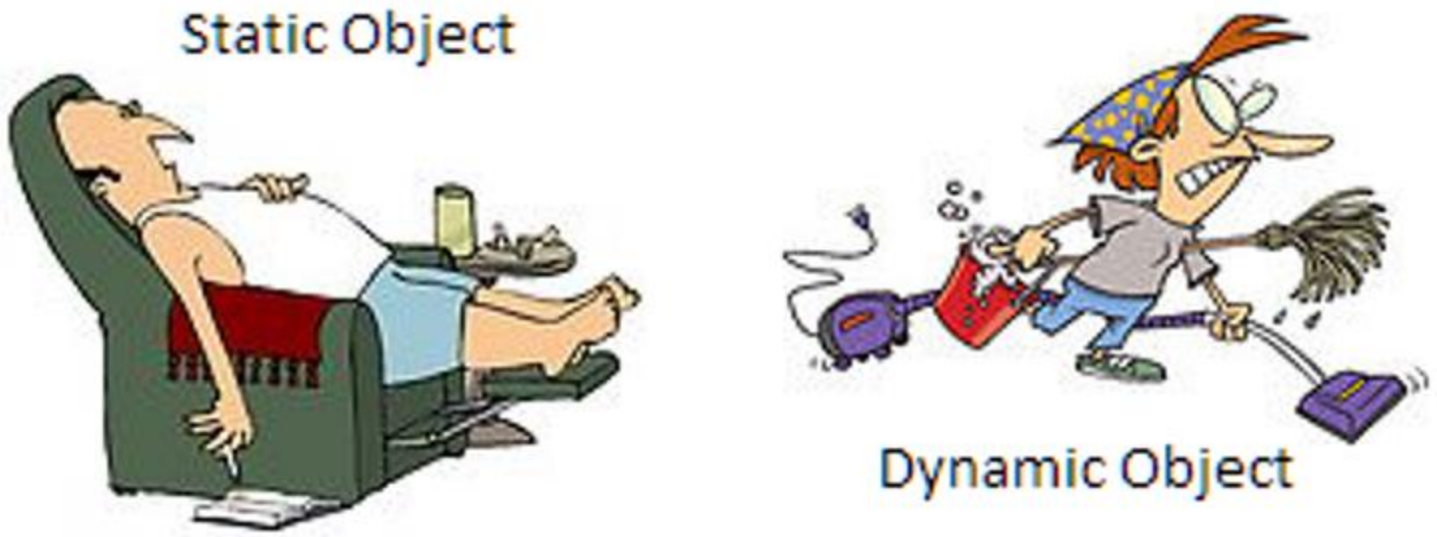
Fakultas Informatika
School of Computing
Telkom University

CII3B4
Pemrograman Berorientasi Objek

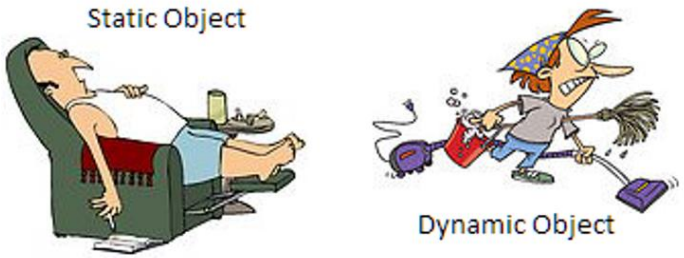
Static and Collection

Static Modifier



Static Modifier

- Static variables
- Static method



Static Variables

- Create variables that will exist independently of any instances created for the class.
- Only one copy of the static variable exists regardless of the number of instances of the class.
- Also known as class variables.
- Local variables cannot be declared static

Static Variables

- One variable shared for all instances
- One value per class, instead of one value per instance

Normal Variables

```
public class Duck{
   private int size;

   public Duck(int size) {
      this.size = size;
   }

   public int getSize(){
      return size;
   }

}
```

```
public class Driver{
   public static void main(String args[]){
      Duck d;
      d = new Duck(4);
      d = new Duck(10);
      d = new Duck(16);
      d = new Duck(5);
   }
}
```

Question: how many Ducks that have been instantiated?

Normal Variables

```
public class Duck{
   private int size;
   private int duckCount = 0;

   public Duck(int size) {
      this.size = size;
      duckCount++;
   }

   public int getSize(){
      return size;
   }

   public int getCount(){
      return duckCount;
   }

}
```

```
public class Driver{
   public static void main(String args[]){
      Duck d;
      d = new Duck(4);
      d = new Duck(10);
      d = new Duck(16);
      d = new Duck(5);

      System.out.println(d.getCount());

   }
}
```

```
Output :
> 1
```

Static Variables

```
public class Duck{
   private int size;
   private static int duckCount = 0;

   public Duck(int size) {
      this.size = size;
      duckCount++;
   }

   public int getSize(){
      return size;
   }

   public int getCount(){
      return duckCount;
   }

}
```

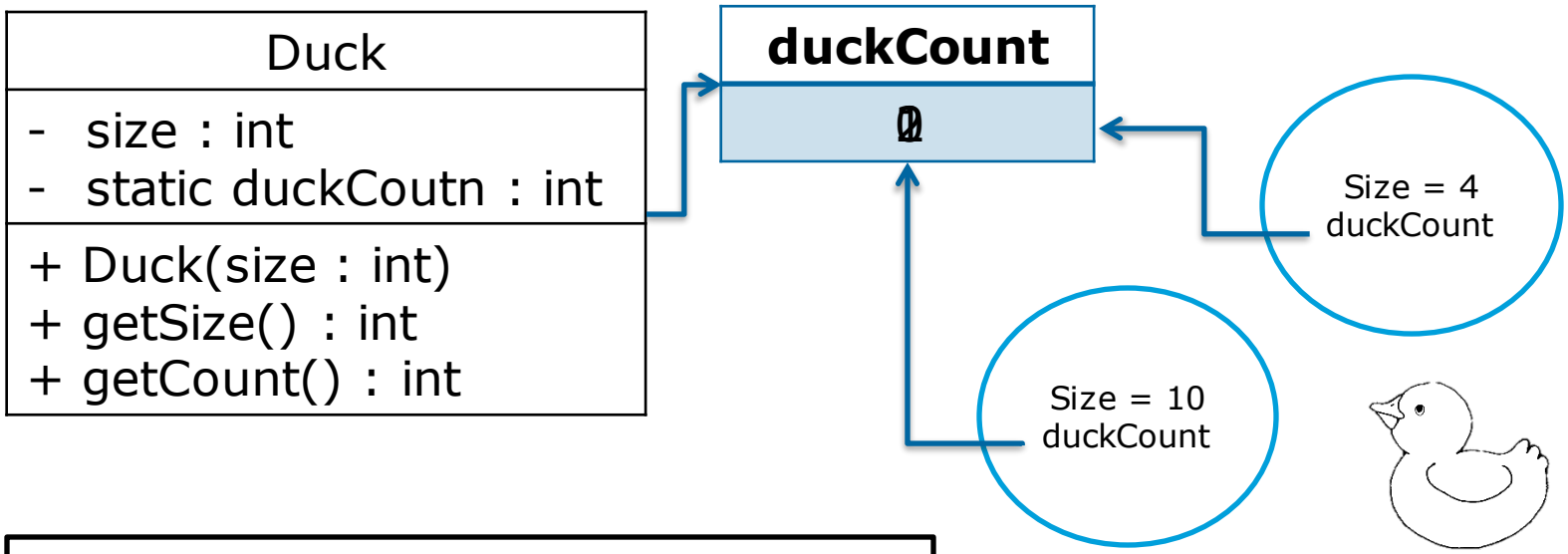
```
public class Driver{
   public static void main(String args[]){
      Duck d;
      d = new Duck(4);
      d = new Duck(10);
      d = new Duck(16);
      d = new Duck(5);

      System.out.println(d.getCount());

   }
}
```

```
Output :
> 4
```

Static Variable



- A duck object doesn't keep its own copy of duckCount
- Duct objects all share a single copy of duckCount
- Variable that lives in a CLASS, instead of in an object

Each Duck object has its own size variable, but there's only one copy of duckCount variable

Static Variables

```
public class Share {
   private int privateInt;
   private static int staticInt;

   public Share(int pr, int st){
      privateInt = pr;
      staticInt = st;
   }

   public String toString(){
      return privateInt +" "+ staticInt;
   }
}
```

Static Variables

```
public class Driver{
   public static void main(String args[]){
      Share s1 = new Share(4,4);
      System.out.println(s1.toString());

      Share s2 = new Share(8,2);
      System.out.println(s1.toString());
      System.out.println(s2.toString());

      Share s3 = new Share(6,22);
      System.out.println(s1.toString());
      System.out.println(s2.toString());
      System.out.println(s3.toString());
   }
}
```

Output :

> 4 4

> 4 2
> 8 2

> 4 22
> 8 22
> 6 22

Static Methods

- Create methods that will exist independently of any instances created for the class
- Static methods do not use any instance variables of any object of the class they are defined in.
- Only can access static variables

Static Methods

- Static methods take all the data from parameters and compute something from those parameters, with no reference to variables
- Class variables and methods can be accessed using the class name followed by a dot and the name of the variable or method
  - Without instantiating the object

Static Variables

```
public class CounterMachine{
   private static int counter = 0;

   public static void count(){
      counter++;
   }

   public static int getCounter(){
      return counter;
   }

}
```

```
public class Driver{
   public static void main(String args[]){

      for(int i = 0; i< 10; i++){
         if( i % 2 == 0){
            CounterMachine.count();
         }
      }

      System.out.println(
            CounterMachine.getCounter());
   }
}
```

```
Output :
> 5
```

Static Variables

```
public class CounterMachine{
   public static int counter = 0;

   public static void count(){
      counter++;
   }

}
```

```
public class Driver{
   public static void main(String args[]){

      for(int i = 0; i< 10; i++){
         if( i % 2 == 0){
           CounterMachine.count();
         }
      }

      System.out.println(
           CounterMachine.counter);
   }
}
```

Final keyword

- Java's final keyword has slightly different meanings depending on the context
- In general it says "This cannot be changed."

Final data and method

- Constant:
  - It can be a compile-time constant that won't ever change.
  - It can be a value initialized at run time that you don't want changed.

- In Java, these constant must be primitive type and with **final** keyword
- Final on method → the method can not be overriden

Final on object

- When final is used with object references rather than primitives, the meaning can be confusing.
- With a primitive, final makes the value a constant, but with an object reference, final makes the reference a constant.
- Once the reference is initialized to an object, it can never be changed to point to another object
- Object declared final → can not be extended
- However, the object itself can be modified

Note

- By convention: fields that are both static and final (that is, compile-time constants) are capitalized and use underscores to separate words

- Example:

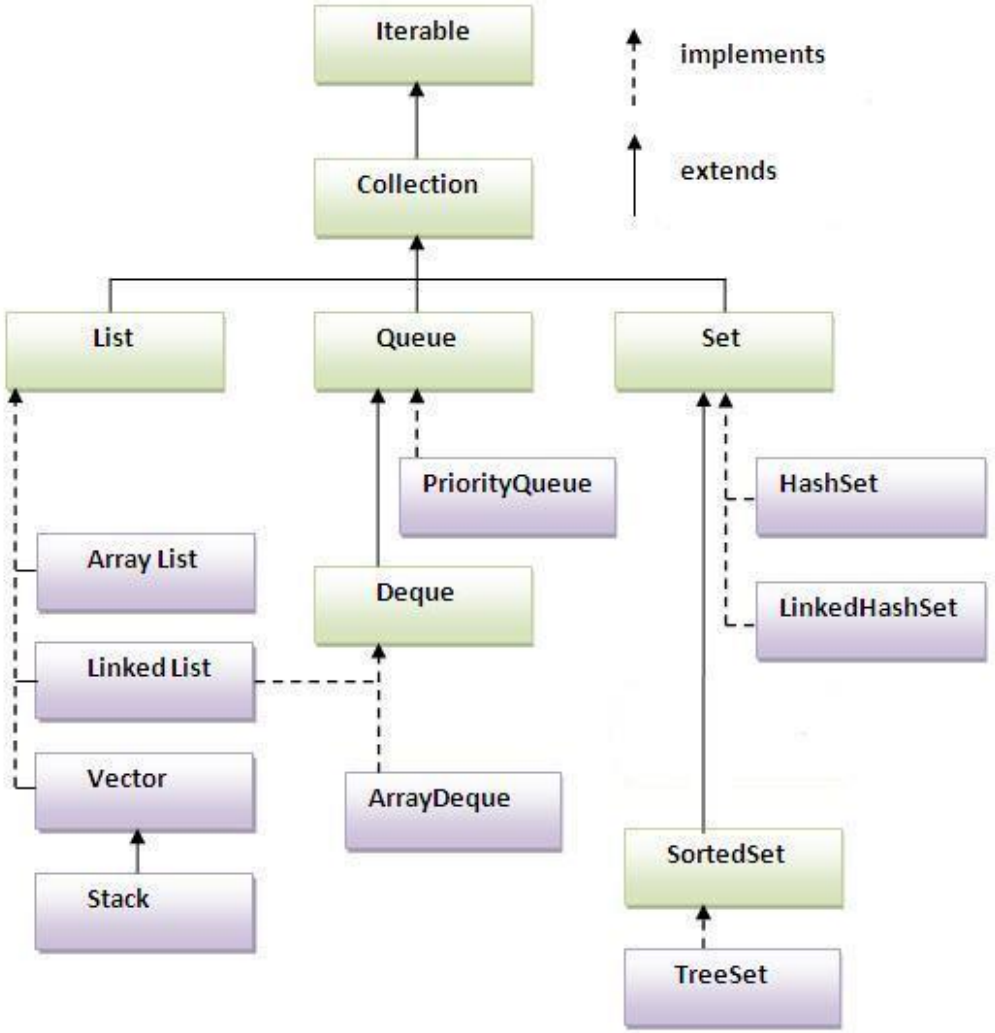
```
public static final RHO_NUMBER = 247;
```

Java Collection

Java Collection

containers of Objects which by polymorphism can hold any class that derives from Object (which is actually, any class)

Using Generics the Collection classes can be aware of the types they store



Java Collection

- add(item)
- remove(item)
- addAll(collection)
- removeAll(collection)
- retainAll(collection)
- contains(item)

Java Collection

```
List list = new ArrayList();

// normal loop
for (int i = 0; i < list.size() ; i++ ) {
    Object o = list.get(i);
    System.out.println(o);
}

// loop using for-element
for (Object o : list) {
    System.out.println(o);
}
```

Java Collection

```
// Loop using iterator
Iterator itr = list.iterator();
while (itr.hasNext()) {
    Object o = itr.next();
    System.out.println(o);
}

// loop using lambda Expression
    list.forEach( o -> System.out.println(o));

// loop using reference
    list.forEach( System.out :: println);
```

Generic Collection

- `Collection<String> str = new HashSet<String>();`
- `List<Integer> arr_i = new List ();`
- `ArrayList<Employee> emp = new ArrayList ();`

Sorting Collection – Comparable

- Collections.sort(list) or list.sort(null)
  - If the List consists of String elements, it will be sorted into alphabetical order.
  - If it consists of Date elements, it will be sorted into chronological order
  - To create a custom ordering for list of objects, the class must implement interface Comparable
    - implement method public int compareTo(Object)
- For arrays, use Arrays.sort(array[])

Sorting Collection – Comparable

- public int compareTo(Object o)
  - Return < 0 if "this" object will be sorted before Object o
  - Return = 0 if "this" object and Object o is equal
  - Return > 0 if "this" object will be sorted after Object o
- If the parameter is either String or Date, use compareTo
  - return this.getString().compareTo(o.getString());
- If the parameter is numerical, use substraction
  - return this.getNumber() – o.getNumber();

Example - sort(list of string)

```
List<String> list = new ArrayList();
list.add("Pineapple");
list.add("Apple");
list.add("Orange");
list.add("Banana");
Collections.sort(list);

int i = 0;
for (String s : list) {
    System.out.println("fruits " + ++i + " : " + s);
}
```

```
String[] fruits
        = new String[]{"Pineapple", "Apple", "Orange", "Banana"};
Arrays.sort(fruits);
int i = 0;
for (String temp : fruits) {
    System.out.println("fruits " + ++i + " : " + temp);
}
```

Example – sort(list of object)

```
public class Employee implements Comparable<Employee> {

    private String name;
    private double salary;

    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    public String getName() {
        return name;
    }

    public double getSalary() {
        return salary;
    }
```

```
    @Override
    public String toString() {
        return "name=" + name + ", salary=" + salary;
    }

    @Override
    public int compareTo(Employee t) {
        return (this.name).compareTo(t.name);
    }

}
```

Example – sort(list of object)

```
public static void main(String[] args) {
    List<Employee> listEmp = new ArrayList();

    listEmp.add(new Employee("bobby", 5));
    listEmp.add(new Employee("erick", 56));
    listEmp.add(new Employee("anna", 15));
    listEmp.add(new Employee("rey", 25));

    Collections.sort(listEmp);
    listEmp.forEach( System.out :: println);
}
```

```
name=anna, salary=15.0
name=bobby, salary=5.0
name=erick, salary=56.0
name=rey, salary=25.0
```

Sorting Collection - Comparator

- Comparable Interface only allows to sort a single property.
- To sort with multiple properties, you need **Comparator class**
- Collections.sort(list, comparator)
  or list.sort(comparator)
- For arrays, use Arrays.sort(array[], comparator)

Sorting Collection - Comparator

- Create a comparator class implements Comparator interface to compare the element
  - public interface Comparator<T> {
    int compare(T object1, T object2);
    }

```
public class SalaryComparator implements Comparator<Employee>{

    @Override
    public int compare(Employee t, Employee t1) {
        return (int) (t.getSalary() - t1.getSalary());
    }

}
```

Sorting Collection Example

```
public static void main(String[] args) {
    List<Employee> listEmp = new ArrayList();

    listEmp.add(new Employee("bobby", 5));
    listEmp.add(new Employee("erick", 56));
    listEmp.add(new Employee("anna", 15));
    listEmp.add(new Employee("rey", 25));

    Collections.sort(listEmp);
    for (Employee emp : listEmp) {
        System.out.println(emp);
    }

    Collections.sort(listEmp, new SalaryComparator());
    for (Employee emp : listEmp) {
        System.out.println(emp);
    }
}
```

```
name=anna, salary=15.0
name=bobby, salary=5.0
name=erick, salary=56.0
name=rey, salary=25.0

name=bobby, salary=5.0
name=anna, salary=15.0
name=rey, salary=25.0
name=erick, salary=56.0
```

Filter elements from Collections

- Select a specific element based on parameter
  - Select all Employee with salary above 20
- Old way

```
List<Employee> temp = new ArrayList();
for (Employee e : listEmp) {
    if (e.getSalary() > 20) {
        temp.add(e);
    }
}
```

Filter elements from Collections

- Select all Employee with salary above 20
- Using Lambda expression
  - list.stream().filter(o -> condition)

```
List<Employee> temp = listEmp.stream()
        .filter(e -> e.getSalary() > 20).collect(Collectors.toList());
```

Filter elements from Collections

- Retrieve an element based on some attributes
  - Select Employee by name
- Old way

```
public Employee getEmployee(String name) {
    for (Employee e : listEmp) {
        if (e.getName().equals(name)) {
            return e;
        }
    }
    return null;
}
```

Filter elements from Collections

- Select Employee by name
- Using Lambda expression
  - list.stream().filter(o -> condition).findFirst()
  - .orElse(x)

```
public Employee getEmployee(String name) {
    return listEmp.stream()
            .filter(o -> o.getName().equals(name))
            .findFirst().orElse(null);
}
```

Question?

Fakultas Informatika
School of Computing
Telkom University

THANK YOU