UNIT 9. DATA TYPES AND MEMORY

Summary

Here you will learn the basics about bits and bytes, and how computers store data. You will also see how to convert between similar numeric types.

In this section, you will…

- Learn that some data types take up more memory than others.
- See nearly the full list of primitive data types.
- Learn about casting, converting between data types.

Learning Objectives

After finishing this unit, you will be able to …

- Make integer or floating point variables that use more or less memory.
- Convert between data types using explicit and implicit casts.

How to Proceed

- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

9.1 Introduction

Now that you have some programming practice let's take a step back and learn a little about bits, bytes, computer memory, and what this means for your data types. We will learn about how data is stored (only briefly!), what different data types are available, and how you go between the data types.

9.2 Bits and bytes and nibbles

Advanced: A computer stores everything as switches that can be either on or off. These switches are called bits, and usually are written as a 0 for off, or a 1 for on. One bit has 2 possible settings. A combination of two bits has 4 possible settings `(00,01,10,11)`. In fact, *n* bits has 2^n possible settings. If you stick with computers, you'll get very comfortable with powers of 2, for this reason.

How do you count with bits? Logically, it follows the same as when you count in regular base-10 numbers, but it takes a bit to wrap your head around it. The first two numbers are easy

```
0
1
```

But now, since we ran out of digits (binary only has `0,1`), we need to add a new column on the left, and add a 1. This is like when we go `0,1,2…9`; if we want to count higher, we add a 1 on a new column to the left, and reset the rest of the number to zero, to get a 10.

```
10 // 2
```

now we just add a 1 again to get 3

```
11 // 3
```

Like above, we ran out of digits so we reset to zeros and add one on the left. Just as 99 becomes 100, then 11 becomes 100 in binary

```
100 // 4
```

Try to keep going with this. How would you write 31?

A group of 8 bits (switches) is called a byte of memory

```
00110101 ← one byte of data
```

A group of 4 bits (half a byte) is called a nibble (seriously!!)

```
0110 ← a nibble of data
```

From here we scale up.

```
2^10 bytes, or 1024 bytes, (8192 bits) is called a kilobyte
2^20 bytes, or 2^10 (1024) kilobytes, make a megabyte
2^30 bytes, or 2^10 (1024) megabytes, make a gigabyte
2^40 bytes, or 2^10 (1024) gigabytes, make a terabyte
```

A terabyte has 1,099,511,627,776 bytes. It has 8,796,093,022,208 switches (bits).

If we line up those many switches using standard 7 cm light switches, we get a line of switches 615 million kilometers long! That is four times the distance from the earth to the sun. Holy cow, that's a lot of switches.

As an aside, some new standard units are moving to even powers of ten, where 1 terabyte = 1,000,000,000,000 bytes. This is not easily represented inside a computer because it's not a power of two, but people like it better.

9.3 Using More or Less Memory

Variables use computer memory, and each variable type has a clearly defined amount of memory that it can use. A variable that uses more memory can store a wider range of numbers, and/or more precision, and a variable with less memory can store less. Although our examples are very small and you may not think you need to worry about computer memory, professional programmers are generally thinking about how they are using memory and keep an eye to being economical.

Let's look at the `int` data type. As we learned previously, the largest number that an `int` can store in Processing is 2147483647 and the smallest number is -2147483648. This seemingly-random number is actually determined by how the number is stored; ask your professor if you are interested:

```
int number = 2147483647;
```

Now, what happens if we increase the number by 1? To try and force it to store that bigger number?

```
number++
println(number);
```

If you try this, you will see that you get a very large negative number. In fact, what we are seeing here is hitting the memory limits of the integer data type. If we go over the limit, the values roll over. The same happens in the other direction – if we keep

going minus, we will see eventually a positive number.

This can be very confusing! If you end up seeing unexpected negative numbers when you are doing calculations, it may be because you had an overflow.

If you come across overflow, you generally need to try and be cleverer with your numbers to avoid such large numbers. Alternatively, you can use a data type with more memory.

9.4 The Primitive Data Types

There are only a few basic, core data types, in most programming languages. We have already seen three basic types: `float`, `char`, and `int`. Remember, `String` is an object and acts differently. This actually covers most of the general territory of primitive types. We will soon learn boolean, a type for true and false only. However, in addition to this, we mainly have variants of the integer and floating point types with more or less memory. For example, in the integer class we have the following variables

type	size	minimum	maximum
`byte`	1 byte	-128	127
`short`	2 byte	-32,768	32,767
`int`	4 byte	-2^{31}	$2^{31}-1$
`long`	8 byte	-2^{63}	$2^{63}-1$

You can use these types just like the other ones we have learned. Also, these are integers, so they follow the same rules regarding integer division, etc.

For the most part, you will only use the larger type (long) if you want a larger range of numbers to store. Except for special circumstances, there is little benefit to using the smaller types.

Advanced: You may think you can save memory by using, for example, the short type. However, you won't likely save memory. Computers work fastest on number sizes that match their processor. For example, in recent years you hear about 32-bit and 64-bit processors. If you have a 2 byte short variable, and a 32 bit computer, your computer is likely to store those 2 bytes alongside 2 empty bytes to make a 32 bit number, to save time and processing to work with it. So, even though a short has 2 bytes, your computer still will probably use 4 to store it. In exceptional circumstances (not in Processing or Java), you can tell your computer how to pack these numbers, for example, when you really need that memory saved, or you need to ensure how many bytes are used.

For floating point numbers, we only have two types. We have the `float` (4 bytes)

and the `double` (8 bytes) types. Unlike integers, more memory in a floating point does not manifest itself only a larger range. Instead, it is also *more precision*. For example, if you perform 2/3 you get the following results:

`float` - 0.6666667

`double` - 0.6666666666666666

As you can see, the double variable gives a closer approximation. In general, we will stick to the `int` and `float` types, but you should be aware of these variants.

Advanced: `float` or `double`? Which is better? Clearly, double has better precision. If you have a 64 bit machine, your 32 bit (4 byte) `float` will likely take 64 bits of memory anyway, so why bother with `float`? If you talk to people who do scientific computing, they will argue strongly to always use `double`. In fact, in regular Java, `double` is the default floating point type. However, if you talk to a graphics person, they may tell you to always use float. Also, processing defaults to `float`. Why is this?

The complication here is graphics cards – your computer has a specialized processor to do graphics. Even if you spend a boatload of cash on a high-end gaming video card, it is very likely optimized for `float` and not `double`. For graphics, speed is more valuable than a bit of extra precision.

9.5 Going Between Data Types: Casting

If you take a metal statue – like a miniature lion – and you want to turn it into a statue of a dog, one way to do it is to first melt the lion down (it's metal, after all), and pour it into a mold, a cast, of a dog. Once it cools, now you have a metal dog. This is called casting, when you take a metal and force it into the shape of the cast.

There is a similar idea in programming. Casting is taking one kind of data and forcing it into the shape of another type. Although we haven't dealt much with any integer type except for `int`, let's try a few things.

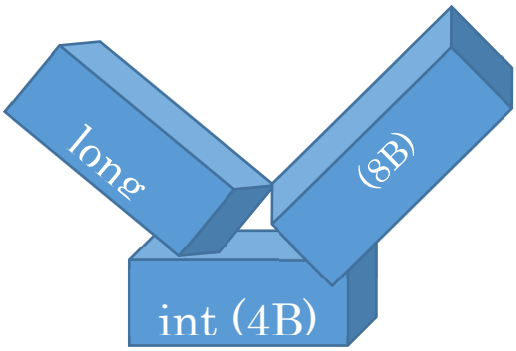
```
int i = 1234;
byte b = i; // store 1234 in b
```

Processing complains and won't compile: `cannot convert int to byte`. We may have guessed this, since we know that byte can only go as large as 128 and cannot store 1234 (check the table on the previous page). Let's try another one:

```
long l = 1234;
int i = l;
```

Even though we know that the integer can handle the number `1234`, Processing still

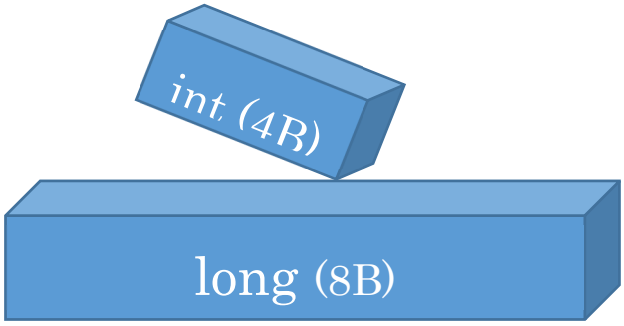
complains: `cannot convert long to int`. In fact Processing does not even look inside to see if the number fits. It just knows that the datatypes are of different sizes, so it complains. The issue here is that you can lose data mistakenly. What if the long actually had a number that was too large for the integer? What should happen then? Should it roll over like in our earlier examples? It is quite confusing, so to play it safe, Processing just stops and doesn't let you do it. This is called a ***narrowing conversion***, since you are moving from a more-capable type to a less-capable type. You can lose data if you are not careful: the diagram illustrates this, showing that you need to break a long up (8 bytes) to make it fit into the 4 byte box.



What about the other direction? What if we want to store an `int` type into a `long`? Or a `byte` into an `int`? Try the following

```
byte b = 12;
int i = b;
long l = i;
```

Here, we take a `byte`, then try and store its value into a 4-byte `int`, and then try to store that into an 8-byte `long`. If you try these in Processing, they all work just fine. This is because we are asking Processing to move from a less capable type to a more capable one. This is called a ***widening conversion*** – as in the inset below, Processing has no problem with this since there is no risk of data loss. The 8 byte `long` can easily accommodate the 4 byte `int`. When data is converted in this way – from a less-capable to a more-capable type – it is called an ***implicit cast***. The conversion happens implicitly, without you asking for it.

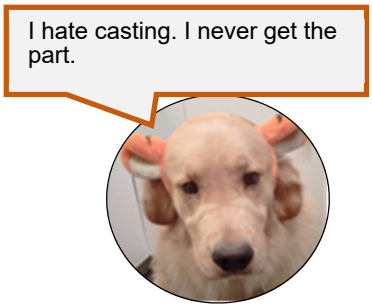


Sometimes, you want to force a narrowing conversion even though you know that data may be lost. For example, you may be working in a `long` datatype to work with large numbers but you know that your result is small enough to fit into an `int` (perhaps you divided it by a large number). In this case, you need to use an ***explicit cast*** to make this happen – you need to tell Processing that you know what you are doing and you want it to cast the data to the new type, even though data may be lost.

Let's do an example:

```
long large = 200;
int small = large;
```

In this case, Processing will not run this code because of the narrowing conversion. However, we know that it is safe, so we want to force a conversion using an explicit cast. The syntax of an explicit cast is as follows:




```
(newType)data
```

For example, we can update the above example:

This tells processing that we know what we are doing, and to go ahead and do the conversion.

```
long large = 200;
int small = (int)large; // explicit cast
```

The same logic above holds for going between floating point types. Since `double` uses more memory than `float`, it is larger. Going from `float` to a `double` is a widening conversion, and can be done with an implicit cast. Going from `double` to `float` is a narrowing conversion and requires an explicit cast:

```
float f = 1.23;
double d = f; // implicit cast OK!
d = 5.123
f = d; // ERROR! Cannot convert, narrowing conversion
f = (float)d; // OK - explicit cast.
```

What about going between the integer and floating point types? In this case, we need to think about it a little differently. Instead of thinking about more or less memory, think about more or less capable. A floating point number can store more detail than an integer, which is limited to whole numbers only. Therefore:

- ***Going from integer -> floating point is a widening conversion, and does not require an explicit cast.***
- ***Going from floating point -> integer is a narrowing conversion, and requires an explicit cast.***

When you go from a floating point to an integer, you need an explicit cast. Java just cuts off the decimal portion and stores the whole number. For example:

```
float f = 1.234;
int i = (int)f;
println(i)
```

What do you think the output will be? Try it out.

One more thing to mention is the order of operations with casts. What about the following?

```
int i = (int)0.5*3.0;
```

What do you expect will happen? If you try to run this, Processing complains that it cannot do the conversion. This is because ***casts happen first in order of operations***. In this example, the 0.5 gets converted to an `int` first, which becomes a zero.

```
int i = 0*3.0;
```

then there is a floating point multiplication, resulting in 0.0, which cannot be stored in an `int`, since it is floating point.

In this class, you will primarily use casting to go between integer and floating point numbers, but you will need to understand the basics of how it works in general, as it is testable. You will use casting a great deal more in future courses in two ways: one, you will do more bits-and-bytes work, and two, casting ends up showing up in object oriented programming, as well.

There are two final notes about casting. First, you can use casting to go between integers and characters, as long as you understand that a character is just a number underneath. Two, when you convert between String and numerical types, we do not call it casting, because a lot of work goes into it; data just isn't forced into a new type (with possible data loss). Someone actually wrote a program to analyze your data and convert it to a string or number.

Check your Understanding

9.6 Check Your Understanding: Exercises

Exercise 1. Make a program to help a shipping company figure out how much oil can fit inside a square tanker. The tanker has dimensions of width: 1234mm, height: 5678, and length: 9012. This company uses barrels that store 1 million cubed millimeters.

a. Calculate the volume of the tanker using integers, by multiplying the width by the height by the length.
b. Calculate how many barrels can be stored by dividing the total volume by a million.
c. `println` the result. What is wrong with it? How can you fix it? Your final answer should be 63143 barrels.

Exercise 2. Your friend wrote the following code to generate a random dice roll (on a standard 6-sided die), but complains that a) they get decimal numbers, and, the dice roll starts from 0 instead of 1.

```
prinln(random(6));
```

a. You know that the decimals happen because `random` gives you a floating point result. How can you fix this using casting?
b. Fix the off-by-one error

Exercise 3. The following code contains two casts: an explicit and an implicit one. Can you identify both?

```
float f = (int)PI;
```

Exercise 4. Up until now, you can avoid the problems of integer division by making one of the operands a float; remember, if both operands are integers, the computer does integer division. Without changing the data types, and only introducing casts, how can you fix the following code bug which unfortunately uses an integer division?

```
int units = 10;
int dollars = 5;
float cost = dollars / units;
println("the cost is: "+cost+" dollars each");
```

Exercise 5. Use casting and a little bit of math to print out a floating point number to two decimal places only. For example, `println(PI)` should give you 3.14.

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.