Advanced Java

Implementing for-each loops
public class HelloWorld {
    public static void main(String[] args) {
        System.out.print("Hello, World!");
    }
}
Iterating over an array

... int[] array = new int[5];
for (int i = 0; i < 5; i++) {
    array[i] = 2 * i;
}
...

...
Iterating over an ArrayList

...  
ArrayList<Integer> primes = new ArrayList<Integer>();
primes.add(2);
primes.add(3);
primes.add(5);
primes.add(7);

for (Integer p : primes){
  System.out.println(p * 2);
}
...
...
Goals

- Benefit from the *for-each* cool syntax in our user defined classes
- Learn several important concepts in Java along the way
A toy example

• By the end of this recitation, the following code will be valid:

```java
Fibonacci fib = new Fibonacci(0, 1, 10);

// Print the first 10 elements of Fibonacci series
// with seed values (0, 1)
for (Integer current : fib) {
    System.out.println(current);
}
```
Introducing: Interfaces

- An interface is a reference type, similar to a class that can contain only\(^1\):
  - constants
  - method signatures
  - default methods
  - static methods
  - nested types

\(^1\) https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html
Interfaces vs. Inheritance

● Thumb rule:
  ○ Inheritance upholds the relation “Is a”
  ○ Interface upholds the relation “Can do”

● A class can extend (inherit) a single class, but can implement many interfaces
Creating a simple interface

// File: Sized.java

public interface Sized {
    public int size();
}

Implementing an interface

// File: CountersArray.java
public class CountersArray implements Sized {
    int numOfElements; int[] array;

    public CountersArray(int range) {
        numOfElements = 0; array = new int[range];
    }

    public void add(int element) {
        array[element] += 1; numOfElements++;
    }

    public int size() { // Must implement all methods from interface
        return numOfElements;
    }
}
The Iterable<T> interface

- Contains only one method named iterator:
  - Iterator<T> iterator()

- Iterator<T> is also an interface
The Iterator<T> interface

- Contains three methods:
  - boolean hasNext()
  - T next()
  - void remove()
Integer vs. int

- **int** is a primitive type following **value semantics**
- **Integer** is a class and therefore follows **reference semantics**
- Every primitive type in Java has an equivalent wrapper class
Several comments on Integer

- Prefer `int` for better performance

- Generics require `reference types`

- Integer supports:
  - Standard math operations
  - Comparison to null
  - Many more useful methods
Implementing Fibonacci class

// Our class
public class Fibonacci implements Iterable<Integer> {
    // Private members for our class
    private int a, b, n;

    // Fibonacci Constructor
    public Fibonacci(int a, int b, int n) {
        this.a = a;
        this.b = b;
        this.n = n;
    }

    public Iterator<Integer> iterator() {
        // Returns a class implementing Iterator<T>
        return new FibonacciIterator();
    }

    ...
Inner classes

- We can define `FibonacciIterator` as a regular class (i.e. as a separate class)
- Java also allows defining classes within classes (i.e. inner classes)
- Useful for grouping related classes
Advantages of inner classes

● Inner classes can access all members and methods defined by the outer class
  ○ Shortens and simplifies the code

● Useful if only one instance is needed
Inner class implementation

...  
// Inner class
private class FibonacciIterator implements Iterator<Integer> {
   // Private members for inner class
   private int current, next, counter;

   // Inner class constructor
   public FibonacciIterator() {
      this.current = Fibonacci.this.a;
      this.next = Fibonacci.this.b;
      this.counter = 0;
   }

   public boolean hasNext() {
      return this.counter < Fibonacci.this.n;
   }
   ...
}
... public Integer next() {
    if (this.hasNext()) {
        int current = this.current;
        this.current = this.next;
        this.next += current;
        this.counter++;
        return current;
    }
    throw new NoSuchElementException();
}

public void remove() {
    throw new UnsupportedOperationException();
}

...
Anonymous classes

- Instead of defining an **inner class**, Java allows us to create an instance of an **anonymous class**

- It saves us the trouble of giving the class a name
Implementing using anonymous class

...  
public Iterator<Integer> iterator() {  
    return new Iterator<Integer>() {  
        private int current, next, counter;  
        {  
            this.current = Fibonacci.this.a;  
            this.next = Fibonacci.this.b;  
            this.counter = 0;  
        }  
        // Replaces a constructor with an instance initializer  
        public boolean hasNext() { /* Same implementation as before */ }  
        public Integer next() { /* Same implementation as before */ }  
        public void remove() { /* Same implementation as before */ }  
    };  
};  
...
Main: The long version

... public static void main(String[] args) {
    Fibonacci fib = new Fibonacci(0, 1, 10);
    
    Iterator<Integer> it = fib.iterator();
    while (it.hasNext()) {
        int current = it.next();
        System.out.println(current);
    }
}
Main: The elegant version

...  
public static void main(String[] args) { 
    Fibonacci fib = new Fibonacci(0, 1, 10);

    for (Integer current : fib) {
        System.out.println(current);
    }
}