Advanced Programming

Using and Misusing References

Copy Constructors
The **new** Operator and Anonymous Objects

- The **new** operator invokes a constructor which initializes an object, and returns a reference to the location in memory of the object created
  - This reference can be assigned to a variable of the object's class type

  ```
  ToyClass variable1 = new ToyClass("Joe", 42);
  ```

- Sometimes the object created is used as an argument to a method, and never used again
  - In this case, the object need not be assigned to a variable, i.e., given a name

  ```
  if (variable1.equals(new ToyClass("JOE", 42)))
      System.out.println("Equal");
  else
      System.out.println("Not equal");
  ```

- An object whose reference is not assigned to a variable is called an **anonymous object**
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, however, adding the private modifier alone is not sufficient
/**
 * Class for a person with a name and dates for birth and death.
 * 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.
 */

public class Person {

    private String name;
    private Date born;
    private Date died; // null indicates still alive.

    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. Aborting.");
            System.exit(0);
        }
    }

    // ...
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);
}

public void set(String newName, Date birthDate, Date deathDate)
<Definition of this method is Self Test Exercise 41.>
public String toString()
{
    String diedString;
    if (died == null)
        diedString = "";  // Empty string
    else
        diedString = died.toString();

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

public boolean equals(Person otherPerson)
{
    if (otherPerson == null)
        return false;
    else
        return (name.equals(otherPerson.name)
                && born.equals(otherPerson.born)
                && datesMatch(died, otherPerson.died));
}
public void setBirthDate(Date newDate) {
  else if (consistent(newDate, died)) 
    born = new Date(newDate);
*/

/**
 * Precondition: newDate is a consistent date of birth.
 * Postcondition: Date of birth of the calling object is newDate.
 */
private static boolean datesMatch(Date date1, Date date2) {
  if (date1 == null) 
    return (date2 == null); 
  else if (date2 == null) 
    return false; 
  else if (date1.equals(date2)) 
    return true; 
  return false; 
}
/**
 * Precondition: newDate is a consistent date of death.
 * Postcondition: Date of death of the calling object is newDate.
 */

public void setDeathDate(Date newDate)
{
    if (!consistent(born, newDate))
    {
        System.out.println("Inconsistent dates. aborting.");
        System.exit(0);
    }

    if (newDate == null)
        died = null;
    else
        died = new Date(newDate);
}

public void setName(String newName)
{
    name = newName;
}
/**
   * Precondition: The date of birth has been set, and changing the year part of the date of birth will give a consistent date of birth.
   * Postcondition: The year of birth is (changed to) newYear.
   */
   
   public void setBirthYear(int newYear)
   {
       if (born == null) //Precondition is violated
       {
           System.out.println("Fatal Error. Aborting.");
           System.exit(0);
       }
       born.setYear(newYear);
       if (!consistent(born, died))
       {
           System.out.println("Inconsistent dates. Aborting.");
           System.exit(0);
       }
   }
/**
 * Precondition: The date of death has been set, and changing the year part of the date of death will give a consistent date of death.
 * Postcondition: The year of death is (changed to) newYear.
 */

public void setDeathYear(int newYear)
{
    if (died == null) //Precondition is violated
    {
        System.out.println("Fatal Error. Aborting.");
        System.exit(0);
    }
    died.setYear(newYear);
    if (!consistent(born, died))
    {
        System.out.println("Inconsistent dates. Aborting.");
        System.exit(0);
    }
}

public String getName()
{
    return name;
}
public Date getBirthDate()
{
    return new Date(born);
}

public Date getDeathDate()
{
    if (died == null)
        return null;
    else
        return new Date(died);
}

/**
 * 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: 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**.

```java
public class Person {
    private String name;
    private Date born;
    private Date died;  //null is still alive
    . . .
}
```
Designing a **Person** Class: Constructor

- Normally, classes contain a no-argument constructor, however...
- 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

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.
A Person Class Constructor

```java
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);
    }
}
```

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.
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
/** 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**
• If a method has a parameter of a class type, then null can be used as the corresponding argument when the method is invoked.

• Sometimes using null as an argument can be the result of an error, but it can sometimes be an intentional argument.

• For example, the class Person uses null for a date of death to indicate that the person is still alive. So null is sometimes a perfectly normal argument for methods such as consistent.

• Method definitions should account for null as a possible argument and not assume the method always receives a true object to plug in for a class parameter.
Designing a **Person** Class: the **equals** Method

```java
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

```java
/**  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

```java
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.
public class Date {

    private String month;
    private int day;
    private int year; //a four digit number.

    public Date(String monthString, int day, int year) {
        setDate(monthString, day, year);
    }

    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;
    }
}
public void setDate(String monthString, int day, int year)
{
    if (dateOK(monthString, day, year))
    {
        this.month = monthString;
        this.day = day;
        this.year = year;
    }
    else
    {
        System.out.println("Fatal Error");
        System.exit(0);
    }
}

public void setYear(int year)
{
    if ( (year < 1000) || (year > 9999) )
    {
        // The method dateOK checks that the date is a legitimate date, such as not having more than 31 days.
        // More code...
    }
}
```java
    System.out.println("Fatal Error");
    System.exit(0);
    }
  }
  }

  public String toString()
    ...
    ...
  public boolean equals(Date otherDate)
    ...
    /
    /**
    Returns true if the calling object date is before otherDate (in time).
    */
  public boolean precedes(Date otherDate)
    ...
  private boolean dateOK(String monthString, int dayInt, int yearInt)
    ...
  }

  The complete definition of equals is given in the answer to Self-Test Exercise 37, and is a better version than the one given in Chapter 4.

  These methods have the obvious meanings. If you need to see a full definition, see Display 4.11 in Chapter 4 and Self-Test Exercise 37 in this chapter.
```
Copy Constructor for a Class with Primitive Type Instance Variables

```java
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;
}
```
Date date1 = new Date("January", 1, 2006);
Date date2 = new Date(date1);

date2.setDate("July", 4, 1776);
System.out.println(date1);
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
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

• 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:
    ```javascript
    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

```java
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:
    ```java
    public Date getBirthDate() {
        return born;  //dangerous
    }
    ```
  - Instead of:
    ```java
    public Date getBirthDate() {
        return new Date(born);  //correct
    }
    ```
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–
Material Covered

• Chapter 5: pages 329 –348
• –Absolute Java, 4\textsuperscript{th} edition, Savitch
Source

• Absolute Java, Savitch
  – Copyright © 2010 Pearson Addison-Wesley. All rights reserved.