CSN-103: Fundamentals of Object Oriented Programming

Object Oriented Programming (OOP)

- OOP language provides us mechanism to implement object oriented model
  - Encapsulation
  - Inheritance
  - Polymorphism

The OOP Principles

- And how all these three work together
  - Robust and scalable

Encapsulation

- Encapsulation
  - Binds together the code and data it manipulates
  - Safety from outside interference and misuse
- Act like a wrapper
  - Access to code and data inside a wrapper is via well defined interfaces

Encapsulation

- Example: A Car

  - Combine all the component under the hood
    - Engine, transmission, cooling system, music system…
  - Well defined interfaces
    - Steering, bakes, accelerating, locking, playing music…
  - Safety
    - Can’t control car’s movement via music system buttons

Encapsulation and Classes

- In OOP, basis of encapsulation is *Class*
  - Defines the structure (Data)
  - Defines the behavior (Code)
- Class contains
  - Data
  - Code

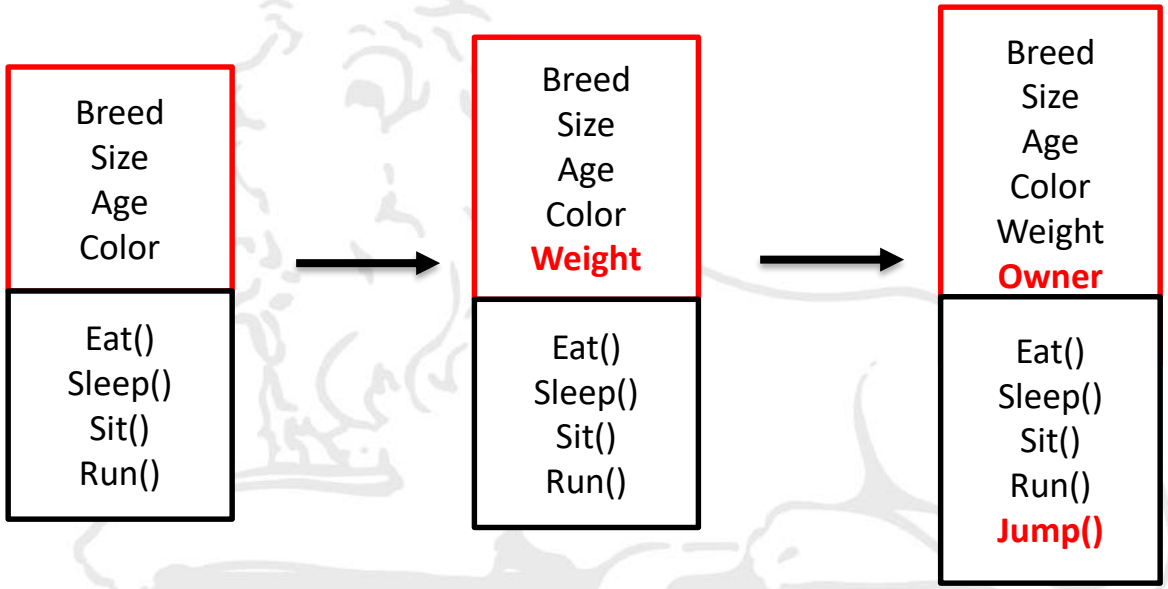
  Members of a Class → Member/Instance Variables; Member Methods
- Class encapsulate complexity
- Encapsulation vs. Abstraction

Abstraction is focused mainly on what should be done, while Encapsulation is focused on how it should be done.

The major difference between abstraction and encapsulation is that abstraction hides the code complexity while encapsulation hides the internal working from the outside world.

Inheritance

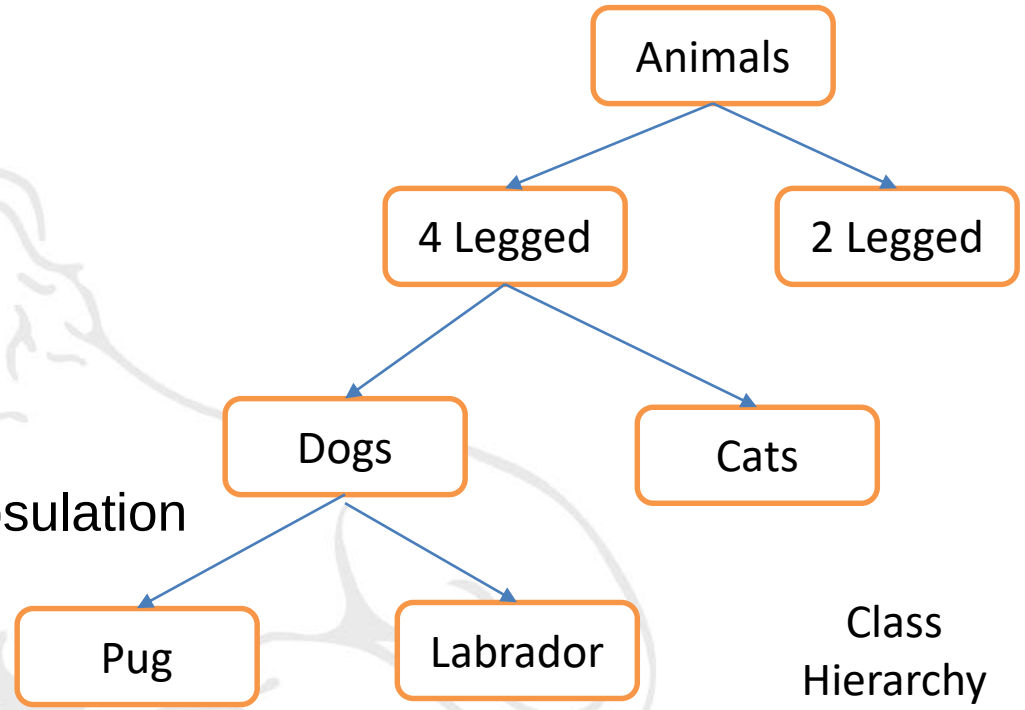
- One object acquire properties of another object



- Support hierarchical classification

Inheritance

- Class hierarchy
  - Superclass (Base Class)
  - Subclass (Derived Class)
- Subclass inherits all the properties of superclass
- Advantages
  - Code reuse
  - Close interaction with Encapsulation



Class Hierarchy

Polymorphism

- “Many Forms”
- One interface for general class of actions
- Example: Dog’s sense of smell is polymorphic
  - Smells a cat → Bark and run after it
  - Smells its food → Wags the tail and run for the bowl
- Sense of smell works in both situation
  - Difference: Type of data (smell) being operated upon by dog’s nose

Polymorphism

- Polymorphism in the programming world
- Example: Calculate the area
  - For Circle: Radius r
  - For Rectangle: Length l, Width w
  - For Sector: Radius r, Angle θ
- A single function to calculate all the above
  - Differentiate: Data Type and Number

Classes

- **Class** is a logical construct
  - Defines the shape and nature of an object
- Class contains:
  - Data
  - Code operating on the data
  - May contain only data or code
- A class is declared by the keyword **class**

General Form

```
class classname {
        type instance-variable1;
        type instance-variable2;
        …
        type instance-variableN;

        type methodname1(parameter-list) {
        // body of method
        }
        type methodname2(parameter-list) {
        // body of method
        }
        ...
        type methodnameN(parameter-list) {
        // body of method
        }
}
```

Data → Instance Variables

Code → Methods

Members

main() Method

- All methods have the same general form as main()
  - Not all methods are **static** or **public**
- General form of a class doesn't specify a main() method
  - Java classes **do not need** to have a **main()**
  - main() is specified if the class is the **starting point** of execution of your program

A Simple Box Class

```
class Box {
    double width;
    double height;
    double depth;
}
```

- Defines a new class: a new type of data
- Creates a **template**, not an actual **instance/object**

- To create an **object** of class **Box**

```
Box mybox = new Box();
```

  It creates a Box object called **mybox**
- **mybox** is an instance/object of class **Box**

A Simple Box Class

- Each object of class will have its own copy of instance variable: `width, height, and depth`
- To access these variables (and methods):
  - Dot(.) operator
- For example:

```
mybox.width=100;
mybox.height=10;
mybox.depth=20;
```

A Complete Program Using the **Box** Class

Box.java BoxDemo.java

Example: BoxDemo.java

- After compilation, two .class files are created, one for **Box** and another for **BoxDemo**
- Single .java file can have multiple class definitions
  - What should be the name of .java file??
- The default access is **package-private**
  - **By default,** all classes within a “folder” (package) can access each others members (data and code).

Declaring Objects

- To obtain an object of a class
  - Declare a variable of that class type
  - Acquire an actual, physical copy of the object and assign it to a variable

```
Box mybox = new Box();
```

OR

```
Box mybox;          // Declaring reference to object, contains null
mybox = new Box();  // Allocate a Box object
```

new Operator

- **new** operator *dynamically* allocates memory for an object

```
class-var = new classname();
```
← Constructor of the class

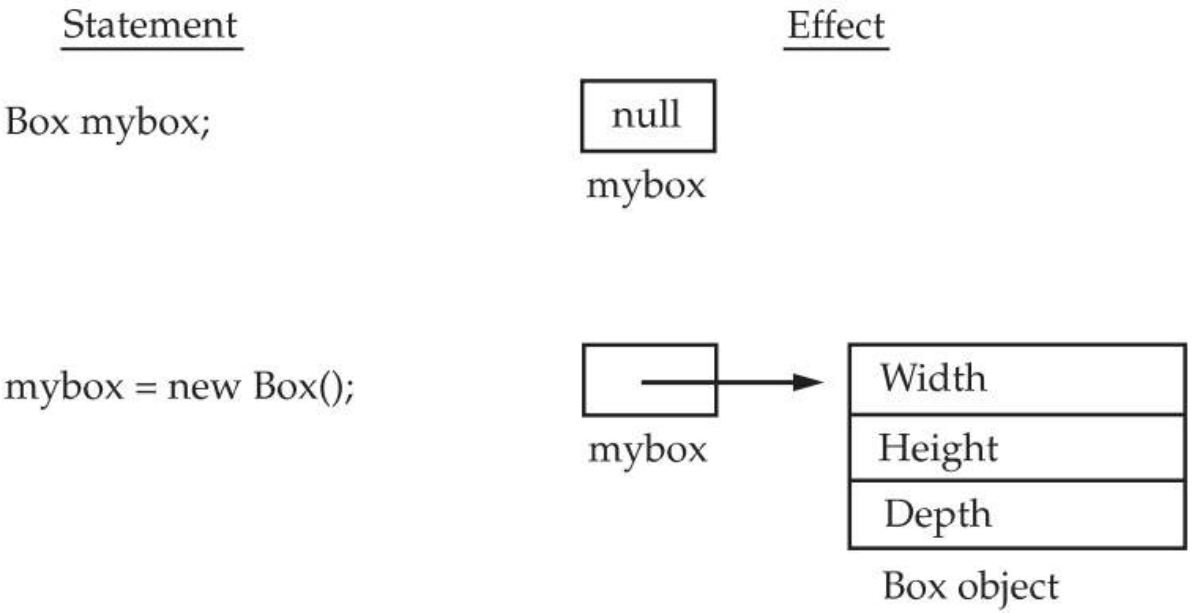
- *Constructor* defines what occurs when an object of **a class** is created

- We **can** define our own constructors
- If no explicit constructor is specified, then Java automatically provide a **default constructor**

- Why we don't use **new** operator for primitive types??
  - Primitive types are not implemented as objects
  - "Normal" variables → For efficiency

new Operator

- **new** allocated memory at the **runtime**



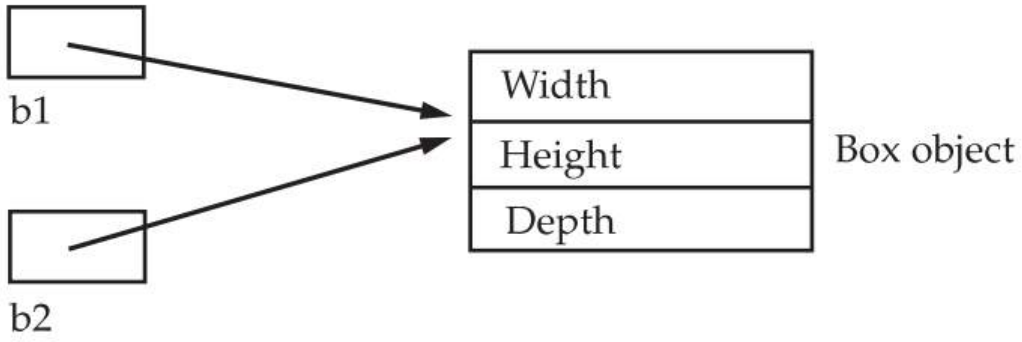
- Can create as many object as we want
  - May cause error at the runtime

Object Reference Variables

- Example:

```
Box b1 = new Box();
Box b2 = b1;
```

- *b1* and *b2* **do not** refer to separate and distinct objects
- Assignment did not allocate any memory or copy any part of the original object
- Any changes made to the object through **b2** will affect the object to which **b1** is referring
- Assignment only creates a copy of the reference not the object



Methods

- General form:

```
type name(parameter-list){
// Body  of the method
}
```

- **type** specify the type of data returned by the method
  - Any valid return type, even **void**
- **parameter-list**: Sequence of **type-identifier** pair
  - Variables that receives the value of the arguments
- **return** statement is the **type** is not **void**

```
return value;
```

Box Class with Parameterized Method

BoxDemo5.java

Constructors

Object Initialization

- Sometimes, it is necessary to **initialize** all the variables of an object **upon creation**
- **One way:** Directly initialize the instance variables

```
class Box
{
    double width=10;
    double height=20;
    double depth=30;
}
```

Not a Good Approach

Object Initialization

- **Another way:** Define and call a special function soon after creating an object

```
void setDim(double w, double h, double d)
{
    width = w;
    height = h;
    depth = d;
}

Box mybox1 = new Box();
mybox1.setDim(10, 20, 15);
```

Defined Inside the Class **Box**

Parameterized Method

Object Initialization

- Simpler and concise (and the best) way:
  - Automatic initialization by the use of a **constructor**
  - A constructor initializes an object immediately upon creation
- Constructor has the **same name** as the class
- A constructor is syntactically similar to a method
- Constructors don't have a return type, **not even void**

new Operator

- Let's re-examine the **new** operator

```
classname class-var = new classname();
```

```
Box mybox1 = new Box();
```

- Parentheses after the class name calls the constructor
- If no constructor is defined for a class then Java creates a **default** constructor

Default Constructor

```
class Box {
        double width;
        double height;
        double depth;
}

class BoxDemo {
        public static void main(String args[])
        {
        Box mybox1 = new Box();
        }
}
```

```
class Box
{
        double width;
        double height;
        double depth;
Box(){
        width = 0;
        height = 0;
        depth = 0;
    }
}

class BoxDemo
{
        public static void main(String args[])
        {
        Box mybox1 = new Box();
        }
}
```

Values Initialized by Default Constructor

Type	Default Value
boolean	false
byte	0
short	0
int	0
long	0L
char	\u0000
float	0.0f
double	0.0d
object	Reference null

Parameterized Constructor

- Last Example: All **Box** objects have the same dimensions
- What if you want to initialize different **Box** objects with different dimensions?
- Adding parameters to the constructor

The “this” keyword

- Sometimes a method needs to refer to the object that invoked it
- For this, Java defines the “this” keyword and it refers to the current object
- “this” is always a reference to the object

// **Redundant** use of this in:

Constructor

```
Box(double w, double h, double d)
{
    this.width = w;
    this.height = h;
    this.depth = d;
}
```

Methods

```
void setDim(double w, double h, double d)
{
    this.width = w;
    this.height = h;
    this.depth = d;
}
```

Exercise

```
class Box {
      double width;
      double height;
      double depth;

Box(double width, double height,
double depth) {
     width = width;
     height = height;
     depth = depth;
}

double volume() {
     return width * height * depth;
     }
}
```

```
class BoxDemo8 {
public static void main(String args[]) {

      Box mybox1 = new Box(10, 20, 30);

      double vol = mybox1.volume();
      System.out.println("Volume is " + vol);
     }
}
```

Instance Variable Hiding

- In Java, two local variables cannot have same name within a scope
- However, local variables and parameters to method can have the same name as class' instance variables
  - Local variable hides the instance variable
- Use ***this*** keyword to resolve name-space collisions