private double fuelLevel;

    public Car(String brandName) {
        this.brandName = brandName;
        fuelLevel = 10;
    }

    public void drive() {
        fuelLevel = fuelLevel - 1;
    }

    public void refuel() {
        fuelLevel = 10;
    }
}
```

```
public class Car {

    private String brandName;
    private doubl

    public Car(St
        this.brar
        fuelLevel
    }

    public void drive() {
        fuelLevel = fuelLevel - 1;
    }

    public void refuel() {
        fuelLevel = 10;
    }
}
```

These are **methods**.

Methods are operations that this object can perform

```java
public class Car {

    private String brandName;
    private double fuelLevel;

    public Car(String brandName) {
        this.brandName = brandName;
        fuelLevel = 10;
    }

    public void drive() {
        fuelLevel = fuelLevel - 1;
    }

    public void refuel() {
        fuelLevel = 10;
    }
}
```

Access modifiers

`public` - anybody can access (same as C++)

`protected` - only code in subclasses can access (same as C++) & code in the same package

default (package) - only code in the same package can access

`private` - only code in the same class can access (same as C++)

Information hiding

The **private** keyword is used to keep all data hidden

But what if I want to access, or to change, the value outside of a class?

We define special methods, **getters** and **setters**

Only define getters and setters if you need them!

```java
public double getFuelLevel() {
    return fuelLevel;
}

public void setBrandName(String brandName) {
    this.brandName = brandName;
}
```

Creating objects

Objects are created with the **new** keyword

```
Car car = new Car("Ford");
```

This invokes the constructor with the right parameters.

Type inference

You omit the variable type and write

```
var car = new Car("Ford");
```

The compiler will infer that `car` is of type `Car`

Method parameters must have a type

`public Car(val brandName){…}` will not compile

Inheritance

Also known as **subclassing** or **subtyping**

Classes can inherit fields and methods from other classes with the **extends** keyword.

We want to model a Sedan, that has all the fields and methods of a car.

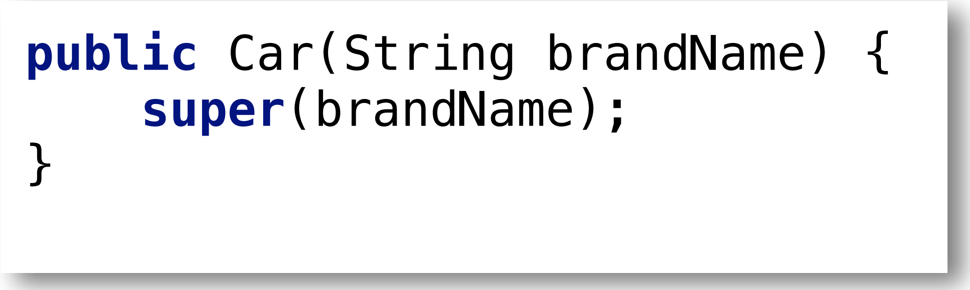
Defines a "is-a" relationship between classes.

```java
public class Sedan extends Car {

    private int noOfDoors = 4;

    public Sedan(String name) {
        super(name);
    }
}
```

```
public class Sedan extends Car {

    private int

    public Sedan
        super(name);
    }
}
```

The class declaration now contains the **extends** declaration

```java
public class Sedan extends Car {

    private int noOfDoors = 4;

    public Sedan(String name) {
        super(name);
    }
}
```



```java
public Car(String brandName) {
    super(brandName);
}
```

The constructor now contains the **super** keyword. **This passes the parameters to Car's constructor.**

```
public cla

    private int noOfDoors = 4;

    public Sedan(String name) {
        super(name);
    }
}
```

```
public Car(String brandName) {
    super(brandName);
}
```

Inheritance

`Sedan` now inherits `Car`'s attributes and method:

```java
Sedan s = new Sedan("Ford");
s.drive();
```

Inheritance

Java only supports **single inheritance** (you can only extend one class)

All classes, by default, extend `Object`.

Polymorphism

Polymorphism means taking different forms

In Java, this refers to the fact that a subclass can always be used instead of a parent class.

e.g. You can use a `Sedan` object, even if a `Car` is required:

```
Car c = new Sedan("Ford");
```

Class hierarchies

We want to model a boat. It has a brand name, a fuel level, but it cannot drive.

We can create an **abstract** class, `Vehicle`, from which we can extend for `Car` and `Boat`

```java
public class Vehicle {

    private String brandName;
    protected double fuelLevel;

    public Vehicle(String brandName) {
        fuelLevel = 10;
        this.brandName = brandName;
    }

    public void refuel() {
        fuelLevel = 10;
    }

    public double getFuelLevel() {
        return fuelLevel;
    }

    public void setBrandName(String brandName) {
        this.brandName = brandName;
    }
}
```

```
public class Vehicle {

    private String brandName;
    protected double fuelLevel;
```

The **protected** keyword allows subclasses to access this field

```
                                      {

    public void refuel() {
        fuelLevel = 10;
    }

    public double getFuelLevel() {
        return fuelLevel;
    }

    public void setBrandName(String brandName) {
        this.brandName = brandName;
    }
}
```

```
public class Vehicle {

    private String brandName;
    protected double fuelLevel;

                                          {

    public void refuel() {
        fuelLevel = 10;
    }

    public double getFuelLevel() {
        return fuelLevel;
    }

    public void setBrandNa
        this.brandName = b
    }
}
```

The **protected** keyword allows subclasses to access this field

We **extracted** all the common functionality between Car and Boat (the name and the fuel) into it's own class

```
public class Car extends Vehicle {

    public Car(String brandName) {
        super(brandName);
    }

    public void drive() {
        fuelLevel = fuelLevel - 1;
        // some other code that "drives" the car
    }

}
```

```
public class Car extends Vehicle {

    public Car(String brandName) {
        super(brandName);
    }

    public void drive() {
        fuelLevel = fuelLevel - 1;
        // some other code that "drives" the car
    }

}
```

We access the protected field in `Vehicle`

```java
public class Boat extends Vehicle {

    public Boat(String name) {
        super(name);
    }

    public void sail() {
        fuelLevel = fuelLevel - 1;
        // some code relating to sailing
    }
}
```

```
public class Boat extends Vehicle {

    public Boat(String name) {
        super(name);
    }

    public void sail() {
        fuelLevel = fuelLev
        // some code relati
    }
}
```

The class **Boat** now has to deal only with Boat specific stuff

Abstract Classes & Methods

You can define **abstract** classes, that cannot be instantiated

```
public abstract class Vehicle {…}
```

`val v = new Vehicle();` will not compile

Abstract methods have no implementation, and can only be declared in abstract classes

```
public abstract void drive();
```

Interfaces

An interface is abstract type, like an abstract class, that only contains method signatures and fields (static of final).

```
public interface Driveable {
    public void drive();
}
```

Like abstract classes, you can not instantiate interfaces.

Interfaces

A class can extend an interface using the implements keyword

```
public class Car implements Drivable {
    …
}
```

A class can implement more than one interface

What's the advantage?

It allows us to write code that is more generic

```
public void refuel(Vehicle v) {
    v.refuel();
}
```

This will work with any vehicle.

It keeps the code clean, and easy to maintain.

What's the advantage?

It allows us to write code that is more generic

```
public void refuel(Vehicle v) {
    v.refuel();
}
```

Dynamic polymorphism

This will work with any vehicle.

It keeps the code clean, and easy to maintain.

Method overloading

In Java, multiple methods can have the same name, as long they have different parameters (type and/or numbers)

```
public void refuel() {
    fuelLevel = 10;
}

public void refuel(int x) {
    fuelLevel = x;
}
```

Method overloading

In Java, multiple methods can have the same name, as long they have different parameters (type and/or numbers)

```java
public void refuel() {
    fuelLevel = 10;
}

public void refuel(int x) {
    fuelLevel = x;
}
```

Static polymorphism