# OOP TA Session 7

Object's methods
Diamond operator
Collection
Collections

## Object's Methods

- toString
- equals
- hashCode

# Object's Methods: equals

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

Perhaps we'd like
complex1.equals(complex2)
to have a different meaning?

# Object’s Methods: equals

```
public class ComplexNumber {
     public boolean equals(Object other) {
          if(other==null ||
               !(other instanceof ComplexNumber))
              return false;
          ComplexNumber complexOther =
               (ComplexNumber)other;

          return (this.getReal()==other.getReal()&&
                 this.getImg()==other.getImg());
     }
}
```

# Object’s Methods: hashCode

- What’s it for?
- Object’s implementation: uses address

# Object's Methods: hashCode

```
Public static void main(String[] args) {
      ComplexNumber a = new ComplexNumber(1,2);
      ComplexNumber b = new ComplexNumber(a);

      hash.add(a);
      syso(hash.contains(b));
}
```

if a.equals(b), it must hold that `a.hashCode() == b.hashCode()`

## Object’s Methods: hashCode

- New hashCode must be a function only of real and img.

```
class ComplexNumber {
     public int hashCode() {
          return 0;
     }
}
```

# Object's Methods: hashCode

- New ~~hashCode must be a function~~ only of real

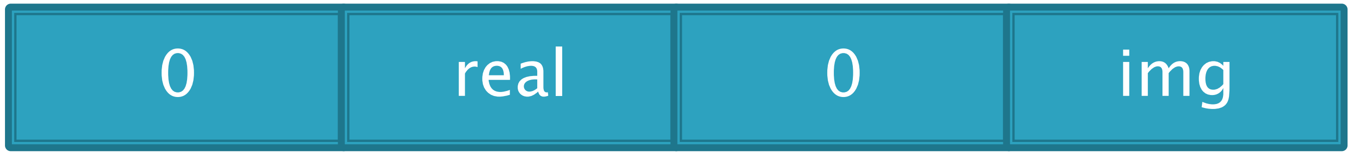
Honey, that's awful.

```
class ComplexNumber {
      public int hashCode() {
            return getReal()*getImg();
      }
}
```

# Object's Methods: hashCode

```
public class ComplexNumber {
      public int hashCode() {
            return 32*getReal() + getImg();
      }
}
```

hashCode:



## Object's Methods: hashCode

```
public class ComplexNumber {
      public int hashCode() {
            return 31*getReal() + 3*getImg();
      }
}
```

Better to use primes

## Diamond Operator

- Some classes have *type parameters*
- This will be thoroughly explained when you'll learn *generics*.

- For now, this is what you need to know:

```
LinkedList<String> list =
     new LinkedList<>();
```

```
LinkedList<String> strList = new LinkedList<>();
strList.add(“word”);
String s = strList.getFirst();
```

No downcasting?

```
LinkedList<String> strList = new LinkedList<>();
strList.add(“word”);
String s = strList.getFirst();

LinkedList<Integer> intList                    >();
intList.add(3);
```

What’s the method’s parameter?

This is not overloading!

## Diamond Operator

- LinkedList<String> is a list of strings.
- LinkedList<Integer> is a completely different class, a list of integers.

LinkedList is only a ***template***
to create ***classes*** from,
using the diamond operator <>

## Diamond Operator: Example

```
public class StringArrayFacade {
      private String[] arr;
      public ArrayFacade(int size) {
            arr = new String[size];
      }
      public String getAt(int i) {
            return arr[i];
      }
      public void setAt(int i, String newElement){
            arr[i] = newElement;
      }
}
```

I'd like a façade for every array!!

## Diamond Operator: Example

```
public class ArrayFacade<E> {
      private E[] arr;
      public ArrayFacade(int size) {
            arr = new E[size];
      }
      public E getAt(int i) {
            return arr[i];
      }
      public void setAt(int i, E newElement) {
            arr[i] = newElement;
      }
}
```

```
public class ArrayFacade<E> {
      private E[] arr;
      // ..
      public sortArray() {
            if(arr[0] <
                  no, wait.
            if(!arr[0].equals(arr[1]))
                  sh*t.
      }
}
```

## Comparable

- We need the object to tell us how it compares (similar to equals).
- A compareTo method, returning
  - –1 if *this* is smaller
  - 0 if this.equals(other)
  - 1 if *this* is bigger

## Comparable

```
public interface Comparable {
      int compareTo(Comparable other);
}
```

implementations will have instanceof & downcasting…

## Comparable<T>

```
public interface Comparable<T> {
        int compareTo(T other);
}


public class ComplexNumber implements
                        Comparable<ComplexNumber> {

        int compareTo(ComplexNumber other) {
                return Double.compare(
                        this.getReal(), other.getReal());
        }
}
```

## Collection<E>

- An interface for collections of E.
- LinkedList<E> implements Collection<E>
- So do java's trees, sets, hash tables…
- So can you!

# Collection<E> Interface:

```
public interface Collection<E> extends Iterable<E> {
        // Basic operations
        int size();
        boolean isEmpty();
        boolean contains(Object element);
        boolean add(E element); //optional
        boolean remove(Object element); //optional
        Iterator<E> iterator();

        // Bulk operations
        boolean containsAll(Collection<?> c);
        boolean addAll(Collection<? extends E> c); //optional
        boolean removeAll(Collection<?> c); //optional
        boolean retainAll(Collection<?> c); //optional
        void clear(); //optional

        // Array operations
        Object[] toArray();
        <T> T[] toArray(T[] a);
}
```

## Collections utility class

- One of the reasons "Collection" is useful
- Collections offers yamba static methods for a Collection:
  - sort
  - reverse
  - swap
  - shuffle
  - min/max
  - …

# Collections utility class

```
LinkedList<ComplexNumber> list = new LinkedList<>();
list.add(new ComplexNumber(5,3);
list.add(new ComplexNumber(2,4);

Collections.sort(list);
```

✓

- Whoa cowboy, how does it know the order?
- Only if ComplexNumber implemented Comparable.

## Collections utility class

```
LinkedList<ComplexNumber> list = new LinkedList<>();
list.add(new ComplexNumber(5,3);
list.add(new ComplexNumber(2,4);

Collections.sort(list);
```

- What if I want to sort by img?
- Complex numbers don't have a natural order.
- Comparable's *compareTo* defines the natural order of a class.

- Still, sometimes I’d like to sort by real,
  sometimes by img,
  sometimes by length.

Comparator.

# Comparator interface

- **Comparator** – encapsulates ordering
- Consists of a single method:

```
public interface Comparator {
    int compare(Object obj1, Object obj2);
}
```

  - Return values: same as **Comprable.compareTo**

# Comparator<T> interface

- **Comparator** – encapsulates ordering
- Consists of a single method:

```
public interface Comparator<T> {
    int compare(T obj1, T obj2);
}
```

  - Return values: same as **Comprable.compareTo**

## Comparator<T> interface

```
public class ImgComparator
         implements Comparator<ComplexNumber> {

    int compare(ComplexNumber num1,
                ComplexNumber num2) {

        return Double.compare(
            num1.getImg(), num2.getImg());
}
```

# Comparator<T> interface

```
private static Comparator<ComplexNumber>
          IMG_COMPARATOR = new ImgComparator();
          ...
public static void main(String[] args) {
     LinkedList<ComplexNumber> list = ...
     Collections.sort(list, IMG_COMPARATOR);
     Collections.sort(list, REAL_COMPARATOR);
     Collections.sort(list, LENGTH_COMPARATOR);
}
```

Note: not necessarily consistent with *equals*!

## Comparator vs. Comparable

- Comparator
  - A class can have many
  - Allows to change the comparison method in runtime
  - The same Comparator class may be used for different classes (prevents code repetition)

- Comparable
  - A class has only one
  - More likely to be consistent with *equals*() (as both methods were written by the same developer)
  - Has access to private variables
    - Might be relevant to the comparison

# Example:PersonName Class

```
import java.util.*;
public class PersonName implements Comparable<PersonName> {
    // fields
    private final String firstName, lastName;
    //constructor
    public PersonName(String firstName, String lastName) {
            // Checking that arguments are not null
            if (firstName == null || lastName == null)
                    throw new NullPointerException();
            this.firstName = firstName;
            this.lastName = lastName;
    }
```

# PersonName Class (cnt'd)

```
// Getter mehtods – no setter methods!
public String firstName() { return firstName; }
public String lastName() { return lastName; }

// Overriding equals of Object
public boolean equals(Object o) {
        // Return false if o is null or is not a PersonName
        if (o == null || !(o instanceof PersonName))
                return false;
        PersonName n = (PersonName) o;
        return n.firstName().equals(firstName) &&
        n.lastName().equals(lastName);
}
```

## PersonName Class (cnt'd)

```
    // Overiding hashCode of Obejct
    public int hashCode() {
        return 31*firstName.hashCode() +  1*lastName.hashCode();
    }
     // Overriding toString of Object
    public String toString() { return firstName + " " + lastName; }

     // Implementing Comparable<PersonName> interface method
    public int compareTo(PersonName n) {
        int cmp = lastName.compareTo(n.lastName());
        return (cmp != 0 ? cmp : firstName.compareTo(n.firstName()));
    }
} //end of class PersonName
```

## Ex4: hash-sets!