**Lecture #13 OOP Part 1**

**Tim Hoffman**

**OOP 1: Writing your own classes**

Object-Oriented Programming (OOP) is the use of objects to combine variables and methods in a single unit of abstraction. More concretely, OOP is the writing and use of classes. A class in Java is simply a definition of a data type where that description weds variables and methods which in turn manipulate those variables. We begin with a simple example.

We want to represent a rational number - a **fraction** in our program. We want to store a numerator and denominator for our fraction and perform operations such as addition, subtraction, multiplication etc on different variables of our Fraction data type.

**How do we represent a Fraction in our program?**

We have several possibilities. The 2 simplest are:

* 1. declare 2 ints named numer and denom, then pass them around to various methods from main
  2. declare an array of 2 ints and just remember that arr[0] is numerator and arr[1] is denominator

The second attempt is actually a  little better than the first. At least it achieves the bundling together of the numerator and denominator into one single object - an array object.  It is still a very poor solution. It would be much better if we could bundle the two values into one container but still call the 2 values by meaningful names like numer and denom rather than arr[0] and arr[1].

**A better solution:     Write a class definition file named "Fraction.java"**

|  |
| --- |
| /\* Fraction.java A class (data type) definition file  This file just defines what a Fraction is  This file is NOT a program  \*/  public class Fraction  {  public int numer;  public int denom;  }// EOF |

As soon as we have written the above Fraction class definition file that describes what a Fraction is, we can then do the following:

|  |
| --- |
| /\*  FractionTester.java A program that declares Fraction variables  \*/  public class FractionTester  {  public static void main( String args[] )  {  // use the word Fraction as if were a Java data type  Fraction f1 = new Fraction(); // create an object of the fraction type  f1.numer = 22; // use dot operator to access numer or denom  f1.denom=7;  System.out.println("f1=" + f1.numer + "/" + f1.denom); // outputs: "f1=22/7"  } // END main  } // EOF |

**You are now a class designer. You just invented your own new data type - the Fraction type (or *class*) and now must learn the rules of good class design and some terminology.**

We start with some terminology to describe what we have already done.

* 1. **class** - a.k.a data type, our new class is named **the Fraction class**
  2. class definition file - the source code file named "Fraction.java" it contains the definition of what a Fraction is
  3. **object** - the chunk of memory allocated by a statement like: **Fraction f1=new Fraction();**. The object is where the actual data is stored in memory
  4. **reference** variable - in our case, **f1** is the reference by which we access the object
  5. **members** - our Fraction class has two members: numer and denom

So.. the  important terms we must understand completely before moving on are: *class, members, object and reference*. Our object is of class (type) Fraction, and we can only access it's members via it's reference variable f1.

**Classes can have methods**

It is often useful to initialize an object at the time of it's creation. In the case of our FractionTester program we would like to be able to do the following:

|  |
| --- |
| /\*  FractionTester.java A program that declares Fraction variables  \*/  public class FractionTester  {  public static void main( String args[] )  {  Fraction f1 = new Fraction( 22, 7 ); // 22/7 gets stored in the object  System.out.println("f1=" + f1.numer + "/" + f1.denom); // outputs: "f1=22/7"  } // END main  } // EOF |

Notice how intuitive and elegant *Fraction f1 = new Fraction( 22, 7 );* is. It is obvious to the reader what the intent of that declaration is. Our Fraction class needs to have a special method added to it's definition file. This method is called a **constructor**. It's job is to receive initial values for the object from the declaration statement and copy those values into their respective member variables.

Our Fraction class definition file after adding our constructor method.

|  |
| --- |
| /\* Fraction.java class (data type) definition file  This file just defines what a Fraction is  This file is NOT a program  \*/  public class Fraction  {  public int numer;  public int denom;  // CONSTRUCTOR method - initializes object  // via values passed in from the object's declaration  public Fraction( int n, int d )  {  numer = n;  denom = d;  }  }// EOF |

Notice this method is peculiar in 2 ways. First it does not have a return type. It is not void, it simply has no return type at all. Secondly, it is named after the class. The class is named Fraction and the constructor method is named Fraction. **Contstructors  can be overloaded**. In other words **you can have multiple constructor methods** - all named Fraction in your Fraction class. They must however differ in their parameter list. Either they must take a different number of args or take different data type args.

Notice how we construct several different Fraction objects in our main program and how each one takes a different number of args. Then look at what we did to our Fraction definition file to accommodate those different forms of declaration in main.

|  |
| --- |
| /\*  FractionTester.java A program that declares Fraction variables  \*/  public class FractionTester  {  public static void main( String args[] )  {  Fraction f1 = new Fraction( 22, 7 ); // 22/7 gets stored in the object  Fraction f2 = new Fraction(); // 0/1 gets stored  Fraction f3 = new Fraction( 5 ); // 5/1 gets stored  System.out.println("f1=" + f1.numer + "/" + f1.denom); // outputs: "f1=22/7"  System.out.println("f2=" + f2.numer + "/" + f2.denom); // outputs: "f2=0/1"  System.out.println("f3=" + f3.numer + "/" + f3.denom); // outputs: "f3=5/1"  } // END main  } // EOF |

And here is the modified Fraction.java file that has those 2 additional constructors written inside.

|  |
| --- |
| /\* Fraction.java class (data type) definition file  This file just defines what a Fraction is  This file is NOT a program  \*/  public class Fraction  {  public int numer;  public int denom;  // DEFAULT CONSTRUCTOR - no args passed in  public Fraction( )  {  numer = 0;  denom = 1; // don't put a ZERO here  }  // 1 arg CONSTRUCTOR - 1 arg passed in  // assume user wants whole number  public Fraction( int n )  {  numer = n;  denom = 1; // don't put a ZERO here  }  // FULL CONSTRUCTOR - an arg for each class data member  public Fraction( int n, int d )  {  numer = n;  denom = d;  }  }// EOF |

**Encapsulation and Data Hiding**

The above Fraction class achieves one of the important advantages of using classes known as **encapsulation**.

Encapsulation means the bundling together or data (variables) and methods. However, our Fraction class falls short  of achieving the second important advantage called **data hiding**.

Data hiding means that we should now allow the users of our class (a.k.a clients of our class) such as the code in main, to directly access the members of the class (numer and denom).

The following code illustrates the vulnerability of having our data members declared public in our class definition.

|  |
| --- |
| /\*  FractionTester.java A program that declares Fraction variables  \*/  public class FractionTester  {  public static void main( String args[] )  {  // use the word Fraction as if were a Java data type  Fraction f1 = new Fraction(); // create an object of the fraction type  f1.numer = 22; // use dot operator to access numer or denom  **f1.denom=0; // BAD! But possible because numer is public !!**  System.out.println("f1=" + f1.numer + "/" + f1.denom); // outputs: "f1=22/7"  } // END main  } // EOF |

**Data Hiding is achieved by declaring class data members private instead of public**

|  |
| --- |
| /\* Fraction.java A class (data type) definition file  This file just defines what a Fraction is  This file is NOT a program  \*\* data members are PRIVATE  \*\* method members are PUBLIC  \*/  public class Fraction  {  **private** int numer;  **private** int denom;  // DEFAULT CONSTRUCTOR - no args passed in  public Fraction( )  {  numer = 0;  denom = 1; // don't put a ZERO here  }  // 1 arg CONSTRUCTOR - 1 arg passed in  // assume user wants whole number  public Fraction( int n )  {  numer = n;  denom = 1; // don't put a ZERO here  }  // FULL CONSTRUCTOR - an arg for each class data member  public Fraction( int n, int d )  {  numer = n;  denom = d;  }  }// EOF |

Now any attempts by the code in main to directly access numer or denom via the dot operator will be caught by the compiler at compile time. Class member data is thus protected from direct access because we declared it as **private** instead of public.

|  |
| --- |
| public class FractionTester  {  public static void main( String args[] )  {  // use the word Fraction as if were a Java data type  Fraction f1 = new Fraction(); // create an object of the fraction type  f1.numer = 22;  f1.denom=0;  } // END main  } // EOF |

Compiling the above FractionTester program thus produces the following expected compilation errors:

|  |
| --- |
| C:\Temp> javac FractionTester.java  C:\Temp\FractionTester.java:8:  numer has private access in Fraction  f1.numer = 22;  ^  C:\Temp\FractionTester.java:10:  denom has private access in Fraction  f1.denom = 0;  ^  2 errors |

**We have now written our class to conform with the good practice of data hiding via private declaration of class data. We now move on to more good design principles.**

**Accessors and Mutators**

Recall above that we just made our class data members private. Users of our class now have a problem that we must solve. They need some way to get to the numer and denom members of the Fraction class. The solution is to write some public methods inside the class definition that allow outside users to access these data members.  These methods are sometimes called **accessors** and **mutators** because they give users of the class the ability to read or modify the member data.

|  |
| --- |
| /\* Fraction.java A class (data type) definition file  This file just defines what a Fraction is  This file is NOT a program  \*\* data members are PRIVATE  \*\* method members are PUBLIC  \*/  public class Fraction  {  private int numer;  private int denom;  // ACCESSORS  public int getNumer()  {  return numer;  }  public int getDenom()  {  return denom;  }  // MUTATORS  public void setNumer( int n )  {  numer = n;  }  public void setDenom( int d )  {  if (d!=0)  denom=d;  else  {  // error msg OR exception OR exit etc.  }  }  // DEFAULT CONSTRUCTOR - no args passed in  public Fraction( )  {  numer = 0;  denom = 1; // don't put a ZERO here  }  // 1 arg CONSTRUCTOR - 1 arg passed in  // assume user wants whole number  public Fraction( int n )  {  numer = n;  denom = 1; // don't put a ZERO here  }  // FULL CONSTRUCTOR - an arg for each class data member  public Fraction( int n, int d )  {  numer = n;  denom = d;  }  }// EOF |

Users of the class can now use these **get** and **set** methods to read and modify the members variables but the integrity of the class is protected because the setDenom() method guards against a ZERO being assigned to the denom member.

|  |
| --- |
| public class FractionTester  {  public static void main( String args[] )  {  // use the word Fraction as if were a Java data type  Fraction f1 = new Fraction(); // create an object of the fraction type  f1.**setNumer**(22);  f1.**setDenom**(7); // if 0 were passed in it would be rejected  System.out.println( "f1= " + f1.**getNumer()** + "/" + f1.**getDenom()** );  } // END main  } // EOF |

**Code re-use and keyword "this"**

Our class still has a big security hole. We just prevented users from putting a zero into the denominator by making our members private and putting a test into the setDenom() method to prevent a zero send in from getting assignmed into denom. Our constructor is still a security hole. A first reaction might be to copy the test code from the setDenom() method into the constructor. This violates the next principle of good class design - **code re-use**.

The code re-use principle states that it is bad programming to rewrite the same code in multiple places - if it's possible to just put it in one place and call it from the other places. Also, in designing a class it is strongly recommended that member variables get modified in only one place - that member's **setXXX()**method. In the case of our full constructor we accomplish this by calling setNumer() and setDenom() instead of direct assignment.

|  |
| --- |
| public class Fraction  {  ...  ...  // FULL CONSTRUCTOR  public Fraction( int n, int d )  {  setNumer( n );  setDenom( d ); // setDenom() catches ZEROs  }  ...  ...  }// EOF |

Our full constructor is now re-using code already written in the set methods of the class. We now have to do the same in our other constructors. We do this by simply having our other constructors call our full constructor. Here is where the keyword **this** comes in.

|  |
| --- |
| // SEVERAL METHODS DELETED FOR BREVITY (CODE WILL NOT COMPILE)  public class Fraction  {  private int numer;  private int denom;  // ACCESSORS...  // MUTATORS ...  // DEFAULT CONSTRUCTOR - no args passed in  public Fraction( )  {  this( 0, 1 ); // **this** means call a fellow constructor  }  // 1 arg CONSTRUCTOR - 1 arg passed in  // assume user wants whole number  public Fraction( int n )  {  this( n, 1 ); // **this** means call a fellow constructor  }  // FULL CONSTRUCTOR - an arg for each class data member  public Fraction( int n, int d )  {  setNumer(n);  setDenom(d);  }  ...  ...  }// EOF |

In the above re-writes of the default and 1 arg constructors, notice we use the keyword **this** to call a fellow constructor of the class. Keyword this is actually a reference to the object of this class type that is executing this code. We will talk more about this later.

**toString() method**

Our simple Fraction lacks one item of convenience that polite class designers put in for the users of the class. The toString() method is automatically invoked by the runtime if a user of the class tries to print a reference variable:

**System.out.println("f1= " + f1 ); // outputs a memory addr something like: "@5341593"**

Without a toString() method present in our Fraction class, the above print statement would output a number. That number would be the address in memory where the Fraction object's data was stored. This makes perfect sense since f1 is a reference variable - which means it simply contains the address of WHERE the data is, NOT the data itself.

|  |
| --- |
| // SEVERAL METHODS DELETED FOR BREVITY (CODE WILL NOT COMPILE)  public class Fraction  {  private int numer;  private int denom;  // ACCESSORS...  // MUTATORS...  // CONSTRUCTORS...  public String toString()  {  return numer + "/" + denom;  }  ...  ...  }// EOF |

In main we can now print out the reference variable and let the runtime automatically call the .toString() of our object for us.

**Fraction f1 = new Fraction(22,7);**

**System.out.println("f1= " + f1 ); // outputs: "22/7"**

**Here is our Fraction class thus far**

|  |
| --- |
| /\* Fraction.java A class (data type) definition file  This file just defines what a Fraction is  This file is NOT a program  \*\* data members are PRIVATE  \*\* method members are PUBLIC  \*/  public class Fraction  {  private int numer;  private int denom;  // ACCESSORS  public int getNumer()  {  return numer;  }  public int getDenom()  {  return denom;  }  // MUTATORS  public void setNumer( int n )  {  numer = n;  }  public void setDenom( int d )  {  if (d!=0)  denom=d;  else  {  // error msg OR exception OR exit etc.  }  }  // DEFAULT CONSTRUCTOR - no args passed in  public Fraction( )  {  this( 0, 1 ); // **this** means call a fellow constructor  }  // 1 arg CONSTRUCTOR - 1 arg passed in  // assume user wants whole number  public Fraction( int n )  {  this( n, 1 ); // **this** means call a fellow constructor  }  // FULL CONSTRUCTOR - an arg for each class data member  public Fraction( int n, int d )  {  setNumer(n);  setDenom(d);  }  }// EOF |

**Passing objects into class methods**

So far, all the values passed into our class methods have been primitives as Strings. We start with passing in other objects of our own type.

**The Copy Constructor**

Let's go back to our FractionTester program and show yet another form of construction, know as **copy construction**.

|  |
| --- |
| public class FractionTester  {  public static void main( String args[] )  {  // use the word Fraction as if were a Java data type  Fraction f1 = new Fraction(22,7); // use full C'Tor  // now create f2 to be exact duplicate of f1 via copy C'Tor  // pass in an already initialied object to copy from  Fraction f2 = new Fraction( f1 ); // use *copy* C'Tor  System.out.println( "f1= " + f1 );  System.out.println( "f2= " + f2 ); // f2 is identical to f1  } // END main  } // EOF |

**The copy constructor looks like this**

|  |
| --- |
| // SEVERAL METHODS DELETED FOR BREVITY (CODE WILL NOT COMPILE)  public class Fraction  {  private int numer;  private int denom;  // ACCESSORS...  // MUTATORS...  // COPY CONSTRUCTOR - takes ref to some already initialied Fraction object  public Fraction( Fraction other )  {  this( other.numer, other.denom ); // call my full C'Tor with other Fraction's data  }  // DEFAULT CONSTRUCTOR - no args passed in  public Fraction( )  {  numer = 0;  denom = 1; // don't put a ZERO here  }  // 1 arg CONSTRUCTOR - 1 arg passed in  // assume user wants whole number  public Fraction( int n )  {  numer = n;  denom = 1; // don't put a ZERO here  }  // FULL CONSTRUCTOR - an arg for each class data member  public Fraction( int n, int d )  {  numer = n;  denom = d;  }  }// EOF |

**Our Fraction class  with full compliment of Constructors**

[**Fraction.java**](http://www.cs.pitt.edu/~hoffmant/401/OOP-1/Fraction/Fraction.java)

|  |
| --- |
| /\* Fraction.java A class (data type) definition file  This file just defines what a Fraction is  This file is NOT a program  \*\* data members are PRIVATE  \*\* method members are PUBLIC  \*/  public class Fraction  {  private int numer;  private int denom;  // ACCESSORS  public int getNumer()  {  return numer;  }  public int getDenom()  {  return denom;  }  public String toString()  {  return numer + "/" + denom;  }  // MUTATORS  public void setNumer( int n )  {  numer = n;  }  public void setDenom( int d )  {  if (d!=0)  denom=d;  else  {  // error msg OR exception OR exit etc.  }  }  // DEFAULT CONSTRUCTOR - no args passed in  public Fraction( )  {  this( 0, 1 ); // **this** means call a fellow constructor  }  // 1 arg CONSTRUCTOR - 1 arg passed in  // assume user wants whole number  public Fraction( int n )  {  this( n, 1 ); // **this** means call a fellow constructor  }  // FULL CONSTRUCTOR - an arg for each class data member  public Fraction( int n, int d )  {  setNumer(n);  setDenom(d);  }  // COPY CONSTRUCTOR - takes ref to some already initialized Fraction object  public Fraction( Fraction other )  {  this( other.numer, other.denom ); // call my full C'Tor with other Fraction's data  }  }// EOF |