Outline

- Prolog
- Introduction
- The Hierarchy
- Code Examples: Person
- The Map Hierarchy
- Utility Classes
- Optional Topics
  - Practical metaphor for contracts and type covariance
  - Streams and Lambdas
  - Range Class
We often mention “type”, but what is it?
- Refers to both interfaces and classes
- “Subtype” refers to:
  - An interface that extends another
  - A class that implements an interface
  - A class that extends another class
Introduction

- Java offers many collection objects
- **Java Collection Hierarchy**
- Characteristics of collection objects:
  - All are generic…
    - must tell it what you are storing
  - If you store type E… you can also store any of its subtypes.
  - Ways of iterating over their elements
Introduction

- Most basic interface **Collection**
- Requires:
  - Adding/removing elements
  - Computing size

```java
public interface Collection {
    boolean add(Object element);
    Collection addAll(Collection collection);
    void clear();
    boolean contains(Object element);
    boolean containsAll(Collection collection);
    boolean equals(Object object);
    int hashCode();
    Iterator iterator();
    boolean remove(Object element);
    Collection removeAll(Collection collection);
    Collection retainAll(Collection collection);
    int size();
    Object[] toArray();
    Object[] toArray(Object[] array);
}
```
Introduction

• All collections offer two ways of iterating
  • New way: Via streams… will discuss this later
  • Old way: Implement an iterator
    • Can be seen as a cursor
    • Cursor is pointing at an element
    • Starts at “first” element in the collection
      • “first” can mean different things
  • Java Collections must be able to create iterators
    • Done via a method called `iterator`
      • `Collection<SomeClass>` gives you an `Iterator<SomeClass>`
Introduction

- Iterators have only 4 methods
  - `next()` - gives you the next elements
    - Also advances the iterator
  - `remove()` - gets rid of the next element
    - Not always implemented
    - New state of iterator depends on the Collection class that made the iterator
  - `hasNext()` - tells you if there are more elements
  - `forEachRemaining()` - will talk about this later
Introduction

- Iterators have only 4 methods
- First and third operations allow for traditional “for” loop

```java
Collection< Foo > coll = ......;
for ( Iterator< Foo > iter = coll.iterator(); iter.hasNext(); ) {
    Foo f = iter.next();
    // Do something with f
}
```

Note: Empty third clause in “for” loop…
`next()` handles this

- Since `Collection` extends `Iterable` interface

```java
for ( Foo f: coll ) {
    // Do something with f
}
```

Note: the item after the colon must be:
- A primitive array
- Item that implements `Iterable`
The Hierarchy

• Recall: Java Collection Hierarchy
  • Each subtype of Iterable adds more and more…
• All Collection subtypes describe one-dimensional collections
  • Because they are Iterable
The Hierarchy

- Classifications of Collections:
  - *Unorderable* collections
    - Order is not guaranteed
    - Example: *HashSets*
  - *Manually orderable* collections
    - Elements have an order, but...
    - It can be changed
    - Example: *Lists*
  - *Automatically ordered* collections
    - Data structure determines order
    - Example: *TreeSets* and *PriorityQueues*
The Hierarchy

- Highlights of *Collection* Hierarchy:
  - *Sets* do not allow duplicates
  - *HashSets* implemented using hash tables
  - *NavigableSets* maintain their own element order
    - based on natural ordering or a *Comparator* object
  - *TreeSet* implements *NavigableSet*
  - *TreeSets* implemented using red-black trees.
    - basically a binary search tree, but...
    - has functionality to maintain $O(\log N)$
  - *Lists* allow access via indices
  - *ArrayList* use contiguous memory
    - like primitive arrays
  - *LinkedLists* use linked nodes
Code Example: Person

• To the code!
  • Versions 1-4:
    • Create **Person** instances
    • Put them in a collection
      • Ordering them one way or another
    • Demonstrate various ways to do the same thing
  • Versions 5-6: Will get to later
• Things to note:
  • **TreeSet** depends heavily on the **Comparable** or **Comparator** class
  • **HashSet** needs Object’s **hashCode()** method
    • **equals()** method is also important
  • In Person the **equals()** method was design from the beginning
    • needs to return true when **compareTo()** returns 0
      • false when non-zero
The Map Hierarchy

- **Maps** have a separate hierarchy
  - **Maps** are similar to “dictionaries”
  - Lists of pairs of values
    - Maps one value to another
  - Input value is the “key”
  - Output value is the “value”
    - We typically call them $K$ and $V$ went doing generics
- Recall **Lists**, or arrays, are a simple form of a map
  - keys are small integers (indices)
  - values are what ever is at the location
- So **List**<E> can be seen as **Map**<Integer,E>…
  - Sort of...
The Map Hierarchy

- A simplified version of the Collection Hierarchy
  - the variations now apply to keys not values
  - The Map Hierarchy
- Example: Person version 5
  - changes Person class
    - only stores name now
  - A map is used to look up a person’s age
    - many similarities between prior examples
Utility Classes

• Other classes worth review
  • Collections and Arrays
    • contain many static methods
  • Collections:
    • contains method that search, rearrange, and enhance various Collection objects
  • Arrays:
    • contains methods for converting primitive arrays to Lists
    • beyond this it gives a lot of list functionality to arrays
Optional Topics

- High-level can do...
  - level cannot
  - Examples:
    - raise `UnsupportedOperationException`
    - “Design by Contract”
    - Why is it not `NavigableSets< T extends Comparable<T> >`?
      - covariance bad
      - allowance of external Comparator
Optional Topics

- Practical metaphor for contracts and type covariance
  - Say you hired Jake to rebuild your kitchen
  - You both signed a contract stating what is to be done
    - and what it will cost
  - Under cost Jake wrote the types of accepted payments
    - Cash, check, and credit card
  - Jake hired a SUBcontractor Tony to do the work
    - Jake said you should treat Tony as him
  - The job is done
    - You offer Tony your credit card
    - But Tony only accepts cash
  - The contract has been violated
Optional Topics

- Let's imagine this as code
  - We have a Payment interface
    - with implementing classes Cash, CreditCard, and Check
  - Your contractor provided you with:

```java
class Contractor {
    void acceptPayment( Payment p ) { ... }
    :
    :
}
```

- Along comes Tony:

```java
class SubContractor extends Contractor {
    @Override
    void acceptPayment( Cash p ) { ... }
    :
    :
}
```
Optional Topics

- You get an instance of that class instead…
  - `Contractor myContractor = new SubContractor(“Tony”);`
  - using “is-a” logic you assume nothing have changed
- You try to pay…
  - `myContractor.acceptPayment( creditCard );`
  - “the system blows up!”
- Java prevents this
  - You cannot further constrain the types of incoming values
  - also known “narrowing” the type
  - This is why you can do `Set<T>`
  - and not `SortedSet< T extends Comparable< T > >`
  - this adds a new constraint to the subclass
- This is known as “covariance”
  - narrowing both type and parameter to the type
Optional Topics

• Streams and Lambdas
  • Recall: Collection implements Iterable
  • Iterable interface provides a void foreach( Consumer<E>)
• Let’s use Consumer to introduce “functional interfaces”
  • a very special group of interfaces
  • they have only one abstract method, their “function”
  • Classes that implement these are basically wrappers around simple functions
• They reside in the java.util.function package
Optional Topics

- Consumer interface

```java
interface Consumer< E > {
    void accept( E element );
    default Consumer< E > andThen( Consumer< E > after ) { .... }
}
```

- andThen is a more advanced operation
  - allows you to chain Consumers together
  - Do not worry about it in this course
  - It has a default implementation
    - So Consumer qualifies as a functional interface
Optional Topics

- Example: Convert a normal collection loop into an application of `forEach`
  - we will print one element per line

```java
Collection<Integer> numbers = .... ;
for ( int n: numbers ) {
    System.out.println( n );
}
```

- How to do this as a `forEach`?
  - You are not going to like it... at first

```java
class IntPrinter implements Consumer<Integer> {
    public void accept( Integer n ) {
        System.out.println( n );
    }
}
```

```
Collection<Integer> numbers = .... ;
numbers.forEach( new IntPrinter() );
```

- Looks like a lot of work! Java does have some shorthand...
Optional Topics

- Shorthand Techniques
  - None make you explicitly declare a class
  - Let's start with one that requires the most typing
  - then go to the least...

```java
numbers.forEach(new Consumer<Integer>() {
    public void accept(Integer n) {
        System.out.println(n);
    }
});

numbers.forEach((Integer n) -> { System.out.println(n); });

numbers.forEach(n -> System.out.println(n));

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

- How does this all work!!!
  - It is magic!
Optional Topics

- It’s actually not magic!
  - It is the Java compiler
  - It is super smart
  - It can infer things
    - Like how to build a class from a small amount of code
- Maybe this way is better than writing that “for” loop?
  - This simplification may not always work
    - Depends on the complexity.
  - As complexity goes up… the need for a normal loop goes up
    - but use the newer loop style whenever possible
Optional Topics

- Lambdas
  - The last three are called lambdas
    - or an unnammed function
  - They look like simple functions…. not classes

```java
numbers.forEach( new Consumer< Integer >() { 
    public void accept( Integer n ) {
        System.out.println( n );
    }
});

numbers.forEach( ( Integer n ) -> { System.out.println( n ); } );

numbers.forEach( n -> System.out.println( n ) );

numbers.forEach( System.out::println );
```
Optional Topics

- Person version 6 example
  - Shows version 4
    - all small classes replaced with lambdas
      - all in the main code now!
  - It may be worth it some times…
    - others it may not be
- In this course you can do what ever form you want…
  - As long as you are not told otherwise
Optional Topics

- The stream() method
  - returns something called a “Stream” on the Collection
  - Streams can do a forEach()
    - `numbers.stream().forEach( System.out::println );`
- Why bother making a Stream?
  - Streams can do a lot more!
  - We will introduce only one: `filter()`
    - Takes in another functional interface
    - Called a `Predicate`

```java
interface Predicate< E > {
    boolean test( E element );
}
```

- It removes things that fail the `test()`
- Example in lecture code...
Optional Topics

- The Range class
  - Designed to mimic as closely as possible the Python `range()`
  - It is iterable
Questions?