Lecture 2.1

Inheritance

References:

- J. Hershberger, “Java Programming: CS491”, 2004
- Angela Chang, http://www.cs.auckland.ac.nz, 2007

Inheritance

- Inheritance is another representation of an OO relationship between two classes. In an inheritance relationship, an object (called a subclass) obtains data and behavior from another object (called the base class or super class)
- Inheritance is implemented in java using the *extends* keyword
- *super* keyword is used to refer base class

Derived class in inheritance

- Derived class (subclass) in an inheritance relationship:
  - Inherits all the public variables and methods of a base class
  - Adds additional variables and methods
  - Can change the meaning of inherited methods (override methods)

Use of inheritance

- Allows for **code reusability**. This is accomplished by allowing the subclasses (or derived classes) to directly use methods that have been implemented by their base class.
- Java does not support multiple inheritance
- In UML diagram inheritance is shown as a clear triangle.

Inheritance in Java

If class A inherits from class B, it can be depicted with java program as:

```
class A extends B{

}
```

Example

```
public abstract class Animal // class is abstract {
    private String name;
    public Animal(String nm) { name=nm; }
    public String getName() { // regular method
        return (name); }
}

public class Dog extends Animal {
    public Dog(String nm) { // builds ala parent
        super(nm);}
    public void work() // this method specific to Dog {
        System.out.println("I can herd sheep and cows"); }
}
```

Multiple Inheritance

- Some programming languages like C++ allow multiple inheritance where a derived class inherits from more then one base class in the same level of inheritance hierarchy.

Access level

- The access level determines to what extent an object will allow another object to access its members (sometimes this is called object visibility).
- Access levels are used to:
  - enforce encapsulation of an object's data by hiding the attributes
  - restrict access to only certain methods
  - restrict access depending upon what packages the object resides in

Types of Access level

- private ('-' notation in UML)
  - Only the class itself has access to its private members
  - Think: "only you know the secret"
- package ('~' notation in UML)
  - Only the class itself, and any other classes in the same package have access to the class's members. This is the default access level if no access level is specified
  - Think: "everyone in the same group knows the secret"
- protected ('#' notation in UML)
  - Only the class itself, any other classes in the same package , and any subclasses have access to the class's members
  - Think: "you know, everyone in the same group knows, and your children know the secret"
  - Note: Protected access level is only concerned with subclasses.
- public ('+' notation in UML)
  - All other classes have access to the class's members
  - Think: "everyone knows the secret"

Method Overloading

- Method overloading is the process of using the same method name for multiple methods
- The signature of each overloaded method must be unique
  - The signature includes the number, type, and order of the parameters
  - The return type of the method is NOT part of the signature
  - The compiler must be able to determine which version of the method is being invoked by analysing the parameters

Example

```
class Circle {

//declaring the instance variable
protected double radius;
                public Circle(double radius) {
                this.radius = radius;
}

// This method can be overidden
public double getArea() {
                return Math.PI*radius*radius;
}//this method returns the area of the circle

} //end of Circle class
```

```
class Cylinder extends Circle {

        //declaring the instance variable
         protected double length;

         public Cylinder(double radius, double length) {
                super(radius);
                this.length = length;
         }

         // This method has been overidden
         public double getArea() { // method overriden here
                return 2*super.getArea()+2*Math.PI*radius*length;
         }//this method returns the cylinder surface area

}// end of class Cylinder
```

Method Overriding

- A derived class can use the methods of its base class(es), or it can override them
- The method in the derived class must have exactly the **same signature** as the base class method to override.
- The signature is number and type of arguments and the constantness (const, non- const) of the method.
- The **return type** must match the base class method to override.

Using Overriding

- If an inherited property or method needs to behave differently in the derived class it can be overridden; that is, you can define a new implementation of the method in the derived class.
- You can change the meaning (override) of the method declared in the superclass
  - Completely, or
  - Add more functionality to the method
    - The new method can call the original method in the parent class by specifying Super before the method name.
- Rules:
  - A Subclass cannot override final methods declared in the base class.
  - The Overridden method must have the same arguments as the inherited method from the base class.

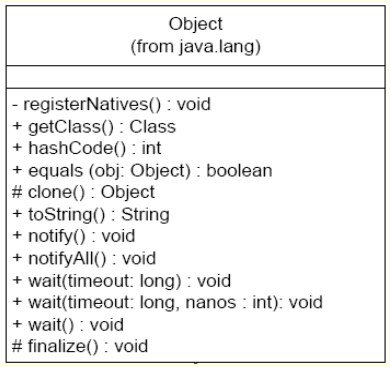
Example

```
public void aMethod( String y){
    //…
}

public void aMethod( int x){ //overloading
    //…
}

public void aMethod( int x, String y ){ //overloading
    //…
}

public int aMethod(float y){ //Not overloading
    //…
}
```

Inheriting from Object class

- In Java, all classes (except primitive data types) inherit from a base class. Each class can have exactly one base class.
- If a class does not explicitly inherit from another class, that class will implicitly inherit from the default base class (which is the *Object* class).
- Class *Object* is the root (i.e. top) of the Java class hierarchy. Every class has *Object* as a superclass (if no other superclass has been given).

UML diagram of Object class

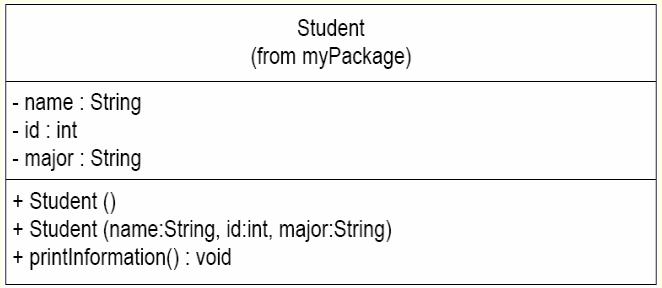


Inheritance with Object class

- The Object class members are not explicitly listed when listing all the members of a Java class. However, if a Java class includes a method which has the same signature of the Object class, you must realize that you are overriding that Object class's member.

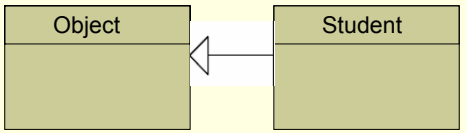
Example

- Student class is declared with some members.



Example cont..

- *Student* class has not explicitly defined any base class but *Object* class will automatically become base class for student.



Example cont..

- if you have the following lines of code:

```
Student S1 = new Student("s1",1,"NOMAJOR");
System.out.println(S1.toString());
// System.out.println(S1);
```

toString() method is a member of Object class which returns a string representation of the object. It can be overloaded to print information about Student.

```
Public String toString(){
    return name + String.valueOf(id) + major;
}
```

Usage of super

- Constructor : super() or super(…)
  - Automatically called in derived constructor if not explicitly called
  - Cannot call super.super()
- super.member
  - Members can be either method or instance variables
  - Refers to the members of the superclass of the subclass in which it is used
  - Used from anywhere within a method of the subclass

Usage of this

- Can be used inside any method to refer to the current object
- Constructor: this(), this(…): refer to its constructor
- this.member
  - Members can be either method or instance variables
  - this.instance_variable:
  - To resolve name-space collisions that might occur between instance variables and local variables

Lecture 2.1

- Dr. Anurag Sharma

Algorithms: Efficiency & Analysis

Algorithms

- An algorithm is a step-by-step procedure used to solve a problem. But making sure the developer is using the most efficient algorithm is very crucial no matter how fast computers become or how cheap memory gets.
- In order to determine how efficiently an algorithm solves a problem, we need to analyze the algorithm.
- Order helps group algorithms according to their eventual behavior.

Algorithms (cont.)

- A computer program is composed of individual modules, understandable by a computer, that solve specific tasks (such as sorting).
- The concentration here is not on the design of entire programs, but rather on the design of the individual modules that accomplish the specific tasks.
- These specific tasks are called **problems**. A problem is a question to which we seek an answer.

Algorithms (cont.)

- A problem may contain **variables** that are not assigned specific values in the statement of the problem. These variables are called parameters to the problem.
- Because a problem contains **parameters**, it represents a class of problems, one for each assignment of values to the parameters. Each specific assignment of values to the parameters is called an **instance** of the problem.

Algorithms (cont.)

- To produce a computer program that can solve all instances of a problem, we must specify a general step-by-step procedure for producing the solution to each instance. This step-by-step procedure is called an **algorithm**. We say that the algorithm solves the problem.

The Importance of Developing Efficient Algorithms

- Efficiency is an important consideration when working with algorithms. A solution is said to be efficient if it solves the problem within the required resource constraints.
- Since one problem can be solved by many different algorithms, it may be important to determine which one is the most efficient. Eg. Searching with sequential search and binary search.

Sequential Search vs Binary Search

- The sequential search algorithm begins at the first position in the array and looks at each value in turn until the item is found.
- The binary search algorithm first compares x with the middle item of the array. If they are equal, the algorithm is done. If x is smaller than the middle item, then x must be in the first half of the array (if it is present at all),and the algorithm repeats the searching procedure on the first half of the array

Binary Search (cont.)

- (That is, x is compared with the middle item of the first half of the array. If they are equal, the algorithm is done, etc.) If x is larger than the middle item of the array, the search is repeated on the second half of the array. This procedure is repeated until x is found or it is determined that x is not in the array.

Binary Search (cont.)

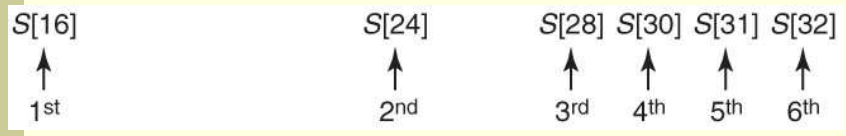


Figure 1.1: The array items that Binary Search compares with x when x is large than all the items in an array of size 32. The items are numbered according to the order in which they are compared.

Sequential Search

- Write the algorithm of sequential search for array of integers.
- Solution will be discussed in the lecture

Sequential Search in Java

```
static int sequentialSearch(final Comparable key,
final ArrayList<? extends Comparable> in){
    int loc = 0;
    while(loc<in.size() && in.get(loc).compareTo(key) != 0){
        loc++;
    }
    if(loc>=in.size())
        loc = -1;
    return loc;
}
```


Binary Search

- Write the algorithm of binary search for array of integers.
- Solution will be discussed in the lecture

Binary Search in Java

```
static int binarySearch(final Comparable key,
final ArrayList<? extends Comparable> in){
    int low, mid, high; //indices
    int loc = -1;

    low = 0; high = in.size()-1;

    while(low<=high && loc == -1){
        mid = (int)Math.floor((low+high)/2.0);
        if(in.get(mid).compareTo(key) == 0)
            loc = mid;
        else if(in.get(mid).compareTo(key) > 0)
            high = mid-1;
        else
            low = mid+1;
    }

    return loc;
}
```


Efficiency comparison

- To compare Sequential Search to Binary Search, Sequential Search does *n* comparisons to determine that *x* is not in the array of size *n*. If *x* is in the array, the number of comparisons is no greater than *n*.
- A Binary Search is an algorithm for locating the position of an element in a sorted list. The method reduces the number of elements that need to be examined by two each time.
- Binary Search appears to be much more efficient than Sequential Search

Efficiency comparison (cont.)

- Compare the number of comparisons for the worst case:

Array Size	Sequential Search	Binary Search
32	32	6
64	64	7
128	128	8
n	*n*	$\log_2 n + 1$
1,024	1,024	11
1,048,576	1,048,576	21

- Therefore Binary Search appears to be much more efficient than Sequential Search, based on the number of comparisons done.

Recursive vs iterative Fibonacci

- The Fibonacci sequence is given as:
- *Fib(n) = Fib(n-1) + Fib(n-2)* for *n* >= 2; *Fib*(0) = 0, *Fib*(1) = 1 Example: 0, 1, 1, 2, 3, 5, 8, …
- Recursive Algorithm (*n*th Fibonacci term):

```
int fib(int n)
{
if (n <= 1) return n;
else
return fib(n-1) + fib(n-2);
}
```

Iterative (n^{th} Fibonacci term)

```
int fib2(int n)
{
  index i;
  int f[0..n]; //array f[0] = 0;
  if (n > 0) {
    f[1] = 1;

 for (i = 2; i <= n; i++)
 f[i] = f[i-1] + f[i-2];
   }
   return f[n];
}
```


Recursive Fibonacci compuations

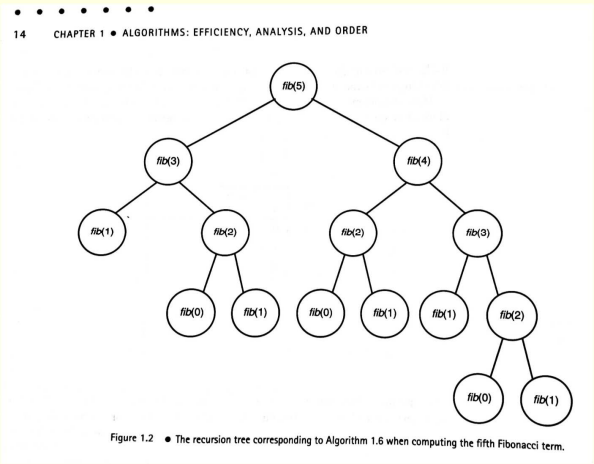


Figure 1.2 • The recursion tree corresponding to Algorithm 1.6 when computing the fifth Fibonacci term.

Efficiency comparison

- Recursive vs iterative Fibonacci.

n	Recursive	Iterative
0	1	1
1	1	2
2	3	3
3	5	4
4	9	5
n	$> 2^{n/2}$	$n + 1$
40	> 1,048,576	41

Lecture 2.2

- Dr. Anurag Sharma

Complexity Analysis of Algorithms

Analysis of Algorithms

- How can we say that an algorithm performs better than another?
- The critical resource for a program is most often its running time and space required to run the program.
  - Time efficiency
  - Space efficiency
- Time is not merely CPU cycles - we want to study algorithms *independent* of *implementations*, *platforms* and *hardware*

Cont.

- When analyzing the efficiency of an algorithm in terms of time, we do not determine the actual number of CPU cycles because this depends on the computer on which the algorithm is run.
- We don't count every instruction executed because the number of instructions depends on the programming language used. We want a measure that is independent of the computer, the programming language, the programmer, and all the complex details of the algorithm.

Cont.

- In general, the running time of the algorithm increases with the size of the input. The total running time is proportional to how many times some basic operation is done. Therefore, we analyze an algorithm's efficiency be determining the number of times some basic operation is done as a function of the size of the input. The size of the input is called the **input size**.
- In general, a time complexity analysis of an algorithm is the determination of how many times the basic operation is done for each value of the input size.

Time Complexity

- A **time complexity analysis** of an algorithm is the determination of how many times the basic operation is done for each value of the input size.

Every-Case Time Complexity

- **every-case time complexity** T(*n*) is defined as the number of times the algorithm does the basic operation for an instance of size *n*
- **Example 1: Adding Array Members**

```
int sum(int n, int array[])
{
   int result = 0;
   for (int i = 0; i < n; i ++)
        result = result + array[i]; // basic operation
    return result;
}
```

  T(*n*) = *n*

Example - 2

Problem: Sort n keys in non-decreasing order

Inputs: Positive integer n, array of keys S indexed from 1 to n
Outputs: Sorted array S

```
void exchangesort (int n, keytype S[])
{
index i, j;
for (i=0; i<n; i++)
  for (j=i+1; j<n; j++)
    if (S[j] < S[i])
      exchange S[i] and S[j];
}
```


T(*n*) for exchange sort

- $T(n) = (n-1) + (n-2) + (n-3) + \ldots + 1$
- $= (n-1)n/2$

Example - 3

- **Problem**: Determine the product of two *n x n* matrices
- **Inputs**: a positive integer n, 2-d arrays of numbers *A* and *B*,each of which has both its rows and columns indexed from *1* to *n*.
- **Outputs**: a 2-d array of numbers *C*, which has both its rows and columns indexed from *1* to *n*, containing the product of *A* and B.
- For c++ and java index starts from 0.

```
void matrixmult (int n, const number A[][],
                 const number B[][], number C[][])
{
   index i, j, k;
   for (i=1; i<=n; i++)
      for (j=1; j<=n; j++) {
         C[i][j] = 0;
         for (k=1; k<=n; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
      }
}
```


T(n) for matrix multiplication

- $T(n) = n * n * n$ //n operations for each loop
- $= n^3$

Worst-Case Time Complexity

- Worst-Case Time Complexity $W(n)$ is defined as the maximum number of times the algorithm will ever do its basic operation for an input size of n. The determination of $W(n)$ is called a worst-case time complexity analysis.
- If T(n) exists, then $W(n) = T(n)$
- Otherwise just take the worst case where maximum computation is required.

Example

- Problem: sequential search

```
static int sequentialSearch(final Comparable key,
final ArrayList<? extends Comparable> in){
    int loc = 0;
    while(loc<in.size() && in.get(loc).compareTo(key) != 0){
        loc++;
    }
    if(loc>=in.size())
        loc = -1;
    return loc;
}
```

W(n) for sequential search

- Worst case?
- When the key to be searched is the last element of the array or
- The key does not exists
- $W(n) = n$

Best-Case Time Complexity

- Best time complexity $\boldsymbol{B(n)}$ is defined as the minimum number of times the algorithm will ever do its basic operation for an input size of n. The determination of $B(n)$ is called a best-case time complexity analysis.
- If T(n) (algorithm does not finish until all n element are considered) exists, then $B(n) = T(n)$

B(n) for sequential search

- Best case?
- When the key to be searched is the first element of the array, is the best case.
- $W(n) = 1$

Average-Case Time Complexity

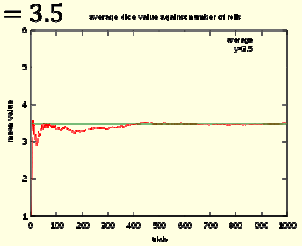
- $\boldsymbol{A(n)}$ is called the average-case time complexity of the algorithm, and the determination of $A(n)$ is called an average-case time complexity analysis.
- If T(n) exists, then $A(n) = T(n)$
- To compute $A(n)$ assign probability to all the input elements of size n.

How to compute A(n)?

- The **expected value** of a discrete random variable X, symbolized as $E(X)$, is often referred to as the *long-term average or* ***mean***. This means that over the long term of doing an experiment over and over, you would expect this average.

$$E(X) = \sum_{i=1}^{n} X_i . P(X_i)$$

- For example, the expected value in rolling a six-sided die is 3.5.
- E(rolling a die) = $1.\frac{1}{6} + 2.\frac{1}{6} + \cdots + 6.\frac{1}{6} = 3.5$
- If n grows, the average will almost surely converge to the expected value, a fact known as the strong law of large numbers.



A(n) for sequential search

- $E(X) = \sum_{i=1}^{N} i \times \frac{1}{N} = \frac{1}{N}\frac{N(N+1)}{2} = \frac{N+1}{2}$

Here we have assumed that an element is in the array. What if in some cases element is not in the list?

Overall A(n) for sequential search

- Element k is either in slot 1 or in slot 2 or in slot i or nowhere in the array.

Prob (slot 1) = 1/n
P(slot 2) = 1/n
P(slot i) = 1/n
P(exists) = p
P(does not exist) = 1-p
E(n) = 1*p/n + 2*p/n + … n*p/n + n*(1-p)
= n(1-p/2)+p/2