Announcements

- Exam 1
  - Wed. Feb 18th 6:30pm -- 7:30pm MTHW 210
- Must have signed Academic Integrity Policy.
- Check newsgroup.
- You are responsible for Chapters 0 -- 5 from the textbook.
- 20 multiple choice; 3 programming questions.
Chapter 5

Selection Statements

CS 180
Sunil Prabhakar
Department of Computer Science
Purdue University
Objectives

After this week, you should be able to

- Implement a selection control using
  - `if` statements
  - `switch` statements

- Understand boolean expressions
- Understand nested `if` statements
- Describe how objects are compared
Choices

Consider a program that:

- repeatedly calculates monthly loan payments until we enter $0; or
- checks that you have not entered a negative value for the LoanCalculator program; or
- inputs a number and reports whether it is even (or a perfect square, prime number etc.).

These represent choices of which statement to execute next.

E.g. if the input is negative, print out an error, else compute the payments as usual.

Java’s `if` and `switch` statements are used for this purpose.
Flow of control

- Once a statement is executed, the next statement of the program is executed.
- Calling a method transfers the control to the statements in the method.
- Once the method returns, control returns to statement that made the call.
- Changing this flow of control is achieved using **if** and **switch** (and other) statements.
- These are called control flow statements.
if ( testScore < 70 )
    JOptionPane.showMessageDialog(null, "You did not pass");
else
    JOptionPane.showMessageDialog(null, "You passed");
if (testScore < 70)
    JOptionPane.showMessageDialog(null, "You did not pass");
else
    JOptionPane.showMessageDialog(null, "You passed ");
Boolean expressions

- `boolean` is a primitive data type.
- A boolean expression can take only two values: `true` or `false`.
- A simple boolean expression compares two values using a relational operator, e.g.
  - `testScore < 70`
  - `j > i`
  - `balance == 100;`
Relational Operators

<  // less than
<=  // less than or equal to
==  // equal to
!=  // not equal to
>  // greater than
>=  // greater than or equal to

testScore < 80

testScore * 2 >= 350

30 < w / (h * h)

x + y != 2 * (a + b)

2 * Math.PI * radius <= 359.99
### Compound Statements

- Use braces if the `<then>` or `<else>` block has multiple statements.

```java
if (testScore < 70)
{
    JOptionPane.showMessageDialog(null, "You did not pass");
    JOptionPane.showMessageDialog(null, "Try harder next time");
}
else
{
    JOptionPane.showMessageDialog(null, "You did pass");
    JOptionPane.showMessageDialog(null, "Keep up the good work");
}
```
if (<boolean expression>) {
    ...
}
else {
    ...
}

if (<boolean expression>) {
    ...
}
else {
    ...
}
else is optional

```java
if ( testScore >= 95 )
    JOptionPane.showMessageDialog(null, "You are an honor student");
```
if (testScore >= 95) {
    JOptionPane.showMessageDialog(null, "You are an honor student");
}

if (testScore >= 95) {
    JOptionPane.showMessageDialog(null, "You are an honor student");
    next statement;
} else {
    previous statement;
    is testScore >= 95 ?
    JOptionPane.showMessageDialog(null, "You are an honor student");
    next statement;
}
The Nested-if Statement

The then and else block of an if statement can contain any valid statements, including other if statements. An if statement containing another if statement is called a nested-if statement.

```java
if (testScore >= 70) {
    if (studentAge < 10) {
        System.out.println("You did a great job ");
    } else {
        System.out.println("You passed"); //test score >= 70
    }
} else { //test score < 70
    System.out.println("You did not pass");
}
```
Control Flow

is testScore >= 70 ?

false

messageBox.show ("You did not pass");

testScore >= 70 ?

true

inner

is studentAge < 10 ?

false

messageBox.show ("You passed");

true

messageBox.show ("You did a great job");

messageBox.show ("You did not pass");
Writing a Proper if Control

```java
if (num1 < 0)
    if (num2 < 0)
        if (num3 < 0)
            negativeCount = 3;
        else
            negativeCount = 2;
    else
        if (num3 < 0)
            negativeCount = 2;
        else
            negativeCount = 1;
else
    if (num2 < 0)
        if (num3 < 0)
            negativeCount = 2;
        else
            negativeCount = 1;
else
    if (num3 < 0)
        negativeCount = 1;
else
    negativeCount = 0;

negativeCount = 0;
if (num1 < 0)
    negativeCount++;
if (num2 < 0)
    negativeCount++;
if (num3 < 0)
    negativeCount++;
```

The statement

```
    negativeCount++;
```

increments the variable by one.
if – else if Control

```java
if (score >= 90)
    System.out.print("Your grade is A");
else if (score >= 80)
    System.out.print("Your grade is B");
else if (score >= 70)
    System.out.print("Your grade is C");
else if (score >= 60)
    System.out.print("Your grade is D");
else
    System.out.print("Your grade is F");
```

<table>
<thead>
<tr>
<th>Test Score</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>90 ≤ score</td>
<td>A</td>
</tr>
<tr>
<td>80 ≤ score &lt; 90</td>
<td>B</td>
</tr>
<tr>
<td>70 ≤ score &lt; 80</td>
<td>C</td>
</tr>
<tr>
<td>60 ≤ score &lt; 70</td>
<td>D</td>
</tr>
<tr>
<td>score &lt; 60</td>
<td>F</td>
</tr>
</tbody>
</table>
Matching else

Are A and B different?

Both A and B mean...

```java
if (x < y) {
    if (x < z) {
        System.out.print("Hello");
    } else {
        System.out.print("Good bye");
    }
} else {
    System.out.print("Good bye");
}
```

Each else paired with nearest unmatched if -- use braces to change this as needed.
Boolean Operators

- Boolean expression can be combined using boolean operators.
- A boolean operator takes boolean values as its operands and returns a boolean value.
- The three boolean operators are:
  - and: `&&`
  - or: `||`
  - not: `!`
  - exclusive-OR: `^`

```java
if (temperature >= 65 && distanceToDestination < 2) {
    System.out.println("Let's walk");
} else {
    System.out.println("Let's drive");
}
```
Boolean Operators (contd)

- `bool1 && bool2` is `true` if both `bool1` and `bool2` are `true`; otherwise it is `false`
  - `(x > 2) && (x<10)` is `true` for `x=3`; `false` for `x=11`;

- `bool1 || bool2` is `true` if either `bool1` or `bool2` (or both) are `true`; otherwise it is `false`
  - `(x>2) || (x<10)` is always true.

- `!bool1` is `true` if `bool1` is `false`, and `false` if `bool1` is `true`
  - `!(x>2)` is `true` for `x=1`; and `false` for `x=3`;

- `bool1 ^ bool2` is `true` if `bool1` and `bool2` are different; otherwise it is `false`
  - `(x>2) ^ (x<10)` is false for `x=3`; and true for `x = 11;`
Semantics of Boolean Operators

Truth table for boolean operators

| p   | q    | p && q | p || q | !p  | p^q |
|-----|------|--------|--------|-----|-----|
| false | false  | false | false  | true| false |
| false | true   | false | true   | true| true |
| true  | false  | false | true   | false| true |
| true  | true   | true  | true   | false| false |

Sometimes true and false are represented by 1 and 0 (NOT in Java).

In C and C++, 0 is false, everything else is true.
Short-Circuit Evaluation

- Consider the following boolean expression:
  
  \[ x > y \; \text{||} \; x > z \]

- The expression is evaluated left to right. If \( x > y \) is true, then there’s no need to evaluate \( x > z \) because the whole expression will be true whether \( x > z \) is true or not.

- To stop the evaluation once the result of the whole expression is known is called short-circuit evaluation.

- What would happen if the short-circuit evaluation is not done for the following expression?

  \[ z == 0 \; \text{||} \; x / z > 20 \]
Short-circuit evaluation

- Sometimes this is useful
  - it is more efficient
  - \( z == 0 \) \(||\) \( x / z > 20 \)

- Can force complete evaluation by using:
  - \& instead of \&
  - | instead of ||

- Short-circuit evaluation is also called lazy evaluation (as opposed to eager evaluation)

- NOTE: \&, \| also denote bitwise and and or!
## Operator Precedence Rules

<table>
<thead>
<tr>
<th>Group</th>
<th>Operator</th>
<th>Precedence</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subexpression</td>
<td>( )</td>
<td>10 (Innermost first)</td>
<td>Left to Right</td>
</tr>
<tr>
<td>Postfix increment and decrement operators</td>
<td>++</td>
<td>9</td>
<td>Right to Left</td>
</tr>
<tr>
<td></td>
<td>--</td>
<td>9</td>
<td>Right to Left</td>
</tr>
<tr>
<td>Unary operators</td>
<td>++</td>
<td>8</td>
<td>Right to Left</td>
</tr>
<tr>
<td>Prefix inc, decre</td>
<td>--</td>
<td>8</td>
<td>Right to Left</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>8</td>
<td>Right to Left</td>
</tr>
<tr>
<td></td>
<td>!</td>
<td>8</td>
<td>Right to Left</td>
</tr>
<tr>
<td>Multiplicative operators</td>
<td>*</td>
<td>7</td>
<td>Left to Right</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>7</td>
<td>Left to Right</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>7</td>
<td>Left to Right</td>
</tr>
<tr>
<td>Additive operators</td>
<td>+</td>
<td>6</td>
<td>Left to Right</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>6</td>
<td>Left to Right</td>
</tr>
<tr>
<td>Relational operators</td>
<td>&lt;</td>
<td>5</td>
<td>Left to Right</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>5</td>
<td>Left to Right</td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
<td>5</td>
<td>Left to Right</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>5</td>
<td>Left to Right</td>
</tr>
<tr>
<td>Equality operators</td>
<td>==</td>
<td>4</td>
<td>Left to Right</td>
</tr>
<tr>
<td></td>
<td>!=</td>
<td>4</td>
<td>Left to Right</td>
</tr>
<tr>
<td>Boolean AND</td>
<td>&amp;&amp;</td>
<td>3</td>
<td>Left to Right</td>
</tr>
<tr>
<td>Boolean OR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assignment</td>
<td>=</td>
<td>1</td>
<td>Right to Left</td>
</tr>
</tbody>
</table>
Precedence Examples

```java
int x = 1, y=10, z=100;
boolean bool, test=false;

- x = -y + y * z;  
- x ==1 && y > 5  
- 4 < x && !test  
- bool = x!=y && y == z  
- x==y || y>4 && z<2
```

```java
x = (-y) + (y*z);
(x ==1) && (y > 5)
(4<x) && (!test)
bool = (x!=y) && (y ==z)
(x==y) || ((y>4) && (z<2))
```
Prefix operators

- The increment (++) and decrement (--) operators can precede the operand
  - `x++; ++x; y--; --y;`

- Their effect on the operand is the same, however, they vary only in terms of the timing of the increment or decrement.

- The postfix operators are applied AFTER the variable's value is used.

- The prefix operator are applied BEFORE
Example

```java
int x=2, y=10;
int x=2, y=10;
int x=2, y=10;
int x=2, y=10;
int x=2, y=10, z;
```
int x = 1, y = 10, z = 100;

boolean bool, test = false;

- x = y++;  
  x: 10   y: 11
- x = ++y;  
  x: 11   y: 11
- x = -++y;  
  x: -11  y: 11
- x = -y++;  
  x: -10  y: 11
- x = -y--;  
  x: -10  y: 9
- x = -(--y);  
  x: -9   y: 9
- x = ++y++;  ERROR!
Prefix vs. postfix.

- A prefix (postfix) operator is equivalent to executing the operator before (after) using the value of the variable:
  
  \[ z = x++ \ast --y; \]
  
  Is equivalent to:
  
  \[
  y = y-1; \\
  z = x \ast y; \\
  x = x + 1; 
  \]

What about:

\[ z = x++ \ast x++; \]
More Examples

\[ z = x++ \times x++; \]

\[ \text{Is equivalent to:} \]
\[ z = x \times (x+1); \]
\[ x = x+2; \]

\[ x = x++ \times --y; \]

\[ \text{Is equivalent to:} \]
\[ y = y - 1; \]
\[ x = x \times (y-1) + 1; \]
int x = 1, y = 10, z = 100;
boolean bool, test = false;

- x = y = z; 
  x: 100 y: 100 z: 100

- x = y = ++z; 
  x: 101 y: 101 z: 101

- bool = (x=11)>y 
  x: 11 y: 10 bool: true

- bool = (x=11)>y++ 
  x: 11 y: 11 bool: true

- bool = (x=11)> ++y 
  x: 11 y: 11 bool: false

- (x=3) > y && (z=5)<10 
  x: 3 y: 10 z: 100

- (x=3) > y & (z=5)<10 
  x: 3 y: 10 z: 5
Boolean Variables

- The result of a boolean expression is either **true** or **false**. These are the two values of data type **boolean**.
- We can declare a variable of data type **boolean** and assign a boolean value to it.

```java
boolean pass, done;
pass = 70 < x;
done = true;
if (pass) {
    ...
} else {
    ...
}
```
**Boolean Methods**

A method that returns a boolean value, such as:

```java
private boolean isValid(int value) {
    if (value < MAX_ALLOWED)
        return true;
    } else {
        return false;
    
    
```
Comparing Objects

There is only one way to compare primitive data types, but with objects (reference data types) there are two:

1. We can test whether two variables point to the same object (use ==), or
2. We can test whether two distinct objects have the same contents.
Using == With Objects (Sample 1)

String str1 = new String("Java");
String str2 = new String("Java");

if (str1 == str2) {
    System.out.println("They are equal");
} else {
    System.out.println("They are not equal");
}

Not equal because str1 and str2 point to different String objects.

They are not equal
String str1 = new String("Java");
String str2 = str1;

if (str1 == str2) {
    System.out.println("They are equal");
} else {
    System.out.println("They are not equal");
}

They are equal here because str1 and str2 point to the same object.
Using equals with String

String str1 = new String("Java");
String str2 = new String("Java");

if (str1.equals(str2)) {
    System.out.println("They are equal");
} else {
    System.out.println("They are not equal");
}

It's equal here because str1 and str2 have the same sequence of characters.
The Semantics of ==

Case 1: different objects

```java
String str1, str2;
str1 = new String("Java");
str2 = new String("Java");
```

```
str1==str2 ?
```

false

Case 2: same object

```java
String str1, str2;
str1 = new String("Java");
str2 = str1;
```

```
str1==str2 ?
```

true
In creating String objects

String word1, word2;
word1 = new String("Java");
word2 = new String("Java");

word1==word2 ? false

Whenever the `new` operator is used, there will be a new object.

String word1, word2;
word1 = "Java";
word2 = "Java";

word1==word2 ? true

Literal String objects such as "Java" will always refer to the same object.
Comparing Strings

- If we want to compare the content of string objects, we can use `equals`

```java
String word1, word2;
if(word1.equals(word2)){
    System.out.print("They are equal");
} else {
    System.out.print("They are not equal");
}
```

- There is also `equalsIgnoreCase` and `compareTo`

- `equalsIgnoreCase` treats upper and lower case letters as the same (e.g. ‘H’ and ‘h’)

compareTo method

- This method compares two strings in terms of their lexicographic order.
  \[ \text{str1}.\text{compareTo}(\text{str2}) \]

- It returns:
  - 0 if the strings are exactly the same;
  - a negative value if \text{str1} comes before \text{str2};
  - a positive value if \text{str1} comes after \text{str2};

- Lexicographic ordering is determined by UNICODE values.
  - ...,!,...,+-,... 0,1,2,...,9,...A,B,...,Z,...,a,b,...,z, ...
Comparing Objects

- The operators <, >=, … cannot be applied to compare objects.
- In order to compare objects, we need to implement an appropriate method.
- For example, the `equals`, `compareTo` methods for strings.
- A default `equals` method exists for each class, but it may not behave as you expect.
if and switch

- The if statement is essential for writing interesting programs.
- Other control flow statements (e.g., switch and loops) can be implemented using if statements.
- They are available since we often need them. Programs are more readable too.
- Next: switch
The **switch** Statement

```java
int recSection;
recSection = Integer.parseInt(JOptionPane.showInputDialog("Recitation Section (1,2,...,4):" ));

switch (recSection) {
    case 1: System.out.print("Go to UNIV 101"); break;
    case 2: System.out.print("Go to UNIV 119"); break;
    case 3: System.out.print("Go to STON 217"); break;
    case 4: System.out.print("Go to UNIV 101"); break;
}
```

This statement is executed if the `gradeLevel` is equal to 1.

This statement is executed if the `gradeLevel` is equal to 4.
Syntax for the **switch** Statement

```
switch ( <integer expression> ) {
    case <label 1> : <case body 1>
        <break;>
    ...
    case <label n> : <case body n>
        <break;>
    default : <default body>
}
```

The **break** statement is *optional* within each case. A case body is also *optional*. The **default** is *optional* for the switch statement.
The `switch` Statement

```java
int recSection;
recSection = Integer.parseInt(JOptionPane.showInputDialog("Recitation Section (1,2,...,4):" ));

switch (recSection) {
    case 1:
        System.out.print("Go to UNIV 101");
        break;
    case 2:
        System.out.print("Go to UNIV 119");
        break;
    case 3:
        System.out.print("Go to STON 217");
        break;
    case 4:
        System.out.print("Go to UNIV 101");
        break;
}
```
Switch statement (cont.)

- The integer expression can have only one of the following types:
  - char, byte, short, or int
- The label must be a literal or named constant of the same type.
- Each case body may end with a break statement.
- A break causes the execution to go to the statement following the switch statement.
- The default case applies when no label matches.
- Each label must be unique.
- Labels may be listed in any order.
Simple **switch** statement

```java
switch ( N ) {
    case 1: x = 10;
    case 2: x = 20;
    case 3: x = 30;
}
```
switch ( N ) {
    case 1: x = 10;
        break;
    case 2: x = 20;
    case 3: x = 30;
        break;
    default: x = 0;
}
switch ( N ) {
    case 1: x = 10;
            break;
    case 2:
    case 3: x = 30;
            break;
    default: x = 0;
}
Characters

- In Java, single characters are represented using the data type `char`.
- Character constants are written as symbols enclosed in single quotes.
- Characters are stored in memory using some form of encoding.
- ASCII, which stands for American Standard Code for Information Interchange, is one of the document coding schemes widely used today.
- Java uses Unicode, which includes ASCII, for representing `char` constants.
## ASCII Encoding

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>nul</td>
<td>soh</td>
<td>stx</td>
<td>etx</td>
<td>eot</td>
<td>enq</td>
<td>ack</td>
<td>bel</td>
<td>bs</td>
<td>ht</td>
</tr>
<tr>
<td>10</td>
<td>lf</td>
<td>vt</td>
<td>ff</td>
<td>cr</td>
<td>so</td>
<td>si</td>
<td>dle</td>
<td>dc1</td>
<td>dc2</td>
<td>dc3</td>
</tr>
<tr>
<td>20</td>
<td>cd4</td>
<td>nak</td>
<td>syn</td>
<td>etb</td>
<td>can</td>
<td>em</td>
<td>sub</td>
<td>esc</td>
<td>fs</td>
<td>gs</td>
</tr>
<tr>
<td>30</td>
<td>rs</td>
<td>us</td>
<td>sp</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
<td>'</td>
</tr>
<tr>
<td>40</td>
<td>(</td>
<td>)</td>
<td>*</td>
<td>+</td>
<td>,</td>
<td>-</td>
<td>.</td>
<td>/</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>:</td>
<td>;</td>
</tr>
<tr>
<td>60</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>?</td>
<td>@</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>70</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td>80</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>90</td>
<td>Z</td>
<td>[</td>
<td>\</td>
<td>]</td>
<td>^</td>
<td>_</td>
<td>`</td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>100</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
</tr>
<tr>
<td>110</td>
<td>n</td>
<td>o</td>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
<td>w</td>
</tr>
<tr>
<td>120</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>{</td>
<td>}</td>
<td></td>
<td>~</td>
<td>del</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Unicode Worldwide Character Standard (Unicode) supports the interchange, processing, and display of the written texts of diverse languages.

A UNICODE character takes up two bytes. ASCII characters take up one byte.

```
char ch1 = 'X';
System.out.println(ch1);  // X
System.out.println( (int) ch1);  // 88
```
Character Processing

definition

```java
char ch1, ch2 = 'X';

System.out.print("ASCII code of character X is " + (int)'X');
System.out.print("Character with ASCII code 88 is " + (char)88);

'A' < 'c'

if (ch1 < 'A' && ch2 == 99)
    System.out.print("Done");
```

- Declaration and initialization
- Type conversion between int and char.
- This comparison returns true because ASCII value of 'A' is 65 while that of 'c' is 99.
- Can compare characters with numbers directly.
Chapter 5 introduces four standard classes related to drawing geometric shapes. They are

- `java.awt.Graphics`
- `java.awt.Color`
- `java.awt.Point`
- `java.awt.Dimension`

These classes are used in the Sample Development section.

Please refer to Java API for details.
import javax.swing.*; //for JFrame
import java.awt.*; //for Graphics and Container

class Ch5SampleGraphics {
    public static void main( String[ ] args ) {
        JFrame win;
        Container contentPane;
        Graphics g;

        win = new JFrame("My First Rectangle");
        win.setSize(300, 200);
        win.setLocation(100,100);
        win.setVisible(true);

        contentPane = win.getContentPane();
        g = contentPane.getGraphics();
        g.drawRect(50,50,100,30);
    }
}
The Effect of `drawRect`

```java
graphic.drawRect(50, 50, 100, 30);
```

```java
graphic.drawRect(x, y, width, height);
```

Position (0, 0)
Problem Statement

Write an application that simulates a screensaver by drawing various geometric shapes in different colors. The user has an option of choosing a type (ellipse or rectangle), color, and movement (stationary, smooth, or random).
Overall Plan

- Tasks:
  - Get the shape the user wants to draw.
  - Get the color of the chosen shape.
  - Get the type of movement the user wants to use.
  - Start the drawing.
Required Classes

- Ch5DrawShape
- DrawingBoard
- JOptionPane
- DrawableShape

- class we implement
- helper class given to us
Development Steps

We will develop this program in six steps:

1. Start with a program skeleton. Explore the DrawingBoard class.
2. Define an experimental DrawableShape class that draws a dummy shape.
3. Add code to allow the user to select a shape. Extend the DrawableShape and other classes as necessary.
4. Add code to allow the user to specify the color. Extend the DrawableShape and other classes as necessary.
5. Add code to allow the user to specify the motion type. Extend the DrawableShape and other classes as necessary.
6. Finalize the code by tying up loose ends.
Step 1 Design

The methods of the DrawingBoard class

- public void addShape(DrawableShape shape)
  Adds a shape to the DrawingBoard. No limit to the number shapes you can add

- public void setBackground(java.awt.Color color)
  Sets the background color of a window to the designated color

- public void setDelayTime(double delay)
  Sets the delay time between drawings to delay seconds

- public void setMovement(int type)
  Sets the movement type to STATIONARY, RANDOM, or SMOOTH

- public void setVisible(boolean state)
  Sets the background color of a window to the designated color

- public void start()
  Starts the drawing of added shapes using the designated movement type and delay time.
Step 1 Code

Program source file is too big to list here. From now on, we ask you to view the source files using your Java IDE.

**Directory:** Chapter5/Step1

**Source Files:** Ch5DrawShape.java
Step 1 Test

- In the testing phase, we run the program and verify that a DrawingBoard window with black background appears on the screen and fills the whole screen.
Step 2 Design

- Define a preliminary DrawableShape class
- The required methods of this class are

  - `public void draw(java.awt.Graphics g)`
    Draws a shape on Graphics object g.
  - `public java.awt.Point getCenterPoint()`
    Returns the center point of this shape
  - `public java.awt.Dimension getDimension()`
    Returns the bounding rectangle of this shape
  - `public void setCenterPoint(java.awt.Point pt)`
    Sets the center point of this shape to pt.
Step 2 Code

Directory: Chapter5/Step2

Source Files: Ch5DrawShape.java
DrawableShape.java
Step 2 Test

- We compile and run the program numerous times
- We confirm the movement types STATIONARY, RANDOM, and SMOOTH.
- We experiment with different delay times
- We try out different background colors
We extend the main class to allow the user to select a shape information.

We will give three choices of shapes to the user: Ellipse, Rectangle, and Rounded Rectangle

We also need input routines for the user to enter the dimension and center point. The center point determines where the shape will appear on the DrawingBoard.

Three input methods are

private int inputShapeType( )
private Dimension inputDimension( )
private Point inputCenterPoint( )
Step 3 Code

Directory: Chapter5/Step3

Source Files: Ch5DrawShape.java
DrawableShape.java
Step 3 Test

- We run the program numerous times with different input values and check the results.
- Try both valid and invalid input values and confirm the response is appropriate
We extend the main class to allow the user to select a color.
We follow the input pattern of Step 3.
We will allow the user to select one of the five colors.
The color input method is
```java
private Color inputColor()
```
Step 4 Code

Directory: Chapter5/Step4

Source Files: Ch5DrawShape.java
DrawableShape.java
Step 4 Test

- We run the program numerous times with different color input.
- Try both valid and invalid input values and confirm the response is appropriate
Step 5 Design

- We extend the main class to allow the user to select a movement type.
- We follow the input pattern of Step 3.
- We will allow the user to select one of the three movement types.
- The movement input method is
  \[
  \text{private int } \text{inputMotionType( )}
  \]
Step 5 Code

Directory: Chapter5/Step5

Source Files: Ch5DrawShape.java
DrawableShape.java
Step 5 Test

- We run the program numerous times with different movement input.
- Try both valid and invalid input values and confirm the response is appropriate.
Step 6: Finalize

- Possible Extensions
  - Morphing the object shape
  - Changing the object color
  - Drawing multiple objects
  - Drawing scrolling text