Java

Iterators, nested classes & functions
Lesson overview

- Function objects in java
- Inner classes
- Java Iterators
Lesson overview

- Function objects in java
- Inner classes
- Java Iterators
Closures in JAVA?

- JAVA functions are second class citizens
- JAVA does not have real closures
  - We will see how we can imitate closures in java
Reminder - Function Objects

```java
class Log {
    final double logBase; // Captured environment
    Log(final double base) {
        logBase = Math.log(base);
    }
    public double apply(double v) {
        return Math.log(v) / logBase;
    }
}
```
public class L {
    public static void main(String[] args) {
        final Log log2 = new Log(2);
        System.out.println("Log base 2 of 1024 is " + log2.apply(1024));
    }
}
Interface for Function Objects

Abstract function type

```java
interface Function<T, R> { R apply(T t); }
```
Interface for Function Objects

class LOG implements Function<
    Double,
    Double //
> {
    // captured environment:
    final Double logBase;
    LOG(final Double base) {
        logBase = Math.log(base);
    }
    public Double apply(Double t) {
        return Math.log(t) / logBase;
    }
}
Lesson overview

- Function objects in java
- Inner classes
- Java Iterators
Smarter - Inner classes

The trick - **inner classes** close on the environment

```java
public class C {
    static Object foo(final int a) {
        class Inner {
            public int hashCode() {
                return (a-1)*a*(a+1)*(2*a + 3);
            }
            return new Inner();
        }
        return foo(a);
    }
    public static void main(String[] args) {
        Object o = foo(6);
        ... System.out.println(o.hashCode());
    }
}
```
Lesson overview

● Function objects in java
● Inner classes
● Java Iterators
Emulating generators using Iterators

- Up to java 7:
  - No closures
  - No generators

- We have seen how to emulate closed and now we will try to simulate generators using iterators.
Java Iterators

We shall have two classes:

- Main class which creates and then uses the iterator.
  - Will create the “environment” for the our poor-man’s generator
  - Environment includes the values 2 and 30. in the C# call Powers(2,30)
- class Powers the iterator object itself.
  - Will save the “environment” that function main passes on to it.
  - Will save the local variables of the generator as class variables.
Invoking the iterator

```
Import java.util.Iterator;

Public class E {
    Public static void main(String[] args) {
        Integer p;
        Iterator<Integer> it;
        for(it = new Powers(2,30);it.hasNext(); ) {
            P = it.next();
            System.out.println(p);
        }
    }
}
```
class Powers implements Iterator<Integer> {
    //saved environment:
    Private final Integer base, count;
    //saved local state:
    Private Integer result = 1, i=0;
    Public PowersIterator(
        //Environment is passes with arguments
        Integer base, Integer count
    ) {
        //Save the environment
        This.base = base; this.count = count
    }
}
class Powers implements Iterator<Integer> {
    ...
    //at each iteration:
    Public Integer next() {
        Result *= base;
        ++i;
        return result;
    }
    //Termination test:
    Public boolean hasNext() {
        return i < count;
    }
}
Iterable vs. Iterator

**Definition (Iterable in JAVA)**

- something on which “iteration” is possible.
- e.g., a list, a set,
- e.g., an arithmetical progression,
- factory of Iterators

**Definition (Iterator in JAVA)**

- provides a service of “iteration” on an Iterable
- at each step of the “iteration”:
  - if there is a “next” item
  - yields the “next” item
Iterable vs. Iterator

- The only thing an **Iterable** does:
  - generates Iterators
- An Iterator is the ultimate disposable construct:
  - provides iteration.
    - single run
    - forward only
  - associated with an **Iterable**:
    - from birth, with **one**, and only **one**
- An **Iterable** can be associated with one **Iterator**, many **Iterators**, or **none** at all.
- An Iterator may also **remove** an item from an Iterable.
  - we will try to ignore this historical accident
Syntactic sugar

In order to use this, we must define a class which implements `Iterable<Thing>`

```java
Foreach syntax
for (Thing thing: things)
    doSomethingWith(thing);

Foreach semantics
Iterator<Thing> it = things.iterator();
while (it.hasNext()) {
    Thing thing = it.next();
    doSomethingWith(thing);
}
```
Pseudo generator

Definition (Pseudo generator)

A pseudo generator is a Java function that returns an `Iterable`;

A pseudo generator

```java
// pseudo generators are almost always public static:
public static
    // a pseudo generator must return an Iterable:
    Iterable<Integer>
powers(final Integer Base, final Integer count) {
    return new POWERS(base, count);
}
```

Looks just like a generator

```java
for (Integer p: powers(2,30))
    System.out.println(p);
```
# Generators vs. Iterators

<table>
<thead>
<tr>
<th></th>
<th>Generators</th>
<th>Iterators</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PL</strong></td>
<td>C#, Python, Icon,...</td>
<td>Java</td>
</tr>
<tr>
<td><strong>Purpose</strong></td>
<td>sequence of values</td>
<td>sequence of values</td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
<td>language construct</td>
<td>by clever programmer</td>
</tr>
<tr>
<td><strong>Underlying concept</strong></td>
<td>function</td>
<td>object</td>
</tr>
<tr>
<td><strong>Environment</strong></td>
<td>builtin</td>
<td>must be managed manually by programmer</td>
</tr>
<tr>
<td><strong>Control</strong></td>
<td>forward</td>
<td>inverted</td>
</tr>
<tr>
<td><strong>Conceptual complexity</strong></td>
<td>low</td>
<td>high</td>
</tr>
<tr>
<td><strong>Pedagogical value</strong></td>
<td>low</td>
<td>high!</td>
</tr>
</tbody>
</table>