Review:

Every class inherits

- boolean equals(Object o)
- String toString()

from Object

To be of any use we should override these classes:

Example:

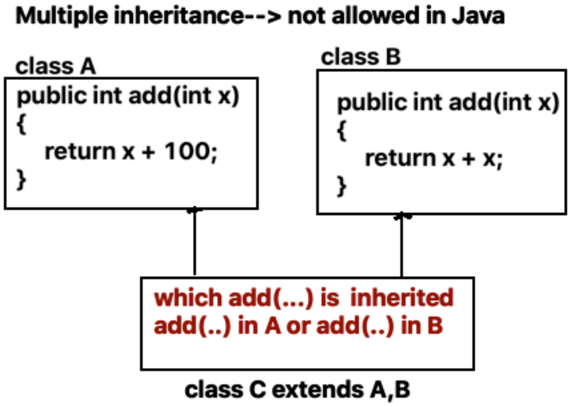
```
public class Circle()
{
    private double radius ;

     // constructors, getters and setters go here

      public double area()
     {
          return 3.14159 * radius*radius;
      }

     // two circles are equal if they have the same area
     public boolean equals(Object o)
    {
         return this.area() == ((Circle)o).area();  // note the downcast and the parentheses
     }
     public  String toString()
     {
           Return “Radius: “+ radius+ “ Area: “ + area();
    }
}

public class ShowCircle
{
     public static void main(String[] args)
     {
          Circle a = new Circle(1); // radius is 1
           Circle b = new Circle(1) ;
           System.out.println (a.equals(b) );
           System.out.println(a);  // calls a.toString()
}
```

Output:

```
true
Radius: 1 Area: 3.14159
```

Multiple Inheritance

Unlike some languages such as C++ a Java class can extend only one class. “Multiple inheritance can lead to ambiguity.


Multiple inheritance--> not allowed in Java
class A
class B
which add(...) is inherited
add(..) in A or add(..) in B
class C extends A,B

```
public int add(int x)
{
    return x + 100;
}
```

```
public int add(int x)
{
    return x + x;
}
```

As an alternative approach Java provides ***Interfaces***

An ***interface*** is a named collection of abstract methods and static constants.

Example,

```
        public interface Geometry  // not a class
        {
                public static final PI = 3.14159;
                 public abstract double area();
                public abstract double perimeter();
         }
```

Note: With an interface, you can omit the word *abstract*. All methods in an interface are assumed to be abstract. So Geometry could also be written as:

```
         public interface Geometry  // not a class
        {
                public static final PI = 3.14159;
                 public double area();
                public double perimeter();
         }
```

This is saved as Geometry.java

A class does not *extend* an interface; a class ***implements*** an interface. A class that implements an interface **must implement all the abstract classes in the interface**
An interface is like an abstract class without data and without fully implemented methods.

Example:

```
public class Square implements Geometry
{
     private double  side;
     // constructors go here
     public double area()
     {
         return side * side;
      }
      public  double perimeter()
      {
            return 4*side;
       }
}
 // notice Square implements the abstract
// methods of Geometry .
```

```
public class Circle implements Geometry
{
     private double radius;
     // constructors go here
     public double area()
     {
         return PI*radius*radius;
      }
      public  double perimeter()
      {
            return 2*PI*radius;
       }
}
// notice Circle implements the abstract
// methods of Geometry
```

1. You can upcast to an interface, just as you can upcast to a parent class. However. Like an abstract class you cannot instantiate/create an object from an interface.

   ```
   public class Figures
   {
                Geometry[] shapes = new Geometry[2];  // create array of references
                 shapes[0] =new Circle(2.0);  // upcast
                 shapes[1] = new Square(4.0);
                 System.out.println(shapes[1].area());
   }
   ```

   Notice the **declared** type of shapes is **Geometry**. So, when compiling the last line, the compiler looks at the interface Geometry, sees an area() method and is happy. No downcast is necessary.

   Even though area() is abstract, the compiler know any class that implements Geometry MUST have area(). ☺

2. A class can extend only one class (multiple inheritance is not allowed) but can implement any number of interfaces, since all the methods of an interface are abstract.

```
public interface Eat
{
   public int mealsPerDay();
   public String  favoriteFood();
}

// there are abstract methods
```

```
public interface Sound
{
     public void speak();
}
```

```
public class Animal
{
   protected String name;
   protected int weight;
   public Animal()
   {
      name = “”;
      weight = 0;
    }
    public Animal (String n, int w)
    {
        name = n;
        weight = w;
     }
     // getters and setters go here
}
```

```
public class cat extends Animal implements Eat, Sound  // implements two interfaces
{

   public Cat()
   {
       super();  // calls default  constructor of Animal
   }
   public Cat(String n, int w)
   {
       super(n, w); // calls the 2 -argument  constructor of Animal
    }
    public void speak()         // must implement all methods of the interface Sound
    {
       System.out.println(“My name is  “+ name +” meow, meow”);
    }
    public int mealsPerDay()      // must implement all methods of Eat
    {
        return 2*weight%25;
    }
    public String favoriteFood()  // must implement all methods of Eat
    {
       return “Mice”;
     }
}
```

```
Public class TestCat
{
   Public static void main(String[] args)
    {
         Cat a = new Cat(“Felix”, 10);           // declared type is Cat
         Eat  b = new Cat(“Tiger”, 8);          // upcast to Eat declared type is Eat
         Sound c = new Cat(“Tubby”, 20);  // upcast to Sound; declared type is Sound
         a.speak();
         ((Cat)b).speak();             // downcast needed, declared type of b is Eat – also  ((Sound)b).speak() is OK
         c.speak();                                          // no downcast needed  declared type of c is Sound
         System.out.println(((Cat)c).favoriteFood());     // could  also say ((Eat)c).favoriteFood()
}
```

The Comparable Interface

Suppose you have two Strings

```
String s1 = “ABC”
String s2 = “XYZ”
```

The String class overrides

boolean equals(Object o)

which is inherited from Object

So s1.equals(s2) returns false.

You can also compare two String objects using compareTo(…)

- s1.compareTo(s2) returns a negative number // “s1 <s2”
- s2.compareTo(s1) returns a positive number //” s2 > s1”
- s1.compareTo(s1) returns 0 // equality

Side note: The comparisons are made by comparing ASCII values so

(“ABC”).compareTo(“abc”) returns a negative number

Because ‘A’ has value 65 and ‘a’ has value 97.

As we add equals(Object o) to each of our classes, we can also add compareTo(…) to our classes to compare objects.

Java provides an interface called Comparable. The Comparable interface has just one abstract method.

public int compareTo(**Object** o)

We can implement this interface in any of our classes. We usually implement compareTo(..) as

- a.compareTo(b) returns negative (-1) if “a is less than b”
- a.compareTo(b) returns positive (1) if “a is less greatern than b”
- a.compareTo(b) returns 0 if “a equals b”

Example:

```
public class Leopard extends Cat implements Comparable
{
 private int numSpots = 0;

 public boolean equals(Object o)  // overrides equals(Object o) from Object
 {
    return this.numSpots == ((Leopard)o).numSpots;
 }

// implements compareTo(Object o) from the Comparable interface
public int compareTo(Object o)
 {
    if (this.numSpots < ((Leopard)o).numSpots)
      return -1;
    else if (this.numSpots > ((Leopard)o).numSpots)
      return 1;
    else
      return 0;
  }
 }
```

Now we can compare Leopards based on the number of spots a Leopard has:

```
Leopard sam = new Leopard(300, 50); // weight 300, 50 spots
Leopard slim =  new Leopard (200, 80); // weight 200, 80 spots


if (sam.compareTo(slim))< 0                 // sam < slim
        System.out.println("Sam has fewer spots");
ese if ( sam.compareTo(slim)> 0)            // sam> slim
        System .out.println("Sam has more spots");
else
        System .out.println("Sam and Slim have the same number of spots");
```

Example

```
public class Box implements Comparable
{
        int length, width, height;
         public Box()
        {
                 length = width = height = 0;
        }
         public Box( int l, int w, int h)
        {
                length = l;
                width = w;
                height = h;
        }
         public int volume()
        {
                return length*width*height;
        }
        public int area()
        {
                return 2*length*width + 2*length*height + 2*width*height;
        }

         public int compareTo(Object o)      // based on volume
        {
                if (volume() < ((Box)o).volume())
                        return -1;
                if (volume() > ((Box)o).volume())
                        return 1;
                 return 0;
         }

         public String toString()  // inherited from Object and overridden
        {
                return "Length: "+ length+ "\t\tWidth: "+ width + "\t\tHeight: "+ height;
        }

        public boolean equals(Object o) // inherited from Object and overridden
         {
                return volume() == ((Box)o).volume();
        }
```

```
     public static void main(String[] args)  // demonstrates Box objects
    {

         Box box1 = new Box(3,4,5);
          Box box2 = new Box(6,3,3);
          System.out.println("The volumes are "+box1.volume() +  "  " + box2.volume());
          System.out.println("box1.compareTo(box2): "+box1.compareTo(box2));
          System.out.println("box2.compareTo(box1): "+box2.compareTo(box1));
         System.out.println("box2.compareTo(box1): "+box1.compareTo(box1));
   }
}
```

.

Output:

```
    The volumes are 60  54
    box1.compareTo(box2): 1
    box2.compareTo(box1): -1
    box2.compareTo(box1): 0
```

>

We can now write a sort method that will sort an array of any kind of objects as long as the objects are comparable, i.e. implement compareTo(..)

On the left is a version of selection sort that sorts an array of int.

On the right is a version that sorts an array of object that implement Comparable

Here is c class SelectionSort with one static method,

void sort(int[] x, int size)

that can **sort an array of int.**

Because sort(…) is static, you call it with the class name

SelectionSort.sort(x, 100)

But this method **sorts integer arrays only**

```
public class SelectionSort
{
 public static void sort(int [] x, int size)
 {
    int max            //  the data stored in x[]
    int maxIndex;   // an index is an int
    for (int i=size-1; i>=1; i--)
    {
     // Find the maximum in the x[0..i]
       max = x[i];      // the "current"  maximum is x[i]
       maxIndex = i;   // index of  "current" max

         for (int j=i-1; j>=0; j--)
         {
            if (max < x[i])
            {
               max = x[j];   // a "new" maximum
               maxIndex = j;
            }
         }
         if (maxIndex != i)
         // place the maximum in its proper position
         {
            x[maxIndex] = x[i];
            x[i] = max;
         }
      }
   }
}
```

Here is a class SelectionSort with one static method,

void sort(Comparable[] x, int size)

that can sort an array of any object whose class implements the Comparable interface, i.e. implements

int compareTo(Object o)

Again, because sort(…) is static you call it with the class name

SelectionSort.sort(x, 100)

But it sorts arrays of Boxes, Leopards, or whatever, as long as the class implements Comparable

```
public class SelectionSort
{
   public static void sort(Comparable[] x, int size)
   {
      Comparable max;       //  the data stored in x[]
      int maxIndex;               // an index is an int
      for (int i=size-1; i>=1; i--)
      {
       // Find the maximum in the x[0..i]
         max = x[i];   // the "current"  maximum is x[i]
         maxIndex = i;      //  index of  "current" max

         for (int j=i-1; j>=0; j--)
         {
            if (max.compareTo( x[j]) <0)   // max < x[i]
            {
               max = x[j];   // a "new" maximum
               maxIndex = j;
            }
         }
         if (maxIndex != i)
        // place the maximum in its proper position
         {
            x[maxIndex] = x[i];
            x[i] = max;
         }
      }
   }
}
```

Here is an example of two programs that use this new generic sort. The same sort routine is used to sort an are of Box objects and an array od Strings

```
import java.util.*;  // for Random
public class SortBoxes
{
 public static void main(String [] args)
 {

     Random rand = new Random();
     Box[] boxes = new Box[10];
     for ( int i = 0; i < 10; i++)
     {
       // get 10 boxes with random dimensions
       int length = rand.nextInt(5) +1;
       int width = rand.nextInt(5) +1;
       int height = rand.nextInt(5) +1;
       boxes[i] = new Box(length,width,height);
     }

     SelectionSort.sort(boxes, 10);
     System.out.println("Boxes sorted by volume:\n");
     for (int i = 0; i <10; i++)
        System.out.println(boxes[i]+ " Volume: "+
                                   boxes[i].volume());
   }  // end main
}
```

```
public class SortStrings
{
 public static void main(String [] args)
 {
       // Make an array of String
        String[] schittsCreek=
        {"Moira ","David", "Alexis", "Johnny",
"Roland","Jocelyn",
        "Twyla","Stevie"  };

      // String implements Comparable
      SelectionSort.sort(schittsCreek, schittsCreek.length);


      System.out.println("Schitts Creek Characters\n");
      for (int i = 0; i <schittsCreek.length; i++)
             System.out.println(schittsCreek[i]);
   }
}
```

Output

```
Boxes sorted by volume:

Length: 1      Width: 3      Height: 1  Volume: 3
Length: 3      Width: 1      Height: 1  Volume: 3
Length: 4      Width: 1      Height: 1  Volume: 4
Length: 2      Width: 2      Height: 1  Volume: 4
Length: 1      Width: 3      Height: 2  Volume: 6
Length: 1      Width: 2      Height: 4  Volume: 8
Length: 1      Width: 5      Height: 2  Volume: 10
Length: 1      Width: 4      Height: 4  Volume: 16
Length: 2      Width: 4      Height: 3  Volume: 24
Length: 3      Width: 5      Height: 5  Volume: 75
>
```

Output:

```
Schitts Creek Characters

Alexis
David
Jocelyn
Johnny
Moira
Roland
Stevie
Twyla
```

The array to be sorted must contain objects that can be compared to each other.
You cannot sort and array that holds Circle objects and Strings, for example.