Programming Concepts II

Summary

VANIER CÉGEP/COLLEGE

Niloufar Salehi

Summer 2023

REFERENCES

These slides have been extracted, modified and updated from the following references:

- Absolute Java by Walter Savitch, 6th Edition or later.
- Absolute Java by Rose Williams, Binghamton University, Kenrick Mock, University of Alaska Anchorage
- Object Oriented Programming course lecture notes of Dr. Nancy Acemian.

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

- A simple **`Person`** class could contain instance variables representing a person's name, the date on which they were born, and the date on which they died
- These instance variables would all be class types: name of type **`String`**, and two dates of type **`Date`**
- As a first line of defense for privacy, each of the instance variables would be declared **`private`**

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

Designing a **`Person`** Class: Constructor

- 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

A **Person** Class Constructor

```
public Person(String initialName, Date birthDate,
                                  Date deathDate)
{
  if (consistent(birthDate, deathDate))
  { name = initialName;
    born = new Date(birthDate);
    if (deathDate == null)
      died = null;
    else
      died = new Date(deathDate);
  }
  else
  { System.out.println("Inconsistent dates.");
    System.exit(0);
  }
}
```

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 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

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

Designing a `Person` Class: the `equals` and `datesMatch` Methods

- 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`**
- Java determines which **`equals`** method is being invoked from the type of its calling object
- Also note that the **`died`** instance variables are compared using the **`datesMatch`** method instead of the equals method, since their values may be **`null`**

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));
}
```

Designing a **Person** Class: the **matchDate** Method

```
/**  To match date1 and date2 must either be the
     same date or both be null.
*/
private static boolean datesMatch(Date date1,
                                   Date date2)
{
  if (date1 == null)
    return (date2 == null);
  else if (date2 == null) //&& date1 != null
    return false;
  else // both dates are not null.
    return(date1.equals(date2));
}
```

Designing a **`Person`** Class: the **`toString`** Method

- Like the **`equals`** method, note that the **`Person`** class **`toString`** method includes invocations of the **`Date`** class **`toString`** method

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

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

Copy Constructors

- 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
- Note how, in the `Date` copy constructor, the values of all of the primitive type private instance variables are merely copied

Copy Constructor for a Class with Primitive Type Instance Variables

```
public Date(Date aDate)
{
  if (aDate == null) //Not a real date.
  {
    System.out.println("Fatal Error.");
    System.exit(0);
  }

  month = aDate.month;
  day = aDate.day;
  year = aDate.year;
}
```

Example: Person.java and Date.java

Copy Constructor for a Class with Class Type Instance Variables

- Unlike the **`Date`** class, the **`Person`** class contains three class type instance variables
- If the **`born`** and **`died`** class type instance variables for the new **`Person`** object were merely copied, then they would simply rename the **`born`** and **`died`** variables from the original **`Person`** object

```
born = original.born //dangerous
died = original.died //dangerous
```

  – This would not create an independent copy of the original object
  – (see *more details* in the next slide)

Copy Constructor for a Class with Class Type Instance Variables

- More details:

We want the object created to be an independent copy of original. That would not happen if we had used the following instead:

```
public Person(Person original) //Unsafe
{
    if (original == null )
    {
        System.out.println("Fatal error.");
        System.exit(0);
    }
    name = original.name;
    born = original.born; //Not good.
    died = original.died; //Not good.
}
```

Copy Constructor for a Class with Class Type Instance Variables

- More details:

Although this alternate definition looks innocent enough and may work fine in many situations, it does have serious problems.

The “Not good.” code simply copies references from original.born and original.died to the corresponding arguments of the object being created by the constructor. So, the object created is not an independent copy of the original object.

Copy Constructor for a Class with Class Type Instance Variables

- More details:

For example, consider the code

```
Person original =
new Person("Natalie Dressed",
new Date("April", 1, 1984), null);

Person copy = new Person(original);

copy.setBirthYear(1800);
System.out.println(original);
```

The output would be

```
Natalie Dressed, April 1, 1800
```

Copy Constructor for a Class with Class Type Instance Variables

- More details:

When we changed the birth year in the object copy, we also changed the birth year in the object original. This is because we are using our unsafe version of the copy constructor. Both original.born and copy.born contain the same reference to the same Date object.

This all happens because we used the unsafe version of the copy constructor. Fortunately, here we use a safer version of the copy constructor that sets the born instance variables as follows:

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

which is equivalent to

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

Copy Constructor for a Class with Class Type Instance Variables

- So, 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 has class type instance variables, copy constructors must already be defined for the instance variables' classes

Copy Constructor for a Class with Class Type Instance Variables

```
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);
}
```

Pitfall: Privacy Leaks

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

```
public Date getBirthDate()
{
    return born;  //dangerous
}
```

  - Instead of:

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

Deep Copy Versus Shallow Copy

- A *deep copy* of an object is a copy that, with one exception, has no references in common with the original

- Any copy that is not a deep copy is called a *shallow copy*
  - This type of copy can cause dangerous privacy leaks in a program

A First Look at the `clone` Method

- Every object inherits a method named `clone` from the class `Object`
  - The method `clone` has no parameters
  - It is supposed to return a deep copy of the calling object
- However, the inherited version of the method was not designed to be used as is
  - Instead, each class is expected to override it with a more appropriate version

A First Look at the `clone` Method

- The heading for the `clone` method defined in the `Object` class is as follows:

  ```
  protected Object clone()
  ```

- The heading for a `clone` method that overrides the `clone` method in the `Object` class can differ somewhat from the heading above
  - A change to a more permissive access, such as from protected to public, is always allowed when overriding a method definition
  - Changing the return type from `Object` to the type of the class being cloned is allowed because every class is a descendent class of the class `Object`
  - This is an example of a covariant return type

A First Look at the `clone` Method

- If a class has a copy constructor, the **`clone`** method for that class can use the *copy constructor* to create the copy returned by the **`clone`** method

```
public Sale clone()
{
  return new Sale(this);
}
```

and another example:

```
public DiscountSale clone()
{
  return new DiscountSale(this);
}
```

Pitfall: Sometime the `clone` Method Return Type is `Object`

- Prior to version 5.0, Java did not allow covariant return types
  - There were no changes whatsoever allowed in the return type of an overridden method
- Therefore, the `clone` method for all classes had `Object` as its return type
  - Since the return type of the clone method of the `Object` class was `Object`, the return type of the overriding clone method of any other class was `Object` also

Pitfall: Sometime the **`clone`** Method Return Type is **`Object`**

- Prior to Java version 5.0, the **`clone`** method for the **`Sale`** class would have looked like this:

```
public Object clone()
{
  return new Sale(this);
}
```

- Therefore, the result must always be type cast when using a **`clone`** method written for an older version of Java

```
Sale copy = (Sale)original.clone();
```

Pitfall: Sometime the `clone` Method Return Type is `Object`

- It is still perfectly legal to use Object as the return type for a clone method, even with classes defined after Java version 5.0
  - When in doubt, it causes no harm to include the type cast
  - For example, the following is legal for the clone method of the Sale class:

```
Sale copy = original.clone();
```

  - However, adding the following type cast produces no problems:

```
Sale copy = (Sale)original.clone();
```

Pitfall: Limitations of Copy Constructors

- Although the copy constructor and **`clone`** method for a class appear to do the same thing, there are cases where only a **`clone`** will work
- For example, given a method **`badcopy`** in the class **`Sale`** that copies an array of sales
  - If this array of sales contains objects from a derived class of **`Sale`**(i.e., **`DiscountSale`**), then the copy will be a plain sale, not a true copy

```
b[i] = new Sale(a[i]); //plain Sale object
```

Pitfall: Limitations of Copy Constructors

- However, if the **`clone`** method is used instead of the copy constructor, then (because of late binding) a true copy is made, even from objects of a derived class (e.g., **`DiscountSale`**):

```
b[i] = (a[i].clone());//DiscountSale object
```

  - The reason this works is because the method **`clone`** has the same name in all classes, and polymorphism works with method names
  - The copy constructors named **`Sale`** and **`DiscountSale`** have different names, and polymorphism doesn't work with methods of different names

The `Cloneable` Interface

- The **`Cloneable`** interface is another unusual example of a Java interface
  - It does not contain method headings or defined constants
  - It is used to indicate how the method **`clone`** (inherited from the **`Object`** class) should be used and redefined

The `Cloneable` Interface

- The method `Object.clone()` does a bit-by-bit copy of the object's data in storage
- If the data is all primitive type data or data of immutable class types (such as `String`), then this is adequate
  - This is the simple case
- The following is an example of a simple class that has no instance variables of a mutable class type, and no specified base class
  - So the base class is `Object`

Implementation of the Method **`clone`**: Simple Case

Display 13.7 Implementation of the Method clone (Simple Case)

```
public class YourCloneableClass implements Cloneable
{
        .
        .
        .
    public Object clone()
    {
        try
        {
            return super.clone();//Invocation of clone
                                 //in the base class Object
        }
        catch(CloneNotSupportedException e)
        {//This should not happen.
            return null; //To keep the compiler happy.
        }
    }
        .
        .
        .
}
```

Works correctly if each instance variable is of a primitive type or of an immutable type like String.

The `Cloneable` Interface

- If the data in the object to be cloned includes instance variables whose type is a mutable class, then the simple implementation of `clone` would cause a *privacy leak*
- When implementing the `Cloneable` interface for a class like this:
  - First invoke the `clone` method of the base class `Object` (or whatever the base class is)
  - Then reset the values of any new instance variables whose types are mutable class types
  - This is done by making copies of the instance variables by invoking *their* clone methods

The `Cloneable` Interface

- Note that this will work properly only if the `Cloneable` interface is implemented properly for the classes to which the instance variables belong
  - And for the classes to which any of the instance variables of the above classes belong, and so on and so forth
- The following shows an example

Implementation of the Method `clone`: Harder Case

Display 13.8 Implementation of the Method `clone` (Harder Case)

```
public class YourCloneableClass2 implements Cloneable
{
    private DataClass someVariable;
        .
        .
        .
    public Object clone()
    {
        try
        {
            YourCloneableClass2 copy =
                              (YourCloneableClass2)super.clone();
            copy.someVariable = (DataClass)someVariable.clone();
            return copy;
        }
        catch(CloneNotSupportedException e)
        {//This should not happen.
            return null; //To keep the compiler happy.
        }
    }
        .
        .
        .
}
```

`DataClass` is a mutable class. Any other instance variables are each of a primitive type or of an immutable type like `String`.

If the clone method return type is `DataClass` rather than `Object`, then this type cast is not needed.

The class `DataClass` must also properly implement the `Cloneable` interface including defining the `clone` method as we are describing.