Numerical Data

CS 18000

Sunil Prabhakar

Department of Computer Science

Purdue University

Problem

Write a program to compute the area and circumference of a circle given its radius.

- Requires that we perform operations on numbers
- Strings or other standard classes are not appropriate for this purpose.
- Instead, we will use a special type of data

Why not Strings?

- We could use strings, but
  - Strings are just encodings of characters
  - with 2 bytes of storage
    - a numeric string can only represent 0,1,…, or 9
    - but, there are potentially 2^{16} = 65,536 combinations (numbers)
  - the String class has no methods for numeric operations
  - better to use a different *type* of representation
    - we same 2 bytes of data can be used to represent two different types of data.

Primitive Data Types

- As all matter is fundamentally composed of atoms, all objects are fundamentally composed of primitive data types.
- Primitive types are the building blocks of all data used in Java.
- Primitive data types are neither classes nor objects.
  - they are the simplest representations of data
- Each type can be processed using only specific operators

Primitive Data Types

- Numeric
  - e.g., 2, 3, 3.1416, -334234.2343242
  - for storing and operating on integer and real valued data
- Character
  - e.g., 'a', 'अ', '℞', '༓', '༊', '𝄢', '⨌', '丙', '齒', 'ש' ,' ض'
  - for representing characters for (almost) all languages
- Boolean
  - logic data type
  - only two allowed values: **true**, **false**
- This week we will study Numeric data.

Area and Perimeter

```
import java.util.Scanner;

public class CircleCalculator {

  public static void main (String[] args){
    double radius, area, circumference;

    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter radius");
    radius = scanner.nextDouble();

    circumference = 2.0 * 3.14 * radius;
    area = 3.14 * radius * radius;

    System.out.println("Given Radius:  " + radius + "\n" +
                       "Area:          " + area   + "\n" +
                       "Circumference: " + circumference);
  }
}
```

Area and Perimeter

```
import java.util.Scanner;

public class CircleCalculator {

  public static void main (String[] args){
    double radius, area, circumference;

    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter radius");
    radius = scanner.nextDouble();

    circumference = 2.0 * 3.14 * radius;
    area = 3.14 * radius * radius;

    System.out.println("Given Radius:  " + radius + "\n" +
                       "Area:          " + area   + "\n" +
                       "Circumference: " + circumference);
  }
}
```

Not a class

Important Points

- Note the use of =
  - do not confuse this with the = symbol from mathematics
  - `circumference = 2 * 3.14 * radius;`
    - computes the product of 2, 3.14, and the numeric value stored in `radius`,
    - and copies this value into `circumference`
  - This is an assignment statement. Causes the value stored in `circumference` to change.

Variables

Variables

- Data items such as `area` are called *variables*.
  - since we can change their values during program execution.

Variables

- Data items such as `area` are called *variables*.
  - since we can change their values during program execution.
- A variable has three properties:
  - A memory location to store the value,
  - The type of data stored in the memory location, and
  - The name used to refer to the memory location.

Variables

- Data items such as `area` are called *variables*.
  - since we can change their values during program execution.
- A variable has three properties:
  - A memory location to store the value,
  - The type of data stored in the memory location, and
  - The name used to refer to the memory location.
- When the declaration **`double`** `area;` is made,
  - memory space is allocated to store a real number value
  - `area` is a reference for this space.

Assignment Statements

- We set the value of a variable using an *assignment statement*.
  - Do not confuse with equality in Algebra!

```
double a, b, c;

a = 3.0;
b = 2.0 * 2.3;
c = a * b;
```

- Compute the value of the right (of =) and copy the result into the variable on the left.

```
a = 2 * a;
```

- Use the current value of `a` to compute result and copy the result back into `a`.
- Can also initialize when declaring

```
double a = 5.9, b = 34;
```

Arithmetic Operators

```
double x, y, z;
x = 5.0;
y = 2.5;
```

Multiplication	`z = x * y;`
Addition	`z = x + y;`
Subtraction	`z = x - y;`
Division	`z = x / y;`
Unary negation	`z = -y;`

Arithmetic Operators

```
double x, y, z;
x = 5.0;
y = 2.5;
```

Multiplication	`z = x * y;`	→ `z = 12.5`
Addition	`z = x + y;`	
Subtraction	`z = x - y;`	
Division	`z = x / y;`	
Unary negation	`z = -y;`	

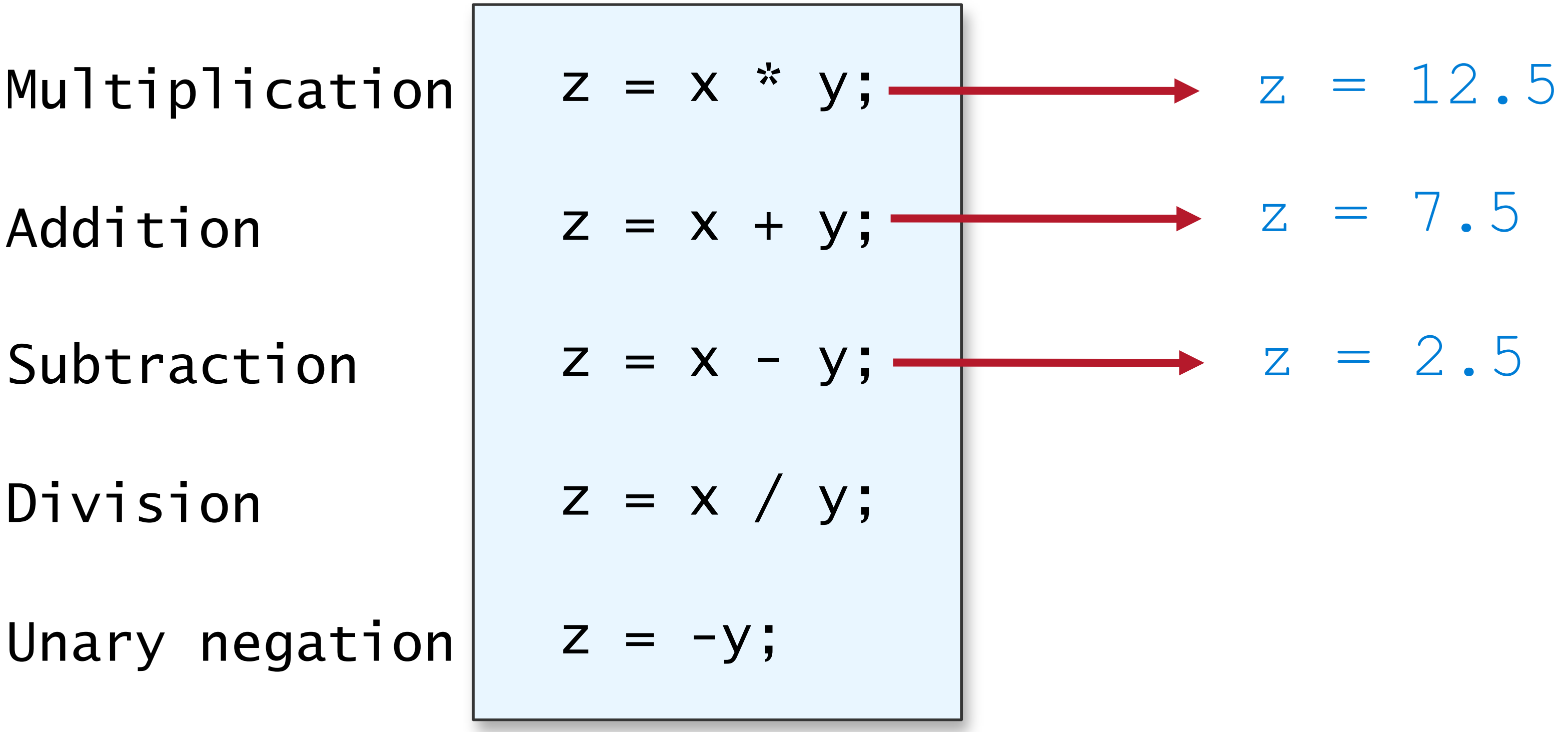
Arithmetic Operators

```
double x, y, z;
x = 5.0;
y = 2.5;
```

Multiplication	`z = x * y;`	`z = 12.5`
Addition	`z = x + y;`	`z = 7.5`
Subtraction	`z = x - y;`	
Division	`z = x / y;`	
Unary negation	`z = -y;`	

Arithmetic Operators

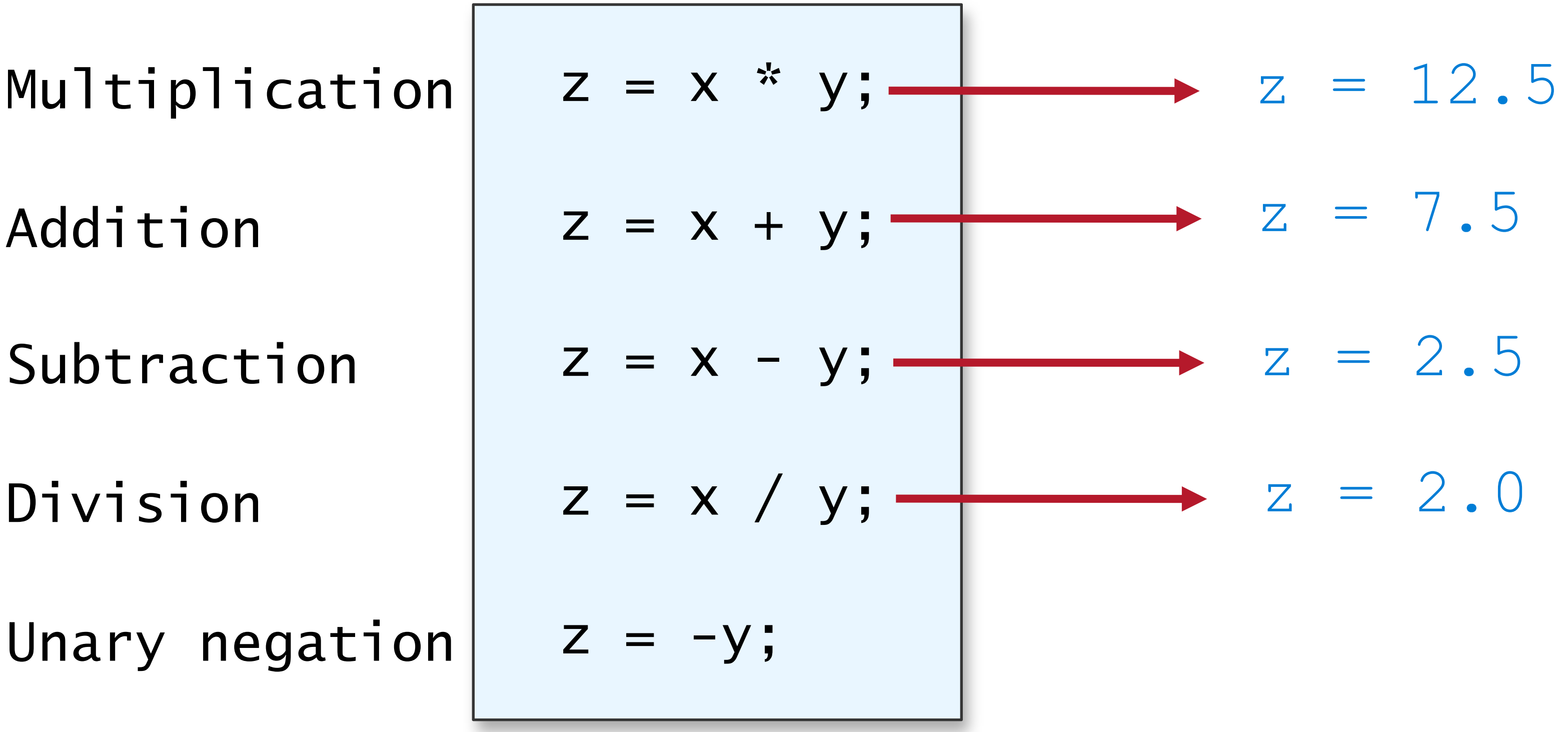
```
double x, y, z;
x = 5.0;
y = 2.5;
```



```
Multiplication      z = x * y;  →  z = 12.5

Addition            z = x + y;  →  z = 7.5

Subtraction         z = x - y;  →  z = 2.5

Division            z = x / y;

Unary negation      z = -y;
```

Arithmetic Operators

```
double x, y, z;
x = 5.0;
y = 2.5;
```



Multiplication	`z = x * y;`	z = 12.5
Addition	`z = x + y;`	z = 7.5
Subtraction	`z = x - y;`	z = 2.5
Division	`z = x / y;`	z = 2.0
Unary negation	`z = -y;`	

Arithmetic Operators

```
double x, y, z;
x = 5.0;
y = 2.5;
```

Multiplication	`z = x * y;`	z = 12.5
Addition	`z = x + y;`	z = 7.5
Subtraction	`z = x - y;`	z = 2.5
Division	`z = x / y;`	z = 2.0
Unary negation	`z = -y;`	z = -2.5

Examples of expressions

```
double tempC, tempF;

tempF = tempC * 9.0/5.0 + 32.0;
```

```
double x, y, z;

z = x * x + y * y / x;

z = x*x+y*y/x;

z = x *    x    + y
*   y        /
x;
```

Whitespaces make no difference.

All these expressions are identical to the compiler.

PURDUE UNIVERSITY

Arithmetic Expressions

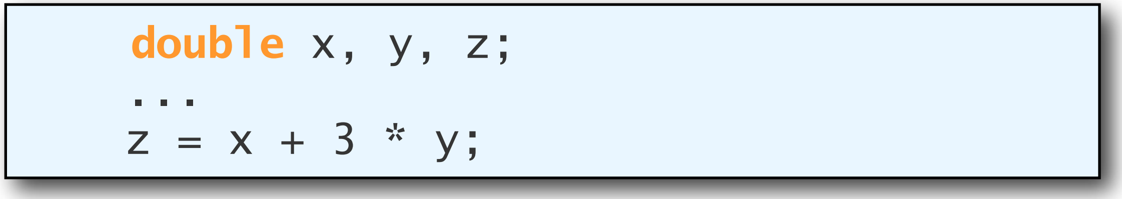
Arithmetic Expressions

- How is the following expression evaluated?

```
double x, y, z;
...
z = x + 3 * y;
```

Arithmetic Expressions

- How is the following expression evaluated?



```
double x, y, z;
...
z = x + 3 * y;
```

Arithmetic Expressions

- How is the following expression evaluated?

```
double x, y, z;
...
z = x + 3 * y;
```

Arithmetic Expressions

- How is the following expression evaluated?

```
double x, y, z;
...
z = x + 3 * y;
```

- Answer: x is added to 3*y .

Arithmetic Expressions

- How is the following expression evaluated?

```
    double x, y, z;
    ...
   z = x + 3 * y;
```

- Answer: x is added to 3*y .
- We determine the order of evaluation by following *precedence rules*.

Arithmetic Expressions

- How is the following expression evaluated?

```
    double x, y, z;
   ...
   z = x + 3 * y;
```

- Answer: x is added to 3*y .
- We determine the order of evaluation by following *precedence rules*.
- Evaluation is in order of precedence.
  - Recall PEMDAS

Arithmetic Expressions

- How is the following expression evaluated?

```
double x, y, z;
...
z = x + 3 * y;
```

- Answer: `x` is added to `3*y` .
- We determine the order of evaluation by following *precedence rules*.
- Evaluation is in order of precedence.
  - Recall PEMDAS
- Operators at same level are evaluated *left to right* for most operators

PURDUE
UNIVERSITY

Precedence Rules

Priority	Group	Operator	Rule
High ↓ Low	Subexpression	( )	Starting with innermost ()
	Unary operators	-, +	Left to right.
	Multiplicative operators	*, /, %	Left to right.
	Additive operators	+, -	Left to right.

PURDUE UNIVERSITY

Precedence Examples

Precedence Examples

```
x + 4*y - x/z + 2/x
```

Precedence Examples

$$x\ +\ \underbrace{4*y}_{1}\ -\ x/z\ +\ 2/x$$

Precedence Examples

$$x \; + \; \underset{1}{\underline{4*y}} \; - \; \underset{2}{\underline{x/z}} \; + \; 2/x$$

Precedence Examples

$$x \;+\; \underbrace{4*y}_{1} \;-\; \underbrace{x/z}_{2} \;+\; \underbrace{2/x}_{3}$$

Precedence Examples

$$x \overset{4}{\overline{+}} \underset{1}{\underline{4*y}} - \underset{2}{\underline{x/z}} + \underset{3}{\underline{2/x}}$$

Precedence Examples

$$x \overset{4}{\overline{\;+\;}} \underbrace{4*y}_{1} \overset{5}{\overline{\;-\;}} \underbrace{x/z}_{2} + \underbrace{2/x}_{3}$$

Precedence Examples

$$x \overset{4}{\overline{+}} \underset{1}{\underline{4*y}} \overset{5}{\overline{-}} \underset{2}{\underline{x/z}} \overset{6}{\overline{+}} \underset{3}{\underline{2/x}}$$

Precedence Examples

$$x \overset{4}{\overline{\;+\;}} \underset{1}{\underline{4*y}} \overset{5}{\overline{\;-\;}} \underset{2}{\underline{x/z}} \overset{6}{\overline{\;+\;}} \underset{3}{\underline{2/x}}$$

same as: `x + (4*y) - (x/z) + (2/x)`

Precedence Examples

$$x \overset{4}{\overline{\;+\;}} \underset{1}{\underline{4*y}} \overset{5}{\overline{\;-\;}} \underset{2}{\underline{x/z}} \overset{6}{\overline{\;+\;}} \underset{3}{\underline{2/x}}$$

same as: `x + (4*y) - (x/z) + (2/x)`

`(x + y * (4 - x) / z + 2 / -x )`

PURDUE UNIVERSITY

Precedence Examples

$$x \overset{4}{\overline{\;+\;}} \underset{1}{\underline{4*y}} \overset{5}{\overline{\;-\;}} \underset{2}{\underline{x/z}} \overset{6}{\overline{\;+\;}} \underset{3}{\underline{2/x}}$$

same as: `x + (4*y) - (x/z) + (2/x)`

$$(x + y * (4 \underset{1}{\underline{\;-\;}} x) / z + 2 / -x)$$

Precedence Examples

$$x \overset{4}{\overline{\;+\;}} \underset{1}{\underline{4*y}} \overset{5}{\overline{\;-\;}} \underset{2}{\underline{x/z}} \overset{6}{\overline{\;+\;}} \underset{3}{\underline{2/x}}$$

same as: $x + (4*y) - (x/z) + (2/x)$

$$(x + y * (4 \underset{1}{\underline{\;-\;}} x) / z + 2 / \underset{2}{\underline{-x}})$$

Precedence Examples

$$x \overset{4}{\overline{\;+\;}} \underset{1}{\underline{4*y}} \overset{5}{\overline{\;-\;}} \underset{2}{\underline{x/z}} \overset{6}{\overline{\;+\;}} \underset{3}{\underline{2/x}}$$

same as: `x + (4*y) - (x/z) + (2/x)`

$$(x + y \underset{3}{\underline{\;*\;}} (4 \underset{1}{\underline{\;-\;}} x) / z + 2 / \underset{2}{\underline{-x}})$$

Precedence Examples

$$x \overbrace{+}^{4} \underbrace{4*y}_{1} \overbrace{-}^{5} \underbrace{x/z}_{2} \overbrace{+}^{6} \underbrace{2/x}_{3}$$

same as: `x + (4*y) - (x/z) + (2/x)`

$$(x + y \underbrace{*}_{3} (4 \underbrace{-}_{1} x) \overbrace{/}^{4} z + 2 / \underbrace{-x}_{2})$$

PURDUE UNIVERSITY

Precedence Examples

$$x \overset{4}{\overline{+}} \underset{1}{\underline{4*y}} \overset{5}{\overline{-}} \underset{2}{\underline{x/z}} \overset{6}{\overline{+}} \underset{3}{\underline{2/x}}$$

same as: `x + (4*y) - (x/z) + (2/x)`

$$(x + y \underset{3}{\underline{\;*\;}} (4 \underset{1}{\underline{\;-\;}} x) \overset{4}{\overline{\;/\;}} z + 2 \overset{5}{\overline{\;/\;}} \underset{2}{\underline{-x}})$$

Precedence Examples

$$x \overset{4}{\overline{+}} \underset{1}{\underline{4*y}} \overset{5}{\overline{-}} \underset{2}{\underline{x/z}} \overset{6}{\overline{+}} \underset{3}{\underline{2/x}}$$

same as: `x + (4*y) - (x/z) + (2/x)`

$$(x \overset{6}{\overline{+}} y \underset{3}{\underline{*}} (4 \underset{1}{\underline{-}} x) \overset{4}{\overline{/}} z + 2 \overset{5}{\overline{/}} \underset{2}{\underline{-x}})$$

Precedence Examples

$$x \overset{4}{\overline{+}} \underset{1}{\underline{4*y}} \overset{5}{\overline{-}} \underset{2}{\underline{x/z}} \overset{6}{\overline{+}} \underset{3}{\underline{2/x}}$$

same as: `x + (4*y) - (x/z) + (2/x)`

$$(x \overset{6}{\overline{+}} y \underset{3}{\underline{\ *\ }} (4 \underset{1}{\underline{\ -\ }} x) \overset{4}{\overline{/}} z \overset{7}{\overline{+}} 2 \overset{5}{\overline{/}} \underset{2}{\underline{-x}})$$

Precedence Examples

$$x \overset{4}{\overline{+}} \underset{1}{\underline{4*y}} \overset{5}{\overline{-}} \underset{2}{\underline{x/z}} \overset{6}{\overline{+}} \underset{3}{\underline{2/x}}$$

same as: `x + (4*y) - (x/z) + (2/x)`

$$(x \overset{6}{\overline{+}} y \underset{3}{\underline{*}} (4 \underset{1}{\underline{-}} x) \overset{4}{\overline{/}} z \overset{7}{\overline{+}} 2 \overset{5}{\overline{/}} \underset{2}{\underline{-x}})$$

same as:

`(x + ((y * (4-x)) / z) + (2 / (-x)))`

PURDUE
UNIVERSITY

Precedence Examples

$$x \overset{4}{\overline{\;+\;}} \underset{1}{\underline{4*y}} \overset{5}{\overline{\;-\;}} \underset{2}{\underline{x/z}} \overset{6}{\overline{\;+\;}} \underset{3}{\underline{2/x}}$$

same as: `x + (4*y) - (x/z) + (2/x)`

$$(x \overset{6}{\overline{\;+\;}} y \underset{3}{\underline{\;*\;}} (4 \underset{1}{\underline{\;-\;}} x) \overset{4}{\overline{\;/\;}} z \overset{7}{\overline{\;+\;}} 2 \overset{5}{\overline{\;/\;}} \underset{2}{\underline{-x}}\;)$$

same as:

`(x + ((y * (4-x)) / z) + (2 / (-x)))`

To be sure, use parentheses!

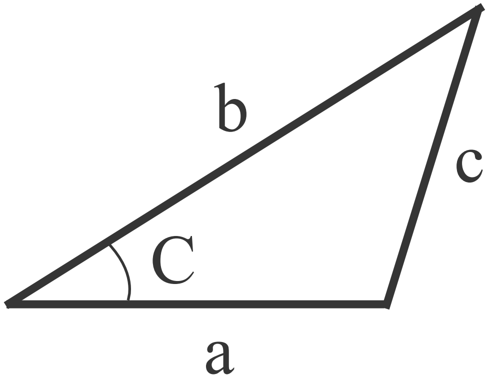
Problem

- *Write a program that when given the lengths of two sides of a triangle, and the angle between these sides, computes the length of the third side.*

- Recall:

$$c^2 = a^2 + b^2 - 2ab \cos C$$

$$c = \sqrt{a^2 + b^2 - 2ab \cos C}$$



Solution

- We know how to get the three inputs.
- But, how do we compute square roots and cosines?
  - Many common functions are available as methods of the Math class defined in the java.lang package.
  - Trigonometric methods require angles to be expressed in Radians (not degrees).
- Most methods take **double** arguments and their return type is **double**

PURDUE UNIVERSITY

Sample Math Class Methods

Method name	Description	Input type	Output type
pow(x, y)	Return x^y	double	double
log(x)	Return natural log of x.	double	double
sqrt(x)	Return the square root of x	double	double
sin(a)	Return sine of angle a (radians)	double	double
asin(a)	Return the arc sine of a (in radians)	double	double
toRadians(d)	Convert d from degrees to radians.	double	double
exp(x)	Return e^x	double	double
max(x, y)	Return larger of x or y.	*	*

See API for details

Step 1: Input and Test

```java
public class ThirdSideStep1 {

  public static void main (String[] args){
    double a, b, c, angleCDegrees;

    Scanner scanner = new Scanner(System.in);

    System.out.println("Enter the length of one side:");
    a = scanner.nextDouble();

    System.out.println("Enter the length of the other side:");
    b = scanner.nextDouble();

    System.out.println("Enter the angle between these two sides
   (in degrees)");
    angleCDegrees = scanner.nextDouble();

    System.out.println("a: " + a + ", b: " + b + ", angle: " +
   angleCDegrees);
  }
}
```

Step 2: Convert to Radians

```
. . .

angleCRadians = Math.toRadians(angleCDegrees);


System.out.println("The angle " + angleCDegrees +
                   " degrees equals: " + angleCRadians +
                   " radians");
```

- The Math class expects arguments in Radians, not degrees
- Use the toRadians method of the Math class to convert, and check.

Step 3: Compute Side and Output

```
. . .

c = Math.sqrt(
      Math.pow(a,2) + Math.pow(b,2) - 2*a*b* Math.cos(angleCRadians)
     );
System.out.println("The length of the third side is: " + c);
```

- Recall: $c = \sqrt{a^2 + b^2 - 2ab\ cosC}$
- Note how the method calls are used within the expression to compute parts of the expression.

Numeric Data Types

- The type **double** that we saw allows us to store a very wide range of real number values:
  - -1.7977 X 10^{308} to +1.797 x 10^{308}
  - 8 bytes are used to store each double variable
  - How? (please wait till this week's Recitation)
- Sometimes, we don't need such a large range.
  - can use the type **float** instead
  - only 4 bytes, but smaller range
  - -3.40282347 x 10^{38} to + 3.40282347 x 10^{38}

Area and Perimeter

```java
import java.util.Scanner;

public class CircleCalculator {

  public static void main (String[] args){
    float radius, area, circumference;

    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter radius");
    radius = scanner.nextFloat();

    circumference = 2.0 * 3.14 * radius;
    area = 3.14 * radius * radius;

    System.out.println("Given Radius:  " + radius + "\n" +
                       "Area:          " + area   + "\n" +
                       "Circumference: " + circumference);
  }
}
```

Note type

Note method name

CAUTION: Imprecision

- It is not possible to exactly represent every possible real valued number in a **double** or **float**
  - Fixed number of bits
    - **float**: 4 bytes -- 32 bits:2^{32} (~1 billion) combinations
    - **double**: 8 bytes -- 64 bits: 2^{64} (~1 million trillion) combinates
  - BUT, how many real numbers
    - between, say 1.0 and 2.0? INFINITE!
- **float**s and **double**s sometimes only store an approximation of the actual number!!!!
- Do not rely on exact values!
- Examples in Recitation

Integer data

- If we are dealing with integer values only, using float or double is unwise:
  - operations are slower
  - maybe using too much space (memory)
  - sometimes there is a (small) error in representation (imprecision)
- Instead, we have completely separate numeric types for integer data
  - **byte, short, int, long**
  - differ in size and range

Numeric Data Types

Type	Content	Size (bytes)	Minimum Value	Maximum Value
byte	Integer	1	-128	127
short		2	-32768	32767
int		4	-2147483648	2147483647
long		8	-9, 223, 372, 036, 854, 780, 000	9, 223, 372, 036, 854, 780, 000
float	Real	4	$-3.40282347 \times 10^{38}$	$3.40282347 \times 10^{38}$
double		8	-1.7977×10^{308}	1.7977×10^{308}

Operators for Integer types

```
int x, y, z;
x = 5;
y = 2;
```

Multiplication	`z = x * y;`
Addition	`z = x + y;`
Subtraction	`z = x - y;`
Division	`z = x / y;`
Modulo	`z = x % y;`
Unary negation	`z = -y;`

Operators for Integer types

```
int x, y, z;
x = 5;
y = 2;
```

Multiplication	`z = x * y;`
Addition	`z = x + y;`
Subtraction	`z = x - y;`
Division	`z = x / y;`
Modulo	`z = x % y;`
Unary negation	`z = -y;`

`z = x * y;` → `z = 10`

Operators for Integer types

```
int x, y, z;
x = 5;
y = 2;
```

Multiplication	`z = x * y;`	`z = 10`
Addition	`z = x + y;`	`z = 7`
Subtraction	`z = x - y;`	
Division	`z = x / y;`	
Modulo	`z = x % y;`	
Unary negation	`z = -y;`	

Operators for Integer types

```
int x, y, z;
x = 5;
y = 2;
```

Multiplication	`z = x * y;`	`z = 10`
Addition	`z = x + y;`	`z = 7`
Subtraction	`z = x - y;`	`z = 3`
Division	`z = x / y;`	
Modulo	`z = x % y;`	
Unary negation	`z = -y;`	

Operators for Integer types

```
int x, y, z;
x = 5;
y = 2;
```

Multiplication	`z = x * y;`	z = 10
Addition	`z = x + y;`	z = 7
Subtraction	`z = x - y;`	z = 3
Division	`z = x / y;`	z = 2
Modulo	`z = x % y;`	
Unary negation	`z = -y;`	

Operators for Integer types

```
int x, y, z;
x = 5;
y = 2;
```

Multiplication	`z = x * y;`	`z = 10`
Addition	`z = x + y;`	`z = 7`
Subtraction	`z = x - y;`	`z = 3`
Division	`z = x / y;`	`z = 2`
Modulo	`z = x % y;`	
Unary negation	`z = -y;`	

Truncation!

Operators for Integer types

```
int x, y, z;
x = 5;
y = 2;
```

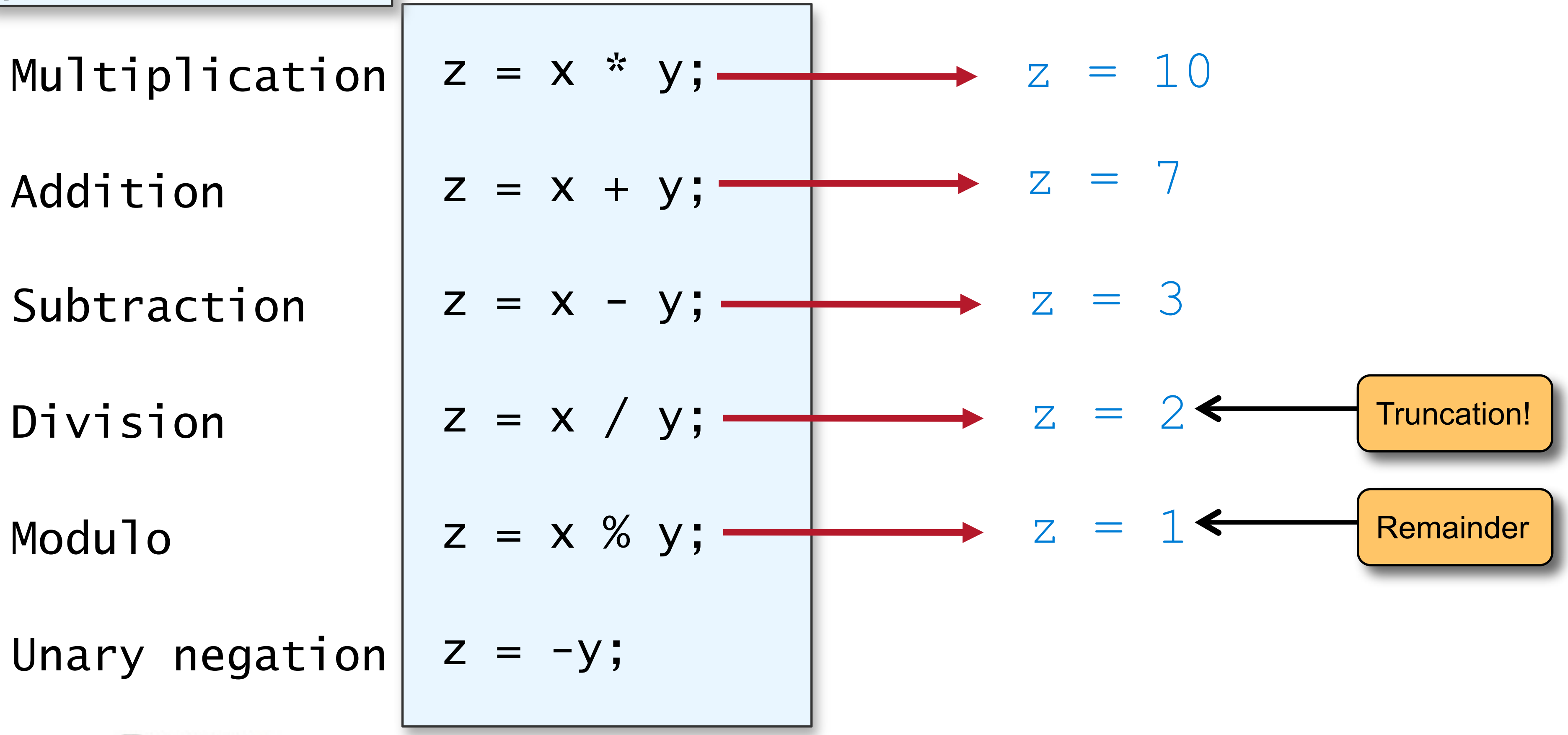
Multiplication	`z = x * y;`	z = 10
Addition	`z = x + y;`	z = 7
Subtraction	`z = x - y;`	z = 3
Division	`z = x / y;`	z = 2
Modulo	`z = x % y;`	z = 1
Unary negation	`z = -y;`	

Truncation!

© Sunil Prabhakar, Purdue University

Operators for Integer types

```
int x, y, z;
x = 5;
y = 2;
```



```
Multiplication   z = x * y;   →   z = 10
Addition         z = x + y;   →   z = 7
Subtraction      z = x - y;   →   z = 3
Division         z = x / y;   →   z = 2  ← Truncation!
Modulo           z = x % y;   →   z = 1  ← Remainder
Unary negation   z = -y;
```

Operators for Integer types

```
int x, y, z;
x = 5;
y = 2;
```

Multiplication	`z = x * y;`	`z = 10`	
Addition	`z = x + y;`	`z = 7`	
Subtraction	`z = x - y;`	`z = 3`	
Division	`z = x / y;`	`z = 2`	Truncation!
Modulo	`z = x % y;`	`z = 1`	Remainder
Unary negation	`z = -y;`	`z = -2`	

Division Operator

- It is important to note the behavior of division when the operands are

  both Integer types (`byte`, `short`, `int`, `long`)

  - in this case we get integer division (truncation of the decimal part)

  or, at least one is of type `float` or `double`

  - in this case we get regular division (no truncation).
  - there may be errors due to inherent problem with float and double representations.
- Division by 0 causes an error.

Integer vs. Real Division

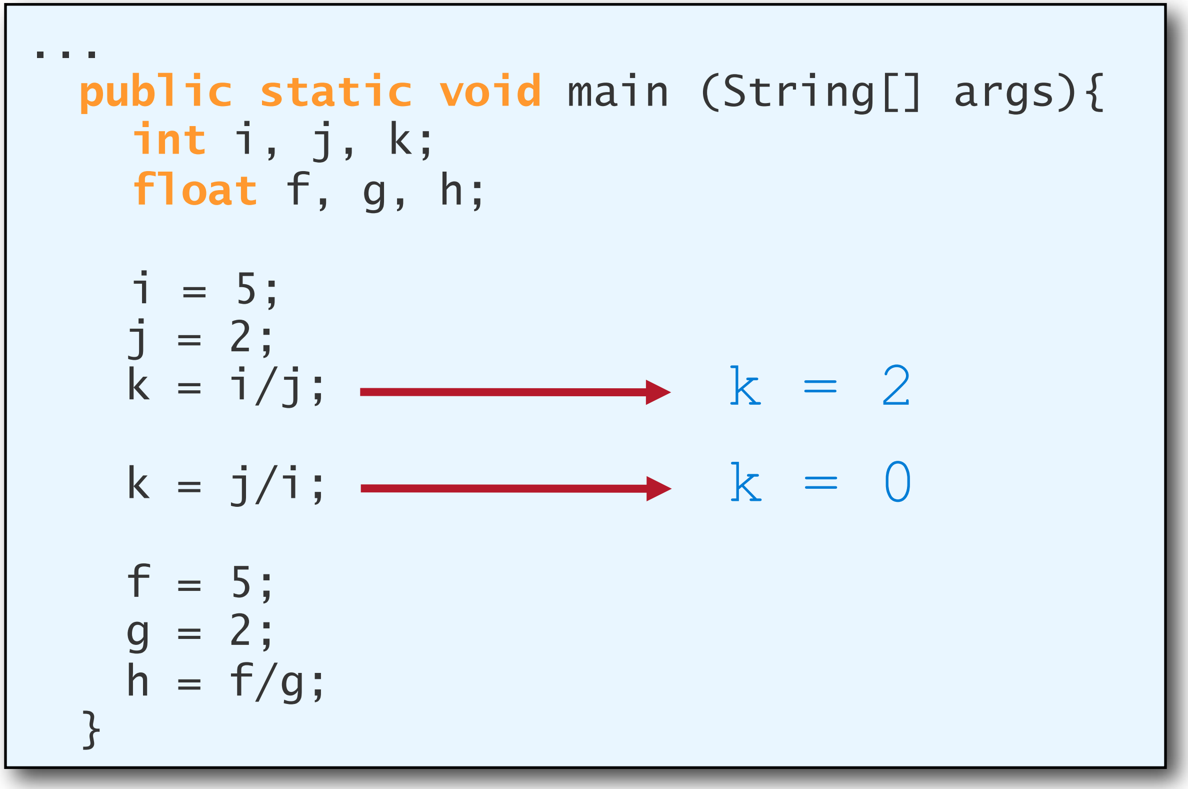
```
...
  public static void main (String[] args){
    int i, j, k;
    float f, g, h;

    i = 5;
    j = 2;
    k = i/j;

    k = j/i;

    f = 5;
    g = 2;
    h = f/g;
  }
```

Integer vs. Real Division

```
...
  public static void main (String[] args){
    int i, j, k;
    float f, g, h;

    i = 5;
    j = 2;
    k = i/j;

    k = j/i;

    f = 5;
    g = 2;
    h = f/g;
  }
```

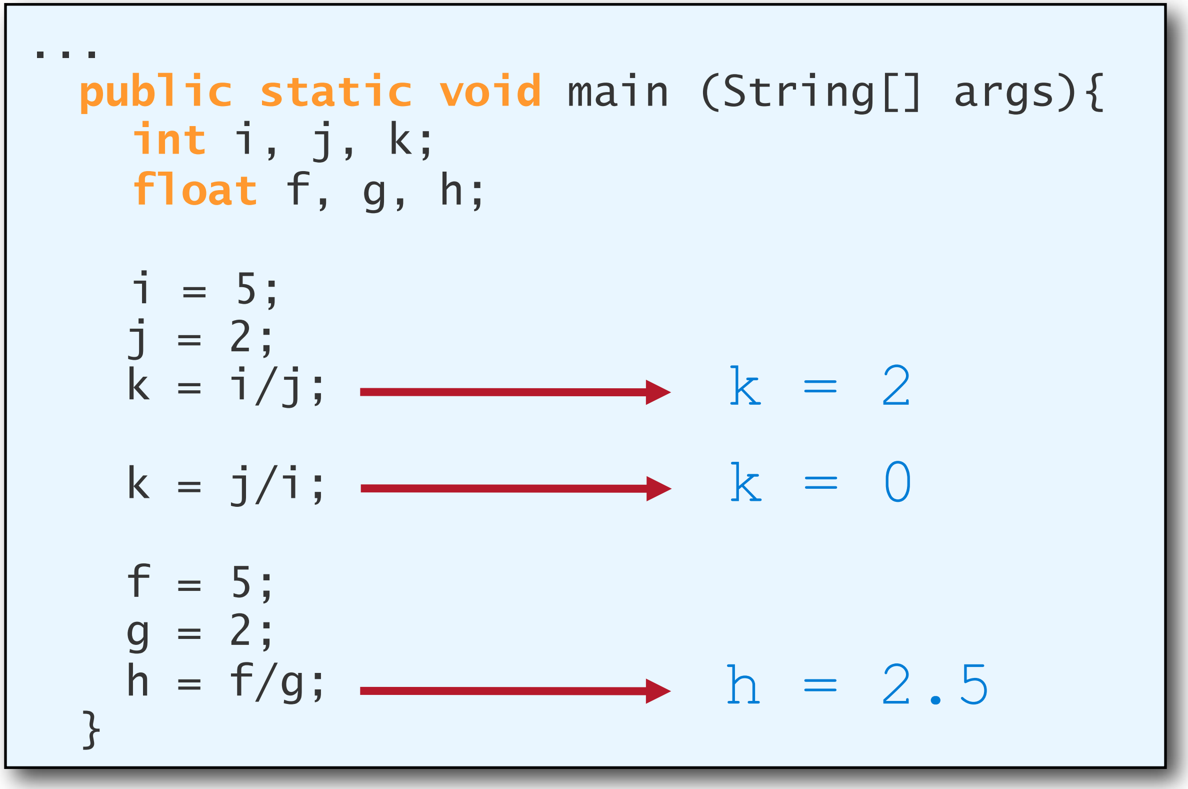
k = 2

Integer vs. Real Division



```
...
  public static void main (String[] args){
    int i, j, k;
    float f, g, h;

    i = 5;
    j = 2;
    k = i/j;     →     k = 2

    k = j/i;     →     k = 0

    f = 5;
    g = 2;
    h = f/g;
  }
```

Integer vs. Real Division



```
...
  public static void main (String[] args){
    int i, j, k;
    float f, g, h;

    i = 5;
    j = 2;
    k = i/j;   →   k = 2

    k = j/i;   →   k = 0

    f = 5;
    g = 2;
    h = f/g;   →   h = 2.5
  }
```

Modulo Operator

- This is simply a remainder operator
  - x % y computes the remainder when x is divided by y.
  - normally only used when both x and y are integer types (**byte**, **short**, **int**, or **long**)
  - can be used with **float** and **double**, but results are not really meaningful

PURDUE
UNIVERSITY

Type Safety

- Why so many different types for numeric data?
  - Integer types are more efficient and 100% accurate, BUT don't handle fractional values.
  - All types have a range
    - larger range implies more memory used
- Can we mix different types in expressions and assignments?
  - Yes, but have to be careful.

Numeric Type Precision

- The numeric types can be arranged in order of their ranges as follows:

`byte` < **`short`** < **`int`** < **`long`** < **`float`** < **`double`**

- The range of each type is strictly more precise than the range of each type to its left
  - E.g., any **`byte`** value can be stored in a **`long`** variable
  - Thus, there is no loss in assigning a smaller typed value to a larger typed variable
  - Going the other way causes losses!

Examples

```
byte b;
short s;
int i;
long l;
float f;
double d;
...

d = f;
d = l;
d = i;
d = s;
d = b;
f = l;
f = i;
f = s;
f = b;
l = i;
l = s;
l = b;
i = s;
i = b;
s = b;
```

Examples

```
byte b;
short s;
int i;
long l;
float f;
double d;
...

d = f;
d = l;
d = i;
d = s;
d = b;
f = l;
f = i;
f = s;
f = b;
l = i;
l = s;
l = b;
i = s;
i = b;
s = b;
```

Each of these assignments is legal -- no data loss.

Examples

```
byte b;
short s;
int i;
long l;
float f;
double d;
...

d = f;
d = l;
d = i;
d = s;
d = b;
f = l;
f = i;
f = s;
f = b;
l = i;
l = s;
l = b;
i = s;
i = b;
s = b;
```

Each of these assignments is legal -- no data loss.

```
byte b;
short s;
int i;
long l;
float f;
double d;
...

f = d;
l = d;
i = d;
s = d;
b = d;
l = f;
i = f;
s = f;
b = f;
i = l;
s = l;
b = l;
s = i;
b = i;
b = s;
```

Examples

```
byte b;
short s;
int i;
long l;
float f;
double d;
...

d = f;
d = l;
d = i;
d = s;
d = b;
f = l;
f = i;
f = s;
f = b;
l = i;
l = s;
l = b;
i = s;
i = b;
s = b;
```

Each of these assignments is legal -- no data loss.

```
byte b;
short s;
int i;
long l;
float f;
double d;
...

f = d;
l = d;
i = d;
s = d;
b = d;
l = f;
i = f;
s = f;
b = f;
i = l;
s = l;
b = l;
s = i;
b = i;
b = s;
```

ERROR!! Each of these assignments is illegal -- could result in data loss.

Type Casting

- It is possible to explicitly change types (type casting)

```
d = (double) i;
i = (int) d;
```

  - Necessary when assigning a more precise type to a less precise one (Demotion).
    - possible data loss
    - assigning a **float** or **double** to an integer type results in truncation (not rounding)

```
i = (int) 3.5;
```

`i` will store 3, not 3.5

  - Automatically done when assigning a less precise type to a more precise type (promotion). No data loss

Expression Types

- Each numeric expression also has a data type. What is the type of i + j ?
- Depends on the types of i and j.
  - If they are both of the same type, then the expression of the same type too
  - Otherwise the operand with the lower type will be automatically promoted to the higher type; the overall expression will be of this higher type too.

Expression types

```
byte b;
short s;
int i;
long l;
float f;
double d;
…
l = b + i;

l = (long) (f * d);

s = (short) f / b;

d = ((s/b) + (i*l))/f;
```

Literal Numeric Values

- What is the type of a literal value such as 3 or 3.45?
- If there is no decimal point, then the type is **int**
  - To make it a **long** type append L or l
  - For **byte** and **short** -- no special type. If the value is an integer within the range of **byte** (**short**), it can be assigned to a **byte** (**short**)
- If it has a decimal point, then its type is **double**.
  - To make it a **float** append F or f

```
byte b = 23;
short s = 145;
int i = -2345;
long l = 234L;
float f = -3.4556F;
double d = 3.4564;
```

Primitive vs. Class assignment

- For assignment, the behavior of primitive variables seems to be different from that of class (reference) variables.

```
double    i,j;
i    =  5.0;
i    =  85.0;


i = j;
```

```
Customer   cust1, cust2;
cust1    =  new  Customer( );
cust2    =  new  Customer( );


cust1 = cust2;
```

Primitive Data: Declaration & Assignment

```
double    i,j;
i    =   5.0;
j    =   8.0;
```

Primitive Data: Declaration & Assignment

```
double   i,j;
i    =   5.0;
j    =   8.0;
```

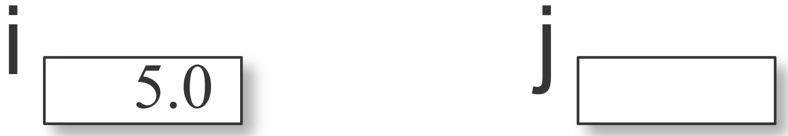
Primitive Data: Declaration & Assignment

```
double    i,j;
i    =  5.0;
j    =  8.0;
```

Memory is allocated.

Primitive Data: Declaration & Assignment

```
double    i,j;
i    =   5.0;
j    =   8.0;
```



Memory is allocated.

Values are stored in those locations.

Primitive Data: Declaration & Assignment

```
double   i,j;
i    =   5.0;
j    =   8.0;
```

i 5.0

j 8.0

Memory is allocated.

Values are stored in those locations.

Primitive Data Assignment

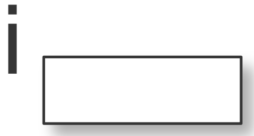
```
double    i;
i    =   5.0;
i    =   85.0;
```

Primitive Data Assignment

```
double    i;
i    =  5.0;
i    =  85.0;
```

Primitive Data Assignment

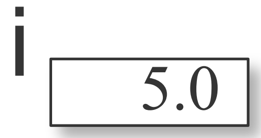
```
double    i;
i    =   5.0;
i    =   85.0;
```



Memory is allocated.

Primitive Data Assignment

```
double    i;
i    =  5.0;
i    =  85.0;
```

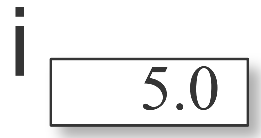


Memory is allocated.

The value 5.0 is stored in i.

Primitive Data Assignment

```
double   i;
i    =   5.0;
i    =   85.0;
```

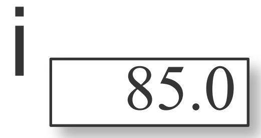


Memory is allocated.

The value 5.0 is stored in i.

Primitive Data Assignment

```
double   i;
i    =   5.0;
i    =   85.0;
```



i

85.0

Memory is allocated.

The value 5.0 is stored in i.

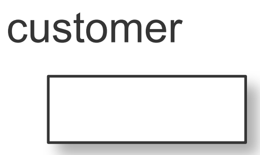
The value 85.0 is stored in i. Old value is lost.

Object Assignment

```
Customer   customer;
customer   =  new  Customer( );
customer   =  new  Customer( );
```

Object Assignment

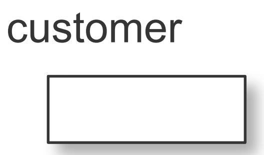
```
Customer    customer;
customer    =   new   Customer( );
customer    =   new   Customer( );
```



The identifier customer is allocated.

Object Assignment

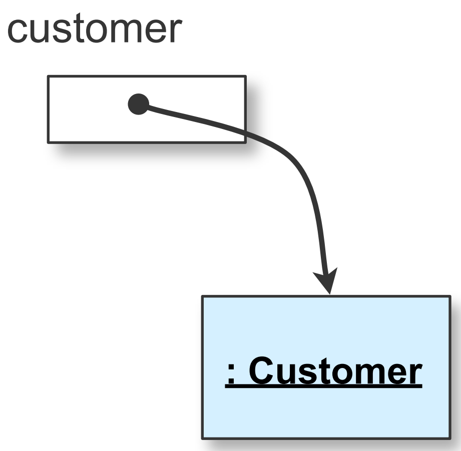
```
Customer    customer;
customer    =  new  Customer( );
customer    =  new  Customer( );
```



The identifier customer is allocated.

Object Assignment

```
Customer    customer;
customer    =   new   Customer( );
customer    =   new   Customer( );
```

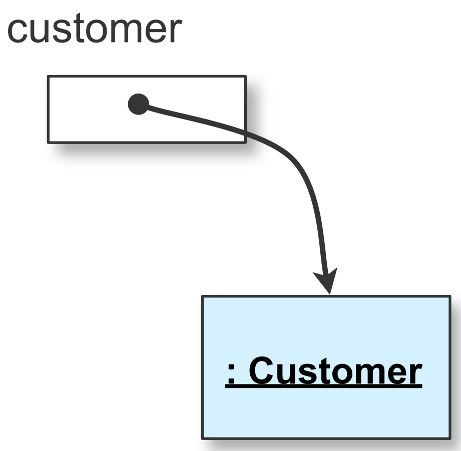


The identifier customer is allocated.

The **reference** to the first object is stored in customer.

Object Assignment

```
Customer   customer;
customer   =  new  Customer( );
customer   =  new  Customer( );
```

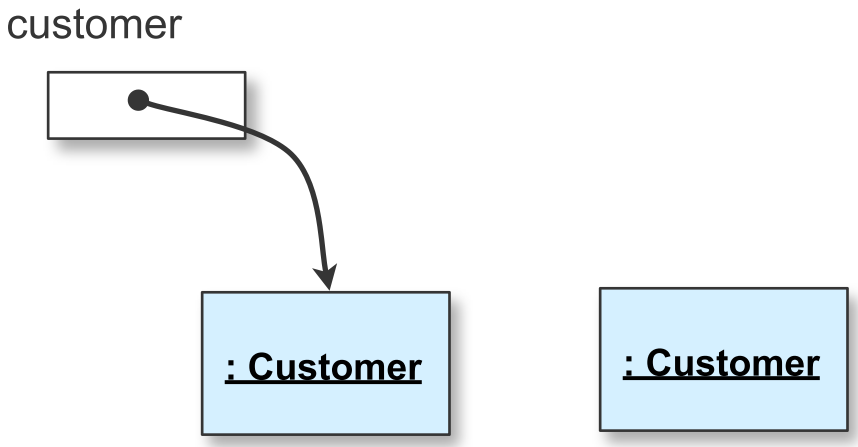


The identifier customer is allocated.

The **reference** to the first object is stored in customer.

Object Assignment

```
Customer   customer;
customer   =  new  Customer( );
customer   =  new  Customer( );
```



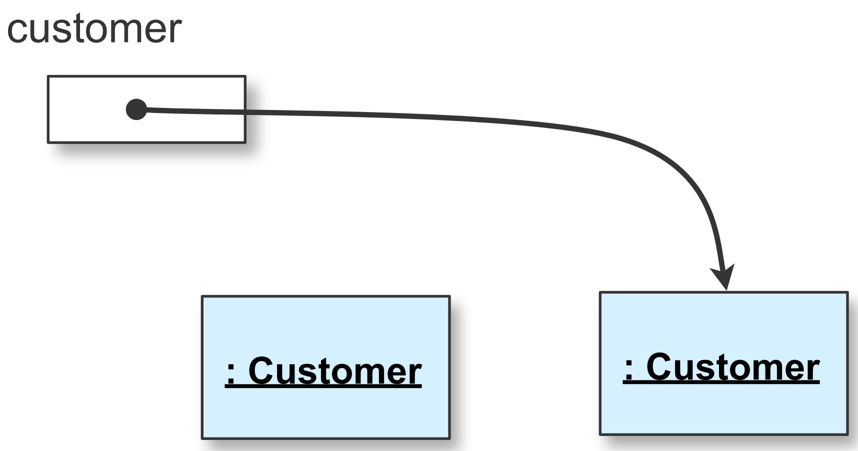
The identifier customer is allocated.

The **reference** to the first object is stored in customer.

© Sunil Prabhakar, Purdue University

Object Assignment

```
Customer   customer;
customer   =  new  Customer( );
customer   =  new  Customer( );
```



The identifier customer is allocated.

The **reference** to the first object is stored in customer.

The **reference** to the second object is stored in customer. The old reference is lost.

Assigning objects

```
Customer   cust1, cust2;
cust1    =  new   Customer( );
cust2    =  cust1;
```

Assigning objects

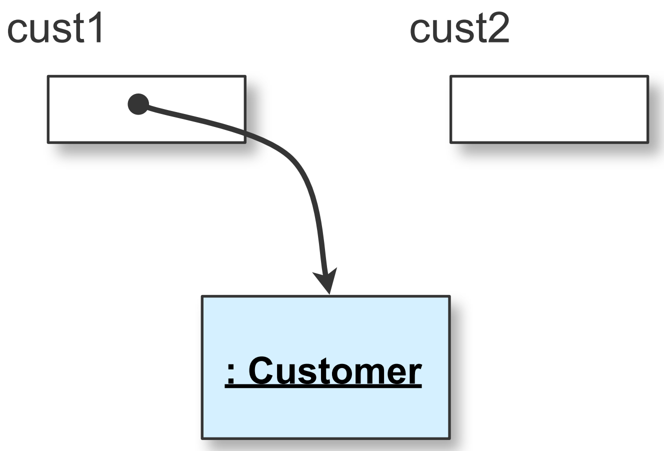
```
Customer   cust1, cust2;
cust1    =   new   Customer( );
cust2    =   cust1;
```

cust1
cust2

The identifiers are allocated.

Assigning objects

```
Customer   cust1, cust2;
cust1    =  new   Customer( );
cust2    =  cust1;
```

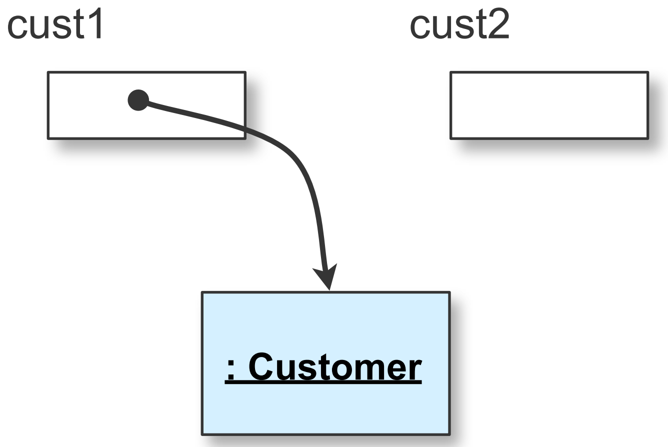


The identifiers are allocated.

The **reference** to the object is stored in cust1.

Assigning objects

```
Customer   cust1, cust2;
cust1    =  new  Customer( );
cust2    =  cust1;
```

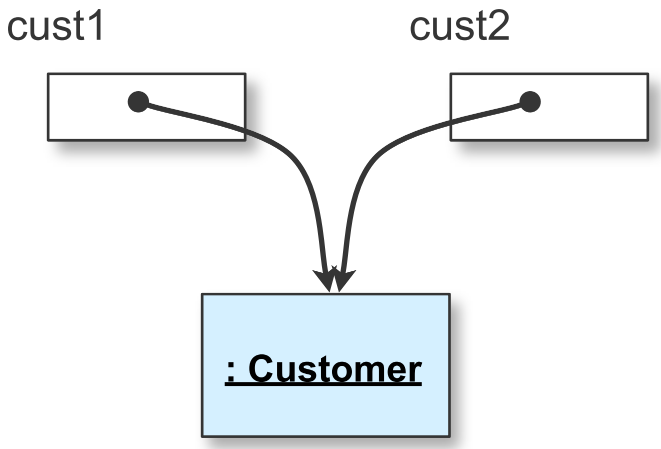


The identifiers are allocated.

The **reference** to the object is stored in cust1.

Assigning objects

```
Customer   cust1, cust2;
cust1    =  new   Customer( );
cust2    =  cust1;
```



The identifiers are allocated.

The **reference** to the object is stored in cust1.

The **reference** stored in cust1. is copied to cust2.

Assigning Primitive Data

```
double   i,j;
i    =  5.0;
j    =  i;
```

Assigning Primitive Data

```
double    i,j;
i    =   5.0;
j    =   i;
```

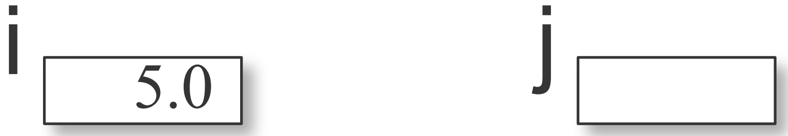
Assigning Primitive Data

```
double    i,j;
i    =   5.0;
j    =   i;
```

Memory is allocated.

Assigning Primitive Data

```
double    i,j;
i     =   5.0;
j     =   i;
```

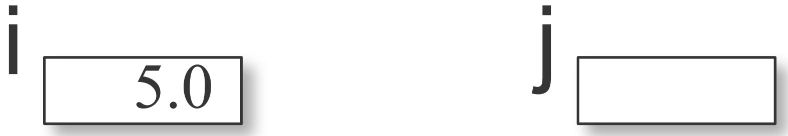


Memory is allocated.

The **value** stored in i is copied to j.

Assigning Primitive Data

```
double    i,j;
i    =   5.0;
j    =   i;
```

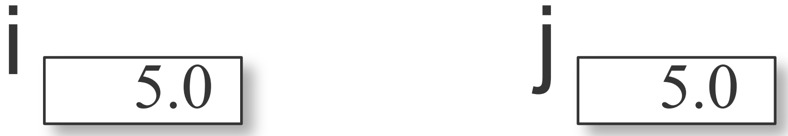


Memory is allocated.

The **value** stored in i is copied to j.

Assigning Primitive Data

```
double    i,j;
i    =  5.0;
j    =  i;
```

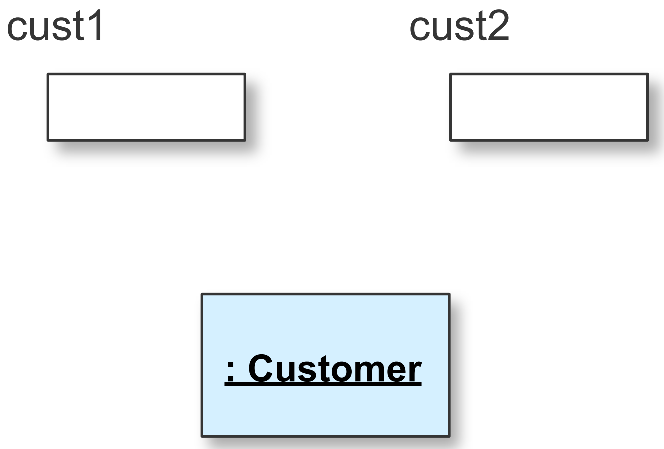


Memory is allocated.

The **value** stored in i is copied to j.

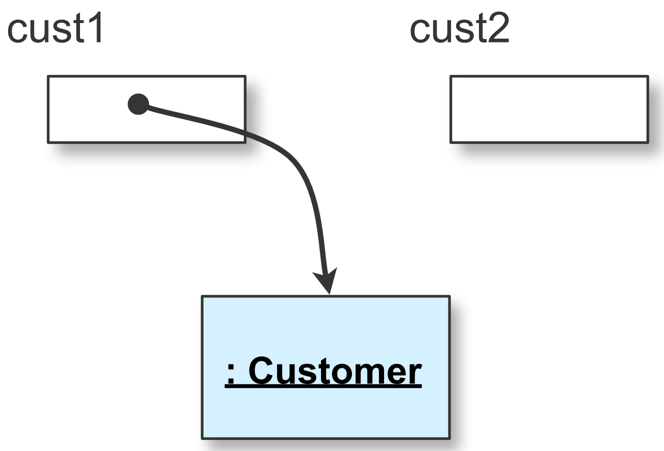
Really the same

```
Customer   cust1, cust2;
cust1    =  new   Customer( );
cust2    =  cust1;
```



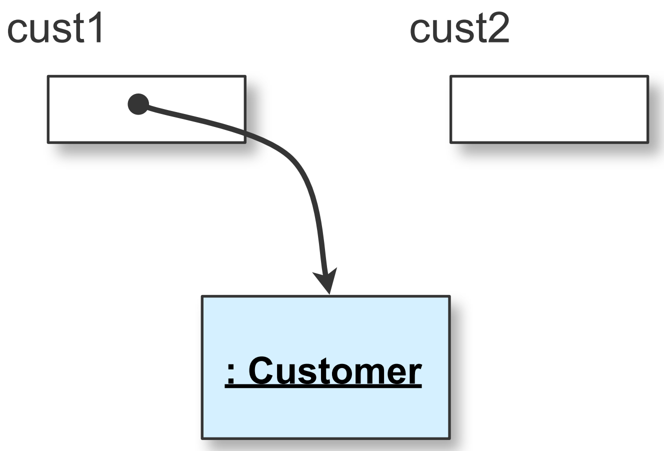
Really the same

```
Customer   cust1, cust2;
cust1    =  new   Customer( );
cust2    =  cust1;
```



Really the same

```
Customer   cust1, cust2;
cust1    =  new   Customer( );
cust2    =  cust1;
```



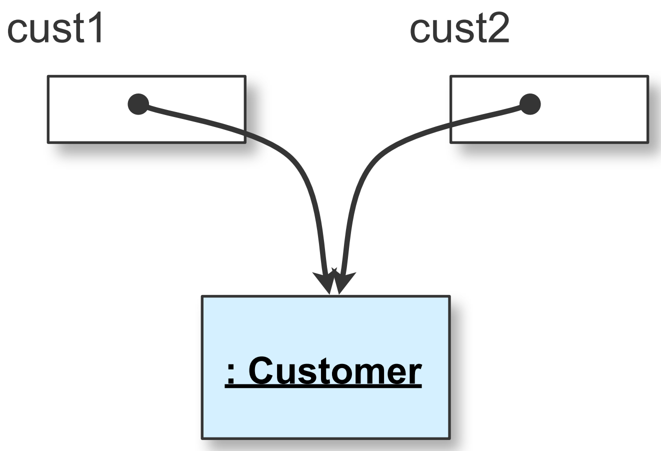
cust1

cust2

: Customer

Really the same

```
Customer   cust1, cust2;
cust1    =  new   Customer( );
cust2    =  cust1;
```



The **value** happens to be a reference to an object.

Hence **reference** type vs. **primitive** type.

The **value** stored in cust1 is copied to cust2.

Area and Perimeter (again)

```java
import java.util.Scanner;

public class CircleCalculator {

  public static void main (String[] args){
    double radius, area, circumference;

    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter radius");
    radius = scanner.nextDouble();

    circumference = 2.0 * 3.1415926535897932 * radius;
    area = 3.1415926535891932 * radius * radius;

    System.out.println("Given Radius:  " + radius + "\n" +
                       "Area:          " + area   + "\n" +
                       "Circumference: " + circumference);
  }
}
```

PURDUE UNIVERSITY

© Sunil Prabhakar, Purdue University

Constants

- Many programs use a constant value that should not be changed during execution.
- To avoid errors and reduce effort, we can define these once and reuse them.

```
final double  PI =3.14159265358979323846264338327950;
…
area = PI * radius * radius;
perimeter = 2 * PI * radius;
```

- The Math class defines PI and E
- Convention: all upper case for constants.

Constants

- Many programs use a constant value that should not be changed during execution.
- To avoid errors and reduce effort, we can define these once and reuse them.

```
final double  PI =3.14159265358979323846264338327950;
…
area = PI * radius * radius;
perimeter = 2 * PI * radius;
```

Note new keyword

- The Math class defines PI and E
- Convention: all upper case for constants.

Why use constants?

- Consistent values
  - No errors due to mistyping
- Easy to manage
  - If we need to change the precision of PI, we need only change it in one place.
- Programs are more readable.

Numeric Types vs. Strings

- Numeric data types are not strings!
  - There are no quotes used for numeric types
- What is the difference between 20 and "20"?
  - They are represented very differently by the computer.
  - 20 is represented in binary equivalent of the value 20. "20" is simply two distinct characters.
  - Doing math on numeric types is direct and fast.
  - Numeric values have special formats.
- We can convert between the two types
  - println() automatically converts numbers to strings

Parsing strings to numbers

- Consider the following attempt to read in the radius value.

```
double radius, area, circumference;
radius = JOptionPange.showInputDialog(null, "Enter radius");
```

- Not allowed by the compiler: wrong type.
- To convert we use a special method defined in a special class:

```
double radius, area, circumference;
String inputString;

inputString = JOptionPange.showInputDialog(null, "Enter radius");

radius = Double.parseDouble(inputString);
```

Wrapper classes

- Useful methods and constants for each of the primitive types are defined in corresponding 'wrapper' classes

Primitive Type	Wrapper class	Sample Method	Constants
byte	Byte	parseByte()	MIN_VALUE MAX_VALUE SIZE
short	Short	parseShort()	
int	Integer	parseInt()	
long	Long	parseLong()	
float	Float	parseFloat()	
double	Double	parseDouble()	

See API for details

CAUTION: + operator

- Recall the + operator for strings?
- It is different than the + operator for numeric data.
- If BOTH operands are numeric data then it is numeric addition
- Otherwise, it is string concatenation
  - if one is numeric it will be converted to a string!

```
double x=5.0, y=6.0, z;
String name = "234.5", str;

str = name + x + y;

str = x + y + name;

z = name + x + y;

z = x + y + name;
```

CAUTION: + operator

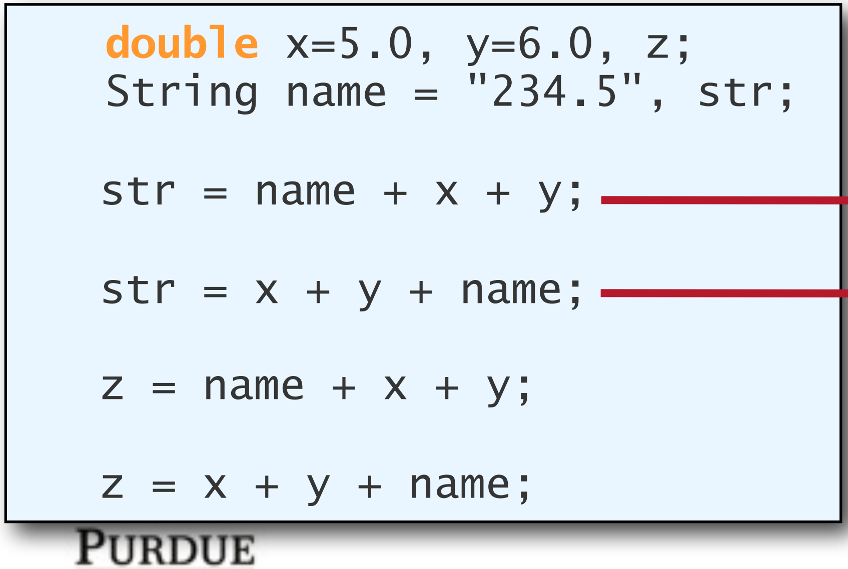
- Recall the + operator for strings?
- It is different than the + operator for numeric data.
- If BOTH operands are numeric data then it is numeric addition
- Otherwise, it is string concatenation
  - if one is numeric it will be converted to a string!

```
double x=5.0, y=6.0, z;
String name = "234.5", str;

str = name + x + y;

str = x + y + name;

z = name + x + y;

z = x + y + name;
```

→ `str = "234.55.06.0"`

CAUTION: + operator

- Recall the + operator for strings?
- It is different than the + operator for numeric data.
- If BOTH operands are numeric data then it is numeric addition
- Otherwise, it is string concatenation
  - if one is numeric it will be converted to a string!



```
double x=5.0, y=6.0, z;
String name = "234.5", str;

str = name + x + y;

str = x + y + name;

z = name + x + y;

z = x + y + name;
```

```
str = "234.55.06.0"
str = "11.0234.5"
```

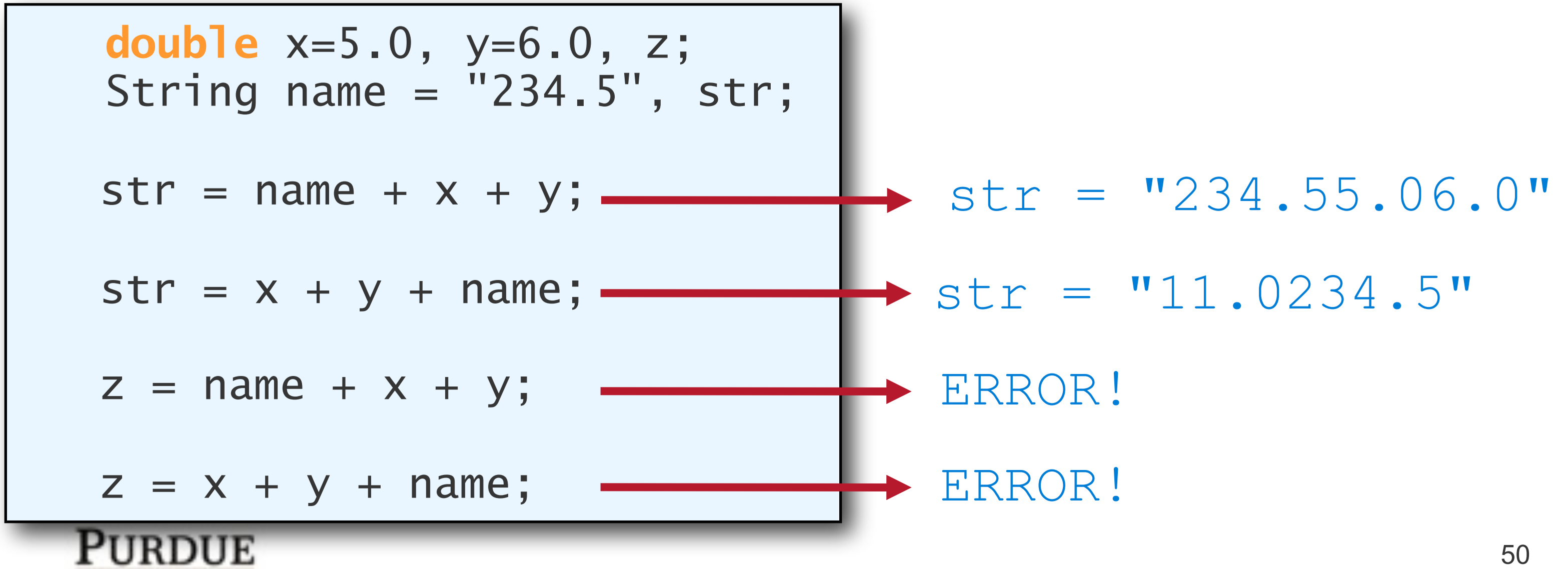
CAUTION: + operator

- Recall the + operator for strings?
- It is different than the + operator for numeric data.
- If BOTH operands are numeric data then it is numeric addition
- Otherwise, it is string concatenation
  - if one is numeric it will be converted to a string!

```
double x=5.0, y=6.0, z;
String name = "234.5", str;

str = name + x + y;      →   str = "234.55.06.0"

str = x + y + name;      →   str = "11.0234.5"

z = name + x + y;        →   ERROR!

z = x + y + name;
```

CAUTION: + operator

- Recall the + operator for strings?
- It is different than the + operator for numeric data.
- If BOTH operands are numeric data then it is numeric addition
- Otherwise, it is string concatenation
  - if one is numeric it will be converted to a string!



```
double x=5.0, y=6.0, z;
String name = "234.5", str;

str = name + x + y;
str = x + y + name;
z = name + x + y;
z = x + y + name;
```

```
str = "234.55.06.0"
str = "11.0234.5"
ERROR!
ERROR!
```

CAUTION: overflow & underflow

```
byte b;
b = 127;
b += 1;

System.out.println("b is" + b);
```

CAUTION: overflow & underflow

```
byte b;
b = 127;
b += 1;

System.out.println("b is" + b);
```

```
b is -128
```

CAUTION: overflow & underflow

```
byte b;
b = 127;
b += 1;

System.out.println("b is" + b);
```

```
b is -128
```

- Why?
- b went out of bounds and wrapped around!
  - Overflow.
- Similarly underflow can occur.
- Pick types wisely! Each has its own range -- be aware of it.
- Note: compiler can catch some problems.

CAUTION: overflow & underflow

```
 byte b;
b = 127;
b += 1;

 System.out.println("b is" + b);
```

```
b is -128
```

- Why?
- b went out of bounds and wrapped around!
  - Overflow.
- Similarly underflow can occur.
- Pick types wisely! Each has its own range -- be aware of it.
- Note: compiler can catch some problems.

```
 byte b;
b = 128;
```

[CAUTION: overflow & underflow]

```
byte b;
b = 127;
b += 1;

System.out.println("b is" + b);
```

→ `b is -128`

- Why?
- b went out of bounds and wrapped around!
  - Overflow.
- Similarly underflow can occur.
- Pick types wisely! Each has its own range -- be aware of it.
- Note: compiler can catch some problems.

```
byte b;
b = 128;
```

→ `will not compile!`

Shorthand operators

- When the right hand side of an assignment uses the same operand as the left hand side, we often use a shorthand form for some operators:

Operator	Example	Shorthand For
+=	x+=y;	x = x+y;
-=	x-=y;	x = x-y;
*=	x*=y;	x = x*y;
/=	x/=y;	x = x/y;
%=	x%=y;	x = x%y;

Shorthand operators

- When the right hand side of an assignment uses the same operand as the left hand side, we often use a shorthand form for some operators:

Operator	Example	Shorthand For
+=	x+=y;	x = x+y;
-=	x-=y;	x = x-y;
*=	x*=y;	x = x*y;
/=	x/=y;	x = x/y;
%=	x%=y;	x = x%y;

Note: no spaces