Object Cloning

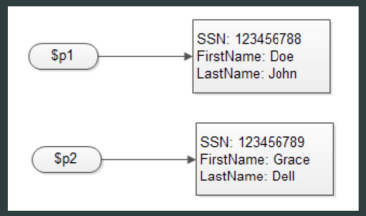
Object Cloning

- We've seen before that making a copy of an object variable just makes a second reference to that object, it doesn't "copy" it

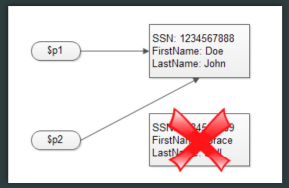
```
Person somePerson=new Person("Student1");
Person otherPerson;
otherPerson=somePerson;  //both otherPerson and somePerson point to the
                         //same object.  A change in either will occur in
                         //both.
```

- This just creates two references to the same exact thing. This isn't very useful.

Clone




```
$p2 = $p1;
```



An Example

In this example, both Date references point to the same Object. A change in one will effect both

```
Date someDate=new Date(1234567);
  Date newDate=someDate;
newDate.setTime(newDate.getTime()+1);
System.out.println(someDate.getTime());   //outputs 1234568
System.out.println(newDate.getTime());    //outputs 1234568
```

In this example, we create a full new object that starts in the same state. Now they are two unique objects

```
Date someDate=new Date(1234567);
  Date newDate=(Date)someDate.clone();
newDate.setTime(newDate.getTime()+1);
System.out.println(someDate.getTime());   //outputs 1234567
System.out.println(newDate.getTime());    //outputs 1234568
```

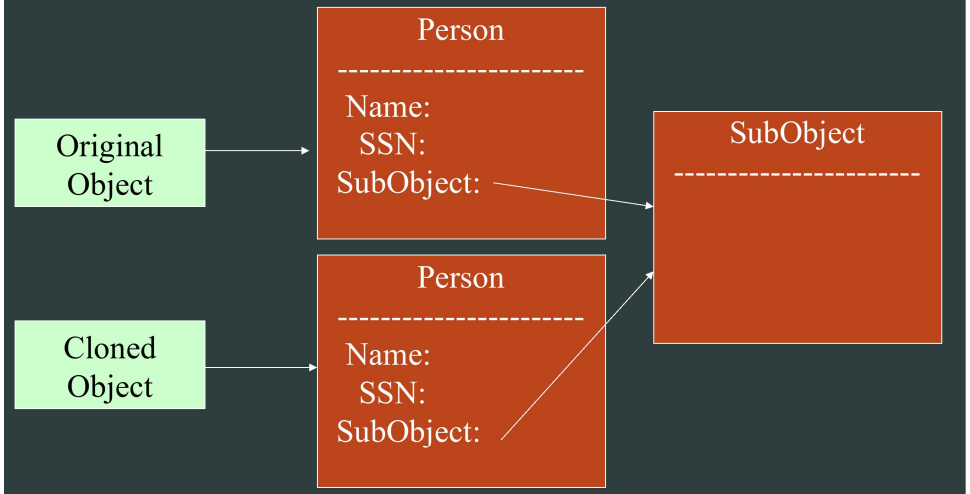
The Clone Method

- This works with Date objects because Date objects implement the Cloneable interface.
- The clone() method is actually inherited from the Object superclass, and is protected by default.
- Since clone() is already implemented in Object, it means that the Cloneable interface is a *tagged interface* – it actually contains no methods.
- Since clone() is inherited from Object, it doesn't know anything about it's subclasses (which is every class in Java).
- Is this a problem?

The Clone Method

- Since the clone() method belongs to the Object superclass, it doesn't know anything about the object it's copying.
- Because of this, it can only do a field-by-field copy of the Object.
- This is fine if all the fields are primitive types or immutable objects, but if the Object contained other mutable sub objects, only the reference will be copied, and the objects will share data.
- This is called shallow copying.

The Clone method



The Cloning Decision

- For every class, you need to decide whether or not
  - 1) The default shallow copy clone is acceptable
  - 2) You need to deep-copy the class
  - 3) The object is not allowed to be cloned
- For either of the first two choices, you must
  - Implement the Cloneable interface
  - Redefine the clone() method with the public access modifier

- The **object cloning** is a way to create exact copy of an object. The clone() method of Object class is used to clone an object.
- The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.
- The **clone() method** is defined in the Object class.

A Cloning Example

- To implement the default clone, you must create a clone() method with a public access modifier and add it to the class

```
public class cloneExample implements Cloneable {

  public Object clone() {
     try {
        return super.clone();
     }
     catch (CloneNotSupportedException exp)
      {  return null; }

      }
}
```

Deep Copy Vs Shallow Copy

- Deep Copy vs Shallow Copy
  - Shallow copy is the method of copying an object and is followed by default in cloning. In this method, the fields of an old object X are copied to the new object Y. While copying the object type field the reference is copied to Y i.e object Y will point to the same location as pointed out by X. If the field value is a primitive type it copies the value of the primitive type.
  - Therefore, any changes made in referenced objects in object X or Y will be reflected in other objects.

    Shallow copies are cheap and simple to make. In the above example, we created a shallow copy of the object.

Deep Copy Vs Shallow Copy

- Usage of clone() method – Deep Copy
  - If we want to create a deep copy of object X and place it in a new object Y then a new copy of any referenced objects fields are created and these references are placed in object Y. This means any changes made in referenced object fields in object X or Y will be reflected only in that object and not in the other. In the below example, we create a deep copy of the object.
  - A deep copy copies all fields and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.

```
class ABC
{
// instance variable of the class ABC
int x = 30;
}
public class ShallowCopyExample
{
// main method
public static void main(String argvs[])
{
// creating an object of the class ABC
ABC obj1 = new ABC();

// it will copy the reference, not value
ABC obj2 = obj1;

// updating the value to 6
// using the reference variable obj2
obj2.x = 6;

// printing the value of x using reference variable obj1
System.out.println("The value of x is: " + obj1.x);
}
}
```

```
The value of x is: 6
Press any key to continue . . .
```

```
class ABC
{
// instance variable of the class ABC
int x = 30;
}
public class DeepCopyExample
{
// main method
public static void main(String argvs[])
{
// creating an object of the class ABC
ABC obj1 = new ABC();

// it will copy the reference, not value
ABC obj2 = new ABC();

// updating the value to 6
// using the reference variable obj2
obj2.x = 6;

// printing the value of x using reference variable obj1
System.out.println("The value of x is: " + obj1.x);
}
}
```

```
The value of x is: 30
Press any key to continue . . .
```

Shallow vs Deep copying

It is fast as no new memory is allocated.
It is slow as new memory is allocated.

Changes in one entity is reflected in other entity.
Changes in one entity are not reflected in changes in another identity.

Cloned object and the original object are not disjoint.
Cloned object and the original object are disjoint.