Table of Contents

Chapter 5. Conversions and Contexts

Every expression written in the Java programming language either produces no result (§15.1) or has a type that can be deduced at compile time (§15.3). When an expression appears in most contexts, it must be *compatible* with a type expected in that context; this type is called the *target type*. For convenience, compatibility of an expression with its surrounding context is facilitated in two ways:

- First, for some expressions, termed *poly expressions* (§15.2), the deduced type can be influenced by the target type. The same expression can have different types in different contexts.
- Second, after the type of the expression has been deduced, an implicit *conversion* from the type of the expression to the target type can sometimes be performed.

If neither strategy is able to produce the appropriate type, a compile-time error occurs.

The rules determining whether an expression is a poly expression, and if so, its type and compatibility in a particular context, vary depending on the kind of context and the form of the expression. In addition to influencing the type of the expression, the target type may in some cases influence the run time behavior of the expression in order to produce a value of the appropriate type.

Similarly, the rules determining whether a target type allows an implicit conversion vary depending on the kind of context, the type of the expression, and, in one special case, the value of a constant expression (§15.28). A 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.

Example 5.0-1. Conversions at Compile Time and Run Time

- *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 non-trivial 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.*

The conversions 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
- Boxing conversions
- Unboxing conversions
- Unchecked conversions
- Capture conversions
- String conversions
- Value set conversions

There are six kinds of *conversion contexts* in which poly expressions may be influenced by context or implicit conversions may occur. Each kind of context has different rules for poly expression typing and allows conversions in some of the categories above but not others. The contexts are:

- Assignment contexts (§5.2, §15.26), in which an expression's value is bound to a named variable. Primitive and reference types are subject to widening, values may be boxed or unboxed, and some primitive constant expressions may be subject to narrowing. An unchecked conversion may also occur.
- Strict invocation contexts (§5.3, §15.9, §15.12), in which an argument is bound to a formal parameter of a constructor or method. Widening primitive, widening reference, and unchecked conversions may occur.
- Loose invocation contexts (§5.3, §15.9, §15.12), in which, like strict invocation contexts, an argument is bound to a formal parameter. Method or constructor invocations may provide this context if no applicable declaration can be found using only strict invocation contexts. In addition to widening and unchecked conversions, this context allows boxing and unboxing conversions to occur.
- String contexts (§5.4, §15.18.1), in which a value of any type is converted to an object of type `String`.
- Casting contexts (§5.5), in which an expression's value is converted to a type explicitly specified by a cast operator (§15.16). Casting contexts are more inclusive than assignment or loose invocation contexts, allowing any specific conversion other than a string conversion, but certain casts to a reference type are checked for correctness at run time.
- Numeric contexts (§5.6), in which the operands of a numeric operator may be widened to a common type so that an operation can be performed.

The term "conversion" is also used to describe, without being specific, any conversions allowed in a particular context. For example, we say that an expression that is the initializer of a local variable is subject to "assignment conversion", meaning that a specific conversion will be implicitly chosen for that expression according to the rules for the assignment context.

Example 5.0-2. Conversions In Various Contexts

```
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);

        // 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);
    }
}
```

This program 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 13 categories.

5.1.1. Identity Conversion

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.

5.1.2. Widening Primitive Conversion

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`

A widening primitive conversion does not lose information about the overall magnitude of a numeric value in the following cases, where the numeric value is preserved exactly:

- from an integral type to another integral type
- from `byte`, `short`, or `char` to a floating point type
- from `int` to `double`
- from `float` to `double` in a `strictfp` expression (§15.4)

A widening primitive conversion from `float` to `double` that is not `strictfp` may lose information about the overall magnitude of the converted value.

A widening primitive conversion from `int` to `float`, or from `long` to `float`, or from `long` 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 `char` to an integral type T zero-extends the representation of the `char` value to fill the wider format.

Despite the fact that loss of precision may occur, a widening primitive conversion never results in a run-time exception (§11.1.1).

Example 5.1.2-1. Widening Primitive Conversion

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

This program 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 Conversion

22 specific conversions on primitive types are called the *narrowing primitive conversions*:

- `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`

A narrowing primitive conversion may lose information about the overall magnitude of a numeric value and may also lose precision and range.

A narrowing primitive conversion from `double` to `float` is governed by the IEEE 754 rounding rules (§4.2.4). This conversion can lose precision, but also lose range, resulting in a `float` zero from a nonzero `double` and a `float` infinity from a finite `double`. A `double` NaN is converted to a `float` NaN and a `double` infinity is converted to the same-signed `float` infinity.

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 `char` 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 chars 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:
     a. 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`.
     b. 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:
     a. 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`.
     b. 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.

Despite the fact that overflow, underflow, or other loss of information may occur, a narrowing primitive conversion never results in a run-time exception (§11.1.1).

Example 5.1.3-1. Narrowing Primitive Conversion

```
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);
    }
}
```

This program 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`.

Example 5.1.3-2. Narrowing Primitive 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));
        // An int value too big for 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 program produces the 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 and Narrowing Primitive Conversion

The following conversion combines both widening and narrowing primitive conversions:

- `byte` to `char`

First, the `byte` is converted to an `int` via widening primitive conversion (§5.1.2), and then the resulting `int` is converted to a `char` by narrowing primitive conversion (§5.1.3).

5.1.5. Widening Reference Conversion

A *widening reference* conversion exists from any reference type S to any reference type T, provided S is a subtype (§4.10) of T.

Widening reference 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.

5.1.6. Narrowing Reference Conversion

Six kinds of conversions are called the *narrowing reference conversions*:

- From any reference type S to any reference type T, provided that S is a proper supertype of T (§4.10).

  An important special case is that there is a narrowing reference conversion from the class type `Object` to any other reference type (§4.12.4).
- From any class type C to any non-parameterized interface type K, provided that C is not `final` and does not implement K.
- From any interface type J to any non-parameterized class type C that is not `final`.
- From any interface type J to any non-parameterized interface type K, provided that J is not a subinterface of K.
- From the interface types `Cloneable` and `java.io.Serializable` to any array type T`[]`.
- From any array type SC`[]` to any array type TC`[]`, provided that SC and TC are reference types and there is a narrowing reference 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.

5.1.7. Boxing Conversion

Boxing conversion converts expressions of primitive type to corresponding expressions of reference type. Specifically, the following nine conversions are called the *boxing conversions*:

- From type `boolean` to type `Boolean`
- From type `byte` to type `Byte`
- From type `short` to type `Short`
- From type `char` to type `Character`
- From type `int` to type `Integer`
- From type `long` to type `Long`
- From type `float` to type `Float`
- From type `double` to type `Double`
- From the null type to the null type

  This rule is necessary because the conditional operator (§15.25) applies boxing conversion to the types of its operands, and uses the result in further calculations.

At run time, boxing conversion proceeds as follows:

- If `p` is a value of type `boolean`, then boxing conversion converts `p` into a reference `r` of class and type `Boolean`, such that `r.booleanValue() == p`
- If `p` is a value of type `byte`, then boxing conversion converts `p` into a reference `r` of class and type `Byte`, such that `r.byteValue() == p`
- If `p` is a value of type `char`, then boxing conversion converts `p` into a reference `r` of class and type `Character`, such that `r.charValue() == p`
- If `p` is a value of type `short`, then boxing conversion converts `p` into a reference `r` of class and type `Short`, such that `r.shortValue() == p`
- If `p` is a value of type `int`, then boxing conversion converts `p` into a reference `r` of class and type `Integer`, such that `r.intValue() == p`
- If `p` is a value of type `long`, then boxing conversion converts `p` into a reference `r` of class and type `Long`, such that `r.longValue() == p`
- If `p` is a value of type `float` then:
  - If `p` is not NaN, then boxing conversion converts `p` into a reference `r` of class and type `Float`, such that `r.floatValue()` evaluates to `p`
  - Otherwise, boxing conversion converts `p` into a reference `r` of class and type `Float` such that `r.isNaN()` evaluates to `true`
- If `p` is a value of type `double`, then:
  - If `p` is not NaN, boxing conversion converts `p` into a reference `r` of class and type `Double`, such that `r.doubleValue()` evaluates to `p`
  - Otherwise, boxing conversion converts `p` into a reference `r` of class and type `Double` such that `r.isNaN()` evaluates to `true`
- If `p` is a value of any other type, boxing conversion is equivalent to an identity conversion (§5.1.1).

If the value `p` being boxed is an integer literal of type `int` between `-128` and `127` inclusive (§3.10.1), or the boolean literal `true` or `false` (§3.10.3), or a character literal between `'\u0000'` and `'\u007f'` inclusive (§3.10.4), then let `a` and `b` be the results of any two boxing conversions of `p`. It is always the case that `a == b`.

Ideally, boxing a primitive value would always yield an identical reference. In practice, this may not be feasible using existing implementation techniques. The rule above is a pragmatic compromise, requiring that certain common values always be boxed into indistinguishable objects. The implementation may cache these, lazily or eagerly. For other values, the rule disallows any assumptions about the identity of the boxed values on the programmer's part. This allows (but does not require) sharing of some or all of these references. Notice that integer literals of type `long` are allowed, but not required, to be shared.

This ensures that in most common cases, the behavior will be the desired one, without imposing an undue performance penalty, especially on small devices. Less memory-limited implementations might, for example, cache all `char` and `short` values, as well as `int` and `long` values in the range of -32K to +32K.

A boxing conversion may result in an `OutOfMemoryError` if a new instance of one of the wrapper classes (`Boolean`, `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float`, or `Double`) needs to be allocated and

insufficient storage is available.

5.1.8. Unboxing Conversion

Unboxing conversion converts expressions of reference type to corresponding expressions of primitive type. Specifically, the following eight conversions are called the *unboxing conversions*:

- From type `Boolean` to type `boolean`
- From type `Byte` to type `byte`
- From type `Short` to type `short`
- From type `Character` to type `char`
- From type `Integer` to type `int`
- From type `Long` to type `long`
- From type `Float` to type `float`
- From type `Double` to type `double`

At run time, unboxing conversion proceeds as follows:

- If `r` is a reference of type `Boolean`, then unboxing conversion converts `r` into `r.booleanValue()`
- If `r` is a reference of type `Byte`, then unboxing conversion converts `r` into `r.byteValue()`
- If `r` is a reference of type `Character`, then unboxing conversion converts `r` into `r.charValue()`
- If `r` is a reference of type `Short`, then unboxing conversion converts `r` into `r.shortValue()`
- If `r` is a reference of type `Integer`, then unboxing conversion converts `r` into `r.intValue()`
- If `r` is a reference of type `Long`, then unboxing conversion converts `r` into `r.longValue()`
- If `r` is a reference of type `Float`, unboxing conversion converts `r` into `r.floatValue()`
- If `r` is a reference of type `Double`, then unboxing conversion converts `r` into `r.doubleValue()`
- If `r` is `null`, unboxing conversion throws a `NullPointerException`

A type is said to be *convertible to a numeric type* if it is a numeric type (§4.2), or it is a reference type that may be converted to a numeric type by unboxing conversion.

A type is said to be *convertible to an integral type* if it is an integral type, or it is a reference type that may be converted to an integral type by unboxing conversion.

5.1.9. Unchecked Conversion

Let G name a generic type declaration with *n* type parameters.

There is an *unchecked conversion* from the raw class or interface type (§4.8) G to any parameterized type of the form $G<T_1,...,T_n>$.

There is an *unchecked conversion* from the raw array type $G[]^k$ to any array type of the form $G<T_1,...,T_n>[]^k$. (The notation $[]^k$ indicates an array type of *k* dimensions.)

Use of an unchecked conversion causes a compile-time *unchecked warning* unless all type arguments T_i ($1 \le i \le n$) are unbounded wildcards (§4.5.1), or the unchecked warning is suppressed by the `SuppressWarnings` annotation (§9.6.4.5).

Unchecked conversion is used to enable a smooth interoperation of legacy code, written before the introduction of generic types, with libraries that have undergone a conversion to use genericity (a process we call generification). In such circumstances (most notably, clients of the Collections Framework in `java.util`), legacy code uses raw types (e.g. `Collection` instead of `Collection<String>`). Expressions of raw types are passed as arguments to library methods that use parameterized versions of those same types as the types of their corresponding formal parameters.

Such calls cannot be shown to be statically safe under the type system using generics. Rejecting such calls would invalidate large bodies of existing code, and prevent them from using newer versions of the libraries. This in turn, would discourage library vendors from taking advantage of genericity. To prevent such an unwelcome turn of events, a raw type may be converted to an arbitrary invocation of the generic type declaration to which the raw type refers. While the conversion is unsound, it is tolerated as a concession to practicality. An unchecked warning is issued in such cases.

5.1.10. Capture Conversion

Let G name a generic type declaration (§8.1.2, §9.1.2) with *n* type parameters $A_1,...,A_n$ with corresponding bounds $U_1,...,U_n$.

There exists a *capture conversion* from a parameterized type $G<T_1,...,T_n>$ (§4.5) to a parameterized type $G<S_1,...,S_n>$, where, for $1 \le i \le n$:

- If T_i is a wildcard type argument (§4.5.1) of the form `?`, then S_i is a fresh type variable whose upper bound is $U_i[A_1:=S_1,...,A_n:=S_n]$ and whose lower bound is the null type (§4.1).
- If T_i is a wildcard type argument of the form `? extends` B_i, then S_i is a fresh type variable whose upper bound is $glb(B_i, U_i[A_1:=S_1,...,A_n:=S_n])$ and whose lower bound is the null type.

  $glb(V_1,...,V_m)$ is defined as V_1 `&` ... `&` V_m.

  It is a compile-time error if, for any two classes (not interfaces) V_i and V_j, V_i is not a subclass of V_j or vice versa.
- If T_i is a wildcard type argument of the form `? super` B_i, then S_i is a fresh type variable whose upper bound is $U_i[A_1:=S_1,...,A_n:=S_n]$ and whose lower bound is B_i.
- Otherwise, $S_i = T_i$.

Capture conversion on any type other than a parameterized type (§4.5) acts as an identity conversion (§5.1.1).

Capture conversion is not applied recursively.

Capture conversion never requires a special action at run time and therefore never throws an exception at run time.

Capture conversion is designed to make wildcards more useful. To understand the motivation, let's begin by looking at the method `java.util.Collections.reverse()`:

```
public static void reverse(List<?> list);
```

The method reverses the list provided as a parameter. It works for any type of list, and so the use of the wildcard type `List<?>` as the type of the formal parameter is entirely appropriate.

Now consider how one would implement `reverse()`:

```
public static void reverse(List<?> list) { rev(list); }
private static <T> void rev(List<T> list) {
    List<T> tmp = new ArrayList<T>(list);
    for (int i = 0; i < list.size(); i++) {
        list.set(i, tmp.get(list.size() - i - 1));
    }
}
```

The implementation needs to copy the list, extract elements from the copy, and insert them into the original. To do this in a type-safe manner, we need to give a name, `T`, to the element type of the incoming list. We do this in the private service method `rev()`. This requires us to pass the incoming argument list, of type `List<?>`, as an argument to `rev()`. In general, `List<?>` is a list of unknown type. It is not a subtype of `List<T>`, for any type T. Allowing such a subtype relation would be unsound. Given the method:

```
public static <T> void fill(List<T> l, T obj)
```

the following code would undermine the type system:

```
List<String> ls = new ArrayList<String>();
List<?> l = ls;
Collections.fill(l, new Object());  // not legal - but assume it was!
String s = ls.get(0); // ClassCastException - ls contains
                      // Objects, not Strings.
```

So, without some special dispensation, we can see that the call from `reverse()` to `rev()` would be disallowed. If this were the case, the author of `reverse()` would be forced to write its signature as:

```
public static <T> void reverse(List<T> list)
```

This is undesirable, as it exposes implementation information to the caller. Worse, the designer of an API might reason that the signature using a wildcard is what the callers of the API require, and only later realize that a type safe implementation was precluded.

The call from `reverse()` to `rev()` is in fact harmless, but it cannot be justified on the basis of a general subtyping relation between `List<?>` and `List<T>`. The call is harmless, because the incoming argument is doubtless a list of some type (albeit an unknown one). If we can capture this unknown type in a type variable `X`, we can infer `T` to be `X`. That is the essence of capture conversion. The specification of course must cope with complications, like non-trivial (and possibly recursively defined) upper or lower bounds, the presence of multiple arguments etc.

Mathematically sophisticated readers will want to relate capture conversion to established type theory. Readers unfamiliar with type theory can skip this discussion - or else study a suitable text, such as Types and Programming Languages by Benjamin Pierce, and then revisit this section.

Here then is a brief summary of the relationship of capture conversion to established type theoretical notions. Wildcard types are a restricted form of existential types. Capture conversion corresponds loosely to an opening of a value of existential type. A capture conversion of an expression `e` can be thought of as an `open` of `e` in a scope that comprises the top level expression that encloses `e`.

The classical `open` operation on existentials requires that the captured type variable must not escape the opened expression. The `open` that corresponds to capture conversion is always on a scope sufficiently large that the captured type variable can never be visible outside that scope. The advantage of this scheme is that there is no need for a `close` operation, as defined in the paper On Variance-Based Subtyping for Parametric Types by Atsushi Igarashi and Mirko Viroli, in the proceedings of the 16th European Conference on Object Oriented Programming (ECOOP 2002). For a formal account of wildcards, see Wild FJ by Mads Torgersen, Erik Ernst and Christian Plesner Hansen, in the 12th workshop on Foundations of Object Oriented Programming (FOOL 2005).

5.1.11. String Conversion

Any type may be converted to type `String` by *string conversion*.

A value `x` of primitive type T is first converted to a reference value as if by giving it as an argument to an appropriate class instance creation expression (§15.9):

- If T is `boolean`, then use `new Boolean(x)`.
- If T is `char`, then use `new Character(x)`.
- If T is `byte`, `short`, or `int`, then use `new Integer(x)`.
- If T is `long`, then use `new Long(x)`.
- If T is `float`, then use `new Float(x)`.
- If T is `double`, then use `new Double(x)`.

This reference value is then converted to type `String` by string conversion.

Now only reference values need to be considered:

- If the reference is `null`, it is converted to the string "`null`" (four ASCII characters `n`, `u`, `l`, `l`).
- Otherwise, the conversion is performed as if by an invocation of the `toString` method of the referenced object with no arguments; but if the result of invoking the `toString` method is `null`, then the string "`null`" is used instead.

  The `toString` method is defined by the primordial class `Object` (§4.3.2). Many classes override it, notably `Boolean`, `Character`, `Integer`, `Long`, `Float`, `Double`, and `String`.

See §5.4 for details of the string context.

5.1.12. Forbidden Conversions

Any conversion that is not explicitly allowed is forbidden.

5.1.13. 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 Contexts

Assignment contexts allow the value of an expression to be 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 one of the following:

- an identity conversion (§5.1.1)
- a widening primitive conversion (§5.1.2)
- a widening reference conversion (§5.1.5)
- a boxing conversion (§5.1.7) optionally followed by a widening reference conversion
- an unboxing conversion (§5.1.8) optionally followed by a widening primitive conversion.

If, after the conversions listed above have been applied, the resulting type is a raw type (§4.8), an unchecked conversion (§5.1.9) may then be applied.

In addition, if the expression is a constant expression (§15.28) of type `byte`, `short`, `char`, or `int`:

- A narrowing primitive conversion may be used if the type of the variable is `byte`, `short`, or `char`, and the value of the constant expression is representable in the type of the variable.
- A narrowing primitive conversion followed by a boxing conversion may be used if the type of the variable is:
  - `Byte` and the value of the constant expression is representable in the type `byte`.
  - `Short` and the value of the constant expression is representable in the type `short`.
  - `Character` and the value of the constant expression is representable in the type `char`.

  The compile-time narrowing of constant expressions 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
  ```

Finally, 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 reference of that type.

It is a compile-time error if the chain of conversions contains two parameterized types that are not in the subtype relation (§4.10).

An example of such an illegal chain would be:

```
Integer, Comparable<Integer>, Comparable, Comparable<String>
```

The first three elements of the chain are related by widening reference conversion, while the last entry is derived from its predecessor by unchecked conversion. However, this is not a valid assignment conversion, because the chain contains two parameterized types, `Comparable<Integer>` and `Comparable<String>`, that are not subtypes.

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 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.

If the type of the variable is `float` or `double`, then value set conversion (§5.1.13) is applied to the value `v` that is the result of the conversion(s):

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

The only exceptions that may arise from conversions in an assignment context are:

- A **`ClassCastException` if, after the conversions above have been applied, the resulting value is an object which is not an instance of a subclass or subinterface of the erasure (§4.6) of the type of the**

variable.

This circumstance can only arise as a result of heap pollution (§4.12.2). In practice, implementations need only perform casts when accessing a field or method of an object of parameterized type when the erased type of the field, or the erased return type of the method, differ from its unerased type.

- **An `OutOfMemoryError` as a result of a boxing conversion.**
- **A `NullPointerException` as a result of an unboxing conversion on a null reference.**
- **An `ArrayStoreException` in special cases involving array elements or field access (§10.5, §15.26.1).**

Example 5.2-1. Assignment Conversion for Primitive Types

```
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);
    }
}
```

This program produces the output:

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

The following program, 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.

Example 5.2-2. Assignment Conversion for Reference Types

```
class Point { int x, y; }
class Point3D extends Point { int z; }
interface Colorable { void setColor(int color); }

class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
}

class Test {
    public static void main(String[] args) {
        // Assignments to variables of class type:
        Point p = new Point();
        p = new Point3D();
          // OK because Point3D is a subclass of Point
        Point3D p3d = p;
          // Error: will require a cast because a Point
          // might not be a Point3D (even though it is,
          // dynamically, in this example.)

        // Assignments to variables of type Object:
        Object o  = p;          // OK: any object to Object
        int[] a   = new int[3];
        Object o2 = a;          // OK: an array to Object

        // Assignments to variables of interface type:
        ColoredPoint cp = new ColoredPoint();
        Colorable c = cp;
          // OK: ColoredPoint implements Colorable

        // Assignments to variables of array type:
        byte[] b = new byte[4];
        a = b;
          // Error: these are not arrays of the same primitive type
        Point3D[] p3da = new Point3D[3];
        Point[] pa = p3da;
          // OK: since we can assign a Point3D to a Point
        p3da = pa;
          // Error: (cast needed) since a Point
          // can't be assigned to a Point3D
    }
}
```

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

```
class Point { int x, y; }
interface Colorable { void setColor(int color); }
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 p = new Point();
        ColoredPoint cp = new ColoredPoint();
        // Okay because ColoredPoint is a subclass of Point:
        p = cp;
        // Okay because ColoredPoint implements Colorable:
        Colorable c = cp;
        // The following cause compile-time errors because
        // we cannot be sure they will succeed, depending on
        // the run-time type of p; a run-time check will be
        // necessary for the needed narrowing conversion and
        // must be indicated by including a cast:
        cp = p;     // p might be neither a ColoredPoint
                    // nor a subclass of ColoredPoint
        c = p;      // p might not implement Colorable
    }
}
```

Example 5.2-3. Assignment Conversion for Array Types

```
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 `veclong` 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`, `Cloneable` or `java.io.Serializable`.*
- *The value of `veclong` 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 cannot 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;  // OK, but may throw an
                               // exception at run time
```

5.3. Invocation Contexts

Invocation contexts allow an argument value in a method or constructor invocation (§8.8.7.1, §15.9, §15.12) to be assigned to a corresponding formal parameter.

Strict invocation contexts allow the use of one of the following:

- an identity conversion (§5.1.1)
- a widening primitive conversion (§5.1.2)
- a widening reference conversion (§5.1.5)

Loose invocation contexts allow a more permissive set of conversions, because they are only used for a particular invocation if no applicable declaration can be found using strict invocation contexts. Loose invocation contexts allow the use of one of the following:

- an identity conversion (§5.1.1)
- a widening primitive conversion (§5.1.2)
- a widening reference conversion (§5.1.5)
- a boxing conversion (§5.1.7) optionally followed by widening reference conversion
- an unboxing conversion (§5.1.8) optionally followed by a widening primitive conversion

If, after the conversions listed for an invocation context have been applied, the resulting type is a raw type (§4.8), an unchecked conversion (§5.1.9) may then be applied.

A value of the null type (the null reference is the only such value) may be assigned to any reference type.

It is a compile-time error if the chain of conversions contains two parameterized types that are not in the subtype relation (§4.10).

If the type of the expression cannot be converted to the type of the parameter by a conversion permitted in a loose invocation context, then a compile-time error occurs.

If the type of an argument expression is either `float` or `double`, then value set conversion (§5.1.13) is applied after the conversion(s):

- 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.

The only exceptions that may arise in an invocation context are:

- **A `ClassCastException` if, after the type conversions above have been applied, the resulting value is an object which is not an instance of a subclass or subinterface of the erasure (§4.6) of the corresponding formal parameter type.**
- **An `OutOfMemoryError` as a result of a boxing conversion.**
- **A `NullPointerException` as a result of an unboxing conversion on a null reference.**

Neither strict nor loose invocation contexts include the implicit narrowing of integer constant expressions which is allowed in assignment contexts. The designers of the Java programming language felt that including these implicit narrowing conversions would add additional complexity to the rules of overload resolution (§15.12.2).

Thus, the program:

```
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 overload resolution. A language that included implicit narrowing of integer constant expressions would need additional rules to resolve cases like this example.

5.4. String Contexts

String contexts apply only to an operand of the binary `+` operator which is not a `String` when the other operand is a `String`.

The target type in these contexts is always `String`, and a string conversion (§5.1.11) of the non-`String` operand always occurs. Evaluation of the `+` operator then proceeds as specified in §15.18.1.

5.5. Casting Contexts

Casting contexts allow the operand of a cast operator (§15.16) to be converted to the type explicitly named by the cast operator.

Casting contexts allow the use of one of:

- an identity conversion (§5.1.1)
- a widening primitive conversion (§5.1.2)
- a narrowing primitive conversion (§5.1.3)
- a widening and narrowing primitive conversion (§5.1.4)
- a widening reference conversion (§5.1.5) optionally followed by either an unboxing conversion (§5.1.8) or an unchecked conversion (§5.1.9)
- a narrowing reference conversion (§5.1.6) optionally followed by either an unboxing conversion (§5.1.8) or an unchecked conversion (§5.1.9)
- a boxing conversion (§5.1.7) optionally followed by a widening reference conversion (§5.1.5)
- an unboxing conversion (§5.1.8) optionally followed by a widening primitive conversion (§5.1.2).

Value set conversion (§5.1.13) is applied after the type conversion.

The compile-time legality of a casting conversion is as follows:

- An expression of a primitive type may undergo casting conversion to another primitive type, by an identity conversion (if the types are the same), or by a widening primitive conversion, or by a narrowing primitive conversion, or by a widening and narrowing primitive conversion.
- An expression of a primitive type may undergo casting conversion to a reference type without error, by boxing conversion.
- An expression of a reference type may undergo casting conversion to a primitive type without error, by unboxing conversion.

- An expression of a reference type may undergo casting conversion to another reference type if no compile-time error occurs given the rules in §5.5.1.

The following tables enumerate which conversions are used in certain casting conversions. Each conversion is signified by a symbol:

- - signifies no casting conversion allowed
- ≈ signifies identity conversion (§5.1.1)
- ω signifies widening primitive conversion (§5.1.2)
- η signifies narrowing primitive conversion (§5.1.3)
- ωη signifies widening and narrowing primitive conversion (§5.1.4)
- ⇑ signifies widening reference conversion (§5.1.5)
- ⇓ signifies narrowing reference conversion (§5.1.6)
- ⊕ signifies boxing conversion (§5.1.7)
- ⊗ signifies unboxing conversion (§5.1.8)

In the tables, a comma between symbols indicates that a casting conversion uses one conversion followed by another. The type `Object` means any reference type other than the eight wrapper classes `Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, `Double`.

Table 5.5-A. Casting conversions to primitive types

To →	byte	short	char	int	long	float	double	boolean
From ↓								
byte	≈	ω	ωη	ω	ω	ω	ω	-
short	η	≈	η	ω	ω	ω	ω	-
char	η	η	≈	ω	ω	ω	ω	-
int	η	η	η	≈	ω	ω	ω	-
long	η	η	η	η	≈	ω	ω	-
float	η	η	η	η	η	≈	ω	-
double	η	η	η	η	η	η	≈	-
boolean	-	-	-	-	-	-	-	≈
Byte	⊗	⊗,ω	-	⊗,ω	⊗,ω	⊗,ω	⊗,ω	-
Short	-	⊗	-	⊗,ω	⊗,ω	⊗,ω	⊗,ω	-
Character	-	-	⊗	⊗,ω	⊗,ω	⊗,ω	⊗,ω	-
Integer	-	-	-	⊗	⊗,ω	⊗,ω	⊗,ω	-
Long	-	-	-	-	⊗	⊗,ω	⊗,ω	-
Float	-	-	-	-	-	⊗	⊗,ω	-
Double	-	-	-	-	-	-	⊗	-
Boolean	-	-	-	-	-	-	-	⊗
Object	⇓,⊗	⇓,⊗	⇓,⊗	⇓,⊗	⇓,⊗	⇓,⊗	⇓,⊗	⇓,⊗

Table 5.5-B. Casting conversions to reference types

To →	Byte	Short	Character	Integer	Long	Float	Double	Boolean	Object
From ↓									
byte	⊕	-	-	-	-	-	-	-	⊕,⇑
short	-	⊕	-	-	-	-	-	-	⊕,⇑
char	-	-	⊕	-	-	-	-	-	⊕,⇑
int	-	-	-	⊕	-	-	-	-	⊕,⇑
long	-	-	-	-	⊕	-	-	-	⊕,⇑
float	-	-	-	-	-	⊕	-	-	⊕,⇑
double	-	-	-	-	-	-	⊕	-	⊕,⇑
boolean	-	-	-	-	-	-	-	⊕	⊕,⇑
Byte	≈	-	-	-	-	-	-	-	⇑
Short	-	≈	-	-	-	-	-	-	⇑
Character	-	-	≈	-	-	-	-	-	⇑
Integer	-	-	-	≈	-	-	-	-	⇑
Long	-	-	-	-	≈	-	-	-	⇑
Float	-	-	-	-	-	≈	-	-	⇑
Double	-	-	-	-	-	-	≈	-	⇑
Boolean	-	-	-	-	-	-	-	≈	⇑
Object	⇓	⇓	⇓	⇓	⇓	⇓	⇓	⇓	≈

5.5.1. Reference Type Casting

Given a compile-time reference type S (source) and a compile-time reference type T (target), a casting conversion exists from S to T if no compile-time errors occur due to the following rules.

If S is a class type:

- **If T is a class type, then either |S| <: |T|, or |T| <: |S|. Otherwise, a compile-time error occurs.**

  Furthermore, if there exists a supertype X of T, and a supertype Y of S, such that both X and Y are provably distinct parameterized types (§4.5), and that the erasures of X and Y are the same, a compile-time error occurs.

- If T is an interface type:
  - **If S is not a `final` class (§8.1.1), then, if there exists a supertype X of T, and a supertype Y of S, such that both X and Y are provably distinct parameterized types, and that the erasures of X and Y are the same, a compile-time error occurs.**

    Otherwise, the cast is always legal 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 a type variable, then this algorithm is applied recursively, using the upper bound of T in place of T.
- **If T is an array type, then S must be the class `Object`, or a compile-time error occurs.**
- If T is an intersection type, T_1 & ... & T_n, then it is a compile-time error if there exists a T_i ($1 \le i \le n$) such that S cannot be cast to T_i by this algorithm. That is, the success of the cast is determined by the most restrictive component of the intersection type.

If S is an interface type:

- **If T is an array type, then S must be the type `java.io.Serializable` or `Cloneable` (the only interfaces implemented by arrays), or a compile-time error occurs.**
- **If T is a class or interface type that is not `final` (§8.1.1), then if there exists a supertype X of T, and a supertype Y of S, such that both X and Y are provably distinct parameterized types, and that the erasures of X and Y are the same, a compile-time error occurs.**

  Otherwise, the cast is always legal at compile time (because even if T does not implement S, a subclass of T might).
- If T is a class type that is `final`, then:
  - **If S is not a parameterized type or a raw type, then T must implement S, or a compile-time error occurs.**
  - **Otherwise, S is either a parameterized type that is an invocation of some generic type declaration G, or a raw type corresponding to a generic type declaration G. Then there must exist a supertype X of T, such that X is an invocation of G, or a compile-time error occurs.**

    Furthermore, if S and X are provably distinct parameterized types then a compile-time error occurs.
- If T is a type variable, then this algorithm is applied recursively, using the upper bound of T in place of T.
- If T is an intersection type, T_1 & ... & T_n, then it is a compile-time error if there exists a T_i ($1 \le i \le n$) such that S cannot be cast to T_i by this algorithm.

If S is a type variable, then this algorithm is applied recursively, using the upper bound of S in place of S.

If S is an intersection type A_1 & ... & A_n, then it is a compile-time error if there exists an A_i ($1 \le i \le n$) such that A_i cannot be cast to T by this algorithm. That is, the success of the cast is determined by the most restrictive component of the intersection type.

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 a type variable, then this algorithm is applied recursively, using the upper bound of T in place of T.
- **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 undergo casting conversion to TC.
- If T is an intersection type, T_1 & ... & T_n, then it is a compile-time error if there exists a T_i ($1 \le i \le n$) such that S cannot be cast to T_i by this algorithm.

Example 5.5.1-1. Casting Conversion for Reference Types

```
class Point { int x, y; }
interface Colorable { void setColor(int color); }
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`.

Example 5.5.1-2. Casting Conversion for Array Types

```
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
    public String toString() { return "("+x+","+y+")"; }
}
interface Colorable { void setColor(int color); }
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 program compiles without errors and produces the output:

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

5.5.2. Checked Casts and Unchecked Casts

A cast from a type S to a type T is *statically known to be correct* if and only if S <: T (§4.10).

A cast from a type S to a parameterized type (§4.5) T is *unchecked* unless at least one of the following is true:

- S <: T
- All of the type arguments (§4.5.1) of T are unbounded wildcards
- T <: S and S has no subtype X other than T where the type arguments of X are not contained in the type arguments of T.

A cast from a type S to a type variable T is unchecked unless S <: T.

A cast from a type S to an intersection type T_1 & ... & T_n is unchecked if there exists a T_i ($1 \le i \le n$) such that a cast from S to T_i is unchecked.

An unchecked cast from S to a non-intersection type T is *completely unchecked* if the cast from |S| to |T| is statically known to be correct. Otherwise, it is *partially unchecked*.

An unchecked cast from S to an intersection type T_1 & ... & T_n is *completely unchecked* if, for all i ($1 \le i \le n$), a cast from S to T_i is either statically known to be correct or completely unchecked. Otherwise, it is *partially unchecked*.

An unchecked cast causes a compile-time unchecked warning, unless suppressed by the `SuppressWarnings` annotation (§9.6.4.5).

A cast is *checked* if it is not statically known to be correct and it is not unchecked.

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

- The cast is statically known to be correct.

  No run-time action is performed for such a cast.
- The cast is a completely unchecked cast.

  No run-time action is performed for such a cast.
- The cast is a partially unchecked or checked cast to an intersection type.

  Where the intersection type is T_1 & ... & T_n, then for all i ($1 \le i \le n$), any run-time check required for a cast from S to T_i is also required for the cast to the intersection type.
- The cast is a partially unchecked cast to a non-intersection type.

  Such a cast requires a run-time validity check. The check is performed as if the cast had been a checked cast between |S| and |T|, as described below.
- The cast is a checked cast to a non-intersection type.

  Such a 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 erasure (§4.6) of 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, via the algorithm in §5.5.3.

  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.

5.5.3. Checked Casts at Run Time

Here is the algorithm to check whether the run-time type R of an object is assignment compatible with the type T which is the erasure (§4.6) of the type named in the cast operator. If a run-time exception is thrown, it is a `ClassCastException`.

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 (§8.1.5) 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.

Example 5.5.3-1. Incompatible Types at Run Time

```
class Point { int x, y; }
interface Colorable { void setColor(int color); }
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);
    }
}
```

This program uses casts to compile, but it throws exceptions at run time, because the types are incompatible.

5.6. Numeric Contexts

Numeric contexts apply to the operands of an arithmetic operator.

Numeric contexts allow the use of:

- an identity conversion (§5.1.1)
- a widening primitive conversion (§5.1.2)
- an unboxing conversion (§5.1.8) optionally followed by a widening primitive conversion

A *numeric promotion* is a process by which, given an arithmetic operator and its argument expressions, the arguments are converted to an inferred target type T. T is chosen during promotion such that each argument expression can be converted to T and the arithmetic operation is defined for values of type T.

The two kinds of numeric promotion are unary numeric promotion (§5.6.1) and binary numeric promotion (§5.6.2).

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`, `Character`, or `Integer`, it is subjected to unboxing conversion (§5.1.8). The result is then promoted to a value of type `int` by a widening primitive conversion (§5.1.2) or an identity conversion (§5.1.1).
- Otherwise, if the operand is of compile-time type `Long`, `Float`, or `Double`, it is subjected to unboxing conversion (§5.1.8).
- Otherwise, if the operand is of compile-time type `byte`, `short`, or `char`, it is promoted to a value of type `int` by a widening primitive conversion (§5.1.2).
- Otherwise, a unary numeric operand remains as is and is not converted.

After the conversion(s), if any, value set conversion (§5.1.13) is then applied.

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

- Each dimension expression in an array creation expression (§15.10.1)
- The index expression in an array access expression (§15.10.3)
- 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).

  A `long` shift distance (right operand) does not promote the value being shifted (left operand) to `long`.

Example 5.6.1-1. 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 program produces the output:

```
a: -1,1
~0xffffffff==0x0
0xffffffff<<4L==0xfffffff0
```

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 that is convertible to a numeric type, the following rules apply, in order:

1. If any operand is of a reference type, it is subjected to unboxing conversion (§5.1.8).
2. Widening primitive conversion (§5.1.2) is applied to convert either or both operands as specified by the following rules:
   - 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 conversion(s), if any, value set conversion (§5.1.13) is then 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)

Example 5.6.2-1. Binary Numeric Promotion

```
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);
    }
}
```

This program 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.