Contents

Java.lang

The core library which exposes most of the basic language features, classes and functionalities

Runtime

The `Runtime` class encapsulates the run-time environment. You cannot instantiate a `Runtime` object. However, you can get a reference to the current `Runtime` object by calling the static method Runtime.getRuntime( ). Once you obtain a reference to the current `Runtime` object, you can call several methods that control the state and behavior of the Java Virtual Machine. Applets and other `untrusted` code typically cannot call any of the `Runtime` methods

Method	Description
Process exec(String progName)	Executes the program specified by progName as a separate process
Process exec(String progName, String environment)	Executes the program specified by progName as a separate process with the environment specified by environment
Process exec(String comLineArray)	Executes the command line specified by the strings in comLineArray as a separate process

Codepoints

Relatively recently, major additions were made to Character. Beginning with `JDK 5`, the Character class has included support for `32-bit` Unicode characters. In the past, all Unicode characters could be held by 16 bits, which is the size of a char (and the size of the value encapsulated within a Character), because those values ranged from 0 to `FFFF`. However, the Unicode character set has been expanded, and more than 16 bits are required. Characters can now range from 0 to `10FFFF`. Here are three important terms. A code point is a character in the range 0 to `10FFFF`. Characters that have values greater than `FFFF` are called supplemental characters. The basic multilingual plane (`BMP`) are those characters between 0 and `FFFF`. The expansion of the Unicode character set caused a fundamental problem for Java. Because a supplemental character has a value greater than a char can hold, some means of handling the supplemental characters was needed. Java addressed this problem in two ways. First, Java uses two chars to represent a supplemental character. The first char is called the high surrogate, and the second is called the low surrogate. New methods, such as `codePointAt` , were provided to translate between code points and supplemental characters.

Object

The clone() method generates a duplicate copy of the object on which it is called. Only classes that implement the `Cloneable` interface can be cloned. The `Cloneable` interface defines no members. It is used to indicate that a class allows a `bitwise` copy of an object (that is, a clone) to be made. If you try to call clone( ) on a class that does not implement `Cloneable`, a `CloneNotSupportedException` is thrown. When a clone is made, the constructor for the object being cloned is not called. As implemented by Object, a clone is simply an exact copy of the original. Some other notable methods exposed by the object class type are

Method	Description
Object clone( ) throws	CloneNotSupportedException Creates a new object that is the same as the invoking object.
boolean equals(Object object)	Returns true if the invoking object is equivalent to object.
void finalize( ) throws Throwable	Default finalize( ) method. It is called before an unused object is recycled.
final Class<?> getClass( )	Obtains a Class object that describes the invoking object.
int hashCode( )	Returns the hash code associated with the invoking object.
final void notify( )	Resumes execution of a thread waiting on the invoking object.
final void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.

Class

Class encapsulates the run-time state of a class or interface. Objects of type Class are created automatically, when classes are loaded. You cannot explicitly declare a Class object. Generally, you obtain a Class object by calling the `getClass()` method defined by Object.

1. `Method[ ] getDeclaredMethods( )throws SecurityException` - Obtains a Method object for each method declared by the class or interface represented by the invoking object and stores them in an array. Returns a reference to this array. (Inherited methods are ignored.)
2. `Field getField(String fieldName)throws NoSuchMethodException, SecurityException` - Returns a Field object that represents the public field specified by fieldName for the class or interface represented by the invoking object.
3. `Field[ ] getFields( )throws SecurityException` - Obtains a Field object for each public field of the class or interface represented by the invoking object and stores them in an array. Returns a reference to this array.
4. `Class<?>[ ] getInterfaces( )` - When invoked on an object that represents a class, this method returns an array of the interfaces implemented by that class. When invoked on an object that represents an interface, this method returns an array of interfaces extended by that interface.
5. `Method getMethod(String methName, Class<?> ... paramTypes)throws NoSuchMethodException, SecurityException` - Returns a Method object that represents the public method specified by methName and having the parameter types specified by paramTypes in the class or interface represented by the invoking object.
6. `Method[ ] getMethods( )throws SecurityException` - Obtains a Method object for each public method of the class or interface represented by the invoking object and stores them in an array. Returns a reference to this array.

7. `String getName( )` - Returns the complete name of the class or interface of the type represented by the invoking object. ProtectionDomain getProtectionDomain( ) Returns the protection domain associated with the invoking object.
8. `Class<? super T> getSuperclass( )` - Returns the superclass of the type represented by the invoking object. The return value is null if the represented type is Object or not a class.
9. `boolean isInterface( )` - Returns true if the type represented by the invoking object is an interface. Otherwise, it returns false.
10. `T newInstance( )throws IllegalAccessException, InstantiationException` - Creates a new instance (i.e., a new object) that is of the same type as that represented by invoking object. This is equivalent to using new with the class' default constructor. The new object is returned. This method will fail if the represented type is abstract, not a class, or does not have a default constructor.
11. `String toString( )` - Returns the string representation of the type represented by the invoking object or interface. throws SecurityException
12. `Class<?>[ ] getClasses( )` - Returns a Class object for each public class and interface that is a member of the class represented by the invoking object.
13. `ClassLoader getClassLoader( )` - Returns the ClassLoader object that loaded the class or interface.
14. `Constructor<T> getConstructor(Class<?> ... paramTypes)throws NoSuchMethodException, SecurityException` - Returns a Constructor object that represents the constructor for the class represented by the invoking object that has the parameter types specified by paramTypes.
15. `Constructor<?>[ ] getConstructors( )throws SecurityException` - Obtains a Constructor object for each public constructor of the class represented by the invoking object and stores them in an array. Returns a reference to this array.

`The abstract class ClassLoader defines how classes are loaded. Your application can create subclasses that extend ClassLoader, implementing its methods. Doing so allows you to load classes in some way other than the way they are normally loaded by the Java run-time system. However, this is not something that you will normally need to do.`

Iterable

The interable interface allows all classes that implement it to take advantage of the for-each loop structures provided by the Java language, the interface has a few methods which have to be implemented by each class that wishes to provide means of being iterated over. This can then be used in wide variety of cases not just with the regular `for-each` structure block provided by the language. It can be used as argument or input parameter to the collections or stream library provided by java.lang.

```
Iterator<T> iterator(); // the main method to be implemented by the class
   wishing to be treated as iterable
```

As noticed, the iterator method above returns an `Iterator` instance, which is itself an interface as well, the iterator interface needs to also be implemented by the `iterable` class, to allow for the iteration process, the iterator interface has the following mandatory methods

```
    boolean hasNext(); // check if there is a `next` iteration element in
       the iterable object, return false if no more next elements
    E next(); // extract the `next` iteration objecdt from the iterable
       object and more the pointer forward to the next object
    default void forEachRemaining(Consumer<? super E> action); // iterates
       over all remaining elements of the current iterator
```

To show more precisely how the iterator object works, to iterate over an iterator one has to know to call only the two methods presented above, `hasNext` and `next`, in a regular while loop, the iterator object will return all elements in sequence until there is a next element to be extracted.

```
    // the general form of iteration over an iterator element
    while (hasNext())
        action.accept(next());
```

The iterable interface has one more feature introduced into java 8, which is what is called a `splititerator`, this type of iterator is used to allow data structures to be iterated concurrently, in a split manner, meaning that the structure can sub-divide itself into multiple chunks, each of which can be iterated independently, which greatly improves efficiency. What is important to note is that the elements in a split iterator must be iterable in such a way that there is no overlap between two splititerators, meaning two splititerators must never return a sub-set of the same iterable class.

```
    boolean tryAdvance(Consumer<? super T> action); // this is a
       combination of hasNext and next, from Iterator
    Spliterator<T> trySplit(); // split the current splititerator in two
       sub-splititerators
    long estimateSize(); // estimate the size of the current splititerator
```

So the general pattern when using split iterators, is to either call `trySplit` until it returns nil, meaning that the iterator can no longer be subdivided, or to use the `estimateSize` as a kill switch, instead of checking for nil, this would allow the client to avoid splitting the iterator too much. After one has obtained all split iterator references, then one can use the `tryAdvance` method, that method is optimized for parallelism, but what it does basically is combine the `hasNext and next` into one, the `tryAdvance` takes in a consumer which in essence is passed in the result of `next`, or the next element of the split iterator, if there is any.

```
// input data source, which one would like to process in parallel,
   splitting the data into split iterators
List<String> names = Arrays.asList("John", "Jane", "Tom", "Lucy", "Mark",
   "Sophie", "Anna", "Paul");

// Create the Spliterator from the initial data set
Spliterator<String> spliterator = names.spliterator();

// Create a list to store the resulting Spliterators
List<Spliterator<String>> spliterators = new ArrayList<>();

// Keep splitting until trySplit returns null or one could instead check
   for the result of estimatedSize
while (spliterator != null) {
    spliterators.add(spliterator);
    spliterator = spliterator.trySplit();  // Try splitting the current
       spliterator
}

// Process each Spliterator in the collected list (e.g., print all
   elements in each)
for (Spliterator<String> sp : spliterators) {
    sp.forEachRemaining(System.out::println);  // Process each
       spliterator's elements
}
```

Collection

This is the very top interface which is the root of all collections in java, this one contains generic methods to control collections, such as iteration, element addition, removal etc. Most of the methods from the `collection` interface are presented below.

Method	Description
size()	reutrn the size of the collection
isEmpty()	check if the collection is empty
contains(Object)	check if collection contains element using equals
iterator()	return an iterator for the current collection
toArray()	convert the collection to a raw dynamic array
toArray(T[]) <T>	copy the collection to array if it fits in
toArray(IntFunction<T[]>) <T>	convert to array, size of which is determined by the IntFunction
add(E)	add element to the collection
remove(Object)	remove element from the collection based on equals
containsAll(Collection<?>)	check if elements from one collection contained into another using equals
addAll(Collection<? extends E>)	add all elements from one collection into the other
removeAll(Collection<?>)	remove all elements from current collection matching the elements from another
removeIf(Predicate<? super E>)	remove element from the collection only if a given predicate for it matches
retainAll(Collection<?>)	retain all elements from the other collections which are contained in the current
clear()	clear all elements from the current collection
spliterator()	returns a split iterator for the current collection
stream()	covert the collection into a stream type object
parallelStream()	covert the collection into a parallel stream type object

List

This interface is an extension of the Collection interface, and represents a sequential collection of elements, it add a few more list related methods which help one work with sequential structures better

Method	Description
get(int)	extract an element based on it's index in the sequence
set(int, E)	set/update an element based on it's index in the sequence
add(int, E)	insert an element on a specific index location in the sequence
remove(int)	remove an element on a specific index location in the sequence
indexOf(Object)	find the index of an element, based on equals, in the sequence
lastIndexOf(Object)	find the last index of an element, based on equals, in the sequence

Set

This interface is another extension of the Collection interface, and it is meant to represent a set, which can hold only unique elements, it adds a few more set related methods. A variation of this interface is another one which extends the Set interface and it is called `SortedSet` which similarly to the `Set` interface does not allow

duplicate elements, however it also provides means of extracting or iterating over the elements of the set in an ordered / sorted manner.

The most notable method, that has to be implemented by all classes which desire to classify themselves as sorted sets, is the `comparator` method, which is what allows the `SortedSet` to be sorted.

```
Comparator<? super E> comparator(); // must be implemented by classes,
   to comply to the `SortedSet` behavior
```

`Note that null elements are not allowed in a Set or SortedSet, as they would violate the definition of the Set in the first place`

Queue

This interface is another extension of the Collection interface, and like the other ones, it provides a specific behavior over the standard Collection interface, in this case the Queue interface defines a collection for which elements are first-in-first-out order.

Method	Description
offer(E)	add new element at the end of the queue, true if succesful, false otherwise
add(E)	same as offer however if queue is capactiy limited throws exception
poll()	remove from the queue, the oldest element
remove()	same as poll, but throws an exception if queue is empty
peek()	only peek the oldest element at the front without removing
element()	same as peek, but throws an exception if queue is empty

`Note that null elements are not allowed in a Queue in first place therefore some of the methods (like add)will throw an exception, further more as noted above, some will also throw if the Queue is empty and one tries to obtain element from it, or if the queue has a capactiy limit and one tries to insert into it`

There are some special types of Queues which the `java.lang` implements, such as the so called `PriorityQueue` which is basically a min/max heap, this is a type of data structure where at the top of the queue or heap the max or min element in the structure are stored, the `PriorityQueue` has a comparator function which will be used to compare the elements, otherwise if no comparator function is specified when the queue is created, the default ordering of the elements will be used. Note that this might mean that the elements have to implement the `Comparable` interface in order for the ordering to work correctly, otherwise exception will be thrown if the contained elements do not implement that interface.

Another data structure which is an extension or variation of the `Queue` is the `ArrayDeque` which is a double ended `Queue` structure, which allows one to put elements at both ends of the queue, at the front and end. This structure is usually implemented using a standard `ArrayList` or `LinkedList` which generally allow for quick insertion at both ends. Another name for this type of Queue is a Deck. The methods this class provides help one work with both ends of the queue, unlike the regular queue interface. Below is a set of the most important methods provided by the `Dequeue|Deck` interface

Method	Descrption
addFirst(E e)	Inserts the specified element at the front of the deque.
addLast(E e)	Inserts the specified element at the end of the deque.
offerFirst(E e)	Inserts the specified element at the front of the deque, returning `true` if successful or `false` otherwise.

Method	Descrption
offerLast(E e)	Inserts the specified element at the end of the deque, returning `true` if successful or `false` otherwise.
removeFirst()	Removes and returns the first element of the deque. Throws an exception if the deque is empty.
removeLast()	Removes and returns the last element of the deque. Throws an exception if the deque is empty.
pollFirst()	Removes and returns the first element of the deque, or returns `null` if the deque is empty.
pollLast()	Removes and returns the last element of the deque, or returns `null` if the deque is empty.
getFirst()	Retrieves, but does not remove, the first element of the deque. Throws an exception if the deque is empty.
getLast()	Retrieves, but does not remove, the last element of the deque. Throws an exception if the deque is empty.
peekFirst()	Retrieves, but does not remove, the first element of the deque, or returns `null` if the deque is empty.
peekLast()	Retrieves, but does not remove, the last element of the deque, or returns `null` if the deque is empty.
removeFirstOccurrence(Object o)	Removes the first occurrence of the specified element from the deque.
removeLastOccurrence(Object o)	Removes the last occurrence of the specified element from the deque.

Map

The map interface provides an easy way to associate a key with a value, where both the key and value pairs can be of any class type, which makes the map pretty powerful, there are many implementations, based on trees, lists, buckets, etc (`TreeMap`, `HashMap` etc). Some of the most notable maps used in practice are listed below, note that there are many more which the java.lang provides out of the box.

Method	Description
EnumMap	specialized map which is meant to be used with Enum type keys
TreeMap	a black-red tree based implementation for a map
HashMap	a hash code based implementation for a map
SortedMap	a map which provides a sorted behavior for its keys
Hashtable	a synchronized version of HashMap
WeakHashMap	a map where the key/value pair will be removed if the key is no longer in ordinary use
NavigableMap	a map which provides means of returning closest matches for a given search target
LinkedHashMap	a linked list based implementation for a map

```
Generally all types of elements are allowed in a map, as long as they comply with the
element requirements of the map, for example for HashMap a key has to have the hashCode
method overriden correctly, for a Tree based map implementation one has to also have
the equals method overriden, on top of that these elements have to also implement the
comparable inteface, or provide a comparator function implementation
```

The Map has the following general interface methods, besides the one inherited from the root Collection interface

Method	Description
containsKey(Object key)	Returns `true` if this map contains a mapping for the specified key.
containsValue(Object value)	Returns `true` if this map maps one or more keys to the specified value.
get(Object key)	Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
getOrDefault(Object key, V defaultValue)	Returns the value to which the specified key is mapped, or `defaultValue` if this map contains no mapping for the key.
put(K key, V value)	Associates the specified value with the specified key in this map.
putIfAbsent(K key, V value)	Associates the specified key with the specified value if it is not already associated with a value.
remove(Object key)	Removes the mapping for a key from this map if present.
clear()	Removes all mappings from this map.
values()	Returns a `Collection` view of the values contained in this map.
keySet()	Returns a `Set` view of the keys contained in this map.
entrySet()	Returns a `Set` view of the mappings contained in this map (key-value pairs).
remove(Object key, Object value)	Removes the entry for the specified key only if it is currently mapped to the specified value.
replace(K key, V oldValue, V newValue)	Replaces the entry for the specified key only if it is currently mapped to the specified value.
replace(K key, V value)	Replaces the entry for the specified key with the given value if it is currently mapped to some value.
computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)	If the specified key is not already associated with a value, computes its value using the given mapping function and puts it in map
computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)	If the key is present and associated with a non-null value, computes a new value and updates the entry.
compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)	Attempts to compute a mapping for the specified key and its current mapped value (or `null` if none).
merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)	Merges the specified value with the existing value (if any) associated with the key, using a remapping function.

The Map.Entry is the entry which represents the pair of key and value that a map an store. The Map.Entry, has the following interface.

Method	Description
getKey()	Returns the key corresponding to this map entry.
getValue()	Returns the value corresponding to this map entry.
setValue(V value)	Replaces the value corresponding to this entry with the specified value.
equals(Object o)	Compares the specified object with this entry for equality.
hashCode()	Returns the hash code value for this map entry.
comparingByKey() <K extends Comparable<? super K>, V>	Returns a comparator that compares `Map.Entry` objects in natural order by key.

Method	Description
comparingByValue() <K, V extends Comparable<? super V»	Returns a comparator that compares `Map.Entry` objects in natural order by value.
comparingByKey(Comparator<? super K>) <K, V>	Returns a comparator that compares `Map.Entry` objects by key using the specified comparator.
comparingByValue(Comparator<? super V>) <K, V>	Returns a comparator that compares `Map.Entry` objects by value using the specified comparator.