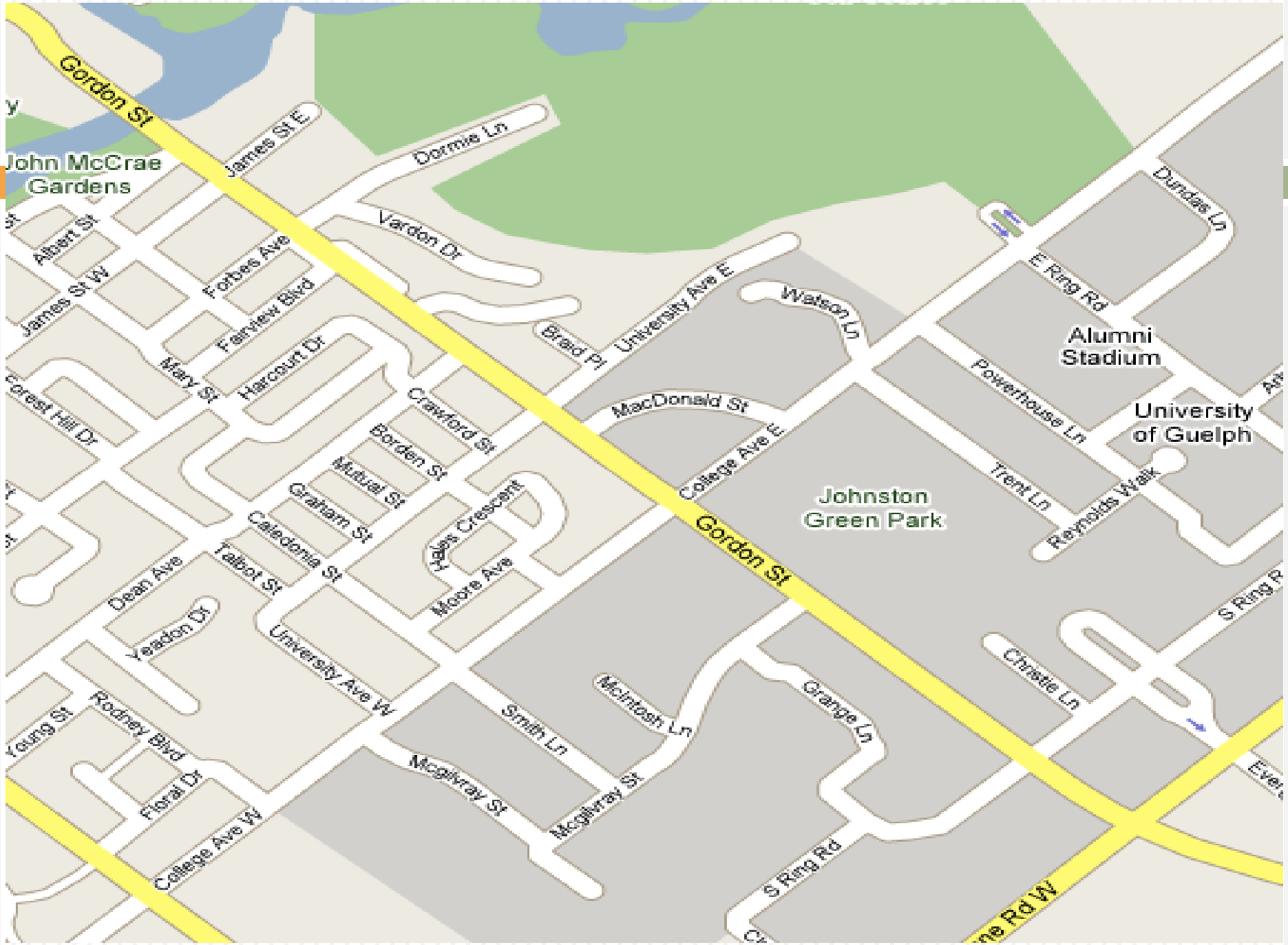
ABSTRACTIONS FOR OOP

CIS*2430 (Fall 2010)

OOP vs. Java

- This course is about OO concepts and techniques
- Java will be used for illustration and practice
  - Will not be taught in the same way as C was taught in CIS*1500/2500
- We will frequently flip between OO concepts and Java examples


Gordon St
John McCrae Gardens
James St E
Dormie Ln
Albert St
Vardon Dr
James St W
Forbes Ave
Fairview Blvd
Harcourt Dr
Mary St
Braid Pl
University Ave E
Watson Ln
E Ring Rd
Dundas Ln
Alumni Stadium
Powerhouse Ln
University of Guelph
MacDonald St
Crawford St
Borden St
Mutual St
Graham St
Caledonia St
Talbot St
Dean Ave
Yeadon Dr
University Ave W
Hales Crescent
Moore Ave
College Ave E
Johnston Green Park
Trent Ln
Reynolds Walk
Gordon St
S Ring Rd
Christie Ln
McIntosh Ln
Grange Ln
Smith Ln
Young St
Rodney Blvd
Floral Dr
College Ave W
Mcgillvray St
Mcgillvray St
S Ring Rd

Complex Systems

- Can't deal with all details at a time
- Need mechanisms for selecting just the appropriate information
- Think about the modern automobile
  - what level of detail do you need to be a driver?
  - How about a mechanic?

Abstraction

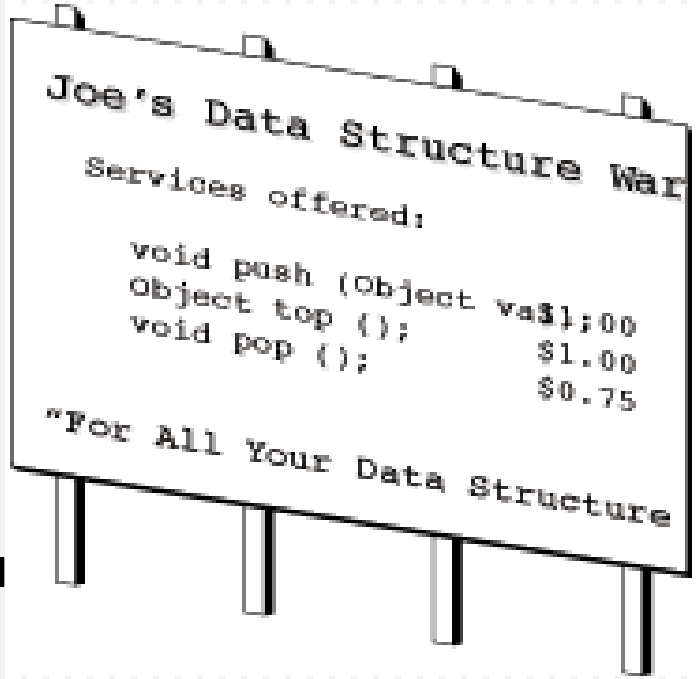
- Abstraction or Information Hiding:
  - the purposeful suppression or hiding of some details of a process or artifact in order to bring out other aspects more clearly
- Layers of abstraction
  - Programs: entire applications
  - Units or groups: packages in Java or namespaces in C++
  - Interacting objects: clients and service providers

Service View of Programming

- Think of your classes as providers and consumers of services
- As a client, your classes needs to know how to make a request, but not what is being done
- As a server, your class must respond to requests
- The API is the contract between the server and the client
- interface and implementation are kept separate

Client Side Abstraction

- Describes what services are provided
- Does not describe how it is done
- Often called an API or an interfa



Server Side Abstraction

- Deals with the way the service is implemented
  - Stack as a linked list stack or an array?
  - Immediate delete or lazy delete?
  - Can go to the level of methods and the individual commands to create the methods

Programming with Abstraction

- All levels of abstraction are important
- Programmers are often required to move between levels of abstraction to complete a single task
- Need to decide which level is most appropriate for the task at hand:
  - What details are important without throwing away critical information
  - Impossible to complete the task with such information

Divide and Conquer

- When programs are thought of as collections of services, individuals can work on a single service
  - The 'transmission' team only needs the API from the 'engine' team, but the transmission team needs full details about the transmission implementation
- The strict division of information between inner and API is called encapsulation
- Components that are well encapsulated can often be substituted for one another
  - putting a different engine into a car

Abstractions in Java

- Method level: flow of control
  - Branches: if-then, if-then-else, switch
  - Loops: for, while, do-while
  - Special controls: break, continue, exit, and assert
- Implementation level: a bit of how but not actual code (e.g., String and Scanner classes)
- API level (e.g., Array and ArrayList)

Branching Example

```
if (myScore > yourScore)
{
   System.out.println("I win!");
   wager = wager + 100;
}
else
{
   System.out.println("I wish these were golf scores.");
   wager = 0;
}
```

Switch Example

```
switch (numberOfFlavors)
{
   case 1:
      System.out.println(“I bet it’s vanila.”);
      break;
   case 2:
   case 3:
   case 4:
      System.out.print(numberOfFlavors + “ flavors”);
      System.out.println(“ is acceptable”);
      break;
   default:
      System.out.print(numberOfFlavors + “ flavors”);
      System.out.println(“ is not acceptable”);
      break;
}
```

Loop Example

```
outerloop:
do
{
   . . .
   while (next >= 0 )
   {
      next = keyboard.nextInt();
      if (next < -100)
         break outerloop;
      . . .
    }
    . . .default:
    answer = . . .
} while (answer.equalsIgnoreCase("yes"));
```

Assertion Check

```
int n = 0;
int sum = 0;
assert (n == 0) && (sum == 0);
while (n < 100)
{
   n++;
   sum = sum + n;
}
```

Javac YourProgram.java
Java –enableassertions YourProgram

Comparing Two Strings

- The equality operator (==) correctly tests two values of a *primitive* type
- For objects (e.g., strings), == tests if they are stored in the same location, not whether they have the same value
- To test if two strings have equal values, use the method **`equals`**, or **`equalsIgnoreCase`**:

```
string1.equals(string2)
string1.equalsIgnoreCase(string2)
```

Lexicographic vs. Alphabetic

- Lexicographic order: the same as *ASCII* order, and includes letters, numbers, and other characters
  - All uppercase letters are in alphabetic order, and all lowercase letters are in alphabetic order, but all uppercase letters come before lowercase letters
- Alphabetic order: use `compareToIgnoreCase` method for a mixture of lower and upper cases.

StringTokenizer Class

- The **`StringTokenizer`** class is used to recover the words or *tokens* in a multi-word **`String`**
  - Most text files are organized by lines, which can be read in with "nextLine" of the Scanner class
  - We can use whitespace characters to separate each token, or specify different delimiters
  - **`StringTokenizer`** needs to be imported:

```
import java.util.StringTokenizer;
```

Methods in StringTokenizer

Display 4.17 Some Methods in the Class `StringTokenizer`

The class `StringTokenizer` is in the `java.util` package.

```
public StringTokenizer(String theString)
```

Constructor for a tokenizer that will use whitespace characters as separators when finding tokens in `theString`.

```
public StringTokenizer(String theString, String delimiters)
```

Constructor for a tokenizer that will use the characters in the string `delimiters` as separators when finding tokens in `theString`.

```
public boolean hasMoreTokens()
```

Tests whether there are more tokens available from this tokenizer's string. When used in conjunction with `nextToken`, it returns `true` as long as `nextToken` has not yet returned all the tokens in the string; returns `false` otherwise.

(continued)

Methods in StringTokenizer

Display 4.17 Some Methods in the Class `StringTokenizer`

`public String nextToken()`

Returns the next token from this tokenizer's string. (Throws `NoSuchElementException` if there are no more tokens to return.)[5]

`public String nextToken(String delimiters)`

First changes the delimiter characters to those in the string `delimiters`. Then returns the next token from this tokenizer's string. After the invocation is completed, the delimiter characters are those in the string `delimiters`.
(Throws `NoSuchElementException` if there are no more tokens to return. Throws `NullPointerException` if `delimiters` is `null`.)[5]

`public int countTokens()`

Returns the number of tokens remaining to be returned by `nextToken`.

StringTokenizer Example

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

public class StringTokenizerDemo
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter last, first, and middle names.");
        System.out.println("Enter \"None\" if no middle name.");
        String line = keyboard.nextLine();
        String delimiters = ", ";    // comma and blank space
        StringTokenizer nameFactory = new StringTokenizer(line, delimiters);

        String lastName = nameFactory.nextToken();
        String firstName = nameFactory.nextToken();
        String middleName = nameFactory.nextToken();
        if( middleName.equalsIgnoreCase("None"))
           middleName = "";   // Empty string
        System.out.println("Hello " + firstName +
                           " " + middleName + " " + lastName);
    }
}
```

StringTokenizer Example

```
Sample Dialogue:
```

Enter last, first, and middle names.

Enter “None” if no middle name.

Savitch, Walter None

Hello Walter Savitch

Creating & Accessing Arrays

- Creating an array with a specific length:

```
double[] score = new double[5];
Person[] specimen = new Person[count];
```

- An array can have indexed variables of any type, including any class type, and all the indexed variables must be of the same type, called the *base type* of the array

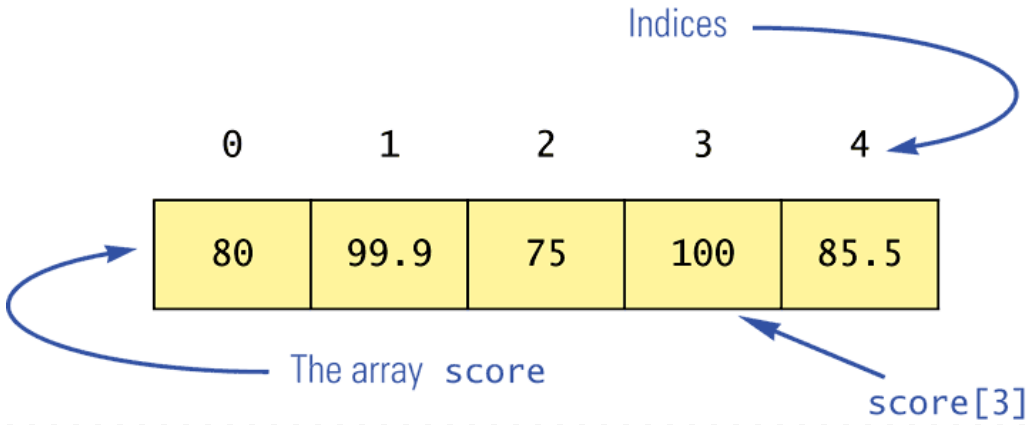
- Java arrays are indexed from zero:

```
score[0], score[1], score[2],
score[3],
score[4]
```

Using Arrays

- The **`for`** loop is ideally suited for performing array manipulations:

```
for (index = 0; index < 5; index++)
   System.out.println(score[index] +
        " differs from max by " +
        (max-score[index]) );
```



Length Instance Variable

- An array is considered to be an object

- Every array has one instance variable named ***length***
  - When an array is created, the instance variable **length** is automatically set to its size
  - The value of **length** cannot be changed (other than by creating an entirely new array with **new**)

```
double[] score = new double[5];
```

  - Given **score** above, **score.length** has a value of 5

Pitfall for Arrays

- The base type of an array can be a class type

```
Date[] holidayList = new Date[20];
```

- The above example creates 20 indexed variables of type `Date`
  - It does not create 20 objects of the class `Date`
  - Each of these indexed variables are automatically initialized to `null`
  - Any attempt to reference any them at this point would result in a "null pointer exception" error message

Pitfall for Arrays

- Each indexed variable requires a separate invocation of the **new** operator to create an object to reference

```
holidayList[0] = new Date();
                 . . .
holidayList[19] = new Date();

for (int i = 0; i < holidayList.length; i++)
  holidayList[i] = new Date();
```

- Each indexed variable can now be referenced since each holds the memory address of a **Date** object