Data types

JC08-4.1

(A. Nguyen)

Values in Java

- A value in Java is:
  - A reference to an object, or
  - One of 8 primitive data types
- We will concentrate on 4 primitive data types:

TYPE	DESCRIPTION	SIZE
`int`	The integer type, with range -2,147,483,648 (`Integer.MIN_VALUE`) ... 2,147,483,647 (`Integer.MAX_VALUE`)	4 bytes
`double`	The double-precision floating-point type, with a range of about $\pm 10^{308}$, and about 15 significant decimal digits	8 bytes
`char`	The character type, representing code units in the Unicode encoding scheme	2 bytes
`boolean`	The type with the two truth values falseand true	1 bit

- Note: a 10-digit phone number would **overflow** an `int` – will need `long`

`int` in Java

	VALUE	COMMENTS
`Integer.MIN_VALUE`	-2^{31} or -2,147,483,648	
`Integer.MIN_VALUE-1`	2147483647	Overflow, becoming positive!
`Integer.MIN_VALUE+1`	-2,147,483,647	
`Integer.MAX_VALUE`	2^{31}-1 or 2,147,483,647	
`Integer.MAX_VALUE-1`	2147483646	
`Integer.MAX_VALUE+1`	-2147483648	Overflow, becoming negative!

Number Literals (& errors)

From Java Concepts, 8e, C. Horstmann

Table 2 Number Literals in Java

Number	Type	Comment
`6`	`int`	An integer has no fractional part.
`-6`	`int`	Integers can be negative.
`0`	`int`	Zero is an integer.
`0.5`	`double`	A number with a fractional part has type `double`.
`1.0`	`double`	An integer with a fractional part .0 has type `double`.
`1E6`	`double`	A number in exponential notation: 1×10^6 or 1000000. Numbers in exponential notation always have type `double`.
`2.96E-2`	`double`	Negative exponent: $2.96 \times 10^{-2} = 2.96 / 100 = 0.0296$
`100000L`	`long`	The `L` suffix indicates a long literal.
`100,000`		**Error:** Do not use a comma as a decimal separator.
`100_000`	`int`	You can use underscores in number literals.
`3 1/2`		**Error:** Do not use fractions; use decimal notation: 3.5

Rounding Errors

- Rounding errors occur when an exact representation of a floating-point number is not possible.
- Floating-point numbers have limited precision. Not every value can be represented precisely, and roundoff errors can occur.
- Example

```
double f = 4.35;
System.out.println(100 * f); // Prints 434.99999999999994
```

Convert between data types, or cast

- The quickest way to convert a number to a string is to concatenate (i.e., with addition symbol) it with an empty string (i.e., a pair of double quotes): `"" + 2 + 3` is the string `23` (i.e., not integer 23)
- Java *automatically* converts a "smaller" number to a "larger" one:

```
int n = 2;
double x = n; // OK to put an int into a double
```

- The programmer must *explicitly* convert or cast a "larger" number to a "smaller" one:

```
double x = 3.6;
int n = (int)x; // cast a double to an int
```

final and static

- Reserved word `final` indicates that the value is a constant, i.e., once assigned, the value cannot be changed (or else syntax error) – note that the reserved word is NOT `const`:

```
final static double QUARTER_VALUE = 0.25;
double circumference = Math.PI * diameter;
```

- Final values are named with upper-case letters and underscores
- Reserved word `static` indicates that the value ***belongs to the class***; i.e., only ***one slot of RAM to be shared by all live objects***:

```
final static double QUARTER_VALUE = 0.25;
static Color.BLACK
```

Why Symbolic Constants?

- Easy (for the programmer) to change the value throughout the program, if necessary
- Easy to change into a variable
- More readable, self-documenting code; e.g., `QUARTER_VALUE` is meaningful, but not .25, embedded in the middle of the code
- Additional data type checking by the compiler

3 kinds of variables: comparison

	FIELD	LOCAL VARIABLE	(FORMAL) PARAMETER
pubic vs. private	must be specified; if not, same as public	not specified, or else compile error	not specified; or else compile error
initial value	defaulted as zero or null if not explicitly initialized	must be initialized before used, or else compile error	value same as passed from caller
assigned	assigned by any method in the class (if private); assigned by any method constructing the object (if public)	assigned after declared, within its scope	used (but not reassigned) by the called method Note: a copy of the param is made locally
scope (shown in different color boxes in BlueJ)	if private, scope is the entire class; if public, accessible & modifiable by clients (i.e., objects of other classes)	scope is from the declaration time until the ending brace of the block containing the declaration	scope is the called method (i.e., the method where the parameter is passed to)

How to choose data type

- For info that is integers (e.g., student ID, room number), choose `int`
- For info that is floating-point (e.g., money amount, GPA), choose `double`
- For info that is one-character text (e.g., initial of middle name), choose `char`
- For info that represents 2 possible values such as true/false or yes/no (e.g., whether an item has been found), choose `boolean`
- **NOTE**: Make sure that the info will fit into the storage by the data type; an `int` is sufficient for a social security number, but not for area code + phone number

Math class – see Java API

- Values (name in upper case only) – static:
  - pi value: **`Math.PI`**
  - Euler's number e: **`Math.E`**
- Some methods (name starts with lower case), returning `double`:
  - Get power value: **`Math.pow`**`(double base, double exponent)`
  - Get random number in [0, 1) interval: **`Math.random`**`()`
  - Get square root value: **`Math.sqrt`**`(double someNumber)`
  - Get rounded value: **`Math.round`**`(double someNumber)`
  - Get natural log value: **`Math.log`**`(double someNumber)`
  - Get common log value: **`Math.log10`**`(double someNumber)`
  - Get sin value: **`Math.sin`**`(double someAngle)`

Math class (cont.) – see Java API

- For absolute value, returning the appropriate data type:
  - `Math.abs(…)`

return statement

JC08-4.1

(A. Nguyen)

Characteristics of `return`

- When a program encounters a `return` statement, it quits that method.
- If the method does not require an output (i.e., `void`), then
  - **`return;`** simply quits the method
  - **`return;`** is NOT REQUIRED if the method ends at the closing brace of the method's body.
- If the method requires an output/return, then
  - **`return …;`** must be followed by a value of the output/return data type
  - **`return …;`** is REQUIRED, and is the last statement of the method for normal end
- A statement that follows a `return` statement (in the same block) will cause a syntax error.
- A method that is supposed to "return" something means a method WITH OUTPUT to its caller; it does NOT mean a method that `print` or `println` something to the console window.

Example of `return` in a method w/o output

```
public void doSomething()
{
    …
    if ( … )
    {
        …
        return;
        // no more statements here, or syntax error
    }
    else
       …
    …
}
```

← `void` indicates no output

← special-case return, in the middle of a method body

← normal return, at the ending brace of a method body; statement is not required

Example of `return` in a method w/ output

```
public double calcArea(double radius)
{
    double area = 0.0;
    if (radius < 0.0 )
        area = -1.0; // neg. area to indicate bad input
    else
      area = Math.PI * radius * radius;
    return area;
    // no more statements here, or syntax error
}
```

← `double` indicates with output, of type double

← REQUIRED return, and WITH output

Operations

JC08-4.2

(A. Nguyen)

Arithmetic

- Operators: `+`, `-`, `/`, `*` , %
- The precedence of operators & parentheses is the same as in algebra (PMDAS) – See App B – Java Operator Summary
- Exp. is `Math.pow(double base, double exp)`
- `m % n` means the remainder when m is divided by n:
  - 17 % 5 is 2
  - 3 % 8 is 3
- To check whether an integer `n` is even: `if (n % 2 == 0)`
- % has the same rank as * and `/`
- Same-rank binary operators are performed in order from left to right

Arithmetic (cont.)

- The type of the result is determined by the types of the operands, not their values; this rule applies to all intermediate results in expressions. If both operands are of the same type, then the result is that type; otherwise, it is of the "larger" type.
- If one operand is an `int` and another is a `double`, the result is a `double`; if both operands are `int` s, the result is an `int`.

Arithmetic (cont.)

- **Caution**: if a and b are ints, then a / b is ***truncated*** to an int…

  17 / 5 gives 3

  3 / 4 gives 0

- …even if you assign the result to a double:

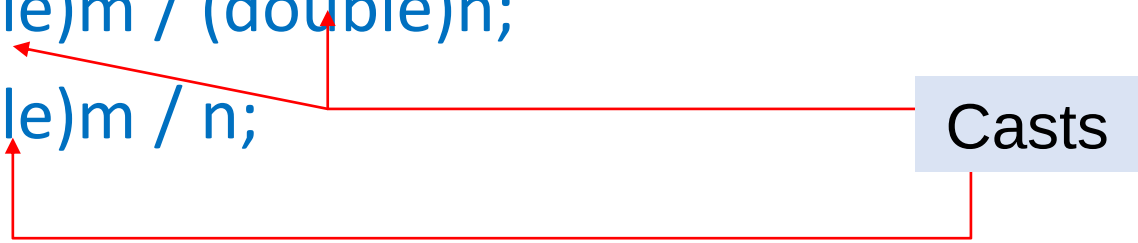
```
double ratio = 2 / 3; // this is int division, then assigned to double
```

The **double** type of the result doesn't help: **ratio** still gets the value **0.0**.

Arithmetic (cont.)

- To get the correct double result, use double constants or the *cast* operator:

```
double ratio = 2.0 / 3;
double ratio = 2 / 3.0;
 int m = ..., n = ...;
double factor = (double)m / (double)n;
double factor = (double)m / n;
double r2 = n / 2.0;
```



Arithmetic (cont'd)

- Compound assignment operators:

```
a = a + b;   <-->   a += b;
a = a – b;   <-->   a –= b;
a = a * b;   <-->   a *= b;
a = a / b;   <-->   a /= b;
a = a % b;   <-->   a %= b;
```

- Increment and decrement operators:

```
a = a + 1;   <-->   a++;
a = a – 1;   <-->   a––;
```

Do not use these in longer expressions

Formatted “print”: printf

- There is `printf` (“f” is for formatted)
- For example:

```
int m = 5, d = 19, y = 2007;
double amt = 123.5;
System.out.printf (
        "Date: %02d/%02d/%d  Amount = %7.2f\n", m, d, y, amt);
```

  displays:

```
Date: 05/19/2007  Amount =  123.50
```

- %2d asks for 2 spaces for a decimal (i.e., base 10), or integer; %d asks for enough spaces to display an integer
- %7.2f asks for 7 spaces for floating-point number, *including the decimal point* and 2 places for the decimal part

Scope

JC08-4.3

(A. Nguyen)

Definition of scope

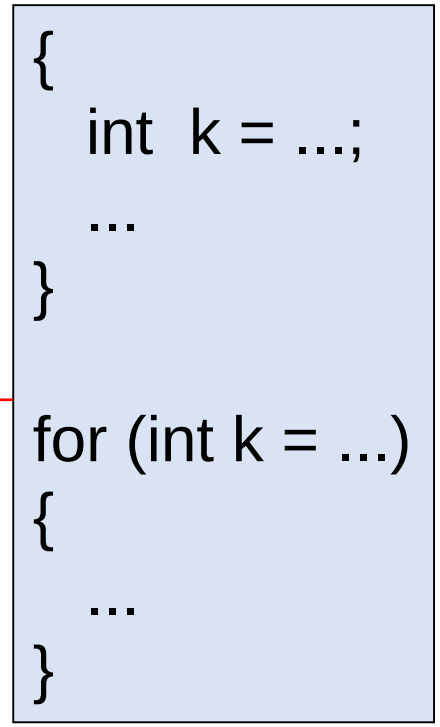
- The scope of a variable is the area of the code where the variable is “visible”
- Two kinds of scopes to consider: for local variables and for fields

Scope of local variables

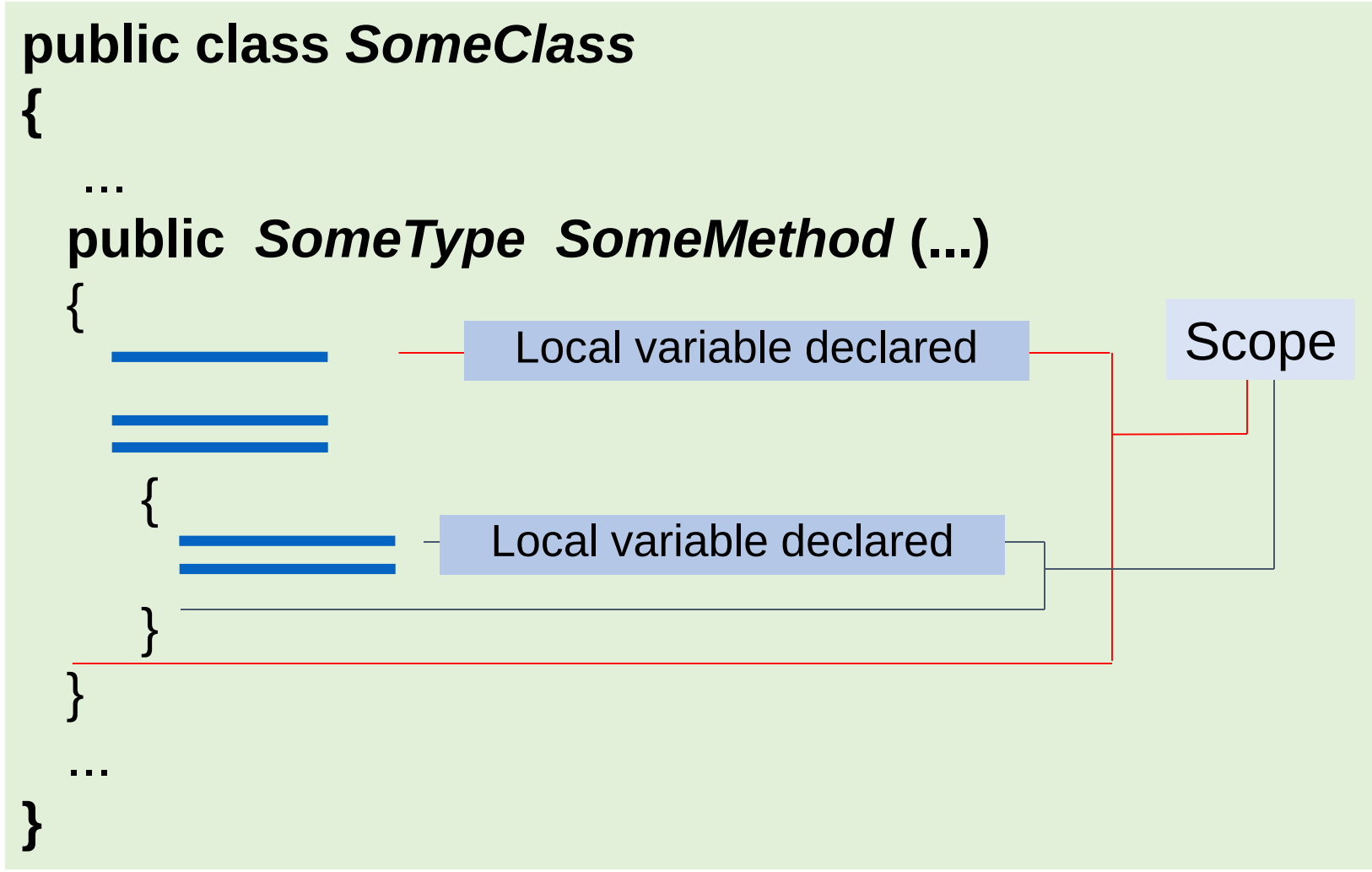
- Local variables are declared inside a constructor or a method.
- Local variables lose their values and are destroyed once the constructor or the method is exited.
- The scope of a local variable is from its declaration down to the closing brace of the block in which it is declared.

Scope of local variables

- Each variable has a scope — the area in the source code where it is "visible."
- If you use a variable outside its scope, the compiler reports a syntax error.
- Variables can have the same name when their scopes do NOT overlap.



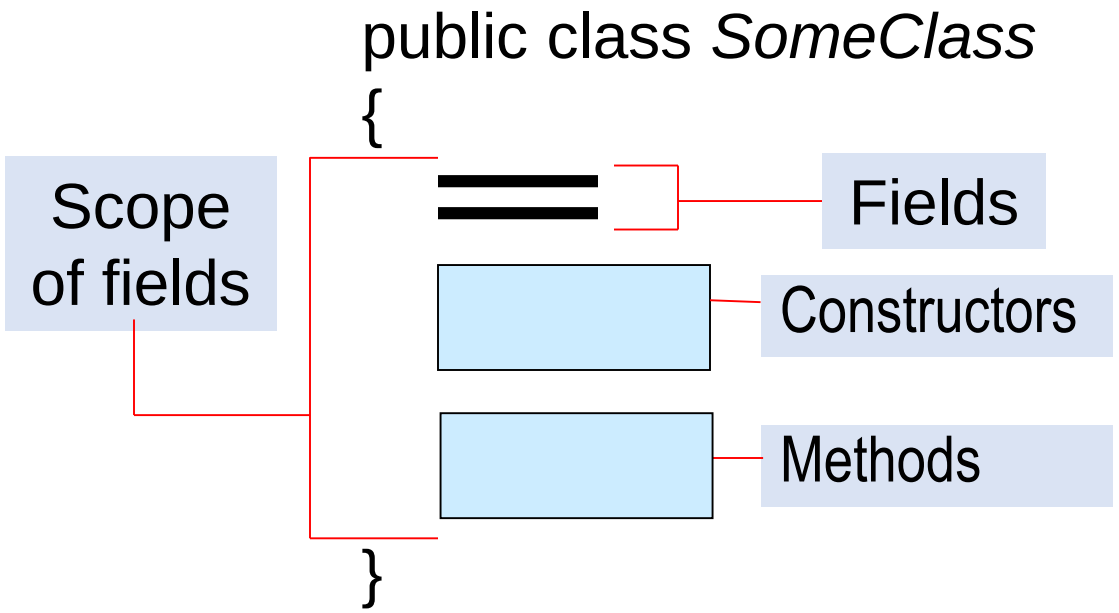
Scope of local variables (cont.)



Scope of fields

- Fields are declared outside all constructors and methods.
- Fields are usually grouped together, either at the top of the class.
- The scope of a field is the whole class, which is why all methods of that class can use all the fields

Scope of fields (cont.)



Error

- Common mistakes – Example: class Circle w/ field `radius`:

```
private double radius; // auto. assigned: 0.0
...
public Circle (...)       // constructor
{
   double radius = 5;
   ...
}
```

Declares a local variable **radius**; the value of field **radius** remains 0.0

Writing a program vs. a class

JC08-4.3

(A. Nguyen)

Writing a program

- When you are asked to write a (console) program, you create a “class” with just the main program, then write the body of the main program
- Example: `HelloWorld`

Writing a class

- When you are asked to write a class, you create a class with 3 main parts: fields, constructors, methods
- Example: `BankAccount`
- You will need to write a client program (like a Tester, Viewer, etc.) with the main program to use that class

String

JC08-4.5

(A. Nguyen)

String operations

- Concatenation operator is +
- Automatic conversion is done for a non-string if concatenated with a string (including an empty string “”); e.g., “” + 17 → “17”, as used in `System.out.println( … );`
- Compound assignment operator is permissible: +=
- Concatenation (if 1 or 2 strings) & addition (if both numeric) are done from left to right:
  - “” + 2 + 3 → “23”
  - 2 + 3 + “” → “5”
- Any object has a `toString` method, which can be overridden by any class

Some popular methods of `String`

Given: `String aStr = "The quick brown fox jumps over the lazy dog."`

METHOD SIGNATURE:	EXAMPLE:
`boolean contains(CharSequence s)` returns **`true`** if the String contains **`s`**	`if (aStr.contains("fox")) ...`
`boolean equals(Object anObject)` returns **`true`** if **`(String)anObject`** has the same contents as self	`if (aStr.equals(otherSent)) ...`
`int length()` returns the length of the **`String`**	`for (int i = 0; i < aStr.length(); i++) ...`
`String substring(int beginIndex, int endIndex)` returns a **`String`** with the contents from location `beginIndex` to location `endIndex -1`, inclusively	`aStr.substring(1, 2)` returns `h` **NOTE: this does NOT return `he`** NOTE: `aStr.substring(4, 4)` returns an empty String.
`String toLowerCase()` **`String toUpperCase()`** converts to lower or upper case, and return a new/different **`String`**	`aStr.toLowerCase()` returns `the quick brown fox jumps over the lazy dog.` **NOTE: this does NOT change `aStr`**

Some popular methods of `String` (cont.)

There are also these popular methods – check the Java API for details:

- `charAt`: gives the character at the given index/location
- `lastIndexOf`: gives the last index/location of the given character/string – NOTE: there is no `firstIndexOf`, only `indexOf`
- `trim`: eliminate white spaces at the beginning & ending (but not inside) of a string
- `replace`: replace all occurrences of a char/string with another
- `replaceFirst`: replace the first occurrence of a string (not `char`) with another

The toString method

- The `toString` method is in the `Object` class, which is the highest superclass of every Java object
- The `toString` method returns the ***text*** that ***best represents*** the object; for example, a `Student` class may have `toString` return the student name and/or ID
- The signature of the method is `String toString()`
- The `println` or `print` method calls the `toString` of the class
- If a class does NOT override the `toString` method, then `println` or `print` will display the hex value of the object; otherwise, the text representing the object will be displayed

The toString method: example

- Given a class called `Student` (with field `theID`, etc.), and an object called `aStudent` of type `Student` created in a client
- If `Student` does NOT override the `toString` method, then the statement in the client:

  ```
  System.out.print(aStudent);
  ```

  will display the hex value of object `aStudent`
- If `Student` overrides the `toString` method, then

  ```
  System.out.print(aStudent);
  ```

  will display `theID` of object `aStudent`

```
public String toString()
{
    return "" + theID;
}
```

THE END