Mixed-type Expressions

Every expression produces a value, and each value produced has a particular type. Thus, when you add or subtract two integers, the **result** is an integer. But what if....

```
1 | a = 5 * 3.5;
```

The CPU uses **different circuitry** for integer and floating-point calculations. To evaluate this expression, **both operands** must be type `int`, **or**, they both must be type `double`. If we convert both to `int`, we **lose information**; converting them to `double` does not.

When your compiler encounters an expression that uses different types, it determines the operand with the greatest **information potential**. It then creates **temporary** values of that type, initializing them with the other values. This is called **promotion**.

Assignment and Mixed Expressions

What is stored in `a` in the example shown above? That depends on the type of `a`. If the variable is other than `double`, the value is again, **implicitly converted** into the same type as the variable. Thus, while the value calculated is `17.5`, if `a` has type `int` then only the `17` will be stored.

- **Widening** conversions occur when the assignment causes a promotion, such as from `int` to `double`. These will always succeed (just as they do in Java or C#).
- **Narrowing** conversions occur when the assignment has the potential for losing information, such as assigning from `double` to `int`.

Narrowing implicit assignment conversions are **prohibited** in Java and C#, but they **are the default behavior** in C++. To turn off such implicit narrowing conversions, C++11 added **brace** or **list assignment**; this makes C++ work more like Java and C#.

```
1 | int a, b;
2 | a = 5 * 3.5;          // 17; implicit narrowing conversion
3 | b = {5 * 3.5};        // c++11+; compiler error
```