Mutable and Immutable Objects

Mutate means *to change*. In object-oriented programming, objects from a class are *mutable* if they can be changed after creation, and *immutable* if they cannot be changed.

For example, instances of the **`java.util.Date`** class are *mutable*, while Strings are *immutable.* This example illustrates:

```
Date date = new Date( 100, Calendar.JANUARY, 15 ); // 15 Jan 2000
String fruit = "apple banana";
System.out.println( now );
System.out.println( fruit );
now.setMonth( Calendar.MAY );
fruit.replaceAll( "apple", "yogurt"); // replace apple with yogurt
System.out.println( now );
System.out.println( fruit );
```

When you run this code it prints:

```
Sat Jan 15 00:00:00 ICT 2000
apple banana
Mon May 15 00:00:00 ICT 2000
apple banana
```

The Date changed, but the String did not. The `replaceAll( )` method creates a new String, but doesn't change the existing string. In fact, *none of the methods* in the String class will change a String. Try some: `toUpperCase()`, `toLowerCase()`, etc.

Some common mutable and immutable classes are:

	Mutable	*Immutable*
string values	StringBuilder	String
dates	java.util.Date	java.time.LocalDate

Exercise: list some mutable and immutable classes in the Java API.

Mutable	*Immutable*

Exercise: for the mutable types, write a code example to prove that the objects are mutable.

Advantages of Immutable Objects

From both a design and implementation point of view, immutable objects have some benefits:

- easier to test
- safe to share references to the same object. Hence, if an object is an attribute of *another* object, that other object can safety return a reference to the attribute (no copy required).
- immutable objects are thread-safe

How Mutable Objects Can Break Encapsulation

Consider this Person class, which has a name and birthday:

```
public class Person {
     private String name;
     private Date birthday;

     /** constructor for new Person objects */
     public Person( String name, Date birth ) {
          this.name = name;
          this.birthday = birth;
     }
     /** get the person's name */
     public String getName( ) {
          return name;
     }
     /** get the person's birthday */
     public Date getBirthday( ) {
          return birthday;
     }
     public String toString() {
          return String.format("%s born on %tD", name, birthday);
     }
}
```

Are Person objects immutable? The class does not have any mutator ("set") methods, but Person is *still* mutable. Here's an example:

```
Date bday = new Date( 55, Calendar.OCTOBER, 28 );
Person bill = new Person("Bill Gates", bday);
System.out.println( bill );
//  now bill.birthday references the same object as bday. What if we change bday?
bday.setYear( 100 );
System.out.println( bill );
//  yeah!  Bill just got younger.
```

We can fix this problem by creating a copy of the birthday parameter instead of just copying the reference. In the constructor:

```
     public Person( String name, Date birth ) {
          this.name = name;
          // create a new date using data from the parameter
          this.birthday = new Date( birth.getTime() );
     }
```

But Person *still* has an encapsulation problem. Consider this example:

```
Person bill = new Person("Bill Gates", new Date(55, 9, 28) );
Date birth = bill.getBirthday( );
birth.setYear( 0 ); // set birthday year to 1900
// Does this change bill's birthday?
System.out.println( bill );
```

getBirthday( ) returns a *reference* to the object's birthday. Now the outside code has a *reference* to the birthday object and can change it (because Date is mutable).

The solution is *accessor methods should not return a reference to mutable attributes. If you want to preserve encapsulation, return a copy or immutable form of the object.*

We can modify `getBirthday( )` to preserve immutability like this:

```
    public Date getBirthday( ) {
        return (Date) birthday.clone( );
    }
```

This example introduces a new way to copy an object: `clone( ). clone` creates a *deep copy* of the object. Cloning can be a time-consuming operation, so not all classes support it. For example, `String` cannot be cloned (since Strings are immutable, there is no need to clone them). Classes that provide a working `clone` method will implement the *`Cloneable`* interface. Check the Java API.

Preserving Immutability Can Be Expensive

Suppose we add an email attribute to Person. Since a Person can have many email addresses, we store them in a Set. We'll provide an `addAddress` that checks for valid email address, and `getAddresses` that returns *all* the person's email addresses:

```
public class Person {
    private String name;
    private Date birthday;
    private Set<String> addresses; // email addresses
    private static final String
        PATTERN = "([\\w\\d]+[\\w\\d\\.]*)@((\\w\\d-)+\\.?)+";

    public Person( String name, Date birth ) {
        this.name = name;
        this.birthday = new Date( birth.getTime() );
        addresses = new HashSet<String>( );
    }

    public boolean addAddress(String address) {
        if ( address == null ) return false;
        if ( ! address.matches(PATTERN) ) return false;
        return addresses.add( address );
    }

    /** get all the email addresses */
    public Set<String> getAddresses( ) {
        return addresses;
    }
```

Can a malicious programmer bypass the addAddress( ) method to modify the person's email addresses? Yes! Since a HashSet is mutable. For example,

```
Set<String> emails = person.getAddresses( );
emails.clear(); // remove all elements!
```

To prevent a user from serrupticiously modifying a collection, we have two choices:

1) return a copy of the collection. If elements of the collection are mutable, you must copy each element, too.

2) return an *immutable form* of the collection. This works if the elements are *immutable* (such as Strings).

How To Write an Immutable Class

1. Declare all attributes (fields) as private.

2. final attributes (constants) may be public *if their type is an immutable class*.

3. Don't provide any mutator methods.

4. If any of the attributes are themselves instances of a *mutable* type, then you create a deep copy of any values passed as parameters to the constructor(s).

5. Accessor ("get") methods that return an attribute that is itself mutable should return a *copy* or clone of the attribute.

5b. Arrays are mutable, so if your class provides an accessor for an array attribute, you must copy the entire array and return the copy. This can be expensive.

5c. Most collections are mutable, so if you class provides an accessor for a collection, you must either copy the entire collection (expensive) or return an *immutable form* of the collection.

Immutable Objects in Software Design: Value Types

In designing software, if a thing represent only a *value*, where you are interested only in the *value* of the object, not its *identity*, then consider making it an immutable type.

Common examples of "value" objects in modeling are:

- an address
- postal code (ZIPcode)
- telephone number
- money

The benefit of doing this is that you can treat the immutable object like a value in your code. For example, an Address represents a value. If Address is immutable, we can copy an Address between objects using assignment, without worrying that one object might change the address.

You can get a (small) performance benefit by declaring immutable classes to be `final` (cannot be subclassed), and declaring the attributes to be `final`, too.

Exercise: write an example of a "value" type object declared as a `final` class with `final` attributes.

Cost of Immutable Objects

Since immutable objects can't be changed, if your application *does* need to change the object then you have to create a new one. An example is strings. Every time your code appends to a String it creates a new String. Consider this loop:

```
// create a String of 10,000 copies of the letter 'a'
String result = "";
int count = 10000;
while( count > 0 ) {
     result = result + "a";
```

```
     count--;
}
```

How many objects does this code snippet create? Every time it appends another "a" to result, it must copy the entire string to a new string object. The old String is discarded.

This is a serious performance and memory issue for applications that build large Strings, such as web applications. The solution is to build the string using a *mutable string*, then convert the result into an *immutable string*.

What's a mutable String? Java has two classes: StringBuffer (which is thread-safe) and StringBuilder (not thread-safe, but slightly faster). For the above code snippet, we'd do:

```
// create a String of 10,000 copies of the letter 'a'
StringBuilder buffer = new StringBuilder();
int count = 10000;
while( count > 0 ) {
     buffer.append("a");
     count--;
}
String result = buffer.toString();
```

Exercise:

Write a program to read the contents of a file into a String.

a) Write a `readfile` method that reads a FileInputStream one character at a time using InputStream's read( ) method, and append the results to a String.

b) Return the String to the calling method (`main` in your program).

c) Output the length of the String and the amount of time used to read the file. To compute the amount of time, call System.nanoTime() before and after invoking readfile. Divide by 1.0E+9 to convert nanoseconds to seconds.

d) repeat the exercise using StringBuffer instead of String. Compare the run times.

You may also want to compare StringBuilder. You'll need a text file of length 10KB - 500KB for the runtimes to be long enough to be meaningful.

Mutable/Immutable Pattern

Many applications would benefit from immutable objects, but some part of the application needs to modify the object. For example, you'd like to make Address be immutable since its conceptually a "value" type, but you also want to be able to modify a Person's address or build addresses from a database.

The Mutable/Immutable Design Pattern(s) addresses this issue. See the references below.

References

"Immutable Objects". article at http://www.javapractices.com/topic/TopicAction.do?Id=29

Wikipedia, *Immutable Objects*.

Horstmann, *Object-oriented Design and Patterns, 2E*.

Mikael Grev, "Mutable/Immutable Patterns". http://www.javalobby.org/articles/immutable/index.jsp