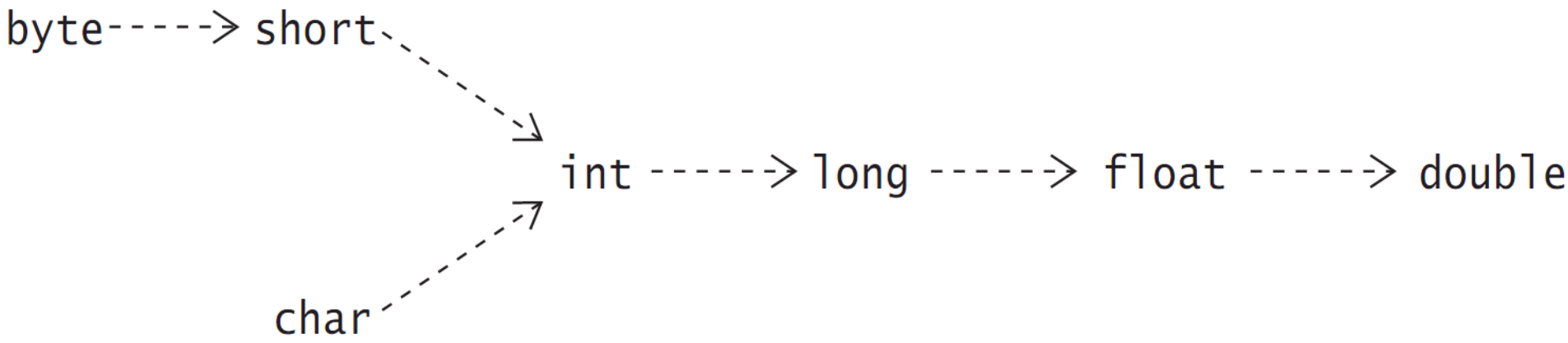
Operators and Expressions

Conversions.

Widening and Narrowing Primitive Conversions



Widening and Narrowing Reference Conversions

Conversions up the type hierarchy are called widening reference conversions (also called upcasting), i.e., such a conversion converts from a subtype to a supertype.

```
Object obj = "upcast me";  // Widening: Object <--- String
```

Conversions down the type hierarchy represent narrowing reference conversions (also called downcasting).

```
String str = (String) obj;
// Narrowing requires cast: String <--- Object
```

Conversions

Widening reference conversions are usually done implicitly, whereas narrowing reference conversions usually require a cast.

The compiler will reject casts that are not legal or issue an unchecked warning under certain circumstances if type safety cannot be guaranteed.

Conversions

Widening reference conversions do not require any runtime checks and never result in an exception during execution.

Narrowing reference conversions require a runtime check and can throw a ClassCastException if the conversion is not legal.

Boxing and Unboxing Conversions

A boxing conversion converts the value of a primitive type to a corresponding value of its wrapper type. If p is a value of a primitiveType, boxing conversion converts p into a reference r of corresponding WrapperType, such that r.primitiveTypeValue() == p.

```
Integer iRef = 10;      // Boxing: Integer <----- int
System.out.println(iRef.intValue() == 10);   // true
```

Boxing and Unboxing Conversions

An unboxing conversion converts the value of a wrapper type to a value of its corresponding primitive type. If r is a reference of a WrapperType, unboxing conversion converts the reference r into r.primitiveTypeValue(), where primitiveType is the primitive type corresponding to the WrapperType.

```
int i = iRef;     // Unboxing: int <----- Integer
System.out.println(iRef.intValue() == i);  // true
```

Other Conversions

- ***Identity conversions*** are always permitted, as they allow conversions from a type to that same type. An identity conversion is always permitted.

```
int i = (int) 10;           // int <---- int
String str = (String) "Hi";    // String <---- String
```

- ***String conversions*** allow a value of any other type to be converted to a String type in the context of the string concatenation operator +
- ***Unchecked conversions*** are permitted to facilitate operability between legacy and generic code
- ***Capture conversions*** aid in increasing the usefulness of wildcards in generic code

Type Conversion Contexts

Conversion Categories	Conversion Contexts			
	Assignment	**Method Invocation**	**Casting**	**Numeric Promotion**
Widening/ Narrowing *Primitive* Conversions	widening narrowing for *constant expressions* of non-`long` integer type, with optional boxing	widening	both	widening
Widening/ Narrowing *Reference* Conversions	widening	widening	both, followed by optional unchecked conversion	Not applicable
Boxing/ Unboxing Conversions	unboxing, followed by optional widening *primitive* conversion boxing, followed by optional widening *reference* conversion	unboxing, followed by optional widening *primitive* conversion boxing, followed by optional widening *reference* conversion	both	unboxing, followed by optional widening *primitive* conversion

Type Conversion Contexts

Assignment Context

An assignment conversion converts the type of an expression to the type of a target variable. An expression (or its value) is assignable to the target variable, if the type of the expression can be converted to the type of the target variable by an assignment conversion.

Note

Special case where a narrowing conversion occurs when assigning a non-long integer constant expression:

```
byte b = 10;   // Narrowing conversion: byte <--- int
```

Type Conversion Contexts

Method Invocation Context

method invocation and *assignment conversions* differ in one respect: method invocation conversions do not include the implicit narrowing conversion performed for integer constant expressions.

Type Conversion Contexts

Casting Context of the Unary Type Cast Operator: (type)

The type cast construct has the following syntax:

`(<type>) <expression>`

The cast operator (<type>) is applied to the value of the <expression>.

```
long l = (long) 10; // Widening primitive conversion: long <--- int
int i = (int) l;    // Narrowing primitive conversion: int <--- long
Object obj = (Object) "Upcast me"; // Widening ref conversion:
                                   //Object <--- String
String str = (String) obj;         // Narrowing ref conversion:
                                   //String <--- Object
Integer iRef = (Integer) i;        // Boxing: Integer <--- int
i = (int) iRef;                    // Unboxing: int <--- Integer
```

Numeric Promotion Context

Unary Numeric Promotion

- If the single operand is of type Byte, Short, Character, or Integer, it is unboxed. If the resulting value is narrower than int, it is promoted to a value of type int by a widening conversion.
- Otherwise, if the single operand is of type Long, Float, or Double, it is unboxed.
- Otherwise, if the single operand is of a type narrower than int, its value is promoted to a value of type int by a widening conversion.
- Otherwise, the operand remains unchanged.

In other words, unary numeric promotion results in an operand value that is either int or wider.

Numeric Promotion Context

Unary numeric promotion is applied in the following expressions:

- operand of the unary arithmetic operators + and -
- array creation expression; e.g., new int[20], where the dimension expression (in this case 20) must evaluate to an int value
- indexing array elements; e.g., objArray['a'], where the index expression (in this case 'a') must evaluate to an int value

Numeric Promotion Context

Binary Numeric Promotion

Binary numeric promotion implicitly applies appropriate widening primitive conversions so that a pair of operands have the widest numeric type of the two, which is always at least **`int`**.

This means that the resulting type of the operands is at least int.

Numeric Promotion Context

Binary numeric promotion is applied in the following expressions:

- operands of the arithmetic operators *, /, %, +, and -
- operands of the relational operators <, <=, >, and >=
- operands of the numerical equality operators == and !=
- operands of the conditional operator ? :, under certain circumstances

Precedence and Associativity Rules for Operators

- The operators are shown with decreasing precedence from the top of the table.
- Operators within the same row have the same precedence.
- Parentheses, ( ), can be used to override precedence and associativity.
- The unary operators, which require one operand, include the following: the postfix increment (++) and decrement (--) operators from the first row, all the prefix operators (+, -, ++, --, ~, !) in the second row, and the prefix operators (object creation operator new, cast operator (type)) in the third row.
- The conditional operator (?:) is ternary, that is, requires three operands.
- All operators not listed above as unary or ternary, are binary, that is, require two operands.
- All binary operators, except for the relational and assignment operators, associate from left to right. The relational operators are nonassociative.
- Except for unary postfix increment and decrement operators, all unary opera-tors, all assignment operators, and the ternary conditional operator associate from right to left.

Precedence and Associativity Rules for Operators

Postfix operators	`[] . (parameters) expression++ expression--`
Unary prefix operators	`++expression --expression +expression -expression ~ !`
Unary prefix creation and cast	`new (type)`
Multiplicative	`* / %`
Additive	`+ -`
Shift	`<< >> >>>`
Relational	`< <= > >= instanceof`
Equality	`== !=`
Bitwise/logical AND	`&`
Bitwise/logical XOR	`^`
Bitwise/logical OR	`|`
Conditional AND	`&&`
Conditional OR	`||`
Conditional	`?:`
Assignment	`= += -= *= /= %= <<= >>= >>>= &= ^= |=`

Evaluation Order of Operands

- Left-Hand Operand Evaluation First

```
int b = 10;
System.out.println((b=3) + b);
```

the value printed will be 6 and not 13

- Operand Evaluation before Operation Execution

Java guarantees that all operands of an operator are fully evaluated before the actual operation is performed.

This rule does not apply to the short-circuit conditional operators &&, ||, and ?:.

- Left to Right Evaluation of Argument Lists

The Simple Assignment Operator =

<variable> = <expression>

which can be read as “the target, <variable>, gets the value of the source, <expression>”.

The previous value of the target variable is overwritten by the assignment operator =

Assigning Primitive Values

```
int j, k;
j = 10; // j gets the value 10.
j = 5;  // j gets the value 5. Previous value is overwritten.
k = j;  // k gets the value 5.
```

The assignment operator has the lowest precedence allowing the expression on the right-hand side to be evaluated before assignment.

```
int i;
i = 5;              // i gets the value 5.
i = i + 1;          // i gets the value 6.
                    // + has higher precedence than =.
i = 20 - i * 2;     // i gets the value 8: (20 - (i * 2))
```

Assigning References

Copying reference values by assignment creates aliases.

```
Pizza pizza1 = new Pizza("Hot&Spicy");
Pizza pizza2 = new Pizza("Sweet&Sour");
pizza2 = pizza1;
```

Assigning a reference value does not create a copy of the source object denoted by the reference variable on the right-hand side.

Multiple Assignments

The assignment statement is an expression statement, which means that application of the binary assignment operator returns the value of the expression on the right-hand side.

```
int j, k;
j = 10; // j gets the value 10 which is returned
k = j;  // k gets the value of j, which is 10,
        // and this value is returned
```

The last two assignments can be written as multiple assignments, illustrating the right associativity of the assignment operator.

```
k = j = 10;         // (k = (j = 10))
```

Multiple assignments are equally valid with references.

```
Pizza pizzaOne, pizzaTwo;
pizzaOne = pizzaTwo = new Pizza("Supreme"); // Aliases.
```

Example of operand evaluation order

```
int[] a = {10, 20, 30, 40, 50}; // an array of int
int index = 4;
a[index] = index = 2;          // (1)
```

What is the value of index, and which array element a[index] is assigned a value in the multiple assignment statement at (1)?

Example of operand evaluation order

```
int[] a = {10, 20, 30, 40, 50}; // an array of int
int index = 4;
a[index] = index = 2;          // (1)
```

What is the value of index, and which array element a[index] is assigned a value in the multiple assignment statement at (1)? The evaluation proceeds as follows:

```
a[index] = index = 2;
a[4]     = index = 2;
a[4]     = (index = 2); // index gets the value 2.
                        // = is right associative.
a[4]     =      2;      // The value of a[4] is changed
                        // from 50 to 2.
```

Arithmetic Operators: *, /, %, +, -

Unary + Addition - Subtraction

Binary * Multiplication / Division % Remainder

+ Addition - Subtraction

Range of Numeric Values

Integer arithmetic always returns a value that is in range, except in the case of integer division by zero and remainder by zero, which causes an `ArithmeticException`

```
int tooBig   = Integer.MAX_VALUE + 1;
// -2147483648 which is Integer.MIN_VALUE.
int tooSmall = Integer.MIN_VALUE - 1;
//  2147483647 which is Integer.MAX_VALUE.
```

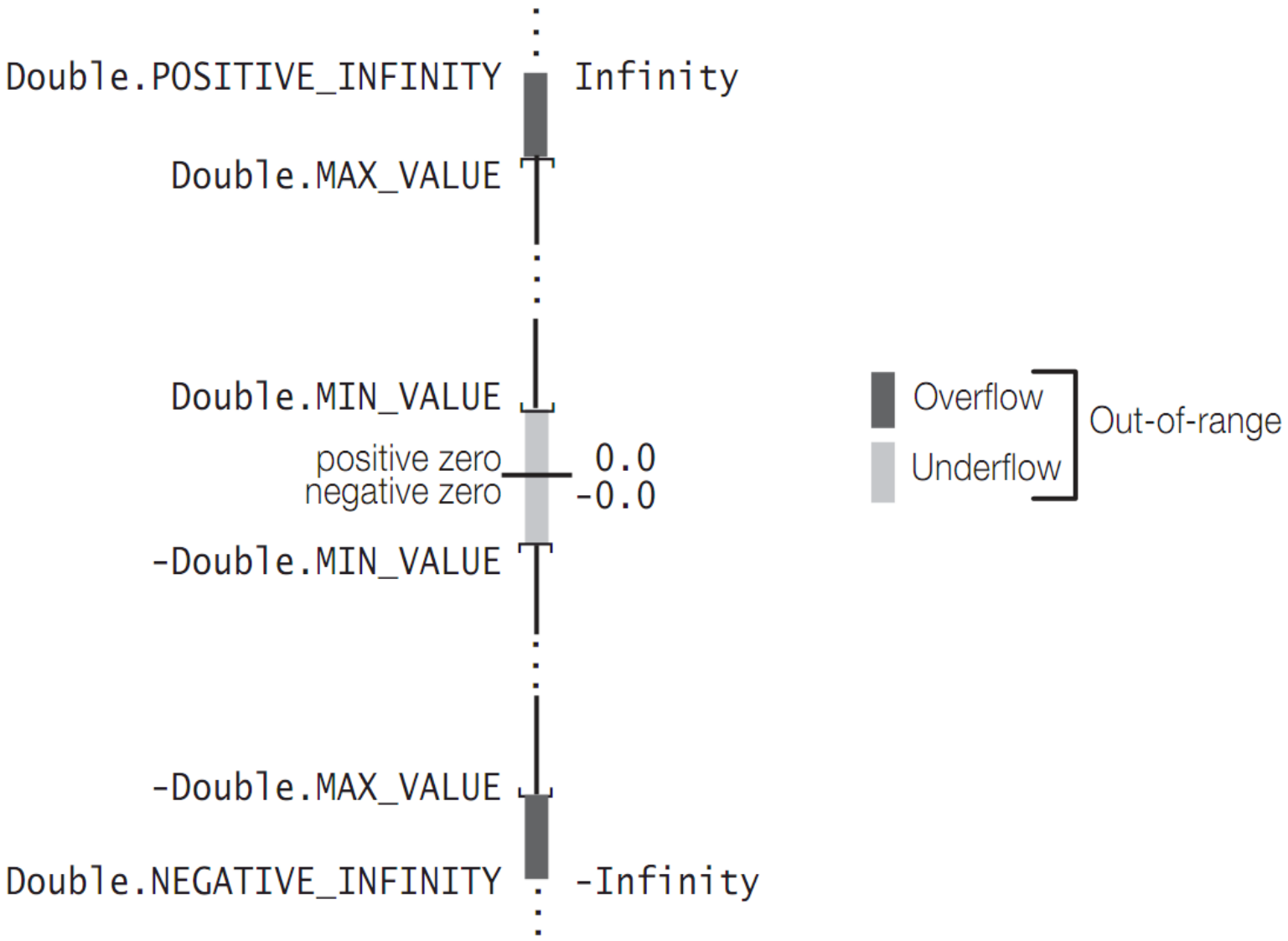
Floating-Point Arithmetic

Certain floating-point operations result in values that are out-of-range. Typically, adding or multiplying two very large floating-point numbers can result in an out-of-range value which is represented by Infinity.

Attempting floating-point division by zero also returns infinity.

```
System.out.println( 4.0 / 0.0); // Prints:  Infinity
System.out.println(-4.0 / 0.0); // Prints: -Infinity
```

Overflow and Underflow in Floating-Point Arithmetic



Underflow and NaN

Underflow occurs in the following situations:

- the result is between Double.MIN_VALUE (or Float.MIN_VALUE) and zero; e.g., the result of (5.1E-324 - 4.9E-324). Underflow then returns positive zero 0.0 (or 0.0F).
- the result is between -Double.MIN_VALUE (or -Float.MIN_VALUE) and zero; e.g., the result of (-Double.MIN_VALUE * 1E-1). Underflow then returns negative zero -0.0 (or -0.0F).

Negative zero compares equal to positive zero, i.e., (-0.0 == 0.0) is true.

Certain operations have no mathematical result, and are represented by NaN

Strict Floating-Point Arithmetic: strictfp

Although floating-point arithmetic in Java is defined in accordance with the IEEE-754 32-bit (float) and 64-bit (double) standard formats, the language does allow JVM implementations to use other extended formats for intermediate results.

This means that floating-point arithmetic can give different results on such JVMs, with possible loss of precision. Such a behavior is termed non-strict, in contrast to being strict and adhering to the standard formats.

To ensure that identical results are produced on all JVMs, the keyword **`strictfp`** can be used to enforce strict behavior for floating-point arithmetic. The modifier **`strictfp`** can be applied to classes, interfaces, and methods.

However, note that strictness is not inherited by the subclasses or subinterfaces.

Constant expressions are always evaluated strictly at compile time.

Unary Arithmetic Operators: -, +

- The unary operators have the highest precedence of all the arithmetic operators.

```
int value = - -10;  // (-(-10)) is 10
```

Notice the blank needed to separate the unary operators; otherwise, these would be interpreted as the decrement operator --

The unary operator + has no effect on the evaluation of the operand value.

Multiplicative Binary Operators: *, /, %

Multiplication Operator: *

The multiplication operator * multiplies two numbers.

```
int sameSigns         = -4 * -8;       // result:  32
double oppositeSigns =  4.0  * -8.0;  // result: -32.0
int    zero           =  0  * -0;      // result:   0
```

Division Operator: /

The division operator / is overloaded. If its operands are integral, the operation results in integer division.

```
int    i1 = 4 / 5;   // result: 0
int    i2 = 8 / 8;   // result: 1
double d1 = 12 / 8;  // result: 1.0,
          //integer division, then widening conversion.
```

Remainder Operator: %

For integer remainder operation, where only integer operands are involved, evaluation of the expression (x % y) always satisfies the following relation:

```
x == (x / y) * y + (x % y)
```

%

```
Calculating (7 % 5):
7 == (7 / 5) * 5 + (7 % 5)
  == (  1  ) * 5 + (7 % 5)
  ==           5 + (7 % 5)
2 ==  (7 % 5)
i.e., (7 % 5) is equal to 2
```

```
Calculating (-7 % 5):
-7 == (-7 / 5) * 5 + (-7 % 5)
   == (  -1  ) * 5 + (-7 % 5)
   ==           -5 + (-7 % 5)
-2 == (-7 % 5)
i.e., (-7 % 5) is equal to -2
```

```
Calculating (7 % -5):
7 == (7 / -5) * -5 + (7 % -5)
  == (  -1  ) * -5 + (7 % -5)
  ==            5 + (7 % -5)
2 == (7 % -5)
i.e., (7 % -5) is equal to 2
```

```
Calculating (-7 % -5):
-7 == (-7 / -5) * -5 + (-7 % -5)
   == (   1   ) * -5 + (-7 % -5)
   ==           -5 + (-7 % -5)
-2 == (-7 % -5)
i.e., (-7 % -5) is equal to -2
```

Remainder can only be negative if the dividend is negative, and the sign of the divisor is irrelevant

An ArithmeticException is thrown if the divisor evaluates to zero.

Floating-Point Remainder

Note that the remainder operator not only accepts integral operands, but floating-point operands as well. The floating-point remainder r is defined by the relation:

r == a - (b * q)

where a and b are the dividend and the divisor, respectively, and q is the integer quotient of (a/b).

```
double  dr0 =  7.0  % 7.0;    //  0.0
float   fr1 =  7.0F %  5.0F;   //  2.0F
double  dr1 =  7.0  % -5.0;    //  2.0
float   fr2 = -7.0F %  5.0F;   // -2.0F
double  dr2 = -7.0  % -5.0;    // -2.0
boolean fpRel = dr2  == (-7.0) - (-5.0) * (long)(-7.0 / -5.0);
// true
float   fr3 = -7.0F %  0.0F;   // NaN
```

Additive Binary Operators: +, -

Arithmetic Expression	Evaluation	Result When Printed
3 + 2 - 1	((3 + 2) - 1)	4
2 + 6 * 7	(2 + (6 * 7))	44
-5+7- -6	(((-5)+7)-(-6))	8
2+4/5	(2+(4/5))	2
13 % 5	(13 % 5)	3
11.5 % 2.5	(11.5 % 2.5)	1.5
10 / 0	ArithmeticException	
2+4.0/5	(2.0+(4.0/5.0))	2.8
4.0 / 0.0	(4.0 / 0.0)	Infinity
-4.0 / 0.0	((-4.0) / 0.0)	-Infinity
0.0 / 0.0	(0.0 / 0.0)	NaN

Numeric Promotions in Arithmetic Expressions

Unary numeric promotion is applied to the single operand of the unary arithmetic operators - and +.

When a unary arithmetic operator is applied to an operand whose type is narrower than int, the operand is promoted to a value of type int, with the operation resulting in an int value.

```
byte b = 3; // int literal in range.
            // Narrowing conversion.
b = (byte) -b; // Cast required on assignment.
```

Numeric Promotion in Arithmetic Expressions

```
public class NumPromotion {
  public static void main(String[] args) {
    byte   b = 32;
    char   c = 'z';    // Unicode value 122 (\u007a)
    short  s = 256;
    int    i = 10000;
    float  f = 3.5F;
    double d = 0.5;
    double v = (d * i) + (f * - b) - (c / s);  // (1)
    System.out.println("Value of v: " + v);
  }
}
```

Numeric Promotion in Arithmetic Expressions

```
public class NumPromotion {
  public static void main(String[] args) {
    byte   b = 32;
    char   c = 'z';    // Unicode value 122 (\u007a)
    short  s = 256;
    int    i = 10000;
    float  f = 3.5F;
    double d = 0.5;
    double v = (d * i) + (f * - b) - (c / s);  // (1)
    System.out.println("Value of v: " + v);
  }
}
Value of v: 4888.0
```

Arithmetic Compound Assignment Operators: *=, /=, %=, +=, -=

A compound assignment operator has the following syntax:

```
<variable> <op>= <expression>
```

and the following semantics:

```
<variable> =
     (<type>)((<variable>) <op> (<expression>))
```

The type of the <variable> is <type> and the <variable> is evaluated only once.

Note the cast and the parentheses implied in the semantics.

Arithmetic Compound Assignment Operators

Expression:	Given T as the Numeric Type of x, the Expression Is Evaluated as:
x *= a	x = (T) ((x) * (a))
x /= a	x = (T) ((x) / (a))
x %= a	x = (T) ((x) % (a))
x += a	x = (T) ((x) + (a))
x -= a	x = (T) ((x) - (a))

```
int i = 2;
i *= i + 4;          // (1) Evaluated as i = (int) ((i) * (i + 4)).
Integer iRef = 2;
iRef *= iRef + 4;
// (2) Evaluated as iRef = (Integer) ((iRef) * (iRef + 4)).
byte b = 2;
b += 10;             // (3) Evaluated as b = (byte) (b + 10).
b = b + 10;          // (4) Will not compile. Cast is required.
```

The Binary String Concatenation Operator +

Non-String operands are converted as follows:

- For an operand of a primitive data type, its value is first converted to a reference value using the object creation expression. A string representation of the reference value is obtained as explained below for reference types.
- Values like true, false, and null are represented by string representations of these literals. A reference variable with the value null also has the string representation "null" in this context.
- For all reference value operands, a string representation is constructed by calling the toString() method on the referred object. Most classes override this method from the Object class in order to provide a more meaningful string representation of their objects.

Variable Increment and Decrement Operators: ++, --

Prefix increment operator has the following semantics:

`++i` adds 1 to the value in i, and stores the new value in i. It returns the new value as the value of the expression. It is equivalent to the following statements:

```
i += 1;
result = i;
return result;
```

Variable Increment and Decrement Operators: ++, --

Postfix increment operator has the following semantics:

`j++` adds 1 to the value in j, and stores the new value in j. It returns the old value in j before the new value is stored in j, as the value of the expression. It is equivalent to the following statements:

```
result = j;
j += 1;
return result;
```

Boolean Expressions

- Boolean expressions, when used as conditionals in control statements, allow the program flow to be controlled during execution.
- Boolean expressions can be formed using **relational** operators, **equality** operators, **bitwise** operators, **boolean logical** operators, **conditional** operators, the **assignment** operator, and the **instanceof** operator

Relational Operators: <, <=, >, >=

a < b a less than b?

a <= b a less than or equal to b?

a > b a greater than b?

a >= b a greater than or equal to b?

Equality

- Primitive Data Value Equality: ==, !=
- Object Reference Equality: ==, !=
- Object Value Equality
  - The Object class provides the method public boolean equals(Object obj), which can be overridden to give the right semantics of object value equality.
  - The default implementation of this method in the Object class returns true only if the object is compared with itself, i.e., as if the equality operator == had been used to compare aliases of an object.

Boolean Logical Operators: !, ^, &, |

Boolean logical operators include:

- the unary operator ! (logical complement)

and

- The binary operators
  - & (logical AND),
  - | (logical inclusive OR), and
  - ^ (logical exclusive OR, also called logical XOR).

Boolean logical operators can be applied to boolean or Boolean operands, returning a boolean value. The operators &, |, and ^ can also be applied to integral operands to perform bitwise logical operations

These operators always evaluate both the operands, unlike their counterpart conditional operators && and ||

Boolean Logical Compound Assignment Operators: &=, ^=, |=

- The left-hand operand must be a boolean variable, and the right-hand operand must be a boolean expression.
- An identity conversion is applied implicitly on assignment.

Conditional Operators: &&, ||

The conditional operators && and || are similar to their counterpart logical operators & and |, except that their evaluation is short-circuited.

The Conditional Operator: ?:

The ternary conditional operator allows conditional expressions to be defined. The operator has the following syntax:

```
<condition> ? <expression1> : <expression2>
```

If the boolean expression <condition> is true then <expression1> is evaluated; otherwise, <expression2> is evaluated.

Of course, <expression1> and <expression2> must evaluate to values of compatible types. The value of the expression evaluated is returned by the conditional expression.

```
boolean leapYear = false;
int daysInFebruary = leapYear ? 29 : 28; // 28
```

The Conditional Operator: ?:

- The conditional operator is the expression equivalent of the if-else statement.
- The conditional expression can be nested and the conditional operator associates from right to left:

(a?b?c?d:e:f:g) evaluates as (a?(b?(c?d:e):f):g)

Other Operators: new, [], instanceof

- The new operator is used to create objects, i.e., instances of classes and arrays. It is used with a constructor call to instantiate classes
- The [] notation is used to declare and construct arrays and also to access array elements
- The boolean, binary, and infix operator instanceof is used to test the type of an object