Day 11

COMP1006/1406

Summer 2016

M. Jason Hinek
Carleton University

today's agenda

- assignments
    - Only the Project is left!
- Generics

last time...

efficiency... searching... sorting...

Generics

consider the List ADT

List ADT

data	operations
an ordered collection of values (need not be unique)	create an empty list ask if the list is empty or not ask what the size of the list is add an item to the list remove an item from the list ask which item is at a given position

We can also describe it as

- an ordered collection with arbitrary access to items in the collection

Generics

the `ArrayList` class implements the List ADT in Java

- the values in an `ArrayList` are `Object`s by default
- this is annoying when using an ArrayList
  - always casting `Object` to the class you are are really using
  - ((String)list.get()).toUpperCase()
- we typically don't store actual `Object` objects in an ArrayList though
- `ArrayList` uses **generics**
- this allows us to specify what kind of list we want to have without requiring a new class for each different kind

Generics

the `ArrayList` class uses generics

```
public class ArrayList<E>{

   public boolean add(E e){...}
     /* Appends specified element to the end of the list */

   public void add(int index,E element){...}
     /* Inserts element to specified position in list */

   public E get(int index){...}
     /* Returns the element at the specified position */
```

Generics

When you specify the type of list

```
   ArrayList<String> list = new ArrayList<String>();
```

it is like Java had a class defined as

```
public class ArrayList<String>{

   public boolean add(String e){...}
     /* Appends specified element to the end of the list */

   public void add(int index,String element){...}
     /* Inserts element to specified position in list */

   public String get(int index){...}
     /* Returns the element at the specified position */
```

Generics

When you specify the type of list

```
    ArrayList<Person> list = new ArrayList<String>();
```

it is like Java had a class defined as

```
public class ArrayList<Person>{

    public boolean add(Person e){...}
      /* Appends specified element to the end of the list */

    public void add(int index,Person element){...}
      /* Inserts element to specified position in list */

    public Person get(int index){...}
      /* Returns the element at the specified position */
```

Generics

think of `E` as an input parameter to the `ArrayList` class

```
public class ArrayList<E>{

   public boolean add(E e){...}
     /* Appends specified element to the end of the list */

   public void add(int index,E element){...}
     /* Inserts element to specified position in list */

   public E get(int index){...}
     /* Returns the element at the specified position */
```

Generics

generics in Java let us use `types` as parameters in our classes, methods and interfaces

generics allows us to implement an abstract data type without having to hard-code the details of a one specific data type.

- this lets us write less code
- we don't need `intList`, `doubleList`, `PersonList`, etc.

Some other befits of using generics

- allows for generic algorithms (like sorting)
- eliminates casting from `Object`
- provides stronger compile time type checks

Note: generics only works with objects (not primitive data types). This is only a minor annoyance since each primitive data type has an associated wrapper class we can use.

Generics

A **generic** class might look like

```
public class Box<T>{
   T data;

   public Box(){...}      // you don’t use the <T> in the constructor
   public Box(T data){    // declarations
      this.data = data;
   }

   public static void main(String[] args){
      Box<String>  bs = new Box<String>("cat");
      Box<Integer> bi = new Box<Integer>(12);

      Box<Box<String>> bbs = new Box<Box<String>>(bs);

      Box<ArrayList<String>> bals = new Box<ArrayList<String>>();
      bals.data = new ArrayList<String>();
      bals.data.add("dog");
   }
}
```

Generics

A **generic** linked list node class might look like

```
public class Node<E>{
   public E       data;
   public Node<E> next;

   public Node(){ this(null, null); }
   public Node(E data){ this(data, null); }
   public Node(E data, Node<E> next){
      this.data = data;
      this.next = next;
   }

 }
```

Note: there can be multiple type parameters also

```
   public class Things<K,V>{ ... }
```

Generics

In a **generic** method

- type parameters specified between the modifiers and the return type
- each type given must appear in either the return type or the input argument types so that the compiler knows what type is specified when the method is called
- the return type might use the type parameters
- the input parameters might use the type parameters
- the body can use the type parameters

```
static <T> T foo0(){...}                          // return type is generic
public <T> int foo1(ArrayList<T> list){...}   // argument type is generic
public <K,V> Box<V> foo2(K key){...}          // both are generic

private <T> Box<T> foo3(ArrayList<T> list){
   Box<T> b = new Box<T>(list.get(2));
   return b;
}
```

Generics

using **generic** methods

- Java use **type inference** so that you do not have to explicitly specify the types of the methods (but you can if you want)

```
/* static <T> T foo0(){...}  */
String s = <String>foo();    // you can specify the type of the method
String s = foo();            // the return type implies String

/* public <T> int foo1(ArrayList<T> list){...}           */
/* object is an object of the class that foo1 is defined */
ArrayList<Integer> list = new ArrayList<Integer>();
int x = object.<Integer>foo1(list);   // you can specify the type
int x = object.foo1(list);            // the type of list implies Integer
```

Generics

For more information see

`https://docs.oracle.com/javase/tutorial/java/generics/`

You add restrictions to the type parameters

```
public static <E extends Comparable<E>> foo(ArrayList<E> list)

public static <E extends Comparable<E> & Person> bar(ArrayList<E> list)
```

You add add bounds to the type parameters (wildcards)

```
public static void processList(List<? extends Foo> list){
    for (Foo elem : list){
        ...
    }
}
```

Here the type is unknown but it must be an descendant of `Foo` final exam is August 23rd at 7:00pm

room to be announced

projects are due next week!!

THE END