

CHAPTER 5

Conversions and Promotions

Every expression written in the Java programming language has a type that can be deduced from the structure of the expression and the types of the literals, variables, and methods mentioned in the expression. It is possible, however, to write an expression in a context where the type of the expression is not appropriate. In some cases, this leads to an error at compile time; for example, if the expression in an `if` statement (§14.9) has any type other than `boolean`, a compile-time error occurs. In other cases, the context may be able to accept a type that is related to the type of the expression; as a convenience, rather than requiring the programmer to indicate a type conversion explicitly, the language performs an implicit *conversion* from the type of the expression to a type acceptable for its surrounding context.

A specific conversion from type *S* to type *T* allows an expression of type *S* to be treated at compile time as if it had type *T* instead. In some cases this will require a corresponding action at run time to check the validity of the conversion or to translate the run-time value of the expression into a form appropriate for the new type *T*. For example:

- A conversion from type `Object` to type `Thread` requires a run-time check to make sure that the run-time value is actually an instance of class `Thread` or one of its subclasses; if it is not, an exception is thrown.
- A conversion from type `Thread` to type `Object` requires no run-time action; `Thread` is a subclass of `Object`, so any reference produced by an expression of type `Thread` is a valid reference value of type `Object`.
- A conversion from type `int` to type `long` requires run-time sign-extension of a 32-bit integer value to the 64-bit `long` representation. No information is lost.

A conversion from type `double` to type `long` requires a nontrivial translation from a 64-bit floating-point value to the 64-bit integer representation. Depending on the actual run-time value, information may be lost.

In every conversion context, only certain specific conversions are permitted. For convenience of description, the specific conversions that are possible in the Java programming language are grouped into several broad categories:

- Identity conversions
- Widening primitive conversions
- Narrowing primitive conversions
- Widening reference conversions
- Narrowing reference conversions
- String conversions
- Value set conversions

There are five *conversion contexts* in which conversion of expressions may occur. Each context allows conversions in some of the categories named above but not others. The term "conversion" is also used to describe the process of choosing a specific conversion for such a context. For example, we say that an expression that is an actual argument in a method invocation is subject to "method invocation conversion," meaning that a specific conversion will be implicitly chosen for that expression according to the rules for the method invocation argument context.

One conversion context is the operand of a numeric operator such as `+` or `*`. The conversion process for such operands is called *numeric promotion*. Promotion is special in that, in the case of binary operators, the conversion chosen for one operand may depend in part on the type of the other operand expression. This chapter first describes the seven categories of conversions (§5.1),

including the special conversions to `String` allowed for the string concatenation operator `+`. Then the five conversion contexts are described:

- Assignment conversion ([§5.2](#), [§15.26](#)) converts the type of an expression to the type of a specified variable. The conversions permitted for assignment are limited in such a way that assignment conversion never causes an exception.
- Method invocation conversion ([§5.3](#), [§15.9](#), [§15.12](#)) is applied to each argument in a method or constructor invocation and, except in one case, performs the same conversions that assignment conversion does. Method invocation conversion never causes an exception.
- Casting conversion ([§5.5](#)) converts the type of an expression to a type explicitly specified by a cast operator ([§15.16](#)). It is more inclusive than assignment or method invocation conversion, allowing any specific conversion other than a string conversion, but certain casts to a reference type may cause an exception at run time.
- String conversion ([§5.4](#), [§15.18.1](#)) allows any type to be converted to type `String`.
- Numeric promotion ([§5.6](#)) brings the operands of a numeric operator to a common type so that an operation can be performed.

Here are some examples of the various contexts for conversion:

```
class Test {  
    public static void main(String[] args) {  
        // Casting conversion (§5.4) of a float literal to  
        // type int. Without the cast operator, this would  
        // be a compile-time error, because this is a  
        // narrowing conversion (§5.1.3):  
        int i = (int)12.5f;  
  
        // String conversion (§5.4) of i's int value:  
        System.out.println("(int)12.5f==" + i);  
  
        // Assignment conversion (§5.2) of i's value to type  
        // float. This is a widening conversion (§5.1.2):  
        float f = i;  
  
        // String conversion of f's float value:  
        System.out.println("after float widening: " + f);  
  
        // Numeric promotion (§5.6) of i's value to type  
        // float. This is a binary numeric promotion.  
        // After promotion, the operation is float*float:  
        System.out.print(f);  
        f = f * i;  
  
        // Two string conversions of i and f:  
        System.out.println("*" + i + "==" + f);  
  
        // Method invocation conversion (§5.3) of f's value  
        // to type double, needed because the method Math.sin  
        // accepts only a double argument:  
        double d = Math.sin(f);  
  
        // Two string conversions of f and d:
```

```

        System.out.println("Math.sin(" + f + ")==" + d);
    }
}

```

which produces the output:

```

(int)12.5f==12
after float widening: 12.0
12.0*12==144.0
Math.sin(144.0)==-0.49102159389846934

```

5.1 Kinds of Conversion

Specific type conversions in the Java programming language are divided into seven categories.

5.1.1 Identity Conversions

A conversion from a type to that same type is permitted for any type.

This may seem trivial, but it has two practical consequences. First, it is always permitted for an expression to have the desired type to begin with, thus allowing the simply stated rule that every expression is subject to conversion, if only a trivial identity conversion. Second, it implies that it is permitted for a program to include redundant cast operators for the sake of clarity.

The only permitted conversion that involves the type `boolean` is the identity conversion from `boolean` to `boolean`.

5.1.2 Widening Primitive Conversion

The following 19 specific conversions on primitive types are called the *widening primitive conversions*:

- byte to short, int, long, float, or double
- short to int, long, float, or double
- char to int, long, float, or double
- int to long, float, or double
- long to float or double
- float to double

Widening primitive conversions do not lose information about the overall magnitude of a numeric value. Indeed, conversions widening from an integral type to another integral type and from `float` to `double` do not lose any information at all; the numeric value is preserved exactly. Conversions widening from `float` to `double` in `strictfp` expressions also preserve the numeric value exactly; however, such conversions that are not `strictfp` may lose information about the overall magnitude of the converted value.

Conversion of an `int` or a `long` value to `float`, or of a `long` value to `double`, may result in *loss of precision*-that is, the result may lose some of the least significant bits of the value. In this case, the resulting floating-point value will be a correctly rounded version of the integer value, using IEEE 754 round-to-nearest mode ([§4.2.4](#)).

A widening conversion of a signed integer value to an integral type *T* simply sign-extends the two's-complement representation of the integer value to fill the wider format. A widening conversion of a character to an integral type *T* zero-extends the representation of the character value to fill the wider format.

Despite the fact that loss of precision may occur, widening conversions among primitive types never result in a run-time exception

[\(§11\)](#).

Here is an example of a widening conversion that loses precision:

```
class Test {
    public static void main(String[] args) {
        int big = 1234567890;
        float approx = big;
        System.out.println(big - (int)approx);
    }
}
```

which prints:

-46

thus indicating that information was lost during the conversion from type `int` to type `float` because values of type `float` are not precise to nine significant digits.

5.1.3 Narrowing Primitive Conversions

The following 23 specific conversions on primitive types are called the *narrowing primitive conversions*:

- `byte` to `char`
- `short` to `byte` or `char`
- `char` to `byte` or `short`
- `int` to `byte`, `short`, or `char`
- `long` to `byte`, `short`, `char`, or `int`
- `float` to `byte`, `short`, `char`, `int`, or `long`
- `double` to `byte`, `short`, `char`, `int`, `long`, or `float`

Narrowing conversions may lose information about the overall magnitude of a numeric value and may also lose precision.

A narrowing conversion of a signed integer to an integral type *T* simply discards all but the *n* lowest order bits, where *n* is the number of bits used to represent type *T*. In addition to a possible loss of information about the magnitude of the numeric value, this may cause the sign of the resulting value to differ from the sign of the input value.

A narrowing conversion of a character to an integral type *T* likewise simply discards all but the *n* lowest order bits, where *n* is the number of bits used to represent type *T*. In addition to a possible loss of information about the magnitude of the numeric value, this may cause the resulting value to be a negative number, even though characters represent 16-bit unsigned integer values.

A narrowing conversion of a floating-point number to an integral type *T* takes two steps:

1. In the first step, the floating-point number is converted either to a `long`, if *T* is `long`, or to an `int`, if *T* is `byte`, `short`, `char`, or `int`, as follows:
 - If the floating-point number is NaN ([§4.2.3](#)), the result of the first step of the conversion is an `int` or `long` 0.
 - Otherwise, if the floating-point number is not an infinity, the floating-point value is rounded to an integer value *V*, rounding toward zero using IEEE 754 round-toward-zero mode ([§4.2.3](#)). Then there are two cases:
 - If *T* is `long`, and this integer value can be represented as a `long`, then the result of the first step is the `long` value *V*.
 - Otherwise, if this integer value can be represented as an `int`, then the result of the first step is the `int` value *V*.

- Otherwise, one of the following two cases must be true:
 - The value must be too small (a negative value of large magnitude or negative infinity), and the result of the first step is the smallest representable value of type `int` or `long`.
 - The value must be too large (a positive value of large magnitude or positive infinity), and the result of the first step is the largest representable value of type `int` or `long`.
- 2. In the second step:
 - If *T* is `int` or `long`, the result of the conversion is the result of the first step.
 - If *T* is `byte`, `char`, or `short`, the result of the conversion is the result of a narrowing conversion to type *T* ([§5.1.3](#)) of the result of the first step.

The example:

```
class Test {
    public static void main(String[] args) {
        float fmin = Float.NEGATIVE_INFINITY;
        float fmax = Float.POSITIVE_INFINITY;
        System.out.println("long: " + (long)fmin +
                           "..." + (long)fmax);
        System.out.println("int: " + (int)fmin +
                           "..." + (int)fmax);
        System.out.println("short: " + (short)fmin +
                           "..." + (short)fmax);
        System.out.println("char: " + (int)(char)fmin +
                           "..." + (int)(char)fmax);
        System.out.println("byte: " + (byte)fmin +
                           "..." + (byte)fmax);
    }
}
```

produces the output:

```
long: -9223372036854775808..9223372036854775807
int: -2147483648..2147483647
short: 0..-1
char: 0..65535
byte: 0..-1
```

The results for `char`, `int`, and `long` are unsurprising, producing the minimum and maximum representable values of the type.

The results for `byte` and `short` lose information about the sign and magnitude of the numeric values and also lose precision. The results can be understood by examining the low order bits of the minimum and maximum `int`. The minimum `int` is, in hexadecimal, `0x80000000`, and the maximum `int` is `0x7fffffff`. This explains the `short` results, which are the low 16 bits of these values, namely, `0x0000` and `0xffff`; it explains the `char` results, which also are the low 16 bits of these values, namely, `'\u0000'` and `'\uffff'`; and it explains the `byte` results, which are the low 8 bits of these values, namely, `0x00` and `0xff`.

Despite the fact that overflow, underflow, or other loss of information may occur, narrowing conversions among primitive types never result in a run-time exception ([§11](#)).

Here is a small test program that demonstrates a number of narrowing conversions that lose information:

```
class Test {
```

```

    public static void main(String[] args) {

        // A narrowing of int to short loses high bits:
        System.out.println("(short)0x12345678==0x" +
                           Integer.toHexString((short)
0x12345678));

        // A int value not fitting in byte changes sign and magnitude:
        System.out.println("(byte)255==" + (byte)255);

        // A float value too big to fit gives largest int value:
        System.out.println("(int)1e20f==" + (int)1e20f);

        // A NaN converted to int yields zero:
        System.out.println("(int)NaN==" + (int)Float.NaN);

        // A double value too large for float yields infinity:
        System.out.println("(float)-1e100==" + (float)-1e100);

        // A double value too small for float underflows to zero:
        System.out.println("(float)1e-50==" + (float)1e-50);

    }
}

```

This test program produces the following output:

```

(short)0x12345678==0x5678
(byte)255==-1
(int)1e20f==2147483647
(int)NaN==0
(float)-1e100==-Infinity
(float)1e-50==0.0

```

5.1.4 Widening Reference Conversions

The following conversions are called the *widening reference conversions*:

- From any class type *S* to any class type *T*, provided that *S* is a subclass of *T*. (An important special case is that there is a widening conversion to the class type `Object` from any other class type.)
- From any class type *S* to any interface type *K*, provided that *S* implements *K*.
- From the null type to any class type, interface type, or array type.
- From any interface type *J* to any interface type *K*, provided that *J* is a subinterface of *K*.
- From any interface type to type `Object`.
- From any array type to type `Object`.
- From any array type to type `Cloneable`.
- From any array type to type `java.io.Serializable`
- From any array type *SC*[] to any array type *TC*[], provided that *SC* and *TC* are reference types and there is a widening conversion from *SC* to *TC*.

Such conversions never require a special action at run time and therefore never throw an exception at run time. They consist simply in regarding a reference as having some other type in a manner that can be proved correct at compile time.

See [§8](#) for the detailed specifications for classes, [§9](#) for interfaces, and [§10](#) for arrays.

5.1.5 Narrowing Reference Conversions

The following conversions are called the *narrowing reference conversions*:

- From any class type *S* to any class type *T*, provided that *S* is a superclass of *T*. (An important special case is that there is a narrowing conversion from the class type `Object` to any other class type.)
- From any class type *S* to any interface type *K*, provided that *S* is not `final` and does not implement *K*. (An important special case is that there is a narrowing conversion from the class type `Object` to any interface type.)
- From type `Object` to any array type.
- From type `Object` to any interface type.
- From any interface type *J* to any class type *T* that is not `final`.
- From any interface type *J* to any class type *T* that is `final`, provided that *T* implements *J*.
- From any interface type *J* to any interface type *K*, provided that *J* is not a subinterface of *K* and there is no method name *m* such that *J* and *K* both contain a method named *m* with the same signature but different return types.
- From any array type *SC*[] to any array type *TC*[], provided that *SC* and *TC* are reference types and there is a narrowing conversion from *SC* to *TC*.

Such conversions require a test at run time to find out whether the actual reference value is a legitimate value of the new type. If not, then a `ClassCastException` is thrown.

5.1.6 String Conversions

There is a string conversion to type `String` from every other type, including the null type.

5.1.7 Forbidden Conversions

- There is no permitted conversion from any reference type to any primitive type.
- Except for the string conversions, there is no permitted conversion from any primitive type to any reference type.
- There is no permitted conversion from the null type to any primitive type.
- There is no permitted conversion to the null type other than the identity conversion.
- There is no permitted conversion to the type `boolean` other than the identity conversion.
- There is no permitted conversion from the type `boolean` other than the identity conversion and string conversion.
- There is no permitted conversion other than string conversion from class type *S* to a different class type *T* if *S* is not a subclass of *T* and *T* is not a subclass of *S*.
- There is no permitted conversion from class type *S* to interface type *K* if *S* is `final` and does not implement *K*.
- There is no permitted conversion from class type *S* to any array type if *S* is not `Object`.
- There is no permitted conversion other than string conversion from interface type *J* to class type *T* if *T* is `final` and does not implement *J*.
- There is no permitted conversion from interface type *J* to interface type *K* if *J* and *K* contain methods with the same signature but different return types.
- There is no permitted conversion from any array type to any class type other than `Object` or `String`.
- There is no permitted conversion from any array type to any interface type, except to the interface types `java.io.Serializable` and `Cloneable`, which are implemented by all arrays.
- There is no permitted conversion from array type *SC*[] to array type *TC*[] if there is no permitted conversion other than a string conversion from *SC* to *TC*.

5.1.8 Value Set Conversion

Value set conversion is the process of mapping a floating-point value from one value set to another without changing its type.

Within an expression that is not FP-strict ([§15.4](#)), value set conversion provides choices to an implementation of the Java

programming language:

- If the value is an element of the float-extended-exponent value set, then the implementation may, at its option, map the value to the nearest element of the float value set. This conversion may result in overflow (in which case the value is replaced by an infinity of the same sign) or underflow (in which case the value may lose precision because it is replaced by a denormalized number or zero of the same sign).
- If the value is an element of the double-extended-exponent value set, then the implementation may, at its option, map the value to the nearest element of the double value set. This conversion may result in overflow (in which case the value is replaced by an infinity of the same sign) or underflow (in which case the value may lose precision because it is replaced by a denormalized number or zero of the same sign).

Within an FP-strict expression (§15.4), value set conversion does not provide any choices; every implementation must behave in the same way:

- If the value is of type `float` and is not an element of the float value set, then the implementation must map the value to the nearest element of the float value set. This conversion may result in overflow or underflow.
- If the value is of type `double` and is not an element of the double value set, then the implementation must map the value to the nearest element of the double value set. This conversion may result in overflow or underflow.

Within an FP-strict expression, mapping values from the float-extended-exponent value set or double-extended-exponent value set is necessary only when a method is invoked whose declaration is not FP-strict and the implementation has chosen to represent the result of the method invocation as an element of an extended-exponent value set.

Whether in FP-strict code or code that is not FP-strict, value set conversion always leaves unchanged any value whose type is neither `float` nor `double`.

5.2 Assignment Conversion

Assignment conversion occurs when the value of an expression is assigned (§15.26) to a variable: the type of the expression must be converted to the type of the variable. Assignment contexts allow the use of an identity conversion (§5.1.1), a widening primitive conversion (§5.1.2), or a widening reference conversion (§5.1.4). In addition, a narrowing primitive conversion may be used if all of the following conditions are satisfied:

- The expression is a constant expression of type `byte`, `short`, `char` or `int`.
- The type of the variable is `byte`, `short`, or `char`.
- The value of the expression (which is known at compile time, because it is a constant expression) is representable in the type of the variable.

If the type of the expression cannot be converted to the type of the variable by a conversion permitted in an assignment context, then a compile-time error occurs.

If the type of the variable is `float` or `double`, then value set conversion is applied after the type conversion:

- If the value is of type `float` and is an element of the float-extended-exponent value set, then the implementation must map the value to the nearest element of the float value set. This conversion may result in overflow or underflow.
- If the value is of type `double` and is an element of the double-extended-exponent value set, then the implementation must map the value to the nearest element of the double value set. This conversion may result in overflow or underflow.

If the type of an expression can be converted to the type of a variable by assignment conversion, we say the expression (or its value) is *assignable* to the variable or, equivalently, that the type of the expression is *assignment compatible* with the type of the variable.

An assignment conversion never causes an exception. (Note, however, that an assignment may result in an exception in a special case involving array elements -see [§10.10](#) and [§15.26.1](#).)

The compile-time narrowing of constants means that code such as:

```
byte theAnswer = 42;
```

is allowed. Without the narrowing, the fact that the integer literal 42 has type `int` would mean that a cast to `byte` would be required:

```
byte theAnswer = (byte)42;           // cast is permitted but not required
```

A value of primitive type must not be assigned to a variable of reference type; an attempt to do so will result in a compile-time error. A value of type `boolean` can be assigned only to a variable of type `boolean`.

The following test program contains examples of assignment conversion of primitive values:

```
class Test {
    public static void main(String[] args) {
        short s = 12;           // narrow 12 to short
        float f = s;            // widen short to float
        System.out.println("f=" + f);

        char c = '\u0123';
        long l = c;              // widen char to long
        System.out.println("l=0x" + Long.toString(l,16));

        f = 1.23f;
        double d = f;            // widen float to double
        System.out.println("d=" + d);
    }
}
```

It produces the following output:

```
f=12.0
l=0x123
d=1.2300000190734863
```

The following test, however, produces compile-time errors:

```
class Test {
    public static void main(String[] args) {
        short s = 123;
        char c = s;              // error: would require cast
        s = c;                   // error: would require cast
    }
}
```

because not all `short` values are `char` values, and neither are all `char` values `short` values.

A value of the null type (the null reference is the only such value) may be assigned to any reference type, resulting in a null

Assignment of a value of compile-time reference type S (source) to a variable of compile-time reference type T (target) is checked as follows:

- If S is a class type:
 - If T is a class type, then S must either be the same class as T , or S must be a subclass of T , or a compile-time error occurs.
 - If T is an interface type, then S must implement interface T , or a compile-time error occurs.
 - If T is an array type, then a compile-time error occurs.
- If S is an interface type:
 - If T is a class type, then T must be `Object`, or a compile-time error occurs.
 - If T is an interface type, then T must be either the same interface as S or a superinterface of S , or a compile-time error occurs.
 - If T is an array type, then a compile-time error occurs.
- If S is an array type $SC[]$, that is, an array of components of type SC :
 - If T is a class type, then T must be `Object`, or a compile-time error occurs.
 - If T is an interface type, then a compile-time error occurs unless T is the type `java.io.Serializable` or the type `Cloneable`, the only interfaces implemented by arrays.
 - If T is an array type $TC[]$, that is, an array of components of type TC , then a compile-time error occurs unless one of the following is true:
 - TC and SC are the same primitive type.
 - TC and SC are both reference types and type SC is assignable to TC , as determined by a recursive application of these compile-time rules for assignability.

See §8 for the specification of classes, §9 for interfaces, and §10 for arrays.

The following test program illustrates assignment conversions on reference values, but fails to compile because it violates the preceding rules, as described in its comments. This example should be compared to the preceding one.

[illegible]

```

        c = p;                                // p might not implement
    Colorable
    }
}

```

Here is another example involving assignment of array objects:

```

class Point { int x, y; }
class ColoredPoint extends Point { int color; }
class Test {
    public static void main(String[] args) {
        long[] veclong = new long[100];
        Object o = veclong;                    // okay
        Long l = veclong;                      // compile-
time error
        short[] vecshort = veclong;           // compile-
time error
        Point[] pvec = new Point[100];
        ColoredPoint[] cpvec = new ColoredPoint[100];
        pvec = cpvec;                          // okay
        pvec[0] = new Point();                 // okay at
compile time,
                                              // but would
throw an
                                              // exception
at run time
        cpvec = pvec;                         // compile-
time error
    }
}

```

In this example:

- The value of `vec long` cannot be assigned to a `Long` variable, because `Long` is a class type other than `Object`. An array can be assigned only to a variable of a compatible array type, or to a variable of type `Object`.
- The value of `vec long` cannot be assigned to `vecshort`, because they are arrays of primitive type, and `short` and `long` are not the same primitive type.
- The value of `cpvec` can be assigned to `pvec`, because any reference that could be the value of an expression of type `ColoredPoint` can be the value of a variable of type `Point`. The subsequent assignment of the new `Point` to a component of `pvec` then would throw an `ArrayStoreException` (if the program were otherwise corrected so that it could be compiled), because a `ColoredPoint` array can't have an instance of `Point` as the value of a component.
- The value of `pvec` cannot be assigned to `cpvec`, because not every reference that could be the value of an expression of type `ColoredPoint` can correctly be the value of a variable of type `Point`. If the value of `pvec` at run time were a reference to an instance of `Point[]`, and the assignment to `cpvec` were allowed, a simple reference to a component of `cpvec`, say, `cpvec[0]`, could return a `Point`, and a `Point` is not a `ColoredPoint`. Thus to allow such an assignment would allow a violation of the type system. A cast may be used ([§5.5](#), [§15.16](#)) to ensure that `pvec` references a `ColoredPoint[]`:

```

cpvec = (ColoredPoint[])pvec;                // okay, but may throw an
                                              // exception at run time

```

5.3 Method Invocation Conversion

Method invocation conversion is applied to each argument value in a method or constructor invocation (§15.9, §15.12): the type of the argument expression must be converted to the type of the corresponding parameter. Method invocation contexts allow the use of an identity conversion (§5.1.1), a widening primitive conversion (§5.1.2), or a widening reference conversion (§5.1.4).

If the type of an argument expression is either `float` or `double`, then value set conversion (§5.1.8) is applied after the type conversion:

- If an argument value of type `float` is an element of the float-extended-exponent value set, then the implementation must map the value to the nearest element of the float value set. This conversion may result in overflow or underflow.
- If an argument value of type `double` is an element of the double-extended-exponent value set, then the implementation must map the value to the nearest element of the double value set. This conversion may result in overflow or underflow.

Method invocation conversions specifically do not include the implicit narrowing of integer constants which is part of assignment conversion (§5.2). The designers of the Java programming language felt that including these implicit narrowing conversions would add additional complexity to the overloaded method matching resolution process (§15.12.2). Thus, the example:

```
class Test {
    static int m(byte a, int b) { return a+b; }
    static int m(short a, short b) { return a-b; }
    public static void main(String[] args) {
        System.out.println(m(12,
2));
compile-time error
    }
}
```

causes a compile-time error because the integer literals 12 and 2 have type `int`, so neither method `m` matches under the rules of (§15.12.2). A language that included implicit narrowing of integer constants would need additional rules to resolve cases like this example.

5.4 String Conversion

String conversion applies only to the operands of the binary `+` operator when one of the arguments is a `String`. In this single special case, the other argument to the `+` is converted to a `String`, and a new `String` which is the concatenation of the two strings is the result of the `+`. String conversion is specified in detail within the description of the string concatenation `+` operator (§15.18.1).

5.5 Casting Conversion

Casting conversion is applied to the operand of a cast operator (§15.16): the type of the operand expression must be converted to the type explicitly named by the cast operator. Casting contexts allow the use of an identity conversion (§5.1.1), a widening primitive conversion (§5.1.2), a narrowing primitive conversion (§5.1.3), a widening reference conversion (§5.1.4), or a narrowing reference conversion (§5.1.5). Thus casting conversions are more inclusive than assignment or method invocation conversions: a cast can do any permitted conversion other than a string conversion.

Value set conversion (§5.1.8) is applied after the type conversion.

Some casts can be proven incorrect at compile time; such casts result in a compile-time error.

A value of a primitive type can be cast to another primitive type by identity conversion, if the types are the same, or by a widening primitive conversion or a narrowing primitive conversion.

A value of a primitive type cannot be cast to a reference type by casting conversion, nor can a value of a reference type be cast to a primitive type.

The remaining cases involve conversion between reference types. The detailed rules for compile-time correctness checking of a casting conversion of a value of compile-time reference type *S* (source) to a compile-time reference type *T* (target) are as follows:

- If *S* is a class type:
 - If *T* is a class type, then *S* and *T* must be related classes—that is, *S* and *T* must be the same class, or *S* a subclass of *T*, or *T* a subclass of *S*; otherwise a compile-time error occurs.
 - If *T* is an interface type:
 - If *S* is not a `final` class (§8.1.1), then the cast is always correct at compile time (because even if *S* does not implement *T*, a subclass of *S* might).
 - If *S* is a `final` class (§8.1.1), then *S* must implement *T*, or a compile-time error occurs.
 - If *T* is an array type, then *S* must be the class `Object`, or a compile-time error occurs.
- If *S* is an interface type:
 - If *T* is an array type, then *T* must implement *S*, or a compile-time error occurs.
 - If *T* is a class type that is not `final` (§8.1.1), then the cast is always correct at compile time (because even if *T* does not implement *S*, a subclass of *T* might).
 - If *T* is an interface type and if *T* and *S* contain methods with the same signature (§8.4.2) but different return types, then a compile-time error occurs.
- If *S* is an array type *SC*[], that is, an array of components of type *SC*:
 - If *T* is a class type, then if *T* is not `Object`, then a compile-time error occurs (because `Object` is the only class type to which arrays can be assigned).
 - If *T* is an interface type, then a compile-time error occurs unless *T* is the type `java.io.Serializable` or the type `Cloneable`, the only interfaces implemented by arrays.
 - If *T* is an array type *TC*[], that is, an array of components of type *TC*, then a compile-time error occurs unless one of the following is true:
 - *TC* and *SC* are the same primitive type.
 - *TC* and *SC* are reference types and type *SC* can be cast to *TC* by a recursive application of these compile-time rules for casting.

See §8 for the specification of classes, §9 for interfaces, and §10 for arrays.

If a cast to a reference type is not a compile-time error, there are two cases:

- The cast can be determined to be correct at compile time. A cast from the compile-time type *S* to compile-time type *T* is correct at compile time if and only if *S* can be converted to *T* by assignment conversion (§5.2).
- The cast requires a run-time validity check. If the value at run time is `null`, then the cast is allowed. Otherwise, let *R* be the class of the object referred to by the run-time reference value, and let *T* be the type named in the cast operator. A cast conversion must check, at run time, that the class *R* is assignment compatible with the type *T*, using the algorithm specified in §5.2 but using the class *R* instead of the compile-time type *S* as specified there. (Note that *R* cannot be an interface when these rules are first applied for any given cast, but *R* may be an interface if the rules are applied recursively because the run-time reference value may refer to an array whose element type is an interface type.) The modified algorithm is shown here:
 - If *R* is an ordinary class (not an array class):
 - If *T* is a class type, then *R* must be either the same class (§4.3.4) as *T* or a subclass of *T*, or a run-time exception is thrown.
 - If *T* is an interface type, then *R* must implement (§8.1.4) interface *T*, or a run-time exception is thrown.
 - If *T* is an array type, then a run-time exception is thrown.
 - If *R* is an interface:
 - If *T* is a class type, then *T* must be `Object` (§4.3.2), or a run-time exception is thrown.
 - If *T* is an interface type, then *R* must be either the same interface as *T* or a subinterface of *T*, or a run-time

exception is thrown.

- If T is an array type, then a run-time exception is thrown.
- If R is a class representing an array type $RC[]$ -that is, an array of components of type RC :
 - If T is a class type, then T must be `Object` ([§4.3.2](#)), or a run-time exception is thrown.
 - If T is an interface type, then a run-time exception is thrown unless T is the type `java.io.Serializable` or the type `Cloneable`, the only interfaces implemented by arrays (this case could slip past the compile-time checking if, for example, a reference to an array were stored in a variable of type `Object`).
 - If T is an array type $TC[]$, that is, an array of components of type TC , then a run-time exception is thrown unless one of the following is true:
 - TC and RC are the same primitive type.
 - TC and RC are reference types and type RC can be cast to TC by a recursive application of these run-time rules for casting.

If a run-time exception is thrown, it is a `ClassCastException`.

Here are some examples of casting conversions of reference types, similar to the example in [§5.2](#):

```
public class Point { int x, y; }
public interface Colorable { void setColor(int color); }
public class ColoredPoint extends Point implements Colorable
{
    int color;
    public void setColor(int color) { this.color = color; }
}
final class EndPoint extends Point { }
class Test {
    public static void main(String[] args) {
        Point p = new Point();
        ColoredPoint cp = new ColoredPoint();
        Colorable c;

        // The following may cause errors at run time because
        // we cannot be sure they will succeed; this possibility
        // is suggested by the casts:
        cp = (ColoredPoint)p;           // p might not reference an
                                        // object which is a
ColoredPoint                               // or a subclass of
ColoredPoint
        c = (Colorable)p;               // p might not be Colorable

        // The following are incorrect at compile time because
        // they can never succeed as explained in the text:
        Long l = (Long)p;               // compile-time error #1
        EndPoint e = new EndPoint();
        c = (Colorable)e;               // compile-time error #2
    }
}
```

Here the first compile-time error occurs because the class types `Long` and `Point` are unrelated (that is, they are not the same, and neither is a subclass of the other), so a cast between them will always fail.

The second compile-time error occurs because a variable of type `EndPoint` can never reference a value that implements the interface `Colorable`. This is because `EndPoint` is a `final` type, and a variable of a `final` type always holds a value of the same run-time type as its compile-time type. Therefore, the run-time type of variable `e` must be exactly the type `EndPoint`, and type `EndPoint` does not implement `Colorable`.

Here is an example involving arrays ([§10](#)):

```
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
    public String toString() { return "("+x+","+y+""; }
}
public interface Colorable { void setColor(int color); }
public class ColoredPoint extends Point implements Colorable
{
    int color;
    ColoredPoint(int x, int y, int color) {
        super(x, y); setColor(color);
    }
    public void setColor(int color) { this.color = color; }
    public String toString() {
        return super.toString() + "@" + color;
    }
}
class Test {
    public static void main(String[] args) {
        Point[] pa = new ColoredPoint[4];
        pa[0] = new ColoredPoint(2, 2, 12);
        pa[1] = new ColoredPoint(4, 5, 24);
        ColoredPoint[] cpa = (ColoredPoint[])pa;
        System.out.print("cpa: {");
        for (int i = 0; i < cpa.length; i++)
            System.out.print((i == 0 ? " " : ", ") + cpa[i]);
        System.out.println(" }");
    }
}
```

This example compiles without errors and produces the output:

```
cpa: { (2,2)@12, (4,5)@24, null, null }
```

The following example uses casts to compile, but it throws exceptions at run time, because the types are incompatible:

```
public class Point { int x, y; }
public interface Colorable { void setColor(int color); }
public class ColoredPoint extends Point implements Colorable
{
    int color;
    public void setColor(int color) { this.color = color; }
}
class Test {
    public static void main(String[] args) {
```

```

        Point[] pa = new Point[100];
        // The following line will throw a ClassCastException:
        ColoredPoint[] cpa = (ColoredPoint[])pa;
        System.out.println(cpa[0]);
        int[] shortvec = new int[2];
        Object o = shortvec;
        // The following line will throw a ClassCastException:
        Colorable c = (Colorable)o;
        c.setColor(0);
    }

}

```

5.6 Numeric Promotions

Numeric promotion is applied to the operands of an arithmetic operator. Numeric promotion contexts allow the use of an identity conversion ([§5.1.1](#)) or a widening primitive conversion ([§5.1.2](#)).

Numeric promotions are used to convert the operands of a numeric operator to a common type so that an operation can be performed. The two kinds of numeric promotion are unary numeric promotion ([§5.6.1](#)) and binary numeric promotion ([§5.6.2](#)). The analogous conversions in C are called "the usual unary conversions" and "the usual binary conversions."

Numeric promotion is not a general feature of the Java programming language, but rather a property of the specific definitions of the built-in operations.

5.6.1 Unary Numeric Promotion

Some operators apply *unary numeric promotion* to a single operand, which must produce a value of a numeric type:

- If the operand is of compile-time type `byte`, `short`, or `char`, unary numeric promotion promotes it to a value of type `int` by a widening conversion ([§5.1.2](#)).
- Otherwise, a unary numeric operand remains as is and is not converted.

In either case, value set conversion ([§5.1.8](#)) is then applied.

Unary numeric promotion is performed on expressions in the following situations:

- Each dimension expression in an array creation expression ([§15.10](#))
- The index expression in an array access expression ([§15.13](#))
- The operand of a unary plus operator `+` ([§15.15.3](#))
- The operand of a unary minus operator `-` ([§15.15.4](#))
- The operand of a bitwise complement operator `~` ([§15.15.5](#))
- Each operand, separately, of a shift operator `>>`, `>>>`, or `<<` ([§15.19](#)); therefore a `long` shift distance (right operand) does not promote the value being shifted (left operand) to `long`

Here is a test program that includes examples of unary numeric promotion:

```

class Test {
    public static void main(String[] args) {

```

```

        byte b = 2;
        int a[] = new int[b];    // dimension expression promotion
        char c = '\u0001';
        a[c] = 1;                // index expression promotion
        a[0] = -c;               // unary - promotion
        System.out.println("a: " + a[0] + ", " + a[1]);
        b = -1;
        int i = ~b;              // bitwise complement

promotion
        System.out.println("~0x" + Integer.toHexString(b)
                                + "==0x" + Integer.
toHexString(i));
        i = b << 4L;            // shift promotion (left operand)
        System.out.println("0x" + Integer.toHexString(b)
                                + "<<4L==0x" + Integer.toHexString
(i));
    }
}

```

This test program produces the output:

```

a: -1,1
~0xffffffff==0x0
0xffffffff<<4L==0xffffffff0

```

5.6.2 Binary Numeric Promotion

When an operator applies *binary numeric promotion* to a pair of operands, each of which must denote a value of a numeric type, the following rules apply, in order, using widening conversion ([§5.1.2](#)) to convert operands as necessary:

- If either operand is of type `double`, the other is converted to `double`.
- Otherwise, if either operand is of type `float`, the other is converted to `float`.
- Otherwise, if either operand is of type `long`, the other is converted to `long`.
- Otherwise, both operands are converted to type `int`.

After the type conversion, if any, value set conversion ([§5.1.8](#)) is applied to each operand.

Binary numeric promotion is performed on the operands of certain operators:

- The multiplicative operators `*`, `/` and `%` ([§15.17](#))
- The addition and subtraction operators for numeric types `+` and `-` ([§15.18.2](#))
- The numerical comparison operators `<`, `<=`, `>`, and `>=` ([§15.20.1](#))
- The numerical equality operators `==` and `!=` ([§15.21.1](#))
- The integer bitwise operators `&`, `^`, and `|` ([§15.22.1](#))
- In certain cases, the conditional operator `?:` ([§15.25](#))

An example of binary numeric promotion appears above in [§5.1](#). Here is another:

```

class Test {
    public static void main(String[] args) {
        int i = 0;
        float f = 1.0f;
    }
}

```

```

        double d = 2.0;
        // First int*float is promoted to float*float, then
        // float==double is promoted to double==double:
        if (i * f == d)
            System.out.println("oops");
        // A char&byte is promoted to int&int:
        byte b = 0x1f;
        char c = 'G';
        int control = c & b;
        System.out.println(Integer.toHexString(control));
        // Here int:float is promoted to float:float:
        f = (b==0) ? i : 4.0f;
        System.out.println(1.0/f);
    }
}

```

which produces the output:

```

7
0.25

```

The example converts the ASCII character **G** to the ASCII control-G (BEL), by masking off all but the low 5 bits of the character. The 7 is the numeric value of this control character.