Java Enterprise Application Development

Lecture 7
Generics

Dr. Fan Hongfei

20 October 2025

Review

- Generic programming in C++
  - Function template
  - Class template
  - Standard Template Library (STL)
    - Containers: vector, list, map, set, …
    - Algorithms: sort, find, binary_search, …
    - Iterators: begin(), end()
    - Function objects

Introduction to Generics

- Example without generics

```
1  class ArrayList {
2      private Object[] elementData;
3      public Object get(int i) {...}
4      public Object add(Object o) {...}
5  }
```

- Supporting any type of object, but always leading to bugs

- **Type casting** is **necessary**

```
1  ArrayList myList = new ArrayList();
2  String str = (String) myList.get(0);
3  //...
4  myList.add(Integer.valueOf(233));
```

3

Introduction to Generics (cont.)

- **Generics**: enabling **types** (classes, interfaces) to be **parameters**

  - Stronger type checks at compile time

  - Elimination of casting

  - Enabling the implementation of generic algorithms

```java
1  List<String> strList = new ArrayList<>();
2  strList.add("Hello");
3  // no cast
4  System.out.println(strList.get(0));
5  // compile-time error
6  // strList.add(Integer.valueOf(233));
```

Generic Types

- A generic class or interface that is parameterized over types
- Defined with the following format

  class ClassName<T1, T2, ..., Tn> {...}

- A type variable can be any **non-primitive** type
- Invoking and instantiating easily

  ClassName<Type> instance = new ClassName<Type>();

- Diamond: compiler **infers** type arguments from context

  List<String> myList = new ArrayList<>();
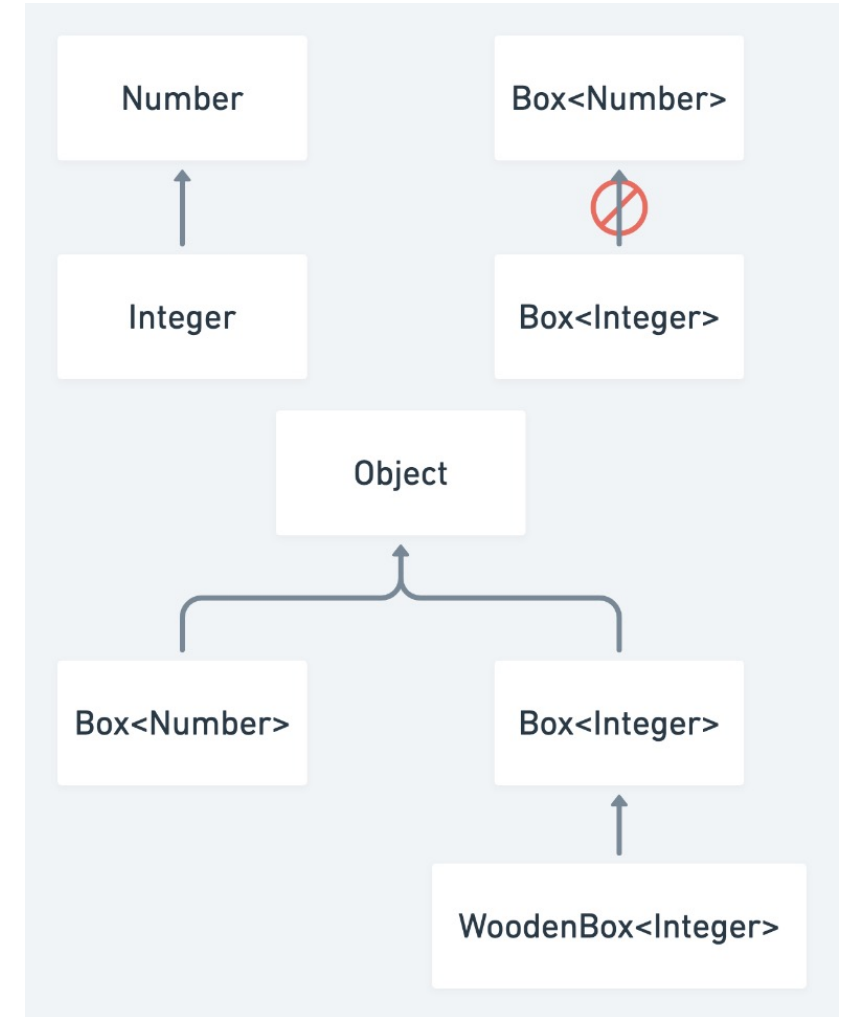
Generic Types (cont.)

- Type parameter naming **conventions**
  - T -> Type
  - S, U, V etc. -> 2nd, 3rd, 4th types
  - E -> Element
  - N -> Number
  - K -> Key
  - V -> Value

Generic Methods

- Methods that introduce their **own** type parameters

- Both static and non-static methods are allowed

- Syntax: a list of type parameters, **inside angle brackets**, appearing before the return type

- For static generic methods, the type parameter section must appear before the method's return type

Generics and Inheritance

- Arrays are **covariant**

  - T[] may contain elements of T or any subtype of T

  - S[] is a subtype of T[] if S is a subtype of T

- Java generics are **invariant**

  - MyClass<S> has no relationship to MyClass<T>

  - Subtype a generic class or interface by extending or implementing it

Bounded Type Parameters and Wildcards

- Bounded type parameters
  - **Restrict** the types that can be used as type arguments
  - List the type parameter's name, followed by *extends*
  - Multiple bounds are supported, **using "&"**
- Wildcard: "?" representing an unknown type
  - Used as the type of a parameter, field, local variable or return type
  - *List<?>, Box<?>, Map<?, ?>...*
  - **Upper bounded, lower bounded, and unbounded**

Bounded Type Parameters and Wildcards (cont.)

- **Unbounded** wildcards
  - Examples: *List<?>, Box<?>*
  - Scenarios
    - Using functionality provided in the *Object* class
    - Using methods in the generic class that do not depend on the type parameter
- **Upper bounded** wildcards
  - Restriction on the variable: **? extends**
  - Restricting the unknown type to be a specific type or a subtype of that type: *List<? extends Number>, Box<? extends Integer>...*
- **Lower bounded** wildcards
  - Restriction on the variable: **? super**
  - Restricting the unknown type to be a specific type or a super type of that type: *List<? super Integer>...*

Type Erasure

- The compiler erases all type parameters, ensuring that no new classes are created for parameterized types

  – Replace all type parameters in generic types with their bounds, or *Object* if the type parameters are unbounded

  – Insert type casts if necessary to preserve type safety

- Incurring no runtime overhead

  – Only raw types during runtime

Best Practice

- Do not use raw types
  - Except for class literals: *List.class, String[].class* (*List<String>.class* is illegal in java)

- Prefer lists to arrays

- Use bounded wildcards to increase API flexibility
  - public void pushAll(Iterable<E> src) (x)
  - public void pushAll(Iterable<? extends E> src) (√)

Further Reading

- Chapter 8 Generic Programming in *Core Java (10th Edition)*

- Item 26-33 in *Effective Java (3rd Edition)*

- Lesson: Generics
  https://docs.oracle.com/javase/tutorial/java/generics/index.html