Basic Computation – Part 2

Forest Agostinelli
University of South Carolina

Outline

- Classes
- Strings
- Processing input text
- Parsing example

Class Types

- Class types group together data with functionality (methods)
- Classes create instances of Objects
- Separated by Reference and Contents
  - Reference is the memory address that "points" to the object's contents in memory
  - A reference is the value stored in the identifier
  - Contents contain the data and functionality

Memory

Identifier	Contents	Byte Address
…	…	…
objectID	28	14
..	…	…
objectID.data01	4	28
objectID.data02	3.0	32
objectID.method01()	-	-
objectID.method02()	-	-
…	…	…

Class Construction

- Objects must be *constructed* before used
  - Default value for class types is NULL
  - NULL means “nothing” as the object does not exist
  - Cannot use a NULL object
  - Reserved word “new” is used to construct instances of most class types, but not usually for Strings
- Methods provide functionality for an object
  - It’s what the object can do
  - Reusing code
- Methods are *called* by using the object’s identifier, followed by a dot “.”, followed by the method name an arguments

Syntax for Calling a Method

```
<<identifier>>.<<method name>>(<<arguments>>);
```

Outline

- Classes
- Strings
- Processing input text
- Parsing example

Strings

- Class type
  - Data = Array of Characters
  - Methods = Built-in Functionality
- Denoted by double quotes (“”)
  - Single Characters are single quotes (’)
- Used to group together single characters into words and phrases
  - Useful for Outputting and Formatting Data
  - Useful for Inputting Data as words or sentences

<u>Syntax</u>

```
String <<identifier>>;//Declare a String
//Assigning a String Value
<<identifier>> = “<<String Value>>”;
```

Strings

- Array of Characters
  - Contiguous Collection of Characters
  - Individual Characters can be accessed by an "index"
  - Indices always start from 0 to Length - 1

<u>Example String</u>

```
String str = "abcdefg";
```

<u>String as an Array</u>

Index	0	1	2	3	4	5	6
Value	'a'	'b'	'c'	'd'	'e'	'f'	'g'

String Representation in Memory

- Object type
- Array of characters

Examples

```
String str = “abcd”;
```

Memory

Identifier	Contents	Byte Address
…	…	…
str	**64**	28
…	…	…
str[0]	‘a’	64
str[1]	‘b’	66
str[2]	‘c’	68
str[3]	‘d’	70
…	…	…

String Operations

- The plus (+) operator concatenates a value with a String
  - Not the same as the mathematical “+”
- Useful methods
  - length()
  - charAt(index)
  - substring(startIndex)
  - substring(startIndex, endIndex)
  - toUpperCase()
  - toLowerCase()
  - split(regular expression)

Examples

```
String str = “abcdefg”;
System.out.println(str.charAt(0));
String str2 = str.substring(2,5);
System.out.println(str2);
```

Console

```
a
cde
```

String Operations

- See JDK

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.

String Operations

Method overloading. Java distinguishes between these methods using their arguments

`lastIndexOf` (a_*string*)	`int`	`i = s.lastIndexOf("a");` `// 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();` `// i = 4`	Returns the length of this string.
`toLower Case()`	`String`	`s2 = s.toLowerCase();` `// 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();` `// 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');` `// 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);` `// 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);` `// 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  ";`
		`s2 = s.trim();`
		`// s2 = "Java"`
			Returns a new string having the same characters as this string, but with leading and trailing whitespace removed.

Review: Java Types

- **Primitive Types**
  - Atomic/irreducible
  - No methods
  - Identifiers contain the assigned value
- **Class Types**
  - Are composed of other types
  - Can have class methods
  - Identifiers are references to the class object

Review: Primitive Types

Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	**4 bytes**	**Stores whole numbers from -2,147,483,648 to 2,147,483,647**
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	**8 bytes**	**Stores fractional numbers. Sufficient for storing 15 decimal digits**
boolean	**1 bit**	**Stores true or false values**
char	**2 bytes**	**Stores a single character/letter or ASCII values**

- The name “floating-point” comes from the fact that the decimal point can be made to “float” to different places in a number in scientific notation
- What explains the range of the primitive number types?
  - Why the difference of 1 in the range of positive and negative numbers?
    - See two’s complement
- In what situation would using a byte be preferable to an int?
- What happens if one has a byte that is 127 and then adds 1 to it?

Quick Quiz

```
String str1 = new String("abc");
String str2 = new String("abc");
String str3 = str1;
System.out.println(str1 == str2);
System.out.println(str3 == str1);
System.out.println(str3 == str2);
System.out.println(str1.equals(str2));
System.out.println(str1.charAt(0) == str2.charAt(0));
```

- true of false?

Quick Quiz

```
String str3 = new String("a");
String str4 = new String("a");
System.out.println(str3 == str4);
System.out.println(str3.equals(str4));
System.out.println(str3.charAt(0) == str4.charAt(0));
```

- true or false?

Interned Strings

```
String str5 = "abc";
String str6 = "abc";
System.out.println(str5 == str6);
System.out.println(str5.charAt(0) == str6.charAt(0));
```

Output

```
true
true
```

- Ensures that all strings that have the same contents refer to the same object
- Can be used to save memory in certain situations
- It is still good practice to always use .equals() to check for equality!
  - Many people spend hours debugging a problem because they used == when they should have used .equals.

Outline

- Classes
- Strings
- Processing input text
- Parsing example

Escape Characters

- Used to better format Strings
- Considered Single Characters
  - Despite there are two individual characters
- Starts with a “\”
- \” - Double Quote
- \’ - Single Quote
- \\ - Backslash
- \n – New Line. Go to beginning of Next line
- \r – Carriage Return. Go to beginning of the Current line
- \t – Tab. Add space until next tab stop

Examples

```
String str = “Hello\n\”World\””;
System.out.println(str);
```

Console

```
Hello
“World”
```

Scanner Class

- Class Type
- Used to “Scan” or “Read”
  - Standard System Input “System.in” (Console)
  - Strings
  - Files
  - Network Traffic
- Must import type Scanner from “java.util” package
  - import java.util.Scanner;
- Before using it must be both Declared and Constructed
  - The “ARGS” part is the item the Scanner will process. It can be the System input, Strings, Files, etc.

<u>Syntax</u>

```
//Declaring and Constructing a Scanner
Scanner <<identifier>> = new Scanner(<<ARGS>>);
```

<u>Example</u>

```
//Declaring and constructing a Scanner for
//Console (System.in)
Scanner keyboard = new Scanner(System.in);
```

Scanner Class

- Once a Scanner has been declared and constructed it can be used by calling its various methods
- Scanner uses *delimiters*
  - Separates information by Special Characters
  - Assumed to be any kind of space unless otherwise declared
  - Types of spaces include
    - Single Spaces
    - Multiple Spaces
    - End Line / Carriage Returns
    - Tabs

Examples

```
Scanner keyboard = new Scanner(System.in);
String name = keyboard.nextLine();
int i = keyboard.nextInt();
keyboard.nextLine();//Useful “fix-up”
double j = keyboard.nextDouble();
keyboard.nextLine();//Useful “fix-up”
System.out.println(name+ “ “ + i + “ “ + j);
```

Console

```
JJ
64
3.14
JJ 64 3.14
```

Scanner Methods

Method Name	Description	Example

next()	Returns a String value up to but not including the first delimiter character	//Assume user enters “1234 3.14 true asdf”
		String str = keyboard.next();
		//str is “1234”

nextLine()	Returns a String value up to but not including the line terminator ‘\n’	//Assume user enters “1234 3.14 true asdf”
		String str = keyboard.nextLine();
		//str is “1234 3.14 true asdf”

nextInt()	Returns the first instance of an integer value. All other characters and delimiters are ignored.	//Assume user enters “1234 3.14 true asdf”
		int i = keyboard.nextInt();
		//int i is 1234

nextDouble()	Returns the first instance of a double value. All other characters and delimiters are ignored.	//Assume user enters “1234 3.14 true asdf”
		dobule j = keyboard.nextDouble();
		//double j is 3.14

nextBoolean()	Returns the first instance of a Boolean value. All other characters and delimiters are ignored.	//Assume user enters “1234 3.14 true asdf”
		boolean b = keyboard.nextBoolean();
		//Boolean b is true

Wrapper Classes

- Classes that “Wrap” or provide functionality to primitive types
- Can be used to convert a String into a primitive type
- Commonly Used
  - Integer.parseInt(<<String>>);
  - Double.parseDouble(<<String>>);
  - Boolean.parseBoolean(<<String>>);

Examples

```
String str = “256”;
int i = Integer.parseInt(str);
i *= 2;
System.out.println(i);
```

Console

```
512
```

Outline

- Classes
- Strings
- Processing input text
- Parsing example

Parsing Example

- We are going to parse input that is delimited by a single space
  - For example: <<name>> <<ID>> <<X>>

Parsing Example

`input =`

0	1	2	3	4	5	6	7	8	9
A	D	A		2	3		2	.	2

Current Line of Code

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

Parsing Example

`input =`

0	1	2	3	4	5	6	7	8	9
A	D	A		2	3		2	.	2

`copyInput =`

0	1	2	3	4	5	6	7	8	9
A	D	A		2	3		2	.	2

Current Line of Code

```
String copyInput = input;
```

Parsing Example

`input =`

0	1	2	3	4	5	6	7	8	9
A	D	A		2	3		2	.	2

`copyInput =`

0	1	2	3	4	5	6	7	8	9
A	D	A		2	3		2	.	2

`workingIndex = 3`

Current Line of Code

```
int workingIndex = copyInput.indexOf(" ");
```

Parsing Example

input =

0	1	2	3	4	5	6	7	8	9
A	D	A		2	3		2	.	2

copyInput =

0	1	2	3	4	5	6	7	8	9
A	D	A		2	3		2	.	2

workingIndex = 3

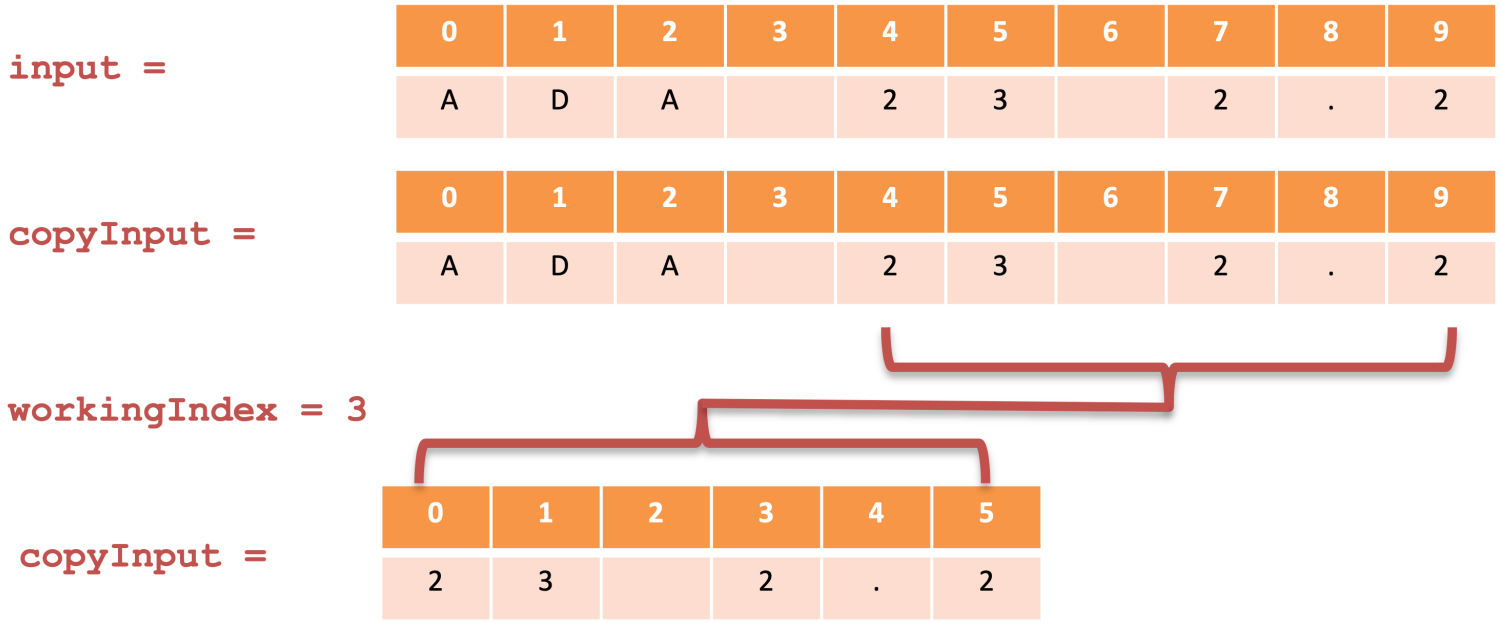
name =

0	1	2
A	D	A

Current Line of Code

```
String name = copyInput.substring(0,workingIndex);
```

Parsing Example



Current Line of Code

```
copyInput = copyInput.substring(workingIndex+1);
```

Parsing Example

- Connection to homework?
- Homework will not use space delimiters
  - 1101
  - 1001
- What about for inputs of variable length for extra credit?

PlayerParser.java

```java
/*
 * Written by JJ Shepherd
 */
import java.util.Scanner;
public class PlayerParser {

    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        //<<name>> <<id>> <<x>> <<y>> <<z>>\n
        System.out.println("Enter the player's name followed by their model id (int), x,y,z position (double)");
        String input = keyboard.nextLine();
        String copyInput = input;

        int workingIndex = copyInput.indexOf(" ");
        String name = copyInput.substring(0,workingIndex);
        copyInput = copyInput.substring(workingIndex+1);

        workingIndex = copyInput.indexOf(" ");
        String sModelID = copyInput.substring(0,workingIndex);
        int iModelID = Integer.parseInt(sModelID);
        copyInput = copyInput.substring(workingIndex+1);

        workingIndex = copyInput.indexOf(" ");
        String sX = copyInput.substring(0,workingIndex);
        double dX = Double.parseDouble(sX);
        copyInput = copyInput.substring(workingIndex+1);

        workingIndex = copyInput.indexOf(" ");
        String sY = copyInput.substring(0,workingIndex);
        double dY = Double.parseDouble(sY);
        copyInput = copyInput.substring(workingIndex+1);

        String sZ = copyInput.substring(0,workingIndex);
        double dZ = Double.parseDouble(sZ);

        System.out.println("The player "+name+" has a model id of "+iModelID+" and is located at\n"+dX+"\t"+dY+"\t"+dZ);
    }

}
```

Good Programming Practices

- Documentation and Style is important
  - Most programs are modified over time to respond to new requirements
  - Programs that are easy to read and understand are easy to modify
  - You have to be able to read it in order to debug it
- Meaningful Identifiers
  - Identifiers should suggest its use
  - Stick to common conventions

- Commenting
  - Self documenting with Clean Style is best
  - Comments are written as needed
  - Used by programmers to explain code, but ignored by the compiler
  - Include your name at the beginning of every file
  - It's good to write an explanatory comment at the beginning of the file
- Indentation
  - Use indentation to "line-up" code within their respective bodies
  - Clearly indicates "nested" statements