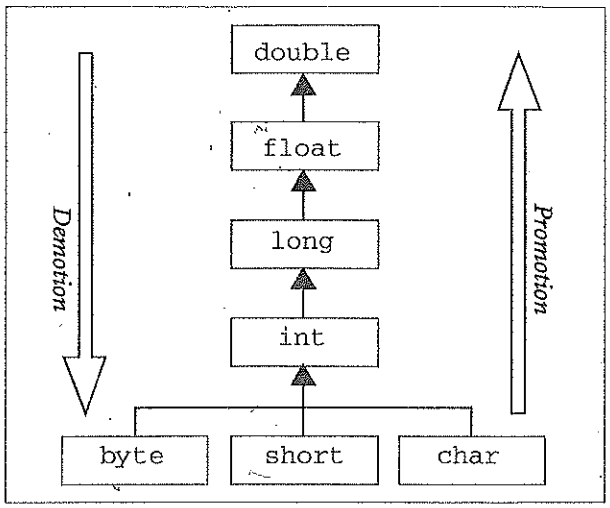

Figure 1.18 The hierarchy of numeric types

The compiler detects the need for conversion in three situations:

- when the two operands of an operator are not the same
- when both operands are of the type `byte`, or
- when the type of the RHS is different from that of the LHS.

If the required conversion involves promotion, it is done automatically; but if it involves a demotion, then a compile-time error is triggered.

If you are willing to take responsibility for the possible consequences, you can manually cast a value from one numeric type to another with the **cast operator.** The cast operator is a unary prefix operator (precedence level: –3; association: right to left) whose symbol is the name of the type sandwiched between two parentheses. For example, you can take responsibility for the conversion in the last assignment of Example 1.5 by rewriting it as

```
int result = (int) (100 * dVar);
```

Example 1.5

Provide a critique of the following fragment:

```
int iVar = 15;
long lVar = 2;
float fVar = 7.6f - iVar / lVar;
double dVar = 1L / lVar + fVar / lVar;
int result = 100 * dVar;
System.out.println(result);
```

Answer:

The first assignment involves a single type and is straightforward. The second has an `int` RHS and a `long` LHS. The compiler will auto-promote the RHS to `long`; that is, the compiler will proceed as if the RHS contains 2L. The third assignment involves a mixed expression, and the computation will begin with the / operator. Since its two operands are not of the same type, one of them will be auto-promoted to the other. This will resolve the overloaded operator to the one offered by the `long` type. The quotient of 15L and 2L is 7L. Still in the same statement, we see a second mixed expression involving a `float` and a `long`, so we convert the `long` to a `float` and subtract 7F from 7.6F. This yields 0.6F, and its type is the same as that of the LHS. The next assignment involves two divisions. The left one has same-type operands (`long`) and yields 0L. The right one requires a promotion from `long` to `float`, and it yields 0.3F. To complete the assignment, this value is auto-promoted to

double so that it matches the LHS. The very last assignment involves an int and a double, and its RHS evaluates to the double value 30.0. Completing the assignment requires demoting this value to an int, and this leads to a compile-time error. □

In this case, the fragment will compile. Pay special attention to the way casting was done. Had we written

```
int result = (int) 100 * dVar;
```

the error would have persisted. This is because the cast operator has a higher precedence than multiplication so it will cast the 100, not the product, and this is not what we wanted.

The output generated by the fragment may surprise you: it is 29, *not* 30. If you consider this difference significant, then you should not have casted.

Casting up the hierarchy of Fig. 1.18 is safe but unneeded since the compiler does it automatically. Casting down, however, should generally be avoided, or used with caution, except for one common (and safe) situation: to select the real / operator instead of the integer one. For example, to compute the average number of hits per second, we may write something like

```
hits / seconds
```

where both operands are long. This will not compute the sought average (due to truncation) so we cast

```
(double) hits / seconds
```

This will cast hits to double, which forces a promotion of seconds to double, thereby selecting the / of the type double. Note that the following two casting variations also result in a double, but only one of them (the second) will lead to the sought result:

```
(double) (hits / seconds)
hits / (double) seconds
```

Assignment Shortcuts

We conclude our coverage of the assignment statement by mentioning a shortcut: any assignment of the form

```
variable = variable operator expression;
```

can be abbreviated as

```
variable operator= expression;
```

Here are some examples:

```
hits += 6; is short for hits = hits + 6;
hits *= count; is short for hits = hits * count;
```

Assignment is treated in Java as an operator with the symbol =, precedence level -15, and right-to-left association. Appendix B contains a precedence table of all operators in Java, and at level –15, assignment has the lowest precedence. This is precisely how assignment is meant to work: evaluate all operators in the expression on the RHS before the assignment operation is carried out. The **assignment shortcuts** are also operators, and they have the same level and association rule as the assignment operator.

Example 1.6

Provide a critique of the following fragment:

```
char letter = 'D';
letter = (char) (letter + 1);
System.out.println(letter);
int code = letter;
System.out.println(code);
int offset = letter - 'A';
System.out.println(offset);
```

Answer:

In the first assignment, the types are compatible (both are `char`) so the code of `'D'` will be stored in `letter`. (According to Appendix A, this code is `68`, but we do not need to know this in order to reason about this fragment.) The second assignment contains the expression

```
letter + 1
```

The two operands have different types (`char` and `int`), so we auto-promote the `char` and use the + of the `int` type. The result is an `int`, and in order to store it in `letter`, we must demote it by a manual cast. This leads to the code of the character that follows `'D'`, which is `'E'`. The first output will thus be `E`. What happened behind the scene is that `'D'` was converted to `68`, `68` was incremented to `69`, and then `69` was converted to `'E'`.

The next assignment,

```
int code = letter;
```

assigns a `char` to an `int`. Since the RHS is *lower*, it gets auto-promoted to an `int`. The end result is to store the code of `'E'` in code. The second output will thus be the code of code of `'E'` (which is `69`).

The next statement includes the expression

```
letter - 'A'
```

Both operands are of the type `char`, but this type has no operators. Hence, both must be promoted up the hierarchy (Fig. 1.18) to `int`. This leads to subtracting the code of `A` from the code of `E`, and since alphabetically consecutive letters have consecutive codes, this difference is the `int` `4`. The third output will thus be `4`.

□

Example 1.6 demonstrates how type casting is used to manipulate values whose type does not provide operator support. The values are first cast to `int`, where they can be manipulated, and then cast back.