java.util

Class Arrays

java.lang.Object
    java.util.Arrays

```
public class Arrays
extends Object
```

This class contains various methods for manipulating arrays (such as sorting and searching). This class also contains a static factory that allows arrays to be viewed as lists.

The methods in this class all throw a `NullPointerException`, if the specified array reference is null, except where noted.

The documentation for the methods contained in this class includes briefs description of the *implementations*. Such descriptions should be regarded as *implementation notes*, rather than parts of the *specification*. Implementors should feel free to substitute other algorithms, so long as the specification itself is adhered to. (For example, the algorithm used by `sort(Object[])` does not have to be a MergeSort, but it does have to be *stable*.)

This class is a member of the Java Collections Framework.

Since:

1.2

Method Summary

Methods

Modifier and Type	Method and Description
`static <T> List<T>`	`asList(T... a)` Returns a fixed-size list backed by the specified array.
`static int`	`binarySearch(byte[] a, byte key)` Searches the specified array of bytes for the specified value using the binary search algorithm.
`static int`	`binarySearch(byte[] a, int fromIndex, int toIndex, byte key)` Searches a range of the specified array of bytes for the specified value using the binary search algorithm.
`static int`	`binarySearch(char[] a, char key)` Searches the specified array of chars for the specified value using the binary search algorithm.
`static int`	`binarySearch(char[] a, int fromIndex, int toIndex, char key)` Searches a range of the specified array of chars for the specified value using the binary search algorithm.
`static int`	`binarySearch(double[] a, double key)` Searches the specified array of doubles for the specified value using the binary search algorithm.
`static int`	`binarySearch(double[] a, int fromIndex, int toIndex, double key)` Searches a range of the specified array of doubles for the specified value using the binary search algorithm.
`static int`	`binarySearch(float[] a, float key)`

	Searches the specified array of floats for the specified value using the binary search algorithm.
static int	**binarySearch**(float[] a, int fromIndex, int toIndex, float key) Searches a range of the specified array of floats for the specified value using the binary search algorithm.
static int	**binarySearch**(int[] a, int key) Searches the specified array of ints for the specified value using the binary search algorithm.
static int	**binarySearch**(int[] a, int fromIndex, int toIndex, int key) Searches a range of the specified array of ints for the specified value using the binary search algorithm.
static int	**binarySearch**(long[] a, int fromIndex, int toIndex, long key) Searches a range of the specified array of longs for the specified value using the binary search algorithm.
static int	**binarySearch**(long[] a, long key) Searches the specified array of longs for the specified value using the binary search algorithm.
static int	**binarySearch**(**Object**[] a, int fromIndex, int toIndex, **Object** key) Searches a range of the specified array for the specified object using the binary search algorithm.
static int	**binarySearch**(**Object**[] a, **Object** key) Searches the specified array for the specified object using the binary search algorithm.
static int	**binarySearch**(short[] a, int fromIndex, int toIndex, short key) Searches a range of the specified array of shorts for the specified value using the binary search algorithm.
static int	**binarySearch**(short[] a, short key) Searches the specified array of shorts for the specified value using the binary search algorithm.
static <T> int	**binarySearch**(T[] a, int fromIndex, int toIndex, T key, **Comparator**<? super T> c) Searches a range of the specified array for the specified object using the binary search algorithm.
static <T> int	**binarySearch**(T[] a, T key, **Comparator**<? super T> c) Searches the specified array for the specified object using the binary search algorithm.
static boolean[]	**copyOf**(boolean[] original, int newLength) Copies the specified array, truncating or padding with `false` (if necessary) so the copy has the specified length.
static byte[]	**copyOf**(byte[] original, int newLength) Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
static char[]	**copyOf**(char[] original, int newLength) Copies the specified array, truncating or padding with null characters (if necessary) so the copy has the specified length.
static double[]	**copyOf**(double[] original, int newLength) Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
static float[]	**copyOf**(float[] original, int newLength) Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
static int[]	**copyOf**(int[] original, int newLength) Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
static long[]	**copyOf**(long[] original, int newLength) Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
static short[]	**copyOf**(short[] original, int newLength)

	Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
static <T> T[]	**copyOf**(T[] original, int newLength) Copies the specified array, truncating or padding with nulls (if necessary) so the copy has the specified length.
static <T,U> T[]	**copyOf**(U[] original, int newLength, **Class**<? extends T[]> newType) Copies the specified array, truncating or padding with nulls (if necessary) so the copy has the specified length.
static boolean[]	**copyOfRange**(boolean[] original, int from, int to) Copies the specified range of the specified array into a new array.
static byte[]	**copyOfRange**(byte[] original, int from, int to) Copies the specified range of the specified array into a new array.
static char[]	**copyOfRange**(char[] original, int from, int to) Copies the specified range of the specified array into a new array.
static double[]	**copyOfRange**(double[] original, int from, int to) Copies the specified range of the specified array into a new array.
static float[]	**copyOfRange**(float[] original, int from, int to) Copies the specified range of the specified array into a new array.
static int[]	**copyOfRange**(int[] original, int from, int to) Copies the specified range of the specified array into a new array.
static long[]	**copyOfRange**(long[] original, int from, int to) Copies the specified range of the specified array into a new array.
static short[]	**copyOfRange**(short[] original, int from, int to) Copies the specified range of the specified array into a new array.
static <T> T[]	**copyOfRange**(T[] original, int from, int to) Copies the specified range of the specified array into a new array.
static <T,U> T[]	**copyOfRange**(U[] original, int from, int to, **Class**<? extends T[]> newType) Copies the specified range of the specified array into a new array.
static boolean	**deepEquals**(**Object**[] a1, **Object**[] a2) Returns true if the two specified arrays are *deeply equal* to one another.
static int	**deepHashCode**(**Object**[] a) Returns a hash code based on the "deep contents" of the specified array.
static **String**	**deepToString**(**Object**[] a) Returns a string representation of the "deep contents" of the specified array.
static boolean	**equals**(boolean[] a, boolean[] a2) Returns true if the two specified arrays of booleans are *equal* to one another.
static boolean	**equals**(byte[] a, byte[] a2) Returns true if the two specified arrays of bytes are *equal* to one another.
static boolean	**equals**(char[] a, char[] a2) Returns true if the two specified arrays of chars are *equal* to one another.
static boolean	**equals**(double[] a, double[] a2) Returns true if the two specified arrays of doubles are *equal* to one another.
static boolean	**equals**(float[] a, float[] a2) Returns true if the two specified arrays of floats are *equal* to one another.
static boolean	**equals**(int[] a, int[] a2) Returns true if the two specified arrays of ints are *equal* to one another.
static boolean	**equals**(long[] a, long[] a2) Returns true if the two specified arrays of longs are *equal* to one another.
static boolean	**equals**(**Object**[] a, **Object**[] a2) Returns true if the two specified arrays of Objects are *equal* to one another.
static boolean	**equals**(short[] a, short[] a2) Returns true if the two specified arrays of shorts are *equal* to one another.
static void	**fill**(boolean[] a, boolean val)

	Assigns the specified boolean value to each element of the specified array of booleans.
static void	**fill**(boolean[] a, int fromIndex, int toIndex, boolean val) Assigns the specified boolean value to each element of the specified range of the specified array of booleans.
static void	**fill**(byte[] a, byte val) Assigns the specified byte value to each element of the specified array of bytes.
static void	**fill**(byte[] a, int fromIndex, int toIndex, byte val) Assigns the specified byte value to each element of the specified range of the specified array of bytes.
static void	**fill**(char[] a, char val) Assigns the specified char value to each element of the specified array of chars.
static void	**fill**(char[] a, int fromIndex, int toIndex, char val) Assigns the specified char value to each element of the specified range of the specified array of chars.
static void	**fill**(double[] a, double val) Assigns the specified double value to each element of the specified array of doubles.
static void	**fill**(double[] a, int fromIndex, int toIndex, double val) Assigns the specified double value to each element of the specified range of the specified array of doubles.
static void	**fill**(float[] a, float val) Assigns the specified float value to each element of the specified array of floats.
static void	**fill**(float[] a, int fromIndex, int toIndex, float val) Assigns the specified float value to each element of the specified range of the specified array of floats.
static void	**fill**(int[] a, int val) Assigns the specified int value to each element of the specified array of ints.
static void	**fill**(int[] a, int fromIndex, int toIndex, int val) Assigns the specified int value to each element of the specified range of the specified array of ints.
static void	**fill**(long[] a, int fromIndex, int toIndex, long val) Assigns the specified long value to each element of the specified range of the specified array of longs.
static void	**fill**(long[] a, long val) Assigns the specified long value to each element of the specified array of longs.
static void	**fill**(**Object**[] a, int fromIndex, int toIndex, **Object** val) Assigns the specified Object reference to each element of the specified range of the specified array of Objects.
static void	**fill**(**Object**[] a, **Object** val) Assigns the specified Object reference to each element of the specified array of Objects.
static void	**fill**(short[] a, int fromIndex, int toIndex, short val) Assigns the specified short value to each element of the specified range of the specified array of shorts.
static void	**fill**(short[] a, short val) Assigns the specified short value to each element of the specified array of shorts.
static int	**hashCode**(boolean[] a) Returns a hash code based on the contents of the specified array.
static int	**hashCode**(byte[] a) Returns a hash code based on the contents of the specified array.
static int	**hashCode**(char[] a) Returns a hash code based on the contents of the specified array.
static int	**hashCode**(double[] a) Returns a hash code based on the contents of the specified array.
static int	**hashCode**(float[] a) Returns a hash code based on the contents of the specified array.
static int	**hashCode**(int[] a)

	Returns a hash code based on the contents of the specified array.
static int	**hashCode**(long[] a) Returns a hash code based on the contents of the specified array.
static int	**hashCode**(Object[] a) Returns a hash code based on the contents of the specified array.
static int	**hashCode**(short[] a) Returns a hash code based on the contents of the specified array.
static void	**sort**(byte[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(byte[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(char[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(char[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(double[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(double[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(float[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(float[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(int[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(int[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(long[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(long[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(Object[] a) Sorts the specified array of objects into ascending order, according to the **natural ordering** of its elements.
static void	**sort**(Object[] a, int fromIndex, int toIndex) Sorts the specified range of the specified array of objects into ascending order, according to the **natural ordering** of its elements.
static void	**sort**(short[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(short[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static <T> void	**sort**(T[] a, **Comparator**<? super T> c) Sorts the specified array of objects according to the order induced by the specified comparator.
static <T> void	**sort**(T[] a, int fromIndex, int toIndex, **Comparator**<? super T> c) Sorts the specified range of the specified array of objects according to the order induced by the specified comparator.
static **String**	**toString**(boolean[] a) Returns a string representation of the contents of the specified array.
static **String**	**toString**(byte[] a) Returns a string representation of the contents of the specified array.
static **String**	**toString**(char[] a) Returns a string representation of the contents of the specified array.
static **String**	**toString**(double[] a)

	Returns a string representation of the contents of the specified array.
static String	**toString**(float[] a) Returns a string representation of the contents of the specified array.
static String	**toString**(int[] a) Returns a string representation of the contents of the specified array.
static String	**toString**(long[] a) Returns a string representation of the contents of the specified array.
static String	**toString**(Object[] a) Returns a string representation of the contents of the specified array.
static String	**toString**(short[] a) Returns a string representation of the contents of the specified array.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Method Detail

sort

```
public static void sort(int[] a)
```

Sorts the specified array into ascending numerical order.

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

a - the array to be sorted

sort

```
public static void sort(int[] a,
        int fromIndex,
        int toIndex)
```

Sorts the specified range of the array into ascending order. The range to be sorted extends from the index `fromIndex`, inclusive, to the index `toIndex`, exclusive. If `fromIndex == toIndex`, the range to be sorted is empty.

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

a - the array to be sorted

fromIndex - the index of the first element, inclusive, to be sorted

toIndex - the index of the last element, exclusive, to be sorted

Throws:

IllegalArgumentException - if `fromIndex > toIndex`

ArrayIndexOutOfBoundsException - if `fromIndex < 0` or `toIndex > a.length`

sort

```
public static void sort(long[] a)
```

Sorts the specified array into ascending numerical order.

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

`a` - the array to be sorted

sort

```
public static void sort(long[] a,
        int fromIndex,
        int toIndex)
```

Sorts the specified range of the array into ascending order. The range to be sorted extends from the index `fromIndex`, inclusive, to the index `toIndex`, exclusive. If `fromIndex == toIndex`, the range to be sorted is empty.

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

`a` - the array to be sorted

`fromIndex` - the index of the first element, inclusive, to be sorted

`toIndex` - the index of the last element, exclusive, to be sorted

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0` or `toIndex > a.length`

sort

```
public static void sort(short[] a)
```

Sorts the specified array into ascending numerical order.

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

`a` - the array to be sorted

sort

```
public static void sort(short[] a,
        int fromIndex,
        int toIndex)
```

Sorts the specified range of the array into ascending order. The range to be sorted extends from the index `fromIndex`, inclusive, to the index `toIndex`, exclusive. If `fromIndex == toIndex`, the range to be sorted is empty.

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

`a` - the array to be sorted

`fromIndex` - the index of the first element, inclusive, to be sorted

`toIndex` - the index of the last element, exclusive, to be sorted

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0` or `toIndex > a.length`

sort

```
public static void sort(char[] a)
```

Sorts the specified array into ascending numerical order.

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

`a` - the array to be sorted

sort

```
public static void sort(char[] a,
        int fromIndex,
        int toIndex)
```

Sorts the specified range of the array into ascending order. The range to be sorted extends from the index `fromIndex`, inclusive, to the index `toIndex`, exclusive. If `fromIndex == toIndex`, the range to be sorted is empty.

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

`a` - the array to be sorted

`fromIndex` - the index of the first element, inclusive, to be sorted

`toIndex` - the index of the last element, exclusive, to be sorted

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0` or `toIndex > a.length`

sort

```
public static void sort(byte[] a)
```

Sorts the specified array into ascending numerical order.

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

`a` - the array to be sorted

sort

```
public static void sort(byte[] a,
        int fromIndex,
        int toIndex)
```

Sorts the specified range of the array into ascending order. The range to be sorted extends from the index `fromIndex`, inclusive, to the index `toIndex`, exclusive. If `fromIndex == toIndex`, the range to be sorted is empty.

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

`a` - the array to be sorted

`fromIndex` - the index of the first element, inclusive, to be sorted

`toIndex` - the index of the last element, exclusive, to be sorted

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0` or `toIndex > a.length`

sort

```
public static void sort(float[] a)
```

Sorts the specified array into ascending numerical order.

The < relation does not provide a total order on all float values: `-0.0f == 0.0f` is `true` and a `Float.NaN` value compares neither less than, greater than, nor equal to any value, even itself. This method uses the total order imposed by the method `Float.compareTo(java.lang.Float)`: `-0.0f` is treated as less than value `0.0f` and `Float.NaN` is considered greater than any other value and all `Float.NaN` values are considered equal.

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

`a` - the array to be sorted

sort

```
public static void sort(float[] a,
        int fromIndex,
        int toIndex)
```

Sorts the specified range of the array into ascending order. The range to be sorted extends from the index `fromIndex`, inclusive, to the index `toIndex`, exclusive. If `fromIndex == toIndex`, the range to be sorted is empty.

The < relation does not provide a total order on all float values: `-0.0f == 0.0f` is `true` and a `Float.NaN` value compares neither less than, greater than, nor equal to any value, even itself. This method uses the total order imposed by the method `Float.compareTo(java.lang.Float)`: `-0.0f` is treated as less than value `0.0f` and `Float.NaN` is considered greater than any other value and all `Float.NaN` values are considered equal.

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

`a` - the array to be sorted

`fromIndex` - the index of the first element, inclusive, to be sorted

`toIndex` - the index of the last element, exclusive, to be sorted

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0` or `toIndex > a.length`

sort

```
public static void sort(double[] a)
```

Sorts the specified array into ascending numerical order.

The < relation does not provide a total order on all double values: `-0.0d == 0.0d` is `true` and a `Double.NaN` value compares neither less than, greater than, nor equal to any value, even itself. This method uses the total order imposed by the method `Double.compareTo(java.lang.Double)`: `-0.0d` is treated as less than value `0.0d` and `Double.NaN` is considered greater than any other value and all `Double.NaN` values are considered equal.

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

`a` - the array to be sorted

sort

```
public static void sort(double[] a,
        int fromIndex,
        int toIndex)
```

Sorts the specified range of the array into ascending order. The range to be sorted extends from the index `fromIndex`, inclusive, to the index `toIndex`, exclusive. If `fromIndex == toIndex`, the range to be sorted is empty.

The < relation does not provide a total order on all double values: `-0.0d == 0.0d` is `true` and a `Double.NaN` value compares neither less than, greater than, nor equal to any value, even itself. This method uses the total order imposed by the method `Double.compareTo(java.lang.Double)`: `-0.0d` is treated as less than value `0.0d` and `Double.NaN` is considered greater than any other value and all `Double.NaN` values are considered equal.

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

`a` - the array to be sorted

`fromIndex` - the index of the first element, inclusive, to be sorted

`toIndex` - the index of the last element, exclusive, to be sorted

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0` or `toIndex > a.length`

sort

```
public static void sort(Object[] a)
```

Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. All elements in the array must implement the `Comparable` interface. Furthermore, all elements in the array must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the array).

This sort is guaranteed to be *stable*: equal elements will not be reordered as a result of the sort.

Implementation note: This implementation is a stable, adaptive, iterative mergesort that requires far fewer than n lg(n) comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately n comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to n/2 object references for randomly ordered input arrays.

The implementation takes equal advantage of ascending and descending order in its input array, and can take advantage of ascending and descending order in different parts of the the same input array. It is well-suited to merging two or more sorted arrays: simply concatenate the arrays and sort the resulting array.

The implementation was adapted from Tim Peters's list sort for Python (TimSort). It uses techiques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474, January 1993.

Parameters:

`a` - the array to be sorted

Throws:

`ClassCastException` - if the array contains elements that are not *mutually comparable* (for example, strings and integers)

`IllegalArgumentException` - (optional) if the natural ordering of the array elements is found to violate the `Comparable` contract

sort

```
public static void sort(Object[] a,
        int fromIndex,
        int toIndex)
```

Sorts the specified range of the specified array of objects into ascending order, according to the natural ordering of its elements. The range to be sorted extends from index `fromIndex`, inclusive, to index `toIndex`, exclusive. (If `fromIndex==toIndex`, the range to be sorted is empty.) All elements in this range must implement the `Comparable` interface. Furthermore, all elements in this range must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the array).

This sort is guaranteed to be *stable*: equal elements will not be reordered as a result of the sort.

Implementation note: This implementation is a stable, adaptive, iterative mergesort that requires far fewer than n lg(n) comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately n comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to n/2 object references for randomly ordered input arrays.

The implementation takes equal advantage of ascending and descending order in its input array, and can take advantage of ascending and descending order in different parts of the the same input array. It is well-suited to merging two or more sorted arrays: simply concatenate the arrays and sort the resulting array.

The implementation was adapted from Tim Peters's list sort for Python (TimSort). It uses techiques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474, January 1993.

Parameters:

`a` - the array to be sorted

`fromIndex` - the index of the first element (inclusive) to be sorted

`toIndex` - the index of the last element (exclusive) to be sorted

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex` or (optional) if the natural ordering of the array elements is found to violate the `Comparable` contract

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0` or `toIndex > a.length`

`ClassCastException` - if the array contains elements that are not *mutually comparable* (for example, strings and integers).

sort

```
public static <T> void sort(T[] a,
            Comparator<? super T> c)
```

Sorts the specified array of objects according to the order induced by the specified comparator. All elements in the array must be *mutually comparable* by the specified comparator (that is, `c.compare(e1, e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the array).

This sort is guaranteed to be *stable*: equal elements will not be reordered as a result of the sort.

Implementation note: This implementation is a stable, adaptive, iterative mergesort that requires far fewer than n lg(n) comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately n comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to n/2 object references for randomly ordered input arrays.

The implementation takes equal advantage of ascending and descending order in its input array, and can take advantage of ascending and descending order in different parts of the the same input array. It is well-suited to merging two or more sorted arrays: simply concatenate the arrays and sort the resulting array.

The implementation was adapted from Tim Peters's list sort for Python (TimSort). It uses techiques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474, January 1993.

Parameters:

`a` - the array to be sorted

`c` - the comparator to determine the order of the array. A `null` value indicates that the elements' natural ordering should be used.

Throws:

`ClassCastException` - if the array contains elements that are not *mutually comparable* using the specified comparator

`IllegalArgumentException` - (optional) if the comparator is found to violate the `Comparator` contract

sort

```
public static <T> void sort(T[] a,
            int fromIndex,
            int toIndex,
            Comparator<? super T> c)
```

Sorts the specified range of the specified array of objects according to the order induced by the specified comparator. The range to be sorted extends from index `fromIndex`, inclusive, to index `toIndex`, exclusive. (If `fromIndex==toIndex`, the range to be sorted is empty.) All elements in the range must be *mutually comparable* by the specified comparator (that is, `c.compare(e1, e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the range).

This sort is guaranteed to be *stable*: equal elements will not be reordered as a result of the sort.

Implementation note: This implementation is a stable, adaptive, iterative mergesort that requires far fewer than n lg(n) comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately n comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to n/2 object references for randomly ordered input arrays.

The implementation takes equal advantage of ascending and descending order in its input array, and can take advantage of ascending and descending order in different parts of the the same input array. It is well-suited to merging two or more sorted arrays: simply concatenate the arrays and sort the resulting array.

The implementation was adapted from Tim Peters's list sort for Python (TimSort). It uses techiques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474, January 1993.

Parameters:

`a` - the array to be sorted

`fromIndex` - the index of the first element (inclusive) to be sorted

`toIndex` - the index of the last element (exclusive) to be sorted

`c` - the comparator to determine the order of the array. A `null` value indicates that the elements' natural ordering should be used.

Throws:

`ClassCastException` - if the array contains elements that are not *mutually comparable* using the specified comparator.

`IllegalArgumentException` - if `fromIndex > toIndex` or (optional) if the comparator is found to violate the `Comparator` contract

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0` or `toIndex > a.length`

binarySearch

```
public static int binarySearch(long[] a,
               long key)
```

Searches the specified array of longs for the specified value using the binary search algorithm. The array must be sorted (as by the `sort(long[])` method) prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements with the specified value, there is no guarantee which one will be found.

Parameters:

`a` - the array to be searched

`key` - the value to be searched for

Returns:

index of the search key, if it is contained in the array; otherwise, `(-(`*`insertion point`*`) - 1)`. The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element greater than the key, or `a.length` if all elements in the array are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

binarySearch

```
public static int binarySearch(long[] a,
               int fromIndex,
```

```
                int toIndex,
                long key)
```

Searches a range of the specified array of longs for the specified value using the binary search algorithm. The range must be sorted (as by the `sort(long[], int, int)` method) prior to making this call. If it is not sorted, the results are undefined. If the range contains multiple elements with the specified value, there is no guarantee which one will be found.

Parameters:

`a` - the array to be searched

`fromIndex` - the index of the first element (inclusive) to be searched

`toIndex` - the index of the last element (exclusive) to be searched

`key` - the value to be searched for

Returns:

index of the search key, if it is contained in the array within the specified range; otherwise, `(-(`*`insertion point`*`) - 1)`. The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element in the range greater than the key, or `toIndex` if all elements in the range are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0 or toIndex > a.length`

Since:

1.6

binarySearch

```
public static int binarySearch(int[] a,
                int key)
```

Searches the specified array of ints for the specified value using the binary search algorithm. The array must be sorted (as by the `sort(int[])` method) prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements with the specified value, there is no guarantee which one will be found.

Parameters:

`a` - the array to be searched

`key` - the value to be searched for

Returns:

index of the search key, if it is contained in the array; otherwise, `(-(`*`insertion point`*`) - 1)`. The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element greater than the key, or `a.length` if all elements in the array are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

binarySearch

```
public static int binarySearch(int[] a,
                int fromIndex,
                int toIndex,
                int key)
```

Searches a range of the specified array of ints for the specified value using the binary search algorithm. The range must be sorted (as by the `sort(int[], int, int)` method) prior to making this call. If it is not sorted, the results are undefined. If the range contains multiple elements with the specified value, there is no guarantee which one will be found.

Parameters:

a - the array to be searched

fromIndex - the index of the first element (inclusive) to be searched

toIndex - the index of the last element (exclusive) to be searched

key - the value to be searched for

Returns:

index of the search key, if it is contained in the array within the specified range; otherwise, (-(*insertion point*) - 1). The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element in the range greater than the key, or toIndex if all elements in the range are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

Throws:

IllegalArgumentException - if fromIndex > toIndex

ArrayIndexOutOfBoundsException - if fromIndex < 0 or toIndex > a.length

Since:

1.6

binarySearch

```
public static int binarySearch(short[] a,
               short key)
```

Searches the specified array of shorts for the specified value using the binary search algorithm. The array must be sorted (as by the sort(short[]) method) prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements with the specified value, there is no guarantee which one will be found.

Parameters:

a - the array to be searched

key - the value to be searched for

Returns:

index of the search key, if it is contained in the array; otherwise, (-(*insertion point*) - 1). The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element greater than the key, or a.length if all elements in the array are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

binarySearch

```
public static int binarySearch(short[] a,
               int fromIndex,
               int toIndex,
               short key)
```

Searches a range of the specified array of shorts for the specified value using the binary search algorithm. The range must be sorted (as by the sort(short[], int, int) method) prior to making this call. If it is not sorted, the results are undefined. If the range contains multiple elements with the specified value, there is no guarantee which one will be found.

Parameters:

a - the array to be searched

fromIndex - the index of the first element (inclusive) to be searched

toIndex - the index of the last element (exclusive) to be searched

key - the value to be searched for

Returns:

index of the search key, if it is contained in the array within the specified range; otherwise, `(-(insertion point) - 1)`. The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element in the range greater than the key, or `toIndex` if all elements in the range are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0 or toIndex > a.length`

Since:

1.6

binarySearch

```
public static int binarySearch(char[] a,
               char key)
```

Searches the specified array of chars for the specified value using the binary search algorithm. The array must be sorted (as by the `sort(char[])` method) prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements with the specified value, there is no guarantee which one will be found.

Parameters:

`a` - the array to be searched

`key` - the value to be searched for

Returns:

index of the search key, if it is contained in the array; otherwise, `(-(insertion point) - 1)`. The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element greater than the key, or `a.length` if all elements in the array are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

binarySearch

```
public static int binarySearch(char[] a,
               int fromIndex,
               int toIndex,
               char key)
```

Searches a range of the specified array of chars for the specified value using the binary search algorithm. The range must be sorted (as by the `sort(char[], int, int)` method) prior to making this call. If it is not sorted, the results are undefined. If the range contains multiple elements with the specified value, there is no guarantee which one will be found.

Parameters:

`a` - the array to be searched

`fromIndex` - the index of the first element (inclusive) to be searched

`toIndex` - the index of the last element (exclusive) to be searched

`key` - the value to be searched for

Returns:

index of the search key, if it is contained in the array within the specified range; otherwise, `(-(insertion point) - 1)`. The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element in the range greater than the key, or `toIndex` if all elements in the range are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0 or toIndex > a.length`

Since:

1.6

binarySearch

```
public static int binarySearch(byte[] a,
               byte key)
```

Searches the specified array of bytes for the specified value using the binary search algorithm. The array must be sorted (as by the `sort(byte[])` method) prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements with the specified value, there is no guarantee which one will be found.

Parameters:

`a` - the array to be searched

`key` - the value to be searched for

Returns:

index of the search key, if it is contained in the array; otherwise, `(-(`*`insertion point`*`) - 1)`. The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element greater than the key, or `a.length` if all elements in the array are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

binarySearch

```
public static int binarySearch(byte[] a,
               int fromIndex,
               int toIndex,
               byte key)
```

Searches a range of the specified array of bytes for the specified value using the binary search algorithm. The range must be sorted (as by the `sort(byte[], int, int)` method) prior to making this call. If it is not sorted, the results are undefined. If the range contains multiple elements with the specified value, there is no guarantee which one will be found.

Parameters:

`a` - the array to be searched

`fromIndex` - the index of the first element (inclusive) to be searched

`toIndex` - the index of the last element (exclusive) to be searched

`key` - the value to be searched for

Returns:

index of the search key, if it is contained in the array within the specified range; otherwise, `(-(`*`insertion point`*`) - 1)`. The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element in the range greater than the key, or `toIndex` if all elements in the range are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0 or toIndex > a.length`

Since:

1.6

binarySearch

```
public static int binarySearch(double[] a,
               double key)
```

Searches the specified array of doubles for the specified value using the binary search algorithm. The array must be sorted (as by the `sort(double[])` method) prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements with the specified value, there is no guarantee which one will be found. This method considers all NaN values to be equivalent and equal.

Parameters:

`a` - the array to be searched

`key` - the value to be searched for

Returns:

index of the search key, if it is contained in the array; otherwise, `(-(`*`insertion point`*`) - 1)`. The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element greater than the key, or `a.length` if all elements in the array are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

binarySearch

```
public static int binarySearch(double[] a,
               int fromIndex,
               int toIndex,
               double key)
```

Searches a range of the specified array of doubles for the specified value using the binary search algorithm. The range must be sorted (as by the `sort(double[], int, int)` method) prior to making this call. If it is not sorted, the results are undefined. If the range contains multiple elements with the specified value, there is no guarantee which one will be found. This method considers all NaN values to be equivalent and equal.

Parameters:

`a` - the array to be searched

`fromIndex` - the index of the first element (inclusive) to be searched

`toIndex` - the index of the last element (exclusive) to be searched

`key` - the value to be searched for

Returns:

index of the search key, if it is contained in the array within the specified range; otherwise, `(-(`*`insertion point`*`) - 1)`. The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element in the range greater than the key, or `toIndex` if all elements in the range are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0 or toIndex > a.length`

Since:

1.6

binarySearch

```
public static int binarySearch(float[] a,
               float key)
```

Searches the specified array of floats for the specified value using the binary search algorithm. The array must be sorted (as by the `sort(float[])` method) prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements with the specified value, there is no guarantee which one will be found. This method considers all NaN values to be equivalent and equal.

Parameters:

`a` - the array to be searched

`key` - the value to be searched for

Returns:

index of the search key, if it is contained in the array; otherwise, `(-(`*insertion point*`) - 1)`. The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element greater than the key, or `a.length` if all elements in the array are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

binarySearch

```
public static int binarySearch(float[] a,
               int fromIndex,
               int toIndex,
               float key)
```

Searches a range of the specified array of floats for the specified value using the binary search algorithm. The range must be sorted (as by the `sort(float[], int, int)` method) prior to making this call. If it is not sorted, the results are undefined. If the range contains multiple elements with the specified value, there is no guarantee which one will be found. This method considers all NaN values to be equivalent and equal.

Parameters:

`a` - the array to be searched

`fromIndex` - the index of the first element (inclusive) to be searched

`toIndex` - the index of the last element (exclusive) to be searched

`key` - the value to be searched for

Returns:

index of the search key, if it is contained in the array within the specified range; otherwise, `(-(`*insertion point*`) - 1)`. The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element in the range greater than the key, or `toIndex` if all elements in the range are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0 or toIndex > a.length`

Since:

1.6

binarySearch

```
public static int binarySearch(Object[] a,
               Object key)
```

Searches the specified array for the specified object using the binary search algorithm. The array must be sorted into ascending order according to the natural ordering of its elements (as by the `sort(Object[])` method) prior to making this call. If it is not sorted, the results are undefined. (If the array contains elements that are not mutually comparable (for example, strings and integers), it *cannot* be sorted according to the natural ordering of its elements, hence results are undefined.) If the array contains multiple elements equal to the specified object, there is no guarantee which one will be found.

Parameters:

a - the array to be searched

key - the value to be searched for

Returns:

index of the search key, if it is contained in the array; otherwise, (-(*insertion point*) - 1). The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element greater than the key, or a.length if all elements in the array are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

Throws:

ClassCastException - if the search key is not comparable to the elements of the array.

binarySearch

```
public static int binarySearch(Object[] a,
               int fromIndex,
               int toIndex,
               Object key)
```

Searches a range of the specified array for the specified object using the binary search algorithm. The range must be sorted into ascending order according to the natural ordering of its elements (as by the sort(Object[], int, int) method) prior to making this call. If it is not sorted, the results are undefined. (If the range contains elements that are not mutually comparable (for example, strings and integers), it *cannot* be sorted according to the natural ordering of its elements, hence results are undefined.) If the range contains multiple elements equal to the specified object, there is no guarantee which one will be found.

Parameters:

a - the array to be searched

fromIndex - the index of the first element (inclusive) to be searched

toIndex - the index of the last element (exclusive) to be searched

key - the value to be searched for

Returns:

index of the search key, if it is contained in the array within the specified range; otherwise, (-(*insertion point*) - 1). The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element in the range greater than the key, or toIndex if all elements in the range are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

Throws:

ClassCastException - if the search key is not comparable to the elements of the array within the specified range.

IllegalArgumentException - if fromIndex > toIndex

ArrayIndexOutOfBoundsException - if fromIndex < 0 or toIndex > a.length

Since:

1.6

binarySearch

```
public static <T> int binarySearch(T[] a,
                  T key,
                  Comparator<? super T> c)
```

Searches the specified array for the specified object using the binary search algorithm. The array must be sorted into ascending order according to the specified comparator (as by the sort(T[], Comparator) method) prior to making this

call. If it is not sorted, the results are undefined. If the array contains multiple elements equal to the specified object, there is no guarantee which one will be found.

Parameters:

`a` - the array to be searched

`key` - the value to be searched for

`c` - the comparator by which the array is ordered. A `null` value indicates that the elements' natural ordering should be used.

Returns:

index of the search key, if it is contained in the array; otherwise, `(-(`*`insertion point`*`) - 1)`. The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element greater than the key, or `a.length` if all elements in the array are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

Throws:

`ClassCastException` - if the array contains elements that are not *mutually comparable* using the specified comparator, or the search key is not comparable to the elements of the array using this comparator.

binarySearch

```
public static <T> int binarySearch(T[] a,
                   int fromIndex,
                   int toIndex,
                   T key,
                   Comparator<? super T> c)
```

Searches a range of the specified array for the specified object using the binary search algorithm. The range must be sorted into ascending order according to the specified comparator (as by the `sort(T[], int, int, Comparator)` method) prior to making this call. If it is not sorted, the results are undefined. If the range contains multiple elements equal to the specified object, there is no guarantee which one will be found.

Parameters:

`a` - the array to be searched

`fromIndex` - the index of the first element (inclusive) to be searched

`toIndex` - the index of the last element (exclusive) to be searched

`key` - the value to be searched for

`c` - the comparator by which the array is ordered. A `null` value indicates that the elements' natural ordering should be used.

Returns:

index of the search key, if it is contained in the array within the specified range; otherwise, `(-(`*`insertion point`*`) - 1)`. The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element in the range greater than the key, or `toIndex` if all elements in the range are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

Throws:

`ClassCastException` - if the range contains elements that are not *mutually comparable* using the specified comparator, or the search key is not comparable to the elements in the range using this comparator.

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0 or toIndex > a.length`

Since:

1.6

equals

```
public static boolean equals(long[] a,
             long[] a2)
```

Returns `true` if the two specified arrays of longs are *equal* to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. In other words, two arrays are equal if they contain the same elements in the same order. Also, two array references are considered equal if both are `null`.

Parameters:

`a` - one array to be tested for equality

`a2` - the other array to be tested for equality

Returns:

`true` if the two arrays are equal

equals

```
public static boolean equals(int[] a,
             int[] a2)
```

Returns `true` if the two specified arrays of ints are *equal* to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. In other words, two arrays are equal if they contain the same elements in the same order. Also, two array references are considered equal if both are `null`.

Parameters:

`a` - one array to be tested for equality

`a2` - the other array to be tested for equality

Returns:

`true` if the two arrays are equal

equals

```
public static boolean equals(short[] a,
             short[] a2)
```

Returns `true` if the two specified arrays of shorts are *equal* to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. In other words, two arrays are equal if they contain the same elements in the same order. Also, two array references are considered equal if both are `null`.

Parameters:

`a` - one array to be tested for equality

`a2` - the other array to be tested for equality

Returns:

`true` if the two arrays are equal

equals

```
public static boolean equals(char[] a,
             char[] a2)
```

Returns `true` if the two specified arrays of chars are *equal* to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. In other words, two arrays are equal if they contain the same elements in the same order. Also, two array references are considered equal if both are `null`.

Parameters:

`a` - one array to be tested for equality

`a2` - the other array to be tested for equality

Returns:

`true` if the two arrays are equal

equals

```
public static boolean equals(byte[] a,
             byte[] a2)
```

Returns `true` if the two specified arrays of bytes are *equal* to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. In other words, two arrays are equal if they contain the same elements in the same order. Also, two array references are considered equal if both are `null`.

Parameters:

`a` - one array to be tested for equality

`a2` - the other array to be tested for equality

Returns:

`true` if the two arrays are equal

equals

```
public static boolean equals(boolean[] a,
             boolean[] a2)
```

Returns `true` if the two specified arrays of booleans are *equal* to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. In other words, two arrays are equal if they contain the same elements in the same order. Also, two array references are considered equal if both are `null`.

Parameters:

`a` - one array to be tested for equality

`a2` - the other array to be tested for equality

Returns:

`true` if the two arrays are equal

equals

```
public static boolean equals(double[] a,
             double[] a2)
```

Returns `true` if the two specified arrays of doubles are *equal* to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. In other words, two arrays are equal if they contain the same elements in the same order. Also, two array references are considered equal if both are `null`.

Two doubles `d1` and `d2` are considered equal if:

```
new Double(d1).equals(new Double(d2))
```

(Unlike the == operator, this method considers `NaN` equals to itself, and 0.0d unequal to -0.0d.)

Parameters:

`a` - one array to be tested for equality

`a2` - the other array to be tested for equality

Returns:

`true` if the two arrays are equal

See Also:

`Double.equals(Object)`

equals

```
public static boolean equals(float[] a,
             float[] a2)
```

Returns `true` if the two specified arrays of floats are *equal* to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. In other words, two arrays are equal if they contain the same elements in the same order. Also, two array references are considered equal if both are `null`.

Two floats `f1` and `f2` are considered equal if:

```
new Float(f1).equals(new Float(f2))
```

(Unlike the == operator, this method considers `NaN` equals to itself, and 0.0f unequal to -0.0f.)

Parameters:

`a` - one array to be tested for equality

`a2` - the other array to be tested for equality

Returns:

`true` if the two arrays are equal

See Also:

`Float.equals(Object)`

equals

```
public static boolean equals(Object[] a,
             Object[] a2)
```

Returns `true` if the two specified arrays of Objects are *equal* to one another. The two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. Two objects `e1` and `e2` are considered *equal* if `(e1==null ? e2==null : e1.equals(e2))`. In other words, the two arrays are equal if they contain the same elements in the same order. Also, two array references are considered equal if both are `null`.

Parameters:

`a` - one array to be tested for equality

`a2` - the other array to be tested for equality

Returns:

`true` if the two arrays are equal

fill

```
public static void fill(long[] a,
        long val)
```

Assigns the specified long value to each element of the specified array of longs.

Parameters:

`a` - the array to be filled

`val` - the value to be stored in all elements of the array

fill

```
public static void fill(long[] a,
        int fromIndex,
        int toIndex,
        long val)
```

Assigns the specified long value to each element of the specified range of the specified array of longs. The range to be filled extends from index `fromIndex`, inclusive, to index `toIndex`, exclusive. (If `fromIndex==toIndex`, the range to be filled is empty.)

Parameters:

`a` - the array to be filled

`fromIndex` - the index of the first element (inclusive) to be filled with the specified value

`toIndex` - the index of the last element (exclusive) to be filled with the specified value

`val` - the value to be stored in all elements of the array

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0` or `toIndex > a.length`

fill

```
public static void fill(int[] a,
        int val)
```

Assigns the specified int value to each element of the specified array of ints.

Parameters:

`a` - the array to be filled

`val` - the value to be stored in all elements of the array

fill

```
public static void fill(int[] a,
        int fromIndex,
        int toIndex,
        int val)
```

Assigns the specified int value to each element of the specified range of the specified array of ints. The range to be filled extends from index `fromIndex`, inclusive, to index `toIndex`, exclusive. (If `fromIndex==toIndex`, the range to be filled is empty.)

Parameters:

`a` - the array to be filled

`fromIndex` - the index of the first element (inclusive) to be filled with the specified value

`toIndex` - the index of the last element (exclusive) to be filled with the specified value

`val` - the value to be stored in all elements of the array

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0` or `toIndex > a.length`

fill

```
public static void fill(short[] a,
        short val)
```

Assigns the specified short value to each element of the specified array of shorts.

Parameters:

`a` - the array to be filled

`val` - the value to be stored in all elements of the array

fill

```
public static void fill(short[] a,
        int fromIndex,
        int toIndex,
        short val)
```

Assigns the specified short value to each element of the specified range of the specified array of shorts. The range to be filled extends from index `fromIndex`, inclusive, to index `toIndex`, exclusive. (If `fromIndex==toIndex`, the range to be filled is empty.)

Parameters:

`a` - the array to be filled

`fromIndex` - the index of the first element (inclusive) to be filled with the specified value

`toIndex` - the index of the last element (exclusive) to be filled with the specified value

`val` - the value to be stored in all elements of the array

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0` or `toIndex > a.length`

fill

```
public static void fill(char[] a,
        char val)
```

Assigns the specified char value to each element of the specified array of chars.

Parameters:

`a` - the array to be filled

`val` - the value to be stored in all elements of the array

fill

```
public static void fill(char[] a,
        int fromIndex,
        int toIndex,
        char val)
```

Assigns the specified char value to each element of the specified range of the specified array of chars. The range to be filled extends from index `fromIndex`, inclusive, to index `toIndex`, exclusive. (If `fromIndex==toIndex`, the range to be filled is empty.)

Parameters:

`a` - the array to be filled

`fromIndex` - the index of the first element (inclusive) to be filled with the specified value

`toIndex` - the index of the last element (exclusive) to be filled with the specified value

`val` - the value to be stored in all elements of the array

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0` or `toIndex > a.length`

fill

```
public static void fill(byte[] a,
        byte val)
```

Assigns the specified byte value to each element of the specified array of bytes.

Parameters:

`a` - the array to be filled

`val` - the value to be stored in all elements of the array

fill

```
public static void fill(byte[] a,
        int fromIndex,
        int toIndex,
        byte val)
```

Assigns the specified byte value to each element of the specified range of the specified array of bytes. The range to be filled extends from index `fromIndex`, inclusive, to index `toIndex`, exclusive. (If `fromIndex==toIndex`, the range to be filled is empty.)

Parameters:

`a` - the array to be filled

`fromIndex` - the index of the first element (inclusive) to be filled with the specified value

`toIndex` - the index of the last element (exclusive) to be filled with the specified value

`val` - the value to be stored in all elements of the array

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0` or `toIndex > a.length`

fill

```
public static void fill(boolean[] a,
        boolean val)
```

Assigns the specified boolean value to each element of the specified array of booleans.

Parameters:

`a` - the array to be filled

`val` - the value to be stored in all elements of the array

fill

```
public static void fill(boolean[] a,
        int fromIndex,
        int toIndex,
        boolean val)
```

Assigns the specified boolean value to each element of the specified range of the specified array of booleans. The range to be filled extends from index `fromIndex`, inclusive, to index `toIndex`, exclusive. (If `fromIndex==toIndex`, the range to be filled is empty.)

Parameters:

`a` - the array to be filled

`fromIndex` - the index of the first element (inclusive) to be filled with the specified value

`toIndex` - the index of the last element (exclusive) to be filled with the specified value

`val` - the value to be stored in all elements of the array

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0` or `toIndex > a.length`

fill

```
public static void fill(double[] a,
        double val)
```

Assigns the specified double value to each element of the specified array of doubles.

Parameters:

`a` - the array to be filled

`val` - the value to be stored in all elements of the array

fill

```
public static void fill(double[] a,
        int fromIndex,
        int toIndex,
        double val)
```

Assigns the specified double value to each element of the specified range of the specified array of doubles. The range to be filled extends from index `fromIndex`, inclusive, to index `toIndex`, exclusive. (If `fromIndex==toIndex`, the range to be filled is empty.)

Parameters:

`a` - the array to be filled

`fromIndex` - the index of the first element (inclusive) to be filled with the specified value

`toIndex` - the index of the last element (exclusive) to be filled with the specified value

`val` - the value to be stored in all elements of the array

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0` or `toIndex > a.length`

fill

```
public static void fill(float[] a,
        float val)
```

Assigns the specified float value to each element of the specified array of floats.

Parameters:

`a` - the array to be filled

`val` - the value to be stored in all elements of the array

fill

```
public static void fill(float[] a,
        int fromIndex,
        int toIndex,
        float val)
```

Assigns the specified float value to each element of the specified range of the specified array of floats. The range to be filled extends from index `fromIndex`, inclusive, to index `toIndex`, exclusive. (If `fromIndex==toIndex`, the range to be filled is empty.)

Parameters:

`a` - the array to be filled

`fromIndex` - the index of the first element (inclusive) to be filled with the specified value

`toIndex` - the index of the last element (exclusive) to be filled with the specified value

`val` - the value to be stored in all elements of the array

Throws:

`IllegalArgumentException` - if `fromIndex > toIndex`

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0` or `toIndex > a.length`

fill

```
public static void fill(Object[] a,
        Object val)
```

Assigns the specified Object reference to each element of the specified array of Objects.

Parameters:

a - the array to be filled

val - the value to be stored in all elements of the array

Throws:

ArrayStoreException - if the specified value is not of a runtime type that can be stored in the specified array

fill

```
public static void fill(Object[] a,
        int fromIndex,
        int toIndex,
        Object val)
```

Assigns the specified Object reference to each element of the specified range of the specified array of Objects. The range to be filled extends from index fromIndex, inclusive, to index toIndex, exclusive. (If fromIndex==toIndex, the range to be filled is empty.)

Parameters:

a - the array to be filled

fromIndex - the index of the first element (inclusive) to be filled with the specified value

toIndex - the index of the last element (exclusive) to be filled with the specified value

val - the value to be stored in all elements of the array

Throws:

IllegalArgumentException - if fromIndex > toIndex

ArrayIndexOutOfBoundsException - if fromIndex < 0 or toIndex > a.length

ArrayStoreException - if the specified value is not of a runtime type that can be stored in the specified array

copyOf

```
public static <T> T[] copyOf(T[] original,
             int newLength)
```

Copies the specified array, truncating or padding with nulls (if necessary) so the copy has the specified length. For all indices that are valid in both the original array and the copy, the two arrays will contain identical values. For any indices that are valid in the copy but not the original, the copy will contain null. Such indices will exist if and only if the specified length is greater than that of the original array. The resulting array is of exactly the same class as the original array.

Parameters:

original - the array to be copied

newLength - the length of the copy to be returned

Returns:

a copy of the original array, truncated or padded with nulls to obtain the specified length

Throws:

NegativeArraySizeException - if newLength is negative

NullPointerException - if original is null

Since:

1.6

copyOf

```
public static <T,U> T[] copyOf(U[] original,
               int newLength,
               Class<? extends T[]> newType)
```

Copies the specified array, truncating or padding with nulls (if necessary) so the copy has the specified length. For all indices that are valid in both the original array and the copy, the two arrays will contain identical values. For any indices that are valid in the copy but not the original, the copy will contain null. Such indices will exist if and only if the specified length is greater than that of the original array. The resulting array is of the class newType.

Parameters:

original - the array to be copied

newLength - the length of the copy to be returned

newType - the class of the copy to be returned

Returns:

a copy of the original array, truncated or padded with nulls to obtain the specified length

Throws:

NegativeArraySizeException - if newLength is negative

NullPointerException - if original is null

ArrayStoreException - if an element copied from original is not of a runtime type that can be stored in an array of class newType

Since:

1.6

copyOf

```
public static byte[] copyOf(byte[] original,
            int newLength)
```

Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length. For all indices that are valid in both the original array and the copy, the two arrays will contain identical values. For any indices that are valid in the copy but not the original, the copy will contain (byte)0. Such indices will exist if and only if the specified length is greater than that of the original array.

Parameters:

original - the array to be copied

newLength - the length of the copy to be returned

Returns:

a copy of the original array, truncated or padded with zeros to obtain the specified length

Throws:

NegativeArraySizeException - if newLength is negative

NullPointerException - if original is null

Since:

1.6

copyOf

```
public static short[] copyOf(short[] original,
             int newLength)
```

Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length. For all indices that are valid in both the original array and the copy, the two arrays will contain identical values. For any indices that are valid in the copy but not the original, the copy will contain `(short)0`. Such indices will exist if and only if the specified length is greater than that of the original array.

Parameters:

`original` - the array to be copied

`newLength` - the length of the copy to be returned

Returns:

a copy of the original array, truncated or padded with zeros to obtain the specified length

Throws:

`NegativeArraySizeException` - if `newLength` is negative

`NullPointerException` - if `original` is null

Since:

1.6

copyOf

```
public static int[] copyOf(int[] original,
           int newLength)
```

Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length. For all indices that are valid in both the original array and the copy, the two arrays will contain identical values. For any indices that are valid in the copy but not the original, the copy will contain `0`. Such indices will exist if and only if the specified length is greater than that of the original array.

Parameters:

`original` - the array to be copied

`newLength` - the length of the copy to be returned

Returns:

a copy of the original array, truncated or padded with zeros to obtain the specified length

Throws:

`NegativeArraySizeException` - if `newLength` is negative

`NullPointerException` - if `original` is null

Since:

1.6

copyOf

```
public static long[] copyOf(long[] original,
            int newLength)
```

Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length. For all indices that are valid in both the original array and the copy, the two arrays will contain identical values. For any indices that are valid in the copy but not the original, the copy will contain `0L`. Such indices will exist if and only if the specified length is greater than that of the original array.

Parameters:

`original` - the array to be copied

`newLength` - the length of the copy to be returned

Returns:

a copy of the original array, truncated or padded with zeros to obtain the specified length

Throws:

`NegativeArraySizeException` - if `newLength` is negative

`NullPointerException` - if `original` is null

Since:

1.6

copyOf

```
public static char[] copyOf(char[] original,
            int newLength)
```

Copies the specified array, truncating or padding with null characters (if necessary) so the copy has the specified length. For all indices that are valid in both the original array and the copy, the two arrays will contain identical values. For any indices that are valid in the copy but not the original, the copy will contain `'\\u000'`. Such indices will exist if and only if the specified length is greater than that of the original array.

Parameters:

`original` - the array to be copied

`newLength` - the length of the copy to be returned

Returns:

a copy of the original array, truncated or padded with null characters to obtain the specified length

Throws:

`NegativeArraySizeException` - if `newLength` is negative

`NullPointerException` - if `original` is null

Since:

1.6

copyOf

```
public static float[] copyOf(float[] original,
             int newLength)
```

Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length. For all indices that are valid in both the original array and the copy, the two arrays will contain identical values. For any indices that are valid in the copy but not the original, the copy will contain `0f`. Such indices will exist if and only if the specified length is greater than that of the original array.

Parameters:

`original` - the array to be copied

`newLength` - the length of the copy to be returned

Returns:

a copy of the original array, truncated or padded with zeros to obtain the specified length

Throws:

`NegativeArraySizeException` - if `newLength` is negative

`NullPointerException` - if `original` is null

Since:

1.6

copyOf

```
public static double[] copyOf(double[] original,
              int newLength)
```

Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length. For all indices that are valid in both the original array and the copy, the two arrays will contain identical values. For any indices that are valid in the copy but not the original, the copy will contain `0d`. Such indices will exist if and only if the specified length is greater than that of the original array.

Parameters:

`original` - the array to be copied

`newLength` - the length of the copy to be returned

Returns:

a copy of the original array, truncated or padded with zeros to obtain the specified length

Throws:

`NegativeArraySizeException` - if `newLength` is negative

`NullPointerException` - if `original` is null

Since:

1.6

copyOf

```
public static boolean[] copyOf(boolean[] original,
               int newLength)
```

Copies the specified array, truncating or padding with `false` (if necessary) so the copy has the specified length. For all indices that are valid in both the original array and the copy, the two arrays will contain identical values. For any indices that are valid in the copy but not the original, the copy will contain `false`. Such indices will exist if and only if the specified length is greater than that of the original array.

Parameters:

`original` - the array to be copied

`newLength` - the length of the copy to be returned

Returns:

a copy of the original array, truncated or padded with false elements to obtain the specified length

Throws:

`NegativeArraySizeException` - if `newLength` is negative

`NullPointerException` - if `original` is null

Since:

1.6

copyOfRange

```
public static <T> T[] copyOfRange(T[] original,
                  int from,
                  int to)
```

Copies the specified range of the specified array into a new array. The initial index of the range (`from`) must lie between zero and `original.length`, inclusive. The value at `original[from]` is placed into the initial element of the copy (unless `from == original.length` or `from == to`). Values from subsequent elements in the original array are placed into subsequent elements in the copy. The final index of the range (`to`), which must be greater than or equal to `from`, may be greater than `original.length`, in which case `null` is placed in all elements of the copy whose index is greater than or equal to `original.length - from`. The length of the returned array will be `to - from`.

The resulting array is of exactly the same class as the original array.

Parameters:

`original` - the array from which a range is to be copied

`from` - the initial index of the range to be copied, inclusive

`to` - the final index of the range to be copied, exclusive. (This index may lie outside the array.)

Returns:

a new array containing the specified range from the original array, truncated or padded with nulls to obtain the required length

Throws:

`ArrayIndexOutOfBoundsException` - if `from < 0` or `from > original.length`

`IllegalArgumentException` - if `from > to`

`NullPointerException` - if `original` is null

Since:

1.6

copyOfRange

```
public static <T,U> T[] copyOfRange(U[] original,
                    int from,
                    int to,
                    Class<? extends T[]> newType)
```

Copies the specified range of the specified array into a new array. The initial index of the range (`from`) must lie between zero and `original.length`, inclusive. The value at `original[from]` is placed into the initial element of the copy (unless `from == original.length` or `from == to`). Values from subsequent elements in the original array are placed into subsequent elements in the copy. The final index of the range (`to`), which must be greater than or equal to `from`, may be greater than `original.length`, in which case `null` is placed in all elements of the copy whose index is greater than or equal to `original.length - from`. The length of the returned array will be `to - from`. The resulting array is of the class `newType`.

Parameters:

`original` - the array from which a range is to be copied

`from` - the initial index of the range to be copied, inclusive

`to` - the final index of the range to be copied, exclusive. (This index may lie outside the array.)

`newType` - the class of the copy to be returned

Returns:

a new array containing the specified range from the original array, truncated or padded with nulls to obtain the required length

Throws:

`ArrayIndexOutOfBoundsException` - if `from < 0` or `from > original.length`

`IllegalArgumentException` - if `from > to`

`NullPointerException` - if `original` is null

`ArrayStoreException` - if an element copied from `original` is not of a runtime type that can be stored in an array of class `newType`.

Since:

1.6

copyOfRange

```
public static byte[] copyOfRange(byte[] original,
                 int from,
                 int to)
```

Copies the specified range of the specified array into a new array. The initial index of the range (`from`) must lie between zero and `original.length`, inclusive. The value at `original[from]` is placed into the initial element of the copy (unless `from == original.length` or `from == to`). Values from subsequent elements in the original array are placed into subsequent elements in the copy. The final index of the range (`to`), which must be greater than or equal to `from`, may be greater than `original.length`, in which case `(byte)0` is placed in all elements of the copy whose index is greater than or equal to `original.length - from`. The length of the returned array will be `to - from`.

Parameters:

`original` - the array from which a range is to be copied

`from` - the initial index of the range to be copied, inclusive

`to` - the final index of the range to be copied, exclusive. (This index may lie outside the array.)

Returns:

a new array containing the specified range from the original array, truncated or padded with zeros to obtain the required length

Throws:

`ArrayIndexOutOfBoundsException` - if `from < 0` or `from > original.length`

`IllegalArgumentException` - if `from > to`

`NullPointerException` - if `original` is null

Since:

1.6

copyOfRange

```
public static short[] copyOfRange(short[] original,
                  int from,
                  int to)
```

Copies the specified range of the specified array into a new array. The initial index of the range (`from`) must lie between zero and `original.length`, inclusive. The value at `original[from]` is placed into the initial element of the copy (unless `from == original.length` or `from == to`). Values from subsequent elements in the original array are placed into subsequent elements in the copy. The final index of the range (`to`), which must be greater than or equal to `from`, may be greater than `original.length`, in which case `(short)0` is placed in all elements of the copy whose index is greater than or equal to `original.length - from`. The length of the returned array will be `to - from`.

Parameters:

`original` - the array from which a range is to be copied

`from` - the initial index of the range to be copied, inclusive

`to` - the final index of the range to be copied, exclusive. (This index may lie outside the array.)

Returns:

a new array containing the specified range from the original array, truncated or padded with zeros to obtain the required length

Throws:

`ArrayIndexOutOfBoundsException` - if `from < 0` or `from > original.length`

`IllegalArgumentException` - if `from > to`

`NullPointerException` - if `original` is null

Since:

1.6

copyOfRange

```
public static int[] copyOfRange(int[] original,
                int from,
                int to)
```

Copies the specified range of the specified array into a new array. The initial index of the range (`from`) must lie between zero and `original.length`, inclusive. The value at `original[from]` is placed into the initial element of the copy (unless `from == original.length` or `from == to`). Values from subsequent elements in the original array are placed into subsequent elements in the copy. The final index of the range (`to`), which must be greater than or equal to `from`, may be greater than `original.length`, in which case `0` is placed in all elements of the copy whose index is greater than or equal to `original.length - from`. The length of the returned array will be `to - from`.

Parameters:

`original` - the array from which a range is to be copied

`from` - the initial index of the range to be copied, inclusive

`to` - the final index of the range to be copied, exclusive. (This index may lie outside the array.)

Returns:

a new array containing the specified range from the original array, truncated or padded with zeros to obtain the required length

Throws:

`ArrayIndexOutOfBoundsException` - if `from < 0` or `from > original.length`

`IllegalArgumentException` - if `from > to`

`NullPointerException` - if `original` is null

Since:

1.6

copyOfRange

```
public static long[] copyOfRange(long[] original,
                 int from,
                 int to)
```

Copies the specified range of the specified array into a new array. The initial index of the range (`from`) must lie between zero and `original.length`, inclusive. The value at `original[from]` is placed into the initial element of the copy (unless `from == original.length` or `from == to`). Values from subsequent elements in the original array are placed into subsequent elements in the copy. The final index of the range (`to`), which must be greater than or equal to `from`, may be greater than `original.length`, in which case `0L` is placed in all elements of the copy whose index is greater than or equal to `original.length - from`. The length of the returned array will be `to - from`.

Parameters:

`original` - the array from which a range is to be copied

`from` - the initial index of the range to be copied, inclusive

`to` - the final index of the range to be copied, exclusive. (This index may lie outside the array.)

Returns:

a new array containing the specified range from the original array, truncated or padded with zeros to obtain the required length

Throws:

`ArrayIndexOutOfBoundsException` - if `from < 0` or `from > original.length`

`IllegalArgumentException` - if `from > to`

`NullPointerException` - if `original` is null

Since:

1.6

copyOfRange

```
public static char[] copyOfRange(char[] original,
                 int from,
                 int to)
```

Copies the specified range of the specified array into a new array. The initial index of the range (`from`) must lie between zero and `original.length`, inclusive. The value at `original[from]` is placed into the initial element of the copy (unless `from == original.length` or `from == to`). Values from subsequent elements in the original array are placed into subsequent elements in the copy. The final index of the range (`to`), which must be greater than or equal to `from`, may be greater than `original.length`, in which case `'\\u000'` is placed in all elements of the copy whose index is greater than or equal to `original.length - from`. The length of the returned array will be `to - from`.

Parameters:

`original` - the array from which a range is to be copied

`from` - the initial index of the range to be copied, inclusive

`to` - the final index of the range to be copied, exclusive. (This index may lie outside the array.)

Returns:

a new array containing the specified range from the original array, truncated or padded with null characters to obtain the required length

Throws:

`ArrayIndexOutOfBoundsException` - if `from < 0` or `from > original.length`

`IllegalArgumentException` - if `from > to`

`NullPointerException` - if `original` is null

Since:

1.6

copyOfRange

```
public static float[] copyOfRange(float[] original,
                  int from,
                  int to)
```

Copies the specified range of the specified array into a new array. The initial index of the range (`from`) must lie between zero and `original.length`, inclusive. The value at `original[from]` is placed into the initial element of the copy (unless `from == original.length` or `from == to`). Values from subsequent elements in the original array are placed into subsequent elements in the copy. The final index of the range (`to`), which must be greater than or equal to `from`, may be greater than `original.length`, in which case `0f` is placed in all elements of the copy whose index is greater than or equal to `original.length - from`. The length of the returned array will be `to - from`.

Parameters:

`original` - the array from which a range is to be copied

`from` - the initial index of the range to be copied, inclusive

`to` - the final index of the range to be copied, exclusive. (This index may lie outside the array.)

Returns:

a new array containing the specified range from the original array, truncated or padded with zeros to obtain the required length

Throws:

`ArrayIndexOutOfBoundsException` - if `from < 0` or `from > original.length`

`IllegalArgumentException` - if `from > to`

`NullPointerException` - if `original` is null

Since:

1.6

copyOfRange

```
public static double[] copyOfRange(double[] original,
                   int from,
                   int to)
```

Copies the specified range of the specified array into a new array. The initial index of the range (`from`) must lie between zero and `original.length`, inclusive. The value at `original[from]` is placed into the initial element of the copy (unless `from == original.length` or `from == to`). Values from subsequent elements in the original array are placed into subsequent elements in the copy. The final index of the range (`to`), which must be greater than or equal to `from`, may be greater than `original.length`, in which case `0d` is placed in all elements of the copy whose index is greater than or equal to `original.length - from`. The length of the returned array will be `to - from`.

Parameters:

`original` - the array from which a range is to be copied

`from` - the initial index of the range to be copied, inclusive

`to` - the final index of the range to be copied, exclusive. (This index may lie outside the array.)

Returns:

a new array containing the specified range from the original array, truncated or padded with zeros to obtain the required length

Throws:

`ArrayIndexOutOfBoundsException` - if `from < 0` or `from > original.length`

`IllegalArgumentException` - if `from > to`

`NullPointerException` - if `original` is null

Since:

1.6

copyOfRange

```
public static boolean[] copyOfRange(boolean[] original,
                    int from,
                    int to)
```

Copies the specified range of the specified array into a new array. The initial index of the range (`from`) must lie between zero and `original.length`, inclusive. The value at `original[from]` is placed into the initial element of the copy (unless `from == original.length` or `from == to`). Values from subsequent elements in the original array are placed into subsequent elements in the copy. The final index of the range (`to`), which must be greater than or equal to `from`, may be greater than `original.length`, in which case `false` is placed in all elements of the copy whose index is greater than or equal to `original.length - from`. The length of the returned array will be `to - from`.

Parameters:

`original` - the array from which a range is to be copied

`from` - the initial index of the range to be copied, inclusive

`to` - the final index of the range to be copied, exclusive. (This index may lie outside the array.)

Returns:

a new array containing the specified range from the original array, truncated or padded with false elements to obtain the required length

Throws:

`ArrayIndexOutOfBoundsException` - if `from < 0` or `from > original.length`

`IllegalArgumentException` - if `from > to`

`NullPointerException` - if `original` is null

Since:

1.6

asList

```
@SafeVarargs
public static <T> List<T> asList(T... a)
```

Returns a fixed-size list backed by the specified array. (Changes to the returned list "write through" to the array.) This method acts as bridge between array-based and collection-based APIs, in combination with `Collection.toArray()`. The returned list is serializable and implements `RandomAccess`.

This method also provides a convenient way to create a fixed-size list initialized to contain several elements:

```
List<String> stooges = Arrays.asList("Larry", "Moe", "Curly");
```

Parameters:

`a` - the array by which the list will be backed

Returns:

a list view of the specified array

hashCode

```
public static int hashCode(long[] a)
```

Returns a hash code based on the contents of the specified array. For any two `long` arrays `a` and `b` such that `Arrays.equals(a, b)`, it is also the case that `Arrays.hashCode(a) == Arrays.hashCode(b)`.

The value returned by this method is the same value that would be obtained by invoking the `hashCode` method on a `List` containing a sequence of `Long` instances representing the elements of `a` in the same order. If `a` is `null`, this method returns 0.

Parameters:

`a` - the array whose hash value to compute

Returns:

a content-based hash code for `a`

Since:

1.5

hashCode

```
public static int hashCode(int[] a)
```

Returns a hash code based on the contents of the specified array. For any two non-null `int` arrays `a` and `b` such that `Arrays.equals(a, b)`, it is also the case that `Arrays.hashCode(a) == Arrays.hashCode(b)`.

The value returned by this method is the same value that would be obtained by invoking the `hashCode` method on a `List` containing a sequence of `Integer` instances representing the elements of `a` in the same order. If `a` is `null`, this method returns 0.

Parameters:

`a` - the array whose hash value to compute

Returns:

a content-based hash code for `a`

Since:

1.5

hashCode

```
public static int hashCode(short[] a)
```

Returns a hash code based on the contents of the specified array. For any two `short` arrays `a` and `b` such that `Arrays.equals(a, b)`, it is also the case that `Arrays.hashCode(a) == Arrays.hashCode(b)`.

The value returned by this method is the same value that would be obtained by invoking the `hashCode` method on a `List` containing a sequence of `Short` instances representing the elements of `a` in the same order. If `a` is `null`, this method returns 0.

Parameters:

`a` - the array whose hash value to compute

Returns:

a content-based hash code for a

Since:

1.5

hashCode

```
public static int hashCode(char[] a)
```

Returns a hash code based on the contents of the specified array. For any two char arrays a and b such that Arrays.equals(a, b), it is also the case that Arrays.hashCode(a) == Arrays.hashCode(b).

The value returned by this method is the same value that would be obtained by invoking the hashCode method on a List containing a sequence of Character instances representing the elements of a in the same order. If a is null, this method returns 0.

Parameters:

a - the array whose hash value to compute

Returns:

a content-based hash code for a

Since:

1.5

hashCode

```
public static int hashCode(byte[] a)
```

Returns a hash code based on the contents of the specified array. For any two byte arrays a and b such that Arrays.equals(a, b), it is also the case that Arrays.hashCode(a) == Arrays.hashCode(b).

The value returned by this method is the same value that would be obtained by invoking the hashCode method on a List containing a sequence of Byte instances representing the elements of a in the same order. If a is null, this method returns 0.

Parameters:

a - the array whose hash value to compute

Returns:

a content-based hash code for a

Since:

1.5

hashCode

```
public static int hashCode(boolean[] a)
```

Returns a hash code based on the contents of the specified array. For any two boolean arrays a and b such that Arrays.equals(a, b), it is also the case that Arrays.hashCode(a) == Arrays.hashCode(b).

The value returned by this method is the same value that would be obtained by invoking the hashCode method on a List containing a sequence of Boolean instances representing the elements of a in the same order. If a is null, this method returns 0.

Parameters:

a - the array whose hash value to compute

Returns:

a content-based hash code for a

Since:

1.5

hashCode

```
public static int hashCode(float[] a)
```

Returns a hash code based on the contents of the specified array. For any two `float` arrays a and b such that `Arrays.equals(a, b)`, it is also the case that `Arrays.hashCode(a) == Arrays.hashCode(b)`.

The value returned by this method is the same value that would be obtained by invoking the `hashCode` method on a `List` containing a sequence of `Float` instances representing the elements of a in the same order. If a is `null`, this method returns 0.

Parameters:

a - the array whose hash value to compute

Returns:

a content-based hash code for a

Since:

1.5

hashCode

```
public static int hashCode(double[] a)
```

Returns a hash code based on the contents of the specified array. For any two `double` arrays a and b such that `Arrays.equals(a, b)`, it is also the case that `Arrays.hashCode(a) == Arrays.hashCode(b)`.

The value returned by this method is the same value that would be obtained by invoking the `hashCode` method on a `List` containing a sequence of `Double` instances representing the elements of a in the same order. If a is `null`, this method returns 0.

Parameters:

a - the array whose hash value to compute

Returns:

a content-based hash code for a

Since:

1.5

hashCode

```
public static int hashCode(Object[] a)
```

Returns a hash code based on the contents of the specified array. If the array contains other arrays as elements, the hash code is based on their identities rather than their contents. It is therefore acceptable to invoke this method on an array that contains itself as an element, either directly or indirectly through one or more levels of arrays.

For any two arrays `a` and `b` such that `Arrays.equals(a, b)`, it is also the case that `Arrays.hashCode(a) == Arrays.hashCode(b)`.

The value returned by this method is equal to the value that would be returned by `Arrays.asList(a).hashCode()`, unless `a` is `null`, in which case `0` is returned.

Parameters:

`a` - the array whose content-based hash code to compute

Returns:

a content-based hash code for `a`

Since:

1.5

See Also:

`deepHashCode(Object[])`

deepHashCode

```
public static int deepHashCode(Object[] a)
```

Returns a hash code based on the "deep contents" of the specified array. If the array contains other arrays as elements, the hash code is based on their contents and so on, ad infinitum. It is therefore unacceptable to invoke this method on an array that contains itself as an element, either directly or indirectly through one or more levels of arrays. The behavior of such an invocation is undefined.

For any two arrays `a` and `b` such that `Arrays.deepEquals(a, b)`, it is also the case that `Arrays.deepHashCode(a) == Arrays.deepHashCode(b)`.

The computation of the value returned by this method is similar to that of the value returned by `List.hashCode()` on a list containing the same elements as `a` in the same order, with one difference: If an element `e` of `a` is itself an array, its hash code is computed not by calling `e.hashCode()`, but as by calling the appropriate overloading of `Arrays.hashCode(e)` if `e` is an array of a primitive type, or as by calling `Arrays.deepHashCode(e)` recursively if `e` is an array of a reference type. If `a` is `null`, this method returns 0.

Parameters:

`a` - the array whose deep-content-based hash code to compute

Returns:

a deep-content-based hash code for `a`

Since:

1.5

See Also:

`hashCode(Object[])`

deepEquals

```
public static boolean deepEquals(Object[] a1,
                 Object[] a2)
```

Returns `true` if the two specified arrays are *deeply equal* to one another. Unlike the `equals(Object[],Object[])` method, this method is appropriate for use with nested arrays of arbitrary depth.

Two array references are considered deeply equal if both are `null`, or if they refer to arrays that contain the same number of elements and all corresponding pairs of elements in the two arrays are deeply equal.

Two possibly `null` elements `e1` and `e2` are deeply equal if any of the following conditions hold:

- e1 and e2 are both arrays of object reference types, and Arrays.deepEquals(e1, e2) would return true
- e1 and e2 are arrays of the same primitive type, and the appropriate overloading of Arrays.equals(e1, e2) would return true.
- e1 == e2
- e1.equals(e2) would return true.

Note that this definition permits null elements at any depth.

If either of the specified arrays contain themselves as elements either directly or indirectly through one or more levels of arrays, the behavior of this method is undefined.

Parameters:

a1 - one array to be tested for equality

a2 - the other array to be tested for equality

Returns:

true if the two arrays are equal

Since:

1.5

See Also:

equals(Object[],Object[]), Objects.deepEquals(Object, Object)

toString

```
public static String toString(long[] a)
```

Returns a string representation of the contents of the specified array. The string representation consists of a list of the array's elements, enclosed in square brackets ("[]"). Adjacent elements are separated by the characters ", " (a comma followed by a space). Elements are converted to strings as by String.valueOf(long). Returns "null" if a is null.

Parameters:

a - the array whose string representation to return

Returns:

a string representation of a

Since:

1.5

toString

```
public static String toString(int[] a)
```

Returns a string representation of the contents of the specified array. The string representation consists of a list of the array's elements, enclosed in square brackets ("[]"). Adjacent elements are separated by the characters ", " (a comma followed by a space). Elements are converted to strings as by String.valueOf(int). Returns "null" if a is null.

Parameters:

a - the array whose string representation to return

Returns:

a string representation of a

Since:

1.5

toString

```
public static String toString(short[] a)
```

Returns a string representation of the contents of the specified array. The string representation consists of a list of the array's elements, enclosed in square brackets (`"[]"`). Adjacent elements are separated by the characters `", "` (a comma followed by a space). Elements are converted to strings as by `String.valueOf(short)`. Returns `"null"` if `a` is `null`.

Parameters:

`a` - the array whose string representation to return

Returns:

a string representation of `a`

Since:

1.5

toString

```
public static String toString(char[] a)
```

Returns a string representation of the contents of the specified array. The string representation consists of a list of the array's elements, enclosed in square brackets (`"[]"`). Adjacent elements are separated by the characters `", "` (a comma followed by a space). Elements are converted to strings as by `String.valueOf(char)`. Returns `"null"` if `a` is `null`.

Parameters:

`a` - the array whose string representation to return

Returns:

a string representation of `a`

Since:

1.5

toString

```
public static String toString(byte[] a)
```

Returns a string representation of the contents of the specified array. The string representation consists of a list of the array's elements, enclosed in square brackets (`"[]"`). Adjacent elements are separated by the characters `", "` (a comma followed by a space). Elements are converted to strings as by `String.valueOf(byte)`. Returns `"null"` if `a` is `null`.

Parameters:

`a` - the array whose string representation to return

Returns:

a string representation of `a`

Since:

1.5

toString

```
public static String toString(boolean[] a)
```

Returns a string representation of the contents of the specified array. The string representation consists of a list of the array's elements, enclosed in square brackets ("[]"). Adjacent elements are separated by the characters ", " (a comma followed by a space). Elements are converted to strings as by `String.valueOf(boolean)`. Returns "null" if a is null.

Parameters:

a - the array whose string representation to return

Returns:

a string representation of a

Since:

1.5

toString

```
public static String toString(float[] a)
```

Returns a string representation of the contents of the specified array. The string representation consists of a list of the array's elements, enclosed in square brackets ("[]"). Adjacent elements are separated by the characters ", " (a comma followed by a space). Elements are converted to strings as by `String.valueOf(float)`. Returns "null" if a is null.

Parameters:

a - the array whose string representation to return

Returns:

a string representation of a

Since:

1.5

toString

```
public static String toString(double[] a)
```

Returns a string representation of the contents of the specified array. The string representation consists of a list of the array's elements, enclosed in square brackets ("[]"). Adjacent elements are separated by the characters ", " (a comma followed by a space). Elements are converted to strings as by `String.valueOf(double)`. Returns "null" if a is null.

Parameters:

a - the array whose string representation to return

Returns:

a string representation of a

Since:

1.5

toString

```
public static String toString(Object[] a)
```

Returns a string representation of the contents of the specified array. If the array contains other arrays as elements, they are converted to strings by the `Object.toString()` method inherited from `Object`, which describes their *identities* rather than their contents.

The value returned by this method is equal to the value that would be returned by `Arrays.asList(a).toString()`, unless `a` is `null`, in which case `"null"` is returned.

Parameters:

`a` - the array whose string representation to return

Returns:

a string representation of `a`

Since:

1.5

See Also:

`deepToString(Object[])`

deepToString

```
public static String deepToString(Object[] a)
```

Returns a string representation of the "deep contents" of the specified array. If the array contains other arrays as elements, the string representation contains their contents and so on. This method is designed for converting multidimensional arrays to strings.

The string representation consists of a list of the array's elements, enclosed in square brackets (`"[]"`). Adjacent elements are separated by the characters `", "` (a comma followed by a space). Elements are converted to strings as by `String.valueOf(Object)`, unless they are themselves arrays.

If an element `e` is an array of a primitive type, it is converted to a string as by invoking the appropriate overloading of `Arrays.toString(e)`. If an element `e` is an array of a reference type, it is converted to a string as by invoking this method recursively.

To avoid infinite recursion, if the specified array contains itself as an element, or contains an indirect reference to itself through one or more levels of arrays, the self-reference is converted to the string `"[...]"`. For example, an array containing only a reference to itself would be rendered as `"[[...]]"`.

This method returns `"null"` if the specified array is `null`.

Parameters:

`a` - the array whose string representation to return

Returns:

a string representation of `a`

Since:

1.5

See Also:

`toString(Object[])`

detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.