Programming Languages and Techniques (CIS1200)

Lecture 29

Enumerations and Iteration

Chapter 25

Announcements

- HW07: PennPals
  - Programming with Java Collections
  - Due Tuesday, April 9th at 11.59pm

- HW08: Twitter bot will be released Wednesday
  - Due Tuesday, April 16th at 11.59pm

Iterating over collections

iterators, while, for, for-each loops

Iterator and Iterable

```
interface Iterator<E> {
    boolean hasNext();
    E next();
    default void delete();   // optional
    default void forEachRemaining(..); // optional
}
```

```
interface Iterable<E> {
    Iterator<E> iterator();
}
```

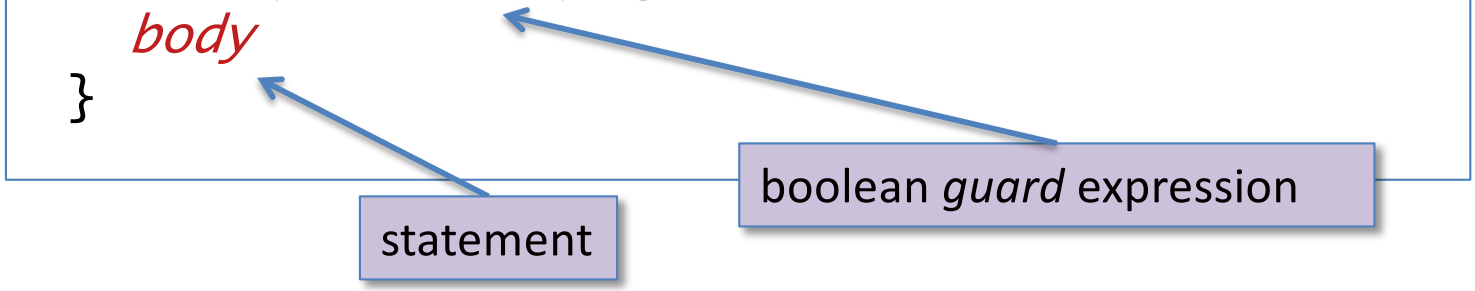
```
interface Collection<E> extends Iterable<E> …
```

Challenge: given a List<Book> how would you add each book's data to a catalogue using an iterator?

While Loops

syntax:

```
// repeat body until condition becomes false
while (condition) {
   body
}
```



example:

```
List<Book> shelf = …  // create a list of Books

// iterate through the elements on the shelf
Iterator<Book> iter = shelf.iterator();
while (iter.hasNext()) {
    Book book = iter.next();
    catalogue.addInfo(book);
    numBooks = numBooks+1;
}
```

For Loops

syntax:

```
for (init-stmt; condition; next-stmt) {
  body
}
```

equivalent while loop:

```
init-stmt;
while (condition) {
  body
  next-stmt;
}
```

```
List<Book> shelf = …  // create a list of Books

// iterate through the elements on the shelf
for (Iterator<Book> iter = shelf.iterator();
     iter.hasNext();) {
   Book book = iter.next();
   catalogue.addInfo(book);
   numBooks = numBooks+1;
}
```

For-each Loops

syntax:

```
// repeat body for each element in collection
for (type var : coll) {
   body
}
```

element type E

Array of E or instance of Iterable<E>

example:

```
List<Book> shelf = …  // create a list of books

// iterate through the elements on a shelf
for (Book book : shelf) {
    catalogue.addInfo(book);
    numBooks = numBooks+1;
}
```

For-each Loops (cont'd)

Another example:

```
int[] arr = …  // create an array of ints

// count the non-null elements of an array
for (int elt : arr) {
    if (elt != 0) cnt = cnt+1;
}
```

For-each can be used to iterate over arrays or any class that implements the `Iterable<E>` interface (notably `Collection<E>` and its subinterfaces).

Iterator example

```
public static void iteratorExample() {
    List<Integer> nums = new LinkedList<>();
    nums.add(1);
    nums.add(2);
    nums.add(7);

    int numElts = 0;
    int sumElts = 0;
    Iterator<Integer> iter =
        nums.iterator();
    while (iter.hasNext()) {
      Integer v = iter.next();
      sumElts = sumElts + v;
      numElts = numElts + 1;
    }

    System.out.println("sumElts = " + sumElts);
    System.out.println("numElts = " + numElts);
  }
```

What is printed by iteratorExample()?

1. sumElts = 0 numElts = 0
2. sumElts = 3 numElts = 2
3. sumElts = 10 numElts = 3
4. NullPointerException
5. Something else

Answer: 3

29: What is printed by iteratorExample()?

0

sumElts = 0 numElts = 0

0%

sumElts = 3 numElts = 2

0%

sumElts = 10 numElts = 3

0%

NullPointerException

0%

something else

0%

29: What is printed by nextNextExample()?

0

sumElts = 0 numElts = 0

0%

sumElts = 3 numElts = 2

0%

sumElts = 8 numElts = 2

0%

NullPointerException

0%

something else

0%

Another Iterator example

```
public static void nextNextExample() {
    List<Integer> nums = new LinkedList<>();
    nums.add(1);
    nums.add(2);
    nums.add(7);

    int sumElts = 0;
    int numElts = 0;
    Iterator<Integer> iter =
        nums.iterator();
    while (iter.hasNext()) {
      Integer v = iter.next();
      sumElts = sumElts + v;
      v = iter.next();
      numElts = numElts + v;
    }
    System.out.println("sumElts = " + sumElts);
    System.out.println("numElts = " + numElts);
  }
```

What is printed by nextNextExample()?

1. sumElts = 0 numElts = 0
2. sumElts = 3 numElts = 2
3. sumElts = 8 numElts = 2
4. NullPointerException
5. Something else

Answer: 5 NoSuchElementException

For-each version

```
public static void forEachExample() {
    List<Integer> nums = new LinkedList<>();
    nums.add(1);
    nums.add(2);
    nums.add(7);

    int numElts = 0;
    int sumElts = 0;
    for (Integer v : nums) {
      sumElts = sumElts + v;
      numElts = numElts + 1;
    }

    System.out.println("sumElts = " + sumElts);
    System.out.println("numElts = " + numElts);
  }
```

Enumerations

Enumerations (a.k.a. Enum Types)

- Java supports *enumerated* type constructors
  - Intended to represent constant data values

```
enum CommandType {
    CREATE, INVITE, JOIN, KICK, LEAVE, MESG, NICK
}
```

- Intuitively similar to a simple usage of OCaml datatypes
  - …but each language provides extra bells and whistles that the other does not

Enums with data

```
public enum ServerResponse {
    OKAY(200),
    INVALID_NAME(401),
    NO_SUCH_CHANNEL(402),
    NO_SUCH_USER(403),
    USER_NOT_IN_CHANNEL(404),
    USER_NOT_OWNER(406),
    JOIN_PRIVATE_CHANNEL(407),
    INVITE_TO_PUBLIC_CHANNEL(408),
    NAME_ALREADY_IN_USE(500),
    CHANNEL_ALREADY_EXISTS(501);

    // The integer associated with this enum value
    private final int value;

    ServerResponse(int value) {
        this.value = value;
    }

    public int getCode() {
        return value;
    }
}
```

Elements of the enum can be declared along with “parameters”

When the object representing each element is created, the associated parameters are passed to the constructor method.

Enums are Classes

- Enums are a convenient way of defining a class along with some standard static methods

`valueOf` : converts a `String` to an Enum

```
CommandType c = CommandType.valueOf("CREATE");
```

`values`: returns an `array` of all the enumerated constants

```
CommandType[] varr = CommandType.values();
```

- Implicitly extend class `java.lang.Enum`
- Can include specialized constructors, fields and methods, as in ServerResponse

Using Enums: Switch

```
// Use of 'enum'
CommandType t = …

switch (t) {
 case CREATE : System.out.println("Got CREATE!"); break;
 case MESG   : System.out.println("Got MESG!"); break;
 default     : System.out.println("default");
}
```

- Multi-way branch, similar to OCaml's match
  - Works for: primitive data 'int', 'byte', 'char', *etc.*, plus enum types and String
  - Not as powerful as OCaml pattern matching! (Yet!*)
- The `default` keyword specifies a "catch all" (wildcard) case
- Must indicate end of each case using break or return

*ML-style pattern matching that binds variables, etc., has been a "preview feature" of recent versions of Java, and is slowly being integrated into the main design.

29: What will be printed by the following program?

0

Got CREATE!

0%

Got MESG!

0%

Got NICK!

0%

default

0%

something else

0%

What will be printed by the following program?

```
CommandType t = CommandType.CREATE;

switch (t) {
  case CREATE : System.out.println("Got CREATE!");
  case MESG   : System.out.println("Got MESG!");
  case NICK   : System.out.println("Got NICK!");
  default     : System.out.println("default");
}
```

1. Got CREATE!
2. Got MESG!
3. Got NICK!
4. default
5. something else

Answer: 5 something else!

break

- GOTCHA: By default, each branch will "fall through" into the next, so that code actually prints:

```
Got CREATE!
Got MESG!
Got NICK!
default
```

- Use an explicit `break` statement to avoid fall-through:

```
switch (t) {
case CREATE : System.out.println("Got CREATE!");
              break;
case MESG   : System.out.println("Got MESG!");
              break;
case NICK   : System.out.println("Got NICK!");
              break;
default: System.out.println("default");
}
```

Alternative Option – switch Expressions

- Introduced in Java 14
- No need for `break` statements to prevent fall through
- Read more here - https://docs.oracle.com/en/java/javase/14/language/switch-expressions.html

```
switch (t) {
  case CREATE -> System.out.println("Got CREATE!");
  case MESG   -> System.out.println("Got MESG!");
  case NICK   -> System.out.println("Got NICK!");
  default     -> System.out.println("default");
}
```

Alternative Option – switch Expressions

- (Similar to OCaml pattern matching), these are *expressions* and evaluate to a single value
- (Similar to OCaml pattern matching), these must be exhaustive

```
int x = switch (t) {
  case CREATE -> 5;
  case MESG   -> 10;
  case NICK   -> 15;
  default     -> 20;
}
```

Some Advice on Debugging

Use the Scientific Method

1. Make an observation / ask a question
   - One of my test cases fails!
   - Which assertion? What exception? What is the stack trace?
2. Formulate a hypothesis
   - Could I have passed null as bar to foo.munge(bar)?
3. Conduct an experiment
   - Modify the program to try to confirm / refute the hypothesis.
   - *Don't* make random changes!
   - You should try to predict the effects of your experiment
   - Re-run test cases
4. Analyze the results
   - Did the modified code behave as expected?
5. Draw conclusions / Report results
   - Create a new test case (if appropriate)

Observing Behavior

- Understand exceptions and the stack trace
  - They give you a lot of information

- If you are using and IDE (like IntelliJ), it is worth taking a little time to learn how to use the debugger!
  - create “breakpoints” that stop the program and let you inspect the state of the abstract machine

- Simple print statements are also very effective!
  - Confirm or disprove hypothesis
  - Can sometimes be easier to control than the debugger
  - e.g.: The code reached "HERE!" (or not)