# CHAPTER 2 -BASIC COMPUTATION

CSC111

DR. SULTAN ALFARHOOD

sultanf@ksu.edu.sa

# OUTLINE

- Java Program Structure
- Hello Program
- Saving, Compiling and Running Java Programs
- Comments
- Discovering what is a variable
- Discovering what is a data type
- Learning about the basic data types
- Constants and variables identifiers
- Get acquainted with how to select proper types for numerical data
- Write arithmetic expressions in Java

# JAVA PROGRAM STRUCTURE

```
public class MyProgramName {

    public static void main( String[] args ){

    }

}
```

# JAVA PROGRAM STRUCTURE

```
// import Section - import used Java libraries
public class MyProgramName {

    // main method
    public static void main( String[] args ){

        // Declaration section - Declare needed variables

        // Input section - Enter required data

        // Processing section - Processing Statements

        // Output section - Display expected results

    } // end main

} // end class
```

# HELLO PROGRAM

```
// import Section – import used Java libraries
public class HelloProgram {

    // main method
    public static void main( String[] args ){

        // Declaration section – Declare needed variables

        // Input section – Enter required data

        // Processing section – Processing Statements

        // Output section – Display expected results

        System.out.println("… Hello …");

    } // end main

} // end class
```

# SAVING, COMPILING AND RUNNING JAVA PROGRAMS

- Saving a Java program
  - A file having a name same as the class name should be used to save the program. The extension of this file is ".java".
    - Hello program must be saved in a file called "HelloProgram.java".
- Compiling a Java program
  - Call the Java compiler **javac**:
    - javac HelloProgram.java
    - The Java compiler generates a file called "HelloProgram.class" (the bytecode).
- Running a Java program
  - Call the Java Virtual Machine **java**:
    - java HelloProgram.class

# COMMENTS

- The best programs are self-documenting.
  - Clean style
  - Well-chosen names
- Comments are written into a program as needed 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.
*/
```

# 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 */
```

# 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.

# HELLO PROGRAM

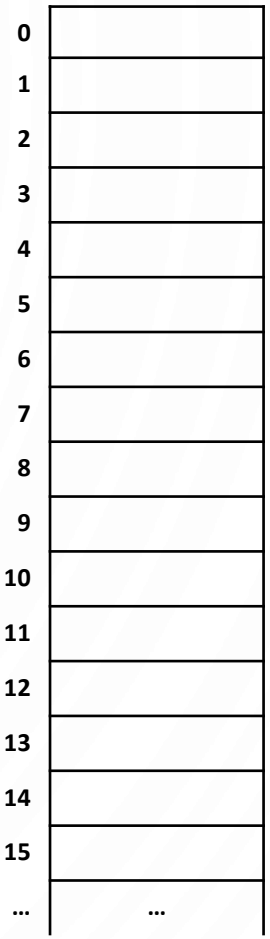
```
// import Section – import used Java libraries
public class HelloProgram {

    // main method
    public static void main( String[] args ){

        // Declaration section – Declare needed variables

        // Input section – Enter required data

        // Processing section – Processing Statements

        // Output section – Display expected results
        System.out.println("… Hello…");

        /* This comment starts here
        continues here
        ends at the end of this line */

    } // end main

} // end class
```

# SUM PROGRAM

```
public class SumProgram {

    public static void main( String[] args ){

        int num1 ;
        int num2 ;
        int sum ;

        num1 = 5 ;
        num2 = 3 ;

        sum = num1 + num2 ;

        System.out.println(sum);
    }
}
```

# PROGRAMS AND DATA

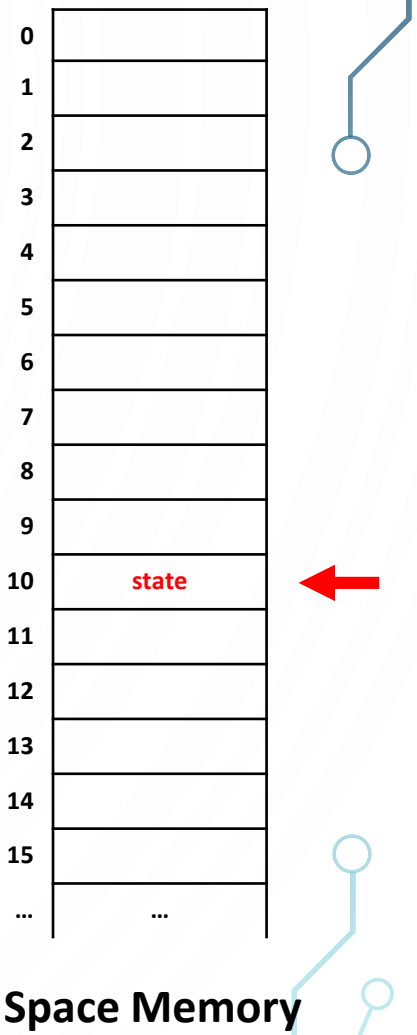
- Most programs require the temporary storage of data
- The data to be processed is stored in a temporary storage in the computer's memory: **space memory**
- A space memory has three characteristics
  - Identifier
  - Data Type
  - State




**Space Memory**

# STATE OF THE SPACE MEMORY

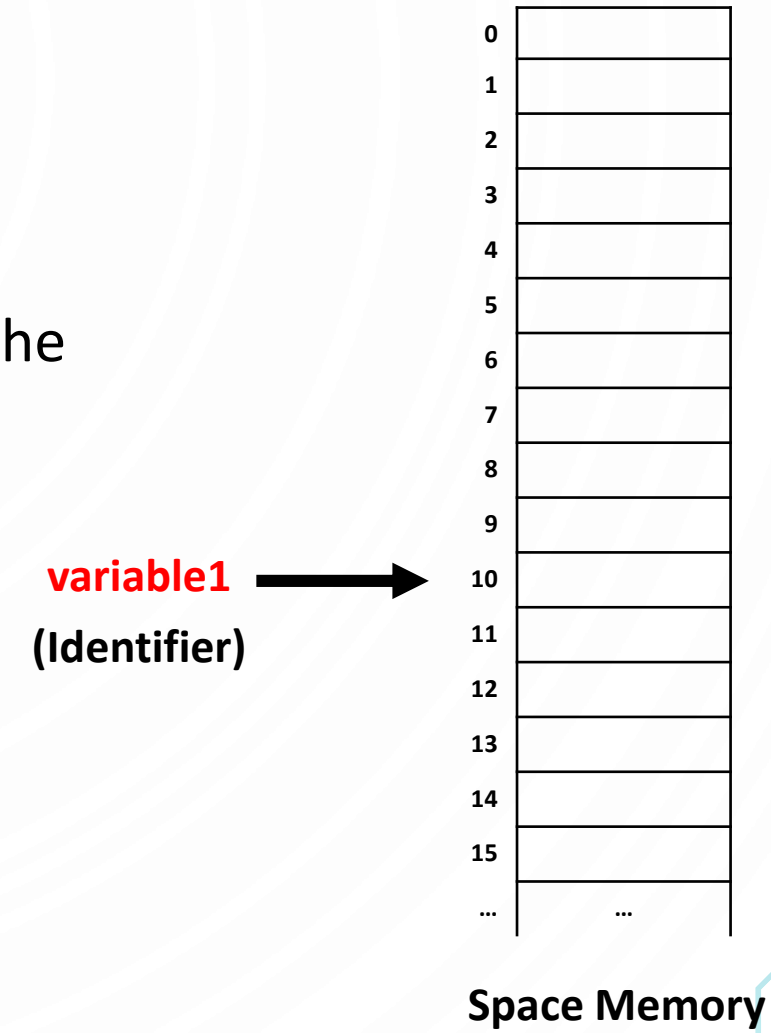
- The state of the space memory is the current value (data) stored in the space memory.
- The state of the space memory:
  - May be changed.
    - In this case the space memory is called **variable**.
  - Cannot be changed.
    - In this case the space memory is called **constant**.




**Space Memory**

# SPACE MEMORY IDENTIFIER

- **Identifier** is a sequence of characters that denotes the name of the space memory to be used
  - This name is unique within a program



# JAVA IDENTIFIERS

- Identifiers may contain only
  - Letters
  - Digits (0 through 9)
  - The underscore character (_)
  - And the dollar sign symbol ($) which has a special meaning (Avoid it)
- The first character cannot be a digit.

# JAVA IDENTIFIERS

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

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

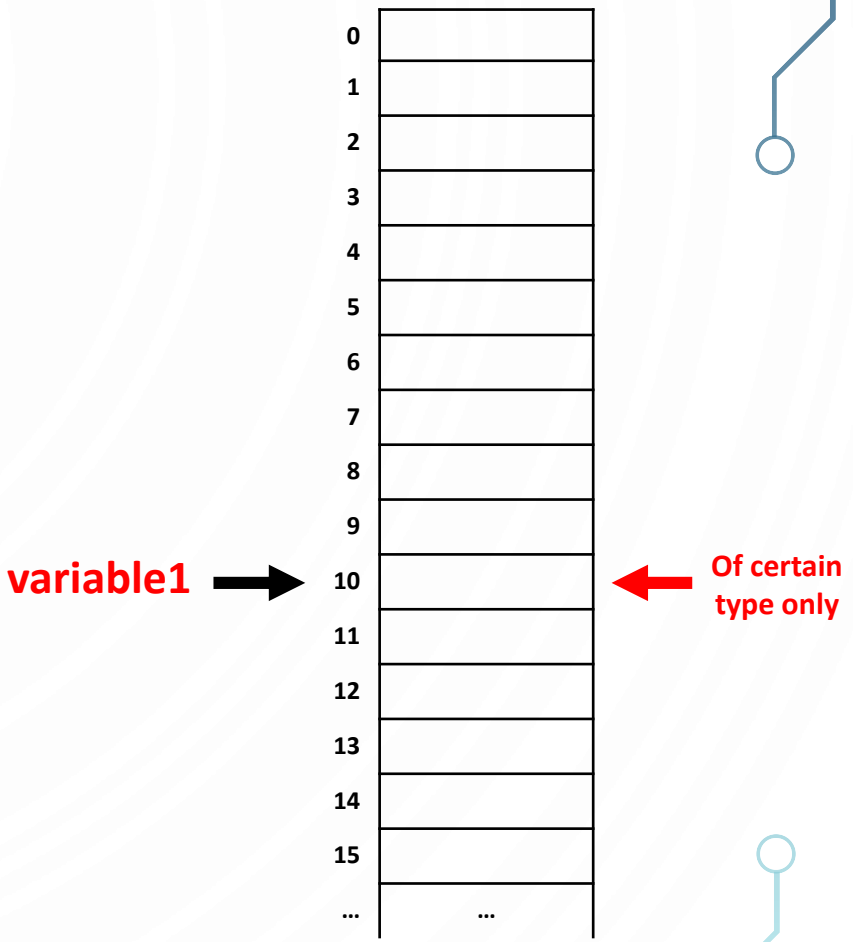
- Identifiers can be arbitrarily long.
- Since Java is *case sensitive*, `stuff`, `Stuff`, and `STUFF` are different identifiers.

# IDENTIFIER CONVENTIONS IN JAVA

- Constants
  - All uppercase, separating words within a multiword identifier with the underscore symbol, _
- Variables
  - All lowercase
  - Capitalizing the first letter of each word in a multiword identifier, except for the first word

# DATA TYPE

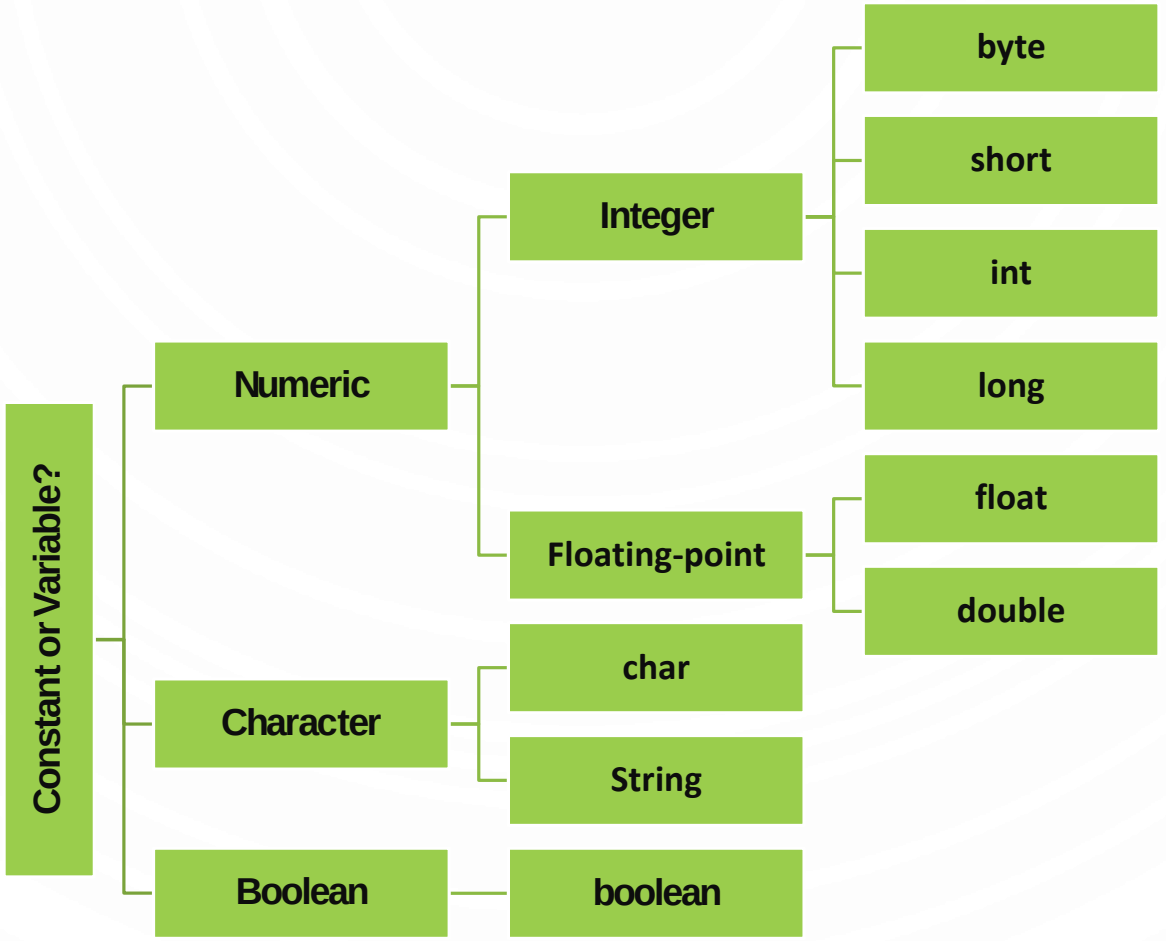
- The **data type** defines what kinds of values a space memory is allowed to store
- All values stored in the same space memory should be of the same data type
- All constants and variables used in a Java program must be defined prior to their use in the program



# 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.

# JAVA BUILT-IN DATA TYPES



# 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`)

# 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 |

# 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
```

# CONSTANT DECLARATION

```
final dataType constIdentifier = literal | expression;
```

```
final double PI                = 3.14159;
final int    MONTH_IN_YEAR     = 12;
final short  FARADAY_CONSTANT  = 23060;
```

The reservation word **final** is used to declare constants

**Constants/ Named constant**

Literals

```
final int MAX                  = 1024;
final int MIN                  = 128;
final int AVG                  = (MAX + MIN) / 2;
```

Expression

# 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.

# 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.

# VARIABLE DECLARATION

- A variable may be declared:
  - With initial value.
  - Without initial value.
- Variable declaration without initial value;

```
dataType    variableIdentifier;
double      avg;
int         i;
```

- Variable declaration with initial value;

```
datatype    variableIdentifier      = literal | expression;
double      avg                     = 0.0;
int         i                       = 1;
int         x                       =5, y = 7, z = (x+y)*3;
```

# 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;
```

# 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 Appendix 1 for a complete 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 */
   . . .
}
```

# ASSIGNMENT STATEMENTS

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

```
answer = 42;
```

- The "equal sign" is called the *assignment operator.*
- We say, "The variable named **`answer`** is assigned a value of 42," or more simply, "**`answer`** is assigned 42."

# ASSIGNMENT STATEMENTS

- Syntax

  `variable = expression`

  where `expression` can be:

  - another variable
  - a *literal* or *constant* (such as a number)
  - something more complicated which combines variables and literals using *operators* (such as + and -)

# ASSIGNMENT EXAMPLES

```
amount = 3.99;

firstInitial = 'W';

score = numberOfCards + handicap;

eggsPerBasket = eggsPerBasket - 2;
```

# 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';
```

# INITIALIZING VARIABLES

- syntax

```
type variable_1 = expression_1, variable_2 =
expression_2, …;
```

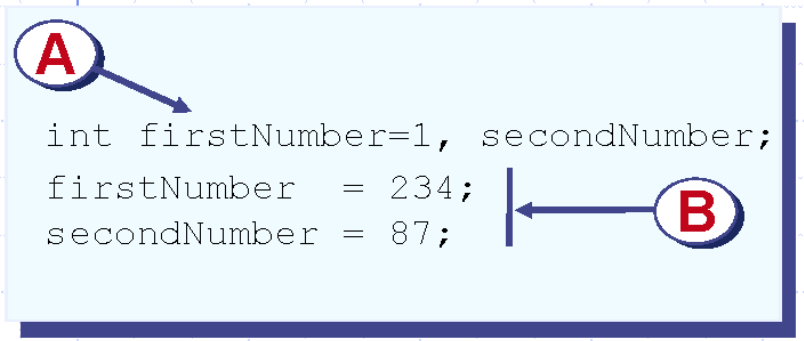
# ASSIGNMENT EVALUATION

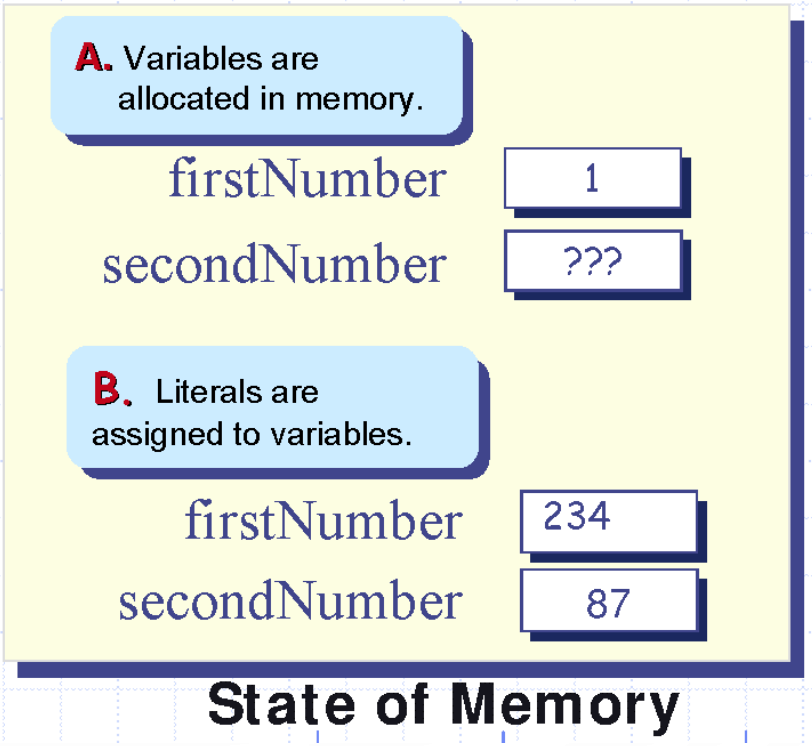
- The expression on the right-hand side of the assignment operator (=) is evaluated first.
- The result is used to set the value of the variable on the left-hand side of the assignment operator.

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

# ASSIGNING LITERALS

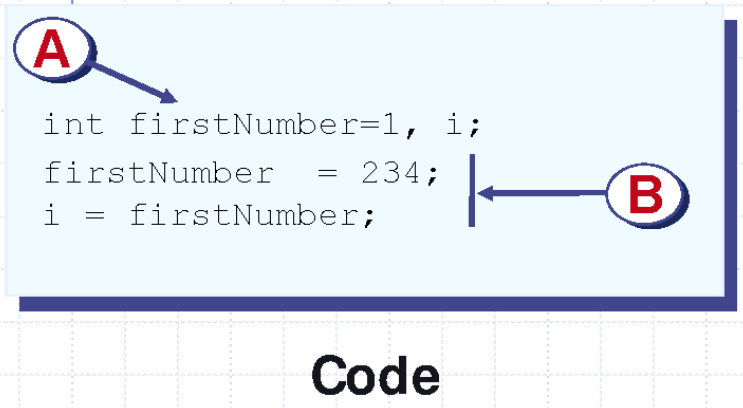
- In this case, the literal is stored in the space memory allocated for the variable at the left side.

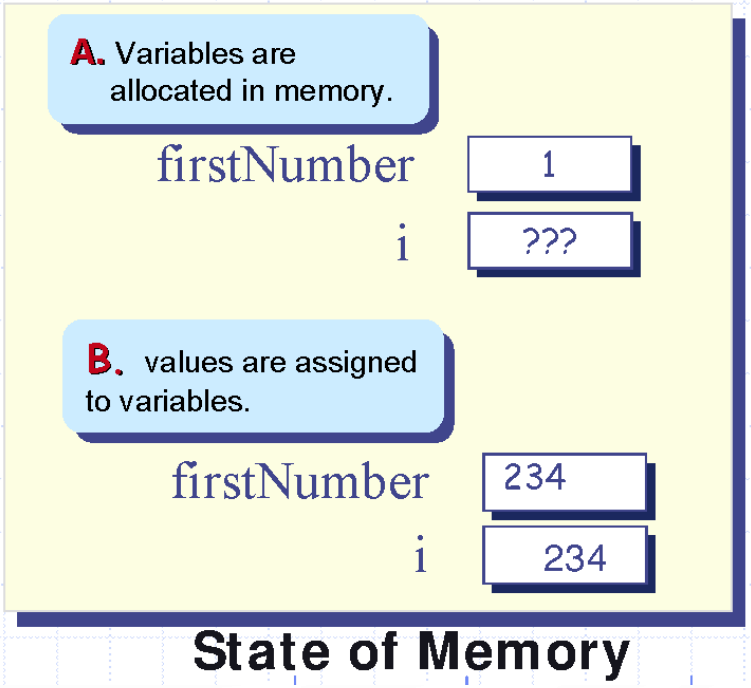


# ASSIGNING VARIABLES

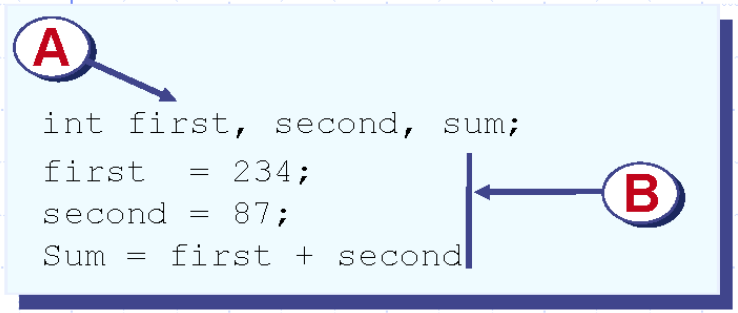
- In this case, the value of the variable at the right side is stored in the space memory allocated for the variable at the left side.

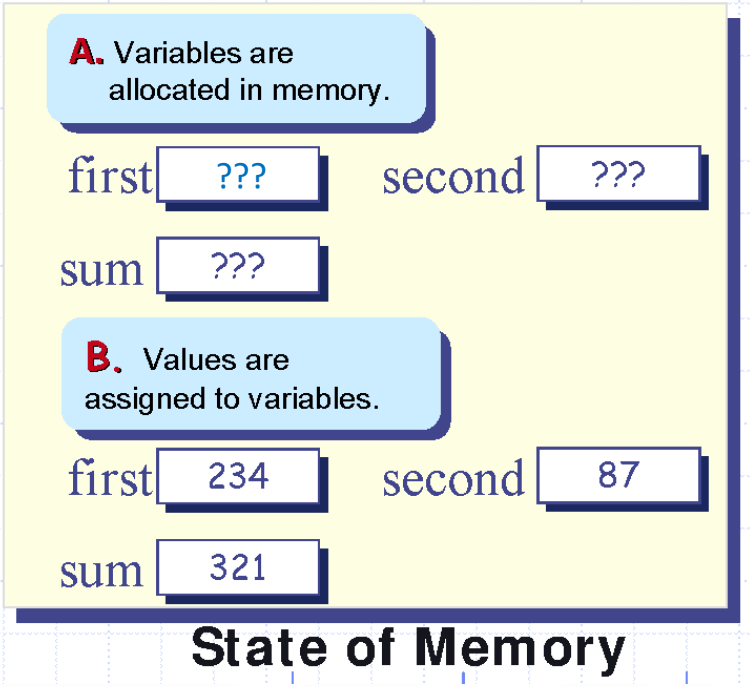


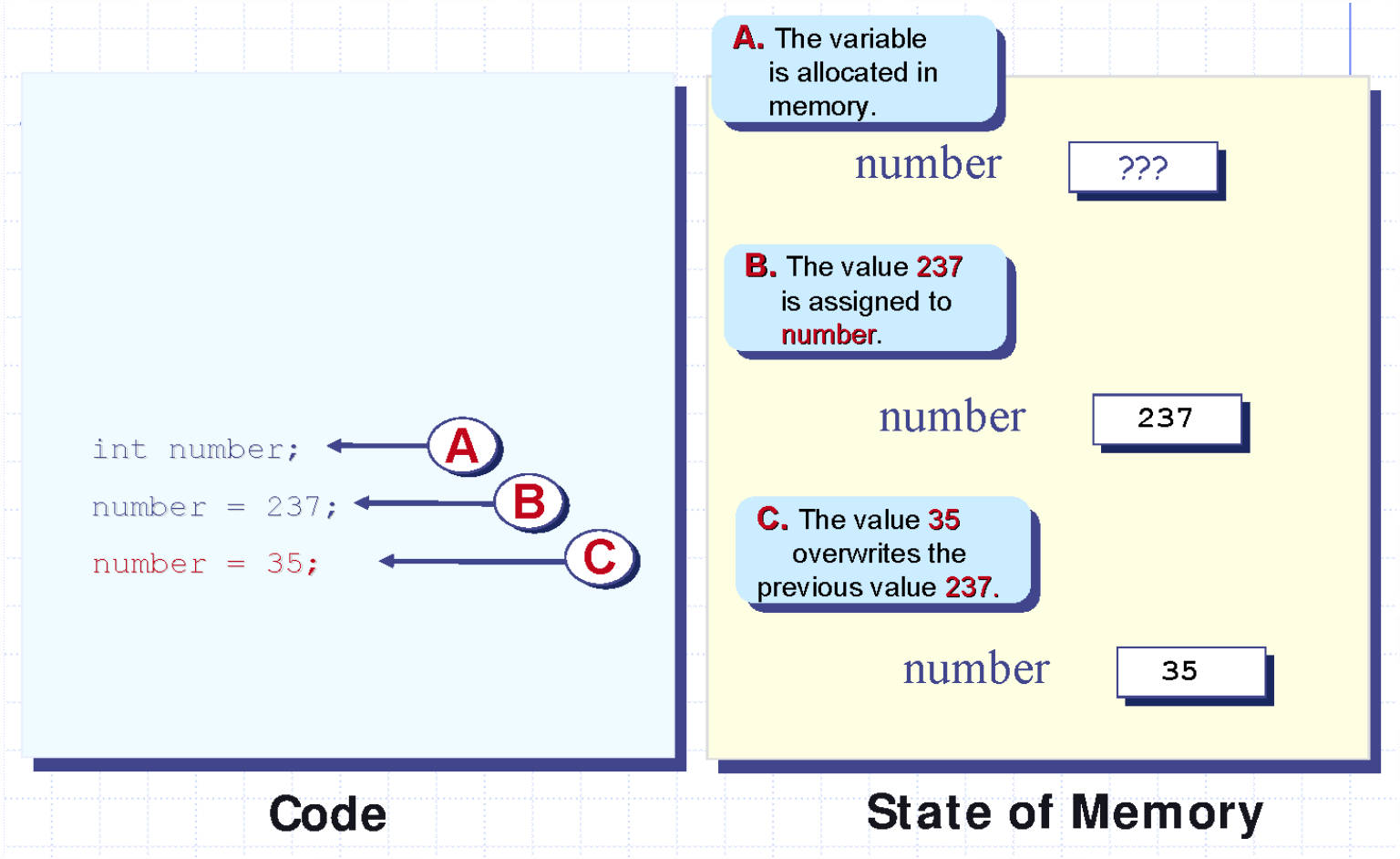
# ASSIGNING EXPRESSIONS

- In this case, the result of the evaluation of the expression is stored in the space memory allocated for variable at the left side.



# UPDATING DATA



# CONTENT

- Group of Operators
- Arithmetic Operators
- Assignment Operator
- Order of Precedence
- Increment/Decrement Operators
- Relational Operators
- Logical Operators

# OPERATORS

- **Operators** are special symbols used for:
  - mathematical functions
  - assignment statements
  - logical comparisons
- Examples of operators:
  - 3 + 5 // uses + operator
  - 14 + 5 - 4 * (5 - 3) // uses +, -, * operators
- **Expressions**: can be combinations of variables and operators that result in a value

# GROUPS OF OPERATORS

- Groups of operators:
  - Arithmetic Operators
  - Assignment Operator
  - Increment / Decrement Operators
  - Relational Operators
  - Logical Operators

# JAVA ARITHMETIC OPERATORS

- Addition +
- Subtraction -
- Multiplication *
- Division /
- Remainder (Modulus) %

# EXAMPLES

| Operation | Java Operator | Example | Value (x = 10, y = 7, z = 2.5) |
|---|---|---|---|
| Addition | + | x + y | 17 |
| Subtraction | - | x - y | 3 |
| Multiplication | * | x * y | 70 |
| Division | / | x / y | 1 |
| | | x / z | 4.0 |
| Modulo division (remainder) | % | x % y | 3 |

This is an integer division where the fractional part is truncated

# DIVISION EXAMPLE

- Example of division issues:

10 / 3 → 3

10.0 / 3 → 3.33333

- As we can see,
  - if we divide two integers we get an integer result.
  - if one or both operands is a floating-point value we get a floating-point result.

# MODULUS (REMAINDER)

- Generates the remainder when you divide two integer values.

5%3 → 2 ; 5%4 → 1 ; 5%5 → 0 ; 5%10→5

- Modulus operator is most commonly used with integer operands
- If we attempt to use the modulus operator on floating-point values we will garbage!

# ORDER OF PRECEDENCE

- ( ) evaluated first, inside-out
- *, /, or % evaluated second, left-to-right
- +, - evaluated last, left-to-right

# BASIC ASSIGNMENT OPERATOR

- We assign a value to a variable using the basic assignment operator (=)
- Assignment operator stores a value in memory
- The syntax is

```
leftSide = rightSide ;
```

- Examples:

```
i = 1;
start = i;
sum = firstNumber + secondNumber;
avg = (one + two + three) / 3;
```

# THE RIGHT SIDE OF THE ASSIGNMENT OPERATOR

- The Java assignment operator assigns the value on the right side of the operator to the variable appearing on the left side of the operator.
- The right side may be either:
  - Literal
    - e.g. `i = 1;`
  - Variable identifier
    - e.g. `start = i;`
  - Expression
    - e.g. `sum = first + second;`

# SUM PROGRAM

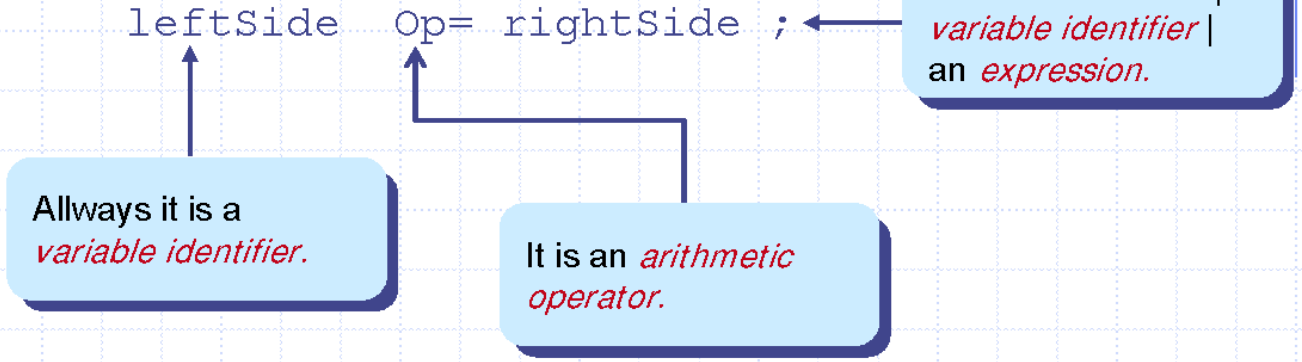
```
public class SumProgram {

    public static void main( String[] args ){

        int num1 ;
        int num2 ;
        int sum ;

        num1 = 5 ;
        num2 = 3 ;

        sum = num1 + num2 ;

        System.out.println(num1 + " + " + num2 + " = " + sum);
    }
}
```

# ARITHMETIC & ASSIGNMENT OPERATORS

- Java allows combining arithmetic and assignment operators into a single operator:
  - Addition & assignment +=
  - Subtraction & assignment -=
  - Multiplication & assignment *=
  - Division & assignment /=
  - Remainder & assignment %=

# ARITHMETIC & ASSIGNMENT OPERATORS

- The syntax is




- This is equivalent to:

```
leftSide = leftSide Op rightSide ;
```

- x%=5; ⇔ x = x % 5;
- x*=y+w*z; ⇔ x = x*(y+w*z);

# INCREMENT/DECREMENT OPERATORS

- Only use ++ or -- when a variable is being incremented/decremented as a statement by itself

`x++;` is equivalent to `x = x+1;`

`x--;` is equivalent to `x = x-1;`

# 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 += 1;
```

- equivalent operations

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

# INCREMENT AND DECREMENT OPERATORS IN EXPRESSIONS

- This expression:

```
int sum = 5 + a++ ;
```

Is equivalent to:

```
int sum = 5 + a ;
a = a + 1 ;
```

- This expression:

```
int sum= 5 + ++a ;
```

Is equivalent to:

```
a = a + 1 ;
int sum = 5 + a ;
```

# CHALLENGE

```
//What does this program print?


public class Challenge {
    public static void main(String args[]) {
        int x=10;
        int y= x +++ (-x);
        System.out.println("x= " + x);
        System.out.println("y= " + y);
    }
}
```

# RELATIONAL OPERATORS

- Relational operators compare two values
- They produce a boolean value (true or false) depending on the relationship

| Operation | Is true when |
|---|---|
| a >b | a is greater than b |
| a >=b | a is greater than or equal to b |
| a ==b | a is equal to b |
| a !=b | a is not equal to b |
| a <=b | a is less than or equal to b |
| a <b | a is less than b |

# EXAMPLE

```
int x = 3;

int y = 5;

boolean result;

result = (x > y);
```

- result is assigned the value false because 3 is not greater than 5

# LOGICAL OPERATORS

| Symbol | Name |
|---|---|
| && | AND |
| \|\| | OR |
| ! | NOT |

| && | True | False |
|---|---|---|
| True | True | False |
| False | False | False |

| \|\| | True | False |
|---|---|---|
| True | True | True |
| False | True | False |

# EXAMPLE

```
boolean x = true;
boolean y = false;
boolean result;

result = (x && y);          //result is assigned the value false

result = ((x || y) && x);   //(x || y) evaluates to true
                            //(true && x) evaluates to true
                            //result is then assigned the value true
```

# OPERATORS PRECEDENCE

| | |
|---|---|
| Parentheses | (), inside-out |
| Increment/decrement | ++, --, from left to right |
| Multiplicative | *, /, %, from left to right |
| Additive | +, -, from left to right |
| Relational | <, >, <=, >=, from left to right |
| Equality | ==, !=, from left to right |
| Logical AND | && |
| Logical OR | \|\| |
| Assignment | =, +=, -=, *=, /=, %= |

# CONTENT

- Type Casting
- Input & Output
- Vending Machine Change Example
- The Class `String`
- Documentation and Style

# ASSIGNMENT COMPATIBILITIES

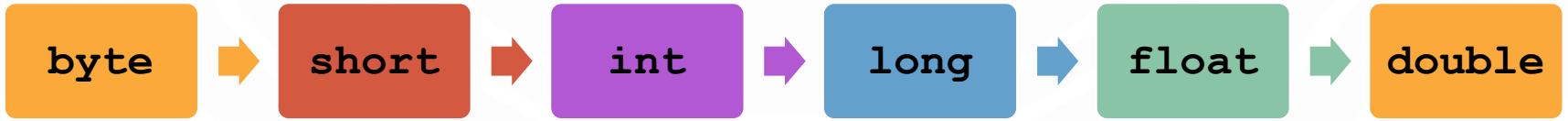
- Java is said to be *strongly typed.*
  - You can't, for example, assign a floating point value to a variable declared to store an integer.
- Sometimes conversions between numbers are possible.

```
doubleVariable = 7;
```

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

# ASSIGNMENT COMPATIBILITIES

- A value of one type can be assigned to a variable of any type further to the right




  - But not to a variable of any type further to the left.
- You can assign a value of type char to a variable of type `int`.

# TYPE CASTING

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

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

- Illegal without `(int)`

# TYPE CASTING

- The value of `(int)distance` is `9`
- The value of `distance`, both before and after the cast, is `9.0`.
- Any nonzero value to the right of the decimal point is *truncated* rather than *rounded.*

# RECTANGLE AREA CALCULATION PROGRAM

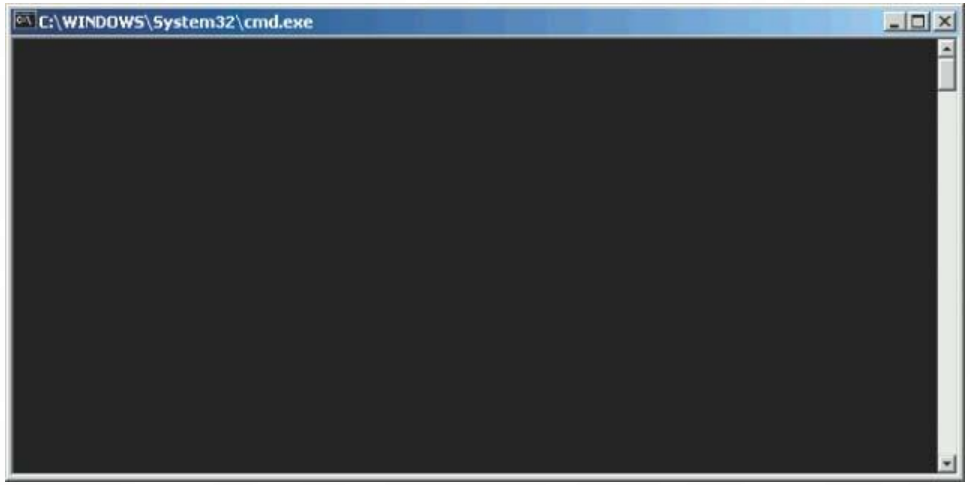
```
public class RecArea {

    public static void main(String args[]) {

        double length=2,width=4,area=0;

        area= length * width;

        System.out.println("The area of this rectangle is: " + area );
    }
}
```

# RECTANGLE AREA CALCULATION PROGRAM

```
import java.util.Scanner;

public class RecArea {
    public static void main(String args[]) {

        double length=0,width=0,area=0;
        Scanner input= new Scanner(System.in);

        System.out.println("Enter the length of the rectangle:");
        length= input.nextDouble();

        System.out.println("Enter the width of the rectangle:");
        width= input.nextDouble();

        area= length * width;

        System.out.println("The area of this rectangle is: " + area );
    }
}
```

# STANDARD OUTPUT WINDOW

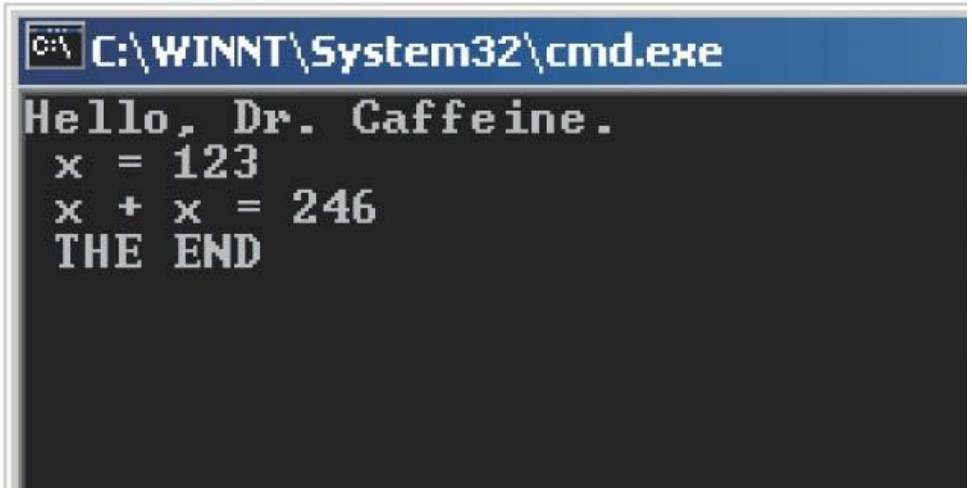
- Using System.out, we can output multiple lines of text to the standard output window.
- The exact style of standard output window depends on the Java tool you use.



# THE PRINTLN METHOD

- The function print is used to print text to the standard output window
- The function println skips a line and prints the text to the standard output window

```
int x = 123, y = x + x;
System.out.println( "Hello, Dr. Caffeine." );
System.out.print( " x = " );
System.out.println( x );
System.out.print( " x + x = " );
System.out.println( y );
System.out.println( " THE END" );
```



# SIMPLE INPUT

- Sometimes the data needed for a computation are obtained from the user at run time.
- Keyboard input requires

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

at the beginning of the file.

# SIMPLE INPUT

- Data can be entered from the keyboard using

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

followed, for example, by

```
anotherVariable = scannerVariable.nextInt();
```

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

# COMMON SCANNER METHODS

| Method | Example |
|---|---|
| nextByte( ) | byte b = input.nextByte( ); |
| nextShort( ) | short s = input.nextShort( ); |
| nextInt( ) | int i = input.nextInt( ); |
| nextLong( ) | long l = input.nextLong( ); |
| nextFloat( ) | float f = input.nextFloat( ); |
| nextDouble( ) | double d = input.nextDouble( ); |
| next() | String str = input.next(); |
| nextLine() | String str = input.nextLine(); |

# VENDING MACHINE CHANGE EXAMPLE

```java
import java.util.Scanner;

public class VendingMachineChange {
    public static void main(String args[]) {

        int n500=0,n100=0,n50=0,n10=0,n5=0;
        int amount=0;
        Scanner input= new Scanner(System.in);

        System.out.println("Enter the amount to be changed:");
        amount= input.nextInt();

        n500= amount/500;
        amount = amount % 500;
        System.out.println("The number of 500's is: "+ n500);

        n100= amount/100;
        amount = amount % 100;
        System.out.println("The number of 100's is: "+ n100);

        n50= amount/50;
        amount = amount % 50;
        System.out.println("The number of 50's is: "+ n50);

        n10= amount/10;
        amount = amount % 10;
        System.out.println("The number of 10's is: "+ n10);

        n5= amount/5;
        amount = amount % 5;
        System.out.println("The number of 5's is: "+ n5);

        System.out.println("The number of 1's is: "+ amount);
    }
}
```

```java
import java.util.Scanner;

public class VendingMachineChange2 {
    public static void main(String args[]) {

        int amount=0;
        Scanner input= new Scanner(System.in);

        System.out.println("Enter the amount to be changed:");
        amount= input.nextInt();

        System.out.println("The number of 500's is: "+ amount/500);

        amount %= 500;
        System.out.println("The number of 100's is: "+ amount/100);

        amount %= 100;
        System.out.println("The number of 50's is: "+ amount/50);

        amount %= 50;
        System.out.println("The number of 10's is: "+ amount/10);

        amount %= 10;
        System.out.println("The number of 5's is: "+ amount/5);

        amount %= 5;
        System.out.println("The number of 1's is: "+ amount);
    }
}
```

# 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.

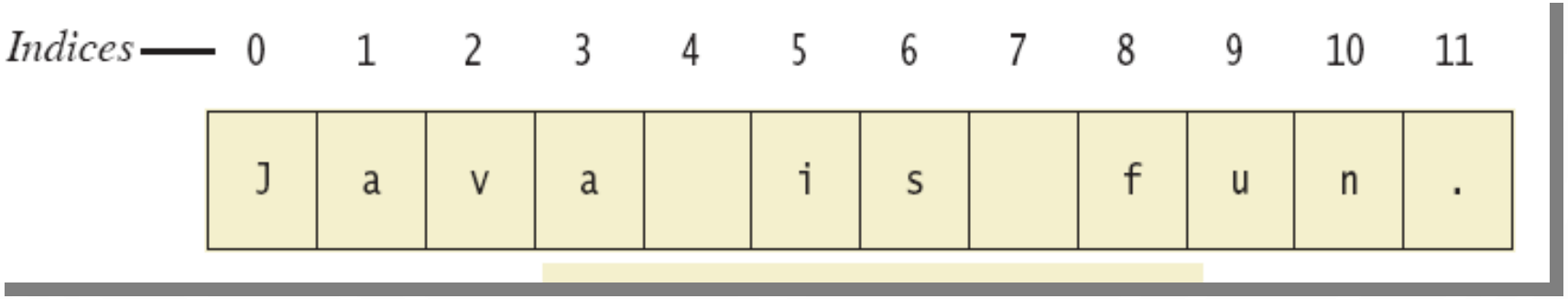
```
String greeting = "Hello";
int n = greeting.length();
```

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

# `String` Indices




- 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.

**FIGURE 2.5 Some Methods in the Class `String`**

| Method | Return Type | Example for `String s = "Java";` | Description |
|---|---|---|---|
| `charAt` (*index*) | `char` | `c = s.charAt(2);` `// c='v'` | Returns the character at *index* in the string. Index numbers begin at 0. |
| `compareTo` (a_*string*) | `int` | `i = s.compareTo("C++");` `// i is positive` | Compares this string with *a_string* to see which comes first in lexicographic (alphabetic, with upper before lower case) ordering. Returns a negative integer if this string is first, zero if the two strings are equal, and a positive integer if *a_string* is first. |
| `concat` (a_*string*) | `String` | `s2 = s.concat("rocks");` `// s2 = "Javarocks"` | Returns a new string with this string concatenated with *a_string*. You can use the + operator instead. |
| `equals` (a_*string*) | `boolean` | `b = s.equals("Java");` `// b = true` | Returns true if this string and *a_string* are equal. Otherwise returns false. |
| `equals IgnoreCase` (a_*string*) | `boolean` | `b = s.equals("Java");` `// b = true` | Returns true if this string and *a_string* are equal, considering upper and lower case versions of a letter to be the same. Otherwise returns false. |
| `indexOf` (a_*string*) | `int` | `i = s.indexOf("va");` `// i = 2` | Returns the index of the first occurrence of the substring *a_string* within this string or -1 if *a_string* is not found. Index numbers begin at 0. |

| | | | |
|---|---|---|---|
| `lastIndexOf` (a_*string*) | `int` | `i = s.lastIndexOf("a");`<br>`// i = 3` | Returns the index of the last occurrence of the substring *a_string* within this string or -1 if *a_string* is not found. Index numbers begin at 0. |
| `length()` | `int` | `i = s.length();`<br>`// i = 4` | Returns the length of this string. |
| `toLower Case()` | `String` | `s2 = s.toLowerCase();`<br>`// s = "java"` | Returns a new string having the same characters as this string, but with any uppercase letters converted to lowercase. This string is unchanged. |
| `toUpper Case()` | `String` | `s2 = s.toUpperCase();`<br>`// s2 = "JAVA"` | Returns a new string having the same characters as this string, but with any lowercase letters converted to uppercase. This string is unchanged. |
| `replace` (*oldchar, newchar)* | `String` | `s2 = s.replace('a','o');`<br>`// s2 = "Jovo";` | Returns a new string having the same characters as this string, but with each occurrence of *oldchar* replaced by *newchar*. |
| `substring` (*start)* | `String` | `s2 = s.substring(2);`<br>`// s2 = "va";` | Returns a new string having the same characters as the substring that begins at index *start* through to the end of the string. Index numbers begin at 0. |
| `substring` (*start,end*) | `String` | `s2 = s.substring(1,3);`<br>`// s2 = "av";` | Returns a new string having the same characters as the substring that begins at index *start* through to but not including the character at index *end*. Index numbers begin at 0. |
| `trim( )` | `String` | `s = "  Java  ";`<br>`s2 = s.trim();`<br>`// s2 = "Java"` | Returns a new string having the same characters as this string, but with leading and trailing whitespace removed. |

# 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

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

| | |
|---|---|
| `\"` | 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. |

# 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 the ASCII character set.
- The Unicode character set includes characters from many different alphabets (but you probably won't use them).

# 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 = "";
```

# 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

# 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`).
  - 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 explain the program.
  - They are useful to the programmer, but they are ignored by the compiler.

# INDENTATION

- Indentation should communicate nesting clearly.
- A good choice is four spaces for each level of indentation.
- Indentation should be consistent.
- Indentation should be used for second and subsequent lines of statements which do not fit on a single line.

# INDENTATION

- Indentation does not change the behavior of the program
- Proper indentation helps communicate to the human reader the nested structures of the program

# 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.

# 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 than another literal with the same value might be changed unintentionally.