8. Java: Generics and Annotations

UNIVERSITÄT
BERN

Generics and Annotations

Sources

- David Flanagan, *Java in a Nutshell*, 5th Edition, O'Reilly.
- GoF, *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison Wesley,1997.
- Gilad Bracha, *Generics in the Java Programming Language, 2004*

Roadmap

- Generics
- Annotations
- Model-Driven Engineering

Roadmap

- **Generics**
- Annotations
- Model-Driven Engineering

Why do we need generics?

Generics allow you to *abstract* over *types*.
The most common examples are container types, the collection hierarchy.

Motivating Example – Old Style

```
List stones = new LinkedList();
stones.add(new Stone(RED));
stones.add(new Stone(GREEN));
stones.add(new Stone(RED));
Stone first = (Stone) stones.get(0);
```

The cast is annoying but essential!

```
public int countStones(Color color) {
    int tally = 0;
    Iterator it = stones.iterator();
    while (it.hasNext()) {
        Stone stone = (Stone)
it.next();
        if (stone.getColor() == color)
{
            tally++;
        }
    }
    return tally;
}
```

Motivating example – new style using generics

List is a *generic interface* that takes a type as a *parameter*.

```
List<Stone> stones = new
LinkedList<>();
stones.add(new Stone(RED));
stones.add(new Stone(GREEN));
stones.add(new Stone(RED));
Stone first = /*no cast*/
stones.get(0);
```

```
public int countStones(Color color) {
    int tally = 0;
    /*no temporary*/
    for (Stone stone : stones) {
        /*no temporary, no cast*/
        if (stone.getColor() == color)
{
            tally++;
        }
    }
    return tally;
}
```

Compile Time vs. Runtime Safety

Old way

```
List stones = new LinkedList();
stones.add("ceci n'est pas un stone");

...

Stone stone = (Stone) stones.get(0);
```

No check, unsafe

Runtime error

New way

```
List<Stone> stones = new LinkedList<Stone>();
stones.add("ceci n'est pas un stone");

...

Stone stone = stones.get(0);
```

Compile time check

Runtime is safe

Stack Example

```
public interface StackInterface {
    public boolean isEmpty();
    public int size();
    public void push(Object item);
    public Object top();
    public void pop();
}
```

Old way

```
public interface StackInterface<E> {
    public boolean isEmpty();
    public int size();
    public void push(E item);
    public E top();
    public void pop();
}
```

New way:
we define a generic interface that takes a type parameter

Linked Stack Example

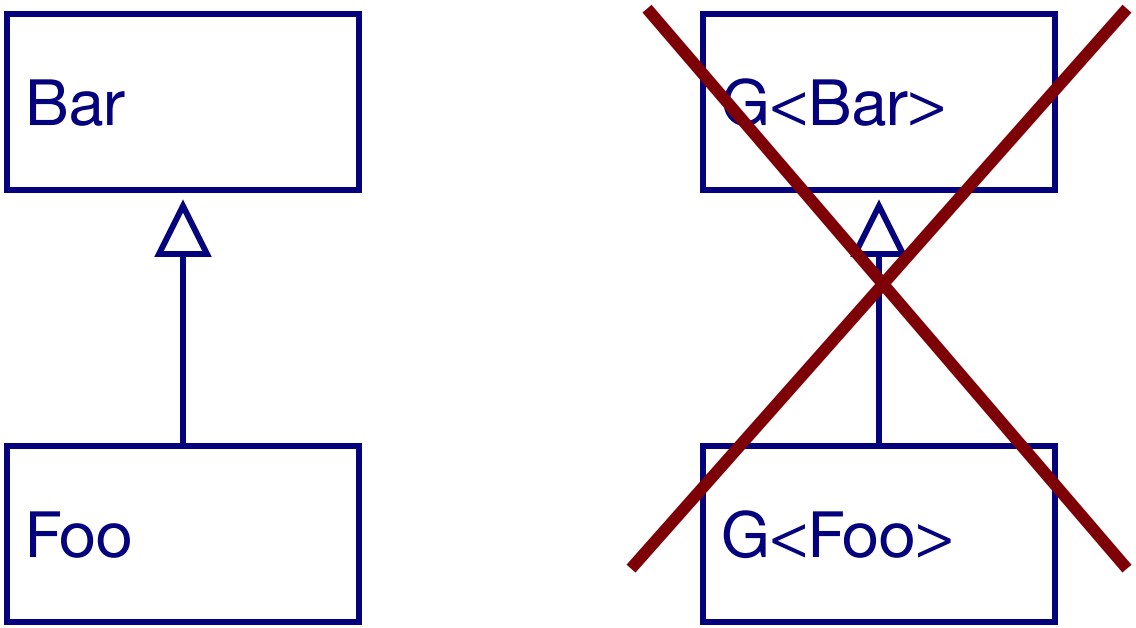
```
public class LinkStack<E> implements StackInterface<E> {
…
    public class Cell {
        public E item;
        public Cell next;
        public Cell(E item, Cell next) {
            this.item = item;
            this.next = next;
        }
    }

…
    public E top() {
        assert !this.isEmpty();
        return top.item;
    }
```

Creating a Stack of Integers

```
Stack<Integer> myStack = new LinkedStack<Integer>();
myStack.push(42); // autoboxing
```

When a generic is instantiated, the *actual type parameters* are substituted for the *formal type parameters*.

Generics and Subtyping



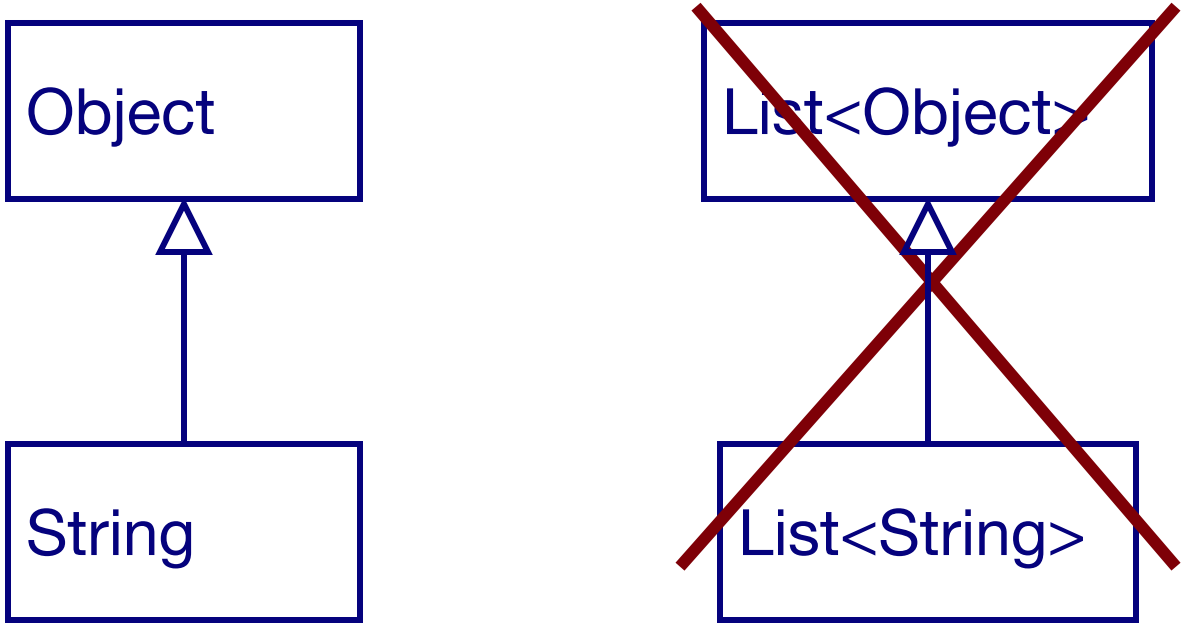
In Java, Foo is a subtype of Bar only if Foo's interface *strictly includes* Bar's interface. Instantiated generics normally have *different* interfaces. *(I.e., if the type parameters are used in the public interface.)*

Generics and Subtyping (II)

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

List<Object> lo = ls;

lo.add(0, new Object()); // legal?!
ls.get(0); // Not a string?!
```

Compile error as it is not type safe!

In other words…



Wildcards

```
void printCollection(Collection c) {
    Iterator i = c.iterator();
    while (i.hasNext()) {
        System.out.println(i.next());
    }
}
```

We want a method that prints our all the elements of a collection

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

Here is a naïve attempt at writing it using generics

```
printCollection(stones);
```

Won't compile!

What type matches all kinds of collections?

```
Collection<?>
```

"collection of unknown" is a collection whose element type matches anything — **a wildcard type**

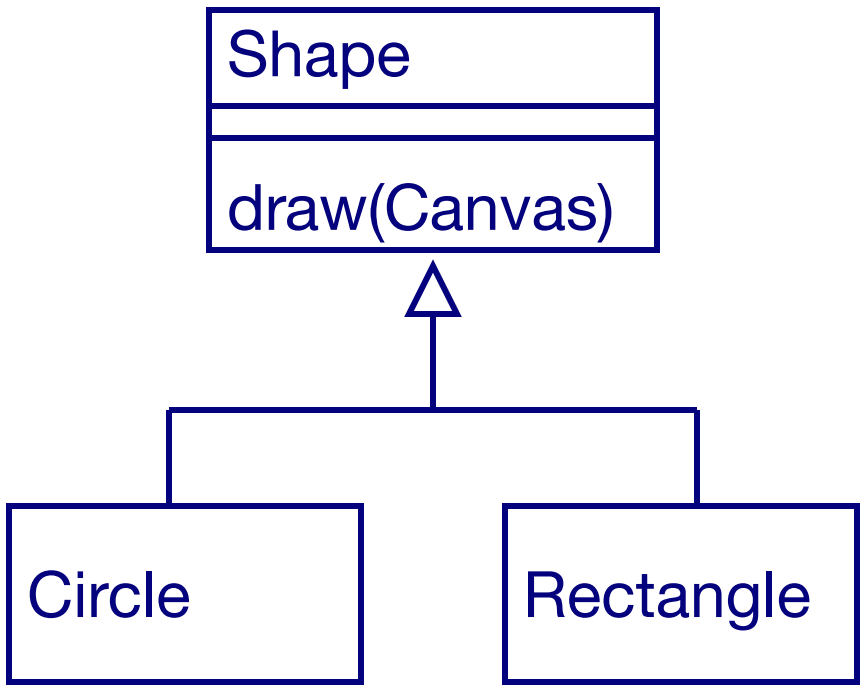
```
void printCollection(Collection<?> c) {
    for (Object e: c){
        System.out.println(e);
    }
}
```

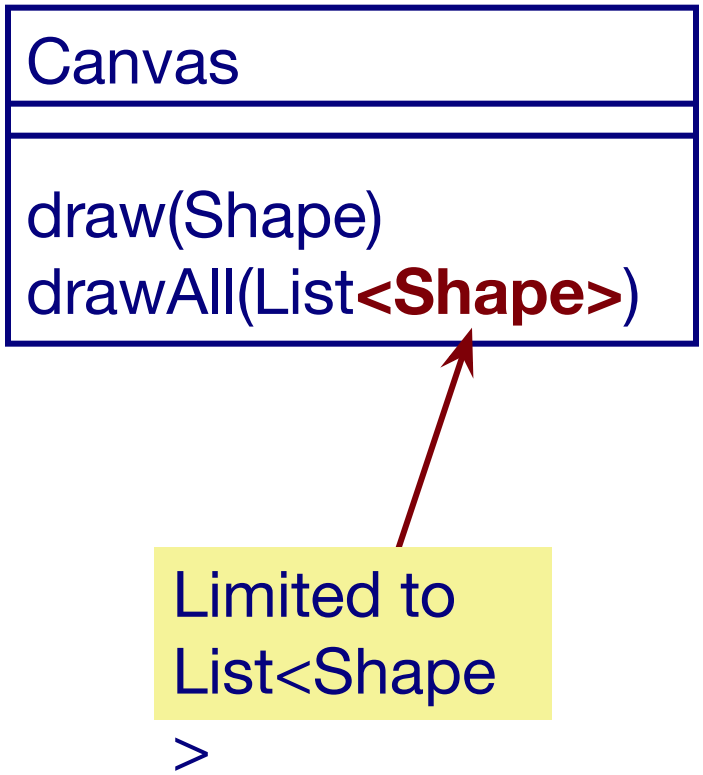
```
printCollection(stones);
```

```
stone(java.awt.Color[r=255,g=0,b=0])
stone(java.awt.Color[r=0,g=255,b=0])
stone(java.awt.Color[r=0,g=255,b=0])
```

Bounded Wildcards

Consider a simple drawing application to draw shapes (circles, rectangles,…)





A Method that accepts a List of any kind of Shape…

```
public void drawAll(List<? extends Shape>) {…}
```

a bounded wildcard

Shape is the ***upper bound*** of the wildcard

More fun with generics

```
import java.util.*;
 …

   public void pushAll(Collection<? extends E> collection)
{
       for (E element : collection) {
           this.push(element);
       }
   }

   public List<E> sort(Comparator<? super E> comp) {
       List<E> list = this.asList();
       Collections.sort(list, comp);
       return list;
   }
```

All elements must be *at least* an E

The comparison method must require *at most* an E

License

http://creativecommons.org/licenses/by-sa/2.5/