Lecture 6: Introduction to Class

Sub: Object Oriented Programming

Course Code: CSE-1205

Object-oriented Programming

- **Object-oriented programming (OOP)** is a programming paradigm that uses “Objects” and their interactions to design applications.

- It simplifies the software development and maintenance by providing some concepts:

  - Object
  - Class
  - Data Abstraction & Encapsulation
  - Inheritance
  - Polymorphism
  - Message Passing

Object-oriented programming

- Basis
  - Create and manipulate objects with attributes and methods that the programmer can specify

- Mechanism
  - Classes

- Benefits
  - An information type is designed and implemented once
    - Reused as needed
      - No need reanalysis and re-justification of the representation

Class

- A class is a definition of a well defined entity, containing attributes and methods.
- In fact, Objects are variables of the type class.
- Once a class has been defined, we can create any number of objects belonging to that class.
- **Classes** are data types based on which objects are created. Objects with similar properties and methods are grouped together to form a Class. Thus a Class represents a set of individual objects.

Known Classes

- Classes we’ve seen
    - String
    - Scanner
    - System

Object

- Objects are the basic run time entities in an object- oriented system.
- They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.

- A class is thus a collection of objects of similar type. for example: mango, apple, and orange are members of the class fruit . ex: fruit mango; will create an object mango belonging to the class fruit.

- An object is an instance of a class. E.g.

Class → Fruit; mango → Object

  - Fruit mango = new Fruit();
  - mango is now an object of type Fruit.

Data Abstraction

- Abstraction refers to the act of representing essential features without including the background details or explanations.

- Since the classes use the concept of data abstraction, they are known as *abstraction data type(ADT).*

- For example, a class Car would be made up of an Engine, Gearbox, Steering objects, and many more components. To build the Car class, one does not need to know how the different components work internally, but only how to interface with them, i.e., send messages to them, receive messages from them, and perhaps make the different objects composing the class interact with each other

Example of Abstraction

- Humans manage complexity through abstraction.
- When you drive your car you do not have to be concerned with the exact internal working of your car(unless you are a mechanic).
- What you are concerned with is interacting with your car via its interfaces like steering wheel, brake pedal, accelerator pedal etc.
- Various manufacturers of car has different implementation of car working but its basic interface has not changed (i.e. you still use steering wheel, brake pedal, accelerator pedal etc to interact with your car). Hence the knowledge you have of your car is abstract.

Inheritance

- *Inheritance* allows to reuse classes by deriving a new class from an existing one

- The existing class is called the *parent class*, or *superclass*, or *base class*

- The derived class is called the *child class* or *subclass*.

- The child class inherits characteristics of the parent class (i.e the child class *inherits* the methods and data defined for the parent class)

Polymorphism

- Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms.

- So polymorphism means many forms.

- Many different (poly) forms of objects that share a common interface respond differently when a method of that interface is invoked:

  - sub-classes have to follow this interface inheritance), but are also permitted to provide their own implementations (overriding)
  - a super-class defines the common interface

Polymorphism Example

- Suppose we create a program that keeps tracks of the movement of several types of animals for a biological study.

- Classes **Fish, Frog** and **Bird** represent the three types of animals under investigation.
  - Each class extends superclass **Animal**, which contains a method move and maintains an animal's current location as *x*-*y* coordinates.
  - Each subclass implements method move.
  - A program maintains an **Animal** array containing references to objects of the various **Animal** subclasses.
  - To simulate the animals' movements, the program sends each object the same message—namely, move.

The Car class

A new example: creating a Car class

- What properties does a car have in the real world?
  - Color
  - Position (x,y)
  - Fuel in tank
- We will implement these properties in our Car class

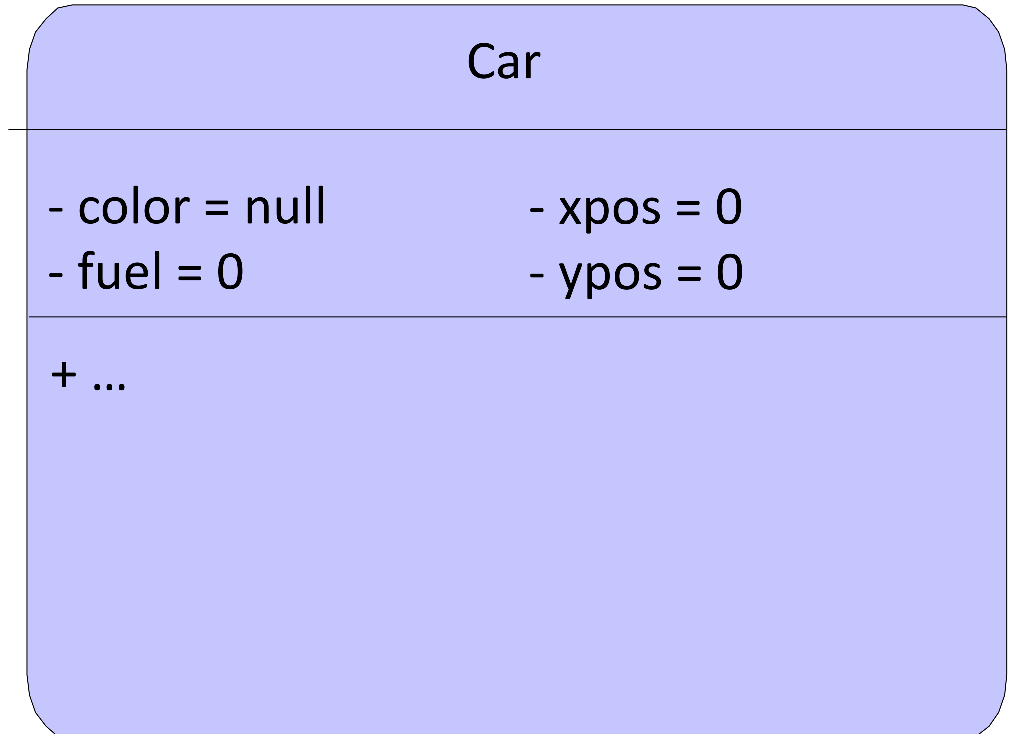
```
class Car {
    private Color color;
    private int xpos;
    private int ypos;
    private int fuel;
    //...
}
```

Car's instance variables

```
class Car {
    private Color color;
    private int xpos;
    private int ypos;
    private int fuel;

    //...
}
```

Instance variables and attributes

- Default initialization
  - If the variable is within a method, Java does NOT initialize it
  - If the variable is within a class, Java initializes it as follows:
    - Numeric instance variables initialized to 0
    - Logical instance variables initialized to false
    - Object instance variables initialized to null



Car behaviors or methods

- What can a car do? And what can you do to a car?
  - Move it
    - Change it's x and y positions
  - Change it's color
  - Fill it up with fuel

- For our computer simulation, what else do we want the Car class to do?
  - Create a new Car
  - Change Car's condition
- Each of these behaviors will be written as a method

Creating a new car

- To create a new Car, we call:

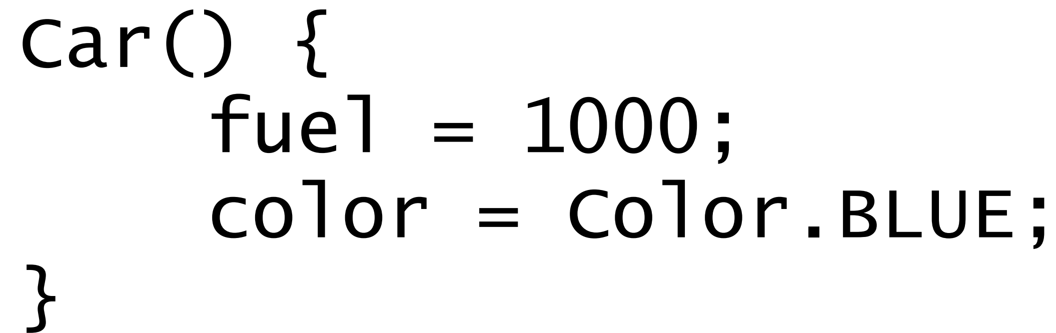
  - Car car = new Car();

- Notice this looks like a method
  - You are calling a special method called a constructor
  - A constructor is used to create (or construct) an object
    - It sets the instance variables to initial values

- The constructor:

```
Car() {
  fuel = 1000;
  color = Color.BLUE;
}
```

Constructors




```
Car() {
    fuel = 1000;
    color = Color.BLUE;
}
```

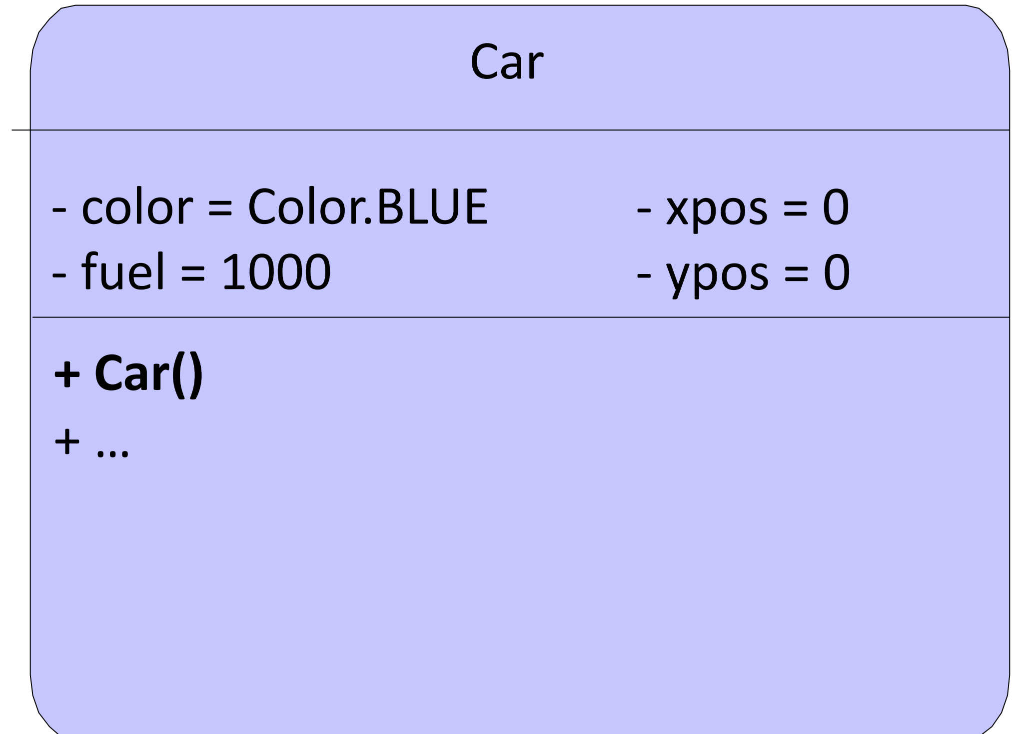
Our Car class so far

```
class Car {
    private Color color;
    private int xpos;
    private int ypos;
    private int fuel;

     Car() {
        fuel = 1000;
        color = Color.BLUE;
      }
}
```

```
class Car {
    private Color color =
              Color.BLUE;
    private int xpos;
    private int ypos;
    private int fuel = 1000;

      Car() {
      }
}
```

Our Car class so far

```
class Car {
    private Color color =
                Color.BLUE;
    private int xpos = 0;
    private int ypos = 0;
    private int fuel = 1000;

    Car() {
    }
}
```



- Called the default constructor
  - The default constructor has no parameters
  - If you don’t include one, Java will SOMETIMES put one there automatically

Another constructor

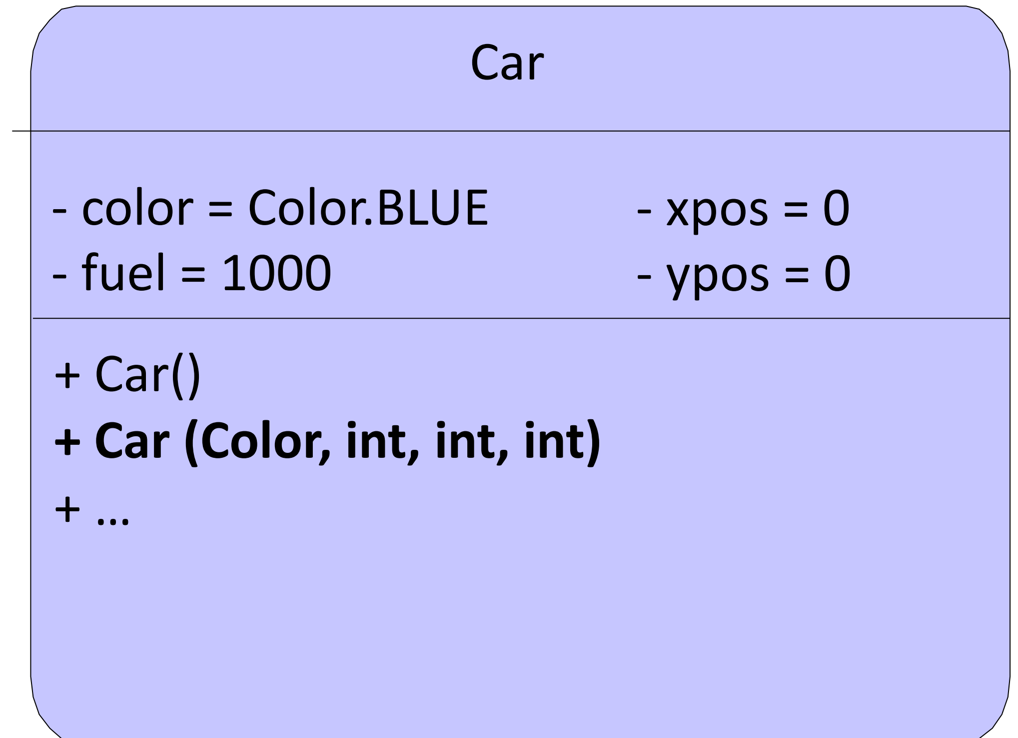
- Another constructor:

```
Car (Color c, int x, int y, int f) {
  color = c;
  xpos = x;
  ypos = y;
  fuel = f;
  }
```

- This constructor takes in four parameters
- The instance variables in the object are set to those parameters
- This is called a specific constructor
  - An constructor you provide that takes in parameters is called a specific constructor

Our Car class so far

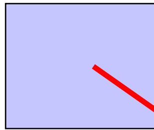
```
class Car {
    private Color color =
     Color.BLUE;
    private int xpos = 0;
    private int ypos = 0;
    private int fuel = 1000;

    Car() {
    }

    Car (Color c, int x, int y, int f) {
    color = c;
    xpos = x;
    ypos = y;
    fuel = f;
    }

}
```

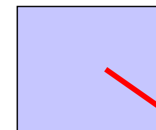


Using our Car class

- Now we can use both our constructors:

```
Car c1 = new Car();
Car c2 = new Car (Color.BLACK, 1, 2, 500);
```





So what does private mean?

- Consider the following code

```
class CarSimulation {
   public static void main (String[] args) {
      Car c = new Car();
      System.out.println (c.fuel);
   }
}
```

- Recall that fuel is a private instance variable in the Car class
- Private means that code outside the class CANNOT access the variable
  - For either reading or writing
- Java will not compile the above code
  - If fuel were public, the above code would work

So how do we get the fuel of a Car?

- Via accessor methods in the Car class:

```
public int getFuel() {
 return fuel;
}

public Color getColor() {
   return color;
}
```

```
public int getYPos() {
   return ypos;
}

public int getXPos() {
     return xpos;
   }
```

- As these methods are within the Car class, they can read the private instance variables
- As the methods are public, anybody can call them

So how do we set the fuel of a Car?

- Via mutator methods in the Car class:

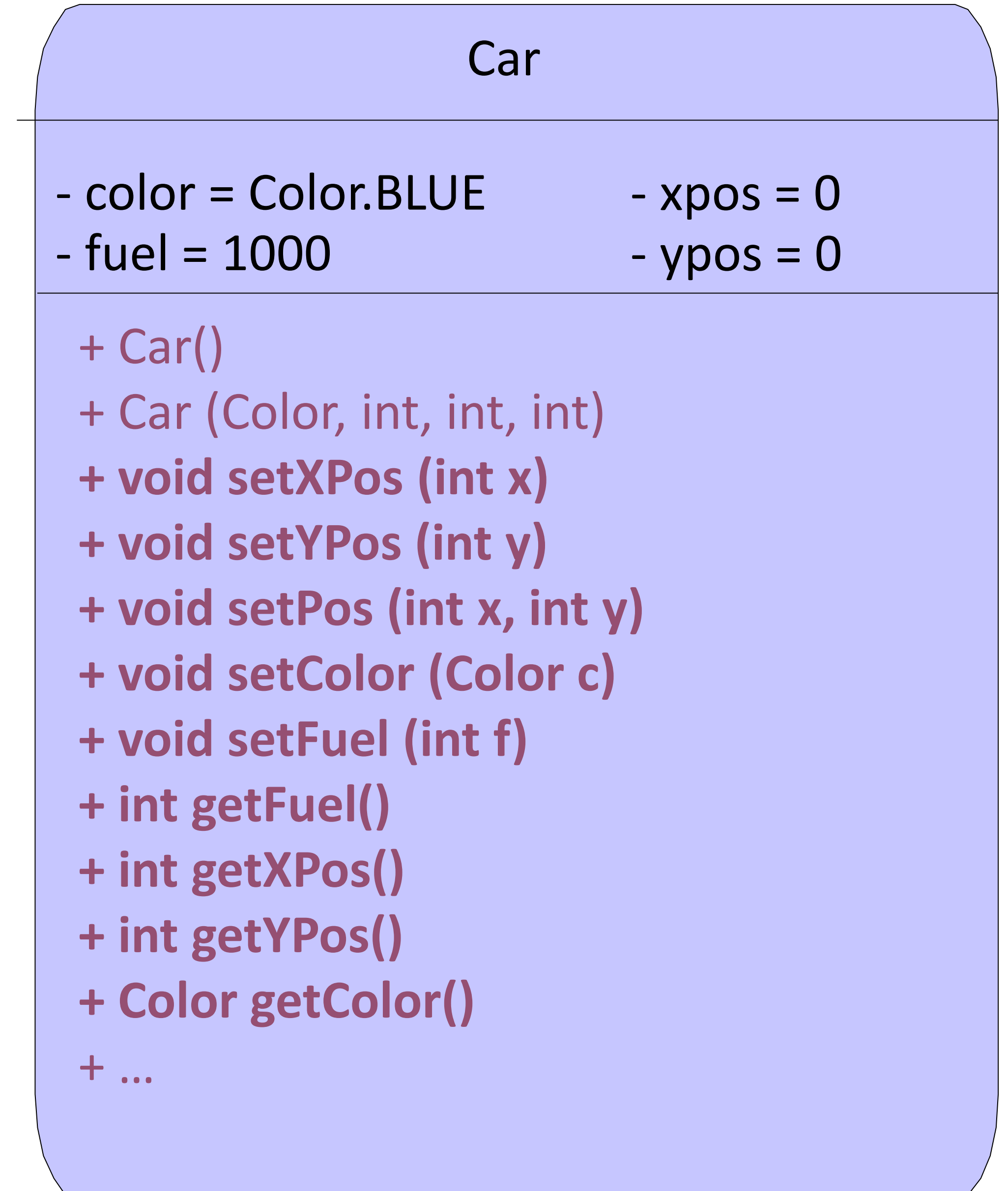
```
public void setFuel (int f) {
 fuel = f;
}

public void setColor (Color c) {
 color = c;
}
```

```
public void setXPos (int
x) {
   xpos = x;
}

public void setYPos (int
y) {
   ypos = y;
}
```

- As these methods are within the Car class, they can read the private instance variables
- As the methods are public, anybody can call them

Why use all this?

- These methods are called a get/set pair
  - Used with private variables
- Our Car so far:



Back to our specific constructor

```
class Car {
  private Color color =
            Color.BLUE;
  private int xpos = 0;
  private int ypos = 0;
  private int fuel = 1000;

  Car (Color c,
      int x, int y, int f) {
      color = c;
      xpos = x;
      ypos = y;
      fuel = f;
    }
}
```

```
class Car {
    private Color color =
            Color.BLUE;
    private int xpos = 0;
    private int ypos = 0;
    private int fuel = 1000;

    Car (Color c,
        int x, int y, int f) {
        setColor (c);
        setXPos (x);
        setYPos (y);
        setFuel (f);
    }
}
```

Back to our specific constructor

- Using the mutator methods (i.e. the ‘set’ methods) is the preferred way to modify instance variables in a constructor

So what's left to add to our Car class?

- What else we should add:
  - A mutator that sets both the x and y positions at the same time
  - A means to "use" the Car's fuel
- Let's do the first:

```
public void setPos (int x, int y) {
 setXPos (x);
 setYPos (y);
}
```

- Notice that it calls the mutator methods

Using the Car's fuel

- Whenever the Car moves, it should burn some of the fuel
    - For each pixel it moves, it uses one unit of fuel
        - We could make this more realistic, but this is simpler

```
public void setXPos (int x) {
   xpos = x;
}



public void setYPos (int y) {
   ypos = y;
}
```

```
public void setXPos (int x) {
   fuel -= Math.abs
          (getXPos()-x);
   xpos = x;
}
public void setYPos (int y) {
   fuel -= Math.abs
          (getYPos()-y);
   ypos = y;
}
```

Using the Car’s fuel

```
public void setPos (int x, int y) {
   setXPos(x);
   setYPos(y);
}
```

- Notice that to access the instance variables, the accessor methods are used
- Math.abs() gets the absolute value of the passed parameter