Basic Computation

Chapter 2

*JAVA: An Introduction to Problem Solving & Programming, 6*th Ed. By Walter Savitch
ISBN 0132162709

Outline

- Variables and Expressions
- The Class `String`
- Keyboard and Screen I/O
- Documentation and Style

Variables

- *Variables* store data such as numbers and letters.
  - Think of them as places to store data.
  - They are implemented as memory locations.
- The data stored by a variable is called its *value.*
  - The value is stored in the memory location.
- Its value can be changed as a program runs

Variables and Values

- Download **`EggBasket.java`** and open in drjava

```
If you have
6 eggs per basket and
10 baskets, then
the total number of eggs is 60
```

Sample Screen Output

Variables and Values

- Variables

```
numberOfBaskets
eggsPerBasket
totalEggs
```

- Assigning values

```
eggsPerBasket = 6;
eggsPerBasket = eggsPerBasket - 2;
```

Naming and Declaring Variables

- Choose names that are helpful such as `count` or `speed`, but not `c` or `s`.
- When you *declare* a variable, you provide its name and type.

```
int numberOfBaskets,eggsPerBasket;
```

- A variable's *type* determines what kinds of values it can hold (`int`, `double`, `char`, etc.).
- A variable must be declared before it is used.

Syntax and Examples

- Syntax

  `type variable_1, variable_2, …;`

  `(variable_1` is a generic variable called a *syntactic variable*)

- Examples

```
int styleChoice, numberOfChecks;
double balance, interestRate;
char jointOrIndividual;
```

Data Types

- A *class type* is used for a class of objects and has both data and methods.
  - `"Java is fun"` is a value of class type `String`
- A *primitive type* is used for simple, nondecomposable values such as an individual number or individual character.
  - `int`, `double`, and `char` are primitive types.

Primitive Types

- Figure 2.1 Primitive Types

Type Name	Kind of Value	Memory Used	Range of Values
`byte`	Integer	1 byte	–128 to 127
`short`	Integer	2 bytes	–32,768 to 32,767
`int`	Integer	4 bytes	–2,147,483,648 to 2,147,483,647
`long`	Integer	8 bytes	–9,223,372,036,8547,75,808 to 9,223,372,036,854,775,807
`float`	Floating-point	4 bytes	$\pm 3.40282347 \times 10^{+38}$ to $\pm 1.40239846 \times 10^{-45}$
`double`	Floating-point	8 bytes	$\pm 1.79769313486231570 \times 10^{+308}$ to $\pm 4.94065645841246544 \times 10^{-324}$
`char`	Single character (Unicode)	2 bytes	All Unicode values from 0 to 65,535
`boolean`		1 bit	True or false

Java Identifiers

- An *identifier* is a name, such as the name of a variable.
- Identifiers may contain only
  - Letters
  - Digits (0 through 9)
  - The underscore character (_)
  - And the dollar sign symbol ($) which has a special meaning (avoid using it)
- The first character cannot be a digit.

Java Identifiers

- Identifiers may not contain any spaces, dots (.), asterisks (*), or other characters:

  `7-11` `netscape.com` `util.*` (not allowed)

- Identifiers can be arbitrarily long.
- Since Java is *case sensitive*, `stuff`, `Stuff`, and `STUFF` are different identifiers.

Keywords or Reserved Words

- Words such as `if` are called *keywords* or *reserved words* and have special, predefined meanings.
  - Cannot be used as identifiers.
  - See e.g. Wikipedia for a list of Java keywords: http://en.wikipedia.org/wiki/List_of_Java_keywords
- Example keywords: `int`, `public`, `class`

Naming Conventions

- Class types begin with an uppercase letter (e.g. `String`).
- Primitive types begin with a lowercase letter (e.g. `int`).
- Variables of both class and primitive types begin with a lowercase letters (e.g. `myName`, `myBalance`).
- Multiword names are "punctuated" using uppercase letters.

Where to Declare Variables

- Declare a variable
  - Just before it is used or
  - At the beginning of the section of your program that is enclosed in **{ }**.

```
public static void main(String[] args)
{
    /* declare variables here */
     . . .
}
```

Primitive Types

- Four integer types (`byte`, `short`, `int`, and `long`)
  - `int` is most common
- Two floating-point types (`float` and `double`)
  - `double` is more common
- One character type (`char`)
- One boolean type (`boolean`)

Examples of Primitive Values

- Integer types

```
0   -1   365   12000
```

- Floating-point types

```
0.99   -22.8   3.14159 5.0
```

- Character type

```
'a'   'A'   '#'   ' '
```

- Boolean type

```
true   false
```

Assignment Statements

- An assignment statement is used to assign a value to a variable.

```
answer = 42;
average = sum / count;
firstInitial = 'W';
done = true;
```

Initializing Variables

- A variable that has been declared, but no yet given a value is said to be *uninitialized.*
- Uninitialized class variables have the value `null`.
- Uninitialized primitive variables may have a default value.
- It's good practice not to rely on a default value.

Initializing Variables

- To protect against an uninitialized variable (and to keep the compiler happy), assign a value at the time the variable is declared.
- Examples:

```
int count = 0;
char grade = 'A';
```

Assignment Evaluation

- The expression on the right-hand side of the assignment operator (=) is evaluated first.
- The result is used to set the value of the variable on the left-hand side of the assignment operator.

```
score = numberOfCards + handicap;
eggsPerBasket = eggsPerBasket - 2;
```

Simple Input

- Download **`EggBasket2.java`**

```
Enter the number of eggs in each basket:
6
Enter the number of baskets:
10
If you have
6 eggs per basket and
10 baskets, then
the total number of eggs is 60
Now we take two eggs out of each basket.
You now have
4 eggs per basket and
10 baskets.
The new total number of eggs is 40
```

Sample screen output

Simple Input

- Keyboard input can be done using a **`Scanner`** object
- At the top of the source file:

```
import java.util.Scanner;
```

- Data can be entered from the keyboard using

```
Scanner keyboard = new Scanner(System.in);
```

followed, for example, by

```
eggsPerBasket = keyboard.nextInt();
```

which reads one int value from the keyboard and assigns it to eggsPerBasket.

Simple Screen Output

```
System.out.println("The count is " + count);
```

- Outputs the sting literal `"the count is "`
- Followed by the current value of the variable `count`.

Constants

- Literal expressions such as `2, 3.7,` or `'y'` are called *constants*.
- Integer constants can be preceded by a `+` or `–` sign, but cannot contain commas.

Named Constants

- Java provides mechanism to …
  - Define a variable
  - Initialize it
  - Fix the value so it cannot be changed

```
public static final Type Variable = Constant;
```

  - Example

```
public static final int MAX_ENTRIES = 100;
```

Assignment Compatibilities

- Java is said to be *strongly typed.*
  - You can't, for example, assign a floating point value to a variable declared to store an integer.
- Sometimes conversions between numbers are possible.

  `doubleVariable = 7;`

  is possible even if `doubleVariable` is of type `double`, for example.

Assignment Compatibilities

- Automatic type conversion takes place if a value of any type in the following list is assigned to a variable of a type further right in the list:

```
byte -> short -> int -> long
                     -> float -> double
```

  - i.e., a value of one type can be assigned to a variable of any type further to the right
- In particular, a value of any integer type can be assigned to a variable of any floating-point type.

Type Casting

- A *type cast* temporarily changes the value of a variable from the declared type to some other type.
- For example,

```
double distance = 9.0;
int points = (int) distance; //truncates
```

- Illegal without `(int)`
- Any nonzero value to the right of the decimal point is *truncated* rather than *rounded.*

Arithmetic Operators

- Arithmetic expressions can be formed using the `+`, `-`, `*`, and `/` operators together with variables or numbers referred to as *operands.*
- When both operands are of the same type, the result is of that type.
- When one of the operands is a floating-point type and the other is an integer, the result is a floating point type.

Arithmetic Operations

- Example

  If `hoursWorked` is an `int` to which the value `40` has been assigned, and `payRate` is a `double` to which `8.25` has been assigned

  ```
  hoursWorked * payRate
  ```

  is a `double` with a value of `330.0`.

The Division Operator

- The division operator (`/`) behaves as expected if one of the operands is a floating-point type.
- When both operands are integer types
  - The result is also an integer type
  - The result is truncated, not rounded.
  - Hence, `99/100` has a value of `0`, and `12/5` has a value of `2`.

The `mod` Operator

- The `mod` (`%`) operator is used with operators of integer type to obtain the **remainder** after integer division.
- 14 divided by 4 is 3 *with a remainder of 2.*
  - Hence, `14 % 4` is equal to `2`.
- The mod operator has many uses, including
  - determining if an integer is odd or even
  - determining if one integer is evenly divisible by another integer.

Parentheses and Precedence

- Parentheses can communicate the order in which arithmetic operations are performed
- examples:

```
(cost + tax) * discount
(cost + (tax * discount)
```

- Without parentheses, an expressions is evaluated according to the *rules of precedence.*

Precedence Rules

Highest Precedence

First: the unary operators +, -, !, ++, and --

Second: the binary arithmetic operators *, /, and %

Third: the binary arithmetic operators + and –

Lowest Precedence

- The *binary* arithmetic operators `*`, `/`, and `%`, have *lower precedence* than the *unary* operators `+`, `-`, `++`, `--`, and `!`, but have *higher precedence* than the binary arithmetic operators `+` and `-`.

Precedence Rules

- When unary operators have equal precedence, the operator on the right acts before the operation(s) on the left.
- Even when parentheses are not needed, they can be used to make the code clearer.

```
balance + (interestRate * balance)
```

- Spaces also make code clearer

```
balance + interestRate*balance
```

but spaces do not dictate precedence.

Sample Expressions

- Figure 2.3 Some Arithmetic Expressions in Java

Ordinary Math	Java (Preferred Form)	Java (Parenthesized)
$rate^2 + delta$	`rate * rate + delta`	`(rate * rate) + delta`
$2(salary + bonus)$	`2 * (salary + bonus)`	`2 * (salary + bonus)`
$\frac{1}{time + 3mass}$	`1 / (time + 3 * mass)`	`1 / (time + (3 * mass))`
$\frac{a-7}{t+9v}$	`(a - 7) / (t + 9 * v)`	`(a - 7) / (t + (9 * v))`

Specialized Assignment Operators

- Assignment operators can be combined with arithmetic operators (including `-`, `+`, `*`, `/`, and `%`).

```
amount = amount + 5;
```

can be written as

```
amount += 5;
```

yielding the same results.

Specialized Assignment Operators

- Try it out:

```
int amount = 20;
amount -= 5;
amount *= 3;
```

Increment and Decrement Operators

- Used to increase (or decrease) the value of a variable by 1
- Easy to use, important to recognize
- The increment operator

  `count++` or `++count`
- The decrement operator

  `count--` or `--count`

Increment and Decrement Operators

- equivalent operations

```
count++;
++count;
count = count + 1;

count--;
--count;
count = count - 1;
```

Increment and Decrement Operators in Expressions

- after executing

```
int m = 4;
int result = 3 * (++m)
```

`result` has a value of `15` and `m` has a value of `5`

- after executing

```
int m = 4;
int result = 3 * (m++)
```

`result` has a value of `12` and `m` has a value of `5`

- Avoid using `++` and `--` in expressions

The Class `String`

- We've used constants of type `String` already.

  `"Enter a whole number from 1 to 99."`

- A value of type `String` is a
  - Sequence of characters
  - Treated as a single item.

String Constants and Variables

- Declaring

```
String greeting;
 greeting = "Hello!";
```

or

```
String greeting = "Hello!";
```

or

```
String greeting = new String("Hello!");
```

- Printing

```
System.out.println(greeting);
```

Concatenation of Strings

- Two strings are *concatenated* using the + operator.

```
String greeting = "Hello";
String sentence;
sentence = greeting + " officer";
System.out.println(sentence);
```

- Any number of strings can be concatenated using the + operator.

Concatenating Strings and Integers

```
String solution;
solution = "The answer is " + 42;
System.out.println(solution);
```

```
The answer is 42
```

String Methods

- An object of the `String` class stores data consisting of a sequence of characters.
- Objects have methods as well as data
- The `length()` method returns the number of characters in a particular `String` object.

```
String greeting = "Hello";
int n = greeting.length();
```

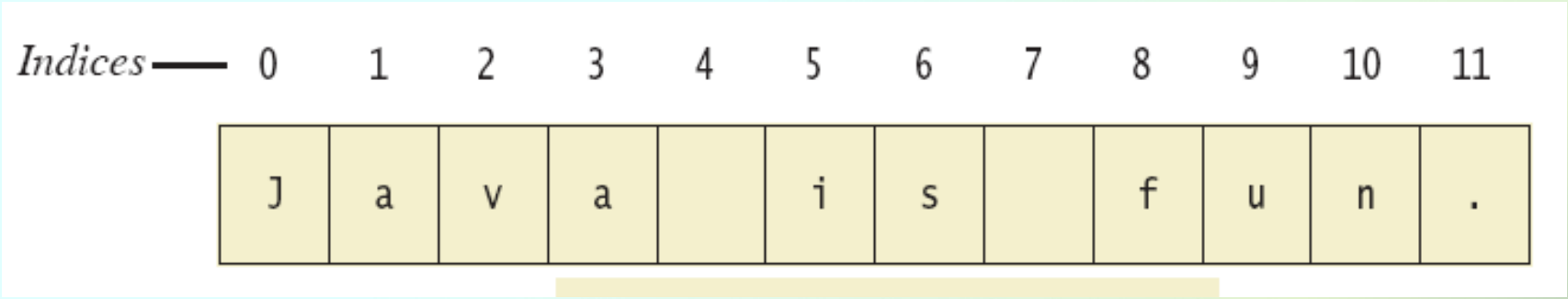
The Method `length()`

- The method `length()` returns an `int`.
- You can use a call to method `length()` anywhere an `int` can be used.

```
int count = command.length();
System.out.println("Length is " +
    command.length());
count = command.length() + 3;
```

`String` Indices

- Figure 2.4



- Positions start with 0, not 1.
  - The 'J' in "Java is fun." is in position 0
- A position is referred to an an *index.*

  The `'f'` in `"Java is fun."` is at index 8.

`String` Methods

- Follow the "Java API" link on the course webpage
- Find the `String` class

`charAt(`*Index*`)` Returns the character at *Index* in this string. Index numbers begin at 0.
`compareTo(`*A_String*`)` Compares this string with *A_String* to see which string comes first in the lexicographic ordering. (Lexicographic ordering is the same as alphabetical ordering when both strings are either all uppercase letters or all lowercase letters.) Returns a negative integer if this string is first, returns zero if the two strings are equal, and returns a positive integer if *A_String* is first.
`concat(`*A_String*`)` Returns a new string having the same characters as this string concatenated with the characters in *A_String*. You can use the ⇓ operator instead of `concat`.
`equals(`*Other_String*`)` Returns true if this string and *Other_String* are equal. Otherwise, returns false.

Figure 2.5a

`String` Methods

`equalsIgnoreCase(`*Other_String*`)` Behaves like the method `equals`, but considers uppercase and lowercase versions of a letter to be the same.
`indexOf(`*A_String*`)` Returns the index of the first occurrence of the substring *A_String* within this string. Returns -1 if *A_String* is not found. Index numbers begin at 0.
`lastIndexOf(`*A_String*`)` Returns the index of the last occurrence of the substring *A_String* within this string. Returns -1 if *A_String* is not found. Index numbers begin at 0.

Figure 2.5b

`String` Methods

`length()` Returns the length of this string.
`toLowerCase()` Returns a new string having the same characters as this string, but with any uppercase letters converted to lowercase.
`toUpperCase()` Returns a new string having the same characters as this string, but with any lowercase letters converted to uppercase.

Figure 2.5c

`String` Methods

`replace(`*OldChar*, *NewChar*`)` Returns a new string having the same characters as this string, but with each occurrence of *OldChar* replaced by *NewChar*.
`substring(`*Start*`)` Returns a new string having the same characters as the substring that begins at index *Start* of this string through to the end of the string. Index numbers begin at 0.
`substring(`*Start*, *End*`)` Returns a new string having the same characters as the substring that begins at index *Start* of this string through, but not including, index *End* of the string. Index numbers begin at 0.
`trim()` Returns a new string having the same characters as this string, but with leading and trailing whitespace removed.

Figure 2.5d

The Empty String

- A string can have any number of characters, including zero.
- The string with zero characters is called the *empty* string.
- The empty string is useful and can be created in many ways including

```
String s3 = "";
```

The Empty String

- Not allowed to call methods on a **`null`** object
- Try this out:

```
String str;
int strLen;
strLen = str.length(); //NullPointerException

str = "";
strLen = str.length();   // strLen is now 0
```

`String` Processing

- No methods allow you to change the value of a `String` object.
- But you can change the value of a `String` variable.

```
String str = "Hello";
str = str + " World"; // 创建了一个新的字符串对象，并将其赋给str
```

- Download, compile, and run `StringDemo.java`

```
Text processing is hard!
01234567890123456789012 3
The word "hard" starts at index 19
The changed string is:
TEXT PROCESSING IS EASY!
```

Sample Screen Output

Escape Characters

- How would you print

```
"Java" refers to a language.
```
?

- The compiler needs to be told that the quotation marks (") do not signal the start or end of a string, but instead are to be printed.

```
System.out.println(
"\"Java\" refers to a language.");
```

Escape Characters

`\"`	Double quote.
`\'`	Single quote.
`\\`	Backslash.
`\n`	New line. Go to the beginning of the next line.
`\r`	Carriage return. Go to the beginning of the current line.
`\t`	Tab. Add whitespace up to the next tab stop.

- Figure 2.6
- Each escape sequence is a single character even though it is written with two symbols.

Examples

```
System.out.println("abc\\def");
```

```
abc\def
```

```
System.out.println("new\nline");
```

```
new
line
```

```
char singleQuote = '\'';
System.out.println(singleQuote);
```

```
'
```

The Unicode Character Set

- Most programming languages use the *ASCII* character set.
- Java uses the *Unicode* character set which includes characters from many different alphabets (e.g. the ASCII character set).

Screen Output

- We've seen several examples of screen output already.
- `System.out` is an object that is part of Java.
- `println()` is one of the methods available to the `System.out` object.
- Use `print` when the next item printed should continue on the same line
- Use `println` when the next item printed should start on a new line

Screen Output

- The concatenation operator (+) is useful when everything does not fit on one line.

```
System.out.println("Lucky number = "
  + 13 +
  "Secret number = " + number);
```

- Do not break the line except immediately before or after the concatenation operator (+).

Screen Output

- Print `One, two, three, four.` :

```
System.out.print("One, two,");
System.out.println(" three, four.");
```

OR

```
System.out.println("One, two," +
                   " three, four.");
```

ILLEGAL to continue string on next line:

```
System.out.println("One, two,
                   three, four.");
```

Keyboard Input

- Java's **`Scanner`** class has reasonable facilities for handling keyboard input.
- The **`Scanner`** class is part of the **`java.util`** package.
  - A *package* is a library of classes.

Using the Scanner Class

- Near the beginning of your program, insert

```
import java.util.Scanner;
```

- Create an object of the `Scanner` class

```
Scanner keyboard =
   new Scanner (System.in)
```

- Read data (an `int` or a `double`, for example)

```
int n1 = keyboard.nextInt();
double d1 = keyboard.nextDouble();
```

Keyboard Input Demonstration

- Download, compile and run
  `ScannerDemo.java`

```
Enter two whole numbers
separated by one or more spaces:
  42   43
You entered 42 and 43
Next enter two numbers.
A decimal point is OK.
 9.99  21
You entered 9.99 and 21.0
Next enter two words:
plastic spoons
You entered "plastic" and "spoons"
Next enter a line of text:
May the hair on your toes grow long and curly.
You entered "May the hair on your toes grow long and curly."
```

Sample Screen Output

Scanner Methods

- The `next` method simply reads all the characters up to the next whitespace
- There is a `next`***`X`*** method for each of the primitive types (`nextInt, nextBoolean, nextDouble...`)
  - skip whitespace, including newlines, until a non-whitespace character is encountered
  - read the characters up to the next whitespace
  - convert the characters read to the appropriate type (int, boolean, double...)

`nextLine()` Method Caution

- The `nextLine()` method reads
  - The remainder of the current line,
  - Even if it is empty.

`nextLine()` Method

- What gets printed?

```
import java.util.Scanner;
public class SandBox {
  public static void main(String[] args) {
    int n;
    String s1, s2;
    Scanner keyboard = new Scanner(System.in);
    n = keyboard.nextInt();
    s1 = keyboard.nextLine();
    s2 = keyboard.nextLine();

    System.out.println(n);
    System.out.println(s1);
    System.out.println(s2);
  }
}
```

```
42 is the answer
and don't you
forget it.
```

```
42
and don't you
forget it.
```

`nextLine()` Method

- What gets printed?

```
import java.util.Scanner;
public class SandBox {
  public static void main(String[] args) {
    int n;
    String s1, s2;
    Scanner keyboard = new Scanner(System.in);
    n = keyboard.nextInt();
    keyboard.nextLine();
    s1 = keyboard.nextLine();
    s2 = keyboard.nextLine();

    System.out.println(n);
    System.out.println(s1);
    System.out.println(s2);
  }
}
```

```
42 is the answer
and don't you
forget it.
```

```
42
and don't you
forget it.
```

Exercise

- Write a program called AddressReader that reads keyboard data in the following format:

lastName age

address

then prints all 3 data values on separate lines.

- Sample input:

Smith 21

Hauptstr 4, 12345 Kleindorf

Exercise

```
import java.util.Scanner;
public class AddressReader {
  public static void main(String[] args) {
    int age = 0;
    String lastName = "", address = "";
    Scanner keyboard = new Scanner(System.in);

    lastName = keyboard.next();
    age = keyboard.nextInt();
    keyboard.nextLine();
    address = keyboard.nextLine();

    System.out.println(age + "\n" + lastName +
                       "\n" + address);
  }
}
```

Documentation and Style

- Most programs are modified over time to respond to new requirements.
- Programs which are easy to read and understand are easy to modify.
- Even if it will be used only once, you have to read it in order to debug it.
- Always use:
  - Meaningful variable names
  - Comments
  - Indentation
  - Named constants

Meaningful Variable Names

- A variable's name should suggest its use.
- Observe conventions in choosing names for variables.
  - Use only letters and digits.
  - "Punctuate" using uppercase letters at word boundaries (e.g. `taxRate, firstName`).
  - Start variables with lowercase letters.
  - Start class names with uppercase letters.

Comments

- The best programs are self-documenting.
  - Clean style
  - Well-chosen names
- Comments are written into a program as needed to explain the program.
  - They are useful to the programmer, but they are ignored by the compiler.

Comments

- A comment can begin with //.
- Everything after these symbols and to the end of the line is treated as a comment and is ignored by the compiler.

```
double radius; //in centimeters
```

Comments

- A comment can begin with `/*` and end with `*/`
- Everything between these symbols is treated as a comment and is ignored by the compiler.

```
/*
This program should only
be used on alternate Thursdays,
except during leap years, when it should
only be used on alternate Tuesdays.
*/
```

When to Use Comments

- Begin each program file with an explanatory comment
  - What the program does
  - The name of the author
  - Contact information for the author
  - Date of the last modification.
- Provide only those comments which the expected reader of the program file will need in order to understand it.

javadoc Comments

- A *javadoc* comment, begins with `/**` and ends with `*/`.
- It can be extracted automatically from Java software.

```
/**
  * method change requires the
  * number of coins to be
  * nonnegative
  */
```

javadoc Comments

- Add the following javadoc comment right above the class definition in the program you wrote earlier:

```
/**
 * Program to practice using a Scanner.
 */
```

- Add this javadoc comment right above the main method:

```
/**
 * Read stuff from the keyboard. Uses
 * nextLine to skip remainder of a line.
 */
```

- Tools -> Javadoc -> Preview Javadoc for Current Document

Comments Example

- View **`CircleCalculation.java`**

```
Enter the radius of a circle in inches:
2.5
A circle of radius 2.5 inches
has an area of 19.6349375 square inches.
```

Sample Screen Output

Indentation

- Programs have a lot of structure
  - Methods are a part of class definitions
  - Variable declarations and code are parts of a method
- We say that some parts are nested within others
- Indentation should be consistent and indicate nesting clearly.
- Indentation does not change the behavior of the program but helps communicate to the human reader the nested structures of the program

DrJava Configuration

- IDEs like drjava can be configured to act the way we want them to:

Edit -> Preferences

Miscellaneous

Indent Level : 4

check box: Automatically Close Block Comments

Resource Locations

set web browser (/usr/bin/konqueror ???)

Display Options

check box: Show All Line Numbers

Notifications

Indentation with DrJava

- Indent a single line of code:
  - Press the **tab key**
- Indent the whole program:
  - Type **Strg-a** (or choose Edit->Select All)
  - Press the **tab key** (or choose Edit->Indent Lines(s))

Using Named Constants

- To avoid confusion, always name constants (and variables).

```
area = PI * radius * radius;
```

  is clearer than

```
area = 3.14159 * radius * radius;
```

- Place constants near the beginning of the program (not inside a method).

Named Constants

- Once the value of a constant is set (or changed by an editor), it can be used (or reflected) throughout the program.

```
public static final double INTEREST_RATE = 6.65;
```

- If a literal (such as 6.65) is used instead, every occurrence must be changed, with the risk that another literal with the same value might be changed unintentionally.

Declaring Constants

- Syntax

```
public static final Type Name = Value;
```

- Examples

```
public static final double PI = 3.14159;
public static final String MOTTO =
    "The customer is always right.";
```

- By convention, uppercase letters are used for constants.

Named Constants Example

- Compare **`CircleCalculation.java and CircleCalculation2.java`**

```
Enter the radius of a circle in inches:
2.5
A circle of radius 2.5 inches
has an area of 19.6349375 square inches.
```

Sample Screen Output