CS2110: Primitive number types

Values of each of the primitive number types

Java has these 4 primitive integral types:

byte: A value occupies 1 byte (8 bits). The range of values is -128..127

short: A value occupies 2 bytes (16 bits). The range of values is -2^15..2^15-1

int: A value occupies 4 bytes (32 bits). The range of values is -2^31..2^31-1

long: A value occupies 8 bytes (64 bits). The range of values is -2^63..2^63-1

and two "floating-point" types, whose values are approximations to the real numbers:

float: A value occupies 4 bytes (32 bits).

double: A value occupies 8 bytes (64 bits).

Values of the integral types are maintained in two's complement notation (see the dictionary entry for *two's complement notation*). A discussion of floating-point values is outside the scope of this website, except to say that some bits are used for the mantissa and some for the exponent and that infinity and NaN (not a number) are both floating-point values. We don't discuss this further.

Generally, one uses mainly types **int** and **double**. But if you are declaring a large array and you know that the values fit in a byte, you can save ¾ of the space using a **byte** array instead of an **int** array, e.g.

byte[] b= **new byte**[1000];

Operations on the primitive integer types

Types **byte** and **short** have no operations. Instead, operations on their values are treated as if the values were of type **int**. For example, suppose b is of type **byte**. Then the expression b+1 has type **int**.

The operations of types **int** and **long** are given in the table to the right. They are what one expects, except for one point. The designers of Java included the principle that **int** operations must produce an **int** and **long** operations must produce a **long**. The minimum and maximum values in type **int** are -2147483648 and 2147483647. Then what is the value of this expression?

2147483647 + 1

The answer is the smallest value, -2147483648, because the numbers "wrap around" —e.g. after the largest **int** value comes the smallest.

int and long operations

- b	negation
+ b	no change
a + b	addition
a - b	subtraction
a * b	multiplication
a / b	integer division 9 / 2 is 4
a % b	remainder 9 % 2 = 1

int/long literals

Below are five ways to write constants of type **int**. Follow one directly with l or L and it is a **long**, e.g. 123L.

1. A decimal integer that doesn't begin with 0: 20
2. An octal integer (begin with 0): 020 is the same as 16
3. A hexadecimal integer (begin with 0x, use 1..9 and A..F or a..f): 0xF is the same as 15. 0x1f is the same as 31
4. A binary integer (begin with 0b): 0b110 is the same as 6
5. Put an underscore '_' anywhere *between* digits to make it more readable: 5_300_000 is the same as 5300000

float/double literals

Here are ways to write a **double** literal. Follow it by F or f and it is a **float** literal.

1. 125.3 (anything with '.' in it is a floating point number)
2. 1.253E2 or 1.253e2 Same as 125.3 but in scientific notation
3. 1253E-1 Same as 125.3 but in scientific notation

no byte/long literals

There are no **byte** or **short** literals. But an assignment like:

byte b= 127;

will be treated like

byte b= (**byte**) 127;

The cast (**byte**) 127 (see the next page for casts) will be evaluated at compile-time. The following won't compile because 128 is outside the range of **byte**:

byte b= 128; // won't compile

If you have a function like this:

public int f(**byte** b) {…}

the call f(127) will not compile; it must be written like this:

f((**byte**) 127)

CS2110: Primitive number types

Narrower-wider types and assignment

We order the primitive number types as follows:

byte → **short** → **int** → **long** → **float** → **double**

narrower → wider

Each type is called *wider* than the ones to its left because it contains all the values of the types to its left and more. For example, every value of type **byte** is also a value of type **short**. The opposite of wider is *narrower*.

Since a **long** value occupies 64 bits and a **float** value only 32, you might question that **float** is wider that **long**. But remember that floating point numbers are only *approximations* to real numbers and are rarely exact. So, converting a **long** to a **float** value produces the closest **float** approximation to the **long**.

In Java, an assignment to a primitive type variable is syntactically correct and will compile only if the expression has the same or a narrower type than the variable. This prohibits information being lost or changed. For example:

```
int k= 55.3;    // will not compile because float (the type of 55.3), is wider than int
long v= 55;     // will compile because long is wider than int.
```

Casting between primitive number types

The expression **(float)** 5 is called a *cast* of **int** value 5 to type **float**. For each primitive number type T, the cast operator (T) is a unary prefix operator. Cast (T) v converts v to its representation in type T.

Widening casts are done automatically when required. Here's information about widening casts.

1. A widening cast to an integral type (e.g. **int** to **long**) preserves the value.
2. A widening cast to a floating point type changes the value to the closest possible approximation to the value in the new type. Further discussion is beyond the scope of these webpages.

Narrowing casts must be explicitly requested. For example, here is how to add 1 to **byte** variable b:

```
b= (byte) (b+1);
```

Here's information about narrowing casts.

1. A narrowing cast from one integral type to another preserves the value if the value can be represented in the resulting type. Otherwise, the resulting value depends on the wrap-around properties of integral types. For example, since 127 is the largest value of type **byte**, the value of (**byte**)128 is -128, the smallest value of type **byte**.
2. A narrowing cast from **double** to **float**, like (**float**) 5.2 , is done using rules that are beyond the scope of this webpage. Some precision will be lost.
3. A narrowing cast from a floating-point type to an integral type, like (**short**) 5.2 , first requires converting the floating-point value to the closest possible integer, using rules that are beyond the scope of this webpage. Then, that value may require a cast to a narrower integral type, using the method shown in point 1. Note this example: The value of expression (**int**) 1E200 is the largest **int** value, 2147483647.

Sequences of casts are performed from right to left, since a cast is a unary prefix operator. We illustrate this:

```
(byte) – (int) -128L      // (1) cast -128L to int, giving -128;
                          // (2) negate -128 giving 128;
                          // (3) cast 128 to byte, giving, because of wrap-around, -128
```

Type char

Primitive type **char** is also an integral number type. Each **char** value is represented by an integer, and one can cast between them, e.g.

```
(int) 'a'       evaluates to 97
(char) 97       evaluates to 'a'
```

Look under the dictionary entry for **char** for a complete discussion of type **char**.