Generics with Vector and HashSet

**Generic** adjective [ˈdʒɛrɪk, -ɪk] 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.

**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

---

### Iterating over a HashSet or Vector

HashSet $\Rightarrow$ new HashSet();

... code to store values in the set ...

**for** (Object e : x) {
    System.out.println(e);
}

A loop whose body is executed once for each element of the set. Don’t know order in which set elements processed

Use same sort of loop to process elements of a Vector in the order in which they are in the Vector.

---

### Using Vector to maintain list of Strings is cumbersome

Vector vs = new Vector();

... Store a bunch of Strings in vs ...

// Get element 0, store its size in n
String obv = (String) vs.get(0).length();
int n = obv.size();

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 Vector@x1 Vector

---

### Generics allow us to say we want Vector of Strings only

API specs: Vector declared like this:

```
public class Vector$\langle E \rangle$ extends AbstractList$\langle E \rangle$
```

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

```
Vector<String> vs = new Vector<String>(); //contain only Strings
Vector<Integer> vs = new Vector<Integer>(); //contain only Integers
```

These are illegal

```
vs.add(3);
vi.add("abc");
```

---

Generics with Vector and HashSet

**Vector** 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!

```java
Vector vs = new Vector();
```

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

**Vector() get(int) size() remove() set(int, Object)**

---

Generics with Vector and HashSet

**HashSet** $\Rightarrow$ 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!

```
HashSet hs = new HashSet();
```

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

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

---

Generics with Vector and HashSet

**HashSet** $\Rightarrow$ new HashSet();

... code to store values in the set ...

```java
HashSet@y2
```

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

---

Generics with Vector and HashSet

**Vector** $\Rightarrow$ new Vector();

An object of class Vector 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!

```java
Vector vs = new Vector();
```

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

**Vector() get(int) size() remove() set(int, Object)**

---

Generics with Vector and HashSet

**HashSet** $\Rightarrow$ new HashSet();

... code to store values in the set ...

```java
HashSet@y2
```

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

---
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 **MUST** be of class `C`, and when retrieving an element from `v`, its class is `C`.

---

**Assignment A1**

Second part of A1 requires use of some sort of list and some sort of set. Used in a generic way.

Understanding what we did in past slides, today, will help you with this.

One step is to create a set of genes from a list of genes. This can be done without a for-loop because there exists a constructor in the class for implementing sets that will do it for you.

Note that class `Gene` overrides `equals` and `hashCode`, so your class `MyGene` does not have to do it!

---

**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 `()`.

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

---

### 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`

- `Object`
- `AbstractCollection`  
- `Collection`
- `List`
- `Iterable`

**Iterateable Not discussed today**

---

### Interface List

abstract methods for dealing with a list of objects (o₀, …, oₙ₋₁).

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

- `Object`
- `AbstractCollection`  
- `Collection`
- `List`
- `Iterable`

**Iterateable Not discussed today**

---

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

---

### 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`  
  *Operation on a “stack machine“*

- `ADD`  
  *Operation on a “stack machine“*

- `ADD`: Remove top 2 stack elements, push their sum onto stack
Infix requires parentheses. Postfix doesn’t

\( (5 + 6) \times (4 - 3) \) \text{ Infix}
\( 5 + 6 \times 3 \) \text{ Postfix}
\( 5 + (6 \times 3) \) \text{ Infix}
\( 5 + 6 \times 3 \) \text{ Postfix}

**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: \(<\text{Exp}>\), \(<\text{Term}>\), \(<\text{Factor}>\)

\(<\text{Factor}>\) has one of 3 forms:
1. integer
2. \(- <\text{Factor}>\)
3. \( ( <\text{Exp}> ) \)

A \(<\text{Term}>\) is:
\(<\text{Factor}>\) followed by 0 or more occurrences of \(<\text{Term}>\)
where \(<\text{addop}>\) is + or -

\(<\text{Exp}>\) := \(<\text{Term}>\) \{ \{ + | - \} 1 \}<\text{Term}>\}

**Class Scanner**

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

Expression: \(3445 \times (20 + 16)\)

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
```java
/** 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) {
    // Implementation
}

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.
```