This recitation

• An interesting point about A2: Using previous methods to avoid work in programming and debugging. How much time did you spend writing and debugging prepend?
• Enums (enumerations)
• Generics and Java’s Collection interfaces and classes
• Parsing arithmetic expressions using a grammar that gives precedence to * and / over + and – (if there is time)

How to use previous methods in A2

The A2 handout contained this:
Further guidelines and instructions:
“Note that some methods that you have to write …. Also, in writing methods 4..7, writing them in terms of calls on previously written methods may save you time.”

Did you read that? Think about it? Attempt it?

A lesson in:
1. Reading carefully, wisely.
2. Thinking about what methods do, visualizing what they do.

Suppose we want to append e to this list:

This is what it looks like after the append:

What if we prepended e instead of appending it?

Legend  
   pred  suc

What append does:

What prepend does:

Therefore: prepend(v); can be done by append(v); head= head.pred; 
body of prepend

About enums (enumerations)

An enum: a class that lets you create mnemonic names for entities instead of having to use constants like 1, 2, 3, 4. The declaration below declares a class Suit.

After that, in any method, use Suit.Clubs, Suit.Diamonds, etc. as constants.

public enum Suit (Clubs, Diamonds, Hearts, Spades)

could be private, or any access modifier
new keyword
The constants of the class are Clubs, Diamonds, Hearts, Spades

An interesting point about A2: Using previous methods to avoid work in programming and debugging. How much time did you spend writing and debugging prepend?

Did you try to write prepend in terms of append?

How much time did you spend writing and debugging prepend?

Morals of the story:
1. Read carefully.
2. Visualize what methods do; understand specs completely.
3. Avoid duplication of effort by using previously written methods


**Four static final variables that contain pointers to objects**

```java
public enum Suit {Clubs, Diamonds, Hearts, Spades}
```

<table>
<thead>
<tr>
<th>Clubs</th>
<th>Suit@0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diamonds</td>
<td>Suit@1</td>
</tr>
<tr>
<td>Hearts</td>
<td>Suit@2</td>
</tr>
<tr>
<td>Spades</td>
<td>Suit@3</td>
</tr>
</tbody>
</table>

Clubs, Diamonds, Hearts, Spades

Are static variables of class enum

**Testing for an enum constant**

```java
public enum Suit {Clubs, Diamonds, Hearts, Spades}
```

```java
Suit s = Suit.Clubs;
```

Then

```java
s == Suit.Clubs is true
s == Suit.Hearts is false
```

```java
switch(s) {
    case Clubs:
    case Spades:
        color= "black"; break;
    case Diamonds:
    case Hearts:
        color= "red"; break;
}
```

Can use a switch statement

Type of s is Suit.

Inside the switch, you cannot write Suit.Hearts instead of Hearts

**Miscellaneous points about enums**

1. **Suit is a subclass of Enum (in package java.lang)**
2. It is not possible to create instances of class Suit, because its constructor is private!
3. It's as if Clubs (as well as the other three names) is declared within class Suit as
   ```java
   public static final Suit Clubs = new Suit(some values);
   ```
   You don't care what values

4. Static function `values()` returns a `Suit[]` containing the four constants. You can, for example, use it to print all of them:
   ```java
   for (Suit s : Suit.values())
     System.out.println(s);
   ```
   Output:
   ```
   Clubs
   Diamonds
   Hearts
   Spades
   ```

Can save this array in a static variable and use it over and over:
```java
private static Suit[] mine = Suit.values();
```

5. Static function `valueOf(String name)` returns the enum constant with that name:
   ```java
   Suit c = Suit.valueOf("Hearts");
   ```
   After the assignment, `c` contains the name of object Hearts

   ```java
   This is the object for Hearts:
   ```
   ```
   Suit@2
   ```

6. Object Clubs (and the other three) has a function `ordinal()` that returns its position in the list
   ```java
   Suit.Clubs.ordinal() is 0
   Suit.Diamonds.ordinal() is 1
   ```

We have only touched the surface of enums. E.g. in an enum declaration, you can write a private constructor, and instead of Clubs you can put a more elaborate structure. All this is outside the scope of CS2110.
Package java.util has a bunch of classes called the Collection Classes that make it easy to maintain sets of values, list of values, queues, and so on. You should spend some time looking at their API specifications and getting familiar with them.

Remember:
A set is a bunch of distinct (different) values. No ordering is implied
A list is an ordered bunch of values. It may have duplicates.

Interface Collection: abstract methods for dealing with a group of objects (e.g. sets, lists)
Abstract class AbstractCollection: overrides some abstract methods with methods to make it easier to fully implement Collection
AbstractList, AbstractQueue, AbstractSet, AbstractDeque overrides some abstract methods of AbstractCollection with real methods to make it easier to fully implement lists, queues, set, and deques

Next slide contains classes that you should become familiar with and use. Spend time looking at their specifications.

There are also other useful Collection classes

Class ArrayList extends AbstractList: An object is a growable/shrinkable list of values implemented in an array
Class Arrays: Has lots of static methods for dealing with arrays — searching, sorting, copying, etc.
Class HashSet extends AbstractSet: An object maintains a growable/shrinkable set of values using a technique called hashing. We will learn about hashing later.
Class LinkedList extends AbstractSequentialList: An object maintains a list as a doubly linked list
Class Stack extends Vector: An object maintains LIFO (last-in-first-out) stack of objects
Class Vector extends AbstractList: An object is a growable/shrinkable list of values implemented in an array. An old class from early Java

Class Arrays:

ArrayList v = new ArrayList();

ArrayList Object defined in package java.util

Fields that contain a list of objects
Object

ArrayList() add(Object) get(int) set(int, Object) remove() ...

HashSet s = new HashSet();

HashSet Object

Fields that contain a set of elements
Object

HashSet() add(Object) contains(Object) containsAll(Object) ...

Don’t ask what “hash” means. Just know that a Hash Set object maintains a set

A loop whose body is executed once with e being each element of the set. Don’t know order in which set elements processed
Use same sort of loop to process elements of an ArrayList in the order in which they are in the ArrayList.
Generics: say we want an ArrayList of only one class
API specs: ArrayList declared like this:
```java
public class ArrayList<E> extends AbstractList<E> implements List<E> ... {
```

Means: Can create Vector specialized to certain class of objects:
```java
ArrayList<String> vs = new ArrayList<String>(); // only Strings
ArrayList<Integer> vi = new ArrayList<Integer>(); // only Integers
```

```java
vs.add(3);  // only Strings, no need to cast
vi.add("abc");
```

Generics and Java's Collection Classes
ge·ner·ic adjective ˈgə-nərk, -rēk-
relating or applied to or descriptive of all members of a genus, species, class, or group; common to or characteristic of a whole group or class; typifying or subsuming; not specific or individual.

From Wikipedia: generic programming: a style of computer programming in which algorithms are written in terms of to-be-specified-later types that are then instantiated when needed for specific types provided as parameters. Read carefully!

In Java: Without generics, every Vector object contains a list of elements of class Object. Clumsy
With generics, we can have a Vector of Strings, a Vector of Integers, a Vector of Genes. Simplifies programming, guards against some errors

Generics allow us to say we want Vector of Strings only
API specs: Vector declared like this:
```java
public class Vector<E> extends AbstractList<E> implements List<E> ... {
```

Full understanding of generics is not given in this recitation. E.g. We do not show you how to write a generic class.

Important point: When you want to use a class that is defined like Vector above, you can write:
```java
Vector<C> v = new Vector<C>();
```
to have v contain a Vector object whose elements HAVE to be of class C, and when retrieving an element from v, its class is C.
Parsing Arithmetic Expressions

We show you a real grammar for arithmetic expressions with integer operands; operations +, -, *, /; and parentheses ( ). It gives precedence to multiplicative operations.

We write a recursive descent parser for the grammar and have it generate instructions for a stack machine (explained later). You learn about infix, postfix, and prefix expressions.

Historical note: Gries wrote the first text on compiler writing, in 1971. It was the first text written/printed on computer, using a simple formatting application. It was typed on punch cards. You can see the cards in the Stanford museum; visit infolab.stanford.edu/pub/voy/museum/pictures/display/floor5.htm

Parsing Arithmetic Expressions

-5 + 6 Arithmetic expr in infix notation
5 – 6 Same expr in postfix notation

PUSH 5 Corresponding machine language for a “stack machine”:
NEG PUSH 6 PUSH; push value on stack
ADD NEG; negate the value on top of stack
ADD: Remove top 2 stack elements, push their sum onto stack

Infix requires parentheses. Postfix doesn’t

(5 + 6) * (4 – 3) Infix
5  6  +  4  3  -  *

5 + 6 * 3 Infix
5  6  3  *  +  Postfix

Math convention: * has precedence over +. This convention removes need for many parentheses

Task: Write a parser for conventional arithmetic expressions whose operands are ints.

1. Need a grammar for expressions, which defines legal arithexp, giving precedence to * / over + -
2. Write recursive procedures, based on grammar, to parse the expression given in a String. Called a recursive descent parser

Use 3 syntactic categories: <Exp>, <Term>, <Factor>

A <Factor> has one of 3 forms:
1. integer
2. <Factor>
3. ( <Exp> )

Show “syntax trees” for
3 – 5 – (3 + 2)

Use 3 syntactic categories: <Exp>, <Term>, <Factor>

A <Term> is:
<Factor> followed by 0 or more occurrences of <Addop> <Factor>

Means: 0 or 1 occurrences of * or /

Use 3 syntactic categories: <Exp>, <Term>, <Factor>

A <Exp> is:
<Factor> followed by 0 or more occurrences of <Addop> <Term>

Means: 0 or more occurrences of things inside ()

Use 3 syntactic categories: <Exp>, <Term>, <Factor>

3 * (5 + 2) * 6

Haven’t shown <Exp> grammar yet

3 + (5 + 2)

11

3 + (5 + 2) 6

28
Class Scanner

Initialized to a String that contains an arithmetic expression.
Delivers the tokens in the String, one at a time.

Expression: 3445*(20 + 16)
Tokens: 3445 *
  20 + 16

An instance provides tokens from a string, one at a time.
A token is either
1. an unsigned integer,
2. a Java identifier
3. an operator + - * / %
4. a paren of some sort: ( ) [ ] { }
5. any seq of non-whitespace chars not included in 1..4.

public Scanner(String s) // An instance with input s
public boolean hasToken() // true iff there is a token in input
public String token() // first token in input (null if none)
public String scanOverToken() // remove first token from input
  // and return it (null if none)
public boolean tokenIsInt() // true iff first token in input is int
public boolean tokenIsId() // true iff first token in input is a
  // Java identifier

/** scanner's input should start with a <Factor> -- if not, throw a RuntimeException.
Return the postfix instructions for <Factor> and have scanner remove the <Factor> from its input.
<Factor> ::= an integer
| - <Factor>
| ( <Expr> ) 
*/

public static String parseFactor(Scanner scanner) {
    String code = parseTerm(scanner);
    while ("+".equals(scanner.token()) ||
        "-".equals(scanner.token())) {
        String op = scanner.scanOverToken();
        String rightOp = parseTerm(scanner);
        code = code + rightOp +
            (op.equals("+") ? "PLUS\n" : "MINUS\n");   }   }
    return code;

/** scanner's input should start with an <Exp> -- if not throw a RuntimeException.
Return corresponding postfix instructions and have scanner remove the <Exp> from its input.
<Exp> ::= <Term> { {+ or -}1 <Term>} 
*/

public static String parseExp(Scanner scanner) {
    return parseTerm(scanner) + "PLUS\n";   }

/** scanner's input should start with a <Factor> -- if not, throw a RuntimeException.
Return the postfix instructions for <Factor> and have scanner remove the <Factor> from its input.
<Factor> ::= an integer
| - <Factor>
| ( <Expr> ) 
*/

public static String parseFactor(Scanner scanner) {
    // An instance with input s
    // true iff there is a token in input
    // first token in input (null if none)
    // remove first token from input
    // and return it (null if none)
    // true iff first token in input is int
    // Java identifier
    //  */

Parser for <Exp>

Parser for <Factor>

The spec of every parser method for a grammatical entry is similar. It states
1. What is in the scanner when parsing method is called
2. What the method returns.
3. What was removed from the scanner during parsing.