3-generics-and-collections

Contents

Basic design **2**
Generics . . . 2
Syntax . . . 3
Raw-types . . . 4
Methods . . . 5
Subtypes . . . 6
Wildcards . . . 6
Limitations . . . 7
Summary . . . 8
Collections . . . 8
Core . . . 8
Concrete . . . 9
ArrayList . . . 9
Set . . . 11
Map . . . 11
Deque . . . 12
Comparable . . . 13
Comparator . . . 13
Stream . . . 14
Primitive streams . . . 14
Creating streams . . . 15
Building streams . . . 15
Existing interface . . . 16
Intermediate operations . . . 16
Terminal operations . . . 17

- Basic design
  - Generics
    * Syntax
    * Raw-types
    * Methods
    * Subtypes
    * Wildcards
    * Limitations
    * Summary
  - Collections
    * Core
    * Concrete
    * ArrayList

    - Set
    - Map
    - Deque
    - Comparable
    - Comparator
  - Stream
    - Primitive streams
    - Creating streams
    - Building streams
    - Existing interface
    - Intermediate operations
    - Terminal operations

Basic design

Every non trivial Java application makes use of data structures and algorithms. The Java collections framework provides a large set of readily usable general purpose data structures and algorithms. These data structures and algorithms can be used with any suitable data type in a type safe manner; this is achieved through the use of a language feature called generics

The generics feature in java can be very familiar to other similar concepts from other languages such as c++, however while this feature is similar in nature it is quite different in the way it is implemented in the Java Language in comparison

Generics

The general syntax of defining generics and generic types is to wrap around any non primitive type in angle brackets such as `<T>`. The `T` name is just a placeholder which the compiler uses during compile time to replace with the actual type declared by the user when a generic type is instantiated, it is matched against the actual definition of the type, mostly to make sure that type adheres to the API and generic rules (e.g. T extends K, T super K), since the entire generic feature in Java works in compile time, once the class is compiled by the compiler the actual generic type information is erased away. This is different than how some other languages implement generics, however this was done to keep backwards compatibility with older versions of the language and avoid introducing breaking changes at such a fundamental level.

```
// this is how the declaration of a generic type looks like, note that the `T
   ` type is simply a placeholder, which is
// replaced during compiler time by the compiler, if there are any
   restrictions on the place holder the compiler makes sure
// to validate them as well i.e one can define the placeholder such that T
   extends OtherType, this rule is enforced by the
// compiler.
class BoxPrinter<T> {
    private T val;

    public BoxPrinter(T arg) {
        val = arg;
    }

    public String toString() {
        return "[" + val + "]";
```

```
    }
}

// create an instance which stores a string, the placeholder is resolved
   during compile time, by the compiler, and
// matched against the type, in this case String
BoxPrinter<String> box = new BoxPrinter<String>("test");

// note that auto-boxing will work with generics, the compiler will correctly
    convert this from a primitive type to
// Integer, before creating the instance, therefore this is allowed and is
   not a compiler error
BoxPrinter<Integer> box2 = new BoxPrinter<Integer>(1);
```

The `<T>` is simply a placeholder, which is used in the declaration and then in the definition of the class type. The example above shows the diamond syntax. As mentioned generics are happening at two stages in the java language - first is declaration, when the generic type is declared, meaning the `signature` of the type is created, the user specified what the generic type / holder class is, the second is the actual creation of the generic type, or in other words an instance of a generic type, where the diamond syntax `<>` specifies the concrete type that the generic class definition will be created with. In a way it works very similarly to how templates work in C++, where the templates are also resolved during compile time, in C++, one can think of it in the following way, the compiler takes in the template and does a sort of `find and replace` action on the generic type signature, to create a new type, where the generic type is instantiated, for each concrete type.

Imagine that in the example presented above, two versions of `BoxPrinter` are created, one that stores an `Integer` and one that stores `String` in the val member variable. For more details see example blow

```
class BoxPrinterString {
    private String val;
    // the rest of the class members
}

class BoxPrinterInteger {
    private Integer val;
    // the rest of the class members
}
```

This is not exactly how generics work, in Java, but fundamentally this is what the compiler does, it creates multiple unique versions of the type, for each unique generic argument, no matter how many generic arguments a given type declares

Syntax

As already explained the diamond syntax is used to both declare a generic type, and define it in code. It is possible to skip the type in the creation of the instance, to make tying less verbose.

```
// those two are functionally equivalent, the second instantiation simply
   skips the contents inside of the brackets on
// the constructor, but they are inferred from the type, the left hand side
BoxPrinter<Integer> box1 = new BoxPrinter<Integer>(1);
BoxPrinter<Integer> box2 = new BoxPrinter<>(1);
```

```
// this however is incorrect, the compiler will spill out an warning, since
   this is trying to declare a non generic
// instance of `BoxPrinter`, since the explicit diamond pattern is not
   specified
BoxPrinter box3 = new BoxPrinter<Integer>(1);

// this is functionally and fundamentally the same as the one above, this is
   not an error, because the language is
// trying to be backwards compatible with older versions, the compiler will
   simply try to replace the type argument with
// Object, when it is omitted
BoxPrinter box4 = new BoxPrinter(1);

// this is also functionally equivalent to the above, however in this case
   the placeholder types are known, the compiler
// will simply complain that there is unchecked cast between the left and
   right expression, since the left expression
// declares the generic types, however the right one produces BoxPrinter<
   Object> effectively
BoxPrinter<Integer> box5 = new BoxPrinter(1);

// this is straight up wrong, the generic placeholder can be inferred only
   from the left hand side of the expression,
// not the right hand, that would produce a compile time error
BoxPrinter<> box5 = new BoxPrinter<Integer>(1);
```

Raw-types

As already mentioned, to retain backwards compatibility certain syntaxes are allowed when defining generics, however the compiler will infer the raw type by default, in this case that is `Object` the top level type from which all types in Java extend off of. This is fine, and the compilation process might complete fine, with warnings, however one looses all benefits of compile time checks when the actual generic type is not known, or is not specified.

```
// while this is possible, all methods which return/work on an element of the
    array would by default return Object,
// which is not very useful, since users loose all types of compile time type
    checking capabilities
List list = new ArrayList();
```

```
List list = new LinkedList();
list.add("First");
list.add("Second");

List<String> strList = list; // #0: generates a compiler warning
strList.add(10); // #1: generates compiler error

for(Iterator<String> itemItr = strList.iterator(); itemItr.hasNext();) {
    System.out.println("Item : " + itemItr.next());
}

List<String> strList2 = new LinkedList<>();
```

```
strList2.add("First");
strList2.add("Second");

List list2 = strList2;
list2.add(10); // #2: compiles fine, results in a runtime exception

for(Iterator<String> itemItr = list2.iterator(); itemItr.hasNext();) {
    System.out.println("Item : " + itemItr.next());
}
```

Using raw types is not recommended for the various reasons displayed above, the worse case is that the code throws at run time due to a class cast exception, there are niche situations where the use of raw types might be necessary, but in general it is never a good idea.

Methods

The way one can define generic types, generic methods are also possible, they work the same way as generic types, however, it depends on the way the generic method is defined, if a method is defined in a generic type, and if the method itself is generic, those are two different constructs. The first one does not define a generic method, the generic method is a side effect of the type being generic, while the second one defines a completely different language construct which is a generic method. Generic methods can be either static or normal non-static ones.

```
public final class ClassName {

    public static <T> void fill(List<T> list, T val) {
        for(int i = 0; i < list.size(); i++)
        list.set(i, val);
    }

    public static void main(String[] args) {
        List<Integer> intList = new ArrayList<Integer>();
        intList.add(10);
        intList.add(20);

        // both calls are semantically identical, however it is worth noting
           that if the list was of raw Object type,
        // one could still provide a stronger type checking by enforcing the
           type in the method call itself, by using the
        // class name as prefix to call the static method, or if it was an
           instance method use the instance name (variable)
        // in place of the class name shown in the example below
        fill(intList, 100);

        // even though this will print all types of warnings, this approach
           can be used to make sure that a generic type
        // that is passed in as argument is treated accordingly, if the
           elements inside the generic are of known type
        List rawList = intList;
        LitteralsAndTypes.<Integer>fill(rawList, 100);
    }
```

```
}
```

The method type is generic, however the type is inferred from the passed in types, by the compiler, it is not required to specify the type of the arguments, also as shown above, if the argument is of raw type, it is possible to provide the actual type of the generic by specially invoking the method.

Subtypes

While it is possible to assign a derived type object to its base type reference. However, for generics the type parameters should match exactly, otherwise a compiler error is guaranteed, In other words subtyping does not work for generic parameters, they have to match exactly.

`Subtyping works for class types, you can assign a derived type object ot its base type reference, however subtyping does not work for generics type parameters - one can not assign a dervied generic type parameter to a base type parameter`

```
// this is not a valid generic declaration, while Number is indeed a valid
   super type for Integer, that is not valid
// when creating generic types, will spill out a compiler error
List<Number> intList = new ArrayList<Integer>();
```

So why is this wrong, here is the reason - the right hand side of the expression creates an array list, internally a dynamic array, which holds Integers, however, the left hand side assings to a less strict type, in this case Number, now imagine if this had worked, then that `intList` (take a note at the name of the variable) can be used in a context where one could simply insert any type of class which extends off of Number, say Float, and the moment one calls `intList.add(new Float(5));` that will produce a class cast exception during run time, since the actual intList points to an `ArrayList` instance which holds only Integers.

Wildcards

To overcome issue above, wild card type parameters are used, they do not specify the exact type in the left hand side expression.

```
// note that wild cards can be used only in the left hand side expression,
   one can not instantiate a generic type of
// wild card, since that is meaningless, the right hand side of the
   expression has to provide a concrete instance of the
// generic with exact type or if non is provided then the raw type is used by
    default - meaning Object
List<?> wildCardList = new ArrayList<Integer>();
```

Wild cards have different semantic meaning, they are not a replacement for the `raw` type, or `Object`, the wild card does NOT tell the compiler that `List` holds different types of elements (heterogeneous), what the wild card tells the compiler is that that variable references a generic `List` type of homogeneous elements that can hold any type of argument - `List<Integer>, List<String>, List<Float> etc`, meaning the underlying generic List instance is always containing/referencing the same type of elements

`Wildcards are not replacement for Object or Raw types, they have very special meaning, they imply that a variable is referencing a generic of an unknown type, but still the instance is of generic class and of a specific homogeneous type`

```
List<?> wildCardList = new ArrayList<Integer>();
wildCardList.add(1); // Compile error - cannot add to List<?>
wildCardList.add(1); // Compile error - cannot add to List<?>
```

The above would fail because the wild card is unbound, why is that a problem. When one uses wildcard type, the code implies to the compiler that the type information is ignored, so <?> stands for unknown type. Every time one tries to apss arguments to a generic type, the java compiler tries to infer the type of the passed in argument as well, as the type of the generics and to justify the type safety. Now calling add method to insert an element in the list, is not valid, since the variable references a list that holds unknown types (specified by the wild card). The compiler does not know which type to infer. That is why it just errors out. If it did not, one might end up adding String, to the list, which was already established will fail at run-time the compiler tries to be pro-active and warn about issues as early as possible, since generics were added to the language to ensure type safety, replacing the old approach of using Object to achieve the same, not forego it.

`In general when one provides a wildcard parameters, methods can not be called that modify the object, there are certain methods which can still be called that access the object's state, for the example with the list above methods like .get().size().empty()and so on, are still valid and can be called safely`

Limitations

There are many limitations of generic types due to type erasure. A few important ones are as follows:

- You cannot instantiate a generic type using a new operator. - this is because the compiler does not know what the default constructor for the type T is at the moment of the generic's definition, that is only known when an actual usage of the generic type is in play i.e when a generic type is instantiated. Remember the type replacement happens at compile time, not at run-time, meaning that after compilation, the resulting byte code will have the type erased, and there is no way to call a constructor of a type erased type. If that were allowed at run-time the expression would be literally - `new Object()`, which is not valid

```
T mem = new T(); // compiler error
```

- You cannot instantiate an array of a generic type. - Remember the type replacement happens at compile time, not at run-time, meaning that after compilation, the resulting byte code will have the type erased, and there is no way to call the array constructor for erased type, of unknown type, what will end up happening if that were allowed is basically this - `new Object[100]`

```
T[] amem = new T[100]; // compiler error
```

- You can declare instance fields of type T, but not of static fields of type T. This is due to the fact that static members are bound to the `ClassType` itself, however generics are bound to a class type instance, there is no notion of an instance for the generic class type definition itself. For example

```
class Clazz<T> {
    static T staticMember; // compiler error
}
```

- It is not possible to have generic exceptions - the reason is pretty simple, the catch statements are evaluated during run-time, there is no way to specify a catch expression that would be able to capture different types of the generic exception, i.e `GenericException<String>, GenericException<Number>, GenericException<Float>`

```
class GenericException<T> extends Throwable {}
```

- Generics can not be instantiated from primitive types - i.e. `List<int>, List<short> etc`, is not valid, and would produce a compile time error

Summary

Implementation of generics is static in nature, which means that the Java compiler interprets the generics specified in the source code and replaces the generic code with concrete types. This is referred to as type erasure. After compilation, the code looks similar to what a developer would have written with concrete types. Essentially, the use of generics offers two advantages: first, it introduces an abstraction, which enables you to write generic implementation; second, it allows you to write generic implementation with type safety.

Collections

In Java 8, the collections framework was greatly overhauled to include the usage of the new Generics feature, and that allows it to provide a big reusable set of collection types which are very robust and flexible

Core

The core of the collections framework is revolving around a handful of interfaces and abstract classes, which are the base blocks of the entire collections framework

Class / Interface	Description
Iterable	A class implementing this interface can be used for iterating with a foreach statement.
Collection	Common base interface for classes in the collection hierarchy. When you want to write methods that are very general, you can pass the Collection interface. For example, max() method in java.util.Collections takes a Collection and returns an object.
List	Base interface for containers that store a sequence of elements. You can access the elements using an index, and retrieve the same element later (so that it maintains the insertion order). You can store duplicate elements in a List.
Set, SortedSet, NavigableSet	Interfaces for containers that don't allow duplicate elements. SortedSet maintains the set elements in a sorted order. NavigableSet allows searching the set for the closest matches.
Queue, Deque	Queue is a base interface for containers that holds a sequence of elements for processing. For example, the classes implementing Queue can be LIFO (last in, first out-as in stack data structure) or FIFO (first in, first out-as in queue data structure). In a Deque you can insert or remove elements from both the ends.
Map, SortedMap, NavigableMap	Interfaces for containers that map keys to values. In SortedMap, the keys are in a sorted order. A NavigableMap allows you to search and return the closest match for given search criteria. Note that Map hierarchy does not extend the Collection interface.
Iterator, ListIterator	You can traverse over the container in the forward direction if a class implements the Iterator interface. You can traverse in both forward and reverse directions if a class implements the ListIterator interface.

The collection interface itself, being the base for all other collections provides a minimal set of behavior and actions expressed in the methods below

Method	Description
boolean addAll(Collection<? extends | Element> coll)	Adds all the elements in coll into the underlying container.
boolean containsAll(Collection<?> coll)	Checks if all elements given in coll are present in the underlying container.
boolean removeAll(Collection<?> coll)	Removes all elements from the underlying container that are also present in coll.
boolean retainAll(Collection<?> coll)	Retains elements in the underlying container only if they are also present in coll; it removes all other elements.

Concrete

Several of the most well known and used concrete classes from the collections framework are listed in the table below, there are certainly more but these reflect the most commonly known data structures in programming

Class	Description
ArrayList	Internally implemented as a resizable array. This is one of the most widely used concrete classes. Fast to search, but slow to insert or delete. Allows duplicates.
LinkedList	Internally implements a doubly linked list data structure. Fast to insert or delete elements, but slow for searching elements. Additionally, LinkedList can be used when you need a stack (LIFO) or queue (FIFO) data structure. Allows duplicates.
HashSet	Internally implemented as a hash-table data structure. Used for storing a set of elements-it does not allow storing duplicate elements. Fast for searching and retrieving elements. It does not maintain any order for stored elements.
TreeSet	Internally implements a red-black tree data structure. Like HashSet, TreeSet does not allow storing duplicates. However, unlike HashSet, it stores the elements in a sorted order. It uses a tree data structure to decide where to store or search the elements, and the position is decided by the sorting order.
HashMap	Internally implemented as a hash-table data structure. Stores key and value pairs. Uses hashing for finding a place to search or store a pair. Searching or inserting is very fast. It does not store the elements in any order.
TreeMap	Internally implemented using a red-black tree data structure. Unlike HashMap, TreeMap stores the elements in a sorted order. It uses a tree data structure to decide where to store or search for keys, and the position is decided by the sorting order.
PriorityQueue	Internally implemented using heap data structure. A PriorityQueue is for retrieving elements based on priority. Irrespective of the order in which you insert, when you remove the elements, the highest priority element will be retrieved first.

ArrayList

Used to store a sequence of elements. `ArrayList` specifically implements a `resizable` array. When creating a native array (i.e new String[10];) the size of the array is known and fixed, at the time of creation. However `ArrayList` is a dynamic array, it can grow in size as required. Internally an `ArrayList` allocates a block of memory and grows as required, So accessing array elements is very fast in `ArrayList`. However, when you add or remove elements internally the rest of the elements are copied, or shifted, so addition and deletion of elements are costly operations.

```
// create a simple list, which is populated with random entries, initially
   the array list is created without any
```

```
// arguments to the constructor, internally the implementation however does
   not create an empty array, since that would be
// inefficient, when specified without capacity the array that is created is
   - 10, unless specified otherwise
ArrayList<String> languageList = new ArrayList<>();
languageList.add("C");
languageList.add("C++");
languageList.add("Java");

// the basic default for-each structure in java, this is a compile time
   structure, which gets compiled to an iterator,
// as shown below, it is usable for all types of collections which implement
   the iterator pattern and iterable interface
for(String language : languageList) {
    System.out.println(language);
}

// the iterator approach can also be used, this is equivalent to the for each
    above, however it is manually controlling
// the iteration process, meaning that one can use the call to next to skip
   over more than one element, stop the iteration
// at specific iterator point, or even use the .remove method from iterator
   to remove elements from the collection
for(Iterator<String> languageIter = languageList.iterator(); languageIter.
   hasNext();) {
    String language = languageIter.next();
    System.out.println(language);
}

// this example shows how one can remove from the array while iterating using
    the iterator, note that, this creates a
// mutable instance of a specific ArrayList iterator, each call to next, will
    mutate the iterator instance in place, making
// it point to the next element in the array, internally.
Iterator<String> language = language.iterator();

// when creating an iterator, the only way to obtain reference to the
   elements in the array is to call the next method,
// but before calling next one has to ensure that there is actually a `next`
   element, that is why the pattern usually is a
// call to hasNext followed by a call to next
while(language.hasNext()) {
    // next has to be called first before the remove method is called, this
       is specified in the API documentation of the
    // remove method in iterator, otherwise the call to remove will throw an
       exception, every remove call has to be
    // preceeded with next, the call to next moves to the next element first,
        mutates the current instance of the iterator,
    // to point to the next element, then remove is called
    language.next();
```

```
    // calling remove only after ensuring one has called the next method
       first, this is mandatory as mentioned in the
    // docs of the iterator interface, otherwise undefined behavior will
       occur, exception as well
    language.remove();
}
```

Set

Sets are collections that contain no duplicates, Unlike List a Set does not remember where you inserted the element, it is not ordered,. There are two important concrete classes for Set - `HashSet` and `TreeSet`, A `HashSet` is for quickly inserting and retrieving elements, it does not maintain any sorting order for the elements it holds, A `TreeSet`, stores the elements in a sorted order, according to the compare function or `compareTo` from the Comparable interface

The example below shows how a `pangram`, can be destructed into its constituent elements which are all the characters of the alphabet and then put into `TreeSet`, and printed out in a sorted order, since `TreeSet` will sort it's elements

```
String pangram = "the quick brown fox jumps over the lazy dog";
Set<Character> aToZee = new TreeSet<Character>();
for(char gram : pangram.toCharArray()) {
    aToZee.add(gram);
}
System.out.println("The pangram is: " + pangram);
System.out.print("Sorted pangram characters are: " + aToZee);
```

A pangram is a sentence that uses all letters in the alphabet at least once. You want to store characters of a pangram in a set. Since you need to use reference types for containers, one way to do this is to create `TreeSet` of Characters.

To obtain the Characters or chars of a String type in Java one can use the `toCharArray`, which returns a char[], which is the primitive version of the boxed type Character, the auto-boxing will occur above, and one can safely add Character elements to the `TreeSet`.

Map

A map stores key and value pairs. The Map interface does not extend the Collection interface. However there are methods in the Map interface that you can use to get the objects classes that implement Collection interface to work around this problem. Also the method names in Map are very similar to the method names in Collection, so it is easy to understand and use Map. There are two important concrete classes of Map - `HashMap` and `TreeMap`.

- A `HashMap` - uses a hash table data structure internally. In HashMap searching is a fast operation. However HashMap neither remembers the order in which the elements were inserted nor does it keep elements in any sorted order.
- A `TreeMap` - uses a red black tree data structure internally. Unlike the HashMap, TreeMap keeps the elements in sorted order, not in the order of insertion, but in the order defined based on the values. Searching and inserting is somewhat slower operation that in the HashMap.

The `NavigableMap` interface extends the `SortedMap` interface. The `TreeMap` class is the widely used clss that implements `NavigableMap`. As the name suggests with `NavigableMap`, one can navigate the Map easily. It

has mnany methods that make Map navigation easy. You can get the nearest value matching a given key, all values less than the given key, all values greater than the given key and so on. Let us look at an example.

```
NavigableMap<Integer, String> examScores = new TreeMap<Integer, String>();
examScores.put(90, "Sophia");
examScores.put(20, "Isabella");
examScores.put(10, "Emma");
examScores.put(50, "Olivea");

// this here would print the map as it is, containing the keys, in sorted
   order, since it is TreeMap and not a HashMap
// it is aware of and knowing how to sort the elements
System.out.println("The data in the map is: " + examScores);

// the call to `descendingMap`, will produce a new map with re-ordered
   elements in a descending order, in this case the
// higher valued keys would be sorted first, the lower one later on
System.out.println("The data descending order is: " + examScores.
   descendingMap());

// the call to tail, will generate a map of all keys and entries in general
   that are at the tail of the map greater than
// 40, remember the `examScores` are stored in ascending order, therefore the
    higher values are towards the end or tail of
// the map, to achieve the reverse, use headMap(40), which would generate a
   new map with all scores lower than 40, from
// the head toward the end of the map
System.out.println("Details of those who passed the exam: " + examScores.
   tailMap(40));

// the call to first, will extract the first entry in this case the one with
   the lowest score, since the map as already
// mentioned is sorted in ascending order
System.out.println("The lowest mark is: " + examScores.firstEntry());
```

Deque

`Deque` or Deck, is a doubly linked queue, that is a data structure that allows you to insert and remove elements from both ends, The `Deque` interface was introduced in java 6, The deue interface extends the Queue interface. Hence all methods provided by Queue are also available in the `Deque` interface. There are three concrete implementations of the `Deque` interface: `LinkedList, ArrayDeque and LihnkedBlockingDeque`.

The Deque interface provides multiple methods to operate on both ends of the queue, such as `addFirst` , `addLast`, `removeFirst`, `removeLast` and so on, which are pretty descriptive based on the name of the method.

```
class SpecialQueue {
    private Deque<String> queue = new ArrayDeque<>();

    void addInQueue(String customer){
        queue.addLast(customer);
    }
```

```
    void removeFront(){
        queue.removeFirst();
    }
    void removeBack(){
        queue.removeLast();
    }
}
```

Comparable

The Comparable and its companion interface Comparator interfaces are meant to provide means of comparing objects. These two interfaces are very important to the general internals of the Collections framework in Java. A lot of containers in the library rely on elements being either comparable, or a special lambda expression comparator to be provided when creating the container to ensure that elements can be compared when inserted/removed/searched.

The Comparable interface provides one single interface method, however this is not a `@FunctionalInterface` it is meant for types to implement it to compare with other types. Note that the method returns an integer value, this is because there are really three states in which a result of the comparison between two objects exists. Either the left $>$ right, left $==$ right, left $<$ right. Which corresponds to positive, zero and negative result.

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

```
class Student implements Comparable<Student> {
    String id;
    String name;
    Double cgpa;

    public Student(String studentId, String studentName, double studentCGPA)
       {
        id = studentId;
        name = studentName;
        cgpa = studentCGPA;
    }
    // this simple example compares only the id of the student, the
       comparison conditions vary greatly between the type
    // of object, and are usually some sort of business rules, which are not
       governed by the language but by real world
    // rules. That is why the need for interfaces like Comparable.
    public int compareTo(Student that) {
        return this.id.compareTo(that.id);
    }
}
```

Comparator

The companion interface to Comparable, which is usually meant to provide a one-shot comparison operations based on some special criteria, for example given different requirements, one might wish to sort the Student type based on different criteria. Want to sort them alphabetically - use the name, based on score - use the `gpa`

score, and so on. Those can not be expressed with a single `compoareTo` method, that is why `comparator` exists which is a `FunctionalInterface`, which can be used to create various criteria based on which to compare

```
class CGPAComparator implements Comparator<Student> {
    public int compare(Student s1, Student s2) {
        return (s1.cgpa.compareTo(s2.cgpa));
    }
}
List<Student> students = new ArrayList<>();
// create and populate an array of students
Arrays.sort(students, new CGPAComparator());
```

`The general rule of thumb is that most real world cases have a natural order to them, meaning that the Comparable interface can be used, in exceptional cases the Compartor interface can be used as a substitute, to make certain customized sorting or ordering decisions.`

Stream

The Stream interface is among the most important interfaces provided in the Java 8 release. The classes `Double|Long|IntStream` are specialization for the int,long and double primitive types.

The streams in java operate on what is called the `stream pipeline`. The stream pipeline is a set of operations which are chained on a stream object, and are used to generate the final transformation and result for a Stream instance.

The pipeline is represented by methods which can be called on an instance of a stream. Calls to each action, part of the pipeline, mutates the source stream instance/object, adding transformations to the pipeline, however the underlying actions are not performed until special terminal or termination operations are invoked on the stream instance, only then the actual pipeline springs into motion executing all actions previously specified on that stream instance, and a final result is obtained (usually the call to the termination action, returns the actual final result of the transformation pipeline)

```
class StreamPipelineComponents {
    public static void main(String []args) {
        Method[] objectMethods = Object.class.getMethods();
        Stream<Method> objectMethodStream = Arrays.stream(objectMethods);
        Stream<String> objectMethodNames = objectMethodStream.map(method ->
           method.getName());
        Stream<String> uniqueObjectMethodNames = objectMethodNames.distinct()
           ;
        uniqueObjectMethodNames.forEach(System.out::println);
    }
}
```

The example above shows all the different components of the Stream interface, the different methods the way they are used, the intermediate calls to methods like `map and distinct` and the termination call to `forEach`. Which makes sure that the stream is terminated, causing all the previously chained operations to execute.

Primitive streams

As already mentioned the primitive types have their own stream interface, `IntStream`, `DoubleStream`, `LongStream`, these do not directly extend from the Stream interface, rather they extend from `BaseStream`.

The API of the primitive streams is slightly different than the regular Stream API since it deals with numeric / integer types, therefore different methods are present on top of the obvious ones (like map or filter).

To create a primitive stream there are several methods that can be used. The ones listed below are the most prolific and deserve more special attention since they are the mostly used in practice, when a new stream of integers has to be generated. It is either generated from an array of integers, or within some sort of predefined range.

```
// Located in the IntStream (analogous implementation in other primitive
   stream types
public static IntStream range(int startInclusive, int endExclusive)
public static IntStream rangeClosed(int startInclusive, int endInclusive)
public static IntStream concat(IntStream a, IntStream b)
public static IntStream of(int... values)
```

Another method which could be wildly used is the iterate method, which is basically a replacement for the standard for-loop, and can be used like so.

```
// the first argument is the start of the range, the second argument is a
   function which increments the starting
// iterator, and the limit in this case is optional, but if one wants to have
    a non-infinite loop is a good idea.
IntStream.iterate(1, i -> i + 1).limit(5)
```

Creating streams

There are other ways to create streams as well, besides the newly provided static utility methods in the Stream or `IntStream` classes. The `Arrays` class was expanded with new capabilities in Java 8 allowing it to create new streams based on the target object, either a primitive (int example only given below) or a regular class type expressed as generic.

```
// Located in the Arrays class, added with Java 8 release
public static IntStream stream(int[] array)
public static <T> Stream<T> stream(T[] array)
```

The methods above are actually what is used in the implementation of the Stream.of methods, they use Arrays.stream to do the same exact thing,

```
// these are basically wrappers around the methods provided by Arrays -
   Arrays.stream
Stream.of(1, 2, 3, 4, 5)
Stream.of(new Integer[]{1, 2, 3, 4, 5})
```

Building streams

There is another way to create streams, element wise, instead of first collecting elements into a List or any other collection and then converting it into stream, if one is aware that a given collection of items will be immediately used for stream operations, then the Stream.Builder interface can be used

```
// The stream builder interface is quite simple it has two methods, both of
   which are doing the same exact thing, they
// are simply aliases, the methods are accept and add, both of which add a
   new element to the stream, the stream builder
// can be used like so.
Stream.builder().add(1).add(2).add(3).add(4).add(5).build()
```

Existing interface

There are many other ways to build streams, most of the Java classes provides a quick way to construct streams right away directly from the API of the class itself. Classes such as Files, Pattern, Random and so on have built in support to generate streams directly

```
// here are some examples which show existing java classes and how they inter
    operate with streams, directly without the
// need for intermediate collection interface or structure to be created
Path.lines(Paths.get("/")).forEach(System.out::println);
Pattern.compile(" ").splitAsStream("java 8 streams").forEach(System.out::
   println);
new RAndom().ints().limit(5).forEach(System.out::println);
"hello".chars().sorted().forEach(ch -> System.out.println("%c ", ch));
```

Note that the chars() method on the String type returns an IntStream, reason being is that chars are treated as integers in Java, factually they are shorts, or at least 2 bytes, since Java stores the String representations in UTF-16, which mandates that the size of the char is at least 2 bytes.

Intermediate operations

Intermediate operations as already shown, simply do not consume the stream but refine it, refining the stream means that the operations which were invoked on the stream object will be executed only when a termination operation is called (such as forEach, collect etc).

Method	Description
Stream<> filter(Predicate<? super T> check)	Removes the elements for which the check predicate returns false.
<> Stream<> map(Function<? super T,? extends R> transform)	Applies the transform() function for each of the elements in the stream.
Stream<> distinct()	Removes duplicate elements in the stream; it uses the equals() method to determine if an element is repeated in the stream.
Stream<> sorted(Comparator<? super T> compare)	Sorts the elements in its natural order. The overloaded version takes a Comparator-you can pass a lambda function for that.
Stream<> peek(Consumer<? super T> consume)	Returns the same elements in the stream, but also executes the passed consume lambda expression on the elements.
Stream<> limit(long size)	Removes the elements if there are more elements than the given size in the stream.

```
// here is a demonstration of the user of intermediate operations to mutate
   the stream, peek the elements, without
// terminating it and then finally invoke the count method which will
   terminate the stream, calling the count method will
// actually make sure all other intermediate operations are executed in order
    as specified
Stream.of(1, 2, 3, 4, 5).map(i -> i * i).peek(i -> System.out.printf("%d ", i
   )).count();
```

Intermediate operations do no produce result, instead they return the initial stream object instance, which can then be used to chain more intermediate operations, further more until a termination operation is invoked on the stream, the intermediate operation's

actions are never executed, the lambda expression (object)which the operation represents is simply stored in the stream. This also allows the stream to internally optimize certain tasks since the execution of the actions happens only upon stream termination)

Terminal operations

Terminating operations as already mentioned are usually the last operations to be called on a stream, before that calling any number of intermediate operations is optional, the terminating operations close the stream, meaning that if a variable references the stream object, and it is used again to call any operation on the stream (intermediate or terminating) an exception will be thrown, Stream object are usually meant for one short transformation pipelines, and they are not meant to be reusable, they are lightweight, since they simply wrap around the initial object or collection, and do not copy or clone the elements of that collection, they simply provide a way to generate a new collection of the original one, which contains the transformed elements in some state. Of course, depending on the terminating operation, and the intermediate operations entirely new objects might be created (i.e using map) after calling the final terminating operation on the stream object.

Method	Description
void forEach(Consumer<? super T> action)	Calls the action for every element in the stream.
Object[] toArray()	Returns an Object array that has the elements in the stream.
Optional<> min(Comparator<? super T> | compare)	Returns the minimum value in the stream (compares the objects using the given compare function).
Optional<> max(Comparator<? super T> | compare)	Returns the maximum value in the stream (compares the objects using the given compare function).
long count()	Returns the number of elements in the stream.

Once the stream has been finalized with calling a terminating operation, the stream is considered consumed, and any attempt to call any other operation on it - terminating or intermediate, will result in IllegalStateException