

# Lecture 3 — Types, Literals, and Operators

## CITS2005 Object Oriented Programming

Department of Computer Science and Software Engineering  
University of Western Australia

# Contents

- See Chapter 2 of the textbook
- Primitive and non-primitive types
- Ints, floats, characters, strings, and booleans
- Literals and Operators

# Variables, types, and operators

- We learned about *variables* and their *types* (e.g., `int x;`)
- The assignment *operator* (e.g., `x=10;`)
- Arithmetic *operators* (e.g., `+`, `-`, `*`, `/`, `%`)
- Relational *operators* (e.g., `==`, `!=`, `<`, `>`, `<=`, `>=`)
- Let's look at these in more detail

# Types

- We have looked at the `int`, `double`, and `String` types
- What other types exist in Java?
- Types are either *primitive* or *non-primitive*
- *Primitive* types are built into Java (e.g., `int`)
- They are usually stored directly and are efficient
- *Non-primitive* types are defined by code in a `class` (e.g., `Scanner` from Lecture 2)
- They are stored as *objects*

# Primitive Types

<code>int</code>	32-bit Integer Values ( $[-2^{31}, 2^{31} - 1]$ )
<code>long</code>	64-bit Integer Values ( $[-2^{64}, 2^{64} - 1]$ )
<code>byte</code>	8-bit Integer Values ( $[-2^8, 2^8 - 1]$ )
<code>short</code>	16-bit Integer Values ( $[-2^{16}, 2^{16} - 1]$ )
<code>float</code>	32-bit Floating Point Values
<code>double</code>	64-bit Floating Point Values
<code>boolean</code>	Either <code>true</code> or <code>false</code>
<code>char</code>	A single character (e.g., a letter)

- Notice that all primitive type names are lowercase
- First 4 all store integers, `int` is most commonly used
- Next 2 store floating point numbers, `double` is most common
- `boolean` is often seen in the context of relational operators like `<`
- `char` is often seen in the context of `String` (more later)

# Literals

- *Literals* are values (not variables) that appear in code
- Often said to be constant
- `int x = -10;`
- This is an *integer literal*
- Literals are (almost) always primitive types

# Integer Types and Literals

- We've already seen some integer literals
- e.g., 10, -10, 100, 324928, -328347, 0
- Any integer value
- By default, they are of type `int` (32-bit)
- Sometimes, we want numbers too big to fit into 32-bits
- We can write a long literal: `long bigValue = 1234567891011L;`
- See the `L` suffix

# Integer Types and Literals

- Integers support the following arithmetic operators
- Increment and decrement can be used prefix (`++i`) or postfix (`i++`)

+	Addition	5+3 (=8)
-	Subtraction	5-3 (=2)
*	Multiplication	5*3 (=15)
/	Division	5/3 (=1), 6/3 (=2)
%	Modulus	5%3 (=2), 6%3 (=0)
++	Increment	i++ (=i+1)
--	Decrement	i-- (=i-1)



# Integer Types and Literals

- You can use `_` to make integer literals easier to read
- e.g., `100_000`, `-89_111`, `12_34_56_78`
- The `_` can be inserted anywhere

# Integer Types and Literals

---

```
public class IntTypes {  
    public static void main(String[] args) {  
        int a = 1_000_000_000_000; // error: integer number too large  
  
        /*  
        error: incompatible types: possible lossy conversion from long to int  
        Note: need to remove the other errors before we see this one.  
        Also note: this is a multiline comment!  
        */  
        int b = 1_000_000_000_000L;  
  
        long c = 1_000_000_000_000; // error: integer number too large  
  
        long d = 1_000_000_000_000L;  
    }  
}
```

# Integer Types and Literals

---

```
public class IntLiterals {  
    public static void main(String[] args) {  
        int a = 1_000_000; // Decimal  
        int b = 0b101; // 0 or 1.  $1*2^2 + 0*2^1 + 1*2^0$   
        int c = 0x1_F; // 0-9, A-F.  $1*16^1 + 15*16^0$   
        int d = 0172; // 0-7.  $1*8^2 + 7*8^1 + 2*8^0$   
        System.out.println(a);  
        System.out.println(b);  
        System.out.println(c);  
        System.out.println(d);  
    }  
}
```

---

- You do not need to write integer literals in base-10
- Binary (base-2, very useful), Hexadecimal (base-16, occasionally useful), Octal (base-8, I've never used it)

# Integer Types and Literals

---

```
public class IntLiterals2 {  
    public static void main(String[] args) {  
        long a = 0b1111111111111111111111111111111111111111L;  
        long b = 0x1FFFFFFFFFFFL;  
        System.out.println(a);  
        System.out.println(b);  
    }  
}
```

---

- You can combine these notations with longs

# Float Types and Literals

+	Addition	5.0+3.5 (=8.5)
-	Subtraction	5.0-3.5 (=1.5)
*	Multiplication	5.0*3.5 (=17.5)
/	Division	5.0/3.5 ( $\approx$ 1.4285)
%	Modulus	5.0%3.5 (=1.5)
++	Increment	i++ (=i+1.0)
--	Decrement	i-- (=i-1.0)

- Floating point numbers are those with a decimal place
- float, and double
- They support the same arithmetic operators as integers but with different results

# Float Types and Literals

---

```
public class Floats {  
    public static void main(String[] args) {  
        float f = 1.1f;  
        double d = 1.0;  
        ++f;  
        d = (d / 7.0)*1000000.0;  
        System.out.println(f);  
        System.out.println(d);  
    }  
}
```

---

- All literals are double by default
- The `f` suffix tells Java this is a float literal
- Scientific notation is also allowed: `double x = 2.3e8;`
- Did you notice that the value of `d` is slightly wrong? (see next slide)

# Float Types and Literals

- Floating point numbers are (often) approximate
- They are stored in binary, and must use a fixed number of bits
- Try representing  $\frac{1}{3}$  in decimal: 0.333333...
- The same thing happens in binary: 0.01010101...

# Characters

- The *character* (`char`) type in Java represents a “letter”
- Specifically, “letters” are *unicode* (UTF-16) symbols
- <https://www.ssec.wisc.edu/~tomw/java/unicode.html>
- Represented like an `int`, but Java knows which number corresponds to which symbol



# Characters

---

```
public class Chars {  
    public static void main(String[] args) {  
        char a = 'a'+1;  
        char b = 'b';  
        System.out.println(a);  
        System.out.println(b);  
    }  
}
```

---

- Character literals use 'single quotes': 'x'
- Dynamic initialisation: char a = 'a'+1;
- Notice that these print out the same value

# Strings

---

```
public class Strings {  
    public static void main(String[] args) {  
        String hello = "Hello";  
        String world = "World";  
        System.out.println(hello + " " + world);  
    }  
}
```

---

- Strings are a sequence of 0 or more chars
- String literals use “double quotes”: "Hello, CITS"
- They support the concatenation + operator

# Strings

---

```
public class StringConcat {  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 20;  
        String s1 = a+b+"";  
        String s2 = ""+a+b;  
        System.out.println(s1);  
        System.out.println(s2);  
    }  
}
```

---

- String concatenation can be used with different types
- It is executed left-to-right
- This sometimes has unexpected consequences

# Strings

```
public class SpecialCharacters {  
    public static void main(String[] args) {  
        char tab = '\t';  
        // Concatenation of a char  
        System.out.println("Hello" + tab + "World");  
        String s = "backslash: \\, double quote: \", single quote: \'  
        System.out.println(s+"\n"+"another line!");  
    }  
}
```

- There are some special characters for those you cannot easily type
- These are achieved using an *escape sequence* starting with a backslash
- <https://docs.oracle.com/javase/tutorial/java/data/characters.html>

# Strings

- Are Strings primitives?
- There are String literals like other primitives: `"literal"`
- But their name is not lowercase: `String` vs `char`
- Some say `String` is a primitive type, others say it is not (including our textbook)
- It shares elements of both
- It is built-in and there are String literals, but Strings are objects

# Booleans

```
public class Booleans {  
    public static void main(String[] args) {  
        boolean a = true;  
        boolean b = false;  
        if (a)  
            System.out.println("a");  
        if (b)  
            System.out.println("b");  
    }  
}
```

- boolean only has two values: true or false
- These are the only two literals
- We saw some boolean logic in if statements
- if (something\_that\_is\_boolean) statement;

# Booleans

- The result of *relational* or *logical* operators are boolean
- For example, `x < y` is true if `x` is less than `y`, and false otherwise
- Also, `x && y` is true if both `x` and `y` are true

# Booleans

---

```
public class BooleanOperators {  
    public static void main(String[] args) {  
        int x = 5;  
        if (x >= 1 && x <= 10)  
            System.out.println("x is between 1 and 10");  
        else if (x >= 11 && x <= 20)  
            System.out.println("x is between 11 and 20");  
    }  
}
```

---

- How does java know to execute <= before &&?
- This is called *operator precedence*, <= has higher precedence



# Relational Operators

<code>==</code>	Equal to	<code>2==3</code> (false)
<code>!=</code>	Not equal	<code>2!=3</code> (true)
<code>&lt;</code>	Less than	<code>2&lt;3</code> (true)
<code>&gt;</code>	Greater than	<code>2&gt;3</code> (false)
<code>&lt;=</code>	Less than or equal	<code>2&lt;=3</code> (true)
<code>&gt;=</code>	Greater than or equal	<code>2&gt;=3</code> (false)

- Relational operators work on numeric types (e.g., `int`, `double`) including `char`
- The `==` and `!=` operators work on `booleans` too

# Relational Operators

---

```
public class RelationalOperators {  
    public static void main(String[] args) {  
        boolean x = 1 < 2;  
        boolean y = 7 != 7;  
        boolean z = x == y;  
        if (z != true)  
            System.out.println("z is false");  
    }  
}
```

---

- Does the println happen? Why or why not?

# Logical Operators

<code>&amp;</code>	AND
<code> </code>	OR
<code>^</code>	XOR
<code>&amp;&amp;</code>	AND (short-circuit)
<code>  </code>	OR (short-circuit)
<code>!</code>	NOT

- More about (some of) these in the first lab
- Also about operator precedence
- We will assume you know it later, so please make sure to do the lab this week!
- For now, let's end with a puzzle

# Logical Operators

---

```
public class LogicalOperators {  
    public static void main(String[] args) {  
        boolean mystery = (1<3) || (3>2) && !(3<4);  
        if (mystery)  
            System.out.println("mystery is true");  
        else  
            System.out.println("mystery is false");  
    }  
}
```

---

- What is mystery?