2b

Control Statements: Part 1

OBJECTIVES

In this lecture you will learn:

- To use basic problem-solving techniques.
- To develop algorithms through the process of top-down, stepwise refinement.
- To use the `if` and `if…else` selection statements to choose among alternative actions.
- To use the `while` repetition statement to execute statements in a program repeatedly.
- To use counter-controlled repetition and sentinel-controlled repetition.
- To use the assignment, increment and decrement operators.
- The primitive data types.

Outline

2b.2 Algorithms

Algorithms

- The actions to execute
- The order in which these actions execute

Program control

- Specifies the order in which actions execute in a program

2b.3 Pseudocode

Pseudocode

- An informal language similar to English
- Helps programmers develop algorithms
- Does not run on computers
- Should contain input, output and calculation actions
- Should not contain variable declarations

2b.4 Control Structures

Sequential execution

- Statements are normally executed one after the other in the order in which they are written

Transfer of control

- Specifying the next statement to execute that is not necessarily the next one in order
- Can be performed by the `goto` statement
  - Structured programming eliminated `goto` statements

2b.4 Control Structures (Cont.)

Bohm and Jacopini's research

- **Demonstrated that `goto` statements were unnecessary**
- **Demonstrated that all programs could be written with three control structures**
  - **The sequence structure,**
  - **The selection structure and**
  - **The repetition structure**

2b.4 Control Structures (Cont.)

UML activity diagram (www.uml.org)

- Models the workflow (or activity) of a part of a software system
- Action-state symbols (rectangles with their sides replaced with outward-curving arcs)
  - represent action expressions specifying actions to perform
- Diamonds
  - Decision symbols (explained in Section 2b.5)
  - Merge symbols (explained in Section 2b.7)

2b.4 Control Structures (Cont.)

- Small circles
  - Solid circle represents the activity's initial state
  - Solid circle surrounded by a hollow circle represents the activity's final state
- Transition arrows
  - Indicate the order in which actions are performed
- Notes (rectangles with the upper-right corners folded over)
  - Explain the purposes of symbols (like comments in Java)
  - Are connected to the symbols they describe by dotted lines

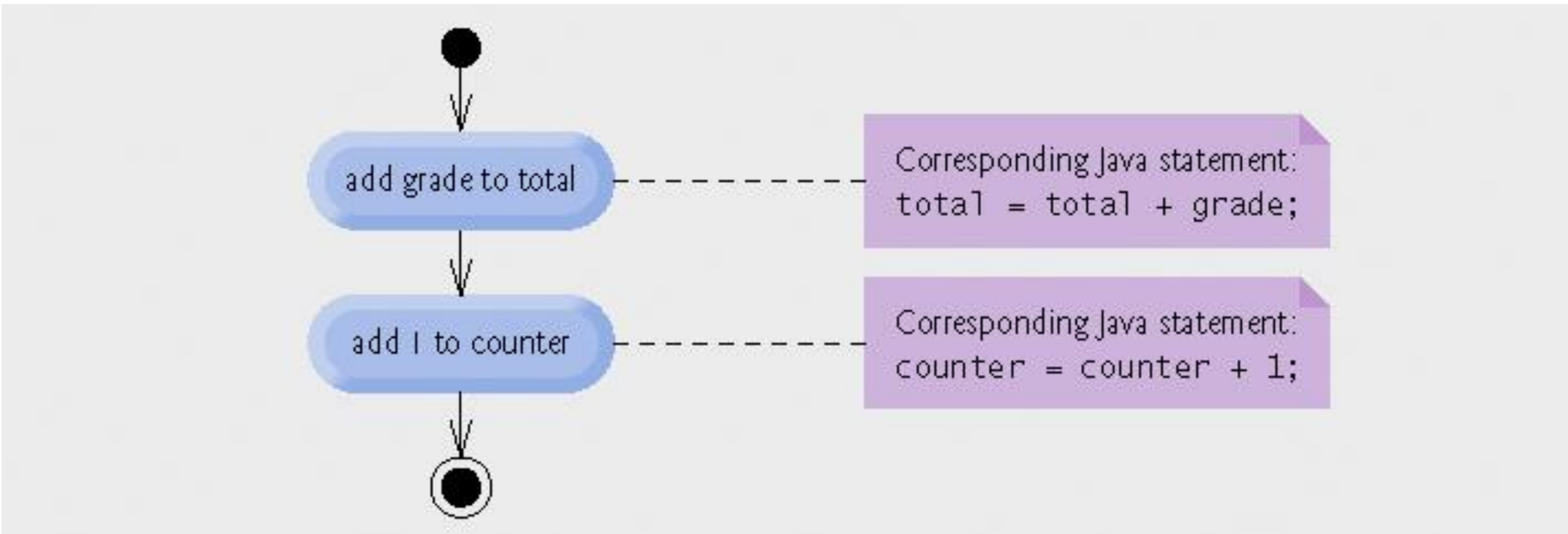


Fig. 2b.1 | Sequence structure activity diagram.

2b.4 Control Structures (Cont.)

Selection Statements

- `if` statement
  - Single-selection statement
- `if`…`else` statement
  - Double-selection statement
- `switch` statement
  - Multiple-selection statement

2b.4 Control Structures (Cont.)

Repetition statements

- Also known as looping statements
- Repeatedly performs an action while its loop-continuation condition remains true
- `while` statement
  - Performs the actions in its body zero or more times
- `do`…`while` statement
  - Performs the actions in its body one or more times
- `for` statement
  - Performs the actions in its body zero or more times

2b.4 Control Structures (Cont.)

Java has three kinds of control structures

- **Sequence statement,**
- **Selection statements (three types) and**
- **Repetition statements (three types)**
- **All programs are composed of these control statements**
  - **Control-statement stacking**
    - **All control statements are single-entry/single-exit**
  - **Control-statement nesting**

2b.5 `if` Single-Selection Statement

`if` statements

- **Execute an action if the specified condition is `true`**
- **Can be represented by a decision symbol (diamond) in a UML activity diagram**
  - **Transition arrows out of a decision symbol have guard conditions**
    - **Workflow follows the transition arrow whose guard condition is true**

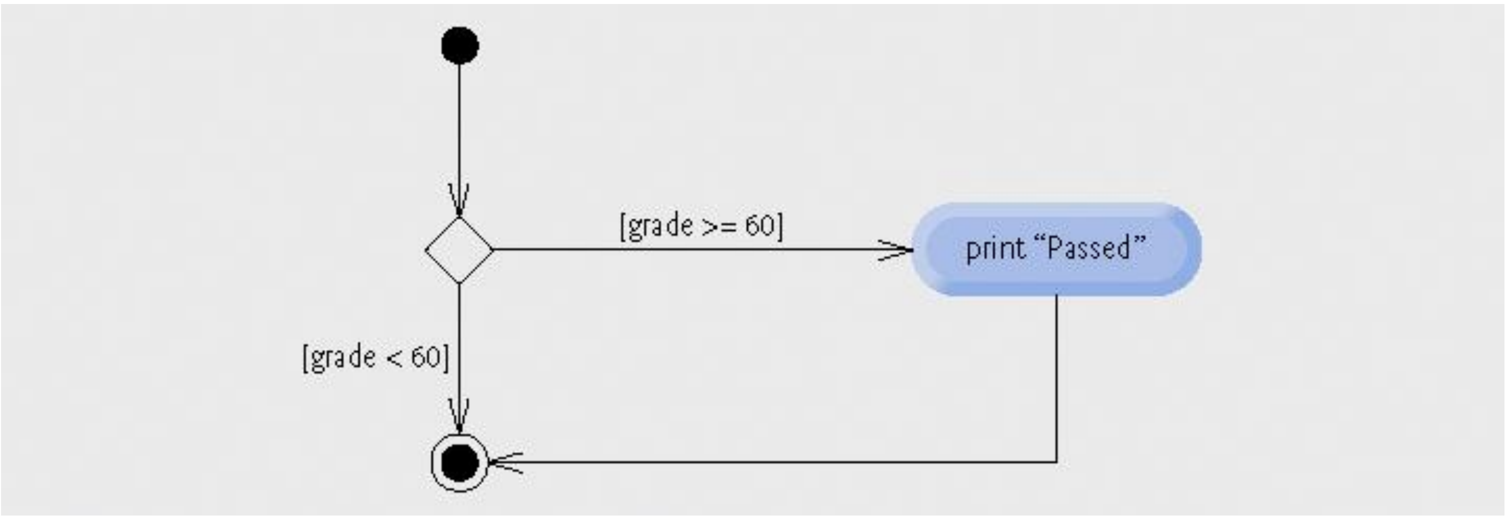


Fig. 2b.2 | `if` single-selection statement UML activity diagram.

2b.6 `if`…`else` Double-Selection Statement

- `if`…`else` **statement**
  - **Executes one action if the specified condition is `true` or a different action if the specified condition is `false`**
- **Conditional Operator (`? :`)**
  - **Java's only ternary operator (takes three operands)**
  - **`? :` and its three operands form a conditional expression**
    - **Entire conditional expression evaluates to the second operand if the first operand is `true`**
    - **Entire conditional expression evaluates to the third operand if the first operand is `false`**

Good Programming Practice 2b.1

Indent both body statements of an `if…else` statement.

Good Programming Practice 2b.2

If there are several levels of indentation, each level should be indented the same additional amount of space.

Good Programming Practice 2b.3

Conditional expressions are more difficult to read than `if…else` statements and should be used to replace only simple `if…else` statements that choose between two values.

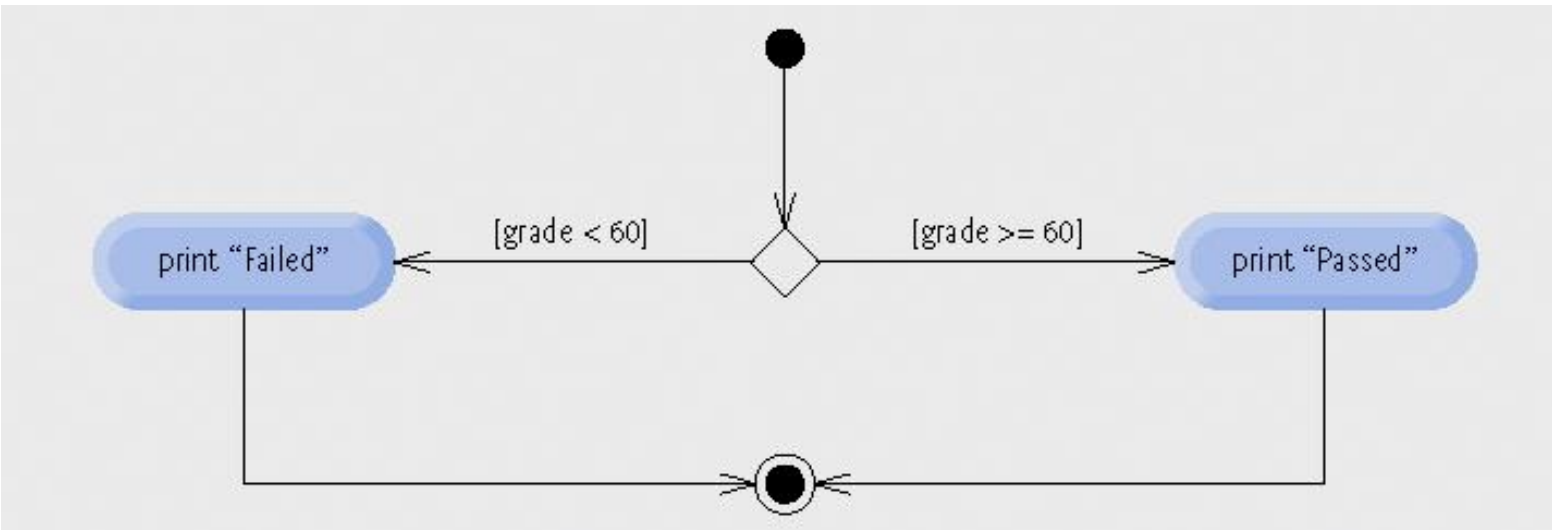


Fig. 2b.3 | `if...else` double-selection statement UML activity diagram.

2b.6 `if…else` Double-Selection Statement (Cont.)

- **Nested `if…else` statements**
  - `if…else` statements can be put inside other `if…else` statements
- **Dangling-`else` problem**
  - `else`s are always associated with the immediately preceding `if` unless otherwise specified by braces `{ }`
- **Blocks**
  - Braces `{ }` associate statements into blocks
  - Blocks can replace individual statements as an `if` body

2b.6 `if`…`else` Double-Selection Statement (Cont.)

- **Logic errors**
  - **Fatal logic errors cause a program to fail and terminate prematurely**
  - **Nonfatal logic errors cause a program to produce incorrect results**
- **Empty statements**
  - **Represented by placing a semicolon (`;`) where a statement would normally be**
  - **Can be used as an `if` body**

Common Programming Error 2b.1

Forgetting one or both of the braces that delimit a block can lead to syntax errors or logic errors in a program.

Good Programming Practice 2b.4

Always using braces in an `if...else` (or other) statement helps prevent their accidental omission, especially when adding statements to the `if`-part or the `else`-part at a later time. To avoid omitting one or both of the braces, some programmers type the beginning and ending braces of blocks before typing the individual statements within the braces.

Common Programming Error 2b.2

Placing a semicolon after the condition in an `if` or `if...else` statement leads to a logic error in single-selection `if` statements and a syntax error in double-selection `if...else` statements (when the `if`-part contains an actual body statement).

2b.7 `while` Repetition Statement

`while` statement

- Repeats an action while its loop-continuation condition remains true
- Uses a merge symbol in its UML activity diagram
  - Merges two or more workflows
  - Represented by a diamond (like decision symbols) but has:
    - Multiple incoming transition arrows,
    - Only one outgoing transition arrow and
    - No guard conditions on any transition arrows

Common Programming Error 2b.3

Not providing, in the body of a `while` statement, an action that eventually causes the condition in the `while` to become false normally results in a logic error called an *infinite loop*, in which the loop never terminates.

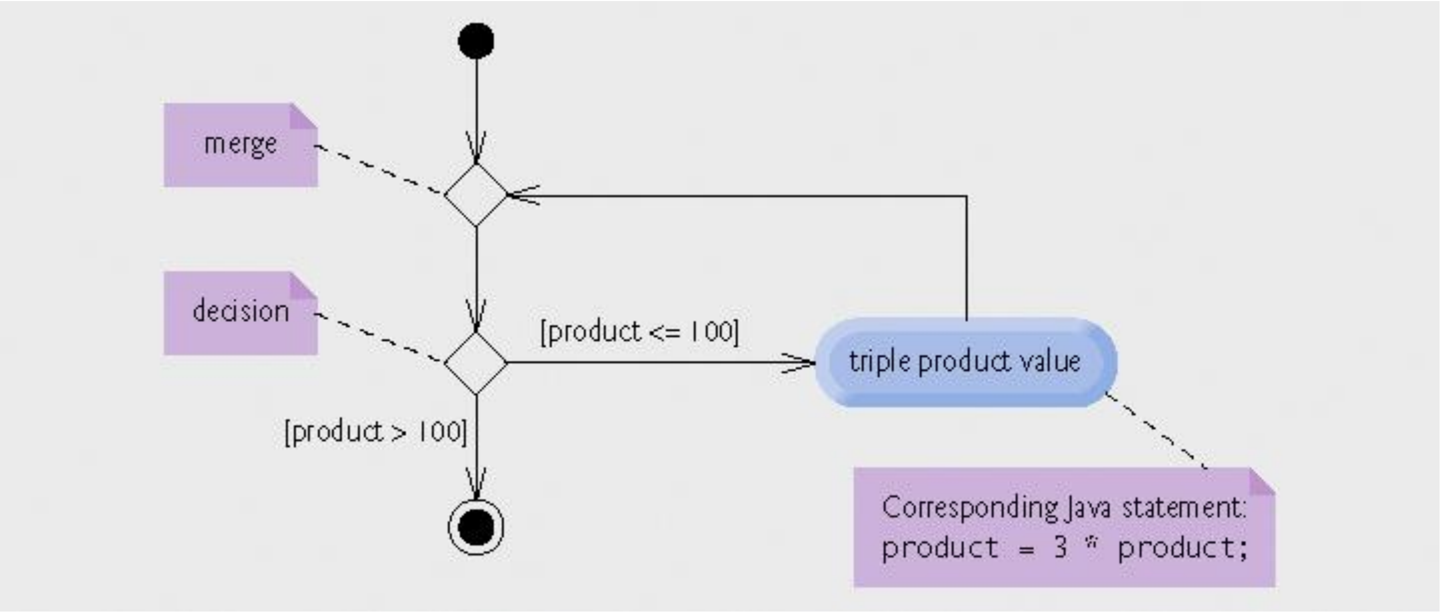


Fig. 2b.4 | `while` repetition statement UML activity diagram.

2b.8 Formulating Algorithms: Counter-Controlled Repetition

Counter-controlled repetition

– Use a counter variable to count the number of times a loop is iterated

Integer division

– The fractional part of an integer division calculation is truncated (thrown away)

```
Set total to zero
Set grade counter to one

While grade counter is less than or equal to ten
    Prompt the user to enter the next grade
    Input the next grade
    Add the grade into the total
    Add one to the grade counter

Set the class average to the total divided by ten
Print the class average
```

Fig. 2b.5 | Pseudocode algorithm that uses counter-controlled repetition to solve the class-average problem.

```java
// Fig. 4.6: GradeBook.java
// GradeBook class that solves class-average problem using
// counter-controlled repetition.
import java.util.Scanner; // program uses class Scanner

public class GradeBook
{
   private String courseName; // name of course this GradeBook represents

   // constructor initializes courseName
   public GradeBook( String courseName)
   {
      setCourseName(courseName); // initializes courseName
   } // end constructor

   // method to set the course name
   public void setCourseName( String name )
   {
      courseName = (name != null)? name: “No name”; // store the course name
   } // end method setCourseName

   // method to retrieve the course name
   public String getCourseName()
   {
      return courseName;
   } // end method getCourseName

```


Assign a value to instance variable **`courseName`**

Declare method **`setCourseName`**

Declare method **`getCourseName`**

```java
// display a welcome message to the GradeBook user
public void displayMessage()
{
   // getCourseName gets the name of the course
   System.out.printf( "Welcome to the grade book for%n%s!%n%n",
      courseName );
} // end method displayMessage

// determine class average based on 10 grades entered by user
public void determineClassAverage()
{
   // create Scanner to obtain input from command window
   Scanner input = new Scanner( System.in );

   int total; // sum of grades entered by user
   int gradeCounter; // number of the grade to be entered next
   int grade; // grade value entered by user
   int average; // average of grades

   // initialization phase
   total = 0; // initialize total
   gradeCounter = 1; // initialize loop counter

```

Declare method **displayMessage**

Declare method **determineClassAverage**

Declare and initialize **Scanner** variable **input**

Declare local **int** variables **total**, **gradeCounter**, **grade** and **average**

•**GradeBook.java**
(2 of 3)

- **GradeBook.java**

(3 of 3)

```java
      // processing phase
      while ( gradeCounter <= 10 ) // loop 10 times
      {
         System.out.print( "Enter grade: " ); // prompt
         grade = input.nextInt(); // input next grade
         total = total + grade; // add grade to total
         gradeCounter = gradeCounter + 1; // increment counter by 1
      } // end while

      // termination phase
      average = total / 10; // integer division yields integer result

      // display total and average of grades
      System.out.printf( "\nTotal of all 10 grades is %d%n", total );
      System.out.printf( "Class average is %d%n", average );
   } // end method determineClassAverage

} // end class GradeBook
```

while loop iterates as long as **gradeCounter <= 10**

Increment the counter variable **gradeCounter**

Calculate average grade

Display results

Good Programming Practice 2b.5

Separate declarations from other statements in methods with a blank line for readability.

Software Engineering Observation 2b.1

Experience has shown that the most difficult part of solving a problem on a computer is developing the algorithm for the solution. Once a correct algorithm has been specified, the process of producing a working Java program from the algorithm is normally straightforward.

Common Programming Error 2b.4

Using the value of a local variable before it is initialized results in a compilation error. All local variables must be initialized before their values are used in expressions.

Error-Prevention Tip 2b.1

Initialize each counter and total, either in its declaration or in an assignment statement. Totals are normally initialized to 0. Counters are normally initialized to 0 or 1, depending on how they are used (we will show examples of when to use 0 and when to use 1).

•GradeBook Test.java

```java
// Fig. 4.7: GradeBookTest.java
// Create GradeBook object and invoke its determineClassAverage method.

public class GradeBookTest
{
   public static void main( String args[] )
   {
      // create GradeBook object myGradeBook and
      // pass course name to constructor
      GradeBook myGradeBook = new GradeBook(
         "CS101 Introduction to Java Programming" );

      myGradeBook.displayMessage(); // display welcome message
      myGradeBook.determineClassAverage(); // find average of 10 grades
   } // end main

} // end class GradeBookTest
```

Create a new **GradeBook** object

Pass the course's name to the **GradeBook** constructor as a **string**

Call **GradeBook**'s **determineClassAverage** method

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

Enter grade: 67
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100

Total of all 10 grades is 846
Class average is 84
```

Common Programming Error 2b.5

Assuming that integer division rounds (rather than truncates) can lead to incorrect results. For example, 7 ÷ 4, which yields 1.75 in conventional arithmetic, truncates to 1 in integer arithmetic, rather than rounding to 2.

2b.9 Formulating Algorithms: Sentinel-Controlled Repetition

Sentinel-controlled repetition

- **Also known as indefinite repetition**
- **Use a sentinel value (also known as a signal, dummy or flag value)**
  - **A sentinel value cannot also be a valid input value**

2b.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

Top-down, stepwise refinement

- **Top: a single statement that conveys the overall function of the program**
- **First refinement: multiple statements using only the sequence structure**
- **Second refinement: commit to specific variables, use specific control structures**

Common Programming Error 2b.6

Choosing a sentinel value that is also a legitimate data value is a logic error.

Software Engineering Observation 2b.2

Each refinement, as well as the top itself, is a complete specification of the algorithm—only the level of detail varies.

Software Engineering Observation 2b.3

Many programs can be divided logically into three phases: an initialization phase that initializes the variables; a processing phase that inputs data values and adjusts program variables (e.g., counters and totals) accordingly; and a termination phase that calculates and outputs the final results.

Error-Prevention Tip 2b.2

When performing division by an expression whose value could be zero, explicitly test for this possibility and handle it appropriately in your program (e.g., by printing an error message) rather than allow the error to occur

```
Initialize total to zero
Initialize counter to zero

Prompt the user to enter the first grade
Input the first grade (possibly the sentinel)

While the user has not yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Prompt the user to enter the next grade
    Input the next grade (possibly the sentinel)

If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the average
else
    Print "No grades were entered"
```

Fig. 2b.8 | Class-average problem pseudocode algorithm with sentinel-controlled repetition.

Software Engineering Observation 2b.4

Terminate the top-down, stepwise refinement process when you have specified the pseudocode algorithm in sufficient detail for you to convert the pseudocode to Java. Normally, implementing the Java program is then straightforward.

Software Engineering Observation 2b.5

Some experienced programmers write programs without ever using program-development tools like pseudocode. They feel that their ultimate goal is to solve the problem on a computer and that writing pseudocode merely delays the production of final outputs. Although this method may work for simple and familiar problems, it can lead to serious errors and delays in large, complex projects.

GradeBook.java
(1 of 3)

```java
// Fig. 4.9: GradeBook.java
// GradeBook class that solves class-average program using
// sentinel-controlled repetition.
import java.util.Scanner; // program uses class Scanner

public class GradeBook
{
   private String courseName; // name of course this GradeBook represents

   // constructor initializes courseName
   public GradeBook( String courseName)
   {
      setCourseName(courseName) ; // initializes courseName
   } // end constructor

   // method to set the course name
   public void setCourseName( String name )
   {
      courseName = (name != null)? name: “no name”; // s
   } // end method setCourseName

   // method to retrieve the course name
   public String getCourseName()
   {
      return courseName;
   } // end method getCourseName

```

Assign a value to instance variable **courseName**

Declare method **setCourseName**

Declare method **getCourseName**

```java
   // display a welcome message to the GradeBook user
   public void displayMessage()
   {
      // getCourseName gets the name of the course
      System.out.printf( "Welcome to the grade book for%n%s!%n%n",
         courseName );
   } // end method displayMessage

   // determine the average of an arbitrary number of grades
   public void determineClassAverage()
   {
      // create Scanner to obtain input from command window
      Scanner input = new Scanner( System.in );

      int total; // sum of grades
      int gradeCounter; // number of grades entered
      int grade; // grade value
      double average; // number with decimal point for average

      // initialization phase
      total = 0; // initialize total
      gradeCounter = 0; // initialize loop counter

      // processing phase
      // prompt for input and read grade from user
      System.out.print( "Enter grade or -1 to quit: " );
      grade = input.nextInt();

```

GradeBook.java

(2 of 3)

Declare method **displayMessage**

Declare method **determineClassAverage**

Declare and initialize **Scanner** variable **input**

Declare local **int** variables **total**, **gradeCounter** and **grade** and **double** variable **average**

```java
      // loop until sentinel value read from user
      while ( grade != -1 )
      {
         total = total + grade; // add grade to total
         gradeCounter = gradeCounter + 1; // increment counter

         // prompt for input and read next grade from user
         System.out.print( "Enter grade or -1 to quit: " );
         grade = input.nextInt();
      } // end while

      // termination phase
      // if user entered at least one grade...
      if ( gradeCounter != 0 )
      {
         // calculate average of all grades entered
         average = (double) total / gradeCounter;

         // display total and average (with two digits of precision)
         System.out.printf( "\nTotal of the %d grades entered is %d %n",
            gradeCounter, total );
         System.out.printf( "Class average is %.2f %n ", average );
      } // end if
      else // no grades were entered, so output appropriate message
         System.out.println( "No grades were entered" );
   } // end method determineClassAverage

} // end class GradeBook
```

while loop iterates as long as **grade !=** the sentinel value, **-1**

GradeBook.java

(3 of 3)

Calculate average grade using **(double)** to perform explicit conversion

Display average grade

Display **"No grades were entered"** message

Good Programming Practice 2b.6

In a sentinel-controlled loop, the prompts requesting data entry should explicitly remind the user of the sentinel value.

Common Programming Error 2b.7

Omitting the braces that delimit a block can lead to logic errors, such as infinite loops. To prevent this problem, some programmers enclose the body of every control statement in braces even if the body contains only a single statement.

2b.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

Unary cast operator

- **Creates a temporary copy of its operand with a different data type**
  - **example: `(double)` will create a temporary floating-point copy of its operand**
- **Explicit conversion**

Promotion

- **Converting a value (e.g. `int`) to another data type (e.g. `double`) to perform a calculation**
- **Implicit conversion**

Common Programming Error 2b.8

The cast operator can be used to convert between primitive numeric types, such as `int` and `double`, and between related reference types (as we discuss in Chapter 10, Object-Oriented Programming: Polymorphism). Casting to the wrong type may cause compilation errors or runtime errors.

•GradeBook Test.java

```java
// Fig. 4.10: GradeBookTest.java
// Create GradeBook object and invoke its determineClassAverage method.

public class GradeBookTest
{
   public static void main( String args[] )
   {
      // create GradeBook object myGradeBook and
      // pass course name to constructor
      GradeBook myGradeBook = new GradeBook(
         "CS101 Introduction to Java Programming" );

      myGradeBook.displayMessage(); // display welcome message
      myGradeBook.determineClassAverage(); // find average of grades
   } // end main

} // end class GradeBookTest
```

Create a new **`GradeBook`** object

Pass the course's name to the **`GradeBook`** constructor as a **`string`**

Call **`GradeBook`**'s **`determineClassAverage`** method

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of the 3 grades entered is 257
Class average is 85.67
```

2b.10 Formulating Algorithms: Nested Control Statements

Control statements can be nested within one another

- **Place one control statement inside the body of the other**

```
Initialize passes to zero
Initialize failures to zero
Initialize student counter to one

While student counter is less than or equal to 10
    Prompt the user to enter the next exam result
    Input the next exam result

    If the student passed
        Add one to passes
    Else
        Add one to failures

    Add one to student counter

Print the number of passes
Print the number of failures

If more than eight students passed
    Print "Raise tuition"
```

Fig. 2b.11 | Pseudocode for examination-results problem.

```java
// Fig. 4.12: Analysis.java
// Analysis of examination results.
import java.util.Scanner; // class uses class Scanner

public class Analysis
{
   public void processExamResults
   {
      // create Scanner to obtain input from command window
      Scanner input = new Scanner( System.in );

      // initializing variables in declarations
      int passes = 0; // number of passes
      int failures = 0; // number of failures
      int studentCounter = 1; // student counter
      int result; // one exam result (obtains value from user)

      // process 10 students using counter-controlled loop
      while ( studentCounter <= 10 )
      {
         // prompt user for input and obtain value from user
         System.out.print( "Enter result (1 = pass, 2 = fail): " );
         result = input.nextInt();

```

Declare `processExamResults`' local variables

`while` loop iterates as long as `studentCounter <= 10`

- **Analysis.java**

(1 of 2)

```
         // if...else nested in while
         if ( result == 1 )             // if result 1,
            passes = passes + 1;        // increment passes;
         else                           // else result is not 1, so
            failures = failures + 1;    // increment failures

         // increment studentCounter so loop eventually terminates
         studentCounter = studentCounter + 1;
      } // end while

      // termination phase; prepare and display results
      System.out.printf( "Passed: %d%nFailed: %d%n", passes, failures );

      // determine whether more than 8 students passed
      if ( passes > 8 )
         System.out.println( "Raise Tuition" );
   } // end method processExamResults

} // end class Analysis
```

Determine whether this student passed or failed and increment the appropriate variable

Determine whether more than eight students passed the exam

•Analysis.java
(2 of 2)

Error-Prevention Tip 2b.3

Initializing local variables when they are declared helps the programmer avoid any compilation errors that might arise from attempts to use uninitialized data. While Java does not require that local variable initializations be incorporated into declarations, it does require that local variables be initialized before their values are used in an expression.

```java
// Fig. 4.13: AnalysisTest.java
// Test program for class Analysis.

public class AnalysisTest
{
   public static void main( String args[] )
   {
      Analysis application = new Analysis(); // create Analysis object
      application.processExamResults(); // call method to process results
   } // end main

} // end class AnalysisTest
```

Create an Analysis object

- **AnalysisTest.java**

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 9
Failed: 1
Raise Tuition
```

More than 8 students passed the exam

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 6
Failed: 4
```

2b.11 Compound Assignment Operators

Compound assignment operators

- **An assignment statement of the form:**
  ***variable* = *variable operator expression* ;**
  where *operator* is `+`, `-`, `*`, `/` or `%` can be written as:
  ***variable operator*= *expression* ;**
- **example: `c = c + 3`*;* can be written as `c += 3`*;***
  - **This statement adds `3` to the value in variable `c` and stores the result in variable `c`**

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`C = c + 7`	`10` to `c`
`-=`	`d -= 4`	`d = d - 4`	`1` to `d`
`*=`	`e *= 5`	`e = e * 5`	`20` to `e`
`/=`	`f /= 3`	`f = f / 3`	`2` to `f`
`%=`	`g %= 9`	`g = g % 9`	`3` to `g`

Fig. 2b.14 | Arithmetic compound assignment operators.

2b.12 Increment and Decrement Operators

Unary increment and decrement operators

- Unary increment operator (++) adds one to its operand
- Unary decrement operator (−−) subtracts one from its operand
- Prefix increment (and decrement) operator
    - Changes the value of its operand, then uses the new value of the operand in the expression in which the operation appears
- Postfix increment (and decrement) operator
    - Uses the current value of its operand in the expression in which the operation appears, then changes the value of the operand

Good Programming Practice 2b.7

Unlike binary operators, the unary increment and decrement operators should be placed next to their operands, with no intervening spaces.

Operator	Called	Sample expression	Explanation
`++`	prefix increment	`++a`	Increment `a` by `1`, then use the new value of `a` in the expression in which `a` resides.
`++`	postfix increment	`a++`	Use the current value of `a` in the expression in which `a` resides, then increment `a` by `1`.
`--`	prefix decrement	`--b`	Decrement `b` by `1`, then use the new value of `b` in the expression in which `b` resides.
`--`	postfix decrement	`b--`	Use the current value of `b` in the expression in which `b` resides, then decrement `b` by `1`.

Fig. 2b.15 | Increment and decrement operators.

```java
// Fig. 4.16: Increment.java
// Prefix increment and postfix increment operators.

public class Increment
{
   public static void main( String args[] )
   {
      int c;

      // demonstrate postfix increment operator
      c = 5; // assign 5 to c
      System.out.println( c );   // print 5
      System.out.println( c++ ); // print 5 then postincrement
      System.out.println( c );   // print 6

      System.out.println(); // skip a line

      // demonstrate prefix increment operator
      c = 5; // assign 5 to c
      System.out.println( c );   // print 5
      System.out.println( ++c ); // preincrement then print 6
      System.out.println( c );   // print 6

   } // end main

} // end class Increment
```

Postincrementing the `c` variable

Preincrementing the `c` variable

```
5
5
6

5
6
6
```

•Increment.java

Common Programming Error 2b.9

Attempting to use the increment or decrement operator on an expression other than one to which a value can be assigned is a syntax error. For example, writing `++(x + 1)` is a syntax error because `(x + 1)` is not a variable.

Operators	Associativity	Type
`++ --`	right to left	unary postfix
`++ -- + - (` *type* `)`	right to left	unary prefix
`* / %`	left to right	Multiplicative
`+ -`	left to right	Additive
`< <= > >=`	left to right	Relational
`== !=`	left to right	Equality
`?:`	right to left	Conditional
`= += -= *= /= %=`	right to left	assignment

Fig. 2b.17 | Precedence and associativity of the operators discussed so far.

2b.13 Primitive Types

Java is a strongly typed language

- **All variables have a type**

Primitive types in Java are portable across all platforms that support Java

Portability Tip 2b.1

Unlike C and C++, the primitive types in Java are portable across all computer platforms that support Java. Thanks to this and Java's many other portability features, a programmer can write a program once and be certain that it will execute on any computer platform that supports Java. This capability is sometimes referred to as *WORA* (Write Once, Run Anywhere).

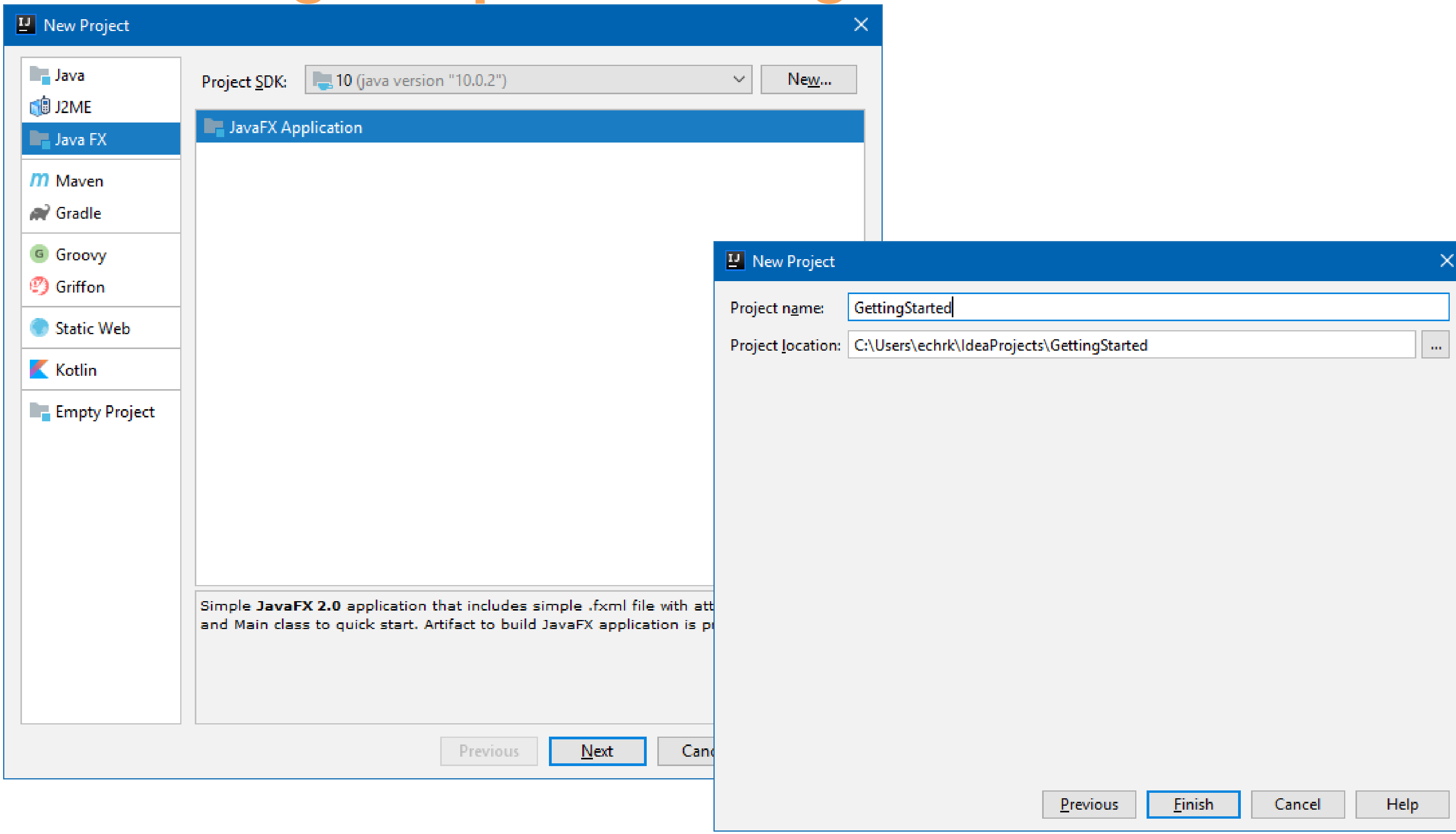
2b.14 GUI and Graphics Case Study: Creating Simple Drawings with JavaFX

The programs we've explored thus far have been text-based.

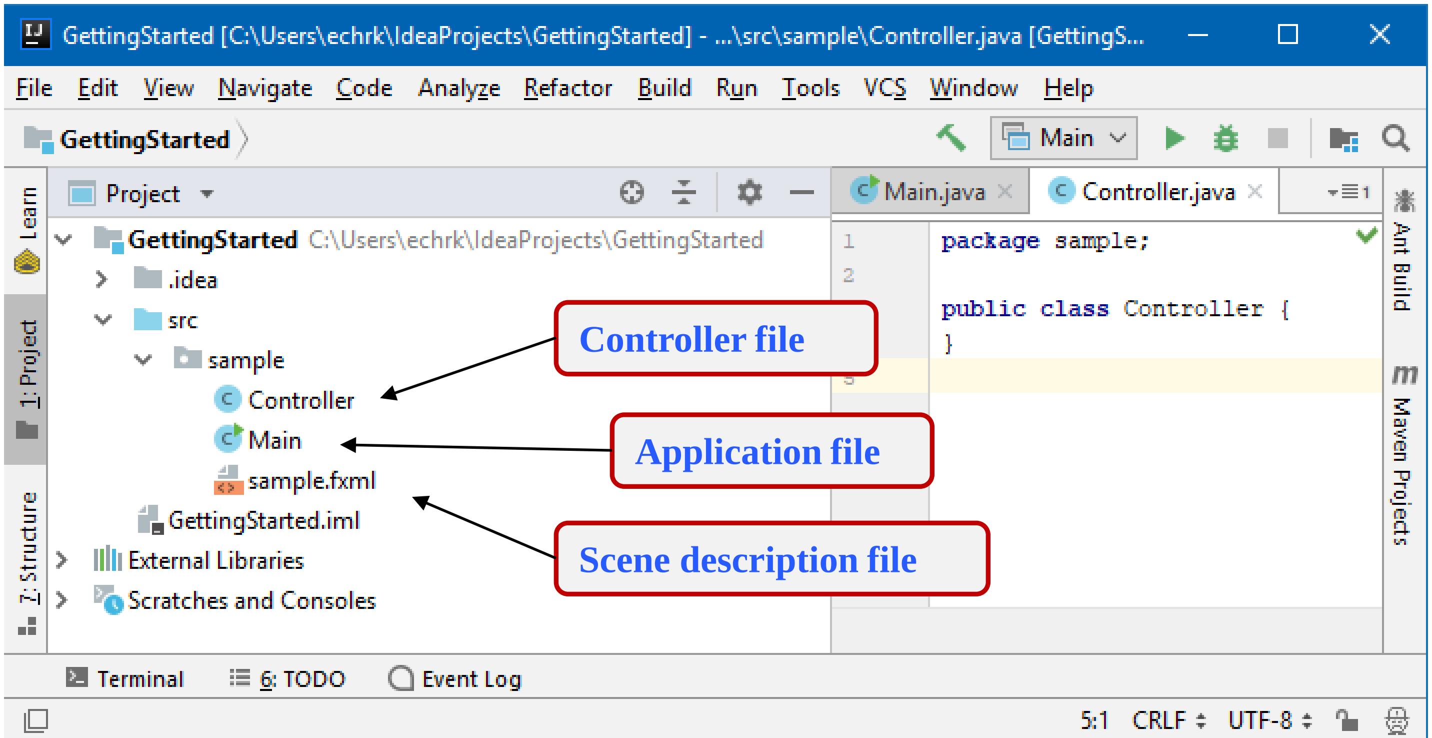
They are called *command-line applications* (Java Console Applications), which interact with the user using simple text prompts.

We’ll now begin to explore programs that use graphics and graphical user interfaces (GUIs). Support for these programs will come from the JavaFX API. JavaFX is the current standard in Java for developing GUI with rich content.

2b.14 GUI and Graphics Case Study: Creating Simple Drawings with JavaFX

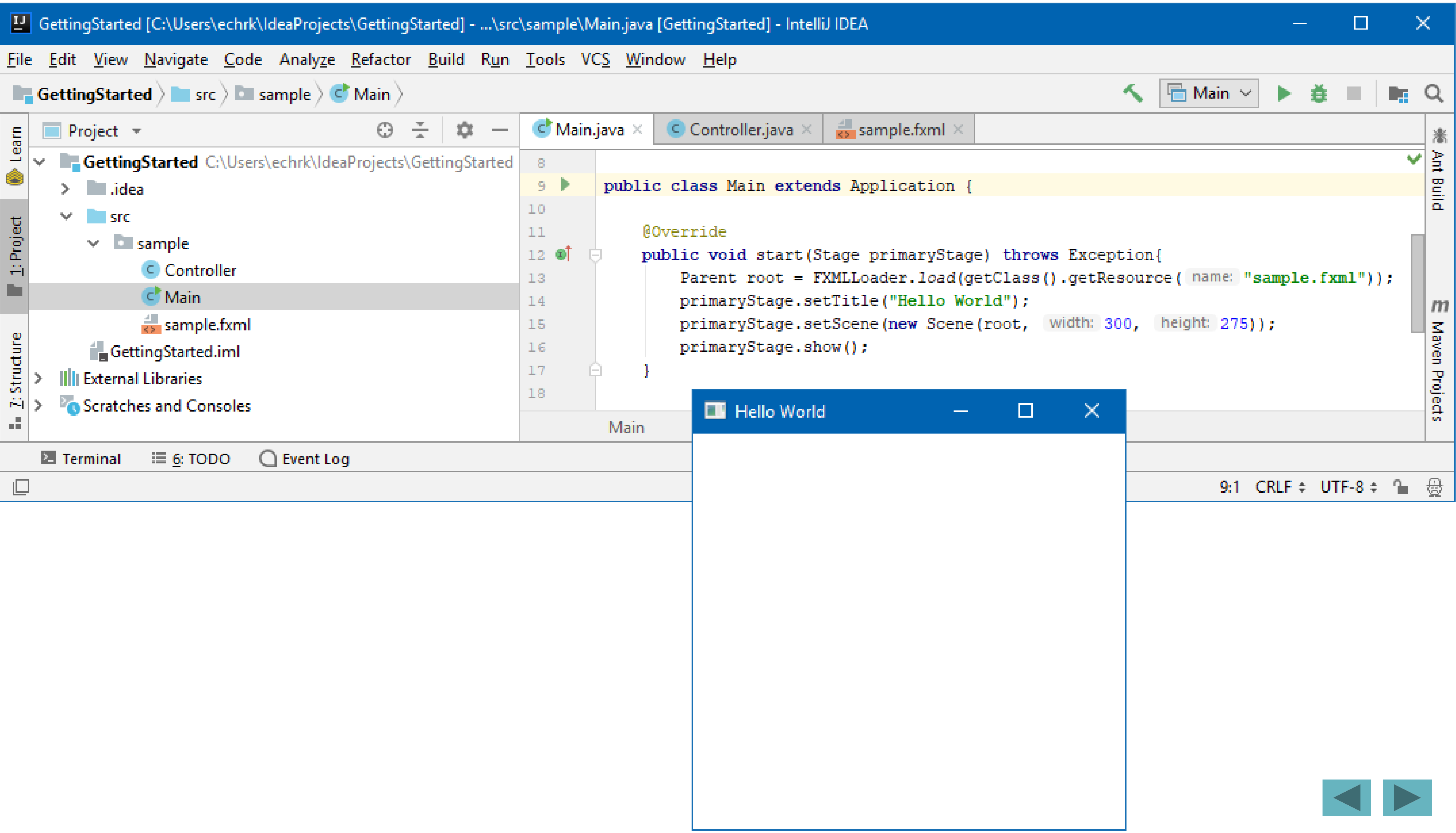


2b.14 GUI and Graphics Case Study: Creating Simple Drawings with JavaFX

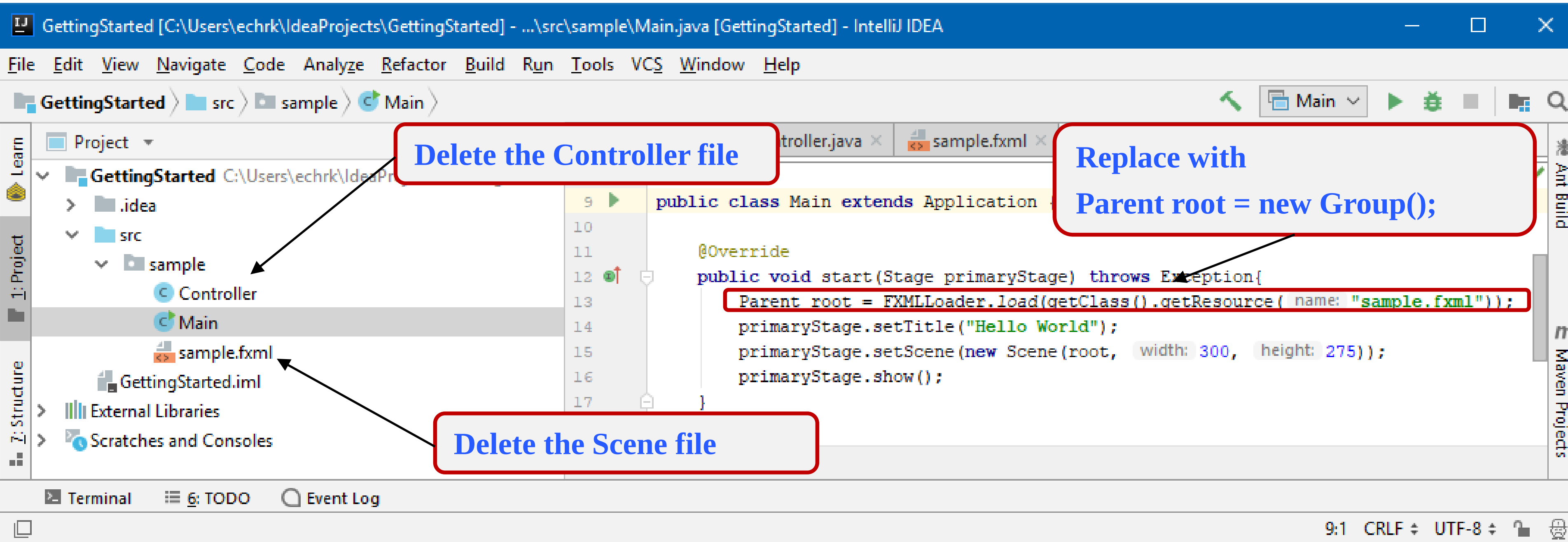


Note: In order to compile and run the Application you must add the JavaFX library to the Module-> Dependencies of the project. Study the attached SetupJFXwithJDK13IntelliJ.pdf

2b.14 GUI and Graphics Case Study: Creating Simple Drawings with JavaFX



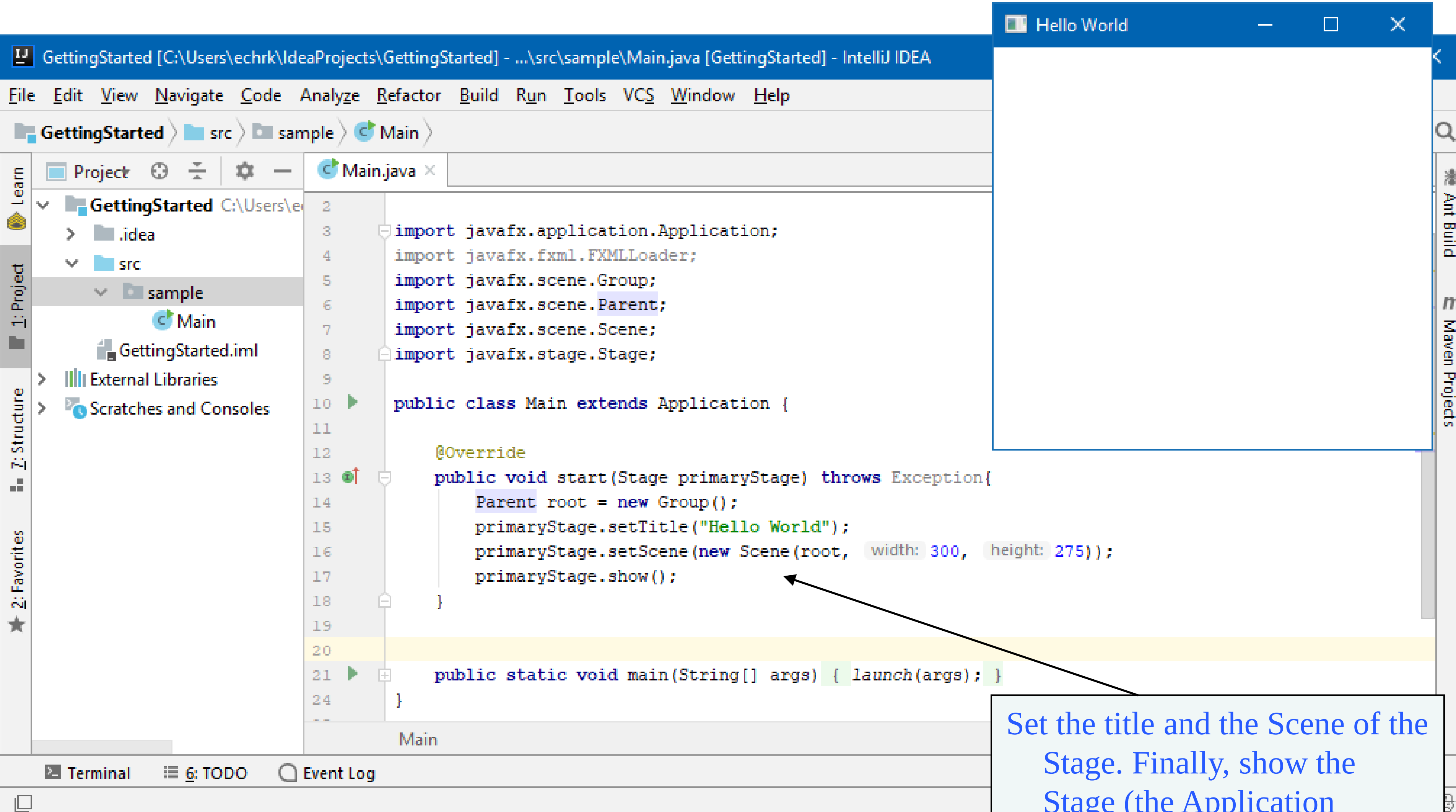
2b.14 GUI and Graphics Case Study: Creating Simple Drawings with JavaFX



Rebuild the application….

and run it.

2b.14 GUI and Graphics Case Study: Creating Simple Drawings with JavaFX



Set the title and the Scene of the Stage. Finally, show the Stage (the Application window)

2b.14 (Optional) GUI and Graphics Case Study: Creating Simple Drawings (Cont.)

- **Inheriting**
  - **`extends` keyword**
  - **The subclass inherits from the superclass**
    - **The subclass has the data and methods that the superclass has as well as any it defines for itself**

- **JavaFX programs extends the `Application` class, providing core graphical functionality**

2b.14 GUI and Graphics Case Study: Creating Simple Drawings with JavaFX

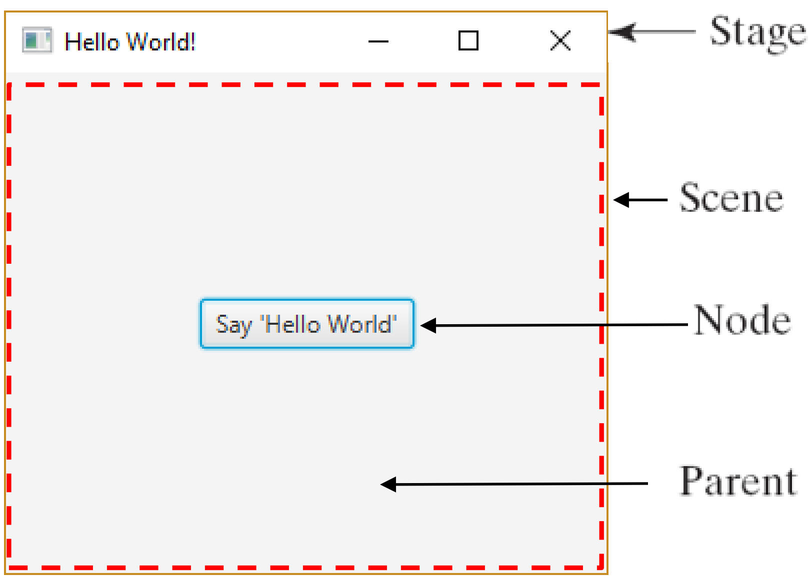
A JavaFX program has a `start()` method

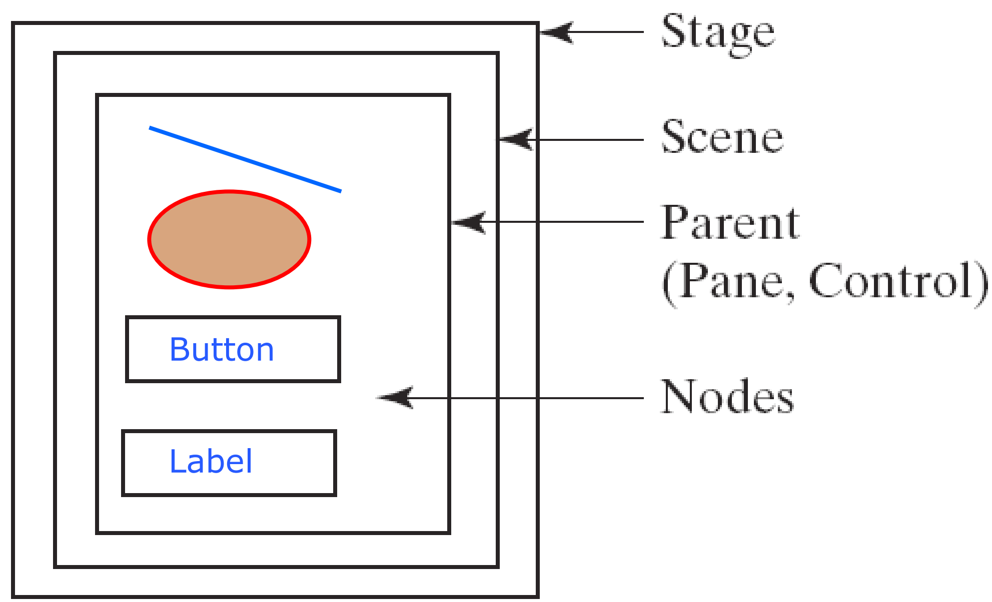
The `main()` method is only needed to `launch()` the JavaFX application

The `start()` method accepts the primary Stage (window) used by the program as a parameter.

JavaFX embraces a theatre analogy

2b.14 GUI and Graphics Case Study: Creating Simple Drawings with JavaFX






```
compile-single:
run-single:
Hello World!
```

2b.14 GUI and Graphics Case Study: Creating Simple Drawings with JavaFX

In this example, a `Button` object is added to a parent `StackPanel`. The pane serves as the *root node* of a `Scene`. Content is added to the pane using the `getChildren()` method of the node.

The `Scene` is displayed on the primary `Stage` (the Application window). Therefore we call this class the JavaFX Application class.

The `title` and the `Scene` of the `Application` window (the `Stage`) are set with the `setTitle`() and the `setScene`() methods correspondingly.

2b.14 **GUI and Graphics Case Study: Creating Simple Drawings with JavaFX**

Java's coordinate system

- **Defined by x-coordinates and y-coordinates**
  - **Also known as horizontal and vertical coordinates**
  - **Are measured along the x-axis and y-axis**
- **Coordinate units are measured in pixels**

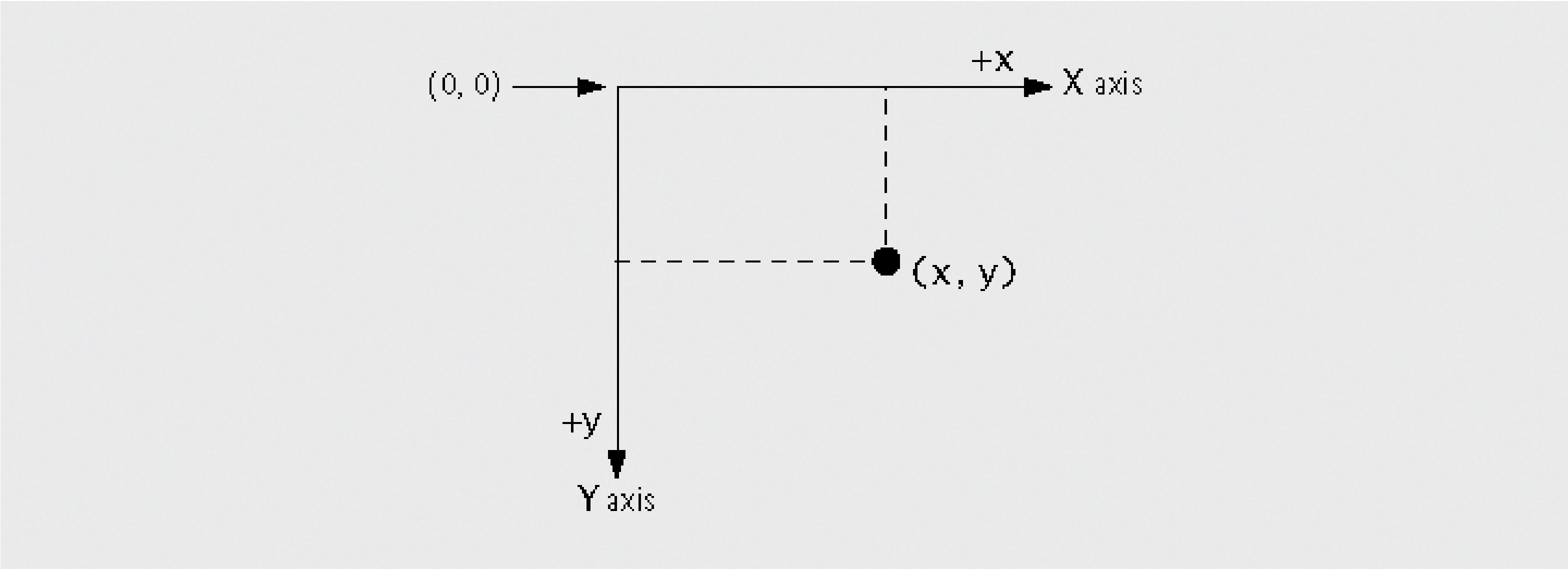


Fig. 2b.18 | Java coordinate system. Units are measured in pixels.

2b.14 GUI and Graphics Case Study: Creating Simple Drawings with JavaFX

JavaFX shapes are represented by classes in package `javafx.scene.shape`

A Line segment is defined by the `Line` class, whose constructor accepts the coordinates of the two endpoints:

```
Line(startX, startY, endX, endY)
```

For example:

```
Line myLine = new Line(10, 20, 300, 80);
```

2b.14 GUI and Graphics Case Study: Creating Simple Drawings with JavaFX

A Rectangle is specified by its upper left corner and its width and height:

```
Rectangle(x, y, width, height)

Rectangle r =
     new Rectangle(30, 50, 200, 70);
```

2b.14 GUI and Graphics Case Study: Creating Simple Drawings with JavaFX

A circle is specified by its center point and radius:

```
Circle(centerX, centerY, radius)
```

```
Circle c = new Circle(100, 150, 40);
```

```java
package drawlines;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Line;
import javafx.stage.Stage;

public class DrawLines extends Application {

    @Override
    public void start(Stage primaryStage) {
        Group group = new Group(); // Create a layout Panel
        Scene scene = new Scene(group, 300, 250);// Create the Scene

```

Class Application provides core functionality of the JavaFX application

Define the root Panel and the Scene

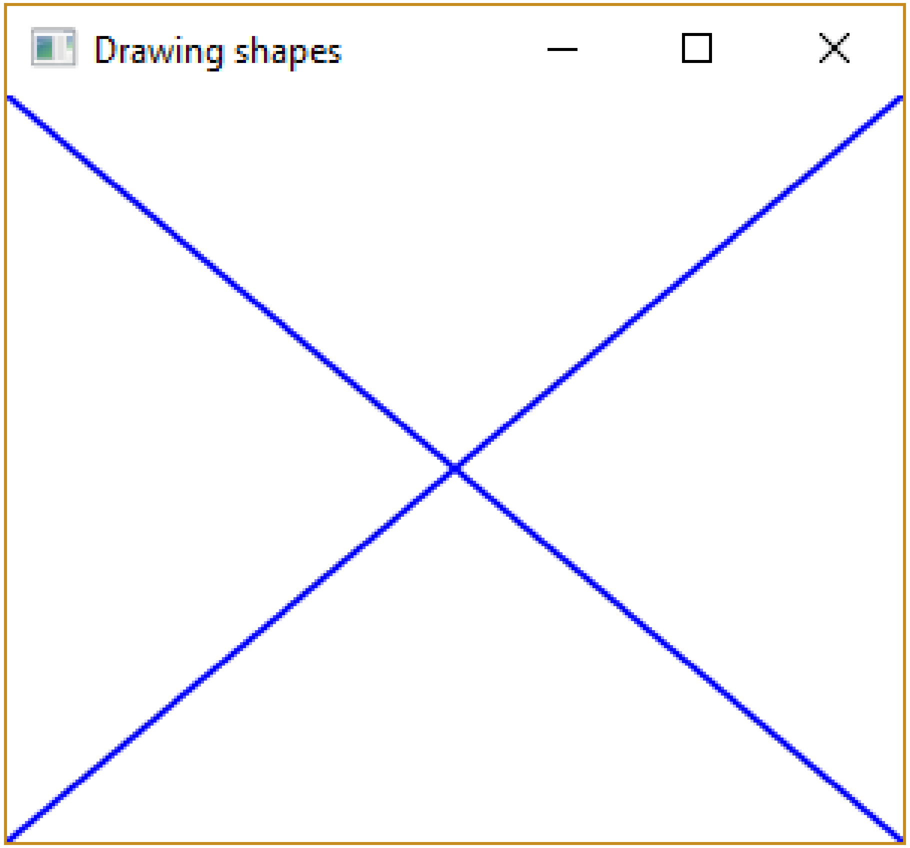
```
        // add content to the Layout panel
        double width = scene.getWidth(); // total width of the scene
        double height = scene.getHeight(); // total width of the scene

        // draw a line from the upper-left to the lower-right
        Line line = new Line(0, 0, width, height);
        line.setStroke(Color.BLUE);
        line.setStrokeWidth(2);
        group.getChildren().add(line);
        // draw a line from the lower-left to the upper-right
        line = new Line(0, height, width, 0);
        line.setStroke(Color.BLUE);
        line.setStrokeWidth(2);
        group.getChildren().add(line);

        // Set the title of the Stage(the application window)
        primaryStage.setTitle("Drawing shapes");
        // Add the Scene to the Stage
        primaryStage.setScene(scene);
        // Show the Stage (the application window)
        primaryStage.show();
    }

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

Draw the two lines


Drawing shapes

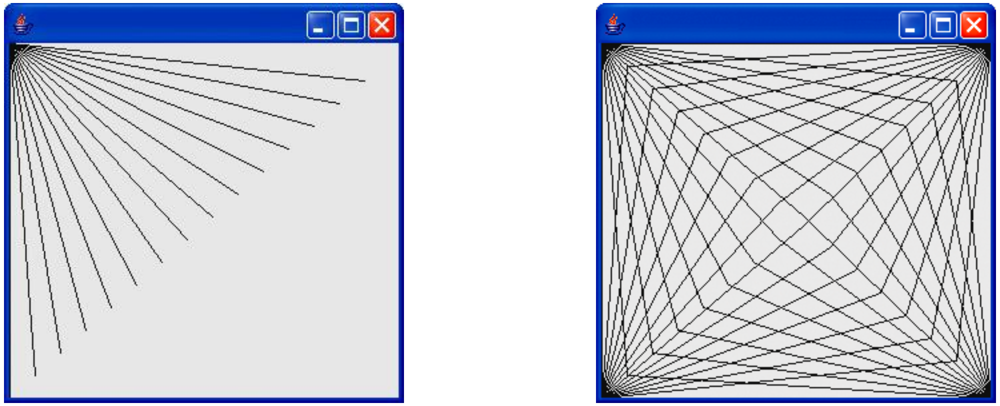


Fig. 2b.21 | Lines fanning from a corner.

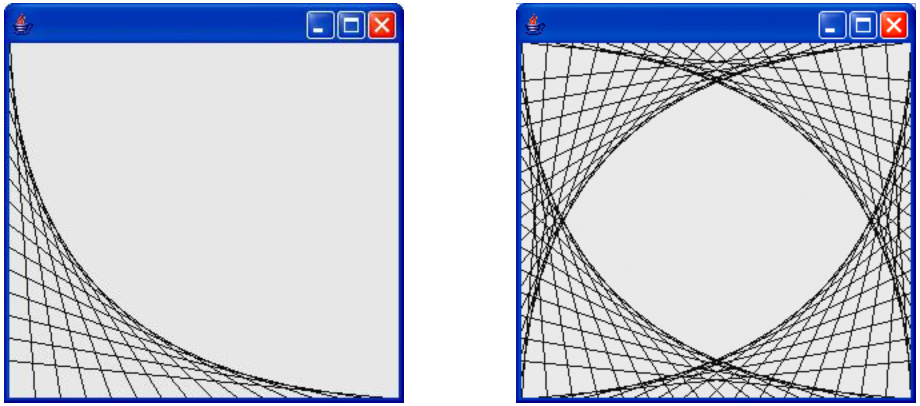


Fig. 2b.22 | Line art with loops and drawLine.