Generics

Lecture 9

What are “generics”?

- In Java, generics enable types (classes and interfaces) to be parameters when defining classes, interfaces, and methods

- The goal: allow code re-use while still allowing for strong error-detection at *compile-time* (i.e. as early as possible)

- Before we dive into the details, let’s examine a motivating example…

Remember: **`Comparable`**

The **`Comparable<T>`** interface allows you to specify that instances your class can be compared to instances of type **`T`** (could be the same). Goals:

- Create ordering among instances, thus…
- Support generic sorting algorithms/data structures

```
public interface Comparable<T> {
    // Returns a negative integer, zero, or a
    // positive integer as this object is less than,
    // equal to, or greater than the specified object.
    int compareTo(T o);
}
```

Example

Circle.java

```
public class Circle implements
                    Measurable, Comparable<Circle> {
      final private double radius;

      public Circle(double radius) {
            this.radius = radius;
      }

      @Override
      public double getArea() {
            return Math.PI*radius*radius;
      }

      @Override
      public double getPerimeter() {
            return 2.*Math.PI*radius;
      }

      @Override
      public int compareTo(Circle o) {
            return Double.compare(this.radius,
                                  o.radius);
      }
}
```

```
final Comparable<Circle> c1 = new Circle(1);

System.out.printf("%d%n",
      c1.compareTo((Circle) new Circle(2)));
```

So What is `<T>`?

- The author of the **`Comparable`** interface has allowed other software developers to explicitly specialize the interface for one or more "types" (i.e. classes/interfaces)

- To get a sense of why this is useful, let's consider what the **`Comparable`** interface looked like prior to generics…

JDK Comparison

JDK4

```
public interface Comparable {
    int compareTo(Object o);
}
```

JDK8

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

Crucial question: when are type errors detected?

```
Comparable c = new Date();
int x = c.compareTo("red");
```

Run Time :(

```
Comparable<Date> c = new Date();
int x = c.compareTo("red");
```

Compile Time :)

Example: Old-Style Implementation

```
public int compareTo(Object o) {
   return Double.compare(this.radius, (Circle o).radius);
}
```

```
int x = (new Circle(1)).compareTo("red");
```

```
Exception in thread "main" java.lang.ClassCastException:
java.lang.String cannot be cast to Circle
```

Consider the Following Warning

```
ArrayList a = new ArrayList();
a.add("red");
a.add(5);
```

```
ArrayList is a raw type. References to
generic type ArrayList<E> should be
parameterized
```

And the Results of Ignoring It…

```
ArrayList a = new ArrayList();
a.add("red");
a.add(5);

String s1 = (String) a.get(0);
String s2 = (String) a.get(1);
int x = s1.compareTo(s2);
```

```
Exception in thread "main"
java.lang.ClassCastException: java.lang.Integer
cannot be cast to java.lang.String
```

Compare To*…

```
ArrayList<String> a = new ArrayList<>();
a.add("red");
a.add(5);

String s1 = (String) a.get(0);
String s2 = (String) a.get(1);
int x = s1.compareTo(s2);
```

```
The method add(int, String) in the type ArrayList<String>
is not applicable for the arguments (int)

*pun intended :)
```

Compare To*…

```
ArrayList<String> a = new ArrayList<>();
a.add("red");
a.add("five");

String s1 = a.get(0);
String s2 = a.get(1);
int x = s1.compareTo(s2);
```

Not to mention, fewer unsightly casts

Lesson

Using generics (i.e. the <>'s) when creating/using types/methods can lead to…

- Generalizable code (i.e. write once, use in many situations), while…
- Fewer *run-time* errors, and…
- Lesser need to explicitly cast

A Small, but Useful, Example

- Consider a `MutableObject` with the following simple methods
  - Constructors (no-arg=null, or value)
  - toString (override)
  - Get (current value)
  - Set (to new value)

Solution

MutableObject.java

```
public class MutableObject<T> {
	private T value;

	public MutableObject(T initValue) {
		value = initValue;
	}

	public MutableObject() {
		this(null);
	}

	@Override
	public String toString() {
		return value==null?null:value.toString();
	}

	public T get() {
		return value;
	}

	public void set(T newValue) {
		value = newValue;
	}
}
```

Foo.java

```
MutableObject<String> s = new MutableObject<>();
System.out.printf("%s%n", s); // null
s.set("hi");
System.out.printf("%s%n", s); // hi
s.set("bye");
System.out.printf("%s%n", s); // bye

MutableObject<Integer> x = new MutableObject<>(5);
System.out.printf("%s%n", x); // 5
x.set(10);
System.out.printf("%s%n", x); // 10
```

Yet Another

Pair.java

```
public interface Pair<K,V> {
	K getKey();
	V getValue();
}
```

OrderedPair.java

```
public class OrderedPair<K,V> implements Pair<K, V> {
	final private K key;
	final private V value;

	public OrderedPair(K key, V value) {
		this.key = key;
		this.value = value;
	}

	@Override
	public K getKey() {
		return key;
	}

	@Override
	public V getValue() {
		return value;
	}
}
```

Usage

```
Pair<String, String> fb =
   new OrderedPair<>("Foo", "Bar");
System.out.printf("%s%s%n",
   fb.getKey(), fb.getValue()); // FooBar

Pair<String, List<Double>> balances =
   new OrderedPair<>("Alice",
      new ArrayList<Double>());
balances.getValue().add(2.7);
balances.getValue().add(3.14);
```

Type Parameter Naming Conventions

Parameter Name	Meaning
`E`	Element
`K`	Key
`N`	Number
`T`	Type
`V`	Value
`S, U, V, etc.`	2nd, 3rd, 4th, … types

An Example Method with Generics

Foo.java

```
public class Foo {
     public static <E> void print(E[] list) {
          for (int i = 0; i < list.length; i++)
               System.out.printf("%s ", list[i]);
     System.out.printf("%n");
}
```

```
String[] aS = {"hi", "there"};
Integer[] aI = {1, 2, 3, 4, 5};

Foo.<String>print(aS);
Foo.<Integer>print(aI);
```

Bounded Type Parameters

- When defining a type/method, you can restrict the types that can be used via the extends attribute

```
<T extends Class>
<T extends ClassA & IntB & IntC>
```

- In this context the **extends** keyword works for both subclass relationships as well as interface implementation
  – If multiple, and any is a class, must be the 1st

Example

```
public class NumericPair<K extends Number,V extends Number> implements Pair<K, V> {
	final private K key;
	final private V value;

	public NumericPair(K key, V value) {
		this.key = key;
		this.value = value;
	}

	@Override
	public K getKey() {
		return key;
	}

	@Override
	public V getValue() {
		return value;
	}

	public double sum() {
		return key.doubleValue() + value.doubleValue();
	}
}
```

Another Example

```
public static <E extends Comparable<E>> E max(E[] l) {
     E m = null;
     for (E v : l) {
          if (m == null || v.compareTo(m)>0) {
               m = v;
          }
     }
     return m;
}


Double[] ds = {1., 3., 2.7, 3.14};
String[] ss = {"a", "b", "c"};

System.out.printf("%s%n", max(ds)); // 3.14
System.out.printf("%s%n", max(ss)); // c
```

An Issue…

```
public class Foo {
    public static double sum(MutableObject<Number> a, MutableObject<Number> b) {
        return a.get().doubleValue() + b.get().doubleValue();
    }

    public static void main(String[] args) {
        final MutableObject<Integer> i1 = new MutableObject<>(1);
        final MutableObject<Integer> i2 = new MutableObject<>(2);

        System.out.printf("%.3f%n", sum(i1, i2));
    }
}
```

```
The method sum(MutableObject<Number>, MutableObject<Number>)
in the type Foo is not applicable for the arguments
(MutableObject<Integer>, MutableObject<Integer>)
```

Introducing Wildcards

- Problem: **`Integer`** is a subtype of **`Number`**, but **`MutableObject<Integer>`** is not a subtype of **`MutableObject<Number>`**
- When calling a method, you can use a wildcard in three ways…
  - Unbound: **`<?>`**
  - Upper bound: **`<? extends X>`**
    - Can be any subclass of **`X`**
  - Lower bound: **`<? super X>`**
    - Can be any class for which X is a subclass

Solution

```
public class Foo {
    public static double sum(MutableObject<? extends Number> a,
                             MutableObject<? extends Number> b) {
        return a.get().doubleValue() + b.get().doubleValue();
    }

    public static void main(String[] args) {
        final MutableObject<Integer> i1 = new MutableObject<>(1);
        final MutableObject<Integer> i2 = new MutableObject<>(2);

        System.out.printf("%.3f%n", sum(i1, i2));
    }
}
```

More Examples

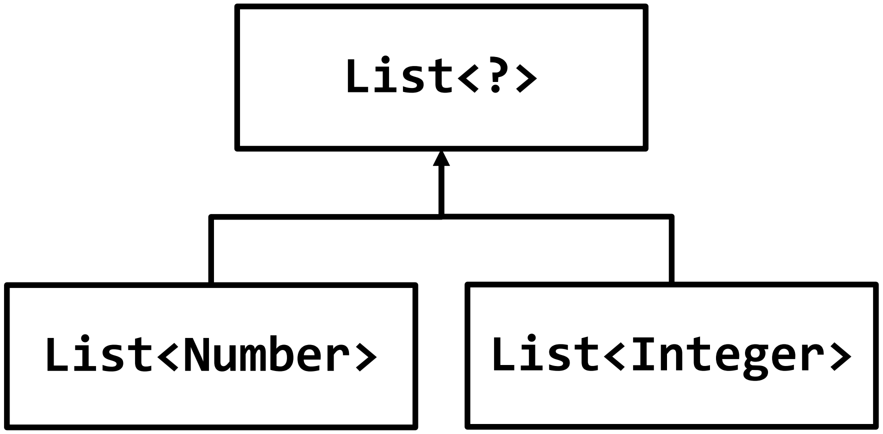
Unbound

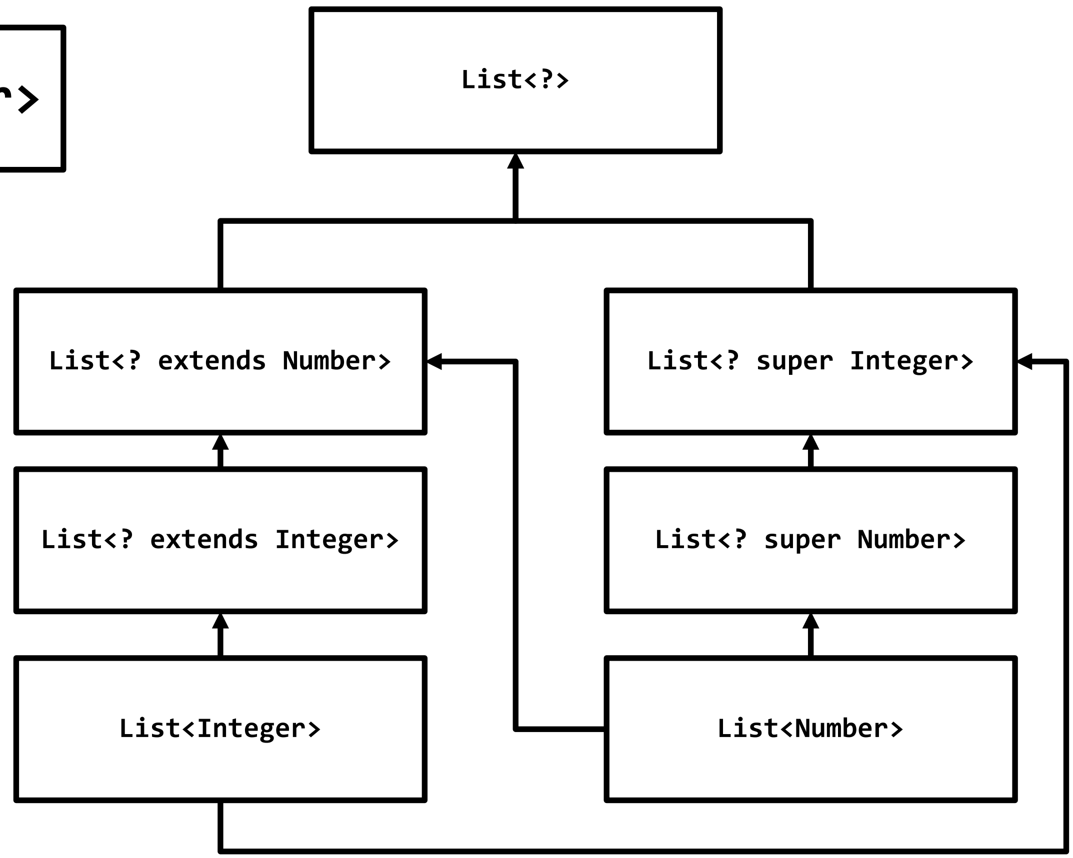
```
public static void print(List<?> list) {
	for (Object v : list)
		System.out.printf("%s ", v);
	System.out.printf("%n");
}
```

Lower Bound

```
public static void addAll(List<? super Integer> list,
                          int l, int u) {
	for (int i=l; i<=u; i++)
		list.add(i);
}
```

Visually


List<?>
List<Number>
List<Integer>


List<?>
List<? extends Number>
List<? super Integer>
List<? extends Integer>
List<? super Number>
List<Integer>
List<Number>

Issue: Type Erasure

- Generics are implemented using an approach called **type erasure**
- The compiler uses the generic type information to compile the code, but erases it afterwards
  - So the generic information is ***not*** available at run time (replaced with **`Object`** or bound)
- Benefits
  - Backward-compatible with legacy code that uses raw types
  - No run-time overhead

Example (1)

Pre-Compile

```
public class MutableObject<T> {
	private T value;

	public MutableObject(T initValue) {
		value = initValue;
	}

	...
}
```

Post-Compile

```
public class MutableObject {
	private Object value;

	public MutableObject(Object initValue) {
		value = initValue;
	}

	...
}
```

Example (2)

Pre-Compile

```
public static <T> int count(T[] anArray, T elem) {
        int cnt = 0;
        for (T e : anArray)
                if (e.equals(elem))
                        ++cnt;
        return cnt;
}
```

Post-Compile

```
public static int count(Object[] anArray, Object elem) {
        int cnt = 0;
        for (Object e : anArray)
                if (e.equals(elem))
                        ++cnt;
        return cnt;
}
```

Example (3)

Pre-Compile

```
public class NumericPair<K extends Number,V extends Number>
                                        implements Pair<K, V> {
        final private K key;
        final private V value;

        public NumericPair(K key, V value) {
                this.key = key;
                this.value = value;
        }

        ...
}
```

Post-Compile

```
public class NumericPair
                implements Pair {
        final private Number key;
        final private Number value;

        public NumericPair(Number key, Number value) {
                this.key = key;
                this.value = value;
        }

        ...
}
```

Restrictions on Generics (1)

A. Cannot create an instance of a generic type (remember: type erasure!), nor check via **instanceof**

```
new E(); // compiler error
x instanceof E // compiler error
```

B. Cannot create a static field whose type is a type parameter (remember: static types shared by all + type erasure!)

Restrictions on Generics (2)

C. Cannot create a generic array

```
new E[100]; // compiler error
```

D. Cannot overload a method where the formal parameter types of each overload erase to the same raw type

```
public void print(Set<String> strSet) { }
public void print(Set<Integer> intSet) { }
```

E. Cannot subclass an exception generically

– Think about the inability to catch differentially due to type erasure

Exercise

Implement a method to return the smallest element in an ArrayList

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

Solution

```
public static <E extends Comparable<E>> E min(ArrayList<E> list) {
    E m = null;
    for (E v : list) {
        if (m == null || v.compareTo(m)<0) {
            m = v;
        }
    }
    return m;
}
```

Take Home Points

- Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces, and methods
  - By explicitly typing, errors from general code can be caught at compile time!
- Wildcards allow methods to express hierarchical generic types
- After compiling, all generics are removed for backwards compatibility
  - Due to this type erasure, there are some unintuitive restrictions to using generics