Generics

COSC 1P03 – Lecture 07 (Spring 2024)

Maysara Al Jumaily
amaysara@brocku.ca

Brock University

Tuesday June 11, 2024

Total slides: 17

Lecture Outline

Overview of Generics

- Think about a stack (last-in-first-out), don't we agree that no matter what the type of the object that will be inserted, it will always behave the same?
- So, if we insert all elements as `String`s or all as `int`s or all as `double`s, the structure will still stay the same.
- In other words, The structure is *independent* of the type being used.
- That is the point of generics. we will not worry about what the type is but just the structure we want to achieve.
- For example, we created a queue that stored `int`egers as the type.
- How about changing the queue to store `double`s instead. We would agree that the type needs to be changed only, correct? Yes, correct.
- When using generics, we **MUST** use the object-equivalent of the primitive types, **NOT** primitive types. This is described in the upcoming slides.

Wrapper classes

- We all realized that the primitive types: `char`, `boolean`, `byte`, `short`, `int`, `long`, `float` and `double` are special.
- For example, we can have `int x = 7;`.
- Can we treat it as an object and perform something like:
  `int x = new int(7);//Nope! Wrong syntax!`
- What if we *really* wanted to do that?
- Luckily, we have the `Integer` object class defined by Java. It is a wrapper class of the primitive type `int`. It can be used as such:
- Why do we have the `Integer` class object that behaves like an `int`?
  - Because generics only accept objects as the type, *not* primitive types.
- Note, each of the primitive types have their corresponding Object: `char` has `Character`, `boolean` has `Boolean`, `byte` has `Byte`, `short` has `Short`, `int` has `Integer`, `long` has `Long`, `float` has `Float` and `double` has `Double`.
- You can use something like:

```
Integer x = new Integer(7);//Similar to initializing an object
Boolean y = new Boolean(false);//Similar to initializing an object
Double z = new Double(3.14);//Similar to initializing an object
```

Parameterizing A Stack

- As mentioned before, a stack is a data structure that allows for a first-in-last-out operations.
- The structure will not change if we are storing `int`egers, as opposed to `double`s, or `String`s.
- Let us parameterize a stack that uses the linked list implementation.
- We will use `Node`s to create the linked list, and we have to also parameterize `item`, as we could store some generic type.
- To use generics, we would need to denote the type we are dealing with as some `E` type (it could be any other name, not just `E`, like `T`).
- We will pass in the type (*e.g.*, `Integer`, `String`, `Turtle`, etc.) later and Java will automatically substitute the type inside of `E`.
- The name of the class (not filename, but class name) will include `<E>` to denote a generic class.
- The `E` in `<E>` is a place holder for the type that will be substituted later. The convention is using `E` or `T`; other letter/words are allowed.
- The interface will also include `<E>`, it will allow for pushing/inserting something of type `E` and remove something of type `E`.
- To substitute the type of a `String`, we write the following:

```
Stack<String> s = new ArrayIntStack<String>();
```

Parameterizing the Node Class

- The `Node` class will be translated to:

```java
import java.io.Serializable;

public class Node<E> implements Serializable {
  public E item;
  public Node<E> next;
  public Node(E item, Node<E> next) {
    this.item = item;
    this.next = next;
  }
}
```

- We will ensure to add `<E>` to the class name (and objects) and `E` as the type instead of `int`.

- Previously, we used the interface `IntStack`, to denote that the implementation of the stack only accepts `int`egers. Now, we will create a structure that is independent of the type. We can call our stack interface `Stack`.
- The `Stack` interface will be translated to:

```
public interface Stack<E> {
  public void push (E item);

  public E top (); //returns last element added

  public E pop ();//returns last element added AND removes it

  public boolean empty ();

  public E size (); //extra for fun!
}
```

- We will ensure to add `<E>` to the class name (and objects) and `E` as the type instead of `int`.

Parameterizing the ArrayStack Class I

- The `ArrayStack` will be a parameterized class that implements the parameterized interface:

```java
public class ArrayStack<E> implements Stack<E> {
  private int index = 0; // current available cell
  private int count = 0; // Number of elements in the stack
  private E[] data;      // The array of type E.

  public ArrayStack() { ... }

  public ArrayStack(int size) { ... }

  public void enter(E item) { ... }

  public E front() { ... }

  public E leave() { ... }

  public boolean empty() { ... }

  public int size() { ... }
}
```

Parameterizing the ArrayStack Class II

- The interesting piece of code is the initialization of the array.
- Java will force us to create an array of type `Object` and then cast it to an array of type `E`:
  `data = (E[]) new Object[100];`
- Let us break it down:
  - Recall that if we wanted to cast from a `double` to an `int`, we write:
    `int x = (int) 3.14;`
  - The `(E[])` part is casting to an array of type `E`, and `new Object[100];` is creating an array containing 100 elements of type `Object`.
  - Hence, cast an array of type `Object` to an array of type `T`
  - Remember that everything (other than primitive types) in Java is seen as an `Object`.

Parameterizing the LinkedStack Class

- The `LinkedStack` class will be similar to the `ArrayStack` class but now will use `head` as the node, which will be of type `Node<E>`.

```
public class LinkedStack<E> implements Stack<E> {
  private Node<E> head = null; // The head of the linked list
  private int count = 0; // Number of elements in the stack

  public LinkedStack() { ... }

  public void enter(E item) { ... }

  public E front() { ... }

  public E leave() { ... }

  public boolean empty() { ... }

  public int size() { ... }
}
```

Parameterizing a Queue

- The same concept follows through when implementing a queue.
- The structure will defer such that the insertion will be placed at the rear, but removal at the front, similar to a stack.
- The `Node` class will be parameterized and the `Queue` interface. Note that we don't have `IntQueue` as the interface, not `StringQueue`, but a generic `Queue` interface.

Autoboxing vs Unboxing I

- Autoboxing/unboxing deals with converting from the primitive type to the corresponding wrapper class and vice versa.
- **Autoboxing:** implicitly/automatically converts from primitive type and store it in the wrapper class type. For example:
  ```
  Integer y = 7;//autoboxing
  ```
  Java would convert the above line as the following proper format:
  ```
  Integer y = new Integer(7);
  ```
- **Unboxing:** converting from wrapper class type and store it in primitive type. For example:
  ```
  int z = new Integer(7);//unboxing
  ```
  Java would automatically/implicitly convert the above line to the following:
  ```
  int z = 7;
  ```
- We should realize that the above will work for *primitive* types **only**.

Autoboxing vs Unboxing II

- Java will deal with primitive types (`int`, `double`, etc.) and their wrapper classes (`Integer`, `Double`, etc.) seamlessly. For example, suppose we have the following code snippet:

  ```
  int x = 7;
  Integer y = new Integer(7);
  if(x == y){
    System.out.println("Yes");
  }
  ```

  Java will recognize that we are comparing a primitive type `x` to an Object `y`, but since that object is the wrapper class of the type `int`, the comparison will be based off of the two `int`s involved. The `if` statement will be true and `"Yes"` will be printed.
- The same logic follows through when dealing with other primitive types.

Autoboxing vs Unboxing III

- The previous slide compared a primitive type to its corresponding wrapper class object.
- What if we compare two objects to each other? Note that there are no primitive types in this comparison:

```
Integer a = new Integer(7);
Integer b = new Integer(7);
if(a == b){
  System.out.println("same");
}
```

- Java tries to compare two objects, yes, they have the same types, but still are two distinct objects.
- In the eyes of Java, they are different objects and are not equal. Hence, the `if` statement will *not* evaluate to true and nothing is printed.

Java's Characters and Integers I

- Java has the primitive type `char`, which holds one and only one character; here are some examples:

```java
char a = 'Z';
char b = 'Q';
char c = 'W';
```

- Would it be surprising to know that we can compare `char`s to `int`s? A `char` is a character, whereas an `int` is an integer, how can we compare them?
- It happens that every character is associated with a number. So, the `char`acter `'Z'` is associated with the value 90, `'Q'` is associated with the value 81 and `'W'` is associated with the value 87.
- We could store the `int`eger value that corresponds to a `char`:

```java
char a = 'Z'; // the character corresponding to the value of 90
int x = a; // store integer value corresponding to 'Z', which is 90
if(x == 90){ // valid statement and will be true
  System.out.println("Correct!");
}
```

- It is also valid to store an `int`eger into a `char`acter, which will correspond to the ASCII value of the character:
`char a = 90; // The 'Z' character`

Java's Characters and Integers II

- These values are based on the ASCII Table, which is found next slide.
- The names `ASCIIDataFile`, `ASCIIDisplayer` and `ASCIIOutputFile` are all derived from the ASCII Table.
- The ASCII Table contains typical Latin (English) upper and lower characters, digits and punctuations.
- Simplified Chinese characters, Arabic characters latin characters with accents (*i.e.*, Ł, ï, ú, ø, ß, etc.) mathematical symbols or Emojis are not a part of ASCII characters, but Unicode Standard.
- The *Unicode Standard* allows for 1 114 112 characters to be added, unlike the ASCII Standard, which allows for 128 characters, only.
- As of the latest version of Unicode 15.1.0 (September 2023), there are 149 813 characters.

Java's Characters and Integers – ASCII Table

- A standard table that *all* of the typical programming languages use.
- It has 128 characters that are numbered, and the first 32 characters (0 to 31) are non-printable characters, such as tab, new line, escape, etc.
- The character ␣ corresponding to the value 32 is the space character.

$0, 1, 2, \ldots, 31$ are special non-printable characters (backspace, tab, etc.)																			
32	␣	33	!	34	"	35	#	36	$	37	%	38	&	39	'	40	(	41	)
42	*	43	+	44	,	45	-	46	.	47	/	48	0	49	1	50	2	51	3
52	4	53	5	54	6	55	7	56	8	57	9	58	:	59	;	60	<	61	=
62	>	63	?	64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O	80	P	81	Q
82	R	83	S	84	T	85	U	86	V	87	W	88	X	89	Y	90	Z	91	[
92	\	93	]	94	^	95	_	96	`	97	a	98	b	99	c	100	d	101	e
102	f	103	g	104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	c	116	t	117	u	118	v	119	w	120	x	121	y
122	z	123	{	124	|	125	}	126	~	127	DEL								