Java Collections

Agenda

- Collections Overview
- Generics
- Vector, ArrayList, HashMap
- Utils
- Special Collections

Collections Overview

Collection classes in Java are containers of Objects which by polymorphism can hold any class that derives from Object (which is actually, any class)

Using Generics the Collection classes can be aware of the types they store

Collections Overview

1st Example:

```
static public void main(String[] args) {
    ArrayList argsList = new ArrayList();
    for(String str : args) {
       argsList.add(str);
    }
    if(argsList.contains("Koko") {
      System.out.println("We have Koko");
    }
    String first = (String)argsList.get(0);
    System.out.println("First: " + first);
}
```

Collections Overview

2nd Example – now with Generics:

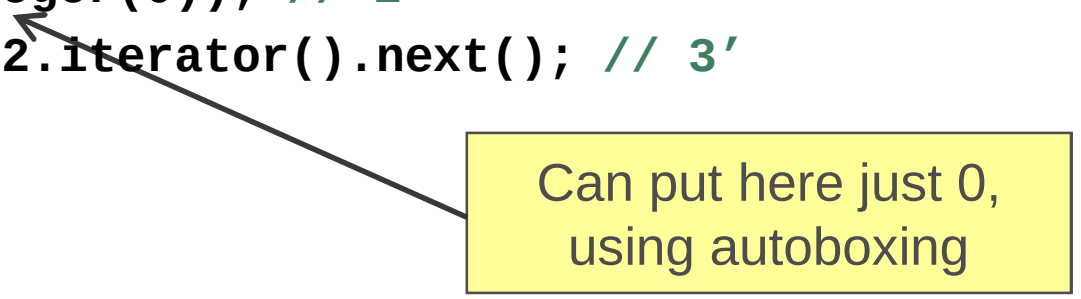
```
static public void main(String[] args) {
    ArrayList<String> argsList =
             new ArrayList<String>();
    for(String str : args) {
       argsList.add(str); // argsList.add(7) would fail
    }
    if(argsList.contains("Koko") {
      System.out.println("We have Koko");
    }
    String first = argsList.get(0); // no casting!
    System.out.println("First: " + first);
}
```

Generics

Generics are a way to define which types are allowed in your class or function

```
// old way
List myIntList1 = new LinkedList(); // 1
myIntList1.add(new Integer(0)); // 2
Integer x1 = (Integer) myIntList1.iterator().next(); // 3

// with generics
List<Integer> myIntList2 = new LinkedList<Integer>(); // 1'
myIntList2.add(new Integer(0)); // 2'
Integer x2 = myIntList2.iterator().next(); // 3'
```



Generics

Example 1 – Defining Generic Types:

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}

public interface Iterator<E> {
    E next();
    boolean hasNext();
}

public interface Map<K,V> {
    V put(K key, V value);
}
```

Generics

Example 2 – Defining (our own) Generic Types:

```
public class GenericClass<T> {
    private T obj;
    public void setObj(T t) {obj = t;}
    public T getObj() {return obj;}
    public void print() {
        System.out.println(obj);
    }
}
```

Main:

```
    GenericClass<Integer> g = new GenericClass<Integer>();
    g.setObj(5); // auto-boxing
    int i = g.getObj(); // auto-unboxing
    g.print();
```

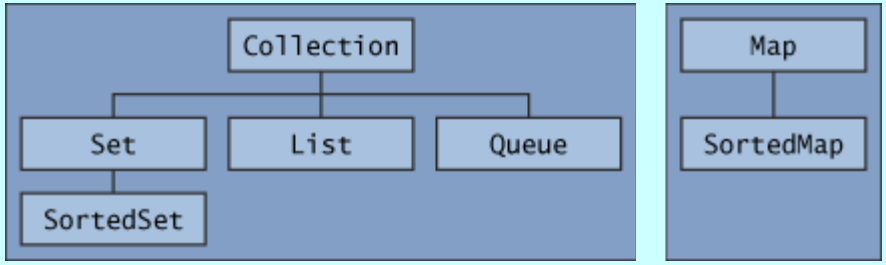
Generics – for advanced students

Generics is a complex topic
to cover it we added some more slides as an appendix

Which Collections do we have?

There are two main interfaces for all the collection types in Java:

- **Collection<E>**
- **Map<K,V>**



List of all Collections and related frameworks:

http://java.sun.com/javase/6/docs/technotes/guides/collections/reference.html

Vector

Vector is a synchronized dynamically growable array with efficient access by index

Example:

initialCapacity is optional

```
Vector<Integer> vec =
    new Vector<Integer>(10/*initialCapacity*/);
vec.add(7);
```

Vector is an old (Java 1.0) container and is less in use today, replaced mainly by ArrayList (Java 1.2) which is not synchronized

ArrayList

ArrayList is a non-synchronized dynamically growable array with efficient access by index

Example:

initialCapacity is optional

```
ArrayList<Integer> arr =
    new ArrayList<Integer>(10/*initialCapacity*/);
arr.add(7);
```

ArrayList is in fact not a list (though implementing the List interface)
If you need a list use the LinkedList class!

How should I know?

When performing many adds and removes

HashMap

HashMap is a non-synchronized key-value Hashtable

Example 1:

```
HashMap<String, Person> id2Person;
...
Person p = id2Person.get("021212121");
if(p != null) {
   System.out.println("found: " + p);
}
```

HashMap is a Java 1.2 class.
There is a similar Java 1.0 class called Hashtable which is synchronized and is less used today

HashMap

Example 2:

```
HashMap<String, Integer> frequency(String[] names) {
   HashMap<String, Integer> frequency =
            new HashMap<String, Integer>();
   for(String name : names) {
      Integer currentCount = frequency.get(name);
      if(currentCount == null) {
         currentCount = 0; // auto-boxing
      }
      frequency.put(name, ++currentCount);
   }
   return frequency;
}
```

HashMap

Example 2 (cont'):

```
public static void main(String[] args) {
   System.out.println(
      frequency(new String[]{
         "Momo", "Momo", "Koko", "Noa", "Momo", "Koko"
      }).toString());
}
```

HashMap has a nice toString!

Print out of this main is:

{Koko=2, Noa=1, Momo=3}

HashMap doesn't guarantee any order!

HashMap

For a class to properly serve as a key in HashMap the equals and hashCode methods should both be appropriately implemented

Example:

Parameter MUST be Object (and NOT Person!)

```
public class Person {
    public String name;
    boolean equals(Object o) {
        return (o instanceof Person &&
                    ((Person)o).name.equals(name));
    }
    public int hashCode() {
        return name.hashCode();
    }
}
```

Collection Utils

Handful Collection utils appears as static methods of the class Collections:

http://java.sun.com/javase/6/docs/api/java/util/Collections.html

A similar set of utils for simple arrays appear in the class Arrays:

http://java.sun.com/javase/6/docs/api/java/util/Arrays.html

Special Collections

BlockingQueue

- Interface, part of java.util.concurrent
- extends Queue with specific operations that:
  - wait for the queue to become non-empty when retrieving
  - wait for queue to have room when storing an element

ConcurrentMap

- part of the new java.util.concurrent
- extends Map with atomic putIfAbsent, remove and replace

CopyOnWriteArrayList

- As its name says…

For more Special Collections see the java.util.concurrent package:

http://java.sun.com/javase/6/docs/api/java/util/concurrent/package-summary.html

Get Strings from the command line, present in the console a vertical bar chart of the frequency of each letter in the input.

- **Treat small and capital letters the same -- as capital**
- **Ignore any char that is not an English letter**

Example

For the following input:

```
Hey how are you?
```

we expect the following chart:

```
A       #
E       ##
H       ##
O       ##
R       #
Y       ##
U       #
W       #
```

Write the necessary classes to support the following main:

```
static public void main(String[] args) {
   Expression e =
      new Sum(
         new Exponent( new Var("X"), new Number(3.0) ),
         new Sum( new Var("X"), new Var("Y") )
      );

   Function f = new Function(e);
   try {
      f.setVar("Z", 3.0);
   } catch(InvalidVariableException e) {
      System.out.println(e.getMessage());
   }

   // the main continues in next page!
```

cont’

```
    try {
       f.evaluate();
    } catch(MissingVariableException e) {
       System.out.println(e.getMessage());
    }

    f.setVar("Y", 1.0);
    for(double d=-1; d<=1; d+=0.5) {
       f.setVar("X", d);
       System.out.println("X=" + f.getVar("X") +
          ", Y=" + f.getVar("Y") + ", " + f + "=" + f.evaluate());
    }

} // end of main
```

--- please continue to next page

cont’

--- the program above should print:

```
Variable ‘Z’ does not exist
Missing the following variable(s): ‘X’, ‘Y’
X=-1.0, Y=1.0, ((X ^ 3.0) + (X + Y)) = -1.0
X=-0.5, Y=1.0, ((X ^ 3.0) + (X + Y)) = 0.375
X=0, Y=1.0, ((X ^ 3.0) + (X + Y)) = 1.0
X=0.5, Y=1.0, ((X ^ 3.0) + (X + Y)) = 1.675
X=1.0, Y=1.0, ((X ^ 3.0) + (X + Y)) = 3
```

Appendix

Special appendix on Generics

Generics
[How does it work? – "Erasure"]

There is no real copy for each parameterized type
(Unlike Templates in C++)

What is being done?

- **Compile time check (e.g. List<Integer> adds only Integers)**
- **Compiler adds run-time casting (e.g. pulling item from List<Integer> goes through run-time casting to Integer)**
- **At run-time, the parameterized types (e.g. <T>) are Erased – this technique is called Erasure**

At run-time, List<Integer> is just a List !

Generics
[Erasure implications #1]

Is the following possible?

```
public class GenericClass<T> {
        private T obj;
        …
        public void print() {
                System.out.println(obj);
        }
}
```

Answer is: NO (compilation error on: `T.class`)

But, the following, however, is possible:

```
System.out.println("obj type: " + obj.getClass().getName());
```

Generics
[Erasure implications #2]

Is the following possible?

```
public class GenericClass<T> {
        private T obj;
        public GenericClass() {
            }
}
```

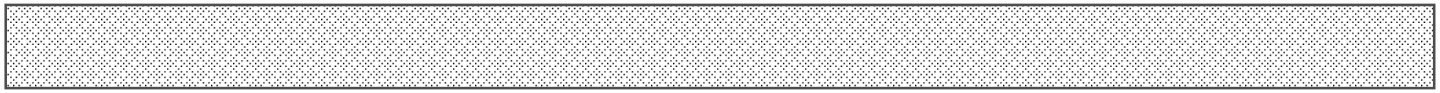
Answer is: NO (compilation error on: `new T();`)

One should either send an instantiated object or go back to reflection and send the class:

```
public GenericClass(Class<T> klass) {
        obj = klass.newInstance(); // handle exceptions..
}
```

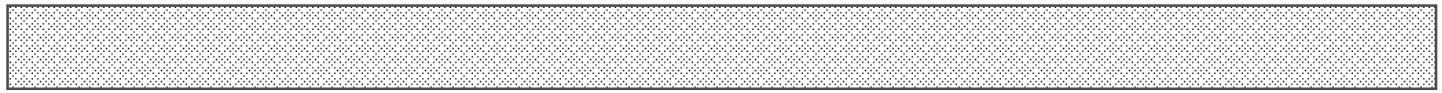
Generics
[Erasure implications #3]

Is the following possible?



```
...
}
```

Or:



```
...
}
```

Answer is: NO (compilation error, **T** is erased)

T is not a known type during run-time.

To enforce a parameter of type T we will have to use compile time checking (e.g. function signature)

Generics
[Erasure implications #4]

Is the following possible?

```
...
}
```

Or:

```
...
}
```

Answer is: NO (compilation error, `List<Integer>` isn't a class)

List<Integer> is not a known type during run-time.

To enforce List<Integer> we will have to use compile time checking (e.g. function signature)

Generics
[Erasure implications #5]

Is the following possible?

```
List myRawList;
List<Integer> myIntList = new LinkedList<Integer>();
```

Needed for backward compatibility

Answer is: Yes (`List<Integer>` is in fact a `List`)

The problem starts here:

Not checked at run-time (erasure…)

```
myRawList.add("oops"); // gets type safety warning
System.out.println(myIntList.get(0)); // OK, prints oops
                    // (though might be compiler dependent)
Integer x3 = myIntList.get(0); // Runtime ClassCastException
```

```
// this explains why operations on raw type
// should always get type safety warning
```

Generics
[Erasure implications #5B]

By the way… is the following possible?

```
List myRawList = new LinkedList();
List<Integer> myIntList;
```

Wow, that's ugly and quite disturbing

Answer is: Yes (with type-safety warning)

And run-time errors risk

The reason is again backward compatibility:

⇒ **myRawList might result from an old library that does not use generics**

⇒ **the following casting should have been the solution:**

```
myIntList = (List<Integer>)myRawList; // illegal casting
```

But: List<Integer> is not a type (as it was "erased")

Generics
[Erasure - Summary]

- There is no real copy for each parameterized type
  (Unlike Templates in C++)

What is being done?

- Compile time check (e.g. List<Integer> adds only Integers – checked against the signature List<T>.add)
- Compiler adds run-time casting (e.g. return type from List<T>.get() goes through run-time casting to T)
- At run-time, the parameterized types (e.g. <T>) are Erased and thus CANNOT BE USED during run-time

At run-time, List<Integer> is just a List !

Generics
[Subtyping]

Parameterized types can be restricted:

```
public class GenericSerializer<                              > {
    …
}
```

- Type T provided for our GenericSerializer class must implement Serializable
- Note that the syntax is always "extends", also for interfaces

Multiple restrictions might be provided, separated by &:

```
public class Foo<                                                  > {
    …
}
```

Generics
[Wildcards and subtyping #1]

Is the following possible?

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

Well, we know that the following is of course fine:

```
String str = "hello";
Object obj = str;
```

Answer is: NO (compilation error)

This comes to avoid the following:

```
listObjects.add(7);
String str = listStrings.get(0); // wd've been run-time error
```

Generics
[Wildcards and subtyping #2]

Suppose we want to implement the following function:

```
void printCollection(Collection col) {
       for(Object obj : col) {
              System.out.println(obj);
       }
}
```

But we want to do it in a "generic" way, so we write:

```
void printCollection(                    col) {
       for(Object obj : col) {
              System.out.println(obj);
       }
}
```

Can get ONLY collection of Objects

(go one slide back for explanation)

Cannot support Collection<String> Collection<Float> etc.

What's wrong with the 2nd implementation?

Generics
[Wildcards and subtyping #3]

The proper way is:

```
void printCollection(                                    col) {
        for(Object obj : col) {
                System.out.println(obj);
        }
}
```

Which is the same, for this case, as:

```
void printCollection(                 col) {
        for(Object obj : col) {
                System.out.println(obj);
        }
}
```

Now we support all type of Collections!

Generics
[Wildcards and subtyping #4]

One more wildcard example:

```
public interface                    {
        …
        void putAll(                                                    )
        …
}
```

And another one:

```
public interface                          {
        …
        void addAll(                                                    )
        …
}
```

Generics
[Wildcards and subtyping #5]

Wildcards can be used also for declaring types:

```
// the following collection might be Collection<Shape>,
// but it can also be Collection<Circle> etc.
Collection<                    > shapes;
...

// the following is OK and is checked at compile-time!
Class<                         > clazz = shapes.getClass();

// the following is not OK (compilation error), why?
Class<                                                 > clazz
                                                       = shapes.getClass();
```

Generics
[Wildcards and subtyping #5 cont']

Wildcards for declaring types, cont':

```
// the following collection might be Collection<Shape>,
// but it can also be Collection<Circle> etc.
Collection<                    > shapes;
...

// Now, what can we do with the shapes collection?
// [1] Add - NOT allowed
shapes.add(new Shape()); // (compilation error)
// [2] but this is OK:
for(Shape shape: shapes) {
   shape.print(); // (assuming of course Shape has print func')
}
```

Generics
[Wildcards and super type]

Take a look at the following function signature in class Class<T>:

```
public Class<? super T> getSuperclass()
```

The keyword 'super' is used here to denote that the return type of

`Class<T>.getSuperclass()`

is going to be an object of type Class<? super T> and ? is obliged to be a super of T

The 'super' refers to any level of T or above (including T itself)

Generics
[Generic Methods]

Parameterized type can be added also to a function, example from the interface Collection:

```
public <T> T[] toArray(T[] arr)
```

The parameterized type T is not specified when calling the function, the compiler guesses it according to the arguments sent

Another Generic Method example, from the class Class:

And another one, from class java.utils.Collections:

Generics
[A final example]

The following is the max function from JDK 1.4 Collections class:

```
static public Object max(Collection coll) {
   Iterator itr = coll.iterator();
   if(!itr.hasNext()) {
      return null;
   }
   Comparable max = (Comparable)itr.next();
   while(itr.hasNext()) {
      Object curr = itr.next();
      if(max.compareTo(curr) < 0) {
         max = (Comparable)curr;
      }
   }
   return max;
}
```

When Sun engineers wanted to re-implement the max function to use generics in Java 5.0, what was the result?

Generics
[A final example]

The following is the JDK 5.0 max function (Collections class):

```
    Iterator<? extends T> itr = coll.iterator();
    if(!itr.hasNext()) {
       return null;
    }
    T max = itr.next();
    while(itr.hasNext()) {
       T curr = itr.next();
       if(max.compareTo(curr) < 0) {
          max = curr;
       }
    }
    return max;
}
```

Look at the &

For other interesting Generic examples, go to java.utils.Collections