CSC207 - Week 11

Ilir Dema

Summer 2016

Topics

1. Singleton Pattern
2. Iterator Pattern
3. Regular Expressions

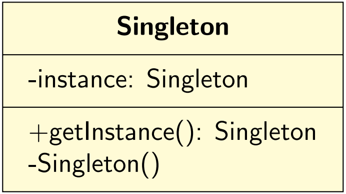
   Note: Those are the last topics to appear on the final.

   Next week: Floating Point numbers, Review.

Singleton Pattern

- Context
  - Classes for which only one instance should exist (singleton).
  - Provide a global point of access.
- Problem
  - How do you ensure that it is never possible to create more than one instance of a singleton class?
- Forces
  - The use of a public constructor cannot guarantee that no more than one instance will be created.
  - The singleton instance must be accessible to all classes that require it.

Singleton: Solution



Singleton

-instance: Singleton

+getInstance(): Singleton
-Singleton()

Clients access a `Singleton` instance solely though `Singleton's` `getInstance()` operation.

Serialization

If the Singleton class implements the `Serializable` interface, when a singleton is serialized and then deserialized more than once, there will be multiple instances of Singleton created. In order to avoid this the `readResolve` method should be implemented.

```java
public class Singleton implements Serializable {
    // Some code
    // This method is called immediately after
    // an object of this class is deserialized.
    // This method returns the singleton instance.
    protected Object readResolve() {
        return getInstance();
    }
}
```

What is an iterator

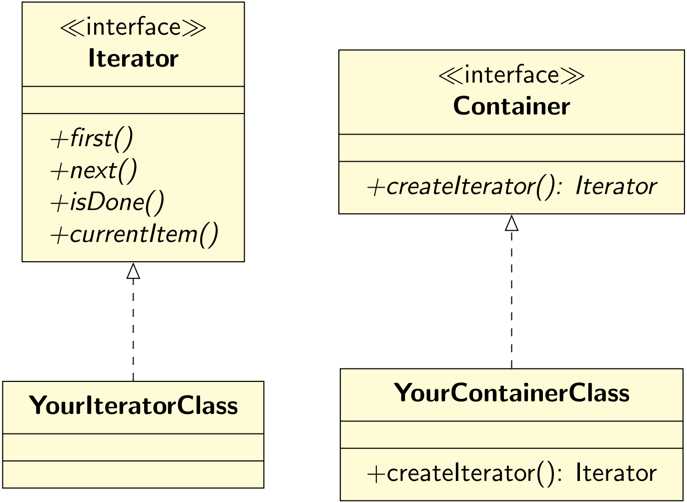
An iterator is a mechanism that permits all elements of a collection to be accessed sequentially, with some operation being performed on each element. In essence, an iterator provides a means of ”looping” over an encapsulated collection of objects. Examples of using iterators include

- Visit each file in a directory (aka folder) and display its name.
- Visit each node in a graph and determine whether it is reachable from a given node.
- Visit each customer in a queue (for instance, simulating a line in a bank) and find out how long he or she has been waiting.
- Visit each node in a compiler’s abstract syntax tree (which is produced by the parser) and perform semantic checking or code generation.

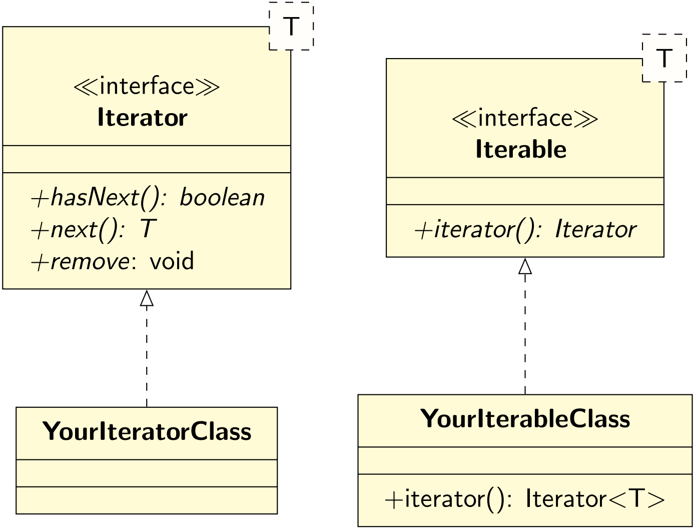
Iterator Design Pattern

- Context
  - A container/collection object
- Problem
  - Want a way to iterate over the elements of the container.
  - Want to have multiple, independent iterators over the elements of the container.
  - Do not want to expose the underlying representation (i.e., should not reveal how the elements are stored).

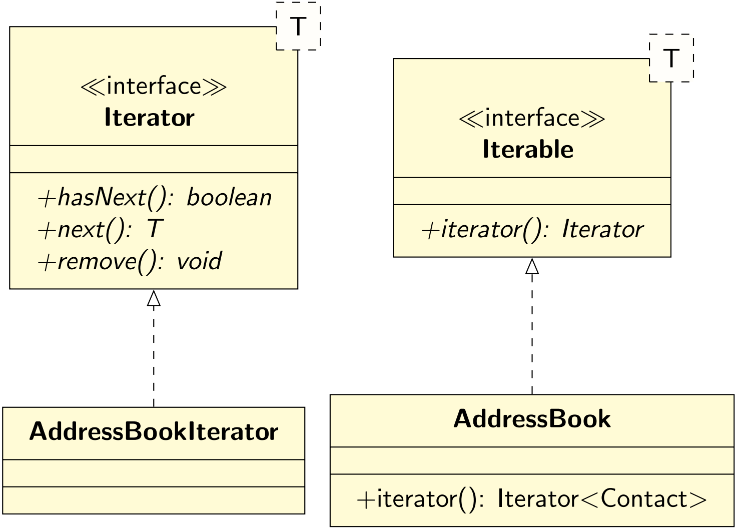
Iterator Design Pattern: Solution



Iterator Design Pattern: Java



Iterator: Example in Java



Regular expressions

- A **regular expression** is a pattern that a string may or or may not match.
- Example: $[0-9]+$

  $[0-9]$ means a character in that range
  $+$ means one or more of what came before
  Strings that match: 91254
  Strings that dont: abc empty string
- Symbols like $[,]$, and $+$ have special meaning. They are not part of the string that matches. (If we want them to be, we escape them with a backslash.)

Example Uses

- Handling white space
  - A program ought to be able to treat any number of white space characters as a separator.
- Identifying blank lines
  - Most people consider a line with just spaces on it to be blank.
- Validating input
  - To check that input is has the expected format (e.g., a date in DD-MM-YYYY format).
- Finding something within some input
  - E.g., finding dates within paragraphs of text.

Who do we need patterns?

- We could accomplish those tasks without using patterns.
- But its much easier to declare a pattern that you want matched than to write code that matches it.
- Therefore many languages offer support for this.
- Bonus: By having the pattern explicitly declared, rather than implicit in code that matches it, its much easier to:
  - understand what the pattern is
  - modify it

Editors, Unix, Python, Java...

- Regular expressions are used in many places.
- Editors like vi, emacs, and Sublime Text allow you to use regular expressions for searching.
- Many unix commands use regular expressions. Example:

  ```
  grep pattern file
  ```

  prints all lines from file that match pattern
- Many programming languages provide a library for regular expressions.
- The syntax varies from context to context, but the core is the same everywhere.

Simple patterns

Pattern	Matches	Explanation
$a*$	” ’a’ ’aa’	zero or more
$b+$	’b’ ’bb’	one or more
$ab?c$	’ac’ ’abc’	zero or one
$[abc]$	’a’ ’b’ ’c’	one from the set
$[a-c]$	’a’ ’b’ ’c’	one from the range
$[abc]*$	” ’acbccb’	combination

Note: In Java, patterns can be used to match an occurrence anywhere in the string, or one that consumes the whole string, among other options.

Anchoring

Lets you force the position of the match.

- ^ matches the begnining of the line
- $ matches the end
- Neither consumes any characters.

Pattern	Text	Result
`b+`	abbc	matches
`^b+`	abbc	Fails (no b at start)
`^a*$`	aabaa	Fails (not all a's)

Escaping

- Match actual

  `^` and `$` and `[` etc.

- using escape sequences

  `\^` and `\$` and `\[` etc.

- Remember, we also use escapes for other characters:

  `\t` is a tab character
  `\n` is a newline

Predefined Caracter Classes

Construct	Description
.	any character
\d	a digit [0-9]
\D	a non-digit [^0-9]
\s	a whitespace char [\t\n\x0B\f\r]
\S	a non whitespace char [^\s]
\w	a word char [a-zA-Z_0-9]
\W	a non word char [^\w]

Defining your own character classes

Construct	Description
[abc]	a,b or c (simple class)
[^abc]	any char except a,b, or c (negation)
[a-zA-Z]	a through z or A through Z inclusive (range)
[a-d[m-p]]	a through d or m through p (union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z except for b and c (subtraction)
[a-z&&[^m-p]]	a through z and not m through p (subtraction)

Quantifiers

Construct	Description
X?	0 or 1 times
X*	0 or more times
X+	1 or more times
X{n}	exactly n times
X{,n}	at least n times
X{n,m}	at least n but no more than m times

Capturing Groups and Backreferences

Capturing groups allow you to treat multiple characters as a single unit.

Use **parentheses** to group.

Capturing groups are **numbered** by counting their opening parentheses from left to right.

((A)(B(C))) has the following groups:

1. ((A)(B(C)))
2. (A)
3. (B(C))
4. (C)

Capturing Groups and Backreferences

The section of the input string matching the capturing group(s) is saved in memory for later recall via backreference.

A backreference is specified in the regular expression as a backslash (\) followed by a digit indicating the number of the group to be recalled.

Pattern	**Example matching string**
`(\d\d)\1`	1212
`(\w*)\s\1`	asdf asdf

Regular expressions in Java

The `java.util.regex` package contains:

- `Pattern`: a compiled regular expression
- `Matcher`: the result of a match

Example: `RegexDemo`

Regular expressions and language theory

- Language theory uses a very restricted form of regular expression.
- The set of strings accepted by a regular expression is said to be a **language**.
- BNF (Backus Normal Form or BackusNaur Form) rules are another way of defining a language. Example:

```
<expr> ::= <term>|<expr> + <term>|<expr> - <term>

<term> ::= <factor>|<term> * <factor>|<term> / <factor>
<factor> ::= <number>|  (<expr>)
```

- BNF is a more expressive notation: There are languages you can describe with BNF that you cant describe with regular expressions.
- E.g.: a sequence of a's followed by the same number of b's.

Topics