Primitives

Overview

The *primitive* types of Java are the basic types:

- `byte`
- `short`
- `int`
- `long`
- `float`
- `double`
- `char`
- `boolean`

Before searching Guava for a method, you should check if it is in `Arrays` or the corresponding JDK wrapper type, e.g. `Integer`.

These types cannot be used as objects or as type parameters to generic types, which means that many general-purpose utilities cannot be applied to them. Guava provides a number of these general-purpose utilities, ways of interfacing between primitive arrays and collection APIs, conversion from types to byte array representations, and support for unsigned behaviors on certain types.

Primitive Type	Guava Utilities (all in `com.google.common.primitives`)
`byte`	`Bytes`, `SignedBytes`, `UnsignedBytes`
`short`	`Shorts`
`int`	`Ints`, `UnsignedInteger`, `UnsignedInts`
`long`	`Longs`, `UnsignedLong`, `UnsignedLongs`
`float`	`Floats`
`double`	`Doubles`
`char`	`Chars`
`boolean`	`Booleans`

Methods that differ in behavior for signed and unsigned bytes are completely skipped in `Bytes`, but only present in the `SignedBytes` and `UnsignedBytes` utilities, since the signedness of bytes is somewhat more ambiguous than the signedness of other types.

Unsigned variants of methods on `int` and `long` are provided in the `UnsignedInts` and `UnsignedLongs` classes, but since most uses of those types are signed, the `Ints` and `Longs` classes treat their inputs as signed.

Additionally, Guava provides "wrapper types" for unsigned `int` and `long` values, `UnsignedInteger` and `UnsignedLong`, to help you use the type system to enforce distinctions between signed and unsigned values, in exchange for a small

performance cost. These classes directly support simple arithmetic operations in the style of `BigInteger`.

All method signatures use `Wrapper` to refer to the corresponding JDK wrapper type, and `prim` to refer to the primitive type. (`Prims`, where applicable, refers to the corresponding Guava utilities class.)

Primitive array utilities

Primitive arrays are the most efficient way (in both memory and performance) to work with primitive types in aggregate. Guava provides a variety of utilities to work with these methods.

Signature	Description	Collection analogue	Availability
`List<Wrapper> asList(prim... backingArray)`	Wraps a primitive array as a `List` of the corresponding wrapper type.	`Arrays.asList`	Sign-independent*
`prim[] toArray(Collection<Wrapper> collection)`	Copies a collection into a new `prim[]`. This method is as thread-safe as `collection.toArray()`.	`Collection.toArray()`	Sign-independent
`prim[] concat(prim[]... arrays)`	Concatenate several primitive arrays.	`Iterables.concat`	Sign-independent
`boolean contains(prim[] array, prim target)`	Determines if the specified element is in the specified array.	`Collection.contains`	Sign-independent
`int indexOf(prim[] array, prim target)`	Finds the index of the first appearance of the value `target` in `array`, or returns `-1` if no such value exists.	`List.indexOf`	Sign-independent
`int lastIndexOf(prim[] array, prim target)`	Finds the index of the last appearance of the value `target` in `array`, or returns `-1` if no such value exists.	`List.lastIndexOf`	Sign-independent
`prim min(prim... array)`	Returns the minimum *element* of the array.	`Collections.min`	Sign-dependent**
`prim max(prim... array)`	Returns the maximum *element* of the array.	`Collections.max`	Sign-dependent

Signature	Description	Collection analogue	Availability
`String join(String separator, prim... array)`	Constructs a string containing the elements of `array`, separated by `separator`.	`Joiner.on(separator).join`	Sign-independent
`Comparator<prim[]> lexicographicalComparator()`	A comparator which compares `prim` arrays lexicographically.	`Ordering.natural().lexicographical()`	Sign-dependent

* Sign-independent methods are present in: `Bytes`, `Shorts`, `Ints`, `Longs`, `Floats`, `Doubles`, `Chars`, `Booleans`. *Not* `UnsignedInts`, `UnsignedLongs`, `SignedBytes`, or `UnsignedBytes`.

** Sign-dependent methods are present in: `SignedBytes`, `UnsignedBytes`, `Shorts`, `Ints`, `Longs`, `Floats`, `Doubles`, `Chars`, `Booleans`, `UnsignedInts`, `UnsignedLongs`. *Not* `Bytes`.

General utility methods

Guava provides a number of basic utilities which were not part of JDK 6. Some of these methods, however, are available in JDK 7.

Signature	Description	Availability
`int compare(prim a, prim b)`	A traditional `Comparator.compare` method, but on the primitive types. *Provided in the JDK wrapper classes as of JDK 7.*	Sign-dependent
`prim checkedCast(long value)`	Casts the specified value to `prim`, *unless* the specified value does not fit into a `prim`, in which case an `IllegalArgumentException` is thrown.	Sign-dependent for integral types only*
`prim saturatedCast(long value)`	Casts the specified value to `prim`, unless the specified value does not fit into a `prim`, in which case the closest `prim` value is used.	Sign-dependent for integral types only

*Here, integral types include `byte`, `short`, `int`, `long`. Integral types do *not* include `char`, `boolean`, `float`, or `double`.

Note: Rounding from `double` is provided in `com.google.common.math.DoubleMath`, and supports a variety of rounding modes. See the article for details.

Byte conversion methods

Guava provides methods to convert primitive types to and from byte array representations **in big-endian order**. All methods are sign-independent, except that `Booleans` provides none of these methods.

Signature	Description
`int BYTES`	Constant representing the number of bytes needed to represent a `prim` value.
`prim fromByteArray(byte[] bytes)`	Returns the `prim` value whose big-endian representation is the first `Prims.BYTES` bytes in the array `bytes`. Throws an `IllegalArgumentException` if `bytes.length <= Prims.BYTES`.
`prim fromBytes(byte b1, ..., byte bk)`	Takes `Prims.BYTES` byte arguments. Returns the `prim` value whose byte representation is the specified bytes in big-endian order.
`byte[] toByteArray(prim value)`	Returns an array containing the big-endian byte representation of `value`.

Unsigned support

The `UnsignedInts` and `UnsignedLongs` utility classes provide some of the generic utilities that Java provides for signed types in their wrapper classes. `UnsignedInts` and `UnsignedLongs` deal with the primitive type directly: it is up to you to make sure that only unsigned values are passed to these utilities.

Additionally, for `int` and `long`, Guava provides "unsigned" wrapper types (`UnsignedInteger` and `UnsignedLong` to help you enforce distinctions between unsigned and signed values in the type system, in exchange for a small performance penalty.

Generic utilities

These methods' signed analogues are provided in the wrapper classes in the JDK.

Signature	Explanation
`int UnsignedInts.parseUnsignedInt(String)` `long UnsignedLongs.parseUnsignedLong(String)`	Parses an unsigned value from a string in base 10.
`int UnsignedInts.parseUnsignedInt(String string, int radix)` `long UnsignedLongs.parseUnsignedLong(String string, int radix)`	Parses an unsigned value from a string in the specified base.
`String UnsignedInts.toString(int)` `String UnsignedLongs.toString(long)`	Returns a string representation of the unsigned value in base 10.
`String UnsignedInts.toString(int value, int radix)` `String UnsignedLongs.toString(long value, int radix)`	Returns a string representation of the unsigned value in the specified base.

Wrapper

The provided unsigned wrapper types include a number of methods to make their use and conversion easier.

Signature	Explanation
`UnsignedPrim plus(UnsignedPrim)`, `minus`, `times`, `dividedBy`, `mod`	Simple arithmetic operations.
`UnsignedPrim valueOf(BigInteger)`	Returns the value from a `BigInteger` as an `UnsignedPrim`, or throw an `IAE` if the specified `BigInteger` is negative or does not fit.
`UnsignedPrim valueOf(long)`	Returns the value from the `long` as an `UnsignedPrim`, or throw an `IAE` if the specified `long` is negative or does not fit.
`UnsignedPrim fromPrimBits(prim value)`	View the given value as unsigned. For example, `UnsignedInteger.fromIntBits(1 << 31)` has the value 231, even though `1 << 31` is negative as an `int`.
`BigInteger bigIntegerValue()`	Get the value of this `UnsignedPrim` as a `BigInteger`.
`toString()`, `toString(int radix)`	Returns a string representation of this unsigned value.