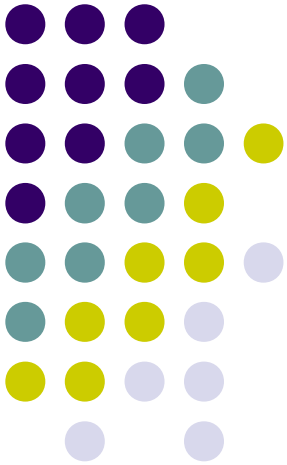
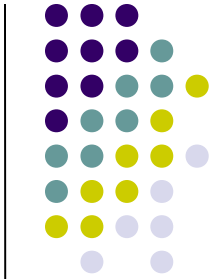
CO225: Software construction

# Collections

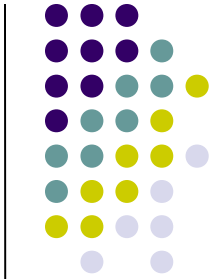


# Plan



- Look at some Java Collections
  - We already looked at them (lists, hashmaps..)
- And their applications
- (slides, thanks to Dr. Bandaranayake)

# Why and what?



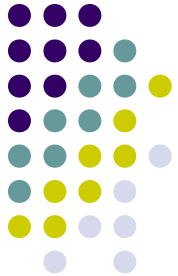
What is a collection?

- An object that groups multiple elements into a single unit, so that we can
  - Store/retrieve and manipulate
  - Transmit data from one methods to another
- Also called as containers

Why a collection?

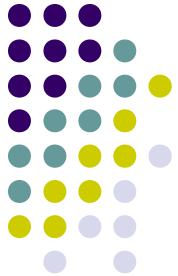
- Typical data belongs to a natural group
  - Example: list of students in a course, list of books in library
  - Different objects in a similar collection!

# Java Collections Framework (JCF)

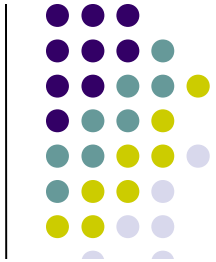


- Unified architecture for representing and manipulating collections, which includes:
  - Interface: how the collection should be manipulated
  - Implementation: implementation of the interface in a reusable manner
  - Algorithm: useful methods to be performed on the objects in the collection (example: sorting, searching etc.)
    - These algorithms are *polymorphic:* same method can be used on different objects

# Advantages of JCF



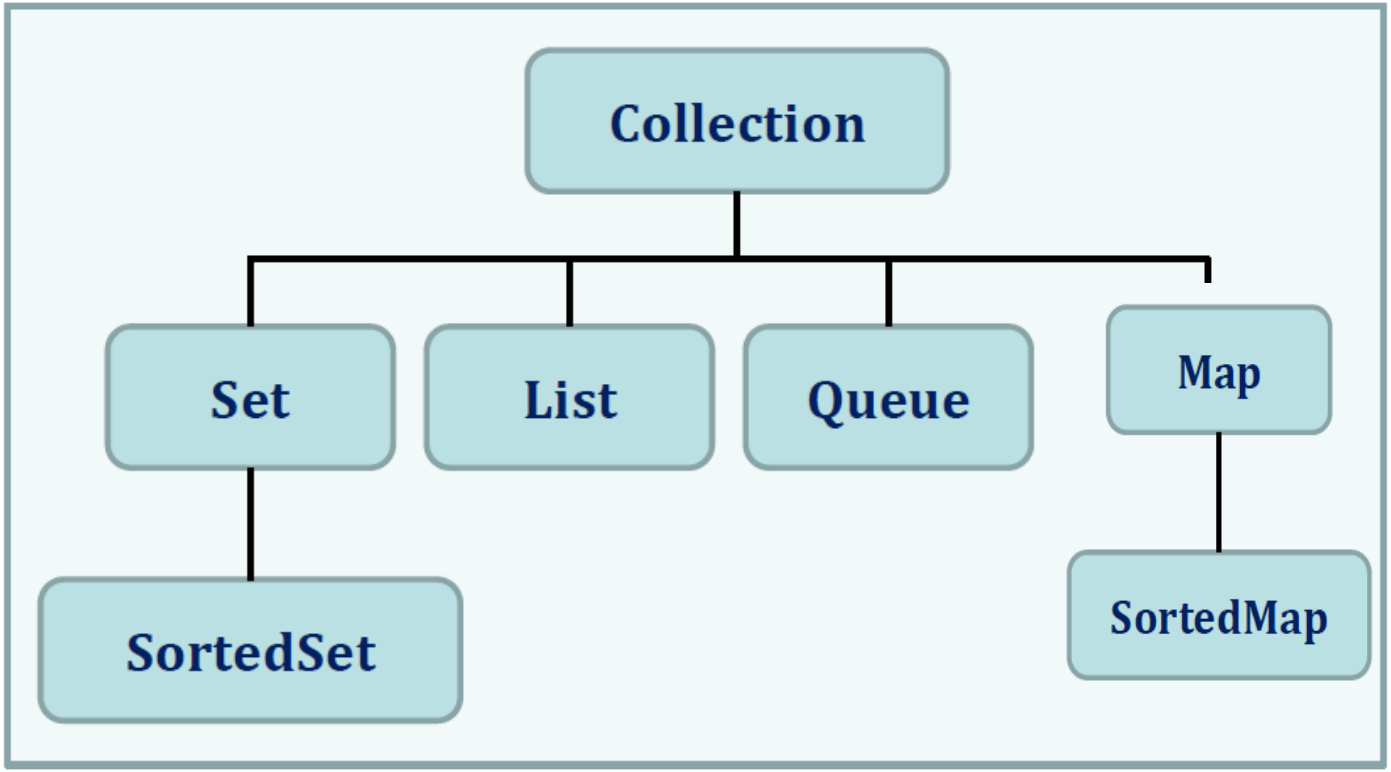
- Reduce programming effort
  - Work on important aspects that basics
- Increase program speed and quality
  - Reuse existing/time tested code
- Allows interoperability among unrelated APIs
- Foster software reuse



# Before moving on

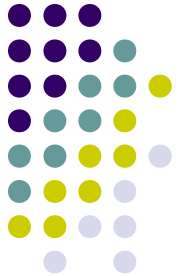
- You can find Java Collections in Package *java.util.*
- In this lecture we will survey the Java collection interfaces.
- We will not cover all of the details
- For additional details, please see
  - **Javadoc**, provided with your java distribution.
  - Comments and code in the specific **java.util.*.java** files, provided with your java distribution.
  - The **Collections Java tutorial**, available at http://docs.oracle.com/javase/tutorial/collections/index.html

# Collection interface hierarchy




- When you understand how to use these interfaces, you will know most of what there is to know about the Java Collections Framework

# Idea of an abstract class



You specify what the class should look like.

Without the implementation

Some class should implement that later

Example: see

Human.java

Student.java

EngStudent.java

# Idea of an abstract class

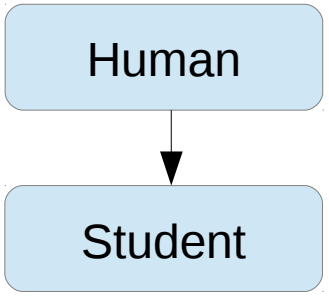
```
abstract class Human {
    private String name;
    Human(String name) {
       this.name = name;
    }
    abstract public void whoAmI(); // should be implemented
    public String getName() {
       return this.name;
    }
}
```

Human

```
Human a = new Human("Dhammika"); // will not work;
```

# Idea of an abstract class

```
abstract class Student extends Human {
    private String studentId;
    Student(String name, String studentid) {
       super(name);
       this.studentId = studentid;
    }
    public String getStudentId() {
       return this.studentId;
    }
}
```

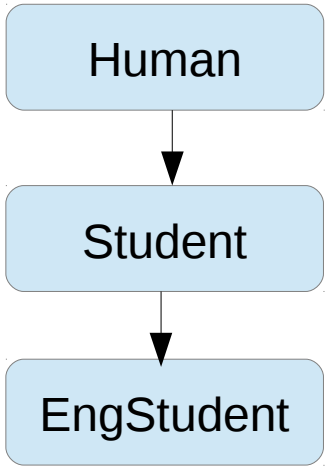



```
Human a = new Human("Dhammika"); // will not work;
Human b = new Student("Dhammika", "E100100");//will
not work
```

# Idea of an abstract class

```
class EngStudent extends Student {
    EngStudent(String name, String reg) {
       super(name, reg);
    }

    public void whoAmI() { // have to provide this method!!!
       System.out.println("I am " + getName() +
                          " Engineering student with reg "
                    + getStudentId());
    }
}
```

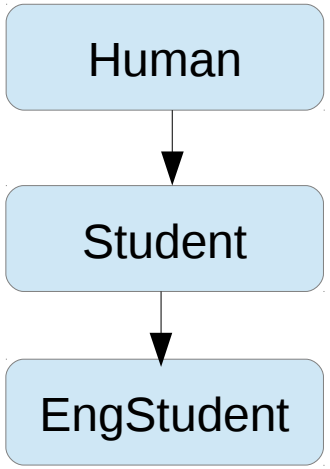


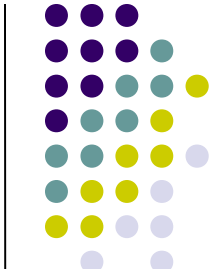

```
Human a = new Human("Dhammika"); // will not work;
Human b = new Student("Dhammika", "E100100");//will
not work
Human c = new EngStudent("Dhammika Elkaduwe",
"E100200");
c.whoAmI(); // You have implemented the interface
```

# Idea of an abstract class

- The Human class makes a promise of some functionality

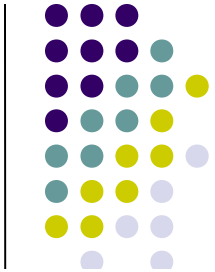
- EngStudent class implements that





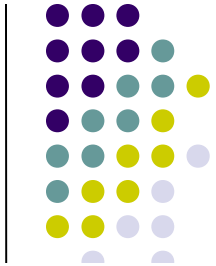
# Types of collection

- Java supplies several types of Collection:
  - **Set**: NO duplicate elements, order is not important
  - **SortedSet**: like a Set, but order is important
  - **List**: may contain duplicate elements, order is important
  - **Queue**: elements held for processing. Will have FIFO (first-in, first-out) or a "priority" first-out removal.
  - **Deque**: A double-ended Queue.
- Java also supplies some "collection-like" things:
  - **Map**: a "dictionary" that associates keys with values, order is not important
  - **SortedMap**: like a Map, but order is important



# Interface

- public int **size**();
  - Return number of elements in collection
- public boolean **isEmpty**();
  - Return true iff collection holds no elements
- public boolean **add**(E x);
  - Make sure the collection includes x; returns true if collection has changed (some collections allow duplicates, some don't)
- public boolean **contains**(Object x);
  - Returns true if & only if collection contains x (uses equals( ) method)
- public boolean **remove**(Object x);
  - Removes a single instance of x from the collection; returns true if collection has changed.



# Interface

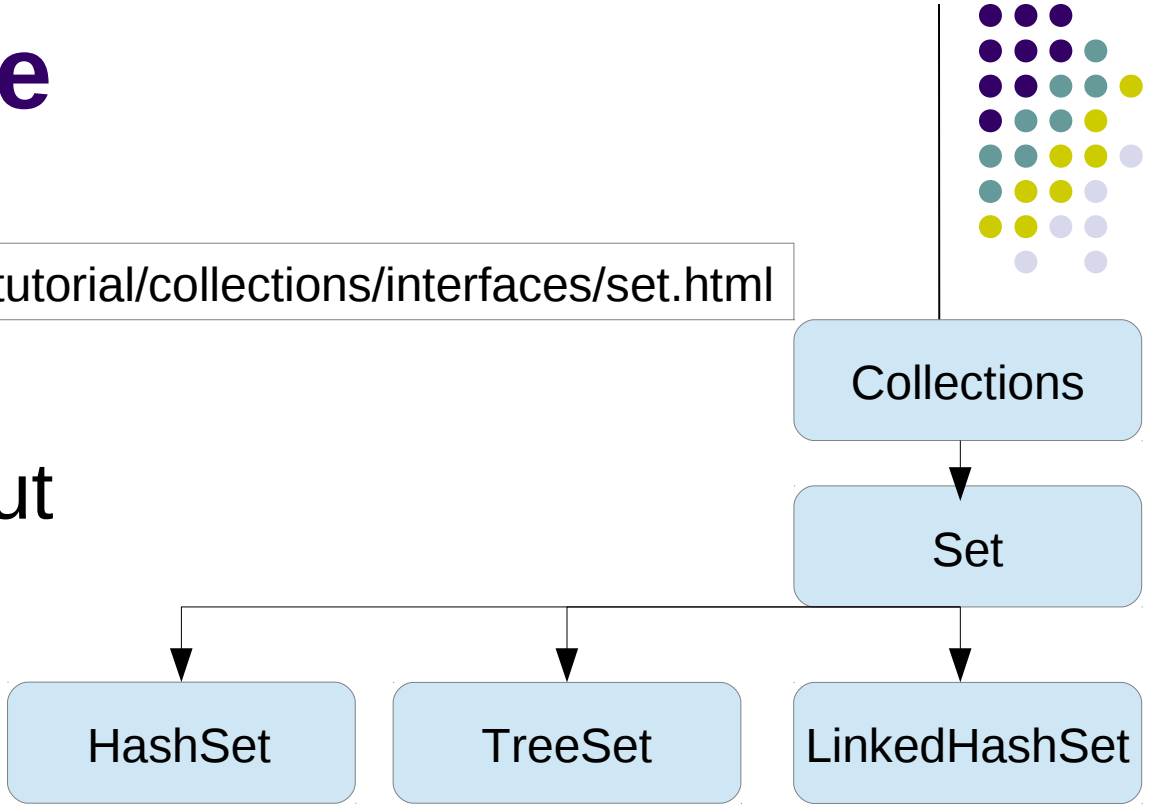
- **public Iterator<E> iterator();**
  - Returns an Iterator that steps through elements of collection
- public Object[] **toArray**()
  - Returns a new array containing all the elements of this collection
- public <T> T[] **toArray**(T[] dest)
  - Returns an array containing all the elements of this collection; uses dest as that array if it can
- Bulk Operations (very powerful!):
  - public boolean **containsAll**(Collection<?> c);
  - public boolean **addAll**(Collection<? extends E> c);
  - public boolean **removeAll**(Collection<?> c);
  - public boolean **retainAll**(Collection<?> c);
  - public void **clear**();

# First example
# Sets

https://docs.oracle.com/javase/tutorial/collections/interfaces/set.html

Set: collection of elements without duplicates

Abstract set is implemented by: HashSet, TreeSet, LinkedHashSet



# Sample use of a set

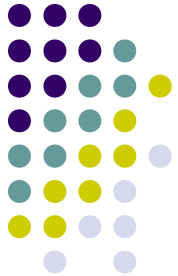
See: SampleSet.java

```
class SampleSet {
    public static void main(String [] args) {
       int [] data = {11, 123, 3, 14, 23, 3, 412, 3, 2};
       Set<Integer> set = new LinkedHashSet<Integer>();
       Collection s = new LinkedHashSet<Integer>();

       for(int i=0; i<data.length; i++) {
           set.add(data[i]);
           s.add(data[i]);
           System.out.printf("Inserted %d element, have %d in set\n",
                        i+1, set.size());
       }

       System.out.println(set);
       System.out.println(s);
  }
}
```

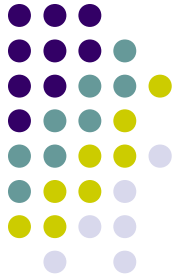
# Traversing collections



- A There are ~~two~~ three ways to traverse collections:
  - Using **Iterators**.
  - With the (enhanced) ***for–each*** construct
  - Using **Aggregate Operations**:
    - Preferred method of iteration in JDK 8 & later
    - Often used in conjunction with *lambda expressions* – another new feature in JDK 8.

# Iteration example

See IterationEx.java



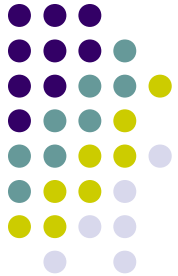
```
class IterationEx {
    public static void main(String [] args) {
       int [] data = {11, 123, 3, 14, 23, 3, 412, 3, 2};
       Set<Integer> set = new LinkedHashSet<Integer>();
       for(int i=0; i<data.length; i++)  set.add(data[i]);

       Iterator<Integer> it = set.iterator();
       while(it.hasNext())
           System.out.println(it.next());
    }
}
```

Synatx:

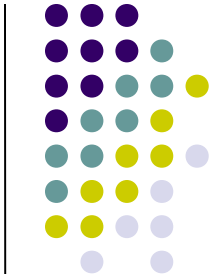
```
Iterator<E> it = collection.iterator();
while (it.hasNext())
        System.out.println(it.next());
```

# java.util.Iterator<E>



- public boolean hasNext()
  - returns true if the iteration has more elements
- public E next()
  - returns the next element in the iteration.
  - throws exception if iterator has already visited all elements.
- public void remove()
  - removes the last element that was returned by next.
  - remove may be called only once per call to next
  - otherwise throws an exception.
  - Iterator.remove() is the only safe way to modify a collection during iteration

# *For-each* construct



- Suppose collection is an instance of a Collection. Then

```
for (Object o : collection)
    System.out.println(o);
```

  - prints each element of the collection on a separate line.
- This code is just shorthand:
  - it compiles to use `o.iterator()`.
- But, the for-each construct hides the iterator
  - You cannot call remove.
  - Therefore, the for-each construct is not usable for filtering.

# Example: for-each

```
class ForEachSample {
    public static void main(String [] args) {
       int [] data = {11, 123, 3, 14, 23, 3, 412, 3, 2};
       Set<Integer> set = new LinkedHashSet<Integer>();
       for(int i=0; i<data.length; i++)  set.add(data[i]);

       for(Object o: set)
           System.out.println(o);
    }
}
```

# More on the set interface

- A **Set** is a Collection that has no duplicate elements
- Set extends collection
  - contains *only* methods inherited from Collection.
  - If you attempt to add() an element twice then the second add() will return ***false*** (i.e., the Set has not changed)

| | | |
|---|---|---|
| S1.**containsAll**(S2); |  | **Subset** |
| S1.**addAll**(S2); | | **Union** |
| S1.**retainAll**(S2); | | **Intersection** |
| S1.**removeAll**(S2); | | **Set difference** |