Java™ Basics
Object-based Programming

Reading Assignment Chapters 1-2

Reading assignment

- Chapters 1-2 in textbook
- 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.

Classes and objects

- The main "actors" in an OO programming language are *objects*
  - Objects are alive ☺
  - An object stores the *state* (i.e., data) of its actor in *fields*
  - An object provides capabilities to its actor with *methods*
  - Methods of an object operate (i.e., access, modify) on its fields
  - Every object is an instance of a *class*
- A class consists of members
  - There are two categories of class members
    - Fields or variables
    - Methods
  - A class defines *types* for all of its members
  - The type of a field can be *primitive* or *reference*

The surroundings of a class

- Package
  - A class belongs to a named package or the default package
    ```
    package cscc115assignment1;
    ```
  - A class can import packages
    ```
    import javax.swing.*;
    Import java.io.*;
    ```
- Inheritance
  - A class can extend another class (i.e., be a *subclass*)
    ```
    public class Manager extends Employee { … }
    public class Model extends Observable { … }
    ```
  - A class can be a *superclass* for another class
- Interfaces
  - A class can implement an interface
    ```
    public class TextView implements Observer { … }
    ```

Class declaration

- Syntax

```
[modifiers] class ClassName [extends SuperClassName]
   [implements Interface1, Interface2, …] {
   class member declarations;
}
```

- Example

```
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; }
}
```


Class modifiers

- Class modifiers are optional keywords preceding the **class** keyword.
- **abstract**
  - The class has **abstract** methods (i.e., no method body and preceded by abstract modifier).
  - A class with only **final** instance variables and only **abstract** methods is called an **interface**
- **final**
  - A final class has no subclasses.
- **public**
  - A **public** class can be instantiated or **extended** by anything in the same package or anything that **imports** this class.
  - Each public class is declared in a separate file; downloadable component.

Creating fields or variables

- Creating fields of primitive types
- Examples

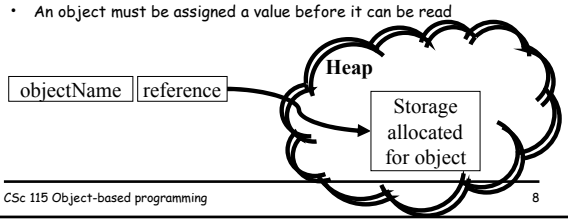
```
int k = 3;
double d = 3.14159;
boolean b = true;
```

- Creating fields of reference types (i.e., objects)
- Examples

```
String s = new String("hi");
Point p = new Point(3, 7);
Course c = new Course(256, "Muller");
```


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

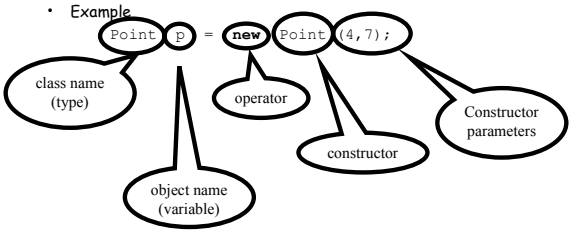


Creating or instantiating objects

- Syntax

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

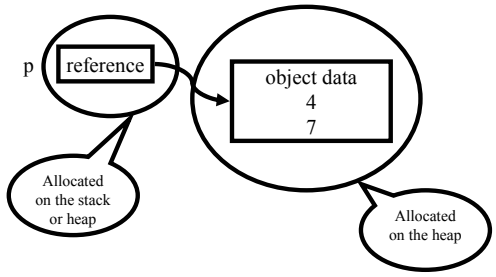
- Example



Creating or instantiating objects

- Example

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



Class members

- Fields
  - Data associated with an object
  - Represent and store the *state* of an object
  - The type of an field or a parameter can be primitive or reference
  - All fields are initialized to default values automatically
  - Examples

```
private int k = 17;
protected Point p = new Point(17,12);
```

- Methods
  - Define the behaviour of the objects instantiated from that class
  - A method definition has two parts: *signature* and *body*
  - Methods have a return type
    - The return type is **void** if the method does not return anyting
  - Methods are also called functions or procedures

```
void doNothing() { }
ComplexNumber makeComplex(double r, double i){ /* … */
int findSock(Color c, Socks[] a) { /* … */ }
double[] getGrades() { /* … */ }
```


public, **protected**, **private**, and package modifiers

- These modifiers apply to both fields and methods
- **public**
  - Any method can access **public** members
- **protected**
  - Only methods of the same package or subclasses can access protected members
- **private**
  - Only methods of the same class can access **private** members
- Package (no modifier)
  - Members, which are not **public**, **protected**, or **private**, are called package members
  - Only methods in the same package can access package members

Field modifiers

- **public, protected, private, package modifiers**
  - As discussed on previous slide
- **static**
  - A **static** variable is associated with its class, is shared by all objects of its class, and its storage exists once (i.e., with the class rather than all the objects)
- **final**
  - A **final** variable must be initialized and is readonly after initialization (i.e., it is constant)
  - **final** variables are usually also declared **static** so that storage is allocated only once for an entire class
  - The naming convention for **final** variables is all upper case
  - **final** variables are often declared in interfaces

Method modifiers

- **public, protected, private, package modifiers**
  - As discussed on a previous slide
- **static**
  - A **static** method is associated with its class and is shared by all objects of its class (i.e., with the class rather than all the objects)
  - The **static** fields can only be changed by **static** methods (as long as they are not declared **final**)
- **final**
  - A **final** methods cannot be overridden by a subclass.
- **abstract**
  - An **abstract** method has no body.
  - The parameter list is followed by a semicolon to terminate the **abstract** method declaration.
  - **abstract** methods may only appear within an **abstract** class.
  - **abstract** methods are typically overridden by subclasses.

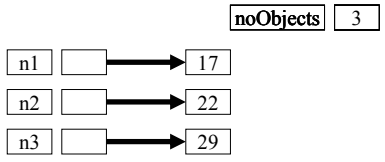
static members

- Example

```
public class Node {
   private static int noObjects = 0;
   private int id;
   public Node(int k) { id=k; noObjects++ }
   public static getNoObjects() {
      return noObjects;
   }
}
System.out.println(Node.noObjects()); // 0
Node n1 = new Node(17);
System.out.println(Node.noObjects()); // 1
Node n2 = new Node(22);
System.out.println(Node.noObjects()); // 2
```


static field

- Execute program



Accessing members

- Dot notation
- Accessing instance members

```
objectName.classMember
objectName.field
objectName.method()
```

- Accessing static members

```
className.classMember
className.field
className.method()
```


Constructors

- A special type of method
- Instantiates and initializes objects
- Has same name as the class and no return type
- The **abstract**, **static**, and **final** modifiers are not allowed for constructors
- A **public**, no-argument constructor is provided by the Java run-time environment if the class does not define one
- A class can have many constructors; all have the same name, but all signatures must be different

main() method

- The main entry point of a Java program
- This is the first routine called by the operating system
- Specific signature:

```
public static void main(String[] args) { … }
```

- Each class can have a `main()` routine for testing purposes

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)
- Example

```
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

Primitive types

- Primitive types are defined by the language:
  - `byte, short, int, long, float, double, boolean, char`
- All primitive types have literals
  - A literal is an unnamed constant value
  - Examples

int	42	052	0x2a	
double	42.0	42.	4.2e1	42d
float	42.0f	.42e2f		
boolean	true	false		
char	'c'	'\n'	'\\'	'\"

- You can *wrap* primitive data inside objects, if necessary
- Sometimes useful to treat all variables uniformly

```
Integer intWrapper = new Integer(3);
int i = intWrapper.intValue();
```


Reference types

- Two kinds
  - Classes
  - Arrays
- String class

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

  - Strings are *not* arrays of characters

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)

```
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
```


Storage allocation for variables

- 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];
```

Variables Quiz

```
int sumSquares(int n) {

   partialSum = 0;

   int i;

   while (i <= n) {
      int square = i*i;
      partialSum += square;
      i++;
   }

   System.out.println("last square = " + square);

   return partialSum;
}
```


Variables Quiz (Solutions)

```
int sumSquares(int n) {

   partialSum = 0;
   // The variable named partialSum has no type

   int i;

   while (i <= n) {
   // The variable named i is not initialized
      int square = i*i;
      partialSum += square;
      i++;
   }

   System.out.println("last square = " + square);
  // The variable named square is not defined
   return partialSum;
}
```


Modifiers (Quiz)

	Modifier	Can be applied to		
		Classes	Fields	Methods
access modifiers	public			
	protected			
	(default)			
	private			
	static			
	final			
	abstract			

- Other modifiers:
  synchronized, native, transient, volatile, strictfp

Modifiers (Quiz)

	Modifier	Can be applied to		
		Classes	Fields	Methods
access modifiers	public	x	x	x
	protected		x	x
	(default)	x	x	x
	private		x	x
	static		x	x
	final	x	x	x
	abstract	x		x

- Other modifiers:
  synchronized, native, transient, volatile, strictfp

Identifiers and Reserved Words

- Identifiers are used as names for variables, constants, classes, methods, etc.
- Must follow certain rules:
  - must begin with a letter
  - may contain additional letters and digits
  - all characters are significant
  - case sensitive
  - must not conflict with a reserved word

Identifier Quiz

Identifier	Valid?
sum	
4you	
salary%	
MEDIUM	
long	
longint	
Double	
NO_VALUE	
_12	
goto	
Rect$1	
始まった	

Identifier Quiz (Solutions)

Identifier	Valid?
sum	√
4you	No - Must begin with letter
salary%	No - No special chars allowed
MEDIUM	√
long	reserved
longint	√
Double	√
NO_VALUE	√
_12	√ - _ is considered a char
goto	Reserved word
Rect$1	√ $ is considered
始まった	√

Naming conventions

- Variables, fields, parameters
  - Mixed case, start with lower case
    - k
    - inputMode
- Classes, constructors
  - Mixed case, start with upper case
    - Person
    - Clock
- Constants
  - All upper case
  - PI, MAXNUMBER, LASTINDEX
- Methods
  - Mixed case, start with lower case, parenthesis
    - getAge()
    - setUserID()
- Packages
  - All lower case
    - awt
    - swingx
    - project

Operators and Expressions

- Operators can be unary, binary or ternary
- Operators are left-associative, except for assignment operators

```
42 - 17 - 5   ➔   (42 - 17) - 5
a = b = 42    ➔   a = (b = 42)
```

  - assignment operators return the value assigned

- Operators have precedence

```
a.length > 5 + 42 / 7
        ➔ (a.length) > (5 + (42 / 7))
x < 5 && y != 6 ? 42 : 0
        ➔ ((x < 5) && (y != 6)) ? 42 : 0
```

  - See table on page 22 in Goodrich and Tamassia.
  - when in doubt, don't skimp on parentheses!

Operator Quiz

```
int x = 42, y = 20, z = 1;  int[] a = {19, 4, 7};
```

Expression	Which Result is correct?		
y * 2 – x / 7 + 1	1	14	35
3 / 2	1.5	1	2
3 / 2d	1.5	1	2
y % 6	3	2	-2
y++	20	21	22
++y	20	21	22
a[z++]	2	4	7
a[0] – a[2]	12	15	19
a.length > 3 && a[3] == 7	true	7	false
x = y /= 5	4	8	5
(y %= 7) == 0 ? -1 : x / ((double) y)	-1	7	7.0

Operator Quiz

```
int x = 42, y = 20, z = 1;  int[] a = {19, 4, 7};
```

Expression	Which Result s correct?		
y * 2 – x / 7 + 1	1	14	35
3 / 2	1.5	1	2
3 / 2d	1.5	1	2
y % 6	3	2	-2
y++	20	21	22
++y	20	21	22
a[z++]	2	4	7
a[0] – a[2]	12	15	19
a.length > 3 && a[3] == 7	true	7	false
x = y /= 5	4	8	5
(y %= 7) == 0 ? -1 : x / ((double) y)	-1	7	7.0

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");
```


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) {
  case 2:
   case 3:
   case 5:
   case 7:  isPrime = true; break;
   case 4:
   case 6:
   case 8:  isPrime = false;
   break;
}
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

- allow 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

Control Flow Quiz

- What is the difference between a `while` statement and a `do while` statement?
- How can you translate a `for` loop into a `while` loop?

Control Flow Quiz

- What is the difference between a `while` statement and a `do while` statement?
  - The block in the `do while` statement is executed at least once.
  - The block in the `while` statement is potentially never executed.
- How can you translate a `for` loop into a `while` loop?

```
for (init; terminationCondition;
  incrementalStep) {
     statements;
  }
init;
  while (terminationCondition) {
     statements;
     incrementalStep;
  }
```

Packages

- Large software systems have many more classes than lines of code per class. Thus, organizing classes is as important as programming individual classes.
- Java offers the notion of a package to aggregate related classes.
- Classes are assigned to a package using a **`package`** directive before the class declaration:

```
package packagename;
package assignment3;
```

- Package names are usually in all lower case.
- Using the **`import`** directive, packages can be imported (i.e., made available) to classes.

```
import packagename.*;
import assignment3.*;
```


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);
```