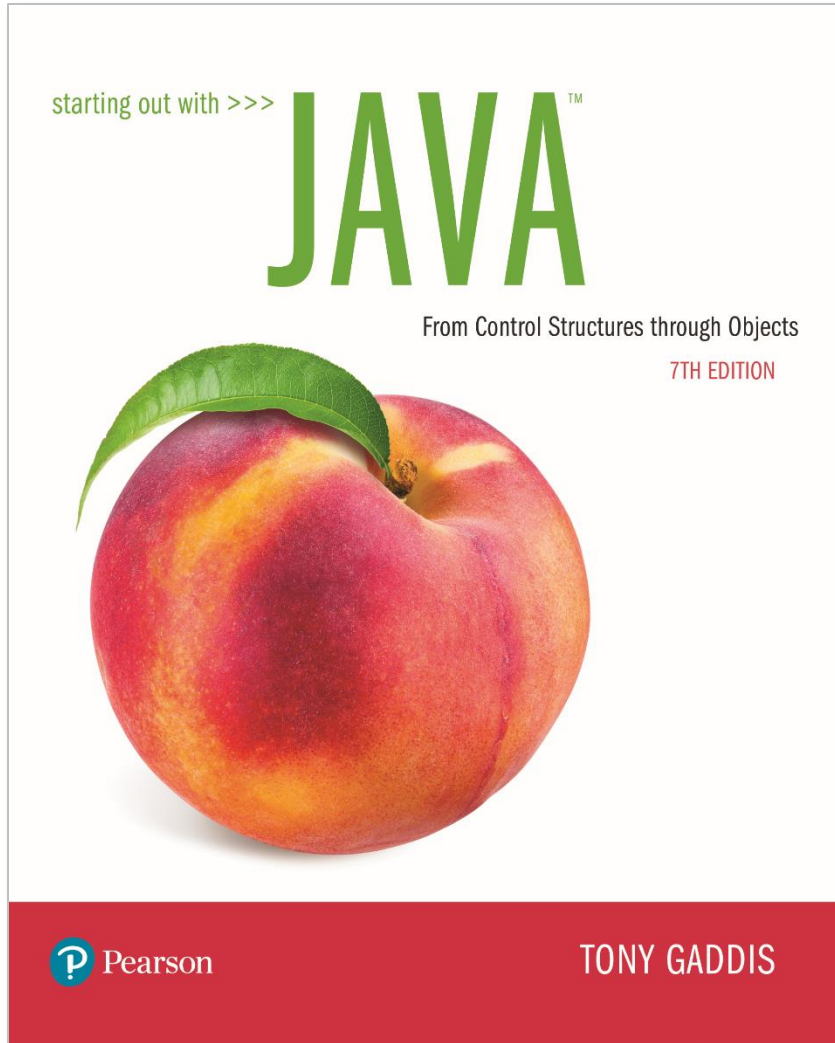


# STARTING OUT WITH JAVA™

7<sup>th</sup> Edition



## Chapter 2

### Java Fundamentals

# Parts of a Java Program (1 of 2)

- A Java source code file contains one or more Java classes.
- If more than one class is in a source code file, only one of them may be public.
- The public class and the filename of the source code file must match.

ex: A class named *Simple* must be in a file named *Simple.java*

- Each Java class can be separated into parts.

# Parts of a Java Program (2 of 2)

- See example: [Simple.java](#)
- To compile the example:
  - `javac Simple.java`
    - Notice the `.java` file extension is needed.
    - This will result in a file named *Simple.class* being created.
- To run the example:
  - `java Simple`
    - Notice there is no file extension here.
    - The *java* command assumes the extension is `.class`.

# Analyzing The Example (1 of 3)

```
// This is a simple Java program.
```

**This is a Java comment. It is ignored by the compiler.**

```
public class Simple
```

**This is the class header for the class Simple**

```
{
```

**This area is the body of the class Simple. All of the data and methods for this class will be between these curly braces.**

```
}
```

# Analyzing The Example (2 of 3)

```
// This is a simple Java program.
```

```
public class Simple  
{
```

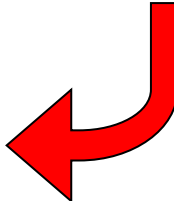
```
    public static void main(String [] args)
```

```
    {
```


```
    }
```

```
}
```

**This is the method header for the main method. The main method is where a Java application begins.**



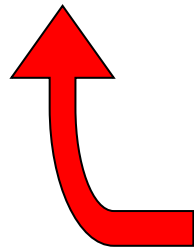
**This area is the body of the main method. All of the actions to be completed during the main method will be between these curly braces.**



# Analyzing The Example (3 of 3)

// This is a simple Java program.

```
public class Simple
{
    public static void main(String [] args)
    {
        System.out.println("Programming is great fun!");
    }
}
```



**This is the Java Statement that  
is executed when the program runs.**

# Parts of a Java Program (1 of 2)

- Comments
  - The line is ignored by the compiler.
  - The comment in the example is a single-line comment.
- Class Header
  - The class header tells the compiler things about the class such as what other classes can use it (**public**) and that it is a Java class (**class**), and the name of that class (**Simple**).
- Curly Braces
  - When associated with the class header, they define the scope of the class.
  - When associated with a method, they define the scope of the method.

# Parts of a Java Program (2 of 2)

- The `main` Method
  - This line must be exactly as shown in the example (except the *args* variable name can be programmer defined).
  - This is the line of code that the *java* command will run first.
  - This method starts the Java program.
  - Every Java application must have a `main` method.
- Java Statements
  - When the program runs, the statements within the `main` method will be executed.
  - Can you see what the line in the example will do?



# Java Statements (1 of 2)

- If we look back at the previous example, we can see that there is only one line that ends with a semi-colon.

```
System.out.println("Programming is  
great fun!");
```

- This is because it is the only Java statement in the program.
- The rest of the code is either a comment or other Java framework code.

# Java Statements (2 of 2)

- Comments are ignored by the Java compiler so they need no semi-colons.
- Other Java code elements that do not need semi colons include:
  - class headers
    - Terminated by the code within its curly braces.
  - method headers
    - Terminated by the code within its curly braces.
  - curly braces
    - Part of framework code that needs no semi-colon termination.

# Short Review (1 of 2)

- Java is a case-sensitive language.
- All Java programs must be stored in a file with a .java file extension.
- Comments are ignored by the compiler.
- A .java file may contain many classes but may only have one public class.
- If a .java file has a public class, the class must have the same name as the file.

## Short Review (2 of 2)

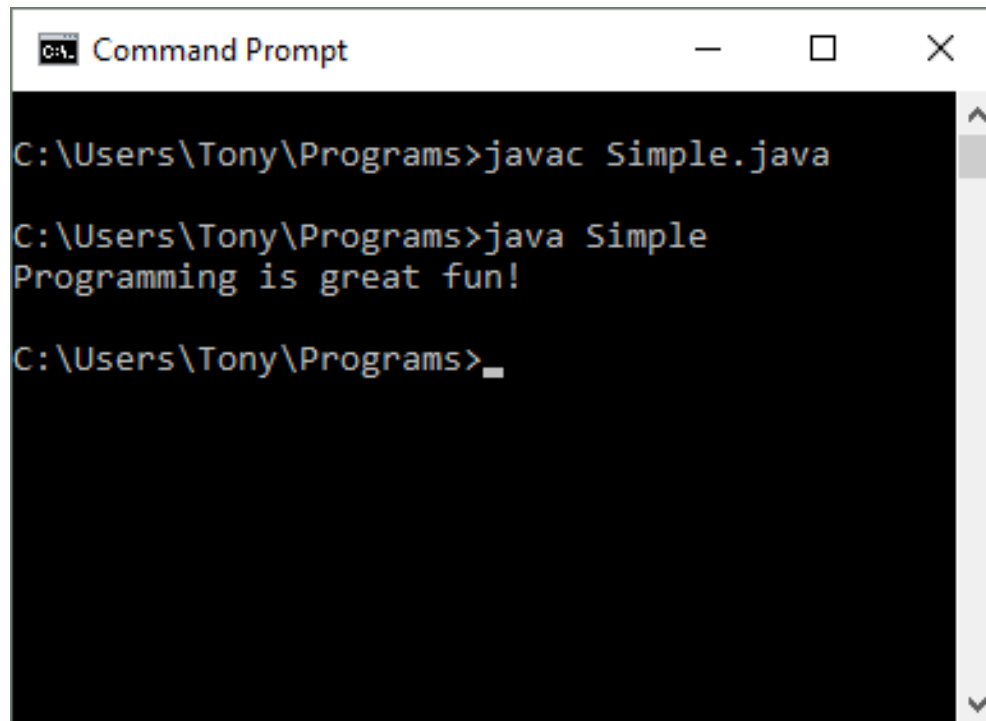
- Java applications must have a `main` method.
- For every left brace, or opening brace, there must be a corresponding right brace, or closing brace.
- Statements are terminated with semicolons.
  - Comments, class headers, method headers, and braces are not considered Java statements.

# Special Characters

//	double slash	Marks the beginning of a single line comment.
( )	open and close parenthesis	Used in a method header to mark the <i>parameter list</i> .
{ }	open and close curly braces	Encloses a group of statements, such as the contents of a class or a method.
" "	quotation marks	Encloses a string of characters, such as a message that is to be printed on the screen
;	semi-colon	Marks the end of a complete programming statement

# Console Output (1 of 8)

- Many of the programs that you will write will run in a console window.



```
C:\Users\Tony\Programs>javac Simple.java  
  
C:\Users\Tony\Programs>java Simple  
Programming is great fun!  
  
C:\Users\Tony\Programs>_
```

## Console Output (2 of 8)

- The console window that starts a Java application is typically known as the *standard output* device.
- The *standard input* device is typically the keyboard.
- Java sends information to the standard output device by using a Java class stored in the standard Java library.

## Console Output (3 of 8)

- Java classes in the standard Java library are accessed using the Java Applications Programming Interface (API).
- The standard Java library is commonly referred to as the *Java API*.



## Console Output (4 of 8)

- The previous example uses the line:  

```
System.out.println("Programming is  
great fun!");
```
- This line uses the `System` class from the standard Java library.
- The `System` class contains methods and objects that perform system level tasks.
- The `out` object, a member of the `System` class, contains the methods `print` and `println`.

## Console Output (5 of 8)

- The `print` and `println` methods actually perform the task of sending characters to the output device.
- The line:  

```
System.out.println("Programming is  
great fun!");
```

  
is pronounced: System dot out dot println ...
- The value inside the parenthesis will be sent to the output device (in this case, a string).

## Console Output (6 of 8)

- The `println` method places a newline character at the end of whatever is being printed out.
- The following lines:

```
System.out.println("This is being  
printed out");
```

```
System.out.println("on two separate  
lines.");
```

Would be printed out on separate lines since the first statement sends a newline command to the screen.

# Console Output (7 of 8)

- The `print` statement works very similarly to the `println` statement.
- However, the `print` statement does not put a newline character at the end of the output.
- The lines:

```
System.out.print("These lines will be");  
System.out.print("printed on");  
System.out.println("the same line.");
```

Will output:

These lines will beprinted onthe same line.

Notice the odd spacing? Why are some words run together?

# Console Output (8 of 8)

- For all of the previous examples, we have been printing out strings of characters.
- Later, we will see that much more can be printed.
- There are some special characters that can be put into the output.

```
System.out.print("This line will have a  
newline at the end.\n");
```

- The `\n` in the string is an escape sequence that represents the newline character.
- Escape sequences allow the programmer to print characters that otherwise would be unprintable.

# Java Escape Sequences (1 of 2)

<code>\n</code>	newline	Advances the cursor to the next line for subsequent printing
<code>\t</code>	tab	Causes the cursor to skip over to the next tab stop
<code>\b</code>	backspace	Causes the cursor to back up, or move left, one position
<code>\r</code>	carriage return	Causes the cursor to go to the beginning of the current line, not the next line
<code>\\</code>	backslash	Causes a backslash to be printed
<code>\'</code>	single quote	Causes a single quotation mark to be printed
<code>\"</code>	double quote	Causes a double quotation mark to be printed

# Java Escape Sequences (2 of 2)

- Even though the escape sequences are comprised of two characters, they are treated by the compiler as a single character.

```
System.out.print("These are our top sellers:\n");  
System.out.print("\tComputer games\n\tCoffee\n ");  
System.out.println("\tAspirin");
```

Would result in the following output:

```
These are our top seller:  
  
    Computer games  
  
    Coffee  
  
    Asprin
```

- With these escape sequences, complex text output can be achieved.

# Variables and Literals (1 of 2)

- A variable is a named storage location in the computer's memory.
- A literal is a value that is written into the code of a program.
- Programmers determine the number and type of variables a program will need.
- See example: [Variable.java](#)

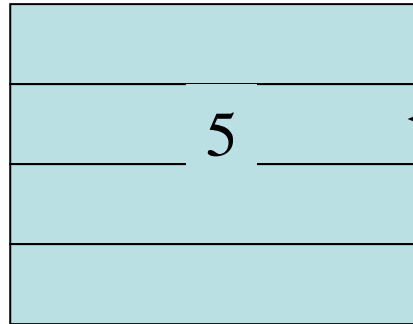


# Variables and Literals (2 of 2)

This line is called  
a *variable declaration*.  
`int value;`

The following line is known  
as an assignment statement.  
`value = 5;`

**0x000**  
**0x001**  
**0x002**  
**0x003**



The value 5  
is stored in  
memory.

This is a string *literal*. It will be printed as is.

```
System.out.print("The value is ");  
System.out.println(value);
```

The integer 5 will  
be printed out here.  
*Notice no quote marks?*

# The + Operator

- The + operator can be used in two ways.
  - as a concatenation operator
  - as an addition operator
- If either side of the + operator is a string, the result will be a string.

```
System.out.println("Hello " + "World");  
System.out.println("The value is: " + 5);  
System.out.println("The value is: " +  
value);  
System.out.println("The value is: " + '/n'  
+ 5);
```

# String Concatenation (1 of 3)

- Java commands that have string literals must be treated with care.
- A string literal value cannot span lines in a Java source code file.

```
System.out.println("This line is too  
long and now it has spanned more than  
one line, which will cause a syntax  
error to be generated by the compiler.  
");
```

# String Concatenation (2 of 3)

- The String concatenation operator can be used to fix this problem.

```
System.out.println("These lines are " +  
                    "are now ok and will not " +  
                    "cause the error as before.");
```

- String concatenation can join various data types.

```
System.out.println("We can join a string to " +  
                    "a number like this: " + 5);
```

# String Concatenation (3 of 3)

- The Concatenation operator can be used to format complex String objects.

```
System.out.println("The following will be printed " +  
    "in a tabbed format: " +  
    "\n\tFirst = " + 5 * 6 + ", " +  
    "\n\tSecond = " + (6 + 4) + ", " +  
    "\n\tThird = " + 16.7 + ".");
```

- Notice that if an addition operation is also needed, it must be put in parenthesis.

# Identifiers (1 of 2)

- Identifiers are programmer-defined names for:
  - classes
  - variables
  - methods
- Identifiers may not be any of the Java reserved keywords.

# Identifiers (2 of 2)

- Identifiers must follow certain rules:
  - An identifier may only contain:
    - letters a–z or A–Z,
    - the digits 0–9,
    - underscores (\_), or
    - the dollar sign (\$)
  - The first character may not be a digit.
  - Identifiers are case sensitive.
    - `itemsOrdered` is not the same as `itemsordered`.
  - Identifiers cannot include spaces.

# Java Reserved Keywords

abstract	double	instanceof	static
assert	else	int	strictfp
boolean	enum	interface	super
break	extends	long	switch
byte	false	native	synchronized
case	for	new	this
catch	final	null	throw
char	finally	package	throws
class	float	private	transient
const	goto	protected	true
continue	if	public	try
default	implements	return	void
do	import	short	volatile
			while



# Variable Names

- Variable names should be descriptive.
- Descriptive names allow the code to be more readable; therefore, the code is more maintainable.
- Which of the following is more descriptive?  

```
double tr = 0.0725;
```

```
double salesTaxRate = 0.0725;
```
- Java programs should be *self-documenting*.

# Java Naming Conventions

- Variable names should begin with a lower case letter and then switch to title case thereafter:

Ex: `int caTaxRate`

- Class names should be all title case.

Ex: `public class BigLittle`

- More Java naming conventions can be found at:  
<http://java.sun.com/docs/codeconv/html/CodeConventions.doc8.html>
- A general rule of thumb about naming variables and classes are that, with some exceptions, their names tend to be nouns or noun phrases.

# Primitive Data Types

- Primitive data types are built into the Java language and are not derived from classes.
- There are 8 Java primitive data types.

byte  
short  
int  
long

# Numeric Data Types

byte	1 byte	Integers in the range -128 to +127 (why not 255?)
short	2 bytes	Integers in the range of -32,768 to +32,767
int	4 bytes	Integers in the range of -2,147,483,648 to +2,147,483,647
long	8 bytes	Integers in the range of -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
float	4 bytes	Floating-point numbers in the range of $\pm 3.410 \times 10^{-38}$ to $\pm 3.41038$ , with 7 digits of accuracy
double	8 bytes	Floating-point numbers in the range of $\pm 1.710 \times 10^{-308}$ to $\pm 1.710308$ , with 15 digits of accuracy

# Signed and Unsigned: short and char

- short and char are closely related, as both are stored as integral types with the same 16-bit length
- The primary difference is that short is *signed*, which means it splits its range across the positive and negative integers. Alternatively, char is *unsigned*

```
short bird = 'd';  
char mammal = (short)83;  
System.out.println(bird); // Prints 100  
System.out.println(mammal); // Prints S  
short reptile = 65535; // DOES NOT COMPILE  
char fish = (short)-1; // DOES NOT COMPILE  
System.out.println((char)-100); //???
```

# Writing Literals

- Java assumes you are defining an int value with a numeric literal.

`long max = 3123456789; // DOES NOT COMPILE`

- **Octal** (digits 0–7), which uses the number 0 as a prefix—for example, 017
- **Hexadecimal** (digits 0–9 and letters A–F/a–f), which uses 0x or 0X as a prefix—for example, 0xFF, 0xff, 0XFf. Hexadecimal is case insensitive so all of these examples mean the same value.
- **Binary** (digits 0–1), which uses the number 0 followed by b or B as a prefix—for example, 0b10, 0B10

# Writing Literals

## Literals and the Underscore Character

- Can add underscores anywhere **except** at the beginning of a literal, the end of a literal, right before a decimal point, or right after a decimal point.
- Can place multiple underscore characters next to each other

```
double notAtStart = _1000.00;      // DOES NOT COMPILE
double notAtEnd = 1000.00_;        // DOES NOT COMPILE
double notByDecimal = 1000_.00;    // DOES NOT COMPILE
int niceLook = 1_000_123_000;
double annoyingButLegal = 1_00_0.0_0; // Ugly, but compiles
double reallyUgly = 1_____2;    // Also compiles
```

# Variable Declarations

- Variable Declarations take the following form:
  - *Data Type VariableName;*
    - `byte inches;`
    - `short month;`
    - `int speed;`
    - `long timeStamp;`
    - `float salesCommission;`
    - `double distance;`



# Declaring Multiple Variables

Which of the following are legal declarations:

1. `boolean b1, b2;`
2. `String s1 = "1", s2;`
3. `double d1, double d2;`
4. `int i1; int i2;`
5. `int i3; i4;`

# Integer Data Types

- `byte`, `short`, `int`, and `long` are all integer data types.
- They can hold whole numbers such as 5, 10, 23, 89, etc.
- Integer data types cannot hold numbers that have a decimal point in them.
- Integers embedded into Java source code are called *integer literals*.
- See Example: [IntegerVariables.java](#)

# Floating Point Data Types

- Data types that allow fractional values are called *floating-point* numbers.
  - 1.7 and -45.316 are floating-point numbers.
- In Java there are two data types that can represent floating-point numbers.
  - `float` - also called *single precision* (7 decimal points).
  - `double` - also called *double precision* (15 decimal points).

# Floating Point Literals (1 of 3)

- When floating point numbers are embedded into Java source code they are called *floating point literals*.
- The default type for floating point literals is `double`.
  - 29.75, 1.76, and 31.51 are `double` data types.
- Java is a *strongly-typed* language.
- See example: [Sale.java](#)

## Floating Point Literals (2 of 3)

- A `double` value is not compatible with a `float` variable because of its size and precision.
  - `float number;`
  - `number = 23.5; // Error!`
- A `double` can be forced into a `float` by appending the letter `F` or `f` to the literal.
  - `float number;`
  - `number = 23.5F; // This will work.`

# Floating Point Literals (3 of 3)

- Literals cannot contain embedded currency symbols or commas.
  - `grossPay = $1,257.00; // ERROR!`
  - `grossPay = 1257.00; // Correct.`
- Floating-point literals can be represented in *scientific notation*.
  - $47,281.97 == 4.728197 \times 10^4$ .
- Java uses *E notation* to represent values in scientific notation.
  - $4.728197 \times 10^4 == 4.728197E4$ .

# Scientific and E Notation

Decimal Notation	Scientific Notation	E Notation
247.91	$2.4791 \times 10^2$	2.4791E2
0.00072	$7.2 \times 10^{-4}$	7.2E-4
2,900,000	$2.9 \times 10^6$	2.9E6

See example: [SunFacts.java](#)

# The `boolean` Data Type

- The Java `boolean` data type can have two possible values.
  - `true`
  - `false`
- The value of a `boolean` variable may only be copied into a `boolean` variable.

See example: [TrueFalse.java](#)



# The `char` Data Type

- The Java `char` data type provides access to single characters.
- `char` literals are enclosed in single quote marks.
  - `'a'`, `'Z'`, `'\n'`, `'1'`
- Don't confuse `char` literals with string literals.
  - `char` literals are enclosed in single quotes.
  - String literals are enclosed in double quotes.

See example: [Letters.java](#)

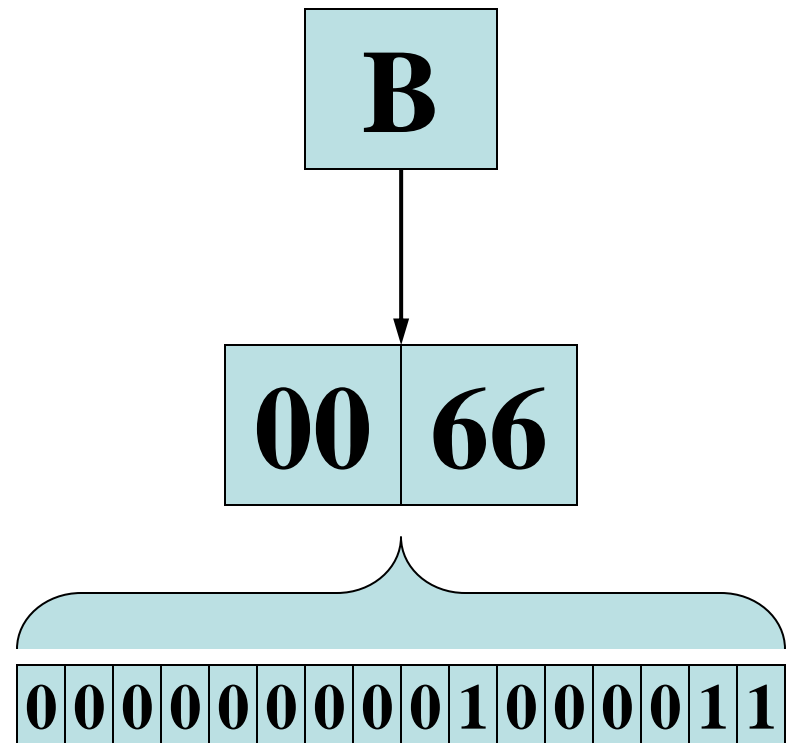
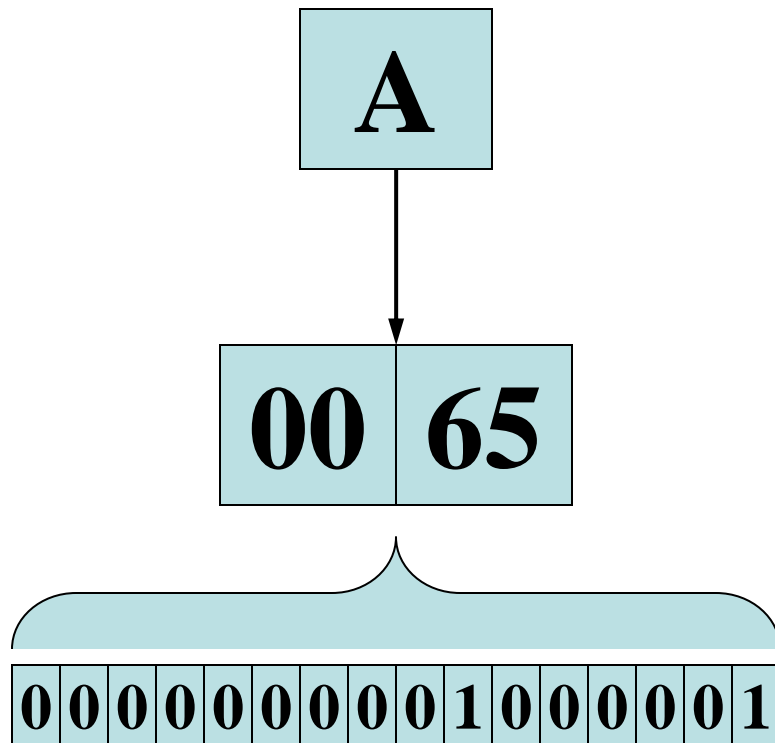
# Unicode (1 of 5)

- Internally, characters are stored as numbers.
- Character data in Java is stored as Unicode characters.
- The Unicode character set can consist of 65536 ( $2^{16}$ ) individual characters.
- This means that each character takes up 2 bytes in memory.
- The first 256 characters in the Unicode character set are compatible with the ASCII\* character set.

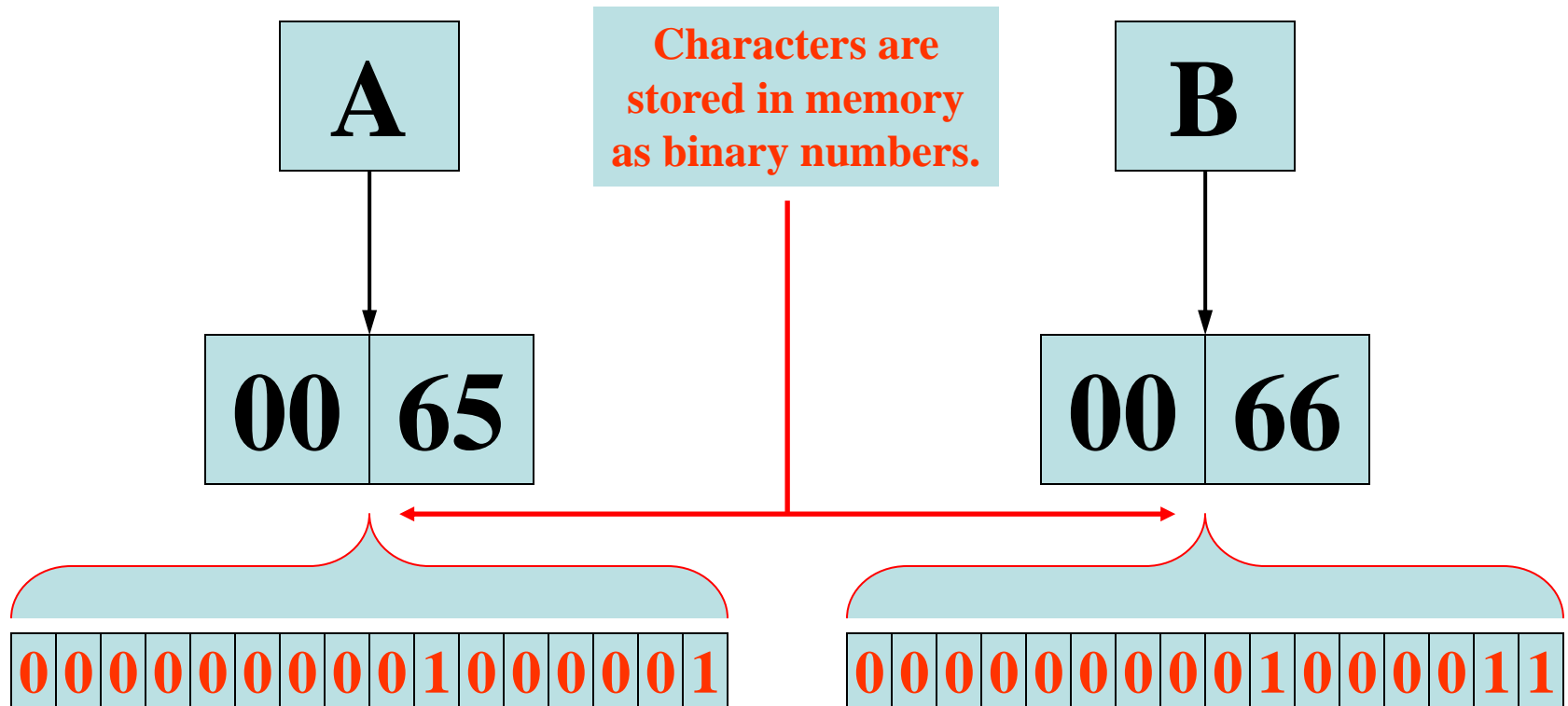
See example: [Letters2.java](#)

\*American Standard Code for Information Interchange

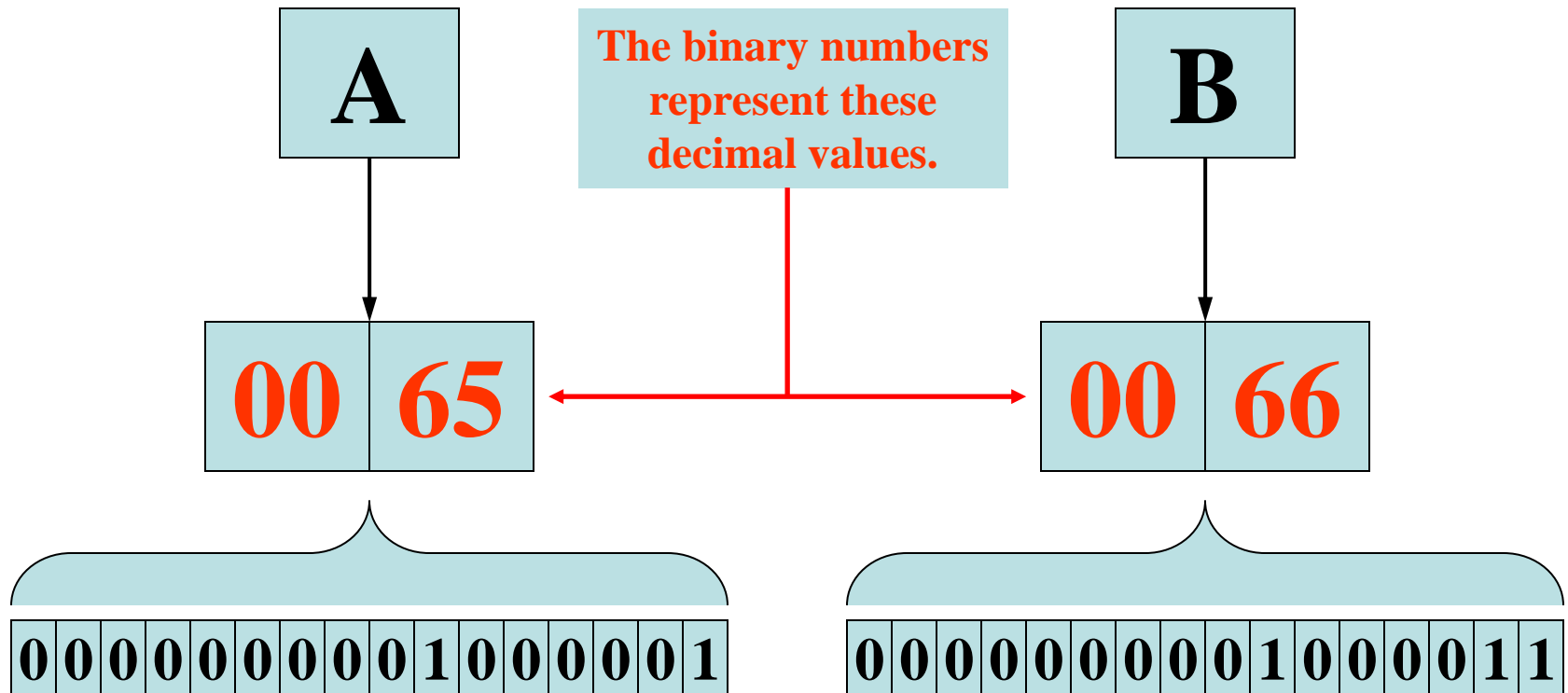
# Unicode (2 of 5)



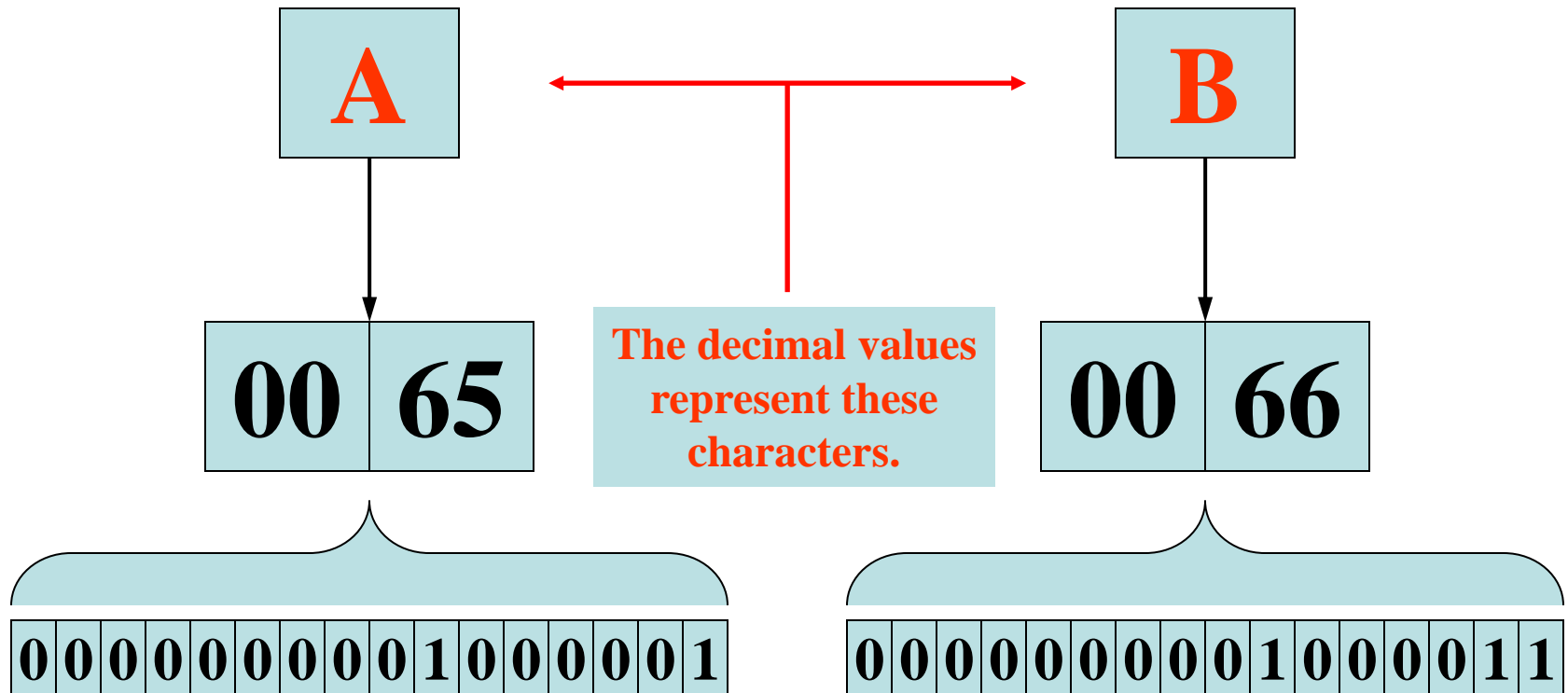
# Unicode (3 of 5)



# Unicode (4 of 5)



# Unicode (5 of 5)



# Type Inference of var

- Starting in Java 10, you have the option of using the keyword var instead of the type for **local variables** under certain conditions.

```
public void reassignment() {  
    var number = 7;  
    number = 4;  
    number = "five"; // DOES NOT COMPILE  
}
```

- Java does not allow var in multiple variable declarations

```
public void twoTypes() {  
    int a, var b = 3; // DOES NOT COMPILE  
    var n = null;    // DOES NOT COMPILE  
}
```

```
var a = 2, b = 3; // DOES NOT COMPILE
```

Will this compile?

```
var o = (String)null;
```

# *var* Rules

- A var is used as a local variable in a constructor, method, or initializer block.
- A var cannot be used in constructor parameters, method parameters, instance variables, or class variables.
- A var is always initialized on the same line (or statement) where it is declared.
- The value of a var can change, but the type cannot.
- A var cannot be initialized with a null value without a type.
- A var is not permitted in a multiple-variable declaration.
- A var is a reserved type name but not a reserved word, meaning it can be used as an identifier except as a class, interface, or enum name.



# Variable Assignment and Initialization

## (1 of 6)

- In order to store a value in a variable, an *assignment statement* must be used.
- The *assignment operator* is the equal (=) sign.
- The operand on the left side of the assignment operator must be a variable name.
- The operand on the right side must be either a literal or expression that evaluates to a type that is compatible with the type of the variable.

# Variable Assignment and Initialization

## (2 of 6)

```
// This program shows variable assignment.

public class Initialize
{
    public static void main(String[] args)
    {
        int month, days;

        month = 2;
        days = 28;
        System.out.println("Month " + month + " has " +
                           days + " Days.");
    }
}
```

**The variables must be declared before they can be used.**

# Variable Assignment and Initialization

## (3 of 6)

```
// This program shows variable assignment.

public class Initialize
{
    public static void main(String[] args)
    {
        int month, days;

        month = 2;
        days = 28;
        System.out.println("Month " + month + " has " +
                           days + " Days.");
    }
}
```

**Once declared, they can then receive a value (initialization); however the value must be compatible with the variable's declared type.**

# Variable Assignment and Initialization

## (4 of 6)

```
// This program shows variable assignment.

public class Initialize
{
    public static void main(String[] args)
    {
        int month, days;

        month = 2;
        days = 28;
        System.out.println("Month " + month + " has " +
                           days + " Days.");
    }
}
```

**After receiving a value, the variables can then be used in output statements or in other calculations.**

# Variable Assignment and Initialization (5 of 6)

```
// This program shows variable initialization.

public class Initialize
{
    public static void main(String[] args)
    {
        int month = 2, days = 28;

        System.out.println("Month " + month + " has " +
                           days + " Days.");
    }
}
```

**Local variables can be declared and initialized on the same line.**

# Variable Assignment and Initialization

## (6 of 6)

- Variables can only hold one value at a time.
- Local variables do not receive a default value.
- Local variables must have a valid type in order to be used.

```
public static void main(String [] args)
{
    int month, days; //No value given...

    System.out.println("Month " + month + " has " +
                       days + " Days.");
}
```

**Trying to use uninitialized variables will generate a Syntax Error when the code is compiled.**

# Primitive value assignments

Which of the following statements will **NOT** compile:

```
int fur = (int)5;  
float egg = 2.0 / 9;  
int hair = (short) 2;  
long feathers = 10(long);  
String type = (String) "Bird";  
int tadpole = (int)5 * 2L;  
short tail = (short)(4 + 10);  
short frog = 3 - 2.0;  
short s = (short)12345678;
```

# Primitive value assignments

Which of the following statements will compile:

```
int fur = (int)5;  
float egg = 2.0 / 9;  
int hair = (short) 2;  
long feathers = 10(long);  
String type = (String) "Bird";  
int tadpole = (int)5 * 2L;  
short tail = (short)(4 + 10);  
short frog = 3 - 2.0;  
short s = (short)12345678;
```



# Primitive value assignments

Which of the following statements will compile:

```
int fish = 1.0;           // DOES NOT COMPILE
short bird = 1921222;     // DOES NOT COMPILE
int mammal = 9f;          // DOES NOT COMPILE
long reptile = 192301398193810323; // DOES NOT COMPILE
```



```
int trainer = (int)1.0;
short ticketTaker = (short)1921222; // Stored as 20678
int usher = (int)9f;
long manager = 192301398193810323L;
```

# JShell

JShell is a command line interactive tool introduced in Java 9. JShell enables you to type a single Java statement and get it executed to see the result right away without having to write a complete class. This feature is commonly known as REPL (Read-Evaluate-Print Loop), which evaluates expressions and executes statements as they are entered and shows the result immediately.

# Arithmetic Operators (1 of 2)

- Java has five (5) arithmetic operators.

Operator	Meaning	Type	Example
+	Addition	Binary	<code>total = cost + tax;</code>
-	Subtraction	Binary	<code>cost = total - tax;</code>
*	Multiplication	Binary	<code>tax = cost * rate;</code>
/	Division	Binary	<code>salePrice = original / 2;</code>
%	Modulus	Binary	<code>remainder = value % 5;</code>

# Arithmetic Operators (2 of 6)

- The operators are called binary operators because they must have two operands.
- Each operator must have a left and right operator.

See example: [Wages.java](#)

- The arithmetic operators work as one would expect.
- It is an error to try to divide any number by zero.
- When working with two integer operands, the division operator requires special attention.

# Integer Division

- Division can be tricky.  
In a Java program, what is the value of  $1/2$ ?
- You might think the answer is 0.5...
- But, that's wrong.
- The answer is simply 0.
- Integer division will truncate any decimal remainder.

# Operator Precedence

- Mathematical expressions can be very complex.
- There is a set order in which arithmetic operations will be carried out.

	Operator	Associativity	Example	Result
Higher Priority	- (unary negation)	Right to left	$x = -4 + 3;$	-1
	* / %	Left to right	$x = -4 + 4 \% 3 * 13 + 2;$	11
Lower Priority	+ -	Left to right	$x = 6 + 3 - 4 + 6 * 3;$	23

# Grouping with Parenthesis

- When parenthesis are used in an expression, the inner most parenthesis are processed first.
- If two sets of parenthesis are at the same level, they are processed left to right.

The diagram illustrates the order of operations for the expression `x = ((4*5) / (5-2)) - 25; // result = -19`. Red curly braces and numbers indicate the sequence of operations:

- 1**: The innermost operation `4*5` is processed first.
- 2**: The next innermost operation `5-2` is processed second.
- 3**: The division operation `(4*5) / (5-2)` is processed third, as it is the next level of grouping.
- 4**: Finally, the subtraction operation `((4*5) / (5-2)) - 25` is processed fourth.

# Numeric Promotion

- If two values have different data types, Java will automatically promote one of the values to the larger of the two data types.
- If one of the values is integral and the other is floating-point, Java will automatically promote the integral value to the floating-point value's data type.
- Smaller data types, namely, byte, short, and char, are first promoted to int any time they're used with a Java binary arithmetic operator, even if neither of the operands is int.
- After all promotion has occurred and the operands have the same data type, the resulting value will have the same data type as its promoted operands.



- What is the data type of `x * y`?  
**long**

```
int x = 1;
long y = 33;
var z = x * y;
```

- What is the data type of `x + y`?  
**Will not compile**

```
double x = 39.21;
float y = 2.1;
var z = x + y;
```

- What is the data type of `x * y`?  
**int**

```
short x = 10;
short y = 3;
var z = x * y;
```

- What is the data type of `w * x / y`?  
**double**

```
short w = 14;
float x = 13;
double y = 30;
var z = w * x / y;
```

- Will the code compile?

```
short mouse = 10;
short hamster = 3;
short capybara = mouse * hamster;
```

**No**

```
short mouse = 10;
short hamster = 3;
short capybara = (short)(mouse * hamster);
```

**Yes**

```
short mouse = 10;
short hamster = 3;
short capybara = (short)mouse * hamster;
short gerbil = 1 + (short)(mouse * hamster);
```

**No**

# Combined Assignment Operators (1 of 2)

- Java has some combined assignment operators.
- These operators allow the programmer to perform an arithmetic operation and assignment with a single operator.
- Although not required, these operators are popular since they shorten simple equations.

# Combined Assignment Operators (2 of 2)

Operator	Example	Equivalent	Value of variable after operation
<b>+=</b>	<code>x += 5;</code>	<code>x = x + 5;</code>	The old value of <b>x</b> plus 5.
<b>-=</b>	<code>y -= 2;</code>	<code>y = y - 2;</code>	The old value of <b>y</b> minus 2
<b>*=</b>	<code>z *= 10;</code>	<code>z = z * 10;</code>	The old value of <b>z</b> times 10
<b>/=</b>	<code>a /= b;</code>	<code>a = a / b;</code>	The old value of <b>a</b> divided by <b>b</b> .
<b>%=</b>	<code>c %= 3;</code>	<code>c = c % 3;</code>	The remainder of the division of the old value of <b>c</b> divided by 3.

# Combined Assignment Operators

```
long x = 10;
```

```
int y = 5;
```

```
y = y * x; // DOES NOT COMPILE
```

```
y = (int) (y * x);
```

```
y *= x;
```

```
/* the compiler will automatically cast the resulting value to  
the data type of the value on the left side of the compound  
operator. */
```

# The Math class

- The Java API provides a class named Math, which contains numerous methods that are useful for performing complex mathematical operations.
- Math.pow(double, double)

$2^8$  -> Math.pow(2.0, 8.0)

- Math.sqrt(double)

$\sqrt{36}$  -> Math.sqrt(36)

# Creating Constants with `final` (1 of 3)

- Many programs have data that does not need to be changed.
- Littering programs with literal values can make the program hard to read and maintain.
- Replacing literal values with constants remedies this problem.
- Constants allow the programmer to use a name rather than a value throughout the program.
- Constants also give a singular point for changing those values when needed.

## Creating Constants with `final` (2 of 3)

- Constants keep the program organized and easier to maintain.
- Constants are identifiers that can hold only a single value.
- Constants are declared using the keyword `final`.
- **Constants need not be initialized when declared**; however, they must be initialized before they are used or a compiler error will be generated.

# Creating Constants with `final` (3 of 3)

- Once initialized with a value, constants cannot be changed programmatically.
- By convention, constants are all upper case and words are separated by the underscore character.

```
final int CAL_SALES_TAX = 0.725;
```



# The `String` Class

- Java has no primitive data type that holds a series of characters.
- The `String` class from the Java standard library is used for this purpose.
- In order to be useful, the variable must be created to reference a `String` object.

```
String number;
```

- Notice the `S` in `String` is upper case.
- By convention, class names should always begin with an uppercase character.

# Primitive vs. Reference Variables (1 of 2)

- Primitive variables actually contain the value that they have been assigned.

```
number = 25;
```

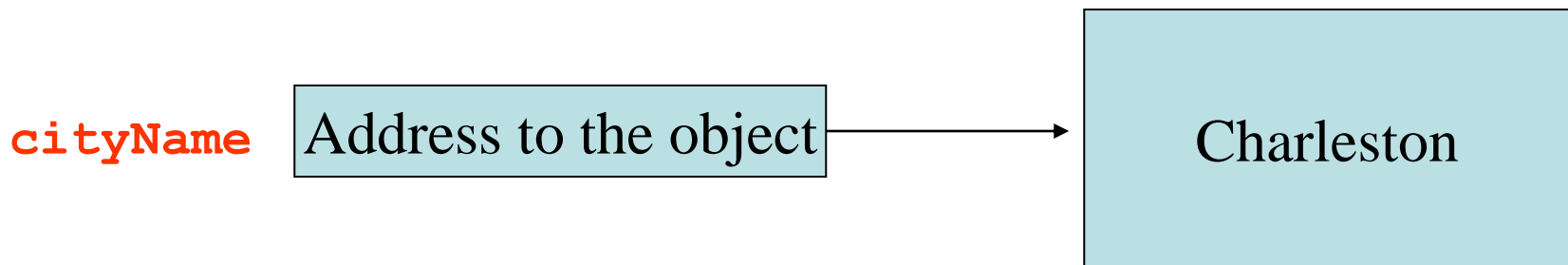
- The value 25 will be stored in the memory location associated with the variable `number`.
- Objects are not stored in variables, however. Objects are *referenced* by variables.

# Primitive vs. Reference Variables (2 of 2)

- When a variable references an object, it contains the memory address of the object's location.
- Then it is said that the variable *references* the object.

```
String cityName = "Charleston";
```

**The object that contains the character string “Charleston”**



# String Objects

- A variable can be assigned a String literal.

```
String value = "Hello";
```

- Strings are the only objects that can be created in this way.

- A variable can be created using the *new* keyword.

```
String value = new String("Hello");
```

- This is the method that all other objects must use when they are created.

See example: [StringDemo.java](#)

# The String Methods

- Since `String` is a class, objects that are instances of it have methods.
- One of those methods is the `length` method.  

```
stringSize = value.length();
```
- This statement runs the `length` method on the object pointed to by the `value` variable.

See example: [StringLength.java](#)

# String Methods

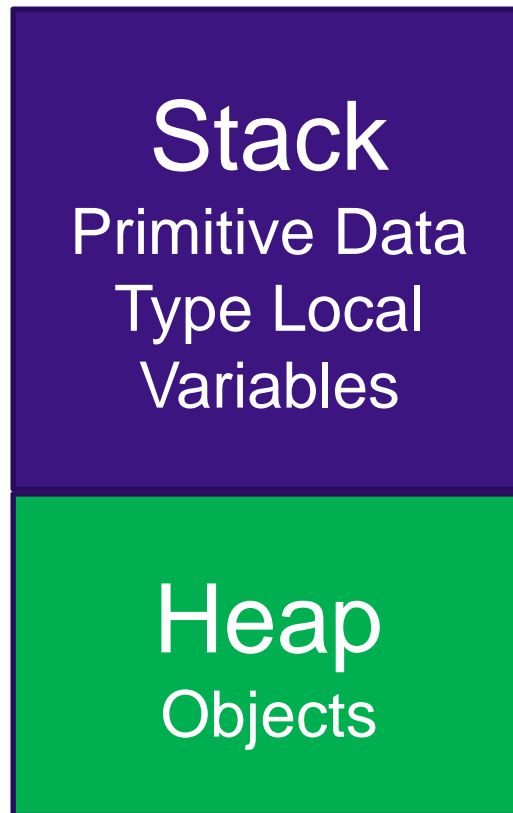
- The `String` class contains many methods that help with the manipulation of `String` objects.
- `String` objects are *immutable*, meaning that they cannot be changed.
- Many of the methods of a `String` object can create new versions of the object.

See example: [StringMethods.java](#)

# String vs StringBuffer vs StringBuilder

# Heap & Stack

- Objects in heap
- Local variables, methods in stack

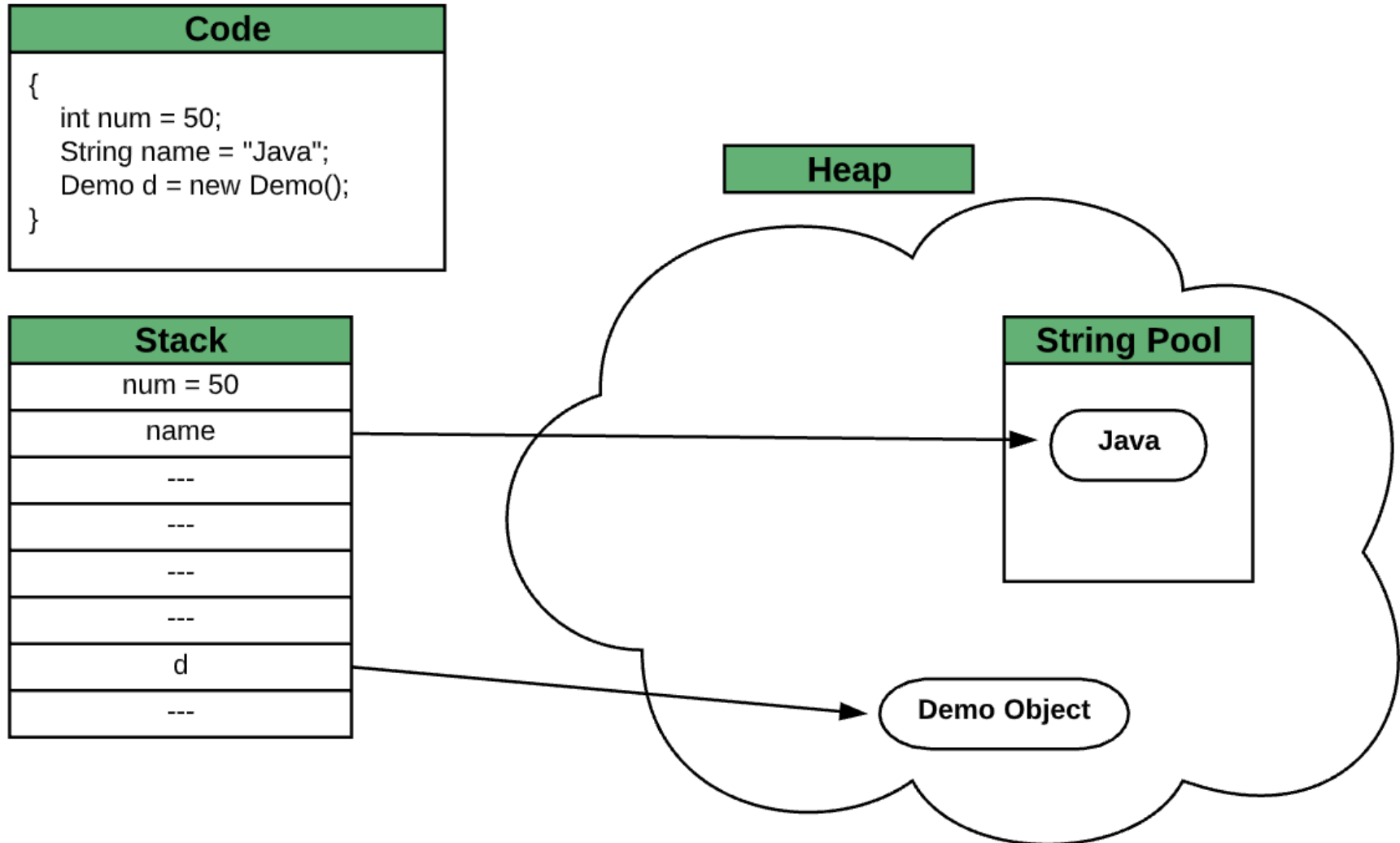




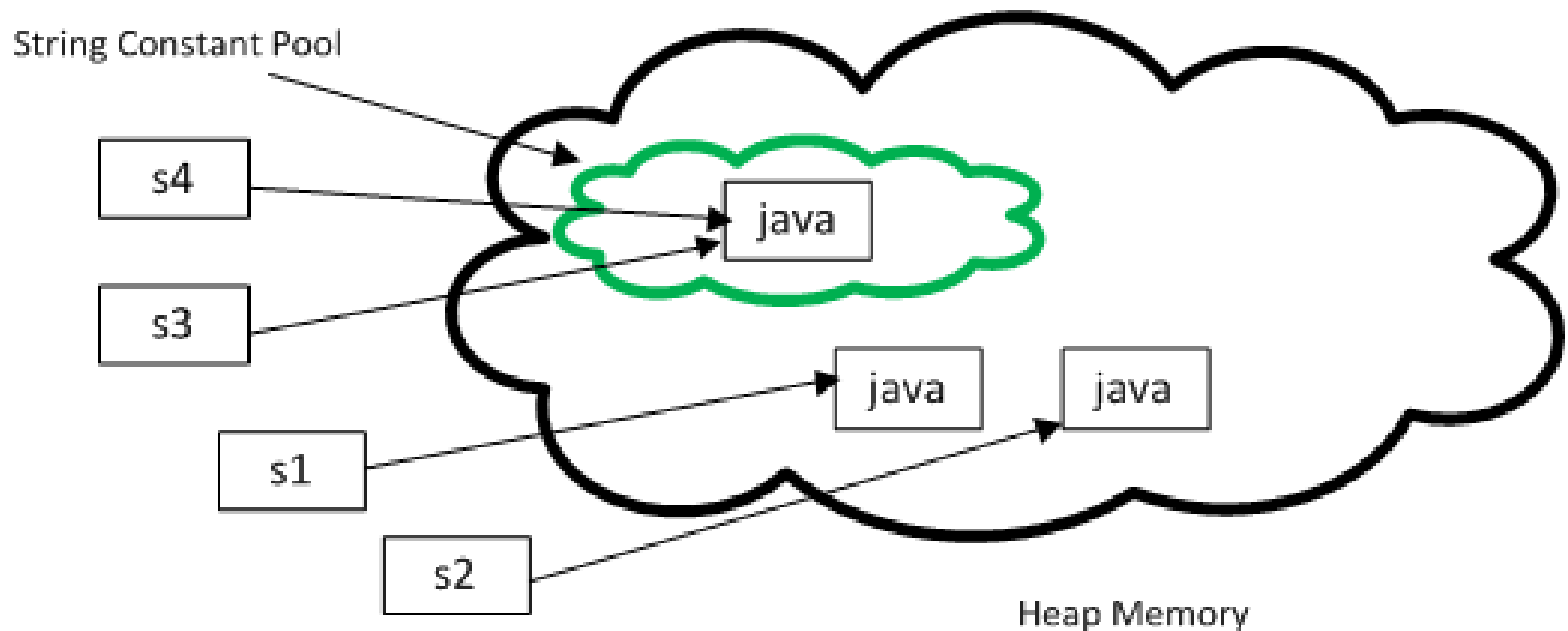
# String

- Strings in java are immutable
- Once created they cannot be altered and hence any alterations will lead to creation of new string object

# Stack vs Heap



```
String s1 = new String("Java");  
String s2 = new String("Java");  
String s3 = "Java";  
String s4 = "Java";
```



```
System.out.println("s1==s3? " + (s1==s2));  
System.out.println("s1==s3? " + (s1==s3));  
System.out.println("s3==s4? " + (s3==s4));
```

# Example

- `String s1 = "Example"`
- `String s2 = new String("Example")`
- `String s3 = "Example"`
- The difference between the three statements is that, `s1` and `s3` are pointing to the same memory location i.e. the string pool. `s2` is pointing to a memory location on the heap.
- Using a `new` operator creates a memory location on the heap.
- Concatenating `s1` and `s3` leads to creation of a new string in the pool.

# StringBuffer

- StringBuffer is a synchronized and allows us to mutate the string.
- StringBuffer has many utility methods to manipulate the string.
- This is more useful when using in a multithreaded environment.
- Always has a locking overhead.

# Example

```
public class mybuffers{  
    public static void main(String args[]){  
        StringBuffer buffer = new StringBuffer("Hi");  
        buffer.append("Bye");  
        System.out.println(buffer);  
    }  
}
```

- This program appends the string Bye to Hi and prints it to the screen.

# StringBuilder

- StringBuilder is the same as the StringBuffer class
- The StringBuilder class is not synchronized and hence in a single threaded environment, the overhead is less than using a StringBuffer.

# Scope

- *Scope* refers to the part of a program that has access to a variable's contents.
- Variables declared inside a method (like the main method) are called *local variables*.
- Local variables' scope begins at the declaration of the variable and ends at the end of the method in which it was declared.

See example: [Scope.java](#) (This program contains an intentional error.)



# Commenting Code (1 of 3)

- Java provides three methods for commenting code.

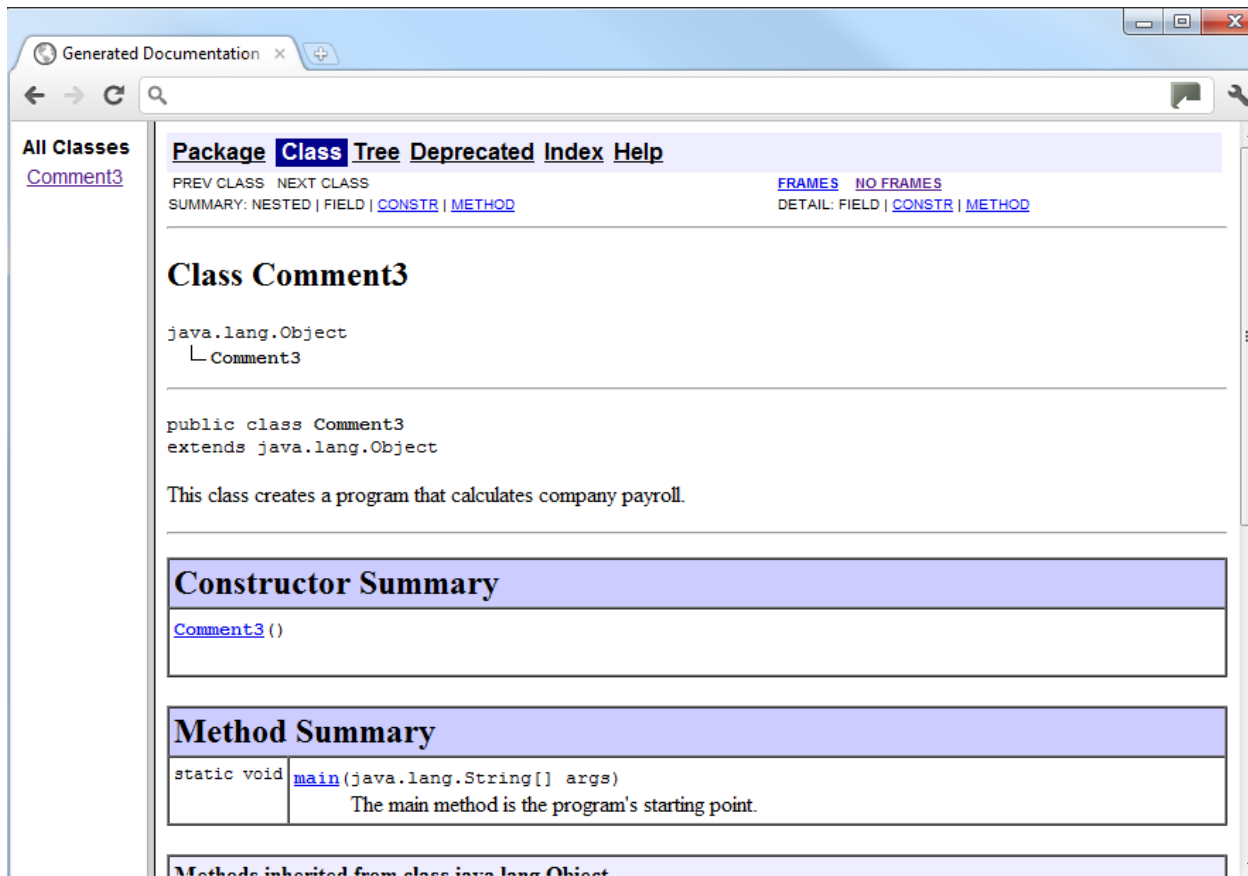
Comment Style	Description
//	Single line comment. Anything after the // on the line will be ignored by the compiler.
/* ... */	Block comment. Everything beginning with /* and ending with the first */ will be ignored by the compiler. This comment type cannot be nested.
/** ... */	Javadoc comment. This is a special version of the previous block comment that allows comments to be documented by the javadoc utility program. Everything beginning with the /** and ending with the first */ will be ignored by the compiler. This comment type cannot be nested.

## Commenting Code (2 of 3)

- Javadoc comments can be built into HTML documentation.
- See example: [Comment3.java](#)
- To create the documentation:
  - Run the `javadoc` program with the source file as an argument
  - Ex: `javadoc Comment3.java`
- The `javadoc` program will create `index.html` and several other documentation files in the same directory as the input file.

# Commenting Code (3 of 3)

- Example [index.html](#):



The screenshot shows a web browser window titled "Generated Documentation". The address bar contains a search icon and a magnifying glass. The page has a navigation bar with links: [Package](#), [Class](#) (selected), [Tree](#), [Deprecated](#), [Index](#), and [Help](#). Below the navigation bar, there are links for [PREV CLASS](#), [NEXT CLASS](#), [SUMMARY: NESTED](#), [FIELD](#), [CONSTR](#), and [METHOD](#). On the right side, there are links for [FRAMES](#), [NO FRAMES](#), [DETAIL: FIELD](#), [CONSTR](#), and [METHOD](#).

The main content area displays the class **Class Comment3**. It shows the inheritance hierarchy: `java.lang.Object` and `└ Comment3`. Below this, the class declaration is shown: `public class Comment3 extends java.lang.Object`. A description follows: "This class creates a program that calculates company payroll."

Below the description, there are two summary sections:

- Constructor Summary**: A table with one row showing the constructor `Comment3()`.
- Method Summary**: A table with one row showing the static void method `main(java.lang.String[] args)`. The description for this method is "The main method is the program's starting point."

At the bottom, there is a section titled "Methods inherited from class java.lang.Object".

# Programming Style

- Although Java has a strict syntax, whitespace characters are ignored by the compiler.
- The Java whitespace characters are:
  - space
  - tab
  - newline
  - carriage return
  - form feed

See example: [Compact.java](#)

# Indentation

- Programs should use proper indentation.
- Each block of code should be indented a few spaces from its surrounding block.
- Two to four spaces are sufficient.
- Tab characters should be avoided.
  - Tabs can vary in size between applications and devices.
  - Most programming text editors allow the user to replace the tab with spaces.

See example: [Readable.java](#)

# The Scanner Class (1 of 2)

- To read input from the keyboard we can use the `Scanner` class.
- The `Scanner` class is defined in `java.util`, so we will use the following statement at the top of our programs:

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

# The Scanner Class (2 of 2)

- Scanner objects work with `System.in`
- To create a Scanner object:  

```
Scanner keyboard = new Scanner  
(System.in);
```
- Scanner class methods are listed in Table 2-18 in the text.
- See example: [Payroll.java](#)

# Console Input Using the Scanner Class

- Starting with version 5.0, Java includes a class for doing simple keyboard input named the **Scanner** class
- In order to use the **Scanner** class, a program must include the following line near the start of the file:

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

- This statement tells Java to
  - Make the **Scanner** class available to the program
  - Find the **Scanner** class in a library of classes (i.e., Java *package*) named **java.util**



# Console Input Using the Scanner Class

- The following line creates an object of the class **Scanner** and names the object **keyboard** :

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

- Although a name like **keyboard** is often used, a **Scanner** object can be given any name
  - For example, in the following code the **Scanner** object is named **scannerObject**

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

- Once a **Scanner** object has been created, a program can then use that object to perform keyboard input using methods of the **Scanner** class

# Console Input Using the Scanner Class

- The method `nextInt` reads one `int` value typed in at the keyboard and assigns it to a variable:

```
int numberOfPods = keyboard.nextInt();
```

- The method `nextDouble` reads one `double` value typed in at the keyboard and assigns it to a variable:

```
double d1 = keyboard.nextDouble();
```

- Multiple inputs must be separated by *whitespace* and read by multiple invocations of the appropriate method
  - Whitespace is any string of characters, such as blank spaces, tabs, and line breaks that print out as white space

# Console Input Using the Scanner Class

- The method **next** reads one string of non-whitespace characters delimited by whitespace characters such as blanks or the beginning or end of a line
- Given the code

```
String word1 = keyboard.next();  
String word2 = keyboard.next();
```

and the input line

**jelly beans**

The value of **word1** would be **jelly**, and the value of **word2** would be **beans**

# Console Input Using the Scanner Class

- The method `nextLine` reads an entire line of keyboard input
- The code,  

```
String line = keyboard.nextLine();
```

reads in an entire line and places the string that is read into the variable `line`
- The end of an input line is indicated by the escape sequence `'\n'`
  - This is the character input when the **Enter** key is pressed
  - On the screen it is indicated by the ending of one line and the beginning of the next line
- When `nextLine` reads a line of text, it reads the `'\n'` character, so the next reading of input begins on the next line
  - However, the `'\n'` does not become part of the string value returned (e.g., the string named by the variable `line` above does not end with the `'\n'` character)

# Keyboard Input Demonstration (Part 1 of 2)

## Display 2.6 Keyboard Input Demonstration

```
1  import java.util.Scanner;
2  public class ScannerDemo
3  {
4      public static void main(String[] args)
5      {
6          Scanner keyboard = new Scanner(System.in);
7
8          System.out.println("Enter the number of pods followed by");
9          System.out.println("the number of peas in a pod:");
10         int numberOfPods = keyboard.nextInt();
11         int peasPerPod = keyboard.nextInt();
12
13         int totalNumberOfPeas = numberOfPods*peasPerPod;
14
15         System.out.print(numberOfPods + " pods and ");
16         System.out.println(peasPerPod + " peas per pod.");
17         System.out.println("The total number of peas = "
18                             + totalNumberOfPeas);
19     }
20 }
```

*Makes the Scanner class available to your program.*

*Creates an object of the class Scanner and names the object keyboard.*

*Each reads one int from the keyboard*

Copyright © 2016 Pearson Inc. All rights reserved.

# Keyboard Input Demonstration

## (Part 2 of 2)

### Display 2.6 Keyboard Input Demonstration

#### SAMPLE DIALOGUE 1

Enter the number of pods followed by  
the number of peas in a pod:

22 10

22 pods and 10 peas per pod.

The total number of peas = 220

*The numbers that are input must be separated by whitespace, such as one or more blanks.*

#### SAMPLE DIALOGUE 2

Enter the number of pods followed by  
the number of peas in a pod:

22

10

22 pods and 10 peas per pod.

The total number of peas = 220

*A line break is also considered whitespace and can be used to separate the numbers typed in at the keyboard.*

# Another Keyboard Input Demonstration (Part 1 of 3)

## Display 2.7 Another Keyboard Input Demonstration

---

```
1  import java.util.Scanner;

2  public class ScannerDemo2
3  {
4      public static void main(String[] args)
5      {
6          int n1, n2;
7          Scanner scannerObject = new Scanner(System.in);

8          System.out.println("Enter two whole numbers");
9          System.out.println("seperated by one or more spaces:");

10         n1 = scannerObject.nextInt();
11         n2 = scannerObject.nextInt();
12         System.out.println("You entered " + n1 + " and " + n2);

13         System.out.println("Next enter two numbers.");
14         System.out.println("Decimal points are allowed.");
```

*Creates an object of the class **Scanner** and names the object **scannerObject**.*

*Reads one **int** from the keyboard.*

# Another Keyboard Input Demonstration (Part 2 of 3)

## Display 2.7 Another Keyboard Input Demonstration

```
15     double d1, d2;
16     d1 = scannerObject.nextDouble();
17     d2 = scannerObject.nextDouble();
18     System.out.println("You entered " + d1 + " and " + d2);

19     System.out.println("Next enter two words:");

20     String word1 = scannerObject.next();
21     String word2 = scannerObject.next();
22     System.out.println("You entered \"" +
23         word1 + "\" and \"" + word2 + "\"");

24     String junk = scannerObject.nextLine(); //To get rid of '\n'

25     System.out.println("Next enter a line of text:");
26     String line = scannerObject.nextLine();
27     System.out.println("You entered: \"" + line + "\"");
28 }
29 }
```

*Reads one double from the keyboard.*

*Reads one word from the keyboard.*

*This line is explained in the Pitfall section "Dealing with the Line Terminator, '\n'".*

*Reads an entire line.*

(continued)



# Another Keyboard Input Demonstration (Part 3 of 3)

## Display 2.7 Another Keyboard Input Demonstration

---

### SAMPLE DIALOGUE

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 57**

You entered 9.99 and 57.0

Next enter two words:

**jelly beans**

You entered "jelly" and "beans"

Next enter a line of text:

**Java flavored jelly beans are my favorite.**

You entered "Java flavored jelly beans are my favorite."

# Pitfall: Dealing with the Line Terminator, ' \n '

- The method `nextLine` of the class `Scanner` reads the remainder of a line of text starting wherever the last keyboard reading left off
- This can cause problems when combining it with different methods for reading from the keyboard such as `nextInt`
- Given the code,

```
Scanner keyboard = new Scanner(System.in);  
int n = keyboard.nextInt();  
String s1 = keyboard.nextLine();  
String s2 = keyboard.nextLine();
```

and the input,

```
2
```

```
Heads are better than
```

```
1 head.
```

what are the values of `n`, `s1`, and `s2`?

# Pitfall: Dealing with the Line Terminator, ' \n '

- Given the code and input on the previous slide  
    **n** will be equal to **"2"**,  
    **s1** will be equal to **" "**, and  
    **s2** will be equal to **"heads are better than"**
- If the following results were desired instead  
    **n** equal to **"2"**,  
    **s1** equal to **"heads are better than"**, and  
    **s2** equal to **"1 head"**

then an extra invocation of **nextLine** would be needed to get rid of the end of line character ( **' \n '** )

# Methods in the Class Scanner

## (Part 1 of 3)

### Display 2.8    **Methods of the Scanner Class**

---

The Scanner class can be used to obtain input from files as well as from the keyboard. However, here we are assuming it is being used only for input from the keyboard.

To set things up for keyboard input, you need the following at the beginning of the file with the keyboard input code:

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

You also need the following before the first keyboard input statement:

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

The *Scanner\_Object\_Name* can then be used with the following methods to read and return various types of data typed on the keyboard.

Values to be read should be separated by whitespace characters, such as blanks and/or new lines. When reading values, these whitespace characters are skipped. (It is possible to change the separators from whitespace to something else, but whitespace is the default and is what we will use.)

```
Scanner_Object_Name.nextInt()
```

Returns the next value of type `int` that is typed on the keyboard.

# Methods in the Class Scanner

## (Part 2 of 3)

### Display 2.8 Methods of the Scanner Class

---

*Scanner\_Object\_Name.nextLong()*

Returns the next value of type `long` that is typed on the keyboard.

*Scanner\_Object\_Name.nextByte()*

Returns the next value of type `byte` that is typed on the keyboard.

*Scanner\_Object\_Name.nextShort()*

Returns the next value of type `short` that is typed on the keyboard.

*Scanner\_Object\_Name.nextDouble()*

Returns the next value of type `double` that is typed on the keyboard.

*Scanner\_Object\_Name.nextFloat()*

Returns the next value of type `float` that is typed on the keyboard.

(continued)

# Methods in the Class Scanner

## (Part 3 of 3)

### Display 2.8 Methods of the Scanner Class

---

*Scanner\_Object\_Name.next()*

Returns the `String` value consisting of the next keyboard characters up to, but not including, the first delimiter character. The default delimiters are whitespace characters.

*Scanner\_Object\_Name.nextBoolean()*

Returns the next value of type `boolean` that is typed on the keyboard. The values of `true` and `false` are entered as the strings `"true"` and `"false"`. Any combination of upper- and/or lowercase letters is allowed in spelling `"true"` and `"false"`.

*Scanner\_Object\_Name.nextLine()*

Reads the rest of the current keyboard input line and returns the characters read as a value of type `String`. Note that the line terminator `'\n'` is read and discarded; it is not included in the string returned.

*Scanner\_Object\_Name.useDelimiter(New\_Delimiter);*

Changes the delimiter for keyboard input with *Scanner\_Object\_Name*. The *New\_Delimiter* is a value of type `String`. After this statement is executed, *New\_Delimiter* is the only delimiter that separates words or numbers. See the subsection "Other Input Delimiters" for details.

# The Empty String

- A string can have any number of characters, including zero characters
  - `""` is the empty string
- When a program executes the `nextLine` method to read a line of text, and the user types nothing on the line but presses the **Enter** key, then the `nextLine` Method reads the empty string