ESS 201 Programming II Java
Term 1, 2019-20

Generics & Collections

T K Srikanth
International Institute of Information Technology, Bangalore

Generic Classes and Methods

Mechanism for implementing classes/methods that can work on different classes, but still provide **compile-time** type safety.

E.g. can we implement a generic method printArray that can iterate through and print an array with any specific type of element

Instead of

```
    static void printArray(Integer[] ints)
    static void printArray(Book[] books)
    static void printArray(String[] strings)
```

Can we have a single implementation of printArray where can
pass in any of Integers, Books, Strings…. Another example of **re-use**!

Other examples

Implement methods to compute the sum (or average) of an array of ints or an array of floats or an array of shorts, but be strongly typed.

Generic print array - example

```
public static <T> void printArray(T[] arr) {
    for(T elem: arr) {
        System.out.println(elem);
    }
    System.out.println();
}
```

Works on any class T for which toString is defined - i.e. any sub-class of Object!

Note: generic methods can only be defined for non-primitive types. For primitives, use wrapper classes

Generics

We pass in type parameters to the class/method, similar to passing in arguments to methods. Allows re-use.

- Stronger type checking
- Elimination of casts

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

```
List<String> list = new
ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

- Implement algorithms such as sort in a generic manner

Bounding the type parameters

What if the generic method implementation uses methods of a certain type - e.g. a method that can compute the average of an ArrayList<Double> or ArrayList<Integer> - in general ArrayList<Number>

I.e. we want a method:

```
static double average(ArrayList<Number> nums) {// assume nums non-zero length
        double sum = 0.0;
        for(number n: nums) {
                sum += n;
        }
        return sum/nums.size();
}
```

Bounding type parameters

What happens when we call this with a list that is of type

ArrayList<Integer> or ArrayList<Double>? Why?

ArrayList<Integer> is not a sub-class of ArrayList<Number>. Why?

Can we pass in an ArrayList<Book>?

Hence, need a way to say that we can pass in any arraylist, so long as the elements are of any sub-type of Number.

```
    static double average(ArrayList<? extends Number> nums){ … }
```

Called a wildcard type-parameter. Can also use:

```
    static <T extends Number> double average(ArrayList<T> nums) { … }
```

Generic Classes

We can implement classes that can have flexibility in the type of objects they handle. ArrayList is an example of this - you can have an ArrayList of any type of elements, and be able to apply its methods consistently.

Another example:

A Stack class: to enhance type safety, we would need variants of the Stack that manage data of specific types:

Stack of Integers: can push only Integers, and pop should return Integer

Stack of Cars: can push only Cars and pop should return Cars

Example: Stack - class or interface

```
public class Stack<T> {

    public void push(T item) { … }

    public T pop() { … }

    public boolean isEmpty() { … }

}
```

```
public interface Stack<T> {

    public void push(T item);

    public T pop();

    public boolean isEmpty();

}
```

Generic Classes

Consider a class Point in 2D. Depending on the context, the coordinates could be in float or integer units (e.g. a continuous space or a pixel-based screen). Yet, most of the operations we perform on these would be “generic” in nature:

- Distance, closest point of a list of points to a given point, etc.

Can we implement this once and re-use it for both scenarios - float and int coordinate spaces?

Generic Classes

```
public class Point<T> {
        Point(T x, T y) { … }
        public static Double dist(Point<T> p2) {  … }
        public Point<T> closest(ArrayList<Point<T>> points) { … }
        …
        private T x, y;
}
```

And use this as

```
Point<Integer> pi = new Point<Integer>(3,4);
Point<Double> pd = new Point<Double>(3.0, 4.3);
```

Note: to be safe, we should strictly define this as

```
        public class Point<T extends Number> { …. }
```

Specifying type argument

Given a class Point<T>,

```
Point<Integer> pi = new Point<Integer>(3,4);
Point<Double> pd = new Point<Double>(3.0, 4.3);
```

We can also use:

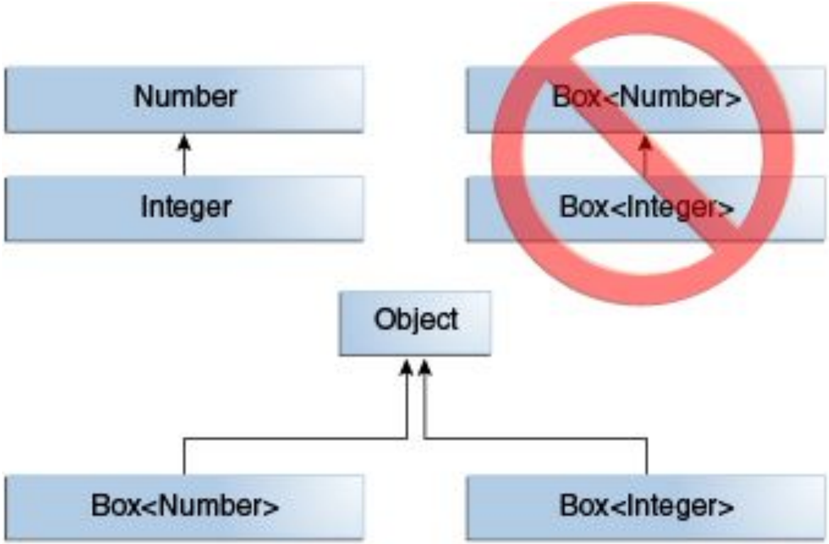
```
Point<Integer> pi = new Point<>(3,4);
Point<Double> pd = new Point<>(3.0, 4.3);
```

Compiler uses *type inference* to decide what type should be passed in when instantiating the Point object (of the right type)

Generics: Inheritance and sub-types

If we have a generic class Box<T>

Is Box<Integer> a sub-type of Box<Number>?



But, if we define NewBox<T> extends Box<T>, then

NewBox<Integer> is a sub-type of Box<Integer>