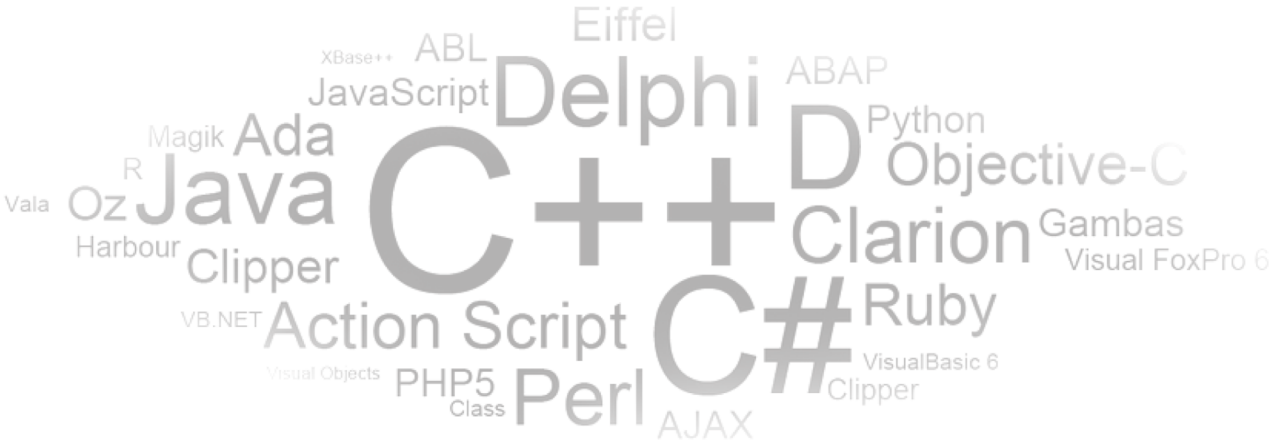
CIS 351-Data Structure-Generics
March 3, 2020

Dr. Farzana Rahman

Syracuse University



ArrayLists and arrays

- A `ArrayList` is like an array of `Objects`, but…

- To create an ArrayList:
  - `ArrayList myList = new ArrayList();`
  - Or, since an `ArrayList` is a kind of `List`,
    `List myList = new ArrayList();`

- To use an ArrayList,
  - `boolean add(Object` ***obj***`)`
  - `Object set(int` ***index***`, Object` ***obj***`)`
  - `Object get(int` ***index***`)`

ArrayLists, then and now

- Starting in Java 5, `ArrayLists` have been genericized
  - That means, every place you used to say `ArrayList`, you now have to say what kind of objects it holds; like this: `ArrayList<String>`
  - If you don't do this, you will get a warning message, but your program will still run

Auto boxing and unboxing

- Java won't let you use a primitive value where an object is required--you need a "wrapper"
  - `ArrayList<Integer> myList = new ArrayList<Integer>();`
  - `myList.add(new Integer(5));`
- Similarly, you can't use an object where a primitive is required--you need to "unwrap" it
  - `int n = ((Integer)myArrayList.get(2)).intValue();`

- With Java Generics makes this automatic:
  - ```
    myArrayList<Integer> myList = new myArrayList<Integer>();
    myList.add(5);
    int n = myList.get(2);
    ```
    ```

Good and Bad

- The bad news:
  - Instead of saying: `List words = new ArrayList();`
  - You'll have to say:

```
List<String> words = new ArrayList<String>();
```

- The good news:
  - No casting;

  instead of

```
String title = (String) words.get(i);
```

  you use

```
String title = words.get(i);
```

- Some classes and interfaces that have been “genericized” are: Vector, ArrayList, LinkedList, Hashtable, HashMap, Stack, Queue, PriorityQueue, Dictionary, TreeMap and TreeSet

Generic programming

- Java Generics is a feature that enables the definition of classes that are implemented independently of some type that they use as an **abstraction** by accepting a type parameter.

- For example, a class like `LinkedList<T>` is a generic type, that has a type parameter T.

- Instantiations, such as `LinkedList<String>` or a `LinkedList<Integer>`, are called **parameterized types**, and `String` and `Integer` are the respective actual **type arguments**.

- Java generics use a technique known as **type erasure**, and the compiler keeps track of the generic definitions internally, hence using the same class definition at compile/run time.

Generic classes

- A class that is defined with a parameter for a type is called a generic class
- For classes, the type parameter is included in **angular brackets** after the class name in the class definition heading.
- When a generic class is used, the specific type to be plugged in is provided in angular brackets.
- Every occurrence of the type parameter is replaced with the highest type applicable to the type parameter.
- If a type bound was specified, this type is applied. If no type bound was specified, **`Object`** is used.

```
/**
 * Generic class that defines a wrapper class around a single
 * element of a generic type.
 */
public class Box<T extends Number> {

    private T t;

    public void set(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
/**
 * Generic method that uses both the generic type of the class
 * it belongs to, as well as an additional generic type that is
 * bound to the Number type.
 */
    public void inspect(){
        System.out.println("T: " + t.getClass().getName());
    }

    public <U> void inspectWithAdditionalType(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(new Integer(10));
        integerBox.inspect();
        integerBox.inspectWithAdditionalType("Hello world");
        Integer i = integerBox.get();
    }
}
```

Generic classes: benefit

- So, what is the difference between a generic class and a class defined using `Object` as the internal type? Consider a **`LinkedList`** class that can contain elements of type **`Object`**:

```
LinkedList list = new LinkedList();
list.add("abc");                  // fine
list.add(new Date());             // fine as well
```

- This seems interesting, until we get the elements from the list:

```
String s = (String)list.get(0);  // cast required
Date d   = (Date)list.get(1);    // cast required
```

- As the elements are of type **`Object`**, we must explicitly cast them to use them as objects of their own type after extraction.

Generic classes: benefit

- The problem is that the compiler cannot check at compile time whether such casts are valid or not.
- Using generic classes, we can define such a **`LinkedList`** and parameterize it for every specific use and ensuring **type safety** for each different use of generic class:

```
LinkedList<String> stringList = new LinkedList<String>();
stringList.add("Hello");                    // fine
// list.add(new Date(1,1));                 // error
String s = stringList.get(0);               // no cast needed

LinkedList<Integer> integerList = new LinkedList<Integer>();
integerList.add(new Integer(10));           // fine
// integerList.add("Hello");                // error
Integer i = integerList.get(0);             // no cast needed
```

- Generic classes and the **type erasure** mechanism allow programmer to:
  - Define classes that are valid in different contexts of use.
  - Ensure that they are used correctly in each specific context of use.

Iterators

- Iterator
  - Gives the ability to cycle through items in a collection
  - Access next item in a collection by using iter.next()
- provides two primary iterator interfaces
  - java.util.Iterator
  - java.util.ListIterator
- Every ADT collection have a method to return an iterator object

Generic Iterators

- An Iterator is an object that will let you step through the elements of a list one at a time
  - ```
    List<String> listOfStrings = new ArrayList<String>();
    ...
    for (Iterator i = listOfStrings.iterator(); i.hasNext(); ) {
        String s = (String) i.next();
        System.out.println(s);
    }
    ```

- Iterators have also been genericized:
  - ```
    List<String> listOfStrings = new ArrayList<String>();
    ...
    for (Iterator<String> i = listOfStrings.iterator(); i.hasNext();
    ) {
        String s = i.next();
        System.out.println(s);
    }
    ```

- If a class implements `Iterable`, you can use the new for loop to iterate through all its objects

Iterators

- `ListIterator` methods
  - `void add(E o)`
  - `boolean hasNext()`
  - `boolean hasPrevious()`
  - `E next()`
  - `int nextIndex()`
  - `E previous()`
  - `int previousIndex()`
  - `void remove()`
  - `void set(E o)`

Writing your own generic types

```
public class Box<T>
 {
    private List<T> contents;

    public Box() {
        contents = new ArrayList<T>();
    }

    public void add(T thing) { contents.add(thing); }

    public T grab()
    {
        if (contents.size() > 0)
            return contents.remove(0);
        else
            return null;
}
```

- Sun's recommendation is to use single capital letters (such as T) for types

New for statement

- The syntax of the new statement is

```
    for(type var : array)  {...}
or  for(type var : collection) {...}
```

- Example:

```
for(float x : myRealArray) {
    myRealSum += x;
}
```

- For a collection class that implements `Iterable`, instead of

```
for (Iterator iter = c.iterator(); iter.hasNext(); )
    ((TimerTask) iter.next()).cancel();
```

  you can now say

```
for (TimerTask task : c)
    task.cancel();
```

New for statement with arrays

- The new `for` statement can also be used with arrays

- Instead of

```
for (int i = 0; i < array.length; i++) {
    System.out.println(array[i]);
}
```

  you can say (assuming `array` is an `int` array):

```
for (int value : array) {
    System.out.println(value);
}
```

- Disadvantage: You don't know the index of any of your values

Summary

- If you think of a genericized type as a ***type***, you won't go far wrong
  - Use it wherever a type would be used
  - `ArrayList myList` becomes `ArrayList<String> myList`
  - `new ArrayList()` becomes `new ArrayList<String>()`
  - `public ArrayList reverse(ArrayList list)` becomes `public ArrayList<String> reverse(ArrayList<String> list)`
- **Advantage:** Instead of having collections of "Objects", you can control the type of object
- **Disadvantage:** more complex, more typing