Java Collections

Written by Amir Kirsh, Edited by Liron Blecher

Agenda

- Collections Overview
- Generics (basics)
- Concrete collections
- Generics (Advance)

Collections

- **Set of items**
- **Maintained in different approaches**
  - List (maintains order., allows duplications)
  - Set (group of unordered items. No duplications)
  - Map (unique key -> value pairs).

Collections

- **Interface Collection**
  - List \ Set
  - Defines general methods and capabilities for collections (iterator)
- **Interface Map**
  - Dictionaries (key=>value pairs)
- **Utility class Collections**
  - Sort
  - Search
- **Collections Concurrency**
  - Old one's are…
  - New ones are not (by default)

Collections Overview

1st Example:

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

Agenda

- Collections Overview
- Generics
- Concrete collections
- Generics (Advance)

Generic – Motivation ?

```java
public class IntegerBox {
    private int item;

    public IntegerBox(int item) {
        this.item = item;
    }

    public int getItem() {
        return item;
    }

    public void setItem(int item) {
        this.item = item;
    }

    public static void main(String[] args) {
        IntegerBox intBox = new IntegerBox( item: 78);
        System.out.println(intBox.getItem());
        intBox.setItem(30);
        System.out.println(intBox.getItem());
    }
}
```

Generic – Motivation ?

```java
public class StringBox {
    private String item;

    public StringBox(String item) {
        this.item = item;
    }

    public String getItem() {
        return item;
    }

    public void setItem(String item) {
        this.item = item;
    }

    public static void main(String[] args) {
        StringBox stringBox = new StringBox( item: "Moshe");
        System.out.println(stringBox.getItem());
        stringBox.setItem("Tikva");
        System.out.println(stringBox.getItem());
    }
}
```

Generic – Motivation ?

```java
public class ObjectBox {
    private Object item;

    public ObjectBox(Object item) {
        this.item = item;
    }

    public Object getItem() {
        return item;
    }

    public void setItem(Object item) {
        this.item = item;
    }
```

```java
public static void main(String[] args) {
    ObjectBox objectBox = new ObjectBox( item: 78);
    System.out.println(objectBox.getItem());
    int x = (Integer)objectBox.getItem();

    objectBox.setItem("Mosh");
    System.out.println(objectBox.getItem());
    //x = (Integer)objectBox.getItem();

    String item = (String)objectBox.getItem();
}
```

Generic – Motivation ?

Generics

- Generics helps abstract over a type
- Allows defining a class\algorithm once and use its essence with multiple ad-hoc implementations, varies only on the actual type(s) used within
- Arrived in JDK 5 (2004)
  - Compile time only feature !
  - Preserves backward compatibility
  - Avoid problematic castings in code

Usage: <Generic type name>

Generics – the solution !

- Upon definition

```
public class Box<T> {
    private T whatsInMe;
```

```
Box<String> aStringBox = new Box<>("A string");
```

- Upon usage

```
Box<Integer> anIntegerBox = new Box<>(5);
```

```
aStringBox.updateWhatsInMe(new Object());
```

```
Box<Integer> anIntegerBox = new Box<>("A string");
```

```
    }
```

collections.generics.motivation

DEMO

Collections Overview

2nd Example – now with Generics:

```
static public void main(String[] args) {
    List<String> argsList = new ArrayList<>();
    for(String str : args) {
        argsList.add(str); // argsList.add(7) would fail
    }
    if(argsList.contains("Koko") {
       System.out.println("We have Koko");
    }


}
```

Generics

Example 1 – Defining Generic Types:

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

}

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

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

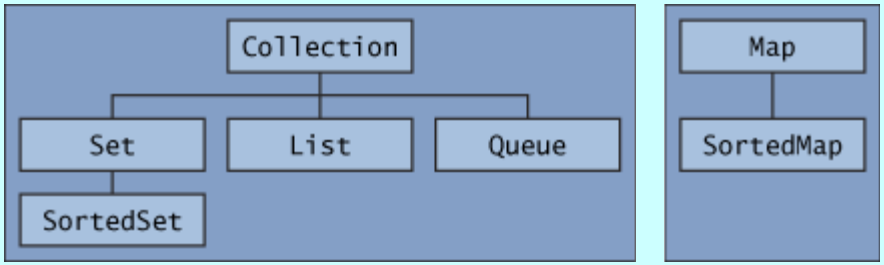
Agenda

- Collections Overview
- Generics
- Concrete collections
- Generics (Advance)

Which Collections do we have?

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

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



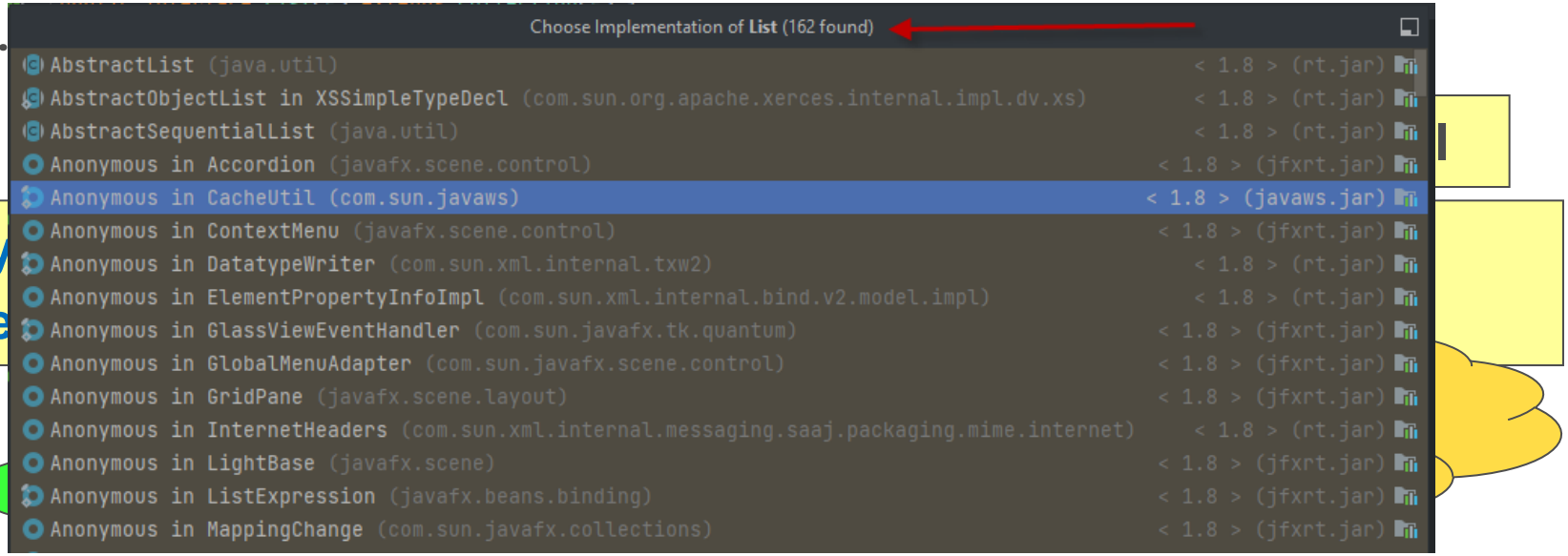
List of all Collections and related frameworks

Collections - Lists

ArrayList is a dynamically growable array with efficient access by index

Example:

```
List<Integer> arr = new ArrayList<>(10);
arr.
```



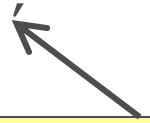
Collections - Lists

Vector also implements List interface.

Similarly to ArrayList, it is a dynamically growable array with efficient access by index

Example:

```
List<Integer> vec = new Vector<>(10);
vec.add(7);
```



initialCapacity is optional

Vector is an old (Java 1.0) container and is less in use today (since it is thread safe and is always synchronized).

It is replaced mainly by ArrayList (Java 1.2) (which is not synchronized)

Collections - Sets

HashSet is an hash table implementation.

Each element can exists exactly once

Example:

initialCapacity is optional

```
Set<Integer> set = new HashSet<>(10);
set.add(7);
```

HashSet is one implementation Set. (doesn't preserve insertion order)

LinkedHasSet is another implementation of Set interface (preserves insertion order)

TreeSet is another implementation of Set interface (maintains order between elements by comparison)

Collections - Maps

HashMap is a key-value hash table

Example 1:

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

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

Collections - Maps

Example 2:

```
Map<String, Integer> frequency(String[] names) {

                                                ;

}
```

Collections - Maps

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!

equals and hashcode revisited

For a class to properly serve as a key in HashMap or to be safely used in an HashSet or List, the equals and hashCode methods should both be appropriately implemented

Example:

```
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();
    }
}
```

Parameter MUST be Object (and NOT Person!)

collections.cats

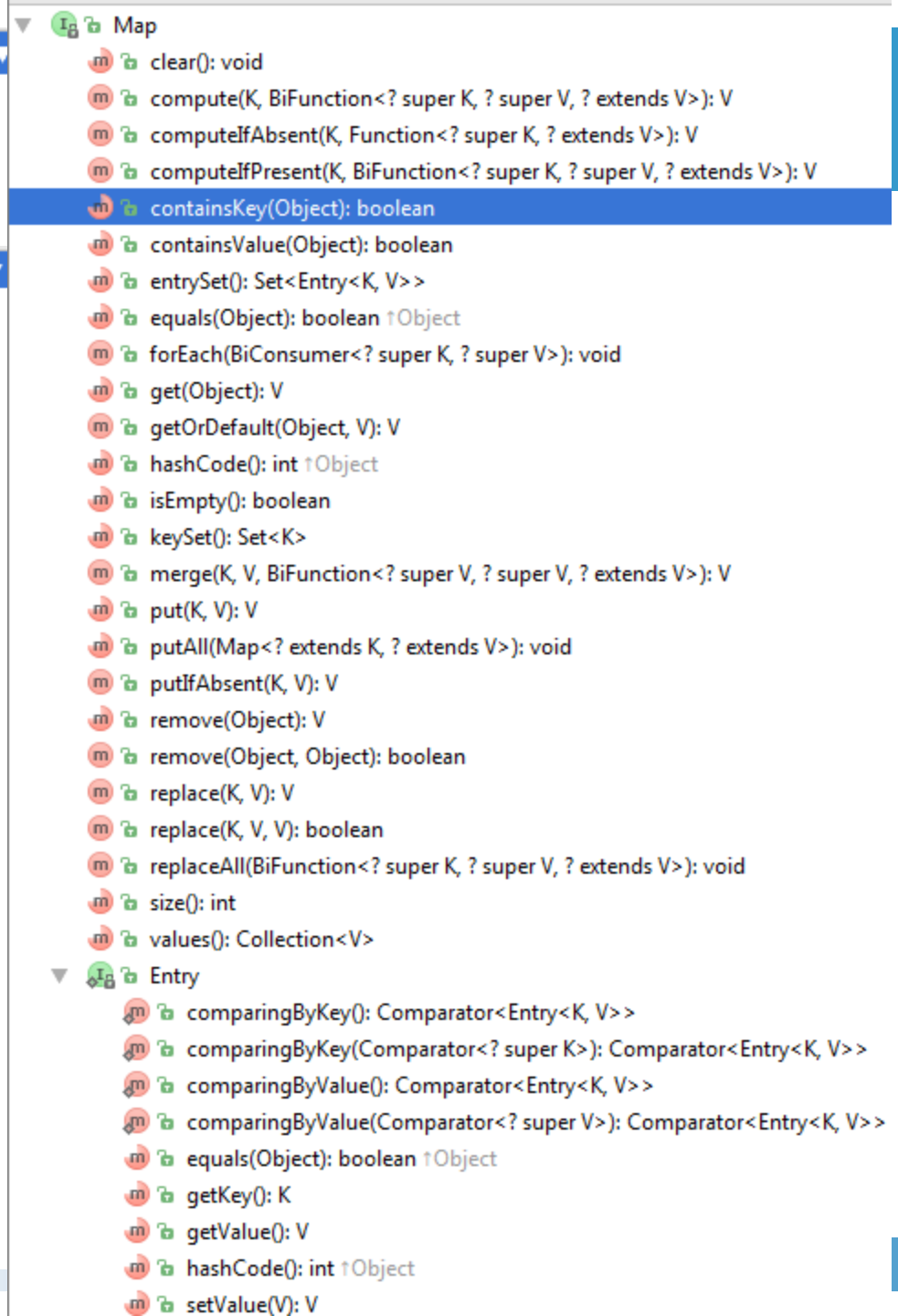
DEMO

Collections inte

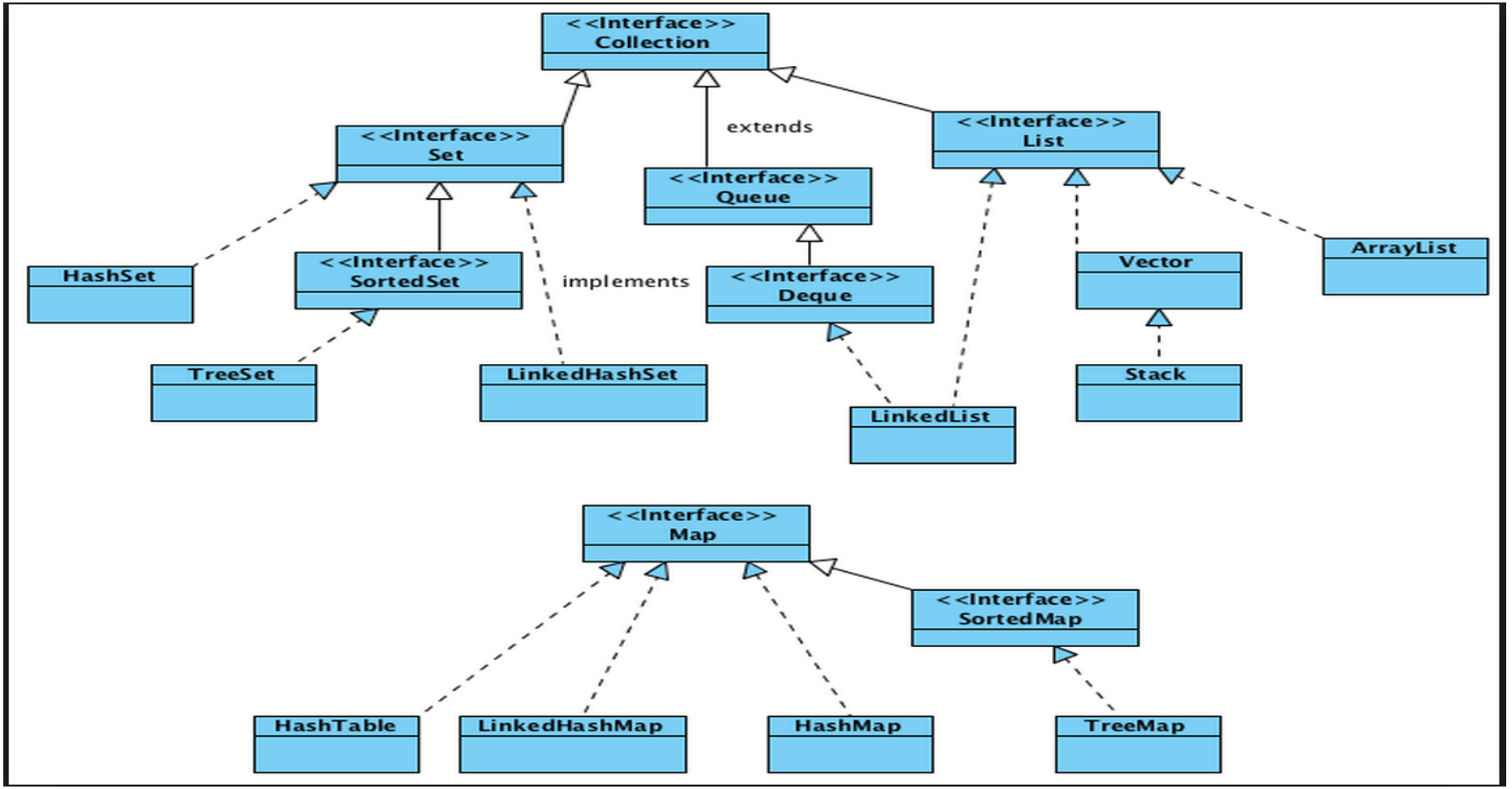
List interface:

Set interface:

Map interface:



Collections



Collections behavior summary

	ArrayList	Vector	LinkedList	HashMap	LinkedHashMap	Hashtable	TreeMap	HashSet	LinkedHashSet	TreeSet
Allows Null?	Yes	Yes	Yes	Yes (But one key and multiple values)	Yes (But one key and multiple values)	No	Yes (But zero keys and multiple values)	Yes	Yes	No
Allows Duplicates?	Yes	Yes	Yes	No	No	No	No	No	No	No
Retrieves Sorted results?	No	No	No	No	No	No	Yes	No	No	Yes
Retrieves same as Insertion order?	Yes	Yes	Yes	No	Yes	No	No	No	Yes	No

Collection Utils

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

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

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

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

collections.persons

DEMO

Agenda

- Collections Overview
- Generics
- Concrete collections
- Generics (Advance)

Generics – how it works

- First compilation phase:
  - Infers the types of generic classes
  - ensures that no wrong calls are done to generic classes

```
public class Box {
    private Object whatsInMe;

    public Box(Object whatsInMe) {
        this.whatsInMe = whatsInMe;
    }

    public Object extract() {
        return whatsInMe;
    }

    public void updateWhatsInMe(Object newThing) {
        whatsInMe = newThing;
    }
```

- Second compilation phase:
  - All generic types **are stripped away** and are replaced with Object class
  - Adding relevant casting statements where needed

```
Box aStringBox = new Box("A string");

String whatsInsideTheStringBox = (String)aStringBox.extract();
```

Generics
[Erasure implications #1]

Is the following possible?

```
public class GenericClass<T> {
        private T obj;
        …
        public void print() {
                System.out.println("obj type: " +
                        T.class.getName());
                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() {
                obj = new T();
        }
}
```

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?

```
if(obj instanceof T) {
...
}
```

Or:

```
if(someClass == T.class) {
...
}
```

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, using (e.g.) function signature)

Generics
[Erasure implications #4]

Is the following possible?

```
if(obj instanceof List<Integer>) {
...
}
```

Or:

```
if(someClass == List<Integer>.class) {
...
}
```

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

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

(To enforce a parameter of type List<Integer> we will have to use compile time checking, using (e.g.) function signature)

Generics
[Erasure implications #5A]

Is the following possible?

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

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;
myIntList = myRawList;
```

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**

⇒ **We could also:**

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

But: would still get an 'unchecked' warning from the compiler

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 !

collections.generics.erasure

DEMO

Generics – generic methods

Introduce a generic type to a specific method

- Exists in the method scope only

```
public static <T extends Number> Integer concatSomething(T something1, T something2) {
    return something1.intValue() + something2.intValue();
}

    int res = concatSomething(4, 5);   // but of course it is
    res = concatSomething(4, "sdf");
    res = concatSomething(4, 4.5);
```

collections.generics.method

DEMO

Generics Bound Types

A generic type can be bound to other types (inheritance-like)

'extends' -> the generic type must extend from a different type

Usage:

T extends <Type>

Generics [Bound Types]

Parameterized types can be restricted:

```
public class GenericSerializer<T extends Serializable> {
    …                                  <=
}
```

- Type T provided for our GenericSerializer class must implement Serializable
- Syntax is always "extends", also for interfaces

Multiple restrictions might be provided, separated by &:

```
public class Foo<T extends Comparable<T> & Iterable<T>> {
    …                    <=
}
```

Generics and inheritance

Inheritance **DOES NOT** apply to generics

Generics
[Wildcards and subtyping #1]

Is the following possible?

We know that the following is acceptable:

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

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

Answer is: NO (compilation error)

This comes to avoid the following:

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

Generics [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(Collection<Object> col) {
        for(Object obj : col) {
                System.out.println(obj);
        }
}
```

Can get ONLY collection of Objects

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

What's wrong with the 2nd implementation?

Generics Wildcard

Wildcard (ג'וקר) stands for ‘any type’ we need

- It can be viewed as the ‘unknown type’
- Use ? To signal the wildcard

e.g.

List<?> x

x is a list of a certain type. THAT type and ONLT that type. We simply don’t know what type it is.

- Used mainly in signatures arguments and\or return types, to ‘relax’ the compiler in some cases

Generics Wildcard

So… instead of this:

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

We would need to write something like this:

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

Now we support all type of Collections!

Generics Wildcard

```
List<Integer> li = Arrays.asList(1, 2, 3);
List<String>  ls = Arrays.asList("one", "two", "three");
printList(li);
printList(ls);
```

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
```

Generics Wildcard and subtyping

? Can be bounded

- Upper bound: ? Extends <type>
- Lower bound: ? Super <type> (pretty rare and far more complicated for this course)

```
public interface Map<K,V> {
        …
        void putAll(Map<? extends K, ? extends V> map)
        …
}
```

```
public interface Collection<E> {
        …
        void addAll(Collection<? extends E> coll)
        …
}
```

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<? extends Shape> shapes;
...

// the following is OK and is checked at compile-time!
Class<? extends Collection> 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<? extends Shape> shapes;
...
// Now, what can we do with the shapes collection?
// [1] Add - NOT allowed
shapes.add(new Square());// compilation error
shapes.add(new Circle()); // compilation error

// [2] but this is OK:
for(Shape shape: shapes) {
   shape.print(); // (assuming of course Shape has print func')
}
```

Why ?
Here and here

examples.collections.shapes

DEMO

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):

```
static public <T extends Object & Comparable<? super T>>
T max(Collection<? extends T> coll) {
   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

Generics
[References and further reading]

- http://docs.oracle.com/javase/tutorial/java/generics/
- http://gafter.blogspot.com/2004/09/puzzling-through-erasure-answer.html
- http://gafter.blogspot.com/2006/11/reified-generics-for-java.html
- http://www.mindview.net/WebLog/log-0058

Generics FAQ:

- http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html
- http://www.javacodegeeks.com/2013/07/the-dangers-of-correlating-subtype-polymorphism-with-generic-polymorphism.html

Concurrent package

- java.util.concurrent package

Collection questions

- 40 Java Collections Interview Questions and Answers