CS2030 Lecture 5

Generics and Variance of Types

Henry Chia (hchia@comp.nus.edu.sg)

Semester 2 2019 / 2020

Lecture Outline

- Generics
  - Generic type, classes and methods
  - Auto-boxing and unboxing
  - Sub-typing and variance of types
    - Covariant, contravariant and invariant
  - Wildcards
    - Upper-bounded and lower-bounded
- Java Collections Framework
  - `Collection` / `List` interfaces
  - `Comparator` interface
- Type erasure

Creating a Generic Box

- Create a `Box` to contain *any* type of object
  - a `static of(..)` method that takes an object as argument and puts it into the box
  - a `get()` method to return the object
- How to define a "generic" `Box` class to hold any object?
  - **abstraction principle**: avoid defining similar classes with duplicate code: `StringBox`, `PointBox`, ...
  - consider declaring the internal object with type `Object` since every other object is an `Object`?
- *Keep in mind this notion of a "box", as it will come in very handy when discussing certain FP concepts later...*

A “Generic” Box using `Object`

```
class Box {
    private final Object t;

    private Box(Object t) {
        this.t = t;
    }

    static Box of(Object t) {
        return new Box(t);
    }

    Object get() {
        return t;
    }

    @Override
    public String toString() {
        return "[" + t + "]";
    }
}
```

```
jshell> Box.of("chocolate")
$.. ==> [chocolate]

jshell> Box.of("chocolate").get()
$.. ==> "chocolate"

jshell> Box.of("chocolate").get().length()
|  Error:
|  cannot find symbol
|    symbol:   method length()
|  Box.of("chocolate").get().length()
|  ^--------------------------^
```

- Calling method `get()` on a `Box` returns an `Object` type
  - requires an explicitly cast to a “known” compile-time type

```
jshell> ((String) Box.of("chocolate").get()).length()
$.. ==> 9
```

Generic Type

"allows a type or method to operate on objects of various types while providing compile-time type safety"

- Since Java 5, generics has been added to eliminate the "drudgery of casting"
- Class/interface or method definitions can now include a type parameter section consisting of type parameters T_i enclosed with angle brackets (e.g. < $\mathsf{T}_1, \mathsf{T}_2, \mathsf{T}_3, \ldots$ >)
- Type parameters can then be used anywhere within the class/interface or method
- Generic typing is also known as **parametric polymorphism**
  - a means to replace type parameters with concrete types when needed

Generic Classes

- Declare the class `Box<T>` with a generic type parameter `T` to support parametric polymorphism, e.g. objects of type `Box<String>`, `Box<Object>`, ...

```
class Box<T> {
    private final T t;

    Box(T t) {
        this.t = t;
    }

    T get() {
        return t;
    }

    @Override
    public String toString() {
        return "[" + t + "]";
    }
}
```

```
jshell> Box<String> box = new Box<String>("abc")
box ==> [abc]

jshell> box.get()
$.. ==> "abc"

jshell> box.get().length()
$.. ==> 3
```

Generic Method

- Can also define generic methods with its own type parameter

```
jshell> <T> Box<T> of(T t) { return new Box<T>(t); }
|  created method of(T)

jshell> Box<String> box = of("abc")
box ==> [abc]
```

- The scope of type parameter T is within the method
- Use the same idea to define a static method in class Box<T>

```
class Box<T> {
    private final T t;
    ...
    static <T> Box<T> of(T t) { // can replace <T> with <U> also
        return new Box<>(t);
    }
```

  - declaration of two type parameters T: one within the class enclosing scope, the other within an inner method scope

Generic `Box` Class

```
class Box<T> {
    private final T t;

    private Box(T t) {
        this.t = t;
    }

    T get() {
        return t;
    }

    static <T> Box<T> of(T t) {
        return new Box<>(t);
    }

    @Override
    public String toString() {
        return "[" + t + "]";
    }
}
```

```
jshell> Box<String> box = Box.of("abc")
box ==> [abc]

jshell> box.get()
$.. ==> "abc"

jshell> box.get().length()
$.. ==> 3
```

Auto-boxing and Unboxing

- Only reference types are allowed as type arguments; primitives are auto-boxed/unboxed, e.g. in the case of `Box<Integer>`

```
jshell> Box<Integer> box = Box.of(123)
box ==> [123]

jshell> Integer x = Box.of(123).get()
x ==> 123

jshell> int y = Box.of(123).get()
y ==> 123
```

  - Placing an `int` value into `Box<Integer>` causes it to be **auto-boxed** into an `Integer` type
  - Getting an `Integer` value out of `Box<Integer>` and assigning to an `int` primitive type variable causes the `Integer` return value to be **(auto-)unboxed**

Variance of Types

- **LSP** re-defined: S is a **sub-type** of T (denoted `S <: T`) if a piece of code written for variables of type T can be used on variables of type S while ensuring type-safety
- Let `S <: T` with simple types `S`, `T` denoting classes/interfaces
  - **covariance**: when subtype relation is preserved in complex types, e.g. Java arrays are covariant `S[] <: T[]`

    ```
    Shape s = new Circle(1); Shape[] shapes = new Circle[10];
    ```
  - **contravariant**: when subtype relation is reversed for complex types *(more on this later..)*
  - **invariant**: when its neither covariant nor contravariant, e.g. Java generics are invariant

    ```
    Box<Shape> shapes = new Box<Circle>();   // error
    Box<Circle> circles = new Box<Circle>(); // ok
    ```

Wildcards

- Due to invariance in generics, the parameterized type can be inferred from the type declaration of the variable, i.e.

  `Box<Integer> box = new Box<Integer>(10);` can simply be

  `Box<Integer> box = new Box<>(10);`

- How do we then sub-type among generic types, in the spirit of

```
jshell> Box<Integer> box = Box.of(10)
box ==> [10]

jshell> Box<Object> anybox = box
|  Error:
|  incompatible types: Box<Integer> cannot be converted to Box<Object>
|  Box<Object> anyoox = box;
```

- Use the wildcard `?`, i.e. `Box<?> anyBox = new Box<Integer>();`
- `?` is not the `Object` type;
  - see what happens when we try: `new Box<?>()`

Bounded Wildcards

- Suppose we have the following classes:

```
jshell> class FastFood { }
|  created class FastFood

jshell> class Burger extends FastFood { }
|  created class Burger

jshell> class CheeseBurger extends Burger { }
|  created class CheeseBurger
```

- Define a method to return a `Burger` from a `Box<Burger>`

```
jshell> Burger getBurger(Box<Burger> box) { return box.get(); }
|  created method getBurger(Box<Burger>)

jshell> Box<Burger> box = Box.of(new Burger())
box ==> [Burger@6093dd95]

jshell> Burger burger = getBurger(box)
burger ==> Burger@6093dd95
```

Upper-Bounded Wildcards

- What other food can be boxed and passed to `getBurger`, `Box.of(new CheeseBurger())` or `Box.of(new FastFood())`?
  - the former since `CheeseBurger` is a `Burger`; however

```
jshell> Box<CheeseBurger> box = Box.of(new CheeseBurger())
box ==> [CheeseBurger@6093dd95]

jshell> getBurger(box)
|  Error:
|  incompatible types: Box<CheeseBurger> cannot be converted to Box<Burger>
|  getBurger(box)
|            ^-^
```

- Let `Burger` form an upper bound of the wildcard — change parameterized type of the argument to `<? extends Burger>`

```
jshell> Burger getBurger(Box<? extends Burger> box) { return box.get(); }
|  replaced method getBurger(Box<? extends Burger>)

jshell> Box<CheeseBurger> box = Box.of(new CheeseBurger())
box ==> [CheeseBurger@5622fdf]

jshell> Burger burger = getBurger(box)
burger ==> CheeseBurger@5622fdf
```

Putting Burgers into Empty Boxes

- Now modify the `Box` class to include the following:

```
class Box<T> {
    ...
    private boolean isEmpty() {
        return this.t == null;
    }

    static <T> Box<T> empty() {
        return new Box<>(null);
    }

    Box<T> add(T t) {
        return isEmpty() ? new Box<>(t) : this;
    }

    @Override
    public String toString() {
        return "[" + (isEmpty() ? "empty" : t) + "]";
    }
}
```

Putting Burgers into Empty Boxes

- Define a method to put a `Burger` into a `Burger<Box>`

```
jshell> Box<Burger> putBurger(Box<Burger> box) {
   ...> return box.add(new Burger());
   ...> }
|  created method putBurger(Box<Burger>)

jshell> Box<Burger> emptybox = Box.empty()
emptybox ==> [empty]

jshell> Box<Burger> fullbox = putBurger(emptybox)
fullbox ==> [Burger@53b32d7]
```

- What other types of box can a `Burger` be added into, `Box<CheeseBurger>` or `Box<FastFood>`?
  - the latter, since `Burger` is a `FastFood`; however

```
jshell> Box<FastFood> emptybox = Box.empty()
emptybox ==> [empty]

jshell> putBurger(emptybox)
|  Error:
|  incompatible types: Box<FastFood> cannot be converted to Box<Burger>
|  putBurger(emptybox)
|            ^------^
```

Lower-Bounded Wildcards

- `Burger` now forms a lower bound of the wildcard — change parameterized type of the argument to `<? super Burger>`

```
jshell> Box<? super Burger> putBurger(Box<? super Burger> box) {
   ...> return box.add(new Burger());
   ...> }
|  replaced method putBurger(Box<? super Burger>)

jshell> Box<FastFood> emptybox = Box.empty()
emptybox ==> [empty]

jshell> Box<? super Burger> fullbox = putBurger(emptybox)
fullbox ==> [Burger@39c0f4a]
```

- Notice that the return type is also `<? super Burger>` since a `Burger` can also be added to any box that holds food more general than a `Burger`

Covariance versus Contravariance

- Let the complex-type `C<T>` be a class/interface having a simple-type parameter `T`
- Covariance:
  - `? extends` is covariant since sub-type relation is **preserved** between simple and complex types
  - if `S <: T`, then `C<S> <: C<? extends T>`

    `CheeseBurger <: Burger` $\implies$ `Box<CheeseBurger> <: Box<? extends Burger>`
- Contravariance:
  - `? super` is contravariant since the sub-type relation is **reversed** between simple and complex types
  - if `S <: T`, then `C<T> <: C<? super S>`

    `Burger <: FastFood` $\implies$ `Box<FastFood> <: Box<? super Burger>`

Java Collections Framework

- The Java API provides **collections** to store related objects
- Interfaces specified as part of the Collections Famework declare common operations that can be performed *generically* on various type of collections

Interface	Description
Collection	The root interface in the collections hierarchy from which interfaces Set, Queue and List are derived.
Set	A collection that does not contain duplicates.
List	An ordered collection that can contain duplicate elements.
Map	A collection that associates keys to values and cannot contain duplicate keys.
Queue	Typically a first-in, first-out collection that models a waiting line; other orders can be specified.

Java Collections Framework

void	**add**(int index, **E** element)	Inserts the specified element at the specified position in this list.
boolean	**add**(**E** e)	Appends the specified element to the end of this list.
void	**clear**()	Removes all of the elements from this list.
boolean	**contains**(**Object** o)	Returns `true` if this list contains the specified element.
E	**get**(int index)	Returns the element at the specified position in this list.
int	**indexOf**(**Object** o)	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**isEmpty**()	Returns `true` if this list contains no elements.
E	**remove**(int index)	Removes the element at the specified position in this list.
boolean	**remove**(**Object** o)	Removes the first occurrence of the specified element from this list, if it is present.
E	**set**(int index, **E** element)	Replaces the element at the specified position in this list with the specified element.
int	**size**()	Returns the number of elements in this list.
void	**trimToSize**()	Trims the capacity of this `ArrayList` instance to be the list's current size.

- Examples of methods specified in interface `Collection<E>`
  - `size`, `isEmpty`, `contains`, `add(E)`, `remove(Object)`, `clear`
- Examples of methods specified in interface `List<E>`
  - `indexOf`, `get`, `set`, `add(int, E)`, `remove(int)`,

The `Collection<E>` Interface

- *Generic interface* parameterized with a type parameter `E`
- `toArray(T[])` is a generic method; the caller is responsible for passing the right type*
- `containsAll`, `removeAll`, and `retainAll` has parameter type `Collection<?>`, we can pass in a Collection of any reference type to check for equality
- `addAll` has parameter declared as `Collection <? extends E>`; we can only add elements that are upper-bounded by `E`

```
public interface Collection<E>
        extends Iterable<E> {
    boolean add(E e);

    boolean contains(Object o);

    boolean remove(Object o);

    void clear();

    boolean isEmpty();

    int size();

    Object[] toArray();

    <T> T[] toArray(T[] a);

    boolean addAll(Collection<? extends E> c);

    boolean containsAll(Collection<?> c);

    boolean removeAll(Collection<?> c);

    boolean retainAll(Collection<?> c);
    :
}
```

**otherwise, an `ArrayStoreException` will be thrown*

The `List<E>` Interface

- `List<E>` interface extends `Collection<E>`
  - for ordered collections of possibly duplicate objects
  - Classes that implement `List<E>` include `ArrayList` and `LinkedList`, e.g.

```
List<Circle> circles = new ArrayList<>();
```

    - **covariance**: `S <: T` $\Longrightarrow$ `S<E> <: T<E>`

- `List<E>` specifies a sort method with a default implementation

```
default void sort(Comparator<? super E> c)
```

- `sort` method takes as argument an object with a generic interface `Comparator<? super E>` (*why super?*)

The `Comparator<T>` Interface

- An implementation of the `Comparator<T>` interface is required to define the `int compare(T o1, o2)` method
  - `compare(o1, o2)` returns 0 if the two elements are equal, < 0 if `o1` is "less than" `o2`, or > 0 otherwise

```
import java.util.Comparator;

class NumberComparator implements Comparator<Integer> {
    @Override
    public int compare(Integer s1, Integer s2) {
        return s1 - s2;
    }
}
```

```
jshell> List<Integer> nums = Arrays.asList(3, 1, 2);
nums ==> [3, 1, 2]

jshell> nums.sort(new NumberComparator())

jshell> nums
nums ==> [1, 2, 3]
```

Type Erasure

- Compiler performs type checking/inference, and generates non-generic bytecode (erasure) for backward compatibility
  - type parameters are replaced with either `Object` if it is unbounded, or the bound if it is bounded

```
class Box { // bytecode generated from this "type-erased" version
    private final Object t;
    private Box(Object t) { this.t = t; }
    Object get() { return t; }
    static Box of(Object t) { return new Box(t); }
```

  - type arguments are erased during compile time; generics allow the creation of raw types (should be avoided)

```
jshell> Box raw = Box.of("abc")  // possible, but bad practice
raw ==> [abc]

jshell> Box<Integer> box = raw   // erasure becomes: Box box = raw
|  Warning: unchecked conversion // warning ensues but still compilable
|    required: Box<java.lang.Integer>
|    found:    Box
box ==> [abc]                    // still runs
```

Lecture Summary

- Appreciate the use of Java generics in classes and methods
- Understand autoboxing and unboxing involving primitives and its wrapper classes
- Understand parametric polymorphism and the sub-typing mechanism, e.g. given `Burger <: FastFood`
  - covariant: `Burger[] <: FastFood[]`
  - covariant: `ArrayList<Burger> <: List<Burger>`
  - covariant: `Box<Burger> <: Box <? extends FastFood>`
  - contravariant: `Box<FastFood> <: Box<? super Burger>`
  - invariant: Neither `Box<Burger> <: Box<FastFood>` nor `Box<FastFood> <: Box<Burger>`
- Familiarity with usage of the Java Collections Framework