Data Structures & Algorithms

Subodh Kumar
(subodh@iitd.ac.in, Bharti 422)
Dept of Computer Sc. & Engg.

Reminder about Email

- Please use your IITD email
  - Entry number will be taken from address
- For mails intended for me
  - Subject line must begin with COL106
- For administrative issues
  - Email: col106admin@cse.iitd.ac.in

Data Structures

Collection of data items and operations

- INSERT
- DELETE
- FETCH
  - FIRST/LAST
  - NEXT/PREVIOUS
    - NORTH-WEST
  - BEST/WORST
- ENQUIRE
  - MORE-STATS

- Types of Data items
- Ways to identify an item
- Relationships between items
- Behavior of Data structure

What you should learn

- Argue about correctness and efficiency
- Data-centric focus
  - Efficient data organization and operations
- Data abstraction
  - Separate behavior and implementation
  - Re-use in similar situations
- Common algorithms
  - Applications
  - Build up a bag of tricks

Intro to Java

- Object oriented
- Similar to C++, but cleaner
  - Easier and faster to program in some ways
  - Popular in industry
- In-built garbage collection
  - No ‘delete’ or ‘free’ of memory
- No Pointer arithmetic
- Platform-independence (in principle)

Not C++

- Pure object oriented: “everything” is an object in Java
  - No struct, enum, unions, template, only classes (but see generic)
  - Single class hierarchy: Everything gets derived from java.lang.Object
- Java source compiles to byte code
  - JVM interprets and executes byte code
- Automatic garbage collection: no destructors, only constructors
- No macros: #ifdef, #ifndef, #include
  - import
- No global function/variable
  - static method/data within a class
- No multiple inheritance
- No pointers but “references"
  - Cannot perform arithmetic on references

Not C++

- Abstract class = interface
- *array* is a class with member .length
- package, instead of namespaces
- Automatic initialization
- No default argument
- No operator overloading
- No scope resolution operator "::"
  - Method defined only within class
- Try/catch *required* if a function may throw an exception
- Keywords const and goto are reserved, but not used
- Built-in support for threads
- Built-in support for documentation comments: /** ... */ (cf javadoc)

Object-oriented Concepts

- Class and Object
  - Visibility: default, protected, private, public
  - Constructor
- Subclass
  - Inherit: “Is A” and extend, Override
- Polymorphism
  - Name overloading, Overriding
  - Use derived object in place of super-class object
- Abstract Class, Interface

```
class Vehicle {
      void start() {
      }
       int number_of_passengers;
       float luggage_capacity;
}

Vehicle buggy = new Vehicle();
Vehicle jetliner = new Vehicle();
buggy.start();
jetliner.start();
```

Objects

- Program = collection of cooperating objects, with specified behavior
- Objects have properties and methods
- Send message to objects to perform a task
- Objects can contain other objects
  - Objects can send message to other objects
  - Or, create other objects
- Objects can be derived from other objects
- Object referenced by “handles”
  - Generated at creation

Object Oriented Programming

- Objects "abstract" *how-to*s
  - Interface of invocation clearly defined
- Each object knows "how to" do a few things
  - Sometimes by "taking help" from other objects
  - Does that thing only when someone asks it to
  - Becomes "inactive" after that thing is done
- Programming =
  - Decide what classes are needed
  - What objects should be created to begin with
  - And sent a message

Object Oriented Programming

- ~~Object~~Classes "abstract" *how-tos*
  - Interface of invocation clearly defined
- Each object knows "how to" do a few things
  - Sometimes by "taking help" from other objects
  - Does that thing only when someone asks it to
  - Becomes "inactive" after that thing is done
- Programming =
  - Decide what classes are needed
  - What objects should be created to begin with
  - And sent a message

Basics of Running

- **Compile:** `javac filename1.java`
  - **Output is** `filename1.class`
  - **Not machine code, but byte-code**
    - **Platform independence, more security**
  - **Executed by Java Virtual Machine**
    - **Actually interpreted**
- **Run java interpreter:** `java filename1`
  - **Only the main class needs to be called**
  - **Other classes must be in the 'search path'**

Hello World!

```
import java.util.*;
class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

File: file.java

- Should it be in the file `HelloWorld.java`?
- Executing `java HelloWorld` arg1 arg2
  - amounts to `HelloWorld.main(args)`
- `main` is public so JVM may call it
  - static, because it is independent of any instance

```
package Transport
class Vehicle {
    void start() {
    }
    int number_of_passengers;
    float luggage_capacity;
}
    Vehicle(int count, int cap) {
        number_of_passengers = count;
        float luggage_capacity = cap;
    }
Vehicle buggy = new Vehicle( );
Vehicle jetliner = new Vehicle( );
```

Visible within package

```
Vehicle buggy = new Vehicle(2, 40);
Vehicle jetliner = new Vehicle(250, 20000);
buggy.luggage_capacity = 100;  ?
```

```
class Vehicle {
    protected void start() {
    }
    public int number_of_passengers;
    private float luggage_capacity;
    Vehicle(int count, int cap) {
        number_of_passengers = count;
        float luggage_capacity = cap;
    }
}

Vehicle buggy = new Vehicle(2, 40);
buggy.luggage_capacity = 41;
```

Visible to subclass and also within package

Visible globally

Visible inside class

- Email to
  - col106quiz@cse.iitd.ac.in
    - Subject: a
    - Or ,Subject: b
    - .. etc.

Abstract classes

```
abstract class Vehicle {
      abstract void start() ;
      public int number_of_passengers;
      private float luggage_capacity;
      Vehicle(int count, int cap) {
            number_of_passengers = count;
            float luggage_capacity = cap;
      }
}

Vehicle buggy = new Vehicle(2, 40);
buggy.start();
```

Interfaces

```
interface ManagedList {
    void add();
    void remove();
}
```

```
class FIFO implements ManagedList {
    public void add() { .. }
    public void remove() { .. }
}
```

```
class LIFO implements ManagedList {
    public void add() { .. }
    public void remove() { .. }
}
```

```
FIFO q1;
ManagedList q2;
```

Generic Typing

```
import java.util.*;

class Data {
    int x;
    public void print() {
        System.out.println("x = " + x);
    }
}

class HelloWorld {
  public static void main(String[] args) {
     Data a = new Data();
     a.print();
  }
}
```

Fixed Type

Generic Typing

```
import java.util.*;

class Data<T> {
    T x;
    public void print() {
        System.out.println("x = " + x);
    }
}

class HelloWorld {
  public static void main(String[] args) {
    Data<Integer> a = new Data<Integer>();
    a.print();
  }
}
```

Inheritance

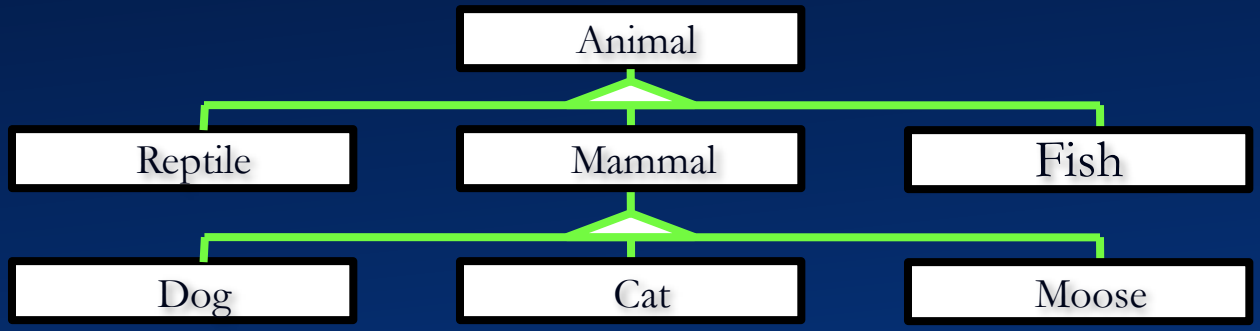
- A subclass specializes its superclass
  - May add new properties and behavior, override existing ones, implement abstract methods
- inherits all capabilities of its superclass
  - `Car`.moves() => `SportsCar`.moves()
- A superclass factors out capabilities common to its subclasses
- subclasses:
  - may not access `private` methods/variables of super
  - but does still inherit them

Class Hierarchy

- **Subclass models "is a" relationships**

  object of subclass "is an" object of superclass: it has all methods and attributes of the superclass

- **Inheritance Hierarchy of classes**



`Mammal` "*is an*" `Animal: Animal animal = someMammal;`

`Cat` "*is a*" `Mammal`

=> a `Cat` "*is an*" `Animal`,

`Reptile, Mammal` and `Fish` "*inherit from*" `Animal`

`Dog, Cat,` and `Moose` "*inherit from*" `Mammal`

Hierarchy Design

- superclass too general to define all behaviour
- superclass specifies some behaviour; subclasses inherit these
  - but may choose to override
    - maybe, using the super's behaviour
- superclass legislates an abstract behaviour and delegates implementation to subclass
- subclass specifies its own new behaviour

JAVA Declaration

- To declare a subclass of `Vehicle`,

```
public class Car extends Vehicle
```

```
class PriorityQ extends FIFO {
    public void add() { .. }
}
```

`super`class Constructor

- Use `super` to call superclass constructor

```
public class RaceCar extends Car() {
   public RaceCar() {
      super();
      // rest of Racecar constructor
      nitrousCapacity = 100;
   }
}
```

Do not need to explicitly call here

  - Any call to the superclass constructor must be the first line of the subclass constructor
  - then, initialize any subclass instance variables
- Java inserts default super() call if no explicit super constructor is called

Email: col106quiz@cse.iitd.ac.in


```
class Aclass {
    void a()
    {
        System.out.print("aclass:a,");
    }

    void b()
    {
        System.out.print("aclass:b,");
        a();
    }
}

class Bclass extends Aclass {
    void a()
    {
        System.out.print("bclass,");
    }
}
```

What prints? Why?

```
public class atest {
   public static void main(String argv[]) {
      Aclass avar = new Bclass();
      avar.a();
      avar.b();
   }
}
```

Overriding Methods

- Subclasses can override, i.e., redefine inherited methods:

```
public class RaceCar extends Car {
  // declarations and constructor elided
  public void start() {
    // new ode to start engine with the
  }
}
```

- Definition of `start()` replaces the one in Car car
  - If this method has identical signature (name and parameter list)
  - Otherwise, its a new method: polymorphism

Method Resolution

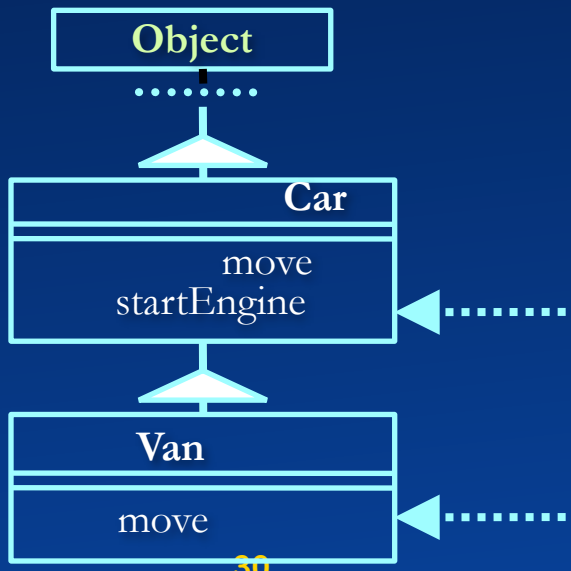
- We can use `Van` in the same way we would use `Car`
  - because it "is a" `Car`
  - `Car mysteryCar = new Van();`
- How does Java determine which method to call when `move()` message is sent to an instance?
  - `mysteryCar.move();`
  - Depends on what actual object `mysteryCar` refers to: `Van` in this case

Method Resolution Rule

- If the instance's class defines the method
  - Call it
- If not, "walk up the class inheritance tree" from class to its superclass until it either
  - finds the method: calls that inherited method
  - doesn't find the method: issues error

`startEngine()` definition found. Call it

No definition of `startEngine()` found in `Van`, looking to `Car`



Email: col106quiz@cse.iitd.ac.in


```
class Aclass {
    void a()
    {
        System.out.println("aclass:a,");
    }

    void b()
    {
        System.out.print("aclass:b,");
        a();
    }
}

class Bclass extends Aclass {
    void a()
    {
        System.out.print("bclass,");
    }
}
```

What prints? Why?

```
public class atest {
   public static void main(String argv[]) {
      Aclass avar = new Bclass();
      avar.a();
      avar.b();
   }
}
```

Polymorphism

Email: col106quiz@cse.iitd.ac.in

```
class Aclass {
    int x = 10;
    void a()
    {
        System.out.printf("aclass:a %d\n", x);
    }

    void b()
    {
        System.out.printf("aclass:b %d\n", x);
        a();
    }
}

class Bclass extends Aclass {
    int x = 33;
    void a()
    {
        System.out.printf("bclass %d\n", x);
    }
}
```

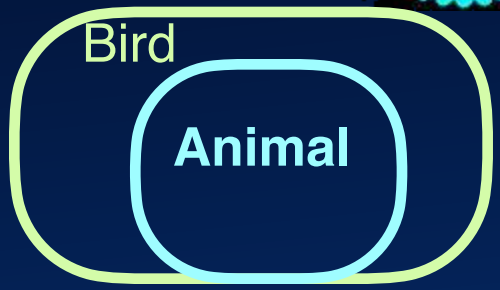
```
public class atest {
    public static void main(String argv[]) {
        Aclass avar = new Bclass();
        avar.a();
        avar.b();
    }
}
```

Email: col106quiz@cse.iitd.ac.in

```
class Aclass {
    private int x = 10;
    void a()
    {
        System.out.printf("aclass:a %d\n", x);
    }

    void b()
    {
        System.out.printf("aclass:b %d\n", x);
        a();
    }
}

class Bclass extends Aclass {
    float x = 33;
    void a()
    {
        System.out.printf("bclass %f\n", x);
    }
}
```

```
public class atest {
    public static void main(String argv[]) {
        Aclass avar = new Bclass();
        avar.a();
        avar.b();
    }
}
```

Another Example



Class `Bird` **extends** `Animal {...}`

```
class Cat {
   public Bird catchBird(Bird birdToCatch) {
      ...
   }
    … Later in another method() …
   {
    Animal pigeon = new Bird();
    catchBird(pigeon);
   }
}
```

Type mismatch: compiler

Is there a problem?

Abstract ➡ Concrete

- Superclass delegates behaviour
  - e.g., all Vehicle instances must be able to move
    - But all Vehicle may move in its own way
- Declaration sp
  - name, parar
  - vehicle.mov clared
- Cannot instantiate an abstract class
  - A class containing an abstract method must itself be declared abstract

```
public class Sportscar extends Vehicle
{
  // provide behavior here
  public void start() { vroom(); }
}
```

```
Vehicle myCar = new Sportscar();
```

```
public abstract class Vehicle{
  // should provide start behavior
  abstract public void start();
}
```

```
// Now this will not compile.
Vehicle myCar = new Vehicle();
```

Quiz

- Polymorphism means that methods return different values for different values of its parameters

No

- Polymorphism allows different objects of a class to respond to the same message (methods with the same signature) differently.

Yes (subclass)

- If a class redeclares a name that is already declared in the parent as private, it is a new name.

Yes

In class Animal:

```
public void eat(Food food) {
    food.cook();
    food = new Food();
    food.digest();
}
```

Somewhere else in the program:

```
myAnimal.eat(apple);// apple is of type Fruit
                    // Fruit is a subclass of Food
```

What happens to object `apple`?

1) Nothing; only a copy of **`apple`** gets passed as a parameter, original is safe
2) It gets cooked, but not digested
3) It gets cooked and digested
4) It gets digested, but not cooked
5) After **`eat`**, **`apple`** refers to the same instance of the **`Fruit`** class as it did before we called **`eat()`**
6) After **`eat`** returns, **`apple`** refers to a different instance of the **`Fruit`** class

Cooked. Some other "Food" object is digested.

apple in caller still refers to the same object

```
class C1 {
    C1()          { System.out.println("C1"); }
    C1(int n)     { System.out.println("C1 : " + n); }
    void print() { System.out.println("C1 print"); }
}
class C2 extends C1 {
    C2()          { System.out.println("C2"); }
    C2(int n)     { this(); System.out.println("C2 : " + n); }
    void print() { System.out.println("C2 print"); }
}
class MainClass  {
    public static void main(String[] args) {
        C1[] c1 = new C1[4];
        c1[0] = new C2();
        c1[1] = new C2(5);
        c1[2] = new C1();
        c1[3] = new C1(4);
        for (int i=0; i<4; i++) {
            c1[i].print();
        }
    }
}
```

DYI. Run it to verify.

What prints?

Simple Loop

```
for(int index=
  process(A[inc
}
```

```
int index;
initialize(index);
while(index < count) {
  process(A[index]);
}
```

```
obj.start();
while obj.hasmore() {
  process(obj.next());
}
```

Iterator interface

```
// Are there more elements left
public boolean hasNext();

// Get the next element (in iteration order)
public Object next();
// Throws NoSuchElementException\
                                    (if there aren't any left)
```

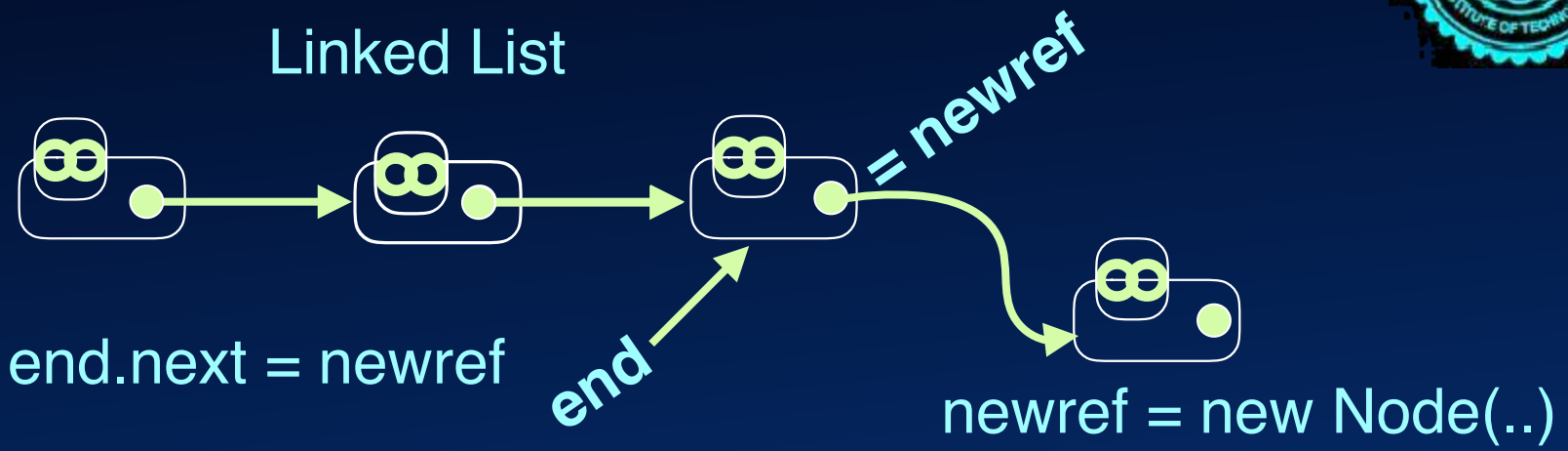
Also see: ListIterator

```
Iterator it = myList.iterator();

while (it.hasNext()) { // there are more
      int v = (Integer) it.next();
      process(v);
}
```

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
  default void remove();
  default void
  forEachRemaining(Consumer<? super E> action)
}
```

```
class Someclass {
    private int[] A;
    Iterator iterator() { return new Itr(); }

    class Itr implements Iterator<Integer> {
        int index = 0;
        public boolean hasNext() {
            r
        }
        publi
            r
        }
    }
}
```

```
Someclass s = new Someclass();
  ...
Iterator sitr = s.iterator();
while (sitr.hasNext()) {
    int v = (Integer) sitr.next();
      ...
}
```



Node

```
class Node {
    private Data data;
    Node next;
}
```