Numerical Data

CS 180
Sunil Prabhakar
Department of Computer Science
Purdue University
Problem

Write a program to compute the area and perimeter 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
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 so simple, that we don't need the overhead of objects
- Each type can be processed using only specific operators
Primitive Data Types

- **Numeric**
  - e.g., 2, 3, 3.1416,
  - for storing and operating on integer and real valued data

- **Character**
  - e.g., 'a', 'अ', 'ר', 'ᵉ', 'ꠏ', '맀', 'Ъ', 'ژ', '(_:)
  - for capturing characters for (almost) all world languages

- **Boolean**
  - only two allowed valued: true, false
  - logic data type

- This week we will study Numeric data.
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 + 
"Area: " + area + 
"Circumference: " + circumference);
    }
}
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

- 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;
```

<table>
<thead>
<tr>
<th>Operator</th>
<th>Equation</th>
<th>Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplication</td>
<td>$z = x \times y;$</td>
<td>$z = 12.5$</td>
</tr>
<tr>
<td>Addition</td>
<td>$z = x + y;$</td>
<td>$z = 7.5$</td>
</tr>
<tr>
<td>Subtraction</td>
<td>$z = x - y;$</td>
<td>$z = 2.5$</td>
</tr>
<tr>
<td>Division</td>
<td>$z = x \div y;$</td>
<td>$z = 2.0$</td>
</tr>
<tr>
<td>Unary negation</td>
<td>$z = -y;$</td>
<td>$z = -2.5$</td>
</tr>
</tbody>
</table>
Examples of expressions

```c
double tempC, tempF;
tempF = tempC * 9.0/5.0 + 32.0;
```

```c
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.
How is the following expression evaluated?

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

Answer: $x$ is added to $3 \times y$.

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.
# Precedence Rules

<table>
<thead>
<tr>
<th>Priority</th>
<th>Group</th>
<th>Operator</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>Subexpression</td>
<td>()</td>
<td>Starting with innermost ()</td>
</tr>
<tr>
<td></td>
<td>Unary operators</td>
<td>-, +</td>
<td>Left to right.</td>
</tr>
<tr>
<td></td>
<td>Multiplicative operators</td>
<td>*, /, %</td>
<td>Left to right.</td>
</tr>
<tr>
<td>Low</td>
<td>Additive operators</td>
<td>+, -</td>
<td>Left to right.</td>
</tr>
</tbody>
</table>
Precedence Examples

\[ \frac{x}{1} + \frac{4\times y}{3} - \frac{x}{z} + \frac{2}{x} = ? \]
\[ x + (4\times y) - \frac{x}{z} + \frac{2}{x} \]

\[ x + \left( \frac{y \times (4-x)}{z} \right) + \frac{2}{-x} \]

To be safe, use parentheses!
Numeric Data Types

- The type **double** that we saw allows us to store a very wide range of real number values:
  - $-1.7977 \times 10^{308}$ to $+1.797 \times 10^{308}$
  - 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
  - $-3.40282347 \times 10^{38}$ to $+3.40282347 \times 10^{38}$
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);
    }
}

© Sunil Prabhakar, Purdue University

Sunday, January 22, 2012
CAUTION: Imprecision

It is not possible to exactly represent every possible real valued number in double or float

- Fixed number of bits
  - **float**: 4 bytes -- 32 bits: \(2^{32}\) (~1 billion) values
  - **double**: 8 bytes -- 64 bits: \(2^{64}\) (~1 million trillion) values

- BUT, how many real numbers
  - between just 1.0 and 2.0? INFINITE!

**floats** and **doubles** may 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 slow
  - maybe using too much space (memory)
  - sometimes there is a (small) error in representation (more shortly)

- Instead, we have completely separate numeric types for **integer** data
  - **byte, short, int, long**
  - differ in size and range
# Numeric Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Content</th>
<th>Size (bytes)</th>
<th>Minimum Value</th>
<th>Maximum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>Integer</td>
<td>1</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>short</td>
<td>Integer</td>
<td>2</td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
<td>4</td>
<td>-2147483648</td>
<td>2147483647</td>
</tr>
<tr>
<td>long</td>
<td>Integer</td>
<td>8</td>
<td>-9,223,372,036,854,780,000</td>
<td>9,223,372,036,854,780,000</td>
</tr>
<tr>
<td>float</td>
<td>Real</td>
<td>4</td>
<td>-3.40282347 \times 10^{38}</td>
<td>3.40282347 \times 10^{38}</td>
</tr>
<tr>
<td>double</td>
<td>Real</td>
<td>8</td>
<td>-1.7977 \times 10^{308}</td>
<td>1.7977 \times 10^{308}</td>
</tr>
</tbody>
</table>
Operators for Integer types

```c
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)
It is important to note the behavior of division when the operands are all Integer types (`byte, short, int, long`).

- In this case we get integer division (`truncation` of the decimal part).

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;  \rightarrow  k = 2

    k = j/i;  \rightarrow  k = 0

    f = 5;
    g = 2;
    h = f/g;  \rightarrow  h = 2.5
}

© Sunil Prabhakar, Purdue University
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
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.
The numeric types can be arranged in order of their ranges as follows:

\[
\text{byte} < \text{short} < \text{int} < \text{long} < \text{float} < \text{double}
\]

The range of each type is strictly more precise than the range of each type to its left.

- E.g., any \text{byte} value can be stored in a \text{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;

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

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

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

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

Sunday, January 22, 2012
Type Casting

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

  - 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)

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

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

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

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

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

  i will store 3, not 3.5
Expression Types

- Each numeric expression has a data type. What is the type of $i + j$?
- Depends on the type 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

```java
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 \textbf{int}
  - To make it a \textbf{long} type append L or l
  - For \textbf{byte} and \textbf{short} -- no special type. If the value is an integer within the range of \textbf{byte} (\textbf{short}), it can be assigned to a \textbf{byte} (\textbf{short})
- If it has a decimal point, then its type is \textbf{double}.
  - To make it a \textbf{float} append F or f

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

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

See API for details
For assignment, the behavior of primitive variables seems to be different from that of class (reference) variables.

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

```java
Customer cust1, cust2;
cust1 = new Customer();
cust2 = new Customer();
cust1 = cust2;
```
Primitive Data: Declaration & Assignment

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

Memory is allocated.

Values are stored in those locations.
**Primitive Data Assignment**

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

```java
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

```java
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`.

© Sunil Prabhakar, Purdue University
Assigning Primitive Data

```c
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;
```

The value stored in cust1 is copied to cust2.

The value stored in cust1 is a reference to an object.

Hence reference type vs. primitive type.
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);
  }
}
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.

```java
final double PI = 3.1415926535897932384626433832795;
...
area = PI * radius * radius;
perimeter = 2 * PI * radius;
```

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.

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

Not allowed by the compiler: wrong type.

To convert we use a special method defined in a special class:

```java
double radius, area, circumference;
String inputString;
inputString = JOptionPane.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

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Wrapper class</th>
<th>Sample Method</th>
<th>Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>Byte</td>
<td>parseByte()</td>
<td>MIN_VALUE</td>
</tr>
<tr>
<td>short</td>
<td>Short</td>
<td>parseShort()</td>
<td>MAX_VALUE</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
<td>parseInt()</td>
<td>SIZE</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
<td>parseLong()</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
<td>parseFloat()</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
<td>parseDouble()</td>
<td></td>
</tr>
</tbody>
</table>

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!

```java
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;  // ERROR!
```

© Sunil Prabhakar, Purdue University
CAUTION: overflow & underflow

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 = 127;
b += 1;
System.out.println("b is" + b);
```

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

```
byte b;
b = 127;
b += 1;
System.out.println("b is" + b);
```

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

will not compile!
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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Shorthand For</th>
</tr>
</thead>
<tbody>
<tr>
<td>+=</td>
<td>x+=y;</td>
<td>x = x+y;</td>
</tr>
<tr>
<td>-=</td>
<td>x-=y;</td>
<td>x = x-y;</td>
</tr>
<tr>
<td>*=</td>
<td>x*=y;</td>
<td>x = x*y;</td>
</tr>
<tr>
<td>/=</td>
<td>x/=y;</td>
<td>x = x/y;</td>
</tr>
<tr>
<td>%=</td>
<td>x%=y;</td>
<td>x = x%y;</td>
</tr>
</tbody>
</table>