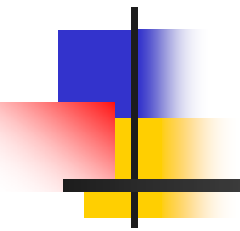
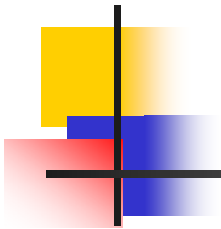
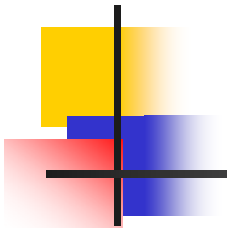


Classes 4/5



Outline

- Using and Misusing References
- Designing A **`Person`** Class
- Copy Constructors
- Mutable and Immutable Classes
- Deep Copy Versus Shallow Copy



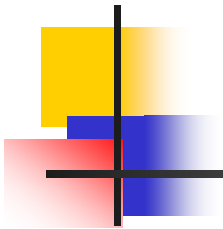
- Using and Misusing References

- 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

-- 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
        . . .
```

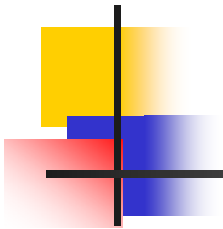


... -- Designing a **`Person`** Class: Constructor ...

- 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

... -- Designing 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);
  }
}
```

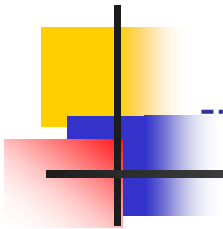


-- Designing a **`Person`** Class: the Class Invariant

- 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

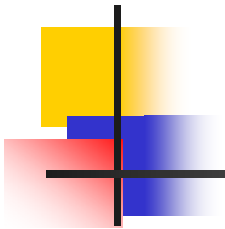
-- Designing a **Person** Class: the Class Invariant

```
/**  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

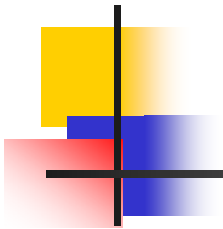
```
/**  To match date1 and date2 must either be the
     same date or both be null.
*/
private static boolean datesMatch(Date date1, Date date2)
{
  if (date1 == null)
    return (date2 == null);
  else if (date2 == null) //&& date1 != null
    return false;
  else // both dates are not null.
    return(date1.equals(date2));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

-- 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);
}
```



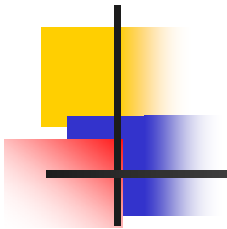
- Copy Constructors

- 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;
}
```

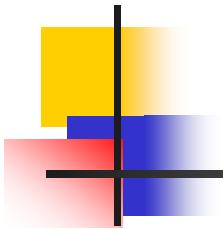


... -- Copy Constructor for a Class with Class Type Instance Variables ...

- 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

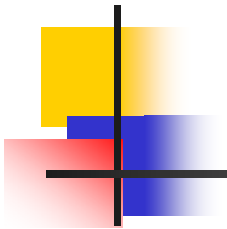


... -- Copy Constructor for a Class with Class Type Instance Variables ...

- 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 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);
}
```

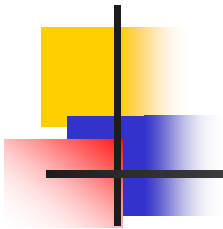
Pitfall: Privacy Leaks

- 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
}
```

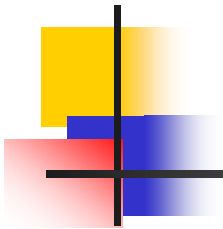


- Mutable and Immutable Classes ...

- The accessor method **`getName`** from the **`Person`** class appears to contradict the rules for avoiding privacy leaks:

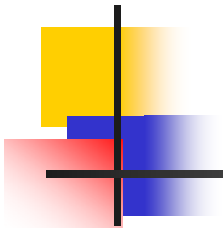
```
public String getName()
{
  return name; //Isn't this dangerous?
}
```

- Although it appears the same as some of the previous examples, it is not: The class **`String`** contains no mutator methods that can change any of the data in a **`String`** object



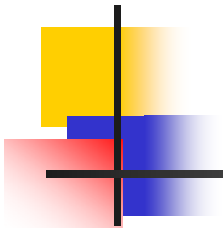
... - Mutable and Immutable Classes ...

- A class that contains no methods (other than constructors) that change any of the data in an object of the class is called an *immutable class*
  - Objects of such a class are called *immutable objects*
  - It is perfectly safe to return a reference to an immutable object because the object cannot be changed in any way
  - The `String` class is an immutable class



... - Mutable and Immutable Classes

- A class that contains public mutator methods or other public methods that can change the data in its objects is called a *mutable class*, and its objects are called *mutable objects*
  - Never write a method that returns a mutable object
  - Instead, use a copy constructor to return a reference to a completely independent copy of the mutable object



- 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
  - Exception: References to immutable objects are allowed to be shared
- 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

THE END