Topic 9
Using Objects, Interactive Programs and Loop Techniques

"There are only two kinds of programming languages: those people always [complain] about and those nobody uses."

— Bjarne Stroustroup, creator of C++

Based on slides for Building Java Programs by Reges/Stepp, found at http://faculty.washington.edu/stepp/book/

Objects and Classes

Objects

- So far, we have seen:
  - **methods**, which represent behavior
  - **variables**, which represent data
  - **types**, which represent categories of data

- In Java and other "object-oriented" programming languages, it is possible to create new types that are combinations of the existing primitive types.
  - Such types are called **object types** or **reference types**.
  - An **object** is an entity that contains data and behavior.
    - There are variables inside the object, storing its data.
    - There are methods inside the object, representing its behavior.

- Today, we will learn how to communicate with certain objects that exist in Java.

Constructing objects

- **construct**: To create a new object.
  - Objects are constructed with the `new` keyword.
  - Most objects other than Strings must be constructed before they can be used.

- Constructing objects, general syntax:
  <type> ***<name>*** `=` `new` ***<type>*** `(` ***<parameters>*** `);`

  - Examples:

```
BigInteger rhs = new BigInteger("123456123456");
Color orange = new Color(255, 128, 0);
Point origin = new Point(0, 0);
Polygon poly = new Polygon();
```

Reminder: primitive variables

- We now need to examine some important differences between the behavior of objects and primitive values.
- We saw with primitive variables that modifying the value of one variable does not modify the value of another.
- When one variable is assigned to another, the value is copied.
  - Example:

```
int x = 5;
int y = x;    // x = 5, y = 5
y = 17;       // x = 5, y = 17
x = 8;        // x = 8, y = 17
```

Reference variables

- However, objects behave differently than primitives.
  - When working with objects, we have to understand the distinction between an object, and the variable that stores it.
  - Variables of object types are called **reference variables**.
  - Reference variables do not actually store an object; they store the address of an object's location in the computer memory.
  - If two reference variables are assigned to refer to the same object, the object is *not* copied; both variables literally share the same object. Calling a method on either variable will modify the same object.

  - Example:

```
Point p1 = new Point(10, 20); // x and y coords.
Point p2 = p1; // does not create a new Point
p2.move(5, 10)' // new x and y coordinates
System.out.println( "x: " + p1.getX() +
        ", y: " + p1.getY() );
```

Modifying parameters

- When we call a method and pass primitive variables' values as parameters, it is legal to assign new values to the parameters inside the method.
  - But this does not affect the value of the variable that was passed, because its value was copied.
  - Example:

```
public static void main(String[] args) {
    int x = 1;
    foo(x);
    System.out.println(x);    // output: 1
}

public static void foo(int x) {
    x = 2;
}
```

value 1 is copied into parameter

parameter's value is changed to 2
(variable `x` in `main` is unaffected)

Objects as parameters

- When an object is passed as a parameter, it is not copied. It is shared between the original variable and the method's parameter.
  - If a method is called on the parameter, it *will* affect the original object that was passed to the method.
  - Example:

```
public static void main(String[] args) {
    Point p1 = new Point(5, 10);
    System.out.println( p1.toString() );
    foo(p);
    System.out.println( p1.toString() );
}

public static void foo(Point p) {
    System.out.println( p.toString() );
    p.move(1, 2);
    System.out.println( p.toString() );
}
```

Strings

- One of the most common types of objects in Java is type String.
  - **String**: A sequence of text characters.
  - Object data types' names are usually uppercase (String), unlike primitives (int).

- String variables can be declared and assigned, just like primitive values:
  - `String` ***<name>*** `=` ***"<text>";***
  - `String` ***<name>*** `=` ***<expression that produces a String>;***

  - Examples:

```
String name = "Tom Danielson";

int x = 3, y = 5;
String point = "(" + x + ", " + y + ")";
```

Indexes

- The characters in a String are each internally numbered with an *index*, starting with 0 for the first character:
  - Example:

```
String name = "M. Scott";
```

name -->

0	1	2	3	4	5	6	7
'M'	'.'	' '	'S'	'c'	'o'	't'	't'

- Individual text characters are represented by a primitive type called `char`. Literal `char` values are surrounded with apostrophe (single-quote) marks, such as `'a'` or `'4'`.
  - An escape sequence can be represented as a `char`, such as `'\n'` (new-line character) or `'\''` (apostrophe).

Calling methods of Strings

- Strings are objects that contain methods.
  - A String contains code inside it that can manipulate or process the String in several useful ways.
  - When we call a method of a String, we don't just write the method's name. We also have to write which String we want to execute the method. The results will be different from one String to another.
- Calling a method of an object, general syntax:
  <name> **.** ***<methodName>*** **(** ***<parameters>*** **)**
  - Examples:

```
String name = "Mike";
System.out.println(name.toUpperCase());     // MIKE

String name2 = "Mike Scott";
System.out.println(name2.length());          // 10
```

String methods

- Here are several of the most useful String methods:

Method name	Description
charAt(*index*)	character at a specific index
indexOf(*String*)	index where the start of the given String appears in this String (-1 if it is not there)
length()	number of characters in this String
substring(*index1*, *index2*)	the characters from *index1* to just before *index2*
toLowerCase()	a new String with all lowercase letters
toUpperCase()	a new String with all uppercase letters

String method examples

```
//      index 012345678901
String s1 = "Olivia Scott";
String s2 = "Isabelle Scott";
System.out.println(s1.length());          // 12
System.out.println(s1.indexOf("i"));      // 2
System.out.println(s1.substring(1, 4));   // liv

String s3 = s2.toUpperCase();
System.out.println(s3.substring(6, 10)); // LE S

String s4 = s1.substring(0, 6);
System.out.println(s4.toLowerCase());  // olivia
```

Methods that return values

- The methods of `String` objects do not print their results to the console.
  - Instead, a call to one of these methods can be used as an expression or part of an expression.

- Recall: **return value**: A value that is produced by a call to a method, and can be used in an expression.
  - Return values are the opposite of parameters. Parameters pass information inward into a method from the caller. Return values give information outward from the method to the caller.
  - The methods of String objects produce (or *return*) a result which is either a new String or a number, depending on the method.
  - The result can be used in a larger expression, stored in a variable, or printed to the console.

Return values example

```
String str = "Kelly Scott";
int len = 2 * str.length() + 1;
System.out.println(len);
  // 23

String first = str.substring(0, 5);
System.out.println("first name is " + first);  // Kelly
```

- What expression would produce the first letter of the String? The last letter?

- What expression would trim any String, not just the one above, to its first word?
  - Does our answer assume anything about the letters in the String?

Modify and reassign

- The various methods that modify Strings return a new String with the new contents.
  - They don't modify the existing String.
- Just like `int` or `double` variables, `String` variables do not change when used in an expression unless you reassign them:
  - Bad Example:

```
String s = "I get it";
s.toUpperCase();
System.out.println(s);    // I get it
```

  - Better Code:

```
String s = "I get it";
s = s.toUpperCase();
System.out.println(s);
```

- Equivalent with an `int`:

```
int x = 3;
x + 1;
System.out.println(x); // 3
```

- Equivalent with an `int`:

```
int x = 3;
x = x + 1;
System.out.println(x); // 4
```

Strings vs. other objects

- Strings are extremely useful objects, but they behave differently than most objects in Java.
  - Strings are created differently than most objects.
    We don't have to use the `new` keyword when constructing Strings.
    (This is because Sun felt that Strings were so important, they should be integrated into the language with a shorter syntax.)
  - Strings can't be modified without reassigning them.
    - An object that cannot be changed after construction is sometimes called an *immutable* object.
  - It is harder to visualize Strings as having data and behavior, but a String's data is its characters, and its behavior is the methods like `toUpperCase` and `length` that manipulate or examine those characters.

Interactive Programs and Scanner Objects

Interactive programs

- We have written several programs that print *output* to the console.
- It is also possible to read text *input* from the console.
  - The user running the program types the input into the console.
  - We can capture the input and use it as data in our program.
- A program that processes input from the user is called an *interactive program*.
- Interactive programs can be challenging:
  - Computers and users think in very different ways.
  - Users tend to do unpredictable and unexpected things!
    - The Mom test.

Input and System.in

- We have now seen code that communicates with objects.
  - Example objects: `Point, String, BigInteger`

- When we print text output to the console, we communicate with an object named **System.out** .
  - We call the `println` (or `print`) method of the `System.out` object to print a message to the console.

- The object that holds the user's console input is named **System.in** . But it is not as easy to use...

Scanner

- Since `System.in` is not easy to use by itself, we will use a second object of a type named `Scanner` to help.
  - Once we construct the Scanner object, we can ask it to read various kinds of input from the console.

- Constructing a Scanner object to read console input:

```
Scanner <name> = new Scanner(System.in);
```

  - Example:

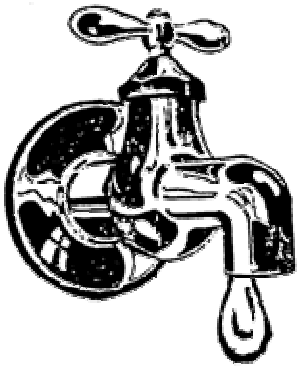
```
Scanner console = new Scanner(System.in);
```

- When you use Scanner, you must include this line:

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

Scanner as data source

- Think of a Scanner like a faucet or showerhead that can be attached to a source of 'water' (data). In our case, the source of that data is `System.in`.
  - Like a faucet must be connected to a water source, the Scanner must be connected to a data source by writing (`System.in`) when constructing it.



Scanner methods

- Methods of `Scanner` that we will use in the near future:

Method	Description
`next()`	reads and returns next token as a `String`
`nextDouble()`	reads and returns next token as a `double`
`nextInt()`	reads and returns next token as an `int`
`nextLine()`	reads and returns next entire line of input as a `String`

- Each of these methods causes your program to pause until the user has typed input and pressed Enter, then it *returns* the typed value to your program.

Example Scanner usage

```
import java.util.*;
public class ReadSomeInput {
    public static void main(String[] args) {
        System.out.print("How old are you? ");
        int age;

        Scanner console = new Scanner(System.in);
        age = console.nextInt();

        System.out.println("Wow, you're " + age);
        System.out.println("That's old!");
    }
}
```

- Output (user input underlined):

```
How old are you? 14
Wow, you're 14
That's old!
```

Scanning tokens

- **token**: A unit of user input. Tokens are separated by whitespace (spaces, tabs, new lines).
  - Example: If the user types the following:

```
23   3.14 John Smith   "Hello world"
          45.2    19
```

  - The tokens in the input are the following, and can be interpreted as the given types:

Token	Type(s)
`23`	int, double, String
`3.14`	double, String
`John`	String
`Smith`	String
`"Hello`	String
`world"`	String
`45.2`	double, String
`19`	int, double, String

Consuming input

- When the Scanner's methods are called, the Scanner reads and returns the next input value to our program.
  - If the type of the token isn't compatible with the type we requested, the program crashes.
- Imagine the Scanner as having an invisible cursor that moves through all the user input.
  - As the scanner reads each input value, it advances forward through the user input until it has passed the given token.
  - This is called *consuming* the input token.

```
double pi = console.nextDouble();
            // 3.14
23\t3.14 John Smith\t"Hello world"\n\t\t45.2      19
   ^
```

- 23 has been consumed (used), 3.14 is the next token, all the text in bold is unconsumed

Consume input example

- Example: If the following input from the user has been typed,

```
23   3.14 John Smith   "Hello world"
          45.2   19
```

The Scanner views it as a linear stream of data, like the following:

```
23\t3.14 John Smith\t"Hello world"\n\t\t45.2  19\n
```

The Scanner positions its 'cursor' at the start of the user input:

```
23\t3.14 John Smith\t"Hello world"\n\t\t45.2  19\n
^
```

- As we call the various `next` methods on the Scanner, the scanner moves forward:

```
int x = console.nextInt();                          // 23
23\t3.14 John Smith\t"Hello world"\n\t\t45.2  19\n
  ^

double pi = console.nextDouble();                   // 3.14
23\t3.14 John Smith\t"Hello world"\n\t\t45.2  19\n
       ^

String word = console.next();                       // "John"
23\t3.14 John Smith\t"Hello world"\n\t\t45.2  19\n
            ^
```

Line-based input

- The Scanner's `nextLine` method consumes and returns an entire line of input as a String.
  - The Scanner moves its cursor from its current position until it sees a `\n` new line character, and returns all text that was found.
    - The new line character is consumed but not returned.

- Example:

```
23   3.14 John Smith   "Hello world"
          45.2 19
```

```
String line1 = console.nextLine();
23\t3.14 John Smith\t"Hello world"\n\t\t45.2  19\n
                                  ^
```

```
String line2 = console.nextLine();
23\t3.14 John Smith\t"Hello world"\n\t\t45.2  19\n
                                          ^
```

Mixing line-based with tokens

- It is not generally recommended to use `nextLine` in combination with the other `next__` methods, because confusing results occur.

```
23   3.14
Joe   "Hello world"
                45.2  19
```

```
int n = console.nextInt();                                  // 23
23\t3.14\nJoe\t"Hello world"\n\t\t45.2  19\n
  ^
double x = console.nextDouble();                            // 3.14
23\t3.14\nJoe\t"Hello world"\n\t\t45.2  19\n
        ^

// User intends to grab the Joe   "Hello world" line
// but instead receives an empty line!
String line = console.nextLine();                           // ""
23\t3.14\nJoe\t"Hello world"\n\t\t45.2  19\n
          ^
// Calling nextLine again will get the line we wanted.
String line2 = console.nextLine();
// "Joe\t\"Hello world\""
23\t3.14\nJoe\t"Hello world"\n\t\t45.2  19\n
                            ^
```

Line-and-token example

- Here's another example of the confusing behavior:

```
Scanner console = new Scanner(System.in);
System.out.print("Enter your age: ");
int age = console.nextInt();

System.out.print("Now enter your name: ");
String name = console.nextLine();

System.out.println(name + " is " + age + " years old.");
```

Log of execution (user input underlined):

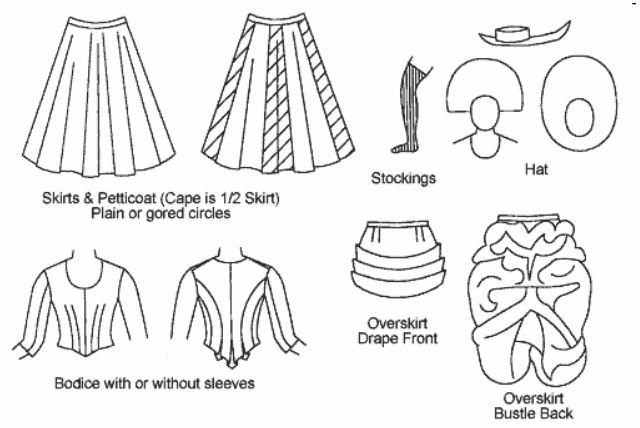
```
Enter your age: 13
Now enter your name: Olivia Scott
 is 13 years old.
```

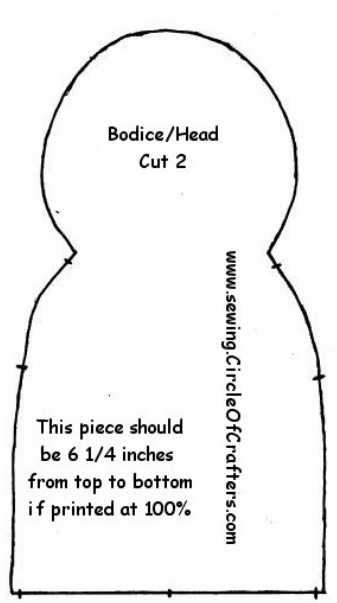
- Why?
  - User's overall input: **12\nOlivia**
  - After nextInt(): *12*\nOlivia Scott
    ^
  - After nextLine(): *12\n*Olivia Scott
    ^

Loop Techniques

Some loop patterns

- As you program you will see common patterns, things you need to do over and over again in various programs
- 2 common programming patterns are
  - cumulative sum
  - fencepost or loop and a half problems





Adding many numbers

- Consider the following code to read three values from the user and add them together:

```
Scanner console = new Scanner(System.in);
System.out.print("Type a number: ");
int num1 = console.nextInt();

System.out.print("Type a number: ");
int num2 = console.nextInt();

System.out.print("Type a number: ");
int num3 = console.nextInt();

int sum = num1 + num2 + num3;

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

A cumulative sum

- You may have observed that the variables num1, num2, and num3 are unnecessary. The code can be improved:

```
Scanner console = new Scanner(System.in);
System.out.print("Type a number: ");
int sum = console.nextInt();

System.out.print("Type a number: ");
sum += console.nextInt();

System.out.print("Type a number: ");
sum += console.nextInt();

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

- **cumulative sum**: A sum variable that keeps a total-in-progress and is updated many times until the task of summing is finished.
  - The variable sum in the above code now represents a cumulative sum.

Failed cumulative sum loop

- How would we modify the preceding code to sum 100 numbers?
  - Creating 100 cut-and-paste copies of the same code would be redundant and unwieldy.
  - Here's a failed attempt to write a loop that adds 100 numbers.
  - It actually declares 100 variables named sum, each of which is created and destroyed in a single pass of the for loop.
  - None of the sum variables lives on after the for loop, so the last line of code is a compiler error.

```
Scanner console = new Scanner(System.in);
for (int i = 1; i <= 100; i++) {
    int sum = 0;
    System.out.print("Type a number: ");
    sum += console.nextInt();
}

// sum is undefined here :-(
System.out.println("The sum is " + sum);
```

Fixed cumulative sum loop

- A corrected version of the sum loop code:

```
Scanner console = new Scanner(System.in);
int sum = 0;
for (int i = 1; i <= 100; i++) {
    System.out.print("Type a number: ");
    sum += console.nextInt();
}
System.out.println("The sum is " + sum);
```

  - Cumulative sum variables must always be declared *outside* the loops that update them, so that they will continue to live after the loop is finished.

User-guided sum, average

- The user's input can guide the number of times the cumulative sum loop repeats:

```
Scanner console = new Scanner(System.in);
System.out.print("How many numbers to average? ");
int count = console.nextInt();

int sum = 0;
for (int i = 1; i <= count; i++) {
    System.out.print("Type a number: ");
    sum += console.nextInt();
}

double average = (double) sum / count;
System.out.println("The average is " + average);
```

Variation: cumulative product

- The same idea can be used with other operators, such as multiplication which produces a cumulative product:

```
Scanner console = new Scanner(System.in);
System.out.print("Raise 2 to what power? ");
int exponent = console.nextInt();

int product = 1;
for (int i = 1; i <= exponent; i++) {
    product *= 2;
}
System.out.println("2 to the " + exponent + " = "
                   + product);
```

  - Exercise: Change the above code so that it also prompts for the base, instead of always using 2.
  - Exercise: Make the code to compute the powers into a method which accepts a base *a* and exponent *b* as parameters and returns a^b .

The fencepost problem

- Problem: Write a static method named printNumbers that prints each number from 1 to a given maximum, which is passed as a parameter, separated by commas. Assume that the maximum number passed in is greater than 0. For example, the method call:

  printNumbers(5)

  should print:

  1, 2, 3, 4, 5

- Let's write a solution to this problem...

Flawed solutions

```
public static void printNumbers(int max) {
    for (int i = 1; i <= max; i++) {
        System.out.print(i + ", ");
    }
    System.out.println();  // to end the line of output
}
```

OUTPUT from printNumbers(5):

```
1, 2, 3, 4, 5,
```

- An incorrect attempt to fix the code:

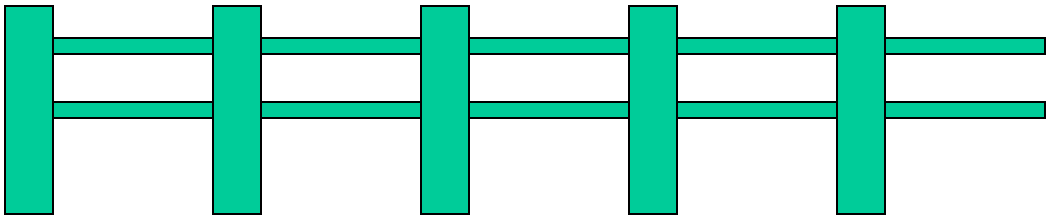
```
public static void printNumbers(int max) {
    for (int i = 1; i <= max; i++) {
        System.out.print(", " + i);
    }
    System.out.println();  // to end the line of output
}
```

OUTPUT from printNumbers(5):

```
, 1, 2, 3, 4, 5
```

Fence posts

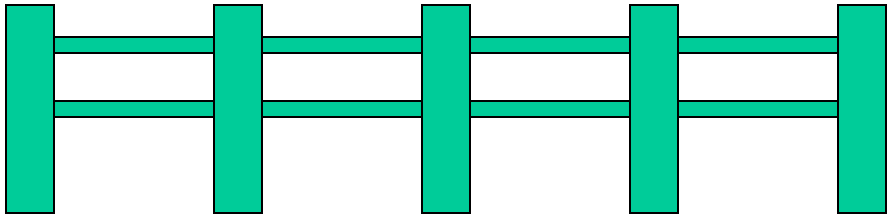
- The problem here is that if we are printing *n* numbers, we only need *n* - 1 commas.
- This problem is similar to the task of building a fence, where lengths of wire are separated by posts.
- If we repeatedly place a post and place a length, we will never have an end post.
  - A flawed algorithm:
    for (length of fence):
        place some post.
        place some wire.



Fencepost solution

- The key to solving the fencepost problem is to add an extra statement outside the loop that places the initial post.
  - This is sometimes also called the "loop-and-a-half" solution.
  - We will encounter this concept many times in this chapter when using indefinite while loops.
  - The revised algorithm:

    place a post.
    for (length of fence - 1):
    place some wire.
    place some post.



Fencepost printNumbers

- A version of printNumbers that works:

```
public static void printNumbers(int max) {
    System.out.print(1);
    for (int i = 2; i <= max; i++) {
        System.out.print(", " + i);
    }
    System.out.println();  // to end the line of output
}
```

OUTPUT from printNumbers(5):

```
1, 2, 3, 4, 5
```

- **fencepost loop**: A loop that correctly handles a "fence post" issue by issuing part of the loop body's commands outside the loop.

Fencepost practice problem

- Write a Java program that reads a base and a maximum power and prints all of the powers of the given base up to that max, separated by commas.

```
Base: 2
Max exponent: 9

The first 9 powers of 2 are:
2, 4, 8, 16, 32, 64, 128, 256, 512
```

Fencepost practice problem

- Write a method named printFactors that, when given a number, prints its factors in the following format (using an example of 24 for the parameter value):

```
[1, 2, 3, 4, 6, 8, 12, 24]
```