CSC301

Iterators & Lambda Expressions

Iterators

- An iterator is an object that lets you traverse (i.e., go through) items in a collection, without accessing the collection itself.
    - Extremely common *design pattern*
    - Interface might vary slightly between programming languages

Iterator in Java

- Part of the Java Collections Framework
- Generic interface `Iterator<T>`
    - `hasNext()`
        - Returns a boolean
        - Indicates whether there is a next item
    - `next()`
        - Return the next item (of type `T`)
        - The type of the item depends on the collection we're iterating over.

          In Java, you can define a generic collection

Iterables

- We usually distinguish between two concepts:
  - `Iterable<T>`, a collection of items that can be traversed using an iterator.
  - `Iterator<T>`, a "utility object" used for traversing an iterable collection.
- In Java, you can use iterables in a for-each loop

```
for (T item : iterable){
    // Loop body ...
}
```

- *Note:* Iterables are **not** a Java-specific concept. For example, the same distinction exists in Python (although it is sometimes a little less clear).

Why Iterators?

- Modularity
    - Looser dependencies - Don’t depend on a specific collection
    - Changing an underlying collection (e.g., instead of a list, use a tree or a set) does not require changes in other pieces of the code
- In some cases, memory efficiency
    - Generate a large (or even unbounded) sequence of items, using little memory space
- Convenience
    - Abstract implementation details such as network communication, caching or lazy-evaluation. E.g.: Infinite scrolling, database cursor
- Clear & Explicit Design
    - Indicates that your code only needs a way to traverse the items, nothing more
      Using Math terminology … Your solution is stronger, because it makes fewer assumptions

Code Examples

- An iterator that generates a range of integers
  - Q: What is the space complexity?
    In other words, how much memory does this iterator use?
- Similar, yet slightly more flexible version
  - Pass the step-size as a constructor argument
  - Use default values for the starting point and/or step-size

Lambda Expressions

- Lambda expressions conveniently define a function inline
    - Were introduced to Java in version 8
    - Avoid the need to create (anonymous) class, when all you need is a function
    - Together with *Functional Interfaces*, they add *functional programming* capabilities to Java
- The same concept exists in many programming languages

Code Examples

- General purpose examples:
  - Mapping iterator that uses a Function
  - Filtering Iterator that uses a Predicate
  - Disclaimer: These are just code examples.
    In a real project, you should check whether the language/framework you are using already offers such general-purpose functionality.
- A couple of examples based on last term's test
  - `Iterator<Product>` represents an assembly line, where products come one at a time.
  - Batch products based on different criteria (specified via lambda expressions)
  - BatchIterator
  - BatchIterator2 - Look ahead and make sure a batch does not go over the weight limit.
    (this example assumes that no single product weighs more than the weight limit)