Lecture 8

- Covers
  - Internal representation of primitive data types
  - Type compatibilities and type casting
  - Integer division and truncation of floating point numbers
- Reading: Savitch 2.1

Lecture overview

- This lecture has 5 main parts
  - Representation of Integers
  - Representation of Real Numbers
  - Type char
  - Type boolean
  - Type Compatibility and Type Conversion

- Internal representation of integers

Integer types

- Integers are whole numbers

  0, -1, 1, -2, 2, etc.

- Java has 4 integer types

Type name	Memory used	Size range
byte	1 byte	-128 to 127
short	2 bytes	-32768 to 32767
int	4 bytes	-2147483648 to 2147483647
long	8 bytes	-9223372036854775808 to 9223372036854775807

- We most commonly use the int type

Representation of integers

- Consider a 2-byte short for illustration purposes

```
00000001 01010110
```

- Addition

```
                1  1  1 1              1
  00000001 01010110          342
+ 00000000 01010111        + 087
-----------------------    -----
  00000001 10101101          429
```

Representation of integers

- Subtraction
  - Can be done as usual but in computing we use the two's complement method
  - First obtain the one's complement of the number to be subtracted by subtracting each digit in that number from 1

```
  11111111 11111111
- 00000000 01010111
  11111111 10101000
```

Representation of integers

- Next obtain the two's complement of that number by adding 1 to the 1's complement

  `= 11111111 10101001`

- Then *add* this number to the number you wanted to subtract from

E.g.

```
  0001 01010110      ➲    0001 01010110
- 0000 01010111           + 1111 10101001
                          ------------------
                          ⁄0000 11111111
```

- Then discard the leftmost 1

Representation of integers

- Positive integer values are stored in their binary representation
- Negative integer values are stored in their two's complement binary representation
- Thus the left-most bit indicates the sign of the integer
  - 0 indicates a positive number
  - 1 indicates a negative number

Representation of integers

- 2's complement form of $-n$ is $2^{16} - n$
- More examples

```
0000 0000 0000 0010 =
0000 0000 0000 0001 =
0000 0000 0000 0000 =
1111 1111 1111 1111 =
1111 1111 1111 1110 =
1000 0000 0000 0000 =
0111 1111 1111 1111 =
```

- Internal representation of real numbers

Type double

- Real numbers are numbers with a fractional part
- Java has two types of real numbers

Type name	Memory used	Size range	Precision
float	4 byte	-3.4×10^{38} to 3.4×10^{38}	≈ 7 sig. digits
double	8 bytes	-1.7×10^{308} to 1.7×10^{308}	≈ 15 sig. digits

- Scientific (floating-point) form

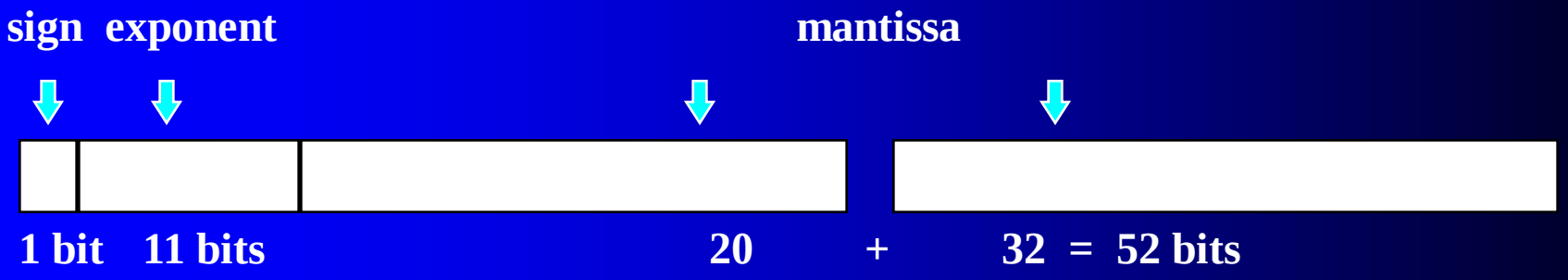
  $45678 = 4.5678 \times 10^{4}$

  $0.0345 = 3.45 \times 10^{-2}$

- We most commonly use the double type

Floating-point representation

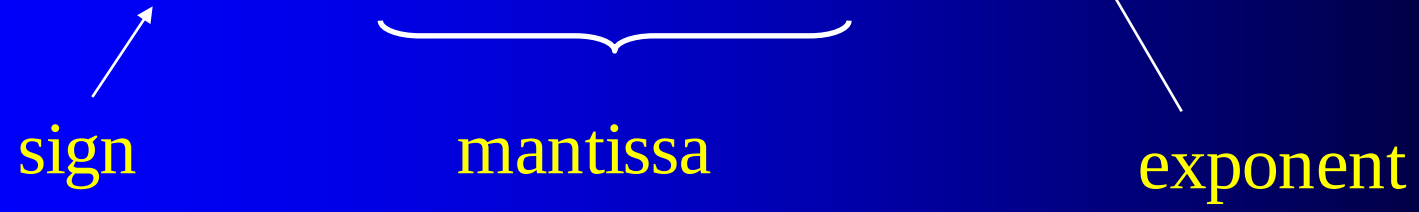
- 64 bits used (i.e. 8 bytes)



Scientific notation

- -12.345 can be written as -0.12345×10^2 (scientific notation, base 10)
- Similarly, a real number can be expressed in base 2 in the form

$$+/- \; 0.b_1 \, b_2 \, b_3 \ldots b_n \; \times \; 2^e$$



- Type char

Type char

- Type char is used to represent a single character (of almost any language)
- Examples

  'a' '+' '3'
- Stored in 2 bytes
- Java uses Unicode scheme to represent characters
- Stored as an unsigned 16-bit integer
- Range of 0 to 216 – 1 possible values (64 K)

Type char

- Each number maps to a character in a predefined manner
- Only a little over half the range is currently mapped to characters
- Characters can appear in programs in 3 forms
  - As a character between a pair of single quotes
  - As an escape sequence
  - As a Unicode* value

* *Unicode is an extension of the earlier ASCII character set that only allowed 256 different characters*

Unicode representation

numeric code in base 10	*numeric code in base 2*	*character*
.	.	.
.	.	.
63	… 0011 1111	‘?’
64	… 0100 0000	‘@’
65	… 0100 0001	‘A’
66	… 0100 0010	‘B’
67	… 0100 0011	‘C’
68	… 0100 0100	‘D’
.	.	.
.	.	.
.	.	.
97	… 0110 0001	‘a’
98	… 0110 0010	‘b’
.	.	.
.	.	.

Characters

- In Java, letters, digits, punctuation marks, and special characters are usually written between a pair of single quotes

'a'	letter a in lower case
'A'	letter A in upper case
'1'	digit 1
'!'	punctuation mark !
'@'	the special “at” character

Characters

- Non-printable characters (control characters) are usually written as escape sequences

'\b'	backspace
'\t'	tab
'\n'	new line (line feed)
'\r'	carriage return
'\" '	double quote
'\' '	single quote
'\\'	backslash

Characters

- Characters (any) can be written in Unicode with value in hexadecimal form
- Example

'\u004E'	letter 'N'
'\u0007'	the 'beep‘

Strings of characters

- String = a sequence of characters
- Example
  - "hello world"
- "a" is not the same as 'a'

- We will look at strings in Java in the next lecture

- Type boolean

Type boolean

- Sometimes we want to store whether or not some expression is true or false
- There is a type boolean that does this

```
boolean enrolled;
enrolled = true;
enrolled = false;
```

- Stored in 1 bit

- Type compatibility and type conversion

Type compatibilities

- In general, a variable of one type cannot store a value of another type
- Examples

```
int counter;
counter = 2.34;
counter = 'a';
```

Incorrect

Not incorrect *but* poor style

Mixing numeric types

- Java allows the mixing of *byte*, *short*, *int*, *long*, *float*, and *double* in arithmetic expressions
  - If one argument of a binary operator is a *double*, the other argument is converted into a *double*, and the result is a *double*
  - Otherwise, if one argument is a *float*, the other will be converted into a *float*, the result is a *float*
  - Otherwise, if one argument is a *long*, the other is converted into a *long*, the result is a *long*
  - Otherwise, if one argument is an *int*, the other is converted into an *int*, the result is an *int*
  - Otherwise if one argument is a *short*, the other is converted into a *short*, but the result is an *int*
  - In the case the two arguments are *bytes*, the result is still an *int*

Examples

- Given

```
double d = 1.2;
float f = 1.2F;
byte b = 123;
int i = 2000000000;
```

1. `d + f` *// valid, result is a double*
2. `f + b` *// valid, result is a float*
3. `b + b` *// valid, result is an int*
4. `i * 2;` *// result is an int and is equal to*
   // -294967296

Mixing with *char*

- Java allows the mixing of *char* with numeric data types
  - A *char* argument of a binary operator is always treated as an *int*. Thus the result is an *int*
  - Given

```
double x = 1.2;
byte b = 123;
char ch = 'A';
```

1. `x + ch` *// valid, result is a double*
2. `b + ch` *// valid, result is an int*
3. `ch + ch` *// valid, result is an int*

Mixed types in assignments

- As a special case, we can assign an int value to a double variable; but not vice versa
- In general, Java performs the following implicit type conversions for assigning a value to a variable of a different type

  byte → short → int → long → float → double

  char → int → long → float → double

Mixed types in assignments

- These are all considered widening conversions as they convert the data into another type that uses more memory to store the value; the magnitude range of the data will not be lost
- In the case of converting an integer type to a floating point type, some precision may be lost

Examples

- Given

```
double x = 1.2;
int i = 2;
byte b = 4;
char ch = 'A';
```

1. `x = i;` *// valid*
2. `i = x;` *// invalid*
3. `x = ch;` *// valid*
4. `b = ch;` *// invalid*
5. `ch = b;` *// invalid*

Explicit type casts

- When it is required by the programming logic, we can explicitly convert a data value of one type to another type
- When converting a data value stored in one type to a type that uses less memory, information can be lost or unexpected results may occur
- These conversions are referred to as narrowing conversions
- To make a narrowing conversion we have to explicitly tell the compiler with a type cast

Examples

- double x = 12.34;
  int i = (int) x; *// i = 12*
  // some precision is lost
- int i = 12345;
  byte b = (byte) i; *// b = 57*
  // unexpected

Integer division

- If a division involves two integers, the result will be an integer with the remainder discarded
- Examples

  8 / 4 $\Rightarrow$ 2

  9 / 4 $\Rightarrow$ 2

Double division

- If a division involves at least one double, the result will be a double
- Examples

  8 / 4.0 $\Rightarrow$ 2.0

  9.0 / 4 $\Rightarrow$ 2.25

Conversion between double and int

- Sometimes, we need to convert double values to int and vice versa
- The conversion can be done with a type cast
- Examples

```
9 / (double) 2   ⇒   4.5
(double) 9 / 2   ⇒   4.5

int numTables = (int) Math.ceil(
            (double) numGuests / tableSize)
```

% operator

- The % operator determines the remainder value of a division (involving 2 integers)
- Examples

  8 % 4 $\Rightarrow$ 0

  9 % 4 $\Rightarrow$ 1

Order of evaluation

- The order in which an expression is evaluated is governed by the rules of precedence and association
- Precedence: from highest to lowest

  * /

  + -
- Association: from left to right
- Parentheses can be used to change the order

Order of evaluation

- Examples

  2 * 2 + 8 * 4 / 2

  2 * 4 / 8

  2 * (4 / 8)

Next lecture

- The String class