Object copy, equals, compare, and misc.

Yulia Newton, Ph.D.

CS151, Object Oriented Programming

San Jose State University

Spring 2020

Agenda

- Deep vs. shallow copy
- Object equals
- Object compare and Comparable interface
- Enum data structure

Deep vs. shallow vs. lazy copying in Java

- Copying an object is creating a complete copy of the current object
- Two ways to create a copy of an object
  - Copy constructor
  - Cloning
- Deep, shallow, and lazy terms refer to cloning
  - Objects that implement *Cloneable* interface
  - Lazy copy is a combination of deep and shallow copy

Example: copying objects using copy constructor

```
class Dog{
    private String name;
    private String breed;
    private String color;
    private int age;

    Dog(String name, String breed, String color, int age){
        this.name = name;
        this.breed = breed;
        this.color = color;
        this.age = age;
    }

    Dog(Dog d){
        this.name = d.name;
        this.breed = d.breed;
        this.color = d.color;
        this.age = d.age;
    }

    public String getName(){return this.name;}
    public String getBreed(){return this.breed;}
    public String getColor(){return this.color;}
    public int getAge(){return this.age;}

    public void setName(String name){this.name = name;}
    public void setBreed(String breed){this.breed = breed;}
    public void setColor(String color){this.color = color;}
    public void setAge(int age){this.age = age;}

    public String toString(){
        return "I am " + this.name + ", a " + this.color + " "
                + this.breed + ", " + this.age + " years old";
    }
}
```

```
public class Test {
    public static void main(String args[]){
        Dog d1 = new Dog("Sparky", "Pug", "brown", 2);
        Dog d2 = new Dog(d1);
        System.out.println(d2.toString());
    }
}
```

Output:

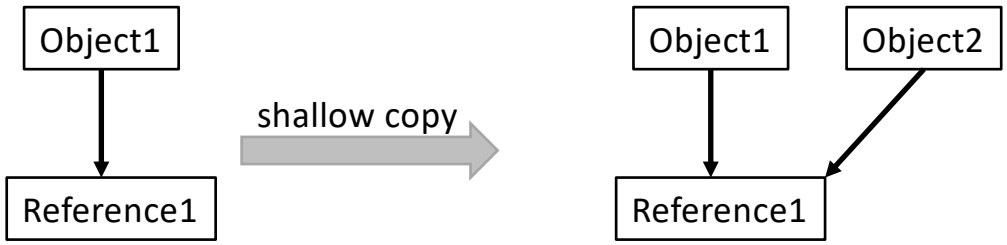
```
I am Sparky, a brown Pug, 2 years old
```

Cloning in Java

- Cloning – creating an exact copy of an object
- We clone using *clone()* method of *java.lang.Object* class
  - Creates a field-by-field copy and returns reference to that object
  - The object must implement *Cloneable* interface in order to be able to clone
    - Remember *Cloneable* marker interface?
    - Marks an object as eligible for cloning
- By default *clone()* method creates a shallow copy of an object
- To create a deep copy the class has to override *clone()* method
- Method *clone()* is has protected access modifier in *Object* class, forcing to override it as it cannot be called from outside of *java.lang* package

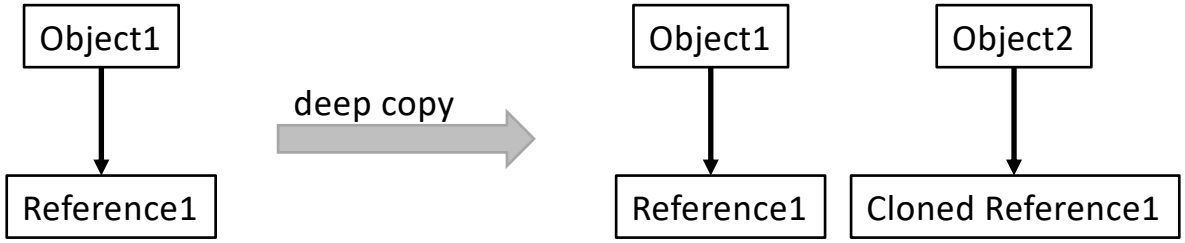
Shallow copy

- Shallow copying is performed whenever we use the default implementation of the clone() method
- In shallow copying non-primitive types in the cloned object reference the same memory location as the original object
  - References do not get replaced



Deep copy

- Copies everything, including creating new references
- Uses *clone()* method but it needs to be overridden by the class that is being cloned
  - All members of the class also need to implement *Cloneable* interface
  - Each member needs to override *clone()* method



```java
class Company{
    private String name;
    private String hqAddress;

    Company(String name, String address){
        this.name = name;
        this.hqAddress = address;
    }

    public String getName(){return this.name;}
    public String getAddress(){return this.hqAddress;}
    public void setName(String name){this.name = name;}
    public void setAddress(String address){this.hqAddress = address;}

}

class Employee implements Cloneable{
    private String name;
    private int id;
    private Company employer;

    Employee(String name, int id, Company employer){
        this.name = name;
        this.id = id;
        this.employer = employer;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public Company getEmployer(){return this.employer;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setEmployer(String name, String address){
        this.employer.setName(name);
        this.employer.setAddress(address);
    }

    protected Object clone() throws CloneNotSupportedException
    {
        // use default implementation of clone() in Object superclass
        return super.clone();
    }
}
```

Example: shallow copy

```java
public class Test {
    public static void main(String args[]){
        Company company1 = new Company("Company1", "123 Hitech Wy, My City CA 99999");
        Employee employee1 = new Employee("John Smith", 100, company1);
        Employee employee2 = null;

        try{
            employee2 = (Employee) employee1.clone();
        }catch (CloneNotSupportedException e)
        {
            e.printStackTrace();
        }

        System.out.println(employee1.getEmployer().getName()+
            ": "+employee1.getEmployer().getAddress());
        employee2.setEmployer("Company2", "789 Pacific Wy, Another City NY 11111");
        System.out.println(employee1.getEmployer().getName()+
            ": "+employee1.getEmployer().getAddress());

    }
}
```

we change company information in employee2
but can see the changes in employee1

Output:

```
Company1: 123 Hitech Wy, My City CA 99999
Company2: 789 Pacific Wy, Another City NY 11111
```

```java
class Company implements Cloneable{
    private String name;
    private String hqAddress;

    Company(String name, String address){
        this.name = name;
        this.hqAddress = address;
    }

    public String getName(){return this.name;}
    public String getAddress(){return this.hqAddress;}
    public void setName(String name){this.name = name;}
    public void setAddress(String address){this.hqAddress = address;}

    protected Object clone() throws CloneNotSupportedException{
        return super.clone();
    }
}

class Employee implements Cloneable{
    private String name;
    private int id;
    private Company employer;

    Employee(String name, int id, Company employer){
        this.name = name;
        this.id = id;
        this.employer = employer;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public Company getEmployer(){return this.employer;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setEmployer(String name, String address){
        this.employer.setName(name);
        this.employer.setAddress(address);
    }

    protected Object clone() throws CloneNotSupportedException{
        Employee employee = (Employee) super.clone();
        employee.employer = (Company) employer.clone();
        return employee;
    }
}
```

now Company is Cloneable too

shallow copy

clone all references

Example: deep copy

```java
public class Test {
    public static void main(String args[]){
        Company company1 = new Company("Company1", "123 Hitech Wy, My City CA 99999");
        Employee employee1 = new Employee("John Smith", 100, company1);
        Employee employee2 = null;

        try{
            employee2 = (Employee) employee1.clone();
        }catch (CloneNotSupportedException e)
        {
            e.printStackTrace();
        }

        System.out.println(employee1.getEmployer().getName()+
            ": "+employee1.getEmployer().getAddress());
        employee2.setEmployer("Company2", "789 Pacific Wy, Another City NY 11111");
        System.out.println(employee1.getEmployer().getName()+
            ": "+employee1.getEmployer().getAddress());
        System.out.println(employee2.getEmployer().getName()+
            ": "+employee2.getEmployer().getAddress());
    }
}
```

Output:

```
Company1: 123 Hitech Wy, My City CA 99999
Company1: 123 Hitech Wy, My City CA 99999
Company2: 789 Pacific Wy, Another City NY 11111
```

Deep vs. shallow copy

Deep copy	Shallow copy
Cloned and original objects are completely disjointed	Cloned and original objects are not completely disjointed (non-primitive members)
Changes made to the cloned object **do not** affect the original object	Changes made to the cloned object affect the original object
Must override the default implementation of *clone()* method	Default implementation of the Object's *clone()* method
Should implement if have non-primitive type class members	OK to use if class has only primitive type members
Slower	Faster

Object equality

- When using comparison operator (==) Java checks if two variables reference the same object in memory
  - All variables in Java are references
- What does it mean for two objects to be equal?
  - E.g. are two employees equal?

Example: motivation for *equals()* method

```java
class MyClass {
    private String myAttribute;

    MyClass(String attribute) {
        this.myAttribute = attribute;
    }
}

public class Test {
    public static void main(String[] args){
        MyClass c1 = new MyClass("Test");
        MyClass c2 = new MyClass("Test");
        MyClass c3 = c2;

        if(c1 == c2){
            System.out.println("c1 and c2 are equal");
        }else{
           System.out.println("c1 and c2 are NOT equal");
        }

        if(c3 == c2){
            System.out.println("c3 and c2 are equal");
        }else{
           System.out.println("c3 and c2 are NOT equal");
        }
    }
}
```


c1 and c2 are seemingly the same

Output:

```
c1 and c2 are NOT equal
c3 and c2 are equal
```

Overriding *equals()* method

- Should override *Object* class' method *equals()* to test equality
  - boolean equals(Object o)
  - You get to define what it means for two objects to be equal
    - Are two employees with the same name but different IDs different?
- Tips for implementing *equals()* method
  - Test if input argument object is null, if so return *false*
  - Test for reference equality, if equals return *true*
  - Test if the same class (*getClass()* method), if different return *false*
  - Finally test attribute equality

Object class implementation of equals() method:

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

Example: compare with *equals()* method

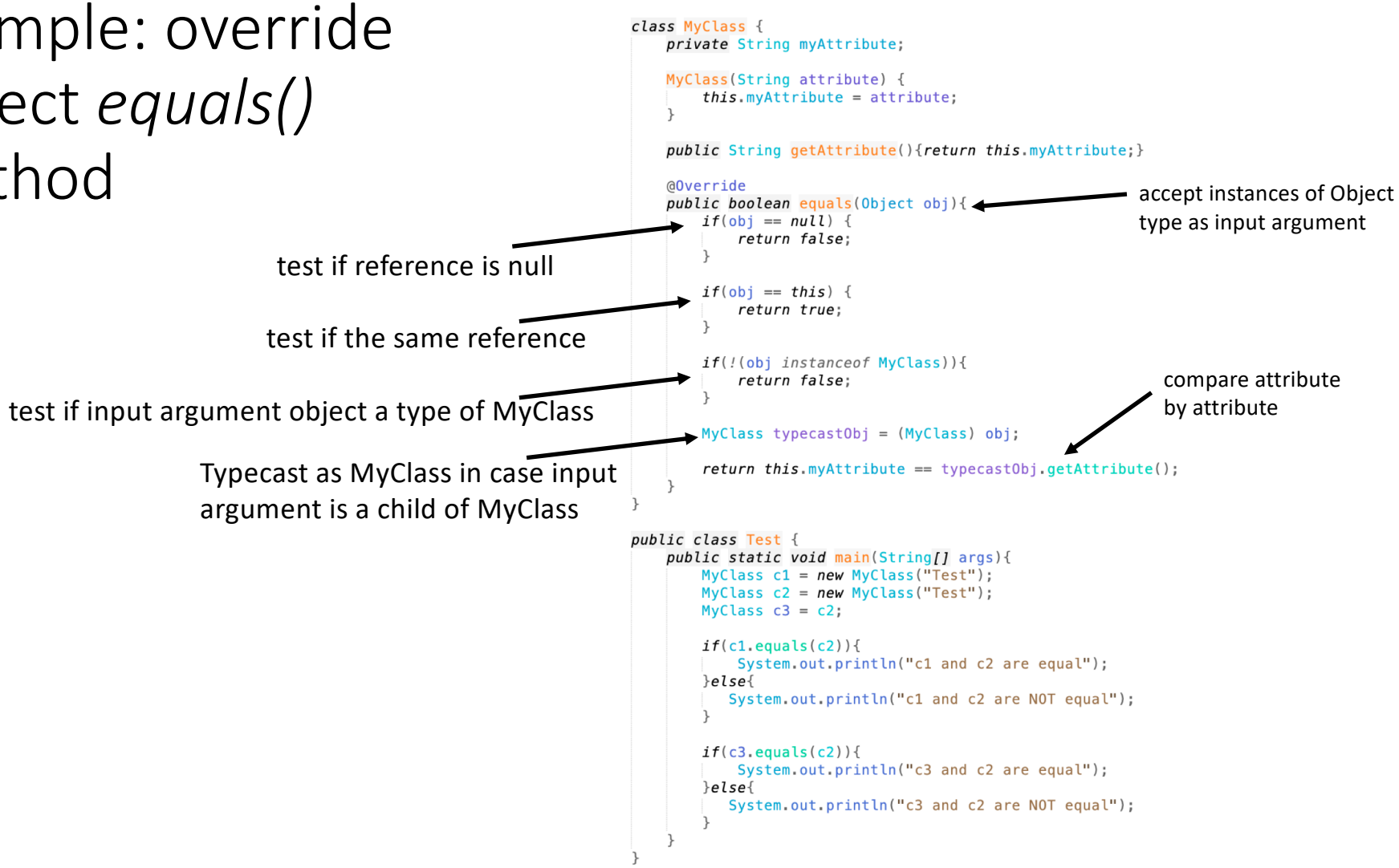
```
class MyClass {
    private String myAttribute;

    MyClass(String attribute) {
        this.myAttribute = attribute;
    }

    public String getAttribute(){return this.myAttribute;}

    public boolean equals(MyClass obj){
        if (obj == this) {
            return true;
        }

        return this.myAttribute == obj.getAttribute();
    }
}

public class Test {
    public static void main(String[] args){
        MyClass c1 = new MyClass("Test");
        MyClass c2 = new MyClass("Test");
        MyClass c3 = c2;

        if(c1.equals(c2)){
            System.out.println("c1 and c2 are equal");
        }else{
           System.out.println("c1 and c2 are NOT equal");
        }

        if(c3.equals(c2)){
            System.out.println("c3 and c2 are equal");
        }else{
           System.out.println("c3 and c2 are NOT equal");
        }
    }
}
```

Output:

```
c1 and c2 are equal
c3 and c2 are equal
```

We implement equals method, however we do not override *Object* class' *equals()* method because we expect MyClass as an input argument. If we try to pass in any other type we will get a compile-time error.

Example: override Object *equals()* method




```
class MyClass {
    private String myAttribute;

    MyClass(String attribute) {
        this.myAttribute = attribute;
    }

    public String getAttribute(){return this.myAttribute;}

    @Override
    public boolean equals(Object obj){
        if(obj == null) {
            return false;
        }

        if(obj == this) {
            return true;
        }

        if(!(obj instanceof MyClass)){
            return false;
        }

        MyClass typecastObj = (MyClass) obj;

        return this.myAttribute == typecastObj.getAttribute();
    }
}

public class Test {
    public static void main(String[] args){
        MyClass c1 = new MyClass("Test");
        MyClass c2 = new MyClass("Test");
        MyClass c3 = c2;

        if(c1.equals(c2)){
            System.out.println("c1 and c2 are equal");
        }else{
           System.out.println("c1 and c2 are NOT equal");
        }

        if(c3.equals(c2)){
            System.out.println("c3 and c2 are equal");
        }else{
           System.out.println("c3 and c2 are NOT equal");
        }
    }
}
```

Overriding *hashCode()* method

- Whenever we override *equals()* it is recommended that we override *hashCode()* as well
- Returns the integer hash code value of the object
  - A way to represent a large object with a compact summary
- Consistency
  - Multiple calls return the same integer value
- Hash code can change between executions or if the object attributes change
- If two objects are equal then they return the same hash code
  - If two objects are not equal they can still have the same hash code
- *hashCode()* should use the same attributes that are used in *equals()* method
- A lot of IDEs provide a way to automatically generate *equals()* and *hashCode()* methods

General rules for computing hash code

- Start with an initial hash, non-zero integer value
- For each attribute used by the *equals()* method compute that field's hash code
  - Primitive data types
    - For byte, char, short or int fields the hash is just the value of the field
    - For boolean fields can do:
      - boolean_hash = (x ? 0 : 1)
    - For double fields can do:
      - Double.doubleToLongBits(x)
  - If you are dealing with an object, use that object's class *hashCode()* method if available
    - E.g. *String.hashCode()*, *Double.hashCode()*, *Integer.hashCode()*, etc.
  - Check if the object reference is null
    - x == null ? 0 : x.hashCode()
  - If you feel really stuck, consider something like this:

```
String hashString = var1.toString() + var2.toString();
hashCodeFromString = hashString.hashCode();
```

Example: overriding *hashCode()* for MyClass

```java
class MyClass {
    private String myAttribute;

    MyClass(String attribute) {
        this.myAttribute = attribute;
    }

    public String getAttribute(){return this.myAttribute;}

    @Override
    public boolean equals(Object obj){
        if(obj == null) {
            return false;
        }

        if(obj == this) {
            return true;
        }

        if(!(obj instanceof MyClass)){
            return false;
        }

        MyClass typecastObj = (MyClass) obj;

        return this.myAttribute == typecastObj.getAttribute();
    }

    @Override
    public int hashCode(){
        return 42 + this.myAttribute.hashCode();
    }
}
```

```java
public class Test {
    public static void main(String[] args){
        MyClass c1 = new MyClass("Test");
        MyClass c2 = new MyClass("Test");
        MyClass c3 = c2;
        MyClass c4 = new MyClass("Different string");

        System.out.println("c1 hash code: "+c1.hashCode());
        System.out.println("c2 hash code: "+c2.hashCode());
        System.out.println("c3 hash code: "+c3.hashCode());
        System.out.println("c4 hash code: "+c4.hashCode());
    }
}
```

Output:

```
c1 hash code: 2603228
c2 hash code: 2603228
c3 hash code: 2603228
c4 hash code: -1287341534
```

Comparing two objects using *compareTo()* method

- Alternative way to compare objects is using *compareTo()* method
  - Overridden in the class whose object types we wish to compare
  - Class has to implement *Comparable* interface
    - *java.lang.Comparable*
- Collections framework heavily utilizes *compareTo()* for ordering elements
  - E.g. sorting objects of Animal type based on their size

compareTo() method

- Accepts an object and returns an integer value indicating how current object and the input argument object compare
  - *public int compareTo(Object obj)*
- Returns 0 if two objects are equal
- Returns >0 if the current object is greater than input argument object
- Returns <0 if the current object is less than input argument object
- The rules of comparing are up to implementation of *compareTo()*
  - E.g. compare employees based on their age
    - Ascending vs. descending order
  - E.g. compare strings based on their length

Example: override *compareTo()* for MyClass

Version 1:

```
class MyClass implements Comparable{
    private String myAttribute;

    MyClass(String attribute) {
        this.myAttribute = attribute;
    }

    public String getAttribute(){return this.myAttribute;}

    @Override
    public int compareTo(Object obj)
    {
        MyClass myObj = (MyClass)obj;
        return this.myAttribute.compareTo(myObj.getAttribute());
    }
}

public class Test {
    public static void main(String[] args){
        MyClass c1 = new MyClass("Test");
        MyClass c2 = new MyClass("Different string");

        System.out.println("c1 compare to c2: "+c1.compareTo(c2));
    }
}
```

uses build-in *String* class *compareTo()*

Output:

```
c1 compare to c2: 16
```

Version 2 (preferred way to implement *compareTo()*):

```
class MyClass implements Comparable<MyClass>{
    private String myAttribute;

    MyClass(String attribute) {
        this.myAttribute = attribute;
    }

    public String getAttribute(){return this.myAttribute;}

    @Override
    public int compareTo(MyClass obj)
    {
        return this.myAttribute.compareTo(obj.getAttribute());
    }
}

public class Test {
    public static void main(String[] args){
        MyClass c1 = new MyClass("Test");
        MyClass c2 = new MyClass("Different string");

        System.out.println("c1 compare to c2: "+c1.compareTo(c2));
    }
}
```

restrict comparisons to MyClass type objects

String *compareTo()* method

- Provided by built-in *String* class
- Compares based on lexicographical order of the string values converted to Unicode
  - Returns 0 if two strings are lexicographically equal
  - Returns >0 of this string is lexicographically bigger
  - Returns <0 of this string is lexicographically less
- Case sensitive
- When comparing to an empty string then the length of the non-empty string is returned with the appropriate sign

Example: compare strings without taking case into account

```java
class MyClass implements Comparable<MyClass>{
    private String myAttribute;

    MyClass(String attribute) {
        this.myAttribute = attribute;
    }

    public String getAttribute(){return this.myAttribute;}

    @Override
    public int compareTo(MyClass obj)
    {
        return this.myAttribute.compareTo(obj.getAttribute());
        //return this.myAttribute.compareToIgnoreCase(obj.getAttribute());
    }
}

public class Test {
    public static void main(String[] args){
        MyClass c1 = new MyClass("Hello");
        MyClass c2 = new MyClass("hello");

        System.out.println("c1 compare to c2: "+c1.compareTo(c2));
    }
}
```

```
c1 compare to c2: -32
```

```java
class MyClass implements Comparable<MyClass>{
    private String myAttribute;

    MyClass(String attribute) {
        this.myAttribute = attribute;
    }

    public String getAttribute(){return this.myAttribute;}

    @Override
    public int compareTo(MyClass obj)
    {
        //return this.myAttribute.compareTo(obj.getAttribute());
        return this.myAttribute.compareToIgnoreCase(obj.getAttribute());
    }
}

public class Test {
    public static void main(String[] args){
        MyClass c1 = new MyClass("Hello");
        MyClass c2 = new MyClass("hello");

        System.out.println("c1 compare to c2: "+c1.compareTo(c2));
    }
}
```

```
c1 compare to c2: 0
```

Example: comparing based on the length of String object

```
class MyClass implements Comparable<MyClass>{
    private String myAttribute;

    MyClass(String attribute) {
        this.myAttribute = attribute;
    }

    public String getAttribute(){return this.myAttribute;}

    @Override
    public int compareTo(MyClass obj)
    {
        return this.myAttribute.length() - obj.getAttribute().length();
    }
}

public class Test {
    public static void main(String[] args){
        MyClass c1 = new MyClass("Test");
        MyClass c2 = new MyClass("Different string");

        System.out.println("c1 compare to c2: "+c1.compareTo(c2));
    }
}
```

Output:

```
c1 compare to c2: -12
```

compare two string attributes based on the length of the string

if want the shorter string to be greater than switch operands:

```
return obj.getAttribute().length() - this.myAttribute.length();
```

Enums

- Useful data structures for enumeration
- Represent groups of related named constants
- Internally implemented by using class objects
- Can use enums to represent such things as days of the week, gender, RGB colors,

```
enum Gender
{
    MALE, FEMALE, NONBINARY, UNSPECIFIED;
}
```

Internally converted at compile time to:

```
class Gender
{
    public static final Gender MALE = new Gender();
    public static final Gender FEMALE = new Gender();
    public static final Gender NONBINARY = new Gender();
    public static final Gender UNSPECIFIED = new Gender();
}
```

Example: working with enums

```java
enum Gender
{
    MALE, FEMALE, NONBINARY, UNSPECIFIED;
}

class Employee{
    private String name;
    private int id;
    private Gender gender;

    Employee(){
        this.name = "Uknown name";
        this.id = 0;
        this.gender = Gender.UNSPECIFIED;
    }

    Employee(String name, int id, Gender gender){
        this.name = name;
        this.id = id;
        this.gender = gender;
    }

    @Override
    public String toString(){
        return this.name+" ("+this.id+") is a "+this.gender;
    }
}
```

```java
public class Test{
    public static void main(String[] args){
        Employee e1 = new Employee();
        System.out.println(e1.toString());

        Employee e2 = new Employee("John Smith", 101, Gender.MALE);
        System.out.println(e2.toString());
    }
}
```

Output:

```
Uknown name (0) is a UNSPECIFIED
John Smith (101) is a MALE
```

Concluding remarks

- In this unit we learned
  - Differences between object deep and shallow copy
    - Overriding *clone()* method
  - Object equality test
    - Overriding *equals()* method
  - Object compare
    - Overriding *compareTo()* method
  - Enums data structure
- Remember that all variables in Java are references to a memory location