COMP-248
Object Oriented Programming I

Defining Classes II

By Emad Shihab, PhD, Fall 2015,
Parts of the slides are taken from Prof. L. Kosseim
Adapted for Section EE by S. Ghaderpanah, Fall 2015

Next:

1. `static` methods and variables
2. Wrapper classes
3. References and class parameters
4. **Using and Misusing references**
5. Packages and `javadoc`

Example: Swapping 2 int

```
public class PassDriver
{
  public static void main(String[] arg)
  {
     int x = 10;
     int y = 20;

     System.out.println("1 " + x + " " + y);
     swap(x, y);
     System.out.println("4 " + x + " " + y);

   }

  public static void swap(int param1, int param2)
  {
    System.out.println("2 " + param1 + " " + param2);
    int temp = param1;
    param1 = param2;
    param2 = temp;
    System.out.println("3 " + param1 + " " + param2);
   }
}
```

```
1 10 20
2 10 20
3 20 10
4 10 20
```

Output

Example: Swapping 2 MyInt

PassDriver.java

```
public class PassDriver {
  public static void main(String[] arg) {
     MyInt a = new MyInt(10);
     MyInt b = new MyInt(20);
     System.out.println("1 " + a.getValue() + " " + b.getValue());
     swap(a, b);
     System.out.println("4 " + a.getValue() + " " + b.getValue());
   }

  public static void swap(MyInt param1, MyInt param2)
  {
    System.out.println("2 " + param1.getValue() + " " + param2.getValue());
    MyInt tmp = new MyInt(param1.getValue());
    param1.setValue(param2.getValue());
    param2.setValue(tmp.getValue());
    System.out.println("3 " + param1.getValue() + " " + param2.getValue());
   }
}
```

```
public class MyInt {
    private int value;

    public MyInt(int data) { this.value = data; }
    public void setValue(int data) { this.value = data; }
    public int getValue() { return value; }
}
```

MyInt.java

```
1 10 20
2 10 20
3 20 10
4 20 10
```

Output

Conclusion

if argument is a primitive type:

A method <u>cannot change</u> the value of the argument

if argument is a reference:

A method <u>can change</u> the value of an instance variable of an objet passed as argument

Example 1: Swapping 2 Account

```
public static void main(String[] arg)
{
  Account a = new Account("ted", 123, 100);
  Account b = new Account("mary", 456, 99);

  System.out.println(a + " " + b);
  swap(a, b);
  System.out.println(a + " " + b);
}

public static void swap(Account a, Account b)
{
      Account tmp;
      tmp = a;
      a = b;
      b = tmp;
}
```

Output

Example 2: Swapping 2 Account

```
public static void main(String[] arg)
{
  Account a = new Account("ted", 123, 100);
  Account b = new Account("mary", 456, 99);

  System.out.println(a + " " + b);
  swap(a, b);
  System.out.println(a + " " + b);
}

public static void swap(Account a, Account b)
{
    Account tmp;
    tmp = (Account) a.clone();
    a = (Account) b.clone();
    b = (Account) tmp.clone();
}
```

Driver

```
public class Account {
  …
  public Object clone() {
    Account copy = new Account();
    copy.acctNumber = this.acctNumber;
    copy.balance = this.balance;

copy.name=(this.name).substring(0);
    return copy;
  }
```

Account.java

Output

Example 3: Swapping 2 Account

```
public static void main(String[] arg)
{
  Account a = new Account("ted", 123, 100);
  Account b = new Account("mary", 456, 99);

  System.out.println(a + " " + b);
  swap(a, b);
  System.out.println(a + " " + b);
}

public static void swap(Account a, Account b)
{
    Account tmp;
    tmp = (Account) a.clone();
    a.changeTo(b);
    b.changeTo(tmp);
}
```

Driver

```
public class Account {
   …
 public Object clone() {
    Account copy = new Account();
    copy.acctNumber = this.acctNumber;
    copy.balance = this.balance;
    copy.name=(this.name).substring(0);
    return copy;
  }
 public void changeTo(Account b) {
    this.acctNumber = b.acctNumber;
    this.balance = b.balance;
    this.name = (b.name).substring(0);
  }
```

Output

Using and Misusing References

It is very important to insure that private instance variables remain truly private

- If an instance variable is:
  - A primitive type, just make it `private`
  - A class type, `private` may not be enough …

```
public class Person
{
  private String name;
  private Date born;
  private Date died;
…
}
```

Constructor for class Person

```
public Person(String initialName, Date birthDate, Date deathDate) {
  if (consistent(birthDate, deathDate))
  {
     name = initialName;
     born = new Date(birthDate); // why not just born = birthdate??
     if (deathDate == null)
       died = null;
     else
       died = new Date(deathDate); // why not just born = deathdate??
  }
  else
      System.exit(0);
}
```

```
Date birth = new Date("April", 1, 1970);
Person original = new Person("john", birth, null);
birth.setMonth("January");
```

Copy Constructors

A copy constructor:

- A constructor with only <u>one argument of the same type as the class</u>

- Creates a separate, independent object that is copy of the argument object

Copy Constructor

For instance variables that are **primitive types OR**
for instance variables that are **objects of an immutable class**

```
public Date(Date aDate) {
  if (aDate == null) System.exit(0); //Not a real date.

  month = aDate.month; // a string
  day = aDate.day;     // an int
  year = aDate.year;   // an int
}
```

Copy Constructor

for instance variables that are **objects of a "regular" class (mutable)**

```
public Person(Person original) {
  if (original == null) System.exit(0);

  name = original.name;            // a string
  born = new Date(original.born);  // a reference to a class
  if (original.died == null)
      died = null;
  else
      died = new Date(original.died); // a reference to a class
}
```

Just Checking …

A copy constructor has ___________ parameters.

A. zero
B. one
C. two
D. three
E. How ever many one needs

Mutable vs. Immutable Classes

- *Immutable class*
  - A class that contains no methods (other than constructors) that change the data in an object of the class
  - It is safe to return a reference to an immutable object because the object cannot be changed

    ex: the `String` class and `Wrapper` classes

Mutable vs. Immutable Classes

- *Mutable class*
  - A class that contains public methods that can change the data in its objects
  - **Never write a method that returns a mutable object**
  - Instead, use a copy constructor, and return a copy of the object

Privacy Leaks again...

Incorrectly defined mutator or accessor methods

ex:

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

Deep Copy Versus Shallow Copy

- A <u>deep copy</u> of an object:
  - A copy that has no references in common with the original
  - Exception: References to immutable objects are allowed to be shared

- A <u>shallow copy</u> of an object:
  - Can cause dangerous privacy leaks in a program

Just Checking …

A condition that allows a programmer to circumvent the private modifier and change the private instance variable is called:

A. a copy constructor
B. a privacy leak
C. a class invariant
D. an anonymous object

Next:

1. `static` methods and variables
2. Wrapper classes
3. References and class parameters
4. Using and Misusing references
5. **Packages and `javadoc`**