

# COMP 249: Object Oriented Programming II

Tutorial 11:  
Generics

# What are generics and why use them?

Generics enable types (Classes and interfaces) to be parameters when defining classes, interfaces and methods. Some advantages of using generics:

- ▶ **Type Safety:** Generics enable you to write code that is type-safe. This means that the compiler can catch type-related errors at compile time, rather than allowing them to manifest at runtime.
- ▶ **Code Reusability:** Generics allow you to write functions, classes, or data structures that can work with different data types without having to duplicate code for each type.

# What are generics and why use them?

- ▶ Abstraction: Generics provide a level of abstraction by allowing you to create generic algorithms or data structures.
- ▶ Performance: Generics can be more efficient than alternatives like using object types, as they can avoid the need for boxing and unboxing (converting between value and reference types) and allow for better compiler optimization.

# A not generic Box class

Consider the following Box class which is not generic:

```
public class Box {  
    private Object object;  
    Box(Object object) {this.object = object;}  
    public void set(Object object) { this.object = object; }  
    public Object get() { return object; }  
}
```

The methods store and return an Object. Although we know the object types when we store them in Box, it becomes harder to keep track of their runtime-types later, especially if they are added to a list and later re-organized...

Whenever we “get” a Box from the list, and then “get” the object it holds, we would have to check its instance type before casting. This adds a lot of execution and painful programming overhead. If many different types of objects are stored in the different Box instances in the list, this becomes even worse.

```
/** In main(...) */
```

```
ArrayList<Box> li = new ArrayList<Box>();
```

```
li.add(new Box( new Object() ));
```

```
li.add(new Box( new String() ));
```

```
li.add(new Box( new Integer(24) ));
```

```
Collections.shuffle(li);
```

```
Box b = li.get(0);
```

```
/** At this point, we don't know what the runtime  
type of b.get() is. It could be a String, an Integer,  
or an Object. If we cast to the incorrect class, we  
will get a runtime error */
```

```
Object o = b.get();
```

```
/** If b.get() is a:
```

- Object in memory, both casts would fail (though the first will cause the crash)
- String in memory, the first cast will pass, the second would fail
- Integer in memory, the first cast will fail, the second would pass (but the first will crash the program).

```
*/
```

```
String s = (String) o;
```

```
Integer l = (Integer) o;
```

# Example of Generics

As an example, consider following code for finding the maximal element in the range [begin, end) of a list.

```
import java.util.*;

public final class Algorithm {
    public static <T extends Object & Comparable<? super T>>
        T max(List<? extends T> list, int begin, int end) {

        T maxElem = list.get(begin);

        for (++begin; begin < end; ++begin)
            if (maxElem.compareTo(list.get(begin)) < 0)
                maxElem = list.get(begin);
        return maxElem;
    }
}
```

# Question 1: A simple generic class

Write the same Box class as a generic. Your generic class should:

- ▶ Store a value of type T as a private variable.
- ▶ Implement accessor and mutator methods for that variable.
- ▶ Implement a toString() method describing the box and its content.

You should be able to use this class like this:

```
Box<Integer> myIntegerBox = new Box<Integer>();  
myIntegerBox.set(new Integer(10));  
System.out.println(myIntegerBox);
```

```
Box<String> myStringBox = new Box<String>();  
myStringBox.set(new String("For sale, baby shoes, never worn"));  
System.out.println(myStringBox);
```

## Question 2: Multiple Types

A generic class can also take multiple type parameters. Let's build an `OrderedPair` generic class. Our generic class should:

- ▶ Store values of type parameters `T` and `S` in private variables.
- ▶ Implement the accessor and mutator for each value.
- ▶ Implement the `toString()` method describing the pair and their values.

You should be able to use this class like this:

```
OrderedPair<Integer,String> myPair = new OrderedPair<Integer,String>();  
myPair.setFirst(new Integer(1));  
myPair.setSecond(new String("So much depends on a red wheelbarrow"));  
System.out.println(myPair);
```

## Question 3: Bounded Type Parameters

At times, you might want to limit the types that can be used by a generic class. This can be accomplished with bounded type parameters using the *extends* word in the type definition. Let's create a `NumberBox` class that limits what our `Box` can contain to a `Number` class object.

We should be able to use the first part of the following code, but generate an error on the second:

```
// This works:
```

```
NumberBox<Integer> myIntegerBox = new NumberBox<Integer>();  
myIntegerBox.set(new Integer(10));  
System.out.println(myIntegerBox);
```

```
// This will result in a compile time error:
```

```
NumberBox<String> myStringBox = new NumberBox<String>();  
myStringBox.set(new String("I am not a Number!"));  
System.out.println(myStringBox);
```



# Bounded Type Parameters with Multiple Bounds

Bounded type parameters can also have multiple bounds. For example, you could ensure that your type is an instance of a certain class and has access to certain interfaces:

```
public class GenericClass<T extends Shape & Interface1 & Interface2>
{
    // ...
}
```

That way, you are ensuring that the variables and methods provided by Shape, Interface1 and Interface2 will be available to your GenericClass.

# Question 4: Bounded Type Parameters with Multiple Bounds

Consider the following Shape class:

```
abstract class Shape {  
    private int height,width;  
  
    public Shape() { this.height = 0; this.width = 0; }  
    public Shape(int h, int w) { this.height = h; this.width = w; }  
  
    public void setHeight(int i) { this.height = i; }  
    public void setWidth(int i) { this.width = i; }  
  
    public int getHeight() { return this.height; }  
    public int getWidth() { return this.width; }  
}  
  
public interface ShapeInterface {  
    String printShape();  
}
```

First, write a Rectangle class which extends Shape and implements ShapeInterface. The printShape method should return a String of height by width characters.

## Question 4: Bounded Type Parameters with Multiple Bounds

Next, create a new generic class which:

- ▶ Bounds its accepted type parameter to the Shape class and the ShapeInterface interface.
- ▶ Works just like our generic Box class (holds a single instance of its generic-type object), with accessors and mutators.
- ▶ Returns the String generated by printShape() in the toString() method.

# References

- ▶ The Java Tutorial on Generics

[docs.oracle.com/javase/tutorial/java/generics/index.html](https://docs.oracle.com/javase/tutorial/java/generics/index.html)

- ▶ Another Tutorial on the same subject, using different examples

[docs.oracle.com/javase/tutorial/extra/generics/index.htm](https://docs.oracle.com/javase/tutorial/extra/generics/index.html)

l