Homework4

4.1
Why cannot type argument be of primitive type?

- In Java the type of any variable is either a primitive type or a reference type. Generic type arguments must be reference types. Since primitives do not extend Object they cannot be used as generic type arguments for a parametrized type.
- Type erasure is a process in which compiler replaces a generic parameter with actual class or bridge method. Because primitive type is not an object, which cannot be replaced during the process.

<T extends E>
That declaration means T can be any type that is subclass of ABC.

<? extends E>
some type which is a subclass of E
? could be E and any subclass of E

<? super E>
some type which is an ancestor (superclass) of E

4.2
Stream API:

1. **map:** The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.

```
List number = Arrays.asList(2,3,4,5);
List square = number.stream().map(x->x*x).collect(Collectors.toList());
```

2. flatmap: Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided

mapping function to each element.

map() vs. flatmap

The difference is that the `map` operation produces one output value for each input value, whereas the `flatMap` operation produces an arbitrary number (zero or more) values for each input value.

3. distinct
    - This method uses *hashCode()* and *equals()* methods to get distinct elements.
    - It's useful in removing duplicate elements from the collection.

4. limit

    restrict the number of stream elements, which take first N elements by limit(N)

    ```
    list.stream().list(N).collect(Collectors.toList());
    ```

5. skip

    an intermediate operation that discards the first *n* elements of a stream.

    ```
    Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
        .filter(i -> i % 2 == 0)
        .skip(2)
        .forEach(i -> System.out.print(i + " "));
    //result: 6 8 10
    ```

6. findFirst

    returns an *Optional* for the first entry in the stream; the *Optional* can, of course, be empty.

7. toArray - terminal operation

If we need to get an array out of the stream, we can simply use *toArray().*

```
Employee[] employees = empList.stream().toArray(Employee[]::new);
```

8. min and max

   return the minimum and maximum element in the stream respectively, based on a comparator.

```
Employee firstEmp = empList.stream()
      .min((e1, e2) -> e1.getId() - e2.getId())
      .orElseThrow(NoSuchElementException::new);
```

9. peek

Perform multiple operations on each element of the stream before any terminal operation is applied.

```
empList.stream()
      .peek(e -> e.salaryIncrement(10.0))
      .peek(System.out::println)
      .collect(Collectors.toList());
```

10. sorted

this sorts the stream elements based on the comparator passed we pass into it.

```
List<Employee> employees = empList.stream()
      .sorted((e1, e2) -> e1.getName().compareTo(e2.getName()))
      .collect(Collectors.toList());
```

Method reference

Method references are a special type of lambda expressions. They're often used to create simple lambda expressions by referencing existing methods.

There are four kinds of method references:

- Static methods
- Instance methods of particular objects
- Instance methods of an arbitrary object of a particular type
- Constructor

Syntax

- *ContainingClass*::*MethodName*

```
List<Integer> numbers = Arrays.asList(5, 3, 50, 24, 40, 2, 9, 18);
// normal
numbers.stream()
  .sorted((a, b) -> a.compareTo(b));
// method reference
numbers.stream()
  .sorted(Integer::compareTo);
```