Intro to Object-Oriented Programming

CIS*2430 (Fall 2021)

Contact Information

- Instructor: Fei Song
  - Email: fsong@uoguelph.ca
  - Office hours: Mon. 2:30 – 3:30 pm, Wed. 3:30 – 4:30 pm, and Fri.: 2:30 – 3:30 pm (Virtual)
- TA's: Benjamin Carlson, Jordan Cartlidge, Liam Fayle, Nadeem Howlader, Parth Miglani and Linda Ngo
  - Duties: teach labs, mark assignments and exams, hold office hours, and monitor discussion forums
  - Email: 2430fs@socs.uoguelph.ca
  - Office hours: Mon. 5:30 – 6:30 pm; Tue. 2:30 – 3:30 pm; Wed. 4:30 – 5:30 pm; Thu. 3:30 – 4:30 pm, 5:30– 6:30 pm; and Fri. 4:30 – 5:30 pm (All Virtual)

Contact Information

- All labs are held virtual between Sept. 13 and Nov. 30 for a total of 11 weeks.
- You can use your own machines or the Linux server at linux.socs.uoguelph.ca for all programming tasks.
- Lab Sections: You are required to ONLY go to the sections assigned to you on WedAdvisor.

Section 0101	Mon, 11:30 am – 1:20 pm	Liam
Section 0102	Tue, 11:30 am – 1:20 pm	Jordan
Section 0103	Wed, 11:30 am – 1:20 pm	Linda
Section 0104	Wed, 1:30 pm – 3:20 pm	Parth
Section 0105	Thu, 11:30 am – 1:20 pm	Benjamin
Section 0106	Fri, 12:30 pm – 2:20 pm	Nadeem
Section 0107	Tue, 3:30 pm – 5:20 pm	Nadeem
Section 0108	Mon, 3:30 pm – 5:20 pm	Jordan

Evaluation Scheme

- Programming (52%):
  - Three assignments @ 14% each (Oct. 18, Nov. 8, Nov. 29)
  - Five lab exercises @ 2% each (due every two weeks)
  - One competency test (to be scheduled individually based on one of the five lab exercises)

- Exams (48%)
  - Two quizzes @ 10% each (Oct. 6 and Nov. 12, 9:30 – 10:20 am)
  - One final exam @ 28% (Dec. 16, 8:30 – 10:30 am)

Competency Tests

- To avoid potential academic misconducts, one of the five lab exercises will also be evaluated through virtual meetings so that we can verify the understanding levels and development skills of all the students.
- If there are noticeable discrepancies between these evaluations and the performance on the programming assignments, further investigation will be triggered to identify possible cases for academic misconducts.
- In addition, all three programming assignments will be checked automatically by Sneakoscope for strong overlaps of code copying or sharing.

Policies

- Fast and reliable internet connections are required for virtual teaching since you will need to view or download course materials, attend online advising, and take online exams.
- Come to the class regularly: the textbook and lecture notes can't replace question-answering and in-class discussions
- Lab attendance is highly beneficial since you can ask questions and get one-on-one help for the lab exercises
- Late assignments are not accepted due to the limited teaching supports and any requests for re-marking will need to be submitted within 5 business days

Policies (continued)

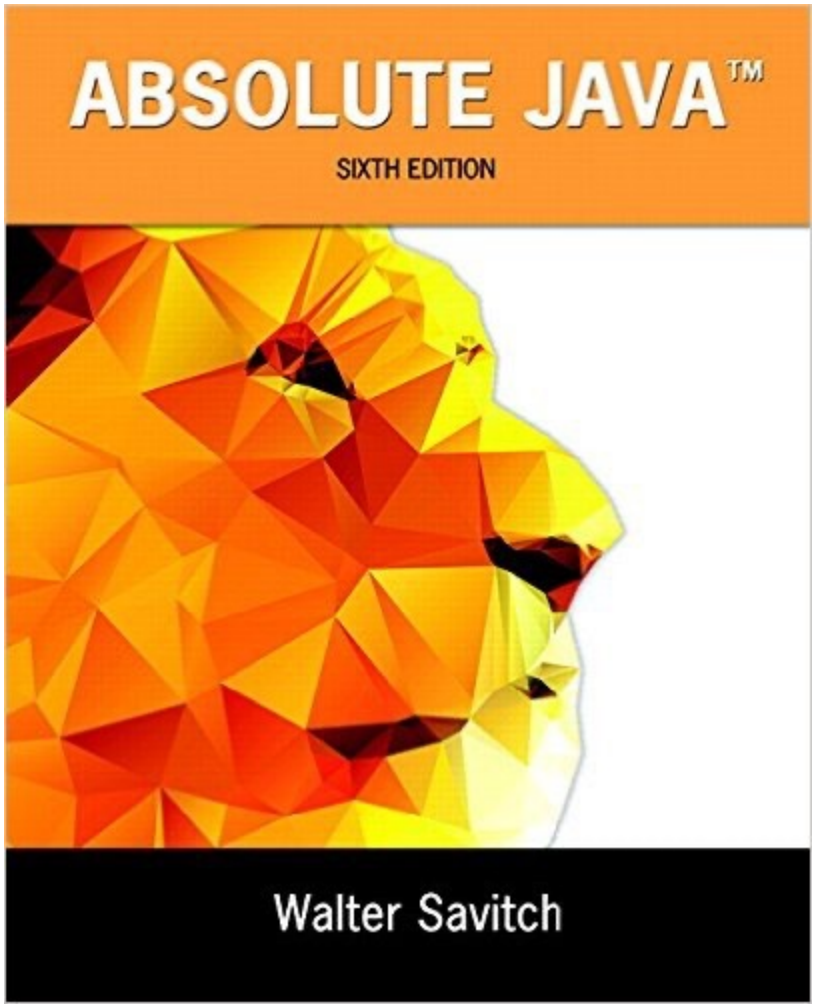
- Do your own work individually. Valid collaboration and code reuse is encouraged but must be documented clearly and appropriately.
- Undocumented use of other people's code is plagiarism, which will incur severe penalty as per the rules on Academic Misconduct: https://www.uoguelph.ca/registrar/calendars/undergraduate/current/.
- Emergencies do happen and you need to contact me as soon as you have a problem - don't wait until it evolves into a disaster.

Textbook

- Walter Savitch and Kenrick Mock. *Absolute Java*. Sixth Edition. Pearson, 2016 (ISBN: 0-13-4041674)

- Other course materials are available in our CourseLink account:

  https://courselink.uoguelph.ca/



Learning Objectives

- Identify major characteristics of different programming paradigms (procedural, functional, logical, and object-oriented)
- Differentiate between procedural and object-oriented paradigms
- Design and implement classes demonstrating correct use of encapsulation, constructors, method overloading, class invariants, accessors, mutators, instance variables and class variables
- Construct class hierarchies that maximize code reuse through inheritance while accommodating differences through method overriding

Learning Objectives

- Describe polymorphism and identify situations in which it is used in an OO program
- Use polymorphism, abstract methods/classes, and interfaces effectively to produce generic code
- Read and understand class diagrams written in UML (Unified Modeling Language)
- Compare event-driven programming with control-driven programming

List of Topics

- Introduction to different programming paradigms
- User-defined classes
- Key OOP mechanisms (encapsulation, inheritance, and polymorphism)
- Overloading vs. overriding methods
- Common data structures (e.g., Collections, Maps, and Iterators)
- Exception handling
- Event-driven programming for graphical user interfaces
- Advanced topics such as UML modeling and design patterns, generics, network programming, database connections, web programming, functional programming

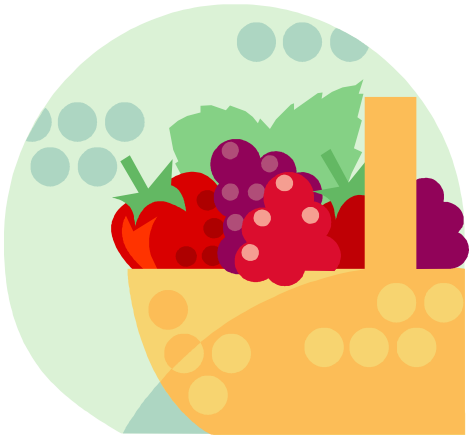
Advice from Previous Classes

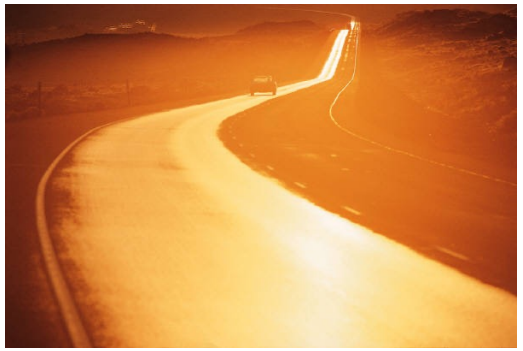
- Start assignments early: big tasks become small tasks and you will learn more and enjoy yourself more.
- Don’t leave assignments to the last minute: small tasks become big tasks and you will become less productive under pressure.
- Buy the textbook: extensive coverage of additional examples and exercises.
- Don't write a line of code until you spend some quality time with pen and paper in hand, thinking about how best to solve the problem.
- Chances are you don't know how to test and debug well. Read ahead and learn it because you can bring programming time down.

Introduction to OOP

- Objects
- Programming Paradigms
- Java
  - History
  - Basics
  - Console programming

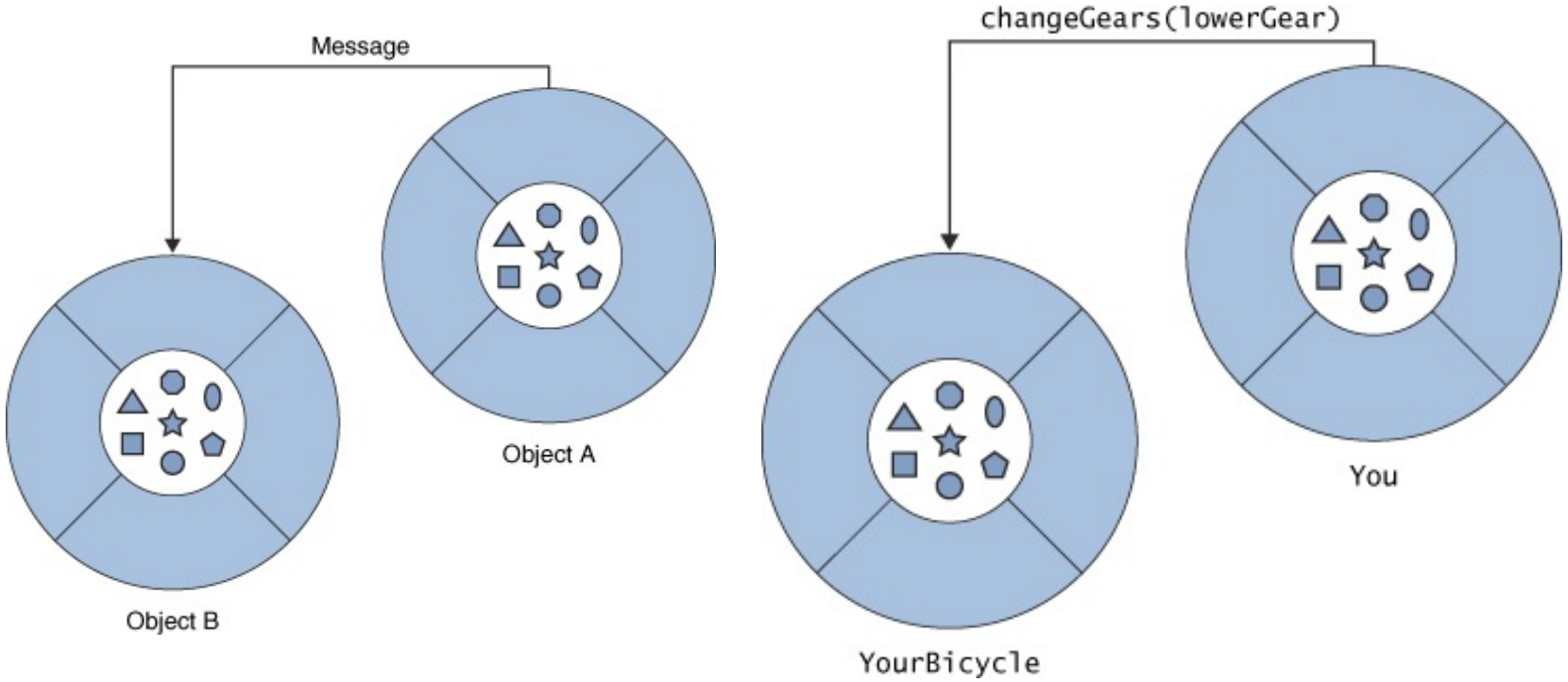
What are Objects?





Message Passing

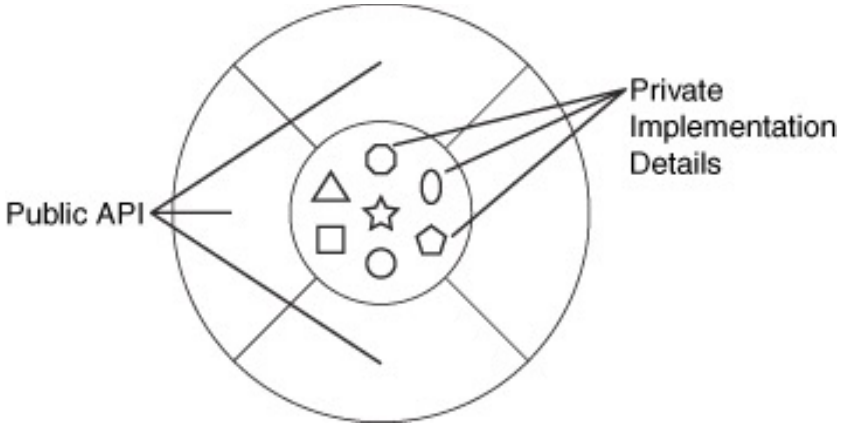
- Programming as a simulation of object interactions.

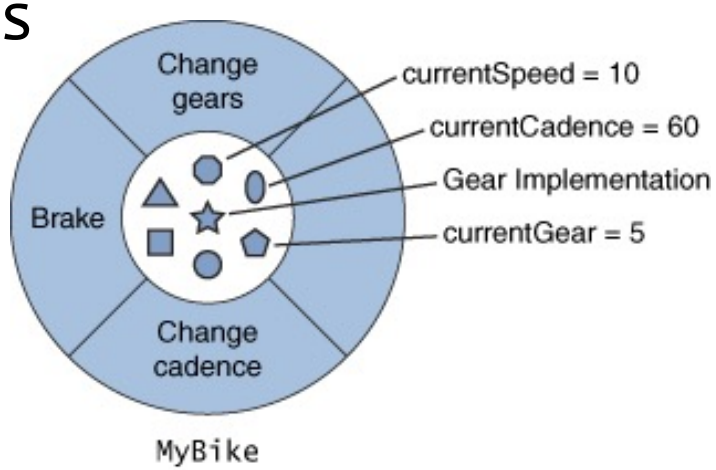


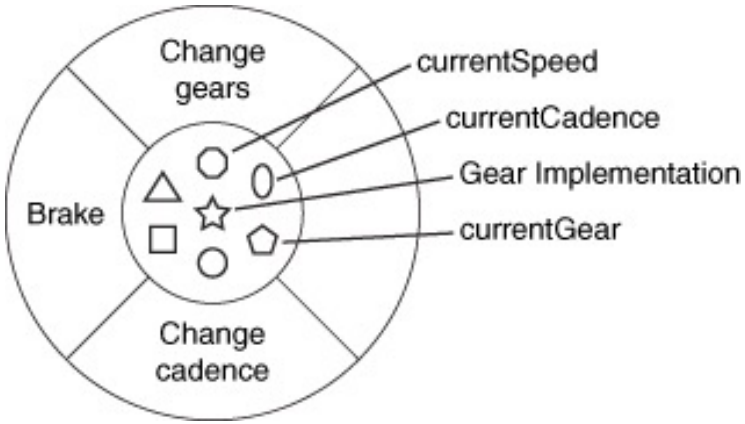
http://java.sun.com/docs/books/tutorial/java/concepts/message.html

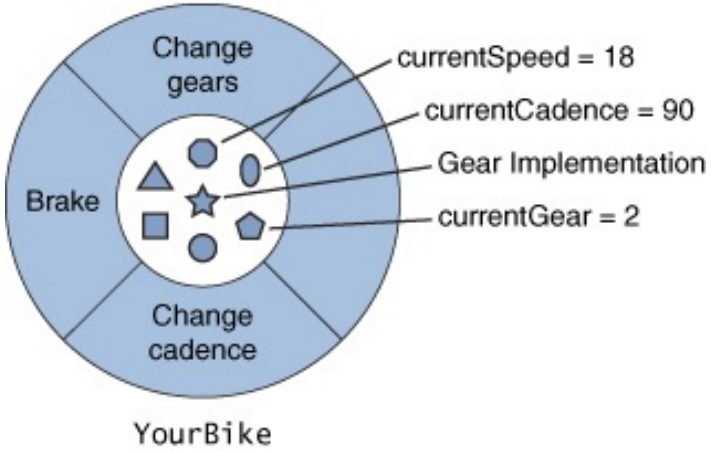
Class and its Objects

- Class as abstraction of objects









http://java.sun.com/docs/books/tutorial/java/concepts/class.html

Programming Paradigms

Procedural or Imperative (e.g., C)	Details of how computation is done	`i = 0; sum = 0;`
		`while (i < n) do {`
		`  i = i + 1;`
		`  sum = sum + i;`
		`}`

Functional (e.g., Lisp)	Describe what rather than how	`int sum(n: int) {`
		`  if n == 0 then 0`
		`  else n + sum(n - 1)`
		`}`

Logical (e.g., Prolog)	Describe rules/facts and answer queries	`edge(a, b).`
		`edge(a, c).`
		`edge(c, a).`
		`path(X, X).`
		`path(X, Y) :- edge(X, Y).`
		`path(X, Y) :-`
		`  edge(X, Z), path(Z, Y).`

Object-Oriented (e.g., Java)	Simulate objects and their interactions	`Price cost = new Price();` `cost.increasePrice(5);`

Procedural vs Object-Oriented

Programs = Data + Algorithms

- Emphasis on procedural abstraction.
- Emphasis on data abstraction.
- Top-down design;

  Step-wise refinement.
- Bottom-up design;

  Reusable libraries.
- Suited for programming in the small.
- Suited for programming in the large.

Introduction to Java

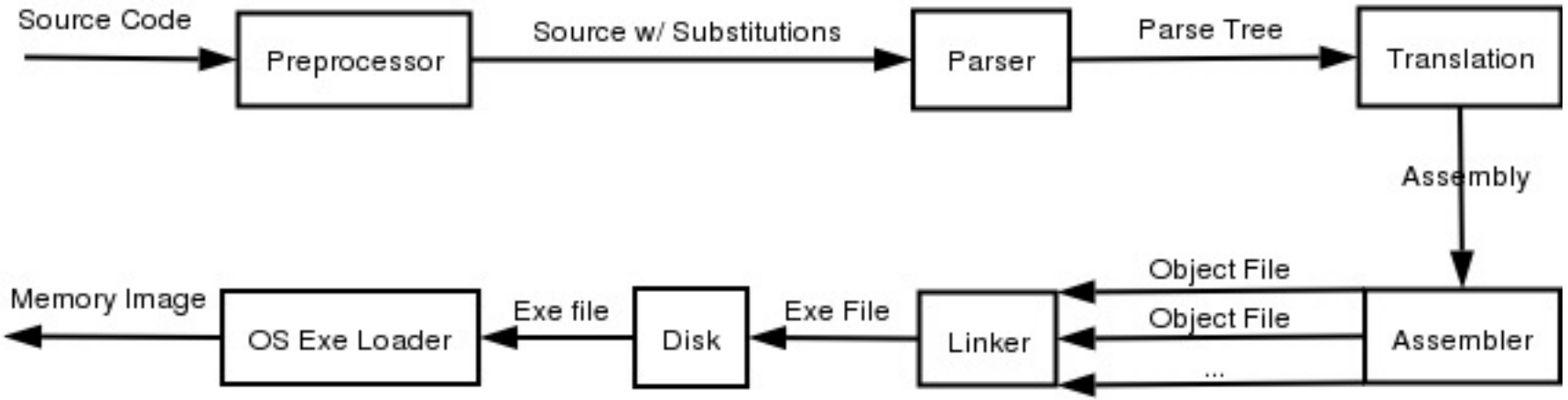
- Java was initially designed as a web programming language (e.g., Applets)

- It is later evolved into a popular general-purpose programming language for applications
  - The syntax for expressions and assignments is similar to that of C
  - The details about objects and classes, their relationships, and other high-level features are quite new

Why Java?

- Pure: A pure object-oriented programming language
- Comprehensive: Built-in supports for Internet, GUI's, concurrent, and network programming
- Extendable: more of a programming platform (e.g., J2EE)
- Highly portable: compiled once and executed everywhere
- …

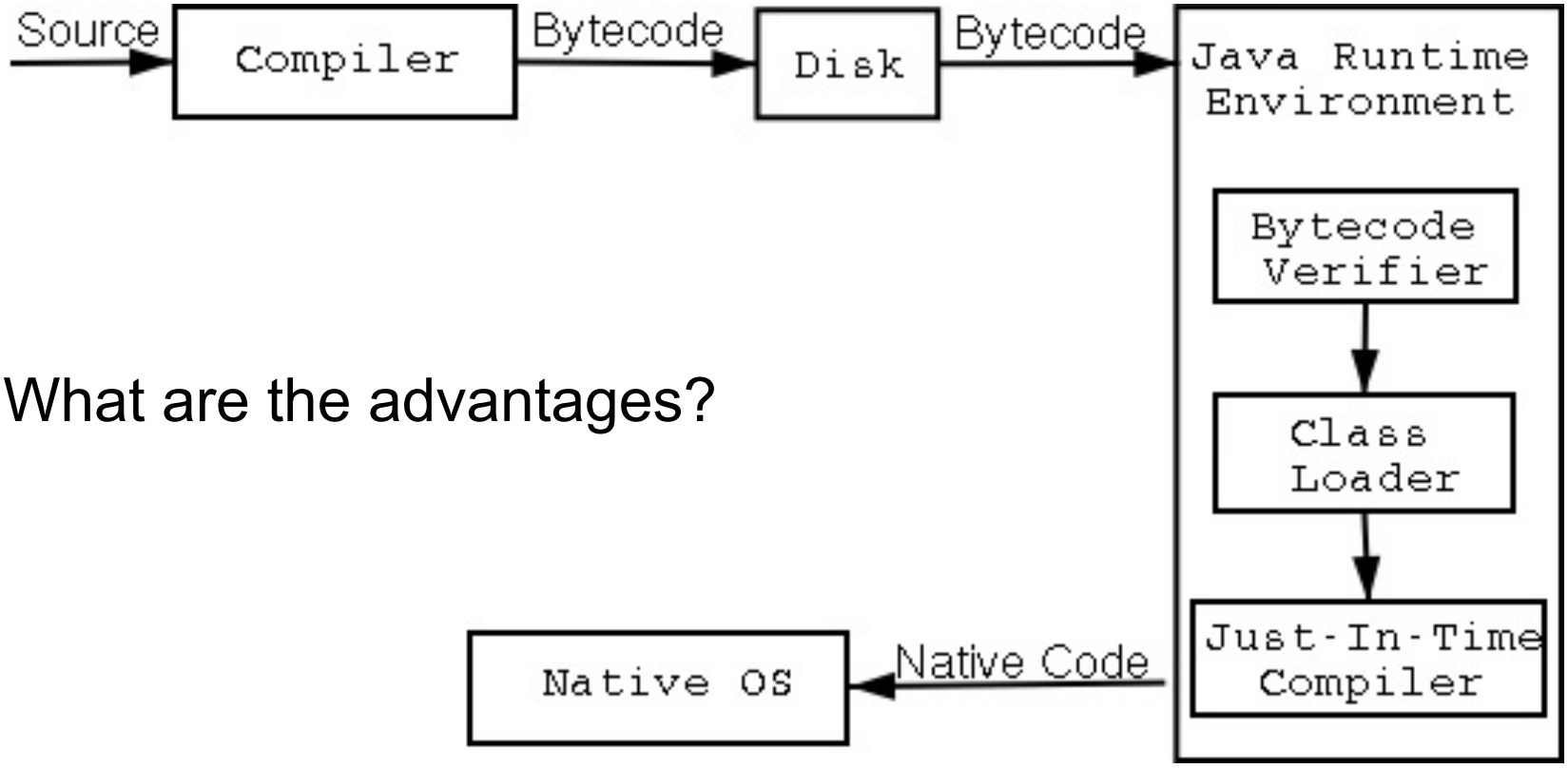
Major Compilation Steps



- What are the roles of a compiler?

http://www.acm.uiuc.edu/sigmil/RevEng/ch02.html

Java Compilation



- What are the advantages?

http://www.acm.uiuc.edu/sigmil/RevEng/ch02.html

Debugging and Errors

- The process of eliminating bugs is called *debugging*
  - *Bug: a mistake in a program*

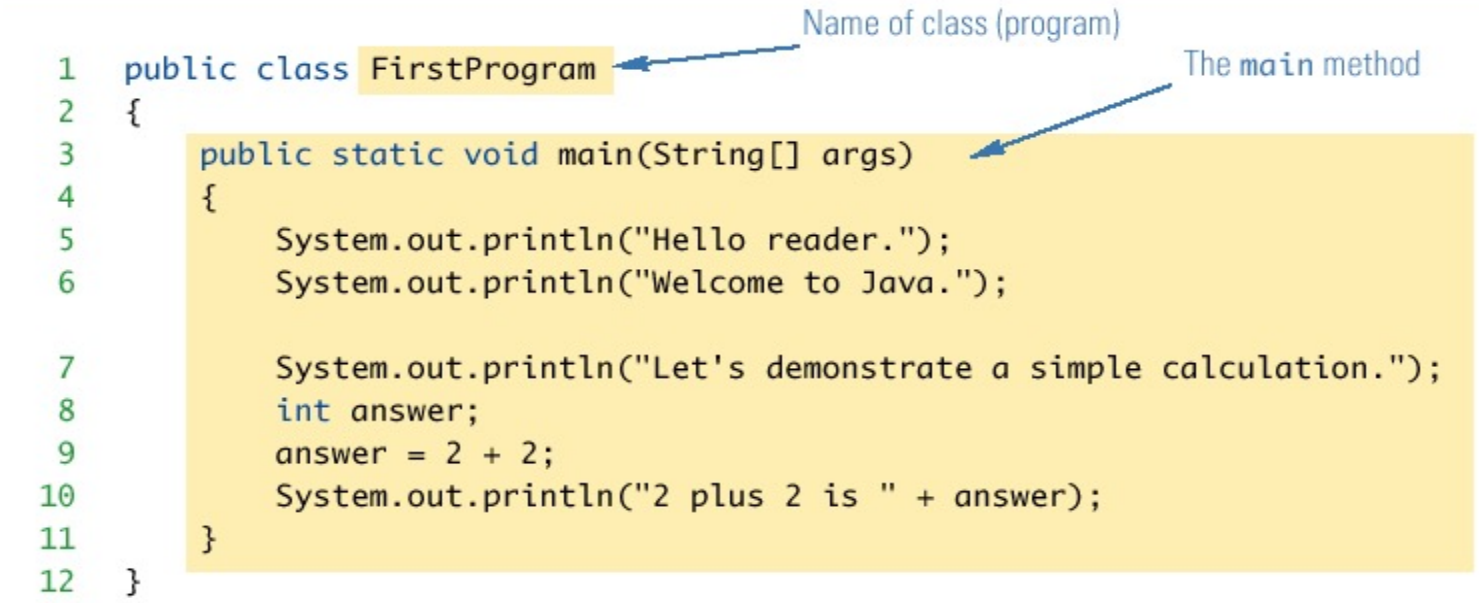
- *Syntax error:* a grammatical mistake in a program
  - The compiler can detect these errors, and will output an error message, explaining what it thinks the error is, and where it thinks the error is located.
  - Syntax errors are relatively easy to fix because they are explicitly identified.

Debugging and Errors

- *Run-time error:* an error that is not detected until a program is run
  - The compiler can't detect these errors: an error message is not generated after compilation, but after execution is ended prematurely.
- *Logic error:* a mistake in the underlying algorithm for a program
  - The compiler can't detect these errors, and no error message is generated after compilation and execution, but the program does not do what it is supposed to do.

Sample Application

Display 1.1 A Sample Java Program




```
1  public class FirstProgram
2  {
3      public static void main(String[] args)
4      {
5          System.out.println("Hello reader.");
6          System.out.println("Welcome to Java.");
7          System.out.println("Let's demonstrate a simple calculation.");
8          int answer;
9          answer = 2 + 2;
10         System.out.println("2 plus 2 is " + answer);
11     }
12 }
```

SAMPLE DIALOGUE 1

```
Hello reader.
Welcome to Java.
Let's demonstrate a simple calculation.
2 plus 2 is 4
```

Java Applications

- An application consists one or more classes and at least one of them should have a "main" method.
- Each class is saved into a file whose name is the same as the classname, e.g., FirstProgram.java.
- Compiling a program: "javac FirstProgram.java" will produce a byte-code file: FirstProgram.class.
- Running a program: "java FirstProgram" (no extension) will activate the "main" method in the given program.

Class Loader

- Java programs are divided into smaller parts called *classes*
  - Each class definition is normally stored in a separate file and compiled separately

- *Class Loader*: A program that loads the byte-code of the classes as needed to run a Java program
  - In other programming languages, the corresponding program is called a *linker.*

Identifiers

- Java identifiers must not start with a digit, and all the characters must be letters, digits, or the underscore symbol

- Java identifiers can theoretically be of any length

- Java is case-sensitive: **Rate**, **rate**, and **RATE** are different names.

Identifiers

- Keywords and reserved words: don't use them to name something else

  public class void static

- Predefined identifiers in libraries required by the Java language standard names: can be redefined but should be avoided

  System String println

Naming Conventions

- Start the names of variables, methods, and objects with a lowercase letter, indicate "word" boundaries with an uppercase letter, and restrict the remaining characters to digits and lowercase letters

  topSpeed bankRate1 timeOfArrival

- Start the names of classes with an uppercase letter

  FirstProgram MyClass String

Variable Declarations

- Every variable must be *declared* before its use:
  - A variable declaration tells the compiler what kind of data (type) will be stored in the variable
  - Variables are typically declared just before they are used or at the start of a block (indicated by a pair of braces { })

```
int numberOfBeans;
double oneWeight, totalWeight;
```

Primitive Data Types

Display 1.2 Primitive Types

TYPE NAME	KIND OF VALUE	MEMORY USED	SIZE RANGE
`boolean`	`true` or `false`	1 byte	not applicable
`char`	single character (Unicode)	2 bytes	all Unicode characters
`byte`	integer	1 byte	−128 to 127
`short`	integer	2 bytes	−32768 to 32767
`int`	integer	4 bytes	−2147483648 to 2147483647
`long`	integer	8 bytes	−9223372036854775808 to 9223372036854775807
`float`	floating-point number	4 bytes	$-3.40282347 \times 10^{+38}$ to $-1.40239846 \times 10^{-45}$
`double`	floating-point number	8 bytes	$\pm 1.76769313486231570 \times 10^{+308}$ to $\pm 4.94065645841246544 \times 10^{-324}$

Initializing Variables

- The declaration of a variable can be combined with its initialization via an assignment statement:

```
int count = 0;
double distance = 55 * .5;
char grade = 'A';
```

- In certain cases, uninitialized variables are given default values:
  - It is best not to rely on this
  - Explicitly initialized variables have the added benefit of improving program clarity.

Shorthand Assignment Statements

Example:	Equivalent To:
count += 2;	count = count + 2;
sum -= discount;	sum = sum – discount;
bonus *= 2;	bonus = bonus * 2;
time /= rushFactor;	time = time / rushFactor;
change %= 100;	change = change % 100;
amount *= count1 + count2;	amount = amount * (count1 + count2);

String Class

- There is no primitive type for strings in Java
- The class **String** is a predefined class used to store and process strings
- Objects of type **String** are made up of strings of characters:

```
String blessing = "Live long and prosper.";
```

Concatenation of Strings

- Use the + operator on two strings to get a longer string:

  - If **greeting** equals **"Hello "** and **javaClass** equals **"class"**, then **greeting + javaClass** equals **"Hello class"**

  - Any number of strings can be concatenated together

- When a string is combined with almost any other type of data, the result is a string:

  "The answer is " + 42 equals **"The answer is 42"**

String Methods

- The **String** class contains many useful methods:

  - A **String** method is called by writing a **String** object, a dot, the name of the method, and a pair of parentheses to enclose any arguments

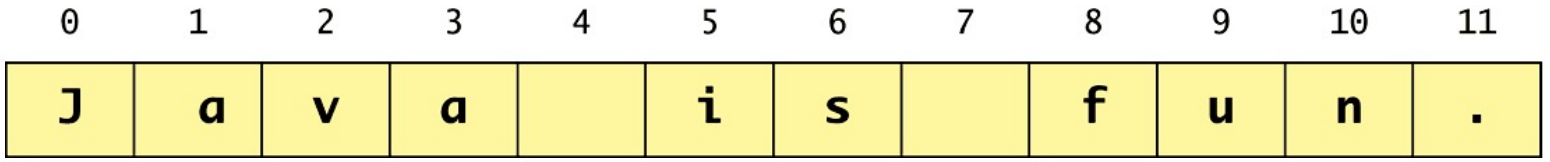
  - If a **String** method returns a value, then it can be placed anywhere that a value of its type can be used

```
String greeting = "Hello";
int count = greeting.length();
System.out.println("Length is " + greeting.length());
```

String Indexes

Display 1.5 String Indexes

The 12 characters in the string `"Java is fun."` have indexes 0 through 11.



Notice that the blanks and the period count as characters in the string.

Some String Methods

Display 1.4 Some Methods in the Class `String`

`int length()`

Returns the length of the calling object (which is a string) as a value of type `int`.

EXAMPLE

After program executes `String greeting = "Hello!";`
`greeting.length()` returns 6.

`boolean equals(Other_String)`

Returns `true` if the calling object string and the *Other_String* are equal. Otherwise, returns `false`.

EXAMPLE

After program executes `String greeting = "Hello";`
`greeting.equals("Hello")` returns `true`
`greeting.equals("Good-Bye")` returns `false`
`greeting.equals("hello")` returns `false`

Note that case matters. `"Hello"` and `"hello"` are not equal because one starts with an uppercase letter and the other starts with a lowercase letter.

(continued)

Some String Methods

Display 1.4 Some Methods in the Class `String`

`boolean equalsIgnoreCase(`*Other_String*`)`

Returns `true` if the calling object string and the *Other_String* are equal, considering uppercase and lowercase versions of a letter to be the same. Otherwise, returns `false`.

EXAMPLE

After program executes `String name = "mary!";`
`greeting.equalsIgnoreCase("Mary!")` returns `true`

`String toLowerCase()`

Returns a string with the same characters as the calling object string, but with all letter characters converted to lowercase.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.toLowerCase()` returns `"hi mary!"`.

(continued)

Some String Methods

Display 1.4 Some Methods in the Class `String`

`char charAt(`*Position*`)`

Returns the character in the calling object string at the *Position*. Positions are counted 0, 1, 2, etc.

EXAMPLE

After program executes `String greeting = "Hello!";`
`greeting.charAt(0)` returns `'H'`, and
`greeting.charAt(1)` returns `'e'`.

`String substring(`*Start*`)`

Returns the substring of the calling object string starting from *Start* through to the end of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position *Start* is included in the value returned.

EXAMPLE

After program executes `String sample = "AbcdefG";`
`sample.substring(2)` returns `"cdefG"`.

(continued)

Some String Methods

Display 1.4 Some Methods in the Class `String`

`String substring(`*Start*`, `*End*`)`

Returns the substring of the calling object string starting from position *Start* through, but not including, position *End* of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position *Start* is included in the value returned, but the character at position *End* is not included.

EXAMPLE

After program executes `String sample = "AbcdefG";`
`sample.substring(2, 5)` returns `"cde"`.

`int indexOf(`*A_String*`)`

Returns the index (position) of the first occurrence of the string *A_String* in the calling object string. Positions are counted 0, 1, 2, etc. Returns −1 if *A_String* is not found.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.indexOf("Mary")` returns 3, and
`greeting.indexOf("Sally")` returns −1.

(continued)

Coding Conventions

```
public class Pet
{
    private String name;
    private int age;          //in years
    private double weight;    //in pounds

    public String toString( )
    {
        return ("Name: " + name + " Age: " + age + " years"
                         + "\nWeight: " + weight + " pounds");
    }

    /**comment about the method here- for api documentation*/
    public Pet(String initialName, int initialAge, double initialWeight)
    {
        name = initialName;
        if ((initialAge < 0) || (initialWeight < 0)) {
            System.out.println("Error: Negative age or weight.");
            System.exit(0);
        } else {
            age = initialAge;  //programmer only (and very few)
            weight = initialWeight;
        }
    }
...
```

Comments and Named Constants

Display 1.8 Comments and a Named Constant

```
1  /**
2   Program to show interest on a sample account balance.
3   Author: Jane Q. Programmer.
4   E-mail Address: janeq@somemachine.etc.etc.
5   Last Changed: September 21, 2004.
6  */
7  public class ShowInterest
8  {
9      public static final double INTEREST_RATE = 2.5;

10     public static void main(String[] args)
11     {
12         double balance = 100;
13         double interest; //as a percent

14         interest = balance * (INTEREST_RATE/100.0);
15         System.out.println("On a balance of $" + balance);
16         System.out.println("you will earn interest of $"
17                                                 + interest);
18         System.out.println("All in just one short year.");
19     }
20
21 }
```

Although it would not be as clear, it is legal to place the definition of INTEREST_RATE here instead.

SAMPLE DIALOGUE

```
On a balance of $100.0
you will earn interest of $2.5
All in just one short year.
```

Formatted Output

Display 2.1 **Format Specifiers for `System.out.printf`**

CONVERSION CHARACTER	TYPE OF OUTPUT	EXAMPLES
d	Decimal (ordinary) integer	`%5d` `%d`
f	Fixed-point (everyday notation) floating point	`%6.2f` `%f`
e	E-notation floating point	`%8.3e` `%e`
g	General floating point (Java decides whether to use E-notation or not)	`%8.3g` `%g`
s	String	`%12s` `%s`
c	Character	`%2c` `%c`

Keyboard Input Demonstration

Display 2.6 Keyboard Input Demonstration

```
1  import java.util.Scanner;

2  public class ScannerDemo
3  {
4     public static void main(String[] args)
5     {
6        Scanner keyboard = new Scanner(System.in);

7        System.out.println("Enter the number of pods followed by");
8        System.out.println("the number of peas in a pod:");
9        int numberOfPods = keyboard.nextInt();
10       int peasPerPod = keyboard.nextInt();

11       int totalNumberOfPeas = numberOfPods*peasPerPod;

12       System.out.print(numberOfPods + " pods and ");
13       System.out.println(peasPerPod + " peas per pod.");
14       System.out.println("The total number of peas = "
15                                          + totalNumberOfPeas);
16    }
17 }
```

Makes the `Scanner` *class available to your program.*

Creates an object of the class `Scanner` *and names the object* `keyboard`.

Each reads one `int` *from the keyboard*

(continued)

Keyboard Input Demonstration

Display 2.6 Keyboard Input Demonstration

SAMPLE DIALOGUE 1

```
Enter the number of pods followed by
the number of peas in a pod:
22 10
22 pods and 10 peas per pod.
The total number of peas = 220
```

The numbers that are input must be separated by whitespace, such as one or more blanks.

SAMPLE DIALOGUE 2

```
Enter the number of pods followed by
the number of peas in a pod:
22
10
22 pods and 10 peas per pod.
The total number of peas = 220
```

A line break is also considered whitespace and can be used to separate the numbers typed in at the keyboard.

Keyboard Input Demonstration (2)

Display 2.7 Another Keyboard Input Demonstration

```
1  import java.util.Scanner;

2  public class ScannerDemo2
3  {
4      public static void main(String[] args)
5      {
6          int n1, n2;
7          Scanner scannerObject = new Scanner(System.in);

8          System.out.println("Enter two whole numbers");
9          System.out.println("seperated by one or more spaces:");

10         n1 = scannerObject.nextInt();
11         n2 = scannerObject.nextInt();
12         System.out.println("You entered " + n1 + " and " + n2);

13         System.out.println("Next enter two numbers.");
14         System.out.println("Decimal points are allowed.");
```

Creates an object of the class `Scanner` *and names the object* `scannerObject`.

Reads one `int` *from the keyboard.*

(continued)

Keyboard Input Demonstration (2)

Display 2.7 Another Keyboard Input Demonstration

```
15          double d1, d2;
16          d1 = scannerObject.nextDouble();
17          d2 = scannerObject.nextDouble();
18          System.out.println("You entered " + d1 + " and " + d2);

19          System.out.println("Next enter two words:");

20          String word1 = scannerObject.next();
21          String word2 = scannerObject.next();
22          System.out.println("You entered \"" +
23                                  word1 + "\" and \"" + word2 + "\"");

24          String junk = scannerObject.nextLine(); //To get rid of '\n'

25          System.out.println("Next enter a line of text:");
26          String line = scannerObject.nextLine();
27          System.out.println("You entered: \"" + line + "\"");
28      }
29  }
```

Reads one `double` *from the keyboard.*

Reads one word from the keyboard.

Reads an entire line.

This line is explained in the Pitfall section "Dealing with the Line Terminator, '\n'"

(continued)

Keyboard Input Demonstration (2)

Display 2.7 **Another Keyboard Input Demonstration**

SAMPLE DIALOGUE

```
Enter two whole numbers
separated by one or more spaces:
  42   43
You entered 42 and 43
Next enter two numbers.
A decimal point is OK.
 9.99  57
You entered 9.99 and 57.0
Next enter two words:
jelly beans
You entered "jelly" and "beans"
Next enter a line of text:
Java flavored jelly beans are my favorite.
You entered "Java flavored jelly beans are my favorite."
```

Pitfall with Line Terminator '\n'

- Method **nextLine** of class **Scanner** reads the remainder of a line of text starting wherever the last keyboard reading left off
- Given the code:

```
Scanner keyboard = new Scanner(System.in);
int n = keyboard.nextInt();
String s1 = keyboard.nextLine();
String s2 = keyboard.nextLine();
```

and the input:

```
2
Heads are better than
1 head.
```

What are the values of **n**, **s1**, and **s2**?

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.
  - Use **compareTo** method to test if one string is ordered before another string.

- 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
```

String Split Example

```
import java.util.Scanner;

public class StringSplitDemo
{
    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
        String[] names = line.split(delimiters);

        String middleName = names[2];
        if( middleName.equalsIgnoreCase("None"))
           middleName = "";    // Empty string
        System.out.println("Hello " + names[0] +
                           " " + middleName + " " + names[1]);
    }
}
```

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

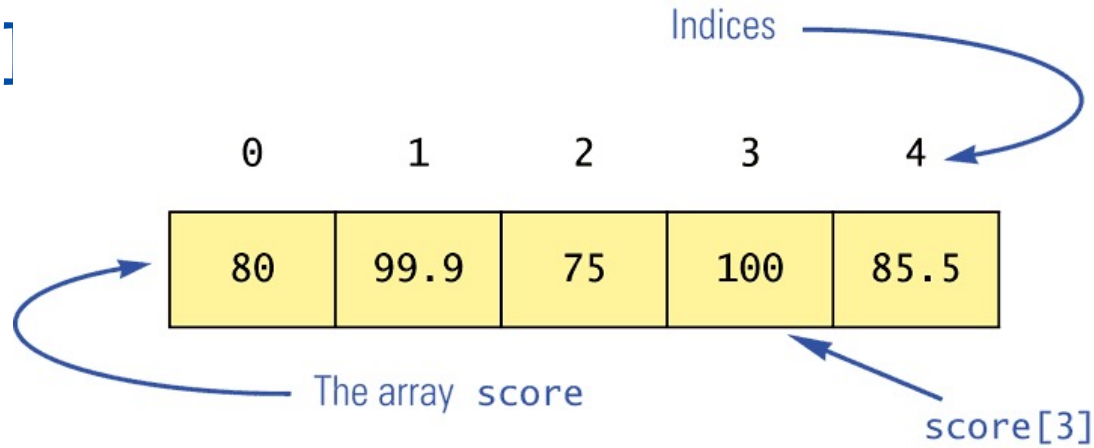
- Java arrays are indexed from zero:

  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 as 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