Java™ Basics and Object-based Programming

Reading assignment

- Chapter 1 in textbook, try some of the exercises and the tutorials online
- Study Java libraries extensively
  - http://java.sun.com/j2se/1.3/docs/api/overview-summary.html
  - `java.lang`
    - `Boolean, Integer`
    - `Math (PI, max, min, sin, cos, random(), round(), sqrt())`
    - `Object (clone(), equals())`
    - `String (CharAt(), CompareTo(), equals(), length())`
    - `System (println(), print(), flush(), Assignment 1)`
  - `java.io`
    - `BufferedReader (Section 1.6 in textbook)`
    - `Stdin, flush(), readLine()`
  - `java.util`
    - `List, LinkedList, Iterator`
    - `Observer`
    - `Calendar, set(), get() (Assignment 1)`
    - `Hashtable`
    - `Random (Assignment 1)`
    - `Stack`
  - The more you know what is in these libraries, the less code you have to write.

Topics to be covered today and next day….

- Done so far….
  - Classes and Objects
  - Methods
  - Primitive Types
  - Variables

- Today (if time)
  - Recap
  - References
  - Parameter passing
  - Program comments
  - Arrays
  - Strings
  - Control Flow
  - Expressions, operators
  - Input and Output
  - Strings

- Followed by…. (on Wed/Thurs)
  - Packages
  - Castings
  - Inheritance
  - Interfaces
  - Modifiers
  - Static members of a class
  - Exceptions

Reference types

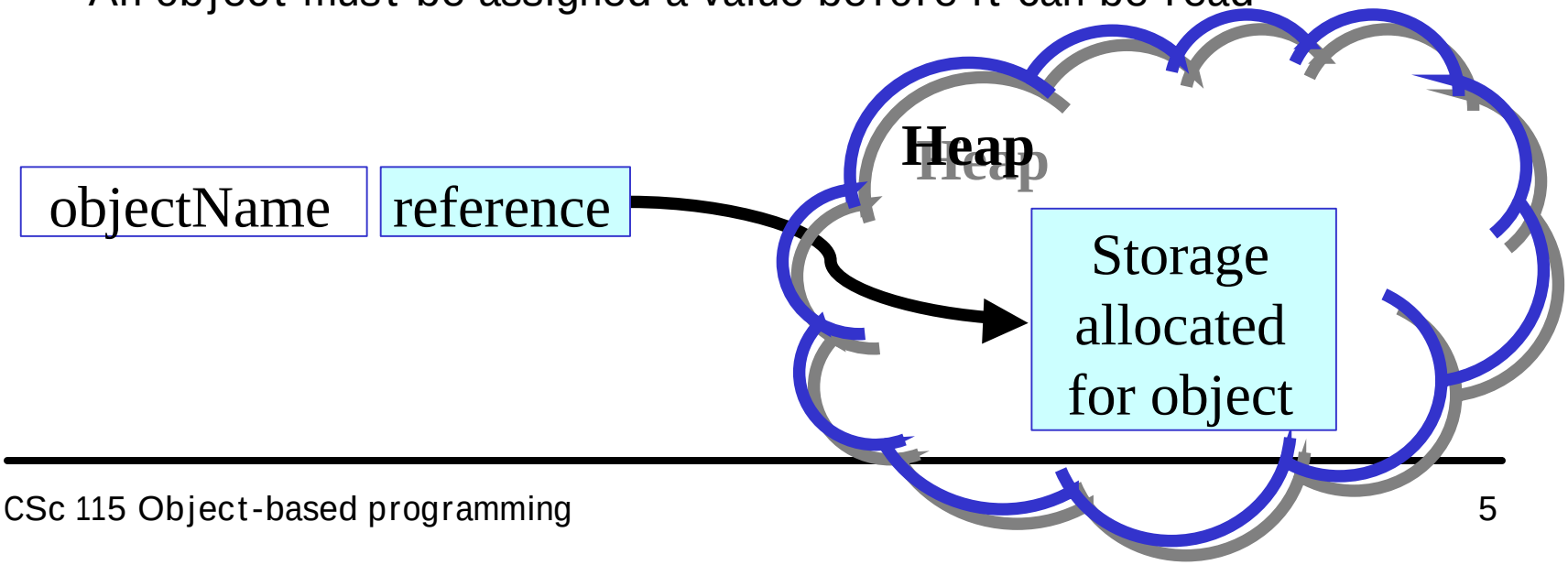
- Two kinds
  - ✍ Classes
  - ✍ Arrays (will look at later)

- String class (will also discuss later)

```
String hello = String("hello");
String hello = "hello"; // short form
hello.charAt(1);        // returns 'e'
```

Creating or instantiating objects

- An object is created from a defined class using the **new** operator
- **new** allocates storage for the object on the *heap* and returns a reference to the object
- An object can be declared anywhere (even within a for loop)
- An object can be accessed from its declaration to the end of the block
  - this is called its *scope*
  - a block ends at its closing curly brace "}"
- An object must be assigned a value before it can be read



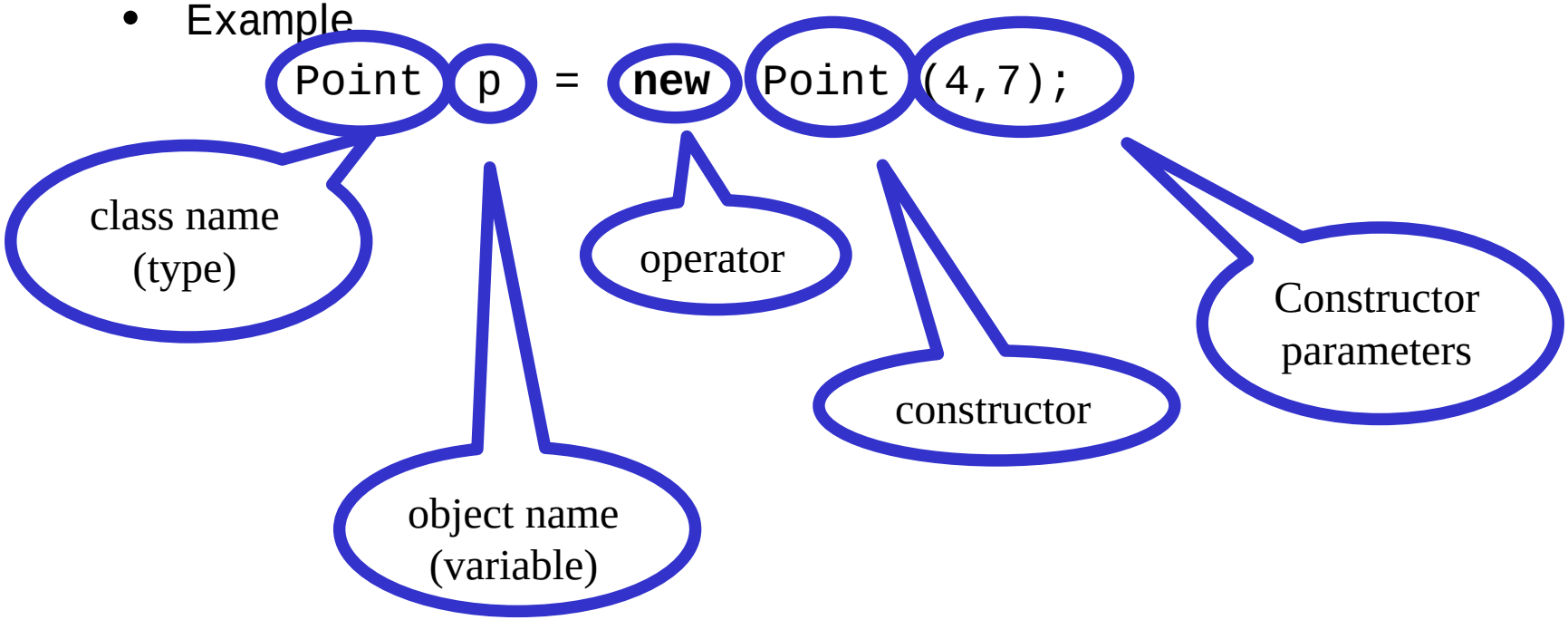
Creating or instantiating objects

- Syntax

```
objectName = new ConstructorClassName(paramters);
```

- Example

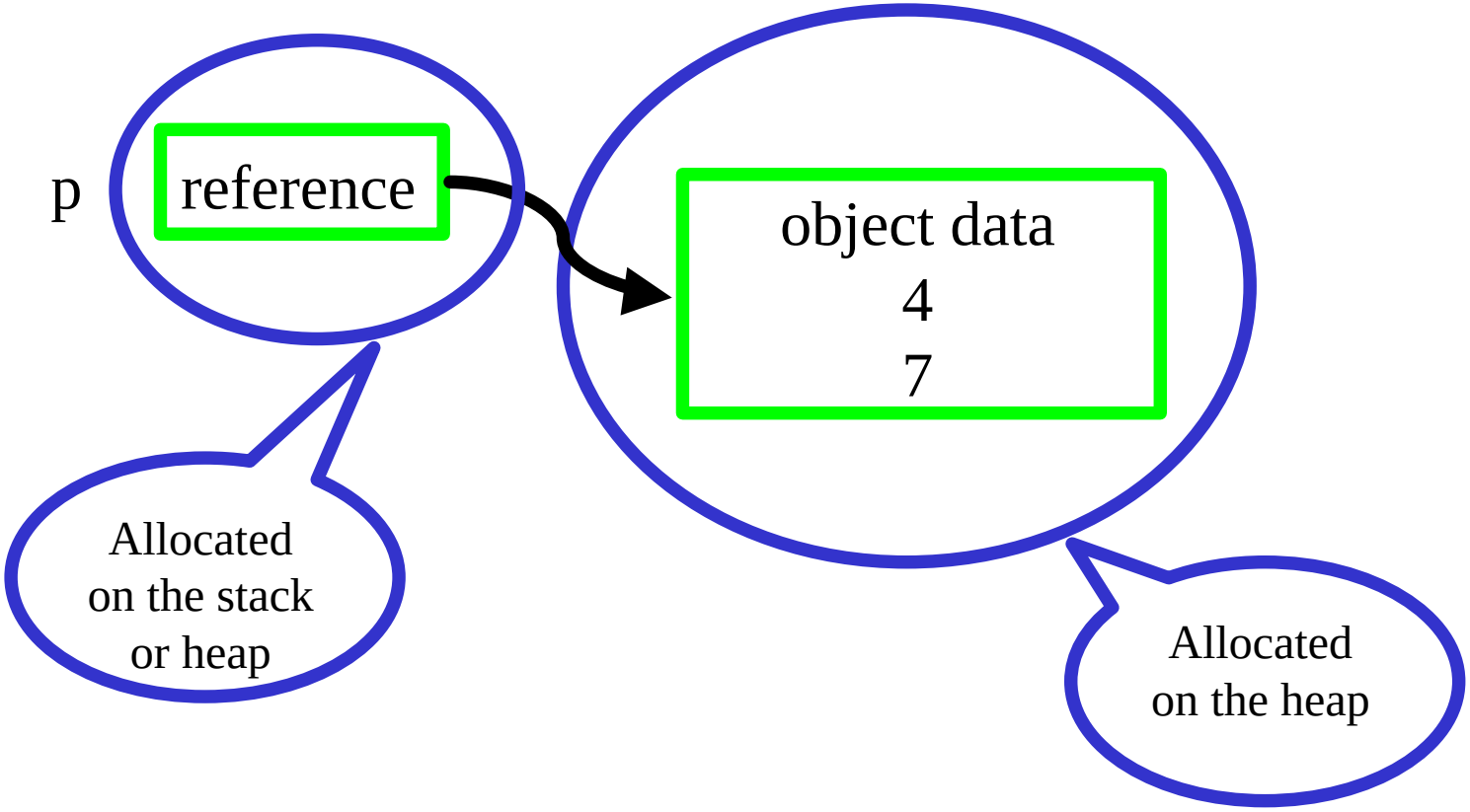
```
Point p = new Point(4,7);
```



Creating or instantiating objects

- Example

```
Point p = new Point (4, 7);
```



Dot Operator

<object_reference>.<method_name([param1, param2, …]);

Or

<object_reference>.<field_name> [= ….;]

An interactive programming example!

- Point.java

Summary of object allocation

- When new is called three things occur…
  1. A new object is dynamically allocated in memory and all instance variables are initialized to standard default values
  2. Constructor for the new object is called with the parameters specified
  3. When the constructor returns, the **new** operator returns a reference (memory address) to the newly created object

Parameters

- All method parameters are passed by value
- As a result
  - ✍ a parameter of a primitive type is *input-only* (i.e., its value is input into the method)
  - ✍ A parameter of reference type is *input-and-output* (i.e., the data of an object parameter can be changed by the method and the changes are visible to the caller of the method)

Parameters (2)

- Example – interactive programming exercise (Parameter.java)

```
void abc(int k; Point q) {
   k++; q.x++; q.y--;
   System.out.println(k, q.x, q.y);
}

int j = 17;  Point p = new Point(3,7);
System.out.println(j, p.x, p.y);
abc(j, p);
System.out.println(j, p.x, p.y);
```

Output:
17 3 7
18 4 6
17 4 6

A Trick Question or?

Look at this code, what do you think the output should be:

(See TrickyParameterPassing.java)

```
static void ChangePoint(Point q) {
    Point tempP = new Point(1,1);
    // q = tempP;
    q.x = 1; q.y = 1;
    q.PrintPoint();
}
// testing references, what happens?
Point testp = new Point(5,5);
testp.PrintPoint();
ChangePoint(testp);
testp.PrintPoint();
```

Summary of classes – another example

- Another Example if time (Course.java)

```
public class Course {
        // two fields and two methods
    private int noStudents;
    private String instructor;
    public Course(int k, String s) {
       noStudents = k; instructor = s; }
    public int getNoStudents() { return noStudents;
      }
    public String getInstructor() { return
      instructor; }
}
```

Program Comments

- Importance of….
- Inline
- Block
- Javadoc
- Assignments – your code must be clearly and verbosely commented! Use Javadoc!

Arrays

- An array is a numbered collection of components all of the same type
- Each component has an index
- The indices range from 0 to length-1
- Every array has a length field (e.g., a.length)
- An index outside this range is referred to as out of bounds and generates an IndexOutOfBounds exception
- Component types can either be primitive or reference (e.g., classes or arrays)
- See the book for a discussion of multi-dimensional arrays

Arrays (2)

- Examples

```
int [] a;
a = new int[5];
a[0] = 42;
a[1] = b[4];
String[] answers = {“yes”, “no”};
Color[] col = new Color[5];
col[0] = new Color( );
int [] b = {12, -15, 42, 12, 10};
b[5] = 11;      // error, throws IndexOutOfBoundsException
b.length == 5;  // b.length returns 5; expression is true
```

Strings

- A string is a set of characters that comes from some alphabet
- Each character c that makes up a string s can be referenced by its index in the string (= to the # of characters that comes *before* c in s, so 1st character is at index 0)
- In Java, our alphabet is the 16 bit unicode international character set (most other languages use a subset of this called ASCII)
- Java has a special built-in class of objects called String objects (so String is not an array)

Storage allocation for variables revisited

- Allocating an object or an array reference

```
String name;
Abc k;
int[] a;
Point[] p;
```

- Allocating a cell for a variable of a primitive type

```
int j;
double d;
```

- Instantiation and initialization

```
j = 3;
d = 3.14159;
name = “Bette”;
k = new Abc();
a = {1, 1, 3, 5, 9, 15, 25, 41, 67, 109};
p = new Point[10];
```

Control Flow — If

```
if (condition) {
   statements_1;
} else {
   statements_2;
}
```

- the *condition* must be a boolean expression
- if it is true, the first block is executed
- otherwise, the second block is executed if present
- execution then resumes after the end of the if statement

Control Flow — Switch

```
switch (expression) {
case constant_1:
   statements_1;
   break;
case constant_2:
   statements_2;
   break;
// …
default:
   statements_default;
}
```

- the *expression* must be of type **char**, **byte**, **short** or **int**
- each case label must be a unique constant
- code is executed starting at the case label whose constant matches the value of the expression
- if no constant matches, the default block is executed
- code is executed until a break statement (or the end of the switch) is reached

Switch Quiz

A **switch** that determines if a number between 2 and 8 is prime.

```
int n = (int) (Math.random()*7)+2;
boolean isPrime;
switch (n) {
   case 2:  isPrime = true; break;
   case 3:  isPrime = true; break;
   case 4:  isPrime = false;
   break;
   case 5:  isPrime = true; break;
   case 6:  isPrime = false;
   break;
   case 7:  isPrime = true; break;
   case 8:  isPrime = false;
   break;
}
System.out.println(n + ” is ” +
   (isPrime ? ”” : ”not ”) +
   ”prime”);
```

```
int n = (int) (Math.random()*7)+2;
boolean isPrime;
switch (n) {

}
System.out.println(n + ” is ” +
   (isPrime ? ”” : ”not ”) +
   ”prime”);
```

Control Flow — While

```
while (condition) {
    statements;
}
```

- the *condition* must be a boolean expression
- if it is true, the *statements* are executed, then the condition is evaluated again
- if it is false, execution resumes after the end of the while statement

```
do {
    statements;
} while (condition);
```

- as above, but the *statements* are executed at least once

Control Flow — For

```
for (initialization ; condition ; increment ) {
    statements;
}
```

- is equivalent to

```
initialization;
while (condition) {
    statements;
    increment;
}
```

- often used for iterating over the elements of an array
- the *initialization* statements can contain a variable declaration:

```
for (int i = 0; i < a.length; i++) {
    sum += a[i];
}
```

Control Flow — Break and Continue

- allows you to change the flow of a **for, while,** or a **do while** loop
  - ✍ **break** will immediately exit the loop
  - ✍ **continue** will skip ahead to evaluating the loop's condition
- Example
  - ✍ Given an array of "sock pair objects" (i.e., a pair can have 1, 2 socks of the same color)
  - ✍ return the *first* index of a *pair* of socks (i.e., two socks) that matches a given color

```
int findPairOfSocks(Color c, Socks[] a) {
   for (int i = 0; i < a.length; i++) {
      if (a[i].isOneSockLost()) continue;
      if (a[i].matchesColor(c)) break;
   }
   return i < a.length ? i : -1;
}
```

if not a pair, keep looking

if match, stop looking

if no match, return -1

Expressions

- Expressions are needed to define new values and to modify variables
- They involve the use of variables, literals and operators

Arithmetic Operators

- Assignment operator: i = j = 5;
- Arithmetic operators:
  - ✍ +, -, * , /
  - ✍ % modulo – n mod m = n – ?n/m?m
  - ✍ Unary minus (inverts the sign of an arithmetic expression
- See book (p. 22) for precedence chart, use ( )'s to change precedence

Arithmetic Operators (2)

- ++ increments
- -- decrements
- If prefix, then value is increased by one before the operation
- If postfix, then the value is increased by one after the operation

Operational Assignment Operators

- Instead of

  <variable> = <variable><op><expression> i = i + 5;

- We can do

  <variable> <op>= <expression> i += 5;

Logical Operators

- Comparison between numbers
- < == != <= >= >
- ++ and !+ can be used to compare object references (result is a boolean)
- Operators that operate on boolean values are:
  - ! Not (prefix)
  - && Conditional And
  - || Conditional Or
  - Note: these will not evaluate the second operand if it is not needed (useful if first test not true would generate an error condition with the second test)

String Operators

- String concatenation is done using '+' operator
  - String rug = "carpet"; String dirt = "spot";
  - String mess = rug + dirt;
  - String answer = mess + " yuck";
  - Answer is "carpetspot yuck"
- Note we can't do the following:
  - If (string1 = string2) { …}
  - Need to do: if (string1.equals(string2)) {…}

Input and output

- Java provides a rich set of classes for performing i/o
- Java provides classes for simple text i/o using a console window

```
import java.io.*;
```

- Java also provides i/o using a Graphical User Interface (GUI)

```
import java.awt.*; // for drawing
import javax.swing.*; // for widgets
```

Simple text I/O

- Output to the console:
  - ✍ Very useful for debugging logical errors in your program.
  - ✍ **`System.out`** is a static object of type **`PrintStream`**
    - **`print()`**, and **`println()`** methods take the following arguments:
      - Any object (provided it has a **`toString( )`**) method
      - Any string or concatenated strings
      - Any base type (automatically cast to String)
- Input from the console
  - ✍ must **`import java.io.*;`**
  - ✍ **`System.in`** is an object of type Input**`Stream`** (abstract class)
    - inputs bytes only (crude)
  - ✍ **`InputStreamReader`** translates bytes to characters.
    - API recommends wrapping an **`InputStreamReader`** within a **`BufferedReader`**
    - See page 35 of text

Simple text I/O

```
import java.io.*;
BufferedReader inp;
String line;
inp = new BufferedReader(new InputStreamReader(System.in));
System.out.print("Type a double: ");
System.out.flush();
if ((line = inp.readLine()) != null) {
   double d = Double.valueOf(line).doubleValue();
System.out.print("Type an int: ");
System.out.flush();
if ((line = inp.readLine()) != null) {
   int k = Integer.valueOf(line).intValue();
double sum = d + k;
System.out.println("Sum is " + sum);
```