

# Generics

Adding Type Safety and Code Reusability



**SoftUni Team**  
Technical Trainers



**SoftUni**



Software University

<https://softuni.bg>

sli.do

**#java-advanced**

# Table of Contents

1. The **Problem** before Java 5.0
2. Generics **Syntax**
3. Generic **Classes** and **Interfaces**
4. Generic **Methods**
5. Type Erasure, Type Parameter Bounds





# Generics

The Problem and the Solution

# The Problem Before Java 5.0

- We need a collection that will store **only strings**

```
List strings = new ArrayList();  
strings.add("1");  
strings.add("2");  
strings.add(3); // Is this correct?  
String e1 = (String) strings.get(0);  
String e2 = (String) strings.get(1);  
String e3 = (String) strings.get(2); // RTE
```

- We need a collection that will store **only strings**

```
List<String> strings = new ArrayList<String>();  
strings.add("1");  
strings.add("2");  
strings.add(3); // Compile time error
```

- Adds **type safety** and provides a powerful way for **code reuse**

```
List<Integer> integers = new ArrayList<>();  
List<Person> people = new ArrayList<>();
```

Type Inference

- Defined with **<Type Parameter 1, Type Parameter 2 ... etc.>**

```
class ArrayList<T> {  
    /* magic */  
}
```

- **Multiple** Type Parameters

```
class HashMap<K, V> {  
    /* magic */  
}
```

- You can use it anywhere inside the **declaring class**

```
class List<T> {  
    public void add (T element) {...}  
    public T remove () {...}  
    public T get(int index) {...}  
}
```



# Problem: Jar of T

- Create a class **Jar<>** that can store **anything**
- Adding should add **on top** of its contents
- Remove should get the **topmost** element
- It should have two public methods:
  - **void add(element)**
  - **element remove()**

```
public class Jar<T> {  
    private Deque<T> content;  
    public Jar() { this.content = new ArrayDeque<>(); }  
  
    public void add(T entity) {  
        this.content.push(entity);  
    }  
  
    public T remove() { return this.content.pop(); }  
}
```

# Subclassing Generic Classes

- Can **extend** to a concrete class

```
class JarOfPickles extends Jar<Pickle> {  
    ...  
}
```

```
JarOfPickles jar = new JarOfPickles();  
jar.add(new Pickle());  
jar.add(new Vegetable()); // Error
```

- **Generic interfaces** are similar to **generic classes**

```
interface List<T> {  
    void add (T element);  
    T get (int index);  
    ...  
}
```

```
class MyList implements List<MyClass> {...}
```

```
class MyList<T> implements List<T> {...}
```

# Problem: Generic Array Creator

- Create a class **ArrayCreator** with a single method:
  - **static T[] create(int length, T item)**
- Add a single overload:
  - **static T[] create(Class<T> class, int length, T item)**
- It should **return an array**
  - with the given length
  - every element should be **set to the given default item**

# Solution: Generic Array Creator (1)

```
public static <T> T[] create(int length, T item) {  
    T[] array = (T[]) new Object[length];  
    for (int i = 0; i < array.length; i++) {  
        array[i] = item;  
    }  
    return array;  
}
```

# Solution: Generic Array Creator (2)

```
public static <T> T[] create(  
    Class<T> cl, int length, T item) {  
    T[] array = (T[]) Array.newInstance(cl, length);  
    for (int i = 0; i < array.length; i++) {  
        array[i] = item;  
    }  
    return array;  
}
```

- Generics are **compile time illusion**

```
List<String> strings = new ArrayList<String>();  
System.out.println(strings instanceof List);  
  
System.out.println(  
    strings instanceof List<String>); // CTE
```

- Compiler **deletes** all angle bracket syntax
- Adds **type casts** for us (presented in byte-code)



# Type Erasure – Example

```
public class Illusion<T> {  
    public void function(Object obj) {  
        if (obj instanceof T) {} // Error  
        T[] array = new T[1]; // Error  
        T newInstance = new T(); // Error  
        Class c1 = T.class; // Error  
    }  
}
```



# Type Parameter Bounds

Upper and Lower Bounds

- **<T extends Class>** - specifies an "Upper bound"

```
class AnimalList<T extends Animal> {  
    private List<T> animals;  
    void add (T animal) {...}  
    void putAnimalsToSleep() {  
        for (Animal a : this.animals)  
            a.sleep();  
    }  
}
```

**T** will be a subclass of  
Animal

Uses methods of **T**

# Problem: Generic Scale

- Create a class **Scale<T>** that:
  - Holds two elements: **left** and **right**
  - Receives the elements through its single constructor:
    - **Scale(T left, T right)**
  - Has a method: **T getHeavier()**
- The greater of the two elements is heavier
- Should return **null** if the elements are equal



# Solution: Generic Scale (1)

```
public class Scale<T extends Comparable<T>> {  
    private T left;  
    private T right;  
    public Scale(T left, T right) {  
        this.left = left;  
        this.right = right;  
    }  
    public T getHeavier() { /* next slide */ }  
}
```

# Solution: Generic Scale (2)

```
public T getHeavier() {  
    if (this.left.compareTo(this.right) == 0)  
        return null;  
    if (this.left.compareTo(this.right) < 0)  
        return right;  
    return left;  
}
```

# Problem: List Utilities

- Create a class **ListUtils** that:
  - Has two static methods:
    - **T getMin(List<T> list)**
    - **T getMax(List<T> list)**
  - Should throw **IllegalArgumentException** if an empty list is passed



# Solution: List Utilities

```
public static <T extends Comparable<T>> T getMax(List<T> list) {  
    if (list.size() == 0) throw new IllegalArgumentException();  
    T max = list.get(0);  
    for (int i = 1; i < list.size(); i++) {  
        if (max.compareTo(list.get(i)) < 0)  
            max = list.get(i);  
    }  
    return max;  
}
```



# Type Parameters Relationships (1)

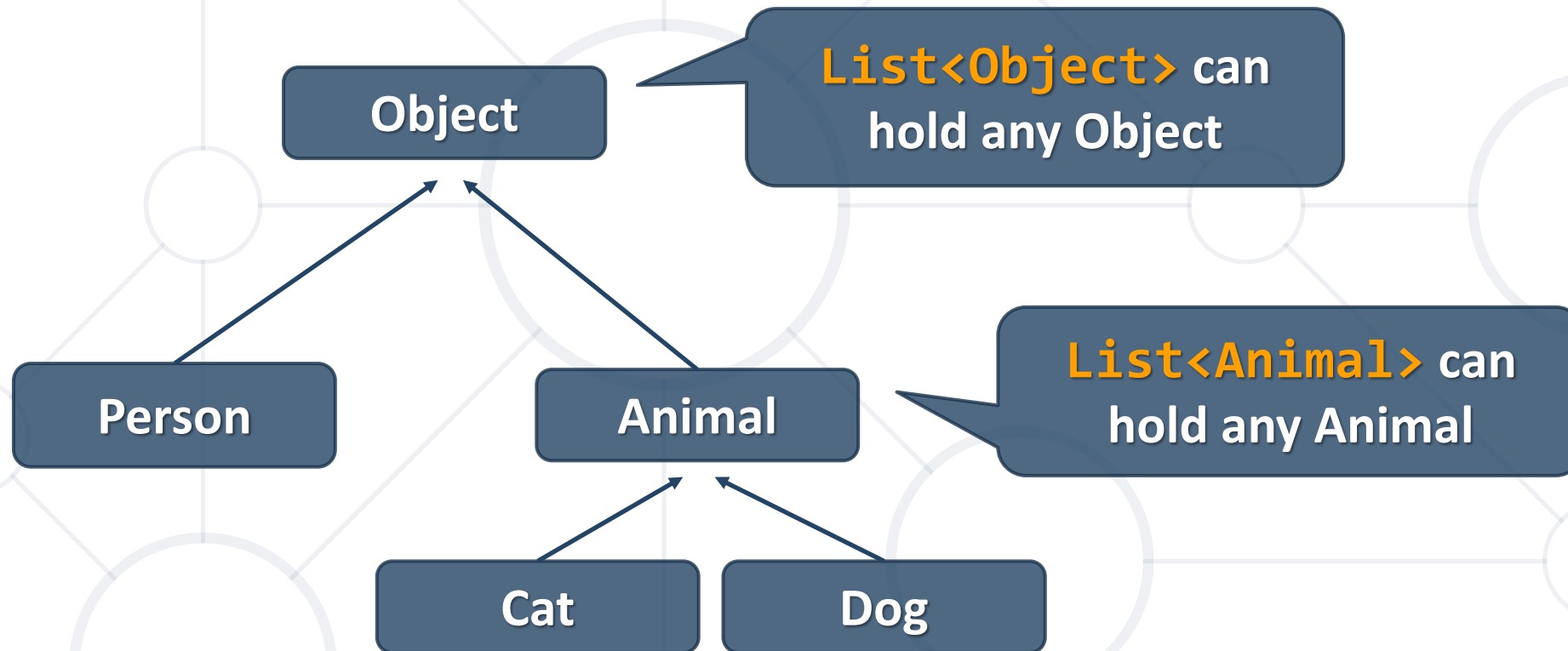
- Generics are **invariant**

```
List<Object> objects = new ArrayList<>>();  
List<Animal> animals = new ArrayList<>>();  
objects = animals; // Compile Time Error!
```

- If the above was possible, then why not:

```
objects = animals;  
objects.add(new Person()); // Impossible!
```

# Type Parameters Relationships (2)



**List<Object> ≠ List<Animal>**

- Generics add **type safety**
- Generic code is **more reusable**
- **Classes, interfaces and methods can be generic**
- Runtime information about **type parameters** is lost due to **erasure**



# SoftUni Diamond Partners



**SCHWARZ**



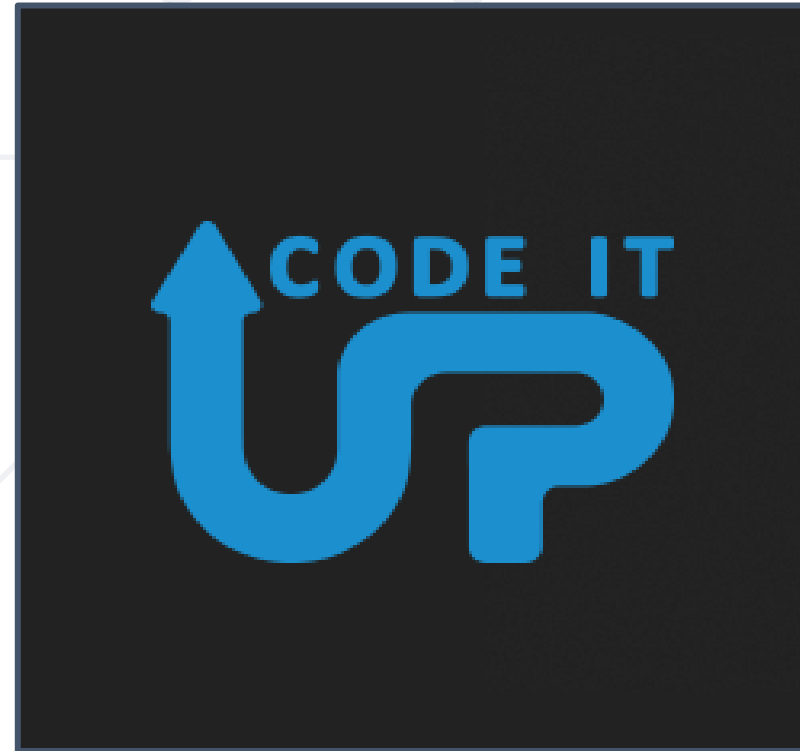
**SUPER  
HOSTING  
.BG**



**INDEAVR**  
Serving the high achievers

**Bosch.io**





- Software University – High-Quality Education, Profession and Job for Software Developers

- [softuni.bg](https://softuni.bg)

- Software University Foundation

- [softuni.foundation](https://softuni.foundation)

- Software University @ Facebook

- [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)

- Software University Forums

- [forum.softuni.bg](https://forum.softuni.bg)



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>

