Generics with ArrayList and HashSet

ge·ner·ic *adjective* \jə·ˈnerik, -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.

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 and Java's Collection Classes

ge·ner·ic *adjective* \jə·ˈnerik, -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.

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 with ArrayList and HashSet

defined in package java.util

ArrayList v= **new** ArrayList ();

An object of class ArrayList contains a growable/ shrinkable list of elements (of class Object). You can get the size of the list, add an object at the end, remove the last element, get element i, etc. More methods exist! Look at them!

v | ArrayList@x1 |

Vector

ArrayList@x1

Object

Fields that contain a list of objects $(o_0, o_1, \ldots, o_{size()-1})$ — ArrayList

ArrayList () add(Object)
get(int) size()
remove(…) set(int, Object)
…

Generics with ArrayList and HashSet

HashSet s= **new** HashSet();

An object of class HashSet contains a growable/ shrinkable set of elements (of class Object). You can get the size of the set, add an object to the set, remove an object, etc. More methods exist! Look at them!

s HashSet@y2

HashSet

Don't ask what "hash" means. Just know that a Hash Set object maintains a set

HashSet@y2

Object

Hashset

Fields that contain a setof objects $\{o_0, o_1, \ldots, o_{size()-1}\}$

HashSet() add(Object)
contains(Object) size()
remove(Object)
…

Iterating over a HashSet or ArrayList

```
HashSet s= new HashSet();

… code to store values in the set …

for (Object e : s) {
    System.out.println(c);
}
```

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 array ArrayList .

HashSet@y2

Object

Fields that contain a setof objects HashSet

$\{o_0, o_1, \ldots, o_{size()-1}\}$

HashSet() add(Object)
contains(Object) size()
remove(Object)
…

s HashSet@y2

HashSet

ArrayList to maintain list of Strings is cumbersome

```
ArrayList v= new ArrayList ();
… Store a bunch of Strings in v …
// Get element 0, store its size in n
String ob= ((String) v.get(0)).length();
int n= ob.size();
```

—Only Strings, nothing else

All elements of v are of type Object. So, to get the size of element 0, you first have to cast it to String.

Make mistake, put an Integer in v? May not catch error for some time.

v ArrayList@x1 ArrayList

ArrayList @x1

Object

ArrayList

Fields that contain a list of objects ($o_0, o_1, \ldots, o_{size()-1}$)

Vector() add(Object)
get(int) size()
remove() set(int, Object)
…

Generics: say we want Vector of ArrayList only

API specs: ArrayList declared like this:

public class ArrayList <E> **extends** AbstractList<E>
implements List<E> … { … }

Means:
Can create Vector specialized to certain class of objects:

ArrayList <String> vs= **new** ArrayList <String>(); //only Strings

ArrayList <Integer> vi= **new** ArrayList <Integer>(); //only Integers

vs.add(3);
vi.add("abc");
These are illegal

int n= vs.get(0).size();

vs.get(0) has type String
No need to cast

Generics allow us to say we want Vector of Strings only

API specs: Vector declared like this:

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

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.

Package java.util has a bunch of classes called e Collection Classes that make it easy to maintain sets of values, list of values, queues, and so on. You should spend dome time looking at their API specifications and getting familiar with them.

Interface Collection: abstract methods for dealing with a group of objects (e.g. sets, lists)

Abstract class AbstractCollection: overrides some abstract methods with real methods to make it easier to fully implement Collection

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

ArrayList extends AbstractList: An object is a growable/ shrinkable list of values implemented in an array

HashSet extends AbstractSet: An object maintains a growable/shrinkable set of values using a technique called *hashing*. We will learn about hashing later.

LinkedList extends AbstractSequentialList: An object maintains a list as a doubly linked list

Vector extends AbstractList: An object is a growable/ shrinkable list of values implemented in an array. An old class from early Java

Stack extends Vector: An object maintains LIFO (last-in-first-out) stack of objects

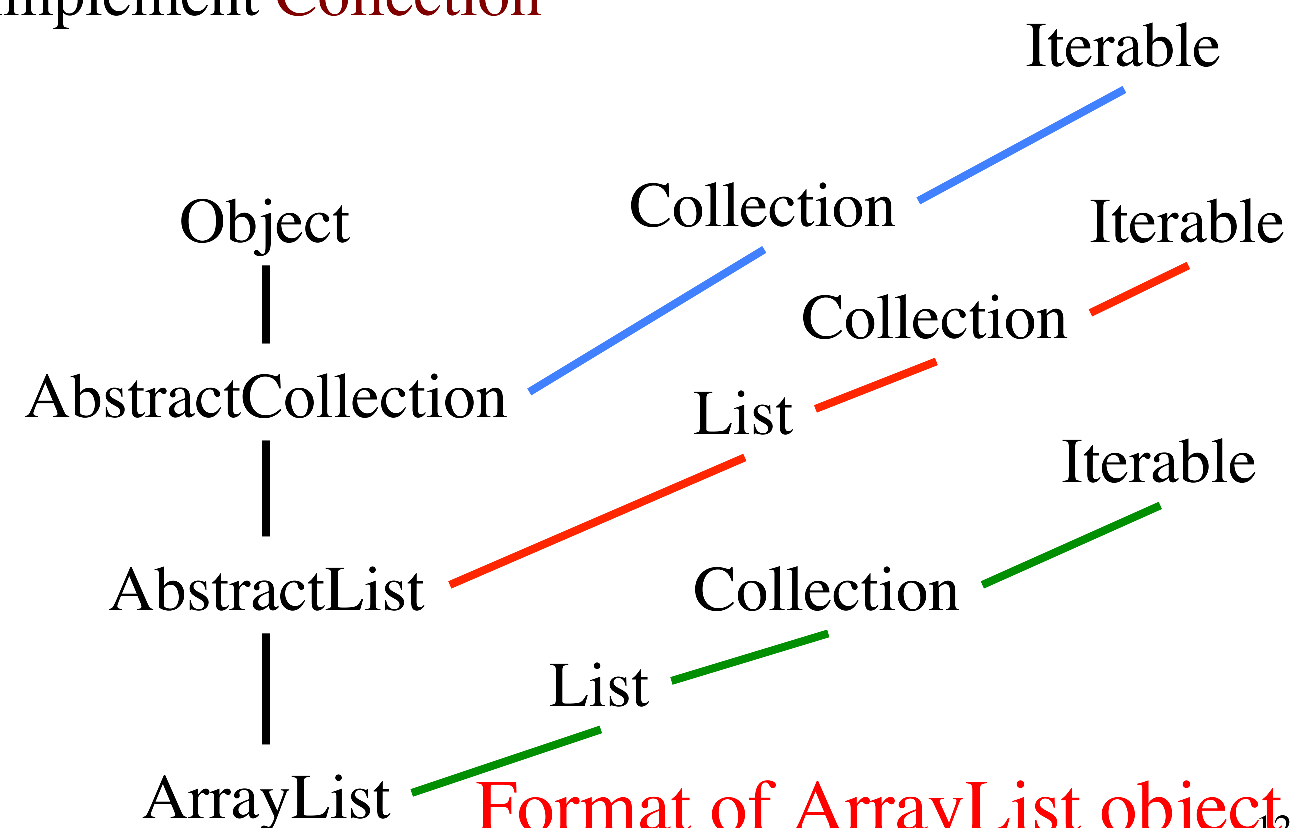
Arrays: Has lots of static methods for dealing with arrays —searching, sorting, copying, etc.

Interface Collection: abstract methods for dealing with a group of objects (e.g. sets, lists)

Abstract class AbstractCollection: overrides some abstract methods with real methods to make it easier to fully implement Collection

Iterable
Not discussed today

ArrayList implements 3 other interfaces, not shown



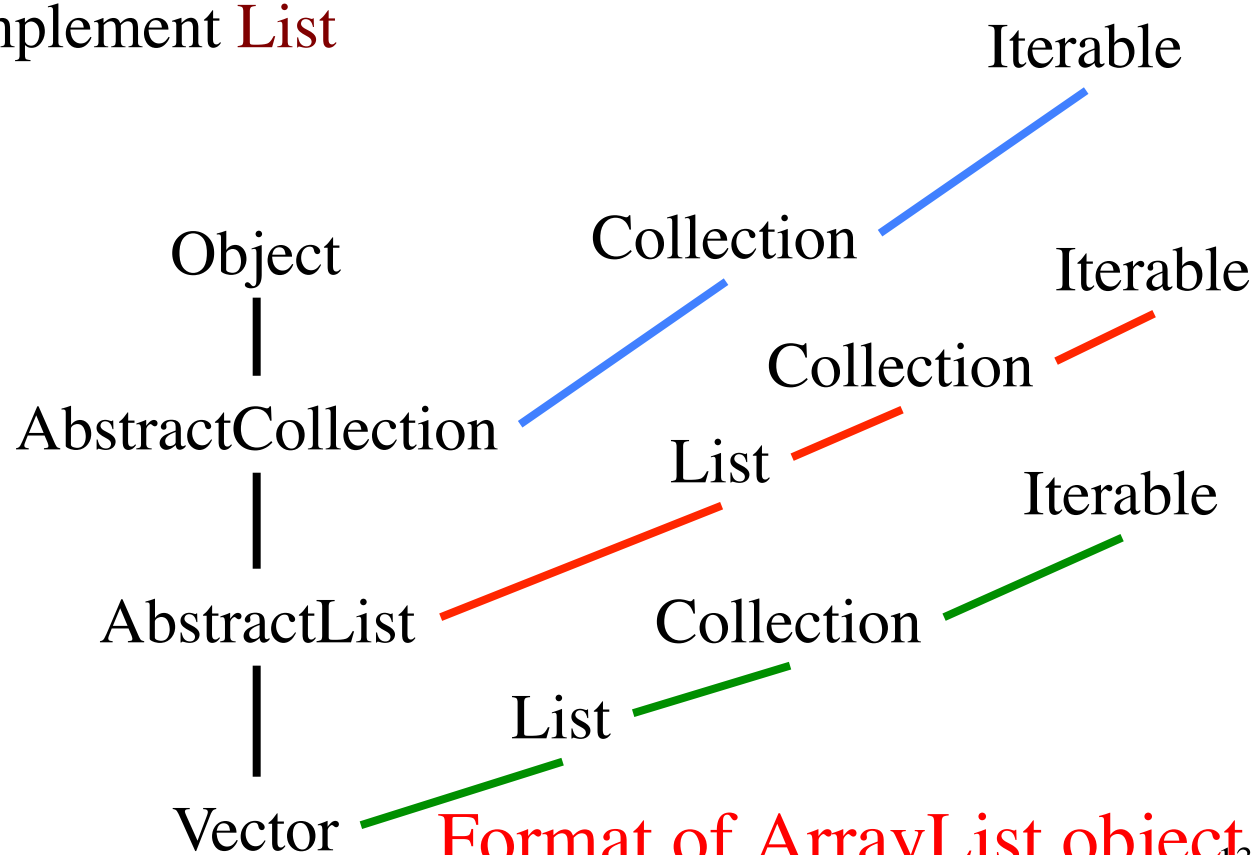
Format of ArrayList object

Interface List: abstract methods for dealing with a list of objects ($o_0, \ldots, o_{n-1}$). Examples: arrays, Vectors

Abstract class AbstractList: overrides some abstract methods with real methods to make it easier to fully implement List

Iterable
Not discussed today

Homework: Look at API specifications and build diagram giving format of HashSet



Format of ArrayList object

Parsing Arithmetic Expressions

Introduced in lecture briefly, to show use of grammars and recursion. Done more thoroughly and carefully here.

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

infix: operation between operands
postfix: operation after operands
prefix: operation before operands

PUSH 5
NEG
PUSH 6
ADD

Corresponding machine language for a "stack machine":

PUSH: push value on stack
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 - *   Postfix

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 arith exps, giving precedence to * / over + -
2. Write recursive procedures, based on grammar, to parse the expression given in a String. Called a recursive descent parser

Grammar

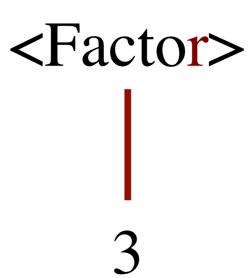
Use 3 syntactic categories: <Exp>, <Term>, <Factor>

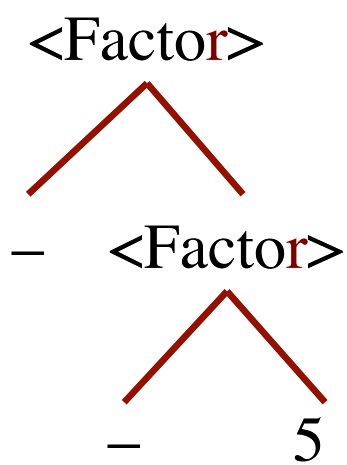
A <Factor> has one of 3 forms:

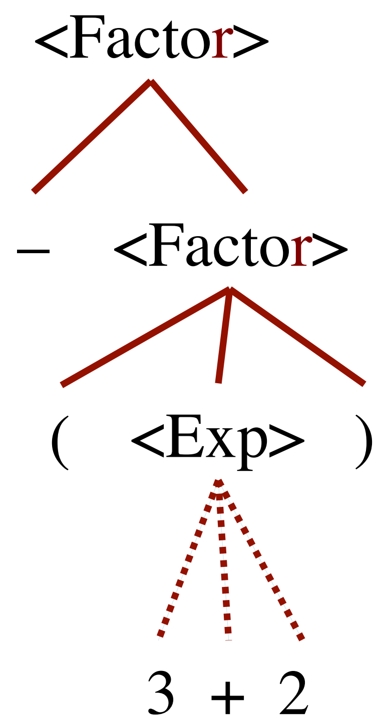
1. integer
2. – <Factor>
3. (<Exp>)

Show "syntax trees" for
3   – – 5   – (3 + 2)

<Factor> ::= int
| <Factor>
| (<Exp>)







Haven't shown <Exp> grammar yet

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

A <Term> is:

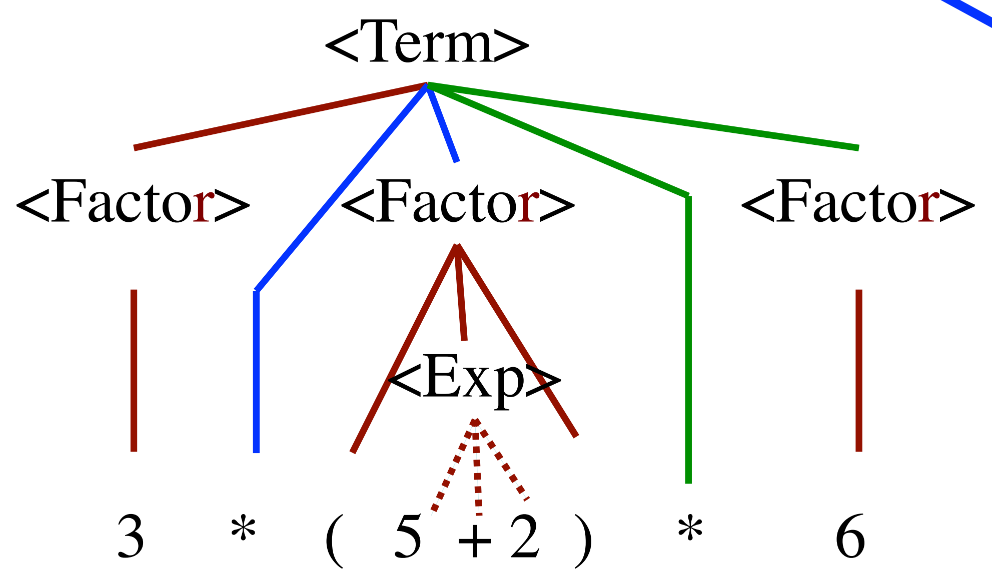
<Factor> followed by 0 or more occurs. of **multop** <Factor>

where **multop** is * or /

Means: 0 or 1 occurrences of * or /

<Term> ::= <Factor> { {* | /}1 <Factor> }

Means: 0 or more occurrences of thing inside { }



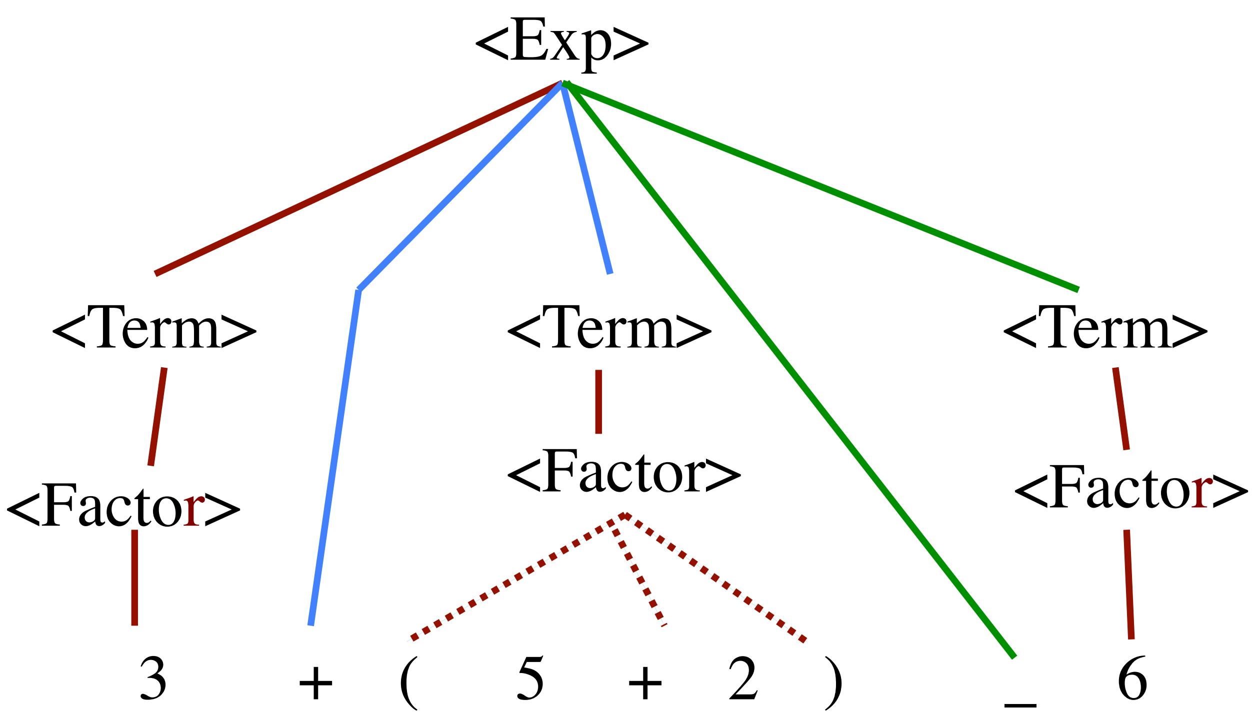
Use 3 syntactic categories: <Exp>, <Term>, <Factor> **Grammar**

A <Exp> is:

<Term> followed by 0 or more occurrences of **addop** <Term>

where **addop** is + or -

<Exp> ::= <Term> **{ {**+ | -**}1** <Term> **}**



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

)

All parsers use a scanner, so they do not have to deal with the input character by character and do not have to deal with whitespace

Class Scanner

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
```

Parser for <Factor>

```
/**  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)
```

The spec of every parser method for a grammatical entry is similar. It states

1. What is in the scanner when paring method is called
2. What the method returns.
3. What was removed from the scanner during parsing.

Parser for <Exp>

```
/**  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) {
     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;
  }
```