# Java Generics, Lambdas, Containers

Dr. Eliahu Khalastchi

2017

המסלול האקדמי
המכללה למינהל

# Java <Generics>

And the “type ensure” technique

# C++ recap on templates

Source code:

```
template<class T>
class Holder{
  T* t;
  public:
  void set(T* t){ this->t = t; }
  T* get(){ return t; }
};
```

```
void main(){
  Holder<Student> hs;
  Holder<Employee> he;
  Holder<int> hi;
  cout << (typeid(hs)==typeid(he)) <<endl;
}
```

false

Complied code:

```
class Holder{
  Student* t;

};
```

```
class Holder{
  Employee* t;

};
```

```
class Holder{
  int* t;
  public:
  void set(int * t){ this->t = t; }
  int* get(){ return t; }
};
```

# Java - before 1.5

```java
public class Holder {
  Object t;
  public void set(Object t){ this.t=t;}
  public Object get(){return t;}
}
```

```java
public static void main(String[] args) {
  Holder h=new Holder();

  h.set(new Student());
  ((Student)h.get()).study();

  h.set(new Employee());
  ((Employee)h.get()).work();
}
```

```java
h.set(new Employee());
((Employee)h.get()).work();

//...

((Student)h.get()).study();
```

Exception! (at runtime! ☹)

# Java - before 1.5

```
public class Holder {
  Object t;
  public void set(Object t){ this.t=t;}
  public Object get(){return t;}
}
```

```
public static void main(String[] args) {
  Holder h=new Holder();

  h.set(new Student());
  ((Student)h.get()).study();

  h.set(new Employee());
  ((Employee)h.get()).work();
}
```

# Since 1.5 – generics!

```
public class Holder<T> {
  T t;
  public void set(T t){ this.t=t;}
  public T get(){return t;}
}
```

```
public static void main(String[] args) {
  Holder<Student> hs=new Holder<Student>();
  hs.set(new Student());
  hs.get().study();

  Holder<Employee> he=new Holder<Employee>();
  he.set(new Employee());
  he.get().work();
}
```

# Ensured type safety

```
public class Holder<T> {
  T t;
  public void set(T t){ this.t=t;}
  public T get(){return t;}
}
```

```
public static void main(String[] args) {
  Holder<Student> hs=new Holder<Student>();
  hs.set(new Student());
  hs.get().study();

  Holder<Employee> he=new Holder<Employee>();
  he.set(new Employee());
  he.get().work();
}
```

# Ensured type safety

```
public class Holder<T> {
  T t;
  public void set(T t){ this.t=t;}
  public T get(){return t;}
}
```

```
public static void main(String[] args) {
  Holder<Student> hs=new Holder<Student>();
  hs.set(new Student());
  hs.get().study();

  Holder<Employee> he=new Holder<Employee>();
  he.set(new Employee());
  he.get().work();
}
```

```
Holder<Student> hs=new Holder<Student>();
hs.set(new Student());
hs.get().study();

//...

hs.set(new Employee());
```

Compilation Error ☺

# “type ensure” - used by Java

```
public class Holder<T> {
  T t;
  public void set(T t){ this.t=t;}
  public T get(){return t;}
}
```

Syntax sugar

Complied code:

```
public class Holder {
  Object t;
  public void set(Object t){ this.t=t;}
  public Object get(){return t;}
}
```

```
public static void main(String[] args) {
  Holder<Student> hs=new Holder<Student>();
  hs.set(new Student());
  hs.get().study();

  Holder<Employee> he=new Holder<Employee>();
  he.set(new Employee());
  he.get().work();
}
```

Complied code:

```
public static void main(String[] args) {
  Holder hs=new Holder();
  hs.set(new Student());
  ((Student)h.get()).study();

  Holder he=new Holder();
  he.set(new Employee());
  ((Employee)he.get()).work();
}
```

# “type ensure” - used by Java

Complied code:

```
public class Holder<T> {
  T t;
  public void set(T t){ this.t=t;}
  public T get(){return t;}
}
```

```
public class Holder {
  Object t;
  public void set(Object t){ this.t=t;}
  public Object get(){return t;}
}
```

Implication: We **can’t** write generic code that requires **runtime information**

- T t = **new** T();
- T[] array = **new** T[10];
- t.doSomething();

Compilation Error ☹

(ok in C++)

```
In addition:

// Holder<int> hi;  - compilation error
Holder<Student> hs=new Holder<Student>();
Holder<Employee> he=new Holder<Employee>();
System.out.println((he.getClass()==hs.getClass()));
```

(again, ok in C++)

true

# Quiz: will this compile in Java?

```
public class GenericException<T> extends Exception {…}
```

```
try {
   throw new GenericException<Integer>();
}
catch(GenericException<Integer> e) {
   System.err.println("Integer");
}
catch(GenericException<String> e) {
   System.err.println("String");
}
```

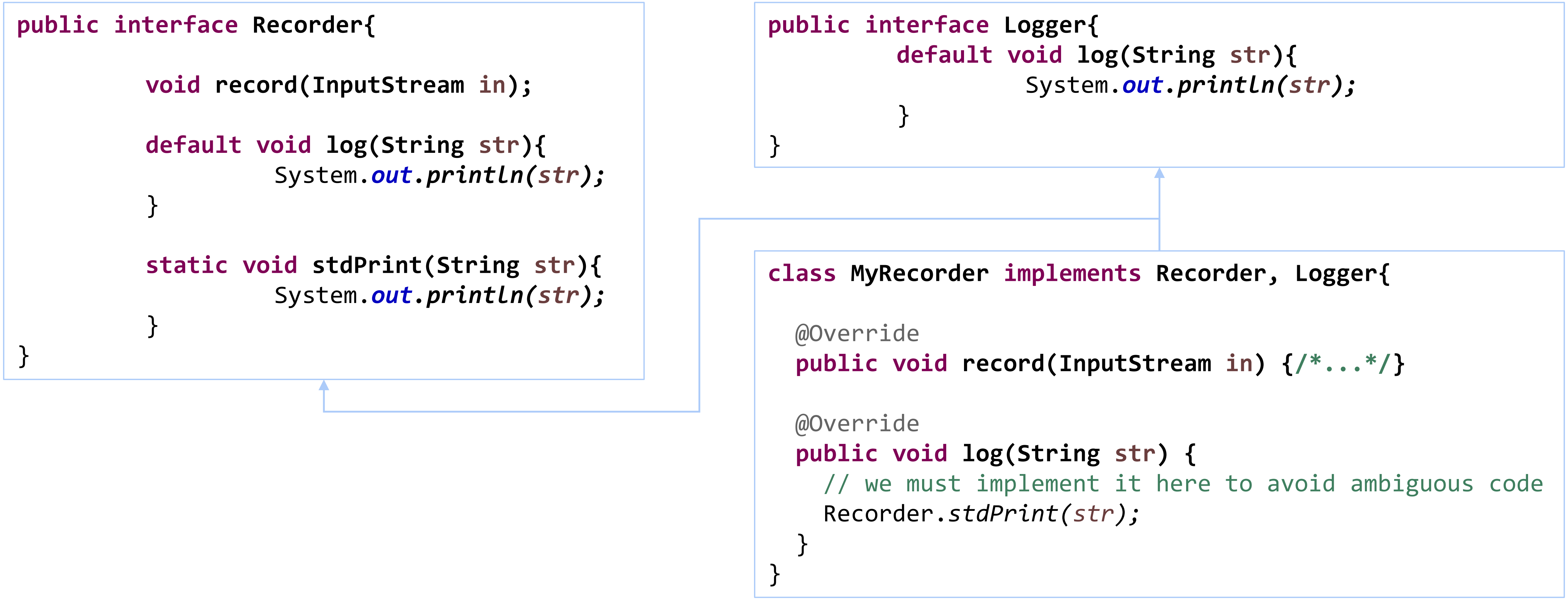
# Java8

Some of the new stuff…

# Default & Static Methods

In Interfaces(!)

# Default & Static Interface Methods



```java
public interface Recorder{

        void record(InputStream in);

        default void log(String str){
                System.out.println(str);
        }

        static void stdPrint(String str){
                System.out.println(str);
        }
}
```

```java
public interface Logger{
        default void log(String str){
                System.out.println(str);
        }
}
```

```java
class MyRecorder implements Recorder, Logger{

  @Override
  public void record(InputStream in) {/*...*/}

  @Override
  public void log(String str) {
    // we must implement it here to avoid ambiguous code
    Recorder.stdPrint(str);
  }
}
```

# Lambda Expressions

Java 8

# Lambda Expressions

```java
public interface FunctionalInterface{
        String func(String str);
}


// Anonymous class...
FunctionalInterface f=new FunctionalInterface() {
        @Override
        public String func(String str) {
                return new StringBuilder(str).reverse().toString();
        }
};

System.out.println(f.func("Hello World!"));
```

# Lambda Expressions

```
public interface FunctionalInterface{
        String func(String str);
}
```

```
FunctionalInterface f;
f=(String str)->{return new StringBuilder(str).reverse().toString();};

System.out.println(f.func("Hello World!"));
```

# Lambda Expressions

```
public interface FunctionalInterface{
        String func(String str);
}
```

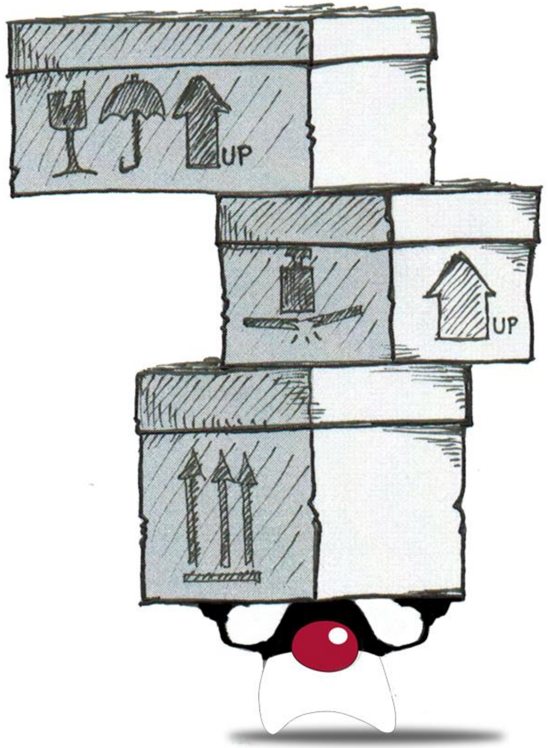
```
FunctionalInterface f;
f=str->new StringBuilder(str).reverse().toString();

System.out.println(f.func("Hello World!"));
```

# Lambda Expressions

- It’s not (just) about the syntax, it’s about the **paradigm!**
- **Separation** of data and functionality
- **Functionality** can be passed as data – more expressive APIs
- **Fluent** (pipelined) operations – better readability
  - Instead of nested loops…
- Libraries are in **control** of computations
  - e.g. **internal iterators** instead of external
  - More **opportunities** for optimizations
    - Laziness
    - Parallelism
    - out-of-order computations

# Java Containers

java.util.*



# Introduction

- Java implemented some very useful data-structures in the *java.util.** library
  - Called containers

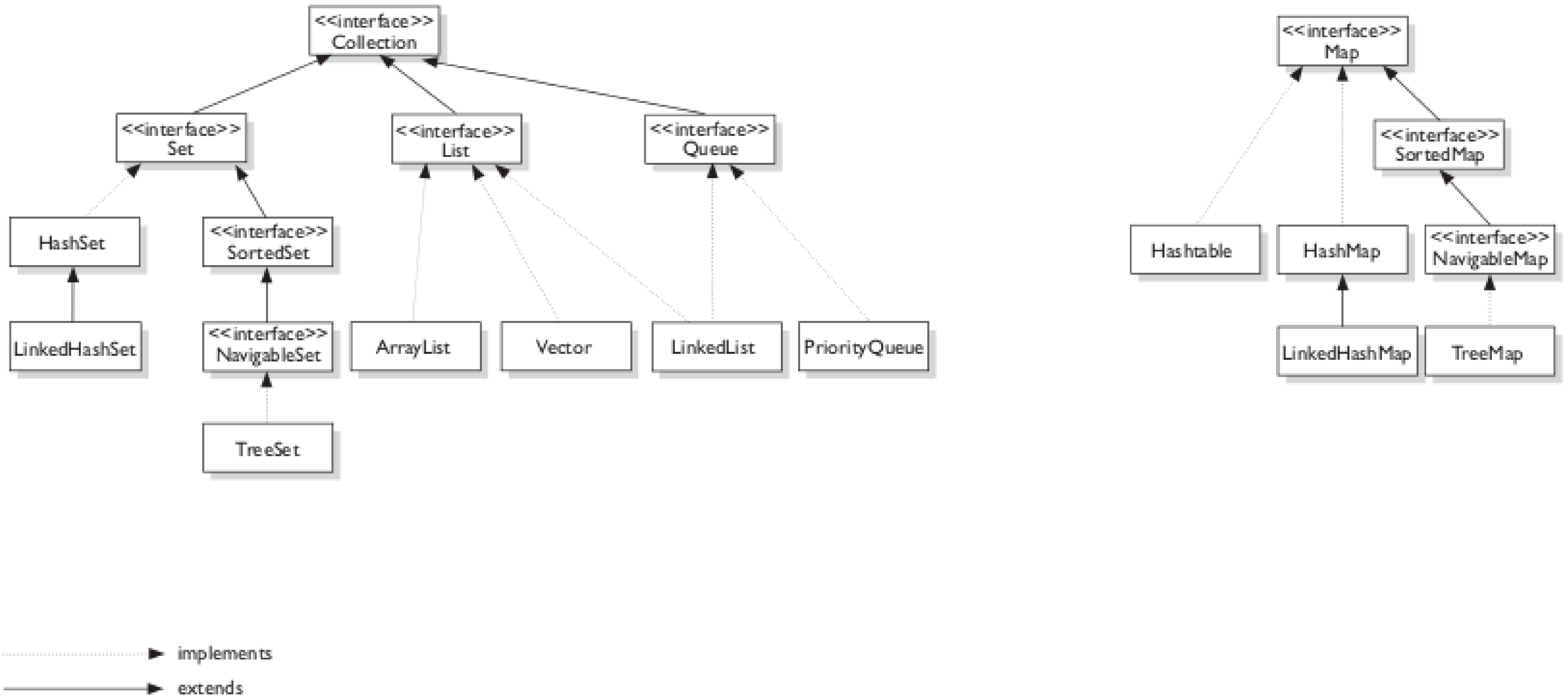
- They only handle objects, not primitive types
  - Instead of *int* we need to use *Integer*
  - Not very memory-efficient

```
ArrayList<Integer> x=new ArrayList<Integer>();
x.add(new Integer(1));// ok
x.add(2);// also ok
```

# Useful Containers

- Java has 2 types of containers:
  - **Collections** – collect single **values**
    - Lists – sequence is important
    - Sets – each element appears only once
  - **Maps** – map **keys** to **values**

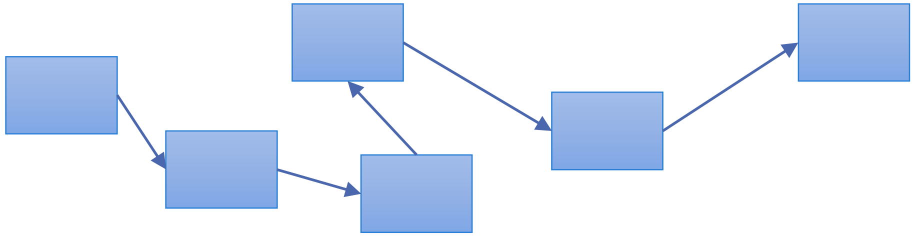
- The implementation is as you have learned in Data-Structures course
- Use them wisely


<<interface>> Collection
<<interface>> Set
<<interface>> List
<<interface>> Queue
HashSet
<<interface>> SortedSet
LinkedHashSet
<<interface>> NavigableSet
TreeSet
ArrayList
Vector
LinkedList
PriorityQueue
<<interface>> Map
<<interface>> SortedMap
Hashtable
HashMap
<<interface>> NavigableMap
LinkedHashMap
TreeMap
implements
extends

# Useful Containers - collections

- Lists:
  - ***ArrayList<E>*** – uses an array
    - Fast random access: O(1)
    - Slow addition / deletion from the middle: O(1) amortized

  - ***LinkedList<E>*** – uses a linked list
    - Slow random access: O(n)
    - Fast addition / deletion from the middle: O(1)



**Example:**

```
List<String> strings=new ArrayList<String>();
strings.add(“hello world”);
```

# Useful Containers - collections

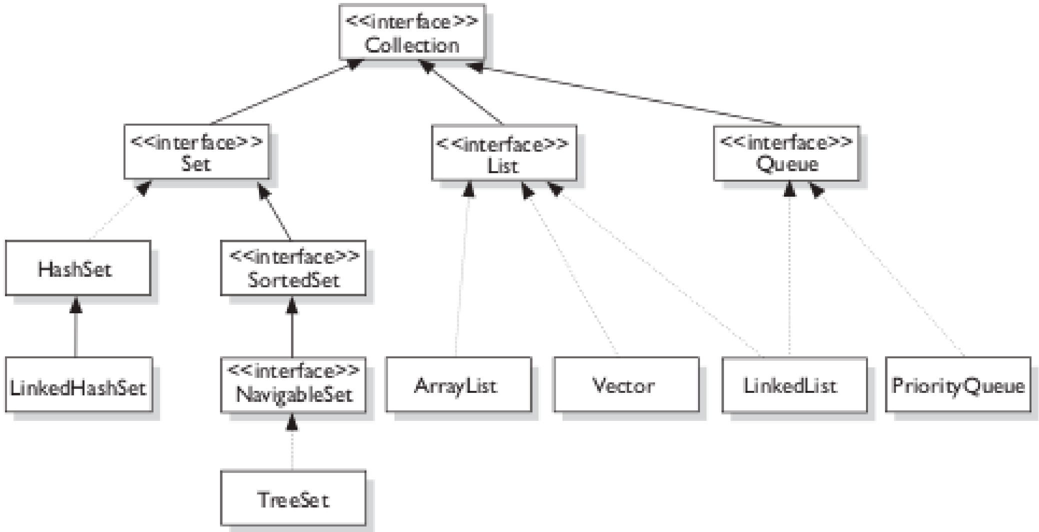
- Sets:
  - ***HashSet<E>*** – uses a hash table
    - Use when search time is important
    - *Object*'s *int HashCode()* method needs to be overridden
    - Usually we'll use something ready as *String*'s hash code
  - ***TreeSet<E>*** – uses a balanced tree
    - O(log(n)) for random access
    - Can easily extract a sorted list

**Example:**

```
Set<String> strings=new HashSet<String>();
strings.add("hello world");
```

# Methods of collections <E>

- boolean add(E e)
- boolean addAll(Collection<? Extends E> c)
- void clear()
- boolean contains(Object o)
- boolean containsAll(Collection<?> c)
- boolean isEmpty()
- Iterator<E> iterator()
- boolean remove(Object o)
- boolean removeAll(Collection<?> c)
- boolean retainAll(Collection<?> c)
- int size()
- Object[] toArray()
- <T> T[]  toArray(T[] a)



**Example:**

```
Set<String> names=new HashSet<String>();
//…
List<String> members=new ArrayList<String>();
members.add("Moshe");
members.addAll(names);
```

# Useful Containers

Maps example:

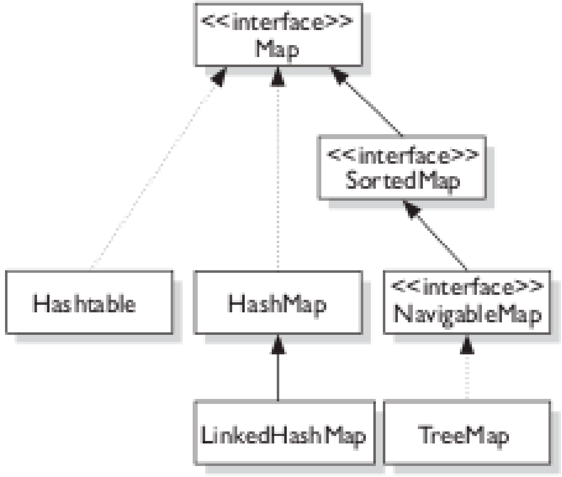
- ***HashMap*** – uses a hash table
  - The ***key*** object needs to implement *hashCode()* method
- ***LinkedHashMap***
  - Also stores the order of entry
- ***TreeMap*** – uses a red-black tree
  - Can easily extract a sorted list

**Example:**

```
Map<Integer, Employee> workers;
workers=new HashMap<Integer, Employee>();
workers.put(123456789, new Employee());
```

# Methods of maps <K,V>

- V put(K **key**, V **value**)
- void putAll(Map<? extends K, ? extends V > m)
- V get(K **key**)
- void clear()
- boolean containsKey(Object **key**)
- boolean containsValue(Object **value**)
- boolean isEmpty()
- V remove(Object **key**)
- int size()
- Collection<V> values()
- Set<K> keySet()
- Set<Map.Entry<K,V>> entrySet()



**Example:**

```
Map<Integer, Employee> workers;
workers=new HashMap<Integer, Employee>();
workers.put(123456789, new Employee());
```

# Sorting Example

ArrayList & Comparable & Comparator…

# Comparators

- Java implemented two interfaces
  - Comparator
  - Comparable
- They are used in a **strategy pattern** to sort various objects in the containers

```
interface Comparator {
      int compare(Object o1, Object o2);
}
```

```
interface Comparable {
      int compareTo(Object o) ;
}
```

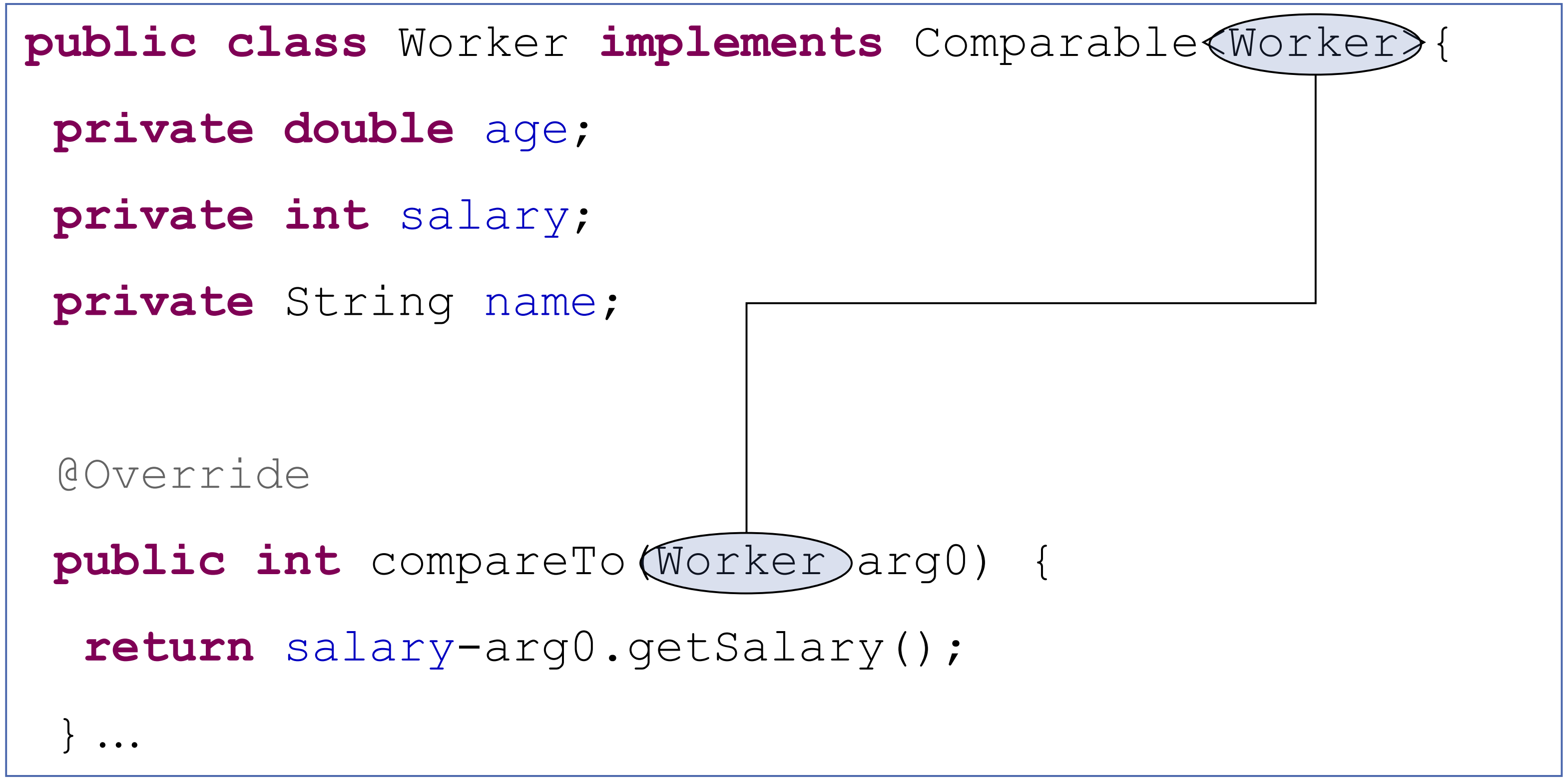
# Comparators

- Java implemented two interfaces
  - Comparator
  - Comparable
- They are used in a **strategy pattern** to sort various objects in the containers

```
interface Comparator <T> {
      int compare(T t1, T t2);
}
```

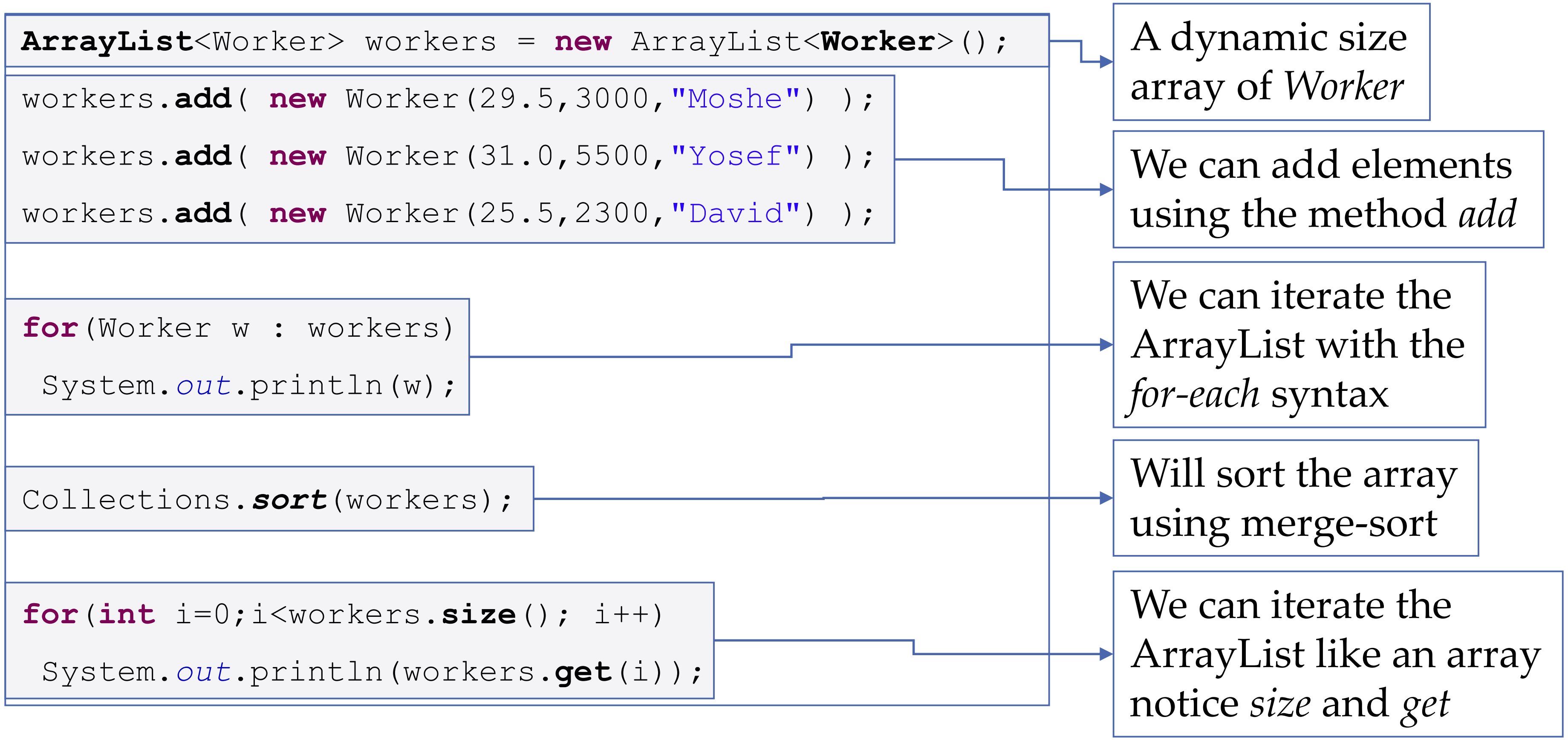
```
interface Comparable <T> {
      int compareTo(T t) ;
}
```

# Comparable



```
public class Worker implements Comparable<Worker>{
 private double age;
 private int salary;
 private String name;

 @Override
 public int compareTo(Worker arg0) {
  return salary-arg0.getSalary();
 } …
```

# Comparable effect on an ArrayList



```java
ArrayList<Worker> workers = new ArrayList<Worker>();
```

A dynamic size array of *Worker*

```java
workers.add( new Worker(29.5,3000,"Moshe") );
workers.add( new Worker(31.0,5500,"Yosef") );
workers.add( new Worker(25.5,2300,"David") );
```

We can add elements using the method *add*

```java
for(Worker w : workers)
 System.out.println(w);
```

We can iterate the ArrayList with the *for-each* syntax

```java
Collections.sort(workers);
```

Will sort the array using merge-sort

```java
for(int i=0;i<workers.size(); i++)
 System.out.println(workers.get(i));
```

We can iterate the ArrayList like an array notice *size* and *get*

# Comparable effect on an ArrayList

- How did *Collections.sort* knew how to sort?
- Because *Worker* is a ***Comparable*** object
  - The sort algorithm used the *compareTo* method
  - It was implemented to compare salaries, thus, the array was sorted by the salary field

```
public class Worker implements Comparable<Worker>{
 public int compareTo(Worker arg0) {
  return salary-arg0.getSalary();
 } …
```

```
Collections.sort(workers);
```

# Comparable effect on an ArrayList

- Why use **merge sort** and not quick sort?
- Its an optimized merge sort

- Always takes O(n·log(n)) time
  - Quick sort might take O($n^2$) in worst case scenario

- Works faster on almost sorted lists
- A sorted group of elements is left alone…

# Comparator

- But what if we want to sort the workers in a different way?
- Would we have to implement new code in each of the *Worker* classes?

- No, we can use a comparator

```
interface Comparator <T> {
    int compare(T t1, T t2);
}
```

# Comparator

- We can implement the class:

```
public class NameComparator implements Comparator<Worker>{
 @Override
 public int compare(Worker w0, Worker w1) {
  return w0.getName().compareTo(w1.getName());
 }
}
```

- And use:

```
Collections.sort(workers, new NameComparator());
```

# Comparator

- We can do the same with an anonymous class, and lambda expression

```
Collections.sort(workers, new NameComparator() );
```

```
Collections.sort(workers, new
    Comparator<Worker>(){
        @Override
        public int compare(Worker w0, Worker w1) {
            return w0.getName().compareTo(w1.getName());
        }
    }
);
```

```
Collections.sort(workers,(w0,w1)->w0.getName().compareTo(w1.getName()));
```

# Using iterators!

Examples of collections and maps

# Iterators

- Earlier we saw the *for-each* loop

```
for(Worker w : workers)
 System.out.println(w);
```

- It is actually a shortcut for an ***Iterator***

```
Iterator<Worker> it=workers.iterator();
while(it.hasNext())
 System.out.println(it.next());
```

# Iterators

- An Iterator is used for:
  - Providing access to a container's elements without publishing its implementation
  - Letting the programmers decide how to Iterate
    - They can extend an Iterator class
  - Enabling the instancing of several itrators
    - Some can go up a list
    - Some can go down
    - Some can skip every two elements
    - Etc…

# Iterators

A HashSet + Iterator example:

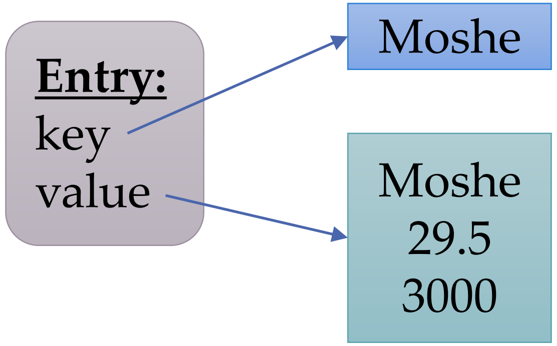
```
HashSet<Worker> hs=new HashSet<Worker>();
hs.add( new Worker(29.5,3000,"Moshe") );
hs.add( new Worker(31.0,5500,"Yosef") );
hs.add( new Worker(25.5,2300,"David") );

for(Worker w : hs)
 System.out.println(w);

Iterator<Worker> it=hs.iterator();
while(it.hasNext())
 System.out.println(it.next());
```

```
In the Worker class:
@Override
public int hashCode(){
 return (name+salary+age).hashCode();
}
```

# Iterators

A HashMap + Iterator example:

```
HashMap<String,Worker> hm = new HashMap<String, Worker>();
hm.put("Moshe" , new Worker(29.5,3000,"Moshe") );
hm.put("Yosef" , new Worker(31.0,5500,"Yosef") );
hm.put("David" , new Worker(25.5,2300,"David") );

Iterator<String> it=hm.keySet().iterator();
while(it.hasNext()){
 String k=it.next();
 System.out.println(k+","+hm.get(k));
}
for(String k : hm.keySet())
 System.out.println(k+","+hm.get(k));

for(Entry<String,Worker> e : hm.entrySet())
 System.out.println(e.getKey()+","+e.getValue());
```



**Entry:**
key
value

Moshe

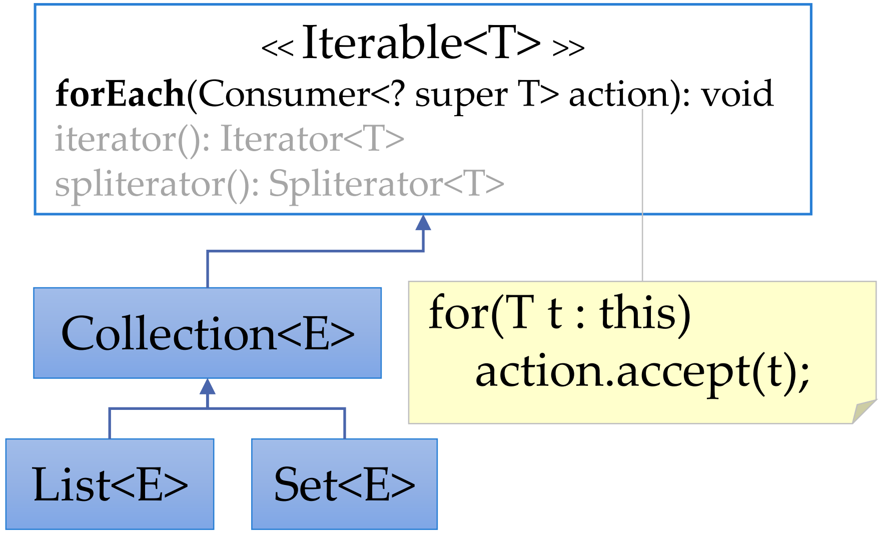
Moshe
29.5
3000

# Collection API

New in java 8

# ForEach

```
List<Integer> list=Arrays.asList(10,12,35);

Consumer<? super Integer> action = new Consumer<Integer>() {

        @Override
        public void accept(Integer i) {
                System.out.println(i);
        }
};

list.forEach(action);

list.forEach(i->System.out.println(i));

list.forEach(System.out::println);
```



And many more interfaces & classes…

# Common Java8 Functional Interfaces

- Predicate<T> - tests the T
- Consumer<T> - applies an action on the T
- Function<T,U> - given a T, returns a U (transformation)
- BiFunction<T,U,V> - transforms (T,U) into a V
- Supplier<T> - provides an instance of a T
- UnaryOperator<T> - a unary operator T → T
- BinaryOperator<T> - a binary oprator (T,T) → T

- java.util.function.*

# ForEach for maps

```
Map<String,Point> points=new HashMap<>();
points.put("init", new Point(0,0));
points.put("max", new Point(10,10));
points.put("min", new Point(-10,-10));

for(Entry<String,Point> e : points.entrySet()){
  System.out.println(e.getKey()+","+e.getValue());
}

points.forEach((K,V)->System.out.println(K+","+V));

points.keySet().forEach(K->System.out.println(K));

points.values().forEach(V->System.out.println(V));
```

# RemoveIf

```java
List<Double> list=new ArrayList(Arrays.asList(10.0,12.5,35.4));
for(Double d : list)
  if(d>15)
    list.remove(d);
```

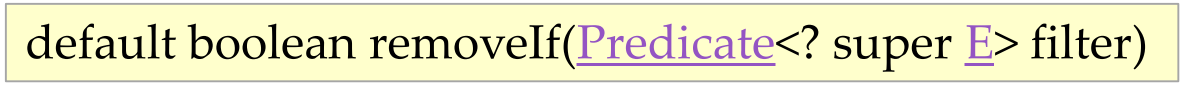
```
Exception in thread "main" java.util.ConcurrentModificationException
   at java.util.ArrayList$Itr.checkForComodification(Unknown Source)
   at java.util.ArrayList$Itr.next(Unknown Source)
```

# RemoveIf

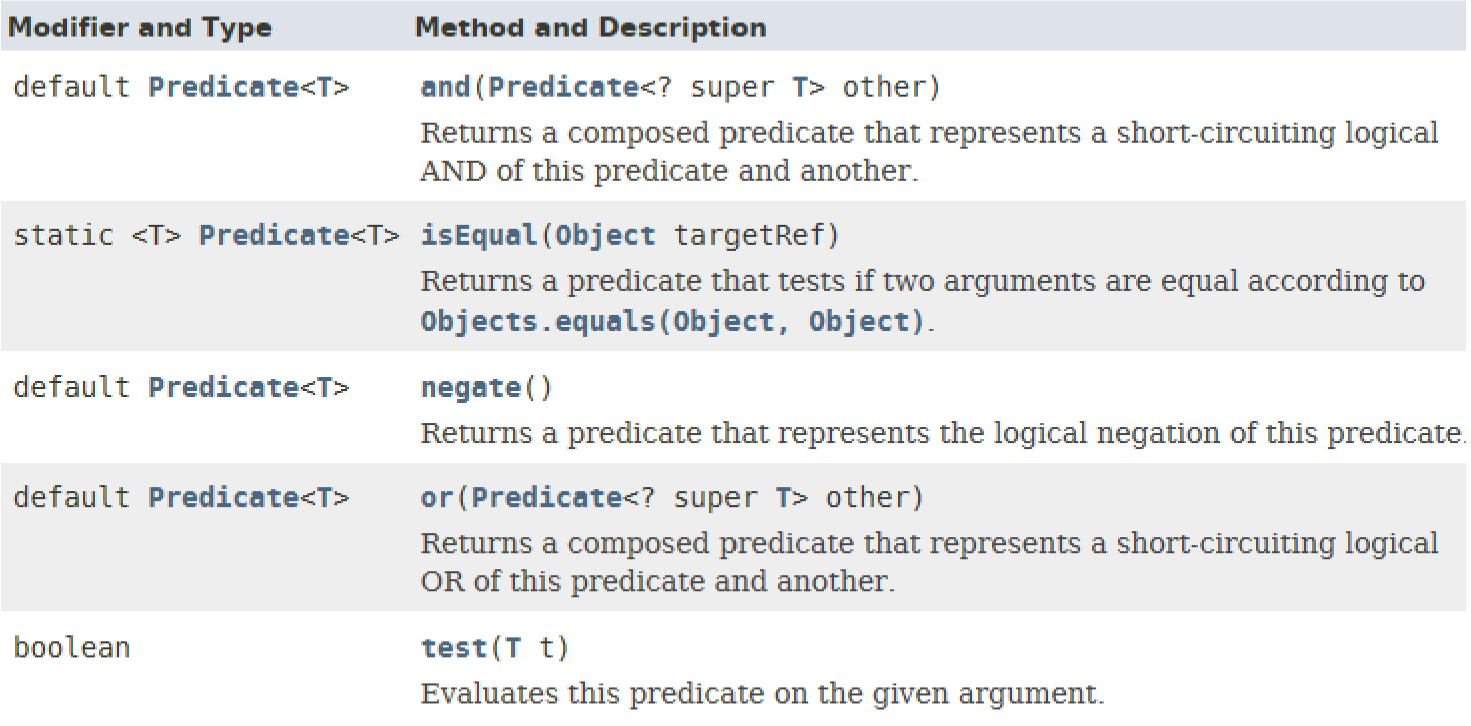
```
List<Double> list=new ArrayList(Arrays.asList(10.0,12.5,35.4));
List<Double> toBeDeleted=new LinkedList<>();

for(Double d : list)
   if(d>15)
      toBeDeleted.add(d);

list.removeAll(toBeDeleted);
```

# RemoveIf

```
List<Double> list=new ArrayList(Arrays.asList(10.0,12.5,35.4));
list.removeIf(d-> d>15);
```



default boolean removeIf(Predicate<? super E> filter)



| Modifier and Type | Method and Description |
|---|---|
| default Predicate<T> | and(Predicate<? super T> other)<br>Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another. |
| static <T> Predicate<T> | isEqual(Object targetRef)<br>Returns a predicate that tests if two arguments are equal according to Objects.equals(Object, Object). |
| default Predicate<T> | negate()<br>Returns a predicate that represents the logical negation of this predicate. |
| default Predicate<T> | or(Predicate<? super T> other)<br>Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another. |
| boolean | test(T t)<br>Evaluates this predicate on the given argument. |