public class java.math

BigInteger

Extends: Number
Implements: Comparable

Immutable arbitrary-precision integers. All operations behave as if BigIntegers were represented in two's-complement notation (like Java's primitive integer types). BigInteger provides analogues to all of Java's primitive integer operators, and all relevant methods from java.lang.Math. Additionally, BigInteger provides operations for modular arithmetic, GCD calculation, primality testing, prime generation, bit manipulation, and a few other miscellaneous operations.

Semantics of arithmetic operations exactly mimic those of Java's integer arithmetic operators, as defined in *The Java Language Specification*. For example, division by zero throws an `ArithmeticException`, and division of a negative by a positive yields a negative (or zero) remainder. All of the details in the Spec concerning overflow are ignored, as BigIntegers are made as large as necessary to accommodate the results of an operation.

Semantics of shift operations extend those of Java's shift operators to allow for negative shift distances. A right-shift with a negative shift distance results in a left shift, and vice-versa. The unsigned right shift operator (>>>) is omitted, as this operation makes little sense in combination with the "infinite word size" abstraction provided by this class.

Semantics of bitwise logical operations exactly mimic those of Java's bitwise integer operators. The binary operators (`and`, `or`, `xor`) implicitly perform sign extension on the shorter of the two operands prior to performing the operation.

Comparison operations perform signed integer comparisons, analogous to those performed by Java's relational and equality operators.

Modular arithmetic operations are provided to compute residues, perform exponentiation, and compute multiplicative inverses. These methods always return a non-negative result, between `0` and `(modulus - 1)`, inclusive.

Bit operations operate on a single bit of the two's-complement representation of their operand. If necessary, the operand is sign-extended so that it contains the designated bit. None of the single-bit

operations can produce a BigInteger with a different sign from the BigInteger being operated on, as they affect only a single bit, and the "infinite word size" abstraction provided by this class ensures that there are infinitely many "virtual sign bits" preceding each BigInteger.

For the sake of brevity and clarity, pseudo-code is used throughout the descriptions of BigInteger methods. The pseudo-code expression `(i + j)` is shorthand for "a BigInteger whose value is that of the BigInteger `i` plus that of the BigInteger `j`." The pseudo-code expression `(i == j)` is shorthand for "`true` if and only if the BigInteger `i` represents the same value as the BigInteger `j`." Other pseudo-code expressions are interpreted similarly.

All methods and constructors in this class throw `NullPointerException` when passed a null object reference for any input parameter.

version 1.75, 06/28/06
since JDK1.1
See also java.math.BigDecimal

Fields

final public static BigInteger	ZERO The BigInteger constant zero. **since** 1.2
final public static BigInteger	ONE The BigInteger constant one. **since** 1.2
final public static BigInteger	TEN The BigInteger constant ten. **since** 1.5

Constructors

public	**BigInteger**(byte[] val) Translates a byte array containing the two's-complement binary representation of a BigInteger into a BigInteger. The input array is assumed to be in *big-endian* byte-order: the most significant byte is in the zeroth element. *val* big-endian two's-complement binary representation of BigInteger. **Throws** NumberFormatException: `val` is zero bytes long.
public	**BigInteger**(int signum, byte[] magnitude) Translates the sign-magnitude representation of a BigInteger into a BigInteger. The sign is represented as an integer signum

	value: -1 for negative, 0 for zero, or 1 for positive. The magnitude is a byte array in *big-endian* byte-order: the most significant byte is in the zeroth element. A zero-length magnitude array is permissible, and will result inin a BigInteger value of 0, whether signum is -1, 0 or 1. *signum* signum of the number (-1 for negative, 0 for zero, 1 for positive). *magnitude* big-endian binary representation of the magnitude of the number. **Throws** NumberFormatException: `signum` is not one of the three legal values (-1, 0, and 1), or `signum` is 0 and `magnitude` contains one or more non-zero bytes.
public	**BigInteger**(String val, int radix) Translates the String representation of a BigInteger in the specified radix into a BigInteger. The String representation consists of an optional minus sign followed by a sequence of one or more digits in the specified radix. The character-to-digit mapping is provided by `Character.digit`. The String may not contain any extraneous characters (whitespace, for example). *val* String representation of BigInteger. *radix* radix to be used in interpreting `val`. **Throws** NumberFormatException: `val` is not a valid representation of a BigInteger in the specified radix, or `radix` is outside the range from `Character#MIN_RADIX` to `Character#MAX_RADIX`, inclusive. **See also** digit
public	**BigInteger**(String val) Translates the decimal String representation of a BigInteger into a BigInteger. The String representation consists of an optional minus sign followed by a sequence of one or more decimal digits. The character-to-digit mapping is provided by `Character.digit`. The String may not contain any extraneous characters (whitespace, for example). *val* decimal String representation of BigInteger. **Throws** NumberFormatException: `val` is not a valid representation of a BigInteger. **See also** digit
public	**BigInteger**(int numBits, Random rnd) Constructs a randomly generated BigInteger, uniformly distributed over the range `0` to $(2^{numBits} - 1)$, inclusive. The uniformity of the distribution assumes that a fair source of random bits is provided in `rnd`. Note that this constructor

	always constructs a non-negative BigInteger. *numBits* maximum bitLength of the new BigInteger. *rnd* source of randomness to be used in computing the new BigInteger. **Throws** IllegalArgumentException: `numBits` is negative. **See also** bitLength()
public	**BigInteger**(int bitLength, int certainty, Random rnd) Constructs a randomly generated positive BigInteger that is probably prime, with the specified bitLength. It is recommended that the `probablePrime` method be used in preference to this constructor unless there is a compelling need to specify a certainty. *bitLength* bitLength of the returned BigInteger. *certainty* a measure of the uncertainty that the caller is willing to tolerate. The probability that the new BigInteger represents a prime number will exceed $(1 - 1/2^{certainty})$. The execution time of this constructor is proportional to the value of this parameter. *rnd* source of random bits used to select candidates to be tested for primality. **Throws** ArithmeticException: `bitLength < 2`. **See also** bitLength()

Methods

public BigInteger	**abs**() Returns a BigInteger whose value is the absolute value of this BigInteger. **return** `abs(this)`
public BigInteger	**add**(BigInteger val) Returns a BigInteger whose value is `(this + val)`. *val* value to be added to this BigInteger. **return** `this + val`
public BigInteger	**and**(BigInteger val) Returns a BigInteger whose value is `(this & val)`. (This method returns a negative BigInteger if and only if this and val are both negative.) *val* value to be AND'ed with this BigInteger. **return** `this & val`

public BigInteger	**andNot**(BigInteger val) Returns a BigInteger whose value is `(this & ~val)`. This method, which is equivalent to `and(val.not())`, is provided as a convenience for masking operations. (This method returns a negative BigInteger if and only if `this` is negative and `val` is positive.) *val* value to be complemented and AND'ed with this BigInteger. **return** `this & ~val`
public int	**bitCount**() Returns the number of bits in the two's complement representation of this BigInteger that differ from its sign bit. This method is useful when implementing bit-vector style sets atop BigIntegers. **return** number of bits in the two's complement representation of this BigInteger that differ from its sign bit.
public int	**bitLength**() Returns the number of bits in the minimal two's-complement representation of this BigInteger, *excluding* a sign bit. For positive BigIntegers, this is equivalent to the number of bits in the ordinary binary representation. (Computes `(ceil(log2(this < 0 ? -this : this+1)))`.) **return** number of bits in the minimal two's-complement representation of this BigInteger, *excluding* a sign bit.
public BigInteger	**clearBit**(int n) Returns a BigInteger whose value is equivalent to this BigInteger with the designated bit cleared. (Computes `(this & ~(1<<n))`.) *n* index of bit to clear. **return** `this & ~(1<<n)` **Throws** ArithmeticException: `n` is negative.
public int	**compareTo**(BigInteger val) Compares this BigInteger with the specified BigInteger. This method is provided in preference to individual methods for each of the six boolean comparison operators (<, ==, >, >=, !=, <=). The suggested idiom for performing these comparisons is: `(x.compareTo(y)` <*op*> `0)`, where <*op*> is one of the six comparison operators. *val* BigInteger to which this BigInteger is to be compared. **return** -1, 0 or 1 as this BigInteger is numerically less than, equal to, or greater than `val`.

public BigInteger	**divide**(BigInteger val) Returns a BigInteger whose value is `(this / val)`. *val* value by which this BigInteger is to be divided. **return** `this / val` **Throws** ArithmeticException: `val==0`
public BigInteger[]	**divideAndRemainder**(BigInteger val) Returns an array of two BigIntegers containing `(this / val)` followed by `(this % val)`. *val* value by which this BigInteger is to be divided, and the remainder computed. **return** an array of two BigIntegers: the quotient `(this / val)` is the initial element, and the remainder `(this % val)` is the final element. **Throws** ArithmeticException: `val==0`
public double	**doubleValue**() Converts this BigInteger to a `double`. This conversion is similar to the *narrowing primitive conversion* from `double` to `float` defined in the Java Language Specification: if this BigInteger has too great a magnitude to represent as a `double`, it will be converted to `Double#NEGATIVE_INFINITY` or `Double#POSITIVE_INFINITY` as appropriate. Note that even when the return value is finite, this conversion can lose information about the precision of the BigInteger value. **return** this BigInteger converted to a `double`.
public boolean	**equals**(Object x) Compares this BigInteger with the specified Object for equality. *x* Object to which this BigInteger is to be compared. **return** `true` if and only if the specified Object is a BigInteger whose value is numerically equal to this BigInteger.
public BigInteger	**flipBit**(int n) Returns a BigInteger whose value is equivalent to this BigInteger with the designated bit flipped. (Computes `(this ^ (1<<n))`.) *n* index of bit to flip. **return** `this ^ (1<<n)` **Throws** ArithmeticException: `n` is negative.
public float	**floatValue**() Converts this BigInteger to a `float`. This conversion is similar to the *narrowing primitive conversion* from `double` to `float` defined in the Java Language Specification: if this BigInteger has too great a magnitude to represent as a `float`, it will be converted to `Float#NEGATIVE_INFINITY` or

	`Float#POSITIVE_INFINITY` as appropriate. Note that even when the return value is finite, this conversion can lose information about the precision of the BigInteger value. **return** this BigInteger converted to a `float`.
public BigInteger	**gcd**(BigInteger val) Returns a BigInteger whose value is the greatest common divisor of `abs(this)` and `abs(val)`. Returns 0 if `this==0 && val==0`. *val* value with which the GCD is to be computed. **return** `GCD(abs(this), abs(val))`
public int	**hashCode**() Returns the hash code for this BigInteger. **return** hash code for this BigInteger.
public int	**intValue**() Converts this BigInteger to an `int`. This conversion is analogous to a *narrowing primitive conversion* from `long` to `int` as defined in the Java Language Specification: if this BigInteger is too big to fit in an `int`, only the low-order 32 bits are returned. Note that this conversion can lose information about the overall magnitude of the BigInteger value as well as return a result with the opposite sign. **return** this BigInteger converted to an `int`.
public long	**longValue**() Converts this BigInteger to a `long`. This conversion is analogous to a *narrowing primitive conversion* from `long` to `int` as defined in the Java Language Specification: if this BigInteger is too big to fit in a `long`, only the low-order 64 bits are returned. Note that this conversion can lose information about the overall magnitude of the BigInteger value as well as return a result with the opposite sign. **return** this BigInteger converted to a `long`.
public BigInteger	**max**(BigInteger val) Returns the maximum of this BigInteger and `val`. *val* value with which the maximum is to be computed. **return** the BigInteger whose value is the greater of this and `val`. If they are equal, either may be returned.
public BigInteger	**min**(BigInteger val) Returns the minimum of this BigInteger and `val`. *val* value with which the minimum is to be computed.

	return the BigInteger whose value is the lesser of this BigInteger and `val`. If they are equal, either may be returned.
public BigInteger	**mod**(BigInteger m) Returns a BigInteger whose value is `(this mod m)`. This method differs from `remainder` in that it always returns a *non-negative* BigInteger. *m* the modulus. **return** `this mod m` **Throws** ArithmeticException: `m <= 0` **See also** remainder
public BigInteger	**modInverse**(BigInteger m) Returns a BigInteger whose value is $(this^{-1} \bmod m)$. *m* the modulus. **return** $this^{-1} \bmod m$. **Throws** ArithmeticException: `m <= 0`, or this BigInteger has no multiplicative inverse mod m (that is, this BigInteger is not *relatively prime* to m).
public BigInteger	**modPow**(BigInteger exponent, BigInteger m) Returns a BigInteger whose value is $(this^{exponent} \bmod m)$. (Unlike `pow`, this method permits negative exponents.) *exponent* the exponent. *m* the modulus. **return** $this^{exponent} \bmod m$ **Throws** ArithmeticException: `m <= 0` **See also** modInverse
public BigInteger	**multiply**(BigInteger val) Returns a BigInteger whose value is `(this * val)`. *val* value to be multiplied by this BigInteger. **return** `this * val`
public BigInteger	**negate**() Returns a BigInteger whose value is `(-this)`. **return** `-this`
public BigInteger	**nextProbablePrime**() Returns the first integer greater than this `BigInteger` that is probably prime. The probability that the number returned by this method is composite does not exceed 2^{-100}. This method will never skip over a prime when searching: if it returns `p`, there is no prime `q` such that `this < q < p`. **return** the first integer greater than this `BigInteger` that is probably prime. **Throws** ArithmeticException: `this < 0`.

	since 1.5
public BigInteger	**not**() Returns a BigInteger whose value is `(~this)`. (This method returns a negative value if and only if this BigInteger is non-negative.) **return** `~this`
public BigInteger	**or**(BigInteger val) Returns a BigInteger whose value is `(this | val)`. (This method returns a negative BigInteger if and only if either this or val is negative.) *val* value to be OR'ed with this BigInteger. **return** `this | val`
public BigInteger	**pow**(int exponent) Returns a BigInteger whose value is $(this^{exponent})$. Note that `exponent` is an integer rather than a BigInteger. *exponent* exponent to which this BigInteger is to be raised. **return** $this^{exponent}$ **Throws** ArithmeticException: `exponent` is negative. (This would cause the operation to yield a non-integer value.)
public static BigInteger	**probablePrime**(int bitLength, Random rnd) Returns a positive BigInteger that is probably prime, with the specified bitLength. The probability that a BigInteger returned by this method is composite does not exceed 2^{-100}. *bitLength* bitLength of the returned BigInteger. *rnd* source of random bits used to select candidates to be tested for primality. **return** a BigInteger of `bitLength` bits that is probably prime **Throws** ArithmeticException: `bitLength < 2`. **since** 1.4 **See also** bitLength()
public BigInteger	**remainder**(BigInteger val) Returns a BigInteger whose value is `(this % val)`. *val* value by which this BigInteger is to be divided, and the remainder computed. **return** `this % val` **Throws** ArithmeticException: `val==0`
public BigInteger	**shiftLeft**(int n) Returns a BigInteger whose value is `(this << n)`. The shift distance, `n`, may be negative, in which case this method performs a right shift. (Computes $floor(this * 2^n)$.)

	n shift distance, in bits. **return** `this << n` **See also** shiftRight
public BigInteger	**shiftRight**(int n) Returns a BigInteger whose value is `(this >> n)`. Sign extension is performed. The shift distance, `n`, may be negative, in which case this method performs a left shift. (Computes `floor(this / 2^n)`.) *n* shift distance, in bits. **return** `this >> n` **See also** shiftLeft
public int	**signum**() Returns the signum function of this BigInteger. **return** -1, 0 or 1 as the value of this BigInteger is negative, zero or positive.
public BigInteger	**subtract**(BigInteger val) Returns a BigInteger whose value is `(this - val)`. *val* value to be subtracted from this BigInteger. **return** `this - val`
public boolean	**testBit**(int n) Returns `true` if and only if the designated bit is set. (Computes `((this & (1<<n)) != 0)`.) *n* index of bit to test. **return** `true` if and only if the designated bit is set. **Throws** ArithmeticException: `n` is negative.
public byte[]	**toByteArray**() Returns a byte array containing the two's-complement representation of this BigInteger. The byte array will be in *big-endian* byte-order: the most significant byte is in the zeroth element. The array will contain the minimum number of bytes required to represent this BigInteger, including at least one sign bit, which is `(ceil((this.bitLength() + 1)/8))`. (This representation is compatible with the `(byte[])` constructor.) **return** a byte array containing the two's-complement representation of this BigInteger. **See also** BigInteger(byte[])
public String	**toString**(int radix) Returns the String representation of this BigInteger in the given radix. If the radix is outside the range from `Character#MIN_RADIX` to `Character#MAX_RADIX` inclusive, it will default to 10 (as is the case for `Integer.toString`). The digit-to-character mapping provided by `Character.forDigit` is used,

	and a minus sign is prepended if appropriate. (This representation is compatible with the `(String, int)` constructor.) *radix* radix of the String representation. **return** String representation of this BigInteger in the given radix. **See also** toString, forDigit, BigInteger(java.lang.String, int)
public String	**toString**() Returns the decimal String representation of this BigInteger. The digit-to-character mapping provided by `Character.forDigit` is used, and a minus sign is prepended if appropriate. (This representation is compatible with the `(String)` constructor, and allows for String concatenation with Java's + operator.) **return** decimal String representation of this BigInteger. **See also** forDigit, BigInteger(java.lang.String)
public static BigInteger	**valueOf**(long val) Returns a BigInteger whose value is equal to that of the specified `long`. This "static factory method" is provided in preference to a (`long`) constructor because it allows for reuse of frequently used BigIntegers. *val* value of the BigInteger to return. **return** a BigInteger with the specified value.
public BigInteger	**xor**(BigInteger val) Returns a BigInteger whose value is `(this ^ val)`. (This method returns a negative BigInteger if and only if exactly one of this and val are negative.) *val* value to be XOR'ed with this BigInteger. **return** `this ^ val`

Properties

public BigInteger	**setBit**(int n) Returns a BigInteger whose value is equivalent to this BigInteger with the designated bit set. (Computes `(this | (1<<n))`.) *n* index of bit to set. **return** `this | (1<<n)` **Throws** ArithmeticException: `n` is negative.
public int	**getLowestSetBit**() Returns the index of the rightmost (lowest-order) one bit in this BigInteger (the number of zero bits to the right of the rightmost

	one bit). Returns -1 if this BigInteger contains no one bits. (Computes `(this==0? -1 : log2(this & -this))`.) **return** index of the rightmost one bit in this BigInteger.
public boolean	**isProbablePrime**(int certainty) Returns `true` if this BigInteger is probably prime, `false` if it's definitely composite. If `certainty` is `<= 0`, `true` is returned. *certainty* a measure of the uncertainty that the caller is willing to tolerate: if the call returns `true` the probability that this BigInteger is prime exceeds $(1 - 1/2^{certainty})$. The execution time of this method is proportional to the value of this parameter. **return** `true` if this BigInteger is probably prime, `false` if it's definitely composite.