Equals Method:

As explained during the first recitation, one of the methods defined in the `Object` class is `equals` method. Its signature is:

```
public boolean equals(Object o)
```

This method tests whether two objects are equal or not. The syntax for invoking it is:

```
object1.equals(object2);
```

the default implementation of the equals method in the `Object` class is:

```
public boolean equals(Object obj) {
	return (this == obj);
}
```

So, as you see, this implementation checks whether two reference variables point to the same object using the == operator. So, as explained in the recitation, you should override this method in your custom class to test whether two distinct objects have the *same content*.

So, here is a simple example:

Let's assume we want to define a class `MyDate` which has `year`, `month` and `day` as its data members (fields). We know that by default all the classes defined by user is (implicily) a subclass of class `Object` i.e., they all extend class `Object`. So, by inheritance, it has the mentioned `equals` method. Following is a simple definition for class `MyDate`:

```
class MyDate { // extends Object
	public MyDate(int year, int month, int day) {
		setDate(year, month, day);
	}
	public void setDate(int year, int month, int day) {
		setYear(year);
		setMonth(month);
		setDay(day);
	}
	public void setYear(int year) { this.year = year;}
	public void setMonth(int month) {this.month = month;}
	public void setDay(int day) {this.day = day;}
	public int getYear() {return this.year;}
	public int getMonth() {return this.month;}
	public int getDay() {return this.day;}
	private int year;
	private int month;
	private int day;
```

```
}
```

But, as mentioned, since we want to check the ***equality of contents*** of two distinct `MyDate` object (contents of `MyDate` object are indeed values of data members `year`, `month` and `day`), we should override `equals` method, as follows:

```
@override
public boolean equals(Object o) {
	if (o instanceof MyDate)
		return ((this.year == ((MyDate)o).year) &&
				(this.month == ((MyDate)o).month) &&
					(this.day == ((MyDate)o).day)
	else
		return false;
}
```

Followings are points you should know regarding this method:

- `@ovverride` annotation is used to ask Java Compiler to check if the method is overridden correctly. This annotation is optional and you might not provide it in your function. However, providing this is beneficial as Java Compiler will check to see if you are correctly overriding any method of any superclass of your class. For example, if you mistakenly put the wrong method name as `equal` (rather than `equals`), then Java Compiler will provide you an error that I can't find any method in any of superclasses of your class (here class `Object`) with this name that you are trying to override ! So, if you didn't provide this annotation and mistakenly put the name incorreclty, when your client code called method `equals`, then the default implementation of this method (which was incorrect and we tried to verride) would be called !!
- `instanceof` operator is used to check if the object passed to this method is of type `MyDate` or not.

Now, lets assume that we have class `Employee` which has `name` (of type `String`), `birthDate` and `hireDate` (both, of type `MyDate` that we defined previosly). Its class definition can be (assuming this class is defined in the same package as class `MyDate` has been defined):

```
public class Employee {
	public Employee (String name, MyDate birthDate, MyDate hireDate){
		setEmplInfo(name, birthDate, hireDate);
	}

	public void setEmplInfo(String name, MyDate birthDate, MyDate hireDate)
	{
		setName(name);
		setBirthDate(birthDate);
		setHireDate(hireDate);
```

```
	}

	public void setName (String name) { this.name = name;}
	public void setBirthDate(MyDate birthDate){this.birthDate = birthDate;}
	public void setHireDate(MyDate hireDate){this.hireDate = hireDate;}

	public String getName(){ return this.name;}
	public MyDate getBirthDate(){return this.birthDate;}
	public MyDate getHireDate(){return this.hireDate;}
}
```

For this class, lets implement equals class as its default implementation provided in `Object` class is incorrect.

First Try:

```
@override
public boolean equals(Object o) {
	if (o instanceof Employee)
		return ((this.name == ((Employee)o).name) &&
				(this.birthDate == ((Employee)o).birthDate) &&
					(this.hireDate == ((Employee)o).hireDate)
	else
		return false;
}
```

Is this implementaion correct?
The answer is no. Why? One typical answer can be:

- W3 W0U1DN'7 PROVID3 H34D3R "**First Try**" UNL355 7H3 4N5W3R W45 NO!! :D

But the valuable answer is:

- we know `==` operator just checks the address stored in the reference data members `name`, `birthDate` and `hireDate` (remember Java memory management discussion we had in recitaion?). So, we should call their `equals` methods rather than using `==` operator.

Second Try:

```
@override
public boolean equals(Object o) {
	if (o instanceof Employee)
		return ((this.name.equals(((Employee)o).name)) &&
				(this.birthDate.equals(((Employee)o).birthDate)) &&
					(this.hireDate.equals(((Employee)o).hireDate))
	else
```

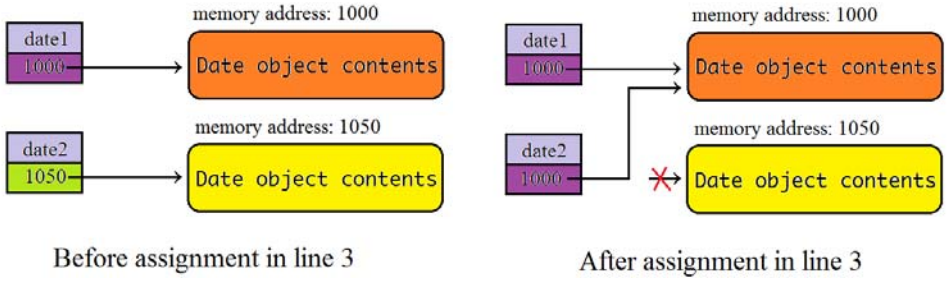
```
            return false;
}
```

Clone Method:

As discussed in class, (because of Java memory management,) simple assignment over the reference data members (or variables) just copies the *reference* of one object to another one. So, for class `Date` (defined in Java library), consider following exmample:

```
1     Date date1 = new Date();
2     Date date2 = new Date();
3     date2 = date1;
```

Here is a schematic explanation of what happens before and after the assignment in line 3:

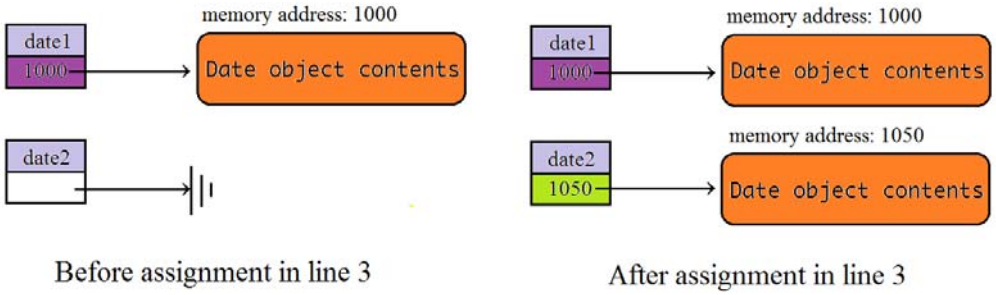


So, as you see, by the assignment in line 3, you didn't copy the contents of `date1` to `date2` !! Indeed you just copy the address of object, referenced (or pointed) by `date1` reference variable to the `date2` reference variable.

However, often it is desirable to create a copy of an object rather than just copying the reference address. Many classes in the Java library (e.g., `Date`, `Calendar`, and `ArrayList`) implement `clone` method to copy the object. So, see the following code:

```
1     Date date1 = new Date();
2     Date date2;
3     date2 = (Date) date1.clone();
```

Here is a schematic explanation of what happens before and after the assignment in line 3:



But regarding user defined classes, in order for your class to be cloned, your class needs to implement `Cloneable` interface and any class that implements this interface, MUST override the `clone` method in the `Object` class. So, your class must implement `clone` method also.

The `Cloneable` interface in the `java.lang` package is defined as follows:

```
package java.lang;

public interface Cloneable {
}
```

This interface is empty. An interface with an empty body is referred to as a *marker interface*. A marker interface doesn't contain constants or methods. It is used to denote that the class posses certain desirable properties. A class that implements the `Cloneable` interface is marked cloneable and its objects can be cloned using the `clone` method (as in `Date` example).

In order to override `clone` method correctly, lets first see what it does by the implementation provided in `Object` class. In class `Object`, what the defined method `clone` does is, it just makes a new object and copies each field (data member) from the original copy to the target copy.

So, if ALL data members defined in your class are of primitive types (e.g., `bool`, `int`, `double` etc) or immutable class (such as `String`), then, you can simply call the method defined in the class `Object`. So, your overridden method will be:

```
@override
public Object clone() throws CloneNotSupportedException {
	return super.clone();
}
```

As an example, if you check again `MyDate` class defined previously, you will see this class has three primitive type (`int`) data members: `year`, `month` and `day`. So, for this class, provided `clone` method can be used but we should inform Java compiler that this class implements `Cloneable` interface by changing the class header as follows:

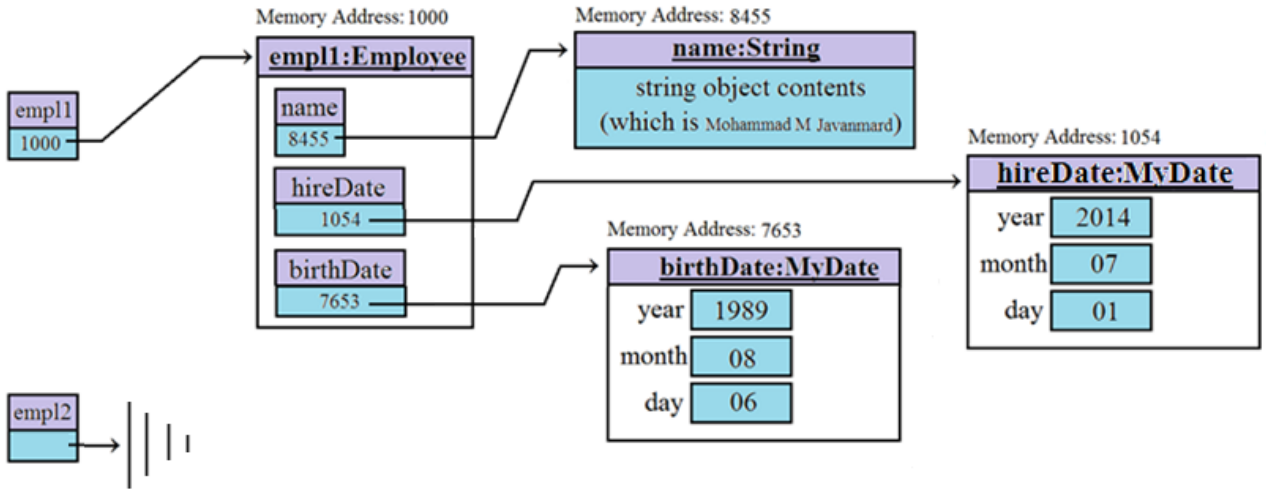
```
class MyDate implements Cloneable {

	// methods defined previously
	...

	// adding clone() method
	@override
	public Object clone() throws CloneNotSupportedException {
		return super.clone();
```

```
        }
}
```

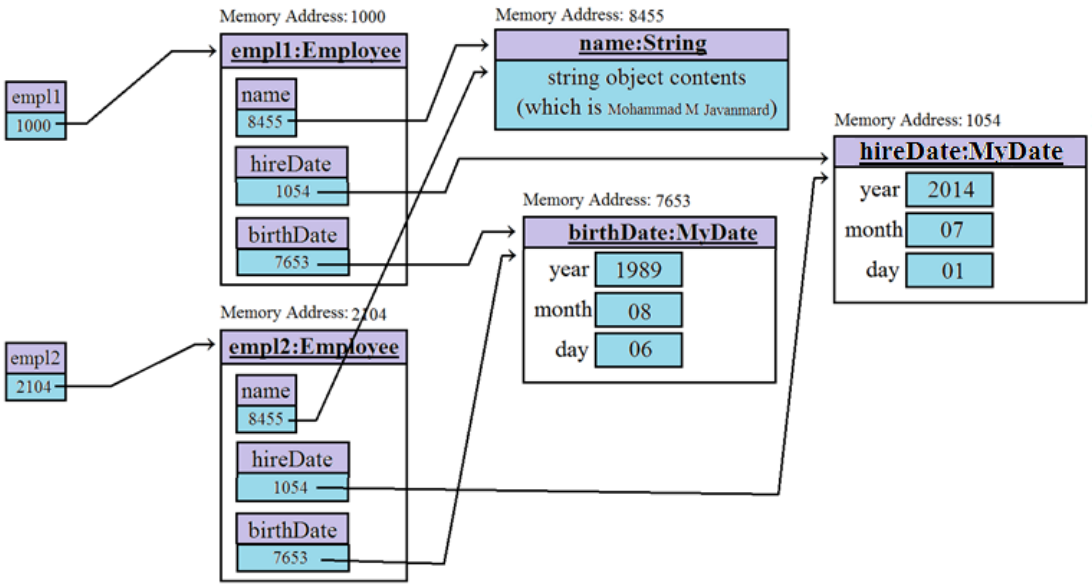
However, if you have at least one reference type data member (or field), then, this method will work incorrectly (as it will copy the address (or reference) of the data member rather than making new object and copying the content of it). As an example, consider class `Employee` that we defined previously. Inside this class, we have two problematic reference type data members: `birthDate` and `hireDate`. (please note that `name` object of type `String` is immutable and cloning it, doesn't make any problem as there is no setter method defined in this class). So, for this class, `clone` method provided (defined for class `MyDate`) will not work. For example, consider following code:

```
1     Employee empl1 = new Employee(new String("Mohammad M. Javanmard"),
                                    new Date(1989, 08, 06),
                                    new Date(2014, 07, 01));
2     Employee empl2;
3     empl2 = (Employee) empl1.clone();
```

See following figures to know why it doesn't work:
Before assignment in line 3:



And after assignment in line 3:



So, as you see because of simple copying, `empl2`'s reference type data members, point to the same objects pointed by data members of object `empl1`. This kind of cloning is called *shallow cloning* which is problematic if you have reference type data member.

So, in order to solve this problem, you need to override this method, as follows:

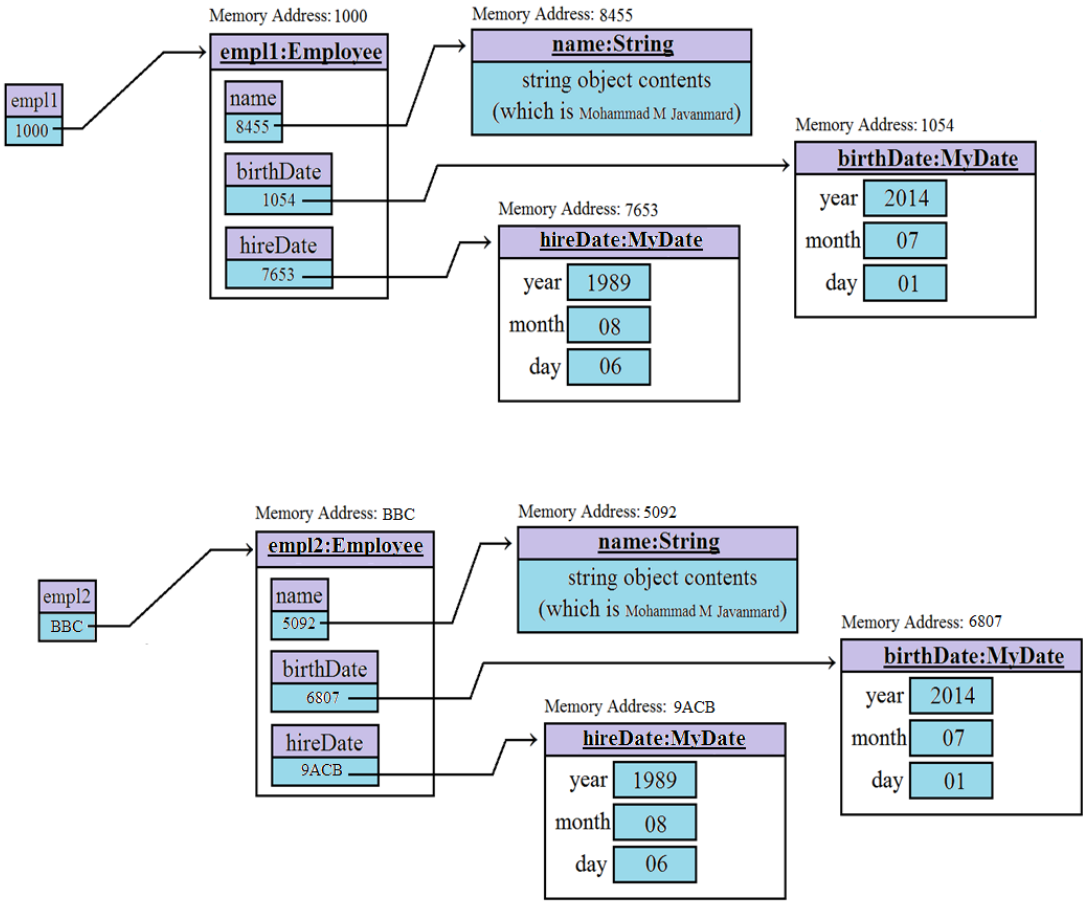
```
@override
public Object clone() throws CloneNotSupportedException {
	Employee employeeClone = (Employee) super.clone();
	employeeClone.birthDate = (MyDate)(birthDate.clone());
	employeeClone.hireDate = (MyDate)(hireDate.clone());
	return employeeClone;
}
```

or

```
@override
public Object clone() {
	try {
		Employee employeeClone = (Employee) super.clone();
		employeeClone.birthDate = (MyDate)(birthDate.clone());
		employeeClone.hireDate = (MyDate)(hireDate.clone());
		return employeeClone;
	}catch(CloneNotSupportedException ex) {
		return null;
	}
}
```

After having such an overridden method in the `Employee` class, we have following figure for the following lines of code:

```
1     Employee empl1 = new Employee(new String("Mohammad M. Javanmard"),
                                    new Date(1989, 08, 06),
                                    new Date(2014, 07, 01));
2     Employee empl2;
3     empl2 = (Employee) empl1.clone();
```



Reference

[1] Introduction to Java Programming, Ninth Edition written by Y. D. Liang.