Iterables, For Each Loops, and Generic Types

The Iterable interface, continued

This interface is used to indicate that we can create an Iterator for the class. The Iterator lets us run through the contents of the class.

The main purpose is to have a way to loop through the data of an abstract data type, without needing to know details of the implementation of the abstract data type.

Recall how we write a loop for a linked list:

LLNode<T> nodeptr = getFront();

while (nodeptr != null) {

T element = nodeptr.getNext();

System.out.println(element);

nodeptr = nodeptr.getNext();

}

Now that we have an Iterator for LinkedList, we can re-write the loop using the iterator:

Iterator<T> it = iterator();

while (it.hasNext()) {

T element = it.next();

System.out.println(element);

}

What if we want to place the loop in a static method, what do we use for the generic:

public static void printList(LinkedList<.....> list) {

Iterator<....> it = iterator();

while (it.hasNext()) {

.... element = it.next();

System.out.print(element + " ");

}

System.out.println();

}

Solution 1, use a generic T. However, we cannot use the generic of the containing class (if it has one).

The generic is specified by the creating of an instance to the class, and there is no instance in a static method.

Instead, we can declare a generic in a method by placing the generic declaration before the return type.

public static <T> void printList(LinkedList<T> list) {

Iterator<T> it = iterator();

while (it.hasNext()) {

T element = it.next();

System.out.print(element + " ");

}

System.out.println();

}

Solution 2 (a better solution) is to use a generic wild card. The ? is a wildcard that means "don't care".

You can use the ? wildcard so that any type will work, but you must have any code that checks the type of the generic. (That would imply that you care what the type is.)

public static void printList(LinkedList<?> list) {

Iterator<?> it = iterator();

while (it.hasNext()) {

Object element = it.next();

System.out.print(element + " ");

}

System.out.println();

}

Incorrect solution: Specify the generic with Object: public static void printList(LinkedList<Object> list) {

- at first glance, this looks like it will accept any linked list (because everything is an object), but it does not.

- it will only accept linked lists with Object as the generic.

Consider this:

LLNode<Object> node = new LLNode<String>("Hello", null);

It looks reasonable because Object is wider than String. HOWEVER, all generic information is only used by the compiler. So, the compiler must be able

to verify, for each statement and expression, the types are used correctly. Consider this line:

node.setElement(new JFrame());

Is it legal as far as the compiler is concerned? Yes, node stores Object and a JFrame is an object.

But is it legal as far as the LLNode object referred to by node is concerned? No! That node was created to only store String, but instead we stored a JFrame!

Or consider this:

node.setNext(new LLNode<Object>(new JFrame(), null));

Same problem, this code passes the type for node, but it does not fit what we inteneded when we created the LLNode<String>.

IMPORTANT FACT:

The generic type only applies to the current type, not the true type.

Consider:

LLNode<String> node = new LLNode<String>("Hi", null);

The current type of the value stored in node is LLNode<String>.

The true type of the value created by new is LLNode.

The compiler uses the generics to verify that all the current types match. Since the generic information is not used when the code is running (i.e. by the true type), the compiler has

to block any code that could potentially cause problems such as typecasting LLNode<String> to LLNode<Object>.

A better solution to printList

Instead of passing in a LinkedList, we could instead pass in an Iterable. Since LinkedList implements Iterable, Iterable is a wider type, and the method will still work with LinkedList, but

now it will work with ALL classes that implement the Iterable interface.

public static void printList(Iterable<?> list) {

Iterator<?> it = iterator();

while (it.hasNext()) {

Object element = it.next();

System.out.print(element + " ");

}

System.out.println();

}

Now, we can pass in any class that extends the Iterable interface:

LinkedList<Integer> list = new LinkedList<Integer>();

list.addToFront(1);

list.addToFront(2);

list.addToFront(3);

printList(list);

Java Collections

Java Collections are a set of classes (that implement the Collections interface) that represent abstract data types that store elements.

The Collection interface extends the Iterable interface.

Two main examples of Collections:

1. LinkedList: just like our LinkedList, but it has arrows going in two directions.

2. ArrayList: a "wrapper" for an array that automatically handles the "allocate a new array and copy data over" when the array runs out of memory.

IMPORTANT: Because both are Collections, we can use them in any code that has Collections. Also, they both have most of the same methods because these methods are inherited

from their parent classes and interfaces. BUT you must remember what they are. Just because a method exists in the API does not mean it is a good idea to use it.

Accessing an element by index in a LinkedList is a terrible idea because the linked list must run from the beginning, counting as it goes.

Starting an ArrayList with minimal capacity and then continually adding elements is also not a good idea if you know what the capacity will be. Instead of one array allocation, the

ArrayList will end up doing many unnecessary allocations and array copies.

Because both are iterable, our printList method will work with both:

ArrayList<Integer> alist = new ArrayList<Integer>();

alist.add(1);

alist.add(2);

alist.add(3);

printList(alist);

LinkedList<Integer> llist = new LinkedList<Integer>();

llist.add(1);

llist.add(2);

llist.add(3);

printList(llist);

Foreach loops:

For each loops are a Java shortcut for Iterable classes and for arrays.

The form of a foreach loop is: for (T i : Iterable<T>)

and it reads as "foreach type T in iterable"

For example, if list is a LinkedList<Integer>, we could have: for (Integer i : list)

which reads as "foreach Integer in list"

Here are two examples:

LinkedList<Integer> list = new LinkedList<Integer>();

list.addToFront(1);

list.addToFront(2);

list.addToFront(3);

for (Integer x : list) {

System.out.print(x + " ");

}

double[] a = {1, 2, 3, 4, 5};

for (double x : a) {

System.out.println(x \* x + " ");

}

Note that arrays are not Iterable. Java just allows foreach loops to work with arrays.

A final example: our printList method using foreach

public static void printList(Iterable<?> list) {

for (Object element : list) {

System.out.print(element + " ");

}

System.out.println();

}