

Java Fundamentals I

Michael C. Hackett

Assistant Professor, Computer Science



Lecture Topics

- Programming Language Concepts
- Primitive Data Types
 - Variables- Declaration, Initialization, Assignment
 - Literals
- Strings
- Standard Streams
- Comments/Documentation

Colors/Fonts

• Local Variable Names	—	Brown
• Primitive data types	—	Fuchsia
• Literals	—	Blue
• Keywords	—	Orange
• Object names	—	Green
• Operators/Punctuation	—	Black
• Field Names	—	Lt Blue
• Method Names	—	Purple
• Parameter Names	—	Gold
• Comments	—	Gray
• Package Names	—	Pink

Source Code — **Consolas**
Output — Courier New

Basics of a Programming Language

- Modern, high-level languages, programming languages incorporate the following concepts:
 - Keywords
 - Operators
 - Punctuation
 - Syntax
 - Variables

Keywords

- A **keyword** (or reserved word) is word that has special meaning to a programming language.
 - The word is *reserved* from being used in other contexts within programs written in the language.
 - Keywords are typically used in a language for performing some specific process.
- For example, in many languages the word “if” is a reserved word.
 - The if keyword begins a special statement that allows a program to make a decision.
 - **if** *this is true* then do *that*
- Many different languages utilize the same keywords.

Operators

- An **operator** is (usually) a symbol that performs an operation on one or more operands(values/data).
- In the mathematical expression $1 + 2$, the plus sign is an operator that adds the two operands together.
 - In this example:
 - **1** and **2** are operands
 - **+** is the operator
 - **Addition** is the overall operation performed by the operator.
- Many languages use the same operators for performing common operations, like arithmetic and comparisons.
- In some cases, keywords can take the form of an operator.

Punctuation

- ***Punctuation*** is characters or symbols used when writing statements in a programming language.
 - A *statement* is like a sentence or an instruction in a programming language.
- Consider the sentence *I went to the park, the mall, and the college.*
 - We used punctuation for listing multiple places (commas) and a period to end the sentence.
 - Programming languages will use characters in similar ways.
 - For example, commas are often used in programming languages when specifying a list of values.
- Punctuation varies among different languages.
 - Some languages, like Java and C++ require ending statements with semicolons.
 - Languages like Python do not require punctuation at the end of statements.

Syntax

- ***Syntax*** is the language's rules for how keywords, operators, punctuation, and identifiers must be arranged in statements.
- Syntax ensures the statements and instructions of a program are correctly executed.
- “Tall, he is.”
 - We can kind-of understand what this English statement is saying.
 - A computer can't “guess” our intentions when we give it instructions.
 - A statement is syntactically correct, or it is not. There can be no ambiguity.

Syntax

- A language's syntax is usually the most notable difference among different programming languages.
 - How languages accomplish tasks is comparable, but how we write those statements to accomplish the task differs.
- Some languages have comparable syntax.
 - Many languages are derived from or inspired by other languages.
 - Java and C++ have comparable syntax as they are both heavily based on the C programming language.
 - Python and Java have some similarities, but overall have many differences in syntax.

Data Types

- A ***data type*** (aka ***type***) specifies the kind of information that data can be.
- It is the *meaning* of the data.
 - The type identifies how the data can be used.
- Data types are used for
 - Specifying the possible values the data can be interpreted as.
 - Specifying what operations can be performed on the data.

Data Types

- All languages have low-level data types for use.
 - Often called ***primitive*** or ***standard data types***.
- These low-level types typically share similarities across different languages.
- The building blocks for more complex types.

Numeric Types

- Programming languages generally have two types for numeric values.
 - Integers
 - Floating Point Numbers (“Floats”)
- Some languages, like Java, have multiple types for integers and floating-point numbers.
 - Other languages, like Python, only have one for each.

Integers

- An ***integer*** is a whole number.
 - 26
 - 0
 - -5
- Integers do not have fractional portions.
 - 45.7 is not an integer.

Floating-Point Numbers

- A floating-point number is used to represent a rational number, or numbers with fractional amounts.
 - 56.7
 - 0.86
 - 4.019999
 - -31.5
- The binary information that makes up a floating-point number (“float”) is organized in a special way.

Signed and Unsigned Numbers

- In computing, there exists signed and unsigned numbers.
 - ***Signed numbers*** are numbers that can be positive or negative.
 - ***Unsigned numbers*** are numbers that can only be positive.
- All numeric primitive data types in Java are signed.

Primitive Data Types

- Java has eight primitive data types
 - Four Integer Types
byte, short, int, long
 - Two Floating-Point Types
float, double
 - One Boolean Type
boolean
 - One Character Type
char

Primitive Data Types

- Integer Types

- byte** (8 bits)

- Can represent any integer between -128 and 127

- short** (16 bits)

- Can represent any integer between -32,768 and 32,767

- int** (32 bits)

- Can represent any integer between -2,147,483,648 and 2,147,483,647
 - Most frequently used integer primitive.

- long** (64 bits)

- Can represent any integer between -2^{63} and $2^{63}-1$

Primitive Data Types

- Floating-Point Types

- float** (32 bits)

- Can represent values between $\sim \pm 3.4 \times 10^{38}$ with 7 significant digits.

- double** (64 bits)

- Can represent values between $\sim \pm 1.7 \times 10^{308}$ with 15 significant digits.

- Boolean Type

- boolean** (1 bit)

- Can be **true** or **false**.
 - Used for decision making.
 - Depending on the OS's memory management, it may not be able to allocate a single bit of memory.
 - The OS may allocate an entire byte (8 bits), though only one of the bits will be used.

Primitive Data Types

- Character type

char (16 bits)

- Can represent a single, 16-bit Unicode character.
- UTF-16 character table for reference:

<http://www.fileformat.info/info/charset/UTF-16/list.htm>

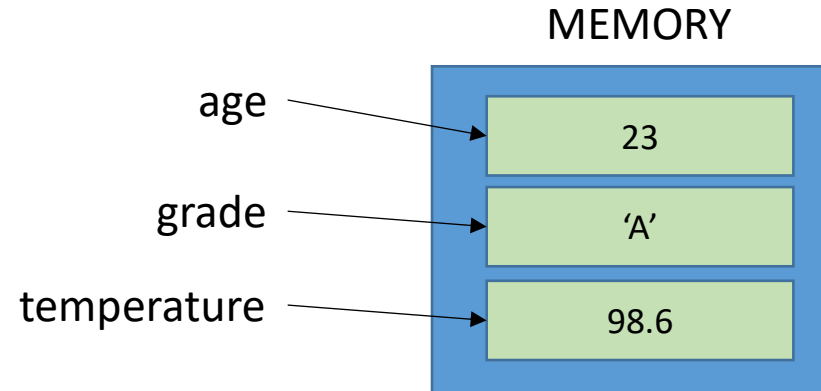
- Together, multiple chars make up a **String** object.
 - Strings are not primitive types in Java, they are *objects* (discussed in later lectures).

Variables

- A ***variable*** (aka ***field*** or ***identifier***) is a programmer-defined name that references the location of an area of memory.
- Like the name suggests, the data referenced by a variable may vary.
 - New values/data can be assigned to the memory location the variable references.
- All variables must have a type associated with it.
 - This restriction ensures the data at that memory location referenced by the variable is interpreted correctly.

Variables

- Variable names are programmer defined.
 - We choose variable names based on the data they represent in our programs.



Declaring a Variable

- When a variable is ***declared***, you are stating:
 - The type of data this variable will reference.
 - The name of the variable.

- Examples:

```
int myExampleInt;  
double myExampleDouble;  
char myExampleCharacter;  
boolean myExampleBoolean;
```

- Declare multiple variables at once:

```
int mySecondExampleInt, myThirdExampleInt, myOtherExampleInt;
```

Variable Name (You make this up)

Data Type

`int myExampleInt;`

Remember, statements in Java end with a semicolon.

Initializing a Variable

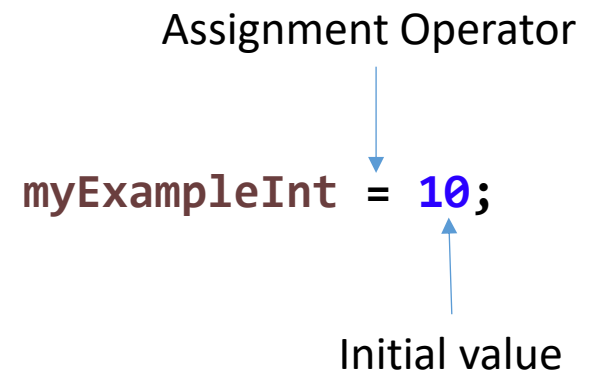
- A variable becomes ***initialized*** when an initial value/data is stored to the memory location referenced by the variable.
- Assignment operator: =
 - Used to store a value to the memory location referenced by a variable.
- Examples (all previously declared in the last slide):

```
myExampleInt = 10;  
myExampleDouble = 15.3;  
myExampleCharacter = 'A';  
myExampleBoolean = false;
```

Assignment Operator

myExampleInt = 10;

Initial value



- Note: You cannot initialize a variable that has not yet been declared!


Declare and Initialize a Variable

- You can assign an initial value when you declare a new variable.

- Examples:

```
int testInt = -457;  
double testDouble = 356.45;  
char testCharacter = 'Z';  
boolean testBoolean = true;
```

Declaration



```
int testInt = -457;
```

Initialization

- You can declare and initialize more than one new variable at once:

```
int testInt1 = 246, testInt2 = -76, testInt3 = 10;
```


(Re)Assignment

- **Assignment** is simply replacing the existing value at the memory location referenced by a variable with a new value.
 - The new data must be of the correct type.
- Like initialization, also uses the Assignment Operator, =
- Example:

```
double currentTemperature = 67.5;  
currentTemperature = 68.2;
```

A few notes on variables

- The default values of the primitive data types (their values before any initialization is performed) are:
 - byte, short, int, long : 0
 - float, double : 0.0
 - char : ' '
 - boolean : false
- Variable names must be unique, regardless of data type.
- A variable's data type cannot be changed after declaration.
- The names of the primitive data types are all Java keywords.
 - They cannot be the name of a variable.

Copying values from one variable to another

- Use the Assignment Operator, =
- Examples:

```
double currentSpeed = 35.2;  
double copyOfCurrentSpeed = currentSpeed;
```

Value located at the memory location referenced by currentSpeed is copied to the memory location referenced by copyOfCurrentSpeed

```
int numberOfApples = 10;  
int copyOfApples;  
copyOfApples = numberOfApples;
```

Value located at the memory location referenced by numberOfApples is copied to the memory location referenced by copyOfApples

Constants

- A ***constant*** is a value that cannot be changed.
- Constants are declared by using the **final** keyword.
- Examples:

```
final int FREEZING_POINT = 32;  
final double PI = 3.14159;  
final char LETTER_ZEE = 'Z';
```

- Trying to assign a constant with a new value will result in a compile time error.
 - A compile-time error occurs when you compile your code.
 - A run-time error occurs when your program is running.

Naming Variables

- Variable names are case-sensitive.

```
int someNumberValue;  
int somenumbervalue;
```

- In the lines of code above, someNumberVariable and somenumbervariable are two separate variables.

```
double myDoubleValue;  
mydoublevalue = 100.1;
```

- These lines of code will NOT work. The variable declared is named my**D**ouble**V**alue not the same as my**d**ouble**v**alue.

Naming Variables

- Names must start with a letter, dollar sign, or underscore.
- Names may contain numbers, but cannot start with numbers!
- Aside from letters, dollar signs, underscores, and numbers, no other characters may be used.

`int someName;` Valid.

`int _someName;` Valid. Can start with underscore.
`int some_Name;` Valid. Can contain any underscores.

`int $someName;` Valid. Can start with dollar sign.
`int some$Name;` Valid. Can contain any dollar signs.

`int 3someName;` INVALID. Can't start with a number.
`int some3Name;` Valid. Can contain any numbers.

Naming Variables

- Names cannot contain spaces. Use underscores, if necessary.

<code>int some Name;</code>	INVALID
<code>int some_Name;</code>	Valid

- Names cannot be a Java keyword.

<code>int double;</code>	INVALID
--------------------------	----------------

Naming Variables

- Variable names normally begin with a lowercase letter; Class/Object names normally begin with an uppercase letter.
- “Camel-case” is the typical convention used for variable names.
 - For variable names that are multiple words long, the first letter of every subsequent word should be capitalized.
- Constants are typically ALL CAPS with underscore spacing.
- This is not mandatory, but these are conventions followed by just about every professional software engineer.

```
int bottlesOfBeerOnTheWall = 99;  
boolean hasGoneToTheMarket;  
final int PENNIES_IN_A_DOLLAR = 100;
```


Literals

- A ***literal*** is a source code representation of a fixed value.
 - It is represented without any computation.
- Sometimes referred to as *hard coded values*.

Literal
↓

```
int exampleLiteralInt = 5432;  
char exampleLiteralCharacter = 'C';
```

↑
Literal

Literals (Numeric)

- byte, int, short, and long literals can be expressed in
 - Decimal (Base 10)
 - Octal (Base 8), or
 - Hexadecimal (Base 16)
- Decimal Literal (No prefix): `int decimalNumber = 100;`
- Octal Literal (0 prefix): `int octalNumber = 0144;`
- Hexadecimal Literal (0x prefix): `int hexNumber = 0x64;`

For the purpose of this course, we will only be using decimal (base 10) values. It's good to know that other numeric literals exist, though.

Literals (Fractionals)

- Double literals (Nothing special needs to be done):

```
double exampleDoubleLiteral = 255.23;
```

- Float literals (Must add lowercase f to the end):

```
float myExampleFloat = 15.5f;
```

The compiler (like with many other programming languages/compilers) interprets literal decimal numbers as doubles by default. To differentiate float literals from double literals, float literals must end with a lowercase f.

Literals (Characters)

- char literals can be expressed as a character literal, a Unicode literal, or a decimal number.
- **Must be in single quotes!!**

```
char exampleCharLiteral = 'A';  
char exampleCharUnicodeLiteral = '\u0041';  
char exampleCharDecimalLiteral = 65;
```

- UTF-16 character table for reference:

<http://www.fileformat.info/info/charset/UTF-16/list.htm>

Any lab examples/homework will use character literals like exampleCharLiteral above.

Strings

- A ***String*** is an object (not a primitive) that contains a sequence of characters.
- The sequence of characters in a String can include any number of:
 - Letters
 - Numbers
 - Symbols
 - Spaces

Strings

- Since a String is an object, it provides ***methods*** (internal processes) we can call.
- A String provides many methods that make the manipulation of its data relatively painless.
 - We will only see a few today and see others later in the course.

Declaring a String object

- Strings, like primitives, are declared:
 - First stating the data type.
 - Then stating the variable name.

 **String** hello;

Capital S

Initializing a String object

- Use the assignment operator: =
- A ***String literal*** is any source code representation of a sequence of characters in double quotation marks.

The diagram shows two lines of Go code. The first line, `String hello;`, is annotated with a bracket above it labeled "Declaration". The second line, `hello = "Hello There!";`, is annotated with a bracket below it labeled "Initialization". A blue arrow points from the text "A String literal" to the double-quoted string `"Hello There!"` in the second line.

```
String hello;  
hello = "Hello There!";
```

Declaration

A String literal

Initialization

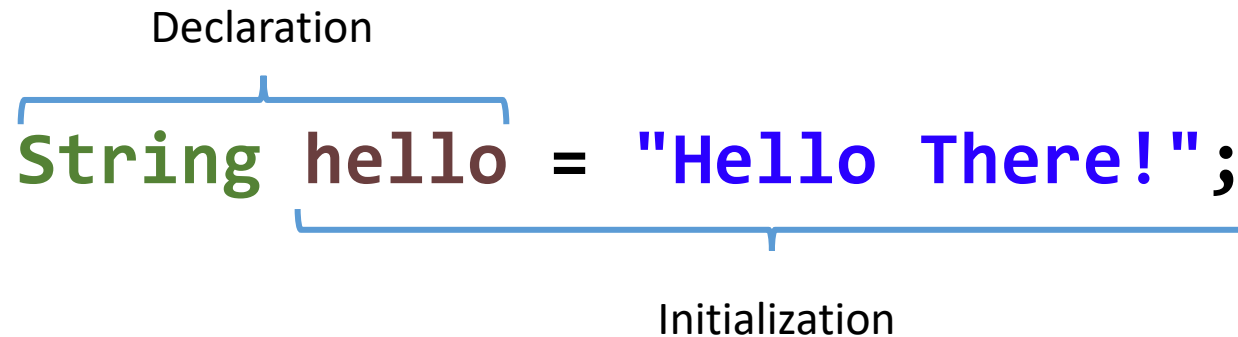
Declare and Initialize a String object

- Declaration and initialization can be done in one statement.

Declaration

```
String hello = "Hello There!";
```

Initialization



Reassigning Strings

- Use the assignment operator: =

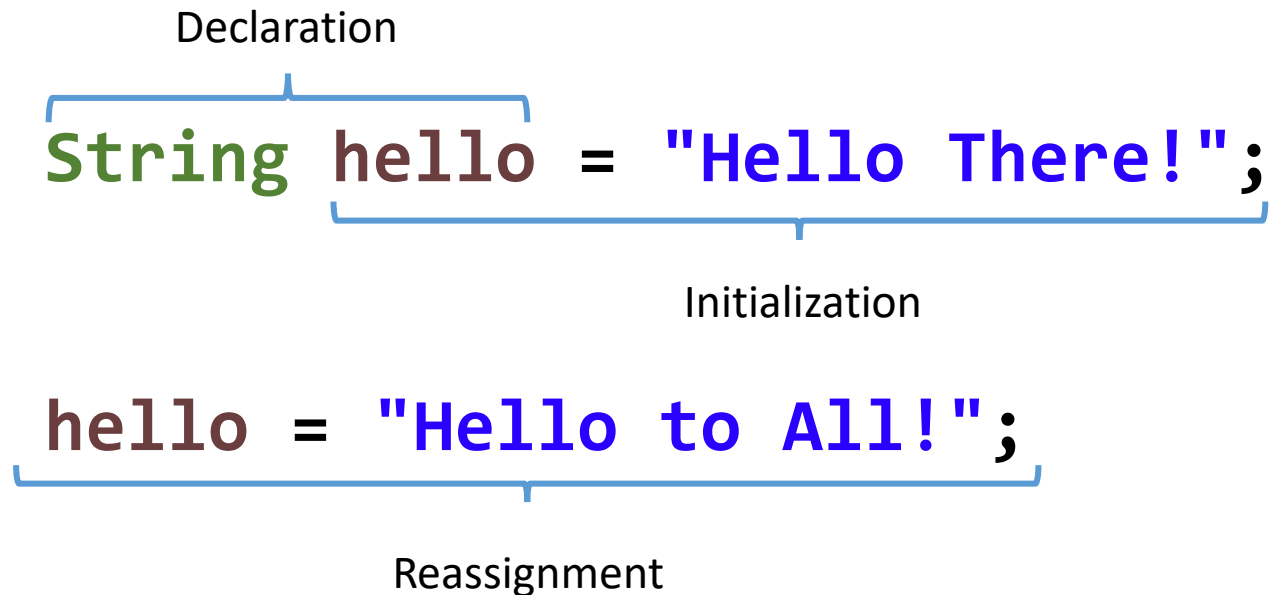
Declaration

```
String hello = "Hello There!";
```

Initialization

```
hello = "Hello to All!";
```

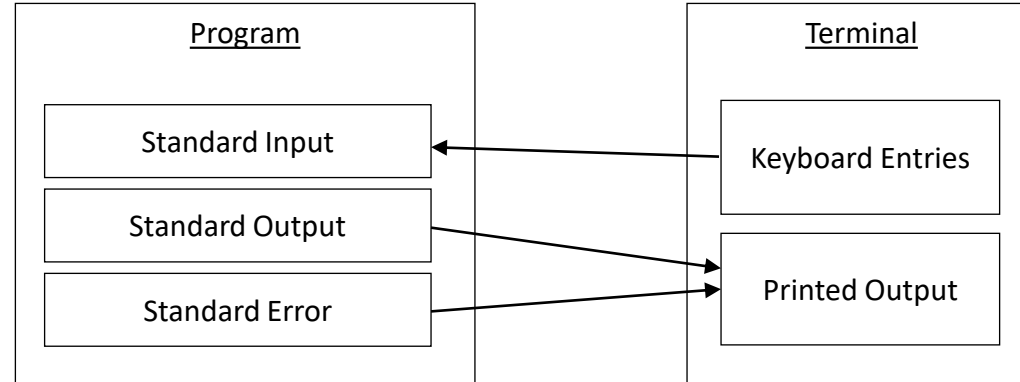
Reassignment

The diagram illustrates three stages of string handling in code. The first line, 'String hello = "Hello There!";', is annotated with a bracket above 'String' labeled 'Declaration' and a bracket below the entire line labeled 'Initialization'. The second line, 'hello = "Hello to All!";', is annotated with a bracket below the entire line labeled 'Reassignment'. The text 'String' is green, 'hello' is brown, and the string literals are blue.

Standard Streams

- A computer program uses data streams for handling incoming and outgoing data.
- Three standard streams
 - Standard Output (“stdout”, “standard out”)
 - Standard Input (“stdin”, “standard in”)
 - Standard Error (“stderr”, “standard error”)

Standard Streams



Standard Output

- ***Standard Output*** refers to the standard data stream used to print information/text to a terminal.
 - The term *console* or *terminal* is used to describe a text-only interface.
- The System object is provided by Java and allows access to standard output, input, and error.

System.out

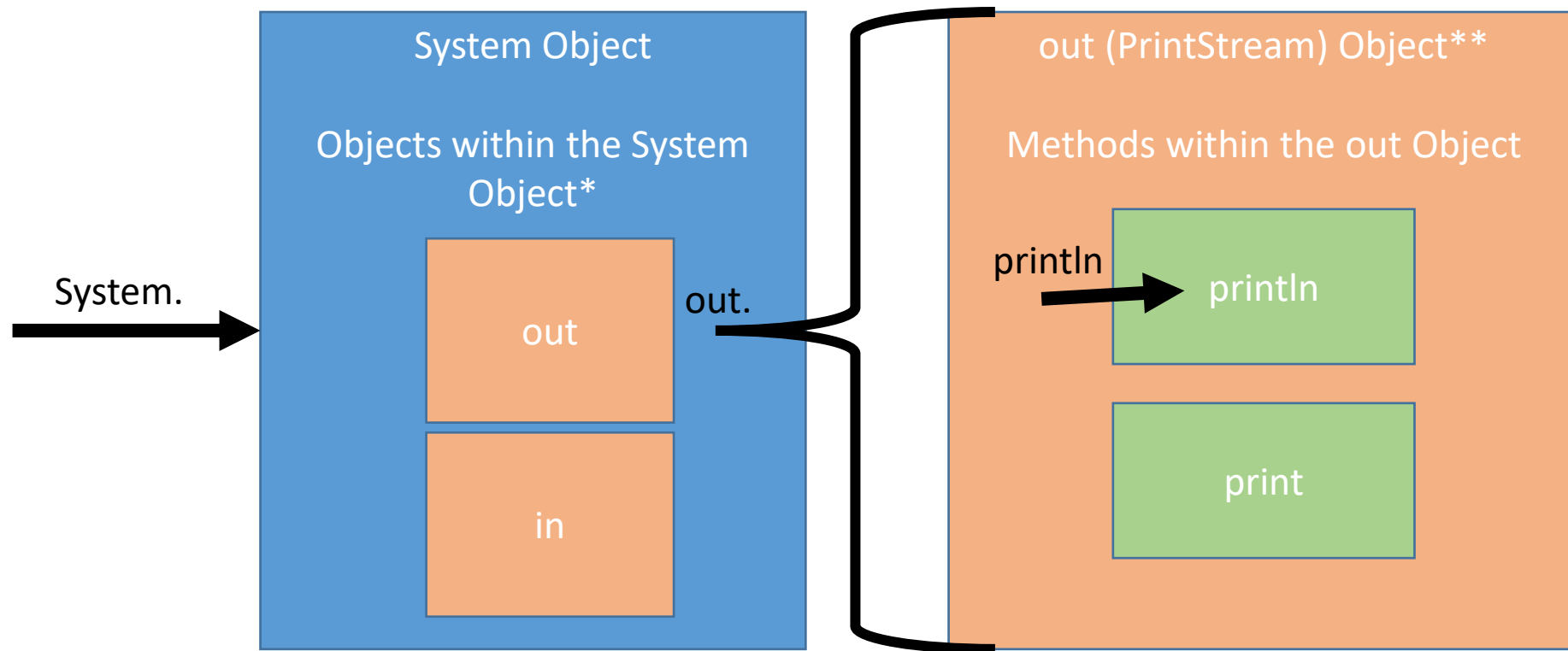
- The System object is the first external class we will be using in the course.
- Another way to look at this is that we are utilizing the objects and code contained in the System object within our own program.
- We need not be concerned with the System object's code. You can just call on it from within your own program whenever you need it.
- Inside of the System object, it has a `PrintStream*` object named “out”.
 - This object handles all default output, which is ultimately printed out to the terminal/console.
- To access an object (or *field*) within an object, we use ***dot notation***.
 - To access the “out” object from within the System object – **System.out**

* You don't have to worry about the technical specifics of a `PrintStream` is right now.

System.out

- There are two methods (sections of pre-written and reusable code) we will be using from System.out – print and println
- To access an object's methods, we again use dot notation - `System.out.println()`
 - We are basically saying, "In the System object's out object, execute the println method's code."
- The parentheses after the method name is for the parameter list.
 - Any data passed as a parameter to the print or println methods will be printed on the screen.

System.out.println() Walkthrough



* System object contains much more than just those two objects. This is just a generalization.

** A PrintStream contains much more than just those two methods. This is also just a generalization.

System.out.println()

- After printing the supplied information, the println method will return to the next line.

```
char gradeLetter1 = 'A';  
char gradeLetter2 = 'B';  
System.out.println(gradeLetter1);  
System.out.println(gradeLetter2);
```

A

B

System.out.print()

- Unlike the println method, the **print** method will stay on the same line.

```
char gradeLetter1 = 'A';  
char gradeLetter2 = 'B';  
System.out.print(gradeLetter1);  
System.out.print(gradeLetter2);
```

AB

Standard Input

- ***Standard Input*** refers to the data stream used to receive keyboard entries.
- Inside of the System object, it has a InputStream* object named “in”.
 - This object handles all default input, which is keyboard entries.
- To access the “in” object from within the System object – **System.in**

Scanner object

- A Scanner object provides an easy way to get input.
- Unlike other objects we have seen, Scanners will need to be *imported*.
 - Some objects provided by Java aren't available by default.
- You must include the following line at the very beginning of your source code (before the class header)

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

This allows the Scanner object to be accessible by your code. The Scanner object is located in Java's util (short for utility) package.

Scanner object

- Unlike other objects we have seen, Scanners will need to be ***instantiated***.
 - When you instantiate, you create a new instance of an object. It then becomes its own entity with its own data.
- Objects are instantiated using the **new** keyword.

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

When using a Scanner to get keyboard input, I usually name it “keyboard.”
You can name it whatever you like, though.

Scanner object

- When instantiating a Scanner, you MUST give it an input source.
- The System object's "in" object is an InputStream object, which a Scanner will accept.
- This allows us access to the default input, which in most cases is the keyboard.

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

Scanner object

```
String someString;  
Scanner keyboard = new Scanner(System.in);  
System.out.print("Enter a String: ");  
someString = keyboard.nextLine();  
System.out.println("The String you entered "  
                    + "is: " + someString);
```

Enter a String: Hello World!

The String you entered is: Hello World!

Scanner object

- A Scanner object can be used more than once:

```
String someString;  
String someString2;  
Scanner keyboard = new Scanner(System.in);  
System.out.print("Enter a String: ");  
someString = keyboard.nextLine();  
System.out.print("Enter another String: ");  
someString2 = keyboard.nextLine();  
System.out.println("The String you entered "  
                    + "is: " + someString);  
System.out.println("The other string you entered "  
                    + "is: " + someString2);
```


Scanner object methods*

Method	Return Type	Description	Possible Exceptions**
nextLine()	String	Returns the next line of input in String form.	NoSuchElementException IllegalStateException

*- There are more than just this one method. We will use more as the semester progresses.

** - It is unlikely you will see these exceptions when using System.in as the input stream.

Comments

- Comments are normally used to document your code.
- This makes it easy to:
 - Leave notes to yourself.
 - Leave notes to other programmers who may work on your code.
 - Describe what a section or line of code does (it may not always be obvious)
- Alternatively, comments are useful for omitting single or multiple lines when debugging.

Comments

- Single line comments begin with //

```
//Single line comment
```

- Multiple line comments begin with /* and end with */

```
/* Everything between slash-asterisk  
and asterisk-slash  
will be  
ignored*/
```

- Comments are entirely ignored by the compiler. You can type whatever you want in a comment.

Comments

`int i = 10; //Comments can be left after a statement.`

- Omit an entire line/statement by adding `//` at the beginning:

```
//int j = 10;
```

- Omit multiple lines/statements by adding `//` at the beginning of each, or use multi-line comments:

```
//int j = 10;  
//int k = 15;
```

```
/*int j = 10;  
int k = 15;*/
```

Comments

- Comments can be useful when debugging and testing.
- They allow you to omit sections of code without actually deleting them.
- You can later uncomment them, or delete them once you are confident you no longer need the lines any more.
 - If you leave in commented lines of code, you will normally leave another comment explaining why you left them in.

Comments (cont.)

```
int length = 5;  
int width = 10;  
int area = length * width;  
  
//Uncomment for debugging the area value  
/*System.out.print("area is ");  
System.out.println(area);*/
```

Typical Convention for Commenting

```
/**  
 * This program will ask users for input  
 * and then display some output.  
 */  
public static void main(String[] args) {  
    int temp = 10; //Initialize temperature to ten.  
    ...  
}
```