CMSC 202H

Classes and Objects:

Reusing Classes with Composition

Code Reuse

- Effective software development relies on reusing existing code.
- Code reuse must be more than just copying code and changing it which is often the case with procedural languages like C.
- The goal with OOP languages is to reuse classes without changing the code within the class - one OOP technique for code reuse is known as ***composition*****.**

A Simple Database

Your favorite cousin wishes to implement a simple database of family relatives. His application has only a few requirements.

He would like record the name, birthday, and date of death of each family relative. A report is required that prints all information for each family member. Family members must be comparable to avoid duplicate entries in the database. For ease of data entry, it must be possible to make a copy of an existing family member.

Your contribution to this project is to design and implement a class named Person that will represent a single family member.

Designing A **`Person`** Class:
Primitive Instance Variables

- To model the kinds of entities we will need for the task at hand, a simple **`Person`** class would contain instance variables representing details about a person's name, the date on which they were born, and the date on which they died.

- As a first pass, we could model this data with instance variables of simple types: primitive types and **`String`**:

```
public class Person
{
  private String name;
  private int born_date, born_month, born_year;
  private Date died_date, died_month, died_year;
        . . .
```

- Also, as a first line of defense for privacy and to provide proper encapsulation, note that we declare each of the instance variables **`private`**.

Designing A **Person** Class: Class Instance Variables

- However, we previously developed a class called **Date** that would be perfect for storing birth and death dates—could we reuse that? Yes, we can, just by including instance variables of class type. (We’ve already been doing this with **String**).

```
public class Person
{
  private String name;
  private Date born;
  private Date died;  //null means still alive
        . . .
```

- Again, we should declare each of the instance variables **private.**

Composition

- Note that the `Person` class contains three class type instance variables.

```
 private String name;
private Date born;
private Date died;
```

- The use of classes as instance variables is a design method known as ***aggregation*** or ***composition***.
- Composition is a fundamental way to reuse code, but there are coding considerations when composition is used.

Composition Considerations

- With composition, Person becomes a user of the Date and String classes.
- The Person class has no special privileges with respect to Date or String.
- The Person class should delegate responsibility to the Date and String classes whenever possible.

Designing a **`Person`** Class: Constraints

- In order to exist, a person must have (at least) a name and a birth date.
  - Therefore, it would make no sense to have a no-argument **`Person`** class constructor.
- A person who is still alive does not yet have a date of death.
  - Therefore, the **`Person`** class constructor will need to be able to deal with a **`null`** value for date of death.
- A person who has died must have had a birth date that preceded his or her date of death.
  - Therefore, when both dates are provided, they will need to be checked for consistency.

Designing a **Person** Class: Behaviors/Services

- After reading the problem description the following behaviors/services have been identified for the Person class.
  - Create a Person with a name, birthday, and date of death
  - Compare two Person objects to determine if they are identical
  - Format a string containing all Person attributes
  - Create a new Person which is the copy of an existing person

Iterative Process

- As we go into more detail in thinking about the behaviors, we might find other attributes that it would be important, or convenient, to model
- The process of designing a class is cyclical and evolving in nature

Designing a **`Person`** Class: The Class Invariant

- A statement that is always true for every object of the class is called a ***class invariant.***
  - A class invariant can help to define a class in a consistent and organized way.
- For the **`Person`** class, the following should always be true.
  - An object of the class **`Person`** has a name, a date of birth (which is not **`null`**), and if the object has a date of death, then the date of death is equal to or later than the date of birth
- Checking the **`Person`** class confirms that this is true of every object created by a constructor, and all the other methods (e.g., the private method **`consistent`**) preserve the truth of this statement.

Class Invariant Summary

- The class invariant is stated as part of the class documentation.
- Error checking in the constructor(s) and mutators insure that the class invariant is not violated.
- Methods of the class which do not change the class’ state may assume the class invariant holds.

A **Person** Class Constructor

```
public Person( String theName, Date birthDate, Date deathDate )
{
   // check that birthDate <= deathDate
   // consistent( ) is a private helper method
  if ( theName != null && consistent(birthDate, deathDate))
  {
    name = theName;
    born = new Date( birthDate );  // copy the birtheDate object
    if( deathDate == null )
      died = null;
    else
      died = new Date( deathDate );
  }
  else
  {
    // later we’ll deal with errors differently
    System.out.println( "Inconsistent Person parameters.“ );
    System.exit( 0 );
  }
}
```

Designing a **`Person`** Class:

The Class Invariant

```
/**  Class invariant: A Person always has a date of birth, and
   if the Person has a date of death, then the date of death is
   equal to or later than the date of birth.
     To be consistent, name and birthDate must not be null. If
   there is no date of death (deathDate == null), that is
   consistent with any birthDate. Otherwise, the birthDate
   must come before or be equal to the deathDate.

   precedes() is a boolean method in Date
*/

private boolean
consistent(Date birthDate, Date deathDate)
{
   if( birthDate == null )  return false;
   if( deathDate == null )  return true;
   return birthDate.precedes( deathDate )
         || birthDate.equals( deathDate );
 }
```

Designing a **`Person`** Class: The **`equals`** Method

- The definition of **`equals`** for the class **`Person`** includes an invocation of **`equals`** for the class **`String`**, and an invocation of the method **`equals`** for the class **`Date`**.
- The Person class passes responsibility for determining equality to the String and Date classes invoking their **`equals`** methods.
  - This is an important example of code reuse arising from the use of composition to implement Person.
- Java determines which **`equals`** method is being invoked from the type of its calling object.
- (Recall: **`equals()`** is special, because methods like **`ArrayList.indexOf()`** expect to call it.)

Designing a **Person** Class: The **equals** Method

```
public boolean equals(Person otherPerson)
{
  if (otherPerson == null)
    return false;
  else
    return  name.equals(otherPerson.name)
      &&    born.equals(otherPerson.born)
      &&    datesMatch(died, otherPerson.died);
}
```

This is the **equals** method for the String class

This is the **equals** method for the Date class

This is a special helper method that handles **null** Date references

Designing a **Person** Class: **datesMatch** Helper Method

```
/**  To match date1 and date2 must either be the
     same date or both be null.
*/

private boolean
datesMatch( Date date1, Date date2 )
{
  if( date1 == null )
    return date2 == null;       // both null is ok
  else if( date2 == null )      // && date1 != null
    return false;               // only one null not ok
  else // both dates are not null
    return date1.equals( date2 );
}
```

Designing a **Person** Class: The **toString** Method

- The **Person** class **toString** method includes invocations of the **Date** class **toString** method.
- Again, an example of code reuse and delegation of responsibility due to composition.

```
public String toString( )
{
  String diedString;
  if( died == null )
    diedString = ""; //Empty string
  else
    diedString = died.toString( );

  return name + ", " + born + "-" + diedString;
}
```

This is the same as **born.toString( )**

Designing a `Person` Class: Making a Copy

- Making a copy of an object requires a special method called a ***copy constructor***.
- A ***copy constructor*** is a constructor with a single argument of the same type as the class.
- The copy constructor should create an object that is a separate, independent object, but with the instance variables set so that it is an exact copy of the argument object.

Copy Constructor for a Class with Primitive Type Instance Variables

```
// a class that does not use composition can
// simply copy the values of the primitive instance
// variables

public Date( Date aDate )
{
  if( aDate == null ) // Not a real date object parameter
  {
     // we’ll handle errors differently later
    System.out.println( "Fatal Error.“ );
    System.exit( 0 );
  }

   // just copy the primitive variables using assignment
   // month is a String which is NOT primitive, but that’s ok
  month = aDate.month;
  day = aDate.day;
  year = aDate.year;
}
```

Copy Constructor for a Class Using Composition

- Because of composition, the technique used with Date will not work correctly with Person.

```
public Person( Person original )
{
   if( original == null )
   {
       System.out.println( "Fatal error.“ );
       System.exit( 0 );
   }
   name = original.name;        // ok
   born = original.born         //dangerous
   died = original.died         //dangerous
}
```

This code would not create an independent copy of the original object. Why not?

Copy Constructor for a Class with Class Type Instance Variables

- The actual copy constructor for the **`Person`** class is a "safe" version that creates completely new and independent copies of **`born`** and **`died`**, and therefore, a completely new and independent copy of the original **`Person`** object.
  - For example:

```
born = new Date( original.born );
```

- Note that in order to define a correct copy constructor for a class that uses composition, copy constructors must already be defined for the instance variables' classes (e.g. Date).

Copy Constructor for a Class Using Composition

```
public Person( Person original )
{
  if( original == null )
  {
    System.out.println( "Fatal error.“ );
    System.exit( 0 );
  }
  name = original.name;
  born = new Date( original.born );
  if( original.died == null )
    died = null;
  else
    died = new Date( original.died );
}
// Why don’t we have to create a new string for name?
```

Using and Misusing References

- When writing a program, it is very important to insure that private instance variables remain truly private.

- For a primitive type instance variable, just adding the `private` modifier to its declaration should insure that there will be no ***privacy leaks***.

- For a class type instance variable, adding the `private` modifier alone is not sufficient.

Pitfall: Privacy Leaks

- The previously illustrated examples from the `Person` class show how an incorrect definition of a copy constructor can result in a ***privacy leak***.
- A similar problem can occur with incorrectly defined mutator or accessor methods.
  - Wrong

    ```
    public Date getBornDate( )
    {
        return born;  //dangerous – why??
    }
    ```

  - Correct

    ```
    public Date getBornDate( )
    {
        return new Date( born );  //correct
    }
    ```

Privacy vs. Efficiency

- As a general rule, privacy considerations trump efficiency in most cases
- A possible class of exceptions is where the enclosing class is primarily implementing a storage model
  - This is very specific, but also very common
  - E.g.: ArrayLists and other collection classes do not by default do deep copies, and this was by design; you have to manage this yourself if desired
- If the included class can be considered a part of the state of the enclosing class, it should definitely be completely protected from privacy leaks.
  - E.g.: “born” instance variable of Person class

Composition and Encapsulation

- Suppose that the user of the Person class has a new requirement: they wish to abbreviate a Person’s birth month by displaying only the first 3 characters.
- How do we provide this feature in an appropriate OOP way keeping in mind the principle of encapsulation?

One Way

- Add an accessor for the born date (getBornDate) to the Person class and add an accessor for the month (getMonth) to the Date class.
- Then the Person class user can write this code.

```
Person bob =
   new Person(“Bob”, new Date(“January”, 14, 1944), null);
String abbrev =
   bob.getBornDate( ).getMonth().substring(0,2);
System.out.println( abbrev );
```

What’s good or bad with this approach?

Another Way

- Add a new method to the Date class to return the month’s abbreviation

  ```
  public String getMonthAbbreviation( )
  { return month.substring(0, 2); }
  ```

- Add a new method to the Person class to return the born date abbreviation

  ```
  public String getBornMonthAbbreviation( )
  { return born.getMonthAbbreviation(); }
  ```

- Now the user of the Person class writes this code.

```
Person bob =
   new Person(“Bob”, new Date(“January”, 14, 1944), null);
String abbrev = bob.getBornMonthAbbreviation();
System.out.println( abbrev );
```

- What’s good or bad with this approach?

Composition with Arrays

- Just as a class type can be used as an instance variable, arrays can also be used as instance variables.

- We can define an array with a primitive base type.

```
        private double[ ] grades;
```

  Or, an array with a class base type.

```
        private Date [ ] dates;
```

Privacy Leaks with Array Instance Variables

- If an accessor method is provided for the array special care must be taken just as when an accessor returns a reference to any private object.

```
public double[ ] getGrades( )
{
   return grades;
}
```

  - The example above will result in a *privacy leak.*
  - Why is this so?

Privacy Leaks with Array Instance Variables

- The previous accessor method would simply return a reference to the array **`grades`** itself.

- Instead, an accessor method should return a reference to a *deep copy* of the private array object.

- Below, **`grades`** is an array which is an instance variable of the class containing the **`getGrades`** method.

```
public double[ ] getGrades( )
{
  double[ ] temp = new double[ grades.length ];
  for( int i = 0; i < grades.length; i++ )
    temp[ i ] = grades[ i ];
  return temp;
}
```

Privacy Leaks with Array Instance Variables

- If a private instance variable is an array that has a mutable class as its base type, then copies must be made of each class object in the array when the array is copied.

```
public Date[ ] getDates( )
{
   Date[ ] temp = new Date[ dates.length ];
   for( int i = 0; i < dates.length; i++ )
      temp[ i ] = new Date( dates[ i ] );
   return temp;
}
```

But what if…

the user really wants to change the array within the class?

a. The user shouldn’t know that the class uses an array.
b. The array must represent some abstract data element in the class (eg student grades).
c. Provide a method that changes the the abstract data element without revealing the existence of an array.

Remember…

“Keep it secret, keep it safe”