

Fall 2017

Fang Yu

Software Security Lab.
Dept. Management Information
Systems,
National Chengchi University

Data Structures

Lecture 2

Object-oriented Design

Abstraction, Modularity, and Encapsulation



Abstraction

- Distills a system to its functionality



[source:onanimation.com]

Abstract Data Type



ADT (Abstract Data Type)

- Define ADT before implementation
- Specifies what each operation does, but not how it does (an interface in Java)
 - An interface is a list of method declarations without their method body
- An ADT is realized by a concrete data structure (which is called “a class” in Java)

Modularity

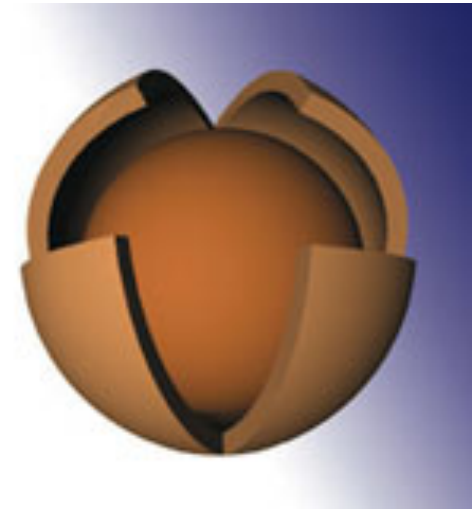
- Splits a large program into a number of smaller, **independent** parts to reduce complexity
- Each part (a module) represents a separate functional unit



[Source: php.jglobal.com]

Encapsulation

- Hide the implementation details of a module from its users
- Each module maintains a consistent interface but reveals **no** internal details for outsiders
- Gives the programmers the freedom in implementing the details



[Source: entertainingcode.com]

Goals

Software implementation should be

- Robustness
- Adaptability
- Reusability



Object-oriented Design



An object

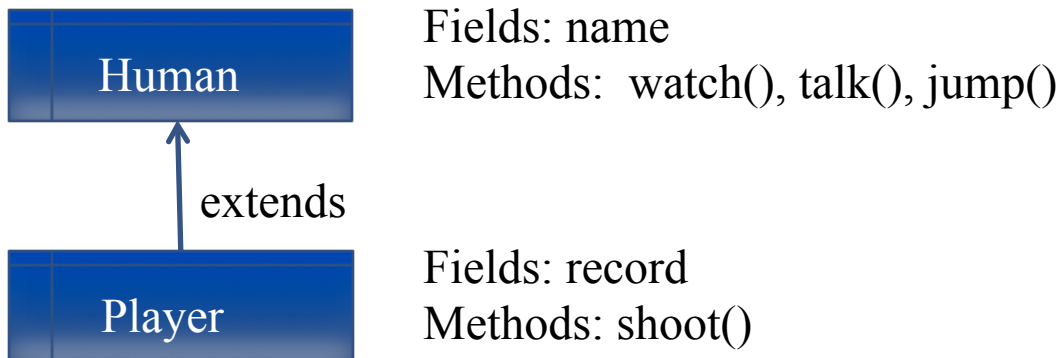
- Is the main actor in the object-oriented paradigm
- is an instance of a class

A Class

- Defines an object
- Consists of fields and methods
- Gives others a consistent and concise view to interact with the object (without knowing details)

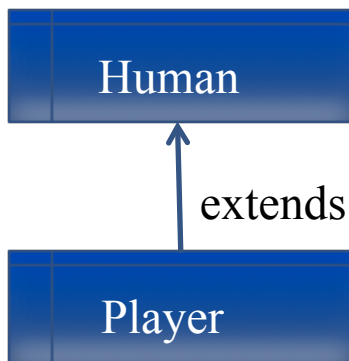
Inheritance

- A way to reuse code (based on a hierarchical structure)
- Player (A subclass) extends Human (A superclass)
- A player has his/her own name and record, and can watch, talk, jump, and shoot.



Overriding

- Redefine a method in the subclass
- A player jumps in a different way
- A player has his/her own name and record, and can watch, talk, **jump**, and shoot.



Fields: name
Methods: watch(), talk(), jump()

Fields: record
Methods: **jump**(), shoot()

Polymorphism

- An object can be polymorphic.
- It may have different forms and behave the same method in different ways depending on which class it refers to

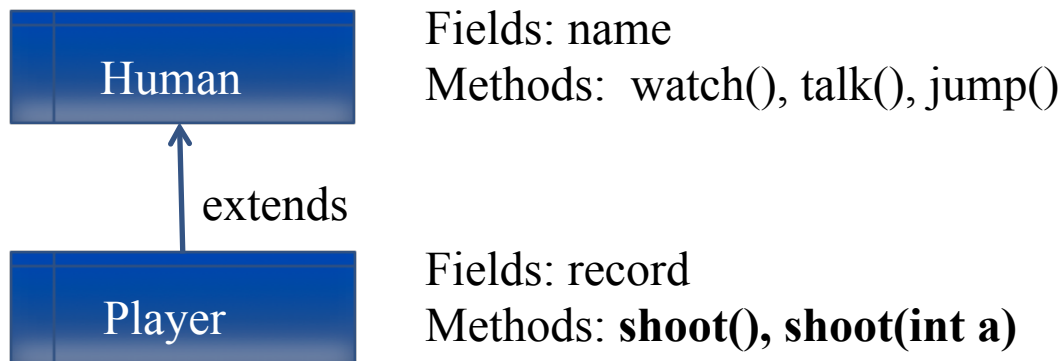
For example,

- An instance of Human (like us) jumps in one way
- An instance of Player (like LeBron James) jumps in a different way
- How does Fang jump?



Overloading

- In the same class, one can define the same method with different signatures
- The signature of a method is a combination of its name, and the type and number of arguments



this

- A keyword in Java
- The reference of the **current instance** of the class

```
public class Example {
    public int dog = 2;
    public void clobber() {
        int dog = 5;
        System.out.println("The local dog value = "+dog);
        System.out.println("The field dog value = "+this.dog);
    }
    public static void main(String[] argv) {
        this.clobber();
    }
}
```

```
javac Example.java
java Example
```

```
The local dog value = 5
The field dog value = 2
```

An inheritance example



- Progression
 - 1, 2, 3, ...
- Arithmetic Progression
 - $f(n) = f(n-1) + d$
 - $f(0) = 1, d = 2$, we have 1, 3, 5, 7, ...
- Geometric Progression
 - $f(n) = f(n-1) * r$
 - $f(0) = 1, r = 2$, we have 1, 2, 4, 8, 16, ...
- Fibonacci Progression
 - $f(n) = f(n-1) + f(n-2)$
 - $f(0) = 1, f(1) = 2$, we have 1, 2, 3, 5, 8, 13, 21, ...

Progression



- Fields:
 - first (the first value)
 - cur (the current value)
- Methods:
 - Progression(): Initialize the field values (A Constructor function)
 - firstValue(): Reset the progression to the first value and return that value
 - nextValue(): Step the progression to the next value and return that value
 - printProgression(int n): Reset the progression and print the first n values

Progression

//1, 2, 3, ...

```
public class Progression {
    protected long first;
    protected long cur;
    Progression(){ //Constructor
        first = cur = 1;
    }
    protected long firstValue(){ //Reset cur
        cur = first;
        return cur;
    }
    protected long nextValue(){ //cur = cur+1; return cur;
        return ++cur;
    }
    protected long printProgression(int n){ ... }
}
```



Progression

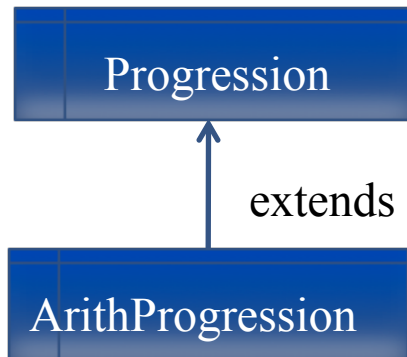


```
//Print the first n values
protected long printProgression(int n){
    System.out.print(firstValue());
    for(int i =2; i<=n; i++)
        System.out.print(" "+ nextValue());
    System.out.println();
}
```

Arithmetic Progression



first, first+d, first+2d, ...



Fields: first, cur
Methods: firstValue(), nextValue(),
printProgression()

Fields: d
Methods: nextValue()

Arithmetic Progression

//refines constructor, replaces nextValue(), and
//inherits Progression(), firstValue(), printProgression(int)

```
class ArithProgression extends Progression{
    protected long d;
    ArithProgression(){ //d =1 by default
        this(1,1); //first=cur=1, d = 1;
    }
    ArithProgression(int a, int increment) { //Set d to increment
        first =cur= a;
        d = increment;
    }
    protected long nextValue(){
        cur += d; //cur = cur+d;
        return cur;
    }
}
```

Geometric Progression



first, first*r, first*r², ...



Fields: first, cur

Methods: firstValue(), nextValue(),
printProgression()

extends



Fields: r

Methods: nextValue()

Geometric Progression

//refines constructor, replaces nextValue(), and
//inherits Progression(), firstValue(), printProgression(int)

```
class GeomProgression extends Progression
{
    protected long r;
    GeomProgression(){ //first =1, r =1 by default
        this(1,1); //first = 1; r = 1;
    }
    GeomProgression(int a, int base) { //Set r to base
        first = a;
        r = base;
    }
    protected long nextValue(){
        cur *= r;    //cur = cur*r;
        return cur;
    }
}
```



Fibonacci Progression



$a_0, a_1, a_2, a_3, \dots (a_{n+1} = a_{n-1} + a_n)$



extends



Fields: first, cur

Methods: firstValue(), nextValue(),
printProgression()

Fields: prev

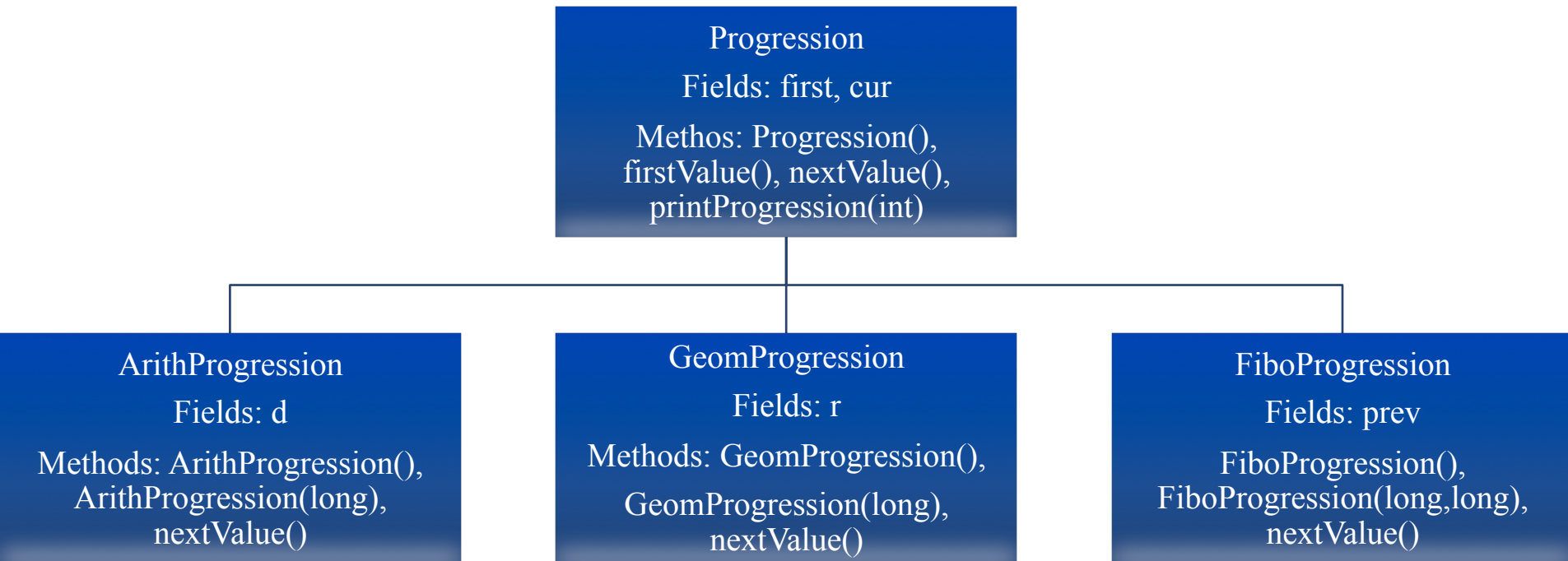
Methods: nextValue()

Fibonacci Progression

```
//inherits Progression(), firstValue(), printProgression(int)
class Fiboprogression extends Progression{
    protected long prev;
    Fiboprogression(){ //a0=1, a1=2 by default
        this(1,2);
    }
    Fiboprogression(long a0, long a1) {
        first = a0; //overwrite the initial value of first (so is cur)
        prev = a1-a0; //fictitious value preceding the first
    }
    protected long nextValue(){ //an+1 = an-1+an
        long temp = prev; //temp = an-1
        prev = cur; //prev = an
        cur+=temp; //cur = an+1 = an-1+an
        return cur;
    }
}
```



Inheritance Diagram



Test Progression

```
class TestProgression{
    public static void main(String[] args){
        Progression prog;
        prog = new ArithProgression(1, 2);
        prog.printProgression(10);
        prog = new GeomProgression(1,3);
        prog.printProgression(10);
        prog = new FibonacciProgression(3,4);
        prog.printProgression(10);
    }
}
```

```
1 3 5 7 9 11 13 15 17 19
1 3 9 27 81 243 729 2187 6561 19683
3 4 7 11 18 29 47 76 123 199
```

DoubleProgression

//1, 2, 3, ...

```
public class DoubleProgression {
    protected double first;
    protected double cur;
    Progression(){ //Constructor
        first = cur = 1;
    }
    protected double firstValue(){ //Reset cur
        cur = first;
        return cur;
    }
    protected double nextValue(){ //cur = cur+1; return cur;
        return ++cur;
    }
    protected double printProgression(int n){ ... }
}
```



DoubleProgression



```
//Print the first n values
protected void printProgression(int n){
    System.out.print(firstValue());
    for(int i =2; i<=n; i++)
        System.out.print(" "+ nextValue());
    System.out.println();
}
```

Geometric Double Progression

//refines constructor, replaces nextValue(), and
//inherits Progression(), firstValue(), printProgression(int)

```
class GeomDoubleProgression extends DoubleProgression
{
    protected double r;
    GeomProgression(){ //first =1, r =1 by default
        this(1,1); //first = 1; r = 1;
    }
    GeomProgression(double a, double base) { //Set r to base
        first = a;
        r = base;
    }
    protected double nextValue(){
        cur *= r;    //cur = cur*r;
        if(cur<0) throw new Exception("negative value!");
        return cur;
    }
}
```

Test Progression

```
class TestProgression{
    public static void main(String[] args){
        Progression prog;
        prog = new ArithProgression(1, 2);
        prog.printProgression(10);
        prog = new GeomProgression(1,3);
        try{
            prog.printProgression(10);
        }catch(Exception abc){
            System.out.println(abc.getValue());
        }
        prog = new GeomDoubleProgression(100,0.024);
        prog.printProgression(10);
        prog = new FibonacciProgression(3,4);
        prog.printProgression(10);
    }
}
```

```
1 3 5 7 9 11 13 15 17 19
1 3 9 27 81 243 729 2187 6561 19683
3 4 7 11 18 29 47 76 123 199
```

Exception Handling



Exceptions are

- unexpected events that occur during the execution of a program (by JRE or programmers)
- Throw an exception in Java

throw new exception_type (param, ...)

- Example:

```
if (insertIndex >= A.length){  
    throw new  
        BoundaryViolationException("No element at index " + insertIndex);  
}
```

Catching Exceptions

- When an exception is thrown, it must be caught
- Otherwise, the program will terminate
- Use try-catch block in Java to catch exceptions

```
int index = Integer.MAX_VALUE;
try{
    String.toBuy = shoppingList[index];
}
catch(ArrayIndexOutOfBoundsException aioobx){
    System.out.println("The index "+index+" is outside the array.");
}
```

Interface



Application Programming Interface (API)

- The methods that each object supports
- A list of method declarations with no data and no bodies

Implementing Interfaces

- An interface enforces the requirements that a class has methods with certain specified signatures
- A class can implement many interfaces (must implement all methods of each interface)

An Interface and Its Implementation



```
public interface Sellable {  
    public String description();  
    public int listPrice();  
    public int lowestPrice();  
}
```

```
public class Photo implements Sellable{  
    private String descript;  
    private int price;  
    private boolean color;  
  
    public Photo(String desc, int p, boolean c){  
        descript = desc; price = p; color = c;  
    }  
    public String description() {  
        return desc;  
    }  
    public int listPrice() {  
        return price;  
    }  
    public int lowestPrice() {  
        return price/2;  
    }  
}
```

Multiple Inheritance

- An interface can have multiple inheritance

(a class cannot)

```
public interface Sellable {  
    public String description();  
    public int listPrice();  
    public int lowestPrice();  
}
```

```
public interface Transportable {  
    public int weight();  
    public int isHazardous();  
}
```

```
public interface InsurableItem extends Transportable, Sellable {  
    public int insuredValue();  
}
```

Generics

- A generic type is not defined at compile time but becomes fully specified at run time
- Define a class with **formal type parameters**
- Instantiate an object of this class by using **actual type parameters** to indicate the concrete types



An Integer Pair Example

```
public class IntPair{
    string key;
    int value;
    public void set(string k, int v){
        key = k;
        value = v;
    }
    public string getKey(){ return key; }
    public int getValue(){ return value;}
    public String toString(){
        return "["+getKey()+"", "+getValue()+" ]";
    }
    public static void main(...){...}
}
```



An Double Pair Example

```
public class DoublePair{
    string key;
    double value;
    public void set(string k, double v){
        key = k;
        value = v;
    }
    public string getKey(){ return key; }
    public double getValue(){ return value;}
    public String toString(){
        return "["+getKey()+" , "+getValue()+" ]";
    }
    public static void main(...){...}
}
```



A Pair Example



```
...
public static void main(String[] args){
    IntPair pair1 = new IntPair();
    pair1.set("age", 20);
    System.out.println(pair1.toString());
    DoublePair pair2 = new DoublePair ();
    pair2.set(new String("grade"), new Double(82.53));
    System.out.println(pair2.toString());
}
...
```

Javac Pair.java
Java Pair

[age, 20]
[grade, 82.53]

A Generic Example

```
public class Pair<K, V>{
    K key;
    V value;
    public void set(K k, V v){
        key = k;
        value = v;
    }
    public K getKey(){ return key; }
    public V getValue(){ return value;}
    public String toString(){
        return "["+getKey()+", "+getValue()+" ]";
    }
    public static void main(...){...}
}
```



A Generic Example



```
...  
public static void main(String[] args){  
    Pair<String, Integer> pair1 = new Pair<String, Integer>();  
    pair1.set(new String("age"), new Integer(20));  
    System.out.println(pair1.toString());  
    Pair<String, Double> pair2 = new Pair<String, Double>();  
    pair2.set(new String("grade"), new Double(82.53));  
    System.out.println(pair2.toString());  
}  
...
```

Javac Pair.java

Java Pair

[age, 20]

[grade, 82.53]

Genetic Progression

```
//1, 2, 3, ...
public class Progression <k> {
    protected k first;
    protected k cur;
    Progression(){ //Constructor
        first = cur;
    }
    protected k firstValue(){ //Reset cur
        cur = first;
        return cur;
    }
    protected k nextValue(){ //cur = cur+1; return cur;
        return cur;
    }
    protected k printProgression(int n){ ... }
}
```



Geometric Progression

//refines constructor, replaces nextValue(), and
//inherits Progression(), firstValue(), printProgression(int)

```
class GeomProgression extends Progression<Long>{
    protected long r;
    GeomProgression(){ //first =1, r =1 by default
        this(1,1); //first = 1; r = 1;
    }
    GeomProgression(long a, long base) { //Set r to base
        first = a;
        r = base;
    }
    protected long nextValue(){
        cur *= r;    //cur = cur*r;
        return cur;
    }
}
```



HW 2 (Due on 9/28)

Generic Geometric Progression

- Output two different types of geometric progressions using inheritance and generics
- Try to throw/catch exceptions



The Growth Population

- Get initial population and years from user inputs (Use Scanner class)
- Every 30 years (a generation), the population becomes double
- Output the Population Progression (by generation)
- E.g., Input: 2 people and 120 years.

Output: 2, 4, 8, 16, 32



The Growth of Capital

- Get initial capital and years from user inputs
- The annual interest rate is 2.4%
- Output the yearly Capital progression
- E.g., Input: 100 and 2 years.

Output: 100, 102.4, 104.8576



Coming up...

- We will discuss the team project, and text processing and pattern matching next week
- Read TB Chapter 12
- **We will have a makeup course on Sep 30**
- **We will not have courses on Oct 5 and Oct. 12**



Team Project: 30%



- 3-5 students as a team
- Send the team list (name and contact) to your TAs before the end of next week
- Develop your application using Eclipse with SVN
 - TAs will help you set up SVN in the next week lab (Sep. 25)
 - Start to work on it
 - You will get extra points for updating your code constantly