



# Primitive Data Types

---

James Brucker



# Primitive Data Types

---

- A **primitive data type** has only a value, such as a number.
- Primitive types are things the CPU can directly manipulate. Example:  $2 + 3$  (cpu can add int)
- Java has 8 primitive types, such as:
  - `int`
  - `char`
  - `double`



# Data Type: Values and Operations

- A data type has a set of *operations* that it supports
- The operations are what make data useful!

## Essential Information About a Data Type

1. what *values* can a data type store?
2. what *operations* can we perform on a data type?

**Operations** for integer, float, and double

*arithmetic*:  $a + b$ ,  $a * b$ ,  $a / b$ ,  $a \% b$  (modulo)

*comparison*:  $a < b$ ,  $a > b$ ,  $a \geq b$ ,  $a == b$  (equality test)

*negate*:  $-a$



# `int` Data Type

---

1. what *values* can the `int` type store?

"int" can store integer values in the range  
-2,147,483,648 to +2,147,483,647



# int Operations

## Arithmetic (return int)

$a + b$

$a - b$

$a * b$

$a / b$

$a \% b$      $a$  modulo  $b$

## Operations that shift bits

$a \ll n$     shift bits left  $n$  times

$a \gg n$     shift bits right w/  
sign

$a \ggg n$  shift bits right

## Comparison Operations

$a < b$

$a > b$

$a \leq b$

$a \geq b$

$a == b$

$a != b$

## Bit mask operations

$a | b$  bitwise "or" of  $a$ ,  $b$

$a \& b$  bitwise "and" of  $a$ ,  $b$

$a \wedge b$  bitwise exclusive or



## Example using "int" type

---

Add the numbers 1 to 100.

```
int max = 100;  
  
int sum = 0;  
  
for( int k=1; k <= max; k++ )  
    sum = sum + k;  
  
System.out.println( "sum is " + sum );
```



# `int` Special Values

---

The Integer *class* has 2 special "int" values:

`Integer.MIN_VALUE` is the minimum value of "int" type.

`Integer.MAX_VALUE` is the maximum value of "int" type.



# Rules for int operations

---

1. If the result is TOO BIG for "int" type, the higher order bits are lost. The result will be incorrect:

1,000,000,000 + 1,000,000,000 is 2,000,000,000

2,000,000,000 + 1,000,000,000 is -1,294,967,296

2. On division of int/int the remainder is discarded.

28 / 10 is 2

-28 / 10 is -2

1 / 2 is 0 even 999999 / 1000000 is 0

1 / 0 is error. Throws DivisionByZero exception.

3. Modulo (%):  $m = a \% b$  is such that  $b * (a / b) + m == a$

7 % 3 is 1, -7 % 3 is -1 but 7 % -3 is 1





# Java Primitive Data Types

---

<u>Name</u>	<u>Values</u>	<u>Examples</u>
<b>boolean</b>	<b>true false</b>	true, false
<b>char</b>	<b>character</b>	'a', 'A', '1', '!', '@', '\n', '\t'
<b>byte</b>	8-bit integer	-127, ..., -1, 0, 1, ..., 127
<b>short</b>	16-bit integer	-32768 ... 0 ... 32767
<b>int</b>	<b>32-bit integer</b>	-400 47 20000000
<b>long</b>	64-bit integer	-1234567890L 0L 888L
<b>float</b>	decimal	3.14159F 0.0F -2.5E-8F
<b>double</b>	<b>64-bit decimal</b>	3.14159265358979E234



# Primitive Data Types: values

---

<u>Data Type</u>	<u>Size in Memory</u>	<u>Range of Values</u>
boolean	1 byte	true false
char	2 bytes	0 (null) - \uFFFF (Unicode)
byte	1 byte	-128 to 127
short	2 bytes	-32,768 to 32,767
int	4 bytes	-2,147,483,648 to 2,147,483,647
long	8 bytes	-9,223,372,036,854,775,808L 9,223,372,036,854,775,807L
float	4 bytes	$\pm 3.402823E+38$
double	8 bytes	$\pm 1.797693134623157E+308$



# double

---

1. Any number written with "." or exponential is automatically of type `double` (not `float`).

`double: 1.0    3.14159    2.99E+8    3e-12`

2. If you do `+`, `-`, `*`, `/` with `int` and `double`, the result is a `double`. the "int" is promoted to `double` first.

`2 * 7.0 --> 14.0 (double)`

`10.0 * 2 / 5 --> 4.0 (double)`

but: `2 / 5 * 10.0 --> 0` ("`2/5`" is done first as `int/int`)

3. `*` and `/` are done before `+` and `-`

`1.5 + 10 * 7.0 --> 71.5`



# special values: Infinity and NaN

---

Java uses the IEEE floating point standard.

There are 3 special values: +Infinity, -Infinity, and NaN (not a number).

`2.5 / 0.0 is +Infinity`

`-2.5 / 0.0 is -Infinity`

`0.0 / 0.0 is NaN (not a number)`

`Infinity * 0.0 is NaN`

For int and long, `n / 0` is error (DivisionByZeroException)  
but for float and double, `x / 0` is +/-Infinity.



# What Data Type?

---

\_\_\_\_\_ 1234, -9999

\_\_\_\_\_ 6010541234 (in Java: 6010541234L)

\_\_\_\_\_ 3.14159 (*what is this?*)

\_\_\_\_\_ 2.99792E+08 (*what is this?*)

\_\_\_\_\_ true

\_\_\_\_\_ '2'

\_\_\_\_\_ "2"

\_\_\_\_\_ '๑'

\_\_\_\_\_ 3 == 4



# Rules for numeric values

---

- Java has rules for how it interprets numerical values.

## Value   Meaning

4      an "int" value 4

4L    a "long" with value 4 (uses 8 bytes)

4.     a "double" with value 4.0

3e4, 3.0E4, 3e+4 a "double" with value 3000.0 ( $3 \times 10^4$ )

0.1    a "double" value 0.1 *approximately*

4.0F a "float" value 4.0 (uses 4 bytes)

'4'    a "char" with value 52 (int)



# Type Conversion

---

If your code contains: `2+3`

then Java sees that you are adding two "int" values and produces an **int** result (5).

But, if your code contains: `2+3.0`

it means to add an "int" and a "double" value.

In this case, Java will first convert 2 to a double (2.0) and add 2.0+3.0. The result is a **double**.

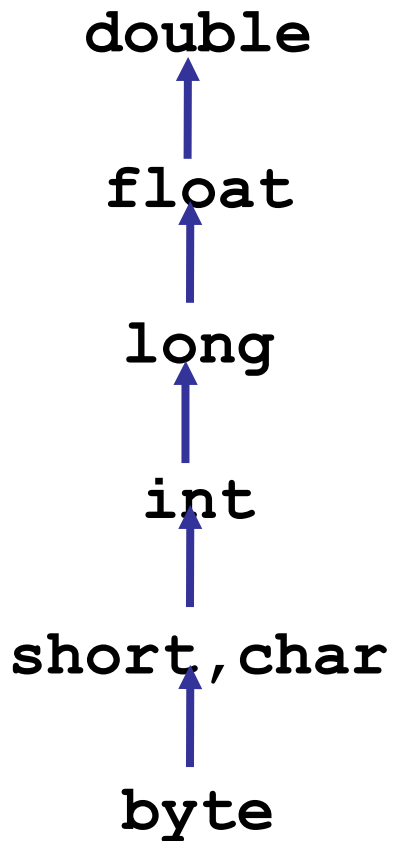
Type conversion may also occur when you call a method. For example: `Math.sqrt(2)`

The `sqrt` method requires a double argument, so Java "promotes" 2 (int) to 2.0 (double).



# Automatic Type Promotion

If you do arithmetic on different data types, Java "promotes" one argument to the type with *widest* range.



## Example

2 + 4L

2 \* 4.0

2F + 3

2.0 \* 3

## Promotion

2 -> (long)2L

2 -> (double)2.0

3 -> (float)3F

3 -> (double)3.0

## Result

6L (long)

6.0 (double)

5.0F (float)

5.0 (double)

## Weird:

'a'+1

'a' -> int (97)

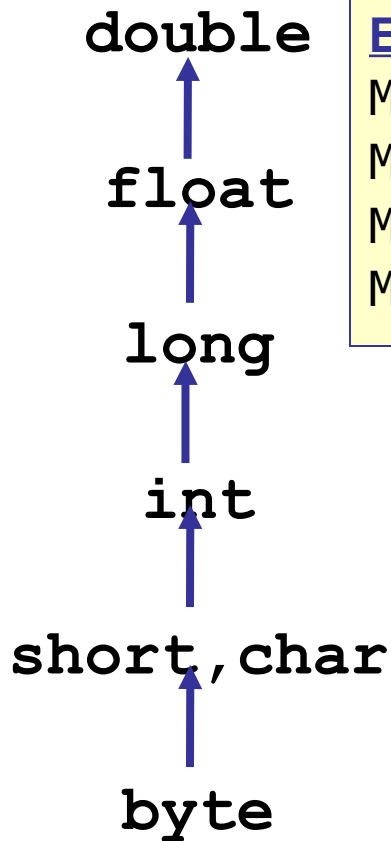
98





# Type Promotion & Functions

If you invoke a function (method) using a numeric value, Java may "promote" the values of arguments.



## Example

`Math.sqrt( 2 )`

`Math.max( 2, 10.0F )`

`Math.max( -1, -4L )`

`Math.max( 3, 2.236 )`

## Promotion

2 to 2.0

2 to 2.0F

-1 to -1L

3 to 3.0

## Then Call

`sqrt(2.0)`

`max(2F, 10F)`

`max( -1L, -4L)`

`max(3.0, 2.236)`

## *Type Conversion May Lose Precision*

Java "type promotion" always perform a *widening conversions* that will never "overflow" the result data type.

But it *may lose precision (accuracy)*.

Example: `(float)123456789 -> 1.2345679E+8`



# What about boolean?

---

boolean type (true, false) **cannot be converted** to any other type!

This is done to prevent accidental errors.

A classic error in C programming is:

```
int n = 1;  
if (n = 2) printf("its true!"); // set n=2, result is true!  
should be:  
if (n == 2) . . . ;
```



# Common Type Errors

---

Here are some common errors.

What is the mistake? How to correct it?

```
// Compute typing speed in words/minute
int wordsTyped = 38; // number of words typed
int time = 45; // time in seconds
double speed = wordsTyped/time * 60.0; // speed = 0

// The midterm exam has a maximum of 90 points.
// "Normalize" the score to be 0-100 (e.g. 90 -> 100%).
int midtermScore = 85;
double score = 100.0 * (midtermScore / 90);
```



# boolean values

- Boolean has 2 values: **true** or **false**
- Used for *conditional execution* of statements.
- Boolean is used in "if", "while", and "for" statements.

```
/** Compute the sales tax on a purchase */
```

← A javadoc  
comment for  
this method.

```
public void getTax( int amount ) {  
    boolean PAY_TAX = true;  
    double tax; // amount of tax owed  
    if ( PAY_TAX ) tax = 0.07 * amount;  
    else tax = 0.0;  
    System.out.println("The tax is: "+tax);  
}
```

← if ( *condition* )  
    *statement1* ;  
    else  
        *statement2* ;



# boolean operations

<code>! b</code>	NOT b ( <code>!true -&gt; false, !false -&gt; true</code> )
<code>b1 &amp;&amp; b2</code>	b1 AND b2
<code>b1    b2</code>	b1 OR b2
<code>b1 ^ b2</code>	b1 XOR b2 true if <i>exactly one</i> of b1, b2 is true

```
boolean hasDog = true;  
boolean hasCat = false;
```

```
// test: does he have a dog or a cat?  
if ( hasDog || hasCat ) petOwner( );  
// test: does he have dog or cat, not both?  
if ( hasDog ^ hasCat ) happyPetOwner( );  
// does he have both dog and cat?  
if ( hasDog && hasCat ) unhappyPetOwner( );
```



# boolean operations

---

It is *always* possible to rewrite  $\wedge$  (exclusive or) using AND, OR, and NOT ( $\&\&$ ,  $\|\|$ ,  $!$ )

Exercise: rewrite expression without using  $\wedge$

```
boolean hasDog = true;
boolean hasCat = false;
happyPetOwner = ( hasDog ^ hasCat );

// write happyPetOwner
// using only &&, ||, and !
happyPetOwner =
```



# char for character data

- The char data type is for character data.
- Java uses 2-byte Unicode for character data, in order to hold the world's alphabets. Including Thai.
- Unicode: <http://www.unicode.org>

```
// Get the first character from a String.  
String word = "George Bush";  
char first;  
first = word.charAt(0);  
System.out.println("The string "+ word  
    + " begins with " + first);  
  
// Get the last character from a String!  
int last = word.length() - 1; // why -1 ??  
first = word.charAt( last );
```

charAt( ) is  
a method of  
the String  
class.

length( )  
returns number  
of chars in a  
string.



# char values

- You can also use `char` to hold special values:
  - `'\t'` tab character
  - `'\n'` new-line character
  - `'\u03C0'` Unicode sequence number for  $\pi$  (pi)

```
char TAB = '\t';
char NEWLINE = '\n';
char PI = '\u03C0';
// Print greek pi symbol
System.out.println("I love cake and "+PI);
// Use tab to align output
System.out.print("Hello" + NEWLINE
                + TAB + "world"+NEWLINE);
```

Must enclose  
character  
values in  
***single quotes***

**NOT**  
*double quotes*





# Escape Sequences for special chars

---

These '\x' values represent special characters:

<u>Code</u>	<u>Name</u>	<u>meaning</u>
\t	Horizontal Tab	advance to next tab stop
\n	New line	start a new line
\v	Vertical Tab	performs a vertical tab (maybe)
\f	Form feed	start a new page on printed media
\r	Carriage return	move to beginning of line
\0	Null	null character, has value 0
\"	Double Quote	use for " inside of String
\'	Single Quote	use for ' inside of char
\\	Backslash	display a \



# byte, short for "raw" data

- `byte` and `short` are for integer data and input/output
- `byte` is used for low-level input, holding character codes (as 1 byte), and groups of "flag" bits
- `byte` and `short` are **not used for arithmetic**.  
Java *promotes* all arithmetic to "int" data type.

```
/* read bytes of data into byte array.  
 * This is soooo boring.  
 */
```

```
byte[] b = new byte[80];  
System.in.read( b );
```

← `read( )` gets  
input data as  
bytes.



# Detailed Look at Float & Double

---

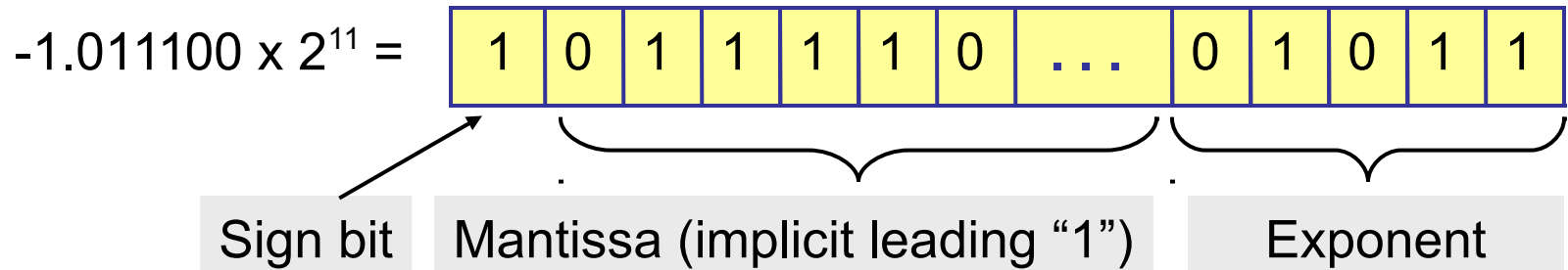
The next few slides explain how float and double values are stored.

You can skip them if you want.

But, to understand the *behavior* of arithmetic operations it helps to know how values are stored.

# float, double: Floating Point Data

Java has 2 data types for storing non-integer values, called *floating point* because they store numeric data as a *mantissa* and *exponent*.



<b>Float:</b>	1 bit	23 bits	8 bits
<b>Double:</b>	1 bit	52 bits	11 bits
		Precision	Range
<b>Float:</b>		24 bits $\approx$ 7 dec. digits	$10^{-38} - 10^{+38}$
<b>Double:</b>		53 bits $\approx$ 15 dec. digits	$10^{-308} - 10^{+308}$



# float, double: Floating Point Data

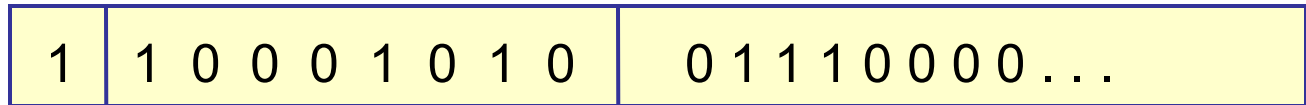
Data Type	Size of mantissa	Accuracy (precision)
<b>float</b>	23 bits	6-7 decimal digits
<b>double</b>	52 bits	15 decimal digits

- ❑ Use **double** for most applications (more accurate).
- ❑ Use **float** where 6-decimal digits is enough, *or* you need to optimize space/performance.

```
// Be careful when using floating point!  
float x = 0.2F;  
float y;  
y = 1.0F - x - x - x - x - x; // should be zero!  
System.out.println("y = "+y); // y = 2.9802322E-8
```

# IEEE Floating Point Data Format

$-1.011100 \times 2^{11} =$



Sign bit

Biased Exponent

Mantissa

**Float:**      1      8 bits    bias= 127      23 bits

**Double:**    1      11 bits    bias=1023      52 bits

Range

Precision

**Float:**       $10^{-38} - 10^{+38}$       24 bits  $\approx$  7 dec. digits

**Double:**     $10^{-308} - 10^{+308}$       53 bits  $\approx$  15 dec. digits

Stored exponent = actual exponent + **bias**



# Wrapper Classes

## Primitive

boolean

char

byte

short

int

long

float

double

## Wrapper

Boolean

Character

Byte

Short

Integer

Long

Float

Double

```
double root = Math.sqrt( 2.0 );
```

```
Double d1 = new Double( root );
```

```
// same thing: automatic boxing
```

```
Double d2 = root;
```

```
// print as a string
```

```
out.println( d2.toString( ) );
```

```
// static method to make a string
```

```
out.println( Integer.toString( 2 ) );
```



# Why Wrapper Classes?

---

1. Some methods and data structures only work with *references* (e.g. *objects*).

**Example:** a List can only contain *references*.

If we want a List of double, we need to "wrap" each double in an object.

```
// ERROR: can't create a list of primitives
ArrayList<double> list = new ArrayList<double>( );

// CORRECT: use wrapper for double
ArrayList<Double> list = new ArrayList<Double>( );

// Java automatically "wraps" 2.0 in a Double
list.add( 2.0 );
```





# Why Wrapper Classes?

---

2. Primitives don't have methods. The wrappers provide useful methods and static constants.

**Example:** get the double value of a String.

```
// convert a String to a double
double x = Double.parseDouble( "2.98E_08" );

// convert double to a String
x = Math.sqrt( x );

String value = Double.toString( x );
```

**Example:** what is the largest value an "int" can store?

```
int max = Integer.MAX_VALUE;
```



# Wrapper to convert to/from String

---

```
int n = 1234;  
// convert n to a String  
String id = Integer.toString(n) ;  
  
String s = "2.5";  
// convert s to a double?
```



# Range limits of numeric types

---

- What is the largest "int" value?
- What is the smallest "long" value?
- What is the range (smallest, biggest) of double?

```
int biggest =
```

```
long smallest =
```

```
double minimum =
```

```
double maximum =
```



# What happens if you go beyond?

```
int n = Integer.MAX_VALUE;  
n = n + 1;  
System.out.println( n );  
double d = Double.MAX_VALUE;  
d = d + 1;  
System.out.println( d );  
d = d * 1.000001;  
System.out.println( d );
```



# What happens if you go beyond?

```
int n = Integer.MAX_VALUE;
```

```
n = n + 1;
```

```
n is -2147483648
```

```
double d = Double.MAX_VALUE;
```

```
d = d + 1;
```

```
no change. +1 insignificant (too small)
```

```
d = d * 1.000001;
```

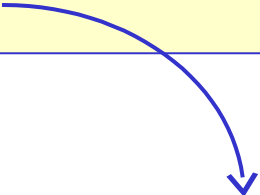


```
d is Infinity
```



# C# numerics are different

- "int", "float", "double" are **struct** types.

```
// This is C#  
int n = int.MaxValue;  
String s = "Biggest int is "  
           + n.ToString( ) ;  
  
// range checking is enforced  
n = n + 1;
```



System.OverflowException: Arithmetic operation resulted in an overflow.



# Review

---

1) Is this correct? Give a reason why or why not.

```
int n = 1234;
```

```
System.out.println( n.toString() );
```

2) How can you convert a String value to a double?

```
String s = "9.8E+6";
```

```
double value = ?
```



# Review

---

Taksin deposited 1,000,000,000 Baht at the bank on 3 occasions. The first 2 times the balance was correct. But the third time the balance was negative. Why?

Here is the code (you can run this in BlueJ codepad):

```
int balance = 0; // initial balance
int deposit = 1000000000; // a small deposit
for(int count=0; count < 3; count++) {
    balance = balance + amount;
    System.out.println("Balance is "+balance);
}
```