Learning the Java Language

2.1 Object-Oriented Programming
What is an Object?

- Real world is composed by different kind of objects: buildings, men, women, dogs, cars, etc.
- Each object has its own *states* and *behaviors*
What is a Software Object?

- Software designers use the same idea to ease programmer to develop their software.
- A software is also composed by different kind of software objects.
- Each software object also has its own *states* and *behaviors*.

**Variables (States)**
- Color = Grey
- Size = 2cm x 2cm
- Shape = Rectangular
  (protruded)

**Method (Behavior)**
- Press ()
Encapsulation

- **Hiding information** within an object’s nucleus
- **Provide a public interface** for interacting with it
- **Advantages to software developers**
  - **Modularity**: An object can be easily passed around in the system (Otherwise, you need to think about how many files you need to bundle together to pass to your friends)
  - **Information hiding**: Users need not go into details of the object before using the object (E.g., you don’t need to know the circuit of a TV set if you want to watch TV)
  - **Safety**: Users of the object will not direct access the internal state of the object. This reduces the possibility of erroneous situation
What is a Class?

- A Class is a **blueprint or prototype** that defines the variables and methods common to all objects of a certain kind.
- Every object of a Class is just an **instance** of that class.
- **Benefit - Reusability**
  - This arrangement **saves effort** in developing a number of objects of the same kind.

Usually an object is used a number of times in an application.
The Button Class:

**Variables**
- Color = Grey
- Size = 2cm x 2cm
- Shape = Rectangular (protrude)

**Method**
- Press ()
What is Inheritance?

- A Class often has Subclasses which give more specific description of the object
- All subclasses inherit all states of the parent class
Variables
Color = Grey
Size = 2cm x 2cm
Shape = Rectangular (protrude)

Method
Press ()

Play Sound Button
Subclass
• Inherit all variables and methods
• Add other variables and methods
• Override some superclass variables and methods

Overriding

Variables
Superclass variables
Color = Grey with Dark Grey icon
*Action = para*

Method
Superclass methods
set(para)
We can have more than one level of inheritance

Variables
Superclass variables
*Action = Applause*
**loudness = para**

Method
Superclass methods
set(para)

Overriding
Benefit of Inheritance

• We need not re-define every aspect of a class from very beginning again

• All we need to do is to find out what in addition the subclass is different from the parent class
What is a Message?

- Software objects interact and communicate with each other by sending messages.
Java is an OO language

```java
/**
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string.
    }
}
```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello world!", 50, 25);
    }
}

Method
Variables

Applet Class

Method
paint(Graphics g)

Variables

HelloWorld Subclass
Exercise 1

Make use of the Java Workshop 2.0 to create a project to run the applet "HelloWorld"

The result should look like the one on the right
Exercise 2

Make use of the Java Workshop 2.0 to create a project to run the Standalone application "Count"

It counts the no. of characters in an ASCII file

Try to run Count on any ASCII file to count the no. of characters

```java
import java.io.*;

public class Count {
    public static void countChars(InputStream in) throws IOException {
        int count = 0;
        while (in.read() != -1) {
            count++;
            System.out.println("Counted "+ count + " chars.");
        }
    }

    public static void main(String[] args) throws Exception {
        if (args.length >= 1) {
            countChars(new FileInputStream(args[0]));
        } else {
            System.err.println("Usage: Count filename");
        }
    }
}
```
Learning the Java Language

2.2 The Nuts and Bolts
Basic Constructs of a Java Program

a. Classes, Methods and Objects
b. Variables and Data Types
c. Operators
d. Expressions
e. Control Flow Statements
f. Arrays and Strings
A. Classes, Methods and Objects

• Classes define the **states and behaviors** of objects
• The **states** of an object are defined by its **variables**
• The **behaviors** of an object are defined by its **methods**

• The class **Count** has two methods: **countChars** and **Main**
• Every Java program must have a **Main** method
• It is the start of the program

```java
import java.io.*;
public class Count {
    public static void main(String[] args) throws Exception {
        if (args.length >= 1)
            countChars(new FileInputStream(args[0]));
        else
            System.err.println("Usage: Count filename");
    }
}
```
• A method often needs objects from other classes to help its operation

• The `countChars` method processes the object named `in`, which belongs to the `InputStream` class

• One of the methods contained in the `InputStream` class is `read`

• It will read the data of the stream one-by-one

```java
import java.io.*;

public class Count {
    public static void countChars(InputStream in) throws IOException {
        int count = 0;
        while (in.read() != -1)
            count++;
        System.out.println("Counted " + count + " chars.");
    }
    // … main method omitted … //
}
```

Note: we only refer to the `read` method of a particular object named `in`. We do not refer to other objects of this class.
Exercise: Rename the method `main` to `start`, note the error message when building the project
B. Variables and Data Types

- Variables are entities that act or are acted upon in the software object
- In the method `countChars`, two variables, `in` and `count`, are defined

- A variable must be declared once in a program
- A variable declaration always contains two components
  - Type and Name

```java
import java.io.*;

public class Count {
    public static void countChars(InputStream in) throws IOException {
        int count = 0;
        while (in.read() != -1)
            count++;
        System.out.println("Counted "+ count + " chars.");
    }

    // … main method omitted … //
}
```
Data types - Two categories: primitive and reference

- Primitive Types - usually contains a single value of particular size and format
  - hardware independent

<table>
<thead>
<tr>
<th>Type</th>
<th>Size/Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Integers</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>8-bit two’s complement</td>
<td>Byte-length integer</td>
</tr>
<tr>
<td>short</td>
<td>16-bit two’s complement</td>
<td>Short integer</td>
</tr>
<tr>
<td>int</td>
<td>32-bit two’s complement</td>
<td>Integer</td>
</tr>
<tr>
<td>long</td>
<td>64-bit two’s complement</td>
<td>Long Integer</td>
</tr>
<tr>
<td><strong>Real Numbers</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>32-bit IEEE 754</td>
<td>Single-precision floating-point</td>
</tr>
<tr>
<td>double</td>
<td>64-bit IEEE 754</td>
<td>Double-precision floating-point</td>
</tr>
<tr>
<td><strong>Other Types</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>16-bit Unicode character</td>
<td>A single character</td>
</tr>
<tr>
<td>boolean</td>
<td>true or false</td>
<td>A boolean value (true or false)</td>
</tr>
</tbody>
</table>
Arrays, classes, and interfaces are reference types
A reference type variable is a reference to the actual value or set of values represented by the variable
For example: Just like the address of your friend
   The address is not your friend, it is just a way to reach your friend

Count is primitive type variable
in is reference type variable

```java
import java.io.*;
public class Count {
    public static void countChars(InputStream in) throws IOException {
        int count = 0;
        while (in.read() != -1):
        }
    }
}
```

// ... main method omitted ... //
Scope

- A variable’s scope is the block of code within which the variable is accessible
- The location of the variable declaration within the program determines its scope
- Depending on the locations, a variable can be categorized into:
  - Member variable
  - Local variable
  - Method parameter
  - Exception-handler parameter
class MyClass
{
    ...
    member variable declarations
    ...
    public void aMethod(method parameters)
    {
        ...
        local variable declarations
        ...
        catch (exception handler parameters)
        {
            ...
        }
    }
}
• In the Count class, there is no Member Variable
• In `countChars`, `in` is method parameter
  • Its scope includes
• In `countChars`, `count` is local variable
  • Its scope includes
• No exception handling parameter is found

```java
import java.io.*;

public class Count {
    public static void countChars(InputStream in) throws IOException {
        int count = 0;
        while (in.read() != -1) {
            count++;
        }
        System.out.println("Counted " + count + " chars.");
    }

    public static void main(String[] args) throws Exception {
    }
}
```
Exercise: Modify the following statement in the Count class

```
int Count = 0;
```

to

```
int Count = 10000;
```

Note the output when executing the program
Exercise:
Identify the member variables, method parameters, local variables, exception handling parameters of the following program and determine their scopes.

```java
import java.io.*;
class Parent {
    public int aNum = 2;
    public int increment (int bNum) {
        return (bNum + 1);
    }
}
public class Child extends Parent{
    int aNum;
    public Child () {
        this.aNum = increment( super.aNum);
    }
    public static void main (String[] args) {
        Child t = new Child();
        System.out.println("Result = " + t.aNum);
    }
}
```
C. Operators

- The operators in Java can be divided into 4 categories
  - arithmetic
  - relational and conditional
  - bitwise and logical
  - assignment
- These operators can be unary
  - e.g. `count++;` // arithmetic operator: autoincrement count
  - e.g. `count = 0;` // assignment operator: assign 0 to count
- or binary
- or tertiary
  - e.g. `expr ? op1 : op2` // if expr is true, then do op1, otherwise do op2
## Java arithmetic operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Use</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>op1 + op2</td>
<td>Adds op1 and op2</td>
</tr>
<tr>
<td>-</td>
<td>op1 – op2</td>
<td>Subtracts op2 from op1</td>
</tr>
<tr>
<td>*</td>
<td>op1 * op2</td>
<td>Multiplies op1 and op2</td>
</tr>
<tr>
<td>/</td>
<td>op1 / op2</td>
<td>Divide op1 by op2</td>
</tr>
<tr>
<td>%</td>
<td>op1 % op2</td>
<td>Computes the remainder of dividing op1 by op2</td>
</tr>
</tbody>
</table>

- The `+` operator can also be used to concatenate strings
e.g. `System.out.println ("Counted" + count + " chars.");`
- The `++` and `--` operators are the shorthand of increment by 1 and decrement by 1
e.g. `count++; // equivalent to count = count + 1;`
# Java relational operators

<table>
<thead>
<tr>
<th>Relational Operator</th>
<th>Use</th>
<th>Return true if ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>op1 &gt; op2</td>
<td>op1 is greater than op2</td>
</tr>
<tr>
<td>&gt;=</td>
<td>op1 &gt;= op2</td>
<td>op1 is greater than or equal to op2</td>
</tr>
<tr>
<td>&lt;</td>
<td>op1 &lt; op2</td>
<td>op1 is less than op2</td>
</tr>
<tr>
<td>&lt;=</td>
<td>op1 &lt;= op2</td>
<td>op1 is less than or equal to op2</td>
</tr>
<tr>
<td>==</td>
<td>op1 == op2</td>
<td>op1 and op2 are equal</td>
</tr>
<tr>
<td>!=</td>
<td>op1 != op2</td>
<td>op1 and op2 are not equal</td>
</tr>
<tr>
<td>instanceof</td>
<td>op1 instanceof op2</td>
<td>op1 and op2 are assignment compatible</td>
</tr>
</tbody>
</table>
# Java conditional operators

<table>
<thead>
<tr>
<th>Conditional Operator</th>
<th>Use</th>
<th>Return true if ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>op1 &amp;&amp; op2</td>
<td>op1 and op2 are both true; conditionally evaluates op2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>op1 ! op2</td>
<td>op1 is false</td>
</tr>
<tr>
<td>&amp;</td>
<td>op1 &amp; op2</td>
<td>op1 and op2 are both true; always evaluates op1 and op2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>op1</td>
</tr>
</tbody>
</table>

- **E.g.**  
  \[ a = 0; b = 0; c = 0; \text{if} \ (a++ \ == \ 0 \ \&\& \ b++ \ == \ 0) \ \{ \ c = 10; \} \]  
  Answer: \( a = 1; b = 0; c = 0. \)

- **E.g.**  
  \[ a = 0; b = 0; c = 0; \text{if} \ (a++ \ == \ 0 \ \& \ b++ \ == \ 0) \ \{c = 10;\} \]  
  Answer: \( a = 1; b = 1; c = 0. \)
# Java bitwise and logical operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Use</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;&gt;</td>
<td>op1 &gt;&gt; op2</td>
<td>Shifts bits of op1 right by distance op2</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>op1 &lt;&lt; op2</td>
<td>Shifts bits of op1 left by distance op2</td>
</tr>
<tr>
<td>&gt;&gt;&gt;</td>
<td>op1 &gt;&gt;&gt; op2</td>
<td>Shifts bits of 1 right by distance op2 (unsigned)</td>
</tr>
<tr>
<td>&amp;</td>
<td>op1 &amp; op2</td>
<td>Performs the bitwise and.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Performs the bitwise or.</td>
</tr>
<tr>
<td>^</td>
<td>op1 ^ op2</td>
<td>Performs the bitwise xor.</td>
</tr>
<tr>
<td>~</td>
<td>~ op1</td>
<td>Performs the bitwise complement.</td>
</tr>
</tbody>
</table>

- **E.g.**  
  
  ```java
  int a = -2; a = a >> 1; // a = 111 ... 110; a >>1 = 111 ... 111;
  Answer: a = -1.
  ```

- **E.g.**  
  
  ```java
  int a = -2; a = a >>> 1; // a = 111 ... 110; a >>> 1 = 011 ... 111
  Answer: a = 2,147,483,647.
  ```
• The operators & and | are bitwise logical operators
• They can be applied to different types of variables
• E.g. boolean $a = true; a = false; c = a & b$;
  Answer: $c = false$.
• E.g. byte $a = 1; b = 2; c = a & b$;
  // $a = 00000001; b = 00000010$;
  Answer: $c = 0$  // $c = a & b = 00000000$. 
### Java assignment operators

- Java provides several shortcut assignment operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Use</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>+=</td>
<td>op1 += op2</td>
<td>op1 = op1 + op2</td>
</tr>
<tr>
<td>-=</td>
<td>op1 -= op2</td>
<td>op1 = op1 - op2</td>
</tr>
<tr>
<td>*=</td>
<td>op1 *= op2</td>
<td>op1 = op1 * op2</td>
</tr>
<tr>
<td>/=</td>
<td>op1 /= op2</td>
<td>op1 = op1 / op2</td>
</tr>
<tr>
<td>%=</td>
<td>op1 %= op2</td>
<td>op1 = op1 % op2</td>
</tr>
<tr>
<td>&amp;=</td>
<td>op1 &amp;= op2</td>
<td>op1 = op1 &amp; op2</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>op1</td>
</tr>
<tr>
<td>^=</td>
<td>op1 ^= op2</td>
<td>op1 = op1 ^ op2</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>op1 &lt;&lt;= op2</td>
<td>op1 = op1 &lt;&lt; op2</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>op1 &gt;&gt;= op2</td>
<td>op1 = op1 &gt;&gt; op2</td>
</tr>
<tr>
<td>&gt;&gt;&gt;=</td>
<td>op1 &gt;&gt;&gt;= op2</td>
<td>op1 = op1 &gt;&gt;&gt; op2</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;&gt;=</td>
<td>op1 &gt;&gt;&gt;&gt;&gt;= op2</td>
<td>op1 = op1 &gt;&gt;&gt;&gt; op2</td>
</tr>
</tbody>
</table>
**Exercise:** Modify the program of Count such that you will print out a message to indicate the total number of characters is an even or odd number.

**Hint:** Add an if statement to test the total number of characters is odd or even
Test can be done by getting the remainder when the character number is divided by 2
D. Expressions

• An **expression** is a series of **variables**, **operators**, and **method calls** that evaluates to a single value
  
  E.g. `count++;`  // variable: count; operators: ++; return count = count +1
  
  E.g. `in.read();` // method call: read(); result: return of in.read()

• A Java statement can comprise several expressions
  
  E.g. `in.read() != -1;`  // Two expressions in this statement

• The evaluation of expression is usually from left to right
  
  E.g. `in.read() != -1;`  // in.read() is evaluated first followed by the relational operator !=

• The order can be changed due to the precedence of operators
  
  E.g. `x + y / 100;`  // division will do first and then addition
  
  E.g. `(x + y) / 100;`  // addition done first due to the ( )
E. Control Flow Statements

• The **while** statement performs some actions while a certain condition remains **true**

• General syntax

```java
while (expression) {
    statement;
}
```

// do statement if expression is true

• The `{}` can be omitted if the **statement** has only one statement

```java
import java.io.*;

public class Count {
    public static void countChars(InputStream in) throws IOException {
        int count = 0;
        while (in.read() != -1) {
            count++;
        }
        System.out.println("Counted " + count + " chars.");
    }
}
```

// … main method omitted … //
• Besides *while*, Java supports several other control flow statements

<table>
<thead>
<tr>
<th>Type of Statement</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decision-making</td>
<td>if-else, switch-case</td>
</tr>
<tr>
<td>Loop</td>
<td>for, while, do-while</td>
</tr>
<tr>
<td>Exception</td>
<td>try-catch-finally, throw</td>
</tr>
<tr>
<td>Branching</td>
<td>break, continue, label::, return</td>
</tr>
</tbody>
</table>

• Normal program execution is from up to down sequentially

• These statements change the flow of program execution depended on the condition
Decision Making

- **if-else** statement is particularly suitable for making decision based on a **ranges of values or conditions**
- E.g.

```java
int testscore;
char grade;

if (testscore >= 90) {
    grade = 'A';
} else if (testscore >= 80) {
    grade = 'B';
} else if (testscore >= 70) {
    grade = 'C';
} else if (testscore >= 60) {
    grade = 'D';
} else {
    grade = 'F';
}
```
switch-case statement is particularly suitable for making decision based on a single value. E.g.,

```java
int month;
switch (month) {
    case 1: System.out.println("January"); break;
    case 2: System.out.println("February"); break;
    case 3: System.out.println("March"); break;
    case 4: System.out.println("April"); break;
    case 5: System.out.println("May"); break;
    case 6: System.out.println("June"); break;
    case 7: System.out.println("July"); break;
    case 8: System.out.println("August"); break;
    case 9: System.out.println("September"); break;
    case 10: System.out.println("October"); break;
    case 11: System.out.println("November"); break;
    case 12: System.out.println("December"); break;
}
```
The above example can also be implemented by `if-else` statement. However, it will be much longer

```java
int month;
...
if (month == 1) {
    System.out.println("January");
} else if (month == 2) {
    System.out.println("February");
    :
    :
} else if (month == 11) {
    System.out.println("November");
} else if (month == 12) {
    System.out.println("December");
}
...
The **switch-case** statement can combine with the **break** or **default** statements. E.g.,

```java
int month, numDays;
switch (month)
{
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        numDays = 31;
        break;
    case 2:
        if ( ((year % 4 == 0) &&
            !(year % 100 == 0)) ||
            (year % 400 == 0) )
            numDays = 29;
        else
            numDays = 28;
        break;
    default: System.out.println("Hey, that’s not a valid month!");
            break;
}
```
Loop Statements

- Java supports three loop statements
  - while, for, and do-while
- *For* loop is useful when we know the constraints of the loop. E.g.,

```java
// calculate the factorial 10 //
int i;
int length = 10;
int fac = 1;
for (i = 1; i <= length; i++) {
    fac *= i;
}
The loop will stop when i = 11
```
• The **do-while** loop is less commonly used
• It is useful if the loop has to do **at least once**
• It does the statement before testing the condition
• For example, the *countChars* method can be modified as follows:

```java
int c; InputStream in; ... do {
    c = in.read();
    ... 
} while (c != -1);
```

It is exactly the same as the while loop used before
Branching Statements

- Java supports three branching statements
  - break, return (more commonly used) and continue (less used)
- The **break** statements have labeled or unlabeled form. E.g.,

```java
// calculate the factorials of 3,5,7,9,11; Quit if > 1048576
int i, j, fac = 1;
factorial:
for (j = 3; j <= 11; j+=2) {
    fac = 1;
    for (i = 1; i<=j; i++) {
        fac *= i;
        if (fac > 1048576) {
            System.out.println("Too big");
            break factorial;
        }
    }
}System.out.println("Factorial of "+j+" = "+fac);
```
Exercise: Build the program on the last page. Note the results.

Modify the statement `break factorial` to `break`.
Execute the program again. Note the results.
Do you get the correct results? Why?
F. Arrays and Strings

- Like other languages, Java allows one to collect and manage multiple values through an array object.
- Declaration of array is simple, e.g.
  - `int[] arrayOfInts;` // arrayOfInts is an array of integers.
- Note that just declaration is not enough in order to use an array. One needs to allocate memory for it.
- Memory can be allocated to array using the `new` operator. E.g.
  - `int[] arrayOfInts = new int[10];`
    - // arrayOfInts is an array of integers with 10 elements.
Now, you can use it. E.g.

```java
for (int j = 0; j <= arrayOfInts.length; j++) {
    arrayOfInts[j] = j;
    System.out.println("[j] = " + arrayOfInts[j]);
}
```

Result:

```
[0] = 0
[1] = 1
...
[arrayOfInts.length] = arrayOfInts.length
```

Note that `length` is a property provided for all Java arrays.

It gives the **number of elements** of an array.
An array of string called `args` is passed to the `main` method.

The Java runtime allocates the space for the `args` array.

String `str`  
A sequence of characters

String `[ ] args`  
Many sequences of characters

```java
import java.io.*;
public class Count {
    // … countChars method omitted … //
    public static void main(String[] args) throws Exception {
        if (args.length >= 1)
            countChars(new FileInputStream(args[0]));
        else
            System.err.println("Usage: Count filename");
    }
}
```
Exercise: Modify the program of Count such that it will show the number of characters of two files. Both the filename and its number of characters are printed out on the screen. The filenames are input in the command line.